Fast Software AES Encryption

Dag Arne Osvik[1], Joppe W. Bos[1], Deian Stefan[2], and David Canright[3]

[1] Laboratory for Cryptologic Algorithms, EPFL, CH-1015 Lausanne, Switzerland
[2] Dept. of Electrical Engineering, The Cooper Union, NY 10003, New York, USA
[3] Applied Math., Naval Postgraduate School, Monterey CA 93943, USA

Abstract. This paper presents new software speed records for AES-128 encryption for architectures at both ends of the performance spectrum. On the one side we target the low-end 8-bit AVR microcontrollers and 32-bit ARM microprocessors, while on the other side of the spectrum we consider the high-performing Cell broadband engine and NVIDIA graphics processing units (GPUs). Platform specific techniques are detailed, explaining how the software speed records on these architectures are obtained. Additionally, this paper presents the first AES decryption implementation for GPU architectures.

Key words: Advanced Encryption Standard (AES), Advanced Virtual RISC (AVR), Advanced RISC Machine (ARM), Cell Broadband Engine, Graphics Processing Unit (GPU), Symmetric Cryptography

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). Today, the AES is one of the most widely used encryption primitives. A wide range of computational devices, from high-end machines, such as dedicated cryptographic servers, to low-end radio frequency identification (RFID) tags, use this encryption standard as a primitive to implement security. Besides its well-regarded security properties[1], the AES is extremely efficient on many different platforms, ranging from 8-bit microcontrollers to 64-bit processors to FPGAs. Indeed, efficiency was a crucial metric in making Rijndael an encryption standard.

There is an active research area devoted to not only creating more efficient and secure implementations, but also evaluating the performance of the AES on different architectures. For example, the recent AES performance records for the Intel Core i7 by Käsper and Schwabe [15] was rewarded with one of the best paper awards at CHES 2009. Many improved performance results are aided by techniques such as byte- and bitslicing, as introduced by Biham [5], and by improved single-instruction multiple data (SIMD) instruction set extensions to commonly available architectures such as x86.

[1] The only attack on the full AES is applicable in the related key scenario to the 192-bit [6] and 256-bit key versions [6,7].

It is expected that the use of lightweight devices, i.e., low-end smart cards and radio frequency identification tags, in electronic commerce and identification will grow rapidly within the near future. For instance, the passive RFID tag market is expected to reach up to US$ 486M by 2013 [12], and the AES has already attracted significant attention due to its capabilities for such devices [11]. This work further investigates the performance of the AES on low-power devices; specifically, 8-bit AVR microcontrollers and 32-bit ARM microprocessors. Other platforms we target in this article are the high-end Cell Broadband Engine architecture (Cell) and the NVIDIA Graphics Processing Units (GPUs). For these platforms, which allow the use of vectorization optimization techniques, multiple input streams are processed at once using SIMD and SIMT (single instruction, multiple threads) techniques for the Cell and GPUs, respectively. Due to the low prices and wide availability of these devices it is interesting to evaluate their performance as cryptologic accelerators.

We present new software implementations of AES-128 with high speed and small code size. To the best of our knowledge, our results set new performance records on all the targeted platforms. These performance records are achieved by carefully mapping the different components of the AES to the various platforms for optimal performance. Our AVR implementation requires 124.6 and 181.3 cycles per byte for encryption and decryption with a code size of less than 2 kilobytes. Compared to the previous AVR records our encryption code is 0.62 times the size and 1.24 times faster. Our ARM implementation requires 34.0 cycles per byte for encryption, which is 1.17 times faster than the previous record on this platform. Our 16-way SIMD byte-sliced implementations for the synergistic processing elements of the Cell architecture achieve speeds of 11.3 and 13.9 cycles per byte, which are 1.10 and 1.23 times faster than the previous Cell records, for encryption and decryption respectively. Similarly, our fastest GPU implementation, running on a single GPU of the NVIDIA GTX 295 and handling many input streams in parallel, delivers throughputs of 0.32 cycles per byte for both encryption and decryption. When running on the older GeForce 8800 GTX our results are 1.2 times and 1.34 times faster than the previous records on this GPU with and without memory transfer, respectively. Furthermore, this is the first AES implementation for the NVIDIA GPU which implements both encryption and decryption.

The paper is organized as follows. Section 2 briefly recalls the design of the AES. In Section 3 our target platforms are described. Section 4 describes the techniques used and decisions made when porting the 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 the AES

The AES is a fixed block length version of the Rijndael block cipher [9,19], 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, taken as a polynomial of degree less than 4 with coefficients 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 [19] and [9]. 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. On-the-fly key generation proves useful in memory constrained environments such as microcontrollers.

For 32-bit (and greater word length) processors, in [9], 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 [9], 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 the 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). We, however, note that since the T_i's are simply rotations of each other, some implementations of (1) benefit from using a single table and performing the necessary rotations.

3 Target Platforms

On the one hand we target low-end lightweight devices—the performance and code-size of the AES on such devices, e.g., RFID-tags, is crucial (see for instance [11]) for the use of this encryption primitive in practice. Contrastingly,

on the other side of the performance spectrum we target the many-core, high performing, Cell and GPU platforms. Fast multi-stream implementations of the AES on these architectures show the potential use of these platforms as AES-accelerators which could be used in high-end servers. Below, we introduce the three target platforms and discuss their overall design and execution models.

