

Accelerating Cryptographic Primitives with GPUs

Patrick Carpenter
Auburn University
Shelby. Ctr. Engr. Tech. Suite 3103
Auburn University, AL 36849
+1 (334) 328-9722
carpept@auburn.edu

ABSTRACT

In this paper, we review the current state-of-the-art in accelerating cryptographic and other computer-security-related primitives using graphics processing units and provide a critical analysis of the appropriateness of graphics accelerators to this task. General-purpose programming of graphics processing units (GPGPUs) has garnered much attention recently in the high-performance computing community, as it offers orders-of-magnitude performance benefits for data-parallel processing tasks. Furthermore, the modern histories of cryptography and high-performance, or super, computing have been non-trivially related since the Second World War. An observation of the computer security community's recent interest in GPGPU seems to indicate that the relationship between high-performance computing and computer security is destined to continue in the near future. First, we introduce necessary prerequisites of graphics processing units and their use for general-purpose, high-performance computation, including their historical development and NVIDIA's now-dominant CUDA programming model. We proceed to discuss and analyze a variety of cryptographic primitives, including cryptographically secure hashing functions and symmetric/asymmetric encryption. Next, we sample the literature for examples of the success – and, possibly failure – of using GPUs to accelerate these algorithms. Finally, analyze what implications GPU acceleration might have on the field of computer security, suggest potential alternative uses of GPUs for accelerating cryptographic or other security-related methods (e.g., steganography) and draw conclusions based on latest research results.

Categories and Subject Descriptors

D.1.3 [Programming Techniques]: Concurrent Programming – *parallel programming, distributed programming*. E.3 [Data Encryption] Data Encryption – *Data Encryption Standard, public key cryptosystems, standards*

General Terms

Algorithms, Performance, Design, Experimentation, Security

Keywords

GPGPU, CUDA, Cryptography, Computer Security, High-Performance Computing

1. INTRODUCTION

The objective of this paper is to acquaint the reader with recent advances in the use of graphics processing units for high-performance, general computation and, in particular, with some of the ways in which graphics devices are being used to accelerate important applications and algorithms from the field of computer security, especially cryptography. Since circa 2005, there has been growing interest to exploit graphics devices for general-purpose, non-graphics calculations in many numerically-demanding applications areas, notably including computational science and finance. There are at least four features shared by modern graphics processors which are desirable from the point of view of high-performance computing: first, graphics processors offer orders of magnitude speedup for many applications; second, they provide astonishing thread- and memory-level parallelism; third, they are commodity – rather than purpose-built – devices, which gives them a strong economical edge; finally, they are designed to provide best performance for large, parallel, data- and arithmetic-intensive calculations, a category of problems which corresponds to a wide array of problems in many fields of applied computing. An application area which is seeing increasing adoption of GPGPU to accelerate a variety of tasks is computer security and, more specifically, those areas of computer security which deal with numerical and semi-numerical algorithms for cryptographic primitives. Three areas in particular – hashing, symmetric encryption and asymmetric encryption – have received significant amounts of attention in recent literature. The use of high-performance computing techniques and the employ of supercomputers and large, distributed, concurrent systems for cryptography and cryptanalysis is not a recent development; by as early as the Second World War, allied forces were using computational resources to perform cryptanalysis of communication via the Reich's Enigma cryptosystem. The relationship between high-performance computing and computer security is therefore likely to remain important in the long term; the issue this paper seeks to address is what, if any, implications the use of GPUs to accelerate cryptography and/or cryptanalysis might have in the short- to mid-term. In order to address this question, we survey the current state-of-the-art in the acceleration of cryptography and cryptanalysis using graphics devices and draw a variety of conclusions.

The rest of the paper is organized as follows. In Section 2, we review background knowledge to provide enough context to understand the key points from the literature review and subsequent analysis and discussion. This section cannot, however, serve as the sole source of preparation for the subsequent material, and should be understood for what it is: a refresher or crash-course for the reader who is already knowledgeable in computing and security. Section 3 contains a detailed review of some of the most representative work and results in recent literature

in relevant areas. The scope of this review is necessarily limited to a few security-related topics, and the authors make no representation that what is presented here constitutes a complete review of the intersection of GPGPU and computer security. Indeed, it is the authors' hope that the limited scope lends itself to a more focused and, therefore, more useful study. To this end, three specific areas are considered: hashing, symmetric cryptography and asymmetric cryptography and closely-related issues. Section 4 contains these authors' interpretation of the findings from the literature, including commentary on the significance of current results and brief discussion of other security-related topics not treated which are nevertheless interesting from theoretical and practical points of view. Section 5 concludes the paper with a synopsis of GPGPU and cryptography, a brief summary of the current literature and a recapitulation of the related analysis. Acknowledgements and references follow.

2. BACKGROUND

2.1 Introduction

Before reviewing related work and presenting analysis and discussion thereof, we review some basic notions in the development and use of graphics accelerators for general-purpose tasks and in cryptographic algorithms for hashing and cryptography. The material in this section is summary in nature and cannot completely obviate the need for further study to fully comprehend subsequent discussion. Interested readers are encouraged to follow-up with the indicated references for more information. The rest of this section is organized as follows. In Subsection 2.2, we introduce GPGPU and CUDA, explaining in detail how and why graphics processors are used for general-purpose computation. In Subsection 2.3, we introduce specific security-related cryptographic and hashing algorithms of which we will make mention later on. The section concludes with a summary of key points to take away from the discussion in Subsection 2.5.

2.2 GPGPU and CUDA

2.2.1 Introduction

This subsection introduces the GPU as a general-purpose computational device. Sub-subsection 2.2.2 details some of the hardware and performance characteristics which make GPUs attractive as high-performance, computational devices, while at the same time pointing out some of the fundamental hardware-based performance limiters in GPGPU. Sub-subsection 2.2.3 gives a shallow treatment of NVIDIA's CUDA programming model and toolkit for GPU computing, so as to ease understanding of the programming challenges involved in writing code for the GPU. The subsection concludes with a summary of the covered material.

2.2.2 Hardware and Performance Characteristics

Over the last 20-30 years, graphics processors have evolved from fixed-function, special-purpose computational devices, into devices programmable through a restricted graphics shader API, and recently into fully programmable computational coprocessors. [13] Because of their humble origins as devices used for graphics rendering, graphics processors have developed with a different set of performance characteristics when compared to CPUs, corresponding to the differences in engineering requirements.

