A Study of Multiple-length Multiplication on the GPU

(GPUにおける多倍長整数乗算に関する研究)

by

Takumi Honda

A dissertation submitted

in partial fulfillment of the requirements for the degree of

Doctor of Engineering

in Information Engineering

Under Supervision of

Professor Koji Nakano

Department of Information Engineering,

Graduate School of Engineering,

Hiroshima University

Martch, 2017

Summary

A GPU (Graphics Processing Unit) is a hardware specialized for graphics processing. Since GPUs have a lot of cores and very high memory bandwidth, we can accelerate graphics processing using GPUs. Modern GPUs are designed for general purpose computing and can perform computations in various applications. Thus, a GPU attracts a lot of attention as a computational accelerator and many studies have been devoted to implement parallel algorithms using GPUs. The GPU has two types of memories: the global memory and the shared memory. The latency of memory access is longer than that of arithmetic operations. Hence, the efficient usage of the global memory and the shared memory is a key to accelerate applications using GPUs.

Applications require arithmetic operations on integer numbers which exceed the range of processing by a CPU directly is called multiple-length numbers or multiple-length precision numbers and hence, computation of these numbers is called multiple-length arithmetic. More specifically, application involving integer arithmetic operations for multiple-length numbers with size longer than 64 bits cannot be performed directly by conventional 64-bit CPUs, because their instruction supports integers with fixed 64 bits. To execute such application, CPUs need to repeat arithmetic operations for those numbers with fixed 64 bits which increase the execution overhead. Suppose that a multiple-length number is represented by w words, that is, a multiple-length number is $64w$ bits on conventional 64-bit CPUs. The addition of such two numbers can be computed in $O(w)$ time. However, the multiplication generally takes $O(w^2)$ time. Multiple-length multiplication is widely used in various applications such as cryptographic computation, and computational science. Since multiple-length numbers of size thousands to several tens of thousands bits are used in such applications, the accelera-

tion of the computation of their multiplications is in great demand. Also, considering practical cases, a large number of multiplications are usually computed. Therefore, the acceleration of a lot of multiple-length multiplications is necessary. Because of the above background, this dissertation shows a GPU implementation for bulk execution of multiple-length multiplication. In addition, this dissertation shows GPU implementations for exhaustive verification of the Collatz conjecture which needs to perform multiple-length multiplications.

We present a GPU implementation for a large number of multiple-length multiplications. The idea of our GPU implementation is to adopt a warp-synchronous programming technique. We assign each multiple-length multiplication to one warp that consists of 32 threads. In warp-synchronous programming technique, execution of threads in a warp can be synchronized instruction by instruction without any barrier synchronous operations. Also, inter-thread communication can be performed by warp shuffle functions without accessing shared memory. We propose 1024-bit multiple-length multiplication method using warp-synchronous programming technique. Our GPU implementation for 1024-bit multiple-length multiplications runs 52 times faster than the sequential CPU implementation. Moreover, we use this 1024-bit multiplication method for larger size of bits as a sub-routine. The GPU implementation attains a speed-up factor of 21 for 65536-bit multiple-length multiplications.

Consider the following operations on an arbitrary positive number: if the number is even, divide it by two, and if the number is odd, triple it and add one. The Collatz conjecture asserts that, starting from any positive number, repeated iteration of the operations eventually produces the value 1. We propose GPU implementations of exhaustive verification of the Collatz conjecture. Our GPU implementation attains a speed-up fac-

tor of 249 over the sequential CPU implementation. Additionally, the number of the above operations until a number reaches 1 is called delay that is one of the mathematical interests for the Collatz conjecture. Using similar ideas, our GPU implementation counting the delay achieves a speed-up factor of 73.

Contents

1 Introduction **1**

1.1 Background and motivation 1

1.2 Contributions 3

1.2.1 A GPU implementation for a large number of multiple-length multiplications 3

1.2.2 GPU implementations of exhaustive verification of the Collatz conjecture 4

1.3 Dissertation organization 5

2 GPU and CUDA **6**

2.1 CUDA parallel programming model 7

3 A GPU implementation for a large number of multiple-length multiplications **15**

3.1 Related work 16

3.2 Multiple-length multiplication 17

3.2.1 School method 17

3.2.2 Comba method 19

3.2.3 Toom-Cook method 19
3.3 Parallel multiple-length multiplication for the GPU 23
3.3.1 Sum-rotate multiplication 23
3.3.2 Toom-k multiplication with sum-rotate multiplication 27
3.3.3 Optimization of the code for arithmetic 29
3.4 Experimental results 29
3.5 Concluding remarks 36

4 GPU implementations of exhaustive verification of the Collatz conjecture 37
4.1 The Collatz conjecture 37
4.2 Related work 39
4.3 Accelerating the verification of the Collatz conjecture 40
4.3.1 Verification algorithm for the convergence 40
4.3.2 Verification algorithm for the delay 48
4.4 GPU implementation 49
4.4.1 A GPU-CPU cooperative approach 50
4.4.2 Efficient memory access for the GPU memory 51
4.4.3 Optimization of the code for arithmetic with larger integers . . . 52
4.4.4 GPU implementation of the computation for the delay 53
4.5 Performance evaluation 54
4.5.1 Performance for the verification of the convergence 55
4.5.2 Performance for the verification of the delay 58
4.6 Concluding remarks 62

5 Conclusion 63

References 64

Acknowledgment 69

List of publications 70

List of Figures

2.1 CUDA hardware architecture 7

2.2 Hierarchy of thread groups 8

2.3 Coalesced and stride access 9

2.4 The structure of the shared memory 10

2.5 Example of intra-warp data exchange like broadcast using a warp shuffle function 13

2.6 Example of intra-warp data exchange like right circular shift using a warp shuffle function 13

2.7 Example of intra-warp data exchange like left circular shift using a warp shuffle function 14

3.1 The order of word-wise multiplication of School method for multiple-length numbers $C = A \cdot B$ 18

3.2 The order of word-wise multiplication of Comba method for multiple-length numbers $C = A \cdot B$ 21

3.3 Parallel column-based multiplication 25

3.4 Sum-rotate multiplication 25

3.5 The computing time of GPU implementations in milliseconds for 10240 multiple-length multiplications 33

4.1 The data format of 64-bit numbers verified by each thread in a block, where *thread_ID* denotes a thread index within a block, *block_ID* denotes a block index within a kernel, and M is a constant. 52

4.2 The number of verified 64-bit numbers per second for various size of base bit of table S in the GPU implementation for the convergence of the Collatz conjecture 56

4.3 The number of verified 64-bit numbers per second for various size of base bit of table S in the CPU implementation for the convergence . . . 57

