Acceleration of Cryptographic Functions using Graphics Hardware

Owen Harrison

A thesis submitted to the

University of Dublin, Trinity College

for the degree of Doctor of Philosophy

February 2010

Declaration

I, the undersigned, declare that this work has not previously been submitted to this or any other University, and that unless otherwise stated, it is entirely my own work.

Owen Harrison

Dated: 18th February, 2010

Permission to Lend and/or Copy

Acknowledgements

I would like to thank my family for always being there, unwavering in support. To John Waldron, my supervisor, thank you for giving me the initial opportunity, the freedom I required and well timed advice dotted over the past years. Thanks to my friends for encouraging, sympathising, distracting, entertaining, especially Barry, Claire, Garrett, Niall, Marie and Saar. To Tony, my best man, for many ideas, many comments, much patience, a debt of gratitude I am incapable of repaying. To Rachel, my reader, my open ears, my support, my friend, my wife, thank you, te quiero mucho.

Abstract

Graphics processing units (GPUs) can act as an attractive alternative to CPUs for general purpose computation in certain scenarios. Traditionally, the GPU has been developed to offload graphics processing from the CPU. In recent years the GPU has continued to become a more flexible and powerful device, responding to the demand of the games industry to execute more and more complex custom graphics algorithms. In the early 2000s a new approach to processing emerged, whereby non-graphics problems that suit a data parallel model could execute on the GPU at competitive or faster rates that the CPU. This performance gap has continued to grow, and as the GPU develops in terms of programming flexibility, the range of application spaces that benefit from GPU assistance widens. Adding to this trend, GPU vendors have started releasing programming frameworks specifically tailored to general purpose computation on GPUs. In light of these developments, there is intense research involving the use of GPUs for acceleration within many problem spaces. We advance the state of the art by presenting the capacity of the GPU to accelerate commonly used cryptographic functions.

We investigate GPU acceleration of symmetric-key and asymmetric-key functions, fundamental components of modern cryptographic systems. We show that AES, a popular example of a symmetric-key function, can be competitive with the CPU on recent GPUs and outperform on contemporary GPUs. We illustrate the issues related to GPU support of symmetric-key modes of operations in various scenarios and present strategies for maintaining performance. We show that RSA, a popular example of an asymmetric-key function, can outperform the CPU when running on the GPU. For both symmetric-key and asymmetric-key approaches presented, not all cryptographic contexts suit the GPU and as such these contexts are highlighted. Also, both approaches are investigated for efficient batching of multiple requests within a single GPU call. Finally, the integration of GPU accelerated cryptography within an operating system abstraction layer and associated costs are presented.

Related Publications

O. Harrison and J. Waldron. Efficient Acceleration of Asymmetric Cryptography on Graphics Hardware. *International Conference on Cryptology in Africa, (AfricaCrypt)*, Gammarth, Tunisia, June 21–25, 2009. LNCS, Volume 5580/2009, Pages 350–367.

O. Harrison and J. Waldron. Public Key Cryptography on Graphics Hardware. *Annual International Conference on the Theory and Applications of Cryptographic Techniques, (Eurocrypt)*, Cologne, Germany, April 26–30, 2009. Appeared in conference booklet, Pages 65–72, and as Poster.

O. Harrison and J. Waldron. Practical Symmetric Key Cryptography on Modern Graphics Hardware. *USENIX Security Symposium*, San Jose, CA, July 28–August 1, 2008. Pages 195–209.

O. Harrison and J. Waldron. AES Encryption Implementation and Analysis on Commodity Graphics Processing Units. *Workshop on Cryptographic Hardware and Embedded Systems, (CHES)*, Vienna, Austria, September 10–13, 2007. LNCS, Volume 4727/2007, Pages 209–226.

O. Harrison and J. Waldron. Optimising Data Movement Rates for Parallel Processing Applications on Graphics Processors. *International Conference on Parallel and Distributed Computing and Networks*, Innsbruck, Austria, February 12–14, 2007. Pages 251–256.

Under Review

O. Harrison and J. Waldron. GPU Accelerated Cryptography as an OS Service, September 2009. Currently available as a technical report [45].

Glossary

AES	Advanced Encryption Standard
CBC	Cipher Block Chaining MOO
CCM	Counter with CBC-Message Authentication Code MOO
CTR	Counter MOO
DES	Data Encryption Standard
DMA	Direct Memory Access.
DX9	DirectX 9. Microsoft suite of media APIs. Also used to identify a generation of compliant graphics hardware.
DX10	DirectX 10.
ECB	Electronic Codebook MOO
ECC	Elliptic Curve Cryptography
FLOP	Floating Point Operation
GPU	Graphics Processing Unit
GPGPU	General Purpose Computation on Graphics Processing Units
Kb,Mb,Gb	2^{10},2^{20},2^{30} bits
KB,MB,GB	2^{10},2^{20},2^{30} bytes
MOO	Symmetric-key Mode of Operation
NIST	National Institute of Standards and Technology
OCF	OpenBSD Cryptographic Framework
RNS	Residue Number System
ROP	Raster Operations
RSA	Rivest Shamir Adleman asymmetric-key algorithm
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
SM	Streaming Multiprocessor
SP	Streaming Processor
Warp	Group of threads issued by the GPU thread scheduler as a single unit.
XOR	Exclusive OR

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphics processing units (GPU) have exhibited a large increase in floating point performance compared with traditional CPUs since the early 2000s. The traditional CPU has levelled off in terms of clock frequency as power and heat concerns have become dominant restrictions. A 2008 GPU released by Nvidia reports a peak throughput of almost 1 TeraFlop. In contrast, a similar era quad core Intel x86 CPU reports a peak throughput in the order of 100 GigaFlops. The continuing divergence of floating point performance is made clear by Figure 1.1. The performance advantage of the GPU over the CPU comes at a price of decreased applicability to general purpose computing. The CPU is optimised for general purpose serial processes. Large on-chip caches and complex control logic are used to hide system memory access latencies. This leads to a large amount of the x86 transistor budget being consumed by caches and control logic. The GPU in comparison contains a reduced amount of on-chip caches and control logic and in return there is an increased percentage of transistor logic spent on execution units. For example, the latest Nvidia GPU contains 240 simple processing cores. This division of transistor budget narrows the GPU's suitability to tasks that are data parallel and exhibit a high compute intensity.

A data parallel task is one where multiple threads can be instantiated, all executing the same programme, across many data elements in parallel. An example of such a task is graphics rendering, where there exists a number of input data elements and a single shading program that executes in parallel across the input elements to generate pixels. A task exhibiting a high compute intensity, or arithmetic intensity, is one with a high ratio of arithmetic to memory access instructions. Memory latency on the GPU is expected to be hidden by a combination of arithmetic instructions and a high number of parallel threads, rather than large caches and control logic. Tasks that fit the data parallel and compute intensity criteria can exhibit higher performance on a GPU compared to a CPU.

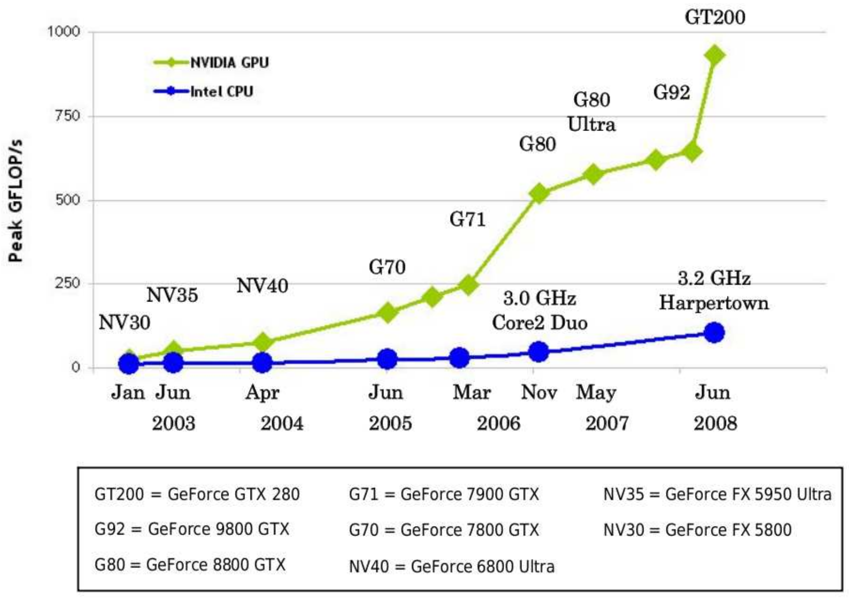


Figure 1.1: Floating point performance for Nvidia GPUs and Intel CPUs [93].

An important trend, which broadens the GPUs suitability to general computation, is the increasing ability to program the processors. Early GPUs were fixed function devices, supporting a graphics pipeline via parametrisation only. Subsequent generations introduced programmable units within the GPU, primarily to allow game programmers to execute custom code for pixel shading. Each generation further increased the flexibility of these programmable units. Examples include increases in supported program length, increases in register counts, addition of branch instructions and improved memory access. Early recognition of the possibility of using the GPU as a general purpose data parallel, or stream, processor came from Buck et al. [12] and Venkatasubramanian [125] in 2003.

An early important milestone for general purpose computation on GPUs (GPGPU) was the release of Shader Model 3.0 [107] compliant processors. Shader Model 3.0 is part of Microsoft's DirectX 9 [71] suite of programming interfaces and details the feature set of the programmable units within a GPU. These GPUs, released by both the major GPU vendors, Nvidia and AMD (formerly ATI), improved the flexibility of the programming units to the extent where they could realistically compete with the traditional CPU with regard to performance on suitable compute intensive parallel problems. Early examples of GPGPU include work in diverse fields such as databases [34], computer vision [30] and data mining [36]. One of the most popular GPGPU projects, Folding@home [28], used a DirectX 9 compliant GPU to achieve a $\sim$2.5 times increase over a comparable CPU. The next major milestone for GPGPU was the introduction of Shader

Model 4.0 [73] complaint GPUs in 2006. This standard further pushed the capabilities of the programmable units within the GPU to resemble fully featured CPUs. GPU vendors also recognised the market for general computation acceleration by releasing non graphical APIs to program their devices for the first time. From this point on, the interest in GPGPU has intensified, spreading to a large number of application contexts [37].

The increase in pervasiveness of confidential online transactions, digital rights management solutions and the secure digitisation of all forms of governmental, corporate and personal data, drives the demand for efficient security systems. The high speed implementation of cryptographic algorithms used to form such systems continues to be an active area of research. Such implementations are not only limited to the CPU, but are also based on an array of diverse hardware devices such as FPGAs, ASICs, PCI co-processor cards, RFID, smart cards and USB devices [26]. Two common types of constituent algorithms in security systems are symmetric-key and asymmetric-key. Symmetric-key algorithms are generally used for confidential communication between parties in the case where a shared secret between the parties exists. Asymmetric-key algorithms are generally used for confidential communication between parties where no shared secret exists. These algorithm types are combined in various ways to produce secure systems providing authenticity, integrity and confidentiality. Symmetric-key and especially asymmetric-key algorithms can exhibit a high compute intensity, and in certain circumstances, a suitability to highly parallel compute devices.

1.1 Thesis Aim

The aim of this thesis is to investigate to what extent the GPU can act as an efficient cryptographic processing platform for symmetric-key and asymmetric-key algorithms. The focal algorithms for this investigation are two of the most popular cryptographic functions, AES [79] (symmetric-key) and RSA [111] (asymmetric-key).

1.2 Motivation

The GPU is now ubiquitous, but for the most part it is grossly underutilised. Unless playing high-end games, in the order of a TeraFlop of processing power is largely going unused. There exists the potential to use this available power in the capacity of a co-processor for non-graphical processing. In particular, it could possibly fulfil the role of existing PCI-based cryptographic accelerator cards. Standard end-user machines could potentially offload their cryptographic requirements to the GPU, freeing up the CPU for other tasks. Applications within a server environment requiring bulk cryptographic processing, such as secure backup/restore or high bandwidth media streaming, could similarly benefit. Also, it is important that potential sources of high performance cryptographic processing are benchmarked. This provides the metrics against which the security assumptions of the various cryptographic primitives are scrutinised. A further motivation for employing the GPU to perform cryptographic tasks is the possibility of creating a reduced trusted computing base that is designed to hide data from the CPU. This could be used for example in the transfer of encrypted video feeds, which are decrypted directly on the GPU. The potential for such a system has been discussed by Cook et al. [14].

A background motivation for this work is related to the traditional CPU design bottlenecking on clock frequency. The main technique employed to tackle this problem is to increase the number of on-chip processing cores, thereby increasing the total amount of available clock cycles. Intel are currently developing "many core" architectures within the Terascale and Larrabee projects. Terascale has demonstrated an 80-core processor as a research prototype. Larrabee is yet to be released as a commercial product, though published architecture designs [117] show it to be a highly parallel processor aimed at graphics tasks and general purpose compute intensive tasks. AMD are also developing a hybrid processor within their Fusion project, combining on a single die a traditional CPU design with a many core GPU-like design for parallel work [3]. It seems likely that future general purpose computation will involve some form of highly parallel compute device. The GPU, currently with 240 on-chip processing cores, can be seen as an early example of such a device. Future parallel devices are likely to follow similar design paths, with many simple arithmetic logic units (ALU) tied to small caches. As such, by mapping cryptography to GPUs, we also hope to expose the issues that this type of processing will encounter in such future highly parallel architectures.

1.3 Contributions

Interest in cryptography on graphics hardware has increased in recent years, especially with the introduction of DirectX 10 compliant GPUs. In Chapter 2 we give the first comprehensive survey of published work related to cryptographic acceleration on graphics hardware. The work in this thesis covers acceleration of cryptographic functions on DirectX 9 and DirectX 10 compliant GPUs. Prior to DirectX 10, the only method of programming graphics processors was via a graphics API such as via DirectX or OpenGL [97]. Because of its operating system agnostic status, we used OpenGL to program the DirectX 9 compliant devices. During the implementation of AES on such hardware, it became apparent that one of the primary performance bottlenecks was efficient movement of data to and from the graphics card over the system bus. Furthermore, achieving optimal transfer rates for a task using a graphics API is complex, involving the careful selection from a vast array of configuration combinations. No tools existed to help with the selection of such configuration states as little demand exists for bi-directional large data transfers from traditional graphics applications. Chapter 3 is based on work from “Optimising Data Movement Rates For Parallel Processing Applications On Graphics Processors” [41]. This introduces new tools that are used to investigate data movement rates, performance cliff avoidance, and the minimisation of data transfer bottlenecks in our AES implementations.

Chapter 4 is based on work presented in “AES Encryption Implementation and Analysis on Commodity Graphics Processing Units” [40]. It describes the first DirectX 9 compliant GPU implementation of a symmetric-key function that gives competitive performance compared to the traditional CPU. AES is implemented using various approaches in an effort to work around the lack of integer support on the GPUs involved. We also investigate an issue raised by Cook et al. [15], whereby the OS reports 100% usage when executing their OpenGL based AES implementation. We look into this more deeply and show the extent to which the CPU can offload cryptographic operations to the GPU and free cycles for other work.

Chapter 5 presents work from “Practical Symmetric Key Cryptography on Modern Graphics Hardware” [43]. Here, a DirectX 10 compliant GPU is used to implement the fastest reported AES implementation on a GPU. The various issues involved with a high-speed implementation on a DirectX 10 compliant GPU are discussed. The results are compared to CPU implementations and are shown to be favourable. Also presented is a new model for mapping symmetric-key client requests to the GPU in a generic manner. We also study the GPU’s ability to process symmetric-key requests within the context of various modes of operation. A mode of operation acts as a wrapping protocol specifying how to use an underlying symmetric-key cipher securely.

Work that first appeared in booklet format in Eurocrypt 2009 as “Public Key Cryptography on Graphics Hardware” [46] and later in Africacrypt 2009 as “Efficient Acceleration of Asymmetric Cryptography on Graphics Hardware” [44] is presented in Chapter 6. This describes one of the first DirectX 10 GPU implementations of efficient big integer modular exponentiation suitable for use in many asymmetric-key cryptographic systems. We show how careful use of the GPU memory system can increase performance and greatly reduce latency over superficially similar GPU implementations. We analyse different number representation systems with an aim to finding a balance between peak performance and minimum latency. In the context of modular exponentiation using residue number systems we introduce new and optimised algorithms for various parts of the implementation. Our GPU exponentiation approaches are employed in an RSA context and are shown to give superior performance compared with CPU based implementations.

Chapter 7 describes the integration of the GPU as an Operating System service for use by both userspace and kernelspace consumers. An existing virtualisation services layer, the OpenBSD Cryptographic Framework (OCF) [55], which abstracts clients from hardware accelerated cryptographic processors, is used as the basis of this work. The novel contributions are, the effective integration of the GPU within the OCF model; the observation that the GPU interface is userspace only and the newly developed mechanisms to allow it to be used as part of a kernel service; the introduction of a new memory management system within the OCF to allow efficient handling of memory transfers between multiple address spaces; and an implementation of a general purpose multi-request batching scheme for asymmetric key requests with regard to the GPU. This work is under peer review and is currently made available as the technical report, "GPU Accelerated Cryptography as an OS Service" [45].

Chapter 2

Background and Related Work

The research presented in this thesis involves the use of two different types of GPU architecture. The first type of architecture concerns GPUs that are DirectX 9 compliant, we label these GPUs as DX9. The second type concerns the use of newer GPUs that are DirectX 10 [70] compliant, labelled DX10. Firstly, this chapter covers an introduction to both architectures, highlighting design features relevant to the implementations presented in subsequent chapters. Secondly, we present the required background to the cryptographic functions explored and related mathematics. Finally, related work in the area of GPUs and cryptography is covered. The GPU information below is generally Nvidia-centric due to the implementations presented in subsequent chapters being based on Nvidia hardware. Nvidia were the first to release DX10 compliant hardware and as such were the vendor of choice for most early DX10 based GPGPU research.

2.1 DX9 Compliant Graphics Hardware

2.1.1 Hardware Overview

With respect to Nvidia hardware, DX9 compliant GPUs include all instances of the NV4X [85] and G7X [88] architecture families. These processor families are largely outdated at the time of writing by newer DX10 compliant processors, however many are still in use. For example, the GPU used within Sony's PlayStation 3 is a DX9 GPU based on the Nvidia GeForce 7800GTX, part of the G7X family. GPUs are designed to suit the concept of a graphics pipeline, which traditionally receives as input a description of a 3D model and outputs a 2D rendering for display on screen. Figure 2.1 shows a heavily simplified hardware view of a DX9 compliant GPU, the GeForce 7900GTX, a high-end GPU from the Nvidia G7X family. The figure depicts the main programmable units within

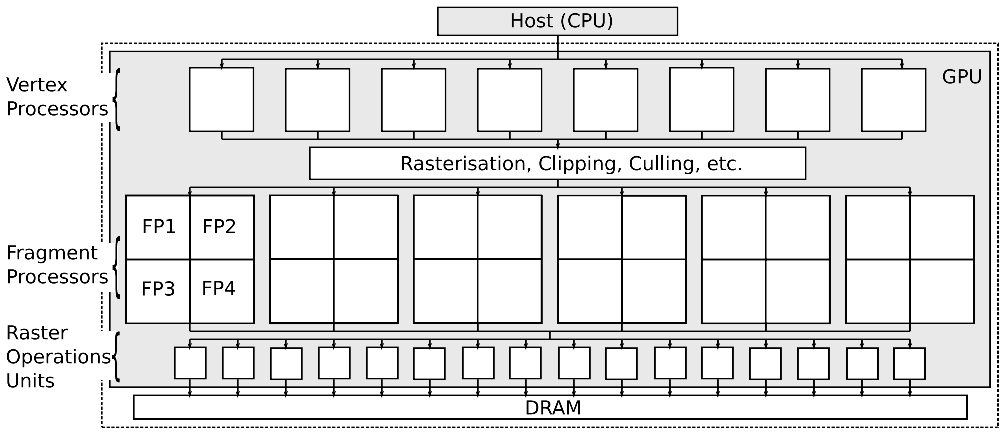


Figure 2.1: Simplified block diagram of the GeForce 7900GTX, a DX9 compliant GPU.

the GPU, the vertex and fragment processors, which support the execution of custom code. The corresponding components within graphics processors prior to DX9 were largely fixed function and were controlled by parameterisation. This allowed little scope for the GPU to be used for anything other than its primary graphics function.

The DX9 vertex and fragment processors contain the majority of the processing power found within a GPU. The distribution of this processing power is not shared evenly between the two types of processing units. Typical graphics applications place more demand for processing on the fragment processors and thus most of the processing power lies within these units. For example, the GeForce 7900GTX is equipped with 24 fragment processors, each of which contains (amongst other components) two 32-bit floating point (FP32) 4-vector arithmetic logic units (ALU). Thus, the fragment processors provide 48 FP32 4-vector ALUs. In comparison, the vertex processors provide 8 equivalent FP32 4-vector ALUs. Figure 2.1 also depicts the raster operations units (ROP). These units are responsible for committing fragment processor output to device DRAM memory.

2.1.2 Programming Interface

Programming DX9 compliant processors requires the use of a graphics APIs such as Direct3D [39], part of the DirectX family of APIs, or OpenGL. These graphics APIs use a programming pipeline, which supports the configuration of the data processing and I/O stages within the GPU. This pipeline, shown in Figure 2.2, is divided into vertex processing, rasterisation, fragment processing and raster

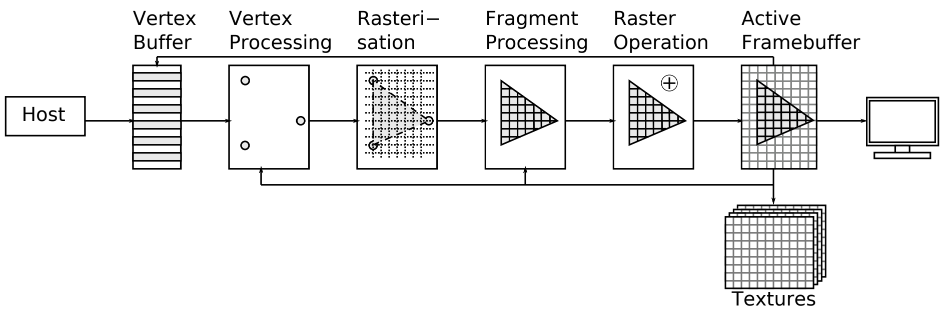


Figure 2.2: The different stages of the DX9 graphics pipeline.

operation processing stages. One can see the parallels between the hardware components discussed above and the programming pipeline. The operation and data in each pipeline stage can be summarised as follows:

- A list of vertices, which exist within a 3 dimensional space, is provide to the graphics driver. These vertices act as descriptors for primitives such as points, triangles or quadrilaterals.
- The vertex processing stage of the pipeline transforms the vertex co-ordinates into screen space co-ordinates.
- The rasterisation stage is responsible for accepting the screen space co-ordinates, which represent primitives, and generating a pixelised view. This pixelised view comes in the form of arrays of fragments. A fragment denotes a potential pixel, which may or may not be rendered to a final output buffer.
- The fragment processing stage accepts fragments as input and can manipulate attributes of each fragment, such as colour.
- These fragments are outputted to the final stage, raster operation, which is ultimately responsible for writing the final pixel colour values to memory for display or recycling back into the pipeline.

As mentioned, the DX9 vertex and fragment processors are programmable. The vertex processing stage runs custom code, called a *vertex shader*, for each input vertex. The code can specify a series of input and output vertex attributes, such as position, normal, texture co-ordinates and colour. The rasteriser interpolates the output attributes of each vertex within a primitive, generating fragments with a set of interpolated attributes. The fragment processing stage runs custom code, called a *fragment shader*, for each fragment handed off to it by the rasteriser. It is responsible for combining the fragment attributes to generate a limited number of colour values for handoff to the raster operations units. Various high level shading languages exist for writing shaders, including Microsoft's DirectX High Level Shading Language HLSL [72], OpenGL's shading language GLSL [113] and Nvidia's Cg [66]. These languages provide a C-like environment to program the GPU's vertex or fragment processors. They also provide explicit support for GPU related features such as textures, vector data types, colours and co-ordinates. Shader Model 3.0, part of the DirectX 9.0c specification, details the minimum feature set that both the vertex and fragment processors support. The raster operation is the last stage of the pipeline and runs on the ROP units. This stage allows parameterised control over how the fragment shader's output is combined with the destination framebuffer. Although this stage does not support custom code execution and can only be controlled by parameterisation, as we shall see, it can be useful when combining fragment processor output with GPU memory.

2.1.3 Textures

Textures are a key component in the programmer's arsenal when programming a GPU. Textures are blocks of memory which exist in off-chip device memory on the GPU card. They can be thought of as similar to bulk data arrays in a normal CPU programming environment. These blocks of memory consist of individual elements called *texels*, each of which has a storage format. An example of a texel's format could be a 4 wide FP32 vector representing the values for red, green, blue and alpha (RGBA). Traditionally textures were accessible only by the fragment processing stage in the GPU, allowing the combination of colours at particular locations within a texture and the colour values output by the rasteriser. DirectX 9 hardware improved texture access by supporting texture reads from both the vertex and fragment processors.

The fragment processor has a somewhat restricted ability to write to textures. This ability is termed as render-to-texture and is important in the context of general purpose processing. During the configuration of the pipeline, via OpenGL or DirectX, the programmer can specify that the fragment processor should output to a texture instead of a frame buffer. The advantage of rendering to texture is that the output of a pipeline pass can be used directly as the input of the next pipeline pass. Without render-to-texture, access to the output of the pipeline requires a copy-to-texture call, which involves the overhead of a memory copy (from frame buffer to texture). Iterative solutions, which require the consumption of the previous pipeline's output as the next pipeline's input can be implemented using a pair of textures, one for input and one for output.

A single texture cannot be used in this scenario due to coherence issues when reading and writing to the same texture in a single pass. After each pipeline pass the role of the texture pair is switched, the input texture becomes the output and vice versa. This implementation pattern is informally called the ping-pong technique [33].

Textures can be declared as 1, 2 or 3 dimensional, though 2 dimensional textures are most commonly used with DX9 GPUs. 1D textures are restrictive in size due to limits of 2048 texels per dimension. 3D texture write support is absent on some DX9 GPUs. Other DX9 GPUs support 3D texture writes, though inefficiently relative to 2D texture writes. The internal data types stored within a texel can be 16-bit or 32-bit floats and also a variety of bit width integers up to 16-bit. The number of data components stored per texel ranges from 1 to 4. It is a common technique to pack multiple data units into each texel to improve I/O efficiency and make effective use of the vector processors.

2.1.4 General Purpose Computation

A problem that suits a parallel processing model can be broken into a large number of independent, or loosely bound tasks. The code which executes the task in this context is commonly referred to as a *kernel*. With regard to GPGPU, the kernel normally takes the form of a fragment shader which runs on fragment processors. To control the number of fragment shader instances, which can be seen as controlling the number of threads executing the kernel, we draw a 2D quadrilateral of size x pixels by y pixels. The aim of this draw command is to create $x \times y$ number of fragments shader instances, corresponding to the number of fragments generated by the rasteriser. To ensure the size of the quadrilateral corresponds in a 1-to-1 manner with the number of generated fragments, we setup an orthogonal projection for the viewing volume which encompasses the quadrilateral. This type of projection preserves the dimensions of objects specified within the viewing volume. The viewing volume is a set of clipping planes which delimit 3D space and determine the active vertices which are sent to the vertex processors. We must also create a viewport with the same dimensions as the quadrilateral. The viewport is a plane onto which the viewing volume is projected. See Figure 2.3 for a depiction of the viewing volume, viewport and an orthogonal projection.

By configuring the graphics pipeline as described above we can create a defined number of threads which run on the GPU. To specify the input data made available to each fragment shader instance we can bind one or more 2D textures to the quadrilateral vertices. The rasteriser is responsible for interpolating the

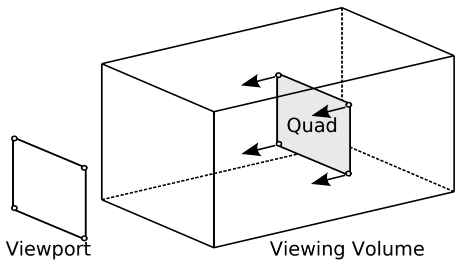


Figure 2.3: Orthogonal projection configured to protect the co-ordinates of the input vertices.

texture co-ordinates bound to the vertices. These interpolated co-ordinates are made available to each fragment instance. A common requirement is to maintain a 1-1 mapping of fragment instances to input data elements. To achieve this the bound texture is made the same size as the quadrilateral, for example x by y as above. In the same way, the output can be stored following a 1-1 mapping between the fragment instances and output data elements by binding a texture of the same size as the quadrilateral to an output framebuffer object. These 1-1 mappings are useful when each thread's input and output data element count and size are symmetric. View volume, viewport and texture setup must be done to configure the graphics pipeline before a draw command is executed. The draw command triggers a single pass of the pipeline. It is equivalent to program execution, executing a potentially large number of fragment program instances and storing their output ready for readback by the CPU or for the next pass of the pipeline. Extensive coverage of general purpose computation on DX9 compliant GPUs and example applications can be found in the survey by Owens et al. [101].

2.1.5 Restrictions and Performance Considerations

DX9 hardware is suited to the execution of data parallel tasks and lacks the flexibility of the standard CPU. We list the major restrictions and related performance considerations associated with using this hardware for general computation. Much of the information regarding DX9 restrictions can be found in Nvidia documentation [87, 107].

Memory: The DX9 GPU memory model provides a relatively complex and performance sensitive environment compared to the CPU. Care has to be taken with regard to memory usage to avoid performance bottlenecks. Points of note regarding the DX9 GPU's memory model include:

- The amount of high speed on-chip memory available for each ALU on a

DX9 GPU is limited to sizes within the estimated range of 16 KB to 24 KB. In comparison, a similar era CPU provides combined ALU access to much greater sizes of on-chip caches, typically in the order of 2-4 MB.

- The GPU's high speed memory mainly consists of a read-only cache for device memory, called the texture cache. Writing to device memory does not update the cache within a single pipeline pass. This causes cache coherence issues if fragment programs within a single pipeline pass read and write to the same memory locations.
- Vertex and fragment shaders have access to local read/write memory in the form of a limited set of temporary registers. These registers are the only form of fast on-chip read/write memory. Also of note, the registers do not support indexed access, limiting their use in certain contexts.
- Vertex and fragment shaders have access to read-only high speed on-chip constant registers. Like the temporary registers, these are limited in number. The numbers of available DX9 temporary and constant registers can be seen later in Table 2.1.
- Writing to device memory is supported only during the last stage of the rendering pipeline. A fragment shader cannot commit data to device memory at any stage, but can only output a limited amount of data to the raster operation stage which is responsible for committing values to device memory.
- The GPU lacks scatter support within its fragment processors. Output locations for data are determined via the pipeline configuration before pipeline execution and as such is fixed during fragment execution. The GPU can output to multiple textures, called render targets, however this is limited to 4 with DX9 GPUs. Thus, the amount of output generated by a single pipeline pass for a single fragment shader is limited to 4 texels.
- Memory allocation and memory free operations can only take place before or after a pipeline execution, but not during.

Thread Co-operation: The ALUs on the GPU, whether those within the vertex or fragment pipelines are incapable of sharing information during a single pass of the pipeline. The only manner of sharing data is to execute multiple pipeline passes using render-to-texture as described above. The execution of each pipeline pass comes with overheads not associated with core computation and thus the more fine grained the requirement for data sharing, the less effective the GPU becomes.

Data Transfer: Any computation using a DX9 or DX10 GPU involves the transmission of data across the system bus. This data is not only input and output for shader execution but also any control operations related to pipeline configuration. The overhead of system bus use becomes more of a performance bottleneck as the amount of computation performed per pipeline pass decreases. Also, as previously mentioned, textures are used as data arrays for shader input and output. Thus, we transfer input and output data via texture transfers to and from the GPU. As we will see in Chapter 3, careful pipeline configuration is required to avoid large drops in transfer rate potential.

Instruction Support: The ALUs on DX9 GPUs support only 32-bit floating point for processing and data access. No native integer or bitwise operations are supported within the vertex and fragment processors. This reduces the effectiveness of the main source of GPU processing power with regards to cryptographic processing due to its heavy requirement for integer processing.

Branching Costs: Branching is supported on both the vertex and fragment processors. The vertex processors support MIMD (multiple instruction multiple data) branching, where each processor can execute a different thread of instructions. The fragment processors however act as a set of SIMD (single instruction multiple data) groups. If branching occurs within a SIMD group, each divergent path must be executed serially and the results predicated. Thus, although fragment processor branching is supported, there can be a large performance penalty if thread divergence is common.

Program Length: The vertex and fragment processors support a minimum of 512 static instructions as specified by the shader model specification. The number of actual executed instructions, i.e. dynamic instructions, is fixed at 65,535. These limitations restrict the complexity of the possible programs that can be run on the GPU in a single pass and as a result can lead to the fragmentation of programs into multiple parts, each called in a separate pipeline pass.

Access to Design Details: The finer details of the GPU architectures designed by vendors, Nvidia and AMD, are often kept secret. Information which may be beneficial to general purpose computation such as low level optimisation can be difficult or impossible to access. Examples of this include the lack of information on the exact cache sizes available to each ALU or the rasterisation patterns used to generate fragments for handoff to the fragment processors. This tradition of secrecy continues with DX10 hardware.

Vertex Processors: Efficiently employing the vertex processors in a general purpose application can be difficult due to their placement and function within the graphics pipeline. Their output values are in the form of vertex attributes. These attributes are used by the fixed function rasteriser to generate the number of fragment shader instances required and also to generate each fragment's interpolated attribute values. The fixed interpolation of the vertex output attributes is beneficial in graphical applications, though has limited use in general computation. This interpolation can be avoided by using point primitives. However, this can result in idle fragment processors as the rasteriser will generate one fragment per vertex. DX10 hardware specifically addresses this issue by allowing more effective use of all the available GPU processors depending on the work required.

2.2 DX10 Compliant Graphics Hardware

2.2.1 Departure from DX9

The DirectX 10 specification details the system architecture for a more flexible programmable graphics processing unit. This specification requires a minimum feature set from compliant GPU's processing cores, which comes close to the flexibility of the traditional CPU. One of the main improvements in DX10 compliant hardware over DX9 in relation to cryptography is the addition of integer operation support. Included in the integer instruction set is support for arithmetic and bitwise operations. Another high profile change includes an increase in complexity of the graphics pipeline with an additional stage called the Geometry Shader. However, we will see in Section 2.2.3, that general purpose use of the GPU no longer has to be concerned with the graphics pipeline.

Shader model 4.0 [73] is included as part of the DX10 specification. The model specifies a unified instruction set for all of the programmable stages of the pipeline: vertex, fragment and geometry. The specification also defines a single virtual machine to be used as a common base for all programmable stages. These specification requirements have important implications for hardware implementations in terms of general purpose processing, see Section 2.2.2. Also included

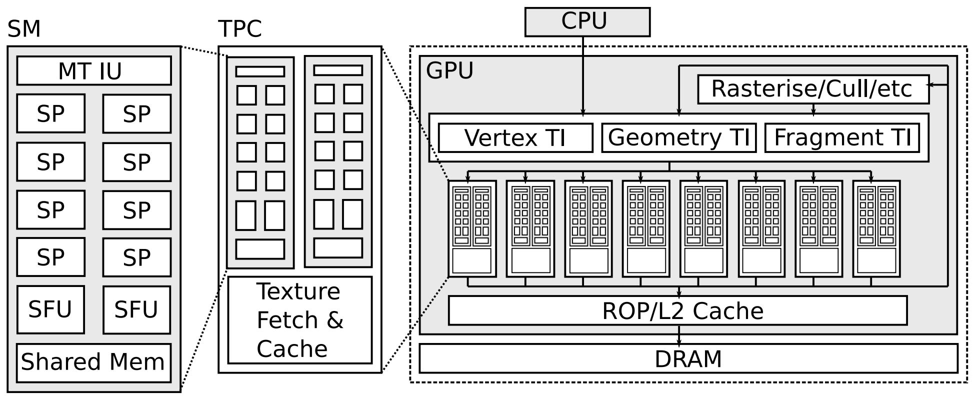


Figure 2.4: Simplified block diagram of the GeForce 8800GTX, the first DX10 compliant GPU.

in the specification are large increases in the number of constant and temporary registers available to each programmable stage. These registers are now indexable, which increases their flexibility compared to previous register usage. Other improvements over DX9 include an increase in size of texture dimensions; increase in number of textures available to programmable stages; increase count of instruction slots, i.e. static program length; increased dynamic program length; new data load instruction to allow unfiltered device memory loads, rather than reading via texture sampling.

2.2.2 Hardware Overview

As mentioned previously, the DX10 specification details both a common virtual machine base and instruction set for all programmable stages of the graphics pipeline. This is referred to as the unified shader model. Both Nvidia and AMD DX10 compliant GPUs reflect these requirements by implementing each programmable stage using the same processing cores. This approach is commonly known as unified shader architecture. The first commercially available DX10 compliant GPU was the Nvidia GeForce 8800GTX, an instance of the G8X [90] architecture family. This architecture provides an homogeneous array of processing cores. The GPU is capable of dynamically configuring the processing cores to assume the role of any of the three programmable stages, depending on the work required. As previously mentioned, it is difficult to make full use of the vertex processors for general purpose computation on DX9 GPUs. Both Nvidia and AMD DX10 compliant GPUs alleviate this issue by dynamically assigning the processing cores to the work type present in the pipeline.

2.2.2.1 Nvidia DX10 Compliant Hardware

In Figure 2.4 we can see that the Geforce 8800GTX has 128 homogeneous stream processing (SP) cores. Each stream processor is a pipelined, in-order, scalar microprocessor capable of executing 32-bit floating point and 32-bit integer operations. The G8X and subsequent Nvidia architecture use scalar processors rather than the tradition vector processors found in graphics hardware. This is due to the realisation that as shader programs get more complex, the active vector width narrows [92, p27]. Thus, complex shader programs fail to fully utilize the entire width of the vector processors and as such leave hardware idle. A scalar approach does not suffer from this scenario. Also, as general purpose computations can commonly involve scalar operations, the full utilisation of vector hardware can require inventive data packing techniques. The change to scalar processors removes this obstacle to achieving performance on the GPU.

We can also see in Figure 2.4 that the stream processors are grouped into units called streaming multiprocessors (SM). Each SM contains 8 SPs; 2 special function units (SFU), which handle transcendental operations; a multithreaded instruction unit (MT IU); and shared memory. The SMs are grouped into pairs to form a thread processing cluster (TPC). Within a TPC the SMs share texture fetch units, texture addressing units, texture cache and other control logic. The architecture scales by a combination of changing the number of SPs per SM, SMs per TPC and the number of TPCs per GPU. The latest Nvidia architecture family at time of writing, the G200 [94], demonstrates this by increasing the processing cores to a total of 240 by combining 3 SMs per TPC and increasing the number of TPCs to 10. The design of the G200 is essentially the same as the original DX10 compliant GPU. In Figure 2.4 we also see the ROP layer, which serves the same purpose as in DX9 hardware. Also present is a level 2 cache (of undefined size) to provide increased performance for texture reads.

SIMT: As mentioned, an SM contains 8 SPs, which are tied to a multithreaded instruction unit. This arrangement functions in a manner called SIMT (single instruction, multiple-threads) [93], which is similar to SIMD. The SM is capable of executing threads, i.e. instances of a kernel execution, each of which run on a single SP. Each thread in effect executes with its own instruction pointer and register state. The similarity of SIMT to SIMD is that the instruction unit issues a single instruction for all SPs to execute at any one time. As such all SPs within an SM execute the same instructions, though can operate on their own unique data. The main difference between SIMT and SIMD is that SIMD instructions expose the vector width to software, whereas SIMT instructions are scalar and

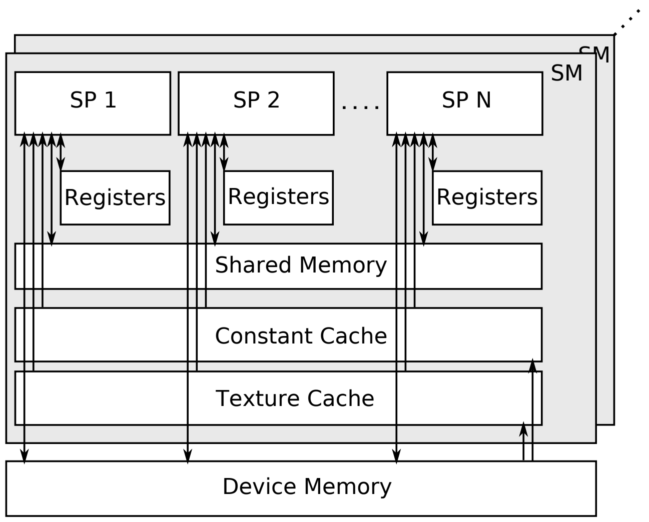


Figure 2.5: Block diagram of the physical memory available to the SPs on a G8X GPU.

determine the execution and branch behaviour of a single thread. SIMT uses an abstraction that allows the SIMD like SM to create multiple independent scalar threads. This abstraction is implemented in hardware by a thread scheduler. The scheduler is capable of dynamically disabling the output of sets of ALUs within the SM during divergent thread execution.

Although the SM ensures correctness of execution of independent scalar threads, for efficiency it creates and schedules in groups of 32 parallel threads, called *warps*. This grouping of threads into warps is not reflected in software, though it is an important consideration in application development with regards to efficiency. The instruction unit issues the same instruction for each group of 32 threads, selectively committing results dependent on thread branching. With regard to performance, the architecture can be viewed as having an effective SIMD width of 32. The performance implications of thread branching on the architecture is discussed in Section 2.2.4.2.

Memory: All types of memory available to each stream processor can be seen in Figure 2.5. This figure shows all the on-chip, high speed memory and the off-chip memory available to each SP. Each memory type is designed for a specific task and generally requires explicit programmer instructions. Each physical memory type can be described as follows:

- Registers - each SP has its own on-chip 32-bit register file.

- Shared Memory - all SPs within a single SM share a small on-chip read-

/write memory. Shared memory can only be written to and read from during kernel execution and is accessibly only by the SPs that reside in the same SM as the shared memory.

- Constant Cache - a read-only cache shared by all SPs that is used to speed up reads from constant memory space.
- Texture Cache - a read-only cache shared by all SPs that is used to speed up reads from texture memory space. All memory read via the texture cache comes from the texture units within the TPC.
- Device Memory - a read/write DRAM memory which exists off-chip. Whereas the on-chip memories are generally small, in the order of kilobytes, current graphics cards contain up to a low number of gigabytes of device memory.

Each SP within an SM has the ability to write to arbitrary locations within the GPU's device memory. This overcomes the DX9 fragment processor's lack of scatter support - the inability of shaders to write directly to device memory. This increases the DX10 GPU's flexibility with regard to general purpose computation. Also, SP access to shared memory has important synchronisation implications. Threads running on the same SM have the ability to co-ordinate via a synchronisation barrier instruction. Combining the synchronisation barrier with the high speed on-chip read/write shared memory can provide a means of implementing fine grained inter-thread co-operation.

2.2.3 Programming Interface - CUDA

Both major GPU vendors have released new software environments for general purpose computation on their DX10 hardware. Nvidia released the Compute Unified Device Architecture (CUDA) [83] and ATI/AMD initially released "Close-To-Metal" (CTM) [1], which was later encapsulated in Stream SDK [2]. These new software environments allow programmers to avoid the use of traditional graphics APIs such as OpenGL and Direct3D to harness the GPU for general computation. All DX10 implementations within this thesis were made using CUDA and as such we focus on this architecture here. CUDA provides the ability to program the GPU using the C++ language with specialised extensions. These extensions can be grouped according to compute capability. Each GPU release that supports CUDA has a particular compute capability release number. The first CUDA supporting devices, and also the first DX10 compliant devices, implemented compute capability 1.0. Subsequent hardware releases have followed with minor updates to the CUDA extensions and as such add new compute capability versions. At the time of writing, the latest CUDA compute capability was 1.3. The CUDA implementations presented in this thesis are based on compute capability 1.0 and thus, are compatible with all CUDA devices.

CUDA extends the C++ language to support the distinction of code which runs on the CPU and the GPU, called host and device code respectively. The CUDA compiler processes CUDA source files according to C++ syntax rules, however device code is restricted to a limited set of C++ syntax. Device and host code can be mixed within a single file, though are separated at the function level. A language extension provides function type qualifiers to indicate whether code runs on the host or device. A CUDA program which executes on the GPU is called a kernel. The "main" function, or starting point, for a kernel is specified by tagging a function with the `__global__` qualifier. A global function can only be called from a host function and can only call device functions. A `__device__` qualifier can be used to denote a function which runs on the GPU and is callable from a global or another device function. A `__host__` qualifier specifies a function which runs on the CPU and can be called by other host functions. Variable type qualifiers have also been added to the language and specify the type of memory that stores the variable. These qualifiers include `__device__`, `__constant__` and `__shared__`. Texture memory is specified using its own type and registers are specified using automatic variables within device functions.

CUDA also provides a language extension to define the number of threads to be created on kernel execution. The number of threads are specified via the combination of a CUDA grid and a CUDA block. Threads are organised into blocks and blocks are organised into a grid. A block is represented by a 3 component vector, each component represents a dimension. The total number of threads in a block are derived from the multiplication of the three dimensions. All blocks within a grid are the same size. Also, all threads within a block execute on a single SM. A grid is represented by a 2 component vector, each component represents a dimension. The total number of blocks in a grid are derived from the multiplication of the two dimensions. The total number of threads spawned for a kernel execution on the GPU is derived from the number of threads per block multiplied by the number of blocks per grid. For example, given a CUDA block value of (1, 3, 100), and a CUDA grid value of (2, 3), the total number of threads for the kernel execution equals $(1 \times 3 \times 100) \times (2 \times 3)$. Note, that there is a hard limit of 512 threads per block, however we later discuss factors that can further restrict the number of threads per block.

Another language extension gives each thread access to a unique identifier via

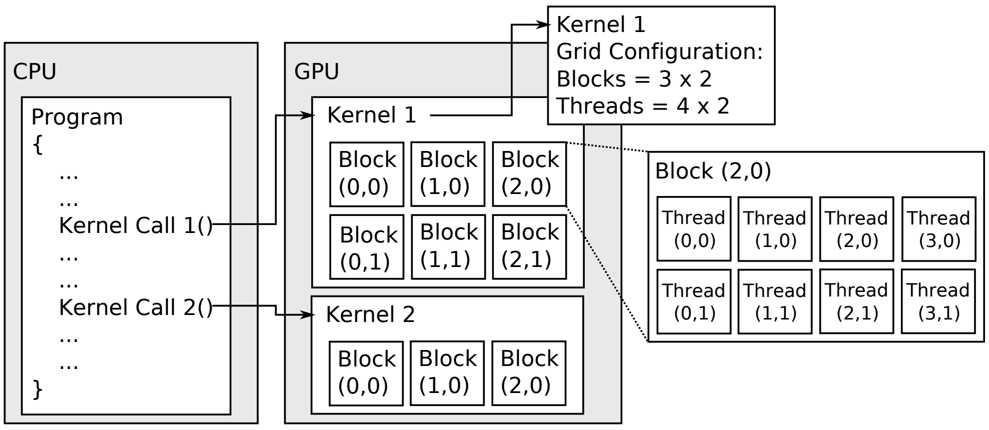


Figure 2.6: Example of the CUDA execution model.

the use of CUDA's build-in `threadIdx` and `blockIdx` variables. The `threadIdx` variable is set to the threads position within the CUDA block it belongs to. The `blockIdx` is set to thread's block within the grid. The total number of threads executed on the GPU for a global function (kernel) call is equal to the number of threads per block multiplied by the number of blocks in the grid. A single C program running on the CPU can execute multiple, different kernels serially on the GPU. This technique can be used to isolate separate parallel and serial portions of a task, executing the serial parts on the CPU and the parallel parts on the GPU. Figure 2.6 illustrates an example of the execution of multiple kernels on the GPU, interleaved with serial code on the CPU. It also illustrates the grid and block configuration for each kernel execution.

CUDA defines a number of memory address spaces. Each thread has its own *private memory* address space. This address space is backed by a combination of on-chip high performance registers and slower off-chip device memory. This space has a lifetime of the thread. Each thread has access to a *shared memory* space, which is shared by threads within the same CUDA block and has a lifetime of the block. The space is implemented by high performance on-chip shared memory. All threads that comprise a kernel execution have access to a single address space called *global memory*. Global memory address space resides in device memory. Threads also have access to read-only *texture memory* and read-only *constant memory* address spaces. These address spaces are shared by all threads in a grid and are implemented using device memory. Unlike the global address space they are accelerated by on-chip caches. The global, texture and constant address spaces are persistent across multiple kernel executions within the same application.

2.2.4 Restrictions and Performance Considerations

There are a number of restrictions and performance pitfalls encountered when designing solutions for acceleration using DX10 compliant GPUs. The main issues of concern are listed below. Many of the points presented below can be found in official Nvidia literature [91, 92, 93].

2.2.4.1 Memory

Careful use of the different memory address spaces available for use by threads on the GPU is an essential part of efficient application development. The limited availability of fast on-chip memory acts as one of the main programming concerns. Appendix B.3 lists the available memories and sizes for the reference DX10 GPU in this work, the GeForce 8800GTX. We detail the performance related concerns for each type of memory space below.

Private Memory: This type of memory is implemented using a combination of on-chip registers and off-chip device memory. Registers are used to store automatic variables declared within a kernel. Automatic variables are variables that are not explicitly specified by the programmer to reside in any other address space and are generally used as fast working memory. These registers are referred to as temporary registers in certain contexts. Local memory is used by the compiler when there is insufficient numbers of registers available to store all automatic variables. This is referred to as “register spillage”. Device memory is used to store local memory, which is relatively slow compared to register access. An SM takes 4 clock cycles to issue a memory instruction per CUDA warp. Register access generally adds no extra clock cycles per instruction, whereas device memory access adds between 400 and 600 extra cycles. From this difference it is clear that register spillage can cause a performance cliff, particularly when taking into account the normally heavy use of registers within a programme, and should be avoided if possible. It is possible to detect this spillage using compiler options to display register usage. Note that, implementations presented in Chapter 5 gain performance by allowing a small amount of register spillage for a increase in concurrent threads per SM.

