Fast Implementations of AES on Various Platforms

Joppe W. Bos[1], Dag Arne Osvik[1], and Deian Stefan[2⋆]

[1] EPFL IC IIF LACAL, Station 14, CH-1015 Lausanne, Switzerland
{joppe.bos, dagarne.osvik}@epfl.ch
[2] Dept. of Electrical Engineering, The Cooper Union, NY 10003, New York, USA
stefan@cooper.edu

Abstract. This paper presents new software speed records for encryption and decryption using the block cipher AES-128 for different architectures. Target platforms are 8-bit AVR microcontrollers, NVIDIA graphics processing units (GPUs) and the Cell broadband engine. The new AVR implementation requires 124.6 and 181.3 cycles per byte for encryption and decryption with a code size of less than two kilobyte. Compared to the previous AVR records for encryption our code is 38 percent smaller and 1.24 times faster. The byte-sliced implementation for the synergistic processing elements of the Cell architecture achieves speed of 11.7 and 14.4 cycles per byte for encryption and decryption. Similarly, our fastest GPU implementation, running on the GTX 295 and handling many input streams in parallel, delivers throughputs of 0.17 and 0.19 cycles per byte for encryption and decryption respectively. Furthermore, this is the first AES implementation for the GPU which implements both encryption and decryption.

Key words: AES, AVR, Cell, GPU

1 Introduction

In 2001, as the outcome of a public competition, Rijndael was announced as the Advanced Encryption Standard (AES) by the US National Institute of Standards and Technology (NIST). Since then, it has become one of the most extensively used encryption primitives for various applications. Besides its well-regarded security properties[3], AES is extremely efficient on many different platforms, ranging from 8-bit microcontrollers to 64-bit processors to FPGAs; the efficiency was a crucial metric in making Rijndael an encryption standard. Much work on efficient and secure implementation of AES has been done, including the evaluation of its performance on different architectures; this work further evaluates such efficient AES implementations.

⋆ This work was done when the third author was visiting EPFL.

[3] The only attack on the full AES is applicable in the related key scenario to the 192-bit [5] and 256-bit key versions [5,6].

We present new, high-speed and small codesize, software implementations of AES for 8-bit AVR microcontrollers, the Cell Broadband Engine architecture (Cell) and NVIDIA Graphics Processing Units (GPUs). To the best of our knowledge, our results set new records on these platforms.

It is expected that the use of lightweight devices, i.e., low-end smart cards and radio frequency identification (RFID) tags, in electronic commerce and identification will grow rapidly in the near future. The passive RFID tag market is expected to reach up to US$ 486M by 2013 [10], and AES has already attracted significant attention due to its capabilities for such devices. This work further investigates the performance of AES on such devices; specifically, 8-bit microcontrollers.

The other target platforms, the Cell and the GPU, are chosen because of their ability to process many streams simultaneously, using single instruction, multiple data (SIMD) and single instruction, multiple threads (SIMT) techniques respectively. Due to the low prices and wide availability of these devices it is interesting to evaluate their performance as cryptologic accelerators.

The paper is organized as follows. Section 2 briefly recalls the design of AES. In Section 3 our target platforms are described. Section 4 describes the techniques used and decisions made when porting AES to the different architectures. In Section 5 we present our results and a comparison is made to other reported results in literature. Finally, we conclude in Section 6.

2 A Brief Introduction to AES

The AES is a fixed block length version of the Rijndael block cipher [8,14], with support for 128-, 192-, and 256-bit keys. The cipher operates on an internal state of 128 bits, which is initially set to the plaintext block, and after transformations, becomes the output ciphertext block. The state is organized in a 4×4 array of 8-bit bytes, which is transformed according to a round function N_r times. The number of rounds is $N_r = 10$ for 128-bit keys, $N_r = 12$ for 192-bit keys, and $N_r = 14$ for 256-bit keys. In order to encrypt, the state is first initialized, then the first 128-bits of the key are `xor`ed into the state, after which the state is modified $N_r - 1$ times according to the round function, followed by the slightly different final round.

The round function consists of four steps: `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey` (except for the final round which omits the `MixColumns` step). Each step operates on the state, at each round r, as follows:

1. `SubBytes`: substitutes every entry (byte) of the state with an S-box entry,
2. `ShiftRows`: cyclically left shifts every row i of the state matrix by $i, 0 \leq i \leq 3$,
3. `MixColumns`: multiplies each column of the state, taken as a polynomial of degree below 4 with coefficient in $\mathbb{F}_{2^8}$, by a fixed polynomial modulo $x^4 + 1$,
4. `AddRoundKey`: `xor`s the r-th round key into the state.

Each transformation has an inverse from which decryption follows in a straightforward way by reversing the steps in each round: `AddRoundKey` (inverse of itself), `InvMixColumns`, `InvShiftRows`, and `InvSubBytes`.