4.4 The number of verified 64-bit numbers per second for various size of base bit of table A in the GPU implementation for the delay 60

4.5 The number of verified 64-bit numbers per second for various size of base bit of table A in the CPU implementation for the delay 61

Chapter 1

Introduction

1.1 Background and motivation

Since general processors do not support instructions of integers larger than 64 bits, arithmetic operations of these integers cannot be performed directly. The integer which exceed the range of processing by a CPU directly is called multiple-length number and the arithmetic operations of these numbers are called multiple-length arithmetic. However, in cryptographic computation and computational science, multiple-length arithmetic operations are necessary. If we use r-bit integers to represent a R-bit multiple-length number, $w = \lceil \frac{R}{r} \rceil$ words are necessary and multiple-length operations for such numbers are performed by repearing arithmetic operations with r-bit. Although the computational cost of the addition of such two numbers is $O(w)$, the cost of the multiplication is $O(w^2)$ in general. Therefore, the acceleration of multiple-length multiplication is a key of the acceleration of computations in cryptographic computation and computational science.

Many multiple-length multiplication algorithms are developed. *School method* is the most popular algorithm. This method multiplies the multiplicand by each word

of the multiplier and then add up all the properly shifted results. If we apply School method to multiple-length multiplication $A \cdot B$, where A and B have w words, the computation cost is $O(w^2)$. *Comba method* [4] is a multiplication algorithm which takes into account a characteristic of computers. In general, the latency of memory access is longer than that of arithmetic operations. Hence, Comba method reduces the number of memory access. Although the computation cost of Comba method is also $O(w^2)$, the number of memory access of this method is less than that of School method. *Toom-Cook method* [28] has lower complexity than School and Comba methods. This method is based on the principles of polynomial multiplication and divides input numbers into several smaller numbers to reduce the number of multiplications. As far as we know, *Fürer method* [10] used FFT (Fast Fourier Transform) has lowest complexity among multiplication methods. When input numbers are very large, Fürer method is fastest in multiplication methods. Although the computation cost of School and Comba methods is $O(w^2)$, when input numbers are not quite large, these methods are faster than other methods. Hence, it is important to use the algorithm which is suited to the length of input numbers.

A GPU (Graphics Processing Unit) is a hardware specialized for image processing and has a lot of cores and very high memory bandwidth. Modern GPUs are designed for general purpose computing and can perform various computations traditionally handled by the CPU. NVIDIA provides a parallel computing architecture called CUDA (Compute Unified Device Architecture) [5] for NVIDIA GPUs. Using CUDA, we can develop parallel processing programs to be implemented in GPUs. Since GPUs have a lot of cores and very high memory bandwidth, a GPU can handle multiple tasks simultaneously, the GPU has a high energy efficiency which denotes performance per watt. Thus,

a GPU attracts a lot of attention as a computational accelerator for high performance computing [7, 15, 18, 24, 27, 29]. GPUs are also used to accelerate computations of applications in cryptographic computation and computational science. Since large integers of size thousands to several tens of thousands bits are used in such applications, the acceleration of the computation of their multiplications on GPUs is in great demand. In addition, a large number of multiple-length multiplications with different inputs are performed in cryptographic computation and computational science, e.g. vector multiplication and matrix multiplication. Therefore, in this work, we consider an efficient algorithm for bulk execution of multiple-length multiplication on a single GPU. This dissertation shows GPU implementations for bulk execution of multiple-length multiplication and exhaustive verification of the Collatz conjecture which needs to perform multiple-length multiplications.

1.2 Contributions

1.2.1 A GPU implementation for a large number of multiple-length multiplications

We present a GPU implementation for bulk execution of multiple-length multiplication. The idea of our GPU implementation is to adopt a warp-synchronous programming technique. We assign each multiple-length multiplication to one warp that consists of 32 threads. In parallel processing using multiple threads, usually, it is costly to synchronize execution of threads and communicate within threads. In warp-synchronous programming technique, however, execution of threads in a warp can be synchronized instruction by instruction without any barrier synchronous operations. Also, inter-thread commu-

nication can be performed by warp shuffle functions without accessing shared memory. We propose 1024-bit multiple-length multiplication method using warp-synchronous programming technique. The experimental results show that our GPU implementation on NVIDIA GeForce GTX 980 attains a speed-up factor of 52 for 1024-bit multiple-length multiplications over the sequential CPU implementation. Moreover, we use this 1024-bit multiplication method for larger size of bits as a sub-routine. In addition, we use Toom-Cook method to reduce the number of multiplications. The GPU implementation attains a speed-up factor of 21 for 65536-bit multiple-length multiplications.

1.2.2 GPU implementations of exhaustive verification of the Collatz conjecture

We propose GPU implementations of exhaustive verification of the Collatz conjecture. Consider the following operations on an arbitrary positive number: if the number is even, divide it by two, and if the number is odd, triple it and add one. The Collatz conjecture asserts that, starting from any positive number, repeated iteration of the operations eventually produces the value 1. We use a CPU-GPU cooperative approach, efficient memory access for the GPU memory, and optimization of multiplication to accelerate the verification. We have implemented on NVIDIA GeForce GTX TITAN X and evaluated the performance. The experimental results show that, our GPU implementation can verify 1.31×10^{12} 64-bit numbers per second. While the sequential CPU implementation on Intel Core i7-4790 can verify 5.25×10^{9} 64-bit numbers per second. Thus, our implementation on the GPU attains a speed-up factor of 249 over the sequential CPU implementation. Additionally, we accelerated the computation of counting the number of the above operations until a number reaches 1, called delay, that is one of the

mathematical interests for the Collatz conjecture by the GPU. Using similar ideas, we achieved a speed-up factor of 73.

1.3 Dissertation organization

The doctoral dissertation is organized as follows. In Chapter 2, we show the details of the GPU and CUDA. We present a GPU implementation for bulk execution of multiple-length multiplication in Chapter 3. In Chapter 4, we show GPU implementations of the exhaustive verification of the Collatz conjecture. Finally, Chapter 5 concludes this dissertation.

