Code Generation for Hardware Accelerated AES

Raymond Manley*, Paul Magrath, David Gregg
School of Computer Science and Statistics
Trinity College Dublin
Dublin, Ireland
manleyr@cs.tcd.ie, magrathp@tcd.ie, dgregg@cs.tcd.ie

***Abstract*—Data must be encrypted if it is to remain confidential when sent over computer networks. Encryption solves many problems involving invasion of privacy, identity theft, fraud, and data theft. However for encryption to be widely used, it must be fast. The problem is so important that new Intel processors provide hardware support for encryption. These instructions implement key stages of the Advanced Encryption Standard (AES), allowing encryption to be completed more quickly and using less power. The AES algorithm consists of several 'rounds' of encryption, each of which involves a relatively complicated computation. This new hardware support allows an entire round to be implemented with just a single instruction.**

An implementation of the AES algorithm using these instructions contains several code sections that can be fine tuned for optimal performance. However, these optimizations are usually done by hand, which can be a lengthy, labour intensive process. We present a system that can generate billions of variants of the AES encryption code to find the best solution for a particular microarchitecture. We apply both common loop optimizations and ones specific to AES. We evaluate the generated code on hardware with built-in AES support using both selective-brute force and guided searches. Our generator achieves significant speedups over a straightforward implementation of the code.

***Keywords*-AES; encryption; code generation**

I. Introduction

There is no shortage of sensitive information that is transmitted on a daily basis. Medical records, military communications, and credit card numbers are all examples of sensitive information that we do not want to freely share with others. To prevent unintended parties from accessing information, encryption must be used. Using encryption can be costly in both time and power requirements.

The Advanced Encryption Standard (AES) is a common and widely used method of encrypting data. The AES algorithm contains a computationally expensive loop which requires large amounts of CPU clock time that provides many avenues for optimization [1]. This paper presents a code generator that creates variants of the AES encryption loop.

The use of code generators to find which combination of optimizations yield a good result is an established technique in optimizing for modern architectures [2]. Code generators, such as Spiral [3] and FFTW [4], have been very successfully applied to their respective domains. Cost and time are factors that result from maintaining hand-tuned assembly are motivating factors when building code generators. A code generator can tune itself to the architecture it is running on to find the best combination of optimizations for that architecture, while remaining readable and maintainable as it can be written in a high level language, such as C++.

AES encryption costs are greatly reduced with Intel's Westmere microarchitecture [5] and its instruction set extension [6] (AES-NI) implements key stages of the AES algorithm. Our code generator optimizes the AES algorithm itself, regardless of architecture, by creating billions of different implementations while maintaining correctness. Our system is a valuable resource to find an optimized AES implementation on a given target architecture. Our contributions are as follows:

- We show our generator finds optimized variants with an average speedup of 1.43x over all the baselines.
- We show that a simple generator can find a good variation of the code without any specific knowledge of the target microarchitecture.
- We offer a viable alternative to maintaining multiple versions of hand-optimized code.
- We show simulated annealing is an effective and quick method to find a solution in a wide search space.

The remainder of this paper is organized as follows: Section II provides background on AES and simulated annealing. Our optimization techniques are described in Section III. Discussion of results are featured in Section IV, while our conclusions are offered in Section V.

II. Background

A. Encryption

AES is one of the most popular algorithms used in symmetric encryption. Originally published as Rijndael [7], AES was adopted as a standard by the U.S. government in November 2001 [8]. The standard comprises three block ciphers: AES-128, AES-192 and AES-256 that each have 10, 12, and 14 keys, respectively. AES is a block cipher which encodes incoming 128-bit blocks of plaintext with a secret key to produce the ciphertext.

*Funded by the Irish Research Council for Science, Engineering and Technology (IRCSET) in collaboration with Intel Ireland Ltd under the National Development Plan.

