Performance Analysis of the SHA-3 Candidates on Exotic Multi-Core Architectures

Joppe W. Bos[1] and Deian Stefan[2]

[1] Laboratory for Cryptologic Algorithms, EPFL, CH-1015 Lausanne, Switzerland
[2] Dept. of Electrical Engineering, The Cooper Union, NY 10003, New York, USA

Abstract. The NIST hash function competition to design a new cryptographic hash standard 'SHA-3' is currently one of the hot topics in cryptologic research, its outcome heavily depends on the public evaluation of the remaining 14 candidates. There have been several cryptanalytic efforts to evaluate the security of these hash functions. Concurrently, invaluable benchmarking efforts have been made to measure the performance of the candidates on multiple architectures. In this paper we contribute to the latter; we evaluate the performance of *all* second-round SHA-3 candidates on two *exotic* platforms: the Cell Broadband Engine (Cell) and the NVIDIA Graphics Processing Units (GPUs). Firstly, we give performance estimates for each candidate based on the number of arithmetic instructions, which can be used as a starting point for evaluating the performance of the SHA-3 candidates on various platforms. Secondly, we use these generic estimates and Cell-/GPU-specific optimization techniques to give more precise figures for our target platforms, and finally, we present implementation results of all 10 non-AES based SHA-3 candidates.

Key words: Cell Broadband Engine, Graphics Processing Unit, Hash function, SHA-3

1 Introduction

The design and analysis of cryptographic hash functions have come under renewed interest with the public competition[3] commenced by the US National Institute of Standards and Technology (NIST) to develop a new cryptographic hash algorithm intended to replace the current standard Secure Hash Algorithm-2 (SHA-2) [28]. The new hash algorithm will be called 'SHA-3' and will be subject to a Federal Information Processing Standard (FIPS), similar to the Advanced Encryption Standard (AES) [27]. The competition is NIST's response to recent advances in the cryptanalysis of hash functions, particularly those affecting widely deployed algorithms, including MD5 and SHA-1. Although these breakthroughs have no direct consequence on the current cryptographic hash standard SHA-2, a successful attack on SHA-2 would have catastrophic effects on the security of applications relying on hash functions (e.g., digital signatures).

[3] See http://csrc.nist.gov/groups/ST/hash/sha-3/index.html

Such attacks are believed to be quite probable due to the structural similarities between SHA-2 and its broken ancestors.

Competition History. The NIST competition officially started in late October 2008 with various contributions from academia, industry and government institutions. A total of 64 proposals were submitted worldwide, of which 51 met the minimum submission requirements and were announced as the first-round candidates. Compared to the AES competition, which had 15 candidates, this number was quite large. In late July 2009, NIST narrowed the number of candidates for the second round to a more manageable size of 14. The total number of candidates is expected to be reduced to about 5 (finalists) by the third quarter of 2010. The new hash function standard(s) will be announced in 2012.

Motivation. The candidates are reviewed based on three main evaluation criteria: security, cost, and algorithmic and implementation characteristics [29]. Through the second round, nearly all of the eliminated algorithms were found to suffer from either efficiency or security flaws. Furthermore, despite suffering from minor security issues, some of the high-performing candidates survived the elimination process [35]; this clearly highlights the importance of efficiency in the evaluation procedure.

One of the motivations behind this work is NIST's predisposition for algorithms with greater flexibility [29]; specifically, NIST states that is it preferable if "*the algorithm can be implemented securely and efficiently on a wide variety of platforms.*" We endeavor to evaluate the performance of the remaining candidates on two *exotic* platforms: 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 of equal length are processed at once using SIMD (single instruction, multiple data) and SIMT (single instruction, multiple threads) techniques for the Cell and GPUs, respectively. Due to the low prices, wide availability, and shift in architecture design towards many-core processors [34], it is of valuable interest to evaluate the performance of the Cell and GPUs as cryptologic accelerators.

There are numerous cryptographic applications in which the computation of a message digest of a fixed-length message is necessary. For instance, the work of Bellare and Rogaway [3], standardized in [36,21,1], proposes a mask generation function used in optimal asymmetric encryption that is based on a hash function which takes a fixed-length input. Further, protocols which use hash-based message authentication codes (HMAC) require the computation of a message digest of fixed-length blocks. Specifically, given hash function H, message m, and key k, HMAC is defined as: $H((k \oplus o_{\text{pad}})||H((k \oplus i_{\text{pad}})||m))$. In this case, $||$ denotes concatenation, and o_{pad} and i_{pad} are fixed-length constants such that the outermost hash is of a fixed-length block (cf. [22] for more details). Thus, computing the message digest of a batch of such fixed-length input messages, e.g., in high-end servers, can be efficiently accomplished with the implementations proposed in this work. Additionally, in a cryptanalytic setting such implementations may be used to speed up brute-force password cracking, allow for hash function cube attack/tester analysis using high-dimensional cubes, among many other applications.