Compared to CPUs, GPUs are very high-throughput, high-latency devices. This is due to the fact that graphics cards were designed primarily for graphics, and humans are more sensitive to low throughput (e.g., forcing lower screen resolutions or smaller monitors) than they are to latency (e.g., viewing graphics many milliseconds after the GPU began processing the scene). For instance, modern NVIDIA Fermi GPUs can achieve over 1 TFLOPS of single-precision floating-point performance and over 100 GB/s sustained on-GPU memory bandwidth. Additionally, GPUs dedicate much more hardware real estate to arithmetic-logic and floating-point units than to control or cache units, meaning that they excel at numerically-intensive applications but not at ones which exhibit complicated control logic (more on this below). Again by way of comparison to CPUs, GPUs exhibit incredible amounts of thread- and memory-level parallelism. For instance, modern NVIDIA Fermi graphics cards contain in the neighborhood of 450 processor cores, each of which can simultaneously access a word of global memory over a wide memory bus. However, all of this hardware performance comes at the cost of programmability; achieving good performance on graphics processors takes effort and expertise. The reason for this is essentially twofold: first, processors on the GPU are grouped into multiprocessors which execute under a single-instruction, multiple-data hardware model, necessitating that programmers take into account their programs' branching behavior; second, a complex memory hierarchy and rules for efficient access – such as GPU-global DRAM memory being accessed in a coalesced, that is, contiguous and aligned fashion – complicate programming for performance. Another complication is that GPUs are typically managed by CPUs over an I/O (e.g., PCI-E v2) bus, requiring that attention be paid to CPU/GPU communication costs. These limitations aside, GPUs have been used to provide speed-up for real applications in high-performance scientific computing (e.g., gene sequence alignment, N-body simulations, climate modeling, etc.) Applications which see improvement on the GPU typically exhibit some or all of the following characteristics: large computational, numerical or arithmetical requirements; substantial thread- and memory-level parallelism; throughput is more important than latency (as is the case for most scientific jobs). GPUs have become viable high-performance, parallel numerical coprocessors, and the fact that they are commodity hardware – consider that modern NVIDIA Fermi products such as those used in previous examples sell for as little as under \$500 – means that they fit well with most computing organizations' desire for modular hardware (e.g., for augmenting existing cluster systems).

NVIDIA Fermi GPUs are organized hierarchically in terms of both processing and memory models. At the highest level, the GPU represents the entire computing device, and contains a device-global memory area hereafter referred to as global memory. Modern GPUs typically include several gigabytes of such memory, which provides high-throughput, high-latency GPU-wide read/write access. At the next level, groups of processor cores, hereafter referred to as streaming multiprocessors, of which NVIDIA Fermi GPUs typically have around 15, contain multiprocessor-local shared memory, which is treated as an explicitly software-managed cache. The size of shared memory varies, but is typically around 64 KB per multiprocessor; regarding performance, shared memory is typically much lower-latency than global memory. At the lowest level, processor cores, also referred to as streaming processors, of

which NVIDIA Fermi GPUs have 32 per multiprocessor, can access large numbers of processor-local registers, which represent the fastest available memory on GPUs.

2.2.3 Programming with NVIDIA's CUDA

NVIDIA's CUDA programming model and toolkit, released in late 2007, allow developers to write general-purpose programs to be executed, in part, on NVIDIA GPUs [11, 12]. CUDA represents a significant step forward in terms of programmability and has led to drastically increased adoption of GPGPU by programmers in several areas of application. It provides a minimal set of extensions to the standard C and C++ programming languages, in addition to several APIs, for efficiently mapping general-purpose computation to their graphics devices. More specifically, CUDA allows programmers explicit access to the hierarchies of processors and memories on GPUs through similarly hierarchical software abstractions. Within a traditional C/C++ program, application programmers define a special function, called a kernel, which will be executed by the GPU. Then, from within main (CPU) code, the application programmer may copy data from CPU memory to GPU global memory and back again, and in addition, invoke, or launch, the kernel, which triggers execution on the GPU over the system I/O bus. Memory transfers and kernel executions happen (or can be made to happen through appropriate flags to API functions) asynchronously to execution of host code, allowing for increased performance via overlapping of communication and computation.

When a kernel is launched on the GPU, many threads (the number and arrangement of which are specified as part of the kernel's invocation) are created and distributed to hardware scheduling units corresponding to the streaming multiprocessors. The set of all threads created in response to a kernel launch is referred to as a grid. Grids are further subdivided into thread blocks, such that all the threads belonging to a given thread block are scheduled on the same multiprocessor. Groups of threads within blocks referred to as warps are scheduled in an SIMD manner across the cores belonging to a multiprocessor. Using built-in variables, threads may access their position in the grid (which thread block they belong to, as well as their position within their thread block) and issue instructions accordingly. Note that for the threads of a warp to make efficient use of hardware, they must abide by the SIMD hardware limitation and execute the same instructions; however, they may operate on independent data elements.

2.2.4 Summary

In this subsection, we have discussed performance and programming characteristics of GPUs in terms of the underlying hardware and programming models. NVIDIA's CUDA provides an efficient and idiomatic interface to harness the power of GPUs for general-purpose computation. However, much work must still be done in order to bridge the performance-programmability gap. Other means of programming GPUs exist: wrappers for CUDA in other languages (Java, Python, etc.) as well as the competing, open-source language and framework by the Khronos group, OpenCL [8]. Despite GPUs' performance potential, they are limited in terms of the kinds of parallelism they can exploit and in terms of the kinds of applications for which they are well-suited. Fundamental limiters include the imposition of an SIMD programming model at the multiprocessor level and restrictions

on access patterns for efficient use of global and shared memories.

2.3 Cryptographic Primitives

2.3.1 Introduction

This section We have opted to present the material of this section in several subsections, each of which corresponds roughly to one of the three kinds of cryptographic algorithms we are including in this study. The reader should, however, bear in mind that the distinctions in some cases can become blurred and that these categorizations, while useful for the purposes of discussion, might not be representative of a thoughtful, academic taxonomy of cryptographic primitives. That being said, the rest of this section is organized as follows. In Sub-subsection 2.3.2, we consider solutions to the problem of cryptographically secure hashing. Sub-subsection 2.3.3 explores common algorithms for symmetric-key cryptography. Sub-subsection 2.3.4 details methods for providing asymmetric key cryptographic primitives. This subsection is concluded with a general summary of the problems these algorithms solve, how they are usually used in practice and other related information.

2.3.2 Cryptographically Secure Hashing

The problem of cryptographic hashing can be stated as follows: given as input a message (which we can assume to be a binary string of length n), produce a hash – or digest – of the message which satisfies the following four constraints: it must be feasible to compute the hash of a message (i.e., polynomial time); it must be infeasible to generate a message for a given hash (this is referred to as pre-image resistance); given one message, it must be infeasible to find another message to which the same hash value is assigned (this is referred to as second pre-image resistance); it must be infeasible to find any two messages to which the same hash value is assigned (referred to as collision resistance). Whether cryptographically secure hashing functions actually exist is unknown, as it depends on whether P – the set of computational problems which can be solved in time given by a polynomial involving the problem size, hence feasible to compute an answer given the question – is equivalent to NP – the set of computational problems which can be answered by a non-deterministic machine in polynomial time, hence feasible to verify an answer given appropriate evidence. It is worth pointing out that the constraints imply that the problem of reversing cryptographically secure hashing functions must be in $NP \setminus P$.

