Design and Implementation of Multiple-precision Integer Library for GPUs

Kaiyong Zhao
Department of Computer Science
Hong Kong Baptist University
Hong Kong, P.R.C
kyzhao@comp.hkbu.edu.hk

Xiaowen Chu
Department of Computer Science
Hong Kong Baptist University
Hong Kong, P.R.C
chxw@comp.hkbu.edu.hk

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 the previous articles, there are some GPU application to accelerate the multiplication of large numbers and model multiplication. Under normal circumstances, our expression of Multiple-precision numbers are in accordance with the order of memory, but the GPU in the thread access case accessed only on a consolidated to achieve maximum bandwidth.. In this case, the traditional arrangement of large numbers method suited GPU demand, we consider the use of alignment on the way to access memory. Similarly, because we are dealing with the data, and there is no order in terms of special requirements, data encryption, or data encoding, we have the data interpreted as a two-dimensional matrix, the traditional approach is to store large numbers in accordance with lines, here it can follow the column to store large integer numbers, for the data itself is not much conflict. Therefore, under the new access methods, we can achieve higher processing efficiency.

Keywords

Multiple-precision, Big Integer, GPU Computing, CUDA, Memory alignment Access, Multiple-precision algorithm

1. Introduction

Non-symmetric encryption is usually used in the network, and his calculation of the core is the processing of large numbers. Large amounts of data on the network to encode or to perform non-symmetric encryption, you need to use a lot of computation time, processing of large numbers will take up much of time. How to reduce computing time, speed up the encoding and encryption of data in the current environment of great challenges. In particular, the amount of data is particularly large when the calculation of encryption or encoding, particularly time-consuming. The emergence of GPU makes this situation changed. GPU is good at handling large amounts of data work.

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.

While the GPU computing power is high, but our actual test results did not meet the highest performance. But how can we make to play a higher performance GPU now? Analysis we have found that the GPU code, GPU computing power of the performance bottleneck is the memory read section. Our research GPU threading model and memory access model of consolidation of large numbers using the new memory layout model to modify the traditional large numbers in the memory access methods, allowing GPU threads can follow the way to access the memory alignment of data to speed up the memory of the read.

The rest of the paper is organized as follows. Section 2 provides background information on Multiple-precision, GPU architecture, and CUDA programming model， GPU thread model and memory access model. Section 3 presents the design of multiple-precision integer arithmetic on GPU. Section 4 we design a new memory access model for multiple-precision. 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 Multiple-precision, GPU architecture and CUDA programming model.

2.1 Multiple-precision Integer

First, we describe that deal with large integer numbers, why do we need to deal with large numbers. In the non-symmetric encryption, using large numbers more difficult to break down characteristics, to

encrypt the data. The number of larger, more difficult to break greater, and now usually used in large numbers are 1024bit bit. Coding in the network will deal with the situation of large numbers, only the larger situation of large numbers, including the amount of information before it was. For example, we will then be used to make the analysis of an algorithm, a vector is multiplied by a data and then seek mode, the process is the encoding process of handling large numbers. Location of large numbers of data over 1024bit position, then we know the computer inside the existing system is 32 bit position, or the 64bit bit. How to represent the present large numbers now? We know that number can be expressed as $A = x_0 \cdot 10^0 + x_1 \cdot 10^1 + \ldots x_n \cdot 10^n$; such a way that a number of the same, we can also be used inside the computer the same way to represent a large numbers, here we have the b-bit 0x100000000, using 2^{32} as a binary. We define binary bit b, then the large numbers can be expressed as $A = x_0 \cdot b^0 + x_1 \cdot b^1 + \ldots x_n \cdot b^n$. These circumstances, we can express the location of large numbers of more than 32bit. For example, large numbers 1024bit bit, we can use 8-bit integer type of representation is 32bit.

