Implementation of Multiple-precision Modular Multiplication on GPU

Kaiyong Zhao

Abstract

Multiple-precision modular multiplications are the key components in security applications, like public-key cryptography for encrypting and signing digital data. But unfortunately they are computationally expensive for contemporary CPUs. By exploiting the computing power of the many-core GPUs, we implemented a multiple-precision integer library with CUDA. In this paper, we will investigate the implementation of two approaches of multiple-precision modular multiplications on GPU. We will analyze the detail of the instructions of multiple-precision modular multiplication on the GPU and find the hit issues, and then we propose to use the inline ASM to improve the implementation of this function. Our experimental results show that the performance of multiple-precision modular multiplication has been improved by 20%.

Keywords

GPU computing, CUDA, Multiple-precision Modular Multiplication, Karatsuba multiplication, Montgomery modular multiplication

1. Introduction

In recent years, peer-to-peer (P2P) content distribution applications such as BitTorrent and ppLive, have become the most popular Internet applications due to their scalability and robustness. Network coding has been proposed as an effective mechanism to improve the performance of such P2P applications [14]. However, P2P applications with network coding suffer from the notorious pollution attacks: a malicious node can send out bogus packets which will be merged into other genuine packets and propagated into the whole network at an exponential speed. To resolve this problem, homomorphic hash functions have to be applied such that the hash of any encoded packet can be effectively derived from the hashes of the original packets, which enables the detection of bogus packets before a peer encodes it with other packets [12]. Unfortunately homomorphic hash functions rely on multiple-precision modular operations and are computationally expensive [10] [12].

Recent advances in Graphics Processing Units (GPUs) open a new era of GPU computing [20]. For example, commodity GPUs like NVIDIA's GTX 280 has 240 processing cores and can achieve 933 GFLOPS of computational horsepower. More importantly, the NVIDIA CUDA programming model makes it easier for developers to develop non-graphic applications using GPU [1] [4]. In CUDA, the GPU becomes a dedicated coprocessor to the host CPU, which works in the principle of Single-Program Multiple Data (SPMD) where multiple threads based on the same code can run simultaneously.

In this paper, we present our study of using GPUs for multiple-precision modular multiplication, which is the key component of not only the homomorphic hashing, but also a number of security applications that make use of public-key cryptography. We implement two methods of multiple-precision modular multiplications on NVIDIA GPUs using CUDA. One is Coarsely Integrated Operand Scanning (CIOS) Montgomery method; the second one is Karatsuba multiplication with Montgomery. In order to achieve the highest performance, we analyze the ptx code (i.e., the assembly code for NVIDIA GPUs) of the implementation. As the CUDA PTX standard is not open to the public, we rely on the decuda tool to disassemble the compiled binary cubin file and search for the bottleneck for an improvement.

The contribution of this work is twofold:

- First, we designed and implemented two approaches of multiple-precision modular multiplication on the CUDA architecture.
- Second, we analyze the implementation of the directive instructions, and use inline ASM to optimize the Montgomery algorithm.

The rest of the paper is organized as follows. Section 2 provides background information on Karatsuba Mulitiplication, Montgomery algorithm, GPU architecture, and CUDA programming model. Section 3 presents the design of multiple-precision modular arithmetic on GPU. Section 4 presents the decuda analysis, and then presents our inline ASM implementation of the 32-bit integer multiplication. Experimental results are presented in Section 5, and we conclude the paper in Section 6.

2. Background and Related Work

In this section, we provide the required background knowledge of Karatsuba multiplication, Montgomery multiplication algorithm, GPU architecture, and CUDA programming model.

2.1 Karatsuba multiplication

The Karatsuba multiplication algorithm has been described by Knuth, which is possible to perform multiplication of large numbers in (many) fewer operations than the usual brute-force technique of "long multiplication". As discovered by Karatsuba [25]， multiplication of two *n*-digit numbers can be done with a bit complexity of less than n^2 steps using identities of the form

$$(a+b\cdot 10^n)(c+d\cdot 10^n)=a\cdot c+[(a+b)(c+d)-a\cdot c-b\cdot d]$$

The inputs x and y are treated as each split into two parts of equal length (or the most significant part one limb shorter if N is odd).

