Efficient Montgomery Multiplication on GPUs

Nicolae Roşia[1,2(✉)], Virgil Cervicescu[2], and Mihai Togan[3]

[1] Advanced Technology Institute, Bucharest, Romania
[2] Military Technical Academy, Bucharest, Romania
{nicolae.rosia,virgil.cervicescu}@gmail.com
[3] certSIGN, Bucharest, Romania
mihai.togan@certsign.ro

Abstract. Public-key cryptosystems and algorithms, including RSA [20], EC and Diffie-Hellman key exchange [5], require efficient large integer arithmetic in finite fields. Contemporary processors are not designed to support such operations in a productive manner, since most of them natively work on 8 to 64 bit word sizes. Thus, an expensive cryptographic accelerator is frequently required to offload the computational burden. In this paper, we focus on a highly parallel architecture which is commonly found in commodity computers, *i.e.* the Graphical Processing Unit (GPU). Recently, GPUs have known an exponential growth in terms of computing power, becoming a cost-effective option for offloading computationally intensive tasks. This paper describes a parallel implementation of the Montgomery Multiplication, as well as optimizations that enable efficient exploitation of the CUDA GPU architecture.

Keywords: Mongtomery multiplication · Modular exponentiation · CUDA · GPGPU

1 Introduction

Asymmetric cryptographic algorithms and protocols, including RSA, EC-based and Diffie-Hellman key exchange, require efficient large integer arithmetic. This implies performing exponentiations and modular reductions, therefore a chain of repetitive operations upon different data. Typically, the sizes of the operands are between 1024 and 4096-bit. Given two operands, A and B, of S_A and S_B bits, the result of $A \times B$ will have a maximum of $S_A + S_B + 1$ bits. Considering the operands sizes and the fact that most processors have a word length between 8 and 64-bits, multiple precision arithmetic is implemented in software, in the detriment of computational performance.

N. Roşia—Author partially supported by the Romanian National Authority for Scientific Research (CNCS-UEFISCDI) under the project PN-II-PT-PCCA-2013-4-1651.

M. Togan—Author partially supported by the Romanian National Authority for Scientific Research (CNCS-UEFISCDI) under the project PN-II-IN-DPST-2012-1-0087 (ctr. 10DPST/2013). All the authors contributed equally to this work.

I. Bica et al. (Eds.): SECITC 2015, LNCS 9522, pp. 119–129, 2015.
DOI: 10.1007/978-3-319-27179-8_9

Large numbers are usually represented in polynomial form as an array of native word size integers:

$$a = \overline{a_{n-1}a_{n-2}\dots a_0}_{(\beta)} = \sum_{i=0}^{n-1} a_i \beta^i \tag{1}$$

General Purpose GPU Programming became easier with the introduction of programming frameworks like OpenCL and CUDA which enable the use of a GPU as a coprocessor. Current GPUs exhibit good Floating-point Operations Per Second (FLOPS) per dollar ratio and represent an attractive way to offload computationally intensive tasks.

1.1 Related Work

One of the first usages of the graphical processing units within the cryptography field was focused on accelerating of symmetric ciphers. The first known implementation of this kind was made by Cook et al. [4]. Further on, researchers have developed various solutions for this purpose using parallel architecture of the GPUs. These implementations have been proven to be more useful than the usual CPU-based implementations. The [13] presents a CUDA-based implementation for AES algorithm which was tested on the NVidia GeForce 8700 and 8800 GTX graphical cards. At the time of writing, the developed solution ran up to 20 times faster than the OpenSSL [1] CPU-based solution. A new block based conventional implementation of AES having a 4-10x speed improvements over CPU solutions is pointed out in [9]. They outlined a general purpose data model for encapsulating cryptographic functions (client requests) which is suitable for an execution on a GPU. They used this general model to investigate how the data input can be mapped to the threading model of the GPUs for several of the AES operation modes.

Currently, there are a number of publications presenting mathematical results and optimizations for the modular multiplication (and exponentiation) algorithms. Various techniques were proposed in addition to practical implementations of these algorithms. These techniques are mainly based on parallelization which is a mechanism perfectly applicable on hardware processing technologies like FPGAs or GPUs.

