Chapter 8
Graphics Processing Units

Peter Schwabe

Abstract This chapter introduces graphics processing units (GPUs) for general-purpose computations. It describes the highly parallel architecture of modern GPUs, software-development toolchains to program them, and typical pitfalls and performance bottlenecks. Then it considers several applications of GPUs in information security, in particular in cryptography and cryptanalysis.

Graphics Processing Units (GPUs) are coprocessors that traditionally perform the rendering of 2-dimensional and 3-dimensional graphics information for display on a screen. In particular, computer games request more and more realistic real-time rendering of graphics data and so GPUs became more and more powerful highly parallel specialist computing units. It did not take long until programmers realized that this computational power can also be used for tasks other than computer graphics. For example already in 1990 Lengyel, Reichert, Donald, and Greenberg used GPUs for real-time robot motion planning [43]. In 2003, Harris introduced the term general-purpose computations on GPUs (GPGPU) [28] for such nongraphics applications running on GPUs. At that time, programming GPGPUs meant expressing all algorithms in terms of operations on graphics data, pixels, and vectors. This was feasible for speed-critical small programs and for algorithms that operate on vectors of floating-point values in a similar way as graphics data are typically processed in the rendering pipeline. The programming paradigm shifted when the two main GPU manufacturers, NVIDIA and AMD, changed the hardware architecture from a dedicated graphics-rendering pipeline to a multi-core computing platform, implemented shader algorithms of the rendering pipeline in software running on these cores, and explicitly supported GPGPUs by offering programming languages and software-development toolchains. This chapter first gives an introduction to the architectures of these modern GPUs and the tools and languages to program them. Then it highlights several applications of GPUs related to information security with a focus on applications in cryptography and cryptanalysis.

P. Schwabe (✉)
Digital Security Group, Radboud University Nijmegen, Nijmegen, The Netherlands
e-mail: peter@cryptojedi.org

8.1 An Introduction to Modern GPUs

GPUs have evolved to coprocessors of a size larger than typical CPUs. While CPUs use large portions of the chip area for caches, GPUs use most of the area for arithmetic logic units (ALUs). The main concept that both NVIDIA and AMD GPUs use to exploit the computational power of these ALUs is executing a single instruction stream on multiple independent data streams (SIMD) [23]. This concept is known from CPUs with vector registers and instructions operating on these registers. For example, a 128-bit vector register can hold four single-precision floating-point values; an additional instruction operating on two such registers performs four independent additions in parallel. Instead of using vector registers, GPUs use hardware threads that all execute the same instruction stream on different sets of data. NVIDIA calls this approach to SIMD computing "single instruction stream, multiple threads (SIMT)". The number of threads required to keep the ALUs busy is much larger than the number of elements inside vector registers on CPUs. GPU performance, therefore, relies on a high degree of data-level parallelism in the application.

To alleviate these requirements on data-level parallelism, GPUs can also exploit task-level parallelism by running different independent tasks of a computation in parallel. This is possible on all modern GPUs through the use of conditional statements. Some recent GPUs support the exploitation of task-level parallelism also through concurrent execution of independent GPU programs. Each of the independent tasks again needs to involve a relatively high degree of data-level parallelism to make full use of the computational power of the GPU, but exploitation of task-level parallelism gives the programmer more flexibility and extends the set of applications that can make use of GPUs to accelerate computations.

The remainder of this section gives an overview of the hardware architectures of modern GPUs, introduces the relevant programming languages, and discusses typical performance bottlenecks and GPU benchmarking issues. The section focuses on NVIDIA GPUs, because most of the implementations of subsequent sections target these GPUs.

8.1.1 NVIDIA GPUs

In 2006, NVIDIA introduced the Compute Unified Device Architecture (CUDA). Today all of NVIDIA's GPUs are CUDA GPUs. CUDA is *not* a computer architecture in the sense of a definition of an instruction set and a set of architectural registers; binaries compiled for one CUDA GPU do not necessarily run on all CUDA GPUs. More specifically, NVIDIA defines different *CUDA compute capabilities* to describe the features supported by CUDA hardware. The first CUDA GPUs had compute capability 1.0. In 2011, NVIDIA released GPUs with compute capability 2.1, which is known as "Fermi" architecture. Details about the different compute capabilities are described in [50, Appendix F].

A CUDA GPU consists of multiple so-called *streaming multiprocessors* (SMs). The threads executing a GPU program, a so-called *kernel*, are grouped in *blocks*. Threads belonging to one block all run on the same multiprocessor but one multiprocessor can run multiple blocks concurrently. Blocks are further divided into groups of 32 threads called *warps*; the threads belonging to one warp are executed in lock step, i.e., they are synchronized. As a consequence, if threads inside one warp diverge via a conditional branch instruction, execution of the different branches is serialized. On GPUs with compute capability 1.x all SMs must execute the same kernel. Compute capability 2.x supports concurrent execution of different kernels on different SMs.

Each SM contains several so-called CUDA cores, 8 per SM in compute capability 1.x , 32 per SM in compute capability 2.0, and 48 per SM in compute capability 2.1. One could think that for example a reasonable number of threads per SM is 8 for compute capability 1.x GPUs or 48 for compute capability-2.1 GPUs. In fact, it needs many more threads to fully utilize the ALUs; the reason is that concurrent execution of many threads on one SM is used to hide arithmetic latencies and up to some extent also memory access latencies. For compute capability 1.x, NVIDIA recommends to run at least 192 or 256 threads per SM. To fully utilize the power of compute capability 2.x, GPUs even more threads need to run concurrently on one SM. For applications that involve a very high degree of data-level parallelism, it might now sound like a good idea to just run as many concurrent threads as possible. The problem is that the register banks are shared among threads; the more threads are executed, the fewer registers are available per thread. Finding the optimal number of threads running concurrently on one SM is a crucial step to achieve good performance.

Aside from registers, each thread also has access to various memory domains. Each SM has several KB of fast shared memory accessible by all threads on this multiprocessor. This memory is intended to exchange data between the threads of a thread block, latencies are as low as for register access but throughput depends on access patterns. The shared memory is organized in 16 banks. If two threads within the same half-warp (16 threads) load from or store to different addresses on the same memory bank in the same instruction, these requests are serialized. Such requests to different addresses on the same memory bank are called *bank conflicts*, for details see [50, Sect. 5.3.2.3]. Graphics cards also contain several hundred MB up to a few GB of device memory. Each thread has a part of this device memory dedicated as so-called local memory. Another part of the device memory is global memory accessible by all threads. Access to device memory has a much higher latency than access to shared memory or registers. For details on latencies and throughput see [50, Sects. 5.3.2.1, 5.3.2.2]. Additionally, each thread has cached read-only access to constant memory and texture and surface memory. Loads from constant cache are efficient if all threads belonging to a half-warp load from the same address; if two threads within the same half-warp loaded from different addresses in the same instruction, throughput decreases by a factor equal to the number of different load addresses. Another decision (aside from the number of threads per SM) that can have huge impact on performance is what data are kept in which memory domain. Access from threads to different memories is depicted in Fig. 8.1.