Listing 1. Psuedocode our implementation of simulated annealing.

```
anneal()
  c0 = cost(default_arguments)
  t = start_temperature
  for i in range (0, iterations) :
    for j in range (0, cooling_steps) :
      new_args = random_arg * weight
      c1 = cost(new_args)
      delta = (c1 - c0) / c0
      if delta < 0
          c0 = c1
      else if (e^(-delta/(k*t) >= random[0,1))
        c0 = c1
    t = t * reduce
```

Listing 2. Psuedocode for the AES CTR encryption loop.

```
aes_encrypt(*source, *dest, nonce, *key, nKeys, blocks)
  for i in range (0, blocks) :
        result = nonce + i
        result = encrypt_initial(result, key[0])
        for j in range (1, nKeys) :
            result = encrypt_round(result, key[j]))
        result = encrypt_final(result, key[j])
        result = _mm_xor_si128(result, source[i])
        dest[i] = result
```

To encrypt data that exceeds the block size, a mode of operation must be used. Our generator produces code for both counter (CTR) and cipher-block chaining (CBC) modes. CTR mode is a stream cipher which encrypts a counter value and is `xor`'d with plaintext, making it ripe for parallelizable optimizations. CBC mode has a cyclic dependency that occurs because the result of each encrypted block is then used as the seed to encode the next block.

To facilitate these operations in hardware, Intel introduced support for AES through AES-NI [6]. Included in AES-NI are six instructions for symmetric encryption/decryption used by AES. Natively supporting AES instructions provides both performance and security benefits.

B. Simulated Annealing

Simulated annealing is a heuristic search algorithm that employs probabilistic reasoning to traverse the search space [9]. In the beginning, the search can be quite erratic in direction, but always testing a neighbouring solution. Bad solutions are accepted in early stages to prevent the search from being stuck in local minima. As the search goes on, the probability of accepting bad solutions reduces through a temperature variable that 'cools down'. The algorithm suggests that it has found a good solution and is more likely to test solutions closer to the good solution when the temperature is sufficiently cool.

The goal of using simulated annealing in conjunction with our generated code is to use a guided search to reduce the time it takes to find a solution. Exhaustively trying all variants takes days of compilation time alone. In Listing 1, we outline the pseudo-code we use to implement simulated annealing. We modify the classic algorithm slightly to keep track of a global best solution.

III. AES Code Generation

Our generator creates C source code variations of the AES encryption loop in Listing 2. Our generated C code uses AES-NI compiler intrinsics when compiling for hardware with AES support. These functions can be substituted on non-AES hardware. Optimizations are turned on and off by a set of flags that traverse a wide search space. These variants are further optimized at low-level by gcc or icc compilers.

The generator can test several implementations, though our results in Section IV showcase 128 and 256-bit versions of both CTR and CBC modes and different optimizations exist for each.

The search space for optimizations is very large. In CTR mode, interleaving combinations number 221,184 variants which is expanded when multiplying by the number of localkey variations (2^{15}), yielding a search space of nearly four billion possible combinations. In CBC mode, there are over $(2.6*10^6)*2^{15}$ possible variants. A full set of generator options in both modes are:

- **Counter (CTR)** localkeys (2^{15}), interleave factor (*IF*) (0-15), interleave distance (0-17), software pipelining initiation interval (0-17), software prefetching to cache OR register, prefetch source (1-[*IF*]), prefetch distance (0-[*IF*-1]), restrict pointers, and streaming store
- **Cipher Block Chaining (CBC)** streams (1-4), localkeys (2^{15}), unroll (*UD*) (0-15), interleave, interleave distance (0-17), software prefetching to cache OR register, prefetch source (2-[*UD*]), prefetch distance (0-[*UD*]), restrict pointers, streaming store, and `xor`

Optimization options that make small differences individually can make large improvements collectively. Options like streaming store that writes to memory without polluting the cache, using C *restrict pointers* that provide the compiler with pointer aliasing information can be turned on and off. Specific to CBC mode, a modification to the first two `xor` instructions can also be enabled.