The integration of multiple-precision multiplication with modular reduction as stated by Montgomery's method [14] along with improvements regarding the interleaving of multiplication and reduction are described in [6]. Five Montgomery Multiplication algorithm flavours along with their space, time requirements and actual performance results are discussed in [10]. The *Coarsely Integrated Operand Scanning (CIOS)* method proved to be the most efficient of all. Given two operands of sizes s, *CIOS* requires $2s^2 + s$ multiplications, $4s^2 + 4s + 2$ additions, $6s^2 + 7s + 2$ reads, and $2s^2 + 5s + 1$ writes and needs $s + 3$ words of memory space.

Many papers present GPU implementations of public key and elliptic curve cryptography needed primitives. All of these were focused on speeding up

operations like modular multiplication, exponentiation or elliptic curve scalar multiplication. One of the first was performed in [21], where by using an NVIDIA 7800 GTX GPU, they reported, at the paper time, a speedup factor of 3 relative to the reference CPU. Other works in this way are referenced by [2,3,7,8,15,21]. A more recent GPU implementation of the Montgomery multiplication algorithm for a field size of 112 to 521 bits is discussed in [12]. Their work, which is an improvement of a previous approach [11], regards the GPU-based NIST prime field multiplication and employs Montgomery algorithm to allow any field prime to be used in this case. They also bring some new implementation techniques which led to eliminating the need for GPU cache accesses and to gaining this way a bigger throughput that could to accelerate EC cryptography. Experiments and measurements have been conducted on an NVIDIA GTX 480 GPU with reported speeds significantly higher than other published CPU and GPU-based implementations. In [22] are proposed several optimizations on modular multiplication algorithms. The implementation uses the OpenCL framework and the tests have been conducted on an AMD Radeon HD5870 graphic card. After applying the optimizations, they could deliver up to 11 % more arithmetical throughput.

Structure of the Paper. The rest of this paper is organized as follows:

Section 2 reviews the basic concepts and introduces the notions used throughout the paper. Namely, summary elements about *Montgomery Reduction*, *Binary exponentiation* and *Montgomery's ladder technique* are presented in Subsects. 2.1, 2.2, and respectively 2.3, while an overview of the GPU architecture used is presented in Subsect. 2.4. Section 3 describes our implementation details regarding the *CIOS Method* on CUDA, along with the results obtained and their interpretation (Sect. 3.1). Finally, conclusions are outlined in Sect. 4.

2 Preliminaries

2.1 Montgomery Multiplication

Commonly used public key cryptographic algorithms imply large integer arithmetic operations, *e.g.* modular multiplication and exponentiation [20]. A straightforward approach when computing a modular product consists of operands' multiplication followed by the reduction of the partial result. Considering the magnitude of the numbers (thousands of bits), multiprecision multiplications and repeated subtractions are necessary steps. From a computational perspective, both of the previously mentioned operations are costly. Modular exponentiation, *i.e.* $a^b \bmod n$, can be computed by multiplying a by itself b times and then reducing the result modulo n. After each multiplication, the memory requirements increase by a number of bits equivalent to the size of a. In practice, this is clearly not a feasible way of tackling the problem. Applying the modulus reduction at each step reduces the required memory. However, by doing so the number of operations greatly increases.

In 1985, Peter Montgomery introduced the *Montgomery Reduction* algorithm [14], which enables the modular multiplication ($c \equiv a \times b \bmod N$) to be computed using a different modulo. This method requires using a *residue* form of the operands, $\overline{a}$ and $\overline{b}$.

The first step of the *Montgomery Reduction* algorithm consists of choosing a number R *s.t.* $R > N$ and $\gcd(R, N) = 1$. Moreover, R is often conveniently chosen to be a power of base β in which the processor operates. In our case, the basis is $\beta = 2$. Assuming that N is an odd prime number of w bits, choosing $R = 2^w$ satisfies the requirements. With such an R, division and remainder operations become bitwise mask and shifting operations.

