OpenSSL acceleration using Graphics Processing Units

Pedro Saraiva

Instituto Superior Técnico

Abstract. Cryptography: The study of techniques focused on security. Typically, an implementation of cryptography is computationally heavy, leading to performance issues on general purpose systems. Adding the possibility of offloading cryptographic operations to a *Graphics Processing Unit* (GPU) onto a widespread, open-source cryptographic library such as OpenSSL would be extremely useful in lightening the CPU load for application logic. GPUs, while originally designed to accelerate graphics processing, have been recently gained usage for unrelated, general purpose computing, due to their massive parallel computing power. As such, two main frameworks designed to take advantage of a GPU for general purpose computing have been developed in the last few years: NVIDIA's proprietary *CUDA* and the Khronos Group's open standard *OpenCL*. In this paper we present high-performance acceleration of the OpenSSL library using both OpenCL and CUDA, specifically for the RSA and AES algorithms. Our evaluation shows that AES decryption can be over forty times faster than the standard CPU implementation, and that RSA keys can be generated over ten times faster than on a CPU. We also study the possibilities of CBC encryption and RSA ciphering, and and conclude why those algorithms are unfeasible to run on a GPU from within OpenSSL.

Keywords: OpenSSL, GPU, OpenCL, CUDA, AES, RSA.

1 Introduction

Cryptography is the study of mathematical techniques focused on information security, including confidentiality, data integrity, and authentication. An implementation of cryptography is typically comprised of computationally intensive algorithms which are used by applications when encrypting, decrypting, and hashing data. Some implementations also include authentication and verification techniques.
One well-known application of cryptography is the *Secure Sockets Layer* (SSL) protocol. Originally developed by Netscape, SSL is a set of rules that govern authentication and encrypted communication between clients and servers. As the growth of secure website deployment rose rapidly, the SSL protocol became the de facto standard for secure electronic commerce. The SSL protocol is built into all popular web browsers. However, due to the computationally intensive nature of SSL technology, many of these sites struggle as demand rises, as large volumes of SSL traffic can impact the performance of even the most powerful general purpose web server systems. Therein, this thesis proposes adding functionality to take advantage of a GPU for cryptography onto an existing, widespread cryptographic framework (OpenSSL), leaving the CPU mostly free for other tasks without requiring the use of specialised, expensive cryptographic accelerator cards.
The goal of this work is to efficiently offload cryptographic operations onto a *Graphics Processing Unit* (GPU). The objective is to add functionality to take advantage of a GPU for cryptography onto an existing, widespread cryptographic framework (OpenSSL), leaving the CPU mostly free for other tasks without requiring the use of specialised, expensive cryptographic accelerator cards. Our focus is on algorithms with a good chance to perform well on a GPU, in order to attempt a significant improvement in performance and lighten the load on the CPU.

2 Background

2.1 OpenSSL

OpenSSL is a robust, commercial-grade, full-featured and open source toolkit implementing the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols, as well as a full-strength, general purpose cryptography library. It is divided into *libssl*, a library implementing the SSL and TLS protocols, and *libcrypto*, a general purpose cryptography library, implementing a wide range of cryptographic algorithms used in various Internet standards, including AES (Advanced Encryption Standard, DES (Data Encryption Standard), RSA (a public-key cryptography algorithm) and many others.

The core library is written in the C programming language, and implements the basic cryptographic functions as well as providing various utility functions. There are also wrappers allowing the use of OpenSSL in other languages. In addition to

implementing various cryptographic algorithms, OpenSSL gives users the possibility to use specialised accelerator hardware. This is done by using *engines* which communicate with said hardware to perform encryption and decryption functions through them.

2.2 GPU

GPU computing has gained momentum during the past years due to the massive parallel processing power that a single GPU contains when compared to a CPU. A high-end graphics card has roughly ten times the single precision floating point processing capability of a high end CPU at roughly the same price. A CPU's processing power increases according to Moore's law, but the increase of GPU processing power is outpacing it.

Another advantage GPUs have over CPUs is their extremely low energy consumption compared to their processing power. For example, an AMD Phenom II X4 CPU operating at 2.8GHz has a ratio of 0.9 gigaflops per watt, whereas a mid-class ATI Radeon 5670 GPU has a ratio of 9.4 gigaflops per watt.

2.3 Programming on the GPU

2.3.1 OpenCL

OpenCL is an open, royalty-free standard for general purpose parallel programming across CPUs, GPUs, and other types of processors, allowing software deelopers portable and efficient access to the power of these heterogeneous processing platforms. It supports a wide range of applications, ranging from embedded and consumer software to *High Performance Computing* (HPC) solutions, due to a low-level portable abstraction. By creating an efficient programming interface, OpenCL forms the foundation layer for a parallel computing system of platform-independent tools. It is maintained by the non-profit technology consortium *Khronos Group*, and has been adopted by Intel, AMD, NVIDIA, and ARM Holdings.