Shared Memory: A strategy for performance improvement regarding repeated access to data is to use shared memory as a staging area for this data. This strategy involves the prefetching of data from slower device memory into on-chip shared memory. Subsequently this data can be accessed efficiently by an algorithm, and can optionally be committed back to device memory. We see an example of this staging strategy in Chapter 5. As shared memory is relatively small compared to device memory a data tiling approach to staging can be used. This splits the data into subsets, each subset being read into shared memory for use one tile at a time [57]. A slight variation on staging is to use shared memory to store intermediate results which require repeated access, again providing a speed up over global memory use. This strategy is seen in Chapter 5.

Each SM's shared memory is physically arranged in a series of 16 banks, or modules, each 32 bits wide. Banks are organised such that successive 32-bit words are assigned to successive banks. A shared memory request is issued on behalf of a 1/2 warp, i.e. one request per 16 threads, and as such each request consists of 16 addresses. Each address within a request maps onto a particular bank. If all addresses map to a unique bank, no bank conflicts occur. As long as no bank conflicts occur, shared memory access is as quick as register access. Bank conflicts occur when addresses within a shared memory request map to the same bank. When a request addresses the same bank multiple times, these accesses are serviced serially. The hardware splits memory requests into as many separate (and serial) conflict free requests as necessary to service the original request, thus the bandwidth is reduced by a factor equal to the number of separate requests made. An exception to the serialisation of memory requests concerns the ability of hardware to choose a broadcast address, which allows a single address to service multiple accesses at the one time. An example of this is where no conflicts occur when all threads access the same shared memory location. The programmer does not have control over the selection of the broadcast bank. Considering the organisation of data within shared memory is deterministic, one can endeavour to arranged storage and retrieval of data to and from shared memory to minimise the number of bank conflicts.

A further concern relating to shared memory is its population with data. There is no mechanism by which a programmer can pre-load data into shared memory for use in a subsequent kernel call. Data must be loaded into shared memory by the threads within a CUDA block during the thread's runtime. This introduces complexity into the design of algorithms which wish to use shared memory as a manual cache for speeding up global memory access. With respect to efficiency, the task of shared memory population should ideally be divided equally across all threads. As thread count per block depends on many factors, such as shared resource contention, this division can lead to inefficiencies. For example, if all threads wish to use a shared lookup table residing in shared memory, the task of populating this table is split across all threads. If this task does not divide evenly amongst the threads within a block, then some threads can be left doing nothing during all or part of the memory copy. This is encountered in Chapter 5.

A note should be made regarding security doubts raised by the use of shared memory for processing of cryptographic functions. Shared memory is only accessibly between threads within a single kernel instance. As CUDA supports no time-slicing, thus each kernel is executed from beginning to end without interruption, and also each kernel is executed by a single CPU process, the sharing of data between threads gives no more access to data than the calling CPU process already has. However, it should also be noted that care should be taken if the GPU were used for cryptographic processing in an untrusted compute environment to "zero" all memories before kernel completion so as to not leak data to subsequent kernels. The previous recommendation should not be considered that doing so makes the use of a GPU in an untrusted environment secure, and such security analysis is beyond the scope of this thesis.

Constant Memory: This memory is a read-only region of device memory, which is accelerated via an on-chip cache. Reads from the cache are as fast as register reads as long as all addresses within a single constant memory request read from the same address. The number of threads serviced by a single constant memory request is 16, a 1/2 warp. All accesses to different addresses within constant memory for a single request are carried out serially. Thus, if threads are expected to access multiple different locations, even if the memory is read-only, it can be worth using a different type of memory such as texture or shared memory. The constant cache is referred to as constant registers in certain contexts.

Texture Memory: This address space maps to a read-only region of device memory. Reads from texture memory are cached and are not subject to the same addressing restrictions to achieve optimal performance as constant memory or global memory, so can prove a flexible alternative for accelerating device memory access. However, the cache and fetch mechanism is optimised for 2D spacial locality, thus achieving good performance with texture memory can depend on good spacial locality of address access. This locality type is as a result of its primary use with textures, where the most common pattern of texture access is to read texels that are spatially close to the currently active texel. Texture memory also provides hardware for high speed filtering of memory reads, such as interpolation between texels, however this was not found useful in the context of the cipher implementations presented in this thesis.

Global Memory: Global memory space can be allocated and used as linear memory, in the same manner that traditional CPU based applications can interact with system memory. Global memory is not cached and thus is relatively slow compared to other cached device memory spaces, however it supports both read and write operations. Also, global memory, when used as linear memory, supports pointer usage, which can be useful for certain pointer based data structures such as trees. Efficient use of global memory requires the adherence of certain access patterns. The compiler will generate a single instruction for a global memory access for a single thread if the data type is 4, 8 or 16 bytes long, and its address is a multiple of the data type size. Failing to meet these requirements can result in multiple instructions being generated, which run serially. Simultaneous access to global memory by a half warp can be coalesced into a single instruction if the following is true: each thread adheres to the access pattern previously described for generating a single access instruction; all data being accessed by the 16 threads is contiguous and aligned to the size of the data; threads access words in sequence, i.e. the n^{th} thread accesses the n^{th} word. Compute capability 1.2 removes this last requirement.

Global memory can be mapped by textures, allowing memory access to potentially benefit from full read/write flexibility and also cached reads. However, writing to global memory mapped by a texture is not necessarily reflected by the texture cache within the same kernel execution. The texture cache is not read-/write coherent, in that writes to a global memory location and subsequent reads for that same memory location via texture fetching gives undefined results. If incoherence is a potential problem it is required to alternate access to the texture and global memory across different kernel executions.

2.2.4.2 Branching

As mentioned previously, an SM unit creates and schedules threads in groups of 32 called warps. On instruction issue, the SM selects an active warp and issues an instruction to all the active threads within the warp. Thread paths within a warp can diverge when they make a conditional branch. When threads diverge, all unique code paths within a warp of threads must be executed serially. When all paths within a warp converge again, all threads return to being executed in parallel. Distinct warps execute independently regardless of code path. Thus, to maintain full efficiency of an SM, threads within a warp should not diverge. Warps are formed by the hardware in a deterministic fashion. Threads within a

CUDA block are split into warps, each warp contains threads with consecutive IDs. Thus, programmers can determine that if code paths taken by threads is consistent based on multiples of the warp size, starting at thread 0, then no thread divergence can happen. The issue of thread divergence has important consequences for asymmetric-key cryptography as shown in Chapter 6.

2.2.4.3 Thread Co-operation

As previously mentioned, threads within a CUDA block have the ability to synchronise. More precisely, all threads within a CUDA block can issue a `__syncthreads()` intrinsic that stops their progress until all threads within the block reach the same point. This intrinsic is allowed within conditional code. However, if threads within a block diverge at the point of synchronisation, the outcome is undefined, possibly resulting in a program crash. Thus, synchronisation and branching can be mixed only if the conditional branch evaluates equivalently across all threads within a block. This factor plays a role in the flexibility of asymmetric-key implementations as seen in Chapter 5. It should also be kept in mind that, as in DX9 GPUs, no global synchronisation of threads is supported. Although atomic instructions were introduced for global memory access in compute capability 1.1, which allows a certain level of global thread collaboration (for example in reduction functions [82]), the only point of synchronisation for all threads within a kernel execution, is when the kernel (i.e. all threads) is finished executing.

2.2.4.4 Occupancy

An important consideration regarding performance on a GPU, or any highly parallel processing device, is that of maintaining a high occupancy. A high occupancy refers to keeping the processing cores (e.g. the SPs in the GPU) busy doing useful work rather than waiting on I/O or other threads for completion. Much of the transistor budget on a traditional CPU is consumed with the aim of keeping the processing cores as busy as possible with the use of units such as branch predictors, re-order buffers and large caches. The GPU gains in theoretical performance by simplifying, reducing or removing these hardware components, and thus it follows that more responsibility for keeping the processing cores occupied shifts to the programmer. To maintain a high occupancy on the GPU the following points require consideration. Although not previously stated, the principles behind the considerations below also apply to DX9 hardware.

Balanced Workload: Where possible, all SMs should have the same amount of work to do. Considering that each CUDA block is assigned to a single SM and given n SMs in a device, the CUDA grid size should specify at least n blocks[1]. If one assumes the work performed per block is equal, then the number of blocks specified should be a multiple of n. Ideally, regardless of work load per block, each SM should perform an equal amount of work. This ensures no SMs are idle while awaiting the completion of work by other SMs before returning to the CPU. The assignment of blocks to SMs is undefined, however we can make certain assumptions to distribute uneven block work loads beneficially, as presented in Chapter 6.

Multiprocessor Occupancy: Nvidia define the term, "multiprocessor occupancy", to be the ratio of the number of warps running concurrently on an SM to the maximum number of concurrent warps supported by an SM. The G8X architectures support up to 24 actively running warps per SM, i.e. 768 threads, thus to be capable of 100% multiprocessor occupancy across all SMs on the GeForce 8800GTX we require at least 12,228 (768×16) threads. Executing a kernel with a high number of threads has the benefit of hiding memory latency, which increases the occupancy of the processing cores. In general it is recommended that the multiprocessor occupancy ratio is maintained as close to 100% as possible, however the following factors are involved in achieving this goal.

- The registers available within an SM are shared amongst all active threads on the SM. The number of active threads is determined by the number of active blocks per SM $\times$ the number of threads per block. Register contention can cause a reduction in both the number of active blocks and number of threads possible per block and hence reduce multiprocessor occupancy. The number of registers used per thread is kernel dependent and can be outputted by the compiler using the `-cubin` switch. The programmer has the ability to specify the number of registers to use, if it is adversely affecting the multiprocessor occupancy, using the `--maxregcount` switch. However, when reducing the number of registers used for a kernel, a performance penalty is paid for each register which spills over to local memory (i.e. device memory). The trade off between local memory spillage and multiprocessor occupancy is seen in Chapter 6.

- Another resource which can restrict multiprocessor occupancy is shared memory. It is divided between all active blocks on an SM. Again, the

[1]Nvidia DX10 compliant GPUs contain a variable number of streaming multiprocessors, ranging for example from 2 with the GeForce 8500GT to 30 with the GeForce GTX280.

amount of shared memory used is kernel dependent and can be output by the compiler using the `-cubin` switch. Relating to this point and the point above, the active block count is determined by the compiler, taking into account the number of blocks specified per grid; the number of threads specified per block; the shared memory usage per block; and the number of registers used per thread.

- For all compute capabilities as of time of writing, the maximum number of threads supported per block is 512. Thus, the active block count per SM must be greater than one to achieve full multiprocessor occupancy. Also, the maximum number of concurrent blocks supported by an SM is 8.

- Thread count per block should always be a multiple of the warp size due to the scheduler issuing threads in groups of warp size. If this is not adhered to, it is guaranteed that SPs will retire unwanted results, thus wasting compute resources.

Nvidia provides an occupancy calculator [84], which accepts as input the number of threads per block, the number of registers per thread and the amount of shared memory used per block. It outputs the level of multiprocessor occupancy. If the occupancy is less than 100%, it indicates which resource is causing the occupancy drop.

Parallelism: A concern related to occupancy is the degree of parallelism inherent in a problem. The ability to split a problem into data parallel tasks can determine the likely level of occupancy a GPU will display in the task's execution. "Embarrassingly parallel" workloads [4] naturally split into a number of parallel tasks, with little or no inter task co-operation. These types of workloads can lead to a high thread count with little to no communications overhead, the ideal for highly parallel devices such as the GPU. Tasks which don't easily divide into independent or loosely coupled units of work, can potentially be divided by increasing the level of inter task co-operation. Increasing inter task communication, in order to increase parallelism and thus the number of active threads, results in a trade off between the potential performance gain due to increased occupancy and performance loss due to the overhead in inter-thread communication. A coarse grained approach to increasing occupancy on a device when a task cannot be further subdivided, is to execute multiple instances of these tasks concurrently. This approach increases the number of concurrent threads, however also increases the requirement for large input data sizes per kernel execution, which can reduce the practicality of the approach. Both fine grained and coarse grained approaches to increase parallelism and occupancy on the GPU appear throughout the implementations in this thesis. It should be noted that security against fault attacks can involve the execution of the same task multiple times taking a vote to determine a more reliable result. This type of technique could increase the appeal of the coarse grained approaches presented if fault attack resilient approaches were implemented on the GPU.

Arithmetic Intensity: The ratio of arithmetic operations to I/O operations is referred to as arithmetic intensity. With regard to occupancy it is desirable for a kernel to have a high arithmetic intensity. As there is little hardware within the GPU dedicated to hiding memory latency, I/O heavy kernels require a large number of threads to effectively hide the I/O cost. A kernel with high arithmetic intensity reduces this pressure to have such a large number of threads as the higher the intensity, the better the arithmetic operations hide the I/O cost. It follows that blindly increasing CUDA grid and block sizes in an attempt to increase occupancy can have no effect if the kernel is not I/O bound.

2.2.4.5 Instruction Throughput

Instruction support on the GPU is not equal across different data types and operations. Regarding Nvidia DX10 hardware, 32-bit integer addition and bitwise operations takes 4 clock cycles to issue per warp, the same for 32-bit floating point add, multiply and multiply-add. 32-bit integer multiplication takes 16 cycles to issue per warp, however there is a faster 24-bit integer multiply which reduces this cost to 4 cycles. Of particular importance to certain asymmetric cryptographic algorithms, integer division and modulo operations are specified by Nvidia as “particularly costly and should be avoided if possible”. We present alternatives to avoid these costs in Chapter 6.

2.2.4.6 Data Transfer

As with DX9 hardware, transfer rates over the system PCI bus remains an impediment to performance. Recent improvements to bandwidth speeds of the system bus with the introduction of PCI Express v2.0 x16 giving a theoretical 8 GB/s still performs poorly relative to device memory bandwidth at a theoretical 142 GB/s for Nvidia’s GeForceGTX 280. It is beneficial to aim to minimise data movements from host (CPU) to device (GPU) where possible. Where multiple kernel calls are required data should remain on device memory. Transfers that are required between host and device should be batched into one transfer if possible to avoid the minimum overhead associated with per transfer call. CUDA supports using page locked memory on the host, which permits the device to execute DMA transfers. This gives the best transfer rate by avoiding an extra memory copy by the device driver. However, as page locked memory (memory which cannot be swapped out and exists in kernel-space), is generally smaller than standard system memory and shared by all OS services, there can be a negative impact to overall system performance if too much is reserved for CUDA use.

Also, as with DX9 hardware, when the CUDA API issues instructions to the GPU there is an associated overhead. This overhead is due to the combination of the relatively slow system bus and the device driver transformation of GPU calls into native instructions. These factors introduce a latency to tasks which may not otherwise exist if executed on the CPU. Thus, the types of applications which are suitable for acceleration on any GPU are generally those which have a certain degree of tolerance to latency. The GPU suits problems where each input data element requires a large amount of processing. This factor is clearly exposed in all result sections presented throughout the thesis.

2.2.4.7 Execution Model

A number of restrictions relating to the execution model employed by DX10 hardware and CUDA play a factor in how solutions for the GPU are designed. The GPU cannot be subdivided to run different kernels simultaneously, either via time slicing or concurrency. This has the effect that resource utilisation is the sole responsibility of a single kernel, in that unused resources during a kernel execution will not be consumed by other kernels. This also has the effect that implementations which have threads running largely differing algorithms during a kernel execution, must split the functionality within a single kernel using conditional branches. This technique is given the term “fat kernels”. Another restriction which should be considered is that results for a kernel execution are not available until all threads are finished. This has implications for occupancy as stated above in Section 2.2.4.4, and plays a role in load balancing of work in Chapter 5. A further restriction, that the G8X family imposes, is that kernel execution cannot happen at the same time as data transfer from host to device, or vice versa. Later architectures removed this restriction allowing the overlap of kernel execution and data transfer. This improvement goes some way to alleviating the adverse affect of latency on throughput as pipelining can be implemented.

	Shader Model 3.0	Shader Model 4.0
Instruction Slots	≥ 512	≥ 65536
Vertex Constant Registers	≥ 256	16 X 4096
Fragment Constant Registers	≥ 224	
Vertex Temporary Registers	16	4096
Fragment Temporary Registers	32	
Render Targets	4	8
2D Texture Size	2048×2048	8192×8192
Integer Operations	-	Yes
Load Operations	-	Yes
Vertex Texture Access	4	128
Fragment Texture Access	16	

Table 2.1: Comparison of select features required by DX9 Shader Model 3.0 and DX10 Shader Model 4.0 .

2.2.4.8 Tool Chain

Nvidia provide a single compiler tool, `nvcc`, which hides a chain of tools used to generate device and host binaries. Of interest is the device binary generating part of this tool chain. nvcc first splits the host and device code into separate files. A device code compiler, `opencc`, is called to generate an intermediate assembly language called PTX (parallel thread execution) from the device code. Finally an assembler, `ptxas`, generates a binary file from the PTX assembly ready for sending to the GPU driver. One restriction is that direct access to the instruction set which actually runs on the device is not made available. PTX is a virtual language, its binary representation gets further optimised and mapped to the ISA of the device by the GPU driver. This restriction, combined with Nvidia's reluctance to release low level architectural details, limits low level optimisation opportunities. Another restriction involves the inability to inline PTX assembly within CUDA files. If an advantage were found to using PTX directly, we must edit an entire kernel represented in PTX assembly. This makes it difficult to use and maintain assembly for large kernels, perhaps making it more beneficial to focus on optimisation at the C++ level.

2.2.5 Shader Model 3.0 versus Shader Model 4.0

Table 2.1 shows a comparison of a subset of features required by shader model 3.0 and shader model 4.0. The information in Table 2.1 is gathered from Microsoft and Nvidia company literature [10, 92]. The table reflects the unification of vertex and fragment shaders in DX10. The term "Render Targets" refers to the number of textures that can be written to by the output of the ROP stage. Render targets have little significance as we have seen that the unified shader can now output to device memory at any point during thread execution. “Load Operations” refers to the ability to read from device memory without using texture filtering. “Texture Access” refers to the number of textures which can be access by a kernel. All other terms have been previously introduced.

2.3 Symmetric-Key Cryptography

Symmetric-key cryptography is based on the use of a single secret key for both the encryption of plaintext messages and the decryption of ciphertext messages. In general, this type of cryptography is relatively fast compared with asymmetric-key cryptography. As such, security systems normally satisfy the bulk cryptographic requirement with the use of symmetric-key functions. A commonly employed building block of symmetric-key based security systems is the block cipher [69, ch7]. A block cipher is an algorithm which operates on input data, split into fixed lengths (i.e. blocks), and outputs a transformed version of the input. The transformation is deterministic, in that given the same inputs, the same output will be produced for each execution of the algorithm. The typical structure of a block cipher is to accept as input, either some form of primary data input such as plaintext or ciphertext and a secondary input known as the key. The combination of the primary and secondary inputs are used to determine the transformation into a single output block, which is ideally impossible to distinguish from a random block of bits of equal length.

Another type of symmetric-key building block is the stream cipher [69, ch6]. This is similar to a block cipher though differentiates itself by accepting inputs in smaller sizes and is typically used in an environment where the size of data is unknown, such as in communication devices. The distinction between block and stream ciphers is blurring as data volumes trend upwards and block ciphers in certain contexts can be used in a stream like fashion. The first widely used and analysed block cipher was the Data Encryption Standard (DES) [76]. In recent years this block cipher has been deprecated in favour of using the new Advanced Encryption Standard (AES). This block cipher is one of the most popular symmetric-key cryptographic functions and is used as the focal algorithm for the symmetric-key implementations presented in this thesis.

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$

Figure 2.7: State for AES-128: input and output for round transformations.

2.3.1 AES

The Advanced Encryption Standard (AES) was introduced in 2001 by the National Institute of Standards and Technology (NIST) in response to security concerns of DES. The standard adopted a restricted version of the Rijndael [20] symmetric block cipher that can encrypt and decrypt data in blocks of 128 bits using a key sizes of 128, 192 or 256 bits. The cipher involves the execution of a number of transformation rounds. Each round accepts a data block of 128 bits and outputs a data block of the same size. Also, each round's output is the next round's input. The number of rounds executed is determined by the key length used: 128-bit uses 10 rounds; 192-bit uses 12 rounds; and 256-bit uses 14. The AES implementations presented throughout this thesis and the following description are based on AES using a 128-bit key.

2.3.1.1 Single Cipher Round

Each round operates on units of 128 bits of data called the *State*, transforming an input State of 128 bits into an output State of the same size ready for the next round. The State, which consists of a 16 byte block, is generally viewed as a 4 x 4 array of bytes. This is shown in Figure 2.7, where $a_{i,j}$ are the bytes within the State, i indicates the row and j the column. Each round consists of a number of stages, which are responsible for transforming the State. These transformations are listed and briefly described as follows.

- *SubBytes:* performs a non-linear byte substitution using a 256×1-byte entry lookup table. Figure 2.8 illustrates an example of the substitution operation. It operates on each byte within the state independently, using a single input byte as the address within the lookup table, traditionally

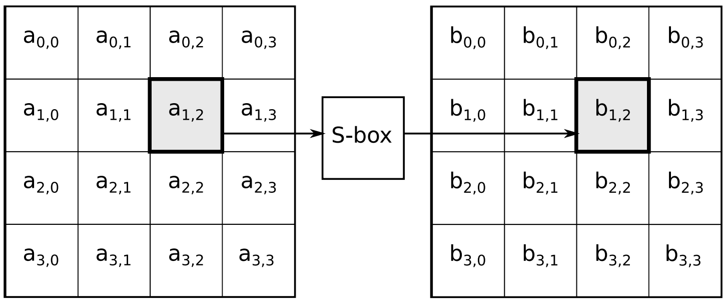


Figure 2.8: AES-128 SubBytes byte substitution using an S-box lookup.

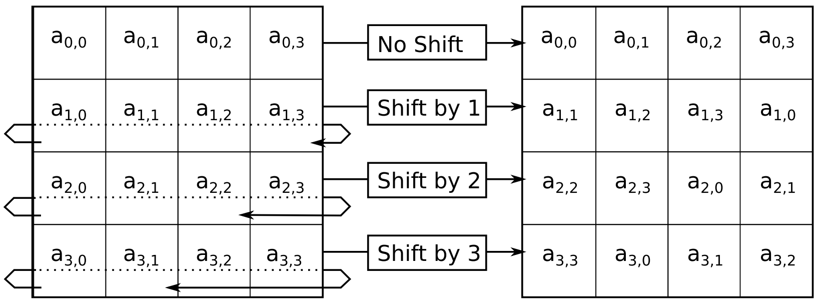


Figure 2.9: AES-128 ShiftRows round transformation stage.

referred to as a Substitution Box (S-box) [25], to retrieve a single output byte. We denote the substitution as $b_{i,j} = S[a_{i,j}]$, where $a_{i,j}$ is an input byte and $b_{i,j}$ is the corresponding output byte.

- *ShiftRows:* shifts the rows of the State cyclically to the left by differing offsets as illustrated in Figure 2.9. In the figure we use $a_{i,j}$ to denote the input and output for ease of reading. Row 0 is not shifted, row 1 is shifted by 1 byte, row 2 by 2 bytes and row 3 by 3 bytes. We denote these shifts as:

$$\begin{bmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{bmatrix} = \begin{bmatrix} a_{0,j} \\ a_{1,|j-1|_4} \\ a_{2,|j-2|_4} \\ a_{3,|j-3|_4} \end{bmatrix}.$$

- *MixColumns:* performs a linear transformation on each of the columns within the State. The linear transformation takes the form of a multiplication of each 4-byte vector with a fixed matrix in the Galois field $GF(2^8)$ [62]. Figure 2.10 shows the multiplication of a single column where $b(x) = c(x) \otimes a(x)$. More precisely the multiplication for a single column

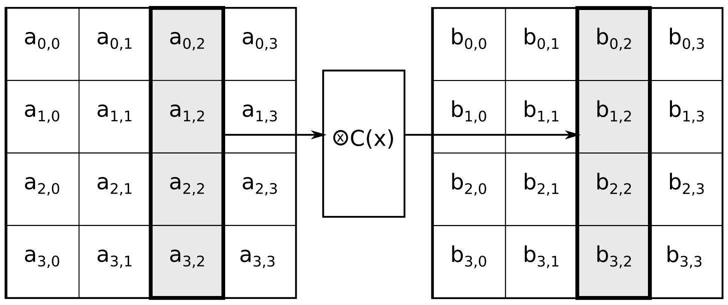


Figure 2.10: AES-128 MixColumns round transformation stage.

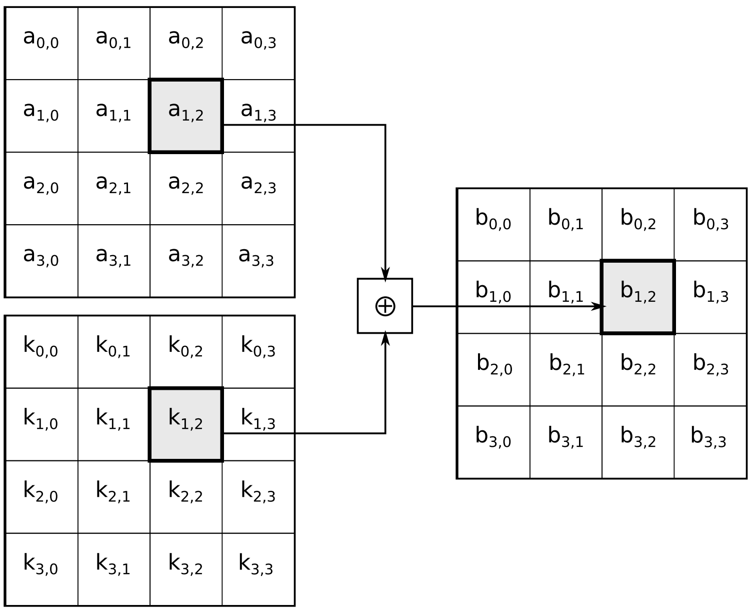


Figure 2.11: AES-128 AddRoundKey round transformation stage.

is as follows, where the constant polynomial $c(x)$ is seen in matrix form:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}.$$

- *AddRoundKey:* adds a *Round Key* (described later) to the current State. Addition of each byte is done in the Galois field $GF(2^8)$, by a bitwise XOR. An example of this is shown in Figure 2.11, where the 128-bit round key is represented in the same manner as the State, i.e. as a 4 x 4 array of bytes.

The following pseudocode describes a round and the order in which the above transformation steps are used. Each of these transformation stages are designed to be reversible, thus allowing the overall cipher to be reversible.

```
Round()
{
    SubBytes();
    ShiftRows();
    MixColumns();
    AddRoundKey();
}
```

2.3.1.2 Full Cipher

The key schedule is a generation process in which the initial cipher key, 128 bits in this case, is expanded into a sufficient number of block sized round keys. The expansion routine is deterministic, dependent solely on the initial cipher key and some constants. Each round consumes one of the round keys in the AddRoundKey stage. One additional round key is used in an AddRoundKey stage before the cipher rounds are executed. The final round execution is different from the main rounds in that it does not execute a MixColumns() transformation stage. All stages of the Rijndael cipher are listed as follows.

```
Rijndael()
{
    Key_Schedule();
    AddRoundKey();
    For(i = 1 to 9) //based on 10 round 128-bit AES
      Round();
    Final_Round();
}
```

2.3.1.3 Implementation

A technique for efficient implementation of the Rijndael cipher on 32-bit machines is the use of lookup tables [21]. Tables can be generated which combine the precomputed transformations of the SubBytes, ShiftRows and MixColumns steps for all possible byte values. These tables are constant and not reliant on the cipher inputs and thus can act as a hard coded lookup tables. To generate the tables we state the full transformation of a round for a single column as:

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = \overbrace{\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \overbrace{\begin{bmatrix} S[a_{0,j} \quad] \\ S[a_{1,|j-1|_4}] \\ S[a_{2,|j-2|_4}] \\ S[a_{3,|j-3|_4}] \end{bmatrix}}^{\text{SubBytes()}}}^{\text{MixColumns()}}_{\text{ShiftRows()}} \underbrace{\oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}}_{\text{AddRoundKey()}}.$$

The matrix multiplication, under the heading MixColumns(), can be written in terms of an XORing of vector multiplies:

$$S[a_{0,j}] \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} \oplus S[a_{1,|j-1|_4}] \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} \oplus S[a_{2,|j-2|_4}] \begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} \oplus S[a_{3,|j-3|_4}] \begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix}.$$

For each vector multiply a table can be generated by using all possible values returned by the S-box substitution stage. This gives the following 4 tables, each containing 256 entries of 4-byte vectors, where a denotes any byte value and $\bullet$ is multiplication in $GF(2^8)$:

$$T_0[a] = \begin{bmatrix} S[a] \bullet 02 \\ S[a] \\ S[a] \\ S[a] \bullet 03 \end{bmatrix}, T_1[a] = \begin{bmatrix} S[a] \bullet 03 \\ S[a] \bullet 02 \\ S[a] \\ S[a] \end{bmatrix}, T_2[a] = \begin{bmatrix} S[a] \\ S[a] \bullet 03 \\ S[a] \bullet 02 \\ S[a] \end{bmatrix}, T_3[a] = \begin{bmatrix} S[a] \\ S[a] \\ S[a] \bullet 03 \\ S[a] \bullet 02 \end{bmatrix}.$$

A lookup can be performed using these tables instead of executing the vector multiplies above. By performing an XOR of each 4-byte vector lookup result, we produce a column transformation for a round save the AddRoundKey() step. As such, we can generate a full round's output for a column as shown in Equation 2.1. These table lookups form an integral part of the AES implementations presented in this thesis. Note that the final round does not perform MixColumns() stage and as such we use the S-box directly as our lookup table for this stage. Also note that decryption can be performed in the same way as encryption, using lookup tables, however inverted S-box data is used to seed the generation of the lookup tables. Related to security, it is worth noting that the use of lookup tables is good practice, giving increased resilience against side channel timing attacks.

$$e_j = T_0[a_{0,j}] \oplus T_1[a_{1,|j-1|_4}] \oplus T_2[a_{2,|j-2|_4}] \oplus T_3[a_{3,|j-3|_4}] \oplus k_j. \qquad (2.1)$$

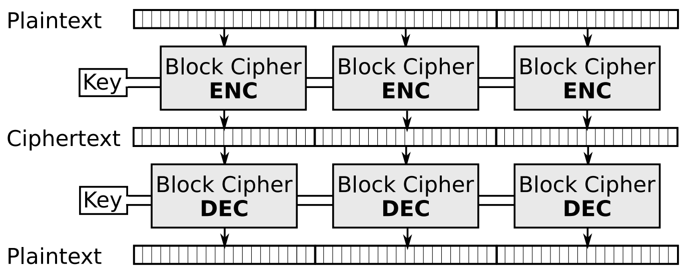


Figure 2.12: Electronic Code Book Mode of Operation: encryption and decryption.

2.3.2 Block Cipher Modes of Operations

Block ciphers are rarely used in isolation to build a secure system. They are normally used in a manner which provides confidentiality and or authentication for messages larger than a single block size. An algorithm that determines the manner in which a block cipher is used is referred to as a mode of operation [116], or simply as a *mode*. A mode is considered a standard building block for creating a secure system using symmetric-key cryptography. NIST currently approves 5 different confidentiality modes [80] which, in the context of this thesis, fall into two different groups. Firstly, modes which are based on chaining of block cipher data with little scope for parallelisation during encryption or decryption. Secondly, modes which are not inherently serial and allow a high degree of parallelisation for both encryption and decryption. We term these modes of operations as *serial modes* and *parallel modes* respectively and discuss them below. In Chapter 5 we explore these two categories of confidentiality modes and their implications for use on the GPU. Although we do not focus on modes that involve authentication, we also include observations regarding their suitability to the GPU.

2.3.2.1 Parallel Modes

Electronic Code Book (ECB) mode [77] is a simple method for using a block cipher, though is considered insecure and not recommended for use in most security systems. The mode splits a message, plaintext or ciphertext, into block sized chunks. Each chunk is passed through the block cipher independently of the rest of the message. This mode is illustrated in Figure 2.12. The advantage of this mode is that it is embarrassingly parallel, the disadvantage is its insecurity due to equal blocks encrypted with the same key produce the same output block.

Counter (CTR) mode [23] is similar to a stream cipher in that it generates a keystream independent of the message being encrypted or decrypted. A nonce is used as a form of initialisation vector for the mode whereby a stream of input data

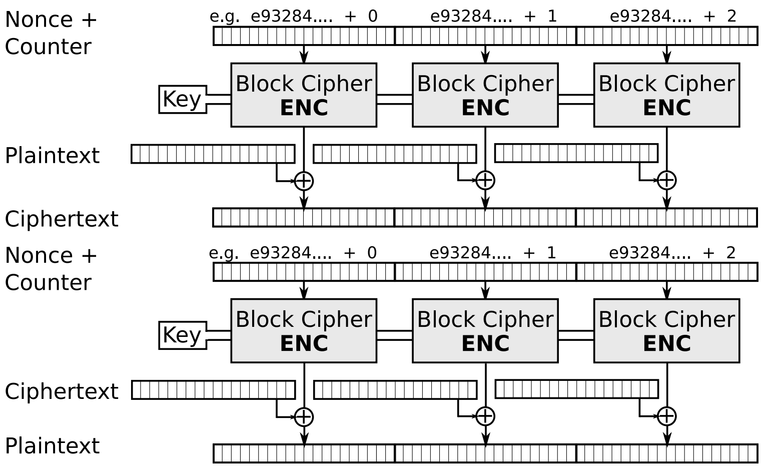


Figure 2.13: Counter Mode of Operation: encryption and decryption.

blocks is produced in a deterministic fashion based on the nonce. The stream of data should have the property that it does not repeat for a long period. A simple and common way to satisfy the data stream requirement is to use the nonce as a starting value for an incremental counter. The data stream is split into blocks and encrypted independently of the other input blocks. The message is also split into blocks and XORed with the output from the block ciphers. Figure 2.13 shows this operation. Advantages of this mode include its ease in making it embarrassingly parallel by using a counter function which can generate a unique input block independently; its ability to precompute a keystream; and as shown in Figure 2.13, the block cipher is required to operate only in a single direction, i.e. there is no need for the inverse of the block cipher.

2.3.2.2 Serial Modes

Cipher Block Chaining (CBC) mode [24] is an example of a confidentiality mode which is serial. A plaintext message is split into blocks, each block is XORed with the output of the previous block cipher output. The first block is XORed with a nonce (initialisation vector). In this way, each encryption of a block relies on having encrypted all preceding message blocks. This chain of dependence for encryption is clearly illustrated in Figure 2.14. Each block decryption relies on two neighbouring ciphertext blocks and thus can be parallelised.

Cipher Feedback (CFB) mode [77] is similar to CBC in that it involves chaining, however it XORs the plaintext after the execution of the block cipher. This allows encryption and decryption of blocks in smaller sizes than the native block size of the cipher. Also this mode only requires the use of the encrypt version

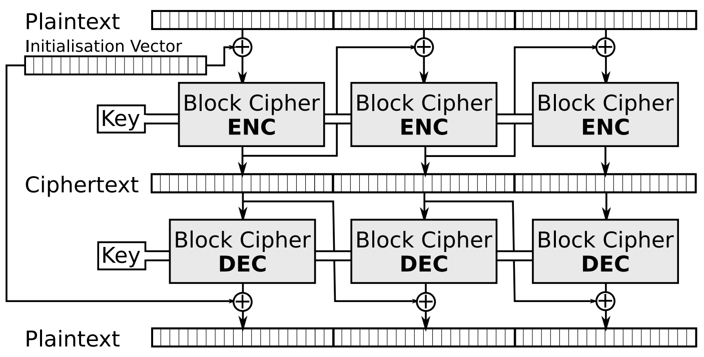


Figure 2.14: Cipher Block Chaining Mode of Operation: encryption and decryption.

of the block cipher. As with CBC, CFB encryption is a serial process. Output Feedback (OFB) mode [77] generates a keystream independent of the plaintext or ciphertext, as in CTR. It does this by chaining the output for each block cipher to the next, starting with a nonce. The plaintext or ciphertext is then XORed to the keystream to encrypt or decrypt. This has the advantage of being able to generate the keystream in advance, however neither the encrypt or decrypt can be executed in parallel.

2.4 Asymmetric-Key Cryptography

Asymmetric-key cryptography, also referred to as public-key cryptography, describe a collection of cryptographic algorithms and protocols which allow the establishment of a secure channel without an initial shared key. In contrast, symmetric-key cryptography is based on prior agreement of a shared secret upon which secure communication is based. In practice the secure sharing of a key can be difficult to achieve, and if compromised, results in an insecure channel. Asymmetric-key cryptography solves this problem through the use of secrets that are not shared between parties, but rather used privately and separately to establish a common secret between the parties. A classic example of a public-key system is the Diffie Hellman Key Exchange protocol (DH) [22], which allows two parties using independently held secret keys, to generate a shared secret key. The Rivest, Shamir, Adleman algorithm (RSA) later advanced public-key cryptography by allowing the secure transmission of a chosen message without an initial shared secret, facilitating encryption and authentication directly.

Asymmetric-key algorithms are generally computationally expensive relative to symmetric-key algorithms. Due to the computational expense of asymmetric algorithms, they are mainly used during the handshake part of a secure session

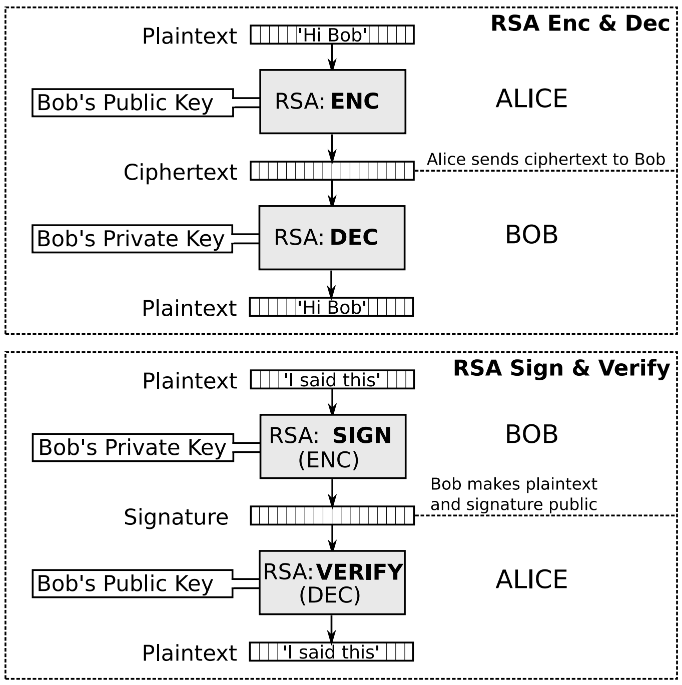


Figure 2.15: Example of Public and Private Key use in RSA.

setup, involving authentication and sharing of a secret key. This shared secret key is then used with symmetric-key algorithms to facilitate the remainder of the session's data transfer. Public-key systems, such as RSA, DH, Digital Security Algorithm (DSA) [78], Elliptic Curve Cryptographic algorithms (ECC) [9] and others, typically involve heavy use of big integer modular arithmetic. Thus, the efficient execution of modular arithmetic is the focus of many public-key cryptosystem implementations. Chapter 6 investigates the efficient execution of modular arithmetic on the GPU using an implementation of RSA as a concrete example. Below we discuss the operational details of RSA, followed by a description of the mathematical fundamentals required for the implementation techniques presented in Chapter 6.

2.4.1 RSA

The RSA cryptosystem is based on the now well established paradigm of the use of asymmetric public and private keys. A public key is used to encrypt a message, whereas the private key is required to decrypt the resulting ciphertext. If Alice wishes to send Bob a message confidentially, she encrypts the message with Bob's public key and transmits the ciphertext. Bob uses his private key to decrypt the ciphertext, thus retrieving the original message. RSA also provides a mechanism for authentication through digital signatures. A digital signature is a digital string which can be used to associate a message with an entity such as Bob. Bob can generate a ciphertext, a digital signature, by encrypting a message with his private key. The digital signature, along with the message is made publicly available. Alice can subsequently decrypt this digital signature using Bob's public key, thus generating the original message. The signature can be verified by comparing the decrypted message with the publicly available message. The relationship of public and private keys used for encryption and signing is depicted in Figure 2.15. In practice a trusted certificate authority is required to bind a public key to an identity (certification), thus providing a basis for reliable authentication.

The RSA public key consists of a modulus n and an encryption exponent e. The private key primarily consists of a decryption exponent d. n is generated such that $n = pq$, where p and q are large primes. e and d are integers generated such that the following relationship holds: $m^{ed}(mod\ n) \equiv m(mod\ n)$, where m is an integer in the range $[0, n-1]$. It is common for the private key to also contain p and q and other prime related data for computational convenience, all of which must be kept secret. RSA encryption is defined in Algorithm 2.1 and decryption in Algorithm 2.2. Functions $o2i()$ and $i2o()$ are responsible for converting a digital string in octet format (i.e. bytes) to integer format and vice versa. The encryption and decryption algorithms are generally used in the context of a padding scheme. The scheme can specify the $o2i()$ and $i2o()$ algorithms used as well as how to produce an encoded version of a message which adheres to certain security recommendations, [5], such as m not being a small value. The employed padding scheme is computationally inexpensive compared to the exponentiation part of RSA and as such is of less interest in the context of asymmetric-key acceleration. A thorough set of recommendations on a secure implementation of RSA can be found in PKCS#1 [114].

Algorithm 2.1 RSA Encryption

```
Require: RSA public key $(n, e)$; message $M$, an octet string.
  $m \leftarrow o2i(M)$
  $c \leftarrow m^e \bmod n$
  $C \leftarrow i2o(c)$
  return $C$
```

The RSA signature and verification primitives are similar to encryption and decryption algorithms, though vary in their intent. The private key is used to encrypt (called *sign* in this context) a message or its hashcode, thus generating a signature. A party wishing to verify the message as being signed by someone in possession of the private key can decrypt (*verify*) the signature using the signer's

Algorithm 2.2 RSA Decryption

```
Require: RSA key pair: $(n, e)$, $(d)$; ciphertext $C$, an octet string.
  $c \leftarrow o2i(C)$
  $m \leftarrow c^d \bmod n$
  $M \leftarrow i2o(m)$
  return $M$
```

public key. This produces a message or its hashcode, which can be compared to the original for verification. We do not cover the topic of key generation or the theory behind RSA as it not in scope here, plenty of material can be found on this, for example [18]. From the encryption and decryption algorithms detailed, we can clearly see that the bottleneck lies in the execution of modular exponentiation operations. Typically the RSA decryption exponent is generally large, from 1024 bits for common use up to 4096 bits. In contrast the encryption exponent is generally set to small values such as 65537. Thus, by far the most computationally expensive RSA primitives are decrypt and sign, i.e. those using the decryption exponent (private key).

2.4.2 Modular Arithmetic Fundamentals

2.4.2.1 Basics

Given integers a, b and n, we can say "a is congruent to b modulo n" if $a - b$ is an integral multiple of n, that is $n|(a - b)$. This relationship is written as:

$$a \equiv b (\mathrm{mod}\ n)$$

where n is referred to as the modulus. We define the binary operation, modulo as:

$$a \bmod n = a - n\lfloor a/n \rfloor, \text{ if } n \neq 0; a \bmod 0 = a$$

where $\lfloor x \rfloor$ is the maximum integer i such that $i \leq x$. During this thesis, for convenience we often use the shorthand notation for the modulo operation, $|a|_n$, which is equivalent to $a \bmod n$. Addition, subtraction and multiplication modulo n can be performed in the conventional way, for example $||a|_n + |b|_n|_n = |a + b|_n$. Division modulo n is performed as multiplication by the multiplicative inverse of the divisor. More precisely, $|a/b|_n = |ab^{-1}|_n$, where b^{-1} is an integer such that $|bb^{-1}|_n = 1$. Here, b^{-1} if referred to as the multiplicative inverse of b modulo n. The inverse of an integer modulo n does not always exist, in such a case division is undefined. It is worth noting that an inverse always exists for $b \bmod n$ if b

is relatively prime to n. That is $gcd(b, n) = 1$ where $gcd()$ returns the greatest common divisor of the two arguments.

2.4.2.2 Chinese Remainder Theorem

Given pairwise relatively prime integers $n_1, n_2, ..., n_k$ and a system of simultaneous congruences,

$$\begin{aligned} b &\equiv a_1 (\text{mod } n_1) \\ b &\equiv a_2 (\text{mod } n_2) \\ &\vdots \\ b &\equiv a_k (\text{mod } n_k) \end{aligned} \tag{2.2}$$

there is a unique solution b modulo $N = \prod_{i=1}^{k} n_i$. We can generate this unique solution using the known constructive proof [58]:

$$b = \sum_{i=1}^{k} (|a_i|_{n_i} |N_i^{-1}|_{n_i} N_i) \text{ mod } N \tag{2.3}$$

where, $N_i = N/n_i$.

2.4.2.3 Integer Representation

Integers can be represented in multiple ways. We detail the following representations which are referred to in this thesis.

Radix Representation: The radix representation is how numbers are normally represented. For example the number 124 has an implicit radix (or base) of 10 and the value of the number is derived from the position of the digits in the manner, $1 \times 10^2 + 2 \times 10^1 + 4 \times 10^0$. More formally, given radix r, a positive integer a can be written uniquely as:

$$a = a_{n-1} r^{n-1} + a_{n-2} r^{n-2} + ... + a_1 r + a_0$$

where a_i is in the range $[0, r - 1]$ and n is the number of *limbs* in the representation. If $n = 1$, a is termed a *single-precision integer*, or a *multiple-precision integer* if $n > 1$.

Mixed Radix Representation: This is a representation where each digit position of a number can be associated with an arbitrary weight, rather than using a conventional weight based on a geometric progression based on a single radix (as in Radix Representation). For example, an integer a can be written as:

$$a = a_{n-1}\prod_{i=1}^{n-2} r_i + ... + a_2(r_1 r_0) + a_1(r_0) + a_0.$$

To make this representation useful, it is normal for restrictions to be placed on the relationship of weights and digits used.

Modular Representation and Residue Number Systems: Given an integer a and a set of relatively prime integers $n_1, n_2, ..., n_k$, we can use modular representation to depict a as a sequence of residues:

$$< a >_n = (|a|_{n_1}, |a|_{n_2}, ..., |a|_{n_k})$$

where, the set $n_1, n_2, ..., n_k$ is called the base. This sequence of residues can be rewritten to form the simultaneous equations according to the Chinese Remainder Theorem (CRT) see Equation 2.2. Thus we know that there is a unique mapping of an integer a to $< a >_n$, i.e. $(|a|_{n_1}, |a|_{n_2}, ..., |a|_{n_k})$, where $0 \leq a < N$ and $N = \prod_{i=1}^{k} n_i$. The set of all modular representations of the integers a within the range $0 \leq a < N$ is called a Residue Number System (RNS). The range $0 \leq a < N$ is called the dynamic range of the system.

A useful feature of an RNS is addition, subtraction and multiplication of multiple-precision integers can be broken into parallel units of work, where each limb is operated on independently of the other limbs. We can carry out these operations as follows:

$$\begin{aligned}
< a >_n + < b >_n &= (||a|_{n_1} + |b|_{n_1}|_{n_1}, ||a|_{n_2} + |b|_{n_2}|_{n_2}, ..., ||a|_{n_k} + |b|_{n_k}|_{n_k}),\\
< a >_n - < b >_n &= (||a|_{n_1} - |b|_{n_1}|_{n_1}, ||a|_{n_2} - |b|_{n_2}|_{n_2}, ..., ||a|_{n_k} - |b|_{n_k}|_{n_k}),\\
< a >_n \times < b >_n &= (||a|_{n_1} \times |b|_{n_1}|_{n_1}, ||a|_{n_2} \times |b|_{n_2}|_{n_2}, ..., ||a|_{n_k} \times |b|_{n_k}|_{n_k}).
\end{aligned}$$

This allows, for example, $|ab|_N$ to be calculated using $< a >_n \times < b >_n$. Also, the integers a and b and the results of the $+, -, \times$ operations can be uniquely mapped to the RNS so long as a, b and the results fall within the dynamic range of the system. The ability to perform these operations in such a fashion can be advantageous on a highly parallel device such as a GPU. There are a number of disadvantages associated with representation using an RNS. RNS is not a weighted number system, so there is no easy way to make size comparison of two numbers. It is also difficult to detect an overflow as a result of an operation, i.e. if the dynamic range of the system has been exceeded. Division is also difficult to perform, limiting the usefulness of RNS for general purpose arithmetic. A method used to overcome these restrictions is to convert a number in RNS into a weighted number system. This conversion can be done using Equation 2.3 or using a more optimised approach such as Garner's algorithm [31]. We look into CRT conversion in more detail in the context of modular exponentiation performed in an RNS in Chapter 6.

2.4.2.4 Montgomery Reduction

Modular reduction, the calculation of $|a|_n$, can be computationally expensive, especially when dealing with large multiple-precision integers. The reason for this is that the classical approach to reduction is to perform division, which is not a fast operation when compared to addition or multiplication. A popular method for fast modular reduction without division is Montgomery reduction [74]. Given a positive modulus n and integers a and R, where $0 \leq a < nR$, $gcd(n, R) = 1$ and $R > n$, Montgomery reduction produces the output $w \equiv aR^{-1} (\text{mod } n)$, where $w < 2n$.