The next step involves transforming the operands a and b into their reduced forms, as illustrated in (2), and finding R's inverse, *i.e.* $RR^{-1} \equiv 1 \bmod N$.

$$\begin{aligned} \overline{a} &\equiv aR \bmod N \\ \overline{b} &\equiv bR \bmod N \end{aligned} \tag{2}$$

Having $\overline{a}$ and $\overline{b}$, we can further compute $\overline{c}$:

$$\overline{c} \equiv cR \equiv (a \times b)R \equiv (aR \times bR)R^{-1} \equiv (\overline{a} \times \overline{b})R^{-1} \bmod N \tag{3}$$

The initial c, can be calculated by applying the inverse Montgomery transformation:

$$c \equiv \overline{c}R^{-1} \bmod N \tag{4}$$

The above presented steps, represent the conversion to and from the Montgomery reduced form, and do not serve in the speed up of the commencing computation. Moreover, converting operands of a single multiplication to their *residue* form, in order to apply *Montgomery Reduction* is disadvantageous compared to the straightforward method, but a substantial gain is obtained in exponentiation operations. Algorithm 1 illustrates the computation of $\overline{c}$. It can be observed that all arithmetic operations are performed modulo R, task which can be easily solved by means of the processor. A performance analysis of the algorithm (together with its multiple implementations) can be found in [10]. The *Coarsely Integrated Operand Scanning (CIOS)* method proved to be the most efficient of all five algorithms analyzed.

The Coarsely Integrated Operand Scanning (CIOS) Method. The Montgomery reduction is intrinsically a right-to-left procedure. This allows us to compute one word at a time of t since $m[i]$ depends only on $t[i]$, [6]. The *CIOS* method (Algorithm 2) takes advantage of this property and integrates the multiplication and reduction steps by alternating between the iterations of the outer loops. Assuming that both $\overline{a}$ and $\overline{b}$ have s words, the *CIOS* variant requires:

- $2s^2 + s$ multiplications
- $4s^2 + 4s + 2$ additions
- $6s^2 + 7s + 2$ reads
- $2s^2 + 5s + 1$ writes
- $s + 3$ words of memory space

Algorithm 1. Montgomery multiplication

Input: $\overline{a}$, $\overline{b}$, N, R
Output: $(\overline{a} \times \overline{b})R^{-1} \bmod N$
1: $n' \equiv -N^{-1} \bmod R$
2: $t \leftarrow (\overline{a} \times \overline{b})$
3: $m \leftarrow t \times n' \bmod R$
4: $t \leftarrow (t + m \times N)/R$
5: **if** $t \geq N$ **then**
6: **return** $t - N$
7: **else**
8: **return** t
9: **end if**

2.2 Binary Exponentiation

Given a large integer exponent, e, with its binary representation $e = \overline{e_{n-1}e_{n-2}\dots e_0}_{(2)}$, the computation of a^e resumes to a series of square and multiply operations, as we can see within the Algorithm 3. The computational complexity of the algorithm is $\mathcal{O}(\log_2 n)$ since there are $log_2 n$ squarings and a maximum of $log_2 n$ multiplications. In asymmetric cryptosystems, the encryption often involves the use of an exponent which must be kept secret. In this the method, the number of multiplications depend on the value of exponent which makes it vulnerable to side-channel attacks.

2.3 Montgomery's Ladder Technique

The technique presented in Algorithm 4 addresses the side-channel vulnerability of Algorithm 3 by performing a fixed sequence of operations regardless of the bit's value in exponent.

2.4 Compute Unified Device Architecture (CUDA)

The GPU and CPU architectures are very different. CPUs have few cores (*e.g.* 1 to 32) running at high clock rates and put a great emphasis on big memory caches, complex control logic including branch prediction, speculative execution but have expensive context switching between threads. In contrast, GPUs have many cores (*e.g.* 128 to 2048) running at lower clock rates and are designed to execute hundreds of threads at the same time [17,18]. These cores have small memory caches and simple control logic. The downside is that GPUs are only efficient in processing tasks which are highly data parallel. In *CUDA*, the basic working unit is the *thread* and is executed by a *CUDA Core*. A *Streaming Multiprocessor (SM)* creates and executes groups of 32 threads called *warps*. The threads are characterized by having their own stack and set of registers including program counter and by being free to branch and execute independently. On the other hand, a *warp* executes a single common instruction at a time, so full

