GPUMP: a Multiple-Precision Integer Library for GPUs

Kaiyong Zhao and Xiaowen Chu
Department of Computer Science, Hong Kong Baptist University
Hong Kong, P. R. China
Email: {kyzhao, chxw}@comp.hkbu.edu.hk

Abstract— **Multiple-precision integer operations are key components of many security applications; but unfortunately they are computationally expensive on contemporary CPUs. In this paper, we present our design and implementation of a multiple-precision integer library for GPUs which is implemented by CUDA. We report our experimental results which show that a significant speedup can be achieved by GPUs as compared with the GNU MP library on CPUs.**

Keywords: Multiple-precision algorithm, GPU, CUDA

I. INTRODUCTION

Public-key encryption plays a critical role in our daily life. The core component of a public-key system is a set of multiple-precision integer operations. A server that relies on public-key encryption (such as an SSL server) needs to process a large number of multiple-precision integer operations, which require huge computing power.

Recent advances in Graphics Processing Units (GPUs) open a new era of GPU computing. 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. 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.

We are motivated by the fact that GPUs could be utilized to speed up multiple-precision integer operations. This is of practical importance to end users as well as application servers. However, it is not easy to achieve high performance on GPUs due to the complicated memory architecture and the relatively slow integer operations[1]. In this paper, we present our design, implementation, and experimental results on a highly optimized multiple-precision integer library. Our library achieved a significant speedup for a number of multiple-precision integer operations.

The rest of the paper is organized as follows. Section II provides background information on GPU architecture and CUDA programming model. Section III presents our design and implementation of multiple-precision integer arithmetic on GPU. Experimental results are presented in Section IV, and we conclude the paper in Section V.

[1] The latest Fermi architecture has a better support on integer operations, but it is out of the scope of this paper.

II. BACKGROUND AND RELATED WORK

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, many core 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).

NVIDIA provided a general-purpose parallel programming model, namely Compute Unified Device Architecture (CUDA) [1] [2], 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 [3]. Intel also exposed Larrabee, a new many-core GPU architecture specifically designed for the market of GPU computing this year [4].

Since the release of CUDA, it has been used for speeding up a large number of applications [8-12]. Given its popularity, we choose CUDA to implement our multiple-precision integer library.

III. MULTIPLE-PRECISION MODULAR ARITHMETIC

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

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

- Multiple-precision comparison
- Multiple-precision addition and subtraction
- Multiple-precision modular addition and subtraction
- Multiple-precision multiplication and division
- Multiple-precision Montgomery reduction
- Multiple-precision Montgomery multiplication
- Multiple-precision exponentiation

Our library implements each operation as a single thread. To make full usage of a GPU, hundreds to thousands of threads are required to be executed simultaneously. It is also possible to implement a complicated operation by multithreading, e.g., a block of threads could be used to perform a single operation such as exponentiation. We leave this as our future work.

A. Comparison, Addition and Subtraction

The pseudo codes of multiple-precision comparison, addition, and subtraction operations are shown in Algorithm 1, 2, and 3, respectively.

Algorithm 1 Multiple-precision Comparison

INPUT: non-negative integers x and y, each with $n+1$ radix b digits.
OUTPUT: 1, if $x > y$; 0, if $x = y$; -1, if $x < y$.

```
1: i ← n ;
2: while ( x_i == y_i and i > 0 )
3:     i ← i − 1;
4: end while
5: if ( x_i > y_i ) then return 1;
6: else if ( x_i == y_i ) then return 0;
7: else return -1;
```

Algorithm 2 Multiple-precision Addition

INPUT: non-negative integers x and y, each with $n+1$ radix b digits.
OUTPUT: $x + y = (z_n z_{n-1} \cdots z_1 z_0)_b$.

```
1: c ← 0 ; /* carry digit */
2: for ( i from 0 to n ) do
3:     z_i ← (x_i + y_i + c) mod b;
4:     c ← (x_i + y_i + c) ÷ b ;
6: end for
7: z_{n+1} ← c ;
8: return (z_n z_{n-1} ··· z_1 z_0)_b ;
```

Algorithm 3 Multiple-precision Subtraction

INPUT: non-negative integers x and y, each with $n+1$ radix b digits, $x \geq y$.
OUTPUT: $x - y = (z_n z_{n-1} \cdots z_1 z_0)_b$.

```
1: c ← 0 ; /* carry digit */
2: for ( i from 0 to n ) do
3:     z_i ← (x_i − y_i + c) mod b;
4:     if ( x_i − y_i + c ≥ 0 ) then c ← 0 ;
5:     else c ← −1 ;
6: end for
7: return (z_n z_{n-1} ··· z_1 z_0)_b ;
```

B. Modular Addition and Subtraction

The pseudo codes of multiple-precision modular addition and subtraction operations are shown in Algorithm 4 and 5, respectively.

