Draft Version

GPU-Acceleration of Block Ciphers in the OpenSSL Cryptographic Library

Johannes Gilger
Research Group IT-Security
RWTH Aachen University
Aachen, Germany
Gilger@ITSec.RWTH-Aachen.de

Johannes Barnickel
Research Group IT-Security
RWTH Aachen University
Aachen, Germany
Barnickel@ITSec.RWTH-Aachen.de

Prof. Dr. Ulrike Meyer
Research Group IT-Security
RWTH Aachen University
Aachen, Germany
Meyer@ITSec.RWTH-Aachen.de

ABSTRACT

The processing power of *graphic processing units* (GPUs) has been applied for cryptographic algorithms for some time. For AES and DES especially, there is large body of existing academic work and some available code which makes use of the CUDA framework.

We contribute to the field of symmetric-key GPU cryptography by implementing and benchmarking multiple block ciphers on CUDA and OpenCL in the form of an OpenSSL cryptographic engine. We show common techniques to implement and accelerate these block ciphers (AES, DES, Blowfish, Camellia, CAST5, IDEA). Another equally important part of our work presents a guideline on how to perform reproducible benchmarks of these ciphers and similar GPU algorithms.

Keywords

gpu, graphics processing unit, block cipher, cryptography, symmetric key, aes, des, blowfish, idea, cast, camellia, openssl, performance, benchmarking

1. INTRODUCTION

Graphic processing units have originally been designed to handle the generation and modification of graphical data. This includes rendering of 2D or 3D scenes by means of *shaders*, relatively small and limited programs to instruct the GPU's programmable rendering pipeline. As GPUs became increasingly powerful (in terms of floating point operations per second) compared to available CPUs in the last few years, the interest in using these devices for tasks other than the interactive generation of graphical output surged.

Although a small number of people had already experimented with using graphically oriented programming APIs such as DirectX and OpenGL to translate more general computation tasks to the GPU, coining the term *GPGPU* (general purpose programming for GPUs), the process was cumbersome and error-prone.

1.1 CUDA & OpenCL

CUDA (Compute Unified Device Architecture) is Nvidia's approach to GPGPU, with its initial release in 2007 aiming at the G80 chip first seen in the GeForce 8800 GTX GPU. CUDA is a programming framework which executes instructions on the CPU ("host") and the GPU ("device") and passes data between the corresponding memory spaces. Programming for the host is done using standard ANSI C, while programming for the GPU introduces some new keywords to the C language.

The *Open Computing Language* (OpenCL [15]) is a standardized GPGPU framework to enable programs to leverage the computing power of GPU and CPU devices by different vendors. The first OpenCL specification was released in 2008 by the Khronos group. In most aspects, OpenCL behaves very similar to CUDA, making the effort to port programs manageable. The important distinction is that OpenCL is a standard rather than a product, and the actual implementation is the responsibility of the respective CPU and GPU vendors.

Many recent research papers describe the acceleration of symmetric block ciphers as well as other cryptographic algorithms like hash functions, public key encryption schemes, and digital signature schemes using GPGPU frameworks. However, with regard to symmetric block ciphers, prior work focused on AES [26] and DES [8] and the CUDA framework, and only very few of these implementations were released as Open Source software.

1.2 Our contribution

In this paper we present our implementations of the symmetric block ciphers AES [26], DES [8], Blowfish [29], IDEA [17], Camellia [22], and CAST5 [1] using CUDA and OpenCL. We chose these algorithms as they are standard encryption algorithms implemented in the OpenSSL cryptographic library [28]. The OpenSSL implementation includes thoroughly tested CPU implementations of these ciphers against which we benchmarked our own implementation efforts. We show that our GPU implementations can accelerate all block ciphers by a factor of ten (compared to CPU implementations) for practical application scenarios. The fact that we implemented the CUDA and OpenCL ciphers as an OpenSSL engine, based on the engine-cuda project [21], makes these ciphers available to software which already employs OpenSSL with relatively little effort. While the original engine-cuda project included a capable AES implementation for CUDA, we show that our improved AES implementation is almost twice as fast.

In addition, we present guidelines for benchmarking cryptographic and other programs with CUDA and OpenCL. We deduce these guidelines from our own experience with the implementation and various shortcomings of the benchmarking presented in related work (see Section 2).

The rest of the paper is structured as follows: Section 2 summarizes previous work on GPU implementations of symmetric ciphers and their integration in frameworks and applications. The results of our implementation efforts are presented in Section 3. Section 4 summarizes our lessons leading into a guideline for developing and evaluating cryptographic algorithms on GPU platforms.

2. STATE OF THE ART

Cryptographic algorithms have been ported to graphics hardware even before general purpose programming tools existed. Using libraries like OpenGL and DirectX, symmetric key algorithms such as AES have successfully been run and accelerated on GPU devices [6], [10], [31].

With the introduction of general purpose computing on GPUs (GPGPU), the number of publications on implementations of cryptographic algorithms on graphics hardware surged.