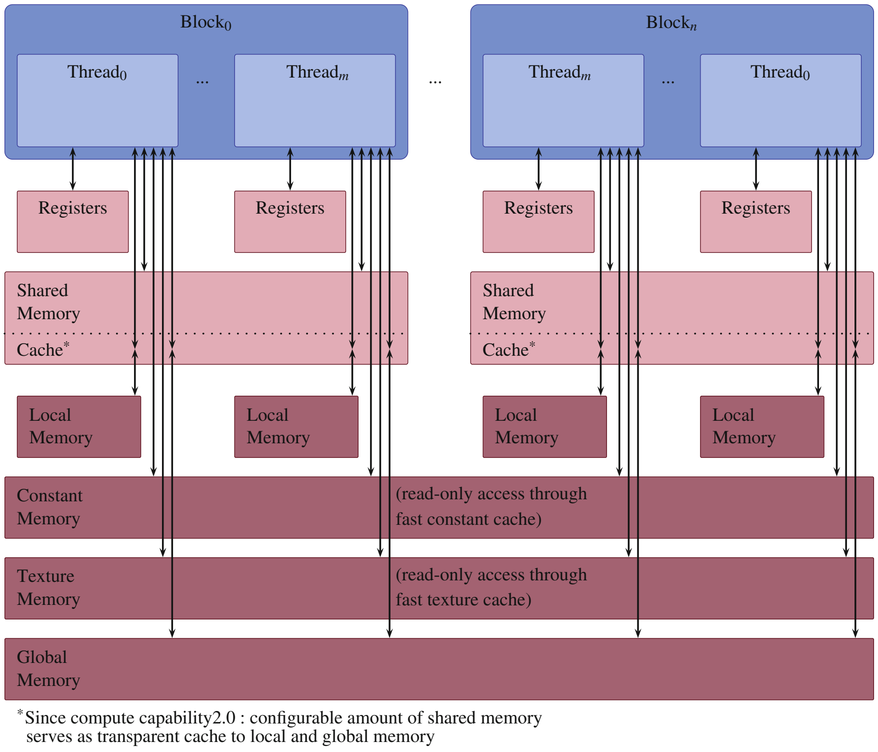

Fig. 8.1 Access to different memories from threads on NVIDIA CUDA devices. *Light-red* memories are fast; *dark-red* memories are parts of device memory with high-latency access

Communication between CPU and GPU is done by transferring data between the host memory and the GPU device memory or by mapping page-locked host memory into the GPUs address space. Asynchronous data transfers between *page-locked* host memory and device memory can overlap with computations on the CPU. For some CUDA devices since compute capability 1.1 they can also overlap with computations on the GPU. For details on data transfers to and from NVIDIA GPUs see [50, Sects. 3.4, 3.5]. Since CUDA 4.0 NVIDIA simplifies data exchange between host memory and device memory of Fermi GPUs by supporting a unified virtual address space. For details see [50, Sects. 3.2.7, 3.3.9]. The unified virtual address space is particularly interesting in conjunction with peer-to-peer memory access between multiple GPUs. This technique makes it possible to access the memory of one GPU directly from another GPU without data transfers through host memory. For details see [50, Sects. 3.2.6.4, 3.2.6.5].

8.1.2 AMD GPUs

The hardware and software technologies that allow programmers to use AMD GPUs for general-purpose computations are called AMD accelerated parallel processing (APP), formerly known as ATI Stream. For a detailed description of the architecture and the programming environment see [3].

Each APP device consists of multiple so-called *compute units*, each compute unit contains multiple *stream cores*, which, in turn, contain multiple *processing elements*. Multiple instances of a GPU program (kernel) are executed concurrently on different data, one such instance of a kernel is called a *work-item*. Multiple work-items are executed by all stream cores of one compute unit in lock step, one such group of work items executed together is called *wavefront*. The number of work items in a wavefront is hardware dependent. The programmer decides how many work items are scheduled to one compute unit in a so-called *workgroup*. Best performance is obtained, if this number is a multiple of the size of a wavefront.

In principle, different compute units can execute different kernels concurrently. However, the number of different kernels running on one APP device may be limited. All stream cores of one compute element execute the same instruction sequence consisting of very-large-instruction-word (VLIW) arithmetic instructions, control-flow instructions, and memory load and store instructions. Then up to four or five (depending on the device) instructions inside a VLIW instruction word are co-issued to the processing elements.

Similar to NVIDIA GPUs, AMD GPUs have various memories with different visibility to work items and different latencies and throughputs. The private memory is specific to each work item and is kept in a register file with very fast access. Work items inside one workgroup, i.e., running on the same compute unit, can communicate through local memory. This “local memory” is not a part of the device memory as on NVIDIA GPUs. In fact, it is very similar to what NVIDIA calls shared memory, a relatively small memory with fast access for efficient exchange of data between work items. Access to local memory is about an order of magnitude faster than access to device memory. Furthermore, all work items executing in one context have access to the global device memory and cached read-only access to a part of the device memory called constant memory. Access from work items to different memories is depicted in Fig. 8.2.

Communication with the host is done through DMA transfers between host and device memory. Computation on both the CPU and the GPU can overlap with DMA transfers.

8.1.3 Programming GPUs in High-Level Languages

With the CUDA architecture NVIDIA introduced language extensions to the C programming language that allowed to write programs that are partially executed on the GPU. The resulting programming language is called “C for CUDA”. Note that

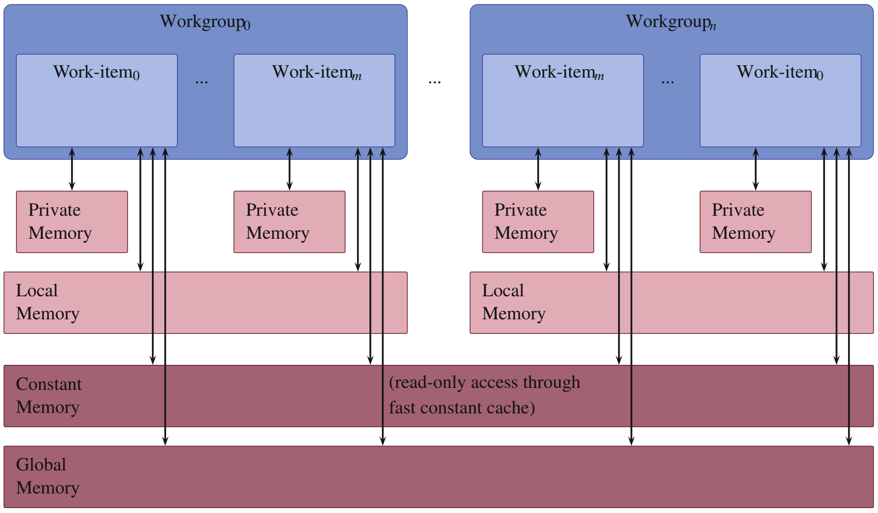

Fig. 8.2 Access to different memories from work items on AMD APP devices. *Light-red* memories are fast; *dark-red* memories are parts of device memory with high-latency access

depending on the compute capability some restrictions apply for the part of the program that is executed by the GPU, for example compute capability 1.x does not support recursive function calls. For details on C for CUDA see [50].

The first software-development tool that AMD offered for GPGPUs was called Close-to-Metal (CTM) which gave low-level access to the native instruction set of the GPU. High-level-language support was first offered in the ATI Stream SDK v1 with the ATI Brook+ language, which is based on BrookGPU developed at Stanford University [14].

Both solutions, C for CUDA and Brook+ could only be used to implement software for the respective manufacturer's GPUs. As a more portable approach, both NVIDIA and AMD now also support the OpenCL programming language and API developed by the Khronos group. This programming language is designed for the development of software for parallel computations on arbitrary heterogeneous systems. Two versions of the language have been released, OpenCL 1.0 in November 2008 [26] and OpenCL 1.1 in June 2004 [27].

Today, the recommended way to program NVIDIA GPUs is using either C for CUDA [50] or OpenCL for CUDA [51]. AMD recommends OpenCL as high-level programming language for their GPUs in their latest APP SDK [3].