Our Contribution. We present a new software performance analysis of all second-round SHA-3 candidates on the Cell and GPU. Our results are three-fold:

1. We present an in-depth performance analysis of all SHA-3 candidates by investigating their internal operations. It is worth noting that the aim of this work is not to claim that our techniques are optimal (hence, the provided estimates are indeed subject to change). Rather, our intended goal is to make a fair, reliable, and accurate comparison between all second-round SHA-3 candidates, which might serve as a reference before the final candidates are announced. Due to the significant number of candidates, all using different techniques, this is not a straightforward task. To facilitate the analysis, we separate the AES-inspired candidates from the others. For the former case, we make extensive use of the work by Osvik et al. [33], which introduced the fastest results of AES on our target architectures. For the latter case, however, a more careful analysis, starting from scratch, is required.
2. We propose specific optimization techniques for each of our target platforms; in combination with our estimation framework, more precise estimates per architecture are given for all second-round SHA-3 candidates.
3. We complement this framework by providing real implementations of all non-AES based candidates on the target platforms. We show that our techniques are indeed applicable, and that the base estimates are usually realistic.

Related Work. The PlayStation 3 (PS3) video game console, which contains the Cell architecture, has been previously used to find chosen-prefix collisions for the cryptographic hash function MD5 [37]. Fast multi-stream implementations of MD5, SHA-1 and SHA-256 for the Cell are presented in [10]; from this work, we use the performance numbers for SHA-256 as a comparison to the performance of the SHA-3 candidates, as they outperform the single stream results from [13] by an order of magnitude. Graphics cards have similarly been used for MD5 collision searches [8], password cracking [26], and accelerating cryptographic applications [38,25]. To the best of our knowledge, there is no previous work implementing second-round SHA-3 candidates on the Cell architecture or NVIDIA GT200 GPUs.

Organization. We start with a brief introduction to our target platforms in Section 2. Several optimization techniques are described in Section 3, directly addressing our main target architectures. Then, in Section 4 and 5 we introduce our performance analysis and implementation results on AES-inspired and other second round candidates, respectively. We conclude in Section 6.

2 Target Platforms

Cell Broadband Engine Architecture. The Cell architecture [20], jointly developed by Sony, Toshiba, and IBM, is equipped with one dual-threaded, 64-bit in-order Power Processing Element (PPE) based on the Power 5 architecture and 8 Synergistic Processing Elements (SPEs). Our interest is in the SPEs [39], the main computational cores of the Cell. Each SPE consists of a Synergistic Processing Unit (SPU), 256 KB of private memory called Local Store (LS), and a Memory Flow Controller (MFC). To avoid the complexity of sending explicit direct memory access requests to the MFC, all code and data must fit within the LS.

The SPU is equipped with a large register file containing 128 registers of 128 bits each. Most SPU instructions work on 128-bit operands denoted as *quadwords*. The instruction set is partitioned into two sets: one set consists of (mainly) 4- and 8-way SIMD arithmetic instructions, while the other consists of instructions operating on the whole quadword (including the load and store instructions) in a single instruction, single data (SISD) manner. The SPU is an asymmetric processor; each set of instructions is executed in a separate pipeline, denoted by the *even* and *odd* pipeline for the SIMD and SISD instructions, respectively. For instance, the {4, 8}-way SIMD left-rotate instruction is an even instruction, while the instruction left-rotating the full quadword is dispatched into the odd pipeline. When dependencies are avoided, a single pair of even and odd instructions can be dispatched every clock cycle.

One of the first applications of the Cell processor was to serve as the heart of Sony's PS3 game console. Although the Cell contains 8 SPEs, in the PS3, one is disabled and a second is reserved by Sony. Thus, with the first generation PS3s the programmer has access to six SPEs, this has been disabled in the current version of the game console. In subsequent applications, serving the supercomputing community, the Cell has been placed in blade servers, with newer variants containing the PowerXCell 8i, a derivative of the Cell that offers enhanced double-precision floating-point capabilities. The SPEs are particularly useful as (cryptographic) accelerators. For this purpose, PCIe cards are available (either equipped with a complete Cell processor or a stripped-down version containing 4 SPEs) so that workstations can benefit from the computational power of the SPEs.

NVIDIA Graphics Processing Units. Unlike the Cell, there are many different GPU architectures, though, most share the primary goal of accelerating 3-dimensional graphics (rendering) applications, such as games. In this work, we focus on programming NVIDIA GPUs using the Compute Unified Device Architecture (CUDA) extension of the C language. With the latest GPUs implementing the Fermi architecture [32], availability and interest in the older G80 series GPUs, which have also been used for cryptologic applications (cf. [25,33,19]), is rapidly decreasing. We therefore restrict our focus to the more-recent GT200 series GPUs.

Each GPU is equipped with several Simultaneous Multiprocessors (SMs), varying from 24 in the GTX 260 to 30 in each of the GPUs of the GTX 295 graphics card. Each SM consists of a large register file (16384 32-bit registers), fast 16-way banked on-chip 16KB shared memory, 8 Scalar Processors (SPs), 2 special function units (used for transcendentals), an instruction scheduler, and (6-8KB) texture and (8KB) constant memory caches. The SPs are capable of executing many instructions, including 32-bit integer arithmetic and bitwise operations, which can be used to implement most cryptologic algorithms.