8-bit Advanced Virtual RISC 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. Additionally, the microcontrollers are equipped with timers, counters, USART, SPI, and many other features and peripherals that make the AVR a favorable platform for embedded applications [3].

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. 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 using the Y or Z registers as base pointers, in addition to the displacement being limited to 64 address locations (including 0) [3]. Similarly, conditional branches have a 6-bit displacement restriction. Hence, it is often necessary for an implementer to use the trampoline technique to address these limitations. Only the Z register may be used for addressing flash memory, e.g., for the 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 the AES remain small.

Most AVRs are clocked between 0 and 20 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 the AES, is of practical consideration for applications requiring cryptographic primitives (e.g., electronic automobile keys).

32-bit Advanced RISC Machine. Advanced RISC Machine (ARM) is an instruction set architecture (ISA) design by ARM for high-performance 32-bit embedded applications. Although ARM cores are usually larger and more complex than AVRs, most ARM designs are also well regarded for their low power consumption and high code density—the ARM is one of the most widely used 32-bit processors in mobile applications [16].

The ARM family has evolved from the ARM1 to the prominent ARM7TDMI core (ARMv4T ISA) to the ARM9 (ARMv5), ARM11 (ARMv6) and most recent Cortex (ARMv7) [30]. The number of pipeline stages increased from 3 on the ARM7 to 8 on the ARM11, with different cores including various caches, memory management units, SIMD extensions, and other capabilities using coprocessors. Different cores even implement different memory architectures: some the von Neumann, while others, such as the StrongARM SA-1110 (ARMv4), are (modified) Harvard architectures. Most of the RISC design features are, however, constant across the families, making code written for the ARM7 core compatible with the ARM11.

Like the AVR, the ARM microprocessor is a load-store architecture with simple, yet powerful, addressing modes. The processor has 37 32-bit registers, of which only 16 are visible at any one time [27]. Register banking is employed to "hide" the registers used in operating modes other than user (e.g., supervisor, abort, etc.). Most instructions on the ARM execute in a single cycle, in addition to their opcode length being fixed to 32-bits[2] To eliminate the need for branching, almost all ARM instructions can be made conditional by adding a suffix to the instruction mnemonic. Furthermore, the condition bits are only modified by an instruction if the programmer desires (expressed by adding the `S` suffix to the instruction). We mention two additional features of the ARM, which are extensively used in this work: the inline barrel shifter and the load-store-multiple instruction. The inline barrel shifter allows for the second operand of most data processing instructions (e.g., arithmetic and logical) to be shifted or rotated before the operation is performed. The load and store-multiple instructions copy any number of general-purpose registers from/to a block of sequential memory addresses.

The Cell Broadband Engine. The Cell architecture [14], 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) [31]. The SPEs are the workhorses of the Cell processor and the target for the AES implementation in this article. Each SPE 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. If one wants to

[2] The fixed-size opcodes simplify decoding, but also increase the overall code size. To address this a separate and reduced (16-bit) instruction set, Thumb-2, can be used instead.

avoid the complexity of sending explicit DMA (Direct Memory Access) requests to the MFC, all code and data must fit within the LS.

Most SPU instructions are 128-bit wide SIMD operations performing sixteen 8-bit, eight 16-bit, four 32-bit, or two 64-bit operations in parallel. Each SPU is also equipped with a large register file containing 128 registers of 128 bits each. This provides space for unrolling and software pipelining of loops, hiding the relatively long latencies of the 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. Most of the arithmetic instructions are executed in the even pipeline while most of the load and store instructions are executed in the odd pipeline. In an ideal scenario, two instructions (one even and one odd) can be dispatched per cycle. The SPUs are in-order processors with no hardware branch-prediction. The programmer (or compiler) must instead notify 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) game console. The Cell contains eight SPEs, but in the PS3 one 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 an external operating system. Hence, we have access to six SPEs when running GNU/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 its use in the PS3, the Cell has been placed on a PCI-Express card such that it can serve as an arithmetic accelerator. The Cell has also established itself in the high-performance market with the Roadrunner supercomputer, in the top 500 supercomputing list [10]. This supercomputer consists of a revised variant of the Cell, the PowerXCell 8i, which is available in the IBM QS22 blade servers.

Graphics Processing Units Using the Compute Unified Device Architecture. Similar to the PS3, Graphic Processing Units (GPUs) have mainly been game- and video-centric devices. Due to the increasing computational requirements of graphics-processing applications, GPUs have become very powerful parallel processors and this, moreover, incited research interest in computing outside the graphics-community. Until recently, however, programming GPUs was limited to graphics libraries such as OpenGL [28] and Direct3D [8], and for many applications, especially those based on integer-arithmetic, the performance improvements over CPUs was minimal, sometimes even degrading. The release of NVIDIA's G80 series and ATI's HD2000 series GPUs (which implemented the unified shader architecture), along with the companies' release of higher-level language support with Compute Unified Device Architecture (CUDA),

Close to Metal (CTM) [24] and the more recent Open Computing Language (OpenCL) [18], however, facilitate the development of massively-parallel general purpose applications for GPUs [21,1]. These general purpose GPUs have become a common target for numerically-intensive applications given their ease of programming (relative 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 using the Stream SDK (successor of CTM) is part of our ongoing work.