The compilation process is very similar for all of the high-level languages. In a first step, the compiler separates the parts of the program that run on the CPU from the parts that run on the GPU. The CPU part is further compiled using native C or C++ compilers for the respective host architecture. The GPU part is first translated to an intermediate low-level language. For NVIDIA this language is called PTX, for AMD it is called IL. The advantage of this intermediate language is that it is somewhat device independent. More specifically, PTX code is compatible across minor revisions of the compute capability; IL code is forward compatible. The GPU driver contains a just-in-time compiler for this intermediate language. Code that needs to run on GPUs with different hardware capabilities can thus be translated only to intermediate language, final compilation to binary code is performed by the respective driver. This last compilation step can also be done off-line to produce binaries for a specific GPU architecture.

8.1.4 Programming GPUs in Assembly

Most software today is written in high-level languages, but some areas of computing still employ hand-optimized assembly routines to achieve best performance. One of these areas is high-performance computing—in computations that run for weeks or months even small performance gains are typically worth the effort of implementing parts of the software in assembly. Now that GPUs explicitly support applications in high-performance computing one would expect that the manufacturers also provide assemblers. However, this is not the case. Until the CUDA 4.0 toolkit was released in May 2011, NVIDIA offered neither an assembler nor a disassembler for their GPUs, an assembler is still not provided by NVIDIA. To fill this gap, van der Laan reverse-engineered the binary format and developed the cubin utilities [57] consisting of the disassembler `decuda` and the assembler `cudasm`.

For the Fermi GPUs (compute capability 2.0 and 2.1) NVIDIA includes the `cuobjdump` disassembler in the CUDA 4.0 toolkit. An assembler for Fermi GPUs is being developed by the asfermi project [33].

AMD documents the instruction-set architecture of their recent GPUs, for example in [2] for the Radeon R600 series, in [5] for the Radeon R700 series, and in [4] for the Evergreen series. AMD does not document the complete ELF format of the binaries and does not provide an assembler for their GPUs. Similar to NVIDIA, community projects work on assemblers that support different families of AMD GPUs [48, 53].

8.1.5 GPU Performance Bottlenecks

What makes GPUs a very interesting computing platform for many algorithms is their pure computing power. For example, an NVIDIA GTX 295 graphics card containing two GT200b GPUs can dispatch a total of 745 billion single-precision floating-point operations per second. For comparison, all four cores of a 2.4 GHz Intel Core 2 Quad CPU can dispatch a total of 57.6 billion single-precision floating-point operations, more than one order of magnitude less. One might thus expect that GPUs speed up computations by a factor of 10 or more, but as the examples in the following sections show this is not the case for many applications. The reason is that in order to make use of the computational power of GPUs, applications need to fulfill two conditions:

- The degree of data-level parallelism required to keep hundreds of threads busy is much larger than the degree of data-level parallelism that is required for the SIMD implementations of current CPUs. For example, keeping 192 threads on each of the 30 multiprocessors of 2 GPUs on an NVIDIA GTX 295 graphics card busy needs 11,520 independent data streams. Keeping the four cores of a CPU busy working on 128-bit vector registers needs just 16 such independent streams. Less data-level parallelism typically requires multiple threads to work on the same data which involves communication and thread synchronization overhead.
- GPU performance depends on memory-access patterns much more than CPU performance does. The reason is that GPUs spend most of their chip area on ALUs while CPUs spend a large part of the chip area on fast caches that reduce load and store latencies. Computations that can keep the active set of data in the available registers benefit from the large computational power of the ALUs, but the high latencies of device-memory loads and stores typically incur huge performance penalties in applications that cannot. Some applications can use the shared memory on NVIDIA GPUs or the local memory on AMD GPUs as cache, Fermi GPUs make this easy by using a configurable amount of shared memory as transparent cache. If the same data are required by all threads this is indeed a very good solution. However, if each thread requires different data in cache (for example, register content temporarily spilled to memory), the amount of shared memory per thread is typically too small. Compilers therefore use device memory for register spills. Another way to deal with high memory latencies is to run more threads and thus hide the latencies. Note that this comes at the price of a smaller number of registers per thread and even higher requirements on data-level parallelism.

Another potential bottleneck is data transfer between host memory and device memory. All modern graphics cards are connected through PCI Express. Throughput rates highly depend on the version of PCI Express, and the number of lanes. For example, the theoretical throughput of PCI Express 2.0 with 16 lanes (commonly denoted x16) is 8 GB/s in both directions. The throughput obtained in practice is considerably lower and depends on the size of data packets transmitted over the bus. For details see, for example, [20]. More serious than throughput limitations can be the latency incurred by data transfers over PCI Express, at least for applications that require frequent communication and cannot interleave communication with computations.

With these limitations in mind, it is interesting to see that GPU advertisements and also various scientific papers claim speedups by a factor of 100 and more of software running on a GPU compared to software running on a CPU. In most of the cases, a careful look at how these speedups were achieved reveals that the CPU implementation is far from state of the art, for example, it does not use the SIMD computing capabilities of modern CPUs, and the CPU implementation is not set up to run on multiple cores.

Despite these misleading comparisons found in many places, GPUs are very powerful computing devices and with careful optimization GPUs can speed up many computations considerably compared to the same computations running on a CPU. The following sections give examples of applications of GPU computing in information security and try to put the performance numbers in a meaningful perspective in comparison to state-of-the-art CPU implementations.

8.2 GPUs as Cryptographic Coprocessors

Cryptographic computations such as encryption and decryption, hashing, signature generation, and signature verification rely on high performance in software for many applications. Furthermore, most of the algorithms involved can be implemented in relatively small code size and it is feasible to hand-optimize code on the assembly level.

This is why for example advanced encryption standard (AES) and RSA encryption were among the algorithms that were implemented using shader instructions of the graphics rendering pipeline of traditional GPUs [29, 47, 62].

In 2006, before CUDA was introduced by NVIDIA, Cook and Keromytis published a book on cryptography on graphics cards [21]. This book claims that using GPUs for cryptography has two additional advantages aside from speeding up computations:

- The authors suggest that GPU implementations may be more resistant to (at least existing) side-channel attacks. They do not claim that GPU implementations are inherently protected against any side-channel attacks that work against CPU implementations. In fact, there is no immediate reason to believe that GPUs generally offer better protection against any side-channel attacks than CPUs. Certainly one of the most relevant class of attacks, namely cache-timing attacks (see, e.g., [56]) will not work on GPUs that have uncached access to memory, but at least the most recent NVIDIA GPUs use part of their shared memory as transparent cache for access to the GPUs main memory [49].
- Chapter 3 of [21] describes a video-streaming service that uses GPUs to decrypt video data that shall only be displayed but never be stored or modified. The system uses the GPU as “the only trusted component in a spyware-safe system”.
  This idea starts from the assumption that GPUs and graphics drivers are more trustworthy than the operating system for computations involving sensitive data such as cryptographic keys. This is a dangerous assumption to make, attackers controlling the operating system can also exchange the graphics driver, there is not even a guarantee that any code really runs on the GPU.

When using GPUs for cryptographic computations, one should keep in mind that GPUs and graphics drivers are not designed for computations on sensitive data and should be used for such computations only with precaution. For instance, on various graphics cards it is possible for a computing kernel to read out parts of the memory content left behind by a previously executed kernel. Keeping cryptographic keys in these parts of the memory can be used to speed up computations—for example, a key can be expanded once and be left in constant memory for all subsequent kernel launches as suggested in [52]. On the other hand, this can also be a serious security threat in multi-user environments if one user manages to launch a GPU kernel that reads out the key of another user.

In environments where data in GPU memory can be protected, for example on a single-user server, or with careful protections to avoid memory readout, modern GPUs can be used as powerful cryptographic coprocessors for throughput-oriented applications.