Let *b* be the power of 2 where the split occurs, ie. if x_0 is k limbs (y_0 the same) then $b=2^{(k \cdot mp\_bits\_per\_limb)}$. With that $x=x_1 \cdot b+x_0$ and $y=y_1 \cdot b+y_0$, and the following holds,

$$x\cdot y=(b^2+b)x_1y_1-b\cdot(x_1-x_0)\cdot(y_1-y_0)+(b+1)\cdot x_0\cdot y_0$$

This formula means doing only three multiplies of $N^2/4$ limbs, whereas a base case multiply of N^2 limbs is equivalent to four multiplies of $N^2/4$. The factors (b^2+b) etc represent the positions where the three products must be added. [3]

The term $(x_1-x_0)\cdot(y_1-y_0)$ is best calculated as an absolute value, and the sign used to choose to add or subtract. Notice the sum $high(x_0 \cdot y_0) + low(x_1 \cdot y_1)$ occurs twice, so it is possible to do $5\cdot k$ limb additions, rather than $6\cdot k$.

Squaring is similar to multiplying, but with $x = y$ the formula reduces to an equivalent with three squares,

$$x^2=(b^2+b)x_1^2-b(x_1-x_0)^2+(b+1)x_0^2$$

The final result is accumulated from those three squares the same way as for the three multiplies above. The middle term $(x_1-x_0)^2$ is now always positive.

A similar formula for both multiplying and squaring can be constructed with a middle term $(x_1+x_0)\cdot(y_1+y_0)$. But those sums can exceed k limbs, leading to more carry handling and additions than the form above.

Karatsuba multiplication is asymptotically an $O(N^{1.585})$ algorithm, the exponent being log(3)/log(2), representing 3 multiplies each 1/2 the size of the inputs. This is a big improvement over the base case multiply at $O(N^2)$ and the advantage soon overcomes the extra additions Karatsuba performs.

2.2 Montgomery Multiplication

The classical modular multiplication is suitable for normal operations. However, when performing modular exponentiations, Montgomery multiplication shows much better performance advantage [5]. The following gives the Montgomery reduction and Montgomery multiplication algorithms.

Montgomery reduction is an algorithm introduced in 1985 by Peter Montgomery that allows modular arithmetic to be performed efficiently when the modulus is large (typically several hundred bits).

Let m be a positive integer, and let R and A be integers such that $R > m$, $gcd(m, R) = 1$, and $0 \le A < m \cdot R$. The Montgomery reduction of A modulo m with respect to *R* is defined as $A \cdot R^{-1} \bmod m$. In our applications, *R* is chosen as b^n to simply the calculation.

Algorithm 1 Multiple-precision Montgomery Reduction

INPUT: integer *m* with *n* radix *b* digits and gcd(*m*, *b*) = 1, $R=b^n$, $m'=-m^{-1} \bmod b$, and integer *A* with 2*n* radix *b* digits and $A<m\cdot R$.

OUTPUT: $T = A\cdot R^{-1} \bmod m$.

1: $T \leftarrow A$;
2: **for** (*i* **from** 0 **to** $n-1$)
3: $u_i \leftarrow T_i \cdot m' \bmod b$;
4: $T \leftarrow T + u_i \cdot m \cdot b^i$;
5: **end for**
6: $T \leftarrow T / b^n$;
7: **if** ($T \ge m$) **then** $T \leftarrow T - m$;
8: **return** *T*;

Algorithm 2 Multiple-precision Montgomery Multiplication

INPUT: non-negative integer *m*, *x*, *y* with *n* radix *b* digits, $x < m$, $y < m$, and gcd(*m*, *b*) = 1, $R=b^n$, $m'=-m^{-1} \bmod b$.

OUTPUT: $T = x\cdot y\cdot R^{-1} \bmod m$.

1: $T \leftarrow 0$;
2: **for** (*i* **from** 0 **to** $n-1$)
3: $u_i \leftarrow (T_0 + x_i \cdot y_0)\cdot m' \bmod b$;
4: $T \leftarrow (T + x_i \cdot y + u_i \cdot m)/b$;
5: **end for**
6: **if** ($T \ge m$) **then** $T \leftarrow T - m$;
7: **return** *T*;

2.3 GPU Computing and CUDA

GPUs are dedicated hardware for manipulating computer graphics. Due to the huge computing demand for real-time and high-definition 3D graphics, the GPU has evolved into a highly parallel, multithreaded, manycore processor. The advances of computing power in GPUs have driven the development of general-purpose computing on GPUs (GPGPU). The first generation of GPGPU requires that any non-graphics application must be mapped through graphics application programming interfaces (APIs).