The generator also takes in a bit-vector parameter to enable individual keys to be held in registers. This can be done in both modes along with software prefetching and preloading. Prefetching uses Intel prefetch instructions to move data into the cache. Preloading loads plaintext early, to help cover the cost of cache misses.

Interleaving is applied to both CTR and CBC modes differently. In CTR, interleaving consists of unrolling the outer AES loop. The code from each unrolled iteration can be interleaved as there is no dependency between them. This allows a single key value to be used more frequently in succession. In CBC, we interleave individual encryption streams instead of successive iterations per stream. This minimizes the cyclic dependency on each stream from one iteration to the next. The generator can also adjust the distance between the interleaved sections.

Exploitable in CTR mode, **software pipelining** divides

several iterations into distinct parts (see Listing 3). The size of the parts depend on the initiation interval (*ii*). Inside the loop, only one full set of AES iteration instructions exist while operating on several different blocks in parallel. Since the same keys and different plaintexts are used in each loop iteration, there are smaller dependencies between rounds.

Listing 3. Software pipelining with an initiation interval of 2

```
//pre software pipelining code
for(i = 0; i < blocks; i++){  //software pipelining loop
 sp0_result = _mm_xor_si128(sp0_result, plaintext[i]);
 sp1_result = encrypt_round(sp1_result, key[13]);
 sp2_result = encrypt_round(sp2_result, key[11]);
 sp3_result = encrypt_round(sp3_result, key[9]);
 sp4_result = encrypt_round(sp4_result, key[7]);
 sp5_result = encrypt_round(sp5_result, key[5]);
 sp6_result = encrypt_round(sp6_result, key[3]);
 sp7_result = encrypt_round(sp7_result, key[1]);
 sp8_result = _mm_shuffle_epi8(counter_block,BSWAP_EPI64);
 counter_block = _mm_add_epi64(counter_block,one);
 ciphertext[i] = sp0_result;
 sp1_result = encrypt_final(sp1_result, key[14]);
 sp2_result = encrypt_round(sp2_result, key[12]);
 sp3_result = encrypt_round(sp3_result, key[10]);
 sp4_result = encrypt_round(sp4_result, key[8]);
 sp5_result = encrypt_round(sp5_result, key[6]);
 sp6_result = encrypt_round(sp6_result, key[4]);
 sp7_result = encrypt_round(sp7_result, key[2]);
 sp8_result = encrypt_initial(sp8_result, key[0]);
 //variable copy cleanup
}
//post software pipelining code
```

IV. Results

The experimental results describe how AES 256, in both CTR and CBC modes, perform on a Core i5 (with AES-NI) running at 3.2ghz with 64-bit Ubuntu. The experiment encrypts 2048 16-byte blocks, or 32KB, of data. In CBC mode, 2048 blocks for each stream are encrypted. All variants are compiled by icc with `-O3`. The performance values are captured by surrounding the encryption loop with reads of the time stamp counter (RDTSC).

A. Selective-Exhaustive searching

AES 256 CTR performance ranged from 1.73 cycles/byte (C/B) to over 6.0 C/B on a subset of possible variants. We find that mid-range (6-12) interleaving values tend to run best. Software pipelining *ii* values of 2 and 3 are also good. Small *ii* values increase instruction level parallelism (ILP)—as does interleaving—but also increases register pressure. Figure 1 shows performance of software pipelining. The graph trends downward with better performance from smaller *ii* values. Lower *ii* values mean higher number of iterations that have little to no dependency inside the encryption loop and maximizes the ability of exploiting ILP.

Nearly all of the fastest 200 variants use 10 or more localkeys. Among this selection, prefetching upcoming sourcetext seems to also be beneficial. Prefetching sourcetext three iterations in advance tends to yield the best results. Prefetching has little effect on performance enabled on its own, but can improve running times by several percentage points when used with other optimizations.