Algorithm 2.3 Montgomery Reduction

```
Require: Integers $a, n, R$, where $0 \leq a < nR$, $\gcd(n, R) = 1$ and $R > n$.
1: $s \leftarrow a(-n^{-1})(\text{mod } R)$
2: $u \leftarrow sn$
3: $w \leftarrow (a + u)/R$
4: return $w$
```

Algorithm 2.3 states the steps required to produce w. Here a is the integer we are trying to reduce modulo n. R is an integer which we can choose freely save that it meets the stated criteria. As we can see there is a divide operation using R as the divisor. Thus, R is typically chosen to be a power of 2, so that on an radix-2 based computer, this divide can be achieved using an efficient shift operation. We can see that $w \equiv aR^{-1} (\text{mod } n)$ because line 3 in Algorithm 2.3 can be written as $w \leftarrow a/R + sn/R$, which when reduced modulo n, the right hand disappears, leaving aR^{-1}. Also, for this algorithm to work $(a + u)$ must be a multiple of R. We can see that is true if we assume that $(a + u) \text{ mod } R = 0$. This can be rewritten to $(a + sn) \text{ mod } R = 0$, and solving for s, we see $s \equiv -an^{-1} \text{ mod } R$. This is the calculation performed in line 1. Lastly, we see $w < 2n$ as it is equal to $w = (a + sn)/R$, where $a < nR$ and $s < R$. So, $w < (nR + nR)/R$, i.e. $w < 2n$.

Concerns: A number of issues are immediately apparent with this approach. First, it does not calculate $|a|_n$, but rather $|aR^{-1}|_n$. Second we require the potentially expensive inverse calculation of $|-n^{-1}|_R$ to perform the algorithm. And third, we require that the input value $a < nR$. These issues are addressed when we consider Montgomery reduction in the context of multiplication and exponentiation.

Montgomery Multiplication: This is the slight modification to the reduction function to perform modular multiplication. We first calculate $a = xy$, where we restrict $x, y < n$ to guarantee $a < nR$. Thus, we use Montgomery Multiplication to calculate $w \equiv xyR^{-1}(\text{mod } n)$, where $w < 2n$.

Montgomery Exponentiation: This is the use of Montgomery multiplication to execute modular exponentiation. Considering the calculation of $|a^2|_n$ using Montgomery multiplication produces $|a^2R^{-1}|_n$, it is difficult to reuse this value to further calculate the powers of a modulo n. However, if we first convert a into Montgomery representation, $a' = |aR|_n$, we find squaring using Montgomery multiplication produces $|a'a'R^{-1}|_n = |aRaRR^{-1}|_n = |a^2R|_n$. This is equivalent to the Montgomery representation of $|a^2|_n$. In effect the additional R in Montgomery representation stops the R^{-1} from "accumulating". In general we can repeatedly apply this to produce $|a^kR|_n$. To generate $|a^k|_n$ we execute a final Montgomery multiplication by 1, $a^kR \cdot 1$. In the context of modular exponentiation, the first two concerns raised previously dissipate when k is sufficiently large, i.e. the savings of not executing division outweigh the once off overheads. Regarding the third concern, we can modulo n the input base a before exponentiation, thus satisfying the quoted restriction for Montgomery reduction $a < nR$ (or $a^2 < nR$ in the case of Montgomery exponentiation).

2.4.2.5 Exponentiation

Montgomery exponentiation looks at ways of improving modular exponentiation by reducing the cost of a single modular multiplication. Another method to improve modular exponentiation, and that of normal exponentiation, is to reduce the number of multiplies required. A naive implementation of exponentiation involves the repeated multiplication using the base, for example a^k requires $k-1$ multiplies. There are many improvements to this approach which are documented in the literature [69]. Here we present a classic improvement to the naive approach, and also describe Sliding Window Exponentiation, which is based on this improvement and used in Chapter 6.

Left-to-right Binary Exponentiation: An insight into the exponentiation problem can be made which removes the need to execute $k-1$ multiplies. Given a, we can see that repeated squaring of this number leads to a doubling of its exponent, $a \cdot a = a^2 \cdot a^2 = a^4$.... Also, we see that repeated multiplication by a leads to an increment of the exponent, $a \cdot a = a^2 \cdot a = a^3$.... Thus, the naive approach "reaches" the exponent, by incrementing by 1 repeatedly. We can see that it requires less steps to reach the exponent by a combination of doubling and increments, and hence exponentiation can be achieved more efficiently by squaring and multiplying, giving rise to algorithms called *repeated square-and-multiply*. We look at left-to-right binary exponentiation, Algorithm 2.4, where the exponent is viewed as radix-2 representation and is traversed from most significant bit to least, squaring and multiplying where appropriate.

Algorithm 2.4 Left-to-right Binary Exponentiation [69, ch14]

```
Require: Integer $a, k = (k_p, k_{p-1}, ..., k_0)_2$, where $k \geq 1$
  $A \leftarrow 1$
  for $i = p$ down to 0 do
    $A \leftarrow A^2$
    if $k_i = 1$ then
      $A \leftarrow A \cdot a$
  return $A$
```

Sliding Window Exponentiation: We can see that binary exponentiation iterates through the exponent a single bit at a time. If we view the exponent in representation $k = (k_p, k_{p-1}, ..., k_0)_{2^n}$, we can substitute the squaring step of Algorithm 2.4 with A^{2^n} and substitute the multiplication step with an unconditional $A \cdot a^{k_i}$. This new algorithm, termed *Left-to-right n-ary exponentiation*, saves on the average number of multiplications performed, assuming the multiplicands, a^{k_i} are precomputed. Extending this, instead of treating n as a fixed bit width, we can use a variable bit width to further save on multiplications. This variable bit width approach is called Sliding Window and is detailed in Algorithm 2.5. Here w can be interpreted as the maximum bit width. The precomputation of values used for multiplication are detailed in lines 1 through 3. Apart from a^2, only the odd powers of a are generated. This is due to the manner in which the bitstring is selected in line 9, ensuring that only odd powers are used in multiplication, a^2 is only used during precomputation.

Algorithm 2.5 Sliding Window Exponentiation [69, ch14]

```
Require: Integer $a, k = (k_p, k_{p-1}, ..., k_0)_2$, where $k_p = 1$, and an integer $w \geq 1$.
1: $a_1 \leftarrow a, a_2 \leftarrow a^2$
2: for $i = 1$ to $2^{w-1}$ do
3:     $a_{2i+1} \leftarrow a_{2i-1} \cdot a_2$
4: $A \leftarrow 1$, $i \leftarrow p$
5: while $i \geq 0$ do
6:     if $k_i = 0$ then
7:         $A \leftarrow A^2$, $i \leftarrow i - 1$
8:     else
9:         Find the longest bitstring $k_i, k_{i-1}...k_l$ such that its length $len \leq w$ and
           $k_l = 1$
10:        $A \leftarrow A^{2^{len}} \cdot a_{(k_i,k_{i-1}...k_l)_2}$, $i \leftarrow l - 1$
11: return $A$
```

2.5 Related Work

Efficient cryptographic implementations on non-traditional general purpose processors continues to be an active research area. Recent SSL [29] and ECC implementations using the vector processors in IBM's Cell Broadband Engine for Sony's Playstation 3 were implemented by Costigan and Scott [17], and Costigan and Schwabe [16]. Field Programmable Gate Arrays (FPGA) have been extensively used for various cryptographic functions. A comparative survey of a large number of symmetric-key and hash algorithm implementations on FPGAs was presented by Jarvinen et al. [51]. Another survey regarding the use of FPGAs in cryptographic acceleration was presented by Wollinger et al. [127]. This survey covers both symmetric-key and asymmetric-key cryptosystems. Of particular interest to the security community is high performing AES implementations on a variety of hardware platforms. A large survey covering AES implementations on Smart Cards, RFID tags, FPGAs, ASICs, USB Devices and PCI based co-processors is presented in [26]. GPU usage in cryptography is a recent phenomenon and as such no comprehensive survey of related work exists. The following aims to list and briefly describe all such works at the time of writing.

2.5.1 Cryptography and Graphics Hardware

2.5.1.1 Symmetric-Key Cryptography

Non-Commodity: Until 2007 there had been little use of graphics processing technology in the field of cryptography due to its previously poor suitability to the problem space. This was due to its lack of one or more of processing power, programmability and integer support. The first reported use of graphics hardware in the field of cryptography was by Kedem and Ishihara [54] in 1999. Here they use the PixelFlow [96] architecture for brute force cryptanalysis of 40-bit RC4 and UNIX passwords. The PixelFlow architecture is a specialised, non-commodity, highly parallel SIMD machine suited to graphics processing. The PixelFlow arrangement used by Kedem and Ishihara in their study consisted of 147,456 8-bit processors running at 100 MHz and was capable of 40-bit RC4 key retrieval in an average time of 3.25 hours. It was also capable of 24 Million UNIX password checks (based on DES) per second.

Commodity, Pre-DX9: The first attempt to use a commodity GPU for the acceleration of any cryptographic primitive was an implementation of AES made in 2005 by Cook et al. [15]. The imaging subset of the graphics pipeline was used to execute the AES table lookups referred to in Section 2.3.1.3. The imaging subset is a fixed function part of the pipeline which allows the construction of colour maps. These colour maps were used by Cook et al. to simulate XOR instructions within the GPU. They presented a successful implementation of AES though the reported peak throughput speed of 1.53 Mb/s. The main reasons for the relatively poor throughput speeds were due to the restrictive feature set available within graphics hardware at that time. For example, there was no ability to program the most powerful components within the GPU (fragment and vertex processors) and thus the reliance on the underpowered imaging subset. The most advanced graphics processor used was the Geforce 3 Ti200, which is pre-DX9.

Cook et al. [14] published a related paper to this work which proposed the use of graphics processors as part of building a secure video transmission system within a mostly untrusted environment. Here they posit that the trusted computing base for visual data can be confined to just the GPU, i.e. excluding the operating system, on an untrusted client. Graphical information is sent encrypted from a remote source to the client. This graphical data is decrypted on the GPU and displayed without making available plaintext or keying information to the CPU. For this the available commodity GPU requires modification. The GPU requires the secure storage of a public-private key pair which can be used to establish a secret key with a foreign party. Also the GPU requires the ability to restrict read access to certain parts of its memory, for example the CPU should not be able to read from the area which stores the decrypted plaintext data. The prototype described in this paper implemented parts or all of the used cryptographic functions, RC4 (the stream cipher used to generate the keystream for video decryption) and RSA decryption, on the CPU as it was not practical to implement them on the GPU. This last restriction is no longer true, recent GPUs are now capable of implementing these functions.

Commodity, DX9: Most reported DX9 symmetric-key implementations are AES based. These implementations were published over a small time period and as such do not directly build upon each other, but rather present independent works. This is a similar trait with regard to DX10 symmetric-key implementations covered later. One of the main factors regarding the AES performance of the implementations listed is whether the lookup tables presented earlier in Section 2.3.1.3 were used. Another important factor relating to AES on DX9 hardware is the implementation strategy for the simulation of XOR. We group the implementations into those that used the AES lookup tables and those that did not. The following implementations did not use the lookup tables, but rather implemented each stage within an AES round transformation separately:

- Rosenberg [112] integrated a DX9 AES implementation into the OpenSSL library using its engine support. The implementation was run on a GeForce 6800GT and written in OpenGL. No objective rates were provided, however relative rates to OpenSSL running on the CPU are reported. The implementation executed 15.4 times slower than OpenSSL on a CPU, using a payload size of 16 MB. The implementation did not avail of the native XOR instruction within the ROP section of the pipeline, but used a 256×256 entry lookup table to carry out the XOR operation one byte at a time.

- Seshadrinathan et al. [118] implemented AES on the GPU to allow ciphertext to be sent to the GPU and for the plaintext to be displayed directly onto the display device. A system for displaying the decrypted ciphertext was presented. The implementation was on a DX9 GPU, the GeForce 7800, and achieved a rate of 426.1 Mb/s with a payload size of 32 MB. Again, a 256×256 entry lookup table was used to carry out the XOR operation.

- Pilkington et al. [103] presented an AES implementation on a GeForce 7900GT GPU and achieved a rate of 12 Mb/s. The same 256×256 entry lookup table XOR approach was used here.

The following implementation incorporated the use of lookup tables:

- Yang and Goodman [129] presented various implementations of AES. One of these implementations was a standard block based implementation using

AMD's X1950 XTX GPU. Yang and Goodman are the only demonstration of AES running on an AMD GPU. The implementation used a "hybrid" 4-bit table lookup to simulate the XOR operation on the GPU's floating point processors. They reported peak throughput rates of 801 Mb/s.

Commodity, DX10: With the release of DX10 GPUs, and integer support, there was heightened interest in the implementation of symmetric-key cryptography. Most implementations involved attempts to accelerate AES. Again, one of the main determinants in the resultant AES performance is the use of AES lookup tables. The XOR approaches used in DX9 implementations are no longer necessary as native XOR is supported in the main ALUs of the GPU. Also, we see that the complex memory system with the GPU requires careful use so as not to lose performance. We group the implementations into those that based on the use of AES lookup tables and those that are not. The following are implementations that do not use the lookup tables:

- An AES encryption and decryption implementation on Nvidia's G8X architecture was presented by Yamanouchi [128]. This implementation was based on OpenGL, where the author is concerned with the performance of vertex and fragment stages separately. The approach presented achieves a peak encryption throughput rate of 760 Mb/s using the fragment processing stage of the OpenGL pipeline. This is achieved by sending and receiving data to and from the GPU in batches of 1 MB. It is noted that decryption performs at the same rate as encryption as the steps involved are computationally equivalent. The GPU used in this publication was a GeForce 8800GTS.

- Rosenberg [112] also presented an implementation of AES using CUDA on a Geforce 8800GTS DX10 GPU. This implementation was integrated into the OpenSSL library and no objective throughput rates are provided. It was reported that the DX10 implementation executed at 1.0365 faster than the CPU using a minimum payload size of 9,830,400 bytes.

The following list the AES DX10 implementations that used AES lookup tables:

- Yang and Goodman [129] use AMD's first DX10 compatible architecture to investigate different implementations of AES. Though they concentrate on a bitslicing implementation of AES, they also provide some details on a standard block based implementation. Using a lookup table based implementation they report rates of 3.25 Gb/s on an HD 2900 XT. The paper does not indicate if these rates include data transfer across the system bus.

- Another notable implementation of AES on a GPU was presented by Manavski [65]. A GeForce 8800GTX, DX10 compatible, GPU is used. A rate of 2.5 Gb/s using a payload size of 8 MB is achieved. If data transfer is not included, i.e. the payload transfer is omitted, then a rate of 8.27 Gb/s is reported. The key schedule is executed on the CPU and stored in shared memory. However, round keys are accessed using a constant address across threads, so constant memory would suit key storage better. The AES lookup tables are stored in constant memory. Threads running on a single SM will concurrently access disparate table locations, which does not suit constant memory as it is single ported and thus serialises concurrent access to different addresses. Shared memory would much better suit the lookup table storage. Shared memory is used to store all intermediate results within a round. It is not clear why the SM registers are not used for this task.

Bitslicing: as mentioned previously, Yang and Goodman [129] presented a bitslicing implementation of AES. This was executed on an AMD DX10 HD 2900 XT GPU and the CTM programming infrastructure. Bitslicing [8] is a technique which involves the transposition of groups of input blocks into a more convenient representation for a processor. They show rates of 17.2 Gb/s processing throughput for their bitsliced AES implementation. These rates do not apply to the normal use of AES, i.e. the encryption of plaintext or decryption of ciphertext. Bitslicing requires a transposition of data overhead, which is not included in the above figures. Also, data transfer is not included in the above rates. The bitslicing techniques presented are useful for key searching algorithms, such as a brute force attack, where the transposed input data can be efficiently generated dynamically.

Non AES: other, non AES, symmetric-key block cipher implementations include an implementation of the ARIA [60] block cipher by Yeom et al. [130]. This block cipher is relatively new and adopted by the Korean Agency for Technology and Standards. They presented an implementation on Nvidia's GeForce 8800GTS (DX10) GPU using CUDA. A throughput rate of 4.8 Gb/s is reported, which is claimed to be the fasted implementation of ARIA. Yang and Goodman [129] also present a bitsliced approach for DES on AMD's first DX10 GPU. They achieved a peak throughput rate of 32.4 Gb/s, proving much faster than a CPU with sufficient data. An RC4 [109] implementation is presented by Boeing [11]. Nvidia's GeForce 9800GX2 (DX10) GPU is used in combination with the CUDA infrastructure. A rate of 6.5×10^6 RC4 key searches per second is reported.

2.5.1.2 Asymmetric-Key Cryptography

The first GPU implementation within the field of asymmetric-key cryptography was presented in a paper by Fleissner [27] concerning the acceleration of Montgomery exponentiation. A radix based pencil and paper [58] multiplication approach was used. Nvidia's GeForce 7800GTX (DX9) GPU was employed, which does not have integer support, resulting in the use of 24-bit limb sizes. Also, the restrictive programming environment forced the implementation to separate the execution of an exponentiation into multiple kernel calls, each instance calculating a single modular product. The execution of multiple kernels to calculate a single modular exponentiation adds overhead to the total run time. The implementation restricted the size of the modulus and base to 192 bits. The paper presents its results in terms of ratio comparisons with its own CPU implementation, and does not state the rates of processing. For example, it reports that the GPU implementation can result in a 168 times speed up compared to a CPU implementation. However, no throughput rates are given for either.

The same GPU, the GeForce 7800GTX (DX9), was used by Moss et al. [75] to implement 1024-bit modular exponentiation. Here they based their approach on Montgomery exponentiation using integers represented in a residue number system (RNS). This paper demonstrated the feasibility of a public-key implementation on a GPU, by achieving 1024-bit modular exponentiation throughput rates of 5.7 ms/op, or 175.4 exponentiations per second. Due to using a DX9 GPU, like Fleissner, Moss et al. implemented a single exponentiation using multiple kernel executions, one per modular product. They report, due to difficulties concerning memory management, a sliding window approach was not possible, resulting in binary exponentiation being used instead. Also, a process known as base extension, see Section 6.3.1, was based on using a mixed radix system rather than using a more efficient Chinese Remainder Theorem based approach.

A more recent contribution, from Szerwinski and Güneysu [121], presented implementations of 1024-bit and 2048-bit modular exponentiations based on both radix and RNS integer representation. All implementations were programmed using CUDA and executed on a DX10 compliant GPU, Nvidia's GeForce 8800GTS. The maximum throughput achieved was via a radix based approach, resulting in 833 1024-bit modular exponentiations per second. However, associated with this throughput is a minimum latency of 6.9 seconds. Due to this high latency the paper concludes that the GPU's maximum throughput can be achieved only in contexts where latency is irrelevant and thus its usefulness is quite restricted. The RNS approach displayed better latency characteristics, though with a reduced throughput rate of 439.8 1024-bit modular exponentiations. Also covered in this paper is an implementation of point multiplication operations for ECC over the prime field P-224 [81]. Here they achieve 1412.6 elliptic curve point multiplications per second. Bernstein et al. [7] subsequently published work which focused solely on GPU acceleration in the area of elliptic curves. Here they primarily focused on techniques for implementing the elliptic-curve method of integer factorisation. They also use their developed techniques for integer factorisation to contrast with results presented in [121]. They reported 2414 280-bit elliptic-curve point multiplications per second for a general 280-bit modulus on a GeForce 8800GTS.

2.5.1.3 Other

An early paper by Jacob and Brodley [50] focused on the use of GPUs to accelerate a problem within the space of computer security. They used OpenGL to program an Nvidia 6800GT (DX9) GPU to accelerate the Snort [122] network intrusion detection system (IDS). The most computationally expensive part of the IDS is signature matching. That is, given a string, its comparison with a set of black listed strings. This string matching was offloaded to the GPU. Their implementation, Pixelsnort, under certain loads from the IDS outperformed Snort by up to 40%. Vasiliadis et al. [124] demonstrated a similar work using an Nvidia GeForce 8600GT (DX10) using CUDA. The also accelerated Snort, called Gnort, achieving a factor of two speed up using the GPU compared to the CPU. They reported a maximum traffic processing throughput of 2.3 Gb/s while monitoring real traffic using a standard Ethernet interface. Data was grouped into payloads consisting of 1024 network packets of 800 bytes each.

Hashing algorithms have also received some attention. Tzeng and Wei [123] presented a method to generate white noise on the GPU using MD5 [110] as a pseudo-random number generator (PRNG). White noise (i.e. random numbers) is useful in graphics applications for certain applications, such as fractal terrain generation. Traditionally this white noise has been "pre-baked" on the CPU and transferred as textures for use on the GPU. However, for large sources of white noise, this use of memory can become a bottleneck, and thus it is desirable for the GPU to be able to dynamically generate its own white noise. The paper showed that MD5 provides a good order independent PRNG, which can run on the GPU using hardware generated texture co-ordinates as input values to the MD5 hash function. An Nvidia GeForce 8800GTX was used and achieved a rate of 2^{20} MD5 128-bit hashes in 6.3 milliseconds. It is also worth noting that there are commercial entities currently selling software packages for password retrieval using GPUs giving reportedly large increases in performance over CPU approaches. The password retrieval process in these applications is generally based on brute force attacks on hashing functions.

Chapter 3

GPU Data Transfer

Traditional use of GPUs typically does not put great demands on data transfer rates across the system bus. Furthermore, if demands are put on the system bus, they tend to be a result of data *download* (CPU to GPU) and not *readback* (GPU to CPU). In contrast, general purpose processing applications running on the GPU can require large amounts of data transfer across the system bus in both directions. Specifically, symmetric cryptography can require the processing of large amounts of data that require equal amounts of download and readback. Asymmetric cryptography tends to have less of a requirement for data transfer compared with symmetric cryptography. This is due to the comparatively high arithmetic intensity and large processing requirement of the functions employed within asymmetric cryptosystems. As stated previously, see Section 2.4, practical cryptosystems employ asymmetric systems for initial exchange of a secret key for session creation. The bulk session data is secured using symmetric functions.

Data transfer and kernel execution on a DX9 and early DX10 GPUs cannot be overlapped, and thus the time required to transfer data to and from the card cannot be hidden from final throughput rates. Also, the data transfer rates across the system bus are slow when compared to the access rates of device memory. To compound this, the readback rates for DX9 GPUs tend to be even slower than their download rates as hardware and driver focus is on the most in-demand transfer direction. Because of these factors and the bi-directional and potentially large data requirements of symmetric cryptography, one of the prime performance bottlenecks to accelerating symmetric functions is transfer speed. During the research into achieving optimal transfer rates for AES on DX9 hardware using OpenGL, we discovered that there was no tool in existence that could fully investigate the graphics' pipeline configuration states and their effect on data transfer performance. We also discovered that achieving high transfer rates using OpenGL is a complex task with many pitfalls. To address this, we developed tools to support the investigation of such states to determine the best configuration for our DX9 symmetric-key implementations. The tools also have applicability to any GPU based application that requires large data transfer using OpenGL. The tools can be used to help find the OpenGL pipeline configuration that optimizes transfer rates for any given application. This chapter presents these tools and their findings. This chapter also discusses how optimising transfer rates is a much more simplified process for DX10 hardware using CUDA.

3.1 OpenGL Imaging Pipeline

Achieving optimum bi-directional data transfer performance is a complex task involving a large array of configuration states and transfer methods. Slight misconfigurations can lead to significant drops in transfer performance, which can be unexpected and difficult to diagnose. We developed two tools, one focusing on optimising download transfer rates, the other on readback transfer rates. They are both written using the OpenGL API and are applicable to applications written using OpenGL, such as the work presented in the following chapter. The tools allow the creation of typical application transfer *scenarios* by providing full control over relevant configuration states related to the OpenGL imaging pipeline. Example states include texture dimension, system memory alignment, transfer method, external and internal texture formats, data type, texture type, the use of fragment buffer objects, pixel buffer objects and the use of fragment programs or fixed function. By providing the ability to have full control over all the relevant configuration states, the user can quickly simulate the majority of application transfer scenarios thus allowing the exploration of relevant states and their performance implications.

The imaging pipeline is part of the OpenGL specification, which concerns the movement of data to and from the GPU. Regarding download, data first resides in system memory, unpacked (read) according to the pipeline configuration, potentially converted to a data format ready for the GPU, sent to the GPU for potential further conversion, and finally written to GPU device memory mapped as a framebuffer or texture. Readback of data is much the same in reverse, ending with data packed (write) into system memory. OpenGL provides a core specification along with a large number of extensions provided by card vendors and the OpenGL Architecture Review Board (ARB). A relevant extension is the Framebuffer Object extension (FBO) [98], which supports a texture being bound to, and used as, the active framebuffer. This extension facilitates render to texture as previously described. Another relevant extension is the Pixel Buffer Object extension (PBO) [99]. This extension permits the use of DMA transfers, which can speed up data transfer and also permits non-blocking readback commands, i.e. the overlap of CPU computation with GPU data transfer.

Other factors concerning the imaging pipeline, which are controlled by the tools presented, are briefly described as follows:

- **Transfer Method:** Data download is triggered using the `glDrawPixels()` function or the `glTexImage()`/`glTexSubImage()` family of functions. `glDrawPixels()` is used to draw pixels existing in system memory to the active framebuffer. This can either render to screen or in combination with an FBO can render to a texture. `glTexImage()` and `glTexSubImage()` are used for transferring all or a portion of a system resident pixels to a texture in GPU memory.

- **External Data Format:** All pixels read from and written to system memory have a designated storage format and type. Format generally specifies the number and purpose of the components used to represent a single pixel, such as `GL_RGBA`. The type specifies the system memory bit width and data type used to store the pixel data, such as `GL_FLOAT`. Packed storage types are also supported, where multiple components are packed into a single system data type, such as `GL_UNSIGNED_SHORT_5_6_5`, which packs three components of 5, 6 and 5-bit widths into a single short.

- **Internal Data Format:** This refers to the resolution and number of components used to represent the pixels stored in GPU memory. For example `GL_RGB16` uses three 16-bit components. There are a large number of possible internal formats, which is further extended by both ARB and vendor specific extensions.

- **Data Alignment:** Data in the host can be stored aligned to specific byte boundaries. Depending on the system architecture, this can have an impact on the performance of data reads and writes. OpenGL supports the specification of a byte boundary alignment that will be followed when data is packed into or unpacked out of system memory.

- **Data Conversion:** If the external data formats or types mismatch with the internal formats supported by the GPU, the OpenGL driver will transparently transform the data. This transformation can take the form of precision conversion, component swizzling or component padding. Swizzling is a term used to denote the swapping of components representing a pixel, or in general the scalars within a data vector. Transparent driver conversion of data can occur in unpredictable scenarios depending on the particular internal formats supported by the GPU. This is further complicated by OpenGL provision of generic internal storage formats such as `GL_RGB`, which are dynamically mapped depending on hardware support.

3.2 Transfer Tools

Due to the typically low demand for large data transfer to and from the GPU, most development assistant tools focus on the shader processors and their optimum ALU and memory usage. At time of development there existed only two notable tools that covered the area of data transfer rates. These are Stanford's GPUBench [13] and Nvidia's PBO Texture Performance Tool [86]. These are designed to be used solely as benchmarking tools for a number of predefined configuration states. They are not designed to give the user the flexibility required to simulate the vast array of application data transfer scenarios. Stanford's GPUBench covers a wide range of scenarios relating to all areas of GPU performance, however provides a limited benchmarking flexibility in relation to data transfer. Nvidia's PBO Texture Performance Tool similarly supports the benchmarking of transfer, however the scenarios are exposed in a limited fashion. For example, in both tools the user cannot specify any OpenGL internal or external pixel formats, or investigate the potential benefits of asynchronous readback. The tools are designed to generate comparative benchmark numbers for card comparison and not to allow the exploration of configuration states applicable to an application in development.

We have implemented a download and readback pair of tools that expose all configuration states relevant to data transfer rates. The download and readback tools are covered in Section 3.3 and 3.4 respectively. Both sections cover the corresponding tool details, tool usage notes and categorised observations. We do not aim to present a survey of hardware and their transfer capabilities. We present an introduction to the transfer tool and how it can be used to optimise bus transfer rates for cryptographic acceleration and also for any application requiring fast transfer. Sample tool output across various scenarios and GPUs, and access to the source code for these tools, can be found on the tool's website [42]. The observations presented below are based on experimentation using the following cards: Nvidia GeForce 5200Go (AGP 4x), 6600GT (AGP 8x), 6600Go (PCIe x16) and 7900GT (PCIe x16). AGP (Accelerated Graphics Port [48]) and PCIe (Peripheral Component Interconnect Express [102]) are system bus standards used by the tested cards, AGP being outdated by the faster PCIe standards.

3.3 Download Tool

3.3.1 Overview

Data can be transferred from the system to the GPU by means of texture transfer or direct rendering of pixels to the active framebuffer. Each execution of the download tool, *downloadBench*, transfers approximately 6.25 GB of data in a series of *frames* from system memory to the active framebuffer. Transferring multiple frames averages out any individual frame transfer impact on the final result. Also, transferring a similar and large amount of data for each tool execution allows for easy comparison of various scenarios. The tool will not transfer the full data allocation to the card if a scenario is estimated to take longer than 3 minutes to complete, in such a case an estimated rate for the full data transfer is output. The tool generates these estimates by first running the frame transfer loop an initial 10 times before conditionally running the full scenario. These estimates are generally pessimistic as the first frame transfer time occupies a larger percentage of the run time.

At the start of each scenario execution an initial test frame is downloaded and subsequently readback from the GPU. The readback data is compared against the downloaded data, taking into account any padding bytes, see `packAlignment` below. If there is any mismatch between the readback data and the downloaded data a warning message is generated indicating that the verification stage has failed and the reported transfer rates are unreliable. This issue is fully discussed as an observation in Section 3.4.4.

3.3.2 Tool Details

The full list and description of configuration states that the tool exposes is presented below. These provide sufficient control to simulate the majority of data download scenarios. The controllable pipeline states are depicted by the following arguments to the download tool.

```
./downloadBench PixelDimension Un/packAlignment TransferMethod
                glExtFormat glExtType glIntFormat glTexTarget
                FBOState PBOState FPState
```

- `PixelDimension:` allows the user to specify the dimension of the buffer sizes and texture sizes involved in the transfer. Currently to simplify the number of parameters this is restricted to square 2D textures, though this could easily be extended.

- `Un/packAlignment:` allows the specification of the alignment in system memory that the data will be unpacked from, this is described in full in The OpenGL Programming Guide [97], see `glPixelStore()`. In general, architectures experience greater system memory read performance when the reads are aligned to some byte boundary, this setting should correspond to the user's system's recommendations.

- `TransferMethod:` indicates the choice of method for downloading data to the graphics card. The tool supports the values `DrawPixels` and `TexSubImage`, which use the `glDrawPixels()` and `glTexSubImage()` functions respectively to transfer data. `glTexSubImage()` is recommended over `glTexImage()` for repeated transfers.

- `glExtFormat, glExtType:` specifies both the format and type of data that is held in system memory. The possible values accepted are string representations of the format and type constants supported by the OpenGL API and extensions, e.g. `GL_RGBA`, `GL_FLOAT`.

- `glIntFormat:` specifies the requested internal storage format and resolution for use in GPU memory. Again the values supported are string representations of internal format constants supported by the core API and extensions. A core API example is `GL_RGBA`, a vendor specific extension example is `GL_FLOAT_RGBA32_NV`.

- `glTexTarget:` specifies the type of texture target to use to bind the texture to, the supported and tested targets are `GL_TEXTURE_RECTANGLE_ARB` and `GL_TEXTURE_2D`.

- `FBOState:` can be set to `FBO_ON` or `FBO_OFF`, indicating whether or not the data is to be transferred into a texture attached framebuffer object. This setting makes sense in the context of DrawPixels transfer method, allowing transfer of data to a texture by using render to texture. When an FBO is not used in the context of DrawPixels, the default visible framebuffer is used as the destination of transferred data. In this case the internal format and texture target can be set to `NONE`.

- `FPState:` specifies whether a fragment program is bound to the graphics pipeline or the default fixed function fragment stage of the pipeline is used. Values supported are `FP_ON` and `FP_OFF`. The bound fragment program is a tool provided shader that simply relays the fragments to the next stage.

The TexSubImage transfer method does not involve the fragment processing stage of the pipeline, and as such FPState is relevant to the DrawPixels mode. During TexSubImage mode, the only use of the fragment processing stage is the rendering of a single point to ensure the texture transfer is not optimised away by the driver.

- `PBOState`: this specifies whether a pixelbuffer object is used to unpack data from system memory during the transfers. The values supported are `PBO_ON` and `PBO_OFF`.

3.3.3 Usage Notes

Artificially high transfer rates for simulated application scenarios is the most common problem encountered when using the download and readback tools. This can occur due to transparent driver optimisations and restrictions, which are difficult to detect, and must be taken into account. When rendering to the default visible framebuffer there are many scenarios that cause false rates to be reported. If the screen display size does not encompass the entire visible rendering window then the data that corresponds to the cropped region of the window need not be transferred. If drawing to a position outside of the area specified when creating the render window, the data transfer does not need to take place. Also, when a foreign window obscures the tool's render window, the obscured region's corresponding data do not need transferring. These issues affect the `DrawPixels` mode by artificially increasing the download rates. `TexSubImage` does not result in a speed up as only a point is drawn to screen in this mode. The texture is still transferred but the omission of a single fragment rendering makes little performance difference. The verification stage of both the `DrawPixels` and `TexSubImage` modes fails when data is not transferred and consequently is not correctly read back. The user of the tool must take care not to obscure the visible window with any GUI objects for any portion of time and to ensure the visible area of the screen can encompass the render window. The use of a framebuffer object removes these obscuring issues as rendering does not target a visible window, thus these usage notes only apply to applications optimising downloads for on-screen rendering.

3.3.4 Observations

The download tool was used on the stated cards, see Section 3.2, to execute an array of scenarios. The following observations, grouped into common related factors, were made. These illustrate the type of insight that can be gained from using the tool.

Pixel Buffer Object: In general using Pixel Buffer Objects while in `TexSubImage` mode, and transferring float data types to an internal format from the OpenGL float buffer extensions (ARB or Nvidia); or transferring byte data types to internal unsigned bytes, results in the best performance. When using PBOs, `TexSubImage` mode for data download is superior, ∼3 times faster, compared with `DrawPixels` mode. Under no scenario did `DrawPixels` experience a speed up when switching between `PBO_OFF` and `PBO_ON`. Part of the difference in speed is attributed to `DrawPixels` resulting in fragment generation and subsequent processing, whereas `TexSubImage` does not produce fragments, but transfers data directly to texture memory. However, when testing `TexSubImage` using texture mapped quad rendering, there still existed a 2 times speed up. Thus, the main difference in speed is explained by the lack of affect PBOs have on the `glDrawPixels()` function.

Pixel Format and Type: There are hundreds of combinations of internal and external texture formats and types, all of which have performance implications. The following covers some points of note regarding the mix of formats and types.

- Important for cryptography, unsigned byte transfers can be accelerated using PBOs, however concerning 4 component transfers, the `GL_BGRA` external format must be used with an `*RGBA*` internal format. This is due to the internal format for bytes being stored in pre-swizzled format according to the Microsoft GDI pixel layout. If this is not adhered to, PBOs have no affect on byte transfer.
- The use of integer or short external data types shows poor performance and is not improved with the use of PBOs.
- Float data types can be accelerated by PBOs, as long as the component ordering between internal and external storage formats is maintained. It is also necessary to use resolution specific internal formats from the OpenGL float buffer extensions. If generic core API internal formats are specified then PBOs have no effect.
- There is up to a ∼7% performance advantage when using internal formats from the `NV_float_buffer` extension over the `ARB_texture_buffer` extension.

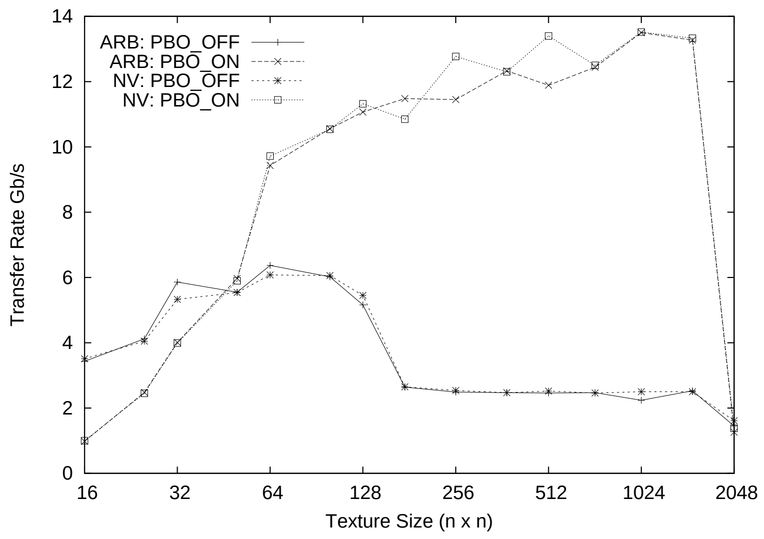


Figure 3.1: Download rates for varying texture sizes, with 4 component FP32 texels, across different scenarios using a 6600GT.

- If padding is avoided for a single byte component by using `GL_LUMINANCE8` as the internal format, PBO speed up can also occur.
- In general it is advisable to use explicit internal format resolutions to avoid possible slow downs caused by transparent data conversion.

Fragment Program: The use of a fragment program (`FP_ON`) has no significant impact on `TexSubImage` mode as expected, however when using `DrawPixels` mode there is a dramatic slow down compared to the fixed function pipeline. The fragment program used is a simple colour in colour out pass through program. No explanation could be found for this behaviour. It is assumed some configuration setting is causing a part of the rendering process to transparently resort to software mode and carry out its function on the CPU.

Data Conversion: In general, scenarios that result in data padding, swizzling, clamping, scaling of precision, lead to no speed up when using PBOs. Also, as expected, with no PBO use, data conversion causes a slow down compared to scenarios that don't undergo any conversion.

Texture Size: Varying texture sizes that are transferred can have a significant effect on overall transfer rates. To demonstrate this, Figure 3.1 shows the results of running four scenarios with varying texture sizes. The scenarios presented are configured with and without the use of PBOs. Also, two seemingly equivalent internal storage formats were used, one from Nvidia's OpenGL extensions and one from the ARB's extensions, `FLOAT_RGBA32_NV` and `RGBA32F_ARB`. It can be seen that for small texture sizes, non PBO use results in higher rates of transfer. Also when using `FLOAT_RGBA32_NV` there is a significant increase in performance at powers of two texture sizes, where as `RGBA32F_ARB` shows no correlation. We can see that in general the larger the texture the faster the download rate. However, we can see that the use of PBOs is not supported over a certain texture size. For example, at 2048 $\times$ 2048 texture size, using 4 component FP32 texels, the figure shows that PBO accelerated transfers offers no improvement in speed over texture transfers without PBOs. In general, the increase in texture size permits the amortisation of the per byte transfer costs. This is a reoccurring factor in the use of GPUs as we will see in the presented cryptographic implementations.

3.4 Readback Tool

3.4.1 Overview

As mentioned, there is a lack of demand for high readback rates in the context of traditional GPU usage. As such the rates tend to be considerably lower than download rates due to hardware and driver optimisations. PCIe and PCIe 2.0 have aimed to alleviated this asymmetry of speeds, however many PCIe graphics cards still suffer from slow readback. To ease the impact of slow readback on system performance, pixel buffer objects support the asynchronous execution of readback functions. Asynchronous, or non-blocking, readback can benefit systems that use the GPU as a co-processor, sharing work between the CPU and GPU. The presented readback tool, *readbackBench*, provides two modes of operation, one for investigating readback transfer rates and the second for analysing the asynchronous readback behaviour. Similar to the download tool, both modes support the execution of various transfer scenarios. In both modes the same amount of data, as in the download tool, is readback from the GPU frame by frame. Also, the same verification stage and initial time limiter trial phase is executed. The specific modes of operation, transfer rate and asynchronous behaviour are discussed in the following sections.

3.4.2 Transfer Rate Mode

The transfer rate mode outputs the rate at which data can be copied from the GPU to system memory. It attempts to minimize the amount of data downloaded to the GPU, using a single non texture mapped quad draw operation per frame to ensure driver optimisation does not omit the readback of any data. The tool uses its calling arguments to calculate the number of readback iterations required to transfer a large, mostly fixed, amount of data from the GPU. The arguments are as follows:

```
./readbackBench PixelDimension Un/packAlignment glExtFormat
                glExtType glIntFormat glTexTarget FBOState
                numberPBOs PBOUsagePattern [workLoad] [drawMode]
                [blockingDetail]
```

- `PixelDimension, Un/packAlignment, glExtFormat, glExtType, glIntFormat, glTexTarget:` these arguments have the same meaning as for the download tool.
- `FBOState:` when set to `FBO_ON` or `FBO_OFF` the `glReadPixels()` function is used to retrieve data from a texture attached FBO or the default framebuffer respectively. The `FBOState` argument can also be set to `FBO_ON_GTI`, which allows the user to specify that the `glGetTexImage()` function should be used to read from the texture attached FBO.
- `numberPBOs:` this turns on and off readback into system memory using a PBO. Increasing the number of PBOs increases the number of parallel readback requests outstanding to the GPU.
- `PBOUsagePattern:` this is used to specify the expected usage pattern of the pixel buffer object. OpenGL allows hints to be passed to indicate the level of modification expected and whether the most common operation will be read, write or copy. For a full list of values see the `glBufferData()` function in the OpenGL Programming Guide.
- The last three parameters are optional. If they are used, the tool changes into asynchronous behaviour mode. These are covered in the asynchronous behaviour section below.

3.4.3 Asynchronous Behaviour Mode

The use of PBOs for system storage permits function calls to readback data from the GPU without blocking. It lends support for multiple outstanding readback requests and also the overlap of GPU and CPU work. To enable asynchronous support, a PBO is created and bound for use as a packing store. A readback function call is made and will return immediately. When the readback data is required, a mapping call is made to the PBO, which returns a pointer to the data

Map Blocking Time	164279 μs
Read Blocking Time	52159259 μs
Map+Read Blocking Time	52323538 μs
Dummy Work Time	44455106 μs
Total Transfer Time	97553896 μs
Bytes Transferred	6710886400
Transfer Rate	524.80 Mb/s

Table 3.1: Example output of the readback tool in asynchronous behaviour mode using a 6600GT.

store. The map call will block until all data is readback to (packed into) system memory. To use the PBO for readback again, it must be unmapped. As indicated above multiple PBOs are supported by OpenGL and can be used to pipeline readbacks while the CPU works on previously readback results. An example of pipelining readback is presented in Nvidia's Fast Texture Transfers article [89]. Ideally no CPU cycles should be wasted on blocking calls, however this is far from achievable in most scenarios. The following lists the `readbackBench` arguments relevant to exposing the level of asynchronous support.

- `workLoad:` To expose a scenario's pipeline potential the tool can insert variable amounts of dummy work to simulate CPU side processing during data readback. `workLoad` can be set to an integer that is used to increase and decrease the amount of dummy CPU cycles spent. Table 3.1 shows an example output of the tool when running in asynchronous mode. It includes the total amount of PBO map and readback command blocking times, and also the time spent in the dummy work loop. To maximise the pipeline potential, a scenario ideally would reduce the total amount of blocking time by the same amount of dummy work time increase, thus maintaining the same transfer rate while gaining free CPU cycles. This can only happen while the amount of dummy work time is less than the total scenario time.

- `drawMode:` The default draw mode used during the execution of the readback tool is a non texture mapped quad render operation as mentioned previously, called `QUAD_DRAW` mode. This is suitable for measuring pure readback rates, however it is not always suitable for measuring the pipeline potential of a scenario. The asynchronous efficiency depends largely on how the particular hardware system handles DMA memory transfers during bus contention and how the graphics driver behaves. `drawMode` can be set to `PIXEL_DRAW` to instruct the tool to use `glDrawPixels()` function to fill the entire window each frame. This generates download traffic to contend with DMA readbacks, thus simulating an application that requires bi-directional data transfer. The default `QUAD_DRAW` mode can be used to more closely simulate the bus traffic patterns of a ping-pong based application, where the results generated on the GPU are used for the next kernel execution's input, i.e. little download traffic.

- `blockingDetail:` this argument, when used, generates logging messages for each frame displaying the blocking times in detail.

3.4.4 Usage Notes

The following usage pitfall is applicable to both download and readback transfers. The internal format used to store data within the graphics card is implementation dependent. According to the OpenGL specification an OpenGL implementation can use the input parameters such as external format, external type and internal format merely as a guide. The graphics driver makes this process transparent and thus when specifying these parameters the user cannot be sure exactly what form the data will take when transferred from and to the card. The result being that data can be down converted into a lower precision representation before being transferred over the bus, leading to false speed-ups.

An example of this pitfall is Nvidia's transparent mapping of the generic internal format `GL_RGBA` to the specific internal format `GL_RGBA8`. As a consequence floats, integers, shorts all get down converted to an 8-bit representation when used in combination with `GL_RGBA`. There is no way of programmatically knowing that a smaller resolution is being used apart from using a strict verification stage that requires an exact match between data downloaded and readback from the GPU. When a verification failure is detected the tools continue, however a warning message is output to notify the user of possible false speed-ups. The reason for not aborting the scenario execution on verification failure is that in some scenarios data conversion will occur but the number of bits per pixel will not change. For example, components can be scaled to a range $[0, 1]$, but can have the same bit width as the original data elements. Also, 32-bit integers can be transferred as 32-bit floats, the data will be down converted to a float, yet the number of bits transferred will be consistent. To allow for better judgement in determining whether the warning message indicates a false speed up, the internal formats natively supported on the GPU should be known. For Nvidia DX9 GPUs, this information can be found in the GeForce 7 Programmers Guide [87].

Window obscuring affects readback in the same manner as it affects download transfers. Care must be taken to ensure when using the default framebuffer that the entire window is visible, otherwise the verification stage will fail and artificial increases in transfer rates can occur. Also, it was noted that for asynchronous behaviour to function the specific graphics drivers must be used. With regard to DX9 Nvidia drivers running on Linux, the driver version should be at least 1.0-8762.

3.4.5 Observations

Similar to the download tool, we present the notable observations based on running a large array of pipeline configuration combinations. We split the observations for the readback tool into transfer rate mode and asynchronous behaviour mode.

3.4.5.1 Transfer Rate Observations

Scenarios that use short or integer external data types under perform by a minimum of 50% compared to byte and float counterparts. Single external components suffer a four times reduction if the internal format is not explicitly requested as a single component due to all four components being transferred across the bus and filtering occurring within the driver. When using `glGetTexImage()` to readback, there is a large drop in performance compared to using `glReadPixels()`. The use of `TEXTURE_RECTANGLE_ARB` as the texture target consistently out performs `TEXTURE_2D` by $\sim$7%. This performance difference does not exist when using internal formats from the OpenGL float buffer extensions, such as `FLOAT_RGBA32_NV` or `RGBA32F_ARB`. Regarding the size of buffer dimensions, there is a notable performance increase using higher numbers of PBOs for small buffer sizes. Also, there are large rates increases when using sizes that are powers of two, though in general the larger the buffer size the faster the readback rate. Finally, as with downloads, it is advisable to use explicit internal format resolutions to avoid the possibility of the driver performing transparent conversions, which can adversely affect transfer rates.

3.4.5.2 Asynchronous Behaviour Observations

Asynchronous Support: On the tested hardware the vast majority of scenarios do *not* support asynchronous readback. The notable groups that do not are: all scenarios with an external type of unsigned byte that do not use the `GL_BGRA` external format; all scenarios with an external data type of signed byte;

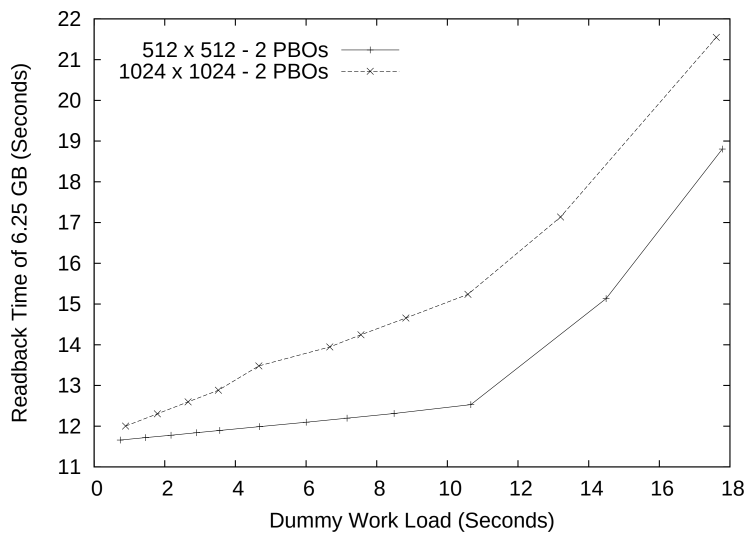


Figure 3.2: PCIe asynchronous behaviour comparison of 512×512 versus 1024×1024 buffers.

all scenarios with external types of signed and unsigned shorts and integers; all scenarios with an external format of `GL_LUMINANCE`; all scenarios that use floats without an explicit internal resolution formats, e.g. those from the float_buffer or arb_texture_float extensions; all scenarios that use `glGetTexImage()` to readback data.

Pipeline Efficiency: The ideal behaviour of scenarios that support asynchronous readbacks would be that for every second added to the CPU dummy work time, blocking time is reduced by a second. As such, for a given amount of readback time, we should be able to hide that amount of additional dummy work time. However, in practice this is not true and varying settings such as the number of PBOs and buffer sizes used affects how close a scenario comes to this ideal. On the AGP cards the number of PBOs used, buffer sizes and drawing mode used have a significant effect on the pipelining efficiency. For example, using a 512×512 buffer size in `QUAD_DRAW` mode, as the number of PBOs used increases the more efficient at hiding the dummy work time the scenario becomes. However, when the same scenario uses `PIXEL_DRAW` mode, the efficiency of dummy work time hiding does not increase as the number of PBOs used increases. Using a PCIe card eliminates the vast majority of these behavioural differences regarding the number of PBOs and draw modes used. However, there is still a significant difference between the asynchronous behaviours when buffer dimensions are varied. Figure 3.2 illustrates 512×512 textures outperforming 1024×1024 textures in terms of pipeline potential on a PCIe card.