Recently one of the major GPU vendors, NVIDIA, announced their new general-purpose parallel programming model, namely Compute Unified Device Architecture (CUDA) [1] [4], which extends the C programming language for general-purpose application development. Meanwhile,

another GPU vendor AMD also introduced Close To Metal (CTM) programming model which provides an assembly language for application development [2]. Intel also exposed Larrabee, a new many-core GPU architecture specifically designed for the market of GPU computing this year [23].

Since the release of CUDA, it has been used for speeding up a large number of applications [17] [18] [20] [21] [22].

The NVIDIA GeForce 8800 has 16 Streaming Multiprocessors (SMs), and each SM has 8 Scalar Processors (SPs), resulting a total of 128 processor cores. The SMs have a Single-Instruction Multiple-Data (SIMD) architecture: At any given clock cycle, each SP of the SM executes the same instruction, but operates on different data. Each SP can support 32-bit single-precision floating-point arithmetic as well as 32-bit integer arithmetic.

Each SM has four different types of on-chip memory, namely registers, shared memory, constant cache, and texture cache. For GeForce 8800, each SM has 8192 32-bit registers, and 16 Kbytes of shared memory which are almost as fast as registers. Constant cache and texture cache are both read-only memories shared by all SPs. Off-chip memories such as local memory and global memory have relatively long access latency, usually 400 to 600 clock cycles [4]. The properties of the different types of memories have been summarized in [4] [17]. In general, the scarce shared memory should be carefully utilized to amortize the global memory latency cost. Shared memory is divided into equally-sized banks, which can be simultaneously accessed. If two memory requests fall into the same bank, it is referred to as bank conflict, and the access has to be serialized.

In CUDA model, the GPU is regarded as a coprocessor capable of executing a great number of threads in parallel. A single source program includes host codes running on CPU and also kernel codes running on GPU. Compute-intensive and data-parallel kernel codes run on GPU in the manner of Single-Process Multiple-Data (SPMD). The threads are organized into blocks, and each block of threads are executed concurrently on one SM. Threads in a thread block can share data through the shared memory and can perform barrier synchronization. Each SM can run at most eight thread blocks concurrently, due to the hard limit of eight processing cores per SM. As a thread block terminate, new blocks will be launched on the vacated SM. Another important concept in CUDA is warp, which is formed by 32 parallel threads and is the scheduling unit of each SM. When a warp stalls, the SM can schedule another warp to execute. A warp executes one instruction at a time, so full efficiency can only be achieved when all 32 threads in the warp have the same execution path. Hence, if the number of threads in a block is not a multiple of warp size, the remaining instruction cycles will be wasted.

3. Multiple-Precision Modular Arithmetic for CUDA

In this section, we present a set of library functions of multiple-precision modular arithmetic implemented on GPUs. These library functions are the cornerstones of the network coding system and homomorphic hash functions. It is of critical importance to implement these library functions efficiently. In modular arithmetic, all operations are performed in a group Z_m, i.e., the set of integers $\{0,1,2,\cdots,m-1\}$. In the following, the modulus m is represented in radix b as $(m_n m_{n-1}\cdots m_1 m_0)_b$ where $m_n \neq 0$. Each symbol m_i, $0 \le i \le n$, is referred to as a radix b digit. Non-negative integers x and y, $x < m$, $y < m$, are represented in radix b as $(x_n x_{n-1}\cdots x_1 x_0)_b$ and $(y_n y_{n-1}\cdots y_1 y_0)_b$ respectively.

We have implemented the following multiple-precision library functions for CUDA:

- Multiple-precision comparison
- Multiple-precision subtraction
- Multiple-precision modular addition
- Multiple-precision modular subtraction
- Multiple-precision multiplication
- Multiple-precision division
- Multiple-precision multiplicative inversion

In this paper, we only present the implementation details in the Montgomery multiplication and the optimization in this paper.

3.1 CIOS Montgomery Reduction

The Coarsely Integrated Operand Scanning method improves on the first one by integrating the multiplication and reduction steps. Specifically, instead of computing the entire product *ab*, then reducing, we alternate between iterations of the outer loops for multiplication and reduction. We can do this since the value of *m* in the *i*th iteration of the outer loop for reduction depends only on the value *t*[*i*], which is completely computed by the *i*th iteration of the outer loop for the multiplication. This leads to the following algorithm:

Algorithm 3 Multiple-precision Montgomery multiplication

INPUT: integer *m* with *n* radix *b* digits and gcd(*m*, *b*) = 1, $R = b^n$, positive integer *x and y* with *n* radix *b* digits and $x < m$.