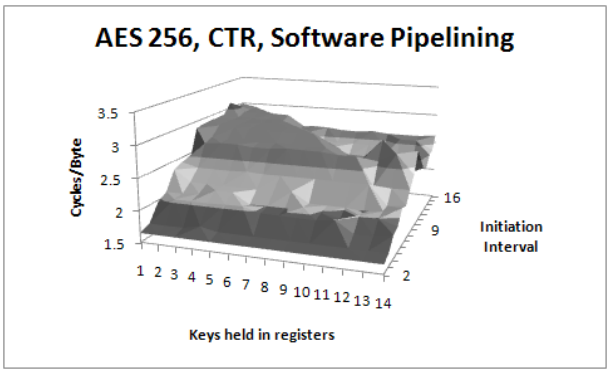

Figure 1. Performance in cycles/byte using software pipelining vs. keys in registers.

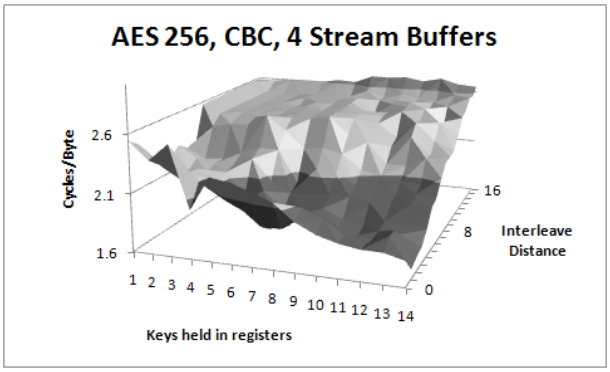

Figure 2. Performance graphs 4 stream CBC mode AES results on interleave distance vs. key values held in registers.

In CBC mode, the fastest running times when generating AES 256 were with 4 stream buffers (CBC-4) and its fastest variant clocked at 1.71 C/B. Figure 2's peaks and valleys show much more texture compared to those in the CTR graph. In Figure 2, the number of localkeys affect performance much more when interleaving 4 stream buffers in CBC mode. As each stream would have their own set of keys, there are far more keys than registers available in 64-bit mode. This pressure will affect how tight interleaving each stream can optimally be generated. In CBC mode, 3 or 4 streams with interleave distances higher than 7 show a significant decrease in performance as the generated code has large amounts of data dependency.

Optimizations like streaming store, restrict and prefetching/preloading can improve a variant by a few percent individually but generally improve performance when used collectively. This shows us that a generator is useful for finding which set of these minor optimizations will improve performance. The combination of these smaller optimizations with the larger impact ones like interleaving, software pipelining, and keys in registers give us a significant improvement over our baselines.

	AES 128 (32K buffer)			AES 256 (32K buffer)		
Mode	**Baseline**	**Best**	**Speedup**	**Baseline**	**Best**	**Speedup**
CTR	1.85	1.17	1.58x	2.79	1.62	1.72x
CTR-SP	1.85	1.23	1.50x	2.79	1.62	1.72x
CBC-1	4.67	3.75	1.25x	6.06	5.14	1.18x
CBC-2	2.02	1.88	1.07x	2.72	2.57	1.06x
CBC-3	1.71	1.26	1.36x	2.67	1.71	1.56x
CBC-4	1.76	1.17	1.50x	2.69	1.65	1.63x

Figure 3. Results of best found variants in cycles/byte. Software pipelining (CTR-SP) is measured seperately from other CTR results. CBC-X modes are measured with 1-4 streams.

B. Simulated Annealing

The selective-exhaustive experiments we ran in CTR mode included over 40,000 variants. This is a very small subset of possible combinations which takes several hours to complete. Our simulated annealing implementation tries 1500 solutions and completes in under an hour. Its initial solution applies no optimizations. When flags are changed, greater weights are given to the number of localkeys and the interleaving factor. Lower weights are given to on/off options. Annealing in CTR mode yielded a top performance of 1.62 C/B with software pipelining and 1.64 C/B using all other possible optimizations as seen in Figure 3.