Although explicit SIMD access of the SM compute units (the SPs) is desirable for many applications, the programmer is limited to writing parallel code at the thread level [31]. Specifically, using CUDA, the programmer writes code for a *kernel* which is executed by many *threads* (all executing the same instructions of the kernel, though operating on different data) on the SPs. In the SIMT programming model, threads are grouped into a *thread block*, which is executed on a single SM and, consequently, these threads may synchronize execution and use the shared memory to communicate. When launching a kernel, it is common (and highly recommended) to execute multiple thread blocks, grouped in a *grid*, which the hardware then assigns to the available SMs; to hide various latencies, it is recommended that at least 2 blocks be available for scheduling on each SM [31]. Note that although each SM has many resources, the shared memory is divided among the 'co-located' thread blocks, and similarly the registers are divided among the individual threads—careful consideration of an application's use of these resources is critical when trying to achieve high performance. Despite these design 'restrictions', GPUs are very commonly being used as accelerators for workstations, given their wide availability as moderately-priced PCIe cards.

3 Porting the SHA-3 Candidates to the Cell and GPU

Cell Broadband Engine Architecture. On the SPE architecture, all distinct binary operations $f : \{0,1\}^2 \to \{0,1\}$ are available, making it a suitable platform to implement hash functions. Operations frequently used by the hash candidates, such as rotations, shifts, and additions, are available as 4-way SIMD instructions operating on the 4 32-bit words of a quadword, in parallel. When possible, we use the 32-bit optimized reference code of the SHA-3 candidates as a base and further optimize this code for the SPE architecture.

To make the code more suitable for execution on the Cell, the use of branches is eliminated or reduced to a minimum, since all four input strings need to be processed in an identical way. Most of the instructions used in the various compression functions are arithmetic instructions, which go in the even pipeline. When naively porting the code to the SPE architecture, this results in a highly unbalanced implementation where the odd pipeline is underutilized. In order to improve performance, some even operations, when feasible, are implemented by a sequence of odd instructions (following a similar approach to that described in [33]). This increases the latency of this operation, but if these instructions can be dispatched for free with the surrounding even instructions, the overall number of cycles decreases (while the number of overall instructions increases).

One obvious way to do this is to make use of the `shuffle` instruction that is dispatched in the odd pipeline. The `shuffle` instruction can pick any 16 bytes of the 32-byte (two 128-bit registers) input or select one of the byte-constants {`0x00, 0xFF, 0x80`} and place them in any of the 16-byte positions of the 128-bit output register. For example, when a 4-way SIMD shift or rotate by x (to the left or right) is required this is typically implemented using the even `shift` or `rotate` instruction. When $x \equiv 0 \bmod 8$, this is simply a reordering of bytes, and can be done for 4 32-bit integer values in parallel using the `shuffle` instruction.

Converting a 4-way SIMD left rotation of a quadword V by $x \not\equiv 0 \bmod 8$ bits to odd instructions can be done using two odd `shuffle` and two odd quadword `shift` instructions. When using an odd quadword `rotate` operation, the bits rotated out from each 32-bit boundary are dislocated. To address this, create a quadword W which contains, on byte positions $4i$ and $4i+1$, the values from the byte location $3+4i$ and $4i$ from V respectively, where $0 \leq i \leq 3$ and the most (least) significant byte position is labeled as 0 (15). The other bytes in W, at byte positions $4i+2$ and $4i+3$, are set to zero. Next, V and W are shifted left by $x \bmod 8$ using the odd quadword `shift` instruction. Finally, shuffle the three bytes from V and single byte from W per word to the correct positions to complete the 4-way SIMD rotation. This technique allows one to trade 1 even `rotate` instruction for 4 odd instructions. Note that the latency of the operation has increased from 4 cycles for the even `rotate` to $4 \times 4 = 16$ for the odd variant.

One of the NIST submission requirements is to provide an implementation of the SHA-3 candidate suitable to run on a 32- and 64-bit platform [29]. However, some of the candidates, e.g., Skein, provide a 32-bit implementation which requires the use of a 64-bit data type in the compression function. This requires to implement fast 64-bit additions and rotations built from 32-bit instructions, since these operations, on the SPE, are only available in 32-bit flavors. A 2-way SIMD addition can be implemented as follows. First, a 4-way SIMD carry generation (even) instruction is used to provide the carries going from the least to the most significant 32-bit word. An odd `shuffle` instruction is then used to put the two carries in the correct position, while the other two carries corresponding to the most significant 32-bit word of each 64-bit integer are ignored. Finally, the 4-way SIMD extended addition, an addition with carry, is used to add the two quadwords consisting of four 32-bit values considering the carries. Thus, a 64-bit addition can be implemented using a single odd and two even instructions.

To implement an efficient 2-way SIMD rotate, the `select` instruction, which is dispatched in the even pipeline, is used when a rotation by $x \not\equiv 0 \bmod 8$ is required. 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. The approach is to first perform a full quadword rotation to the left by x bit-positions and store this in a quadword V_1. Then, put the incorrectly-positioned rotated bits in the correct positions of a separate quadword V_2 by swapping the 64-bit double-words. Use the `select` instruction to get the correct bits from the two quadwords, using a pattern, defined by concatenating twice the 64-bit unsigned integer value $2^x - 1$, selecting the corresponding bit position from V_1 or V_2 if the bit position in the pattern is set to zero or one respectively. Since the SPE architecture has a quadword rotation instruction up to 7 bits and another instruction rotating by bytes, the 2-way SIMD rotation costs 3 odd rotations and one even selection for rotating by $x > 8$. When $x < 8$, the cost is reduced by one odd rotation.