8.2.1 AES on GPUs

In particular, the possibility to implement the AES, the most widely used symmetric encryption algorithm, on GPUs has attracted a lot of attention. AES is a block cipher with supported key sizes of 128, 192, and 256 bits and a block size of 128 bits. Most implementations focus on AES with 128-bit keys. In this setting, the key is first expanded into 11 round keys $K_0, \ldots, K_{11}$. Each 128-bit input block (*state*) is then transformed into 10 rounds, each round involving one of the 11 round keys. The first round key K_0 is xored to the block before the first round. The most common implementation technique for AES, described in [22, Sect. 5.2.1], operates on 32-bit words and uses 4 lookup tables `T0`, `T1`, `T2`, and `T3` of size 1 KB (256 32-bit words) each. The 128-bit state is represented as 4 such 32-bit words. The operations of one round of AES in C notation is given in Listing 1.

Listing 1 One round of AES encryption in C, the 128-bit input state is in 32-bit unsigned integers `y0`, `y1`, `y2`, `y3`, the output state is in 32-bit unsigned integers `z0`, `z1`, `z2`, `z3`; the 128-bit round key is in 32-bit unsigned integers `k0`, `k1`, `k2`, `k3`.

```
z0 = T0[ y0 >> 24         ] ^ T1[(y1 >> 16) & 0xff] \
   ^ T2[(y2 >>  8) & 0xff] ^ T3[ y3         & 0xff] ^ k0;
z1 = T0[ y1 >> 24         ] ^ T1[(y2 >> 16) & 0xff] \
   ^ T2[(y3 >>  8) & 0xff] ^ T3[ y0         & 0xff] ^ k1;
z2 = T0[ y2 >> 24         ] ^ T1[(y3 >> 16) & 0xff] \
   ^ T2[(y0 >>  8) & 0xff] ^ T3[ y1         & 0xff] ^ k2;
z3 = T0[ y3 >> 24         ] ^ T1[(y0 >> 16) & 0xff] \
   ^ T2[(y1 >>  8) & 0xff] ^ T3[ y2         & 0xff] ^ k3;
```

To achieve the required degree of parallelism, GPU implementations of AES typically either consider many independent streams that are encrypted in parallel or they use a parallel mode of operation such as ECB or CTR that allows to encrypt blocks of a single stream independently. The most important decision to make for high-performance AES encryption on GPUs is how to use the available memory domains. CPU implementations store lookup tables and expanded keys in RAM, after some rounds of AES the tables will be in level-1 cache and lookups are fast. On most GPUs a straightforward adaptation of this approach, placing tables and expanded keys in device memory, will incur high latency penalties, because access to device memory is uncached (except for NVIDIA Fermi GPUs where part of the shared memory is used as transparent cache). A better approach is to place the lookup tables in the fast shared memory of NVIDIA GPUs or the local memory of AMD GPUs. Recall that loads from shared memory on NVIDIA GPUs can be as fast as register access, but that throughput and latency depend on the access pattern. AES table lookups have an unpredictable access pattern, so one must expect penalties due to memory bank conflicts. One solution to avoid these penalties is to store multiple copies of the lookup tables in the fast memory, such that each entry is available on each memory bank. If shared memory is not large enough to hold these copies of the tables, it may still be possible to store copies of only one of the tables and obtain entries of the other tables through rotations (see, e.g., [22, Sect. 5.2.1]). The best combination of optimization techniques depends on the target GPU.

Not only the decision about location and layout of the lookup tables is important, also handling of the round keys influences performance. This is relatively easy if one big stream is encrypted in a parallel mode of operation. In this case, all threads use the same key and it can be stored in constant memory. Unlike lookups from the tables, the round keys are accessed in a completely predictable pattern; they are broadcasted to all threads which is exactly what the constant memory is made for. The situation is different for the encryption of many independent streams under different keys. If each thread needs different round keys, there is not enough fast memory on most GPUs to store all these round keys. Instead of loading round keys from slow device memory, it may be a better choice to expand the key on the fly. Again, the best solution highly depends on the specific target GPU.

A completely different approach to implement AES is bitslicing. This technique was first introduced for the Data Encryption Standard (DES) by Biham in [12] and has also been used for various AES implementations [37, 41, 45]. The idea of this technique is transposition of data: Instead of storing a 128-bit state in, e.g., 4 32-bit registers, it uses 128 registers, 1 register per bit. This representation of data allows to simulate a hardware implementation, logical gates become bit-logical instructions. For just one computation this is not efficient, but if all n bits of registers are used to perform computations on n independent streams, this can be very efficient. Note that on top of the high degree of parallelism required for GPU computations, bitslicing requires another factor of n of parallelism, n being the register width.

Various GPU implementations of AES are described in the literature. In [63], Yang and Goodman describe different implementations of AES for AMD GPUs. Their bitsliced implementation aims at key search, so keys need to be expanded into round keys on the fly. On an AMD HD 2900 XT GPU this implementation performs encryption of one block under 145 million keys per second, this corresponds to a throughput of 18.5 Gbit/s. For the lookup-table-based implementation, they report an AES encryption throughput of 3.5 Gbit/s on an AMD HD 2900 XT GPU.

The implementation by Manavski described in [44] uses a lookup-table-based approach to achieve a peak throughput of 8.28 Gbit/s on an NVIDIA 8800 GTX graphics card (G80 GPU); to achieve this peak throughput at least 8 MB of data need to be encrypted under the same key. This implementation exploits parallelism inside AES, four threads perform the transformation of one 128-bit block. Harrison and Waldron report a throughput of 15.423 Gbit/s in [30] for their lookup-table-based implementation of AES on an NVIDIA G80 GPU. This peak performance is achieved for input messages of $\geq$ 65 MB, overhead from data transfers to and from the GPU are not included in the benchmarks. Both the implementation in [44] and the implementation in [30] achieve a significantly lower throughput when data transfers are included in the benchmarks: 2.5 Gbit/s for [44] and 6.9 Gbit/s for [30].

Two more recent papers report speeds beyond 30 Gbit/s on NVIDIA GPUs. Osvik, Bos, Stefan, and Canright in [52] describe an implementation of AES with 128-bit keys that achieve 30.9 Gbit/s throughput on one GPU of an NVIDIA GTX 295 graphics card (containing 2 GT200b GPUs). The implementation interleaves data transfers with computations by using page-locked host memory. Interleaving data transfers with kernel execution were not possible for the GPUs used for benchmarking in [30, 44]. This throughput is achieved for encryption under one key in constant memory, but the paper also describes an implementation with on-the-fly key schedule suitable for key-search applications, that achieve a throughput of 23.8 Gbit/s. Jang, Han, Han, Moon, and Park present a GPU-accelerated SSL proxy in [36]. For the AES implementation included in this proxy, they report 32.8 Gbit/s on an NVIDIA GTX 285 graphics card (GT200b GPU), not including data transfers. They also report detailed performance numbers of AES encryption in the nonparallel CBC mode for different numbers of independent streams on an NVIDIA GTX 580 graphics card (GF110 GPU).

Note that these high throughputs of AES on GPUs can only be achieved by performing AES encryption on thousands of blocks in parallel. This amount of data-level parallelism can certainly be found for some database applications or when writing large amounts of data to an encrypted hard disk. The encryption of typically small Internet packages in applications that do not just need high throughput but also low latency will still do better with a CPU-based approach, not only when using CPUs that support AES in hardware. For example, the bitsliced implementation for Intel processors presented in [37] encrypts 1,500-byte packets in 7.27 cycles per byte on a 2,668 MHz Intel Core i7 920 CPU. This corresponds to a throughput of more than 11.7 Gbit/s on 4 cores.