We can see large integer is an expression of the polynomial. So we have large numbers of addition and subtraction multiplication and division rules. In fact, the conversion to polynomial multiplication and division addition and subtraction. In our realization of the large numbers library, we used a way to carry out large numbers polynomial arithmetic. Because large numbers have been a lot of processing algorithms, where we do not deal with large numbers of the algorithm to do too much to explain, but we choose the best and most suitable for GPU algorithms. Here we will discuss what kind of large numbers algorithm is suitable for GPU, for the present CUDA programming. Processing of large numbers, especially multiplication, exponentiation, modular computing, and power-mode operation in practical application a large proportion of, so large numbers in our database, we will focus on the realization of multiplication, power-mode operation.

2.2 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.

Of particular importance is the CUDA model of GPU threads to access the memory model. Under normal circumstances this place will become a bottleneck GPU acceleration. Each thread reads a 32 bit data, put the rules into alignment here. When all the threads in a half-warp (16 threads) read the collocation memory the access will be make one step.

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

Due to the space limitation, we do not present the implementation details in this paper.

3.1 Modular Addition and Subtraction

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: $i \leftarrow n$;
2: **while** ($x_i == y_i$ **and** $i > 0$)
3: $i \leftarrow i - 1$;
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 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: $c \leftarrow 0$; /* carry digit */
2: **for** (i **from** 0 **to** n) **do**
3: $z_i \leftarrow (x_i - y_i + c)$ **mod** b;
4: **if** ($x_i - y_i + c \geq 0$) **then** $c \leftarrow 0$;
5: **else** $c \leftarrow -1$;
6: **end for**
5: **return** $(z_n z_{n-1} \cdots z_1 z_0)_b$;

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

Algorithm 4 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: $t \leftarrow (m - y)$;
4: **return** $(x + t)$ **mod** m;
5: **end else**

Complexity Analysis: Obviously all the above algorithms have computational complexity of $O(n)$.

3.2 Modular Multiplication

One straightforward method to implement modular multiplication of $x \cdot y \bmod m$ is to calculate $x \cdot y$ first and then calculate the remainder of $x \cdot y$ divided by m. Hence we first give two algorithms to calculate multiple-precision multiplication and division respectively.

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

Algorithm 6 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 [7]. The following gives the Montgomery reduction and Montgomery multiplication algorithms.

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 $A \cdot R^{-1}$ mod m. In our applications, R is chosen as b^n to simply the calculation.

Algorithm 7 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** ($T \ge m$) **then** $T \leftarrow T - m$;
8: **return** T;

Algorithm 8 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** ($T \ge m$) **then** $T \leftarrow T - m$;
7: **return** T;

3.3 Barrett Modular Reduction Algorithm

Barrett Reduction is a method of reducing a number modulo another number. Barrett reduction by precomputing some values, one can easily far exceed the speed of normal modular reductions.

Barrett reduction's benefits are most visible when it is used to reduce various numbers modulo a single number many times. Barrett reduction is not particularly useful when used with small numbers (32 or 64 bits); it's benefits occur when using numbers that are implemented by multiple precision arithmetic libraries, such as when implementing the network coding in GF(2^{32}).

Next I will give the implementation of the Barrett reduction algorithm. But first, keep in mind that Barrett Reduction can only reduce numbers that are, at most, twice as long (in words) as the modulus. We define the modulus, called m, which is k words long (numbered k-1...0, with 0 being the least significant word). First we need pre-calculate the value: $u = \lfloor b^{2k} / m \rfloor$ where b is the "base" of the integers used. For example, if you represented the numbers as a sequence of 32-bit values, b is 2^{32}, or 0x100000000. You will keep this value u across function calls so you can reuse it.

Now, given a number x, which is an arbitrary integer of size (at most) 2k words (2k-1...0), this procedure (in pseudocode) will return the value of x mod m:

Algorithm 9 Barrett Modular Reduction Algorithm

INPUT: positive integers $x = (x_{2k-1} \dots x_1 x_0)_b$, $m = (m_{k-1} \dots m_1 m_0)_b$ (with $m_{k-1} \ne 0$), and $u = \lfloor b^{2k} / m \rfloor$.