The key expansion into the N_r 128-bit round keys is accomplished using a key scheduling algorithm, the details of which can be found in [14] and [8]. The design of the key schedule allows for the full expansion to precede the round transformations, which is advantageous if multiple blocks are encrypted using the same key, while also providing the option for on-the-fly key generation, a method that proves useful in memory constrained environments such as microcontrollers.

For 32-bit (and greater word length) processors, in [8] Daemen and Rijmen detail a fast implementation method that combines the `SubBytes`, `ShiftRows`, and `MixColumns` transformations into four 256-entry (each entry is 4 bytes) look-up tables, T_i, $0 \leq i \leq 3$. Following [8], the "T-table" approach reduces the round transformations to updating the j-th column according to:

$$\left[s'_{0,j}, s'_{1,j}, s'_{2,j}, s'_{3,j}\right]^{\mathrm{T}} = \bigoplus_{i=0}^{3} T_i[s_{i,j+C_i}], \text{ for } 0 \leq j \leq 3, \tag{1}$$

where $s_{j,k}$ is the byte in j-th row and k-th column of the state, and C_i is a constant equivalently doing the `ShiftRows` in-place. After the columns are updated, the remaining transformation is `AddRoundKey` (which is a single 4-byte look-up and `xor` per column).

3 Target Platforms

We target AES for low-end lightweight devices and evaluate its performance on an 8-bit AVR, both in terms of speed and code size; independently, we also target the many-core, high performing, Cell and GPU platforms. Below, we introduce the three target platforms and discuss their overall design and execution models.

3.1 8-bit AVR microcontroller

Advanced Virtual Risc (AVR) is a family of 8-bit microcontrollers designed by Atmel, targeting low-power embedded systems. Although a lightweight microcontroller, the AVR has 32 8-bit registers, a large number of instructions (125 for the AT90USB82/162), between 512B and 384KB in-system programmable flash (ISP), 0 to 4KB of EEPROM and 0 to 32KB SRAM, timers, counters, USART, SPI, and many other features and peripherals that make it a favorable platform for embedded development [2].

The AVR CPU is a modified Harvard architecture (program and data memories are separate) with a two stage single level pipeline supporting instruction pre-fetching. The (memory) parallelism and pipelining greatly improve the microcontroller's performance. Moreover, the majority of AVR instructions take up only a single 16-bit word and execute with a single cycle latency; only a small number of instructions require two or four cycles to complete. Features like free pre-decrement and post-increment of pointer registers also contribute towards small and efficient program code.

The data memory consists of the register file, I/O memory and SRAM, and as such the various direct and indirect addressing (through 16-bit pointer registers X, Y, Z) modes can be used to not only access the data memory, but also the 32 registers. The ability to access the register file as memory, in addition to the optimized direct register access, provides a designer with additional flexibility in optimizing an application implementation. We note, however, that although direct addressing can access the whole data space, indirect addressing (with displacement) is limited to 63 address locations from only one of the pointer registers [2], and this restriction may require the implementer to use techniques such as double-jumping. Another limitation is that only the Z register may be used for addressing flash memory, e.g., for AES S-box lookups, and in some AVR devices this is not possible at all. Additionally, the flash memory is relatively small and because in practical applications it is of little interest to dedicate the whole flash to a cryptographic primitive, it is critical that the code-size of AES remain small.

Most AVRs are clocked between 0 and 20 MHz, with some of the higher-end ones reaching 32 MHz, and with the ability to execute one instruction per cycle the embedded microcontrollers can achieve throughputs up to 20 MIPS (16 MIPS for the AT90USB162). Thus, given the relatively high computation power of these low-cost and low-power devices, the performance of block ciphers, such as AES, is of practical consideration for applications requiring cryptographic primitives (e.g., automobile electronic keys).

3.2 The Cell Broadband Engine

The Cell architecture [12], jointly developed by Sony, Toshiba, and IBM, is equipped with one dual-threaded, 64-bit in-order "Power Processing Element" (PPE), which can offload work to the eight "Synergistic Processing Elements" (SPEs) [24]. The SPEs are the workhorses of the Cell processor. Each consists of a Synergistic Processing Unit (SPU), 256 kilobyte of private memory called Local Store (LS) and a Memory Flow Controller (MFC). The latter handles communication between each SPE and the rest of the machine, including main memory, as explicitly requested by programs. All code and data must fit within the LS if one wants to avoid the complexity of sending explicit DMA (Direct Memory Access) requests to the MFC.

Most SPU instructions are 128-bit wide SIMD operations performing sixteen 8-bit, eight 16-bit, four 32-bit, or two 64-bit computations in parallel. Each SPU is also equipped with a large register file containing 128 registers of 128 bits each, providing space for unrolling and software pipelining of loops, hiding the relatively long latencies of its instructions. Unlike the processor in the PPE, the SPUs are asymmetric processors, having two pipelines (denoted by the odd and the even pipeline) which are designed to execute two disjoint sets of instructions. In the ideal case, two instructions can be dispatched per cycle.