8.2.2 Asymmetric Cryptography on GPUs

Asymmetric cryptographic primitives can be accelerated by laying off the computations from the CPU to the GPU. As for symmetric primitives like AES one way to obtain the necessary degree of parallelism is to consider operations on many independent messages. However, there is another source for parallelism inherent in the algorithms. Most state-of-the-art asymmetric algorithms involve operations on large integers, for example RSA signature generation is the computation of m^d mod n, where m, d and n are integers of 1,024 bits or larger. Arithmetic on such integers, in particular multiplication, squaring and modular reduction, needs to be decomposed in many operations on machine words. Elliptic-curve cryptography involves modular arithmetic on integers of smaller size, typically between 160 and 256 bits, but arithmetic on those integers still decomposes into many operations on machine words. For example, when using a multiplier with 32-bit output, schoolbook multiplication of two 256-bit integers requires 256 multiplications of 16-bit limbs and 240 additions of the 32-bit multiplication outputs. Most of these operations are independent and can be done in parallel by multiple threads. Exploiting such parallelism inside one computation has some obvious advantages. If multiple threads process one input stream together, fewer independent input streams are required to make use of the computational power of the GPU. This makes GPU computations attractive also for applications that require low latency rather than high throughput. Furthermore, when multiple threads carry out one computation together the overall amount of data involved in the computations is smaller; this can be used to fit all data into memory domains that offer low-latency access. However, exploiting data-level parallelism inside computations like big-integer multiplication comes with the disadvantage that it involves overhead from thread synchronization and exchange of data between treads.

Several papers describe implementations of RSA on modern graphics cards. In [55], Szerwinski and Güneysu describe a CUDA implementation that performs 813 modular exponentiations (RSA encryption) of 1,024-bit integers on a NVIDIA 8800 GTS graphics card. This paper furthermore reports a throughput of 104.3 modular exponentiations for 2,048-bit RSA encryption. Harrison and Waldron in [31] focus on RSA decryption and report 5536.75 RSA-1024 decryptions per second on an NVIDIA 8800 GTX graphics card. This computation can make use of the Chinese Remainder Theorem to perform arithmetic on half-size integers. The RSA implementation included in the SSL proxy described in [36] can perform for example 74732 RSA-1024 encryptions or 12044 RSA-2048 encryptions per second on an NVIDIA GTX 580 graphics card. What is particularly interesting about this implementation is that it does not purely focus on throughput but also needs to keep the latency low enough for the application in the SSL proxy. For RSA-1024 the latency is at 3.8 ms, for RSA-2048 it is 13.83 ms.

To put this into perspective to what is currently possible on CPUs, the eBACS benchmarking project [11] reports, for example, more than 11,000 1,024-bit integer exponentiations per second on all six cores of an AMD Phenom II X6 1090T. Again, this speed does not require the large number of independent parallel computations that GPU implementations need and although it is much slower from a pure throughput perspective, it may be the better choice for applications that do not process multiple messages in parallel.

Elliptic-curve cryptography has been implemented on GPUs. Szerwinski and Güneysu report 1,412 scalar multiplications on the NIST P-224 elliptic curve on an NVIDIA 8800 GTS graphics card in [55]. On the same curve but the more recent

NVIDIA GTX 285 graphics card Antão, Bajard, and Sousa report 9990 scalar multiplications per second. More than an order of magnitude slower at significantly lower security is the implementation of scalar multiplication on an elliptic-curve over a binary field described in [19]. Cohen and Parhi report only 96.5 scalar multiplications per second.

Elliptic-curve scalar multiplication has received more attention on CPUs, for example [10] reports 226872 cycles for a scalar multiplication on a 255-bit elliptic curve on an Intel Xeon E5620 CPU running at 2.4 GHz. This corresponds to more than 40,000 scalar multiplication per second on all four cores. Even faster speeds for CPU implementations are reported in [34] for scalar multiplication on elliptic curves with efficiently computable endomorphisms. These comparative numbers may suggest that GPU implementations of elliptic-curve cryptography cannot compete with state-of-the-art CPU implementations, not even in throughput-oriented applications. However, the next section describes implementations of elliptic-curve operations on GPUs for cryptanalysis that outperform CPU implementations. The reason that there are no faster GPU implementations targeting constructive applications may be that there are simply not many applications that require only throughput and can ignore latency.

An asymmetric cryptosystem that appears to be much better suited for implementation on GPUs than elliptic-curve cryptography or RSA is NTRU. The central operation for encryption and decryption is convolution which can be carried out by many threads without significant communication or synchronization due to its parallel structure. In [32], Hermans, Vercauteren, and Preneel describe an implementation of NTRU with a set of parameters that aims at the 256-bit security level. This implementation is able to perform 218,000 encryption operations per second on an NVIDIA GTX 280 graphics card (GT200 GPU).

8.3 GPUs in Cryptanalysis

Cryptanalytical computations are in many ways similar to cryptographic computations. In many cases, breaking a cryptographic system means executing the same or very similar computations that are used in the constructive use of the cryptosystem. One example is brute-force key recovery of symmetric ciphers that simply performs encryption with many different keys. Another example is hash-function collision search with the computationally most expensive part being computing hashes. An example in the cryptanalysis of asymmetric systems is Pollard's rho algorithm to solve the discrete logarithm problem (DLP). Again the computationally most expensive part are the same or very similar operations in the same mathematical structures that are involved in the legitimate use of the DLP-based system.

In three very important points, cryptanalytical computations are different from cryptographic computations and all three make them even better suited for GPUs. First, they typically involve an arbitrary amount of data-level parallelism, the same computations are carried out on huge amounts of independent data; this is exactly the sort of computations that GPUs are best at. Second, many of these computations do not care about latency, they are purely throughput oriented. Third, there is no confidential data involved that needs to be protected, one could say that the opposite is true, revealing the confidential data is the target of the computation.

The most obvious applications of GPUs for cryptanalysis are attacks against symmetric encryption and hash functions. Various commercial solutions for password recovery already include GPU implementations to speed up the computations. These tools typically try out many different passwords from a given word list and either compare with given hash values or derive symmetric keys from a list of known passphrases to recover the content of encrypted files.

The power of GPUs was also used by the winner of Engineyard's SHA-1 programming contest: The task was to find an input to SHA-1 that has minimal Hamming distance to a given hash value. Lange in [42] reports that code by Bernstein is able to compute more than 328 million hashes per second on an NVIDIA GTX 295 graphics card. Each of these hashes required computation of only one 64-byte block of input, so this corresponds to a throughput of more than 167 Gbit/s. As a comparison, all four cores of a 2.4 GHz Intel Core 2 Quad Q6600 CPU involved in the same computation computed 47 million hashes per second. Also the SHA-3 candidates have been implemented on GPUs, password recovery being the most obvious application. In [13], Bos and Stefan describe implementations of all of the SHA-3 round-2 candidates on NVIDIA GT200 GPUs. The reported throughputs reach from 0.9 Gbit/s for Cubehash 16/1 up to 36.8 Gbit/s for Blake-32 and BMW-256 on one GPU of an NVIDIA GTX 295 graphics card. Again to put this into perspective, on a recent CPU, the Intel Core i7-2600K, hashing with Blake-32 takes 6.68 cycles/byte [11]; this corresponds to a throughput of 16.29 Gbit/s.

These applications in password recovery are quite straightforward, but GPUs have also been used for cryptanalysis of asymmetric systems. One of the most famous problems closely related to the RSA cryptosystem is the factorization of large numbers. A critical step inside the factorization of large RSA numbers with the number-field sieve is the factorization of many smaller numbers using the elliptic-curve factorization method (ECM). In [9], Bernstein, Chen, Cheng, Lange, and Yang describe an implementation of ECM for 280-bit numbers. This implementation running on both GT200b GPUs of an NVIDIA GTX 295 graphics card outperforms a state-of-the-art CPU implementation running on all 4 cores of an Intel Core 2 Quad Q9550 by a factor of more than 2.8. The GPU implementation tries 400.7 curves per second, the CPU implementation 142.17 curves per second. A much higher ECM throughput for slightly smaller numbers is reported in [8]. For example, for 210-bit numbers a GTX 295 graphics card is reported to try 4,928 curves per second. Although these numbers are not as impressive as the speedups achieved by using GPUs in symmetric cryptanalysis, the results show that GPUs can also be used to speed up elliptic-curve arithmetic.