2.1 Symmetric key ciphers

For symmetric key algorithms, AES is the block cipher most often investigated. The earliest publication using CUDA to accelerate AES was by Manavski [20]. Manavski closely followed the Rijndael reference implementation, as did almost every team after him. He identified the optimal location for storing the T-tables and provided impressive benchmark performances, measuring raw kernel execution time as well as overall data rate, compared to an OpenGL implementation of AES. In 2008, Harrison et al. released their work with AES-CTR and CUDA GPUs [11]. Aside from similar performance benchmarks, their work also discussed general purpose data structures for scheduling serial and parallel execution of block ciphers on GPUs. Luken et al. implemented and benchmarked the AES and DES [19] on an NVIDIA Tesla C870 GPU, a specifically designed GPGPU card. In some cases, AES was sought to be parallelized below the block-level, and this approach was compared to the more common block-level parallelism in [4], which also proposed a different distribution for T-tables in shared memory in order to avoid memory access conflicts. A textbook implementation of AES for the OpenCL framework was given in 2010 by Gervasi et al. [9]. This implementation did not make use of the T-tables and consequently suffered from unsatisfactory performance. Coincidentally, this publication also employed the OpenSSL framework to integrate their GPU implementation, with the goal of ultimately making it available to a broad audience by having it included in the official OpenSSL development tree. Other work on AES includes [23], which closely examines the different memory locations for the T-tables, correctly points out the low ratio of computation vs. memory access of AES and provides benchmarks which reportedly are able to reach the limit imposed by the memory bandwidth on an Nvidia GeForce 9200M.

DES, which is deprecated for new cryptographic systems, has been implemented on GPUs as well. Yang and Goodman published an AES and DES key breaker realized using DirectX and AMD's Close To Metal framework [32]. Another publication about a DES software breaker, this time using CUDA, was released in 2010 by Agosta et al. [3]. The cost of breaking DES using CUDA-enabled GPUs was compared to the COPACOBANA system [16], which is a special-purpose multi-FPGA DES breaker, and found to perform in the same order of magnitude. Noer et al. have announced that they have improved the results of [3], with a 20-fold performance advantage of searching DES keys on a GPU compared to a CPU [27]. They are furthermore planning to port their efforts to OpenCL.

The CAST5 block cipher was implemented as part of an GPU-based dictionary attack on OpenPGP secret keyrings by Milo et al[24]. In their paper, they present a very detailed and well-written explanation of how they implemented their CAST5 key searcher. The optimization techniques described in their paper apply equally well to similar software and will likely provide noticeable performance improvements.

A CUDA implementation of the Serpent block cipher was presented in [25]. Li et al. investigated MD5-RC4 encryption on GPUs [18].

2.2 Integration and frameworks

Some researchers investigated GPU cryptography in terms of the framework needed to actually integrate it into existing systems. In 2010, Jang et al. introduced their implementation of a reverse SSL-proxy called *SSLShader* [13], which is able to perform the necessary crypto-operations (RSA, AES128-CBC, HMAC-SHA1) for a large number of independent SSL connections by using the GPU starting from a certain threshold, where the latency incurred by the GPU is lower than queueing the operations on the CPU. Recently, Jang et al. built on their previous efforts and released a paper with a detailed account of how they implemented their SSLShader which provides accelerated AES, RSA and HMAC-SHA1 using a combination of GPU and CPU techniques [14].

Harrison and Waldron also investigated how well NVIDIA GPUs can be used to provide a kernel-level service for cryptographic operations [12]. Using the OpenBSD cryptographic framework (OCF) and a userspace daemon they were able to provide GPU-cryptography for kernel- and userspace processes with minimal overhead. The kgpu project [30] also investigates the feasibility of a kernel-level GPU framework using a userspace daemon which to dispatch CUDA operations. To eliminate the latency of kernel invocations, they introduce the concept of a *non-stop kernel* (NSK) which continuously runs on the GPU and can be instructed by passing messages through allocated memory. Their implementation is demonstrated by accelerating eCryptfs with AES using the GPU.

Agosta et al. investigated how GPGPUs can be integrated into the *full-disk encryption* software *TrueCrypt*, using the XTS block mode and Twofish as the block cipher [2].

2.3 Practical GPU-accelerated software

Much of the work using GPU software and cryptographic algorithms has been of a strictly academic nature. However, there do exist some end-user software products which have incorporated GPU components to accelerate their functionality.

Hash algorithms and symmetric ciphers in particular can be found in a number of proprietary consumer software products aimed at breaking password protected files using brute force, or finding hash collisions in general. These implementations therefore do not provide facilities to actually use the GPU for the normal application of the cryptographic algorithm, and the nature of the software forbids opening the source code to competing companies. Another popular use of GPU hardware is the generation of *rainbow-tables*, data-structures for pre-computed hashes that provide a tradeoff between storage space and computation time to find a hash collision.

Open Source software employing cryptography has seen relatively few patches to use GPU devices. The engine-cuda project [21] by Paolo Margara is a notable exception to the large number of proprietary software, as it set out to create an Open Source portable library to be used with the popular OpenSSL crypto library which provides GPU-accelerated versions of the algorithms included in the standard library. This project was chosen as the foundation for implementing the algorithms presented in this paper.

2.4 Benchmarking methodology

To benchmark their implementations of cryptographic algorithms on GPU devices, the involved researchers used a variety of different metrics, which makes it hard to compare the merits of the approaches with one another. While some teams solely focused on the theoretical speed by counting the instructions needed to perform the operations [5], others went a step further and implemented working GPU-versions of these algorithms. For symmetric ciphers, some implemented the key-scheduling on the GPU [9], [14], while others used the CPU for this task [11], [20]. Benchmarking of the implementations differed as well, with some teams only measuring instruction throughput (presumably using the fastest available on-card memory), others doing calculations to and from the GPU main memory [19], and yet other teams measuring the whole computation chain starting from the host to the device and back to the host [19], [20].