Cryptographic hash functions are often used in order to verify the integrity of data. For instance, if Alice publishes the hash of a secret message online and sends a message to Bob, Bob can verify that Eve hasn't replaced Alice's message by checking the result of hashing the received message against the publically available hash. Since it should be difficult for Eve to tamper the message in such a way as to preserve the original hash, or to find other messages with the same hash, Alice and Bob can rest assured that their message will be safe. Important cryptographic hashing functions – SHA and MD – are discussed. We also consider homomorphic hashing functions, which are commonly used in network-encoded peer-to-peer networks.

2.3.2.1 Message Digest and Secure Hash Algorithm

The MD4 family of hash functions, which includes MD4, MD5 and SHA1, began with Rivest's 1990 implementation of MD4 [19]. These algorithms are computationally intensive and require infrequent memory access, which makes them perform well on a variety of computer architectures. The basic functioning of this family of hash functions is described in the next paragraph for SHA1, and the MD4 & MD5 algorithms function in similar ways. Notable differences include the following: MD4 and MD5 use 4-word buffers, whereas SHA1 uses 5-word buffers (hence $32 \times 5 = 160$ bit digests); MD4 uses three rounds of computation, whereas MD5 and SHA1 use four.

SHA1 was designed to compute 160-bit digests for messages of length up to 2^{64} bits [20]. The algorithm consists of two stages: preprocessing and computation of the actual hash function. The preprocessing stage involves padding the input message so as to guarantee that the total message length is an even multiple of 512 bits, and parsing the message into N 512-bit chunks. Computation of the hash function is an iterative process performed on each of the blocks, with intermediate hash value propagation and using the final propagated value (after processing all N blocks). Specifically, the processing of each block involves four stages: a left rotation of the block's bits, the computation of several predefined bitwise functions using the block as input, repeating these steps 80 times and finally generating the intermediate hash value using module-32 addition. The message digest is then computed as the concatenation of the five resulting 32-bit values.

SHA-1 was replaced by SHA-2, in order to have hashing capability matching AES's symmetric-key cryptographic capabilities. The government is currently in the process of replacing SHA-2 with a new algorithm, and results will be announced in 2012 [3]. Of the fourteen second-round SHA-3 candidates, several use AES-inspired byte-oriented operations rather than the traditional SHA-1-style block manipulations. Others employ various techniques.

2.3.2.2 Homomorphic Hashing Functions

Homomorphic hash functions are used in network-coded peer-to-peer networks to defend against pollution attacks [18]. Pollution attacks consist of malicious users injecting corrupted or otherwise bad packets into the network in order to frustrate file sharing. In non-coded networks, regular hashing functions – e.g., SHA1 – can be used to provide a means of verifying the integrity of message data. However, in coded networks, packets may be combined on-the-fly in ways unpredictable to the message source, making accurate a priori hashing infeasible. Homomorphic hash functions overcome this difficulty by possessing a unique property: the homomorphic hash of a linear combination of a set of messages can be obtained from the hashes of the individual messages. Despite their appropriateness to the task of providing a way to ensure integrity of messages exchanged in a network-encoded P2P network, this class of hash functions suffers from poor performance characteristics: it is computationally expensive, so that the throughput of representative methods on a 3GHz Pentium 4 processor is around 300 Kbps.

2.3.3 Symmetric Key Cryptography

The problem of symmetric key cryptography can be stated as follows: Given a message (which we can assume to be a binary

string of length n) and a key (which we can assume to be a binary string of length m), provide a feasible means of encrypting the message such that decrypting is feasible if and only if the key is known. Since encrypting and decrypting occur using the same key, this must be shared by the sender and receiver of messages; it is assumed that keys are exchanged securely, while encrypted messages may be intercepted in flight. A certain symmetric key cipher which uses random keys of the same length as the message to be encoded, referred to as the one time pad, is provably unbreakable (under the idealized, though tenable, assumption that the adversary has no way side-channel attack to recover the key). It consists in performing a bitwise XOR operation between the key and the message in order to encrypt, and again in order to decrypt. Given a plaintext message of length n , encryption can result in any of the 2^n binary strings of length n (which string is obtained depends only on the message and the encryption key). Decryption can similarly result in any of the 2^n binary strings of length n (which string depends only on the ciphertext and decryption key). Therefore, not even a brute-force attack based on guessing keys can work, since there would be no way to determine which of the meaningful messages of length n was the correct one. However, due to the fact that the one-time pad requires truly random keys of the same length of the message, its usefulness is rather limited in practical applications for logistical reasons (e.g., it is difficult to generate and distribute keys, all the while defending against side-channel and social-engineering attacks). More practical schemes involve fixed-length keys for variable-length messages; of these, we will primarily consider AES, as well as a few other representative techniques. Applications of symmetric key cryptography usually involve efficient encryption of large amounts of block data, guaranteeing confidentiality of data being communicated over insecure channels.

2.3.3.1 Advanced Encryption Standard – AES

AES is a symmetric block cipher introduced by NIST in 2001 [6]. Rijndael, the selected algorithm, beat out competitors such as Serpent and Twofish in an international competition to replace the aging DES with a new, more secure solution for symmetric key cryptography [16]. These algorithms share a similar, substitution-permutation based structure. In particular, Rijndael uses 9, 11 or 13 rounds (depending on the key size, which can range from 128 to 256 bits) with the round key computed by key expansion and using fixed S -boxes. Each round consists of four stages: SubBytes, ShiftRows, MixColumns, AddRoundKey. Serpent uses 31 rounds per block with fixed S -boxes, and does not require any preprocessing. A round of the Serpent cipher includes key mixing, application of S -boxes and the linear transformation step. Twofish uses a 16 round Feistel-like structure with round keys and S -boxes computed from the input key and with input and output whitening. All of these methods are block-based ciphers, meaning that they operate on individual blocks and, for messages larger than a block, work on the message in a segmented fashion.

2.3.3.2 Miscellaneous

SEAL belongs to the pseudorandom function family of cryptosystems, and works differently from AES [14]. Using the key, SEAL generates a pseudorandom bit string which it XORs with blocks of plaintext in order to produce ciphertext. The algorithm works in two rounds: table generation and pseudorandom bit stream computation. Tables are generated and

don't change for the duration of the communication. To generate the pseudorandom bit stream, the compression function of SHA1 is used to expand a position index into a bit string of the desired length (this process depends also on the tables generated in the first step). This procedure exhibits near ideal thread-level parallelism, and should therefore be bounded only by limited bandwidth of the memory subsystem.

ARIA, developed in 2003, was selected in 2004 as the standard encryption algorithm of the nation of Korea [17]. It is based on a multi-round substitution/permutation network where each round can be divided into three phases: a round key step, an S-box step, and a diffusion step. As is the case with AES, the number of rounds performed per-block depends on the key size; for keys of length 128, 192 and 256 bits, 12, 14 and 16 rounds should be used, respectively.