Chapter 2

GPU and CUDA

NVIDIA provides GPUs which are hardware specialized for image processing. Recent GPUs are designed for general purpose computing. NVIDIA GPUs consist of DRAM (Dynamic Random Access Memory) and *Streaming Multiprocessors* (SMs). Each SM contains many cores and on-chip memory. Hence, it can execute multiple threads in parallel. In addition, GPUs have high memory bandwidth. Therefore, a lot of data can be processed simultaneously by multiple threads on the GPU. Since 2006, NVIDIA has provided a parallel computing architecture, called CUDA, on NVIDIA GPUs. CUDA provides a comprehensive development environment for C and C++. CUDA includes a compiler for NVIDIA GPUs, many libraries and tools for debugging and optimizing of applications. Hence, we can develop parallel algorithms implemented on NVIDIA GPUs using CUDA. Figure 2.1 illustrates the CUDA hardware architecture. CUDA uses three types of memories in the NVIDIA GPUs: *the global memory*, *the shared memory*, and *the register* [21]. The global memory is implemented as an off-chip DRAM of the GPU, and has large capacity, say, 1.5-12 Gbytes, but its access latency is very long. The shared memory is an extremely fast on-chip memory with

lower capacity, say, 16-112 Kbytes. The registers in CUDA are placed on each core in SM and the fastest memory, that is, no latency is necessary. However, the size of the registers is the smallest during them. The efficiency usage of the global memory and the shared memory is a key for CUDA developers to accelerate applications using GPUs.

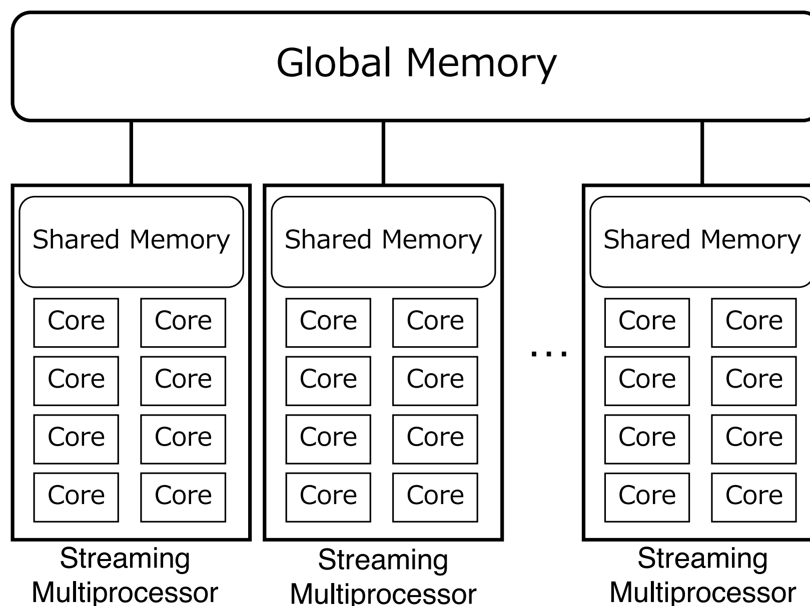

Figure 2.1: CUDA hardware architecture

2.1 CUDA parallel programming model

CUDA parallel programming model has a hierarchy of thread groups, called *grid*, *block*, and *thread* as shown in Figure 2.2. A single grid is organized by multiple blocks, each of which has an equal number of threads. When a program is executed on the GPU, a grid is assigned to a GPU. The blocks are allocated to SMs such that all threads in a block are executed by the same SM in parallel. All threads can access to the global memory. However, as we can see in Figure 2.1, threads in a block can access to the

shared memory of the SM to which the block is allocated. Since blocks are arranged to multiple SMs, threads in different blocks cannot share data in the shared memories.

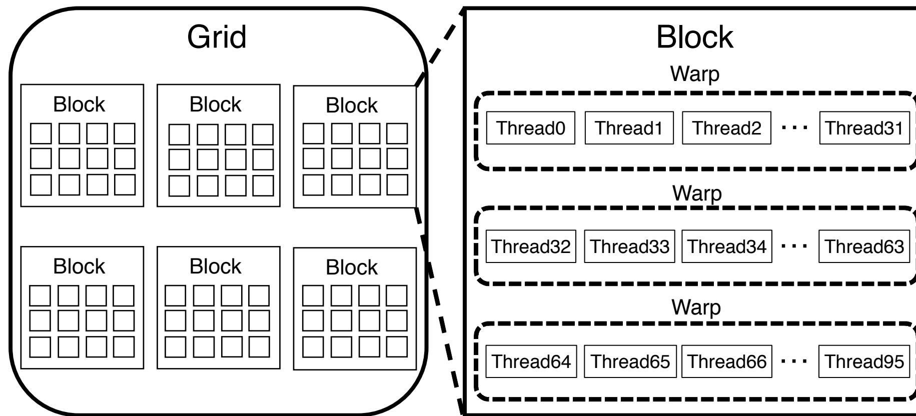

Figure 2.2: Hierarchy of thread groups

The latency of the global memory access is several hundreds clock cycles. Since a thread stalls when one of the operands is not ready, the global memory access tends to stall threads. To accelerate the computation, the coalesced access to the global memory is a key issue. Suppose that there is 2-dimensional array in the global memory. The 2-dimensional array is actually represented as 1-dimensional array. All elements of the array are stored row by row in the global memory. Hence, horizontal neighboring elements are continuous locations in address space. As illustrated in Figure 2.3, when threads access to continuous locations in a row of a 2-dimensional array (*horizontal access*), the continuous locations in address space of the global memory are accessed at the same time (*coalesced access*). The access to the same elements of the array are also coalesced access. However, if threads access to continuous locations in a column (*vertical access*), the distant locations are accessed at the same time (*stride access*). From the structure of the global memory, the coalesced access maximizes the bandwidth of memory access. On the other hand, the stride access needs a lot of clock cycles. Thus,

we should avoid the stride access (or the vertical access) and perform the coalesced access (or the horizontal access) whenever possible.