3.5 DX10 Data Transfer

The complexities of achieving high data transfer have largely been removed with regard to DX10 compatible GPUs. This is mainly due to the GPU natively supporting a wider range of data types and widths removing most conversion issues encountered previously. With internal data formats being scalar, and more external data types matching internal data types, format mismatch is less likely. Also, the use of CUDA removes the need to deal with graphics API issues such as of pixel formats, transfer methods, frame buffer objects, etc. Nvidia have released a simple open source demonstration application within their CUDA SDK, which shows peak transfer rates both to and from the GPU. Through experimentation it is easy to achieve maximum bandwidth for use with cryptographic functions. We can use 32-bit integer external and internal data types with no loss of transfer performance. One remaining performance factor, which must still be taken into account, is the use of DMA memory. This is provided via pixel buffer objects in OpenGL. CUDA supports DMA memory via allocation through the `cudaMallocHost()` function. All copies and reads from this memory are accelerated. The drawback of using too much of this type of memory is a possible decrease in system wide performance.

3.6 Conclusions

One of the primary bottlenecks to overcome in effectively accelerating applications that require large data streams using the GPU is the efficient movement of said data onto and off of the graphics card. The use of the tools presented in this chapter provide a means of exposing a great deal of idiosyncrasies related to data transfer via the OpenGL API. These can be used to identify optimal state configuration for data movement and thus help avoid such bottlenecks. We cover the application of these tools in relation to AES development on DX9 hardware in Section 4.1.

Chapter 4

Symmetric Cryptography on DX9 Hardware

Early generations of graphics processors were controlled through parametrised function calls. This is ill suited for the level of hardware control required to implement symmetric-key functions as we saw with Cook et al. in Section 2.5.1. A key innovation since has been an increase in the graphics processor's programmability. DX9 GPUs support the ability to create and run custom programs. However, these processors provide floating point processing capabilities only and as such cryptography is not an obvious target application. Despite this, this chapter shows that it is possible to achieve respectable secret-key cryptographic performance using the DX9 generation of GPU.

We have selected the Advanced Encryption Standard (AES) symmetric block cipher as our example cryptographic algorithm for implementation. AES was selected due to its popularity, compact nature, well documented implementation techniques and optimisations. We have simplified our investigation to cover AES using 128-bit key size only, which provides sufficient detail to demonstrate the feasibility and performance of the proposed implementation approaches. The implementation executes multiple instances of the block cipher in isolation, effectively in ECB mode. This mode has the features of being simple and embarrassingly parallel, though it is insecure. The presented implementations can be extended to other parallel modes of operation such as CTR, which is considered secure [64] and can also result in performance improvements as we will see in Chapter 5. The ability to parallelise an application is necessary for achieving performance on a GPU architecture.

Sections 4.1 covers GPU related issues that have a direct bearing on the presented implementations. Section 4.2 demonstrates the core operation of AES, a bitwise exclusive-or (XOR), and various implementation approaches along with

	Download Rate (Gb/s)	Readback Rate (Gb/s)
6600GT AGP 8x	12.71	1.39
7900GT PCIe x16	21.24	8.42

Table 4.1: Peak data transfer rates using 1024×1024, 4 component, byte buffers.

their performance. We present the details of three different AES implementation approaches in Section 4.3 including results and analysis. Section 4.4 investigates the effectiveness of using a DX9 GPU as a parallel co-processor and its interference with overlapping CPU processes.

4.1 The GPU and AES

The following is a discussion of various GPU related factors that have a direct impact on the design of our AES implementation approaches.

Data Throughput: There is a high data throughput requirement, both to and from the graphics card. This data transfer must occur across the relatively slow system bus, which has improved with the introduction of the PCIe bus standard, however still remains one of the major potential bottlenecks. We used the tools presented in the previous chapter to investigate the pipeline configuration states for our needs. The selected states give relatively good performance for both download and readback transfers and also match the requirements of the specific implementations. We highlight the peak performance of data transfers for the AES implementation cards in Table 4.1. Note that test scenarios were executed on all cards referenced in Chapter 3 to ensure good transfer performance for all cards. Asynchronous readback is not used in the reported benchmarking of the AES implementations, however the selected configuration states were tested to ensure good non-blocking behaviour.

The main data transfer requirement for AES on the GPU is the use of a data format that is compatible with our input, output and processing requirements, i.e. those that maintain precision and do not require conversion. As such, we focus on integer based formats, bytes, shorts and integers. From the previous chapter we have noted that shorts and integers give poor transfer performance. Also, integers lose precision during driver conversion. As an example of the extreme transfer rate variation experienced across differing pipeline configuration, downloading shorts to a 6600Go PCIe achieved 0.84 Gb/s, where under an equivalent scenario using unsigned byte transfer achieved 18.59 Gb/s. To achieve this byte transfer rate one must use an internal format set to `GL_RGBA8` and an ex-

	Download	Readback
`PixelDimension`	Peak at 1024×1024	
`Un/packAlignment`	4	
`TransferMethod`	`glTexSubImage()`	`glReadPixels()`
`glExtFormat`	`GL_BGRA`	
`glExtType`	`GL_UNSIGNED_BYTE`	
`glIntFormat`	`GL_RGBA8`	
`glTexTarget`	`GL_TEXTURE_RECTANGLE_ARB`	
`FBOState`	`FBO_ON`	
`PBOState`	`PBO_ON`	-
`numberPBOs`	-	4
`FPState`	`FP_ON`	
`PBOUsagePattern`	`GL_DYNAMIC_DRAW`	`GL_DYNAMIC_READ`

Table 4.2: *downloadBench* and *readbackBench* parameters used in pipeline configuration of AES implementations.

ternal format of `GL_BGRA`. As another example of the transfer rate variation, if `GL_RGBA` were used for both external and internal formats, the rate of download is 4.51 Gb/s. `GL_DYNAMIC_READ` and `GL_DYNAMIC_DRAW` were used to represent application's data access pattern's in a practical environment: data stores are read from and written to frequently. We used 4 pixel buffer objects to suit the number of output textures used in all implementations. Also, it was necessary to ensure that all states worked well with the use of framebuffer objects as their use is non-optional in one of the AES approaches. The full list of states used for the AES implementations is shown in Table 4.2.

Texture Lookups: Our implementation approaches rely heavily on texture lookups. These lookups come largely in the form of sequential and dependent lookups. Dependent texture lookups are those that use retrieved data from an initial texture lookup to form the basis of new texture co-ordinates to execute a further lookup. This type of lookup generally results in random gather patterns from the accessed texture and results in large slowdowns to read performance. Example results from the GPUBench [13] tool shows the dramatic fall off in access speed depending on the different types of texture access, ranging from over 480 Gb/s for sequential access to less than 32 Gb/s for random access on DX9 cards. The reason for such a reduction in speed is due to the small cache sizes on the GPUs. These caches are normally sufficient for graphical purposes, which show a high degree of spacial locality of reference. There is an emphasis on all implementation techniques presented to try to reduce the memory footprint of lookup tables used and to increase the reuse patterns of memory access. Thus, we try to minimise the last two types of cache misses as discussed by Hill and Smith [47], namely conflict and capacity misses.

Scatter: Gather is supported in terms of texture reads from various locations, however, a previously noted DX9 restriction is the lack of native scatter support within the fragment processors. Each fragment processor can output a small number of results (between 1 and 4 pixels), however, these results must be written to a predetermined memory location within the active output framebuffers. This is due to traditional graphics programming where each potential pixel is associated with only one pixel location on the screen/framebuffer. This, as we will see in Section 4.3.3, restricts our block cipher output format for our AES implementation strategies and also causes a further restriction on the cipher input format for one of them.

XOR: Another relevant area of the GPU is the availability of a bitwise operations within the final stage of the pipeline. There is hardware support for this type of operation, and in particular XOR, within the raster operation units of Nvidia DX9 GPUs. This allows the combination of the fragment processor output and the existing data within the active framebuffer to be combined using XOR. Note, as mentioned previously there does not exist support for XOR within DX9 fragment processors. ROPs can only be used at the end of the rendering pipeline and exist in fewer numbers compared to the fragment processors. We use the ROP XOR functionality in both Section 4.2.3 and Section 4.3.3 and further discuss the restrictions imposed by its availability in the ROP only.

Swizzle: The last GPU feature of note is the fragment processors ability to implement swizzle operations for free. Data can be stored, addressed and operated upon within the various processing stages of the GPU as groups of scalar components, or vectors up to 4 wide. This vector support is due to traditional graphics processing commonly requiring to work with RGB or RGBA (red, green, blue, alpha) component groups. The fragment processor has the ability to arbitrarily access and permute the RGBA components within a vector during instruction execution. This provides a useful means for cheaply executing byte rotates, which we use to further optimise the AES implementations to reduce the active memory footprint. This is explored in Sections 4.3.1 and 4.3.3.

4.2 XOR Approaches

The AES implementations presented rely on efficient execution of XOR. Here we present three different approaches used for performing this operation on the DX9 GPUs.

4.2.1 8-bit XOR

This approach involves the use of a lookup table to perform the XOR operation on two 8-bit values. The table uses 65,536 (256×256) byte entries, representing the precomputed results of the XOR operation for all 8-bit values. The lookup table is stored as a texture, which uses the single component GL_ALPHA external and internal format. This format is used to reduce the internal memory necessary to represent the lookup table.

To test this approach, two four-component 1024×1024 textures are used to store the input data. Each component is a byte, thus each texel holds four bytes. The bytes at corresponding locations within these two textures are XORed together, thus using a sequential data access pattern across the input textures. For example the first component of the texel at location (x, y) in texture 1 is XORed with the first component of the texel at the same (x, y) location in texture 2. A quadrilateral with dimensions 1024×1024 to match the size of the input textures is rendered to generate a fragment program instance per texel pair. The fragment program loads a single pair of texels from the two input textures corresponding to the fragment's texture co-ordinate. Each of the four corresponding pairs of bytes from the pair of texels are used in turn to execute a dependent texture lookup within the 256×256 XOR lookup texture. The result of each XOR lookup form one of the four components of the output fragment.

4.2.2 4-bit XOR

We have noted that dependent texture lookups have a severe performance penalty. One way to reduce this penalty is to make the dependent texture lookups access a reduced lookup space and thus ease the caching requirements. This approach uses a similar method to the above 256×256 8-bit lookup table, however to help reduce the size of the table we split each 8-bit input value into two 4-bit values and use a smaller precomputed 256 entry table. The issue with this approach is that there is no integer support or bitwise operators, all values read into the fragment processor are represented as floating point numbers. Thus splitting the input floating point values representing the high and low 4-bit values must be achieved using a different method than bit masking or shifting.

The method requires a 16×16 entry texture with the wrap mode set to `GL_REPEAT` to store our precomputed XOR values. We use the `text2D()` function to execute the XOR texture lookups instead of the more flexible `textRECT()` function. The `text2D()` function scales the width and height of a square texture into the range $[0, 1]$. It also scales the co-ordinates used as lookup values in the range $[0, 1]$. In comparison the `textRECT()` function supports non-square textures and does not perform any scaling on width, height or co-ordinates, and as such we use this texture lookup function more frequently for general purpose use. Regarding a texture's wrap mode, this dictates what happens when a co-ordinate is used for lookup that is outside the range of the texture's width or height. In the case of `GL_REPEAT`, a texture co-ordinate that falls outside its border, wraps around to the opposite border removing the width or height from the co-ordinate. This repeats until the co-ordinate falls within the texture. For a square texture, we observe that the repeat wrapping behaviour acts as a mod n operation on the (x, y) lookup co-ordinates, where n is the height/width of the texture.

The `text2D()` scaling behaviour provides an automatic manner of isolating the 4 high bits of an 8 bit input value. The XOR texture has a dimension size of 16 pixels in both directions. The use of `text2D()` to access this texture effectively scales each dimension from $[0, 15]$ to $[0, 1]$. Also, consider that the XOR input operands have the range, $[0, 255]$, these are also scaled to $[0, 1]$. As the original input operands with a 256 distinct values are mapped to the same range as the texture dimensions with 16 distinct values, the 4 high bits of the input operands dictate the texel fetched from the XOR texture. Thus, the use of the original operands in the 4-bit precomputed XOR texture returns the XOR of the operands' high 4 bits. If we multiply the input operands by 16, we effectively shift the 4 low bits into the place of the 4 high bits. However, the operands are stored in registers as floats, since the fragment processor is a floating point only processor. Thus, unlike left bit shifts on byte data types, the high bits are still affecting the value of the operand. The `GL_REPEAT` removes this by effectively executing a mod 16 on the operands. The operands multiplied by 16 can be used to lookup the XOR texture and return the XOR of the operands' low 4 bits. After retrieving the two high and low 4-bit results, they are recombined by multiplying the 4 high bit value by 16 and adding the low bit value.

4.2.3 ROP XOR

We use the native XOR instruction found in the ROP units at the end of the rendering pipeline. This allows the output from the fragment processors to be XORed with the values within the active framebuffer. The advantage of this is that the XOR instruction itself will have good performance. The disadvantage is that the XOR operation can only be applied to the final stage of the rendering process, meaning that to reuse previously XORed values a full render pass must occur. In comparison, the previous two approaches that simulate XORs within the fragment processors can immediately reuse the values as input to subsequent operations within the fragment program. The ping-pong method must be employed when requiring the previous render pass output to be used within the next pass input. This involves making the output textures the input textures of the next rendering pass, and switching the current pass's input texture to be the output textures of the next pass.

All input and output textures use unsigned bytes for their format. This data type suits the small size of the lookup tables, the ability to divide the value into high and low 4-bit pairs and also the use of the bitwise component within the ROP. A by-product of using the XOR function in the ROP stage is that only a single input can be XORed to the existing results in the framebuffer. We make sure that all tests are timed transferring the same amount of data and performing the same number of XOR operations. Also all tests use input and output texture sizes of 1024×1024.

4.2.4 Results

Table 4.3 shows the results of the three approaches using a GPU via OpenGL. The results are quoted in Gb/s of XOR output production so as to be consistent with the AES rates quoted later. A single test consists of two 4 MB input buffers being consumed on the GPU in a sequential manner, XORing corresponding data elements from each buffer one byte at a time. The rates shown are generated from the average XOR throughput for 256 serial executions of said test. The results from the GeForce 6600GT were generated on *System 1*, and the results for the GeForce 7900GT were generated on *System 2*. System 1 and System 2 refer to the hardware and software configurations used to execute the experiments and are detailed in Appendix D. Also included in Table 4.3, is the performance of executing XOR on the System 2 CPU. The 8-bit and 32-bit CPU results show the performance of using a standard C implementation using bytes and integers respectively as the data units. No optimisations were applied to the CPU XOR

	GeForce 6600GT			GeForce 7900GT			CPU	
	8-bit	4-bit	Native	8-bit	4-bit	Native	8-bit	32-bit
W/O Round Trip	1.41	8.34	32.5	5.25	27.42	95.69	0.92	3.41
With Round Trip	0.62	0.98	1.10	2.61	3.69	3.71		

Table 4.3: Results of the various XOR implementation approaches quoted in Gb/s of XOR output.

implementation, as such the CPU results should not be considered optimal. The GPU results include figures for running the approaches with and without full data round trip. As one would expect the full data round trip approaches incur large slow downs due to the transmission of input and results across the system bus. In the context of an AES implementation, the output for a single XOR operation will generally not have to undergo transfer across the system bus.

We can see that the native XOR results far exceed those of the other two approaches. However, it is worth bearing in mind the previously mentioned restrictions to using the ROP XOR approach. The native speeds as expected are close to the full rendering speeds less the additional overhead of a texture lookup and a framebuffer read per pixel per pass. The theoretical pixel fill rate of the 7900GT is 7200 Mpixels/s, which in the case of 4 byte components, is equivalent to 214 Gb/s. We can also see that there is a significant increase in XOR performance when using the 4-bit lookup table over the 8-bit lookup table. This increase and the fact that the major difference between the table lookup approaches and the native approach is the execution of dependent texture reads, suggest that the lookup table approaches are memory bound.

4.3 AES on DX9 Hardware

4.3.1 AES Lookup Tables

The AES cipher is introduced in Section 2.3.1. As noted earlier, the number of rounds is determined by the key length, 128-bit uses 10 rounds, 192-bit uses 12 and 256-bit uses 14. We have opted to use a 128-bit key length and thus 10 rounds in all AES implementations within this chapter. These rounds can be reduced into a number of simplified equations, one for each column of the state, see Equation 2.1. This equation reduces the number of operations involved by using 4 1 KB lookup tables, the results of which need to be XORed with each other and the round key. In an attempt to reduce the active memory footprint used within each round we also have adopted a variation of Equation 2.1, shown in Equation 4.1 as stated in the original Rijndael proposal [21]. This equation is based on the observation that the tables, T_i, can be generated from any one of the tables with byte rotations. A byte rotation is denoted by `Rot()`. This can be seen in Equation 4.1, in that a single table is used, the results of which are rotated. The use of a single 1 KB table reduces the caching demands for this part of the implementation. This equation incurs a penalty of three extra rotates per column per round. The rotates can be implemented using the free swizzle operations. The presented AES implementations are based on the use of both equations, which are described as noROT and ROT (no-rotate and rotate). Thus we can see that the operations involved in the AES implementations are byte selects (swizzle), XORs, and table lookups (denoted by $T_i[\,]$).

$$\begin{aligned} e_j &= T_0[a_{0,j}] \oplus \mathrm{Rot}(T_0[a_{1,|j-1|_4}] \oplus \\ &\quad \mathrm{Rot}(T_0[a_{2,|j-2|_4}] \oplus \mathrm{Rot}(T_0[a_{3,|j-3|_4}]))) \oplus k_j \ . \end{aligned} \tag{4.1}$$

4.3.2 AES Input Data

In general we read message data (plaintext/ciphertext) from textures that have an internal format of four bytes (components) per texture element (texel). Each texel makes up a single column within the cipher input state. Each texel is written to the destination framebuffer when the cipher or stage processing, depending on the approach used, is finished and represents the new state of the corresponding column. Within all AES implementations we attempt to increase the patterns of memory access by altering the layout of the message data across the input textures. We explore three different input gather techniques, which include the use of multiple tables and single tables. These techniques are included in the AES implementations where appropriate.

In Figure 4.1(a) we can see that the input data is read from four different textures at the same texture co-ordinate, which provides for good predictability. However, as Govindaraju et al. [35] point out, texture memory is read in the form of blocks of data. This would mean that 4 independent texture blocks are requesting residency within the texture cache at all times. We label this technique as *Multi Input* in the results section.

In Figure 4.1(b) we have adopted a different memory gather approach reading all input plaintext from a single texture. The layout of the 16 byte blocks use four component texels one after the other in a horizontal fashion, which we hope would require less active memory blocks within the texture cache at the one time. The rasterisation pattern, which is responsible for handing off fragments to the

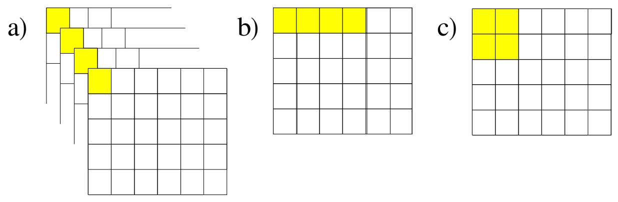


Figure 4.1: Illustrations of the different gather techniques employed for message input data across the AES approaches.

fragment processor in a normally cache friendly order, is proprietary so we can only guess at the most efficient access patterns. To reduce the overhead in calculating the 4 different gather texture co-ordinates within the fragment program we construct the rendered quadrilateral with multiple texture co-ordinates per vertex. We configure each texture co-ordinate set to be appropriately out of line with the rendered quadrilateral so that the interpolated co-ordinates generated within the rasterisation stage will automatically fall on the correct texel. This allows the rasterisation stage of the pipeline to calculate the required co-ordinates, thus incurring no computational overhead within the fragment processors. This technique is labelled as *Single Input Hgather*.

The third gather approach, shown in Figure 4.1(c), is similar to the previous technique. It reads from a single texture. However, to cater for an access pattern that suits 2 dimensional block structures, we organise the input message data into squares. This has similar gather requirements to the standard texture filter reads used within traditional graphics programming. The same method involving multiple texture co-ordinates as stated in the previous technique is also used here. This technique is labelled as *Single Input Sgather*.

4.3.3 AES Implementations

Here we present three different AES approaches on DX9 hardware. All approaches implement the AES cipher using 128-bit key length and thus 10 rounds. All approaches use pipeline configuration as specified for efficient data transfer both to and from the graphics card. The data round trip overhead is included in reported performance to show realistic results in a practical environment. As introduced in Section 4.3.1 both forms of the optimised table lookup techniques of AES cipher implementation are implemented within each approach. All approaches use a technique called multiple render target, which involves the use of 4 output textures as output targets for the fragment programs. The AES implementation approaches are presented as follows:

Approach 1: This approach is based on the 8-bit implementation of XOR as described in Section 4.2. Each execution of the fragment program reads a full 16 byte block via 4 texels using all the gather techniques described above. The other input textures used within the fragment program are the round key texture, the XOR texture and the T_i lookup textures. The round key texture is a 1D texture that contains a pre-generated schedule of round keys, which is provided by the CPU part of the implementation. The appropriate texture co-ordinates for the round key are dynamically generated within the fragment program. There are either 5 or 2 1D T_i lookup textures, representing the first form of the cipher optimisation lookup tables (noROT) or the second form, which only involves a single lookup table (ROT). The extra lookup table is used for the last round, which consists of different data than the other T_i tables as the round MixColumn() step is omitted. A single execution of the fragment program processes the 16 input bytes (a full AES state) and produces the final round output of 16 bytes.

Table 4.4 contains pseudocode that represents a single column, single round transformation using the noROT optimisation. We can see the starting lines, which use the previous columns represented by four 4-way components: s0-3. These are looked up using x,y,z,w (synonymous with rgba) to access the particular row involved in this column transformation. After the initial T_i and round keys are looked up they are repeatedly XORed with each other via the XOR texture dependent texture lookup. It should be noted that all retrieved texels used for texture lookups require multiplication by a correction factor of slightly less that 1 to avoid rounding error. This is labelled as `applyCorrectionFactor()` in the pseudocode. This correction factor did not adversely affect performance as the operations could be removed without any change in throughput. This indicates that the algorithm is I/O bound due to the large number of dependent texture lookups. The implementation mostly executes this pseudocode repeatedly for all columns of the state and all rounds of the cipher. The final four 4-way components are written to the active 4 output textures ready for readback after all fragments have been processed.

Approach 2: Based on the 4-bit version of the XOR approaches described in Section 4.2. The vast majority of implementation detail is the same as the above 8-bit AES approach. The number of XOR lookups are doubled due to both the high and low bit values being dealt with separately. The high and low bit values are only recombined at the end of each round when necessary for use as a single

```
/* AES Table lookup */
te0 = tex1D(te0Tex, s0.z);
te1 = tex1D(te1Tex, s1.y);
te2 = tex1D(te2Tex, s2.x);
te3 = tex1D(te3Tex, s3.w);

/* Round key data - i is the round counter */
rk = tex1D(rkTex, 1/44*((i*4)+1));
applyCorrectionFactor();

/* First 4 byte xors for a single column */
t0.x = tex2D(xorTex,float2(te0.x,te1.x)).w;
t0.y = tex2D(xorTex,float2(te0.y,te1.y)).w;
t0.z = tex2D(xorTex,float2(te0.z,te1.z)).w;
t0.w = tex2D(xorTex,float2(te0.w,te1.w)).w;
applyCorrectionFactor();

t0.x = tex2D(xorTex,float2(t0.x,te2.x)).w;
t0.y = tex2D(xorTex,float2(t0.y,te2.y)).w;
t0.z = tex2D(xorTex,float2(t0.z,te2.z)).w;
t0.w = tex2D(xorTex,float2(t0.w,te2.w)).w;
applyCorrectionFactor();

t0.x = tex2D(xorTex,float2(t0.x,te3.x)).w;
t0.y = tex2D(xorTex,float2(t0.y,te3.y)).w;
t0.z = tex2D(xorTex,float2(t0.z,te3.z)).w;
t0.w = tex2D(xorTex,float2(t0.w,te3.w)).w;
applyCorrectionFactor();

t0.x = tex2D(xorTex,float2(t0.x,rk.z)).w;
t0.y = tex2D(xorTex,float2(t0.y,rk.y)).w;
t0.z = tex2D(xorTex,float2(t0.z,rk.x)).w;
t0.w = tex2D(xorTex,float2(t0.w,rk.w)).w;
applyCorrectionFactor();
```

Table 4.4: Pseudocode for a single single column, single round transformation using simulated XOR in-fragment processor operation.

8-bit T_i table lookup value. This recombining could be further delayed by using 2D T_i lookup tables based on 4-bit by 4-bit lookups, though was deemed unnecessary as the ALU instructions are not presenting a bottleneck. This can be shown by the removal of all ALU instructions within the algorithm implementation resulting in no performance difference.

Approach 3: This approach is based on the ROP provided XOR native implementation shown in Section 4.2. As scatter is not supported within fragment programs, the output is restricted to writing to a fixed location within the four textures bound to the active framebuffers. Recall that the XOR operation is executed at the end of the pipeline pass, after the fragment program has terminated. To use the XOR results we must initiate another pipeline pass using the output textures as the new input. This restriction dictates that the input format must match the output format and thus use the Multi Input gather technique as described above. The Single Input Gather techniques are not incorporated into the implementation of this approach.

As only one XOR operation per fragment output can be executed per pass, we require five rendering passes per round of execution plus one initial clear fragments command to reset all output values to zero. The input and output textures are swapped for each pass, see ping-pong in Section 2.1.3. Each stage of the optimised AES equations are implemented by a different fragment program specifically written to execute the correct component based lookup within the T_i textures. To save having to read in all four input textures for each render pass, and due to the free nature of the swizzle operation we can rearrange the ROT version of the AES implementation technique to permit only a single active input texture per pass. This reduces the active cache footprint of a single pass. Equations 4.2 and 4.3 show the Rot equation suitably rotated to facilitate a single state column reference per rendering pass. Note that the column references are matching vertically. This in effect means that we are only referring to a single column at each stage and generating full stage output for all columns, XORing it with the appropriately rotated result. The OpenGL Vertex Buffer Object extension was employed when implementing this approach to reduce the overhead of vertex transfer due to the high number of render passes.

$$k_0 \oplus T_0[a_{0,\mathbf{0}}] \oplus Rot(T_0[a_{1,\mathbf{1}}]) \oplus Rot2(T_0[a_{2,\mathbf{2}}]) \oplus Rot3(T_0[a_{3,\mathbf{3}}]) \ . \tag{4.2}$$

$$k_1 \oplus Rot3(T_0[a_{3,\mathbf{0}}]) \oplus T_0[a_{0,\mathbf{1}}] \oplus Rot(T_0[a_{1,\mathbf{2}}]) \oplus Rot2(T_0[a_{2,\mathbf{3}}]) \ . \tag{4.3}$$

		8-bit	4-bit	ROP
Multi Input	ROT	49.92	91.76	**361.20**
	noROT	48.88	89.52	359.12
Single Input Sgather	ROT	49.76	91.20	-
	noROT	48.88	89.76	-
Single Input Hgather	ROT	49.60	91.28	-
	noROT	49.20	90.40	-

Table 4.5: GeForce 6600GT results for the various AES approaches quoted in Mb/s.

		8-bit	4-bit	ROP
Multi Input	ROT	206.88	313.84	**870.99**
	noROT	205.68	312.08	868.50
Single Input Sgather	ROT	208.48	313.44	-
	noROT	207.36	312.96	-
Single Input Hgather	ROT	207.92	313.28	-
	noROT	205.52	312.64	-

Table 4.6: GeForce 7900GT results for the various AES approaches quoted in Mb/s.

4.3.4 Results

Table 4.5 shows the peak throughput of AES encryption using the various implementation strategies described above running on the GeForce 6600GT (System 1, see Appendix D). Table 4.6 shows the same for implementations running on the GeForce 7900GT (System 2, see Appendix D). The results are derived from the average peak AES throughput rate taken from 256 serial executions of each test. The input data is randomised for each test, thus ensuring caches behave in a manner expected within a practical system. We can see that the performance figures follow the results trend presented in the XOR Approaches in Section 4.2. There is a consistent slight speed up when using the ROT version of the AES implementation over the noROT version. There is no appreciable difference in speed when using the different gathering techniques, which suggests that the bottleneck lies with the XOR table lookups or that the sequential nature of all gather techniques do not incur cache misses in the first place. The large speed increase in the ROP XOR approach over the other XOR approaches suggest the primary bottleneck is the texture dependent lookups to simulate the XOR operation. As mentioned previously the improvement of 4-bit XOR over 8-bit XOR shows that the cache is failing to accelerate these texture lookups. Note that the implementations presented in all results sections within the thesis

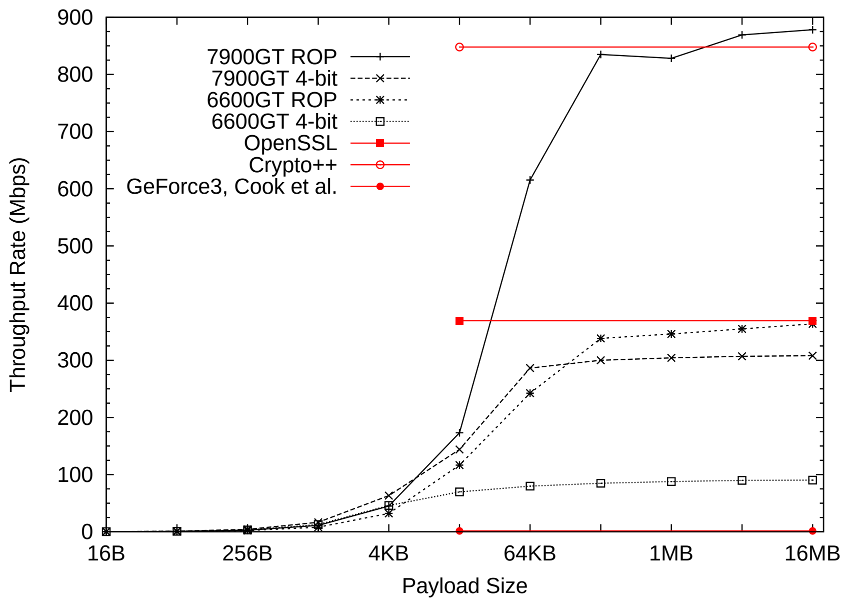


Figure 4.2: Effects of payload size variation on AES encryption throughput.

have been confirmed as functionally correct using comparisons against standard implementations of the corresponding algorithms.

Figure 4.2 demonstrates the encryption throughput effectiveness of the GPUs studied when using different payload sizes. A payload defines a block of data, which is delivered to the GPU in isolation and delivered back to the CPU after the entire data block is encrypted. In practice the payload size refers to the amount of data transferred in the form of input and output textures across the system bus for kernel processing in a single pipeline pass. The figure clearly shows that as the payload size reduces the throughput also reduces. Transferring multiple small data loads across the system bus leads to an increase in the number of required CPU-GPU interactions, increasing inefficiency. More importantly, in general terms, as data workloads reduce in size it becomes increasingly difficult to ensure all processors in a many processor environment are kept busy, which in turn leads to difficulty in effectively leveraging the potential processing power. The implication of the noted behaviour in Figure 4.2 with regard to cryptography is an ineffectiveness of the DX9 GPUs to assist in small data unit encryption and decryption. Applications such as IPsec rely heavily on this type of behaviour and thus for the GPU to assist in this context, the overhead of data transfer and API calls would have to be significantly reduced. Applications that require bulk data encryption and decryption, or latency insensitive applications in general are thus more suited to these GPUs.

Included in Figure 4.2 are the 7900GT and 6600GT ROP and 4-bit approaches, along with the CPU based AES throughput rates. It is assumed that the payload size does not affect the throughput rate of AES on a CPU in any significant way. The performance rates of AES decryption are not listed separately for either GPU or CPU as they produce the same throughput rates. The essential difference between the AES encryption and decryption algorithms when using the optimised equations presented, is a change in the lookup table data. This data reflects the inverse S-box operation. Also included in Figure 4.2 is the only GPU based cipher implementation reported before this work was carried out. The aforementioned Cook et al. AES implementation on a GeForce 3 Ti200 reports a rate of 1.53 Mb/s and is barely visible in the figure. This highlights the advances in programming flexibility and processing capability of GPUs since the release of the GeForce 3.

Comparison Note: As part of the results presented here we include comparative results from CPU based AES implementations. Throughout this thesis we compare results with CPU based implementations, however we should highlight the inexactness of this approach. There are a number of ways to compare a CPU to a GPU. For example, comparisons using transistor counts or die area gives a comparative performance boost to CPUs as they generally consume less transistors per chip than GPUs. Another comparison could be made related to research cost per architecture, which according to informal reports swings the comparative advantage to the GPU. A more useful comparison could be made using the final system price per bit processed, which loosely reflects all constituent factors. However, this is difficult to state precisely. A multi CPU socket system costs more than a single socket, where both include extra PCI slots. Being able to extend a single system with use of GPUs can save costs in expanding to another full system. Also, a further issue with this approach is that it does not serve an academic evaluation where in the near future it may be possible to use a GPU directly in the motherboard in the same way a CPU does [3]. This removes the cost of all GPU card components that are not the chip itself, making it more equitable to the way in which a CPU is used.

A further issue with using price based comparisons is that prices are subject to a number of factors, such as marketing strategy, market conditions, vendor conditions, etc. As such, these types of comparison are not scientific and give questionable value. However, it can be argued that a comparison with the CPU should be made in some form, if even to give rough placement of the GPU in context with existing implementations. To enable such a rough comparison we select where possible similar era, similar priced CPU and GPU chips in isolation.

Looking at the chips in isolation removes factors such as the cost of GPU card components such as RAM, and other system integration issues. CPU chip factory gate prices are generally available in units of 1000. However, GPU prices are only available for end consumer retail prices, which includes the price of the retail channel and all PCI board components. A guess at the price of an individual chip can be determined from the difference in price between single and dual GPU boards. Using an example of the latest high end Nvidia processor, the average difference in price between the single GPU GeForce GTX 285 card and the dual GPU GeForce GTX 295 card is ∼\$140US. As stated this is a rough guess at the factory gate price of a new GPU release, though it could easily be at cost price or a discounted price. The CPUs used for comparison throughout the thesis have a release date typically within one year of the GPU release, and a factory gate price range of \$200-300US. In summary, the CPU comparisons are not exact and act as a background context against which the GPU's performance can be roughly judged.

Also, note that throughout this thesis the cryptographic processing rates are measured in terms of throughput, rather than the more common format, cycles per byte. The reason for this is that on a many core architecture, cycles per byte per core does not give a useful indication of processor throughput. When processing on a many core device, the ALU processing rate is just one factor that determines the throughput rate. Factors such as shared resource contention and availability of threads can have a dramatic impact on the final throughput. A more informative metric is required than cycles per byte for a single core. Thus, we have tended to use the final system throughput for different data sizes.

Comparison: Figure 4.2 includes the performance of running the built in OpenSSL [100] speed test on System 2, see Appendix D. The speed test reported is the rate of repeatedly encrypting in-memory plaintext using AES in ECB mode. The OpenSSL version used was 0.9.8g. It is common to see CPU comparisons using OpenSSL in the literature. However, OpenSSL performance tends to include a lot of API overhead and thus under report the CPU performance potential. To complement these results, Figure 4.2 also includes the throughput rates for AES in ECB mode as reported by the popular Crypto++ [19] API. The Crypto++ result included here is taken from the website's benchmarking pages. The benchmarked CPU is a 2.93 Ghz Intel Prescott, though it is not clear the exact version. Table 4.7 shows the exact CPU throughput rates achieved . The CPU release date and factory gate pricing at release date, where available, are shown in Appendix B.1.

OpenSSL	369.04
Crypto++	**848.0**

Table 4.7: CPU based AES results quoted in Mb/s.

4.4 GPU as an AES Co-Processor

The results shown in Section 4.3.4 are somewhat encouraging in that DX9 GPUs can provide assistance as a cryptographic co-processor. However, it was noted that the operating system reported CPU utilisation at 100% during all runs of the above approaches. Cook et al. [15] also reported the same issue for their implementations. There is little point in using the GPU as a cryptographic co-processor if it must be run in series with CPU tasks. We present a formalised investigation into this behaviour and corresponding results in this Section.

We define `%CPU Idle Time` as the amount of idle CPU time during the execution of a GPU task as a percentage of the total runtime of the GPU task. For example, if a CPU task must run in series with a GPU task, the GPU task has a `%CPU Idle Time` of 0%. Conversely GPU tasks that can run perfectly in parallel with CPU tasks have a `%CPU Idle Time` of 100%. `%CPU Idle Time` for GPU tasks can be calculated as follows:

1. Create a CPU bound task, which requires a known amount of runtime, called `CPU Task Time`.

2. Record the length of time the GPU tasks takes on an otherwise idle CPU, called `GPU Task Time`. The `CPU Task Time` must be sufficiently longer than the `GPU Task Time` such that it starts first and always finishes last.

3. Run both the CPU and GPU tasks together starting the CPU task first. Record the total run time of the CPU task, called `Combined Task Time`.

4. `GPU Task Used CPU Time` = `Combined Task Time` − `CPU Task Time`. This follows as the amount of CPU time demanded by the GPU must be the extra time the CPU task takes to finish when run in parallel with the GPU task.

5. `GPU Task Idle CPU Time` = `GPU Task Time` − `GPU Task Used CPU Time`. This also follows as the amount of time the GPU task consumes that it does not require running on the CPU must be in the form of idle CPU cycles.

6. `%CPU Idle Time` = `GPU Task Idle CPU Time` / `GPU Task Time` × 100.

		GeForce 6600GT			GeForce 7900GT		
		8-bit	4-bit	ROP	8-bit	4-bit	ROP
Multi Input	ROT	96.69%	94.19%	86.75%	87.42%	90.61%	74.84%
	noROT	95.96%	94.10%	85.98%	88.79%	89.79%	74.57%
Single Input SGather	ROT	99.18%	96.75%	N/A	88.06%	93.54%	N/A
	noROT	98.24%	95.32%	N/A	88.65%	92.34%	N/A
Single Input HGather	ROT	98.76%	96.59%	N/A	88.70%	93.02%	N/A
	noROT	98.56%	96.46%	N/A	88.49%	93.34%	N/A

Table 4.8: `%CPU Idle Time` based on 16MB payload sizes.

4.4.1 Results

In Table 4.8 we present the previous GPU based AES approaches in terms of `%CPU Idle Time`. It can be seen that in general the GPU performs well as a co-processor as most of the percentages are quite high. This indicates a high percentage of idle CPU time while the GPU is performing AES operations. There is a notable reduction in `%CPU Idle Time` for scenarios that demonstrate a high throughput rate. This is expected as the amount of CPU overhead will remain more or less consistent across the different GPU approaches for a fixed amount of data processed while the overall execution time has dropped. This results in a higher percentage of the tasks total running time occupying the CPU. As such, care has to be taken when interpreting these figures given that the faster AES approaches are not necessarily disadvantaged over the slower ones in terms of `%CPU Idle Time`, but rather there is a price to pay for the increased throughput rates. Although not practical, the GPU tasks that process at faster rates can artificially generate the same `%CPU Idle Time` as the slower GPU tasks by adding sleep cycles. This table demonstrates that the high throughput rates come at a price of increased interference with CPU tasks per unit of time that the GPU task is running.

4.5 Conclusions

Within this chapter we presented new approaches to solving AES block cipher encryption on DX9 GPU hardware. We have compared each approach's resulting performance to each other and to standard CPU implementations. We have achieved rates of up to 870.99 Mb/s using a Raster Operations Unit based approach and 313.84 Mb/s using a fragment processor based XOR simulation on a GeForce 7900GT. The throughput rates achieved are competitive with the CPU rates presented, achieving similar rates when the amount of input data per payload becomes large. Given the rates achieved on DX9 GPUs, it is possible to use them to alleviate AES loads, or potentially similar cryptographic loads, from a CPU allowing it to spend time on other tasks. It was demonstrated that the GPU performs best using large payload sizes and thus suits applications that require bulk data encryption/decryption. This chapter also demonstrated that the GPU can be used effectively as a co-processor contrary to the operating system reports of 100% CPU load during GPU task execution. The 6600GT results were derived from System 1 and the 7900GT results from System 2, see Appendix D. The CPU timing mechanism used was the `gettimeofday()` function. The results are based on the average timings from multiple independent test executions.

Chapter 5

Symmetric Cryptography on DX10 Hardware

The Nvidia G80 architecture is DX10 standard compliant. It belongs to the first generation of GPU architectures that support integer data units and bitwise operations, key features for cryptographic implementations. The programming model for this architecture has also improved with the release of CUDA, which provides a C-like programming environment for shader definition and execution. Using CUDA and an Nvidia GeForce 8800GTX, the first DX10 compliant GPU and part of the G80 architecture family, we implement 128-bit AES. This implementation is compared to the previous effort in Chapter 4 and other CPU and GPU implementations. This implementation is based on various hardcoded assumptions relating to message sizes, message location, key location, single key, etc. We call this implementation an optimised one and use it for contrast with a more general purpose approach later in the chapter.

A practical consideration, with regards to symmetric-key cryptography, is the GPU's suitability as an accelerator when not used in the context of the hardcoded assumptions. Another practical consideration is the current development overhead associated with using a GPU for cryptographic acceleration. Client applications would benefit from the ability to map their general cryptographic requirements onto GPUs in an easy manner. Also, a challenge exists to achieving high efficiency when processing payloads that include messages using different modes of operation. As we will see, problematic modes of operation are those that are serial in nature. We call these modes, *serial MOO*, as opposed to parallel modes such as CTR, which we call *parallel MOO*.

We present a data model for encapsulating symmetric-key functions in a general manner, which is suitable for use with the GPU. The application of this data model and the details of its interaction with the underlying GPU implementations are outlined. In particular we investigate how input and output data can be mapped to the threading model of the GPU for both serial and parallel MOOs. We show the performance of these modes and use our optimised AES implementation to determine the overhead associated with using a flexible data model and its mapping to the GPU. Also included in the chapter is a study of the issues related to the mixing of symmetric-key modes of operation within a single GPU call.

The main goal in achieving good performance on a GPU processor is to ensure that all processing units are busy executing instructions. This is a challenge with the 8800GTX processor when the following are considered: it can only execute a single kernel at a time; it follows a SIMD-like programming model, thus threads can dictate the runtime of neighbouring threads; its on-chip caches are limited, therefore there can be a demand for a large number of threads to hide memory latency; and it contains 128 execution units. These considerations have a significant influence on design decisions and implementations presented here.

Outline: Section 5.1 presents an optimised implementation of AES, running in ECB and CTR modes of operation, on the Nvidia G80 architecture and shows its performance improvements over comparable CPU and GPU implementations. In Section 5.2 we introduce the generic data model suited to GPUs, which is used to encapsulate application oriented symmetric-key requirements. Section 5.3 describes in detail the steps of mapping from the generic data structure to underlying GPU implementations. We implement different modes of operation using the outlined data model and the optimised AES implementation in Section 5.4. This exposes the overhead associated with moving from a hardcoded to a more general purpose implementation.

5.1 Block Based AES Implementation

5.1.1 Mapping AES to CUDA

As previously mentioned the G80 architecture supports integer bitwise operations and 32-bit integer data types. These new features, which are shared by all DX10 compatible GPUs, simplify the implementation of AES and other block ciphers. This allows for a more conventional AES approach compared to implementations on previous generations of graphics processors. We based our implementations around both the single 1 KB and 4 $\times$ 1 KB pre-calculated lookup tables, which were presented earlier, see Equation 4.1 and Equation 2.1 respectively. Again,

```
/* Index generation */
int index = threadIdx.x + (blockIdx.x × blockDim.x);

/* Input state read */
uint4 state = plaintext[index];
        ⋮
/* Output state write */
ciphertext[index] = state;
```

Table 5.1: Mapping of CUDA threads to input and output message data.

we label these as ROT and noROT approaches. Also, similar to the previous chapter all approaches using the ROT and noROT include the use of an implicit extra table for the last AES round.

Each thread that is created, calculates its own input and output address for a single AES data block. The simple thread to I/O data mapping scheme used for all implementations reported in this section is shown in Table 5.1. Each thread's index relative to all threads within the CUDA grid for a kernel execution is used as the thread's offset into the input and output data buffers. In the table, `blockDim` is the number of CUDA blocks within the CUDA grid, `blockId` is the current CUDA block that the thread exists within, and `threadId` is the current thread index within the CUDA block. Input and output blocks are read and written as 4 component 32-bit integer vectors.

As ECB and CTR are parallel modes of operation, each thread runs largely in isolation from other threads in a single pass to generate an AES output block. As mentioned, to achieve high performance on a GPU or on any highly multi-threaded processor, an important programming goal is to increase occupancy. It is for this reason that we create a single thread for each input block of data rather than processing multiple blocks in a single thread. There is the possibility to create even more threads by allowing multiple threads to co-operate on a single AES block. However, as the generation of each AES state column requires access to all other columns for each round, the synchronisation overhead is too high when compared to the amount of arithmetic work done.

XORs are supported in the programmable section of the graphics pipeline. As such there is no need to use the ROP XOR support, which requires multiple passes of the pipeline, one for each XOR operation. Data is processed as 32-bit integers also improving the performance of XOR over the previous GPU generation. One side effect of using 32-bit integer processing is that the rotations performed in the ROT version of the AES incur an overhead. Switching to bytes would alleviate this by allowing arbitrary selection of components, however this would increase the number of XORs required. We will see that some of the AES implementations are not fully I/O bound on the G80 and the extra rotates affect the final throughput rate.

The following optimised AES implementations are simplified by using a single key for all data to be encrypted, with the key schedule generated on the CPU. The reason for implementing the key schedule process on the CPU rather than the GPU is that it is serial in nature, thus the generation of a key schedule must be done within a single thread. There would be a very high overhead per thread, i.e. per AES block, to generate its own schedule. We address the overheads of using multiple different keys, for use in rekeying and distinct cryptographic sessions, within a single payload later in the chapter.

5.1.2 AES and G80 Memory

Device Memory: We investigated using both textures and linear global memory to store the input and output message data on the device. Through experimentation we found global memory to be slightly faster than texture memory for message data access due to the sequential and non-repeating nature of the memory access. Also, we have ensured that all global memory reads and writes per half warp are coalesced into a single global memory access. Recall that the CUDA 1.0 requirements for full coalescing is for each thread to individually access 4, 8 or 16 byte data units; for memory addresses of each of the data units to be aligned to the data unit's width; for all threads within a half warp to access the data units consecutively; and finally for the data accessed concurrently by the threads within a half warp to be aligned to the total size of the data requested.

The first constraint is easily met considering the G80 supports a single 4-wide integer read. Each thread within a warp accesses a single AES state using a single 16-byte `uint4` read instruction. The data index generated using the thread and block IDs is done so that each thread obtains a consecutive index, that is, thread n will access data at `(baseAddress + (`n `×` `sizeof(uint4)))` and thread n+1 will access data at `(baseAddress + ((`n`+1) × sizeof(uint4)))`. When allocating device memory on the GPU for storing the entire data payload, we use `cudaMalloc()`, which ensures that the returned `baseAddress` from the malloc function is aligned to at least a 256 byte boundary. These conditions ensure that the read and write commands shown in Table 5.1 coalesce fully. All our implementation approaches are based on using linear global memory reads and writes for plaintext and ciphertext data.

Host Memory: An important factor in the performance of transferring data to and from the GPU is whether one uses page locked memory or not. Page locked memory is substantially faster than non page locked memory as it can be used directly by the GPU via DMA. The disadvantage is that systems have a limited amount of page locked memory as it cannot be swapped out to disk, though this is normally seen as advantageous to secure applications as it avoids paging sensitive information to disk. We have seen the DX9 equivalent of this type of memory, pixel buffer objects, in the previous chapter. For CUDA we simply allocate host memory via `cudaMallocHost()` instead of the standard `malloc()` function. All implementation results are quoted using this accelerated system memory. Another minor observation regarding host memory use is that a 3% AES throughput improvement can be experienced when the message data buffer is reused to store the AES cipher output. There are no coherency issues with input and output buffer reuse as each data element is treated by a single thread in isolation. This technique may not be possible to employ depending on the calling API. If the API requires the input buffer to remain unchanged then separate input and output buffers must be maintained.

On-chip Memory: As lookup table access is one of the main performance bottlenecks for our AES approaches, we implemented both ROT and noROT versions using all available types of on-chip memory for the G80. The types used are texture cache, constant cache and shared memory. Shared memory is shared between threads in a CUDA block and is limited to 16 KB of memory per multiprocessor. As previously mentioned, shared memory is divided into 16 banks, where memory locations are striped across the banks in units of 32 bits. 16 parallel reads are supported if no bank conflicts occur and for those that do occur, they must be resolved serially. The constant memory cache working set is 8 KB per multiprocessor. It is also single ported and as such it only supports a single memory request at one time. Texture memory cache is used for all texture reads and is 8 KB in size per multiprocessor. To investigate the read performance characteristics of these types of memory we devised read tests to access the three types of memory in two different ways. We split the tests into random and coherent read memory access patterns, each test accessing 5 billion integers per kernel execution. Coherent access patterns refer to read requests that execute in parallel and are not serialised due to memory porting constraints. We include this access type as there are opportunities to exploit coherent reads within shared memory, i.e. reads with no bank conflicts per half warp request.