2.3.4 Asymmetric Key Cryptography

The problem of asymmetric key cryptography can be stated as follows: given a message (which we can assume to be a binary string of length n) and (minimally) private keys for both the sender and the receiver, provide a means of encrypting and decrypting the message which satisfies the following constraints: it must be feasible for the sender to encrypt the message with his private key; it must be feasible to decrypt the encrypted message if and only if the receiver's private key is known. Note that asymmetric key cryptography is a simple extension of symmetric key cryptography in which we don't require that the sender and receiver share a common key (indeed, the definition given here is somewhat relaxed, and allows any symmetric key algorithm to be interpreted as an asymmetric key algorithm in which the sender's and receiver's private keys happen to be the same; most often, though, asymmetric key cryptography is concerned only with the more general case in which this assumption is not made). This makes asymmetric key cryptography very attractive when compared to symmetric key cryptography, since the latter requires a secure method to exchange the shared secret – a requirement which may necessitate a complicated (and possibly expensive) alternative solution (e.g., delivery by armed guards). However, the price for this is paid in two ways: first, unlike for symmetric-key algorithms, there is no asymmetric key cryptographic procedure which provides provably unbreakable security (as was the case for the one time pad); second, the means of encryption and decryption tend to be more costly even with the private keys. Common applications of asymmetric key cryptography are to provide a means of giving digital signatures, thereby providing a degree of accountability (e.g., by proving that a party send a particular message) and to allow for secure key exchange, enabling the use of symmetric key cryptography for the bulk of the message (thereby improving performance). We will concern ourselves primarily with two asymmetric key cryptographic techniques – elliptic curve cryptography and the RSA algorithm – as well as a lattice-based method, NTRUEncrypt.

2.3.4.1 Elliptic Curve Cryptography – ECC

Developed by Miller and Koblitz in the mid 1980s and standardized by NIST in 2000, the elliptic curve family of cryptosystems provides security comparable to that provided by RSA using shorter keys [Cohen]. An elliptic curve $E(a, b, p)$ over a Galois field $GF(p)$ for prime p is a set composed of a point at infinity and all points (x, y) in $GF(p)^2$ where $y^2 = x^3 + ax + b$,

where a and b are in $GF(p)$ [1]. To ensure smoothness properties of the resulting curve, a and b are constrained such that $4a^3 + 27b^2$ is not an even multiple of p . By defining point addition and doubling for the field of curves, it is possible to define point multiplication via repeated addition. The security of elliptic curve cryptography relies on the fact that it is believed to be infeasible.

2.3.4.2 RSA Algorithm

RSA, widely used in E-commerce applications and the first algorithm known to be suitable for signing as well as encryption, is another - more traditional - public key cryptosystem [4]. Each RSA participant possesses a public key and a private key. The public key is made available to the world and is used to encrypt data sent to the public key's recipient. The private key must be kept secret by the recipient and is required to decrypt messages encrypted using the public key. Key generation occurs by selecting uniformly at random two large prime numbers p and q of comparable bit length, and computing the modulus $n = pq$. Next, the value x is computed using Euler's totient function on p and q , which is $(p - 1)(q - 1)$ since p and q are relatively prime. Next, the public key e is chosen greater than 1 and less than n , so that it is also less than x . Choosing e with shorter bit length results in more efficient encryption, but may result in reduced security. The private key d corresponding to public key e is chosen such that $ed - 1$ is evenly divisible by x ; properties of arithmetic over Z_n guarantee that the value of d will be unique, and d can be found using the extended Euclidean algorithm. Encryption and decryption are then carried out via exponentiation by the keys: $C = M^e$ and $M = C^d$.

2.3.4.3 NTRUEncrypt and Lattice Based Methods

NTRUEncrypt is a representative member of the family of parameterized lattice-based asymmetric key cryptographic algorithms [7]. This algorithm involves simple polynomial multiplication and is therefore significantly faster than RSA and ECC methods. NTRUEncrypt is parameterized using three integers, (N, p, q) , where N is prime, $\gcd(p, q) = 1$ and $p \ll q$. Using these, define polynomial rings – one generic, one based on each of p and q – as well as a convolution multiplication and special class of polynomials for which the numbers of coefficients equal to each of 1, 0 and -1 are known. Key selection occurs as follows. First, choose a ternary polynomial f of the special form just defined which is invertible in the p - and q -based rings. Next, choose a private ternary polynomial g . Compute inverse functions F_q and F_p of f in the rings based on p and q . Letting $h(x) = F_q(x)g(x) \bmod q$ (where juxtaposition indicates convolution multiplication), $h(x)$ is used as the public key and the private key is taken to be the pair $(f(x), F_p(x))$. To encrypt, compute $e(x) = pr(x)h(x) + m(x)$ for random ephemeral key $r(x)$. To decrypt, first compute $a(x) = f(x)e(x) \bmod q$, centerlift $a(x)$ into the generic ring, and compute $m(x) = F_p(x)a(x) \bmod p$. The two convolution multiplications represent the most time-consuming part of both encryption and decryption processes.

2.3.5 Summary

In this subsection, we have introduced three classes of cryptographic problems – cryptographic hashing, symmetric key cryptography and asymmetric key cryptography – and a few representative solutions to each. Additionally, we have indicated common applications of these techniques to real problems in computer security, and have discussed issues of correctness and

efficiency. Moreover, we have endeavored to point out where certain techniques and usages might and might not be amenable to parallelization on a massive scale, and at what granularities we might wish to decompose these cryptographic primitives.

2.4 Summary

In this section, we have sought to provide background and context sufficient to provide the already technically knowledgeable reader with enough information to readily absorb the information presented in the next sections. As has already been mentioned, it is not the authors' intent – and is well beyond the scope of this paper – to provide in-depth coverage of any of the material here outlined (any of these topics could be the subject of several textbooks). Interested readers are encouraged to refer to materials cited in this section for more complete coverage of background material.

We have introduced GPUs as high-performance, massively parallel computational coprocessors. This parallelism is achieved via a highly scalable, hierarchical architecture which tolerates high latencies to achieve high throughput in computational tasks. Also sacrificed is a degree of programmability-performance, which is lost due to the SIMD restriction at the streaming multiprocessor level. By dedicating more die space to arithmetic and floating-point computation compared to control logic and cache, GPUs allow for better efficiency of arithmetically-intensive algorithms, and at the same time put the burden of achieving good performance squarely on the shoulders of the application programmer. The result is a device which can provide orders-of-magnitude speed-up for certain, highly data-parallel applications common throughout the computational sciences, if care is taken to avoid GPUs' performance pitfalls.