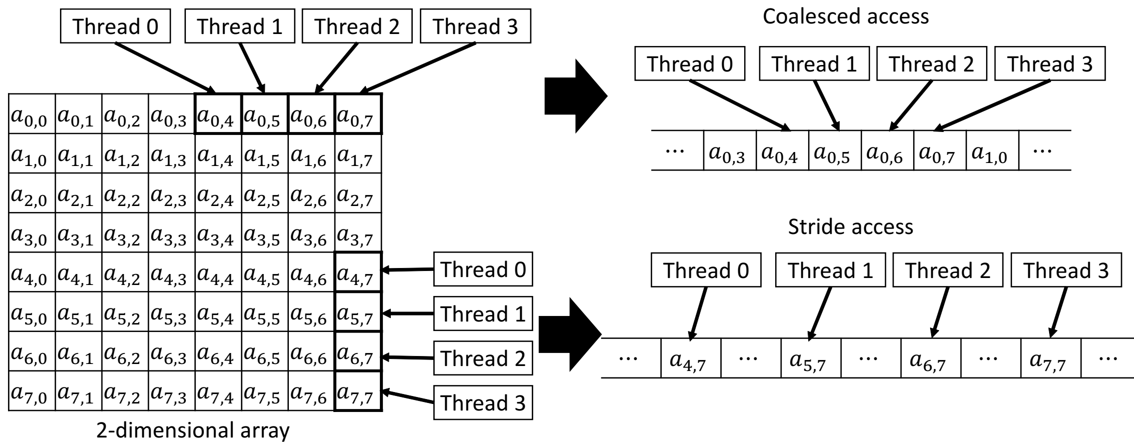

Figure 2.3: Coalesced and stride access

The shared memory is a sort of on-chip memory and which is located within each SM. It has almost no access latency and only visible to the block which is executed by the corresponding SM. The shared memory is divided into 32 equally-sized modules of 32 (or 64)-bit width, called banks (Figure 2.4). In the shared memory, the successive 32 (or 64)-bit words are assigned to successive banks. To achieve maximum throughput, concurrent threads of a block should access different banks. If concurrent threads access to different addresses in the same bank, bank conflicts occur and these memory access are executed sequentially. However, CUDA supports broadcast access. Therefore, when concurrent threads access to the same address in the same bank, bank conflict does not occur. In practice, the shared memory can be used as a cache to hide the access latency of the global memory.

In CUDA, we use CUDA C to implement parallel algorithms on GPUs. CUDA C extends C language by allowing the programmer to define C functions, called *kernels*.

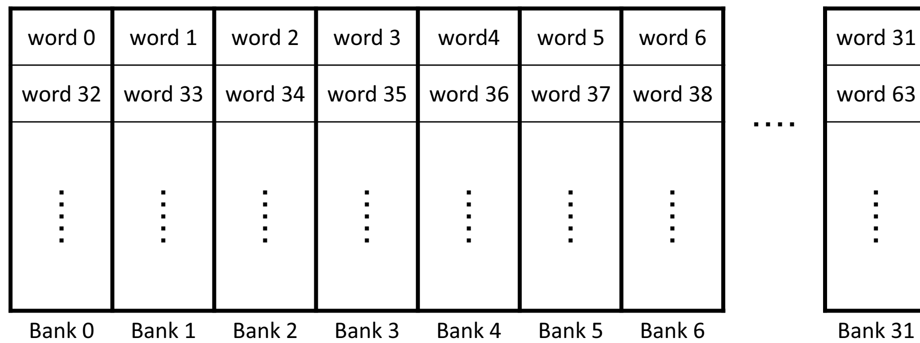

Figure 2.4: The structure of the shared memory

A kernel is a function which is executed on the GPU. By invoking a kernel, all blocks in the grid are allocated in SMs, and threads in each block are executed by processor cores in a single SM. In the execution, threads in a block are split into groups of threads called *warps*. A warp is an implicitly synchronized group of threads. Each of these warps contains the same number of threads and is executed independently. When a warp is selected for execution, all threads execute the same instruction. This characteristic is called SIMT (Single Instruction Multiple Threads). SIMT is an execution model used in parallel computing which combines SIMD (Single Instruction Multiple Data) with multithreading. Any flow control instruction (e.g. if-statements in C language) can significantly impact the effective instruction throughput by causing threads of the same warp to diverge, that is, to follow different execution paths. If this happens, the different execution paths have to be serialized. When all the different execution paths have completed, the threads back to the same execution path. For example, for an if-else statement, if some threads in a warp take the if-clause and others take the else-clause, both clauses are executed in serial. On the other hand, when all threads in a warp branch in the same direction, all threads in a warp take the if-clause, or all take the else-clause. Therefore, to improve the performance, it is important to make branch behavior of all

threads in a warp uniform.

There is a metric, called *occupancy*, related to the number of active warps on an SM. The occupancy is the ratio of the number of active warps per SM to the maximum number of possible active warps. It is important in determining how effectively the hardware is kept busy. There is a measure, called ILP (Instruction-level Parallelism), of how many instructions in a program can be executed simultaneously. If ILP is low, a warp cannot fill instruction pipeline since there is dependence between instructions and a warp frequently stalls. Therefore, to keep hardware busy, the occupancy should be high. If the occupancy is high, when several warps are stalled, other warps can hide latencies. On the other hand, when ILP is high, a single warp can fill instruction pipeline. In this case, it is not necessary that the occupancy is high. The occupancy depends on the numbers of threads and blocks, utilization of the register per thread and the size of the shared memory used in a block. Namely, utilizing too many resources per thread or block may limit the occupancy. There is the trade-off between the occupancy and the utilization of resources per thread and block. To obtain good performance with the GPUs, the occupancy should be considered.

The kernel calls terminate, when threads in all blocks finish the computation. Since all threads in a single block are executed by a single SM, the barrier synchronization of them can be done by calling CUDA C `syncthreads()` function. However, there is no direct way to synchronize threads in different blocks. One of the indirect methods of inter-block barrier synchronization is to partition the computation into kernels. Since continuous kernel calls can be executed such that a kernel is called after all blocks of the previous kernel terminate, execution of blocks is synchronized at the end of kernel calls. On the other hand, all threads of a warp perform the same instruction at the same

time. More specifically, any synchronizing operations are not necessary to synchronize threads within a warp. *Warp-synchronous programming technique* [22] is a parallel programming technique such that one warp is used as an execution unit. The characteristic of this technique is that any synchronous operations are not necessary. Namely, we can parallelize computations without any synchronous operations using warp-synchronous programming technique.

Inter-thread communication is generally performed via shared memory. First, each thread writes data to shared memory. After that, threads read data from shared memory. Therefore, inter-thread communication using shared memory needs memory access. However, in CUDA, *warp shuffle functions* allow the exchange of 32-bit data between threads within a warp, which become available on relatively recent GPUs with compute capability 3.0 and above [21]. Threads in the warp can read other threads' registers without accessing the shared memory. The exchange is performed simultaneously for all threads within the warp. Of particular interest is the `shfl()` function, that is one of the warp shuffle functions. This function takes as parameters a local register variable x and a thread index id. As an example, consider the following function call `shfl`(x,4). The `shfl`(x,4) allows to transfer the data stored in the local register variable x from a thread whose id is 4 (Figure 2.5). This function call corresponds to broadcasting a register variable in a thread to the other threads in a warp. We note that each thread has its own local register x, that is, each x cannot be accessed from other threads. As another example, consider the function call `shfl`(x, $(id + 1)\%w$), where w is the number of threads in a warp. The function call performs data transfer like right circular shift between threads as illustrated in Figure 2.6. In the similar way, the `shfl`(x, $(id + w - 1)\%w$) allows to transfer data like left circular shift (Figure 2.7). The above

data exchange can be performed via shared memory. However, the latency of shared memory access is longer than that of the warp shuffle functions. Since the use of shared memory may cause for decreasing occupancy, if the warp shuffle functions can be used, they should be used.