Algorithm 4 Multiple-precision Modular Addition

INPUT: non-negative integers x and y, each with $n+1$ radix b digits, $x < m, y < m$.
OUTPUT: $(x+y) \bmod m = (z_n z_{n-1} \cdots z_1 z_0)_b$.

```
1: c ← 0 ; /* carry digit */
2: for ( i from 0 to n ) do
3:     z_i ← (x_i + y_i + c) mod b;
4:     if ( x_i + y_i + c < b ) then c ← 0 ;
5:     else c ← 1 ;
6: end for
7: z_{n+1} ← c ;  m_{n+1} ← 0 ;
8: if ( (z_{n+1} z_n z_{n-1} ··· z_1 z_0)_b >= (m_{n+1} m_n m_{n-1} ··· m_1 m_0)_b ) then
9:     (t_{n+1} t_n t_{n-1} ··· t_1 t_0)_b ← (z_{n+1} z_n z_{n-1} ··· z_1 z_0)_b −
                                             (m_{n+1} m_n m_{n-1} ··· m_1 m_0)_b ;
10:    return (t_n t_{n-1} ··· t_1 t_0)_b ;
11: else return (z_n z_{n-1} ··· z_1 z_0)_b ;
```

Algorithm 5 Multiple-precision Modular Subtraction

INPUT: non-negative integers x and y, each with $n+1$ radix b digits, $x < m, y < m$.
OUTPUT: $(x-y) \bmod m = (z_n z_{n-1} \cdots z_1 z_0)_b$.

```
1: if ( x >= y ) then return x − y ;
2: else
3:     t ← (m − y);
4:     return (x + t) mod m;
5: end else
```

C. Multiplication, Division, and Modular Multiplication

One straightforward method to implement modular multiplication of $x \cdot y$ mod m is to calculate $x \cdot y$ first and then calculate the remainder of $x \cdot y$ divided by m. Hence modular multiplication can be implemented by using multiplication and division operations. Next, we give the pseudocode for calculating multiple-precision multiplication and division in Algorithm 6 and 7, respectively.

Algorithm 6 Multiple-precision Multiplication

INPUT: non-negative integers x and y, each with $n+1$ radix b digits and $s+1$ radix b digits respectively.
OUTPUT: $x \cdot y = (z_{n+s+1} z_{n+s} \cdots z_1 z_0)_b$.

```
1: for ( i from 0 to n+s+1 ) do
2:     z_i ← 0 ;
3: end for
4: for ( i from 0 to s ) do
5:     c ← 0 ; /* carry digit */
6:     for ( j from 0 to n ) do
7:         (uv)_b ← z_{i+j} + x_j · y_i + c ;
8:         z_{i+j} ← v ; c ← u ;
9:     end for
10:    z_{n+i+1} ← u ;
11: end for
12: return (z_{n+s+1} z_{n+s} ··· z_1 z_0)_b ;
```

Algorithm 7 Multiple-precision Division

INPUT: non-negative integers x and y, each with $n+1$ radix b digits and $s+1$ radix b digits respectively, $n \geq s \geq 1$, $y_s \neq 0$.

OUTPUT: the quotient $q = (q_{n-s} \cdots q_1 q_0)_b$ and remainder $r = (r_s \cdots r_1 r_0)_b$ such that $x = q \cdot y + r$, $0 \leq r < y$.