OUTPUT: $x*y*R^{-1} \bmod m$.

```
1.  for (i from 0 up to s-1)
2.      C: = 0
3.      for ( j from 0 up to s-1)
4.          (C,S) := t[j] + a[j]*b[i] + C
5.          t[j] := S
```

```
6.      end for
7.      (C,S) := t[s] + C
8.      t[s] := S
9.      t[s+1] := C
10.     C := 0
11.     m := t[0]*n'[0] mod W
12.     for (j from 0 up to s-1)
13.        (C,S) := t[j] + m*n[j] + C
14.        t[j] := S
15.     end for
16.     (C,S) := t[s] + C
17.     t[s] := S
18.     t[s+1] := t[s+1] + C
19.     for (j from 0 up to s)
20.        t[j] := t[j+1]
21.     end for
22. end for
```

Note that the array t is assumed to be set to 0 initially. The last j-loop is used to shift the result one word to the right (i.e., division by 2^w), hence the references to *t[j]* and *t[0]* instead of *t[i+j]* and *t[i]*. A slight improvement is to integrate the shifting into the reduction as follows:

```
m := t[0]*n'[0] mod W
(C,S) := t[0] + m*n[0]
for (j from 1 up to s-1)
   (C,S) := t[j] + m*n[j] + C
   t[j-1] := S
end for
(C,S) := t[s] + C
t[s-1] := S
t[s] := t[s+1] + C
```

The auxiliary array *t* uses only *s* + 2 words. This is due to the fact that the shifting is performed one word at a time, rather than *s* words at once, saving *s* - 1 words. The final result is in the first *s*+1 words of array *t*. A related method, without the shifting of the array (and hence with a larger memory requirement), is described in [2].

The CIOS method (with the slight improvement above) requires $2s^2+s$ multiplications, $4s^2+4s+2$ additions, $6s^2+7s+2$ reads, and $2s^2+5s+1$ writes, including the final multi-precision subtraction, and uses $s+3$ words of memory space. The memory reduction is a significant improvement over the SOS method.

We say that the integration in this method is "coarse" because it alternates between iterations of the outer loop. In the next method, we will alternate between iterations of the inner loop [24].

3.2 Karatsuba Montgomery Reduction

Modular multiplication can be spared into two parts, one is multiplication and one is modular reduction. In this method, we choose the Karatsuba multiplication to implement the multiplication, and then perform Montgomery reduction.

Algorithm 4 Multiple-precision Karatsuba and Montgomery Multiplication

INPUT: integer *m* with *n* radix *b* digits and gcd(*m*, *b*) = 1, $R = b^n$, positive integer *x and y* with *n* radix *b* digits and $x < m$.

OUTPUT: $x*y*R^{-1} \bmod m$.

```
1.  Karatsuba(x,y)
2.  for (i from 0 up to s-1)
3.     C := 0
4.     m := t[i]*n'[0] mod W
5.     for (j from 0 up to s-1)
6.        (C,S) := t[i+j] + m*n[j] + C
7.        t[i+j] := S
8.     end for
9.     ADD (t[i+s],C)
10. end for
11. for (j from 0 up to s)
12.    u[j] := t[j+s]
13. end for
14. B := 0
15. for (i from 0 up to s-1)
16.    (B,D) := u[i] - n[i] - B
17.    t[i] := D
18. end for
19. (B,D) := u[s] - B
20. t[s] := D
21. if B=0 then return t[0], t[1], ... , t[s-1]
22. else return u[0], u[1], ... , u[s-1]
```

4. Improving the Montgomery Multiplication

To achieve a high performance, we analyze the implementation of the instructions of the Montgomery Multiplication. We use the decuda tool to disassemble the cubin file of the CUDA binary codes. We will tell the details as below.

4.1 ASM of Integer Multiplication

When we use the decuda tool disassemble the cubin. We get that the 32bit multiplication 32bit integer is not only using one instruction but using ten to twenty instructions.

We can see the MULT64X64LO need more than 20 instructions, but the MULT32X32WIDE only need 10 instructions. So, we use inline ASM to limit the compiler

compile the 32bit multiplication 32bit work with the 10 instructions. eg.:

Algorithm 5 32bit integer multiplication

INPUT: 32bit integer *A* multiplicative with 32bit integer *B*.