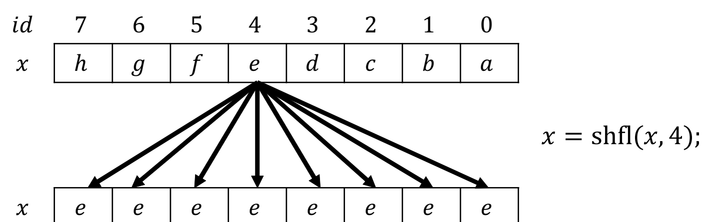

Figure 2.5: Example of intra-warp data exchange like broadcast using a warp shuffle function

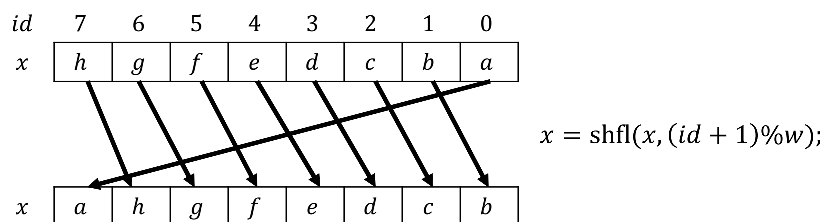

Figure 2.6: Example of intra-warp data exchange like right circular shift using a warp shuffle function

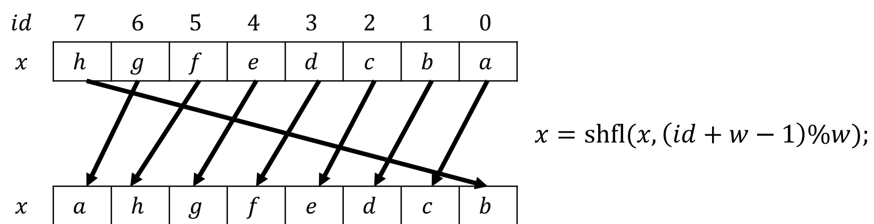

Figure 2.7: Example of intra-warp data exchange like left circular shift using a warp shuffle function

Chapter 3

A GPU implementation for a large number of multiple-length multiplications

In this chapter, we present an implementation of multiple-length multiplication optimized for CUDA-enabled GPUs. The idea of our GPU implementation is to adopt *warp-synchronous programming technique* [22]. We assign each multiple-length multiplication to one warp that consists of 32 threads. In parallel processing using multiple threads, usually, it is costly to synchronize execution of threads and communicate within threads. In warp-synchronous programming technique, however, execution of threads in a warp can be synchronized instruction by instruction without any barrier synchronous operations. Also, inter-thread communication can be performed by warp shuffle functions without accessing shared memory. Using these ideas, we propose a warp synchronous implementation of 1024-bit multiplication on the GPU. In addition, we show multiple-length multiplication methods for more than 1024 bits using the 1024-bit mul-

tiplication method as a sub-routine. To reduce the number of multiplications, we use *Toom-Cook method* [28].

3.1 Related work

There are GPU implementations to accelerate multiple-length multiplications. In papers [8, 3], GPU implementations of very large integer multiplications using FFT are shown. FFT-based multiplication methods have a small time complexity, however, it is efficient for quite large numbers that consists of more than several hundred thousand bits [9]. Zhao *et al.* proposed multiple-length multiplication only for 512 to 2048-bit integers on the GPU as one of library functions [31]. Since this implementation is based on School method that is a naive multiplication method, it is not efficient for more than several thousand-bit numbers. Kitano *et al.* proposed a GPU implementation of parallel multiple-length multiplication also based on School method [16]. In the implementation, load of each thread is equalized by reordering the computation of partial products. However, since warp divergence occurs frequently to perform this reordering, its parallel algorithm is not suitable for GPU architecture. Although several GPU implementations including the above implementations have been proposed, as far as we know, there is no GPU implementation that focuses on bulk execution of multiple-length multiplication. More specifically, many researchers have been devoted to develop and implement parallel algorithms for one input. Although we can obtain outputs for many different inputs by repeating them, their efficiency is not discussed. For example, the aim of the above works [8, 3, 16] is to accelerate the computation of one multiplication for quite large integers by many threads on the GPU. By repeating these methods for many inputs, the bulk execution can be performed. However, there is no research that is premised on the

bulk execution.

3.2 Multiple-length multiplication

In the following, we will represent multiple-length numbers as arrays of r-bit words. In general, $r = 32$ or 64 for conventional CPUs. Let R denote the bit-length of numbers and w be the number of r-bit words. Therefore, $w = \lceil \frac{R}{r} \rceil$. If $r = 32$, a 1024-bit integer consists of 32 ($= \lceil \frac{1024}{32} \rceil$) words. Next, we will introduce several multiplication methods for such multiple-length numbers.

3.2.1 School method

Suppose A and B represent two multiple-length numbers. We are multiplying A by B and the result is stored in C, that is $C = AB$. To compute this multiplication, *School method* is often used. The algorithm of School method is shown in Algorithm 1. For simplicity, in the algorithm, the sizes of the multiplicand and the multiplier are the same and $\{x, y\}$ denotes a concatenation of x and y. School method multiplies the multiplicand by each word of the multiplier and then adds up all the properly shifted results illustrated in Figure 3.1. As illustrated in the figure, calculation of School method is preformed in the row order and some storage needs to be allocated to store intermediate results that are partial products. In School method, intermediate data that are partial products need to be stored to the memory as described at line 6 in Algorithm 1 is necessary.