1: **for** (i **from** 0 **to** $n-s$) **do**
2: $q_i \leftarrow 0$;
3: **end for**
4: **while** ($x \geq y \cdot b^{n-s}$) **do**
5: $q_{n-s} \leftarrow q_{n-s} + 1$;
6: $x \leftarrow x - y \cdot b^{n-s}$;
7: **end while**
8: **for** (i **from** n **down to** $t+1$) **do**
9: **if** ($x_i == y_s$) **then** $q_{i-s-1} \leftarrow b-1$;
10: **else** $q_{i-s-1} \leftarrow \lfloor (x_i \cdot b + x_{i-1}) / y_s \rfloor$;
11: **while** $(q_{i-s-1} \cdot (y_s \cdot b + y_{s-1}) > x_i \cdot b^2 + x_{i-1} \cdot b + x_{i-2})$ **do**
12: $q_{i-s-1} \leftarrow q_{i-s-1} - 1$;
13: **end while**
14: $x \leftarrow x - q_{i-s-1} \cdot y \cdot b^{i-s-1}$;
15: **if** ($x < 0$) **then**
16: $x \leftarrow x + y \cdot b^{i-s-1}$;
17: $q_{i-s-1} \leftarrow q_{i-s-1} - 1$;
18: **end if**
19: **end for**
20: $r \leftarrow x$;
21: **return** (q, r);

The classical modular multiplication is suitable for normal operations. However, when performing modular exponentiations, Montgomery multiplication shows much better performance advantage [6]. Montgomery multiplication makes uses of Montgomery reduction. Hence the following gives the pseudocode of Montgomery reduction and Montgomery multiplication in Algorithm 8 and 9 respectively.

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

Algorithm 8 Multiple-precision Montgomery Reduction

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

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

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

Algorithm 9 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, $R = b^n$, $m' = -m^{-1}$ mod b.

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

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

D. Modular Exponentiation

Modular exponentiation has found a lot of application [7]. There are different ways to implement modular exponentiation. We choose to implement the Montgomery exponentiation because it avoids using division operations which are very inefficient in GPUs. The pseudocode of Montgomery exponentiation is shown in Algorithm 10.

Algorithm 10 Multiple-precision Montgomery Exponentiation

INPUT: integer m with n radix b digits and gcd(m, b) = 1, $R = b^n$, positive integer x with n radix b digits and $x < m$, and positive integer e = $(e_t \cdots e_0)_2$.

OUTPUT: x^e mod m.

1: $\tilde{x} \leftarrow Mont(x, R^2 \ mod \ m)$;
2: $A \leftarrow R$ mod m;
3: **for** (i **from** n **down to** 0)
4: $A \leftarrow Mont(A, A)$;
5: **if** $e_i == 1$ **then** $A \leftarrow Mont(A, \tilde{x})$;
6: **end for**
7: $A \leftarrow Mont(A, 1)$;
8: **return** A;

IV. Implementation and Experimental Result

In this section, we first briefly discuss the data structure of multiple-precision (MP) integer and optimization techniques used by our library, and then report our experimental results. More details can be found in [13].

A. Data Structure of Multiple-precision Integer

We represent a MP integer as a sequence of 32-bit integers, since most GPUs support 32-bit integer operations. There are two ways to arrange this sequence of 32-bit integers in memory. One is to put the data of a MP integer in an array. Then a group of MP integers will be stored as a two-dimensional array. The second way is to transpose the two-dimensional array described previously, so that each MP integer is stored in a column instead of a row. This is to achieve coalesced memory access on GPUs.

In our implementation, a group of MP integers are organized in two parts. The first one is an array, which keeps the length of each MP integer. The second part is a matrix. Suppose the number of MP integer is n, the maximum length of the MP integer is l, the set of MP integers could be regarded as $matrix[(n/w)\times l][w]$, in which w is the number of columns.

B. *Optimization Techniques*

- *Using Constant Value with Cache Memory*

Most algorithms will use the same data multiple times during the calculations. Under these cases, the utilization of memory via cache mechanism can increase the calculation efficiency. On GPUs, texture and constant memory adopt cache mechanism. Thus, those frequently accessed data can be kept in texture or constant memory in order to achieve high reading efficiency.

- *Using Shared Memory for Temp Value*

From the algorithms listed in Section III, we notice that some algorithms (Algorithm 4 to 10) need to use temporary variables. Using local or global memory to store these variables will cause long reading latency. But if we use shared memory, the reading latency can become much shorter. Hence, we adopt shared memory to store the temporary variables as much as possible.

- *Balancing the Computing Resource*

In CUDA programming model, the number of registers and shared memory is limited in a single SM (Stream Multiple-processor), which only can make 8 blocks be active simultaneously. Consequently, in order to maximize the number of threads running in a single SM, we need to reasonably manage the number of registers and shared memory in each block.

C. *Experimental Results*

We tested our library on XFX GTX280 graphics card. It contains a NVIDIA GT200 GPU which has 240 processing cores working at 1.24 GHz. We also give the results of GNU MP library running on an i7 CPU (2.80GHz) for comparison.