The SPUs are in-order processors with no hardware branch-prediction. The programmer (or compiler) must instead tell the instruction fetch unit in advance

where a (single) branch instruction will jump to. Hence, for most code with infrequent jumps and where the target of each branch can be computed sufficiently early, perfect branch prediction is possible.

One of the first applications of the Cell processor was to serve as the heart of Sony's PlayStation 3 (PS3) video game console. The Cell contains eight SPEs, but in the PS3 one of them is disabled, allowing improved yield in the manufacturing process as any chip with a single faulty SPE can still be used. One of the remaining SPEs is reserved by Sony's hypervisor, a software layer providing a virtual machine environment for running e.g., Linux. In the end we have access to six SPEs when running Linux on (the virtual machine on) the PS3. Fortunately, the virtualization does not slow down programs running on the SPUs, as they are naturally isolated and protection mechanisms only need to deal with requests sent to the MFC.

Besides being used in the PS3, (parts of) the Cell has been placed on an PCI-Express card such that it can serve as an arithmetic accelerator. Such cards are available with a complete Cell (by Mercury), one PPE and eight SPEs running at 2.8GHz, equipped with 5 GB of memory or with four SPEs (by Leadtek) running at 1.5GHz with access to 128 MB of memory. The Cell has also established itself in the high-performance market by occupying the first place, with the Roadrunner supercomputer, in the top 500 supercomputing list [9].

3.3 GPU Using CUDA

Similar to the PS3, Graphic Processing Units (GPUs) have mainly been game- and video-centric devices. Moreover, due to the increasing computational requirements of graphics-processing applications, GPUs have become very powerful parallel processors and thus incited research interest in computing outside the graphics-community. Until recently, however, programming GPUs was limited to graphics libraries such as OpenGL [22] and Direct3D [7], and for many applications, especially those based on integer-arithmetic, the performance improvements over CPUs was minimal or even degrading. The release of NVIDIA's G80 series and ATI's HD2000 series GPUs, both of which implemented the unified shader architecture, along with the companies' release of higher-level languages Compute Unified Device Architecture (CUDA), Close to the Metal (CTM) [19] and the more recent Open Computing Language (OpenCL) [**?**], however, facilitate the development of massively-parallel general purpose applications for GPUs [16,1]. These general purpose GPUs (GPGPUs) have become a common target for numerically-intensive applications given their ease of programming (compared to previous generation GPUs), and ability to outperform CPUs in data-parallel applications, commonly by orders of magnitude. In this paper we focus on NVIDIA's GPU architecture with CUDA, programming ATI GPUs with OpenCL is part of our ongoing work and results will be reported in the final version of this paper.

In addition to the common floating point processing capabilities of previous-generation GPUs, starting with the G80 series, NVIDIA's GPU architecture added support for integer arithmetic, including 32-bit addition/subtraction and

bit-wise operations, scatter/gather memory access and various memory spaces (global, texture, constant and shared) [15,16]. Each GPU contains between 10 and 30 streaming multiprocessors (SMs) each equipped with: eight scalar processor (SP) cores, fast 16-way banked on-chip shared memory (16KB/SM), a multithreaded instruction unit, large register file (8192 for G80-based GPUs, 16384 for the newer GT200 series), read-only caches for constant (8KB/SM) and texture memory (varying between 6 and 8 KB/SM), and two special function units for transcendentals. We refer to [16] for further details.

CUDA is an extension of the C language that employs the new massively parallel programming model, single-instruction multiple-thread. SIMT differs from SIMD in that the underlying vector size is hidden and the programmer is restricted to writing scalar code that is parallel at the thread-level. CUDA extends the C language to support SIMT by allowing the programmer to define kernel functions, which are compiled for and executed on the SPs of each SM, in parallel: each light-weight thread executes the same code, operating on different data. A maximum of 512 threads can be grouped into a thread block which is scheduled on a single SM, the threads of which time-share the SPs. This additional hierarchy provides for threads within the same block to communicate using the on-chip shared memory and synchronize their execution using barriers (specifically, the `__syncthreads()` intrinsic blocks thread execution until all threads within the block have reached the synchronization point). Moreover, multiple thread blocks can be executed simultaneously on the GPU, a maximum of eight thread blocks can be scheduled per SM and in order to hide instruction and memory (among other) latencies, it is recommended that a minimum of two blocks be available for scheduling on each SM.

4 Porting AES

When porting the AES to our target platforms different implementation decisions have to be made. These decisions are influenced by the features, or restrictions, of the instruction sets and the available memory.