Algorithm 2. CIOS method for Montgomery multiplication

Input: $\overline{a}$, $\overline{b}$, N, $R = 2^{s \cdot w}$, w being the processor word size and s the number of words.
Output: $(\overline{a} \times \overline{b})R^{-1} \bmod N$

$n' \equiv -N[0]^{-1} \bmod R$
$t \leftarrow 0$
for $i = 0 \rightarrow s - 1$ **do**
    $C \leftarrow 0$
    for $j = 0 \rightarrow s - 1$ **do**
        $(C, S) \leftarrow t[j] + \overline{a}[j] \cdot \overline{b}[i] + C$
        $t[j] \leftarrow S$
    end for
    $t[s] \leftarrow S$
    $t[s + 1] \leftarrow C$
    $C \leftarrow 0$
    $m \leftarrow t[0] \cdot n' \bmod 2^w$
    $(C, S) \leftarrow t[0] + m \cdot N[0]$
    for $j = 1 \rightarrow s - 1$ **do**
        $(C, S) \leftarrow t[j] + m \cdot N[j] + C$
        $t[j - 1] \leftarrow S$
    end for
    $(C, S) \leftarrow t[s] + C$
    $t[s - 1] \leftarrow S$
    $t[s] \leftarrow t[s + 1] + C$
end for
if $t \geq N$ **then**
    return $t - N$
else
    return t
end if

Algorithm 3. Binary Exponentiation

Input: a, $e = \overline{e_{n-1} e_{n-2} \cdots e_0}_{(2)}$
Output: a^e

$x \leftarrow 1$
for $i = n - 1 \rightarrow 0$ **do**
    $x \leftarrow x \cdot x$
    if $e_i = 1$ **then**
        $x \leftarrow x \cdot a$
    end if
end for
return x

efficiency is achieved when all threads of a *warp* follow a common path. Multiple *warps* compose into *thread blocks (TB)* which in turn reside on a *SM*. The *SMs* have limited resources and developers using *CUDA* must take in account these hardware limitations in order to maximize the occupancy and to exploit the hardware-based task switching designed to hide memory access latency.

Algorithm 4. Montgomery's ladder technique

Input: a, $e = \overline{e_{n-1}e_{n-2}\dots e_0}_{(2)}$
Output: a^e
1: $x_1 \leftarrow a$
2: $x_2 \leftarrow a^2$
3: **for** $i = n-2 \rightarrow 0$ **do**
4: **if** $e_i = 0$ **then**
5: $x_2 \leftarrow x_1 \cdot x_2$
6: $x_1 \leftarrow x_1^2$
7: **else**
8: $x_1 \leftarrow x_1 \cdot x_2$
9: $x_2 \leftarrow x_2^2$
10: **end if**
11: **end for**
12: **return** x_1

The level of occupancy depends on the amount of registers and shared memory used by the *kernel* and the generation of the *CUDA Architecture* being used. The current *GPUs* tend to have multiple *SMs*. Communication between *SMs* is not recommended since it is done through the *global memory* which is slower than the *shared memory* available per *thread block*. Unique IDs are given to *threads* and *blocks*, which are accessible through built-in variables, *threadIdx* and *blockIdx*, thus allowing the *threads* to uniquely identify the data which is going to operate on. Threads within a *thread block* can communicate efficiently through shared memory and synchronize through hardware barriers invoked by calling the intrinsic function, `__syncthreads()`.

CUDA C [18] allows developers to use the C programming language to create C functions called *kernels* which are executed in parallel by *CUDA Cores* on the device. A *CUDA Program* consists of a device kernel and a host program. Since the CPU and GPU have their own separate memory, the host program is responsible for transferring the required data necessary for execution. A typical workflow consists of the following steps achieved by calling the relevant *CUDA Application Programming Interface (API)*:

1. Allocate memory on the device;
2. Transfer data from host to device;
3. Start the execution of the kernel;
4. When the kernel is done executing, transfer the result from the device.