In the following figures (Figure 1 to 8), the x-axis denotes the number of multiple-precision integers, and the y-axis denotes the achieved number of operations per second. Figure 1 to 6 and 9 respectively represent the operations per second results about the addition, subtraction, multiplication, division, modular addition, modular subtraction in GPU MP library running on GPU and CPU. In order to guarantee GPU running with full load, we select five groups of data, and each group contains 1920, 3840, 7680, 15360, and 30720 multiple-precision integers, respectively. In each group, we select multiple-precision integers with three different lengths, including 512-bit, 1024-bit and 2048-bit. Figure 7 and 8 list the results about Montgomery reduction and Montgomery multiplication algorithm. Since GNU MP library has no individual algorithm about Montgomery reduction and Montgomery multiplication, we only presents our results on GPU.

All results show that the GPU MP library can achieve significant speedup on GPU, far better than the GNU MP library running on CPU.

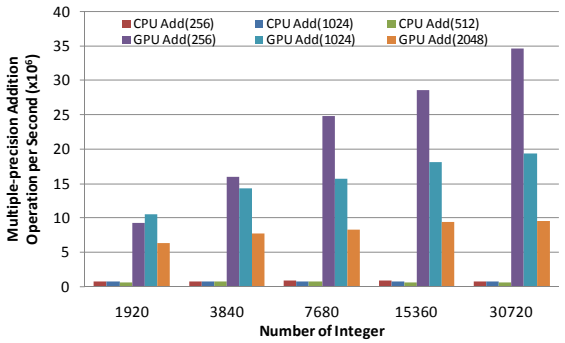

Figure 1. Multiple-precision Addition running on CPU & GPU

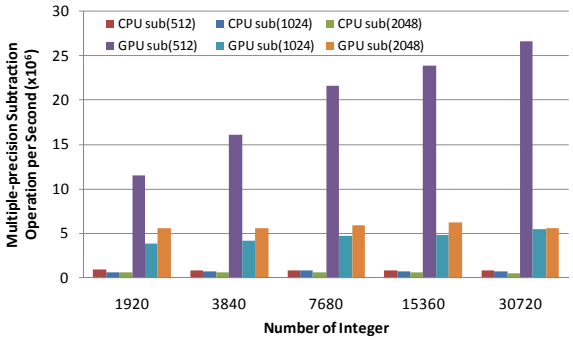

Figure 2. Multiple-precision Subtraction running on CPU & GPU

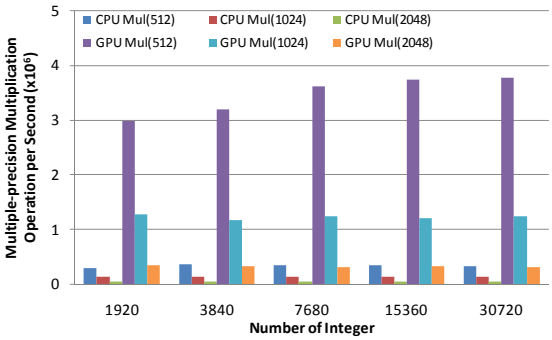

Figure 3. Multiple-precision Multiplication running on CPU & GPU

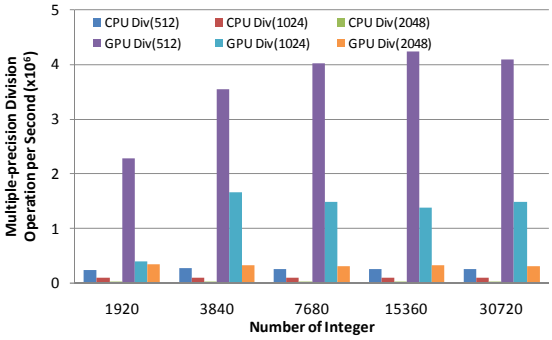

Figure 4. Multiple-precision Division running on CPU & GPU

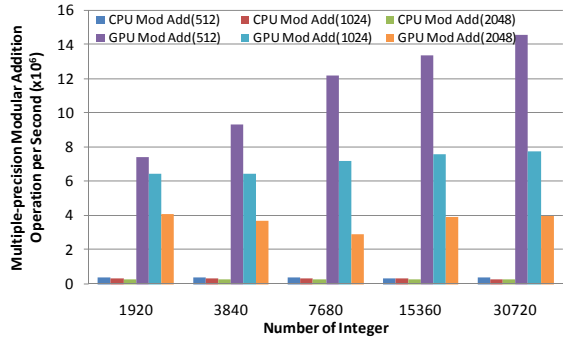

Figure 5. Multiple-precision Modular Addition running on CPU & GPU

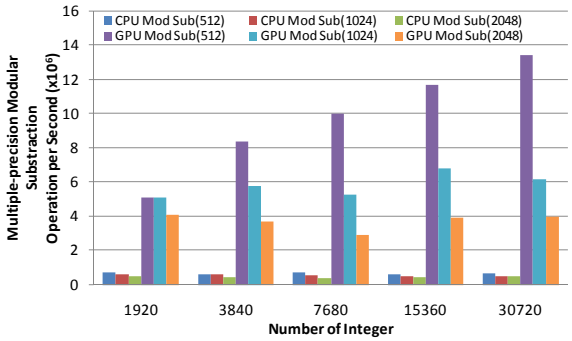

Figure 6. Multiple-precision Modular Subtraction running on CPU & GPU

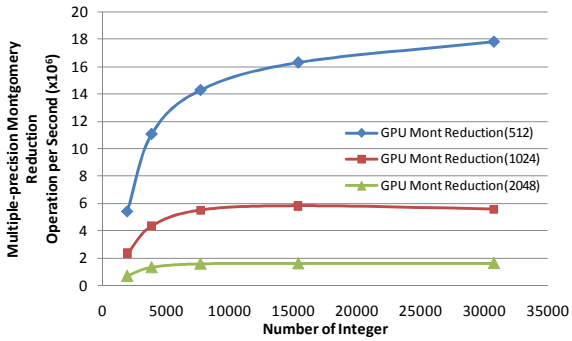