While the approach of *microbenchmarking*, only measuring the operation on the GPU itself, is useful when designing and optimizing the algorithm itself, it should not be used to compare the performance to a CPU-implementation, as it does not represent the performance available for encrypting data stored in global memory of the host system.

2.5 Benchmarking problems

Much of the previous work suffered from minor flaws or omissions in the description of the benchmark process used.

Often, the specific CPU implementation used for comparison was not mentioned, and parameters such as number of CPU cores and compilation were omitted as well. In one case, the performance for the CPU was measured using a Java implementation of the block-cipher in question [7].

Some publications did not clearly state which time interval they measured in order to compute performance results. Also the exact timing method (using either system timers or GPU-specific timing facilities) was often not mentioned, and we suspected this to be a source of error as well. Almost every team used a different GPU, making a direct comparison of the results all the more difficult.

The GPU benchmarks including the host-section of computations often did not state whether the data was already stored in (DMA-accessible) allocated host-memory or if it had to be fetched from background memory first, including the costly memory allocation of a large block. The structure of the payload data also makes a significant difference in performance, since it can affect which memory locations need to be accessed during encryption. Only one team provided a description of the payload data used for their benchmarks.

3. IMPLEMENTATION AND BENCHMARKING RESULTS

To implement and evaluate several popular block ciphers, we used the existing engine-cuda [21] codebase as a foundation for our implementation efforts. The original project was implemented as an OpenSSL engine (i.e. a shared library) and already included the AES for CUDA in ECB and CBC mode [26] for all key sizes. We implemented the following popular block ciphers:

- ▷ **AES** [26]: AES-128, AES-192, AES-256, 128 bits blocksize
- ▷ **DES** [8]: 64 bits keysize, 64 bits blocksize
- ▷ **Blowfish** [29]: 128-448 bits keysize, 64 bits blocksize
- ▷ **IDEA** [17]: 128 bits keysize, 64 bits blocksize
- ▷ **Camellia** [22]: 128, 192 and 256 bits keysize, 128 bits blocksize
- ▷ **CAST5** [1]: 128 bits keysize, 64 bits blocksize

All of these algorithms were implemented with the electronic codebook (ECB) block mode for encryption and decryption, as well as the cipher block chaining (CBC) mode for decryption. We did not implement CBC encryption on the GPU, as the CBC block mode cannot be parallelized during encryption and would not benefit from the GPU platform. Other block modes which can be computed in parallel (e.g. Counter mode (CTR)) could use our implementations with only minimal modifications and a small constant overhead. We limited ourselves to ECB and CBC as these are the block modes available in the current OpenSSL packages.

Since the original engine-cuda project only supported AES with CUDA, we had to significantly extend it to support OpenCL and other block ciphers as well.

To the best of our knowledge, the Blowfish, IDEA, Camellia and CAST5 implementations represent the first published CUDA and OpenCL versions of these algorithms. Although there have been previous efforts to implement AES and DES using CUDA, none of these were readily available to be benchmarked against our implementation, as none of them were Open Source.

3.1 Optimizations

We started implementing each block cipher by porting the implementation included in OpenSSL to CUDA and OpenCL. After we ensured basic functionality, we began to optimize the block cipher for these GPU platforms in terms of performance. Most importantly, we looked at register usage per GPU thread, uncoalesced memory access and diverging paths, trying to optimize each aspect using a variety of techniques, some of which could certainly be classified as "compiler workarounds":

1. Removal of unnecessary flexibility. For example, we implemented separate GPU kernels for ECB, CBC and the different key lengths rather than passing these as parameters to a single kernel. In general, we passed as few parameters as possible, e.g. by storing data such as T-Tables in one big chunk which can be referenced using a single pointer. This resulted in reduced register use, enabling us to better utilize the SMs of the GPU.
2. Removing uncoalesced memory access which would otherwise require multiple memory transactions. Figure 1 shows how we were able to get rid of all the misaligned memory accesses for the Blowfish cipher with a simple change. Sometimes, independent 32-bit values were fetched from memory into a 64-bit variable and split up manually, so that the compiler would generate a single

With BF_KEY structure

```
__shared__ BF_KEY bs;
__device__ BF_KEY bsg;

if(TX < 18)
  bs.P[TX] = bsg.P[TX];

bs.S[TX] = bsg.S[TX];
bs.S[TX+256] = bsg.S[TX+256];
bs.S[TX+512] = bsg.S[TX+512];
bs.S[TX+768] = bsg.S[TX+768];
```

Disregarding BF_KEY structure

```
__shared__ uint32_t bs[1042];
__device__ uint32_t bsg[1042];

bs[TX] = bsg[TX];
bs[TX+256] = bsg[TX+256];
bs[TX+512] = bsg[TX+512];
bs[TX+768] = bsg[TX+768];

if(TX < 18)
  bs[TX+1024] = bsg[TX+1024];
```

Figure 1: Misaligned and aligned memory copy of the Blowfish key schedule

```
#ifndef TX
  #if (__CUDA_ARCH__ < 200)
    #define TX (__umul24(blockIdx.x,blockDim.x) + threadIdx.x)
  #else
    #define TX (blockIdx.x * blockDim.x + threadIdx.x)
  #endif
#endif
```

Figure 2: Preprocessor macro for calculating the thread-ID TX

coalesced ld.global.u64 instruction instead of two strided ld.global.u32 instructions.