3 Implementation and Results

Our implementation leverages the Montgomery Reduction and Montgomery Ladder technique to efficiently compute the exponentiation required by the RSA encryption. The GPU used in our work, *Gefore GTX 750* [16], has a *CUDA Compute Capability 5.0* architecture, with 512 *CUDA Cores* running at a base clock of 1.14 GHz and a memory bandwidth of 80 GB/s. These cores are partitioned

over 4 *Streaming Multiprocessors*, each having its own resources. To obtain the maximum device utilization and to ensure that the memory latency doesn't affect performance, 8 blocks per *Streaming Multiprocessor* were allocated and the number of registers per thread was limited to 32. Each thread operates on word level and calculates a word of the output. The CUDA Architecture has a 32 bit word size, hence the number of threads needed to operate on a number can be calculated by diving the number's bit size to 32.

Memory coalescing of GPU RAM operations is mandatory to obtain peak memory transfer bandwidth. This is guaranteed by having consecutive threads accessing consecutive memory locations, *i.e.* thread 0 reads and writes to word index 0. This is illustrated in Fig. 1, where a block of $n \cdot s$ threads are processing operands of s words.

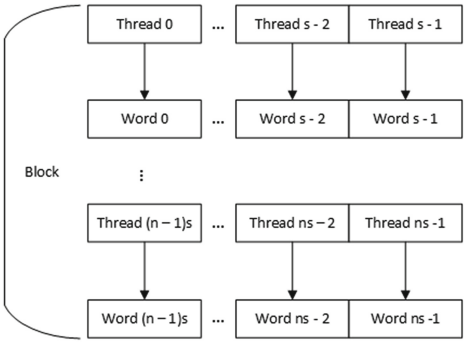

Fig. 1. Threads organization inside a thread block.

Square and multiply are handled differently, exploiting the fact that when computing, $\overline{a} \cdot \overline{a}$, the processor needs a single memory access.

In the scenario where batch operations are performed, *e.g.* RSA encryptions with the same private key, we can precompute the value of N and $n'[0]$ used in Algorithm 2 and embed the results in the code. The compiler is then able to reduce the number of required registers and memory accesses, resulting in a relevant performance gain.

Loop unrolling eliminates the overhead implied by loop counters and loop arithmetic and facilitates additional compiler optimizations.

Shared memory at block level was used for its low latency access times. The 64KB shared memory [19] is big enough to cache the input operands and store the intermediate results accessed by all worker threads.

3.1 Results

The Table 1 presents the speedups obtained, relative to a CPU-based implementation (Intel i7-4790K at 4.0 GHz). The variable factors taken into account were

the *Loop Unroll depth*, whether the precomputed n' is embedded in the code, and the operands bit-size.

It is worth noting that these results were obtained at 100 % GPU utilization. In order to obtain this degree of utilization, it is necessary to queue multiple requests which can lead to increased latency in processing. Depending on the actual setup, it might not always be beneficial to offload the computations if low latency is required.

Firstly, it can be observed that increasing the operands bit-size, degrades performance. This can be explained by the increased number of threads required to compute the result which need to synchronize. The biggest speedup is obtained with 1024 bit operands because 32 threads are needed to compute the result, resembling a *warp*, which is inherently synchronized.

Secondly, (as expected) precomputing n', yields a significant boost in speedup since we eliminate the memory access penalty associated with reading the n' variable.

Finally, loop unrolling depth affects performance in a not so obvious way. The best results for 1024 bit operands, were obtained at a depth of value 8. This is not the case for 2048/4096 bit operands since the best speedup was obtained when performing full loop unrolling.

Table 1. Results for 1024/2048/4096 exponentiation

Unroll depth	n' constant	Speedup		
		1024	2048	4096
1	false	5.83	5.10	4.70
1	true	5.86	5.35	4.73
2	false	5.98	5.39	4.88
2	true	6.18	5.56	4.91
4	false	6.17	5.48	4.98
4	true	6.25	5.68	5.02
8	false	6.25	5.57	5.03
8	true	6.55	5.63	5.26
16	false	6.20	5.61	5.07
16	true	6.52	5.66	5.28
32	false	6.29	5.59	5.06
32	true	6.44	5.67	5.29

4 Conclusions

This paper has presented a high throughput GPU implementation of modular exponentiation, as well as optimization suitable for the SIMT (*Single Instruction Multiple Threads*) architecture. This design could be used in the context

of a cryptographic accelerator. As shown in this article, parallel architecture has proven to be a feasible approach to overcome operating frequency limitations imposed by the current state of technology in CPUs. Despite the increased latency that they may cause, considering the encouraging results obtained, we are confident that further research, coupled with the increase of core count in GPUs, will only increase performance of many-core architectures. As further steps, we plan to embed this work within an open cryptographic API (*e.g.* OpenSSL) in order to evaluate the real acceleration gained at the high level protocols and applications like SSL and web servers.