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 bitwise operations, scatter/gather memory access and different memory spaces [20,21]. 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 memories (varying between 6 and 8 KB/SM), and two special function units (for transcendentals). We refer to [21] 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. The programmer defines *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 number of threads (less than 512) are 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. Moreover, multiple thread blocks can be executed simultaneously on the GPU as part of a *grid*; a maximum of eight thread blocks can be scheduled per SM and in order to hide instruction and memory (among other) latencies, it is important that at least two blocks be scheduled on each SM.

4 Porting the AES

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

We started by optimizing the AES for 8-bit AVR microcontroller architecture. This 8-bit version of the AES is used as a framework to create a byte-sliced implementation on the SIMD-architecture of the SPE. Hence, 16 instances of the AES are processed in parallel per SPE using 16-way SIMD arithmetic on

the 128-bit registers. Unlike the Cell and AVR, the ARM and GPU architectures do not directly benefit from the byte-sliced framework and, instead, the T-table approach is used, see Section 2.

In order to illustrate some of the techniques used we often use the functionality of `xtime` as an example. The multiplication of a variable $b \in \mathbb{F}_{2^8}$, using its polynomial representation, by the constant value `0x02` (which corresponds to the polynomial x) is denoted by `xtime`. The functionality of `xtime` can be implemented by a shift and a conditional `xor` with the constant `0x1B`, depending on whether the most-significant bit of b is set.

8-bit Advanced Virtual RISC Microcontroller. Our implementations of the AES on 8-bit AVR microcontrollers use the conventional lookup-based approach. The lookup tables used are the forward and inverse S-boxes, each 256 bytes; to keep the memory requirements of the implementation low, no other tables are used. Lookups are performed by putting the index value in the lower half of a pointer register; when the table is in flash memory, this needs to be in the Z pointer register. In order to allow for this simple way of calculating a lookup address, the tables must be 256-byte aligned, and in our implementation, the S-box pointer is always in the Z register. This allows the placement of the S-box in flash or SRAM memory to be selected at compile time, without any additional code modifications. Load instructions require 2 cycles when loading from SRAM and 3 cycles when loading from flash memory, so the former provides higher performance in cases where SRAM is available.

The functionality of `xtime` is implemented as a left shift followed by a conditional branch (depending on the shifted-out bit) which skips the `xor` (with `0x1B`) if the most significant bit is 0. On the AVR, a branch costs 2 cycles if taken and 1 cycle if not. In the latter case, the `xor` operation is performed, taking an additional cycle. Hence, the total number of required cycles (including the left shift) is independent of the branch and is always 3 cycles.

The `MixColumns` step is implemented without the use of lookup tables as a series of register copies, `xor`s and `xtime` operations, taking a total of 26 cycles. The `InvMixColumns` step is implemented similarly, but is more complicated and takes a total of 42 cycles.

On the AVR, conditional branches are limited to jumping 63 instructions back or 64 instructions forward. Hence, in our implementation, the branching to the code for the last round cannot be accomplished in a single step. However, since this branch is close to the beginning of the encryption loop we, instead, branch to an unconditional `rjmp` (relative jump) instruction just *before* the encryption loop. The relative jump instruction is able to jump 2047 instructions backward or 2048 instructions forward, and using this trampoline technique we efficiently reach the code for the last round.

We use our own internal, assembly only, calling convention for the interface between mode-of-operation code and the AES encryption core. However, the mode-of-operation code fully supports the C-programming language calling convention.

32-bit Advanced RISC Machine. For the ARM microprocessor we only implemented the AES-128 encryption. As the ARM has considerably more resources than the AVR, the main design goal is to optimize the implementation for speed; we do not take any code-size or memory restrictions into account when designing the code for this platform. Furthermore, we do not restrict ourselves to any specific ARM processor—the code is portable among the ARM family of processors.

The 32-bit based AES-128 encryption implementation for the ARM uses the T-table approach, see Section 2, storing only one of the four tables. This saves the required storage size for the T-table at no additional performance cost by taking advantage of the inline barrel shifter. Other features of the ARM are considered as building blocks of the AES implementation, including conditional execution of instructions and ability to perform multiple load or store operations using a single instruction. The code has been designed to work without slowdown for processors with up to 3 cycles load-to-use latency and with a minor slowdown when 4 cycles are required before a loaded value can be used. In practice, these latencies are often lower; for example, the ARM SA-1110 we used for benchmarking has a 2 cycle latency for cache hits.

In addition, all the steps in the round function and all the rounds have been completely unrolled to maximize speed. We would like to emphasize that an iterated version would have a much smaller code size and still be nearly as fast as the implementation described. Furthermore, although the encryption code is unrolled, the key expansion is not. The key-expansion is implemented such that a single byte from each 4-byte T-table element is used when an S-box entry is required. This is possible since each 4-byte entry of the T-tables is composed of four S-box multiples (e.g., $T_0[a] = (\texttt{0x02} \cdot \text{S-box}[a], \text{S-box}[a], \text{S-box}[a], \texttt{0x03} \cdot \text{S-box}[a])$).