3. Removal of bank conflicts when using shared memory. This was the case for the AES version included with the original engine-cuda codebase. Figure 3 shows the time spent due to warp serialization for the original and our improved AES implementation.
4. Reordering statements to reduce register use. In some cases, statements could be reordered without influencing the computation. A variant of this approach was to re-use variables for different steps of the computation.
5. Modification of the key schedule to avoid endian conversion of the payload data, as many algorithms were developed with Big-Endian CPU architectures in mind.
6. Use of native integer functions and synchronization. We used the preprocessor heavily to employ functions only available on specific compute capabilities, such as native 24-bit or 32-bit multiplication. A simple example is given in Figure 2.
7. Unrolling loops to reduce register use. Unrolling is a technique often performed by CPU compilers, which we had to perform manually for CUDA, for example for the encryption rounds.

Table 2 shows how much resources are used by each kernel of our implementations of the block-ciphers as reported by nvcc. For better comparability, the table lists the *decryption* kernels for ECB and CBC. Register use relates to each thread, while the shared memory is counted block-wise and the constant memory is reported for the whole kernel, following the visibility of each class of memory.

Depending on the compute capability the kernels are compiled for, nvcc will use more or fewer registers for each kernel, up to twice as many registers for our newer (CC 1.3) cards compared to the older card (CC 1.1).

Optimizations were also performed in the backend, which includes the memory transfer functions, the setup of the GPU context and the memory allocation. It is important to mention that we did not implement any stage of key-scheduling on the GPU but left that task to be performed by the CPU. We ensured that our GPU implementations worked correctly by comparing their output to the stock OpenSSL CPU output for a variety of different payload data and keys.

3.2 Benchmark system

We thoroughly benchmarked the resulting implementation of each algorithm on two systems with different CPUs and GPUs, using an older GeForce 8600 GT as well as a more recent GeForce GTX 295. The results presented in this paper are based on the benchmark system described in Table 1.

CPU	Intel Core i7 960 3.20GHz
GPU	GeForce GTX 295 (CC 1.3)
RAM	12 GB DDR3 RDIMM
HDD	Intel X-25 M II SSD (160GB)
Kernel	Linux 3.0.0-17-generic x86_64
CUDA	CUDA toolkit 4.1
Driver	NVIDIA UNIX x86_64 285.05.33
CC	GCC 4.4.6

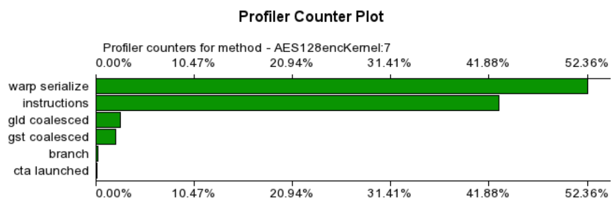
Table 1: Benchmark system

The results for our second system were omitted due to space constraints and will be provided as part of an extended version of this paper. The GeForce GTX 295 in the benchmark system contains two discrete GPUs on two different PCBs (printed circuit boards), each of which is presented as an independent GPU to the system. We only used one of these GPUs for our benchmarks. CPU reference speeds were obtained using OpenSSL v1.0.1 and using one of the four CPU cores of the benchmark system. We measured the theoretical kernel execution speed, which does not take into account the time needed to transfer the data to and from the GPU, as well as measuring the practical encryption speed, which includes transfers from and back to host memory. In the following, we discuss the results of both measurement techniques.

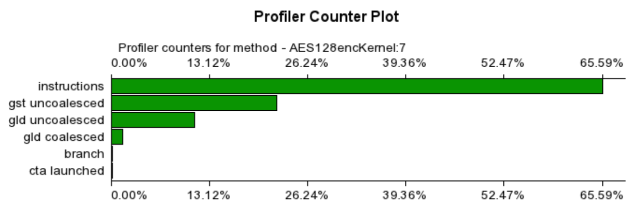
3.3 Impact of payload on kernel benchmarks

The *microbenchmarks* of our cipher kernels are shown in Table 3[1] and Table 4. These tables measure the theoretical ECB

[1]We achieved slow results for Blowfish OpenCL because we



(a) Original AES-128 ECB implementation in engine-cuda



(b) Our improved AES-128 ECB implementation for engine-cuda

Figure 3: The CUDA Visual Profiler showing the original engine-cuda (Figure 3(a)) and our improved version of AES-128 ECB (Figure 3(b))

Cipher	Mode	Registers	Shared Memory	Constant Memory
AES-{128,192,256}	ECB	13 / 14	4376 bytes	264 bytes
	CBC	15	4384 bytes	264 bytes
Blowfish	ECB	10	4176 bytes	8 bytes
	CBC	12	4184 bytes	8 bytes
DES	ECB	9	2056 bytes	136 bytes
	CBC	10	2064 bytes	136 bytes
CAST5	ECB	10	4104 bytes	144 bytes
	CBC	12	4112 bytes	144 bytes
Camellia-128	ECB	14	4104 bytes	296 bytes
	CBC	14	4112 bytes	296 bytes
IDEA	ECB	10	224 bytes	216 bytes
	CBC	12	232 bytes	224 bytes