NVIDIA Graphics Processing Units. Compared to the SPE instruction set architecture (ISA), the GPU parallel thread execution (PTX[4]) ISA [30] is considerably less rich. With respect to integer arithmetic operations, programmers have access to 32-bit bitwise operations (`and`, `or`, `xor`, etc.), left/right shifts, 32-bit additions (with carry-in and carry-out), and 32-bit multiplication (sometimes implemented using several 24-bit multiplication instructions).

Given the simplicity of PTX, to gain the most speedup from the raw computational power, it is imperative that the kernels be very compact (especially with respect to register utilization and shared memory allocation). Compact and non-divergent kernels allow for the execution of more simultaneous threads, and can thus increase the performance of the target hash function. Thus, when implementing common hash function building blocks, a simple approach is also usually the most optimal. For example, a rotation of a 32-bit word is implemented using two shifts (`shl` and `shr`), and an `or` instruction. Furthermore, for many hash functions we can store the full internal state, and sometimes even the input message block, in registers. Although this limits the number of simultaneous threads per SM, it also lowers the copies to and from (shared) memory and thereby contributes to a faster implementation, overall. Additionally, when possible, we manually unroll the compression functions since branching on the SMs can lead to a degradation in performance when threads of a common thread block take divergent paths and execution is serialized. Moreover, conditional statements consisting of a small number of operations in each basic block are implemented using predicate instructions, instead of branches—PTX allows for the predication of almost all instructions. Nevertheless, when branching is necessary (e.g., the compression function of Skein-512), the thread execution is synchronized (at a barrier near the branch) and the branch instruction is executed uniformly by all the threads.

For algorithms with small-to-medium sized chain values (e.g., 256- or 512-bits), we buffer the chain values in registers. To avoid multiple kernel launches, each thread processes multiple message blocks. This, in conjunction with the caching of the chaining values, not only simplifies the multi-block hashing, but also results in a faster implementation (than, for example, executing multiple kernels and having to read/write chain values from/to global memory). For algorithms with larger-sized chain values or internal states, we cache the chain values in shared memory. In implementing algorithms that use shared memory, we require that the thread block size always be a multiple of 16 threads (usually at least 64 threads) and further (implicitly) assert that the n-th thread (counting from 0) loads/stores any shared memory cached values from/to bank $n \bmod 16$, as to avoid bank conflicts.

When considering algorithms using 64-bit operations, the number of registers and instructions usually doubles. For example, a 64-bit addition is performed using two additions with carry (`add.cc`). Similarly, rotations by $x \not\equiv 0 \bmod 32$ is implemented using 4 `shift` and 2 `or` 32-bit instructions. For these algorithms,

[4] We note that the PTX is an intermediate description and not the actual GPU ISA. The latter is not publicly available.

Table 1. The number of AES-like operations per b bytes for all AES-inspired candidates and the performance estimation on the SPE and single GTX 295 GPU. `(R)`: One AES encryption round, `SB`: Substitution operation, `MC`X: Mix-Column operation over X bytes (i.e., X=4 is identical to the one used in AES). Note that Shift-Row operations are ignored because it can be dispatched through the Mix-Column operation. `C/B`: Cycles per byte, `Gb/sec`: 10^9 bits per seconds. The SPE estimates do not use the T-table approach.

Hash function	b	`(R)`	`SB`	`MC4`	`MC8`	`MC16`	`xor` (byte)	SPE		GPU	
								`C/B`	`Gb/sec`	`C/B`	`Gb/sec`
SHA-256 [10]	-	-	-	-	-	-	-	8.2	3.1	-	-
AES-128 [33]	16	10	-	-	-	-	16	11.3	2.3	0.32	30.9
ECHO-256	192	256	-	512	-	-	448	29.6	0.9	0.85	11.7
Fugue-256	4	-	32	-	-	2	60	15.1	1.7	0.62	16.1
Grøstl-256	64	-	1280	-	160	-	1472	41.4	0.6	1.23	8.1
SHAvite-3-256	64	52	-	-	-	-	1280	16.5	1.6	0.42	23.7

rather than using expensive registers to cache chain values or message blocks, we resort to using shared memory for caching. We, again, stress that the restriction on shared memory bank access applies to all our algorithms, and thus a 64-bit cache value requires 2 (non-conflicting) memory accesses per 64-bit word.

4 AES-Inspired SHA-3 Candidates

A popular design choice of the SHA-3 hash function designers was to use AES-like byte oriented operations (and, in some cases the AES round function itself) as building blocks in the compression function of their hash function. The second-round SHA-3 candidates following this paradigm include ECHO [4], Fugue [18], Grøstl [16], and SHAvite-3 [9]. The motivation for using AES-like operations is mainly because AES has successfully withstood much cryptanalytic effort and, moreover, one can exploit the high capabilities of AES-like functions on a wide variety of architectures. Moreover, many of the design teams have pointed out the new Intel AES instruction set and claimed several performance figures outperforming the other candidates (for a more detailed analysis, cf. [5]). Considering the possible widespread use of these processors in the future, these designs will likely have a clear advantage.