OUTPUT: $r = x \bmod m$
1: $q_1 = \lfloor x / b^{k-1} \rfloor$, $q_2 = q_1 \cdot u$, $q_3 = \lfloor q_2 / b^{k+1} \rfloor$;
2: $r_1 = x \bmod b^{k+1}$, $r_2 = q_3 \cdot m \bmod b^{k+1}$, $r = r_1 - r_2$;
3: **if** ($r < 0$) **then** $r = r_1 + b^{k+1}$;
4: **while** ($r \ge m$) **do** $r = r_r - m$;
5: **return** r;

Note that the divisions and modular reductions in this procedure can be replaced by right shifts and AND operations because the b is 2^{32}. This results in the remaining operations being addition and multiplication, both of which are much cheaper than division for multiple precision integers.

3.4 Multiplicative Inversion

Traditionally multiplicative inversion is obtained through extended Euclidean algorithm. In order to avoid the expensive multiple-precision division operations, we implement multiple-precision multiplicative inversion using an extended binary GCD algorithm.

Algorithm 10 Multiple-precision Multiplicative Inversion

INPUT: odd prime number m with n radix b digits, positive integer a with n radix b digits, $a < m$.

OUTPUT: integer $b \in Z_m$ such that $a \cdot b \equiv 1 \pmod{m}$.

1: $u \leftarrow m$; $v \leftarrow a$; $B \leftarrow 0$; $D \leftarrow 1$;
2: **while** u is even
3: $u \leftarrow u / 2$;
4: **if** B is even **then** $B \leftarrow B / 2$;
5: **else** $B \leftarrow (B - m) / 2$;
6: **end while**
7: **while** v is even

8: $v \leftarrow v/2$;
9: **if** D is even **then** $D \leftarrow D/2$;
10: **else** $D \leftarrow (D-m)/2$;
11: **end while**
12: **if** ($u \geq v$) **then** $u \leftarrow u-v$; $B \leftarrow B-D$;
13: **else** $v \leftarrow v-u$; $D \leftarrow D-B$;
14: **if** $u == 0$ **then return** D;
15: **else go to** Step 2.

3.5 Modular Exponentiation

Algorithm 11 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: $x^e \bmod m$.

1: $\tilde{x} \leftarrow Mont(x, R^2 \bmod m)$;
2: $A \leftarrow R \bmod m$;
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;

3.6 Exponentiation with Multi-Exponentiation

There exist multi-exponentiation algorithms which perform much better than calculating the exponents individually. The following algorithm is a variation of Straus's algorithm [5], by integrating the Montgomery method. To evaluate $\prod_{i=0}^{k-1} g_i^{e_i}$ where the maximum bit-length of all the exponents is n, we first form a $k \times n$ bit matrix whose rows are the binary representations of e_i, $0 \leq i \leq k-1$. Let C_j be the non-negative integer whose binary representation is the jth column, $1 \leq j \leq n$, of the bit matrix, where the least significant bit is at the top of the column.

Algorithm 12 Montgomery Multi-Exponentiation

INPUT: integers $g_0, g_1, ..., g_{k-1}$, $e_0, e_1, ..., e_{k-1}$, R, and m.

OUTPUT: $\prod_{i=0}^{k-1} g_i^{e_i} \bmod m$.

1: **for** (i **from** 0 **to** $2^k - 1$)
2: $\tilde{G}_i \leftarrow (\prod_{j=0}^{k-1} g_j^{i_j})R \bmod m$, where $i = (i_{k-1} \cdots i_0)_2$
3: **end for**
4: $A \leftarrow R \bmod m$;
5: **for** (j **from** 1 **to** n)
6: $A \leftarrow Mont(A, A)$;
7: **if** $C_j \neq 0$ **then** $A \leftarrow Mont(A, \tilde{G}_{C_i})$;
8: **end for**
9: $A \leftarrow Mont(A, 1)$;
10: **return** A;

The above algorithm requires at most $2^k + 2(n-2)$ multiplication operations, with a memory space of $2^k - 1$. The optimal value of k depends on the value of n, and it is usually small for contemporary cryptography applications.

Given the structure of homomorphic hash function, we can divide the product $h(e) = \prod_{i=1}^{m} g_i^{e_i}$ into groups of multi-exponentiations: $h(e) = \prod_{i=1}^{\lceil m/k \rceil} (\prod_{j=1}^{k} g_{(i-1)k+j}^{e_{(i-1)k+j}})$, and assign each multi-exponentiation to an individual processing core.