First, an optimized 8-bit implementation of AES for the AVR was designed. This version was implemented in AVR assembler, with each of the transformations independently optimized to minimize register usage and execution cycles. Furthermore, this 8-bit version of the AES was used as a framework to create a byte-sliced [4], implementation on the SIMD-architecture of the SPE. Hence, 16 instances of the AES are processed in parallel per SPE.

Unlike the Cell and AVR, the GPU does not directly benefit from the byte-sliced framework and, instead, the T-table approach is used, see Equation (1). We observe that this approach requires 4 look-ups and 3 `xor`s per column; thus, a total of 16 look-ups and 12 `xor`s per round. Moreover, for applications targeting GPUs older than the GT200 series, with additional rotations a single T_i table can be used as the tables are simply rotations of each other.

[4] The notion of bitslicing was introduced by Biham in [4].

Algorithm 1 Pseudo-code for the Cell architecture to fetch 16 S-box constants simultaneously in a SIMD-fashion. All variables are 128-bit wide registers, variable *in* contains 16 bytes which are used as indices to look up substitution values in the S-box which is 16×16 bytes; the 16-byte rows are denoted by S_i, $0 \leq i \leq 15$.

Input: *in*
Output: *out*

```
    r := and (in, 0x1F);
   a0 := shuffle (S0, S1, r);
   a1 := shuffle (S2, S3, r);
   a2 := shuffle (S4, S5, r);
   a3 := shuffle (S6, S7, r);
   a4 := shuffle (S8, S9, r);
   a5 := shuffle (S10, S11, r);
   a6 := shuffle (S12, S13, r);
   a7 := shuffle (S14, S15, r);
  sel := cmpgt (in, 0x7F);
   a0 := select (a0, a4, sel);
   a1 := select (a1, a5, sel);
   a2 := select (a2, a6, sel);
   a3 := select (a3, a7, sel);
    r := shift_left (in, 1);
  sel := cmpgt (r, 0x7F);
   a0 := select (a0, a2, sel);
   a1 := select (a1, a3, sel);
    r := shift_left (r, 1);
  sel := cmpgt (r, 0x7F);
  out := select (a0, a1, sel);
```

4.1 Optimizing Code for the Cell

In this work we use the naming convention of the Cell when discussing the implementation for the SPE. That is, a word consists of 32 bits and a 128-bit register on the SPE is called either a vector or quadword. The SPE has a rich instruction set which operates simultaneously on single-, double- or quadwords. All distinct binary operations $f : \{0,1\}^2 \rightarrow \{0,1\}$ are available, other instructions of particular interest are the `shuffle` and `select`. The `shuffle` instruction can rearrange (shuffle) 16 bytes of the 32-byte input (two quadwords) specified by a pattern or select one of the constants {`0x00, 0xFF, 0x80`} to the output quadword. The `select` instruction acts as a multiplexer; depending on the input pattern the corresponding bit from either the first or the second input quadword is selected as output.

The 16-way SIMD capabilities of the SPE, working on 16 bytes simultaneously, are used to create a byte-sliced implementation of AES. In an optimistic scenario one expects to achieve a 16-fold speedup compared to machines which natively have a wordsize of one byte and a comparable instruction set. For many of the operations required by AES, e.g., bitwise operations, this speedup holds

on the Cell with the additional advantage that the Cell has a more powerful instruction set when compared to other architectures.

Some operations, however, are not so trivial to perform in SIMD. A prime example is the S-box lookup. Typically this can be done in a few instructions, depending on the index, by calculating the address to load from and then performing the load. Doing this for 16 values in parallel in a straight-forward (naive) way can become a bottleneck. In [17] some techniques are briefly described on how to do this efficiently on the SPE architecture. The idea is to use the five least significant bits as input to eight different table lookups – one for each possible value of the three most significant bits. Then we extract those three bits one by one, and for each of them we build a value we use to select one or the other half of the outputs from those eight table lookups. In the end we have the correct output for the full 8-bit input, performed 16 times in parallel. See Algorithm 1 for the SPE pseudo-code.

4.2 Optimizing Code for the GPU

As the instruction set of the GPU is substantially less rich than that of the Cell, when optimizing an implementation using CUDA, it is essential to be able to execute many threads concurrently and thereby maximally utilize the device. Hence, our GPU implementation processes thousands of streams. Moreover, we also consider implementations with on-the-fly key scheduling, key-expansion in texture memory, key expansion in shared memory, and variants of the first two with storage of the T-tables and S-box (or inverse S-box) in shared memory. To maximize the throughput between the device and host, our implementations use page-locked memory with concurrent memory copies and kernel execution on the GT200 GPUs; the older G80 series GPUs do not support these features.