Although several optimization methods for these hash functions are possible on particular processors, such as using the Intel AES instruction set, we analyze the performance of AES-inspired candidates in a more generic setting. More precisely, we simply count the number of 'AES-like' operations required for the compression function of each candidate, as this gives an intuition of how these designs behave in architectures without native AES-instructions, such as the PowerPC, SPARC, and most low-power microcontrollers. Table 1 provides these rough estimates. Note that since the operations may differ per candidate, we clearly differentiate all possibilities, particularly the variants of the 'Mix-Column' (MC) operation used in AES.

Table 2. Straight-forward estimates for the different mix-column operations without (left) and with (right) the use of T-tables. Note that the `xor` and `rotate` instruction counts for the T-table approach in MCX operate on $(8 \cdot X)$-bit values.

		`XTIME`	`xor` (byte)	size of table(s) in bytes	`xor`	`rotate`
MC4	(AES)	4	16	1,024	3	3
				4,096	3	0
MC8	(Grøstl)	16	104	2,048	7	7
				16,384	7	0
MC16	(Fugue)	32	148	4,096	15	15
				65,536	15	0

The estimates given in Table 1 provide a good indication on the performance of the AES-inspired candidates, especially for hashing extremely long messages, where we simply focus on the compression functions. It should, however, be noted that the techniques used to implement the MC operations used by these candidates account for the largest performance loss/gain. Typically, the MC operation is implemented using a number of `xor` operations and the `XTIME` function. The latter treats a byte-value as a polynomial in the finite field $\mathbf{F}_{2^8}$ and performs modular multiplication by a fixed modulus and multiplier. In practice, `XTIME` can be implemented using a `shift` and a conditional `xor`. An upper bound on the required MC-operations, working on single byte-values, is given in Table 2. First, the double and quadruple of the X elements are computed in MCX for $X \in \{8, 16\}$; the octuple for MC16 is not needed since all the constants in Fugue are below 8. We note that these require $2 \cdot X$ `XTIME` operations, and that the number of required `xor` operations depend on the constants. Counting the latter, for MC4 in AES and MC8 in Grøstl, there are at most $4 \times 5 - 4 = 16$ and $14 \times 8 - 8 = 104$ `xor` instructions, since the rows are simply rotations of each other. Similarly, in Fugue there are $4 \times (10+8+14+9-4) = 148$ `xor` instructions, corresponding to its constants. We stress that these (naive) estimates should be treated as an upper bound; as illustrated by the implementation of MC4 in [33], the number of times `XTIME` and `xor` are required is lower: 3 and 15, respectively.

Following the "T-table" approach [14], the MC and substitution steps can be performed by using lookup tables on 32-bit (and larger) processors. The use of T-tables can greatly reduce the number of required operations; estimates of the cost of the different MC steps using a varying number of T-tables (as the different tables are simply rotations of each other) are also stated in Table 2. The MCX T-table variants require $X - 1$ `xor`, and 0 or $X - 1$ `rotate` instructions (depending on the number of tables used) operating on X-byte values. The use of T-tables is, however, not always favorable where, for example, in memory constraint environments, the tables might be too big. This is also the case for certain SIMD environments, such as the SPE, where as indicated in [33], fetching data for multiple streams in parallel is not trivial and may be more expensive than actually computing the MC operation.

Among the four AES-inspired second-round SHA-3 candidates, ECHO and SHAvite-3 make use of the AES round itself and can highly benefit from Intel AES instruction set. Therefore, it is relatively easy to infer the speed estimates for these two hash functions once we have those for AES. We use the recent work by Osvik et al. [33] on AES to obtain estimates for our target platforms. Based on their results, the corresponding workload required to implement the compression function of the AES-inspired candidates is given in Table 1. As an example of how SHAvite-3 performs under this result (given the estimates of Table 1), one requires 52 AES round function evaluations plus 1280 8-bit `xor`s to perform one compression function invocation of SHAvite-3, compressing a 64 byte message block. From [33] we learn that one AES round can be implemented in 300 and 78600 cycles on the SPE and GPU when hashing 16 simultaneous streams and 600 blocks of 256 streams, respectively. Hence, SHAvite-3 is estimated to achieve performance of $\frac{52 \cdot 300 + 1280}{64 \cdot 16} = 16.5$ cycles/byte on a single SPE, and $\frac{52 \cdot 78600 + 1280}{64 \cdot 256 \cdot 600} = 0.42$ cycles/byte on a single GTX 295 GPU.

We note that the performance estimates given in Table 1 for Grøstl and Fugue are conservative. This is because the naive estimates for MC8 and MC16 use the estimate from Table 2, leaving room for significant optimizations. These numbers can be further improved on platforms where a T-table approach is faster than computing the Mix-Column operation. For example, on the GPU, placing the smaller (2KB) table in shared memory, Grøstl would require two 32-bit lookups in addition to the 7 `xor` and 7 `rotate` (64-bit) instructions.

5 Other SHA-3 Candidates

The non-AES based SHA-3 candidates use a variety of techniques and ideas in their hash function designs. From a performance perspective, it is interesting to have an indication of the number of required instructions per byte. An approximation of this is given in Table 3. We note that operations ending with a '`c`' indicate that one of the input parameters is complemented before use, `eqv` denotes bitwise equivalence (i.e., `xorc`) and `csub` denotes conditional subtraction. These *raw* instruction counts are obtained from the optimized implementations as submitted to NIST and only the number of instructions in the compression function are considered. Since load and store operations are hard to predict (due to possible cache misses), and may be incomparable between platforms, only arithmetic instructions are taken into account (i.e., the required moves, loads/stores, including all the possible table-lookups, are ignored).