3.7 Exponetiation with Precomputation

When applying homomorphic hash function in network coding enabled P2P applications, the same homomorphic hash function, i.e., with the same set of parameters, will be used for a large data set such as a whole file or a video streaming session. Under this special circumstance, it is possible to speed up the modular exponentiations by precomputation [8]. To calculate g^e, we first represent the exponent e using radix $b = 2^k$: $e = \sum_{i=0}^{n-1} a_i b^i$, where $0 \leq a_i < b$ and $a_{n-1} \neq 0$. It is easy to see that $n = \lceil (\lfloor \log_2 e \rfloor + 1)/k \rceil$. The fast modular exponentiation algorithm requires the precomputation of $g^{2^{ki}} \bmod m$ for $1 \leq i \leq n-1$. Then we can use the following algorithm to calculate $g^e \bmod m$.

Algorithm 13 Exponentiation with Precomputation

INPUT: integers m, g, $e = \sum_{i=0}^{m-1} a_i b^i$, R, and $Rg^{2^{ki}} \bmod m$ for $1 \leq i \leq n-1$

OUTPUT: $g^e \bmod m$.

1: $A \leftarrow R$, $B \leftarrow R$;
2: **for** (j **from** $b-1$ **down to** 1)
3: **for** i **from** 0 **to** $m-1$
4: **if** $a_i == j$ **then** $B \leftarrow Mont(B, Rg^{2^{ki}}) \bmod m$;
5: **end for**
6: $A \leftarrow Mont(A, B)$;
7: **end for**
8: $A \leftarrow Mont(A, 1)$;
9: **return** A;

As shown by [8], the above algorithm takes $m + b - 3$ multiplications. For e with 257-bit, the optimal value of b is 16, which takes only 78 multiplications in the worst case, as compared with 512 multiplications required by the binary method.

4. Implementation Multiple-precision Integer library on GPUS

This chapter, we will discuss the use of memory with alignment approach to the implementation of Multiple-precision integer numbers. In the GPU, in order to achieve maximum access to bandwidth, we need a CUDA half-warp access the data inside the data, when, in accordance with coalescing way to access memory.

Next part we will discuss the way, how the half-warp access the memory by coalescing .

4.1 Data Structure of Multiple-precision Integer

There are two ways to organize the data, one way is the CPU data structure, one is suitable for GPU data structures. We define large numbers over more than 64bit or 32bit, so we need to re-definition of the structure of the big integer. It based on 2^{32}, each element of big integer is 32bit, according to 2^{32} as a binary, a total number of bits into the data structure of the last one. In our implementation, each of a large numbers are all composed of two parts, first part is the actual data of large numbers, the second part of the numbers of bits, these two can be separated is placed, it is not released to the same array.

The first way is to put the data in accordance with the habit of CPU, in order, put each number into a 32bit memory inside. Another way to put the data in accordance with GPU appropriate manner, and put 32 large numbers in each row, each a large numbers of data are arranged according to the column.

Our actual calculations will be used simultaneously to both structures, a number of constants, or only once the data, we can put inside GPU computing, or stored in a number of high-performance memory inside, place the data in accordance with the first approach. Be counted some of the data, we will follow for GPU-series data to store data

4.2 Using Constant Value with Cache Memory

In the calculation process, we usually use to duplicate data, or to be multiple uses of data. The data we need to try to put to efficient memory cache inside. In Nvidia's GPU we can use texture and constant memory storage of constants. Texture and the constant has two caches, multiple use of the data will be loaded into the cache inside, so that you can efficiently use the constant data. For example, exponentiation of the base, or are seeking modules modulus.

4.3 Using Shared Memory for Temp Value

We can see that some of the above formula to the process need to use temporary variables. Because the performance of local memory and global memory as well, so this part of the variable we will use shared memory to be stored. All of the intermediate process is the need to reiterate the use of variables. And then passed to a function as a temporary variable to use it.