This is confirmed for elliptic curves over binary fields in [7]. As part of a large effort to solve Certicom's elliptic-curve discrete-logarithm-problem (ECDLP) challenge ECC2K-130 [15, 16], this paper presents an implementation of Pollard's rho algorithm for GT200b GPUs. On the two GPUs inside the GTX 295 graphics card, this implementation is able to perform 63 million Pollard rho iterations per second. As a comparison, the CPU implementation computing the same iteration function described in [6] performs 22.45 million iterations per second on all 4 cores of an Intel Core 2 Extreme Q6850 CPU.

GPUs have also been considered for solving the DLP on elliptic curves over large prime fields. The implementation described in [17] targets an ECDLP on a 109-bit prime curve and is reported to "have generated about 320.000 points/second" on an NVIDIA 8800 GTS graphics card with a G92 GPU. This probably means 320,000 iterations per second, but it is unclear what the exact performance of the implementation is.

8.4 Malware Detection on GPUs

Similar to cryptographic applications, malware-detection software is expected to operate in the background with as little influence on the system's performance as possible. A large computational task of virus detection is pattern matching of byte sequences found in files with known signatures of malware. This task is highly parallel, so it is an application that can run at high speed on GPUs.

Seamans and Alexander describe an implementation of parallel virus signature matching for NVIDIA GPUs in [54]. The authors integrated this implementation into the ClamAV virus scanner [18] and compare the performance of this implementation running on an NVIDIA GTX 7800 graphics card to the original CPU implementation running on an unspecified 3-GHz Intel Pentium 4 CPU; the authors do not specify the number of CPU cores used for this comparison. The speedup obtained by running the pattern matching on the GPU depends on the number of matches, because matches need to be communicated back to the CPU. If no matches are found, the GPU implementation is 27 times faster than the CPU implementation; this factor drops to 17 at a match rate of 1 % and further to 11 at a match rate of 50 %.

In [59], Vasiliadis and Ioannidis describe an implementation of virus-signature pattern matching targeting more recent NVIDIA GPUs. Their implementation filters out clean, unsuspicious regions, it is included as a preprocessing step into the ClamAV [18] virus scanner. The authors achieve a 100-times higher throughput with this approach running on an NVIDIA GTX295 graphics card compared to the CPU-only virus scanner running on 1 core of an Intel Xeon E5520 CPU. Compared to the CPU implementation running on 8 cores of 2 CPUs, the speedup is still 10-fold.

The approach of using the GPU as a coprocessor for malware detection is not purely academic. In December 2009, Kaspersky announced that they incorporated an implementation of the "similarity service" for NVIDIA Tesla cards into their infrastructure. The press release [38] does not give much detail but claims a 360-times speedup of the GPU implementation running on an NVIDIA Tesla S1070 compared to the a CPU implementation running on a 2.6 GHz Intel Core 2 Duo processor. This comparison does not give details about the number of CPU cores used, it also does not say whether the speedup is obtained from running the GPU code on one or all four GPUs included in the Tesla S1070.

Signature matching is also one of the main performance bottlenecks of network-intrusion-detection systems. Consequently, GPUs can also be used to speed up such systems. This was first described by Jacob and Brodley who use a traditional GPGPU approach targeting the NVIDIA 6800 GT graphics card in [35]. They conclude that with their GPU pattern-matching extension to the open-source intrusion detection system Snort "there was no appreciable speedup in packet processing under normal-load conditions". A more efficient approach targeting the NVIDIA 8600 GT graphics card is described in [58]. Vasiliadis, Antonatos, Polychronakis, Markatos, and Ioannidis present a GPU pattern-matching extension of Snort that increases the overall Snort throughput capacity by a factor of two compared to CPU-only Snort running on a 3.4 GHz Intel Pentium 4 processor. The most comprehensive solution for intrusion detection involving GPUs to date is presented in [61]. Vasiliadis, Polychronakis, and Ioannidis describe a Snort-based intrusion detection solution that exploits parallelism on multiple levels. The system makes use of multiple GPUs and multiple CPU cores and copes with a network throughput of 5.2 GBit per second. This performance number was achieved on a system with two NVIDIA GTX 480 graphics cards and two Intel Xeon E5520 CPUs. The pure pattern-matching step reaches a peak performance of more than 70 GBit per second on the two graphics cards.

8.5 Malware Targeting GPUs

GPUs can not only be used to accelerate malware detection, malware itself can also use GPUs to hide from virus scanners. In [60], Vasiliadis, Polychronakis, and Ioannidis describe an implementation of a malware unpacker running on an NVIDIA GPU. The complete malware package consists of two parts, the unpacker running on the GPU and the actual malware that runs on the CPU. These two parts communicate through host memory mapped into the GPUs address space.

Unpackers are one of the most common techniques to hide malware from scanners: The malware code is packed or encrypted in some way and gets unpacked (decrypted) only when it is actually executed. The advantage from the malware author's perspective of using GPU code for the unpacker is twofold as it offers better protection against detection by both static and dynamic malware-detection systems. Static systems try different known unpacking techniques to recover the original malware. This becomes harder if the computational power of the GPU is used for computationally more expensive unpacking algorithms. Dynamic unpacking tools use the unpacker that is included in the malware, for example inside a sandbox or virtual machine. At least existing dynamic tools do not support GPU binaries and would thus fail.

As a second step [60], Yang and Goodman also describe GPU-assisted run-time polymorphism on the function level. The malware binary is never fully decrypted, only the currently executed function resides in memory, when returning from a function call the function is encrypted again and the next function context is decrypted.

The implementations are still just a proof of concept and there have been no reports of real-world malware using the GPU to hide from scanners. Some of the claimed advantages of using the GPU to hide malware can obviously be addressed by malware-detection tools also using the GPU. Others will require better tools for static and dynamic analysis of GPU code. It will be interesting to see whether or how much GPUs become a new battlefield in the everlasting fight between malware and malware detection.

8.6 Accessing GPUs from Web Applications

Software becomes more and more Web centric; programs such as office suites, image-processing software, and games, which traditionally run directly on a computer, are now implemented as applications running inside a Web browser. The most consistent implementation of this approach is Google's Chromium OS, an operating system that is designed to run a Web browser as only application—all other software is Web applications running inside this browser.

As a consequence of higher demands for advanced graphics in Web applications, various technologies have been developed to let those applications access the GPU. The most prominent three approaches are WebGL developed initially by Mozilla and now by the Khronos group [39], Silverlight 5 developed by Microsoft [46], and Flash 11 developed by Adobe.

All of these approaches have in common that they expose the graphics driver and hardware to software originating from the Internet and thus from typically untrusted sources. The implications for security of this approach have so far been discussed primarily for WebGL. In March 2011, version 1.0 of the specification of WebGL was released by the Khronos group. Browsers supporting this specification include Mozilla's Firefox and Google's Chrome. Only about 2 months later, Forshaw publicized an article [24] that describes several security issues in these implementations and claims that these are actually caused by design flaws in WebGL. One of these issues is the possibility to remotely exploit vulnerabilities in the graphics driver to crash or freeze the system. Another one is a cross-site timing attack that extracts image data processed on the GPU. A follow-up article by Forshaw, Stone, and Jordon [25] describes an attack targeting the WebGL implementation of Firefox. In this attack, a malicious website can take screenshots of arbitrary applications running on the client computer.