We would like to stress that the performance figures presented in Table 3 are estimates for a hypothetical 32-bit architecture, the instruction set of which includes all the operations shown in the columns of Table 3. Moreover, we assume that such a machine can dispatch one instruction clock cycle. Estimating the actual performance number on modern platforms is considerably more difficult because they often have access to a separate SIMD unit, which is ignored by our estimates. However, these estimates can be used as a starting point to create more accurate platform-specific speed estimations, for instance for the Cell and

Table 3. Performance estimates for all non-AES inspired SHA-3 candidates based on the number of 32- and 64-bit arithmetic instructions used in the various compression functions (which process b bytes). The † indicates an alternative implementation approach (on-the-fly interleaving) for Keccak. We assume that all operations stated in the columns are single instruction operations.

Hash function	b	`add`	`sub` *csub*	`mul`	`and`	`nand` *andc*	`eqv`	`or` *orc*	`rotate`	`shift`	`xor`	Cycles / byte
Hash functions operating on 32-bit words												
BLAKE-32	64	480	-	-	-	-	-	-	320	-	508	20.4
BMW-256	64	296	58	-	-	-	-	-	212	144	277	15.4
CubeHash-16/1	1	512	-	-	-	-	-	-	512	-	512	1536.0
CubeHash-16/32	32	512	-	-	-	-	-	-	512	-	512	48.0
Hamsi-256	4	-	-	-	24	12	-	24	72	24	287	110.8
JH-256	64	-	-	-	1792	*1152*	288	688	-	800	4024	136.6
Keccak-256	136	-	-	-	684	*96*	144	480 *144*	1248	204	3810	50.1
Keccak-256†	136	-	-	-	756	384	-	624	1248	360	4224	55.9
Luffa-256	32	-	-	-	144	-	96	96	392	-	756	46.4
Shabal-256	64	52	16	96	-	*48*	48	-	112	-	242	9.6
SIMD-256	64	817	901 *256*	419	852	-	-	256	288	804	176	74.5
Hash functions operating on 64-bit words												
Skein-512	64	497	-	-	1	-	-	-	288	-	305	17.0

GPU architectures. Note that while the multiplications by the candidate SIMD operate on 16-bit operands, the multiplications in Shabal are by one of the constants $\{3, 5\}$. Each of the latter multiplications can be converted into a shift and addition, if cheaper than native multiplication.

Cell Broadband Engine Architecture. Ignoring moves and assuming perfect circumstances, i.e., all even and odd pairs of instructions can be dispatched simultaneously without stalls, an estimate for hashing four messages of equal length in parallel on a single SPE may be obtained by dividing the performance numbers in Table 3 by a factor of four. Note that these are pessimistic estimates, as the balancing techniques from Section 3 are not (implicitly) considered. In Table 4 we present actual implementation results of all non-AES based candidates, with the fine-tuned estimates in parentheses. The performance results are obtained by hashing thousands of long messages (25 KB) and measuring the complete hash function (not only the compression function), in addition to the benchmarking overhead.

The number of shifts and rotations which are replaced by their odd variants is often close to the expected value required to balance the number of odd and even instructions. It might happen that this introduces stalls due to instruction dependencies, the optimal number of operations which are replaced is then decided experimentally. This information is taken into account in the estimates in Table 4. For some candidates, additional optimizations are possible.

Table 4. Performance results and estimates (in parentheses) for the non-AES based SHA-3 candidates for the SPE and the GPU architecture. The SPE implementations process four or two (for Skein) messages of equal length. The GPU implementations process 680 blocks of 64 threads on a single NVIDIA GTX 295 GPU. Measurements of only the compression function are shown in [brackets].

Algorithm	SPE		GPU	
	Cycles per byte	Throughput (Gb/sec)	Cycles per byte	Throughput (Gb/sec)
SHA-256 [10]	8.2	3.1	-	-
[2] BLAKE-32	5.0 (4.5)	5.1 (5.7)	0.27 [0.13] (0.13)	36.8 (76.4)
[17] BMW-256	4.2 (3.7)	6.2 (6.9)	0.27 [0.27] (0.10)	36.8 (99.4)
[6] CubeHash-16/1	326.7 (316.0)	0.1 (0.1)	11.1 [11.0] (10.9)	0.90 (0.91)
[6] CubeHash-16/32	11.6 (9.9)	2.2 (2.6)	0.36 [0.35] (0.34)	27.6 (29.2)
[23] Hamsi-256	32.2 (26.9)	0.8 (1.0)	5.19 [0.66] (0.64)	1.91 (15.5)
[40] JH-256	31.5 (29.8)	0.8 (0.9)	0.76 [0.75] (0.67)	13.1 (14.8)
[7] Keccak-256	13.0 (11.1)	2.0 (2.3)	0.56 [0.56] (0.31)	17.7 (32.1)
[12] Luffa-256	11.5 (10.1)	2.2 (2.5)	0.35 [0.34] (0.32)	28.4 (31.1)
[11] Shabal-256	3.5 (2.8)	7.2 (9.2)	0.69 [0.56] (0.07)	14.4 (141.9)
[24] SIMD-256	22.6 (19.0)	1.1 (1.4)	3.60 [3.60] (0.43)	2.76 (23.1)
[15] Skein-512	13.7 (12.1)	1.9 (2.1)	0.46 [0.29] (0.22)	22.1 (45.2)