Also introduced were three classes of cryptographic problems – hashing, symmetric cryptography and asymmetric cryptography – and algorithms solving each problem. Issues of correctness, performance and amenability to massive parallelization were addressed on a case-by-case basis. In the next section, we will see how researchers have been attempting to use GPUs in order to improve performance of these algorithms for a variety of usage scenarios.

3. RELATED WORK

3.1 Introduction

Having already introduced relevant background information in Section 2 above, we proceed to give an account of some of the current research efforts directed toward the problem at hand. The arrangement of topics intentionally mirrors that of Subsection 2.3; whereas in that section we introduced the primitives themselves, we will now focus on work which has been done to use graphics accelerators in order to improve the performance of these algorithms. We conclude with a summary of some the performance, programmability and other observations and results from the research.

3.2 Cryptographically Secure Hashing

3.2.1 Introduction

In this subsection, we review work which seeks to accelerate cryptographically secure hashing algorithms to achieve high data processing throughput. We again pay special attention to the

popular MD/SHA family of hash algorithms, and include acceleration of more exotic hash functions for additional perspective. The subsection concludes with a review of the salient performance results obtained by the authors.

3.2.2 Message Digest and Secure Hash Algorithm

Zhou et al. [19] investigated performance of the MD4 family of cryptographic hash functions when implemented on GPUs. Because of the algorithmic and performance characteristics of this family of algorithms – they are computationally intensive and require infrequent memory accesses – these algorithms are particularly well-suited to implementation on a GPU. Despite this, the authors point out a difficulty in doing GPU optimization – striking the right performance balance between the number of thread blocks and the number of threads per block – in order to achieve good performance. Testing GPU implements in CUDA on three different NVIDIA devices (9600 and 9800 GT and GTX), the authors find results to be quite promising (speedups of 100x, 170x and 200x for SHA-1, MD5 and MD4, respectively, as well as good scalability with increasing core count and improved completion time on newer hardware).

Zhou et al. [20] also implemented SHA-1 on a GeForce 9800 GX, the Cell broadband engine, and FPGAs. Despite the fact that the SHA-1 algorithm is no longer used for secure applications and that the GeForce 9800 GX is an outdated graphics card, their implementation details and performance results are nonetheless instructive. Using a minimal set of bitwise logical instructions (AND, OR, NOT, XOR), unrolled loops and efficient on-multiprocessor constant memory caching, they were able to obtain 20 GB/s throughput. Compared to results from FPGAs and Cell processors, they noted the following: GPUs provided the best price/performance ratio of any solution, although they used significant amounts of power. However, the power/performance ratio was competitive with the other two technologies.

Bos et al. [3] compared performance of the fourteen second-round SHA-3 candidates on two exotic architectures, Cell processors and NVIDIA GPUs. Specifically, they only implemented the non-AES-like candidate algorithms on a GTX 295, which represents much newer graphics technology than was available to Zhou et al. Different candidates varied greatly in terms of the achieved speed-up under GPU acceleration: throughputs of between ~1 GB/s and ~37 GB/s were observed for the worst and best candidates, respectively. The authors found that performance was limited by hardware characteristics, especially the limitation on per-multiprocessor shared memory cache and performance degradation due to warp divergence.

3.2.3 Homomorphic Hashing Functions

Zhao et al. [18] implemented a multiple-precision arithmetic API for NVIDIA GPUs using CUDA including Montgomery reduction, Montgomery multiplication and Montgomery exponentiation, and used this to implement a portable homomorphic hashing method. They measured the performance of this method and compared it to that of a CPU-based method. The results were good, though not so impressive: on a 1.6 GHz quad-core Intel CPU, the authors were able to achieve 130 Kbps using one core, therefore comparable to the 300 Kbps result on a 3GHz Intel Pentium 4; on NVIDIA GTX 260 and GTX 280 GPUs, the authors achieved throughputs of 1.35 Mbps and 1.9

Mbps, respectively. However, the authors were able to exploit a feature of homomorphic hashes to optimize the GPU implementation by adding a pre-computation phase to the Montgomery exponentiation, thereby increasing performance: they observed throughputs of 8.98 Mbps and 11.6 Mbps on the GTX 260 and GTX 280, respectively. Aside from representing, maximally, a 38x speedup, they noted good scaling characteristics from the GTX 260 to GTX 280, which increased almost linearly compared to the increasing in streaming processor (core) count.

3.2.4 Summary

For the MD4 family of hash functions, impressive speed-ups of between 100x and 200x were observed in one case, and in another, some SHA3 candidates exhibited excellent throughput on the GPU of up to 37 GB/s. Homomorphic hashing was accelerated up to 38x, making it much more viable as a way to do hashing in environments where regular hashing is not possible. Programmability of GPUs using CUDA was, however, noted in several instances as being a performance limiter; indeed, this is one of the primary limitations of CUDA and GPUs in a broader sense. Despite these limitations, good results have been observed for uses of cryptographic hashing requiring high data throughput.

3.3 Symmetric Key Cryptography

3.3.1 Introduction

In this subsection, we review work which seeks to accelerate symmetric key cryptographic algorithms to achieve high data processing throughput. As before, AES is treated separately, given its prominent place in the symmetric key cryptographic landscape. Other more exotic techniques are then briefly mentioned to give the reader some feeling for the dizzying number of such algorithms. We conclude with a review of important performance and programmability observations that, along with material from the rest of Section 3, will form the basis for the reasoned analysis in Section 4.

3.3.2 Advanced Encryption Standard

Manavski [9] was the first to implement AES for GPUs using CUDA, although other researchers (e.g. Cook et al.) had worked on implementing AES using programmable shaders and/or the graphics pipeline. Using a minimalistic implementation which stored S-boxes in constant memory and blocks/keys in global memory, and working with an NVIDIA GeForce 8800 GT graphics card, he observed 6.65 Gbps throughput for AES-256, a then 19.6x speedup compared to the optimized OpenSSL CPU-only version. Manavski commented that the single greatest limiter of performance for symmetric key encryption on graphics processors might be the system I/O bus bottleneck.

Building on the work of Manavski and others, Iwai et al. [6] performed a systematic search of the thread- and memory-level optimization spaces for accelerating AES on GPUs. Thread granularities of 16, 8, 4 and 1 bytes/thread were tested, as well as the relative performance benefits of storing S-boxes in constant versus shared memory. Additionally, the opportunity to pipeline the process over the PCI-E based I/O bus was evaluated. Experiments were performed using an NVIDIA GeForce GTX 285, a newer generation hardware device than was available to Manavski at the time of his study. When considering the I/O bus

transfer time, the maximum achieved throughput was only 13.4 Gbps when not overlapping communication and computation; this was raised to 22 Gbps by including that optimization to address the I/O bottleneck, equivalent to a 28.39x speed-up when compared to an Intel i7 920 2.66 GHz processor.