In Table 5.2 we can see the average execution times measured in seconds to perform the 5 billion reads. Although we are only interested in the coherent read

	Coherent Reads	Random Reads
Shared Memory	0.204319s	0.433328s
Constant Memory	0.176087s	0.960423s
Texture Memory	0.702573s	1.237914s

Table 5.2: On-chip Memory Reads: Average execution times for 5 billion 32-bit integer reads.

performance of shared memory we have included the coherent read performance of texture and constant caches. We can see that constant memory performs best with regard to coherent reads. However, as the AES lookup tables will be accessed randomly within a warp we are mainly interested in random reads, of which shared memory gives the best performance. This is due to its high number of ports. There is an extra overhead to using shared memory to store lookup tables. Both texture and constant memory can be loaded with data before the kernel is called and used by all threads within the grid of threads, and also by subsequent kernel calls within a CUDA context. Shared memory must be populated once per CUDA block, and repeatedly done so for each new block. Shared memory is designed for use as inter-thread communication memory within the one multiprocessor and is not designed for pre-loading of data. This is most likely an API limitation. If a mechanism existed to populate shared memory separate to a kernel execution it would reduce the overhead of using shared memory for pre-baked lookup tables.

Lookup Tables in Shared Memory: The reason for including the coherent performance of shared memory in the results in Table 5.2 is that there is the possibility of arranging multiple copies of the AES lookup tables to avoid bank conflict in the ROT approach. This arrangement, assuming overhead was limited, would be desirable as we could expect an increase of the final AES throughput rates due to the improved lookup performance. Considering shared memory in the G80 is stated as 16 KB in size, it should be able to store 16 1 KB lookup tables. In this case, each thread within a half warp can use a modified index to lookup the 16 tables such that they access unique shared memory banks, thus avoiding conflicts. We modify the lookup index as illustrated in Table 5.3.

Recall that shared memory is arranged so that consecutive 32-bit words are stored in consecutive banks. As such we can view shared memory as a two dimensional array where there are 16 columns and 256 rows, each column being 32 bits wide. From the brief code in Table 5.3 we can see that `oldindex` is acting as the row selector and `bankId` is acting as the column selector. The new

```
/*Use threadId to select a unique bank.*/
bankId = threadId & 0xf;

/*Modify the original index value.*/
newindex = oldindex × 16 + bankId;
```

Table 5.3: Modification of values used for AES lookup table indexes to avoid bank conflicts.

index guarantees that each thread within a half warp accesses their own column. However, the overhead associated with the extra operations to generate the new index if coded as presented in Table 5.3, under performs the scenario using a single 1 KB table with bank conflicts. Careful optimisation of the operations used to generate the new index can almost entirely eliminate the overhead.

First, the `bankId` can be calculated once and reused for all table lookups. Next, note that `oldindex` in a standard ROT implementation is used to lookup shared memory, for example `shmAESLookup[oldindex]`. If we look at this in PTX CUDA assembly we can see that it breaks down into `oldindex` $\times$ 4 + the address of `shmAESLookup`. We can add the `bankId` offset at the start of the cipher execution to the address of `shmAESLookup`, thus in effect pre-selecting the column (bank) within the shared memory array. The "$\times$ 4" is used by the compiler to calculate the byte offset from the base address. This offset is required as each element is 4 bytes long. This multiplication step is needed regardless of storing integers or bytes, as ultimately we require 4 bytes returned per lookup. Thus, if we modify the PTX assembly to use $\times$ 64, we incorporate the row select for free. It is worth noting that the PTX comments made in Chapter 2 regarding the lack of inline assembly makes this optimisation and code management difficult. We are forced to modify over 1000 lines of assembly, which included over 900 register aliases.

Unfortunately, CUDA and the G80 does not permit full use of all of the 16 KB of shared memory advertised. We only have access to slightly less than 16 KB, as the first 28 bytes are reserved for system use. Many attempts were made to use these few bytes by backing up and restoring them suitably, however system instability could not be avoided. The full 16 $\times$ 1 KB single table approach is thus not feasible. We used a similar approach, which omitted the last entry from each of the 16 1 KB tables. A conditional was added to check the lookup value. If the last entry was being sought, a hardcoded entry was used instead of executing the table lookup. The previously described assembly language optimisations were also applied to this approach. However, the extra conditional adds sufficient overhead to ultimately result in similar performance to the ROT approach with

	Shared Memory	Constant Memory	Texture Memory
ROT	6,214 Mbps	3,964 Mbps	4,378 Mbps
noROT	**7,234 Mbps**	3,942 Mbps	4,336 Mbps

Table 5.4: AES CTR peak throughput rate on the G80, including data transfer, using different types of on-chip memory for lookup tables.

random access patterns.

We also investigated using a compressed lookup table. We can see that the T_n tables listed in Section 2.3.1.3 have values derived from each other. Take table $T_0[a]$ from said list of tables. We can symbolise the values within the table as ABBC, where $A = B \bullet 2$ and $C = B \bullet 3$. Clearly $3 = (1 \oplus 2)$, and considering we are using arithmetic within a field we can write the distributive relationship $C = B \bullet (1 \oplus 2) = A \oplus B$. Thus, given just AB, i.e. 2 bytes instead of the 4-byte ABBC, we can quickly generate the rest of the returned values from the lookup table. However, this extra XOR combined with additional complexity of index modification, leads to throughput rates of less than the 1 KB ROT approach. We include theoretical performance figures given changes both to CUDA and to hardware related to shared memory access at the end of the following results section.

5.1.3 Results

Implementations: Here we present the throughput rates for AES in CTR mode using the ROT and noROT approaches with random access patterns on all types of on-chip memory. Table 5.4 shows the maximum performance of these implementations including payload transfer across the system bus. It can be seen that the noROT approach using shared memory gives the fastest throughput rates. This approach requires the four 1 KB tables to be setup within shared memory for each CUDA block of threads running. The overhead of this setup was minimised by fixing the number of threads per block to allow even distribution of the loads from global memory into shared memory. We use 256 threads per block, thus each thread is responsible for loading four 32-bit integers, which require little overhead for read index offset generation. The ROT shared memory approach under performs due to the extra rotates that must be performed. We could alleviate the rotates if bytes were used, and as such could be accessed arbitrarily, however using bytes would increase the number of XORs to perform as swizzle is not free. In contrast to the difference in performance of the ROT and noROT approaches using shared memory, the approaches using constant memory and texture memory show little variation. This suggests that they are mostly I/O bound on cache reads as the extra rotates have little effect and are being hidden by waits on reads.

As the noROT approach using shared memory shows best peak performance, we focus on this approach for further analysis. Figure 5.1 shows the performance of AES CTR using this approach under different payload sizes. The figure exposes the requirement for a large number of threads to maintain a high occupancy within the GPU, and also for a small amount of GPU-CPU interactions per data element being processed. We also display the throughput rate of this approach without the overhead of message data transfer across the PCIe x16 bus. A maximum rate of 17,571 Mbps was recorded without transfers, highlighting the data transfer overhead when compared to a maximum of 7,234 Mbps with transfers included. We have included the rates without data transfer for the following reasons: some cryptanalysis techniques do not require a pre-baked data element per cipher execution; newer hardware than the G80 supports the overlapping of transfer and computation on the GPU, thus if efficient pipelining is possible on these cards the throughput could approach those reported here; there is a possibility that future GPUs will share the system memory with the CPU, either in the form of a combined processor or a direct motherboard processor slot.

All the results in Table 5.4 and Figure 5.1 are generated by taking the average of multiple kernel executions. The number of executions is determined by the payload size, varying from 50 for the largest payloads to 50,000 for the smallest. The reason for the variance in the number of executions, is that for small payloads each execution runtime is small. External factors, such as OS scheduling, can have a proportionally large influence on the reported runtime, thus requiring a high number of iterations to give a reliable average throughput rate. Also, all the results were generated by executing the tests on *System 3*, see Appendix D.

CTR and ECB: Also included in Figure 5.1 is the noROT approach running in ECB mode with data transfer included. The peak performance achieved was 6,989 Mb/s. AES in CTR mode can perform better due to using the input message data at the end of the block cipher, rather than at the start. Recall that CTR mode performs the cipher on a nonce combined with a counter. The output of which is XORed with the message data. Thus we can issue the read for message data early. While this read is being carried out, we can proceed with the cipher execution. In contrast, ECB requires the result of the message data read before it can proceed with cipher execution. Thus, in CTR mode we can hide, or partially hide, the read data read latency. This can benefit the overall throughput if the amount of time saved hiding the read latency is greater than

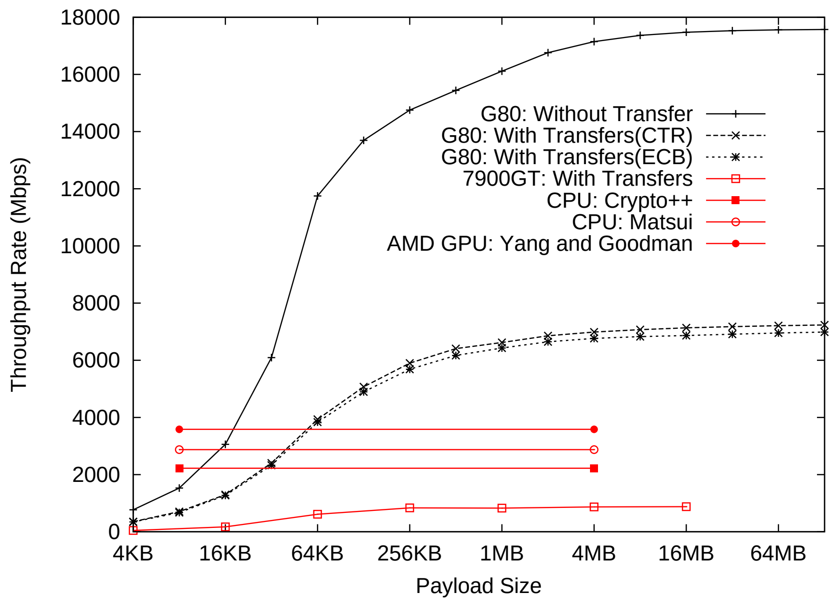


Figure 5.1: G80 AES implementations with and without data transfers across varying payload sizes.

the time lost handling the counter and nonce.

We optimise the CTR implementation so that the application of the counter to the 128-bit nonce consumes a single 32-bit addition without carry per AES message block. Each thread is responsible for adding their offset within the CUDA grid to the least significant 32-bit value of a 128-bit pre-offset nonce without regard for carry. The CPU is responsible for adding an offset to the nonce, reflecting the number of message blocks that have been processed within a single cryptographic session. This produces the 128-bit pre-offset nonce for use by the threads. In this way threads do not calculate the global offset within the cryptographic session. To handle the issue of the abandoned carry, which would have to be manually handled on GPU hardware, we employ the CPU. The CPU is responsible for detecting that the number of threads within a grid will cause a wrap of the 32-bit least significant integer of the pre-offset nonce. In this case the CPU splits the CUDA kernel call into two. The first kernel execution uses up the remaining values in the least significant integer. The CPU then performs the increment with carry on the higher significant integers. Then the second grid is executed with the remaining threads allowing the least significant integer to wrap without carry. This way the CPU carries out this check once per kernel call and avoids each thread performing the addition with carry across the 128-bit value once per message block.

Related to the ability to concurrently execute kernels and transfer data on newer GPUs, the decoupling of data reads from cipher execution in CTR mode can be used to not only increase throughput, but also to reduce latency. This can be achieved by executing a transfer of message data asynchronously. This is followed by the execution of a kernel that generates the key stream. A second kernel call is then executed to perform the XORing of message data with the key stream. Thus the transfer of input data for use in the same cipher execution are overlapped.

Comparison: Figure 5.1 also compares our results with other AES implementations on CPUs and GPUs. Again, we use the Crypto++ benchmarking figures available on the website to report a similar era CPU to the GPU. Figure 5.1 includes Crypto++ AES throughput rate for CTR mode running on an Intel Core 2 Duo E6300 1.86 GHz. The benchmarked figure only reports performance for a single core. We simply double the reported rate to arrive at 2,224 Mbps, though this may be overly optimistic as there will be bus contention between cores. The rate for the same processor running AES in ECB mode is 1,774 Mbps. Matsui [67] claimed the fastest AES speeds on a 64-bit AMD Athlon processor producing a rate of 170 cycles per 128-bit AES block. We scale this to the dual core Athlon 64 X2 3800+ 2.0 GHz to produce speeds of 2,872 Mbps. Yang and Goodman [129] cite a speed of 3,584 Mbps on an AMD HD 2900 XT GPU (AMD's DX10 compliant GPU) for their block based AES implementation. We do not have access to the throughput rates as data sizes increase, hence its illustration as a horizontal straight line, which does not reflect the performance as the buffer size increases. We have included the rates achieved in Chapter 4 for AES on a GeForce 7900GT, which serves to provide a guide to the rate progression from one generation of Nvidia GPU to the next.

With data transfers included for the G80 implementation presented, we see a significant speed up over the fastest CPU reported implementations. Without transfer rates included we see a speed up of over 6 times. When compared to the block based AES implementation on AMD's first DX10 GPU by Yang and Goodman we can see approximately a 2x and 5x speed up with and without data transfers respectively. The increase in speeds, when data transfer over the system bus are not included, clearly illustrate the benefit that would be gained from sharing system memory, or a fully pipelined implementation using newer Nvidia GPUs.

Potential: Regarding important advances to current Nvidia hardware and their implications on AES execution. An increase in available shared memory to the full 16 KB would allow an implementation using the zero conflict shared memory packing approach explained in Section 5.1.2. If shared memory were increased on future GPUs to 64 KB we could implement the noROT approach similarly. Another improvement in performance would be gained if the CUDA API supported pre-population of shared memory, rather than requiring each block to do so. To estimate the performance of such improvements, assuming all other hardware components remained the same, we executed both the ROT and noROT approaches with the same block data to simulate zero shared memory conflicts. These resulted in 18,704 Mb/s for the ROT approach and 25,560 Mb/s for the noROT approach assuming perfect pipelining is possible. These figures serve no immediate practical use apart from highlighting the bottlenecks in the current implementations and an indication of what conditions will help alleviate them.

5.2 Payload Data Model

In this section we present a data model, which we use to explore the issues involved in mapping a generic symmetric-key cryptographic service to GPU implementations. The aim of this section is to outline the data model, its design criteria and the usage implications in the context of GPUs. The data model is seen as being used within an encapsulating runtime, facilitating the client's abstraction from underlying GPU based cryptographic implementations. The runtime can either expose the data model directly to the client, or via a wrapper layer adding higher level functional support such as session management.

5.2.1 The Data Model

Table 5.5 illustrates a payload data model suitable for processing symmetric-key data on the GPU. In this context, the term payload indicates a single grouping of data that contains both data for processing and related meta data. The data model described is similar to standard symmetric-key models and their associated APIs. The GPU, as we have seen, requires a large amount of data to achieve high throughput rates. This requirement can be met using large messages or the combination of multiple smaller messages. Standard cryptographic data models expose support for multiple messages through disparate data pointers. This type of model does not suit the GPU for smaller messages as it requires either multiple data transfers across the system bus, or multiple copies to perform one single system bus transfer. Both are inefficient, especially in the context of AES where we have seen it is already I/O bound. Also, it requires the conversion

```
struct payload                       struct msgDscr
{                                    {
 unsigned char        *data;          struct   elementDscr *msg;
 struct   payloadDscr *dscr;          struct   elementDscr *iv;
};                                    struct   keyValue  **msgMode;
                                      unsigned char        *key;
struct payloadDscr                    unsigned int         keySize;
{                                    };
 unsigned int         id;
 struct   keyValue    *payloadMode;  struct elementDscr
 unsigned int         msgCount;      {
 unsigned int         size;           unsigned int count;
 struct   msgDscr     *msgs;          unsigned int offset;
};                                    unsigned int size;
                                     };
```

Table 5.5: The Payload Data Model

of data pointers into offsets as pointers lose their meaning when sent across the system bus. We present a data model that supports multiple messages within a single data stream. The use of a contiguous block of virtual memory for message data storage facilitates a single copy as in the optimised implementations shown previously. The data model also supports indexing into said streams using offsets rather than pointers, and the mixing of different types of symmetric-key functions for use within a single kernel call.

In Table 5.5 we can see the primary data model components used. The main `payload` structure contains a pointer to a single message data stream, either plaintext or ciphertext. This stream is designed to hold one or more messages, with potentially varying keys. A payload descriptor structure, `payloadDscr`, is also referenced, which is used to provide all information related to the data stream required for its processing. The payload descriptor uses an ID to uniquely identify payloads in an asynchronous runtime environment. The cryptographic service to use can be set within the payload descriptor or within the individual message descriptors. We include a higher level mode, `payloadMode`, to describe such values as cryptographic families. This allows an encapsulating runtime to quickly match a payload to suitable hardware accelerators. A lower level property, `msgMode`, can also be used to describe the cryptographic service on a per message basis as can be seen in the message descriptor structure, `msgDscr`. The message descriptor structure is responsible for describing all information required to carry out the specified cryptographic function for a message. The message descriptor shown contains the primary meta data required such as references to the message data, initialisation vector (IV) and key. The references into the data stream use the generic `elementDscr` descriptor, which allows the description of any data unit within the data stream using address space independent offsets. The element descriptor separates the concept of element size and count as the size of elements can sometimes indicate a functional difference in the used cipher. The return payload is similar to the payload structure.

5.2.2 General Use Implications

A consideration regarding the use of per message properties to indicate cryptographic function, is that it can severely impact performance. The GPU can only execute a single kernel across all threads, any variation in function must be implemented using conditional branches. The technique used for the execution of a large variation in kernel code is called fat kernels. A fat kernel inserts conditional branches at the start of each thread indicating the algorithm to run. As the GPU is a form of SIMD processor, any mix of functions within a warp requires the algorithms to be executed serially. Also, any mix of functions within a CUDA block, ensures that all threads within that block consume the time required to run the longest thread. In general it is best for performance if all messages within a payload use the same function, which is determined before kernel execution.

Another concern when employing this data model for use with an attached processor, such as a GPU, is memory allocation for I/O buffers. For the G80 it is important to use page locked memory, which requires a request to be made to the CUDA library. The CUDA library then returns a pointer to the memory requested that can be used within the calling process. Both the input and output buffers should use page locked memory and also reuse the same buffers where possible for maximum performance. It is presumed that one of the motivations for using a generic data model is to abstract the client from implementation details such as CUDA. Thus, there is a need to support client application requests for buffers in the runtime. The encapsulating runtime should take the responsibility of executing the CUDA allocations as required on behalf of client requests. Allowing the application to provide its own standard system buffers would require an extra copy into accelerated CUDA memory, which would counteract the use of accelerated memory in the first place.

5.3 Applied Data Model

In this section we cover implementation concerns when bridging between the previously described data model and specific GPU cipher implementations. In particular we focus on our implementation of a *runtime layer*, which maps the data model to our specific cipher modes of operation presented in Section 5.4. The overhead of providing a general purpose interface to a GPU implementation is the addition of abstraction layers that need to be resolved within each kernel thread. Throughput is lost when message functions, sizes, element types, etc., can vary within a payload. Each thread must perform extra memory accesses, calculations and conditional branches to act upon the dynamic settings. These per thread calculations can be offset by an implementation using the CPU as a preprocessing stage which optimises a payload for thread parsing before the payload is dispatched. Naturally there is a practical limit to the amount of CPU preprocessing employed, as one of the potential uses of a GPU is to act as a co-processor, which should speed up the overall throughput of a system. The per thread overheads and CPU pre-processing can be easily seen in the following sections.

5.3.1 Descriptor Serialisation

Each element in each message descriptor is serialised on the CPU into a form that can be used independently and quickly by each thread on the GPU. The runtime determines a fixed byte size for a serialised message descriptor. The serialisation of all message descriptors produces a *message descriptor stream*. Thus, given a descriptor ID, which start at 0 and increment for each subsequent message descriptor in the stream, a thread can calculate the required offset into the message descriptor stream using a fixed multiplier. A *key schedule stream* is generated on the CPU, either from cache lookups or dynamically executing key schedules, avoiding redundant entries. A corresponding *key descriptor stream* is generated, which contains a key descriptor entry for each message in the payload. Each key descriptor contains the offset and size information required to support thread recovery of the appropriate round keys from the key schedule stream. Given a descriptor ID, threads calculate an offset into the key descriptor stream in a similar manner as with the message descriptor stream. The serialised message descriptor, key descriptor and key schedule streams are transferred to the GPU and stored within texture memory address space. This gives the best size flexibility of the cacheable memory types.

LThread2Dscr Index: During the descriptor serialisation process, we can order message descriptors in various ways. This can have an influence over the order in which messages are processed by threads. We will see later how message ordering can be important to performance. However, we first require an efficient way for threads to map to a particular message and key descriptor. We adopt the term *physical thread ID* to denote the normal ordering of threads within a CUDA grid, returned by the calculation `threadIdx.x + (blockIdx.x` $\times$ `blockDim.x)`. A naive method for mapping threads to descriptors is to use the physical thread ID directly as the descriptor ID. However, we quickly see that this requires heavy duplication of descriptors in the descriptor stream, as parallel mode messages use multiple threads per message. An improvement to this is to use a separate index, which maps physical IDs to descriptor IDs. However, the index is large as it stores one entry per thread, and each entry stores both the descriptor ID and thread offset into the message for use with parallel mode messages. A large index leads to poor cache behaviour during lookup and a high CPU overhead in its generation. The GPU approaches maximum AES performance at 512 KB payloads, which if containing parallel mode messages would result in an index with 2^{16} entries.

A further improvement to the index is to use a compressed version, where only the physical ID of the first thread of a message is stored. Thus, for parallel mode messages, the physical ID of the thread that processes the first block of the message is entered into the index, whereas the thread that processes the whole serial mode message is used. A thread's offset into its message is generated by subtracting the thread's physical ID from the physical ID retrieved from the index. The descriptor ID is the offset into the index itself as there is one entry per message. Thus only a single integer per index entry is required. This produces a small index with good caching behaviour and a reduced overhead during index generation at the expense of having to search the index. In tests the benefits of the compressed index outweigh the search overhead.

The disadvantage to the previously described approach is that it does not permit non-consecutive physical IDs to map to the same descriptor ID. We show later that it is desirable to allow non-consecutive threads to work on a single parallel mode message for balancing work across the GPU processing elements. To support this, we maintain the same compressed index as before, however instead of using physical IDs to lookup the index, each thread generates a new ID according to a load-balancing scheme. We call this new ID a *logical thread ID*. The compressed index is generated as previously described, however the IDs stored

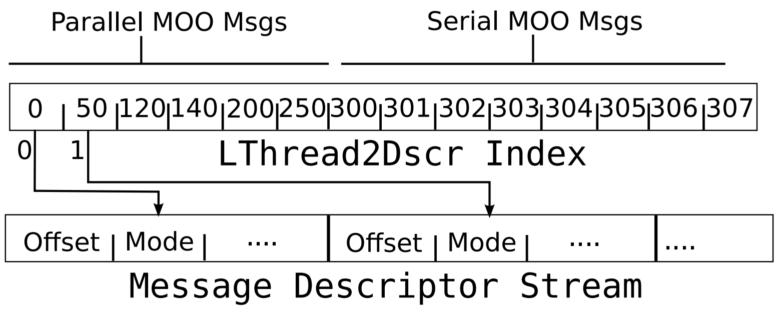


Figure 5.2: Serialised streams used by each thread for data and key retrieval.

in the index are now used as logical IDs, thus we call it the *LThread2Dscr index.* The primary load-balancing schemes investigated are presented in Section 5.3.2. Figure 5.2 shows an example of the LThread2Dscr index and how it relates to the message descriptor stream.

Key Support: As outlined previously, the GPU is a highly parallel device and the key schedule generation is inherently serial, thus in general it makes most sense to implement key expansion on the CPU prior to payload dispatch. Only in the case where serial modes of operation are being executed on the GPU with sufficient numbers of messages with sufficiently varied keys does it make sense to implement the key schedule on the GPU. Our runtime layer implementation handles the generation of the key stream and uses a CPU based cache for storing key schedules to ensure key reuse across messages. This is not just to aid efficiency at the key schedule generation stage on the CPU but also to generate the smallest key schedule stream possible. This is important for on-chip GPU caching of the key schedules. An alternative approach is to cache the entire key schedule stream to save on rebuilding the stream each client call, though this was not implemented here.

5.3.2 Thread to Message Mapping

The full process for mapping a physical thread ID to a descriptor ID and its underlying data is the following, this is also shown in simplified form in Figure 5.3.

1. *Generate the LThread2Dscr index as outlined previously.* This work is carried out on the CPU.

2. *Map physical thread IDs to logical thread IDs within each kernel thread according to a load-balancing scheme.* Firstly, we should note that CUDA

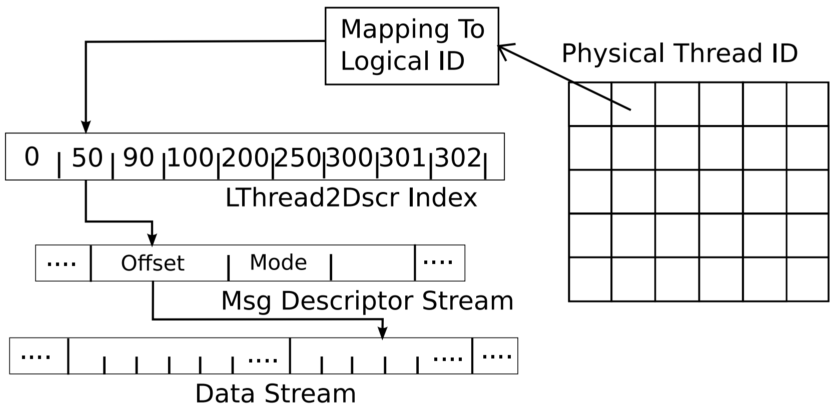


Figure 5.3: Mapping physical thread IDs to descriptor IDs to message data.

only supports a grid with the same number of threads per block. Also of note is that the programmer cannot control the assignment of CUDA blocks to the SMs. This process is controlled automatically by hardware. A further consideration is that the number of threads per CUDA block used for the AES implementations described earlier is 256. This number of threads is to ensure the simplest form of shared memory configuration for lookup table population, see Section 5.1. As such, each CUDA block within the grid contains a full 256 threads. Given these restrictions it is still desirable to influence the distribution of workload across threads. For example, given a payload, the runtime should aim to distribute the associated workload equally across all SMs. If the work in a payload is assigned to each thread in physical ID order, the first CUDA block of 256 threads would be assigned to before moving on to the next. This assignment scheme would result in a kernel execution with threads competing for resources on occupied SMs while other SMs remain idle.

An approach to solving this potential imbalance of work between the SMs is to first generate a CUDA grid size in multiples of the number of SMs on the GPU, i.e. in thread groups of 16×256 for the 8800GTX. This ensures that there is at least one CUDA block per SM. As the number of threads configured to run on the GPU are assigned in groups of 16×256, a lot of scenarios will result in groups of threads in which only some threads are required to do productive work. This we call a "partially occupied" group. Conversely if each thread within a group is occupied we call this a "fully occupied" group. Given the total number of threads required for processing a payload, each thread can determine if they are assigned to a fully occupied group, or a partially occupied group.

If a thread belongs to a fully occupied group, it uses the physical thread ID as its logical thread ID. If the thread belongs to a partially occupied group it calculates a logical thread ID such that an equal number of threads per CUDA block are considered active. That is, there is not enough work for the number of threads in the group of 16 CUDA blocks and as such we assign an equal number of threads per block to execute productive work. Within such a group, the number of productive threads per CUDA block is calculated. If a thread's ID within the CUDA block is above this number, it only participates in populating shared memory and returns. This load-balancing scheme distributes productive threads equally across CUDA blocks. For large numbers of threads, the scheme is quick to execute as a single check can eliminate the case of fully occupied groups where physical and logical IDs are the same. The shortcoming of this approach is that it assumes the work done per thread is equal. It assigns threads on a CUDA block by CUDA block basis, effectively assigning messages to threads within a block (or partial block) before moving onto the next. This makes it difficult or impossible to group costly threads working on serial MOO messages in a manner that distributes them across the SMs.

A second load-balancing approach aims to alleviate this problem. This approach maps physical thread IDs in groups of 32 striped across each CUDA block to consecutive logical threads. By striping the threads across CUDA blocks it allows the possibility of balancing groups of 32 serial MOO messages across SMs, then mapping the rest of the threads within a 16×256 thread group to parallel MOO messages. The size of 32 threads is used to match the warp size. Using groups of non-warp size could unnecessarily mix threads working on serial and parallel MOO messages within a warp, resulting in idle SPs. We use the second approach in our reporting of results later as it is $\sim$0.25% slower than the first, and as we will see the advantage can significantly outweigh the overhead. See Section 5.4.3 for the effects of load-balancing for mixed mode payloads. Note that both load-balancing schemes presented here are examples of scenarios where non-consecutive physical IDs are potentially mapped to a single descriptor ID.

3. *Search the LThread2Dscr index with logical thread ID to determine descriptor ID.* Due to storing a compressed form of the logical thread IDs within the index the search is implemented as a binary search. A direct lookup table could be used for seemingly better performance, however as mentioned previously the overhead of the resultant lookup table size is too high. Each

thread uses its logical thread ID to find the index entry with the closest, but not greater than, value. This entry's offset into the index is used as the descriptor ID. This step also calculates the thread's offset into its message, i.e. which block within the message it should operate on. The difference between the thread's logical ID and the value of the entry retrieved from the index is used as the message offset.

4. *Use the descriptor ID to offset into the message descriptor stream and the key descriptor stream.* The descriptors are used to retrieve the input data and other message settings required to perform the cryptographic function.

5.3.3 Padding

Some modes of operations can require padding when the input data is not a multiple of some bit length. As the ability to generate a linked list of addresses for use during DMA transfer is not supported in CUDA, it is not practical for the CPU based serialization process to support the pre-padding of messages directly into the data stream for sending to the attached device. The reason for this is that the CPU would have to generate a new single stream of contiguous memory based on the new padding insertions and the original data stream. An alternative more efficient approach is to delegate padding to the GPU. This requires that each thread checks if padding is required and to generate the extra input data itself. In relation to CUDA this extra check causes thread divergence for the single thread that must execute the padding. However the overhead is generally very low as the divergence only lasts for a single cipher block across 32 threads. There is an issue with GPU delegation as a thread cannot allocate its own memory. This is problematic when the output buffer is too small to hold the padded output. One can enforce output buffers that have a size of some multiple, however some scenarios can require a full extra block for padding, as in PKCS#5 [115]. The CPU interface must ensure there is enough allocated output memory for padding requirements.

5.3.4 Payload Combining

The runtime layer implementation can easily implement payload combining in the scenario where payloads are queued via an encapsulating framework. The multiple data and key schedule streams within host memory space can be copied into consolidated input buffers on the attached device. During serialisation stage, the serialised message and key descriptors are appended and offsets are recalculated taking into account the combined input streams on the attached device. Similarly processed payloads can be read from a consolidated output buffer on the attached device and read into separate host buffers. Generally ciphers do not change the size of the plaintext and ciphertext, padding aside, allowing efficient reuse (directly or copies) of the input payload descriptors. Although combining is possible, it has questionable value as the overhead of the extra copies and recalculation could outweigh benefit of less GPU calls and higher occupancy.

5.4 Modes Of Operation

In this section we present the implementation and results of symmetric-key modes of operation built using the previously described data model, runtime layer and AES implementation. Modes of operation determine how the underlying block cipher is used to implement a cryptographic system which supports messages greater than one block in length. We have analysed the most common encryption modes, specifically CTR, CBC, CFB and OFB. Using these modes on a highly multithreaded device, the major overriding characteristic which determines throughput is whether the mode can be implemented in parallel or must be done serially. We focus on the throughput of the two main categories of MOOs: serial MOO (CBC and CFB encryption and OFB), and parallel MOO (CBC and CFB decryption and CTR). All implementations are based on the optimised AES implementation presented in Section 5.1 using CUDA. Also, all results are based on using the same system and methodology as specified in Section 5.1.3. Discounting block cipher performance variation, these results should provide a guide to the general behaviour of the investigated MOOs using other block ciphers on a GPU.

5.4.1 Parallel MOOs

It is easier to achieve full occupancy on a highly parallel processor such as a GPU when processing parallel MOO messages compared to serial MOO messages. Each message can be split into blocks and assigned its own thread, thus the number of threads equals the total number of blocks within the payload. Figure 5.4 shows the throughput rates of different message sizes used within payloads containing parallel MOO messages. The results shown are based on CFB decryption. CTR and CBC decryption were also implemented, though the throughput rates did not vary excluding the aforementioned performance benefit of CTR. The number of messages indicates the number used within a single

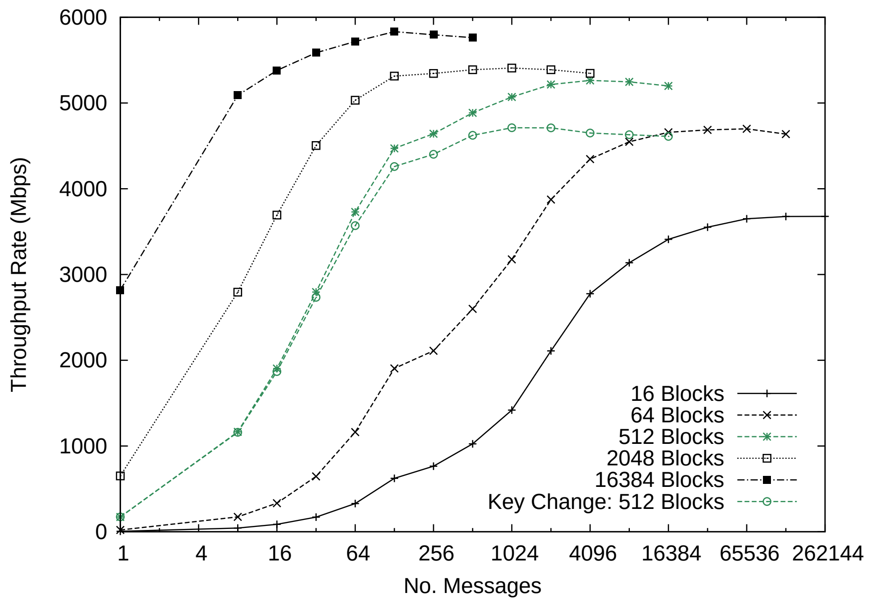


Figure 5.4: Throughput rates for parallel MOO messages across varying message numbers and sizes.

payload. As we can see, the greater the payload size the better the performance. This is to be expected as the increased resource occupancy and memory latency become more effectively hidden. We can also see that at a certain throughput rate the per message overhead of using a generic data model becomes the dominant overhead. As a result, increasing the payload message count past a certain point results in a drop in performance.

All results are based on multiple executions of a single payload with the reuse of accelerated message input and output buffers both for host and on device storage. This simulates the scenario where a client application either uses CUDA calls itself or the encapsulating runtime executes the required CUDA calls on the client application's behalf. Most results are also based on key reuse, simulating a scenario where all messages are from within a single cryptographic session. In contrast we include in Figure 5.4 throughput rates for an extreme mixed key scenario, whereby each message uses its own unique key. We have highlighted the comparison of rates with and without key change for payloads with a message size of 512 blocks. As expected, an increasing message count results in an increasing overhead on total throughput.

The maximum throughput achieved for a parallel MOO under the generic data model was 5,860 Mbps. An important observation to be made from these figures is that there is an overhead associated with using the described generic data model for abstracting the underlying implementation details. We use the optimised ECB implementation presented in Section 5.1.3 to generated overhead percentages. ECB is more comparable to CFB decryption as it immediately uses data read from device memory. When using large messages (16,384 blocks) the overhead is ∼16%, with medium sized messages (512 blocks) the overhead is ∼22%, and in the worst case when using small messages (16 blocks) the overhead is ∼45%. The reason for this increase in overhead is partly due to the degradation in caching behaviour when using larger index and descriptor streams. Also as the number of messages increases the overhead associated with index generation and search, and the descriptor stream serialisation increases.

5.4.2 Serial MOOs

The payload must have a high message count when processing serial MOO messages on the GPU to give good performance. Given a small number of messages, there will be a shortage of threads to maintain a high occupancy level on the GPU and thus performance will suffer. The serial implementations follow the same thread to message mapping process as described previously. The message descriptor contains the message size for serial messages, which is used to set the number of input blocks to be processed by a single thread starting with the initialisation vector (referenced via the message descriptor). The thread starts at the message offset within the data stream and consumes single blocks per cipher iteration. This creates a memory access pattern where neighbouring threads access memory locations separated by the size of the message they are processing. This access pattern has an important impact on throughput as will be seen.

Figure 5.5 shows the performance rates for a serial MOO using different message sizes. All results are based on the CBC MOO in encrypt mode, other serial MOOs using the same block cipher performed equivalently with regards to bulk throughput rates. All messages within a single payload were the same size, see Section 5.4.3 for detail on mixing sizes of messages within a payload. We have included in the figure the performance for CBC MOO from Crypto++ on the same CPU used in Section 5.1.3. We have also included the results for a parallel MOO for a payload with a message size of 2048 blocks from Figure 5.4. The figure highlights the comparison of this parallel MOO with the corresponding serial MOO message size. We can see the penalty paid for a small number of serial messages. The parallel MOO scenario significantly outperforms the serial MOO scenario at low message counts. We also see that performance can be gained by grouping message blocks per thread. This reduces the per message overheads such as index and stream lookups. The overhead reduction accounts for the better performance achieved by the high message count serial payloads

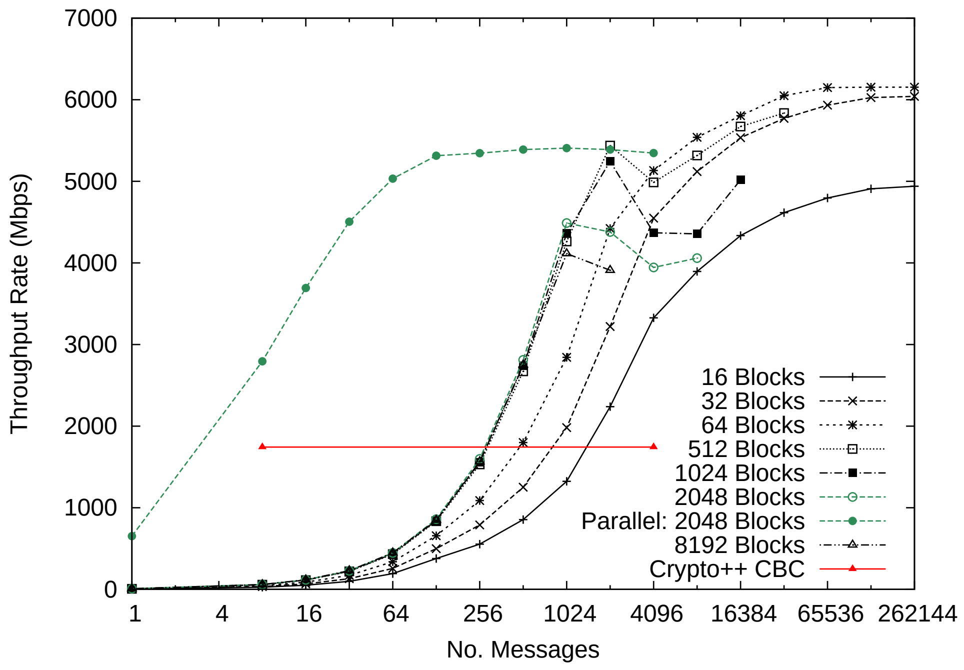


Figure 5.5: Throughput rates for serial MOO messages across varying message numbers and sizes.

over parallel payloads.

A negative performance trend can be observed for larger serial MOO message sizes: as the number of messages increase a performance bottleneck is hit. This may be explained by the memory access pattern created by such executions. Neighbouring threads within a CUDA warp use increasingly disparate memory address locations for their input and output data as the message size increases. We have isolated this behaviour with a separate memory test in which each thread performs a series of sequential reads from global memory starting at an offset from the previous neighbouring thread equal to the number of sequential reads performed. Figure 5.6 presents these results for different offsets and corresponding sequential reads in increments of 16-byte blocks. Each block is read in the same manner as the AES implementations, i.e. with a single `uint4` read. For block counts of 128 and over the memory read performance drops dramatically as the the number of active threads increase. There is not enough publicly available information on the G80 to definitively explain this behaviour. It is possibly a combination of a level 2 cache bottleneck and a limit on the number of separate DRAM open pages supported by the DRAM controllers.

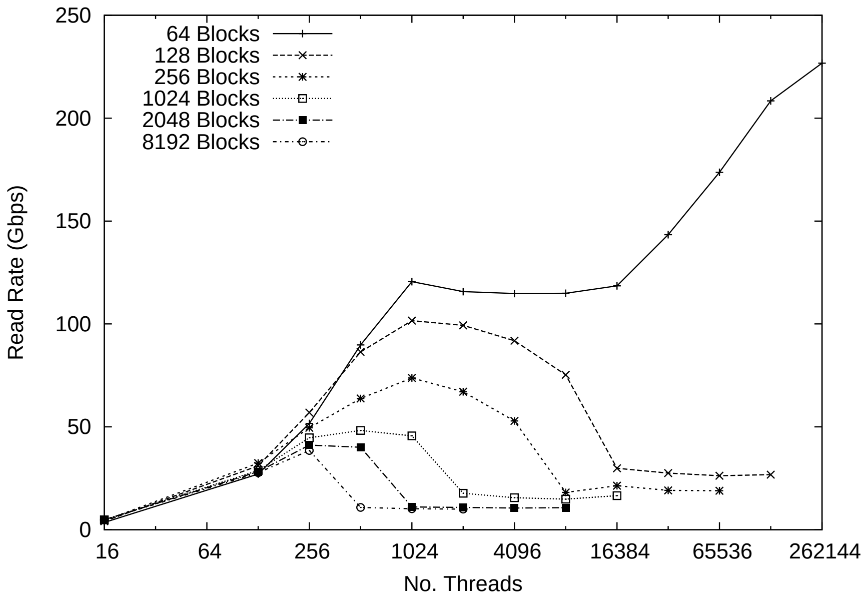


Figure 5.6: Global memory read performance with varying stride patterns.

5.4.3 Mixed MOOs and Message Sizes

Here we investigate the issues involved in mixing both MOO types and message sizes used within a single payload. The mixing of serial and parallel MOO messages within a payload can be beneficial, for example if a small number of serial MOO messages are present in a payload, the presence of parallel MOO messages can help increase occupancy. However, mixing can also result in poor performance, for example, if threads within a CUDA warp work on different MOO types, it can result in idle SPs and a reduction in occupancy. Threads working on serial MOO messages are, in general, extremely costly compared with threads working on a parallel MOO. The first thread type works iteratively across all blocks within a message, whereas the second thread type works on a single block. Thus when mixing MOO types within a payload, the positioning of serial threads (those working on serial MOO messages) within the CUDA grid is important. A reasonable load-balanced scenario is for all serial threads to be divided evenly across the multiprocessors. Also, as the warp size is 32 threads, serial threads should also be grouped into 32 so as not to mix MOO types within a warp. Message size variation within a warp of threads working on serial MOO messages can lead to idle SPs. To minimize this, neighbouring serial threads should operate on messages of a similar size.

To satisfy this distribution of serial threads, we use the striping load-balancing scheme presented in Section 5.3.2. This scheme ensures that consecutive logical thread IDs are striped in groups of 32 across CUDA blocks. Thus, to distribute

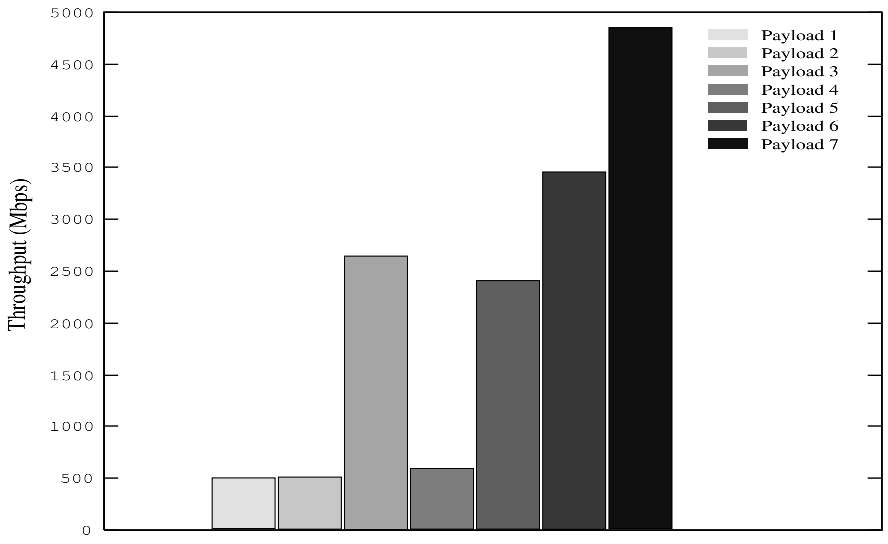


Figure 5.7: Throughput rates for different payload packing configurations.

serial threads in groups of 32 across CUDA blocks we group all serial MOO messages together. Considering logical thread IDs map consecutively to message descriptors, we can group all serial MOO messages together by either the client application directly ensuring that messages are in a suitable order within the data stream, or by the re-ordering of the descriptors on serialisation. For our tests we used the simpler first approach. To ensure that serial threads work on similar sized messages, we order the group of serial MOO messages according to their size. This re-ordering of messages is achieved in the same manner as the aggregation of the serial MOO messages. Using the striping load-balancing scheme, parallel MOO messages do not require any further re-positioning.

Using the striping load-balancing scheme we processed a number of tests to highlight the importance of message ordering to performance. Each payload consisted of the same messages, only the ordering of the messages within the payload was changed. The messages used were of varying MOO type and sizes. Figure 5.7 shows the throughput rates of different payload configurations. The relative difference between the scenarios' performance clearly shows the importance of correct ordering of message types and sizes within a payload. The absolute throughput rates are not important as the payloads used were configured to fit the test requirements and not for performance. All payloads used 960 512-block parallel MOO messages, 992 32-block parallel MOO messages and 1024 serial MOO messages with 8 variations in message size ranging from 16 to 2048 blocks. Here is a description of each of the payload configurations used in Figure 5.7.

Payload Configuration 1: The payload messages are ordered such that the thread distribution results in a single serial thread per warp. All other threads within the warp are parallel threads, thus creating 31 idle SPs. Also, recall that threads are created in groups of 16 $\times$ 256, i.e. 16 CUDA blocks per group. The ordering of this configuration is such that these warps are only placed in the first CUDA block of each group of 16 blocks. Thus, all other blocks contain parallel threads. This configuration is devised as a worst case scenario.

Payload Configuration 2: Similar to configuration 1 except that the warps are spread evenly across all CUDA blocks. The performance is similar to configuration 1 as the GPU dynamically schedules the CUDA blocks when an SM becomes available.

Payload Configuration 3: All serial MOO messages are ordered such that the serial threads are assigned to the minimum number of CUDA blocks. This scenario is much faster than 1 and 2 as all SPs within an SM are occupied, even though not all SMs are occupied with work.

Payload Configuration 4: This configuration randomly distributes the serial MOO messages across the payload.

Payload Configuration 5: This is the same as configuration 4 except that the serial MOOs are grouped into units of 32. This is to ensure that serial threads are grouped into warps.

Payload Configuration 6: All serial MOO messages are grouped together, thus the resultant serial threads are grouped into units of 32 and distributed evenly across all CUDA blocks.

Payload Configuration 7: This is the same as configuration 6 except that all serial MOO messages are also ordered according to their size. All other payload configurations use a random ordering of message sizes.

From the results one can see the impact of serial MOO message positioning within a payload and the resultant serial thread distribution. It is clear the importance of serial thread grouping within the device's SIMD width to ensure the SIMD slots are occupied. It can also be seen the importance of distributing serial threads across the GPU multiprocessors. Furthermore, it is clear the impact of ordering serial MOO messages according to their size. A separate and notable concern when mixing function types within a payload is that the underlying implementation can suffer from increased resource pressure. The G80 only supports a single block of code that executes for all threads, thus the support of multiple functions within a single kernel via conditional code blocks can increase resource usage and also increase overhead for the execution of such conditions.

5.5 Conclusions

In this chapter we have presented an optimised AES implementation using ECB and CTR mode of operation on an Nvidia G80 GPU, which when including data transfer rates, shows a ∼2.5x speed up over a comparable CPU implementation. With transfer rates across the PCIe bus not included, the speed up increases to ∼6x. Comparing to the next fastest GPU based AES implementation, we see a ∼2x increase in performance. We have also investigated the use of the GPU for serving as a general purpose symmetric-key cryptographic processor. The investigation covers the details of a suitable general purpose data structure for representing client requests and how this data structure can be mapped to underlying GPU implementations. Also covered is the implementation and analysis of both major types of encryption modes of operation, serial and parallel. We expose the issues and potentially preventable caveats when mixing these modes of operation within a single kernel execution.

We show that the use of a generic data model and mapping scheme results in an overhead ranging from 16% to 45%. This overhead occurs most acutely during the processing of parallel MOO payloads with small messages and a high message count. It could be argued that in such a case a client would be better implementing a hardcoded approach if the input data structures are known in advance. Overall we can see that the GPU is suitable for bulk AES processing. It can also be employed in a general manner while still maintaining its performance in many circumstances for both parallel and serial modes of operation messages. Despite the overheads of using a generic data model and mapping scheme for the GPU, the performance is greater than competing implementations assuming chip occupancy can be maintained. However, when small payloads are used, the GPU underperforms executing both the general and hardcoded implementations. This is due to resource underutilisation and the transfer overheads associated with data movement across the system bus.

Chapter 6

Asymmetric Cryptography on DX10 Hardware

Asymmetric-key systems typically involve heavy use of big integer modular arithmetic. An example of this is RSA, where integer sizes range from 1024-bit for common application use to 4096-bit for conservative systems. Recall that RSA relies on the generation of two related exponents, e and d, such that $m^{e^d}(\text{mod } n) \equiv m(\text{mod } n)$. Using this relationship RSA encryption is performed by $c = m^e(\text{mod } n)$, where m is the input message and c is the output ciphertext. The relationship can then be used to decrypt the ciphertext by performing $m = c^d(\bmod\ n)$, thus retrieving the original message. In the context of 1024-bit RSA (RSA-1024) m and c are typically encoded as 1024-bit integers, with values less than n. The decrypt exponent d is generated so that it is also 1024 bits wide. The encrypt exponent is normally small, fitting within a single 32-bit integer. The exponent size is the major determining factor of the cost of a modular exponentiation, and thus the cost of an RSA operation. As such, of interest with regards to RSA acceleration are the costly decrypt and sign operations, which both use the large decrypt exponent. The RSA encrypt and verify operations use the smaller encrypt exponent.