For instance, the candidate SIMD uses the select operation (bitwise "if X then Y else Z"), $(X \wedge Y) \oplus (\bar{X} \wedge Z)$, and the majority operation on three operands, $(X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z)$, which can be implemented using one and two `select` instructions, respectively. This optimization is counter-balanced by the fact that the conditional subtraction requires three instructions (a comparison, subtraction and a select) to avoid branching. Another example where instructions on the SPE can be saved is in JH: the swapping of (multiple) bytes requires just a `shuffle` instruction, and the swapping of bits requires two `shift` and a single `select`.

We observe that one of the main reasons the actual performance numbers are slightly higher than the given estimates is that the four input streams of bytes need to be converted to a 4-way SIMD representation. This introduces noticeable overhead, similar to all candidates, which is not accounted for in the estimates. In Hamsi, the overhead is even larger because the message-data is used as an index for a table look-up which further gives rise to extra arithmetic instructions needed to calculate the locations of the loads. Doing this in 4-way SIMD, even when pre-fetching data for subsequent blocks, introduces ample overhead that is not considered in our estimates since all load and store operations are ignored.

NVIDIA Graphics Processing Units. As discussed in Section 3, the PTX ISA is considerably more limited than the Cell's ISA, and therefore some of the instructions in Table 3 will have to be implemented by multiple, simpler, instructions. For example, each `rotate` is implemented using two `shift` instruction and an `or`; each `andc` is implemented using a `not` and an `and`, etc. Taking the implementation of these non-native instructions into account, in addition to the fact that each GPU on the GTX 295 contains 30 SMs (for a total of 240 SPs) we divide the (slightly higher) instruction count of Table 3 by a factor of 240. These estimates are presented in Table 4, along with actual implementation results.

As in the Cell, the GPU estimated performance results of Table 4 do not account for message memory-register copies or moves. Furthermore, they do not account for kernel launch overhead, host-to/from-device copies, or possible table-setup timings (e.g., copying a table to shared memory). For fair comparison, we, however, do account for the chain value copies to/from registers and global memory; this rough figure was measured for the different sizes using a kernel that simply copied the state to registers and back to global memory. Nevertheless, our GPU estimates are certainly optimistic and implementation results, measuring the full hash function, are higher. Additionally, for algorithms with huge internal states or expanded messages, e.g., SIMD, the use of local storage might not be easily avoided and the implementation results are expected to be much worse than the estimates.

Along with considering the techniques of Section 3 when implementing the candidates, we further emphasize the details of Keccak and Hamsi. Since using large tables on the GPU is prohibited, we estimate and implement Keccak with on-the-fly interleaving (Keccak-256† in Table 4) and divide the execution of Hamsi into two kernels. The latter requires the use of a very large 32KB table (which is larger than all the fast memories on the SMs) for the message expansion, and, thus, necessitates a less direct implementation approach. The proposed two-part approach requires: (i) a kernel in which 16 threads expand the 32-bit message to 256-bits (each using 2 $1KB$ tables and an atomic `xor`), and (ii) a kernel implementing the actual compression function. Because the message expansion requires random access reads and uses atomic instructions (to global memory), estimates without considering the effects of memory operations are expected to diverge.

As expected, we observe that the actual performance numbers in Table 4 are slightly higher than the corresponding estimated figures. In most cases, however, the performance overhead is a result of the memory copies (host-to-device and global memory-to-registers). We confirmed this conjecture by measuring the throughput of the compression functions working on a single message block, the results of which are shown [in brackets] in Table 4. We note that the implementation result of SIMD does not, however, agree with our estimated figure—we attribute the extremely low performance to using local memory for the message expansion (4096 bits) and having a single thread do the full compression; splitting the compression function across multiple threads would likely improve SIMD's performance. Additionally, we highlight the Shabal implementation, for which we heavily used the optimized reference code, required the use of a non-inline function in the permutations as to address a compiler optimization bug; the fully-inlined, but buggy, implementation is twice as fast.

6 Conclusion

Efficiency of hash function algorithms is a very important design criterion, almost parallel with security. This work presents a generic framework for analyzing and evaluating the performance of such algorithms; specifically, we estimate the performance of the second-round candidates in the ongoing competition to establish a new cryptographic hash standard, SHA-3. Using this framework as a base, we then take advantage of platform-specific optimization techniques to provide more precise performance estimates for two *exotic* many-core architectures: the Cell Broadband Engine and NVIDIA Graphics Processing Units. We further support our analysis by presenting multi-stream implementation results of all the non-AES based candidates. Finally, we believe that this work can assist in the decision process of the SHA-3 competition.

Acknowledgements. We gratefully acknowledge useful suggestions by Dag Arne Osvik, Onur Özen and the CHES reviewers.