OpenCL consists of an API for coordinating parallel computation across heterogeneous processors, and a cross-platform programming language. It supports data and task-based parallel programming models, and uses a subset of ISO C99 with extensions designed for parallel programming, but ommitting the use of function pointers, recursion, bit fields, variable-length arrays, and standard C99 header files.

2.3.2 CUDA

CUDA is a proprietary hardware and software architecture designed by NVIDIA for issuing and managing computations on the GPU as a data-parallel computing device without the need to map them to a graphics API. The CUDA software stack is composed of several layers: a hardware driver, an *Application Programming Interface* (API) with its runtime, and two high-level mathematical libraries (CUFFT and CUBLAS). The hardware was designed to support lightweight driver and runtime layers, resulting in high performance. It is programmed with 'C for CUDA' (C with NVIDIA extensions and restrictions, as well as some C++ extensions), compiled through a PathScale Open64 C compiler, to code algorithms for execution on a GPU.

3 Implementation

3.1 OpenSSL

The GPU engine implements three different operations: AES, RSA Key Generation, and RSA Cipher. A *bind_helper* function lets OpenSSL know what algorithms are supported by the engine.
First, the ID name and description of the engine (in this case, "gpu" and "GPU-accelerated engine") are set. Following that, the engine must inform OpenSSL of what algorithms it implements, with pointers to the functions that implement said algorithms. Finally, the engine must also inform the OpenSSL library of what cipher modes it supports. An engine object can be either built into OpenSSL or called from within it as a dynamic library --- we opted for the latter, as it means users would be able to use the engine with a standard, unmodified system installation of OpenSSL. For an application to use the engine, it only needs to load the engine library and define that it should be used by default for a specific type of operation.

3.2 AES

The AES cipher function receives an OpenSSL context, a pointer to the input and output buffers, and the number of bytes to proceed. Thus, the function must initialise the GPU, allocate host device memory for the input data, key and IV, and call a modified library of SSLShader's libgpucrypto. After the operation is completed, the last block of ciphertext is stored, as the function may be called again to continue the operation, should the buffer size be smaller than the complete message.
When libgpucrypto is called, it must allocate memory on the GPU, transfer all the input data into it, call the GPU kernel, and wait for the result. The GPU kernel being called behaves differently depending on whether it's a CBC Encryption operation, a CBC Decryption operation, or an ECB operation.

3.2.1 CBC Encryption

During a CBC encryption operation, one thread on the GPU is called that goes through every block individually and encrypts it. The previous block is then used as

an IV for the next block. This is all done serially (no parallelisation is possible with CBC Encryption). Control is given back to the CPU afterwards.

3.2.2 CBC Decryption (or ECB operation)

During a CBC decryption operation, one thread on the GPU is called for every individual block, which are then processed in parallel. Once all operations are completed, control is given back to the CPU.
Once the GPU threads are complete, the output data is transferred from the GPU memory to the host. This is then returned to the OpenSSL engine, which returns the data.

3.3 RSA Key Generation

3.3.1 CPU side

The RSA Key Generation code in *eng_gpu.so* is similar to the normal, non-GPU based code in OpenSSL, with only two major differences: At the start of the process, the *gpu_genprimes* function is called which calls the GPU to generate a large number of large numbers (80 by default). Afterwards, in any point of the code where the standard OpenSSL *BN_generate_prime_ex* would be called, an alternative function is called instead, which merely picks the next generated prime from the list of previously-generated primes and uses it, deleting it afterwards to ensure it won't be reused again.

3.3.2 GPU side

Initially, when *gpu_genprimes* is called, a GPU random number generator is initialised with the current time in milliseconds as the seed (Random123 in OpenCL, curand in CUDA). Afterwards, device memory is allocated for the defined number of BIGNUMs to generate, followed by a call to the GPU kernel, *generatePrimes_kernel*. This kernel runs in parallel for a number of threads identical to the desired number of BIGNUMs to generate. After all threads are done executing the kernel, the output data is copied from the GPU to the host.

3.4 RSA Cipher

The standard Multi-Precision algorithm is the most convenient way to represent large integers in a computer. A *k*-bit integer *A* is broken into *s=k/64* words.