References

1. OpenSSL: The Open Source toolkit for SSL/TLS. http://www.openssl.org
2. Antao, S., Bajard, J.C., Sousa, L.: Elliptic curve point multiplication on GPUs. In: Charot, F., Hannig, F., Teich, J., Wolinski, C., (Eds.) ASAP, pp. 192–199. IEEE (2010)
3. Cohen, A.E., Parhi, K.K.: GPU accelerated elliptic curve cryptography in GF(2^m). In: Proceedings of the 2010 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS), Seattle, WA, pp. 57–60 (2010)
4. Cook, D.L., Ioannidis, J., Keromytis, A.D., Luck, J.: CryptoGraphics: secret key cryptography using graphics cards. In: Menezes, A. (ed.) CT-RSA 2005. LNCS, vol. 3376, pp. 334–350. Springer, Heidelberg (2005)
5. Diffie, W., Hellman, M.: New directions in cryptography. IEEE Trans. Inf. Theor. **22**(6), 644–654 (2006). http://dx.doi.org/10.1109/TIT.1976.1055638
6. Dussé, S.R., Kaliski, Jr., B.S.: A cryptographic library for the Motorola DSP 56000. In: Damgård, I.B. (ed.) EUROCRYPT 1990. LNCS, vol. 473, pp. 230–244. Springer, Heidelberg (1991)
7. Fleissner, S.: GPU-accelerated Montgomery exponentiation. In: Shi, Y., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2007, Part I. LNCS, vol. 4487, pp. 213–220. Springer, Heidelberg (2007)
8. Giorgi, P., Izard, T., Tisserand, A.: Comparison of modular arithmetic algorithms on GPUs. In: Proceedings of International Conference on Parallel Computing ParCo, Lyon, France (2009)
9. Harrison, O., Waldron, J.: Practical symmetric key cryptography on modern graphics hardware. In: 17th USENIX Security Symposium, pp. 195–209 (2008)
10. Koç, C., Acar, T., Kaliski, B.J.: Analyzing and comparing Montgomery multiplication algorithms. IEEE Micro **16**(3), 26–33 (1996)
11. Leboeuf, K., Muscedere, R., Ahmadi, M.: High performance prime field multiplication for GPU. In: 2012 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 93–96, May 2012
12. Leboeuf, K., Muscedere, R., Ahmadi, M.: A GPU implementation of the Montgomery multiplication algorithm for elliptic curve cryptography. In: IEEE International Symposium on Circuits and Systems (ISCAS 2013), pp. 2593–2596, May 2013
13. Manavski, S.A.: CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In: IEEE International Conference on Signal Processing and Communications (ICSPC 2007), 24–27 November 2007, Dubai, United Arab Emirates, pp. 65–68 (2007)

14. Montgomery, P.L.: Modular multiplication without trial division. Math. Comput. **44**(170), 519–521 (1985)
15. Moss, A., Page, D., Smart, N.P.: Toward acceleration of RSA using 3D graphics hardware. In: Galbraith, S.D. (ed.) Cryptography and Coding 2007. LNCS, vol. 4887, pp. 364–383. Springer, Heidelberg (2007)
16. NVIDIA Corporation: GeForce GTX 750 Specifications
17. NVIDIA Corporation: CUDA C Best Practices Guide, 7.0 edn. (2015)
18. NVIDIA Corporation: CUDA C Programming Guide, 7.0 edn. (2015)
19. NVIDIA Corporation: Tuning CUDA Applications for Maxwell, 7.0 edn. (2015)
20. Rivest, R., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**, 120–126 (1978)
21. Szerwinski, R., Güneysu, T.: Exploiting the power of GPUs for asymmetric cryptography. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 79–99. Springer, Heidelberg (2008)
22. Trei, W.: Efficient Modular Arithmetic for SIMD Devices. In: IACR Cryptology ePrint Archive 2013, 652 (2013)