Similar to the implementation of [13], for the first three variants we layout the tables in constant memory. Because the constant memory cache size is 8KB, the full tables can be cached with no concern for cache misses. Although this approach has the advantage of simplicity, unless all the threads of a half-warp (16 threads executing concurrently) access the same memory location, the accesses must be serialized and thus the gain in parallelism is severely degraded. Nonetheless, we improve on [13], by including on-the-fly key scheduling and key-expansion in texture and shared memory variants. The AES implementation of [13] assumed the availability of the expanded key in global memory, which is of practical interest for single-stream cryptographic applications; for multi-stream cryptographic and cryptanalytic applications, however, key scheduling is critical as deriving many keys on the CPU is inefficient.

For the on-the-fly key scheduling variants, each thread is independent and the designs are thus ideal for key search applications or multi-stream applications with many thousand streams. For many applications, however, having on the order of a few hundred to a few thousand streams is sufficient and thus further speedup can be achieved by doing the key expansion in texture or shared memory. Texture memory, unlike constant memory, allows for multiple non-broadcast accesses and, like constant memory, has the advantage that it can be written

once and have the expanded keys live across multiple kernel launches, making its use very advantageous in the encryption and decryption of multiple blocks.

Regarding shared memory access, we note that although a single instruction is issued per warp, the warp execution is split into two half-warps and if no threads in the half-warp access the same shared memory location, i.e., there is no bank conflict, 16 different reads/writes can be completed in two cycles (per SM). This further increases the throughput of AES using precomputed round keys. For the key expansion into shared memory we create 16 stream-groups per block, each group consisting of multiple threads that share a common expanded key. Although bank conflicts on the GT200 series results in only serializing the conflicting accesses, as opposed to the G80s serialization of all the threads in the half warp, we carefully implemented the shared memory access to avoid any bank conflicts. Furthermore, although the number of groups per block can be increased, we instead increased the number of blocks to allow for the hiding of block-dependent latencies. Of course, for the key expansion into texture and shared memory no additional gain is achieved unless multiple blocks are encrypted.

As previously mentioned, the throughput of constant memory for random access is quite low when compared to shared and texture memory and so we further optimize AES by placing the T-tables and S-box in shared memory. To avoid bank conflicts one T_i (of size 1KB) must be stored in each bank; this of course, is not directly possible because kernel arguments are also usually placed in shared memory, and furthermore if most of the table (save a few entries), as in [11], is placed in shared memory, the maximum number of blocks assigned to that SM would be limited to one. Thus, the overall gain would not be much higher than using constant memory. The authors in [11] further propose a quad-table approach in shared memory, but limit the details of whether the design contains bank conflicts. Our shared memory table approach is a "lazy" approach in simply laying out the tables in-order. Because we are targeting the newer generation GPUs, a bank conflict is resolved by serializing only the colliding accesses; thus, although bank conflicts are expected (simulations show that roughly 35% of the memory accesses are serialized, so 6 of the 16), on average, the gain in using shared memory is much higher than constant memory. Further optimizing the T-table lookup method is part of our current ongoing work.

For on-the-fly key generation we buffered the first round key in shared memory, from which the remaining round keys are derived. Adopting the method of [8], during each round four S-box lookups and five `xor`s are needed to derive the new round key. Additional caching (e.g., the last round) can further improve the performance of the implementation. The key scheduling for decryption consists of running the encryption key scheduling algorithm and then applying `InvMixColumns` to all, except the first and last, round keys; it is apparent that on-the-fly key generation for decryption is considerably more complex. For decryption we buffer the first round key, and (after running the encryption key scheduler) the `InvMixColumns` of the final key. We derive all successive keys from the second to last round key using six `xor`s, a `MixColumns` transforma-

Reference	Key scheduling	Encryption (cycles)	Decryption (cycles)	Code size (bytes)	Notes
[25] Fast	on-the-fly	1,259	1,259	1,708	Hardware ext.
[25] Compact	on-the-fly	1,442	1,443	840	cost: 1.1 kGates
[21]	precompute	3,766	4,558	3,410	
[20] Fast	precompute	2,474	3,411	3,098	Key setup:
[20] Furious	precompute	2,739	3,579	1,570	Enc 756 cycles
[20] Fantastic	precompute	4,059	4,675	1,482	Dec 4,977 cycles
[18]	precompute	2,555	6,764	2,070	Key setup:
[18]	precompute	2,555	3,193	2,580	2,039 cycles
New - Low RAM	precompute	2,153	2,901	1,912	Key setup: 789 cycles
New - Fast	precompute	1,993	2,901	1,912	747 cycles

Table 1. *AES-128 implementation results on the 8-bit microcontroller AVR.*

tion (of part of the key), and a transformation combing `InvMixColumns` and `InvSubBytes` per round. These complex transformations nonetheless take advantage of the T-tables, with the additional need for an S-box[S-box[·]] table to further optimize the memory pressure. This efficient on-the-fly key scheduling is not GPU specific, and can be further applied to any other T-table based implementations.