We use the Nvidia 8800GTX G80 GPU with CUDA to investigate the possibility of accelerating modular exponentiation with a focus on RSA-1024 decryption. Montgomery exponentiation is employed in all implementations presented. Its constituent Montgomery reduction expedites modular reduction by removing the requirement for an expensive divide. In summary, given a positive modulus n and integers a and R, where $0 \leq a < nR$, $gcd(n, R) = 1$ and $R > n$, Montgomery reduction produces the output $w \equiv aR^{-1}(\text{mod } n)$, where $w < 2n$. The details of how to generate $aR^{-1}(\bmod\ n)$ and how to use it in the context of exponentiation are covered in Section 2.4.2.4. We present exponentiation approaches using

Montgomery exponentiation for integers in both radix and modular representation. Modular representation is used to build a residue number system (RNS), which as we will see imposes changes on the Montgomery technique. We investigate both types of number representation showing how GPU occupancy and inter thread communication play a central role to performance. Regarding RNS based implementations, we present an optimised base extension algorithm and also a new approach for efficient single-precision modular multiplication in an RSA RNS context. We show that using an RNS based exponentiation approach on the GPU can outperform radix based approaches at smaller payload sizes.

Section 6.1 notes a number of implementation factors that are common across all implementations presented in this chapter. In Section 6.2 we present implementations of modular exponentiation based on numbers in radix representation. Section 6.3 covers the implementation details concerning exponentiation using numbers in a residue number system. We also include in this section an exploration into the most efficient single precision modular multiplication techniques suitable for RNS on the GPU.

6.1 Implementation Commonalities

Key Support: All implementations presented in this chapter process a payload containing multiple messages. Each message is associated with a key, which among other values contains an exponent. The use of a CUDA compatible device imposes restrictions on the rate of key change supported. The reason for this is that the exponent largely determines the flow of control through the code. For example, in binary exponentiation, the exponent determines a conditional branch to execute an additional multiply per iteration. In Sliding Window exponentiation, the exponent determines whether to square or to execute a batch of squares plus a cumulated multiply. These conditional code paths, which depend on the exponent, cause thread divergence. When threads within a CUDA warp diverge on a streaming multiprocessor, all code paths are executed serially, thus a large performance overhead is incurred for threads that diverge for large portions of code.

Concerning the implementations presented later, depending on the degree of parallelism applied, threads can co-operate or act independently to execute an exponentiation. For implementations that use independent threads, the key should be the same for all threads within a CUDA warp. This ensures good performance, in that a single path is executed for all threads within the warp. For implementations that use co-operative threads, a synchronisation barrier is used to ensure consistent results when co-operating. All threads within a CUDA block that perform a synchronisation barrier must not be divergent at the point of synchronisation. Thus, all threads within a single CUDA block are required to execute the same path at points of synchronisation. It follows that for exponentiation that uses inter-thread communication, only one key can be used per CUDA block. This is another reason to focused on RSA decryption performance comparisons as it is common to require several decryptions per single key and thus suitable for GPU acceleration. We indicate the specific implications of the above restrictions within each implementation section later on.

RSA-CRT: RSA-1024 decryption consists of raising a 1024-bit number to the power of a 1024-bit exponent, so a naive approach would use 32 32-bit limbs to store the integers. However, we use a method based on the Chinese Remainder Theorem (CRT) to reduce the size of both the base and exponent as described by Quisquater and Couvreur [106]. Given the ciphertext c, decryption exponent d and modulus n we can write:

$$\begin{array}{ll} c_1 = |c|_p & c_2 = |c|_q \\ d_1 = |d|_{p-1} & d_2 = |d|_{q-1} \\ m_1 = |m|_p = |c_1^{d_1}|_p & \\ m_2 = |m|_q = |c_2^{d_2}|_q & \end{array}$$

where, $n = pq$, p and q are prime, m is the message, and $m = |c^d|_n$. We can see that we have a residue representation of m, $< m >_n = (|m|_p, |m|_q)$. Thus, given $|m|_p$ and $|m|_q$ we can use the constructive proof of CRT to generate $|m|_n$. The advantage of this is that instead of using $|c^d|_n$ to calculate the message, we can calculate $|c_1^{d_1}|_p$ and $|c_2^{d_2}|_q$ where c_1, c_2, d_1 and d_2 are half the bit width of c and d. This reduces the cost of generating the message from the ciphertext by up to four times [69, ch14]. It is common practice to store p, q, d_1, d_2 and other related data within the private key to facilitate the efficient employment of this approach. This CRT optimisation approach is used in both the radix and RNS based implementations of RSA presented here.

6.2 Radix Based Modular Exponentiation

This section presents large integer modular exponentiation implementations suitable for RSA based on the standard pencil-and-paper technique of multiple-precision multiplication [58]. The integers involved are represented in radix form, $x = (x_n, x_{n-1}...x_1, x_0)_b$, where b is the radix and x_i are the coefficients or limbs.

Specifically, on the GPU the radix b is 2^{32}. Following the CRT approach above, our implementations of 1024-bit RSA decryption represent the ciphertext as a pair of 16-limb 32-bit integers. Each 16-limb number represents the 1024-bit ciphertext modulo each of the primes, p and q, whose product comprise the public key's modulus. We have developed two radix based GPU implementations with varying degrees of parallelism incorporating Montgomery reduction and pencil-and-paper multiplication. One implementation acts in a mostly serial manner, while the other distributes parts of the exponentiation across multiple threads in an attempt to increase parallelism, and thus GPU occupancy.

The motivation for using the two approaches is to explore different performance trends, which depend on the number of decryption operations required for processing at any one time. We have focused on the basic pencil-and-paper multiplication approach rather than the Karatsuba [52] or Toom-Cook [58] approaches. These are applicable for a larger number of limbs than is required for a 1024-bit RSA implementation, i.e. 16 limbs. For example, within the GMP project [32], the threshold for using Karatsuba multiplication on a generic architecture is set to 32 limbs, the Toom-Cook approach's threshold is even higher. Even though we focus on RSA decrypt (and equivalently sign), and thus deal with ciphertext input and data such as p, q and other derived data that belong to the private key, the core exponentiation approaches hold for the encrypt and verify RSA functions or any exponentiation based algorithms.

With regards to Montgomery reduction, considering p and q are less than 2^{512}, we can select $R = 2^{512}$. This satisfies the condition that the modulus $n < R$ and also the requirement for division by R be an efficient process. Also, as p and q are prime, there are no common factors with R satisfying the relative prime requirement. Another restriction of Montgomery reduction is $0 \leq a < nR$, where a is the number we are reducing. From the above CRT approach we see that the powers of c_1 and c_2 require reduction in p and q respectively. Given that c_1 and c_2 are derived from $|c|_p$ and $|c|_q$, it is easy to see that all powers of c_1 and c_2 are less than pR and qR respectively.

6.2.1 Serial Approach

Each thread within this implementation performs a full exponentiation without any inter thread communication or cooperation. This is a standard optimised implementation of an exponentiation using the CRT approach, operating on two independent pairs of 16-limb numbers. The approach also uses the Sliding Window technique to reduce the number of Montgomery multiplies and squares required. As a single thread computes an exponentiation independently, a single

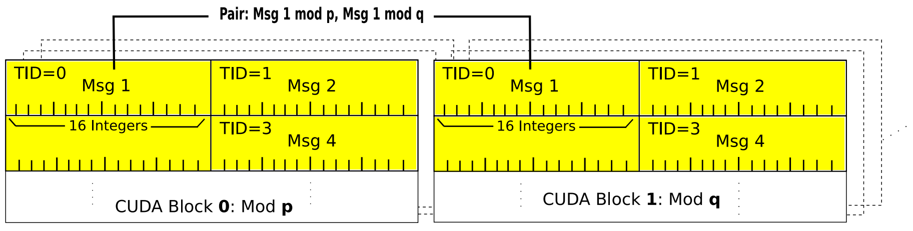


Figure 6.1: Serial Thread Model.

exponent should be used across groups of 32 threads. In terms of RSA, assuming peak performance, this implementation is restricted to scenarios that process at least 32 messages per key. As we are using the CRT based approach to split the input messages in two, we also use two different exponents for a single message, see d_1 and d_2 above. Thus a message must be split into different groups of 32 threads to avoid guaranteed thread divergence. We have adopted a simple strategy to avoid this divergence, whereby CUDA blocks are used in pairs. The first block handles all 16-limb numbers relating to the modulus p, while the second block handles all numbers relating to the modulus q, where $n = pq$ and n is the original modulus. The values for p and q can change within a CUDA block, though again, should follow the restrictions on change frequency. The threading model employed is illustrated in Figure 6.1.

The added support for integers, bitwise operations and increased memory flexibility such as scatter operations, in the 8800GTX, allows this implementation to execute largely in a single kernel call. The byte and bit manipulation operations required for the efficient implementation of Sliding Window are straightforward as opposed to the difficulties encountered using a DX9 GPU [75]. In the referenced DX9 implementation, Sliding Window was deemed impractical. Binary exponentiation was used instead where the main exponent traversing loop was implemented on the CPU, which resulted in a large number of kernel executions to perform a single exponentiation. The implementation we use on the G80 is more standard, consisting of the following macro level details:

- Input data is split into pairs according to the CRT approach.
- The base data is then converted to Montgomery representation.
- Exponentiation is executed using multiplies or squares according to the exponent following the Sliding Window technique.
- All modular reductions of multiplication or squaring outputs use the Montgomery method.

- The final output for each pair is Montgomery multiplied by 1 to undo the initial Montgomery representation.
- The output pair is recombined to the final 1024-bit datum using CRT.

As the steps are well known we do not go into further detail except to draw attention to the following optimisations that were applied within the implementation:

- All $N \times N$ limb multiplies used cumulative addition to reduce memory pressure [58].
- All $N \times N$ limb squaring are optimised to reduce the number of required multiplies [69].
- All $N \times N$ limb multiplies $\bmod R$ are truncated, again to remove redundant multiplies.
- The final two steps within Montgomery multiplication were combined into a single $N \times N$ multiply and accumulate.

These optimisations are listed here as they are relevant to the implementation in Section 6.2.2.

CUDA does not provide an add with carry operation. This causes an overhead for all additions to multiple precision numbers as these must be handled manually by checking for overflows. We expect that this performance issue will be alleviated in the near future as some Nvidia employees informally state that the hardware supports such an operation but it is not yet exposed by the PTX virtual assembly language. All operations within the radix serial approach (and parallel approach) use 32-bit integers. Multiplies can be performed using the standard 32-bit multiply as there is a `umulhi` intrinsic that supports the retrieval of the high 32-bit word from a 32-bit multiplication. The 32-bit multiply is quoted as executing in 16 cycles, so we require 16 cycles for the low word multiply and 16 cycles for the high word multiply. There is a faster `umul24` instruction, which multiples 24-bit integers in 4 cycles. This is an attractive option, however there is no equivalent `umul24hi` instruction and we require the high word results. 16-bit shorts can be multiplied in 4 cycles without overflow, however this gives no performance advantage as it requires four times more operations than 32-bit multiplies for the same resolution output. We assume that Nvidia transparently transpose the 32-bit multiplies into multiple operations, hence their execution in 16 cycles.

6.2.1.1 Memory Usage

As we have seen, the concept of a uniform, hierarchical read-write memory system such as a CPU's does not exist on the GPU, and performance cliffs can be encountered without careful memory planning. The following are the highlights of the various memory access techniques, which are used to optimise the implementation's performance. Note that the implementations in Section 6.2.2 and Section 6.3 use similar memory techniques where appropriate (RNS does not implement $N \times N$ limb multiplies for example), and as such are not repeated later unless significantly different.

Sliding Window Data: The Sliding Window technique requires that various powers of the input data are pre-calculated. This data is used during the exponentiation process to act as one of the N-limb inputs into an $N \times N$ multiple-precision multiplication. The number of powers that require pre-calculation is dependent on the window size, however the size is a multiple of the input data size and as such is generally impractical to store in on-chip memory. We investigated two options on how to handle the storage and retrieval of this pre-calculated data. In both cases each thread running on the GPU is responsible for the pre-calculation work relevant for the threads input data before exponentiation begins.

1. Each thread writes its pre-calculated data to global memory. The data is stored in a single array with a stride width equal to the number messages being processed in a single kernel call multiplied by the message size. Reads are subsequently made from this array directly from global memory. In this scenario only a single kernel call is required for the exponentiation process.

2. In the previous approach the data reads are not coalesced as each thread reads a single limb which is separated by 16 32-bit integers from the next message's pre-calculated power. Coalesced global reads require the data to be contiguous in memory with a stride of up to 16 bytes per thread. Non-coalesced reads generate separate memory transactions significantly reducing load/store throughput. To ameliorate this the Sliding Window pre-calculation data is first generated in an initialisation kernel, writing its results to global memory. A texture can then be bound to this memory and the subsequent exponentiation kernel can use the pre-calculation data via texture references. Note that texture access uses the texture cache, which is a local on chip cache, however textures cannot be written to directly hence the need for a separate initialisation kernel.

The first approach described above is suited to small amounts of data. The second approach is beneficial for larger amounts of data when the advantage of texture use outweighs the fixed overhead of the extra kernel call. Another approach could be to convolve the pre-calculated data to allow coalescing of memory, however this was not implemented. It is not known if the overhead of such convolution and subsequent data read offset calculations would be worth the coalescing improvement.

Exponent Data: Another adaptive memory approach concerns the exponent. As mentioned, the exponent must be the same across a warp number of threads, thus all threads within a warp, when reading the exponent, access the same memory location at any one time. Constant memory has by far the best performance under this scenario, see Table 5.2, however it is limited to 64 KB on the G80. As each exponent requires 32 integers worth of storage in an RSA 1024-bit context, we can use constant memory for up to 512 different keys. If the amount of exponents exceed this threshold then texture memory is used. In practice the threshold is slightly lower than 512 different keys as a small amount of constant memory is used for other purposes.

Multiplication Operands: In an effort to increase the $N \times N$ multiplication performance, we have allocated all of the on-chip fast shared memory for storing and retrieving the most frequently accessed N-limb number of the $N \times N$ operation. Each thread is allocated 16 integers worth of space within shared memory to store a 16-limb multiplicand. These shared memory storage techniques are used for all versions of $N \times N$ multiple precision multiplies (i.e. standard, square, truncated). The less frequently accessed multiplier is retrieved from textures where beneficial. For example the multiplications where the multiplier is p, q or Montgomery related values $|-p^{-1}|_R$, $|-q^{-1}|_R$, $|R^2|_p$ and $|R^2|_q$ (for use in generating the Montgomery representation of the input) are all read from textures as the memory locations accessed can repeat within and across threads. In the context of RSA decryption these variables are assumed to be pre-calculated as they are invariant with private key data. If these values are not stored within the key they are pre-calculated on the CPU before handoff to the GPU.

Message Data: The input and output message data is not exceptional in this implementation save that it cannot be coalesced due to the message stride of 16 32-bit integers within device memory. A convolution of multiple messages could be an option to offset the lack of coalescing though this has not been explored.

It should be noted that during our tests, the cost of reading and writing message data is relatively negligible, therefore coalescing would have little effect.

6.2.1.2 Results

The results for this radix based implementation is presented in Section 6.2.3 in conjunction with the parallel approach described below. Note that two parts of the exponentiation are not included in these implementations, the initial $|c|_p$ and $|c|_q$ and the final CRT to recombine. This is also the case for all implementations reported in this chapter. These steps contribute little to the overall exponentiation runtime and so the performance impact is expected to minor.

6.2.2 Parallel Approach

This approach uses the same macro structure as the algorithm used above, however it executes the various stages within the algorithm in parallel. Each thread is responsible for loading a single limb of the input data, with 16 threads combining to calculate the exponentiation. Each thread undergoes the same high level code flow, following the Sliding Window main loop, however here Montgomery multiplication stages are implemented in parallel. This approach relies heavily on inter thread communication, which has a performance overhead and also the side effect that only one exponent is supported per CUDA block. As the number of threads per block in this implementation is limited to 240 due to shared resource constraints, the number of 1024-bit RSA primitives per key is limited to a minimum of 16. This is a hard limit in that CUDA code has undefined behaviour if threads are divergent at points of synchronisation. The threading model uses the same separation of message pairs, for p and q, as in Figure 6.1, however, a single thread reads only a single integer.

The intensive $N \times N$ multiplies within Montgomery multiplication are parallelised by their separation into individual $1 \times N$ limb multiplications. Each thread is independently responsible for executing a single $1 \times N$ limb multiply. This is followed by a co-operative reduction across all threads to calculate the partial product additions. This parallel reduction carries with it an overhead where more and more threads are idle. Figure 6.2 shows the distribution of the $N \times N$ operation across the 16 threads and its subsequent additive reduction. It also shows the use of shared memory to store the entire operation's output and input at each stage. As previously noted, the number of threads per block is limited to 240, thus each RSA primitive can use up to 16 KB / (240/16) worth of shared memory. This allows for the entire $N \times N$ calculation to fit within shared memory.

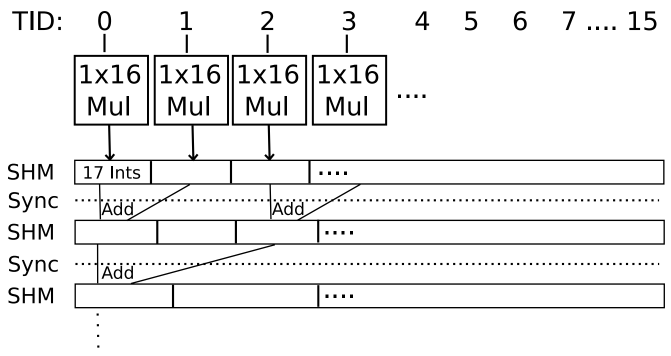


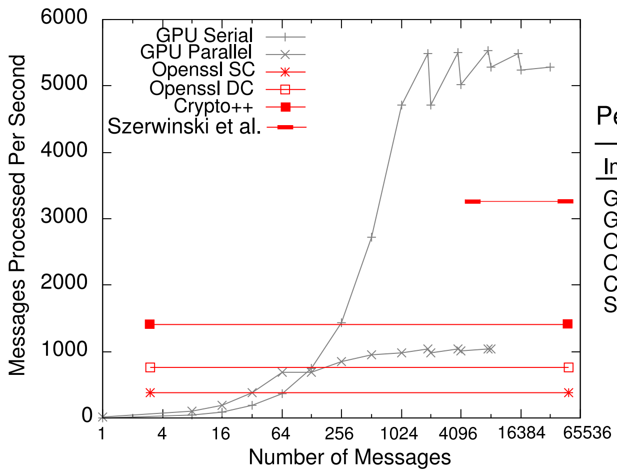
Figure 6.2: $N \times N$ limb multiplication in parallel on a CUDA device.

Also shown in the Figure 6.2 are the synchronisation points used to ensure all shared memory writes are committed before subsequent reads are performed. As this code is the most intensive part of the exponentiation, these synchronisation calls add a significant burden to the performance.

The optimisations applied to the different $N \times N$ multiplies, listed in the serial approach, are not possible in the parallel approach. The squaring optimisation, and also the modulo multiplication step, in general only execute half the limb multiplies that are required compared to a full $N \times N$ multiply. However, the longest limb within the $N \times N$ multiply dictates its overall execution time as all threads within a warp must execute in lock step. Thus, although one thread only executes a single multiply, it must wait until the largest $1 \times N$ multiply finishes. Also, as each thread executes its own $1 \times N$ multiply separately, the cumulative approach to addition must also be separated from the multiplication process. Another issue with the parallel approach is that there are parts of the algorithm which are duplicated across all threads, such as the main exponentiation loop itself. All duplicated instructions are a loss of potential performance. Also, the conditional subtract at the end of Montgomery reduction is executed in all threads as the borrowed bit must propagate making the operation serial. The results for this approach are presented below.

6.2.3 Radix Results

Figure 6.3 illustrates the performance of both the parallel and serial approaches presented above. All measurements presented represent the number of 1024-bit RSA decrypt primitives executed per second. The throughput rates are based on the repeated execution of the various scenarios taking the average performance. The exponents used were selected at random, though filtered to ensure the hamming weight was close to bit width / 2. The x-axis refers to the number



Peak Performance

Impl.	Msgs p/s	ms/op
GPU Serial	5536.75	0.18
GPU Parallel	1041.55	0.96
Openssl SC	379.7	2.63
Openssl DC	762.5	1.31
Crypto++	1369.8	0.73
Szerwinski et al.	3252.0	0.30

Figure 6.3: GPU Radix based Montgomery Exponentiation: 1024-bit RSA Decryption.

of messages processed as a single batch by a single kernel execution. The serial and parallel RSA results were generated by the execution of tests on System 3, see Appendix D. All results presented within this chapter were generated from running tests on the same system, unless otherwise indicated. It can be seen that the GPU implementations depend on an increasing number of messages per payload to approach their peak performance. This is due to having sufficient numbers of threads to occupy the GPU, thus hiding memory read/write latency and to a lesser extent to offset the fixed overheads associated with data transfer and kernel calls. Figure 6.3 shows the advantage of the parallel approach over the serial approach at lower primitives per kernel call due to an higher level of occupancy. However, the performance bottlenecks of synchronisation, lack of $N \times N$ multiply optimisations and operation duplication, limit the parallel approach. The peak performance was achieved using the serial approach at 5536.75 RSA 1024 primitives per second.

Also included in Figure 6.3, is the performance reported for the Crypto++ API for a 1024-bit RSA decrypt. Here we report the throughput rate for the same processor as in Chapter 5, the Intel Core 2 Duo E6300 1.86 GHz running on both cores. The Crypto++ API employs the CRT approach described previously to split the message in two. Also included are the performance measurements for Openssl's [100] speed test for 1024-bit RSA decryption running in both single (SC) and dual core (DC) modes on an AMD Athlon 64 X2 Dual Core 3800+. The OpenSSL version used was 0.9.8g. As can be seen at peak performance, the serial approach on the GPU is over 4 times the speed of the fastest CPU implementation. We can see that the serial approach becomes competitive with the fastest CPU implementation at batches of 256 primitives.

Similar work was independently carried out by Szerwinski and Güneysu [121] on an Nvidia GeForce 8800GTS GPU where throughput rates were stated for 1024-bit modular exponentiation. Their peak performance used a radix implementation, though they did not employ the CRT approach. To make the figures somewhat comparable we apply a correction factor of $\times 4$ to their throughput rates and correspondingly divide their minimum reported latency by four. This compensates for the computational reduction given by using the CRT approach [69, p613]. We observe that they report results for both 1024 and 2048-bit modular exponentiation and the performance scaling is in line with the correction factor used here. Thus, the maximum throughput reported is equivalent to 3252 primitives per second for RSA-1024 CRT, though with a minimum latency of approximately 1.7 seconds regardless of number of primitives executed concurrently. As a point of comparison we achieve a throughput rate of 4707 primitives per second at a latency of 218ms or 1024 primitives per batch of processing, i.e. a latency reduction of over 7 times. It is unclear as to the reason for our latency improvement though it could be attributed to a decreased reliance on global memory. We have only plotted the graph of Szerwinski and Güneysu's results from the lowest number of message per batch corresponding to the reported minimum latency, i.e. the time taken to process smaller batches of messages remains the same. It is also worth noting that Nvidia unhelpfully use the name 8800GTS to denote two different GPUs, one faster ($\sim$120%) and one slower ($\sim$70%) than the 8800GTX. It is unclear as to which was used in the work by Szerwinski and Güneysu as they quote a number of cores in their device that match neither release (though is closer to the slower version of the GPU).

We observe that the parallel approach at no point is faster than both the serial GPU approach and the CPU implementations. All figures include the transfer of data to and from the GPU. We have not included the figures for execution without transfer, however the overhead is small, in the order of 1-10 primitives per second. This is a marked change from symmetric-key acceleration on GPUs where the data transfer is a significant overhead due to its lower level of arithmetic intensity. Asymmetric-key cryptography is more suited to an offload accelerator card due to its inherent high arithmetic intensity. A final observation is the zigzag pattern created with the serial approach. This is due to the coarse grained parallelism of the approach, whereby the performance is heavily dependent on the number of primitives per batch matching the number of available processors in the GPU. This is a consistent feature of the serial approach though we have only highlighted the effect, via purposeful selection of batch sizes, at the higher range of throughputs for visual clarity.

6.3 RNS Based Modular Exponentiation

The motivation for using a residue number system (RNS) to perform exponentiation is an attempt to improve on the parallel approach in Section 6.2. The goal is to reduce the number of messages per payload required for the GPU to become competitive with the CPU. We recall briefly that a number x within an RNS, is denoted as $< x >_a$, where $< x >_a = (|x|_{a_1}, |x|_{a_2} ... |x|_{a_n})$ and a is a set, $\{a_1, a_2 ... a_n\}$, called the RNS base whose members are co-prime. $< x >_a$ can be converted into radix form, $|x|_A$, using the Chinese Remainder Theorem (CRT), where $A = \prod_{i=1}^{n} a_i$ and is called the RNS's dynamic range. Numbers in RNS can perform multiplication, addition and subtraction as: $< x >_a$ op $< y >_a = (||x|_{a_1}$ op $|y|_{a_1}|_{a_1}, ... ||x|_{a_n}$ op $|y|_{a_n}|_{a_n})$, where op is $+$, $-$ or $\times$. $< x >_a$ op $< y >_a$ is equivalent to $|x$ op $y|_A$ in radix form. As each RNS operation is executed independently, it is suited to parallel processing. The division operation can be problematic as we will see.

To illustrate RNS and its usefulness we use an example. Take a 16-limb integer x and a modulus n whose factors are single limb co-primes, $\{n_1, n_2, ... n_{16}\}$. To generate a power of x mod n in radix representation we execute a 16×16 limb multiply followed by a Montgomery reduction. Using RNS, we first convert x to modular representation $< x >_n$. Then we execute 16 independent single-precision modular multiplications. The result is then converted back to radix representation using CRT. The conversion into and out of RNS is expensive, however if the number of operations performed while in RNS form is high, the savings can be large. Although RSA supports the use of moduli using multiple factors, in practice RSA is normally implemented with two factors, p and q. The use of RNS with the base $\{p, q\}$ is essentially the basis of the Quisquater and Couvreur RSA optimisation presented previously. As such, to use RNS to further reduce the cost of modular exponentiation in the context of RSA another approach is required.

6.3.1 Montgomery in RNS

As Montgomery multiplication consists largely of multiplication and addition, there is a temptation to perform this using an RNS. However, two parts of the algorithm cause problems within RNS. Referring to Algorithm 2.3, step 1 is a multiply mod R. Considering R is only restricted to be relatively prime with

Input: $< x >_{a\cup b}, < y >_{a\cup b}$, (where x, $y < 2N$)		
Output: $< w >_{a\cup b}$ (where $w \equiv xyB^{-1} (\mathrm{mod}\ N), w < 2N$)		
	Base a Operation	Base b Operation
1:	$< s >_a \leftarrow < x >_a . < y >_a$	$< s >_b \leftarrow < x >_b . < y >_b$
2a:	—	$< t >_b \leftarrow < s >_b . < -N^{-1} >_b$
2b:	$< t >_{a\cup b} \Longleftarrow < t >_b$	
3:	$< u >_a \leftarrow < t >_a . < N >_a$	—
4:	$< v >_a \leftarrow < s >_a + < u >_a$	—
5a:	$< w >_a \leftarrow < v >_a . < B^{-1} >_a$	—
5b:	$< w >_a \Longrightarrow < w >_{a\cup b}$	

Table 6.1: Kawamura et al. [53]: Montgomery multiplication in an RNS.

and larger than the modulus, there is scope to select an R that also consists of relatively prime factors suitable for use as an RNS base. Given a sufficient dynamic range for this base, all multiplies and additions could be successfully performed in the RNS. In the same algorithm, step 3 requires a division by R. A divide in modular arithmetic is performed by multiplication by the inverse. However, the inverse of R, R^{-1}, does not exist for mod R. Thus, these two operations in step 1 and step 3 cannot be performed within the one RNS base. Montgomery reduction in RNS approaches presented in papers by Posch and Posch [105] and Kawamura et al. [53], show a way around this issue. They use 2 bases, one for the execution of the mod R operation, and the other for execution of the divide. Thus one of the bases in effect acts as Montgomery's R, the other acts as a facilitator to represent R^{-1} and perform a division by inverse multiplication. Table 6.1 contains an outline of Montgomery multiplication in an RNS as presented by Kawamura et al. [53], which we largely base our implementations upon. The RNS bases are denoted as a and b, where $B = \prod_{i=1}^{n} b_i$ is equivalent to R in the standard algorithm.

The calculations in Table 6.1 are carried out in the two bases $a \cup b$, in essence acting as a single large base. Step 2a executes a multiplication in base b alone, achieving the mod B operation. We require conversion of the result into base a to continue the calculations. This conversion from one base into another is called base extension. Step 5a performs the division by B within base a alone, and is followed by a second base extension. The most computationally expensive part of this algorithm are the two base extensions. A naive base extension can perform a full conversion of a number in an RNS into its fixed radix form using CRT, and then convert into another RNS. However, the full conversion via CRT is inefficient and is impractical to form part of the inner loop of an exponentiation. A number of base extension techniques have been proposed. The earliest approach proposed, by Szabo and Tanaka [120], was based on conversion using a mixed radix representation as the intermediate format. Although faster than the traditional CRT approach and parallelisable, it requires a high degree of inter-thread communication. Improvements have been proposed based on a form of CRT as shown in Equation 6.1, which aim to calculate k, the reduction factor, in an efficient manner [53, 104, 119]. The base extension used for our RNS implementations is described below.

6.3.2 Exponentiation using Kawamura on the GPU

Our Montgomery RNS implementation is based on the relatively efficient approach presented by Kawamura et al. [53], listed in Table 6.1. Its base extension algorithm relies on the following representation of CRT.

$$x = \sum_{i=1}^{n} (||x|_{m_i} |M_i^{-1}|_{m_i}|_{m_i}) M_i - kM \tag{6.1}$$

where m is a set of n single limb moduli, $M = \prod_{i=1}^{n} m_i$, $M_i = M/m_i$ and $x < M$. If we compare this equation to the original CRT equation we can see that it is just removing the mod M and replacing it with an explicit subtraction. This allows the conversion process to involve only single precision operations. To see this we look at an example where we convert $< x >_m$ into a new RNS with moduli m'. To simplify the reading of Equation 6.1 further, let $E_i = ||x|_{m_i} |M_i^{-1}|_{m_i}|_{m_i}$, thus we now can write

$$x = \sum_{i=1}^{n} E_i M_i - kM. \tag{6.2}$$

To further simplify, consider just the extension of $< x >_m$ into a single modulus m'_1, i.e. we are calculating $|x|_{m'_1}$. This is performed by the following.

$$|x|_{m'_1} = |\sum_{i=1}^{n} (|E_i M_i|_{m'_1}) - k|M|_{m'_1}|_{m'_1} \tag{6.3}$$

Here E_i, for each i, can be calculated independently and thus in parallel. Its calculation uses $|M_i^{-1}|_{m'_i}$, which consists of invariant moduli. As such, it can be pre-calculated for all moduli in m and m', and used via a lookup table. $|M_i|_{m'_1}$ and $|M|_{m'_1}$ are also based on invariant moduli and can be pre-calculated for all moduli in m and m' and used via lookup tables. To calculate the base extension for multiple moduli, m', each residue, $|x|_{m'_i}$, is assigned to a separate thread and calculated independently. However, on the right hand side of Equation 6.3, k is unknown. We can rewrite Equation 6.2 by dividing across by M to give

$k = \sum_{i=1}^{n} E_i/m_i - x/M$. Considering $x/M < 1$ and k is a whole number we can say $k = \lfloor \sum_{i=1}^{n} E_i/m_i \rfloor$.

Kawamura et al. calculate this divide, $\lfloor \sum_{i=1}^{n} E_i/m_i \rfloor$, using an approximation based on the observation that m_i can be chosen close to a power of 2. m_i is substituted for the nearest higher power of two, allowing a divide by bit shift. Also E_i is approximated, using a restricted number of its most significant bits (the emphasis for Kawamura's approach is on VLSI design), via a function called `trunc()`. k is calculated in a cumulative manner whereby the above approximated divide is carried out each iteration of the summation in Equation 6.3. The result of the divide is added to a cumulation counter, `counter`. Each iteration, k is set to $\lfloor$`counter`$\rfloor$ and then is subtracted from `counter`. Thus k is $\{0, 1\}$ for each iteration, and as such selectively subtracts M. In Algorithm 6.1 we present a version of Kawamura's base extension algorithm which does not use the `trunc()` approximation of E_i as there is no need for it in a 32-bit processor. Also, as the divide by the nearest higher power is always set to 2^{32}, we can let the counter accumulate E_i without modification. A check for 32-bit wrap on the addition takes care of the application of k for each iteration. Note that the algorithm only calculates a base extension for a single modulus for clarity. This must be executed for each modulus in the new base, m'.

Algorithm 6.1 Kawamura base extension modified for a 32-bit processor.

```
Require: $< x >_m$, $m$, $m'_1$, $\alpha$
  $E_i = ||x|_{m_i}|M_i^{-1}|_{m_i}|_{m_i} (\forall i)$
  for $j = 1$ to $n$ do
    $\alpha 0 = \alpha$
    $\alpha += E_j$ /* note $\alpha$ wraps at $2^{32}$ on GPU */
    if $\alpha < \alpha 0$ then
      $r = |r + (|-M|_{m'_1})|_{m'_1}$
    $r = |r + E_j|M_j|_{m'_1}|_{m'_1}$
  return $|x|_{m'_1} = r$
```

In Algorithm 6.1, α is used to compensate for the approximations introduced in the calculation of k. It must be higher than the maximum error caused by the approximations, however lower than 2^N [53], where N is the word bit length of the GPU. As we have removed the error due to approximation of E_i the only determinant of the size of the error is the distance between the moduli used and their next power of 2. This puts a restriction on the number of moduli that can be used with this technique. In effect, this base extension algorithm can be used in the context of 1024-bit RSA with 32 and 16-bit moduli, while 12-bit moduli requires the use of a different method. We use 32-bit moduli as we will see in

Section 6.3.3, this size provides the best performance.

For RSA-1024 using the CRT approach, we require two sets of 17 32-bit moduli for use as the two RNS bases a and b from Table 6.1. This is due to the moduli p and q being 512 bits, and a requirement that the dynamic range of the RNS bases is sufficient to represent numbers mod p and q without overflow. Kawamura et al. also specify further constraints (see paper for more details [53]) on the dynamic range, though two sets of 17 32-bit moduli where the moduli are the closest 34 primes to 2^{32} is sufficient. Note we interleave the moduli in the two different sets to give a reasonable distribution of the dynamic ranges.

Groups of 17 consecutive threads within a CUDA block execute co-operatively to calculate a modular exponentiation. Given that there are two RNS bases consisting of 17 moduli, each message is split into 2 groups of 2×17 residues. That is 17 residues for each base, and 2 groups for c_1 and c_2 related to p and q. Each thread reads in two residues, $|x|_{a_i}$ and $|x|_{b_i}$. Thus, a single thread executes both the left and right sides of the Montgomery RNS algorithm, see Table 6.1, for a pair of residues. This ensures each thread is continuously busy. We make the general observation that RNS exponentiation requires the use of bases that have dynamic ranges larger than some multiple of the input data units. Therefore for data units with a size based on a power of 2, the number of moduli in the RNS base will be slightly higher than a power of 2. Using an RNS based algorithm, as above, on highly parallel devices such as the GPU can lead to an awkward fit as many such devices are designed to suit threads used in groups of powers of 2.

As the residues are in groups of 17, we employ a padding scheme for the input message data whereby the start address used by the first thread of a CUDA block is aligned to a 128-bit boundary. We also pad the number of threads per block to match this padding of input data, which allows a simple address mapping scheme while allowing for fully coalesced reads. The CUDA thread allocation scheme for ensuring even distribution across available SMs and also correct padding is show in Table 6.2, where RNS_SIZE is the number of moduli per base, MAX_THREADS_PER_BLOCK is a predefined constant dependant on shared resource pressure of the kernel and BLOCK_GROUP is the number of SMs in the GPU.

Revisiting Algorithm 6.1, we see that for each new modulus we are extending to, the calculation of E_i is required for all m_i members of m. Also recall that a single thread is responsible for performing the extension to a single new modulus. Thus all E_is will be calculated for all new moduli, leading to much redundancy. To reduce the overhead of calculating E_i, as the number of moduli in the new base match the number in the originating base, we can distribute its calculation one

$total\_threads = noMsgs * RNS\_SIZE * 2$
$max\_msgs\_per\_block = \lfloor MAX\_THREADS\_PER\_BLOCK / RNS\_SIZE \rfloor$
$blocks\_required = \lceil noMsgs * 2 / max\_msgs\_per\_block \rceil$
$blocks\_required = \lceil blocks\_required \rceil^{BLOCK\_GROUP}$
$threads\_per\_block = \lceil total\_threads / blocks\_required \rceil$
$threads\_per\_block = \lceil \lceil threads\_per\_block \rceil^{RNS\_SIZE} \rceil^{WARP\_SIZE}$

Table 6.2: CUDA thread allocation scheme for RNS based modular exponentiation.

E_i per thread. First, each thread is responsible for calculating a single E_i, after which a synchronisation barrier is used to ensure all new E_i values can be safely used by all threads. As discussed in Section 6.2, this synchronisation barrier, along with the general performance issues with thread divergence, dictates that only a single exponent can be used for each CUDA block of threads. Thus for RSA-1024 using RNS on the G80, a single exponent must be used a maximum of once for every 15 primitives (256 threads per block / 17 threads per primitive). With regards to the shared memory use, we use two different locations for storing the values of E_i. The storage locations are alternated for each subsequent base extension. This permits a single synchronisation point to be used, rather than two: one before and one after the generation of E_i, which is necessary when only one location is used for storing the values of E_i.

We also use shared memory to accelerate the most intensive table lookup corresponding to $|M_j|_{m'_i}$ in the base extension. Specifically regarding the bases a and b used in the Montgomery multiplication in an RNS algorithm, Table 6.1, at the start of each kernel call all threads within a block co-operate in loading into shared memory the entirety of the two arrays, $|A_j|_{b_i}$ and $|B_j|_{a_i}$ via texture lookups. This load into shared memory must be performed during kernel execution as unfortunately shared memory has no way of being initialised via the CUDA runtime API before kernel execution. The exponent is treated in the same manner as in the radix based pencil-and-paper approach. The base related variables, a_i, b_i, $|-A|_{b_i}$, $|-B|_{a_i}$, $|A_i^{-1}|_{a_i}$, $|B_i^{-1}|_{b_i}$, $|-p^{-1}|_{b_i}$, $|-q^{-1}|_{b_i}$, $|p|_{a_i}$ and $|q|_{a_i}$ are all stored in textures and accessed at the start of each thread. As each thread only requires access to a single word for each of these variables and they are constant across the thread's execution of the entire exponentiation they are stored in registers. These variables are based on the moduli a and b and as such never change and can be pre-calculated once. Those based on p and q are pre-calculated once per key and either retrieved from the key itself or from a cache.

Each thread executes an exponentiation of the input data for its own pair of

RNS moduli using Sliding Window. The Sliding Window data is generated in much the same manner as in the radix approaches except that each thread is responsible for generating the Sliding Window powers for its own RNS moduli only. The threads work independently apart from the collaborative generation of E_i for base extension. Each thread is also responsible for generating a Montgomery representation of its inputs and at the end, the undoing of the Montgomery representation. The results for 17 threads make up the 34 residues that are used to generate the final output either mod p or q using CRT. This step is performed on the CPU. Future work involves the investigation of an efficient mechanism for performing a full CRT on the GPU.

6.3.3 Single Precision Modular Multiplication on the GPU

The most executed primitive operation within Montgomery RNS is single-precision modular multiplication. On the Nvidia CUDA hardware series the integer operations support operands up to a maximum size of 32 bits. Integer multiplies are reported to take 16 cycles, where divides are not quoted in cycles but rather a recommendation to avoid if possible [93]. In light of this, we present an investigation into 6 different techniques for achieving single-precision modular multiplication suitable for RNS based exponentiation implementations.

1. 32-bit Simple Long Division: Given two 32-bit unsigned integers we use the native multiply operation and the `__umulhi(x,y)` CUDA intrinsic to generate the low and high 32-bit parts of the product. Because we cannot divide a 64-bit number directly we treat the dividend as a 4 16-bit limb number and the divisor as a 2 16-bit limb divisor. Then we use standard multiple-precision division to generate the remainder [58].

2. 32-bit Division by Invariant Integers using Multiplication: We make the observation that the divisors within an RNS Montgomery implementation, i.e. the base's moduli, are static. Also, as we select the moduli, they can be chosen to be close to the word size of the GPU. Thus we can assume two things. Firstly, all moduli can be treated as invariant divisors, and secondly, all invariant divisors can be treated as normalised (i.e. they have their most significant bit set). These two observations allow us to use an optimised variant of Granlund and Montgomery's [38] approach for division by invariants using multiplication. The basic concept used to calculate n/d is to find a sufficiently accurate approximation of $1/d$ in the form $m/2^x$. Thus the division can be performed by the multiplication of $n*m$ and cheap byte shifts for division. The requirement for the

```
$n = x \times y, n1 = \text{hiword}(n), n0 = \text{loword}(n)$
$ns = n0 >> (N - 1)$
$if(ns > 0)\ n0 = n0 + d$
$t = \text{hiword}((m \times (n1 + ns)) + n0)$
$q1 = n1 + t$
$dr = (n - (d << N)) + ((2^N - 1 - q1) \times d)$
$r = \text{loword}(dr) + (d \wedge \text{hiword}(dr))$
```

Table 6.3: Granlund and Montgomery's division by invariants optimised for GPU and RNS.

divisors to be invariant is due to the overhead in calculating m outweighing the savings using multiplication if m is calculated for each division. We pre-calculate m for each of the base moduli used and transfer them to the GPU for use via texture lookups.

The algorithm in Table 6.3 removes all normalisation related calculations from the original algorithm. It also rearranges some of the calculations to suit the efficient predication available on the G80. Inputs: N is the word bit length on the GPU; single word multiplier and multiplicand x and y; m is a pre-calculated value dependent on d alone; d is the divisor. Output: r, the remainder. Some of these operations require 2 word precision and thus require extra instructions on the GPU. `hiword()` indicates the most significant word of a two word integer, where `loword()` indicates the least significant word. For a thorough explanation of the concepts involved, refer to Granlund and Montgomery [38].

3. 32-bit Reduction by Residue Multiplication: In this approach we design an algorithm that executes modular multiplication without division in the context of select residue number systems, which is faster than the previous approach. Recall that the moduli comprising the RNS bases used can be selected close to the GPU's maximum single word value. There are 93 32-bit primes between 2^{32} and $2^{32} - 2048$ of which we use 34 to act as the moduli of our two RNS bases. Thus, in the context of the residue number systems used, we can state for all moduli, d, the following holds:

$$|2^{32}|_d < 2^{11}. \tag{6.4}$$

Note, we use the convention that given a 2-limb number a, we write a_1 to signify the high 32-bit limb and a_0 to signify the low 32-bit limb. Given 32-bit single limb inputs x, y and d, we wish to calculate $|z|_d$, where $z = xy$. Let $r = |2^{32}|_d$. Note r can be calculated efficiently using a single subtract as d is normalised.

Using r, we can rewrite $|z|_d$ as:

$$||z_1 r|_d + |z_0|_d|_d. \tag{6.5}$$

Given the upper bound on r from 6.4, we can state that $z_1 r < 2^{43}$. If we further rewrite $|z_1 r|_d$ as:

$$||(z_1 r)_1 r|_d + |(z_1 r)_0|_d|_d \tag{6.6}$$

we can state that $(z_1 r)_1 r < 2^{22}$. Combining 6.5 and 6.6 we can write:

$$|z|_d = |||(z_1 r)_1 r|_d + |(z_1 r)_0|_d|_d + |z_0|_d|_d.$$

Considering that $(z_1 r)_1 r < 2^{22}$, $(z_1 r)_0 < 2^{32}$, $z_0 < 2^{32}$ and that $\forall d : d - 2^{22} > (2^{32} - d) \times 2$, we can state:

$$z' = (z_1 r)_1 r + (z_1 r)_0 + z_0$$

, where $z' \equiv z (\text{mod } d)$ and $z' < 3d$. Thus we can perform the modular multiplication $|xy|_d$ via multiplies and conditional subtraction. We list the steps in algorithmic form in Algorithm 6.2. The functions `hiword()` and `loword()` are the same as previously described. This approach benefits from being able to use CUDA's `umul24` limited precision fast multiply instruction in the calculation of $(z_1 r)_1 r$.

Algorithm 6.2 32-bit Reduction by Residue Multiplication.

```
Require: Single limb integers $x$, $y$, $d$, $r$, where $d > 2^{32} - 2048$, $r = 2^{32} - d$.
  $z = x \times y$
  $z_0 = \text{loword}(z)$
  $z_1 = \text{hiword}(z)$
  $z_1 r = z_1 \times r$
  $(z_1 r)_0 = \text{loword}(z_1 r)$
  $(z_1 r)_1 = \text{hiword}(z_1 r)$
  $(z_1 r)_1 r = (z_1 r)_1 \times r$
  $z' = (z_1 r)_1 r + (z_1 r)_0 + z_0$
  if $z' > d$ then
    $z' = z' - d$
  if $z' > d$ then
    $z' = z' - d$
  return $z'$
```

4. 32-bit Native Reduction using CRT: Recall that the RNS bases consist of moduli that are relatively prime. We can also further stipulate that each modulus is the product of two co-prime factors. Using a modulus with two co-prime factors p and q, we can represent the modular multiplication input values, x and y, as $|x|_p$, $|x|_q$, $|y|_p$, $|y|_q$. Thus we have a mini RNS representation and as such can multiply these independently. We use CRT to recombine to give the final product. As p and q can be 16-bit, we are able to use the GPU's native integer modulus operator while maintaining 32-bit operands for our modular multiplication. This approach is described in the Moss et al.'s paper [75].

5. 16-bit Native Reduction: We can use 16-bit integers as the basic operand size of our modular multiplication, both input and output. We can then simply use the GPU's native 32-bit integer multiply and modulus operators without any concern of overflow. However, we need to maintain the original dynamic range of the RNS bases when using 32-bit moduli. We can achieve this by doubling the number of moduli used in each base. Note that as a larger number of relatively prime moduli is required to represent the bases, and the moduli are capped at 2^{32}, the contribution of each subsequent modulus to the dynamic range diminishes. However, there is sufficient surplus dynamic range when using the original 17×2 32-bit integers, such that the simple doubling of the number of 16-bit moduli meets the dynamic range requirements.

6. 12-bit Native Reduction: This is the same concept as the 16-bit native approach above, though using 12-bit moduli and as such 12-bit operand sizes. We can then use the much faster floating point multiply and modulus operators without overflow concerns. Again we need to maintain the dynamic range by approximately tripling the original 32-bit moduli. Also as mentioned, there is an issue where the Kawamura approximations require the base moduli to be within a certain range of the next power of 2. This is not discussed further here, though note that a full 12-bit implementation would require the use of a different base extension method than the one described previously.

6.3.3.1 Results

We tested the above approaches by processing the same amount of data, 2^{32} bytes, executing modular multiplication operations, reading and accumulating from and to shared memory. The results can be seen in Table 6.4. We can see that the 12-bit and 16-bit approaches show the best performance, however considering the figures report the number of 12-bit and 16-bit modular multipli-

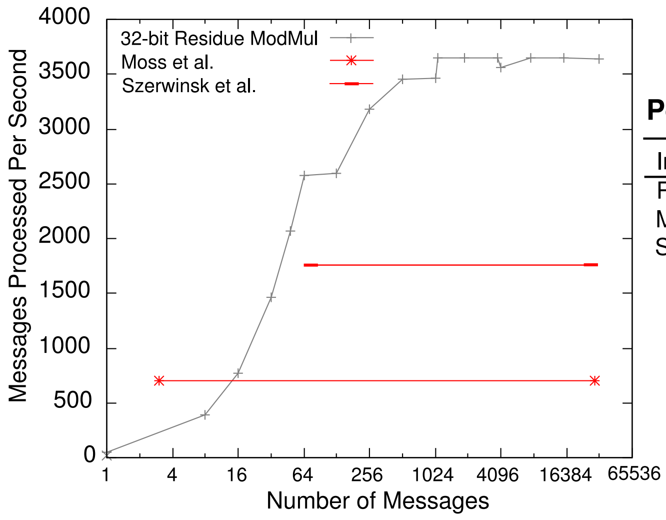
	Modular Multiplication Approach	Modular multiplications per second
1.	32-bit LongDiv	$2.89 * 10^9$
2.	32-bit Inverse Mul	$3.63 * 10^9$
3.	32-bit Residue Mul	$4.64 * 10^9$
4.	32-bit Native+CRT	$1.12 * 10^9$
5.	16-bit Native	$9.42 * 10^9$
6.	12-bit Native	$23.97 * 10^9$

Table 6.4: GPU Modular Multiplication throughput using a variety of techniques.

cations respectively, we must compensate for the loss of resolution to compare the results. Firstly, the data unit sizes used in the modular multiplications determine the sizes of the moduli used in the residue number systems. Secondly, as we have seen, the base extension executes in $O(n)$ time across n processors, where n is the number of moduli in the RNS base. Concretely, assuming a fixed number of processors, throughput reduces at a rate of n^2 as n increases. Lastly, from experimentation, the base extension step in Montgomery RNS is the most intensive part of our implementations consuming over 80% of execution time. Given 12-bit moduli we require approximately three times the number of moduli and two times the moduli for 16-bit moduli compared to 32-bit moduli. The 12-bit reported figures are compensated by an approximated factor of $(9 \times 0.8) = 7.2$, and the 16-bit figures are compensated by a factor of $(4 \times 0.8) = 3.2$. When we reduce the 12-bit and 16-bit results according to these factors we can see that the most efficient approach for use in Montgomery RNS is *32-bit Reduction by Residue Multiplication*.