OUTPUT: *A*B*.

```
1.  static inline __device__ unsigned __int64
    mul_32x32(unsigned A, unsigned B) {
2.    unsigned __int64 out;
3.    asm("mul.wide.u32  %0, %1, %2;" : "=l"(out) : "r"(A),
    "r"(B));
4.    return out;
5.  }
```

5. Implementation and Experimental Results

We tested these implementations on T61 NVIDIA Quadro NVS140M graphic card which contains an NVIDIA Quadro G86M GPU. The G86M GPU uses the G80 architecture with 16 processing cores working at 0.8 GHz. The implementation of modular multiplication is in 1024-bit integer. All the implementation is 1024-bit multiplied by 1024-bit modular 1024-bit.

5.1 Comparing Karatsuba Method and CIOS Method

In Fig. 1, the X-coordinate represents the number of threads. As Fig. 1 shows, the karatsuba Montgomery multiplication is slow than the CIOS method of Montgomery multiplication. The karatsuba Montgomery multiplication method needs 60 registers and 5132 local memories. But the CIOS method only needs 14 register and no local memory at all. The blue block shows the CIOS method of the Montgomery Multiplication. The red block shows the method with K-MM (Karatsuba Montgomery Multiplication). The K-MM method needs more variables in the implementation, because that can't use callback function in the CUDA model recently, so the 1024bit integer multiplication will be translate into 2 256bit integer function, and 256bit to 128bit, and so on. So the K-MM method is slow than the CIOS method on the recently GPU model.

The CIOS method only needs 14 registers and using unrolling method unroll the loop.

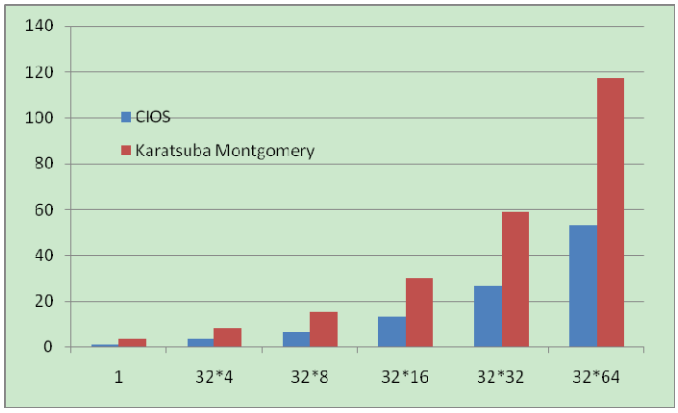

Figure 1. Throughput of Montgomery multiplication and karatsuba Montgomery multiplication on GPU using Algorithm 3&4, with different number of threads.

5.2 Montgomery Multiplication with inline ASM

We use the inline ASM to limit the instruction work with the Short instruction. The experimental results are shown in Fig. 2. The performance has been improved by 20% as compared with the one without using our inline ASM code.

The blue and red block all use the CIOS method. But the red one, as the Fig. 2 shows is 20% faster than blue one. Because the inside ASM function used to solve the 32bit multiplicative 32bit integer. In the decuda code we can see that each loop the CIOS-ASM method is 11 instructions little than the CIOS method.

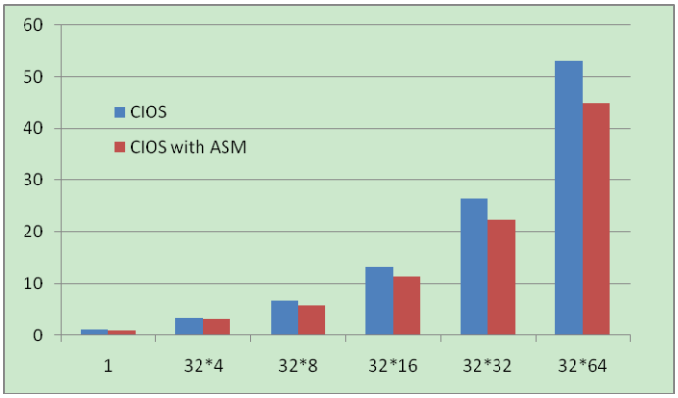

Figure 2. Throughput of inline ASM in CIOS on GPU, with different number of threads

6. Conclusions

Multiple-precision modular multiplication is an important component in public-key cryptography for encrypting and signing digital data. In this paper, we describe the design, implementation and optimization of multiple-precision modular multiplication using GPU and CUDA. Although Karasuba multiplication is theoretically advantageous, we

found that it is not practical for the current GPU platform due to the high cost of comparing large integers. Nevertheless, we improved the performance of the CIOS method by developing an inline ASM implementation of 32-bit integer multiplication.