Synergistic Processing Elements. The 16-way SIMD capabilities of the SPE, working on 16 bytes simultaneously, are used to create a byte-sliced implementation of the AES. In an optimistic scenario one may expect to achieve roughly a 16-fold speedup compared to machines with a word-size of one byte and a comparable instruction set. For many of the operations required by the AES, e.g., bitwise operations, this speedup holds on the Cell. Unlike most other modern architectures, on the SPE all distinct binary operations $f : \{0,1\}^2 \to \{0,1\}$ are available. Other instructions of particular interest for the implementation of the AES are the `shuffle` and `select` instructions. The `shuffle` instruction can pick any 16 bytes of a 32-byte (two 128-bit registers) input or select one of the constants {`0x00, 0xFF, 0x80`} and place them in any of the 16 byte positions of the 128-bit output register. We note that this allows 16 lookups in a 32 entry one-byte table using a single instruction. The `select` instruction acts as a 2-way multiplexer: depending on the input pattern the corresponding bit from either the first or the second input quadword is selected as output.

Some operations which are necessary to implement the AES are not, however, so trivial to perform in a SIMD fashion. A prime example is the S-box lookup. Typically this can be done in a few instructions, depending on the index, by

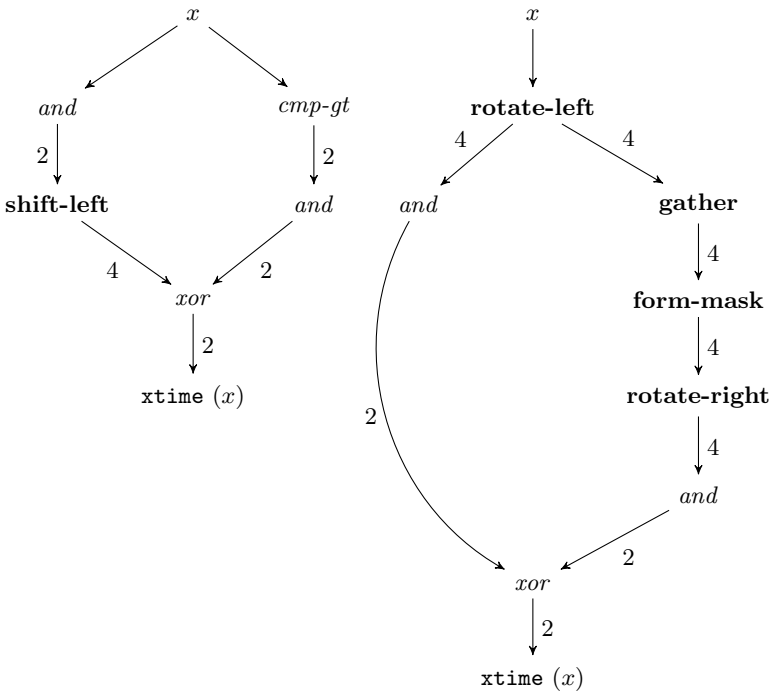

Fig. 1. Two different program flows, with cycle latencies, of an 16-way SIMD implementation of `xtime` on the Cell. Even and odd instructions are denoted by *italic* and **bold** text respectively. The unbalanced implementation requires 4 even and 1 odd instruction and has a latency of 8 cycles. The balanced implementation requires 3 even and 4 odd instructions and has a latency of 20 cycles.

calculating the address to load from and loading the S-box value from this address. Neither the Cell nor any other current mainstream architecture supports parallel lookups, so this approach would need to be performed sequentially. In [22], Osvik describes a technique on how to do this efficiently on the SPE architecture. The idea is to use the five least significant bits, of each of the 16 bytes in parallel, as input to eight different table lookups – one for each possible value of the three most significant bits. Then, one-by-one, extract the three remaining bits. Depending on the values of each of these bits, one or the other half of the lookup results are selected. After all three bits have been processed, the correct S-box output has been fetched, 16 times in parallel.

Simply implementing the AES on the Cell architecture in this way results in an imbalanced number of even and odd instructions, since the majority of the instructions are even (mainly due to all the required `xor`s). Rebalancing the instructions requires an unconventional way of thinking: increasing the number of instructions and the latency of a given part of the encryption might result in

faster overall code if we are able to "hide" these extra instructions in the odd pipeline.

Consider the implementation of `xtime` as an example. Recall that the functionality of `xtime` can be implemented by a shift and a conditional `xor`. However, when performing SIMD arithmetic (where the same steps need to be performed for all concurrent streams), conditional statements are to be avoided. We present an approach (which can be applied in 16-SIMD) that eliminates the conditional instruction by creating masks using the compare-greater-than (`cmp-gt`) instruction. This instruction compares the 16 8-bit values of two vectors simultaneously and returns either all ones or all zeros in every corresponding output byte, depending whether the value is greater or not. The constant value is selected depending on this select-mask using a single `and` instruction. As there is no 16-SIMD shift instruction on the Cell, this operation is mimicked by using an `and`, to clear the most-significant bit of the 16 entries, and performing a global 128-bit left shift. The resulting `xor` of these two values is the output of `xtime`. We note that this approach requires four even and one odd instruction, with a total latency of eight cycles.