References

1. American National Standards Institute. ANSI X9.44-2007: Key Establishment Using Integer Factorization Cryptography, 2007.
2. J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan. SHA-3 proposal BLAKE, 2008.
3. M. Bellare and P. Rogaway. Optimal asymmetric encryption. In *Eurocrypt 1994*, volume 950 of *LNCS*, pages 92–111, 1994.
4. R. Benadjila, O. Billet, H. Gilbert, G. Macario-Rat, T. Peyrin, M. Robshaw, and Y. Seurin. SHA-3 Proposal: ECHO, 2009.
5. R. Benadjila, O. Billet, S. Gueron, and M. J. B. Robshaw. The Intel AES instructions set and the SHA-3 candidates. In *Asiacrypt 2009*, volume 5912 of *LNCS*, pages 162–178, 2009.
6. D. J. Bernstein. CubeHash specification (2.B.1), 2009.
7. G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Keccak specifications, 2009.
8. M. Bevand. MD5 Chosen-Prefix Collisions on GPUs. Black Hat, 2009. Whitepaper.
9. E. Biham and O. Dunkelman. The SHAvite-3 Hash Function, 2009.
10. J. W. Bos, N. Casati, and D. A. Osvik. Multi-Stream Hashing on the PlayStation 3. In *PARA*, LNCS, 2008. To appear, http://documents.epfl.ch/users/b/bo/bos/public/PARA2008.pdf.
11. E. Bresson, A. Canteaut, B. Chevallier-Mames, C. Clavier, T. Fuhr, A. Gouget, T. Icart, J.-F. Misarsky, M. Naya-Plasencia, P. Paillier, T. Pornin, J.-R. Reinhard, C. Thuillet, and M. Videau. The Hash Function Shabal, 2008.
12. C. D. Canniere, H. Sato, and D. Watanabe. Hash Function Luffa, 2009.
13. T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation: A performance view. http://www.ibm.com/developerworks/power/library/pa-cellperf/, November 2005.
14. J. Daemen and V. Rijmen. *The design of Rijndael.* Springer-Verlag New York, Inc. Secaucus, NJ, USA, 2002.
15. N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker. The Skein Hash Function Family, 2009.

16. P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S. S. Thomsen. Grøstl – a SHA-3 candidate, 2008.
17. D. Gligoroski, V. Klima, S. J. Knapskog, M. El-Hadedy, J. Amundsen, and S. F. Mjolsnes. Cryptographic Hash Function BLUE MIDNIGHT WISH, 2009.
18. S. Halevi, W. E. Hall, and C. S. Jutla. The Hash Function Fugue, 2009.
19. O. Harrison and J. Waldron. Practical Symmetric Key Cryptography on Modern Graphics Hardware. In *USENIX Security Symposium*, pages 195–210, 2008.
20. H. P. Hofstee. Power Efficient Processor Architecture and The Cell Processor. In *HPCA 2005*, pages 258–262. IEEE Computer Society, 2005.
21. IEEE Std 1363-2000. *IEEE Standard Specifications for Public-Key Cryptography*. IEEE, New York, 2000.
22. H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, IETF, 1997.
23. O. Küçük. The Hash Function Hamsi, 2009.
24. G. Leurent, C. Bouillaguet, and P.-A. Fouque. SIMD Is a Message Digest, 2009.
25. S. A. Manavski. CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography. In *ICSPC 2007*, pages 65–68. IEEE, November 2007.
26. S. Marechal. Advances in password cracking. *Journal in Computer Virology*, 4(1):73–81, 2008.
27. NIST. FIPS-197: Advanced Encryption Standard (AES), 2001. http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf.
28. NIST. Secure hash standard. FIPS 180-2, http://www.itl.nist.gov/fipspubs/fip180-2.htm, August 2002.
29. NIST. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. Technical report, Department of Commerce, http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf, November 2007.
30. NVIDIA. NVIDIA Compute. PTX: Parallel Thread Execution, March 2008.
31. NVIDIA. NVIDIA CUDA Programming Guide 2.3, 2009.
32. NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. *Whitepaper*, September 2009.
33. D. A. Osvik, J. W. Bos, D. Stefan, and D. Canright. Fast software AES encryption. In *FSE 2010*, volume 6147 of *LNCS*, pages 75–93, 2010.
34. D. Patterson and J. Hennessy. *Computer organization and design: the hardware/software interface*. Morgan Kaufmann, 2008.
35. A. Regenscheid, R. Perlner, S. jen Chang, J. Kelsey, M. Nandi, and S. Paul. Status report on the first round of the SHA-3 cryptographic hash algorithm competition. Technical Report 7620, NIST, http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/documents/sha3_NISTIR7620.pdf, September 2009.
36. RSA Laboratories. PKCS #1 v2.1: RSA Cryptography Standard, 2002.
37. M. Stevens, A. Sotirov, J. Appelbaum, A. Lenstra, D. Molnar, D. A. Osvik, and B. de Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In *Crypto 2009*, volume 5677 of *LNCS*, pages 55–69, 2009.
38. R. Szerwinski and T. Güneysu. Exploiting the power of GPUs for asymmetric cryptography. In *CHES 2008*, volume 5154 of *LNCS*, pages 79–99, 2008.
39. 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.
40. H. Wu. The Hash Function JH, 2009.