7. References

[1] NVIDIA CUDA. http://developer.nvidia.com/object/cuda.html

[2] AMD CTM Guide: Technical Reference Manual. 2006. http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf

[3] GNU MP Arithmetic Library. http://gmplib.org/

[4] NVIDIA CUDA Compute Unified Device Architecture: Programming Guide, Version 2.0beta2, Jun. 2008.

[5] Montgomery, P., 1985. Multiplication without trial division, Math. Computation, vol. 44, 1985, 519-521.

[6] Menezes, A., van Oorshot, P., and Vanstone S., 1996. Handbook of applied cryptography. CRC Press, 1996.

[7] Ahlswede, R., Cai, N., Li S. R., and Yeung, R. W. 2000. Network information flow. IEEE Transactions on Information Theory, 46(4), July 2000, 1204-1216.

[8] Ho, T., Koetter. R., Médard, M., Karger, D.R. and Effros, M. 2003. The benefits of coding over routing in a randomized setting. In Proceedings of IEEE ISIT, 2003.

[9] Li, S.-Y.R., Yueng, R.W., and Cai, N. 2003. Linear network coding. IEEE Transactions on Information Theory, vol. 49, 2003. 371-381.

[10] Krohn, M., FreedMan, M., and Mazieres, D. 2004. On-the-fly verification of rateless erasure codes for efficient content distribution. In Proceedings of IEEE Symposium on Security and Privacy, Berkeley, CA, 2004.

[11] Gkantsidis, C. and Rodriguez, P. 2005. Network coding for large scale content distribution. In Proceedings of IEEE INFOCOM 2005.

[12] Gkantsidis, C. and Rodriguez, P. 2006. Cooperative security for network coding file distribution. In Proceedings of IEEE INFOCOM'06, 2006.

[13] Li, Q., Chiu, D.-M., and Lui, J. C.S. 2006. On the practical and security issues of batch content distribution via network coding. In Proceedings of IEEE ICNP'06, 2006, 158-167.

[14] Chou, P. A. and Wu, Y. 2007. Network coding for the Internet and wireless networks. Technical Report. MSR-TR-2007-70, Microsoft Research.

[15] Wang, M. and Li, B. 2007. Lava: a reality check of network coding in peer-to-peer live streaming. In Proceedings of IEEE INFOCOM'07, 2007.

[16] Wang, M. and Li, B. 2007. R^2: random push with random network coding in live peer-to-peer streaming. In IEEE Journal on Selected Areas in Communications, Dec. 2007, 1655-1666.

[17] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W. 2008. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In Proceedings of ACM PPoPP'08, Feb. 2008.

[18] Falcao, G., Sousa, L., and Silva, V. 2008. Massiv parallel LDPC decoding in GPU. In Proceedings of ACM PPoPP'08, Feb. 2008.

[19] Yu, Z., Wei, Y., Ramkumar, B., and Guan, Y. 2008. An efficient signature-based scheme for securing network coding against pollution attacks. In Proceedings of IEEE INFOCOM'08, Apr. 2008.

[20] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. 2008. GPU computing. IEEE Proceedings, May 2008, 879-899.

[21] Al-Kiswany, S., Gharaibeh, A., Santos-Neto, E., Yuan, G., and Ripeanu, M. 2008. StoreGPU: exploiting graphics processing units to accelerate distributed storage systems. In Proceedings of IEEE Symposium on High Performance Distributed Computing (HPDC), Jun. 2008.

[22] Silberstein, M., Geiger, D., Schuster, A., Patney, A., Owens, J. D. 2008. Efficient computation of sum-products on GPUs through software-managed cache. In Proceedings of the 22nd ACM International Conference on Supercomputing, Jun. 2008.

[23] Seiler, L., et. al., 2008. Larrabee: a many-core x86 architecture for visual computing. ACM Transactions on Graphics, 27(3), Aug. 2008.

[24] Cetin Kaya Koc and Tolga Acar， Burton S. Kaliski Jr， Analyzing and Comparing Montgomery Multiplication Algorithms, IEEE Micro, 16(3):26-33, June 1996

[25] A. Karatsuba and Yu. Ofman (1962). "Multiplication of Many-Digital Numbers by Automatic Computers". In Proceedings of the USSR Academy of Sciences 145: 293–294.