An alternative implementation approach to `xtime` is by first rotating the entire 128-bit vector one bit to the left, making the most-significant bit of byte number i the least-significant bit of byte number $i - 1 \bmod 16$. In a separate register this bit is then cleared using an `and` instruction which is then used for the final `xor`. Just as in the previous approach a select-mask is created, however, in this case we are using the least-significant bit of the 16 elements of the rotated value. This is achieved by first gathering these least-significant bits, using the `gather` instruction, and forming the mask with the help of the form-select-byte-mask instruction `maskb` (both odd instructions). Next, these masks are rotated one byte to the right such that the index of the mask corresponds to the index of the original value of the least-significant bit of the rotated value. Following this, the masks are in the correct position and are used to select the constant value. Finally, this masked constant is used to create (using an `xor`) the output of `xtime`.

Figure 1 shows the program flow of both discussed methods; we denote the first as the *unbalanced* and the second as the *balanced* approach. The figure also highlights their respective latencies. Despite the longer latency of the balanced approach (20 versus 8 cycles) it requires one fewer even instructions in comparison to the unbalanced approach. Moreover, the three extra odd instruction can, in most instances of `xtime` in the AES encryption, be dispatched for "free" in pairs with surrounding even instructions.

In order to make an overall well-balanced (in terms of odd and even instruction count) implementation for the SPE on the Cell, similar techniques are applied when implementing a more balanced variant of the S-box fetching algorithm.

Graphics Processing Units. As the instruction set of the GPU is substantially less rich than that of the Cell and ARM, 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. Additionally, we also consider implementations with on-the-fly key scheduling, key-expansion in texture memory, key-expansion in shared memory, and variants of the former two with storage of the T-tables in shared memory. To maximize the throughput between the device and host, our GT200 implementations use page-locked host memory with concurrent memory copies and kernel execution. Since kernel execution and copies between page-locked host memory and device memory are concurrent, the latency of the memory copies is hidden (except for first and last kernel). The older G80 series GPUs do not, however, support concurrent memory transfers and kernel execution.

Our first three, and simplest, variants are similar to the implementation of [17] in placing the T-tables in constant memory. Because the constant memory cache size is 8KB, the full tables can be cached with no concern for having 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 (broadcast), the accesses must be serialized and thus the gain in parallelism is severely degraded. Hence, to lower the memory access penalties, in implementing the transformation according to (1), we only used as single T table (specifically: T_2)—unlike the ARM, which allows for inline rotates, the rotates on the GPU were implemented with two shifts and an `or`. Moreover, we improve on [17] by including on-the-fly key scheduling and key-expansion in texture and shared memory. The AES implementation of [17] assumes the availability of the expanded key in global memory, which is of practical interest only 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.

Our on-the-fly key scheduling variants are ideal for key search applications and multi-stream applications with many thousand streams since each thread independently encrypts/decrypts multiple (different) blocks with a separate key. Since our implementation processes multiple blocks, for the on-the-fly key generation we buffer the first round key in shared memory, from which the remaining round keys are derived. Adopting the method of [9], 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.

For many applications 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 retain the contents (expanded keys) across multiple kernel launches. Similarly, when accessing shared memory it is important to understand that although a single instruction is issued per warp, the warp execution is split into two half-warps and if no threads of a half-warp access the same shared memory location, i.e., there are no bank conflicts, 16 different reads/writes can be completed in two cycles (per SM). As with

using texture memory, this can further increase the throughput of the AES when encrypting/decrypting multiple blocks. Although shared memory bank conflicts on the GT200 series result 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.

For the key expansion into texture and shared memory we create 16 stream-groups per block, each group consisting of multiple threads that share a common expanded key. The key expansion for the texture memory variant is performed using a separate kernel and all subsequent kernel executions reuse the expanded key. The shared memory variant, however, performs the key expansion at the start of every kernel. Furthermore, the number of stream-groups per block was chosen between 8 and 24 (most commonly 16) to allow for a higher number of blocks to be scheduled concurrently and thus hide block-dependent latencies. We emphasize that for all variants (including on-the-fly), in addition to launching multiple grids, each kernel processes multiple blocks. Of course, for the key expansion into texture and shared memory no additional speedup is attained unless multiple blocks are processed.

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 the AES by placing the T-tables 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 [13], 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 very high. The authors in [13] also propose a quad-table approach in shared memory, though they do not specify whether the design contains bank conflicts. Our shared memory version 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. For these variants we store all four T tables, as the measured performance over using a single T table was higher.