The selected-exhaustive experiments we ran in CBC was again only a subset of billions of possible combinations and annealing is used to reduce search time. We tune the argument weights for CBC-specific optimizations. Annealing found variants for CBC with 1 to 4 stream buffers all achieved speedups over their baselines (Figure 3). Unoptimized CBC-2 has a speedup over unoptimized CBC-1 of 2.23x. However, our optimizations to CBC-2 only increase perfromance by 6-7%. We found better speedups with CBC-3 and CBC-4 at 1.56x and 1.63x, respectively.

In both CTR and CBC modes, annealing was able to traverse much more of the search space to find variants than ran faster than the selective-exhaustive searches found. Running exhaustive searches on even small subsets took hours to complete, whereas simulated annealing took under an hour to complete and came up with a better result.

V. Conclusion

In this paper, we presented a code generator that creates optimized AES implementations. Our system can generate billions of versions the AES encryption loop. Searching even small subsets of the possible combinations yields an average speedup of 1.43x over all baselines. The generator applies both generalized and specific-to-AES optimizations as mentioned in detail in Section III of this paper.

To evaluate our system, we evaluated the generated code on hardware with native AES support. Architectural properties can influence the search for the best optimizations for several AES modes with varying keysizes. The generator can search this space without any insight into hardware specifics, such as cache sizes, number of registers, or even the instruction set available to implement AES.

In order to implement a version which runs efficiently, AES is often optimized by hand. The generator is a viable alternative to maintaining hand-optimized code when new microarchitectures are introduced—saving both time and money. To save additional time, we implemented simulated annealing as a guided search heuristic. This heuristic performed well in both CTR and CBC modes. Exploring only a small fraction of the search space of only 1500 variants, the algorithm found variants that performed better than ones in our selective-exhaustive searches.

Using a code generator to find optimized implementations is a good system to find which version runs fast on a target platform in order to perform an already very common everyday task—AES encryption.

Acknowledgment

We would like to thank Mike O'Hanlon at Intel Shannon for his help, input, and facilitating access to test hardware.

References

[1] E. Käsper and P. Schwabe, "Faster and Timing-Attack Resistant AES-GCM," in *Cryptographic Hardware and Embedded Systems - CHES 2009*, ser. Lecture Notes in Computer Science 5747. Springer Verlag, 2009, pp. 1–17.

[2] H. Massalin, "Superoptimizer: A Look at the Smallest Program," *SIGPLAN Not.*, vol. 22, no. 10, pp. 122–126, 1987.

[3] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code Generation for DSP Transforms," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232– 275, 2005.

[4] M. Frigo, Steven, and G. Johnson, "The Design and Implementation of FFTW3," in *Proceedings of the IEEE*, 2005, pp. 216–231.

[5] R. Benadjila, O. Billet, S. Gueron, and M. J. B. Robshaw, "The Intel AES Instructions Set and the SHA-3 Candidates," in *Advances in Cryptology - ASIACRYPT 2009*, ser. Lecture Notes in Computer Science 5912. Springer Verlag, 2009, pp. 162–178.

[6] S. Gueron, "Intel's New AES Instructions for Enhanced Performance and Security," in *Fast Software Encryption - FSE 2009*, ser. Lecture Notes in Computer Science 5665. Springer Verlag, 2009, pp. 51–66.

[7] J. Daemen and V. Rijmen, "The Block Cipher Rijndael," in *CARDIS '98: Proceedings of the The International Conference on Smart Card Research and Applications*. London, UK: Springer-Verlag, 2000, pp. 277–284.

[8] "Announcing the Advanced Encryption Standard (AES)," Federal Information Processing, 2001.

[9] S. S. Skiena, *The Algorithm Design Manual*. New York, NY, USA: Springer-Verlag New York, Inc., 1998.