Figure 7. Multiple-precision Montgomery Reduction running on GPU

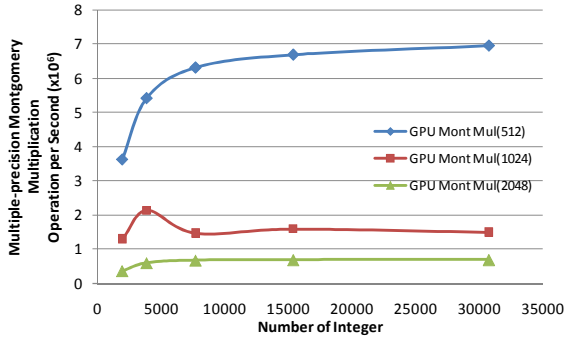

Figure 8. Multiple-precision Montgomery Multiplication running on GPU

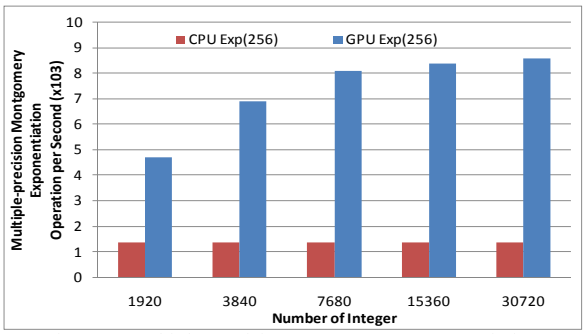

Figure 9. Multiple-precision Montgomery Exponentiation running on CPU & GPU

V. CONCLUSIONS

Multiple-precision integer operations are 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 integer library for GPUs using CUDA. In the future, we will explore how to make use of the new Fermi architecture to further optimize the performance of our library. We will also port our library to OpenCL.

ACKNOWLEDGMENT

This work is supported by FRG Grant frg0809: FRG2/08-09/098 from Hong Kong Baptist University.

REFERENCES

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

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

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

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

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

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

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

[8] 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.

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

[10] 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.

[11] X.-W. Chu, K. Zhao, and M. Wang. Massively Parallel Network Coding on GPUs. In Proceedings of IEEE IPCCC'08, Austin, Texas, USA, Dec 2008.

[12] X.-W. Chu, K. Zhao, and M. Wang. Practical Random Linear Network Coding on GPUs. In Proceedings of IFIP Networking'09, Archen, Germany, May 2009.

[13] K. Zhao and X.-W. Chu. GPUMP: a Multiple-Precision Integer Library for GPUs. Technical Report, Department of Computer Science, Hong Kong Baptist University, 2010.