We recall that 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 clear that on-the-fly key generation for decryption is considerably more complex than for encryption; the key expansion for all other variants is a direct application of this method. For on-the-fly 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` transformation (of part of the key), and a transformation combing `InvMixColumns` and `InvSubBytes` per round. These complex transformations also take advantage of the T-tables, with the additional need for an S-box[S-box[$\cdot$]] table to further

Reference	Key scheduling	Encryption (cycles)	Decryption (cycles)	Code size (bytes)	Notes
8-bit AVR microcontroller					
[32] Fast	on-the-fly	1,259	1,259	1,708	Hardware ext.
[32] Compact	on-the-fly	1,442	1,443	840	cost: 1.1 kGates
[26]	precompute	3,766	4,558	3,410	
[25] Fast	precompute	2,474	3,411	3,098	Key setup:
[25] Furious	precompute	2,739	3,579	1,570	Enc 756 cycles
[25] Fantastic	precompute	4,059	4,675	1,482	Dec 4,977 cycles
[23]	precompute	2,555	6,764	2,070	Key setup:
[23]	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
32-bit ARM microprocessor					
[2] Atasu et. al	precompute	639	638	5,966	
New	precompute	544	-	3,292	

Table 1. AES-128 implementation results on an 8-bit AVR microcontroller and a 32-bit ARM microprocessor.

lower the memory pressure. We note that this efficient on-the-fly key scheduling for decryption is not GPU specific and can be further applied to any other T-table based implementations.

5 Results

Table 1 shows AES-128 performance and code size (including lookup tables) for our implementations on 8-bit AVR microcontroller and 32-bit ARM microprocessor. Depending on the AVR model used, the availability of RAM and flash memory varies. We created two variants: a fast and a compact version. The compact version only stores the key (176 bytes) in RAM—no additional tables; this version is designed for AVR implementations with low RAM requirements. The faster version trades RAM usage for speed by placing the 256 byte S-box in RAM. Our timing 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 we also include estimates of an AVR implementation using hardware extensions [32] in Table 1. Figure 2 graphically shows the code size versus the required cycles for decryption and encryption of different AVR implementations of AES-128. Our AVR encryption and decryption routines are 1.24 and 1.10 times faster compared to the previous fastest results. In both cases we also achieve a smaller code size.

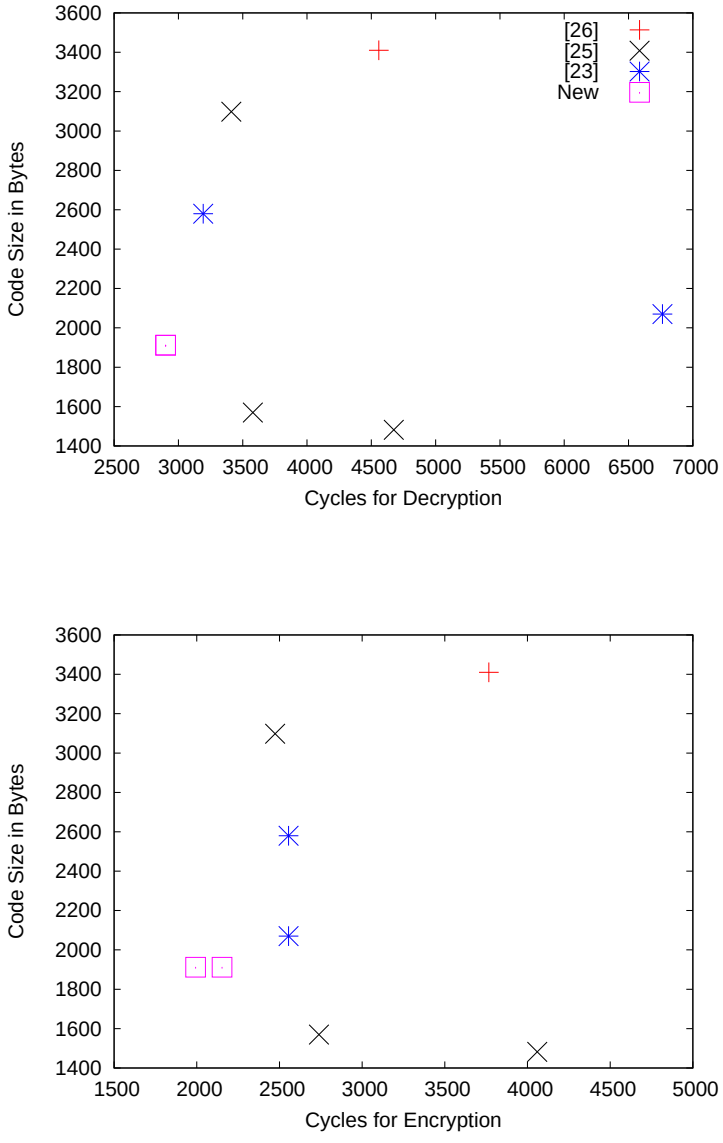

Fig. 2. Code size versus cycle count for decryption and encryption of different AES-128 AVR implementations.