Algorithm 1 School method

Input: $A = (a_{w-1}, ..., a_1, a_0)$, $B = (b_{w-1}, ..., b_1, b_0)$

Output: $C = AB$

1: $C \leftarrow 0$

2: **for** $j \leftarrow 0$ **to** $w - 1$ **do**

3: $\{u, v\} \leftarrow 0$

4: **for** $i \leftarrow 0$ **to** $w - 1$ **do**

5: $\{u, v\} \leftarrow a_i b_j + c_{i+j} + u$

6: $c_{i+j} \leftarrow v$

7: **end for**

8: $c_{w+j} \leftarrow u$

9: **end for**

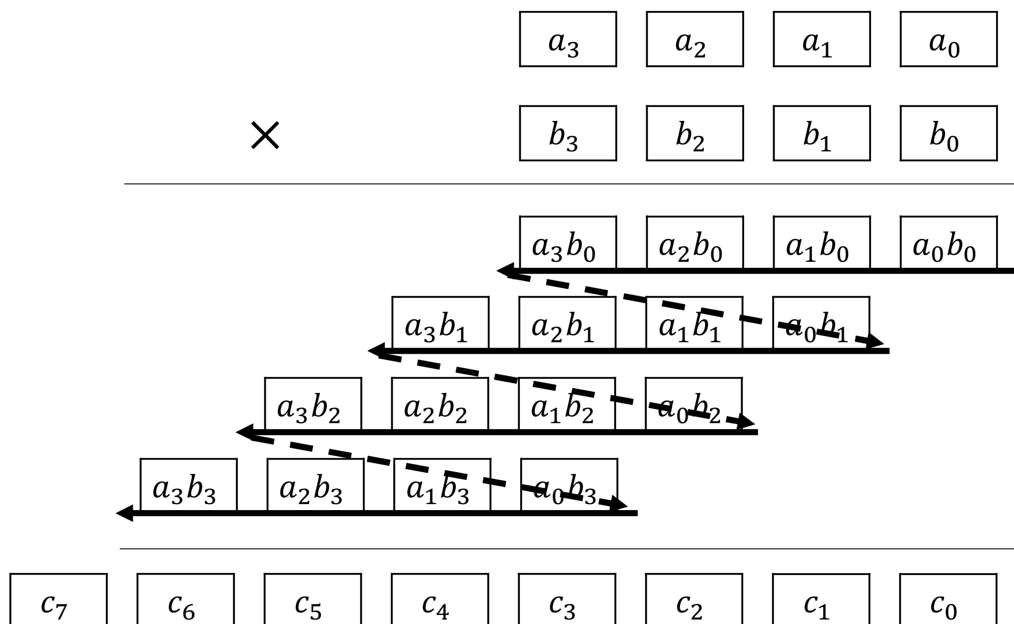

Figure 3.1: The order of word-wise multiplication of School method for multiple-length numbers $C = A \cdot B$

Table 3.1: The number of multiplications and memory read/write for multiplying two w-word numbers using School and Comba methods

method	multiplication	memory read	memory write
School	w^2	$2w^2$	$w^2 + w$
Comba	w^2	$2w^2 - 2$	$2w$

3.2.2 Comba method

To avoid storing the partial products, *Comba method* [4] is used. The algorithm of Comba method is shown in Algorithm 2. According to the algorithm, the readers may think that it is more complicated than School method. However, the difference is only the order of multiplications of words and the number of multiplications of words is the same as illustrated in Figure 3.2. More specifically, calculation of Comba method is preformed in the column order. In Comba method, intermediate data also has to be stored. However, the data corresponds to carry data for the next column. Since the size of the carry data does not depend on the size of numbers and it is only one or two words, its storage can be placed to the register. Table 3.1 shows the number of word-wise multiplications and memory access of School and Comba methods. From the table, the number of memory access, especially memory write, of Comba method is greatly reduced.

3.2.3 Toom-Cook method

Toom-Cook method [28] is an algorithm for multiplying two numbers that reduce the number of multiplications compared with School and Comba methods. Toom-Cook

Algorithm 2 Comba method

Input: $A = (a_{w-1}, ..., a_1, a_0)$, $B = (b_{w-1}, ..., b_1, b_0)$

Output: $C = AB$

1: $\{t, u, v\} \leftarrow 0$
2: **for** $i \leftarrow 0$ **to** $w - 1$ **do**
3:     **for** $j \leftarrow 0$ **to** i **do**
4:         $\{t, u, v\} \leftarrow a_j b_{i-j} + \{t, u, v\}$
5:     **end for**
6:     $c_i \leftarrow v$
7:     $v \leftarrow u, u \leftarrow t, t \leftarrow 0$
8: **end for**
9: **for** $i \leftarrow w$ **to** $2w - 2$ **do**
10:     **for** $j \leftarrow i - w + 1$ **to** $2 - 1$ **do**
11:         $\{t, u, v\} \leftarrow a_j b_{i-j} + \{t, u, v\}$
12:     **end for**
13:     $c_i \leftarrow v$
14:     $v \leftarrow u, u \leftarrow t, t \leftarrow 0$
15: **end for**
16: $c_{2w-1} \leftarrow v$

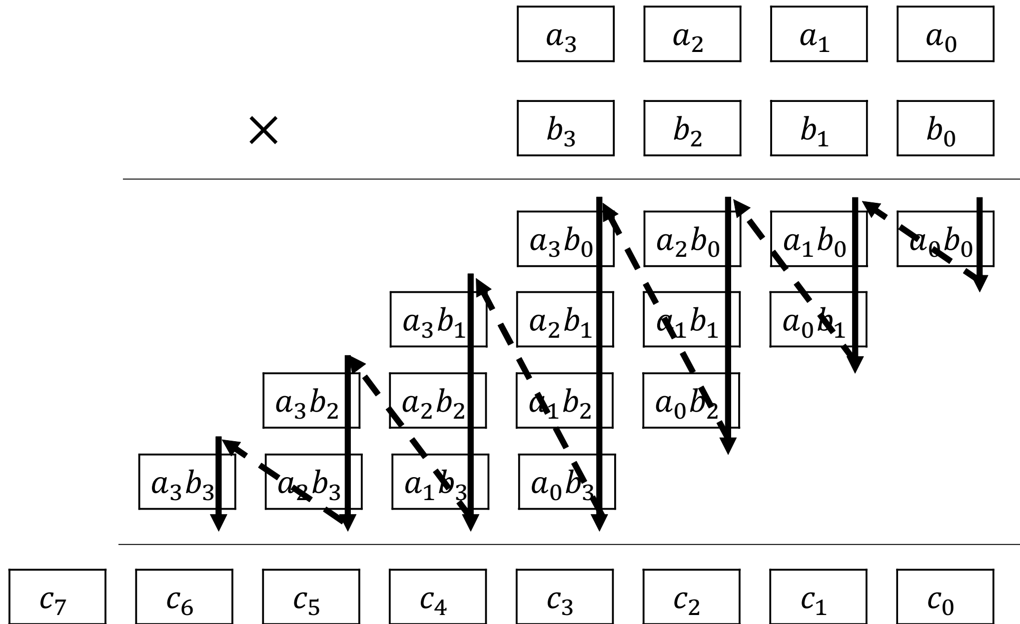

Figure 3.2: The order of word-wise multiplication of Comba method for multiple-length numbers $C = A \cdot B$

method splits each number into multiple parts of equal length. A k-way Toom-Cook method (called Toom-k) can do a multiplication by dividing an integer into k parts. Toom-3 reduces the number of multiplications to from 9 to 5.