Wang et al. [16] compared the performance on four different hardware architectures – an NVIDIA GeForce GTX 285, an ATI HD5970, an Intel i7 Quad Core and an AMD Phenom 4X – of the three AES finalists Rijndael, Serpent and Twofish, using both NVIDIA’s CUDA as well as Khronos Group’s OpenCL framework. Using 256-bit keys and compared to a sequential version of the algorithms, the authors observed between 12x and 23x speed-up for OpenCL (about 10% to 20% worse than CUDA) for all algorithms. Throughput was around 12 MB/s.

3.3.3 Miscellaneous

Theoharoulis et al. [14] implemented the SEAL encryption algorithm using several hardware devices and implementation techniques: an Intel i7 870 processor with and without using the SIMD capabilities of the chipset, FPGAs, and an NVIDIA GeForce GTX 480 GPU. The authors were aware of the potential PCI-E bottleneck and attempted to address this potential performance limiter through CUDA’s stream API. Although they observed over 8x speedup over the sequential CPU version of the code, performance suffered greatly due to the I/O bottleneck, so much so that when considering the PCI-E transfer time, parallel implementations using multiple cores on Intel’s i7 outperformed the GPU.

Yeom et al. [17] implemented the ARIA cryptosystem on an NVIDIA GeForce 8800 GTS, varying the amount of encryption work assigned to each thread. They found that ARIA outperformed a comparable, optimized version of AES running on similar graphics hardware, achieving up to 4.8 Gbps. This is an interesting result, given that ARIA is known to be slower than AES in sequential settings. The authors attribute this difference to ARIA’s intrinsic algorithmic qualities which better lend themselves to GPU-based parallelization.

3.3.4 Summary

In this subsection, we have looked at some recent research efforts aimed at providing speed-ups for symmetric key cryptographic algorithms including AES candidates, ARIA and SEAL. In almost all cases, authors reported significant success in terms of providing real speed-up to real applications, as high as nearly 30 times. However, a common thread running through these papers’ analyses is that the I/O bottleneck (typically over PCI-E) is a significant limiter of performance in this context and, in at least the case of ARIA, leads to a rejection of GPUs as head.

3.4 Asymmetric Key Cryptography

3.4.1 Introduction

In this subsection, we review research targeting GPU acceleration of asymmetric key cryptographic algorithms to achieve high data processing throughput. As in Section 2 above, we take a closer look at cryptography via elliptic curves and the RSA algorithm, in addition to visiting another exotic scheme treated in the literature. The subsection concludes with a summary which highlights key

points, similarities and differences between research groups' efforts.

3.4.2 Elliptic Curve Cryptography – ECC

Antao et al. [1] observed that elliptic curve techniques do not possess all the intrinsic algorithmic properties which lend themselves to good performance on GPUs. In particular, modular inversions are costly. They tried to overcome this limitation using the residue number system approach. By representing inputs in RNS form and using the Montgomery ladder algorithm, the authors were able to achieve good performance on an NVIDIA GeForce GTX 285 GPU: 24.3 ms latency and 9,990 point multiplications per second. This represents an order-of-magnitude improvement in latency and a 122% improvement in throughput compared to previous attempts to put ECC onto GPUs.

Berstein et al. [2] used schoolbook multiplication on 280-bit input elements via floating-point operations on GPUs, rather than Karatsuba multiplication, in an attempt to increase performance. Using Montgomery representation modulo m , they evaluated performance on several systems: Intel Core 2 Duo E6850, Quad Q6600 and Q9550, as well as NVIDIA GeForce GTX 260, 280 and 295. Also tested was a purpose-built system consisting of a Q6600 and two GTX 295 units. The best performance and price-performance results were achieved on this system: about 97 million Mulmod (modular multiplications) per second, translating to 926 curves - over 6x better than the best CPU-only system. Performance per cost was only about 0.42 Hz/\$, about 2.5x better than a CPU-only system..

3.4.3 RSA Algorithm

Moss et al. [10] implemented on the GPU modular arithmetic functions required by RSA. Using residue number system representations and an NVIDIA GeForce 7800 GTX GPU, they were able to obtain about 3x speed-up compared to comparable CPU versions running on a 2.2 GHz AMD-64 3200+. The GPU version of this code was done using programmable shaders, not CUDA, so implementation was more complex than – and performance may have suffered more than – would be the case with a modern CUDA implementation.

Harrison et al. [5] commented that asymmetric key cryptography is inherently better-suited to the GPU than symmetric key cryptography due to its higher computational requirements. In their study, residue number systems were also used, and they achieved similar (4x) improvement over comparable CPU code.

Fan et al. [4] implemented RSA with multi-level parallelism using Hadoop and jCUDA. For the task of decrypting 40,000 character messages, a CPU-only implementation took a little over a minute to complete, whereas a GPU completed the same task in a little over two seconds. Furthermore, the authors noted that the GPU version demonstrated good scalability and that the I/O bottleneck was not nearly as large an issue in this case, reaffirming the conclusion of Harrison et al..

3.4.4 NTRUEncrypt and Lattice-Based Methods

Kamal et al. [7] implemented the NTRUEncrypt algorithm on an NVIDIA GeForce GTX 275 GPU. In order to achieve the most benefit from parallelization, they encrypted many messages in parallel, achieving up to 77 MB/sec – or 616 Mbps – throughput

for the decryption task. The authors noticed that NTRUEncrypt suffered somewhat from the limited global memory bandwidth, and using shared memory increased performance.

3.4.5 Summary

In this subsection, we have reviewed attempts to accelerate via GPGPU several representative asymmetric key cryptographic algorithms to increase the data processing throughput. Results are good in all cases, with several of the authors observing that asymmetric cryptographic primitives are by their very nature more suited to off-chip acceleration since they involve more computation (compared to symmetric key techniques). Several efforts in ECC and RSA cryptography which make use of GPUs use the residue number system to efficiently encode input data to promote parallel processing on GPU architectures.

3.5 Summary

In this section, we have reviewed a sampling of the contemporary literature on the use of GPGPU for accelerating cryptographic primitives. Specific examples of obtained performance serve to illustrate both the successes and shortcomings of the current state-of-the-art, and serve to point the way to future advances which might – or might not – solidify the status of GPGPU as a viable technique for high-performance cryptography.

In subsection 3.2, we reviewed several cryptographic hashing functions: the MD4 family (including the original MD4, MD5 and SHA1 algorithms), SHA3 candidates, and a special class of hash functions well adapted for a particular use referred to as homomorphic hashing functions. Despite good performance, programmability remains a problem for practitioners in their adoption of GPUs for general-purpose semi-numerical work.

In subsection 3.3, we reviewed attempts to port to GPUs and other exotic architectures several common symmetric key cryptographic algorithms, including AES finalists, SEAL, and ARIA. Although good performance improvements in terms of throughput and speedup were observed in many cases, the I/O bandwidth bottleneck was observed to be a common performance limiter for many of these applications, as it indeed is in many other application areas.