Reference	Algorithm	Architecture	Cycles/ byte	Gb/sec
[17], 2007	Enc (P)	NVIDIA 8800 GTX, 1.35GHz	1.30	8.3
[33], 2007	Enc (F)	ATI HD 2900 XT, 750MHz	1.71	3.5
[13], 2008	Enc (P)	NVIDIA 8800 GTX, 1.35GHz	0.70	15.4
This article, T-smem	Enc (P)	NVIDIA 8800 GTX, 1.35GHz	0.52	23.3
This article, T-smem	Enc (F)		0.84	12.9
This article, T-smem	Enc (T)		0.64	16.8
This article, T-smem	Dec (F)		1.23	8.8
This article, T-smem	Dec (T)		0.65	16.6
This article	Enc (F)	NVIDIA GTX 295, 1.24GHz	1.13	8.8
This article	Enc (T)		0.91	10.9
This article	Enc (S)		0.92	10.8
This article, T-smem	Enc (F)		0.42	23.8
This article, T-smem	Enc (T)		0.32	30.9
This article	Dec (F)		2.46	4.0
This article	Dec (T)		1.37	7.2
This article	Dec (S)		1.38	7.1
This article, T-smem	Dec (F)		0.66	15.1
This article, T-smem	Dec (T)		0.32	30.8
[29], 2005	Enc (P)	SPE, 3.2GHz	12.4	2.1
[29], 2005	Dec (P)		17.1	1.5
This article	Enc (P)	SPE, 3.2GHz	11.3	2.3
This article	Dec (P)		13.9	1.8

Table 2. Different AES-128 performance results obtained when running on a single SPE of a PS3 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

For comparison, let us briefly analyze the previous fastest AVR AES implementation: the fast variant from [25]. This implementation integrates `xtime` in its S-box lookup tables, resulting in twice the table size for encryption, all elements of the S-box multiplied by `0x01` and `0x02`, and five times for decryption, all elements of the inverse S-box multiplied by `0x01`, `0x0E`, `0x09`, `0x0D` and `0x0B`. Hence seven tables are used, while we use only two. Although more tables may seem like a good way of speeding up the code, it also requires frequent changing of the S-box pointer. Furthermore, computing `xtime` costs the same as a lookup from flash (3 cycles), but does not require the input value to be copied to a particular register nor change any pointer registers.

To our knowledge, based on a review of the literature, the previous record for the ARM microprocessor is reported in [2] by Atasu, Breveglieri and Macchetti, where performance results are measured on the StrongARM SA-1110. Our implementation is benchmarked on this same ARM processor, the results of which are presented at the bottom of Table 1. The code size for our AES-128

implementation, using a T-table of 1024 (256×4) bytes, is 2156 bytes. The code size for the key-expansion is 112 bytes and is calculated to run in 260 cycles. This could not be accurately measured due to practical limitations: this ARM lacks a timestamp counter. The encryption code is calculated to run in 540 cycles and in practice, when performing repeated encryption of large messages, we measured 544 cycles per encryption using wall-clock time. The code-size of the implementation from [2] includes encryption, decryption and key-scheduling code for the AES working on 128-, 192- and 256-bit key lengths. Our result for encryption is 1.17 times faster compared to their 128-bit variant.

Table 2 gives AES-128 performance results obtained when running on the SPE architecture and various GPUs. We note that although we use sophisticated techniques to implement the AES on the Cell and the GPUs, see Section 4, we did not implement the AES using hand-optimized assembly (except for one GPU variant). Despite the fact that we implement the AES using the C-programming language, we are able to set new performance records for both platforms.

Few benchmarking results of the AES on the SPE architecture are reported in the literature. The only reported performance data we could find are from IBM [29]. This single-stream high-performance implementation is optimized for the SPE-architecture to take full advantage of the SIMD properties. Compared to the IBM implementation our SPE implementation, benchmarked on a PS3, is 1.10 and 1.23 times faster for encryption and decryption, respectively. Our byte-sliced key generation routine runs in 62 clock cycles per stream.

Our AES-SPE implementation uses the pipeline balancing techniques described in Section 4. For the encryption, in the `MixColumns` step, the `xtime` implemented with the long latency but fewer even instructions is used to achieve the best performance results. When decrypting, in the `InvMixColumns` step, `xtime` is called six times. In this case, calling each variant three times leads to optimal performance.

There have been numerous implementations of the 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 [17,33,13]. The GTX 295 results in the table include the memory transfer along with the kernel execution, each stream encrypting 256 random blocks. We further note that although the GTX 295 contains two GPUs (clocked-down version of the GTX 280 GPU), we benchmark on only one of them—performance on both GPUs scales linearly (assuming the memory transfer over PCI-Express is not a limitation).

Since the GT200 series GPUs address many of the limitations of the G80 series GPUs, a direct comparison is not appropriate, nonetheless, we note that our implementations using the shared memory to store the T tables outperform previous GPU implementations of the AES. To our knowledge the previous record on the 8800 GTX is that presented in [13]. As shown in Table 2, our fastest GTX 295 (single-GPU) implementation is roughly 4.1 times faster than [17], 5.3 times faster than [33] and 2.2 times faster than [13]. Additionally, although our implementations target the GT200 GPU, which in addition to the previously-

mentioned advantages over the G80 have more relaxed memory accesses pattern restrictions and ability for concurrent memory access and kernel execution, for completeness we benchmark our fastest implementations on the 8800 GTX with no modification. We also implemented a comparable parallel thread execution (PTX) assembly design using a pre-expanded key (in constant memory). The PTX implementation includes an additional double-buffering (to register) optimization which allows for the encryption of a block, while (hiding the latency in) reading the subsequent block. The peak throughput of our 8800 GTX implementation, measuring only the kernel run time, is 2.5, 3.3 and 1.3 times faster than [17], [33], and [13], respectively. With memory transfer, our PTX implementation is about 1.2 times faster than that of [13], delivering 8.6 Gb/sec versus 6.9 Gb/sec. Moreover, with respect to [17] and [13], our C implementation results are comparable and also include key scheduling methods. Compared to [13], we do not limit our implementation to CTR, for which additional improvements can be made [4]. Finally, when compared to the AES implementation in [33], our streams encrypt different plaintext messages with different keys; tweaking our implementations for applications of key searching as in [33] would further speed up the AES implementation by at least 35% as only one message block would be copied to the device.