In Montgomery multiplication, the multiplication of two *s*-word integers is performed three times. A serial multiplication algorithm has a complexity of $O(S^2)$. This implementation is instead an $O(s)$ parallel algorithm with linear scalability, running in *s* threads working in two phases. The first phase accumulates *s* partial products in *2s* steps (one step for each high bit and one for one low bit), with carries being accumulated in a separate array. Each step translates into a small number of GPU instructions without involving any cascading carry propagation. The second phase repeatedly adds the carries to the intermediate result and renews the carries, and stops when all carries become 0. The number of iterations is *s-1* in the worst case, but it usually only takes one or two iterations, since small carries rarely produce additional ones.

4 Performance Evaluation

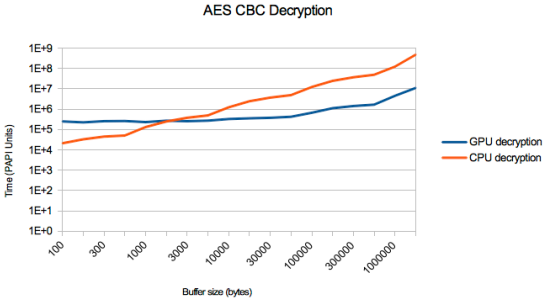
All tests in this subsection were performed on an Intel Core i7 950 CPU clocked at 3.07GHz, with an NVIDIA GeForce GTX 580 GPU. AES tests were performed with PAPI (Performance Application Programming Interface), whereas RSA tests were performed with the UNIX time tool by taking ten results and averaging the result. For results with a heavy CPU load, the *stress* tool was executed running 300 threads, 100 looping on *sqrt*, 100 on *malloc/free*, and 100 on *sync*.

4.1 AES

4.1.1 CBC Decryption

Given how the implementation of AES CBC Decryption is parallelised at the block level, we felt it important to test how much data needs to be decrypted at once to net a performance increase, as well as how much data can be decrypted at one time before the GPU memory limit and/or thread limit is exceeded.

Figure 1. AES CBC Decryption Times.



The results presented in figure 1 clearly show that while CPU-based decryption of AES is linear, GPU-based decryption is logarithmic. Whereas with 100 bytes (only six blocks), the GPU is around ten times slower than the CPU, when the amount of data to be decrypted reaches 3KB (18750 blocks), decryption becomes faster. With the maximum buffer size of 3.7MB (a higher size results in too many threads for this particular GPU to handle), CBC decryption on a GPU can be up to 43 times faster than on the CPU.

4.1.2 CBC Encryption

Given that CBC Encryption cannot be parallelised at all as encrypting a block requires the ciphertext for the previous block, we opted to run two sets of tests — one where the operation is executed normally, and one where the operation is executed while the CPU is under heavy load.

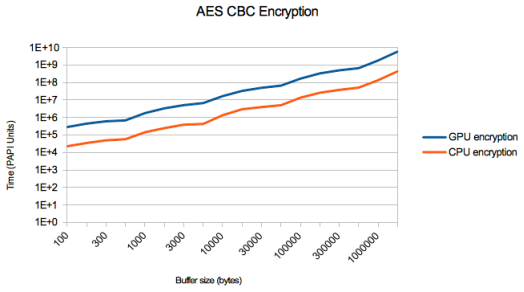


Figure 2. AES CBC Encryption Times.

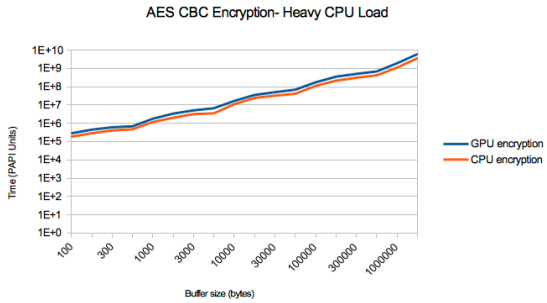


Figure 3. AES CBC Encryption Times — Heavy CPU Load

4.1.3 ECB

Unlike in CBC mode, in ECB mode blocks are encrypted individually, with no inter-dependencies. This means that both encryption and decryption can be parallelised. Additionally, since this lack of inter-dependencies also means that there will be less memory access clashes.