Khronos has reacted to these articles in a WebGL security whitepaper [40] that describes approaches to address the security issues. These approaches can not all be implemented only on the browser side but need support on the graphics-driver side.

Even without any vulnerabilities in the framework, the computational power of GPUs enables attacks that would otherwise be infeasible. For example, the JavaScript bitcoin miner of bitp., it has been discontinued because "Javascript is just too slow to mine bitcoins" [1]. This would certainly be different with the computing power of GPUs open to Web applications. Mining bitcoins on the GPU in the background while a user is visiting a website could on the one hand be a legitimate new way of funding websites (if the user is asked for permission), on the other hand it would most likely also be done silently and thus become sort of a Web Trojan.

The discussion about WebGL security and more general security issues related to exposing the GPU and the graphics driver to untrusted code from the Internet is still ongoing. On the one hand WebGL, Silverlight 5, and Flash 11 are still very young technologies and maybe some of the vulnerabilities are just teething troubles. On the other hand, the concept of letting Web applications access the driver layer of a client's operating system flies in the face of conventional wisdom that tells us that untrusted code should be kept as far away from any critical parts of a system as possible. The future will have to show what changes are required to browsers, operating systems, and drivers to deal with current and future security vulnerabilities and whether it is actually possible to establish these technologies without exposing their users to severe risks.

References

1. "1bitc0inplz". Edited forum post on bitcointalk.org, 2011. https://bitcointalk.org/index.php?topic=9042.0.
2. Advanced Microdevices *Inc. R600-Family Instruction Set Architecture*, 2008. http://developer.amd.com/gpu_assets/r600isa.pdf.
3. Advanced Microdevices Inc. *AMD Accelerated Parallel Processing OpenCL Programming Guide, rev. 1.3f*, 2011. http://developer.amd.com/sdks/AMDAPPSDK/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf.
4. Advanced Microdevices Inc. *Evergreen Family Instruction Set Architecture Instructions and Microcode*, 2011. http://www.amddevcentral.com/sdks/AMDAPPSDK/assets/AMD_Evergreen-Family_Instruction_Set_Architecture.pdf.
5. Advanced Microdevices Inc. *R700-Family Instruction Set Architecture*, 2011. http://developer.amd.com/gpu_assets/R700-Family_Instruction_Set_Architecture.pdf.
6. Daniel V. Bailey, Lejla Batina, Daniel J. Bernstein, Peter Birkner, Joppe W. Bos, Hsieh-Chung Chen, Chen-Mou Cheng, Gauthier Van Damme, Giacomo de Meulenaer, Luis Julian Dominguez Perez, Junfeng Fan, Tim Güneysu, Frank Gürkaynak, Thorsten Kleinjung, Tanja Lange, Nele Mentens, Ruben Niederhagen, Christof Paar, Francesco Regazzoni, Peter Schwabe, Leif Uhsadel, Anthony Van Herrewege, and Bo-Yin Yang. Breaking ECC2K-130. Cryptology ePrint Archive, Report 2009/541, 2009. http://eprint.iacr.org/2009/541/.
7. Daniel J. Bernstein, Hsieh-Chung Chen, Chen-Mou Cheng, Tanja Lange, Ruben Niederhagen, Peter Schwabe, and Bo-Yin Yang. ECC2K-130 on NVIDIA GPUs. In Guang Gong and Kishan Chand Gupta, editors, *Progress in Cryptology - INDOCRYPT 2010*, volume 6498 of LNCS, pp. 328–346. Springer, 2010. http://cryptojedi.org/papers/#gpuev1l.
8. Daniel J. Bernstein, Hsueh-Chung Chen, Ming-Shing Chen, Chen-Mou Cheng, Chun-Hung Hsiao, Tanja Lange, Zong-Cing Lin, and Bo-Yin Yang. The billion-mulmod-per-second pc. In *Workshop Record of SHARCS'09: Special-purpose Hardware for Attacking Cryptographic Systems*, pp. 131–144, 2009. http://www.hyperelliptic.org/tanja/SHARCS/record2.pdf.
9. Daniel J. Bernstein, Tien-Ren Chen, Chen-Mou Cheng, Tanja Lange, and Bo-Yin Yang. ECM on graphics cards. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009*, volume 5479 of LNCS, pp. 483–501. Springer, 2009. http://cr.yp.to/papers.html#gpuecm.
10. Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures, 2011. http://cryptojedi.org/papers/#ed25519.

11. Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems. http://bench.cr.yp.to (Accessed Nov. 3, 2011).
12. Eli Biham. A fast new DES implementation in software. In Eli Biham, editor, *Fast Software Encryption*, volume 1267 of *LNCS*, pp. 260–272. Springer, 1997. http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/1997/CS/CS0891.pdf.
13. Joppe W. Bos and Deian Stefan. Performance analysis of the SHA-3 candidates on exotic multi-core architectures. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems - CHES 2010*, volume 6225 of *LNCS*, pp. 279–293. Springer, 2010. http://www.ee.cooper.edu/stefan/pubs/conference/ches2010.pdf.
14. BrookGPU. http://graphics.stanford.edu/projects/brookgpu/, Accessed Nov. 5, 2011.
15. Certicom ECC Challenge, 1997. http://www.certicom.com/images/pdfs/cert_ecc_challenge.pdf, Accessed Nov. 6, 2011.
16. ECC Curves List, 1997. http://www.certicom.com/index.php/curves-list, Accessed Nov. 6, 2011.
17. Marta Chinnici, Salvatore Cuomo, Maurizio Laporta, Alberto Pizzirani, and Silvio Migliori. CUDA based implementation of parallelized Pollard's rho algorithm for ECDLP. In *Final Workshop of Grid Projects*, *"Pon Ricerca 2000–2006, Avviso 1575"*, 2009. http:///www.cresco.enea.it/Documenti/web/presentazioni/ProceedingsCatan%ia2009/7chinnici.pdf.
18. Clam AntiVirus. http://clamav.net, Accessed Nov 1, 2011.
19. Aaron E. Cohen and Keshab K. Parhi. GPU accelerated elliptic curve cryptography in $GF(2^m)$. In *53rd IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 57–60. IEEE, 2010.
20. James Coleman and Perry Taylor. Hardware level IO benchmarking of PCI Express. White Paper, Intel Corporation, 2008. ftp://download.intel.com/design/intarch/PAPERS/321071.pdf.
21. Debra L. Cook and Angelos D. Keromytis. *CryptoGraphics: Exploiting Graphics Cards For Security*, volume 20 of *Advances in Information Security*. Springer, 2006.
22. Joan Daemen and Vincent Rijmen. AES proposal: Rijndael, version 2, 1999. http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf.
23. Michael J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966. http://ieeexplore.ieee.org/iel5/5/31091/01447203.pdf.
24. James Forshaw. WebGL - a new dimension for browser exploitation. Blog entry on the Context Information Security Ltd. blog, 2011. http://www.contextis.com/resources/blog/webgl/.
25. James Forshaw, Paul Stone, and Michael Jordon. WebGL - more WebGL security flaws. Blog entry on the Context Information Security Ltd. blog, 2011. http://www.contextis.com/resources/blog/webgl2/.
26. Khronos OpenCL Working Group. *The OpenCL Specification, Version* 1.0, 2008. http://www.khronos.org/registry/cl/specs/opencl-1.0.pdf.
27. Khronos OpenCL Working Group. *The OpenCL Specification, Version* 1.1, 2010. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf.
28. Mark Harris. *Real-Time Cloud Simulation and Rendering*. Ph.D. thesis, University of North Carolina at Chapel Hill, 2003. http://www.markmark.net/dissertation/index.html.
29. Owen Harrison and John Waldron. AES encryption implementation and analysis on commodity graphics processing units. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, volume 4727 of *LNCS*, pages 209–226. Springer, 2007.
30. Owen Harrison and John Waldron. Practical symmetric key cryptography on modern graphics hardware. In *USENIX Security Symposium*, pages 195–209. Usenix Association, 2008.
31. Owen Harrison and John Waldron. Efficient acceleration of asymmetric cryptography on graphics hardware. In Bart Preneel, editor, *Progress in Cryptology - AFRICACRYPT 2009*, volume 5580 of *LNCS*, pages 350–367. Springer, 2009.
32. Jens Hermans, Frederik Vercauteren, and Bart Preneel. Speed records for NTRU. In Josef Pieprzyk, editor, *Topics in Cryptology - CT-RSA 2010*, volume 5985 of LNCS, pages 73–88. Springer, 2010.