6 Conclusion

New software speed records for encryption and decryption when running AES-128 on 8-bit AVR microcontrollers, the synergistic processing elements of the Cell Broadband Engine architecture, 32-bit ARM microprocessor and NVIDIA graphics processing units are presented. To achieve these performance records a byte-sliced implementation is employed for the first two architectures, while the T-table approach is used for the latter two. Each implementation uses platform specific techniques to increase performance; the implementations targeting the Cell and GPU architectures process multiple streams in parallel. 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. K. Atasu, L. Breveglieri, and M. Macchetti. Efficient AES implementations for ARM based platforms. In *Symposium on Applied Computing 2004*, pages 841–845. ACM, 2004.
3. Atmel Corporation. 8-bit AVR Microcontroller with 8/16K Bytes of ISP Flash and USB Controller. Technical Reference Manual, 2008.
4. D. J. Bernstein and P. Schwabe. New AES software speed records. In *INDOCRYPT 2008*, volume 5365 of *LNCS*, pages 322–336, 2008.
5. E. Biham. A Fast New DES Implementation in Software. In *FSE 1997*, volume 1267 of *LNCS*, pages 260–272, 1997.

6. 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/.
7. 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.
8. D. Blythe. The Direct3D 10 system. *ACM Trans. Graph.*, 25(3):724–734, 2006.
9. J. Daemen and V. Rijmen. *The design of Rijndael*. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 2002.
10. J. Dongarra, H. Meuer, and E. Strohmaier. Top500 Supercomputer Sites. http://www.top500.org/.
11. M. Feldhofer, S. Dominikus, and J. Wolkerstorfer. Strong authentication for RFID systems using the AES algorithm. In *CHES 2004*, volume 3156 of *LNCS*, pages 85–140, 2004.
12. Frost & Sullivan. Asia Pacific's Final Wireless Growth Frontier. http://www.infoworld.com/t/networking/passive-rfid-tag-market-hit-486m-in-2013-102.
13. O. Harrison and J. Waldron. Practical Symmetric Key Cryptography on Modern Graphics Hardware. In *USENIX Security Symposium*, pages 195–210, 2008.
14. H. P. Hofstee. Power Efficient Processor Architecture and The Cell Processor. In *HPCA 2005*, pages 258–262. IEEE Computer Society, 2005.
15. E. Käsper and P. Schwabe. Faster and timing-attack resistant AES-GCM. In *CHES 2009*, volume 5747 of *LNCS*, pages 1–17, 2009.
16. K. Klami, B. Hammond, and M. Spencer. ARM Announces 10 Billionth Mobile Processor, 2009. http://www.arm.com/news/24403.html.
17. S. A. Manavski. CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography. In *ICSPC 2007*, pages 65–68. IEEE, November 2007.
18. A. Munshi. The OpenCL Specification. *Khronos OpenCL Working Group*, 2009.
19. 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.
20. NVIDIA. NVIDIA GeForce 8800 GPU Architecture Overview. *Technical Brief TB-02787-001 v0*, 9, 2006.
21. NVIDIA. NVIDIA CUDA Programming Guide 2.3, 2009.
22. D. A. Osvik. Cell SPEED. SPEED 2007, 2007. http://www.hyperelliptic.org/SPEED/slides/Osvik_cell-speed.pdf.
23. D. Otte. AVR-Crypto-Lib, 2009. http://www.das-labor.org/wiki/Crypto-avr-lib/en.
24. J. Owens. GPU architecture overview. In *SIGGRAPH 2007*, page 2. ACM, 2007.
25. B. Poettering. AVRAES: The AES block cipher on AVR controllers, 2006. http://point-at-infinity.org/avraes/.
26. 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.
27. D. Seal. *ARM architecture reference manual: Second Edition*. Addison-Wesley Professional, 2001.
28. M. Segal and K. Akeley. The OpenGL graphics system: A specification (version 2.0). *Silicon Graphics, Mountain View, CA*, 2004.
29. 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.

30. A. Sloss, D. Symes, and C. Wright. *ARM system developer's guide: designing and optimizing system software*. Morgan Kaufmann Pub, 2004.
31. 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.
32. S. Tillich and C. Herbst. Boosting AES Performance on a Tiny Processor Core. In *CT-RSA*, volume 4964 of *LNCS*, pages 170–186, 2008.
33. J. Yang and J. Goodman. Symmetric Key Cryptography on Modern Graphics Hardware. In *Asiacrypt 2007*, volume 4833 of *LNCS*, pages 249–264, 2007.