Table 2: Memory consumption for the implemented decryption kernels in engine-cuda (compiled for CUDA Compute Capability 1.3)

Cipher	Engine	Random bytes		Zero bytes		Δ
		Kernel ms	MB/s	Kernel ms	MB/s	
AES-128	CUDA	2.16	29613	1.36	47021	1.59
	OpenCL	2.52	25408	1.72	37248	1.47
AES-192	CUDA	2.57	24922	1.58	40461	1.62
	OpenCL	2.94	21760	1.98	32384	1.49
AES-256	CUDA	2.97	21572	1.87	34234	1.59
	OpenCL	3.35	19072	2.23	28736	1.51
Blowfish	CUDA	2.12	30249	1.49	43077	1.42
	OpenCL	30.07	2112	5.34	11968	5.67
Camellia-128	CUDA	2.43	26302	1.66	38647	1.47
	OpenCL	2.43	26304	1.68	38016	1.45
CAST5	CUDA	2.19	29203	1.54	41519	1.42
	OpenCL	2.45	26112	1.79	35648	1.37
DES	CUDA	4.14	15460	2.75	23279	1.51
	OpenCL	4.12	15488	2.74	23296	1.50
IDEA	CUDA	1.75	36512	1.71	37388	1.02
	OpenCL	1.60	40064	1.59	40128	1.00

Table 3: ECB encryption kernel performance for different input data (8192 KB, GeForce GTX 295)

performance of the cipher kernels, which only includes the execution time of the CUDA (resp. OpenCL) kernel itself. The resulting time is given in milliseconds, from which the theoretical performance (given in megabytes per second) can be derived. To demonstrate the importance of the payload data, we measured performance with pseudo-random data (obtained from `/dev/urandom`) and zero-bytes and used Δ to denote the speed ratio of zero- over random bytes.

The payload data is used as to determine the memory access for certain operations (such as an index in a lookup table). When the payload consists only of uniform bytes, the same memory area will be queried for every byte of payload data, which results in excellent performance for constant memory (which is cached after the first memory access). However, when the payload data exhibits a certain amount of entropy, different memory areas will be queried in each request. In this case, shared memory shows a clear advantage over constant memory.

No previous publication measured and compared both payload types and some of the previous work did not even mention at all which payload they used.

3.4 Improved AES implementation

Table 4 shows our improved AES-128 ECB implementation benchmarked against the implementation which was already included with engine-cuda. Originally, AES was implemented using a “fine-grained” approach, which employs four threads for each cipher block of 128 bits. Our “coarse-grained” approach uses just one GPU thread for each cipher block.

Method	T-Table	Random bytes		Zero bytes		Δ
		Kernel	MB/s	Kernel	MB/s	
Fine	Const.	15.86	4036	3.54	18097	4.48
	Shared	3.44	18584	3.53	18138	0.97
Coarse	Const.	14.35	4460	1.45	44123	9.89
	Shared	2.16	29613	1.36	47021	1.58

Table 4: Fine- and coarse-grained AES-128 ECB (8192 KB, CUDA)

As expected in [4], our coarse-grained approach delivered a noticeable performance increase. We also measured the speed of storing the AES T-Tables, which represent the AES functionality in the form of static lookup tables, in constant and in shared GPU memory. The results are nearly identical when zero bytes are used as payload data, while random data (which is much more practically relevant) shows the clear superiority of using shared memory. This obvious discrepancy prompted our investigation into the benchmark methodology of other teams, leading to the advice given in Section 4.

3.5 OpenSSL benchmarks

To evaluate the practical benefit of using GPU-accelerated block ciphers, we also measured the performance using the OpenSSL `speed` command, which schedules runs of increasingly large blocks of zero-byte data and measures the achieved throughput. These results were averaged over five consecutive runs. This kind of benchmarks includes the time needed to transfer the payload from the host to the device memory, the cipher kernel execution, and the time for transferring the data back to the host, and as such can be directly compared to the execution time on the CPU. Figure 5 shows the results of GPU and CPU benchmarks for CUDA and the ECB mode of each block cipher. It is obvious that because of the latency incurred by invoking a GPU kernel and the necessary memory transfer, GPU-accelerated block ciphers can only outperform the CPU if the payload data is large enough, larger than 16KB in our case.

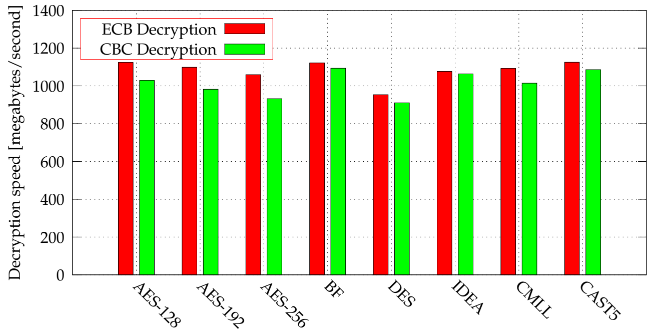


Figure 4: ECB and CBC decryption on 8MB blocks (CUDA, GeForce GTX 295, OpenSSL `speed`)

Although we also implemented the CBC-decryption, we chose to use the ECB mode as the primary tool for benchmarking as it can be more easily compared to existing and new academic work with the same algorithms. The CBC mode in our implementation only adds a small performance penalty in the form of an additional memory access and does not invalidate the performance results compared to the CPU, as shown in Figure 4.