5 Results

All our implementations, for the three architectures, are designed to reduce the number of clock cycles required for encryption and decryption. Table 1 states AES-128 performance results obtained on the AVR microcontroller. Depending on the AVR model used, the availability of RAM and flash memory varies. The code size of the implementation, which needs to fit in the flash memory, is presented in the table as well. We created two versions: a fast and a compact version. The compact version only stores the key (176 bytes) in RAM but, no additional tables. The faster version trades RAM usage for speed by placing the S-box in RAM, and thus increasing the required size by 256 bytes. Our results are obtained by running our compact version on the AT90USB162 (16 MHz, 512 byte RAM and 16 kilobyte flash) and the fast version on the larger AT90USB646 (16 MHz, 4 kilobyte RAM and 64 kilobyte flash). Although a direct comparison is not possible, for completeness, estimates of an AVR implementation using hardware extensions are also shown in Table 1. Figure 1 graphically shows the code size versus the required cycles for decryption and encryption of different AVR implementations of AES-128.

Table 2 gives AES-128 performance results obtained when running on the various GPUs and the SPE architecture. Not many benchmarking results of AES on the SPE architecture are reported in the literature. We therefore compare our results to the performance data given by IBM in [23]. This single-stream

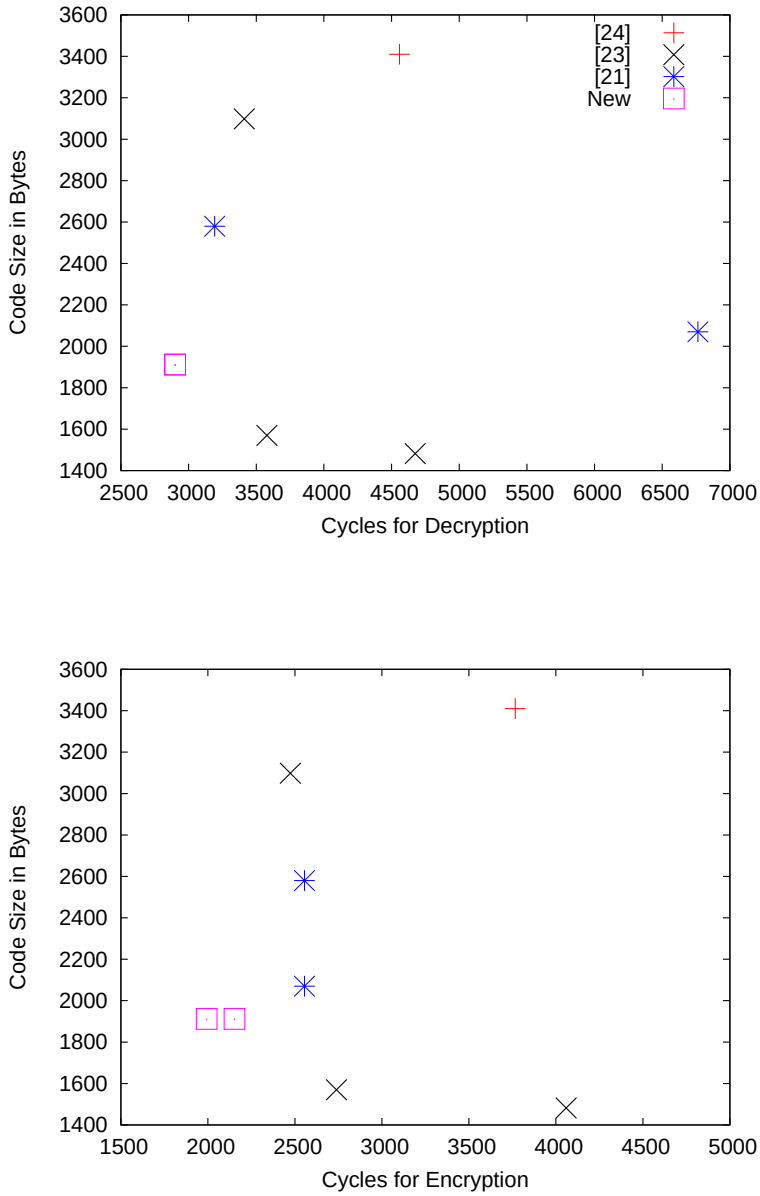

Fig. 1. Code size versus cycle count for decryption and encryption of different AES-128 AVR implementations.