6.3.4 RNS Results

Figure 6.4 shows the throughput of our RSA-1024 RNS implementation using CRT, Sliding Window and 32-bit modular reduction using the Reduction by Residue Multiplication as described previously. The tests were executed multiple times with selection of the exponent as in the radix tests. The peak throughput achieved is 3653 message decryptions per second. Comparing the peak performance of the Crypto++ CPU implementations listed in Figure 6.3 we can see that the peak performance for our RNS implementation has over 2.5 times higher throughput. Also included in Figure 6.4 is the peak RNS performance from independent work by Szerwinski and Güneysu [121] executing on an 8800GTS GPU at 1759 primitives per second for RSA-1024 CRT. Like in the radix implementation we have applied a correction factor of $\times 4$ to their performance rating of 1024-bit modular exponentiation using RNS without CRT. The figures reported



Peak Performance

Impl.	Msgs p/s	ms/op
Residue Mul	3653.54	0.27
Moss et al.	704.22	1.42
Szerwinsk et al.	1756.0	0.57

Figure 6.4: GPU RNS based Montgomery Exponentiation: 1024-bit RSA Decryption.

include this correction factor. As in the radix results, we have only plotted the graph for [121] from the point of the reported minimum latency. We have also included as a historical reference, a previous RNS implementation on an Nvidia 7800GTX by Moss et al. [75] to show the improvements possible due to the advances in GPU hardware and software libraries. They report a performance of 175.5 true 1024-bit exponentiation operations per second, which we have again multiplied by 4 and use as an estimate of the runtime for RSA-1024 CRT decryption. As we can see from the figure there is pressure to increase the number of concurrent messages sent to the GPU to achieve performance. However, as we will see in the next section this pressure is more relaxed when compared to the radix approach presented earlier in the chapter.

6.4 Radix vs RNS on the GPU

We use Figure 6.5 to illustrate the effectiveness of our RNS implementation at accelerating RSA in comparison to our radix based implementations. As can be seen the RNS implementation gives superior throughput with much smaller number of messages per kernel call. The point at which the serial radix approach becomes faster than the CPU is at ∼256 messages, where the RNS approach has better performance at 32 messages per kernel. The greater performance at smaller message numbers is due to a higher GPU occupancy for the RNS approach over the serial radix approach. The RNS approach also does not suffer from the extreme levels of synchronisation during a Montgomery multiplication

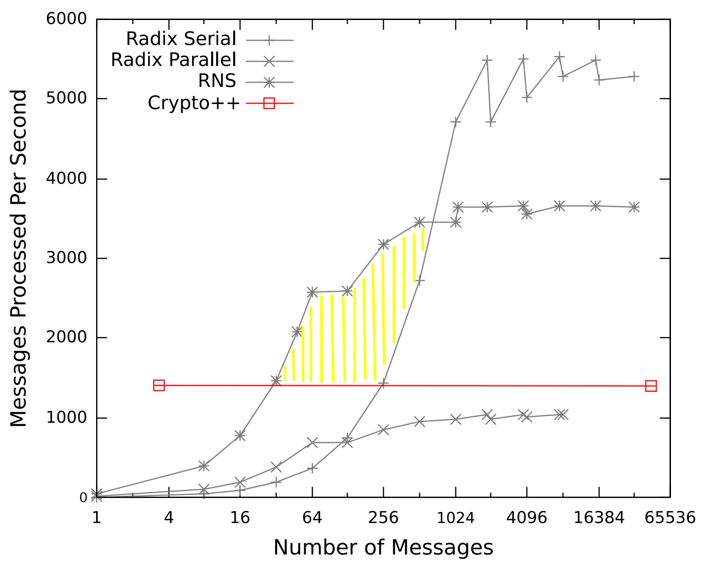


Figure 6.5: RNS vs Radix: 1024-bit RSA Decryption.

as the parallel radix approach. Using RNS can greatly improve the GPU's ability to provide feasible acceleration for RSA decryption, or any public-key cryptographic scheme where the modulus and exponent change with a low frequency. It is clear from Figure 6.5 that an adaptive approach to exponentiation would provide the best overall performance, switching from an RNS based implementation at low message requirements, to a radix based approach at high message requirements.

6.5 Conclusions

In this chapter we have presented implementations of modular exponentiation suitable for asymmetric-key cryptography. We have focused on 1024-bit RSA decryption running on an Nvidia 8800GTX and demonstrated a peak throughput of 0.18 ms/op giving a 4 times improvement over a comparable CPU implementation. We have shown that an adaptive approach to modular exponentiation on the GPU provides the best performance across a range of usage scenarios. A radix based serial implementation of Montgomery exponentiation gives the best performance in the context of a high number of parallel messages, while an RNS based Montgomery exponentiation gives better performance with fewer messages. We show that an optimised RNS approach gives better performance than a CPU implementation at 32 messages per kernel call and that the pencil-and-paper approach proves better than the RNS approach at 256 messages.

Also covered in the chapter is the applicability of the GPU to general asymmetric-key cryptography, where the observation is made that peak performance is only achievable in the context of substantial key reuse. In the case of 1024-bit RSA using RNS, peak performance requires the key to change at a maximum rate of once per 15 messages, and once per 32 messages when using a serial pencil-and-paper approach. Thus the GPU can be effectively used in an RSA decryption and signature capacity where it is common for a server to use a limited number of keys.

RNS based approaches are highly dependent on efficient support of single precision modular multiplication, which the chapter illustrates is non trivial on the GPU. In this context we have explored a variety of techniques for achieving efficient modular multiplication and show that a novel approach suited to RNS and the GPU gives the best performance. The GPU could offer improved performance with RNS based approaches if future architectures provide efficient native modular operations. Also the performance of the radix based approach suffers greatly due to the 16 cycles required to execute a 32-bit integers multiplication. If this is improved in the future to operate at a similar rate to floating point processing, we could see a $4\times$ improvement in the reported rates.

Chapter 7

GPU Accelerated Cryptography as an OS Service

Symmetric-key algorithms such as AES, DES, ARIA; symmetric-key modes of operations; and asymmetric-key algorithms such as RSA, DSA and those based on ECC have recently been explored in the context of GPU acceleration [27, 40, 43, 44, 65, 75, 121, 129, 130]. It has been demonstrated that the GPU can act as an effective accelerator of symmetric-key algorithms using sufficiently large buffers and of asymmetric-key algorithms using a sufficient number of concurrent primitives. Despite the existence of these new approaches, there remains no way for OS kernel services or userspace applications to make use of these implementations in a practical manner. The use of these implementations require interaction with GPU specific interfaces such as the CUDA API, which is inconvenient for application developers and unavailable to kernel services. With the increasing number of GPU accelerated cryptographic algorithms, there is a need to provide an efficient and standardised operating system wide interface to these implementations. To overcome this shortcoming, this chapter investigates the integration of GPU accelerated cryptographic algorithms with an established service virtualisation layer within the Linux kernel. The OpenBSD Cryptographic Framework (OCF) provides the basis for such a virtualisation layer.

The original OCF was developed for OpenBSD and has since been ported to FreeBSD [61], NetBSD and Linux [95]. It was created to provide uniform access to cryptographic accelerator functionality by hiding hardware specific details behind a standardised API. It provides access to this functionality for kernelspace services as well as normal userspace applications and APIs. For our investigation we use the Linux port of the OCF and the 2.6.26 Linux kernel. Although we do not directly use the native linux-crypto (Crypto API) project [63], which has in-built support for some crypto-cards, we note that the OCF acts as a wrapper for this library. We did not use linux-crypto for this work due to its current lack of support for asymmetric algorithms and the fledgling status of its userspace interface, however the main contributions in this chapter are also relevant to this project.

The main contributions of this chapter are: the effective integration of the GPU within the OCF model; the observation that the GPU interface is userspace only and the mechanisms introduced to allow it to be part of a kernel service; the introduction of a new memory management system within the OCF to allow efficient handling of memory transfers between multiple address spaces; and also an implementation of a general purpose multi-request batching scheme for asymmetric-key requests with regard to the GPU. The only previous attempt to provide a form of uniform access to GPU crypto acceleration involved AES via an OpenSSL engine by Rosenberg et al. [112]. This implementation was applicable to userspace applications only and reported a 0 to 3% improvement over the CPU.

The motivation for this work is to provide a standard method of access to the latest GPU crypto acceleration work to all components within an operating system, with minimal loss of performance. This will allow application, kernel and driver developers to transparently include the GPU as part of their cryptographic solutions. As previously observed the GPU has a requirement of high work loads to achieve its peak performance. By using a centralised framework, which is used for all system-wide cryptographic needs, we increase the likelihood of high occupancy on the GPU and thus its potential to act as an effective crypto-accelerator. OS constructs and functions mentioned throughout this chapter are briefly described in Appendix C.

7.1 OCF Background

Figure 7.1 shows a high level view of the OCF framework. The core component of the framework, the main "Crypto" layer, provides two APIs - the producer API for use by crypto-card device drivers and the consumer API for use by other kernel subsystems. An ioctl interface, which uses the /dev/crypto device file, provides a mechanism through which normal userspace applications can issue cryptographic requests. This interface is provided by the "Cryptodev" layer and uses the consumer API to pass on userspace requests to the Crypto layer. Device drivers can register their support for various cryptographic algorithms with the Crypto layer. Cryptographic requests received directly by the Crypto layer or sent via the Cryptodev layer are matched with capable devices and issued to the

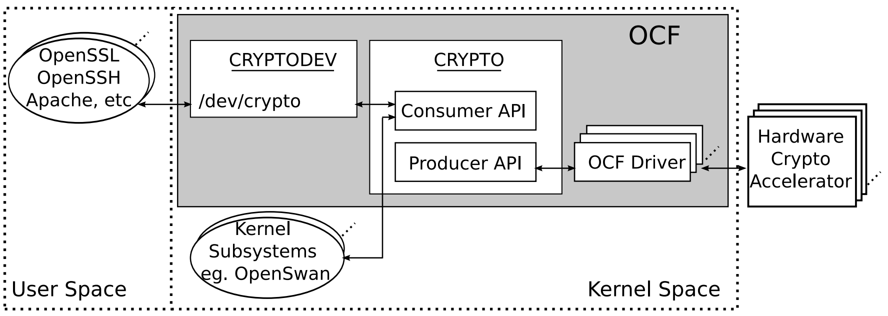


Figure 7.1: Original OCF Architecture.

corresponding device driver. The device driver ID is recorded within the request, which is returned to the requesting application or kernel component along with the results of the processed request. Further requests can be issued to the same device within the OCF by maintaining the driver ID within the request or if left unset the OCF will again select a suitable device dynamically.

7.2 Integration of GPU and OCF

7.2.1 Overview

The OCF provides a standardised method for the integration of any cryptographic accelerator device driver using its producer API. This API allows a device driver to register itself and its supported algorithms with the OCF, making it a target for processing cryptographic requests. The device driver is responsible for registering four callback functions with the OCF, which are used for the set up and tear down of symmetric algorithm sessions and also for the processing of symmetric and asymmetric requests. We have created a GPU cryptographic driver that fulfils the producer API requirements. The driver currently supports AES and modular exponentiation with CRT, suitable for RSA-1024. Supporting these two algorithms allows an analysis of the main issues arising from GPU integration with the OCF for both symmetric and asymmetric functions.

The algorithms supported by the GPU driver are the peak performing CUDA implementations presented in Chapters 5 and 6. The CUDA interface is provided via a userspace runtime library and as such requires its usage to be from userspace processes. Unfortunately Nvidia do not provide a driver that allows the direct control of their cards from within the kernel. This restriction forces all interactions with their cards to originate from userspace processes, making the provision of CUDA services from within the kernel a challenge. To overcome this restriction, we have split our GPU driver into two parts, a kernelspace driver

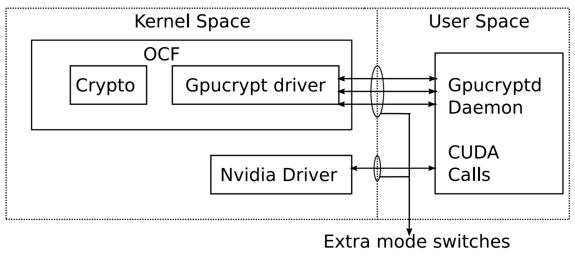


Figure 7.2: OCF and GPU: High Level View - Different Address Space Problem.

and a userspace daemon.

Figure 7.2 shows a high level overview of the GPU driver integration into the OCF. It illustrates the separation of the GPU driver into the kernelspace part, *Gpucrypt*; and the userspace part, *Gpucryptd*. Gpucryptd follows the normal daemon convention, and as such runs as a high privilege background OS process. Gpucryptd is responsible for receiving cryptographic requests from Gpucrypt and processing them using Nvidia's userspace runtime API. A major disadvantage of this separation is the use of extra address spaces within the processing pipeline making data transfer more complex. Extra address spaces can result in a critical bottleneck in performance when processing requests unless memory is carefully managed. We explore this issue in full within the next topic. Another disadvantage of the driver separation is the introduction of two extra OS mode switch points within the processing pipeline. This becomes more of an overhead when the number of cryptographic requests increase, particularly for small request buffer sizes. Unfortunately there is no way to avoid these mode switches, however since the GPU only suits cryptographic acceleration with large workloads and high arithmetic intensity we will see that this overhead has limited effect.

7.2.2 Memory Management

When using devices that handle high volumes of data transfer it is common practice to ask the driver to allocate the memory used in these transfers. This has the advantage that the driver knows what type of memory (contiguous/non-contiguous, zone location) suits the corresponding device for DMA transfers. It is also common practice that allocated memory is shared between the driver and the calling process, either by driver allocation (`mmap()` kernel function) or by mapping userspace pages (`get_user_pages()` kernel function). If memory is not shared then userspace processes must undergo a copy of memory between user address space and kernel address space using the `copy_from_user()` and

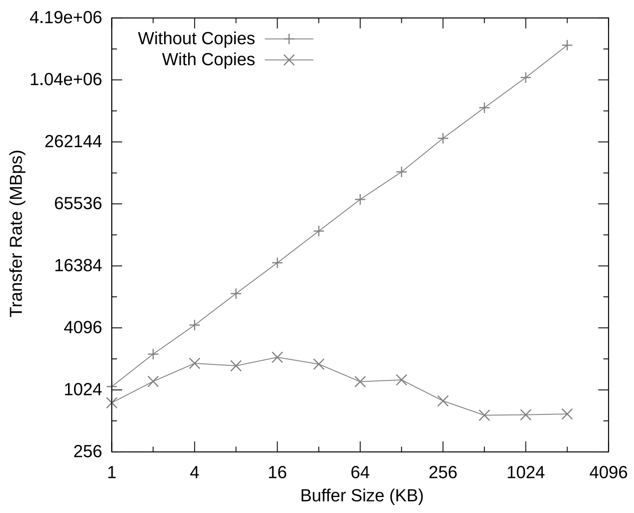


Figure 7.3: Illustration of the Cryptodev Layer Memory Management Overhead.

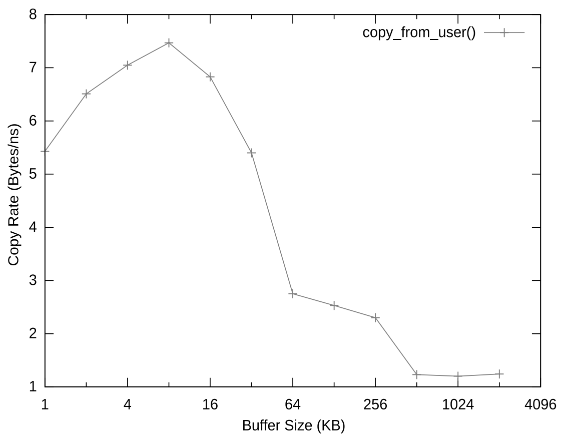


Figure 7.4: Performance of the `copy_from_user()` Function.

`copy_to_user()` Linux kernel functions. Using an abstraction framework like the OCF, or the linux-crypto project, removes the direct line of communication required for standard requests for driver memory allocation, either by userspace processes or kernelspace subsystems.

Integration of the GPU with such a framework emphasises this deficiency due to two factors. First, the GPU requires large volumes of data for symmetric algorithms to reach its performance potential as seen in Chapter 5. The larger the volumes of data, the worse the memory copy overhead. The OCF Cryptodev layer implements a copy from and to userspace policy for data transfer. Figure 7.3 shows the Cryptodev layer's performance with and without these copies as the buffer sizes increase. To explain the drop in performance of the Cryptodev layer, we note that it uses the `copy_from_user()` and `copy_to_user()` kernel functions to transfer memory between userspace and kernelspace. We test the performance of `copy_from_user()`, shown in Figure 7.4, and can see that the memory copy loses efficiency as the buffer sizes increase. Second, one cannot give memory to the Nvidia driver and request it to be used for DMA acceleration, the memory must be requested from the driver. Thus, even using a direct I/O approach (as in the new linux-crypto userspace API), where userspace pages are mapped in by the kernel on request, we must still perform a memory copy into and out of GPU DMA memory. Thus for any device that has DMA memory restrictions, it can be beneficial to have a mechanism for allowing the framework's drivers to manage their own memory.

We have added a new memory management system to the OCF that allows a consumer component (userspace application or kernelspace component) to directly use memory that is managed by the OCF drivers. This system allows the GPU driver to reduce the number of memory copies required during request processing to zero. Each memory allocation is recorded centrally by the OCF as a new memory mapping, which stores the driver's address (`map_ptr`) and the consumer component's address (`app_ptr`) of the allocated memory. `map_ptr` is the address the driver uses to refer to the allocated memory, which would normally be a kernelspace address, however in the case of the GPU it will belong to the Gpucryptd daemon address space. The `app_ptr` is the address the consumer component uses to refer to the memory and will belong to a userspace processes' address space if the OCF was called via the Cryptodev interface, or otherwise belong to the kernel address space.

Figure 7.5 illustrates our implementation of the memory mappings for all consumer components. We create a separate mapping space, indexed by the `current` thread's thread group ID, to store all mappings for each consumer component (the ID is zero in the case of kernel consumer components). This ensures allocated memory can only be accessed by the process that requested the allocation. Within each space the mappings are grouped according to the device that allocated the memory. If memory allocation fails due lack of device support for the new memory management system, where possible we still wish to avoid the memory copies performed by the Cryptodev layer. An allocation request can be tied specifically by the consumer to a particular underlying device. In this case, if the device fails to allocate memory, then this failure is reported to the consumer. Otherwise, when a memory allocation fails, the Cryptodev layer allocates its own memory for sharing with the userspace consumer. As such, the Cryptodev layer maintains its own memory mappings. The following is a detailed account of how mappings are created, removed and used within the new

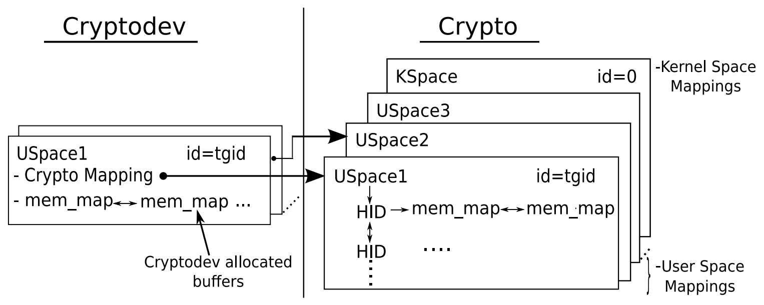


Figure 7.5: Crypto and Cryptodev Layers: Memory Mapping Internal Structure.

memory management system. The new memory management API is listed in Appendix A.1.

7.2.2.1 Memory Map Creation

Userspace, allocation request: A userspace process makes a request for memory via a new ioctl added to the Cryptodev layer. This allocation request can suggest a specific device to carry out the allocation or allow the OCF to choose a device and report the used device back to the consumer. The standard `mmap()` system call is not used as it cannot support device specification in this way. The OCF relays the allocation request to the device driver, which performs the allocation and returns the underlying memory pages and `map_ptr` to the OCF. The OCF takes these pages and manually calls the internal kernel version of `mmap()`, which generates a new virtual memory area (VMA) for the userspace process. We use the returned pages from the device driver to map to this VMA, and return the VMA start address to the userspace process. In the case where no device can be found to fulfil the allocation request and a device identifier is not explicitly stated in the request by the user, the Cryptodev layer allocates its own kernel memory, and maps this to the calling process' VMA. The VMA start address is the pointer used by the userspace process and is the aforementioned `app_ptr`. The OCF uses the `app_ptr` and `map_ptr` to add a memory map to the appropriate “USpace” as shown in Figure 7.5. We illustrate an example of a userspace process allocation request to the GPU in Figure 7.6. We can see six cases of mode switch between user and kernel mode; four of them involve the added trip out of and back into the kernel due to userspace CUDA calls. Here we have highlighted mode switch 1 and 2. The above explains the mapping of kernel memory to userspace memory to eliminate a potential memory copy at mode switch 1. We discuss the GPU driver part of the new memory management system, which eliminates the copy at mode switch 2 in Section 7.2.3. The

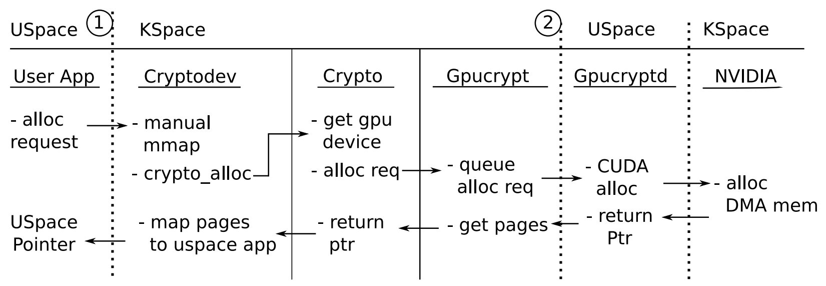


Figure 7.6: New OCF Memory Allocation: Cryptodev to GPU.

remaining mode switch is handled internally by the CUDA runtime.

Userspace, fork: On fork, a child process will have shared access to OCF allocated memory. Allocated memory returned by the OCF is set as always shared. Supporting private memory by implementing a copy-on-write procedure or executing a new allocation for each fork were deemed unnecessary considering that a child process can allocate its own memory. We share the memory between parent and child by overriding the VMA's `vm_open()` kernel function, which is called by the kernel when a new memory reference is created, such as on `fork()`. When this function is called, we create a new Cryptodev or Crypto memory mapping, within a new mapping space representing the process' new address space. Note that light-weight processes automatically share allocated memory as the VMAs of the processes are shared.

Kernelspace, allocation request: A process in kernel mode, or a kernel component, requests memory directly from the Crypto layer using the new `crypto_alloc()` function. This processes the request as above, selecting an appropriate device. However, instead of dealing with a userspace process' VMA, the OCF returns a kernelspace pointer, which references the new memory. The memory returned is not necessarily within the kernel address space, in the case of the GPU it belongs to the Gpucryptd address space. Thus, the OCF selectively performs a `vmap()` to map the memory into the kernel virtual address space if necessary. The kernelspace pointer returned is the `app_ptr` in this context, and is used to create a new memory map in the Crypto layer within the kernel space mappings, see "KSpace" in Figure 7.5.

7.2.2.2 Memory Map Removal

Userspace: A userspace process can free OCF allocated memory by executing the `munmap()` Unix command or by terminating. On such an event the kernel

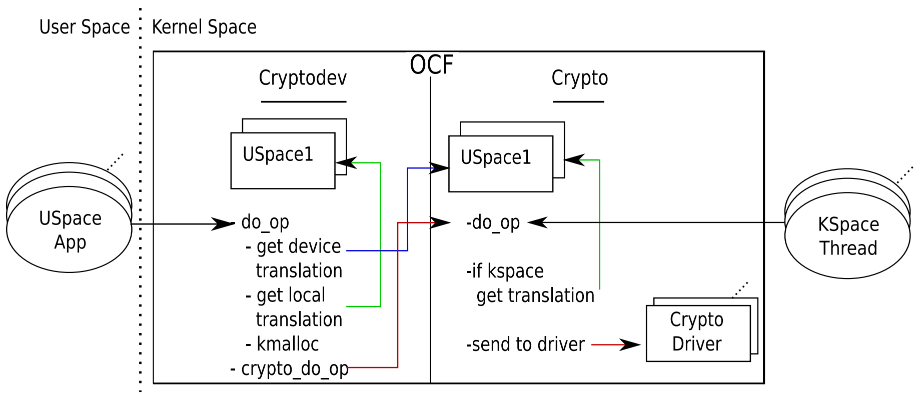


Figure 7.7: Crypto and Cryptodev Layers: Memory Mapping Translation Process.

calls the `vm_close()` kernel function for the allocated memory's VMA, which we have overwritten. If this process is the last to hold an open reference to the shared memory we issue a free command to the Crypto layer for device allocated memory or to the Cryptodev layer for Cryptodev allocated memory. After the memory is freed, the memory map is removed from the appropriate Crypto or Cryptodev memory map USpace.

Kernelspace: Kernelspace components issue a free directly to the Crypto layer via the new `crypto_free()` function. Unlike the Cryptodev free process above, the OCF must ensure that there exists a valid mapping for the kernelspace pointer for the specified device. If found, a free command is issued to the device driver and on return the memory mapping is removed from the kernel mapping space, KSpace.

7.2.2.3 Memory Map Translation

Userspace: All buffer pointers within cryptographic requests received via the Cryptodev layer interface are processed for potential address translation. First, the Crytpo layer is called to find a mapping that matches the userspace pointer (`app_ptr`) and the device specified within the request. This is done by finding the mapping space corresponding to the calling process using its thread group ID. Once the space is found we scan for a matching map within the corresponding device list of memory mappings. If a match is found, the device address (`map_ptr`) recovered replaces the userspace pointer within the cryptographic request and can be used directly by the device without any copies taking place. If no match is found, we repeat a similar process for any Cryptodev allocated memory. If still no match is found we default to the original OCF behaviour of using `kmalloc()` and the `copy_from/to_user()` kernel functions to copy the userspace buffers into the new kernelspace buffers. Figure 7.7 gives a brief illustration of the interactions between the Crypto and Cryptodev layer during this translation. The original OCF behaviour should possibly be upgraded to use the `get_user_pages()` call, as in the linux-crypto project, to default to using direct I/O when no mapping is found thereby eliminating the use of the expensive `copy_from/to_user()` kernel functions. The cryptographic request is always tagged internally to ensure the device driver can detect if a request pointer is a native device address or a normal kernelspace address.

Kernelspace: Cryptographic requests received via the Crypto kernel interface undergo a similar procedure as above, except only the kernel mapping space is searched and only the Crypto memory mappings are searched. Considering that, as kernel mode processes are trusted, we provide the ability for these processes to translate the allocated memory before the request is sent to the Crypto layer. This allows the kernel processes to use native device driver pointers in their requests with tagging thus avoiding translation overhead.

Existing consumers: Care has been taken to ensure legacy consumers can continue to use the OCF with minimal impact to performance. Regarding cryptographic requests made via the Cryptodev interface, the new memory management system will only impose a small translation overhead for userspace applications not using OCF allocated memory. This overhead consists of the failure to retrieve a USpace for requests, which is very fast. For processes in kernel mode using the Crypto layer directly, the overhead depends on how many mappings are being used by other kernel components, as this determines the size of the translation search space.

7.2.3 GPU Driver and Daemon

Here we discuss in detail the GPU driver component within the OCF and in particular its separation into a kernel driver and a userspace daemon. As previously mentioned this separation is necessary due to the requirement of using a userspace API to communicate with Nvidia's GPUs. If Nvidia provided kernel level access to its device drivers this separation could be avoided, as would the extra kernel to user mode switches. Figure 7.8 illustrates both the Gpucrypt driver and Gpucryptd daemon components and an overview of how they cooperate to fulfil the requests delivered by the Crypto layer. This is further discussed

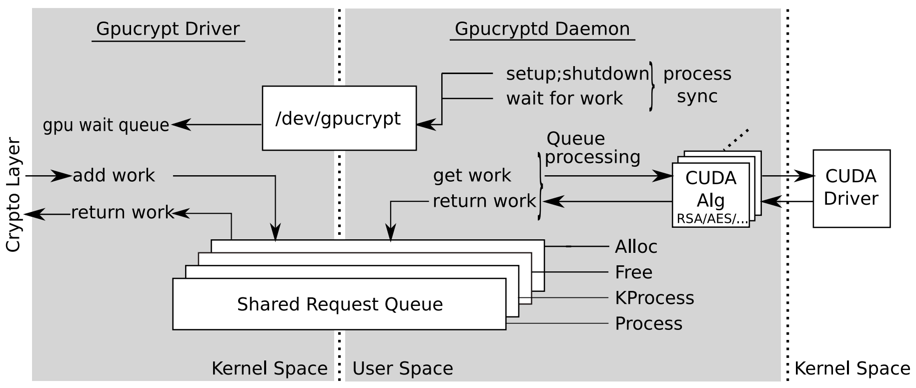


Figure 7.8: GPU Driver Gpucrypt and GPU Daemon Gpucryptd.

below.

/dev/gpucrypt: for the purpose of providing a communications channel between the two components we have created a new OS character device file called /dev/gpucrypt. On OCF start-up, the Gpucrypt driver module is initialised and connects itself with the /dev/gpucrypt device file. The Gpucryptd component can subsequently open this device file and communicate with Gpucrypt via ioctls. These ioctls are used for initial handshake of Gpucryptd with Gpucrypt when the daemon sets up shared buffers for use in request processing. It also uses the interface to send a "ready for work" and "shutdown" signals. When the Gpucrypt driver receives these signals it correspondingly registers and unregisters with the OCF Crypto layer. The /dev/gpucrypt device is most intensively used to co-ordinate the processing of cryptographic and memory requests. When no work is available on the request queues, the Gpucryptd daemon calls the driver to passively wait for more work by putting itself to sleep. Thus, whenever work is received from the Crypto layer the driver calls wake on the daemon process' wait queue. Whenever work is finished and requires returning to the driver, the daemon uses an ioctl to signal that the work is finished, to remove the work from the queue and to call the Crypto layer for request return. The ioctls are listed and detailed in Appendix A.2.

Processing Requests: the Gpucrypt driver implements four shared request queues, one for each type of OCF request supported: symmetric, asymmetric, alloc and free requests. The advantage of using separate queues for each request type is that it simplifies queue management. It allows a straightforward grouping of cryptographic requests for batching purposes rather than dealing with a single queue of mixed requests. These queues are allocated by the Gpucryptd daemon at start-up and memory mapped into the Gpucrypt driver, thus allowing efficient transmission of request data. When the Gpucrypt driver receives a requests from the OCF, it copies all the necessary instructions into the relevant queue. All pointers used in the requests at this stage have undergone address translation, and the addresses used within the queue are from the Gpucryptd daemon address space. Thus, the Gpucryptd daemon does not have to worry about address mapping, it can treat all pointers as native in a normal manner.

Cryptographic Requests: the Gpucrypt driver supports multi-threaded and asynchronous cryptographic requests, helping to increase the concurrency of requests on the process queues. Calls from the Crypto layer to process a symmetric or asymmetric request are returned immediately after the Gpucrypt driver has queued the request and signalled for the Gpucryptd process to awaken if necessary and process the request. All manipulations of the queues are thread safe. The only time a cryptographic request blocks is when the corresponding queue is full. The results of the processed requests are returned asynchronously when the Gpucryptd daemon issues an ioctl to instruct the driver that it is finished. This in turn calls the Crypto layer to inform it that the request is finished.

Memory Requests: as with standard memory allocation and free operations, we have implemented these as blocking requests. Apart from blocking the consumer thread, memory requests do not block any other request from being processed within the OCF. Figure 7.6, which served as an example of an OCF alloc request, can now be discussed in the context of Gpucrypt and Gpucryptd. To service an allocation request, the Gpucrypt driver first puts the allocation details on the shared alloc request queue. The Gpucryptd daemon processes this by executing the `cudaMallocHost()` function call, which allocates pinned DMA accelerated memory. The returned address is placed back on the shared request queue, which is then used by the Gpucrypt driver to access the underlying pages. On initialisation of the Gpucryptd daemon, it registers with the Gpucrypt driver its internal task kernel pointer. This is used to retrieve access to the daemon's underlying virtual memory areas and pages. A note should be made that the virtual memory area used to reference the CUDA allocated pages is flagged with `VM_IO`. Device driver programmers commonly use this flag to prevent memory from being included in core dumps, however it also has the effect of treating the memory area as backed by non system RAM. For I/O mapped memory it is necessary to restrict access to the underlying pages as they don't exist in RAM, however in our experience, CUDA only returns RAM backed memory. We must temporarily disable this flag in order to retrieve the underlying pages, though we take the precaution of acquiring the Gpucryptd's memory map semaphore during this period. Our experience is that this technique has successfully returned the underlying pages to the Crypto layer in all of our tests.

Request Order: maintaining separate shared queues has advantages as stated above, however it has a disadvantage of not automatically preserving the original request ordering between the differing types of requests. This can cause faults when memory requests are run out of order with respect to cryptographic requests. If we solve this problem using a single request queue, then batching is less effective as the queue requires processing in order. This can lead to memory requests unnecessarily splitting groups of cryptographic requests. The solution adopted for this problem was the use of read-write semaphores within the OCF. We have used a read-write semaphore for each mapping space (i.e. one per consumer process) within the OCF and found the solution to give minimal overhead. Each cryptographic request is responsible for acquiring a read-write semaphore for *reading* if a memory translation has occurred and releasing the semaphore on request completion. Each memory request must acquire a read-write semaphore for *writing*, which ensures the memory request is the only request for the consumer process within the OCF pipeline. This ensures that any translations that were valid at the start of the processing of request, remain so until the end. The use of read-write semaphores as opposed to normal semaphores allows the most common type of request, i.e. cryptographic requests, that share a mapping space to exist concurrently within the OCF pipeline. Also if no memory translation is used, e.g. legacy consumer processes, then no semaphores are used as in the original OCF.

Driver Removal: a driver can be removed at any time, and thus we must deal with the case of allocated memory when such an event occurs. Requests can be migrated to another device by the OCF and thus memory allocated for one device can be sent to another device. The Cryptodev layer sees this event as a failed translation and defaults to copying the memory from the userspace process, thus the requests will continue to proceed, however at a slower pace. To avoid this slow down the consumer process must monitor the requests for a change in device used and if a change occurs the OCF allocated memory should be freed and allocated again by the new device. If no OCF allocated memory is used then no action is required. Note that even though the device may have freed the memory, its pages are kept alive due to the consumer process' reference. Requests sent directly to the Crypto layer will also fail the translation stage and the memory will be treated as a standard kernelspace memory pointer. Again the kernel consumer thread must monitor requests for changes in the device used and reallocate memory when this occurs.

7.2.4 Security

We look at each of the changes made to the OCF in terms of their security implications. The new memory allocation functionality requires that all memory returned is automatically zeroed to protect from leaking information. The use of memory translation for userspace requests bypasses the need for the `copy_to/from_user()` kernel functions. These kernel functions perform important validation, ensuring that the addresses are part of the calling process' address space. We ensure that this validation is maintained by only searching for translations within a mapping space which is indexed by the thread group ID. This combined with the fact that the mappings within the space only contain userspace pointers, which are generated by the kernel on behalf of the process, ensure that any match found during translation are valid userspace addresses for that process. During translation we also check the size of the buffers specified within the cryptographic requests, to ensure no buffer overflow will occur. Regarding the Gpucrypt driver, it must be ensured that the /dev/gpucrypt file is accessible by the root user only. If this is not the case, then any userspace program may connect to the Gpucrypt driver and receive OCF cryptographic requests.

7.3 Concurrent Request Processing

7.3.1 Symmetric Request Batching

Symmetric-key request batching has already been covered via a runtime component in Chapter 5 and is not discussed further in the context of the OCF. However, it is worth noting that the OCF API model is based on separate storage of data and keys, and so does not immediately suit the payload data model. This leads to the requirement of a new API, or a modification of the current one to include such a concept and is left for future work. Also of note, and not covered previously, is that apart from the overheads shown in Chapter 5, e.g. the direction of threads via a generic mapping layer and mode of operation type, there is a consideration of OS process switching. As we will see in Section 7.4.2, the OS process switching overhead in a multithreaded environment can become a limiting factor in request queue filling, and thus a factor in GPU throughput.

7.3.2 Asymmetric Request Batching

The OCF is more suitable to the batching of asymmetric-key requests than symmetric-key requests. This is due to the lack of sensitivity of throughput rates to data movement, i.e. asymmetric-key requests have a high arithmetic intensity. Thus, we can use the OCF's existing API for asymmetric requests and flexibly pre-process the input data to suit the GPU. We present a mapping layer in which threads can process multiple different requests in a general manner, similar to the symmetric mapping layer in Section 5.3.2. We also include a selection of input data pre-processing methods, which the OCF performs, that significantly affect the final GPU throughput.

Single Request: Currently the OCF does not support a method of executing more than one asymmetric cryptographic operation within a single request. The framework does provides the ability to chain multiple requests with a link list. This gives the ability to send in a single call multiple requests to the Crypto or Cryptodev layer. We have made a small change to the OCF API to allow the request's input buffers to contain multiple instances of its input vectors. For example, in relation to modular exponentiation, this permits a request to contain multiple bases for each exponent/modulus pair (analogous to multiple messages per key). Although only giving a slight performance improvement, it simplifies the process for clients to send multiple requests with a single key. We are reluctant to make any modifications to existing API structures for reasons of compatibility with existing applications, thus this change is suggestive and can be safely omitted if compatibility is required.

General Purpose Request Batching: We have based our asymmetric-key implementation on the serial radix algorithm for CRT modular exponentiation suitable for RSA-1024 presented within Chapter 6. This involves spawning a new CUDA thread to handle the exponentiation of each base. As there can be multiple bases per request, and one thread per base, we require a mechanism that allows each thread to dynamically discover its request data. We must also take into consideration that the base, modulus and exponent for each operation is split into two, due to the CRT technique. Recall that this involves using the prime factors p and q of the modulus n to generate smaller pairs of bases, mod p and q, and smaller pairs of exponents, mod $p-1$ and $q-1$.

Figure 7.9 illustrates the mechanism used to direct the threads to their corresponding data. As in Chapter 5, we direct all even numbered CUDA blocks to p related data and all odd CUDA blocks to q related data. The base data is config-

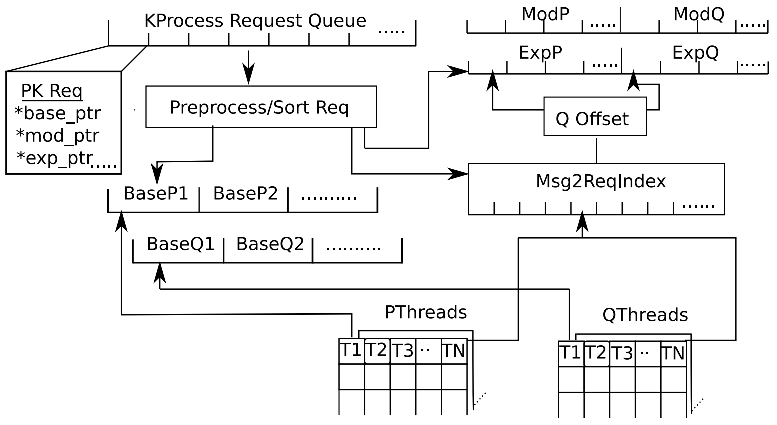


Figure 7.9: Mechanism for Processing Multiple Distinct Asymmetric-Key Requests.

ured in a manner so that each CUDA thread can simply scale their global thread ID to find the offset of their base data. During the preprocessing stage (discussed next), we generate a message to request index, labelled Msg2ReqIndex. This index is used to translate the message number, i.e. the base number within the full batch of requests, to the OCF request number. The request number is used to generate an offset into the modulus, exponent and related per request data. In the figure we can see that the modulus, exponent and related data is split into two groups. This allows a simple conditional addition of a single offset to the request offset to direct a thread to the p or q related data depending on whether the CUDA block is odd or even.

Request Preprocessing: The Gpucryptd daemon can have access to multiple asymmetric requests at any one time. The GPU's processing performance of these requests can depend greatly on the order in which they appear within the GPU buffers. Concerning an efficient modular exponentiation implementation, the code path taken is largely dependent on the exponent. When the GPU executes modular exponentiations with different exponents within the same CUDA warp, we experience thread divergence and a cost is incurred. This is because of having to execute the separate code paths serially rather than concurrently. The more varied the exponents within a warp, the higher the warp cost, and thus the higher the CUDA block cost, up to a limit. We have measured the different warp costs for a GPU execution of a modular exponentiation in various divergent scenarios and the cost ranges between 1 and 2.5, where 1 is equal to the minimum run time of a non-divergent modular exponentiation. Ideally we would be able to efficiently take any array of varying sized and keyed requests and reorder them to derive the minimum total cost, or runtime.

We can draw a loose analogy between this problem and the perfect packing version of the 2-dimensional strip packing problem [108]. If we let the cost of each CUDA block become the height of an object, the width of the object is 1 and the width of the container is the number of available SMs, then we wish to minimise the height of the container holding all the objects. The analogy is not exact as we also have the added complexity that the height of each object, i.e. the CUDA block cost, can vary depending on how requests are ordered. To find the optimal solution to this is computationally impractical. However, we can use heuristics to arrive at a reasonable solution. If we first consider that the block cost increases whenever an exponent changes within the array of requests, we should sort all requests according to their exponent, thus creating a list of non-divergent groups of requests. We perform this sort by an approximation, using only the first integer of the exponent, giving a good accuracy/efficiency trade off. We label this approach as “1 pass”.

We do not have control over the order in which the Nvidia driver chooses its CUDA blocks for execution when an SM becomes free, however it is reasonable to assume it follows a first fit approach, i.e. whenever an SM is free it takes the next lowest block by ID and assigns it to the SM. A reasonable close to optimal approach to solving the strip packing problem is to use the first fit descending heuristic. We follow this heuristic by sorting the non-divergent groups in increasing order of the number of operations within each group. This ensures that the most costly CUDA blocks occur in the lower block IDs. We call this approach “2 pass” as it involves the 1 pass sort above and an extra sort.

Both the 1 and 2 pass techniques are contrasted with no sorting (0 pass) in various scenarios in Figure 7.10. The scenarios are run outside of the OCF as they concern modular exponentiation batching on the GPU in general and not just in the context of the framework. The tests consisted of sending multiple requests to the GPU for concurrent execution. The size of each request within each test was randomly chosen in a guided manner. The “Large” tests restricted the sizes of the requests (the number of bases per request) to be high, typically 100-300; the “Small” tests contained only small requests, typically 1-10; and the “Mixed” tests contained a random mixture of large and small request sizes. Each test was run with a varying probability for each request to be followed by a request with the same exponent and modulus, i.e. the same key. This is labelled “Collision Probability”, with 1 meaning all requests are using the same key and 1/512 meaning a 1 in 512 chance of two requests chosen at random from the

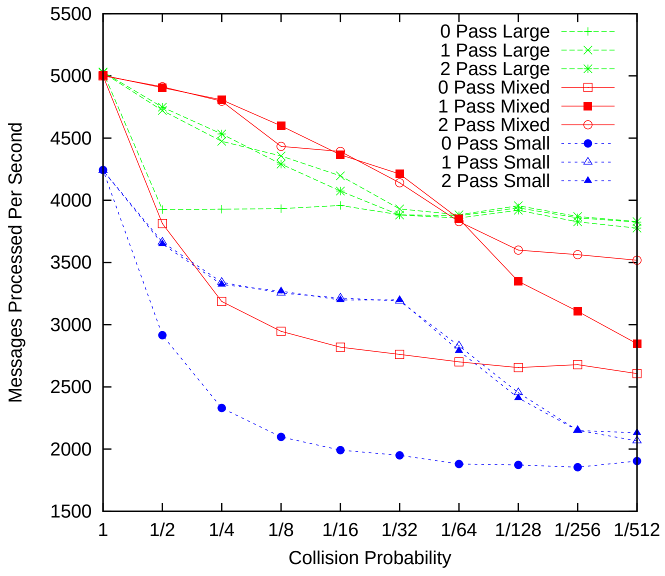


Figure 7.10: Comparison of Pre-processing techniques for RSA-1024 Request Batching.

test having the same key. This collision probability simulates a multithreaded environment sending requests to the OCF with differing numbers of system wide keys.

Figure 7.10 presents the relative performance of the 0, 1 and 2 pass techniques within each scenario. It can be seen that the 0 pass approach underperforms in all scenarios. The 1 and 2 pass techniques mostly perform the same with a general slight overhead noticeable for the 2 pass approach. The 2 pass approach substantially outperforms the 1 pass approach when the collision probability is low and there is a mix of request sizes. The performance improvement of 2 pass at a collision probability of 1/512 is 24%. Small requests are more costly than large requests as the rate of change of the exponent is higher. These small messages when mixed randomly between lower cost large requests, form a layer of thinly distributed costly warps. It is beneficial to move these costly small requests into a small number of high cost blocks as the block costs converge on a relatively small overhead. This increases the number of low cost blocks that can run concurrently and finish while the high costs blocks complete. We recommend the use of this 2 pass approach due to its better performance in this scenario and relatively small overhead in the general case.

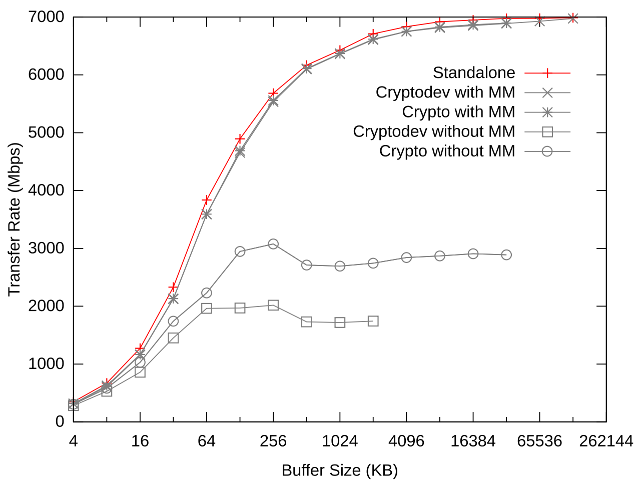


Figure 7.11: Performance of GPU accelerated AES using the OCF.

7.3.3 Request Pipelining

In the scenario where we have multiple cryptographic requests outstanding on the Gpucryptd daemon queue, we have the opportunity to split the processing and return of requests into two concurrent operations. The CUDA API allows for the execution of a kernel on the GPU asynchronously. The Gpucryptd daemon permits both asymmetric and symmetric CUDA modules (see Figure 7.8) to retrieve more requests from the queue without returning. The daemon also supports callbacks to return completed requests. This allows the Gpycryptd daemon to delegate the flow control of request retrieval and request return to the cryptographic modules. Thus, when implementing a cryptographic algorithm for the GPU, it is straight forward to overlap the return of previously completed requests with the execution of the next requests. We present the effects of pipelining in Section 7.4.1.

7.4 Performance

7.4.1 Symmetric-Key Performance

To analyse the overhead of using the OCF for symmetric-key performance, we use an AES module based on the peak performance implementation presented in Section 5.1.3 combined with OCF symmetric-key requests. Figure 7.11 shows the performance of AES when operating on different sized buffers with and without going through the OCF. The non-OCF version of the implementation running in ECB mode, is labelled as "Standalone". We compare this standalone version to four other tests. Two tests were performed using normal userspace processes to initiate the requests and thus go via the Cryptodev layer of the OCF, labelled as "Cryptodev with/without MM". The remaining two tests were performed using a kernel thread which initiated the requests directly via the Crypto layer of the OCF, labelled "Crypto with/without MM". The "with MM" and "without MM" tags, refer to variants of the tests whereby we either include our new memory management system or use the original OCF memory management respectively.

We can see that the two tests which use the OCF and the new memory management system, perform with a small overhead compared to the standalone version. Based on the Cryptodev interface, the average percentage overhead of using the OCF is 3.4%, with a range of 9.3% for the smallest request buffers through to 0.2% for the largest buffers. The spread in overhead percentage is due to the smaller request buffers requiring more calls through the OCF to perform the same amount of processing compared with larger request buffers. Although it cannot be seen here, there is a slight advantage to executing the cryptographic requests from the kernel as the Cryptodev layer overhead is removed.

The two tests, which are performed without the new memory management system, experience a substantial reduction in performance as the buffer sizes increase. This is due to having to perform extra memory copies for each transition between address spaces. The shape of the graphs can be understood when compared to Figure 7.4. The reason for "Crypto without MM" outperforming "Cryptodev without MM", is that the direct calls to the Crypto layer from kernelspace eliminates one of the address space transitions, thus reducing the number of memory copies performed. The reason for "Crypto without MM" not covering the full range of buffer sizes is that it hits the default maximum `vmap()` limit in the kernel. "Cryptodev without MM" is also limited in the buffer sizes used due to the original limit imposed by the OCF. These limits can be removed, though the results show little difference.