In subsection 3.4, we summarized efforts to parallelize asymmetric key cryptographic algorithms of three kinds – elliptic-curve-based methods, the RSA algorithm and a member of the family of parameterized lattice-based algorithms – and observed that these algorithms may be better-suited to GPU acceleration by virtue of their higher computational load. We also observed that several authors employ a common representation strategy, residue number system, in order to streamline GPU processing of input data sets. For computationally demanding encryption/decryption, the I/O bottleneck is less troubling than for other applications.

4. ANALYSIS & DISCUSSION

4.1 Introduction

Having reviewed prerequisite knowledge in Section 2 and a representative sample of the state-of-the-art in the last section, we present a critical analysis of the following: the potential of GPGPU for cryptography and cryptanalysis in terms of performance and feasibility; whether or not GPGPU is ready for

the mainstream computer security community in terms of programmability and reception/adoption of this novel architectural unit; other ways, if any, GPGPU might become relevant in the realm of computer security, at least in the short-term. These three topics are treated separately, and corresponding treatment can be found in Subsections 4.2 – 4.4. We conclude with a summary of our analyses and key points.

4.2 Performance and Feasibility

Several performance limitations were witnessed in Section 3 which have hindered past attempts and, if left unaddressed, may restrict the significance of results in the use of GPUs for accelerating cryptographic primitives. A few of these performance limitations, and these authors' observations of trends in the CUDA development model for NVIDIA GPUs [11, 12] relevant to overcoming these limitations (where such trends exist!), are the following:

- **I/O Bottleneck.** As long as the GPU remains a device accessed via the system I/O bus, the CPU-GPU communication involved in computations will be a potential performance issue. We, however, do not see this as being too dangerous from a long-term performance point of view. First, several efforts are aimed at easing the difficulty of arranging efficient host/device transfers, and recently introduced features in CUDA – as well as independent research efforts – seek to address this in a general fashion. Additionally, the common technique of increasing the per-thread workload should be effective for increasing the efficiency of GPU kernels. For instance, if a GPU thread is capable of processing (on average) 10 Kbps of encryption data, and the I/O latency is 20 ms, it might take 0.12 s to process 480 Kb of data (4 Mbps throughput); on the other hand, if each thread processes twice as much data and the same number of threads is used, the total time can be equal to 0.22 s, a throughput of over 4.3 Mbps. Also, as new I/O technologies are adopted (consider the advance from PCI to PCI-E with PCI-E v2, with PCI-E v3 on the horizon), the number and variety of cryptographic primitives which can efficiently be encrypted should increase.
- **Limited Device Memory.** Limited device global and shared memory was cited in some cases as being a performance hurdle which was difficult to overcome. Unsurprisingly, this is a common – and valid – performance-limiting issue on GPUs. We do not find this all that troubling for the following reason: while the amount of on-device memory in GPUs continues to increase at an impressive rate, cryptographic standards are significantly slower to change and, when they do change, the amount of the change is often a logarithmic function of the added security (so that a multiplicative increase in security would be accompanied by only an additive increase in data load). In many respects, this is a preferable situation to many other GPGPU application areas in which the data requirements of the algorithms scale in a linear or super-linear fashion with the problem complexity. On-device memory should be expected to exceed demands by cryptographic

primitives as technology is improved much faster to accommodate other applications.

4.3 Programmability and Adoption

The interplay between programmability and performance is a subtle one and affects adoption of novel technologies by new users; in any event, this has certainly been the case with GPUs, and rightly so. Researchers and designers have made dramatic strides towards increasing the generality and programmability of CUDA and GPUs; recent releases of CUDA have brought with them the inclusion in NVIDIA's runtime of function pointers, recursion, class and template inheritance and virtual functions [11, 12]. Some configurable automatic caching is available for global memory, obviating the need to explicitly control transfers to and from shared memory in many cases. Improved compiler-based optimization techniques eliminate some instances of warp divergence, uncoalesced memory access and the like. These improvements allow the application programmer to focus on correctness rather than on performance, without sacrificing overly much of the latter.

That being said, programmability for performance is the single greatest challenge to be overcome for making GPUs suitable for more wide-scale use in the cryptographic community. Although performance of the underlying hardware shouldn't be expected to remain relevant long into the future – since, for the most part, GPUs are still experiencing exponential growth in almost all performance-related areas – effectively making those capabilities available to end users through idiomatic and efficient language and API routines remains a challenge.

4.4 GPGPU in Computer Security

In this paper, we have considered the rather narrow technical question of whether GPUs can be used to accelerate popular cryptographic primitives. A few of the other ways in which GPUs might have a significant impact on the field of computer security are the following:

- **Cryptanalysis.** By accelerating security primitives, GPUs may serve to decrease the amount of time needed to perform brute-force-based attacks on primitives – for instance, finding hash collisions via birthday attacks or searching AES keyspaces. Better programmability and ever increasing performance of GPUs may portend a quickening of these brute-force cryptanalytic attacks, necessitating more frequent and/or more complex updates to cryptographic standards.
- **GPU-Assisted Malware.** Vasiliadis et al. [15] describe ways in which GPUs can be used to facilitate the concealment of malware programs. They describe and implement two techniques – unpacking and run-time polymorphism – using GPUs, and argue that existing virus scanners are ill-equipped to detecting and dealing with such kinds of behavior. Moreover, they point the way towards future exploits, including manipulation of the framebuffer (for deceiving users by adding misleading text or graphics to the display, or by reading information from the display in a much less obtrusive fashion than taking a screen capture). In the future, they argue, the trend towards GPUs which are less dependent

on CPUs may mean that malware writers begin targeting these devices themselves; the dominance of two companies' (NVIDIA and AMD) products makes this all the more likely. We believe that this work and the predictions it makes should be taken seriously.

4.5 Summary

In this section, we have looked at a few of the most common and troubling issues facing security practitioners when it comes to using GPUs as parallel accelerators for cryptographic primitives. Of the three main challenges – hardware issues of the I/O bottleneck and lack of on-device memory capacity and programmability issues – we find the last of these to be the most significant. In a larger sense, the transition from the era of sequential computation to that of parallel computation is still happening and far from over; helping programmers manage the complexity of large-scale parallel execution and finding ways of exploiting concurrency are hard problems with no clear answer. The high-performance computing community has been struggling to push the limits of performance and have so far maintained fairly steady progress; in many ways, that researchers in other fields are using architectures as exotic as the GPU says much not only concerning the feasibility of GPGPU but also about increasing adoption of parallel computing by a broader technical audience.

5. CONCLUSION

5.1 Introduction

In this section, we review some of the key goals and contributions of this work in the hopes of solidifying in the minds of the readers what points we believe should be taken away from this discussion. We begin by summarizing key points from the sections on background and the review of the current literature. We then reiterate some of these authors' findings and interpretations in Subsection 5.4. We conclude with some closing remarks in Subsection 5.5.