We intentionally refrain from making statements concerning the performance relative to existing work for the reasons stated in Section 4. We want to emphasize that the 8-10 fold improvement in speed shown in Figure 5 was actually measured on our test system, but that the superiority of the GPU heavily depends on the GPU generation, the CPU and the application area.

Other features of our extended GPU crypto library include timing facilities for CUDA and OpenCL, benchmarking tools for trying different kinds of payload data, keys and payload size and different levels of debug output.

4. PERFORMING REPRODUCIBLE BENCHMARKS

During the process of implementing the cryptographic algorithms presented in this work, we gathered a number of benchmarking details which are important for the production of reproducible comparable benchmark results. In the following, we provide an overview on the details as guideline for future implementation efforts, not only in the are of cryptography.

Kernel and framework.

Depending on the framework, host operations and the time for memory transfers dwarfs the actual kernel execution time. Any research which does not only focus on the raw kernel should therefore include benchmarks of both the kernels and the complete chain of operation.

Structure of payload.

When benchmarking the speed of a block cipher on a GPU, it is important to mind the kind of data being used. While zero bytes are the obvious choice for ease of generation and reproducibility, they can result in artificially good result with un-

had to work around an apparent compiler bug which forced us to use slower memory.

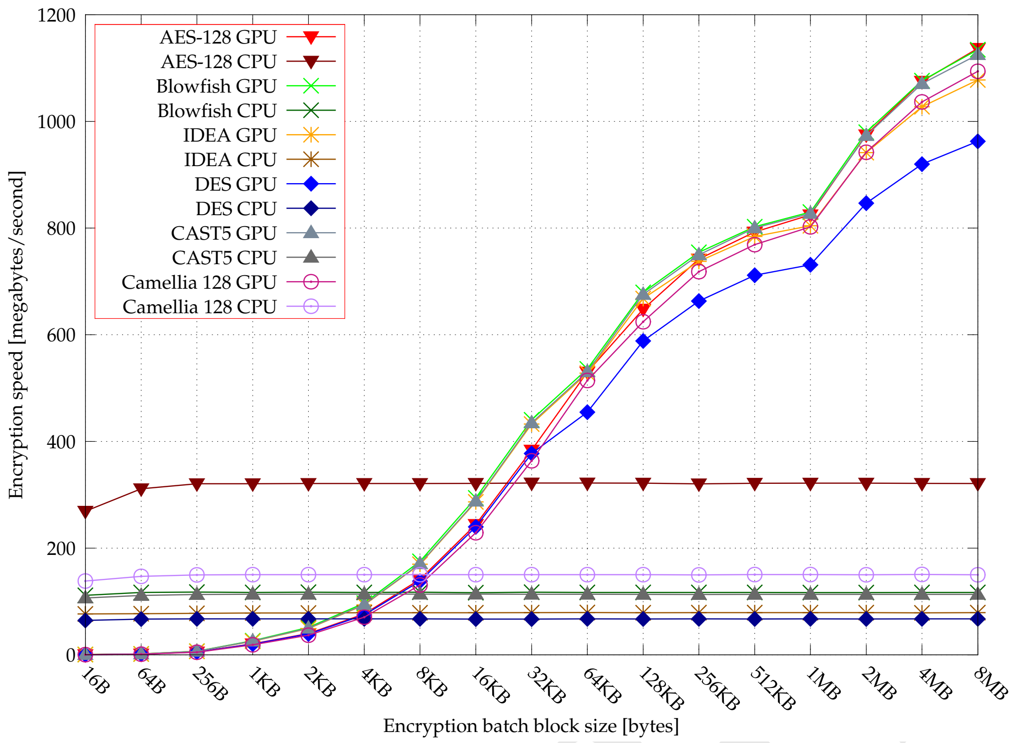


Figure 5: ECB encryption with CUDA on one PCB of a GeForce GTX 295 (OpenSSL `speed`)

known average-case behaviour. Block ciphers should be developed using zero bytes as well as random data, and publications should clearly indicate the source of payload data. Especially the behaviour with random data should influence the choice of storing read-only lookup tables.

Ensuring correctness.

If block ciphers are implemented on GPU platforms, the correctness of the cipher should be ensured at every step of the development process. GPUs are hard to debug, so a combination of different keys, key sizes, payload structure and payload length should be tested at all times. Especially payload sizes which are not multiples of the thread block configurations are often subject to failure. For platforms like OpenCL, tests for correctness should be run repeatedly, since problems do occur non-deterministically.

Measuring time.

CUDA and OpenCL include their own timing functions for the simple reason that kernel calls are nonblocking and may return before the kernel is finished.

Scheduling priorities.

CUDA has an option to use spinning to wait on the return of the GPU kernel, which for many repeated small invocations can make a measurable difference. Understanding and verifying the method to poll for the GPU kernel is important.

Eliminating side effects.

When performing benchmarks, the GPU should be switched to compute exclusive mode and any running X server should be stopped. Benchmarking cryptography means allocating large blocks of page locked memory on the host, so the system should have enough free RAM. The CPU should not be busy with other tasks except executing the host-thread. It is important to use the GPU driver supplied by the vendor alongside the GPGPU framework. Other drivers might work correctly as well but perform much worse when used with CUDA or OpenCL.

Using reference implementations.