33. Yunqing Hou. asfermi: An assembler for the NVIDIA Fermi instruction set, 2011. http://code.google.com/p/asfermi/, Accessed Nov. 1, 2011.
34. Zhi Hu, Patrick Longa, and Maozhi Xu. Implementing 4-dimensional GLV method on GLS elliptic curves with j-invariant 0. Cryptology ePrint Archive, Report 2011/315, 2011. http://eprint.iacr.org/2011/315.
35. Nigel Jacob and Carla Brodley. Offloading IDS computation to the GPU. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pp. 371–380. IEEE Computer Society, 2006. http://www.acsac.org/2006/papers/74.pdf.
36. Keon Jang, Sangjin Han, Seungyeop Han, Sue Moon, and KyoungSoo Park. SSLShader: cheap SSL acceleration with commodity processors. In David G. Andersen and Sylvia Ratnasamy, editors, *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*. ACM Press, 2011. http://www.usenix.org/events/nsdi11/tech/full_papers/Jang.pdf.
37. Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, volume 5747 of LNCS, pp. 1–17. Springer, 2009. http://cryptojedi.org/papers/#aesbs.
38. Kaspersky Lab. Kaspersky Lab utilizes NVIDIA technologies to enhance protection, 2009. http://www.kaspersky.com/about/news/business/2009/Kaspersky_Lab_utilizes_NVIDIA_technologies_to_enhance_protection.
39. WebGL - OpenGL ES 2.0 for the web, 2011. http://www.khronos.org/webgl/.
40. WebGL security, 2011. http://www.khronos.org/webgl/security/, Accessed Nov. 4. 2011.
41. Robert Könighofer. A fast and cache-timing resistant implementation of the AES. In Tal Malkin, editor, *Topics in Cryptology* - CT-RSA 2008, volume 4964 of LNCS, pages 187–202. Springer, 2008.
42. Tanja Lange. CodingCrypto's page on Engineyard's programming contest, 2009. http://www.win.tue.nl/cccc/sha-1-challenge.html, Accessed Nov. 5, 2011.
43. Jed Lengyel, Mark Reichert, Bruce R. Donald, and Donald P. Greenberg. Real-time robot motion planning using rasterizing computer graphics hardware. *SIGGRAPH Computer Graphics*, 24(4):327–335, 1990. http://dl.acm.org/citation.cfm?id=97915&CFID=67002344&CFTOKEN=69396923.
44. Svetlin A. Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In 2007 *IEEE International Conference on Signal Processing and Communications (ICSPC 2007)*, pages 65–68. IEEE, 2007. http://www.manavski.com/downloads/PID505889.pdf.
45. Mitsuru Matsui. How far can we go on the x64 processors? In Matthew Robshaw, editor, Fast *Software Encryption*, volume 4047 of LNCS, pp. 341–358. Springer, 2006. http://www.iacr.org/archive/fse2006/40470344/40470344.pdf.
46. What's new in Silverlight 5, 2011. http://www.silverlight.net/learn/overview/what%27s-new-in-silverlight-5.
47. Andrew Moss, Daniel Page, and Nigel P. Smart. Toward acceleration of RSA using 3d graphics hardware. In Steven D. Galbraith, editor, *Cryptography and Coding*, volume 4887 of LNCS, pp. 364–383. Springer, 2007. http://www.cs.bris.ac.uk/Publications/Papers/2000772.pdf.
48. Ruben Niederhagen. Calasm, 2011. http://www.polycephaly.org/projects/calasm/.
49. NVIDIA Corporation. *Tuning CUDA Applications for Fermi, Version 1.0*, 2010. http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_FermiTuningGuide.pdf.
50. NVIDIA Corporation. *NVIDIA CUDA - NVIDIA CUDA C Programming Guide, Version 4.0*, 2011. http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf.
51. NVIDIA Corporation. *OpenCL Programming Guide for the CUDA Architecture*, 2011. http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Programming_Guide.pdf.
52. Dag Arne Osvik, Joppe W. Bos, Deian Stefan, and David Canright. Fast software AES encryption. In Seokhie Hong and Tetsu Iwata, editors,*Fast Software Encryption*, volume 6147 of LNCS, pages 75–93. Springer, 2010.

53. Ádám Rák. AMD-GPU-Asm-Disasm, 2011. https://github.com/rakadam/AMD-GPU-Asm-Disasm/, Accessed Nov. 1, 2011.
54. Elizabeth Seamans and Thomas Alexander. Fast virus signature matching on the GPU. In Hubert Nguyen, editor,*GPU Gems* 3, pp. 771–784. Addison-Wesley, 2007. http://http.developer.nvidia.com/GPUGems3/gpugems3_ch35.html, Accessed Nov. 1, 2011.
55. Robert Szerwinski and Tim Güneysu. Exploiting the power of GPUs for asymmetric cryptography. In Elisabeth Oswald and Pankaj Rohatgi, editors, *Cryptographic Hardware and Embedded Systems -CHES 2008*, volume 5154 of *LNCS*, pages 79–99. Springer, 2008.
56. Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010. http://people.csail.mit.edu/tromer/papers/cache-joc-official.pdf.
57. Wladimir J. van der Laan. Cubin utilities, 2007. https://github.com/laanwj/decuda/wiki, Accessed Nov. 1, 2011.
58. Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In Richard Lippmann, Engin Kirda, and Ari Trachtenberg, editors, *Recent Advances in Intrusion Detection*, volume 5230 of *LNCS*, pp. 116–134. Springer, 2008. http://www.ics.forth.gr/_pdf/brochures/gnort.raid08.pdf.
59. Giorgos Vasiliadis and Sotiris Ioannidis. GrAVity: A massively parallel antivirus engine. In Somesh Jha, Robin Sommer, and Christian Kreibich, editors, Recent Advances *In Intrusion Detection, LNCS*, pp. 79–96. Springer, 2010. http://dcs.ics.forth.gr/Activities/papers/gravity-raid10.pdf.
60. Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. GPU-assisted malware. In Jean-Yves Marion, Noam Rathaus, and Cliff Zou, editors, *Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 2010. dcs.ics.forth.gr/Activities/papers/gpumalware.malware10.pdf.
61. Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. MIDeA: A multi-parallel intrusion detection architecture. In George Danezis and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM/SIGSAC Conference on Computer and Communications Security*, pp. 297–308. ACM Press, 2011. http://dcs.ics.forth.gr/Activities/papers/midea.ccs11.pdf.
62. Takeshi Yamanouchi. AES encryption and decryption on the GPU. In Hubert Nguyen, editor, *GPU Gems 3*, pp. 785–804. Addison-Wesley, 2007. http://http.developer.nvidia.com/GPUGems3/gpugems3_ch35.html, Accessed Nov. 1, 2011.
63. Jason Yang and James Goodman. Symmetric key cryptography on modern graphics hardware. In Kaoru Kurosawa, editor, *Advances in Cryptology - ASIACRYPT 2007*, volume 4833 of *LNCS*, pages 249–264. Springer, 2007.