5.2 GPGPU and Cryptography

Over the last few decades, GPUs have evolved from fixed-function devices to fully-programmable general-purpose arithmetic coprocessors. They possess the capacity to perform computationally-intensive processing of large amounts of data in a very efficient fashion, but only for problems which can be efficiently mapped onto the data-parallel GPU processing model. NVIDIA's CUDA has made use of these devices for general-purpose computing, including cryptography and other semi-numerical computing, considerably easier than under the programmable shader model of GPU computing.

Three classes of cryptographic primitives – cryptographically secure hash algorithms, symmetric key cryptographic algorithms and asymmetric key cryptographic algorithms – exhibit different degrees of performance improvement under GPU-based parallelization efforts. Efficient use of GPUs for these problems requires identifying massive parallelism, often at many levels. This may involve exploiting parallelism at fine granularities – within the process of encrypting or decrypting a single message – or at coarse granularities – by performing many encryptions and decryptions in a batch-based fashion.

5.3 The State of the Art

We have observed several instances in which GPU parallelization of cryptographic primitives resulted in good – in some cases, extraordinarily so – speed-ups for real usage scenarios. We observed some instances in which GPU parallelization seemed poorly suited. The performance and programmability limitations of GPUs have indeed manifested in these fields, as they have in other, more varied areas of application. Indeed, the radical performance improvements GPUs bring to certain classes of algorithms may make previously unattractive algorithms more tenable (e.g., ARIA).

5.4 Significance and Projections

Performance limitations of GPUs, while real, do not – in these authors opinion – represent the most pressing concern for near- to mid-term adoption of GPGPU for computer security and cryptography. Rather, the programmability-performance (that is, the lack of performance and the result this has on obtained performance) represents a much more significant hurdle for which the computing community has yet to determine a good solution. GPUs may also be expected to play an increasing role in other areas of computer security as their autonomy and capabilities are improved.

5.5 Summary

In this paper, we have introduced GPGPU and several representative classes of cryptographic primitives, and explored the ways in which these areas of research have begun to overlap over the last few years. While GPUs offer enormous potential benefits to the acceleration of many algorithms of interest to practitioners and researchers in cryptography, wider adoption depends on the GPU becoming a more generally programmable device. The road ahead is not an easy one; only time will tell what the fate of GPU-assisted cryptography will be.

6. ACKNOWLEDGMENTS

The authors would like to thank the following individuals: Dr. John A. Hamilton, for providing the impetus to carry out this research and for aiding in the authors' preparation regarding computer security and cryptographic algorithms, and Dr. Weikuan Yu, for cultivating the authors' knowledge of and passion for high-performance computing and computer architecture.

7. REFERENCES

- [1] Antao, S., Bajard, J.C., Sousa, L. Elliptic Curve point multiplication on GPUs. In proceedings of 21st IEEE International Conference on Application-specific Systems Architectures and Processors (ASAP), 2010. pp 192-199.
- [2] Bernstein, D.J., Tien-Ren Chen, Chen-Mou Cheng, Lange, T., Bo-Yin Yang. ECM on Graphics Cards. In Proceedings of the 28th Annual International Conference on Advances in Cryptology: the Theory and Application of Cryptographic Techniques (EUROCRYPT), 2009.
- [3] Bos, J.W., Stefan, D. Performance analysis of the SHA-3 candidates on exotic multi-core architectures. In Proceedings of the 12th International Conference on Cryptographic Hardware and Embedded Systems (CHES), 2010.

- [4] Wenjun Fan, Xudong Chen, Xuefeng Li. Parallelization of RSA Algorithm Based on Computer Unified Device Architecture. In Proceedings of the 9th International Conference On Grid and Cooperative Computing (GCC), 2010.
- [5] Harrison, O., Waldron, J. Public key cryptography on modern graphics hardware. In Booklet of posters, Eurocrypt 2009, April 2009.
- [6] Iwai, K., Kurokawa, T., Nisikawa, N. In Proceedings of the First International Conference on Networking and Computing (ICNC), 2010. pp 209-214.
- [7] Kamal, A.A., Youssef, A.M. Enhanced implementation of the NTRUEncrypt algorithm using graphics cards. In Proceedings of the 1st International Conference On Parallel Distributed Grid Computing (PDGC), 2010. pp 168-174.
- [8] Khronos Group. OpenCL - The open standard for parallel programming of heterogeneous systems. Retrieved online 11/1/2011 from <http://www.khronos.org/opencv/>
- [9] Manavski, S.A. CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography. In Proceedings of IEEE International Conference on Signal Processing and Communications, 2007. pp 65-68.
- [10] Moss, A., Page, D., Smart, N.P. Toward acceleration of RSA using 3D graphics hardware. In Proceedings of the 11th IMA international on Cryptography and Coding (2007).
- [11] NVIDIA Corporation. CUDA 4.0 Best Practices Guide. Retrieved online 11/1/2011 from <http://developer.nvidia.com/nvidia-gpu-computing-documentation>
- [12] NVIDIA Corporation. CUDA 4.0 Programming Guide. Retrieved online 11/1/2011 from <http://developer.nvidia.com/nvidia-gpu-computing-documentation>
- [13] Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., Phillips, J. GPU Computing. Proceedings of the IEEE, vol. 96, no. 5, pp. 879-899, May 2008.
- [14] Theoharoulis, K., Antoniadis, C., Bellas, N., Antonopoulos, C.D. In Proceedings of the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 65-68.
- [15] Vasiliadis, G., Polychronakis, M., Ioannidis, S. GPU-assisted Malware. In Proceedings of the 5th IEEE International Conference on Malicious and Unwanted Software (MALWARE). October 2010.
- [16] Xingliang Wang, Xiaochao Li, Mei Zou, Jun Zhou. AES finalists implementation for GPU and multi-core CPU based on OpenCL. In Proceedings of the 2011 IEEE International Conference on Anti-Counterfeiting, Security and Identification (ASID), pp. 38-42.
- [17] Yongjin Yeom, Yongkuk Cho, Moti Yung. High-Speed Implementations of Block Cipher ARIA Using Graphics Processing Units. In Proceedings of the 2008 International Conference on Multimedia and Ubiquitous Engineering.
- [18] Kaiyong Zhao, Xiaowen Chu, Mea Wang, Yixin Jiang. Speeding up Homomorphic Hashing Using GPUs. In Proceedings of IEEE International Conference on Communications, 2009. pp 1-5.
- [19] Wenchao Zhou, Hongwei Wu, Xiaochao Li, Donghui Guo. Implementations of hardware acceleration for MD4-family algorithms based on GPU. In Proceedings of 3rd International Conference on Anti-counterfeiting, Security, and Identification in Communication, 2009. pp 571-574.
- [20] Lin Zhou, Wenbao Han. A Brief Implementation Analysis of SHA-1 on FPGAs, GPUs and Cell Processors. In Proceedings of International Conference on Engineering Computation, 2009. pp 101-104.

■