When directly comparing the GPU to existing CPU implementations, an established and publicly accessible CPU implementation of the block cipher should be used. It is also important to verify whether the reference implementation uses more than one CPU core, whether it uses native instructions tailored to the CPU platform and if special instructions like AES-NI are employed.

5. APPLICATION AREAS FOR GPU CRYPTOGRAPHY

Previous work has inarguably shown that GPUs can offer substantial performance benefits even in the domain of symmetric key cryptography. In contrast to programs running on a CPU however, GPU programming requires the careful consideration of the application area before the actual implementation. We want to highlight some possible scenarios in which we think GPU assisted symmetric key cryptography might be used in the future and point out the peculiarities of each.

▷ **Key breaker**: Using brute-force or dictionary attacks, a key for data encrypted with a symmetric block cipher is to be found. This requires the use of independent threads which work with a distinct key each and can signal success through global memory on the GPU. The amount of data that needs to be transferred back and forth is small. Block ciphers which make use of a large key-dependent S-Box (such as Blowfish or Twofish) will not benefit as much as other ciphers, since the large schedule should be stored in the limited amount of fast memory for the breaker to work efficiently. Commercial software for breaking keys has been available since the introduction of CUDA and OpenCL, while Open Source software like the popular *John the Ripper* are in the process of incorporating existing patches to enable GPU bruteforcing into their main development tree.

- ▷ **SSL accelerator**: Especially in the face of slow and only partial HTTPS adoption by many popular websites, methods to cheaply accelerate SSL operations are one of the most interesting topics for the future of GPU cryptography. A number of options to use GPUs in this settings can be identified:
  - ▷ **Builtin**: The *server* software itself is linked against CUDA or OpenCL and directly calls device code to handle cryptographic operations.
  - ▷ **Library**: One step above a built-in solution, a widely used cryptographic library (like OpenSSL or libgcrypt) is a prime target for GPU acceleration since all the software using it would benefit instantly.
  - ▷ **Standalone**: A standalone daemon linked against CUDA or OpenCL which can interact with the back-end software and the user, much like [14], is a third option. This approach is probably the most flexible, since it does not require any changes to the server software and can be scaled independently of the server.
- ▷ **Disk encryption**: Using a GPU to accelerate and offload the cryptographic operations when encrypting the content of a harddisk might be another worthy area of research [2]. For GPUs to accelerate disk-encryption, the write speed of the disk has to surpass the cryptographic performance of a CPU, which is not the case in current consumer systems, even less so when taking into account nascent techniques like the AES-NI of current Intel CPUs which is already being used inside the Linux kernel and thus by subsystems like dm-crypt. The inclusion of AES circuitry in CPUs and increasingly often directly within the hard drive means that there are a multitude of options to securely store data on these devices.
- ▷ **Kernel-level service**: For operating systems such as the Linux kernel, offering GPU accelerated cryptography at the kernel level would be an option. Some software relies on the kernel cryptography services, such as the popular disk encryption solution dm-crypt. Currently, using a GPU from kernel space means going through the CUDA libraries in user space and back to the kernel [30], a process which includes two costly context switches. The fact that drivers for NVIDIA GPUs are still only available as binary blobs and the necessary detour through userspace makes this topic nothing more than an interesting area of research for the time being. Other than providing cryptographic algorithms, a kernel-level GPU service might also be used to accelerate other compute intensive tasks, either as a service for userland software of for tasks performed by the kernel.

6. CONCLUSION

In this work, we showed the potential and limitations of GPU-accelerated block ciphers as implemented within the OpenSSL cryptographic library. We were able to clearly accelerate symmetric block ciphers using GPUs compared to traditional CPU implementations, in terms of theoretical as well as practical speed. As the first contribution we used the exact same setup to implement these ciphers using CUDA and OpenCL. Perhaps not surprisingly, we were able to show that both GPU frameworks are equally capable of delivering practical performance for all of the implemented block ciphers.

In addition to the implementation and benchmarking we discuss common problems encountered when trying to compare the benchmarking results of different implementations of symmetric ciphers. From this discussion, we compile and present a list of recommendations for future implementations.

We hope that our paper and the future release of our source-code to the engine-cuda project can assist in presenting correct benchmark results in this and other GPU-related research areas, with the ultimate goal of raising the scientific standard of similar work and to further the cause of open standards within the GPGPU community.

7. ACKNOWLEDGEMENTS

This work was supported by the UMIC Research Centre, RWTH Aachen University.

8. REFERENCES

[1] C. Adams. The CAST-128 Encryption Algorithm. RFC 2144 (Informational), May 1997.

[2] G. Agosta, A. Barenghi, F. D. Santis, A. D. Biagio, and G. Pelosi. Fast Disk Encryption through GPGPU Acceleration. In *PDCAT*, pages 102–109. IEEE Computer Society, 2009.

[3] G. Agosta, A. Barenghi, F. D. Santis, and G. Pelosi. Record Setting Software Implementation of DES Using CUDA. In *ITNG*, pages 748–755. IEEE Computer Society, 2010.

[4] A. D. Biagio, A. Barenghi, G. Agosta, and G. Pelosi. Design of a parallel AES for graphics hardware using the CUDA framework. In *IPDPS*, pages 1–8. IEEE, 2009.