For simplicity, we will explain how to perform Toom-3 as follows. Let us consider two numbers A and B to be multiplied are split into three parts of size $\frac{R}{3}$ bits each such that $A = A_2 \cdot 2^{\frac{2R}{3}} + A_1 \cdot 2^{\frac{R}{3}} + A_0$ and $B = B_2 \cdot 2^{\frac{2R}{3}} + B_1 \cdot 2^{\frac{R}{3}} + B_0$. The product C $(= A \times B)$ is also divided such that $C_4 \cdot 2^{\frac{4R}{3}} + C_3 \cdot 2^{\frac{3R}{3}} + C_2 \cdot 2^{\frac{2R}{3}} + C_1 \cdot 2^{\frac{R}{3}} + C_0$. In Toom-Cook method, these formulae are considered as polynomials by replacing $2^{\frac{R}{3}}$ with a variable x, as follows:

$$
\begin{aligned}
A(x) &= A_2 x^2 + A_1 x + A_0, \\
B(x) &= B_2 x^2 + B_1 x + B_0,
\end{aligned}
$$

$$C(x) = C_4x^4 + C_3x^3 + C_2x^2 + C_1x + C_0.$$

If the coefficients C_0, C_1, C_2, C_3 and C_4 are determined, the product $A \times B$ can be computed from $C(2^{\frac{R}{3}})$. Since the number of coefficients is 5, we can determine them by solving 5 equations obtained by assigning 5 distinct values to x in $C(x)$. For example, the following 5 equations are produced by $x = 0, 1, -1, 2, \infty$;

$$\begin{aligned}
C(0) &= C_0, \\
C(1) &= C_4 + C_3 + C_2 + C_1 + C_0, \\
C(-1) &= C_4 - C_3 + C_2 - C_1 + C_0, \\
C(2) &= 16C_4 + 8C_3 + 4C_2 + 2C_1 + C_0, \\
C(\infty) &= C_4.
\end{aligned}$$

Note that $C(\infty)$ is equivalent to $\lim_{x\to\infty} \frac{C(x)}{x^4}$, that is, $C(\infty)$ equals to the value of the highest-degree coefficient of $C(x)$. By solving these equations, we have the following values of coefficients:

$$\begin{aligned}
C_0 &= C(0), \\
C_1 &= \frac{1}{6}(-3C(0) + 6C(1) - 2C(-1) - C(2) + 12C(\infty)), \\
C_2 &= \frac{1}{2}(-2C(0) + C(1) + C(-1) - 2C(\infty)), \\
C_3 &= \frac{1}{6}(3C(0) - 3C(1) - C(-1) + C(2) - 12C(\infty)), \\
C_4 &= C(\infty).
\end{aligned}$$

On the other hand, from the definition we have

$$\begin{aligned}
C(0) &= A_0B_0, \\
C(1) &= (A_2 + A_1 + A_0)(B_2 + B_1 + B_0),
\end{aligned}$$

$$
\begin{aligned}
C(-1) &= (A_2 - A_1 + A_0)(B_2 - B_1 + B_0), \\
C(2) &= (4A_2 + 2A_1 + A_0)(4B_2 + 2B_1 + B_0), \\
C(\infty) &= A_2 B_2.
\end{aligned}
$$

Using these formulae, $C(0), C(1), C(-1), C(2)$, and $C(\infty)$ are computed from A and B. After that, by computing C_0, C_1, C_2, C_3, and C_4, the final result of the product can be obtained. According to the result, the number of multiplications excluding small constant numbers to be multiplied is reduced from 9 to 5. For $k \geq 2$, we can perform Toom-k multiplication in a similar way.

3.3 Parallel multiple-length multiplication for the GPU

First, we propose a parallel 1024-bit multiple-length multiplication method, called *Sum-rotate multiplication*. After that, we show Toom-k multiplication methods for more than 1024 bits with the 1024-bit multiple-length multiplication method as a sub-routine.

3.3.1 Sum-rotate multiplication

We propose a parallel 1024-bit multiple-length multiplication method using a warp. The method is based on warp-synchronous programming technique. In the following, w threads, which correspond to one warp, are used and work in parallel without any barrier synchronize operations since threads within a warp execute the same instruction and synchronize for each instruction. Also, the proposed parallel multiple-length multiplication does not use any shared memory. It is a parallel algorithm that parallelizes School method basically, called *Sum-rotate multiplication*. To achieve this, we employ warp shuffle functions as described in Chapter 2. More specifically, data exchange methods,

broadcast and right/left circular shift, as shown in Figures 2.5, 2.6 and 2.7 using warp shuffle function `shfl()` are utilized. The detail of the parallel algorithm is presented next.

In our approach, a product $C = (c_{2w-1}, ..., c_1, c_0)$ of two w-word numbers $A = (a_{w-1}, ..., a_1, a_0)$ and $B = (b_{w-1}, ..., b_1, b_0)$ is computed, where the size of each word is 32 bits. Since $w = 32$ unless the value of w is not changed for changing the GPU architecture in the future, this algorithm supports a multiplication of two 1024-bit numbers.

Let us consider how to perform the computation using multiple threads. A simple idea is to assign threads column by column as illustrated in Figure 3.3. In the figure, threads are assigned to two columns to balance the computation load of each threads. However, since threads have to switch columns in distinct timings during the computation, warp divergence, described in Chapter 2, occurs. This parallel approach is not suitable for GPUs.

On the other hand, in the proposed approach, w threads, that correspond to one warp, are used. Each thread is assigned to one of the partial products in each row. More specifically, when w threads (thread 0, thread 1,..., thread $w - 1$) are launched, thread i computes partial products $a_i b_0$, $a_i b_1$,..., $a_i b_{w-1}$ for each row as illustrated in Figure 3.4. Using this assignment of threads, almost all operations are the same between threads, that is warp divergence can be avoided mostly.

In the proposed approach, since each thread takes partial products shifting to the upper digits row by row, it is necessary to obtain the partial products, except the carry, from a thread assigned to the upper digits. To achieve this, we use the inter-thread right circular shift described in Chapter 2. In each row, thread 0 obtains the final product