Reference	Algorithm	Architecture	Cycles/ byte	Gb/sec
[13], 2007	Enc (P)	NVIDIA 8800 GTX, 1.35GHz	1.30	8.3
[26], 2007	Enc (F)	ATI HD 2900 XT, 750MHz	1.71	3.5
[11], 2008	Enc (P)	NVIDIA 8800 GTX, 1.35GHz	1.56	6.9
This article, T-smem	Enc (F)	NVIDIA 8800 GTX, 1.35GHz	0.94	11.5
This article, T-smem	Enc (T)		0.74	14.6
This article, T-smem	Dec (F)		1.43	7.6
This article, T-smem	Dec (T)		0.76	14.3
This article	Enc (F)	NVIDIA GTX 295, 1.24GHz	1.21	8.2
This article	Enc (T)		1.02	9.7
This article	Enc (S)		0.66	15.1
This article, T-smem	Enc (F)		0.21	47.1
This article, T-smem	Enc (T)		0.17	59.6
This article	Dec (F)		7.16	1.4
This article	Dec (T)		0.70	14.2
This article	Dec (S)		0.67	14.9
This article, T-smem	Dec (F)		0.34	29.3
This article, T-smem	Dec (T)		0.19	52.4
[23], 2005	Enc (P)	SPE, 3.2GHz	12.4	2.1
[23], 2005	Dec (P)		17.1	1.5
This article	Enc (P)	SPE, 3.2GHz	11.7	2.2
This article	Dec (P)		14.4	1.8

Table 2. *Different AES-128 implementations results on a single SPE and various GPU architectures. (P) = key scheduling is pre-computed, (F) = key scheduling is on-the-fly, (T) = key expansion in texture memory, (S) = key expansion in shared memory*

high-performance implementation is not byte-sliced, as our implementation, but optimized for the SPE-architecture to take full advantage of the SIMD properties. Compared to the IBM implementation, our SPE implementation is 6% and 16% faster for encryption and decryption, respectively. Our byte-sliced key generation routine runs in 62 clock cycles per stream.

To give an indication how well our C-implementation of AES runs on the SPE we estimate a rough lower bound of encryption using the techniques employed in our implementation. Every round the `AddRoundKey`, `SubBytes`, `ShiftRows` and the `MixColumns` need to be computed four times. A single instance of the first three steps can be computed with four loads and four `xors` and four times the operations as outlined in Algorithm 1 to load the S-box constants. The computation of a single instance of the `MixColumns` is done with 3 odd and 27 even instructions. Hence, nine full rounds and one final round (without the `MixColumns`) plus post-processing (the so-called output whitening) require 1752 odd and 2764 even instructions. This is without taking the loading and storing of the input and output into account. Assuming everything is perfectly scheduled, dispatching one pair of an odd and even instruction per cycle (if available), the

encryption should run in 2764 cycles for sixteen streams in parallel. This gives a lower bound of $2764/(16 \cdot 16) = 10.8$ cycles per byte for our encryption implementation. The 11.7 cycles per byte experienced in practice is slightly higher and can be explained by the fact that some instructions are imperfectly scheduled, thus causing stalls, and due to a possible imbalance in the pipelines which effectively prevents the dispatching of a pair of odd and even instructions every clock cycle.

There have been numerous implementations of AES on GPUs, we however, only compare against GPUs with support for integer arithmetic. Table 2 compares our GTX 295 and GeForce 8800 GTX implementations with those in [13,26,11]. The results in the table include the memory transfer along with the kernel execution, each stream encrypting 256 random blocks. We further note that the GTX 295 contains two GPUs (clocked-down version of the GTX 280 GPU). We point out the poor performance of decryption for the on-the-fly with tables in constant memory (the simplest method) is due to cache misses; the decryption with on-the-fly key scheduling requires the T-tables for encryption and decryption in addition to the S-box and inverse S-box, which are overall greater than 8KB. We, of course, only present these results for comparison in evaluating AES's performance on the GPU using the various memory spaces available.

Since the GT200 series GPUs address many of the limitations of the G80 series GPUs a direct comparison is not appropriate, withal, we note that with the exception of our simplest implementation our results outperform all previous GPU implementations of AES. Additionally, although our implementations target the GT200 GPUs, which in addition to the previously-mentioned advantages over the G80 have more relaxed memory accesses pattern restrictions, for completeness we benchmark our fastest implementations on the 8800 GTX. As shown in Table 2, our fastest GTX 295 implementation is roughly 8 times faster than [13], 10 times faster than [26], and 9.4 times faster than [11]. Similarly, our fastest 8800 GTX implementation is 43%, 57% and 53% faster than [13], [26], and [11], respectively. Moreover, unlike the implementations of [13] and [11], our implementations also include key scheduling methods, without which the throughput would increase by roughly 23% (see [26]). Compared to [11], we do not limit our implementation to CTR, for which additional improvements can be made [3]. When compared to the AES implementation in [26], our streams encrypt different plaintext messages with different keys; tweaking our implementations for applications of key searching as in [26] would further speed up the AES implementation by at least 35% as only one message block would be copied to the device.