[5] J. W. Bos and D. Stefan. Performance Analysis of the SHA-3 Candidates on Exotic Multi-core Architectures. In *CHES*, volume 6225 of *Lecture Notes in Computer Science*, pages 279–293. Springer, 2010.

[6] D. L. Cook, J. Ioannidis, A. D. Keromytis, and J. Luck. CryptoGraphics: Secret Key Cryptography Using Graphics Cards. In *Topics in Cryptology – CT-RSA 2005*, volume 3376 of *Lecture Notes in Computer Science*, pages 334–350. Springer Berlin / Heidelberg, 2005.

[7] T. R. Daniel and S. Mircea. AES on GPU using CUDA. In *2010 European Conference for the Applied Mathematics & Informatics*. World Scientific and Engineering Academy and Society Press, 2010.

[8] N. I. for Standards and Technology. Data Encryption Standard (DES). *NIST FIPS PUB 46-3*, 1999.

[9] O. Gervasi, D. Russo, and F. Vella. The AES Implantation Based on OpenCL for Multi/many Core Architecture. *Computational Science and its Applications, International Conference*, 0:129–134, 2010.

[10] O. Harrison and J. Waldron. AES Encryption Implementation and Analysis on Commodity Graphics Processing Units. In *CHES*, volume 4727 of *Lecture Notes in Computer Science*, pages 209–226. Springer, 2007.

[11] O. Harrison and J. Waldron. Practical Symmetric Key Cryptography on Modern Graphics Hardware. In *USENIX Security Symposium*, pages 195–210. USENIX Association, 2008.

[12] O. Harrison and J. Waldron. GPU accelerated cryptography as an OS service. *Transactions on Computational Science*, 11:104–130, 2010.

[13] K. Jang, S. Han, S. Han, S. Moon, and K. Park. Accelerating SSL with GPUs. In *SIGCOMM*, pages 437–438. ACM, 2010.

[14] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL acceleration with commodity processors. *Proceedings of NSDI'11*, 2011.

[15] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.1*, 30 September 2010.

[16] S. S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler. Breaking Ciphers with COPACOBANA - A Cost-Optimized Parallel Code Breaker. In L. Goubin and M. Matsui, editors, *CHES*, volume 4249 of *Lecture Notes in Computer Science*, pages 101–118. Springer, 2006.

[17] X. Lai and J. L. Massey. A Proposal for a New Block Encryption Standard. In *Proceedings of EUROCRYPT '90 — Advances in cryptology*, volume 473 of *Lecture Notes in Computer Science*, pages 389–404, Århus, Denmark, May 1991. Springer-Verlag.

[18] C. Li, H. Wu, S. Chen, X. Li, and D. Guo. Efficient implementation for MD5-RC4 encryption using GPU with CUDA. In *Proceedings of the 3rd international conference on Anti-Counterfeiting, security, and identification in communication*, pages 167–170. IEEE Press, 2009.

[19] B. P. Luken, M. Ouyang, and A. H. Desoky. AES and DES encryption with GPU. In *ISCA PDCCS*, pages 67–70. ISCA, 2009.

[20] S. A. Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *IEEE International Conference on Signal Processing and Communications, 2007. ICSPC 2007.*, pages 65–68. IEEE, 2008.

[21] P. Margara. engine-cuda. `http://code.google.com/p/engine-cuda/`, 2011. [Online; accessed 25-April-2012].

[22] M. Matsui, J. Nakajima, and S. Moriai. A Description of the Camellia Encryption Algorithm. RFC 3713 (Informational), Apr. 2004.

[23] C. Mei, H. Jiang, and J. Jenness. CUDA-based AES parallelization with fine-tuned GPU memory utilization. In *IPDPS Workshops'10*, pages 1–7, 2010.

[24] F. Milo, M. Bernaschi, and M. Bisson. A fast, GPU based, dictionary attack to OpenPGP secret keyrings. *Journal of Systems and Software*, 84(12):2088–2096, 2011.

[25] A. M. Nazlee, F. A. Hussin, and N. B. Z. Ali. Serpent encryption algorithm implementation on Compute Unified Device Architecture (CUDA). In *IEEE Student Conference on Research and Development (SCOReD), 2009*, pages 164–167. IEEE, 2010.

[26] NIST. *Advanced Encryption Standard (AES)*. National Institute of Standards and Technology, 2001.

[27] D. Noer, A. Engsig-Karup, and E. Zenner. Improved Software Implementation of DES Using CUDA and OpenCL.

[28] OpenSSL. OpenSSL: The Open Source toolkit for SSL/TLS. `http://www.openssl.org/`, 2011. [Online; accessed 25-April-2012].

[29] B. Schneier. Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish). In *Fast Software Encryption, Cambridge Security Workshop Proceedings*, volume 809 of *Lecture Notes in Computer Science*, pages 191–204, Cambridge, UK, December 1993. Springer-Verlag.

[30] W. Sun. kgpu: Augmenting Linux with the GPU. `http://code.google.com/p/kgpu/`, 2011. [Online; accessed 25-April-2012].

[31] T. Yamanouchi. AES encryption and decryption on the GPU. In H. Nguyen, editor, *GPU Gems 3*, chapter 36. Addison Wesley Professional, Aug. 2007.

[32] J. Yang and J. Goodman. Symmetric Key Cryptography on Modern Graphics Hardware. In *ASIACRYPT*, volume 4833 of *Lecture Notes in Computer Science*, pages 249–264. Springer, 2007.