Figure 7.12 is used to investigate both multithread scalability and pipelining, as discussed in Section 7.3.3, for symmetric-key processing on the GPU. The "Single Thread" test is the same as "Standalone" in Figure 7.11, involving multiple iterations of symmetric-key requests with varying buffer sizes. The multithreaded tests consist of executing the same amount of operations as the Single Thread test, using the same sized requests, however the requests are split across 20 threads. One of the multithreaded test runs using the previously mentioned request pipelining, and the other without. Note that the multiple threads referred to are the consumer threads making requests to the OCF, the Gpucryptd daemon itself remains a single thread. It can be seen at small buffer sizes

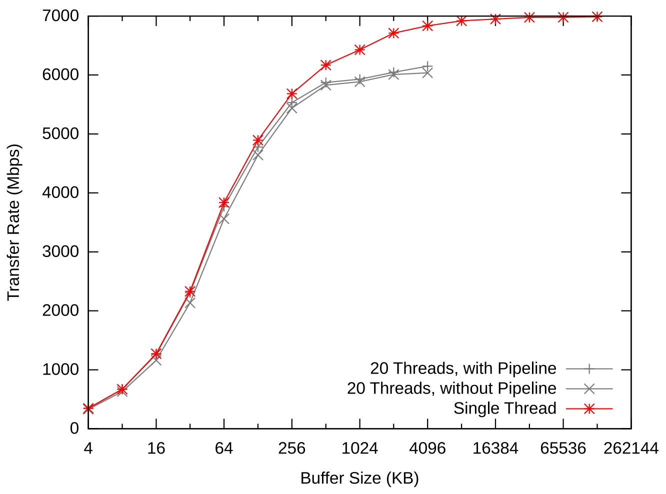


Figure 7.12: Multithreaded performance of GPU accelerated AES using the OCF.

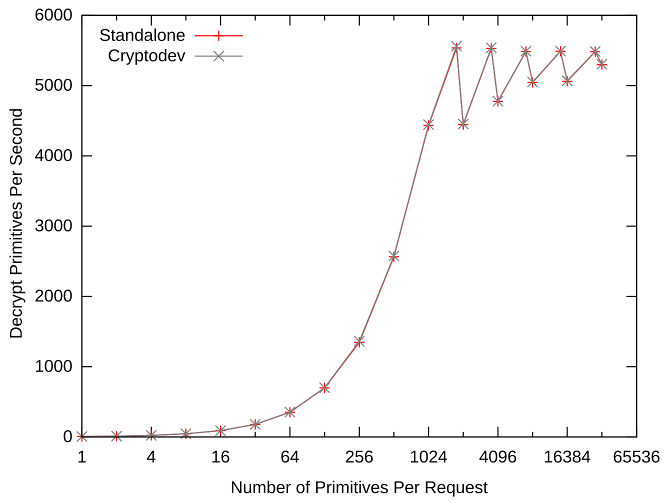


Figure 7.13: Performance of GPU accelerated RSA-1024 using the OCF.

that the multithreaded scenario using the pipeline slightly out performs the scenario without pipeline use. It achieves this improvement from asynchronously returning request results, thus hiding (or partially hiding) the return cost to the consumer. Both of the multithreaded tests taper prematurely in performance as the buffer size increases. This is presumed to be due to the multithread version of these tests requiring 20 times more active memory at any one time than the single thread version. Thus the performance degrades due to increased pressure on system memory.

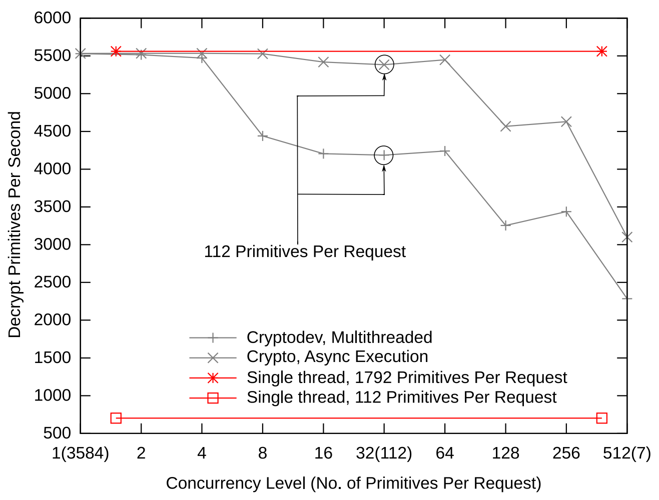


Figure 7.14: Concurrency and GPU accelerated RSA-1024 using the OCF.

7.4.2 Asymmetric-Key Performance

For our tests of asymmetric-key performance we use OCF asymmetric-key requests in combination with a CUDA module based on the peak performance modular exponentiation approach for RSA-1024 presented in Chapter 6. Figure 7.13 shows a comparison of running a standalone version of this implementation and using the implementation via the OCF Cryptodev layer from a userspace process. We can see here that there is no discernible difference in performance, in fact it is difficult to see there are two plotted graphs in the figure. This is due to the high arithmetic intensity inherent in the modular exponentiation algorithm and thus the OCF overhead is relatively quite small. The average percentage overhead of using the OCF via the Cryptodev interface compared to the standalone version is 0.4%, with a range of 0.6% for the smallest number of messages per request to 0.1% for the largest. A related point is that we have performed these tests with and without the new memory management system and also with and without pipelining as in the symmetric-key tests above. The results were indistinguishable from the standalone version due to the small overhead associated with data transfer through the OCF compared to the work done on the GPU.

Figure 7.14 illustrates the behaviour of the OCF when processing multiple asymmetric-key requests with the same key concurrently. We achieve concurrency by using multiple threads via the Cryptodev interface. As it is a blocking interface there is no other way a userspace thread can achieve concurrency. We also test concurrency via direct kernel calls to the Crypto layer, which permits asynchronous request execution. This permits multiple outstanding requests within the Gpucrypt request queue at one time using a single kernel thread. The "Concurrency Level" label in the figure refers to either the number of threads (Cryptodev test) or the number of concurrent requests sent asynchronously (Crypto test). All tests, both multithreaded and single threaded, perform the same total number of asymmetric operations. Thus, as the concurrency increases the number of messages per request decreases, shown in brackets on the x-axis. We have highlighted the performance improvement when processing requests consisting of 112 primitives concurrently versus serially. This improvement is due to the use of batching as described in Section 7.3.2. From Figure 7.14 we can also see that the multithreaded tests lose performance as the concurrency level increases. The main reason for this performance degradation is the inability to maintain an occupied GPU. In the tests, as the concurrency increases the request sizes decrease, the OS has a harder time to deliver sufficient numbers of requests to the queue for batching due to process switching overheads. This relates to the reason the Crypto test outperforms the Cryptodev test. The OS does not have to reschedule processes as frequently to deliver the same amount of data to the GPU.

7.5 Conclusions

We have seen that the GPU can be effectively integrated into the OCF with careful design of a driver consisting of a kernelspace OCF driver and a userspace daemon. The chapter shows that there is an average overhead of 3.4% when using the OCF for AES over a standalone implementation. In the context of RSA-1024 we see that there is a very low 0.3% average overhead when compared to a standalone version. A new memory management system within the OCF was shown to be critical in maintaining this performance for symmetric-key operations. Without its use we see a drop in performance of over 50% when using the OCF's kernelspace Crypto interface, and over a 70% drop via the OCF's userspace Cryptodev interface.

We presented a new general purpose mechanism for processing multiple asymmetric-key requests on the GPU and found that the preprocessing of mixed key requests is crucial to maintaining performance. We have also shown the effectiveness of integrating this mechanism as part of the OCF and its use within multithreaded and asynchronous scenarios. The most important factor regarding performance in these scenarios is the ability of the OS to schedule multiple threads efficiently so as to provide enough work for the GPU to reach peak performance. We have seen that GPU accelerated cryptographic functions can be made available in a uniform manner to all OS components, both in-kernel and userspace, via the OCF without excessive overhead.

Chapter 8

Review and Outlook

The aim of this thesis was to investigate the feasibility and performance of using commodity graphics processors for the execution of cryptographic functions. We have seen that it is possible to use standard off-the-shelf GPUs to provide significant performance improvements over CPUs for both symmetric-key and asymmetric-key algorithms. However, we have also seen that such improvements are not applicable to all cryptographic scenarios. The GPU presents a number of obstacles to achieving high performance. The majority of the work in this thesis has been concerned with implementation techniques that minimise the effect of these obstacles, and the highlighting of the cryptographic scenarios that reduce the exposure to these obstacles. The following is a summary of the conclusions reached in each of the chapters. Afterwards we present thoughts on broader implications, which we hope are relevant to cryptography in the context of future generations of highly parallel devices similar to the GPU.

Appendix B.2 lists the disparity between system bus (CPU to GPU) bandwidth and the on-board bandwidth between the GPU and device memory. For example, the GeForce 8800GTX has 86.4 GB/s device memory bandwidth, whereas the PCIe system bus has a peak bandwidth of 4 GB/s. In Chapter 3 we point out that the relatively slow system bus speeds is one of the main GPU performance bottlenecks for applications that require large amounts of data transfer. We have seen that efficient data transfer across the system bus is critical to performance for symmetric-key processing. Chapter 3 discusses the relatively straightforward manner in achieving maximum system bus transfer rates for DX10 GPUs via CUDA. However, when using DX9 GPUs via OpenGL, it is necessary to interact with a complex array of configuration states that can affect transfer rates in unpredictable ways. We introduced two new tools that can be used to investigate this array of configuration states. Using these tools, we outline the numerous pitfalls that can lead to transfer rate reduction through slight misconfiguration of the OpenGL pipeline. The tools can also be used to investigate the asynchronous efficiency of data readback from the GPU to the CPU. We present the many pitfalls where configuration states can unexpectedly disable asynchronous readbacks or diminish asynchronous readback efficiency. These tools were used to minimising the data transfer bottleneck in all of the presented DX9 symmetric-key implementations.

Chapter 4 presents a number of AES implementations on DirectX 9 hardware using OpenGL. One of the main bottlenecks to AES performance on DX9 hardware is texture access rates. Two types of texture lookups are used, one for assisting in the execution of an optimised AES approach, and one for simulating XOR operations. The second type is used because XOR operations are not supported by the programmable units within DX9 GPUs. We presented two approaches to simulate the XOR operation using 8-bit and 4-bit lookups from pregenerated tables. We can avoid the use of these lookup tables if XOR is performed in the ROP stage of the pipeline. However, using the ROP stage requires one pipeline pass per XOR. Our results show that the overhead of multiple pipeline passes per AES block is a lot less than using table lookups. Texture lookups are the primary bottleneck to performance of AES, even when the ROP XOR approach is used. The presented implementations are I/O bound as the addition of a small number of extra arithmetic instructions has no effect on the throughput rate, meaning that the kernel programs are blocked on memory access.

The fastest AES approach uses the ROP XOR technique, and gives a peak throughput rate of 870 Mb/s on a GeForce 7900GT. This implementation was the first demonstration of AES, or any symmetric-key algorithm, on DX9 hardware with similar speeds to the traditional CPU. We show that a large payload size is required to reach this peak performance. This reduces the applicability of the GPU to latency insensitive applications. The chapter also discussed a previously reported issue of an OpenGL based AES implementation consuming 100% of CPU cycles while processing on the GPU. This is of concern if the GPU is to be used as a cryptographic co-processor. We show that for our fastest ROP based AES implementation, $\sim$75% of the CPU cycles are free to execute other work while the GPU is executing.

Chapter 5 shows that AES can execute efficiently on DirectX 10 hardware using CUDA. A number of different AES implementations are explored, with an approach based on multiple lookup tables residing in shared memory proving the fastest. Using the CRT mode of operation, the implementation achieved a throughput rate of 7,234 Mb/s, and 17,571 Mb/s without data transfer across the system bus. These are the fastest AES rates reported on a GPU. Comparing to a similar era CPU, the GPU implementation provides a ∼2.5x and ∼6x increase in performance with and without data transfer respectively. In comparison with the next fastest GPU based AES implementation, we see a ∼2x increase in performance. The speed improvement over the DX9 generation of GPUs is in the order of ∼8 times. Again, we see the need for large payloads of data in the range of 256 KB to 512 KB before throughput rates approach peak performance. Many cryptographic applications use much smaller buffer sizes and as such do not suit direct offload to the GPU unless multiple such buffers can be batched together efficiently. Another concern is that some security protocols use serial modes of operation, such as IPSec and CBC. We have shown the GPU can implement serial modes, however one thread is responsible for the processing of a full message. As such, the requirement for multiple messages, and large payloads, is even more applicable than when executing parallel modes.

The peak throughput rates reported in this chapter are based on an optimised implementation of AES. That is, with many hardcoded assumptions such as data location, data size, cryptographic algorithm used, single message, single key, etc. We introduce a data model and a serialisation approach, which generates data streams representing meta data for symmetric-key processing, suitable for offload to the GPU. Using this data model, serialisation and a mapping scheme we can use the GPU via a cryptographic API in a practical context. This abstraction layer incurs a performance overhead of 16% to 45% compared to the optimised implementation. The higher the message count and the smaller the message size within a payload, the more expensive the overhead is. The main cause of this overhead is an index and descriptor stream generation and lookup. These support the mapping between threads and associated data and processing instructions. An increase in message count sees an increase in index and descriptor stream size, which reduces the effectiveness of the on-chip caches. Also presented in Chapter 5 is an analysis of the execution of different modes of operation. The ordering of serial and parallel mode message execution within a payload is important with regard to processing performance. This is illustrated by a comparison of different message arrangements, which shows that careful ordering results in large performance improvements over random ordering.

Chapter 6 presents a number of successful implementations of large integer modular exponentiation on DirectX 10 hardware suitable for asymmetric-key algorithms. The two main categories of techniques we employed were based on different number representation systems, radix and RNS. In comparison with a CPU implementation of RSA-1024, we see a ∼4x and ∼2.6x performance improvement for the peak radix and RNS based approaches respectively. The recurring constraint of high data amounts is also applicable to these results. Using the same CPU comparison, the RNS approach requires 32 messages per payload to be faster, whereas the radix approach requires 256 messages per payload. RNS proves faster at reduced data sizes due to its increased parallelism. The RNS approach presented, splits RSA-1024 across 34 threads, whereas the fastest radix approach is split across just two threads. An increase in parallelism for the radix based approach was investigated. We distributed the work across 32 threads, however at no payload size point did this approach prove faster than either the RNS or serial radix approaches. For best modular exponentiation performance on the GPU an adaptive approach should be used, whereby smaller payload sizes use an RNS implementation and larger payload sizes transition to a radix implementation.

The effect of using multiple different keys within a payload was analysed in the context of RSA-1024 and the peak RNS and radix based approaches. To achieve peak performance, assuming sufficient data is available, it was determined that the key should change at a maximum rate of once per 15 messages when using the RNS based approach and once per 32 messages when using the radix based approach. These restrictions are due to the underlying SIMD architecture and the synchronisation functionality of the GPU. In general, the GPU based implementations of modular exponentiation are bound by the speed of device memory read and writes. In the context of the RNS approach, we presented multiple attempts to improve the rate of single-precision modular arithmetic due to the slow speed of integer divide execution on the GPU. Also, in the context of the radix approach, one of the main performance issues is the slow integer multiply operation, taking 4 times the number of cycles as floating point multiply. These factors may improve in the future, however the GPU is primarily a graphics acceleration device and as such focuses the transistor budget on efficient floating point operation. Perhaps with the growth in GPGPU popularity we will see an increase in integer efficiency within these devices.

We saw the successful integration of the peak AES and RSA implementations presented in previous chapters as part of an operating system service in Chapter 7. The implementations were integrated into the OpenBSD Cryptographic Framework, an established OS virtualisation layer used to abstract users from cryptographic implementations, both software and hardware. This integration allows transparent use of GPU implementations by both kernelspace components and userspace applications. One of the main stumbling blocks with the integration is that CUDA is a userspace runtime library, and as such cannot be interacted with directly from kernelspace. We implemented a kernelspace driver and userspace daemon pair to provide a mechanism for the kernel to “request” processing from CUDA. The OCF performs poorly when processing large buffers due to memory copy costs. To remove this cost, we implemented a new memory management system within the OCF to eliminate all memory copies. This is an important improvement for the GPU, which would otherwise incur multiple memory copies between address spaces. In comparison to the optimised AES and RSA implementations, the OCF integrated versions result in an average overhead of 3.4% for AES and 0.3% for RSA. Also shown in this chapter was the integration of a new general purpose mechanism for processing multiple asymmetric-key requests on the GPU with the OCF. The ordering of mixed key requests is shown to be important to performance. Also highlighted was the significant impact of OS thread scheduling efficiency on performance with regard to multithreaded and asynchronous client requests.

8.1 General Lessons

The GPU can be viewed as an example of a highly parallel processor for general purpose computation, and as such some of the conclusions related to this work can be seen as being applicable to similar devices. We can view the following conclusions as applicable to current highly parallel devices as well as to future highly parallel devices, such as future GPU architectures. We have seen a constant pressure to maintain a high occupancy on the GPU. A failure to do so results in substantial performance loss. Regarding similar highly parallel processors, occupancy will continue to be a concern and affecting their suitability to general problems. One of the main criteria for high cryptographic throughput on the GPU is the processing of large numbers of blocks or messages for each kernel execution. In data parallel processing, the number of data elements largely determines the number of active threads. Also, in highly parallel devices the ALUs are generally simplified. It follows that this type of device relies on both a high arithmetic intensity and a high number of threads for memory latency hiding. As such, for highly parallel devices in general, we expect their ability to out-perform standard CPU implementations to be dependant on scenarios that require bulk cryptographic processing. We expect that symmetric-key and asymmetric-key processing performance on highly parallel devices will produce throughput rates dependent on payload size as has been presented in the figures throughout this thesis.

A technique used to increase the computational density of an architecture is to use a SIMD or vector design, where a single instruction is issued to a group of simple execution units. This approach is commonly used in the design of highly parallel devices. Nvidia DX10 GPUs, as we have seen, use 8-wide SIMD multiprocessors, with the same instruction issued to each of the 8 ALUs for 4 consecutive cycles. AMD's first DX10 GPU uses 16-wide SIMD clusters, where each cluster processes a 5-wide very long instruction word instructions. Intel's Larrabee is based around a 16-wide vector processing unit. We have seen the issues this wide processing width can have on performance. If thread branching occurs within a thread group, the group size tending to relate to the vector or SIMD width, the processor can suffer from occupancy problems where processing units are left idle. Relating to asymmetric-key processing, this occupancy issue occurs when a key is changed within a payload. The exponent determines the flow of control, thus different exponents can cause thread divergence. We have also seen this for symmetric-key processing when serial and parallel modes of operations are mixed, or when serial mode messages of different sizes are mixed. The conclusions drawn in this thesis regarding key change, message sorting and thread mapping for efficient cryptographic processing are expected to be relevant to similar highly parallel processors.

We have seen that thread synchronisation presents a functional restriction on the key change rate supported for RSA. This is due to a general problem related to branching within a group of co-operating threads. If threads are to co-operate as a group, the group size tending to relate to the underlying hardware, some form of synchronisation barrier is required at which point all threads wait to proceed. However, if some threads within the group branch and never reach such a synchronisation barrier, unexpected results may occur. It is likely that highly parallel processors will provide some form of synchronisation mechanism between threads and will run into a similar issue regarding multiple key support. Also, throughout this research we have endeavoured to make efficient use of scarce per-thread on-chip memory, for example, the use of a compressed index for general symmetric-key mapping. Similar memory restrictions are likely on other highly parallel devices and as such careful on-chip memory use will continue to be an area of optimisation. In general, it is expected that for payloads with small amounts of cryptographic work, or those with a high rate of key change for asymmetric-key work, the traditional CPU will continue to perform best. However, in terms of relatively large and homogeneous cryptographic workloads, the best performance is likely to continue to be found on the GPU and other highly parallel processors.

8.2 Future Work

It would be of interest to implement AES and RSA on new and upcoming architectures. AMD have recently released the first DX11 compliant GPU. As of yet there are no cryptographic implementations using the Stream SDK, AMD's CUDA equivalent. Also of interest would be an investigation into the new OpenCL [56] standard. This standard is a vendor neutral approach for parallel programming on heterogeneous platforms. AMD GPUs, Nvidia GPUs, and Intel's Larrabee, currently do, or shortly will, support OpenCL. While cryptographic implementations based on OpenCL are preferable to developing and maintaining per vendor implementations, the performance overhead of being vendor neutral needs to be gauged. Another architectural improvement of note is the upcoming native AES instructions part of Intel's Advanced Vector Extensions [49]. Informal reports suggest that a performance improvement of 3x is expected over an equivalent CPU without such instructions.

An investigation is required into the suitability of using the GPU in security sensitive contexts. Possible attacks and best practices for secure use should be detailed. For example, CUDA provides a unique memory space for each thread executing kernel calls. The robustness of using this for memory protection in the context of all the GPU access methods should be explored. The work presented in this thesis has only considered the feasibility and performance issues related to accelerating cryptographic functions. Until a security analysis is performed, it can only be recommended that the GPU be used within a trusted computing context where untrusted users do not have access to the system.

Future work specific to symmetric-key cryptography on the GPU includes an investigation into authenticated encryption modes of operation. This is expected to draw similar conclusions to the modes analysis in Chapter 5, as the authenticated encryption modes largely divide into parallel (e.g. GCM [68], CWC [59]) and serial (e.g. CCM [126], EAX [6]) approaches. Regarding asymmetric-key cryptography it would be interesting to see the scalability of the RSA implementations for larger key sizes, such as 2048 or 4096-bit, in light of the increase of certain on-chip memory resources in newer GPUs. Finally, an ongoing observation should be made regarding the gap in performance between the CPU and GPU. Recently this performance gap appears to be widening in favour of the GPU. Benchmarking the presented implementations on newer GPUs and CPUs would help confirm, in a cryptographic context, whether this performance trend is continuing or is the increase in CPU cores negating this difference.

Bibliography

[1] AMD. ATI CTM Guide: Technical Reference Manual, Version 1.01. http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf.

[2] AMD. ATI Stream Technology. http://www.amd.com/stream.

[3] AMD. Fusion Project. http://fusion.amd.com/.

[4] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks. Technical report, NASA Ames Research Center, Moffett Field, CA, USA, March 1994. RNR Technical Report RNR-94-007.

[5] M. Bellare and P. Rogaway. Optimal Asymmetric Encryption - How to Encrypt with RSA. In *International Conference on Advances in Cryptology - Eurocrypt*, pages 92–111, Saint-Malo, France, May 1995. Springer.

[6] M. Bellare, P. Rogaway, and D. Wagner. A conventional authenticated-encryption mode, September 2003.

[7] D. Bernstein, T.-R. Chen, C.-M. Cheng, T. Lange, and B.-Y. Yang. ECM on Graphics Cards. In *International Conference on Advances in Cryptology - Eurocrypt*, pages 483–501, Cologne, Germany, April 2009. Springer.

[8] E. Biham. A Fast New DES Implementation in Software. In *Fast Software Encryption*, pages 260–272, Haifa, Israel, January 1997. Springer.

[9] I. Blake, G. Seroussi, and N. Smart, editors. *Advances in Elliptic Curve Cryptography. Second Edition.* Cambridge University Press, 2005.

[10] D. Blythe. The Direct 3D 10 System. *ACM Transactions on Graphics*, 25(3):724–734, July 2006.

[11] A. Boeing. Survey and future trends of efficient cryptographic function implementations on GPGPUs. In *Australian Digital Forensics Conference*, pages 59–69, Perth, Western Australia, December 2008. Security Research Centre, Edith Cowan University.

[12] I. Buck. Data Parallel Computing on Graphics Hardware. In *Siggraph: Graphics Hardware Panel*, San Diego, USA, July 2003. `http://graphics.stanford.edu/~ianbuck/GH03_datapargfx.pdf`.

[13] I. Buck, K. Fatahalian, and P. Hanrahan. Gpubench: Evaluating gpu performance for numerical and scientific applications. In *ACM Workshop on General Purpose Computing on Graphics Processors (Poster Session)*, LA, USA, August 2004. ACM.

[14] D. Cook, R. Baratto, and A. Keromytis. Remotely Keyed Cryptographics Secure Remote Display Access Using (Mostly) Untrusted Hardware. In *ICICS05 Conference Proceedings*, pages 363–375, Beijing, China, December 2005. Springer.

[15] D. Cook, J. Ioannidis, A. Keromytis, and J. Luck. CryptoGraphics: Secret Key Cryptography Using Graphics Cards. In *RSA Conference, Cryptographer's Track (CT-RSA)*, pages 334–350, San Francisco, CA, USA, February 2005. Springer.

[16] N. Costigan and P. Schwabe. Fast Elliptic-Curve Cryptography on the Cell Broadband Engine. In *AFRICACRYPT: International Conference on Cryptology in Africa*, pages 368–385, Gammarth, Tunisia, June 2009. Springer-Verlag.

[17] N. Costigan and M. Scott. Accelerating SSL Using the Vector Processors in IBM's Cell Broadband Engine for Sony's PlayStation 3. *Cryptology ePrint Archive*, 2007/061, 2007.

[18] S. Coutinho. *The Mathematics of Ciphers: Number Theory and RSA Cryptography.* A. K. Peters, Ltd., 1999.

[19] Crypto++ Library. `http://www.cryptopp.com/`.

[20] J. Daemen and V. Rijmen. The Block Cipher Rijndael. In *The International Conference on Smart Card Research and Applications*, pages 277–284, Louvain-la-Neuve, Belgium, September 1998.

[21] J. Daemen and V. Rijmen. AES submission document on Rijndael, Version 2, September 1999.
`http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf`.

[22] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22:644–654, November 1976.

[23] W. Diffie and M. Hellman. Privacy and Authentication: An Introduction to Cryptography. *IEEE*, 67(3):397–427, March 1979.

[24] W. Ehrsam, C. Meyer, J. Smith, and W. Tuchman. Message verification and transmission error detection by block chaining. *US Patent 4074066*, February 1978.

[25] H. Feistel. Cryptography and Computer Privacy. *Scientific American*, 228(5):15–23, May 1973.

[26] M. Feldhofer, K. Lemke, E. Oswald, F.-X. Standaert, T. Wollinger, and J. Wolkerstorfer. D.VAM.2: State of the Art in Hardware Architectures. *ECRYPT: European Network of Excellence in Cryptology*, IST-2002-507932, September 2005.

[27] S. Fleissner. GPU-Accelerated Montgomery Exponentiation. In *International Conference of Computational Science*, pages 213–220, Beijing, China, May 2007. Springer.

[28] Folding@home Distributed Computing.
`http://folding.stanford.edu/`.

[29] A. Freier, P. Karlton, and P. Kocher. The SSL Protocol, Version 3.0, November 1998.
`http://www.mozilla.org/projects/security/pki/nss/ssl/`.

[30] J. Fung and S. Mann. OpenVIDIA: parallel GPU computer vision. In *ACM International Conference on Multimedia*, pages 849–852, New York, USA, November 2005.

[31] H. Garner. The Residue Number System. In *IRE-AIEE-ACM '59 (Western): Papers presented at the western joint computer conference*, pages 146–153, San Francisco, California, March 1959. ACM.

[32] GNU Multiple Precision Arithmetic Library.
`http://gmplib.org/`.

[33] D. Göddeke. GPGPU: Basic Math Tutorial. Technical report, FB Mathematik, Universität Dortmund, November 2005. Ergebnisberichte des Instituts für Angewandte Mathematik, Nummer 300.

[34] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: High performance graphics coprocessor sorting for large database management. In *ACM SIGMOD International Conference on Management of Data*, pages 325–336, Chicago, USA, June 2006.

[35] N. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A Memory Model for Scientific Algorithms on Graphics Processors. In *ACM/IEEE Supercomputing Conference*, page 89, Florida, USA, November 2006. ACM.

[36] N. Govindaraju, N. Raghuvanshi, and D. Manocha. Fast and Approximate Stream Mining of Quantiles and Frequencies Using Graphics Processors. In *ACM SIGMOD International Conference on Management of Data*, pages 611–622, New York, USA, June 2005.

[37] GPGPU Online Community.
`http://gpgpu.org/`.

[38] T. Granlund and P. Montgomery. Division by Invariant Integers using Multiplication. In *Conference on Programming Language Design and Implementation*, pages 61–72, Orlando, Florida, June 1994. ACM.

[39] K. Gray. *Microsoft DirectX 9 Programmable Graphics Pipeline*. Microsoft Press, 2003.

[40] O. Harrison and J. Waldron. AES Encryption Implementation and Analysis on Commodity Graphics Processing Units. In *CHES: International Workshop on Cryptographic Hardware and Embedded Systems*, pages 209–226, Vienna, Austria, September 2007. Springer-Verlag.

[41] O. Harrison and J. Waldron. Optimising Data Movement Rates for Parallel Processing Applications on Graphics Processors. In *25th International*

Conference on Parallel and Distributed Computing and Networks, pages 251–256, Innsbruck, Austria, February 2007.

[42] O. Harrison and J. Waldron. TransferBench Tool, 2007. `http://www.cs.tcd.ie/∼harrisoo/transferBench/`.

[43] O. Harrison and J. Waldron. Practical Symmetric Key Cryptography on Modern Graphics Hardware. In *USENIX Security Symposium*, pages 195–209, San Jose, CA, USA, July 2008. Usenix Association.

[44] O. Harrison and J. Waldron. Efficient Acceleration of Asymmetric Cryptography on Graphics Hardware. In *AFRICACRYPT: International Conference on Cryptology in Africa*, pages 350–367, Gammarth, Tunisia, June 2009. Springer-Verlag.

[45] O. Harrison and J. Waldron. GPU Accelerated Cryptography as an OS Service, September 2009. `https://www.cs.tcd.ie/publications/tech-reports/reports.09/TCD-CS-2009-36.pdf`.

[46] O. Harrison and J. Waldron. Public Key Cryptography on Graphics Hardware. In *Eurocrypt: Annual International Conference on the Theory and Applications of Cryptographic Techniques (booklet)*, pages 65–72, Cologne, Germany, April 2009.

[47] M. Hill and A. Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, 38(12):1612–1630, 1989.

[48] Intel. Accelerated Graphics Port, V3.0, Interface Specification, 2002. `http://download.intel.com/support/motherboards/desktop/sb/agp30.pdf`.

[49] Intel. AVX: New Frontiers in Performance Improvements and Energy Efficiency, March 2008. `http://software.intel.com/en-us/articles/intel-avx-new-frontiers-in-performance-improvements-and-energy-efficiency`.

[50] N. Jacob and C. Brodley. Offloading IDS Computation to the GPU. In *Annual Computer Security Applications Conference*, pages 371–380, Miami Beach, FL, USA, December 2006. IEEE Publishing.

[51] K. Järvinen, M. Tommiska, and J. Skyttä. Comparative Survey of High-Performance Cryptographic Algorithm Implementations on FPGAs. *IEE Proceedings - Information Security*, 152(1):3–12, 2005.

[52] A. Karatsuba and Y. Ofman. Multiplication of Many-Digital Numbers by Automatic Computers. *Doklady Akad. Nauk SSSR*, 145:293–294, 1962.

[53] S. Kawamura, M. Koike, F. Sano, and A. Shimbo. Cox-Rower Architecture for Fast Parallel Montgomery Multiplication In Advances in Cryptology. In *International Conference on Advances in Cryptology Eurocrypt*, pages 523–538, Bruges, Belgium, May 2000. Springer.

[54] G. Kedem and Y. Ishihara. Brute force attack on UNIX passwords with SIMD computer. In *USENIX Security Symposium*, pages 8–8, Washington, D.C., USA, 1999. USENIX Association.

[55] A. Keromytis, J. Wright, and T. de Raadt. The Design of the OpenBSD Cryptographic Framework. In *USENIX Annual Technical Conference*, pages 181–196, San Antonio, Texas, USA, June 2003. Usenix Association.

[56] Khronos Group. OpenCL - The open standard for parallel programming of heterogeneous systems.
`http://www.khronos.org/opencl/`.

[57] D. Kirk and W-m. Hwu. *ECE 498 AL: Programming Massively Parallel Processors*. 2009. Chapter 4: CUDA Memories.

[58] D. Knuth. *The Art of Computer Programming, Volume 2*. Addison-Wesley, 3 edition, 1997.

[59] T. Kohno, J. Viega, and D. Whiting. CWC: A high-performance conventional authenticated encryption mode, January 2004.

[60] D. Kwon, J. Kim, S. Park, S. H. Sung, Y. Sohn, J. H. Song, Y. Yeom, E-J. Yoon, S. Lee, J. Lee, S. Chee, D. Han, and J. Hong. New Block Cipher: ARIA. In *International Conference on Information Security and Cryptology*, pages 432–445, Seoul, Korea, November 2003. Springer.

[61] S. Leffler. Cryptographic device support for FreeBSD. In *Usenix, BSD Conference*, pages 69–78, San Mateo, California, USA, September 2003. Usenix Association.

[62] R. Lidl and H. Niederreiter. *Introduction to finite fields and their applications. Revised Edition.* Cambridge University Press, 1994.

[63] linux-crypto (Crypto API).
`http://mail.nl.linux.org/linux-crypto/`.

[64] H. Lipmaa, P. Rogaway, and D. Wagner. Comments to NIST concerning AES Modes of Operations: CTR-Mode Encryption, 2000.

[65] S. Manavski. CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography. In *IEEE International Conference on Signal Processing and Communications*, pages 65–68, Dubai, November 2007. IEEE Publishing.

[66] W. Mark, R. Glanville, K. Akeley, and M. Kilgard. Cg: A system for programming graphics hardware in a C-like language. *ACM Transactions on Graphics*, 22(3):896–907, July 2003.

[67] M. Matsui. How Far Can We Go on the x64 Processors? In *Fast Software Encryption*, pages 341–358, Graz, Austria, March 2006. Springer.

[68] D. McGrew and J. Viega. The Galois/Counter Mode of Operation (GCM), 2005.

[69] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, October 1996.

[70] Microsoft. DirectX 10.
`http://msdn.microsoft.com/directx/`.

[71] Microsoft. DirectX 9.0.
`http://msdn.microsoft.com/en-gb/library/bb318659(VS.85).aspx`.

[72] Microsoft. Microsoft High-Level Shading Language.
`http://msdn.microsoft.com/en-us/library/bb509561(VS.85).aspx`.

[73] Microsoft. Shader Model 4.0.
`http://msdn.microsoft.com/en-us/library/bb509657(VS.85).aspx`.

[74] P. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44:519–521, 1985.

[75] A. Moss, D. Page, and N.P. Smart. Toward Acceleration of RSA Using 3D Graphics Hardware. In *IMA International Conference on Cryptography and Coding*, pages 364–383, Cirencester, UK, December 2007. Springer.

[76] National Institute of Standards and Technology. FIPS-46-3: Data Encryption Standard, July 1977.

[77] National Institute of Standards and Technology. FIPS-81: DES Modes of Operation, December 1980.

[78] National Institute of Standards and Technology. FIPS-186-2: Digital Signature Standard (DSS), January 2000.

[79] National Institute of Standards and Technology. FIPS-197: Advanced Encryption Standard, November 2001.

[80] National Institute of Standards and Technology. Special Publication 800-38A: Approved Block Cipher Modes of Operation, December 2001.

[81] National Institute of Standards and Technology. Special Publication 800-56A - Prime Curve Examples, March 2007. `http://csrc.nist.gov/groups/ST/toolkit/documents/Examples/KS_ECC_Prime.pdf`.

[82] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *ACMQueue*, 6(2), April 2008.

[83] Nvidia. CUDA - Compute Unified Device Architecture. `http://developer.nvidia.com/object/cuda.html`.

[84] Nvidia. CUDA Occupancy Calculator. `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`.

[85] Nvidia. Nvidia NV4X Family: The GeForce 6 Series, 2004. `http://www.nvidia.com/page/geforce6.html`.

[86] Nvidia. PBO Texture Performance Tool, 2004. `http://developer.download.nvidia.com/SDK/9.5/Samples/samples.html`.

[87] Nvidia. GeForce 7 Series: Nvidia GPU Programming Guide, Version 2.5.0, 2005. `http://developer.nvidia.com/object/gpu_programming_guide.html`.

[88] Nvidia. Nvidia G7X Family: The GeForce 7 Series, 2005. `http://www.nvidia.com/page/geforce7.html`.

[89] Nvidia. Technical Brief: Fast Texture Downloads and Readbacks using Pixel Buffer Objects in OpenGL. TB-02011-001_v01, August 2005.
http://developer.nvidia.com/object/fast_texture_transfers.html.

[90] Nvidia. Nvidia G8X Family: The GeForce 8 Series, 2006.
http://www.nvidia.com/page/geforce8.html.

[91] Nvidia. Technical Brief: Microsoft DirectX 10: The Next-Generation Graphics API. TB-02820-001_v01, November 2006.
http://www.nvidia.com/page/8800_tech_briefs.html.

[92] Nvidia. Technical Brief: NVIDIA GeForce 8800 GPU Architecture Overview. TB-02787-001_v01, November 2006.
http://www.nvidia.com/page/8800_tech_briefs.html.

[93] Nvidia. Nvidia CUDA: Programming Guide, Version 2.1, December 2008.
http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.1.pdf.

[94] Nvidia. Nvidia G200 Family: The GeForce 200 Series, 2008.
http://www.nvidia.com/object/geforce_family.html.

[95] OCF-Linux Project Homepage.
http://ocf-linux.sourceforge.net/.

[96] M. Olano and A. Lastra. A Shading Language on Graphics Hardware: The PixelFlow Shading System. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 159–168, Orlando, Florida, USA, July 1998. ACM.

[97] OpenGL ARB, D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2.1.* Addison-Wesley Professional, sixth edition, 2007.

[98] OpenGL ARB, Framebuffer Object Extension: ARB_frambuffer_object, August 2008.

[99] OpenGL ARB, Pixel Buffer Object Extension: ARB_pixel_buffer_object, December 2004.

[100] OpenSSL Open Source Project.
http://www.openssl.org/.

[101] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum: The International Journal of the Eurographics Association*, 26(1):80–113, March 2007.

[102] PCI-SIG. PCI Express Specifications, 2009.
`http://www.pcisig.com/specifications/`.

[103] N. Pilkington and B. Irwin. A Canonical Implementation of the Advanced Encryption Standard on the Graphics Processing Unit. In *"Research in Progress Papers" Section of the ISSA 2008 Innovative Minds Conference*, Johannesburg, South Africa, July 2008.

[104] K. Posch and R. Posch. Base Extension Using a Convolution Sum in Residue Number Systems. *Computing*, 50(2):93–104, 1993.

[105] K. Posch and R. Posch. Modulo Reduction in Residues Numbers Systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):449–454, May 1995.

[106] J-J. Quisquater and C. Couvreur. Fast Decipherment Algorithm for RSA Public-Key Cryptosystem. *Electronics Letters*, 18(21):905–907, 1982.

[107] A. Rege. Nvidia: Shader Model 3.0, 2004.
`ftp://download.nvidia.com/developer/presentations/2004/GPU_Jackpot/Shader_Model_3.pdf`.

[108] M. Riffa, X. Bonnairea, and B. Neveub. A Revision of Recent Approaches for Two-dimensional Strip-packing Problems. *Engineering Applications of Artificial Intelligence*, 22(4-5):823–827, June 2009.

[109] R. Rivest. The RC4 Encryption Algorithm. RSA Data Security Inc., March 1987.

[110] R. Rivest. The MD5 Message-Digest Algorithm, April 1992. RFC 1321.

[111] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[112] U. Rosenberg. Using Graphic Processing Unit in Block Cipher Calculations. Master's thesis, University of Tartu, 2007.

[113] R. Rost. *OpenGL Shading Language*. Addison-Wesley Professional, second edition, 2006.

[114] RSA Laboratories. PKCS #1 v2.1: RSA Cryptography Standard, June 2002.

[115] RSA Laboratories. PKCS #5 v2.1: Password-Based Cryptography Standard, October 2006.

[116] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition*. John Wiley & Sons, Inc., 1996.

[117] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. *ACM Transactions on Graphics*, 27(3):1–15, August 2008.

[118] M. Seshadrinathan and K. Dempski. Implementation of Advanced Encryption Standard for encryption and decryption of images and text on a GPU. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–6, Anchorage, AK, USA, June 2008. IEEE Publishing.

[119] P. Shenoy and R. Kumaresan. Fast Base Extension Using a Redundant Modulus in RNS. *IEEE Transactions on Computers*, 38(2):292–297, 1989.

[120] N. Szabo and R. Tanaka. *Residue Arithmetic and its Applications to Computer Technology*. McGraw-Hill, 1967.

[121] R. Szerwinski and T. Güneysu. Exploiting the Power of GPUs for Asymmetric Cryptography. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 79–99, Washington DC, USA, August 2008. Springer.

[122] The Snort Open Source Network Intrusion Prevention and Detection System.
`http://www.snort.com/`.

[123] S. Tzeng and L.-Y. Wei. Parallel white noise generation on a GPU via cryptographic hash. In *Symposium on Interactive 3D Graphics and Games*, pages 79–87, Redwood City, CA, USA, February 2008. ACM.

[124] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. Markatos, and S. Ioannidis. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *International Symposium On Recent Advances In Intrusion Detection (RAID)*, pages 116–134, Boston, MA, USA, September 2008. Springer.

[125] S. Venkatasubramanian. The Graphics Card as a Stream Computer, June 2003.

[126] D. Whiting, N. Ferguson, and R. Housley. Counter with CBC-MAC (CCM), 2002.

[127] T. Wollinger, J. Guajardo, and C. Paar. Security on FPGAs: State-of-the-art implementations and attacks. *ACM Transactions on Embedded Computer Systems*, 3(3):534–574, 2004.

[128] T. Yamanouchi. AES Encryption and Decryption on the GPU. *GPU Gems 3*, pages 785–804, 2007.

[129] J. Yang and J. Goodman. Symmetric Key Cryptography on Modern Graphics Hardware. In *ASIACRYPT*, pages 249–264, Kuching, Sarawak, Malaysia, December 2007. Springer.

[130] Y. Yeom, Y. Cho, and M. Yung. High-Speed Implementations of Block Cipher ARIA Using Graphics Processing Units. In *International Conference on Multimedia and Ubiquitous Engineering*, pages 271–275, Busan, Korea, April 2008. IEEE Computer Society.

Appendix A

OCF Extensions

A.1 New Memory Management Interface

The following lists the interface extension for the Crypto and Cryptodev layers within the OCF to support the new memory management system.

A.1.1 Crypto Layer Interface

`crypto_alloc()`: pass in the size and optionally the device ID for memory allocation. Returns a kernelspace pointer and the device that performed the allocation. If the allocation fails null is returned.

`crypto_free()`: pass in a pointer returned by crypto_alloc().

`crypto_translate()`: pass in a pointer returned by crypto_alloc(). Returns a device space pointer if a mapping is found.

A.1.2 Cryptodev Layer ioctl Interface

`CIOCALLOC`: this takes in an allocation request structure as a parameter, which specifies the requested buffer size and suggested device ID. The same structure is used to return the userspace pointer and actual device used for the allocation.

A.2 Gpucrypt ioctl Interface

The following ioctls are used to communicate with the Gpucrypt device via `/dev/gpucrypt`. These are used by the Gpucryptd userspace daemon to co-operate with the OCF to complete cryptographic requests.

`GPU_REGISTER_*_REQ_BUF`: this is a series of ioctls which Gpucryptd driver executes on startup to register shared request queues for each type of OCF request supported by the GPU driver. * refers to `ALLOC`, `FREE`, `KPROCESS` (asymmetric request processing) and `PROCESS` (symmetric request processing).

`GPU_READY`: after the request buffers are created, shared and initialised and all state is ready for operation, the Gpucryptd daemon registers itself as ready for work with the Gpucrypt driver. The driver, on receipt of this ioctl, issues a register command to the OCF to inform it that it is ready to start receiving requests.

`GPU_WAIT_FOR_WORK`: the Gpucryptd daemon process cycles through the request queues, continually processing any available work. When there are no more requests to process, rather than continually scanning it calls the Gpucrypt driver to wait for work using this ioctl. On receipt of this request the Gpucrypt sleeps the calling process on a kernel wait queue. When work is subsequently received from the OCF, the Gpucryptd is woken by calling wake on this wait queue, thus releasing Gpucryptd to finish the rest of the ioctl and return to userspace to process the new work.

`GPU_RETURN_*_REQ`: on finishing of a request, the Gpucryptd daemon uses this series of ioctls to deliver the work back to the OCF. This ioctl calls the OCF crypto_done() function, which can either process the registered callback function for the request immediately or allow the OCF return queue kernel thread to do so later. An application that has a long callback function may configure the cryptographic request to not execute an immediate callback as the callback is normally run in interrupt context. However, when the GPU is used, crypto_done is called from within process context, specifically the Gpucryptd context, and thus long callback functions are less problematic and thus the use of the separate return queue kernel thread can be avoided. * refers to `ALLOC`, `FREE`, `KPROCESS` (asymmetric request processing) and `PROCESS` (symmetric request processing).

`GPU_SHUTDOWN`: this ioctl is called on shutdown, which in turn unregisters the Gpucrypt driver from the OCF.

Appendix B

Hardware

B.1 Processor Details

Table B.1 lists the factory gate prices and release dates of the CPUs used for comparisons against the GPU following the methodology stated in Chapter 4. The Intel Pentium 4 processor used for comparison in Chapter 4 was referenced from the Crypto++ project website. The exact processor model is not listed on the website, though it is quoted as a Prescott 2.93 GHz. CPUs matching these details have release dates ranging from 12/2004 to 09/2005. Also, we could not source the price for any of these matching CPUs. To give an estimation, we list a Pentium 4 Prescott chip for which a price could be sourced.

B.2 Memory Bandwidth

Tables B.2 and B.3 highlight the difference in bandwidth between system bus transfer rates and on-card device memory transfer rates of GPUs and system bus types used in this thesis.

B.3 GeForce 8800GTX Memory

Table B.4 lists the physical memory types and sizes available on the DirectX 10 GPU used in this thesis, the Nvidia GeForce 8800GTX.

Processor	Clock	Factory Price (1000s)	Release Date
AMD Athlon X2 3800+	2.0 GHz	\$299	May 2006
AMD Athlon 64 3700+	2.2 GHz	\$272	May 2005
Intel Core 2 Duo E6300	1.86 GHz	\$224	Aug 2006
Intel Pentium 4 630	3.0 GHz	\$224	Mar 2005

Table B.1: List of CPUs used for comparisons.

GeForce 6600GT	14.4 GB/s
GeForce 7900GT	42.2 GB/s
GeForce 8800GTX	86.4 GB/s

Table B.2: Bandwidth rates between relevant GPUs and on-card device memory.

AGP 8x	2 GB/s
PCIe v1.x x16	4 GB/s

Table B.3: Bandwidth rates of relevant system bus types.

Type	Location	Size
Registers	On-chip	8912 32-bit registers $\times$ 16 SMs
Shared Memory	On-chip	16KB $\times$ 16 SMs
Constant Cache	On-chip	8KB $\times$ 16 SMs
Texture Cache	On-chip	8KB $\times$ 16 SMs
Device Memory	On-card	768MB

Table B.4: Physical memory types and sizes for the GeForce 8800GTX

Appendix C

Operating System Terms and Functions

Kernelspace **Userspace**	Most operating systems operate in two different modes, kernel and user mode. These modes are used to give a basic privilege separation, where processes running in kernel mode have direct access to the underlying hardware, and processes running in user mode only have access to these resources indirectly. A kernel refers to the processes that make up the core services of an OS such as process management, memory management, hardware interaction. The kernel processes run in kernel mode. All other processes, such as normal applications, run in user mode and can only access memory, create other processes, access the hardware by requesting these services from the kernel. The kernel has exclusive access to a reserved portion of virtual memory, kernel memory, which all processes running in kernel mode share. Processes running in user mode have their own memory address space allocated from a separate portion of virtual memory, user memory. *Kernelspace* is an informal term commonly attached to processes running in kernel mode and the memory that is reserved for its use. *Userpsace* is used to refer to user mode processes and their accessible memory.

Mode Switch	A switch between user and kernel mode. For example, this can occur when a userspace process requests a kernel service via a system call. There is an overhead associated with a mode switch, in that the CPU state must be backed up and subsequently restored to support seamless execution of the userspace process.
VMA	Virtual Memory Area: this is a kernel structure, which is used to denote a contiguous region of virtual memory assigned to a process.
Page	A basic unit of memory used by a memory management system, which can exist within RAM or in secondary storage.
ioctl	I/O Control. Userspace access mechanism to Kernelspace.
`kmalloc()`	Kernelspace function that allocates contiguous memory using the SLAB allocator, which is designed to optimise sub-page sized allocations.
`get_free_pages()`	Kernelspace function that allocates contiguous memory for processes in kernel mode in multiples of the system page size.
`get_user_pages()`	Kernelspace function that returns the underlying pages given a userspace process and pointer to memory.
`mmap()`	Userspace function that maps a portion of a file (or existing allocated memory in the case of shared memory), into the virtual address space of the calling process.
`munmap()`	Userspace function that releases the mapping of a userspace pointer to the underlying mapped object.
`vmap()`	Kernelspace function that maps pages into kernelspace memory.
`fork()`	Userspace function that spawns a child process.
`vm_open()`	Kernelspace function, attached to a VMA, that is called by the kernel when a new reference is created to said VMA.
`vm_close()`	Kernelspace function, attached to a VMA, that is called by the kernel when an existing reference is removed to said VMA

`copy_from_user()`	Kernelspace function that copies data from userspace memory to kernelspace memory.
`copy_to_user()`	Kernelspace function that copies data from kernelspace memory to userspace memory.

Appendix D

System Specifications

The following is a list of system specifications that were used to generate the experimental results presented within the thesis.

Operating System	Fedora Core 4, 32-bit
OpenGL Version	2.1
Cg Version	1.4.0.4
Nvidia Driver Version	1.0-8762
System CPU	1 GHz AMD Athlon 800
System Memory	512 MB
Graphics Card	GeForce 6600GT
Graphics Bus	AGP 8x
Graphics Card Memory	128 MB

Table D.1: System 1

Operating System	Fedora Core 4, 32-bit
OpenGL Version	2.1
Cg Version	1.4.0.4
Nvidia Driver Version	1.0-8762
System CPU	2.2 GHz AMD Athlon 64 3700+
System Memory	2 GB
Graphics Card	GeForce 7900GT
Graphics Bus	PCIe x16
Graphics Card Memory	256 MB

Table D.2: System 2

Operating System	Fedora Core 9, 32-bit
CUDA Version	2.0
Nvidia Driver Version	1.0-9755
System CPU	2.4 GHz AMD Athlon 64 X2 3800+
System Memory	2 GB
Graphics Card	GeForce 8800GTX
Graphics Bus	PCIe x16
Graphics Card Memory	768 MB

Table D.3: System 3