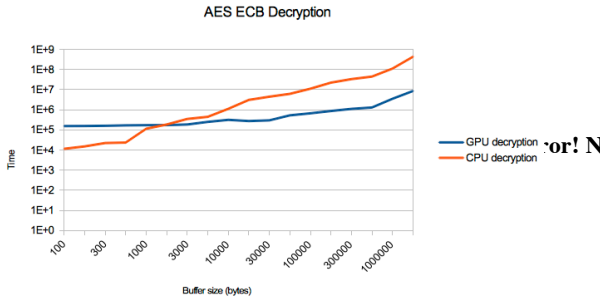


Figure 4. AES ECB Decryption Times

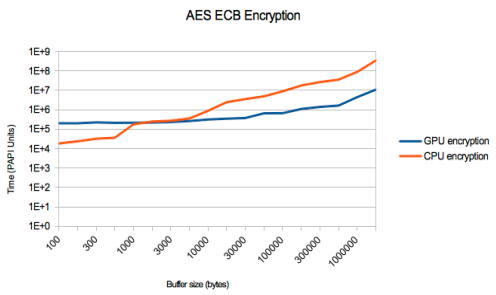


Figure 5. AES ECB Encryption Times

The results presented in figures 4 and 5 show that similarly to CBC decryption, ECB operations, while being slower than the CPU for very small block sizes, gain an extremely large advantage when executed on the GPU in parallel when the buffer is 2KB or larger. At its peak, AES ECB encryption is 31 times faster than on the GPU than on the CPU, and ECB decryption is over 50 times faster than on the CPU.

4.2 RSA

4.2.1 Key Generation

Since key generation is largely dependent on a random factor, we chose to run the key generation 10 times and average the results. We also averaged the CPU usage of each execution, to ascertain how much the CPU might be lightened by moving this operation to the GPU.

Table 1. RSA Key Generation times.

-	GPU time	CPU time	GPU – CPU Cycle Time	CPU – CPU Cycle Time
1024 bits, 1 key	0.107s	0.038s	0.009s	0.027s
1024 bits, 2 keys	0.109s	0.084s	0.009s	0.066s
1024 bits, 5 keys	0.111s	0.181s	0.010s	0.108s
1024 bits, 10 keys	0.117s	0.360s	0.012s	0.288s
1024 bits, 100 keys	0.816s	4.347s	0.092s	2.889s
2048 bits, 1 key	0.131s	0.192s	0.009s	0.119s
2048 bits, 2 keys	0.133s	0.393s	0.009s	0.370s
2048 bits, 5 keys	0.135s	1.064s	0.010s	1.016s
2048 bits, 10 keys	0.137s	1.908s	0.011s	1.815s
2048 bits, 100 keys	0.997s	16.098s	0.154s	14.564s
4096 bits, 1 key	0.225s	2.180s	0.009s	2.078s
4096 bits, 2 keys	0.228s	4.119s	0.009s	4.085s
4096 bits, 5 keys	0.232s	10.592s	0.011s	10.516s
4096 bits, 10 keys	0.233s	18.782s	0.012s	18.648s
4096 bits, 100 keys	11.594s	2m38.819s	6.824s	2m36.880s

Table 2. RSA Key Generation times with a heavy CPU load.

-	GPU time	CPU time	GPU – CPU Cycle Time	CPU – CPU Cycle Time
1024 bits, 1 key	0.108s	0.065s	0.009s	0.026s
1024 bits, 2 keys	0.109s	0.159s	0.009s	0.092s
1024 bits, 5 keys	0.110s	0.206s	0.010s	0.108s
1024 bits, 10 keys	0.119s	0.539s	0.012s	0.285s
1024 bits, 100 keys	0.856s	6.145s	0.012s	2.889s

When generating 1024-bit keys, a prime number is found relatively quickly. As such, the CPU performs better than the GPU when generating a single, or even two 1024-bit keys, as seen in table 1. However, even when generating 1024-bit keys, the GPU gains ground as the number of generated keys is increased, due to the aforementioned prime caching system. When the number of bits in the key is increased, the more attempts have to be made to generate prime numbers. As such, even when generating a single key, the GPU is slightly faster than the CPU for 2048-bit keys, and significantly faster for 4096-bit keys.

4.2.2 Cipher

Table 3. RSA Cipher Execution – Single message

Key size (bits)	GPU time	CPU time
1024	1.654s	0.074s
2048	6.634s	0.448s
4096	12.2833s	2.902s

Table 4. RSA Cipher Execution – Heavy CPU load

Key size (bits)	GPU time	CPU time
1024	1.742s	0.142s
2048	6.682s	0.543s
4096	12.395s	3.645s

Table 5. RSA Cipher Execution – multiple messages (4096-bit key)

Number of Messages	GPU time	CPU time
1	12.283s	2.900s
5	1m02.673s	3.727s
10	2m17.780s	5.777s
15	2m36.600s	5.841s

The results in table 3 show that RSA performs significantly poorly on the GPU. However, the results under heavy CPU load (table 4) show that for 4096-bit keys, it's worth using the GPU when RSA execution is lower priority and the CPU needs to be free for other, more important tasks.