4.4 Balance the Computing Resource

Balance the number of threads for each block, making a maximum of active threads. CUDA programming model Stream Multiprocessor each one simultaneously active 8 blocks, but the need to registers for each block and the shared memory used in the calculation. The need to co-inside with the 4.3, sometimes without direct use of shared memory as global memory can achieve higher performance

4.5 Example Implementation in GPUs

This part, we design a example for benchmarking this library. There is a vector array A multiplication with big integer matrix B, and then mod 1024bit big integer. Large numbers in the original matrix arranged in memory when the inside is based on a lump sum for each line to represent. A lump sum for each of a lump sum and the next will be the location of 256bit interval, so that the GPU inside the access and causing access can't be merged. This will allow each half-warp access memory produce 16 times the time to visit. Each visit is approximately 500 cycle, if it is can be merged into one visit, then these 16 threads will be merged into one visit, which will reduce the access time period, increasing access to bandwidth.

So in this case, we have designed to store large numbers under the column of the memory layout. Can see the figure below, we show that the yellow unit One-big-integer-256bits, says that a large numbers. So, 16 threads to access the matrix B when the alignment means in accordance with the order would be to visit each of the first data row, so that you can make 16 threads to access the data alignment.

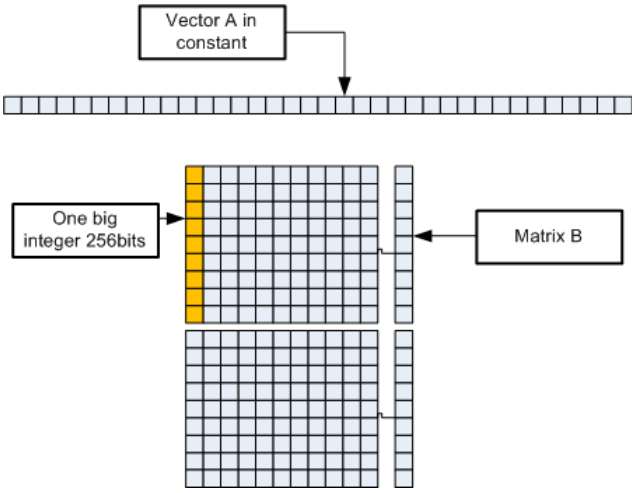

Figure 1. Vector A multiplication with Big integer matrix B

Because the vector A will be used multiple times, so we will put into the vector A into constant memory, because there will be two constant caches, you can speed up the data read

5. Implementation and experimental Result

We have implemented this function using CUDA. We tested these implementations on XFX GTX280 graphic card which contains an NVIDIA GeForce GTX280 GPU. The GTX280 GPU uses the GT200 architecture with 240 processing cores working at 1.24 GHz.

5.1 Result of the Example

Figure 2 shows the results of our CUDA version of network coding encoding with GT200 GPU. Since most of the

computing task is the modular, the throughput is almost independent of the value of m. The encoding speed about 400 MBPs.

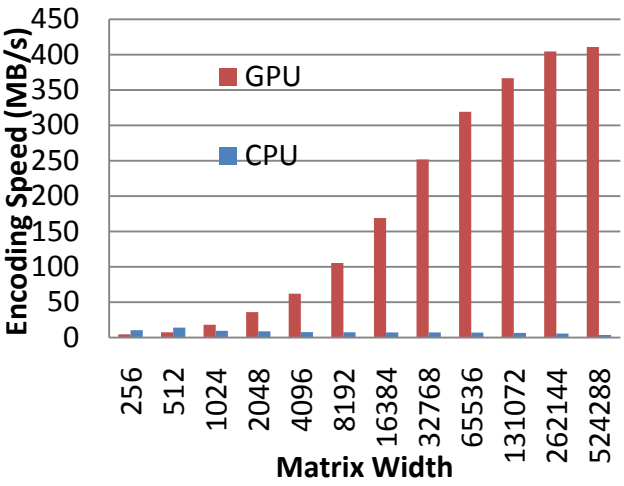

Figure 2. Network Coding Encoding Speed with CUDA

6. Conclusions

Multiple-precision modular 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 using GPU and CUDA. Especially in the case of dealing with large amounts of data, how to optimize the access speed of GPU, we have changed the traditional way of their large numbers of memory, which is more suitable for GPU access the memory data.

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.