6 Conclusion

We presented new software speed records for encryption and decryption when running AES-128 on the 8-bit AVR microcontroller, the Cell broadband engine architecture and NVIDIA graphics processing units. To achieve these performance records a byte-sliced implementation is employed for the first two ar-

chitectures, while a T-table approach is used for our GPU implementations. The implementations targeting the Cell and GPU architectures process multiple streams in parallel to obtain the results. Furthermore, this is the first AES implementation for the GPU which implements both encryption and decryption.

References

1. AMD. ATI CTM Reference Guide. Technical Reference Manual, 2006.
2. Atmel Corporation. 8-bit AVR Microcontroller with 8/16K Bytes of ISP Flash and USB Controller. Technical Reference Manual, 2008.
3. D. J. Bernstein and P. Schwabe. New AES software speed records. In *Progress in Cryptology - INDOCRYPT 2008*, volume 5365 of *Lecture Notes in Computer Science*, pages 322–336. Springer, 2008.
4. E. Biham. A Fast New DES Implementation in Software. In *FSE 1997*, volume 1267 of *LNCS*, pages 260–272, 1997.
5. A. Biryukov and D. Khovratovich. Related-key Cryptanalysis of the Full AES-192 and AES-256. Cryptology ePrint Archive, Report 2009/317, 2009. http://eprint.iacr.org/.
6. A. Biryukov and D. K. I. Nikolic. Distinguisher and Related-Key Attack on the Full AES-256. In *Crypto 2009*, volume 5677 of *LNCS*, pages 231–249, 2009.
7. D. Blythe. The Direct3D 10 system. *ACM Trans. Graph.*, 25(3):724–734, 2006.
8. J. Daemen and V. Rijmen. *The design of Rijndael*. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 2002.
9. J. Dongarra, H. Meuer, and E. Strohmaier. Top500 Supercomputer Sites. http://www.top500.org/.
10. Frost & Sullivan. Asia Pacific's Final Wireless Growth Frontier. http://www.infoworld.com/t/networking/passive-rfid-tag-market-hit-486m-in-2013-102.
11. O. Harrison and J. Waldron. Practical Symmetric Key Cryptography on Modern Graphics Hardware. In *USENIX Security Symposium*, pages 195–210, 2008.
12. H. P. Hofstee. Power Efficient Processor Architecture and The Cell Processor. In *HPCA 2005*, pages 258–262. IEEE Computer Society, 2005.
13. S. A. Manavski. CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography. In *ICSPC 2007*, pages 65–68. IEEE, November 2007.
14. National Institute of Standards and Technology (NIST). FIPS-197: Advanced Encryption Standard (AES), 2001. http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf.
15. NVIDIA. NVIDIA GeForce 8800 GPU Architecture Overview. *Technical Brief TB-02787-001 v0*, 9, 2006.
16. NVIDIA. NVIDIA CUDA Programming Guide 2.3, 2009.
17. D. A. Osvik. Cell SPEED. SPEED 2007, 2007. http://www.hyperelliptic.org/SPEED/slides/Osvik_cell-speed.pdf.
18. D. Otte. AVR-Crypto-Lib, 2009. http://www.das-labor.org/wiki/Crypto-avr-lib/en.
19. J. Owens. GPU architecture overview. In *SIGGRAPH 2007*, page 2. ACM, 2007.
20. B. Poettering. AVRAES: The AES block cipher on AVR controllers, 2006. http://point-at-infinity.org/avraes/.
21. S. Rinne, T. Eisenbarth, and C. Paar. Performance Analysis of Contemporary Light-Weight Block Ciphers on 8-bit Microcontrollers. SPEED 2007, 2007. http://www.hyperelliptic.org/SPEED/record.pdf.

22. M. Segal and K. Akeley. The OpenGL graphics system: A specification (version 2.0). *Silicon Graphics, Mountain View, CA*, 2004.
23. K. Shimizu, D. Brokenshire, and M. Peyravian. Cell Broadband Engine Support for Privacy, Security, and Digital Rights Management Applications. https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/3F88DA69A1C0AC40872570AB00570985, October 2005.
24. O. Takahashi, R. Cook, S. Cottier, S. H. Dhong, B. Flachs, K. Hirairi, A. Kawasumi, H. Murakami, H. Noro, H. Oh, S. Onish, J. Pille, and J. Silberman. The circuit design of the synergistic processor element of a Cell processor. In *ICCAD 2005*, pages 111–117. IEEE Computer Society, 2005.
25. S. Tillich and C. Herbst. Boosting AES Performance on a Tiny Processor Core. In *CT-RSA*, volume 4964 of *LNCS*, pages 170–186, 2008.
26. J. Yang and J. Goodman. Symmetric Key Cryptography on Modern Graphics Hardware. In *Asiacrypt 2007*, volume 4833 of *LNCS*, pages 249–264, 2007.