5 Conclusions

OpenSSL is a widely-used library used by a large number of applications around the world. Rewriting them all to use a different library in order to take advantage of the GPU would be an unfeasible, monumental task. With this work, we present an engine that allows existing applications to skip that complex stage and take advantage of the GPU to increase performance and lighten CPU load on the most commonly used encryption algorithms, implemented with both OpenCL and CUDA. Our evaluation shows that significant performance gains are obtained on RSA key generation (especially when generating a large number of keys), AES ECB

operations, and AES CBC decryption, and these gains can be applied right now to existing applications with minimal effort. Unfortunately, the RSA cipher did not gain any performance, given the way OpenSSL works, the GPU has to be called individually for each operation which results in too large a performance loss. However, even in such a situation, the GPU can be used as a co-processor to lighten CPU load for other, more important tasks. The same can be applied for AES CBC Encryption, which while performing over ten times slower on the GPU, only requires 2.7% of the CPU processing power. CBC Decryption can be up to 43 times faster, whereas ECB encryption can be 31 times faster, and ECB decryption can be over 50 times faster.

Given the way OpenSSL works, the GPU has to be called individually for each operation, resulting in too large a performance hit. However, it may be feasible to create a manager that caches RSA requests and sends them all at once to the GPU. Another possibility that was not implemented was AES's CTR cipher mode, which similarly to ECB, can be parallelised in both encryption and decryption --- this was due to the fact that CTR mode in OpenSSL is still not considered stable (and is in fact disabled by default), however, it seems to be gaining more usage lately. We leave these issues to future work.

Acknowledgments

References

1. ATI Stream Computing, Programming Guide, http://developer.amd.com/gpu_assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf
2. The OpenCL Specification, version 1.2, http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf
3. About the OpenSSL Project, http://www.openssl.org/about/
4. Porting CUDA Applications to OpenCL, http://developer.amd.com/zones/OpenCLZone/programming/pages/portingcudatoopencl.aspx
5. OpenSSL, http://openssl.org
6. OpenSSL-GPU, http://labs.sasslantis.ee/openssl-gpu/
7. Using Graphic Processing Unit in Block Cipher Calculations, http://labs.sasslantis.ee/wp-content/uploads/2011/03/mastersthesis.pdf
8. SHA1 Hash Algorithm OpenCL implementation, http://royger.org/opencl/?p=12
9. Omegaice (James Sweet) GitHub, https://bitbucket.org/Omegaice/
10. Fast Implementation of Two Hash Algorithms on nVidia CUDA GPU, http://upcommons.upc.edu/pfc/bitstream/2099.1/7933/1/Masteoppgave.pdf
11. THIRD (FINAL) ROUND CANDIDATES, http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/submissions_rnd3.html
12. OCLCrypto, https://github.com/kazuki/oclcrypto
13. Camellia algorithm, http://info.isl.ntt.co.jp/crypt/eng/camellia/index.html

14. NVIDIA's Next Generation CUDA Compute Architecture, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
15. OpenSSL: Documents, engine(3), http://www.openssl.org/docs/crypto/engine.html
16. Introduction to Cryptography: Principles and Applications, Hans Delfs and Helmut Knebl
17. OpenSSH, http://www.openssh.org/
18. Using the Cryptographic Accelerators in the UltraSPARC T1 and T2 processors, http://www.oracle.com/technetwork/server-storage/archive/a11-014-crypto-accelerators-439765.pdf
19. SSLShader: Cheap SSL Acceleration with Commodity Processors, http://shader.kaist.edu/sslshader/sslshader.pdf
20. Porting CUDA Applications to OpenCL, http://developer.amd.com/resources/heterogeneous-computing/opencl-zone/programming-in-opencl/porting-cuda-applications-to-opencl/
21. CU2CL: A CUDA-to-OpenCL Translator for Multi-and Many-Core Architectures, http://scholar.lib.vt.edu/theses/available/etd-07282011-122302/unrestricted/Martinez_GE_T_2011.pdf
22. OpenSSL: engine(3), http://www.openssl.org/docs/crypto/engine.html
23. OpenSSL: BN_generate_prime(3), https://www.openssl.org/docs/crypto/BN_generate_prime.html
24. PAPI: Performance Application Programming Interface, http://icl.cs.utk.edu/papi/
25. OpenCL Reference Pages, http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/
26. CUDA Documents, http://docs.nvidia.com/cuda
27. cuRAND, https://developer.nvidia.com/curand
28. Random123: a Library of Counter-Based Random Number Generators, http://www.thesalmons.org/john/random123/releases/1.08/docs/index.html
29. D. E. Knuth. The Art of Computer Programming, volume 2, Addison-Wesley, 3rd edition, 1997.