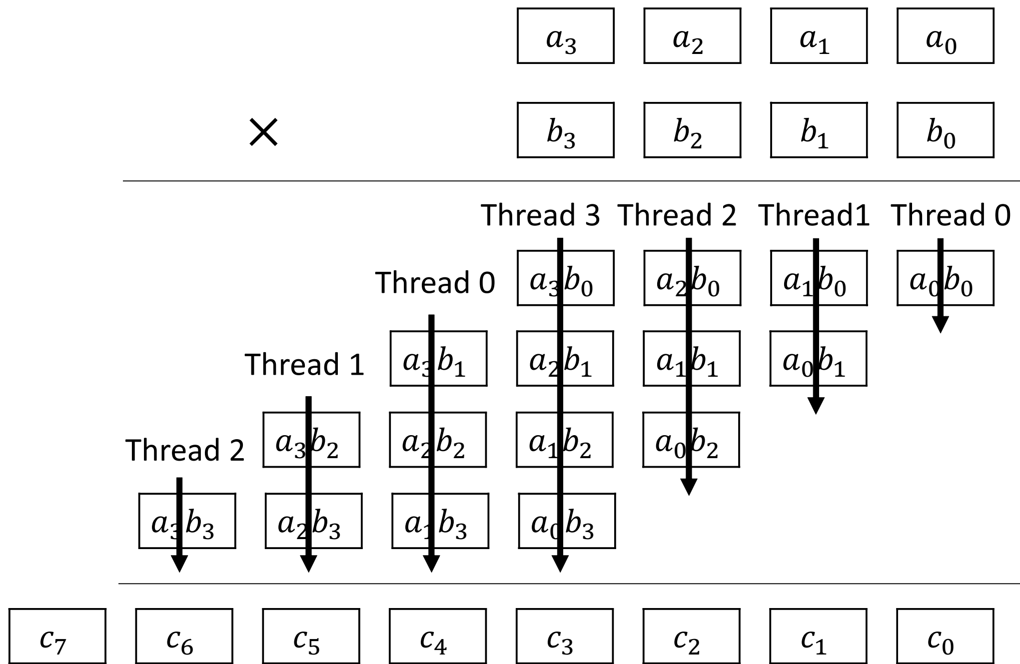

Figure 3.3: Parallel column-based multiplication

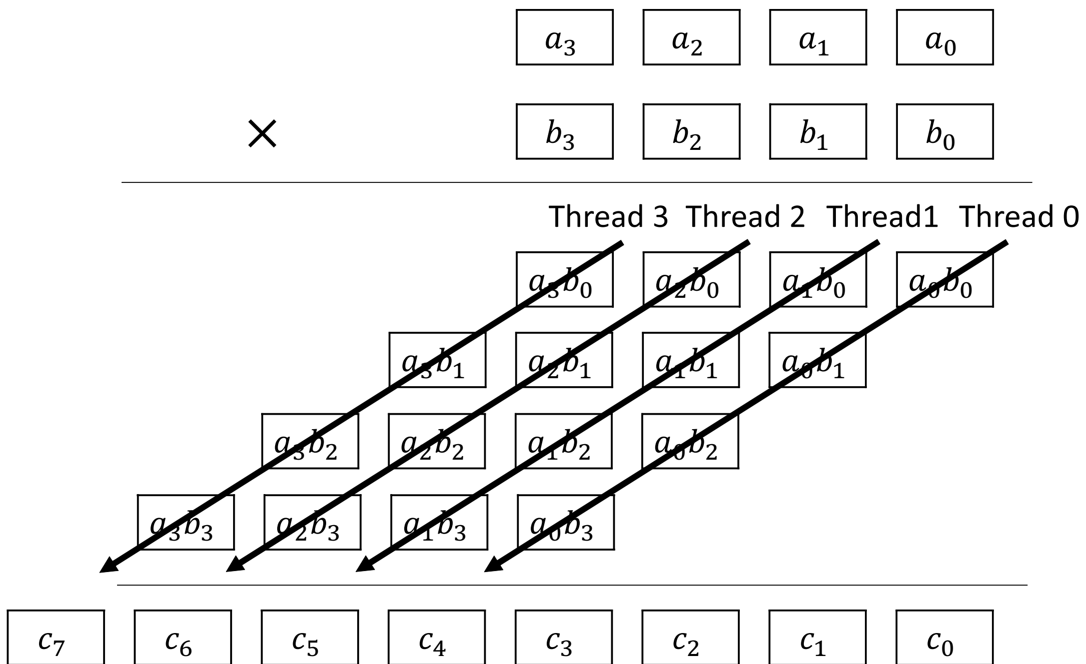

Figure 3.4: Sum-rotate multiplication

of c_j. According to Figure 3.4, a thread assigned to the lowest digits can obtain the lower words of the final product $c_0, ..., c_{w-1}$ for each row. On the other hand, the upper words of $c_w, ..., c_{2w-1}$ are finally computed by thread 0, ..., thread $w-1$, respectively. After completing the multiplication, $2w$ words of the final results are placed such that thread i has two words c_i and c_{i+w} to store the results to consecutive address of the global memory using coalescing access in parallel.

The details of sum-rotate multiplication are shown in Algorithm 3. Each step of the algorithm is executed by w threads in parallel. First of all, in lines 3 and 4, each thread loads one word of each from A and B stored in the global memory and stores them to its own registers a and b. After that, the multiplication is performed row by row as illustrated in Figure 3.4. In line 6, thread j broadcasts b_j to local register b' using the warp shuffle function to compute the product $a \cdot b'$ in the next step. In line 7, partial products are computed including the addition of the carry from the upper digits. Each thread obtains the partial products except the carry from a thread assigned to the upper digits as the carry for the next digits by right circular shift of register v in line 8. In line 9, product c_i of the final product computed by thread 0 is transferred to the right thread using right circular shift of register c'. Next, thread $w-1$, that is assigned to the leftmost thread in Figure 3.4, set registers c' and v to v and 0, respectively. This is for the right circular shift operations in lines 8 and 9. Since this operation is performed only by thread $w-1$, warp divergence occurs, but the effect to the performance seems to be very small. After that, each thread obtains the value of the next digits in line 14. After for loop, each thread has the lower digits of the final products $c_0, ..., c_{w-1}$, respectively. At that time, the upper digits $c_w, ..., c_{2w-1}$ has not been computed yet since each thread still has the carry. Therefore, the while loop in lines 16 to 19, carry propagation is performed

using left circular shift until any threads have no carry. In order to check whether any threads have no carry, we use warp vote function `any()` that evaluates truth values given from all threads of the warp and return non-zero if any of the truth values is non-zero [21]. This while loop is iterated at most $w - 1$ times. After the loop, since thread i has two words c_i and c_{i+w} , they are stored to the global memory with coalesced access in lines 20 and 21.

3.3.2 Toom-*k* multiplication with sum-rotate multiplication

In our GPU implementation, we use Toom-k multiplication method for more than 1024-bit numbers. For simplicity, we explain our GPU implementation with Toom-3 as follows. The GPU implementation consists of two kernels. In the first kernel, five blocks that consist of one warp, that is five warps in total, are assigned to each multiplication. Each warp computes one value of $C(0)$, $C(1)$, $C(-1)$, $C(2)$, and $C(\infty)$ shown in Section 3.2 with sum-rotate multiplication. When the size of multiplication is more than 1024 bits, each warp sequentially performs Comba method by repeating sum-rotate multiplication. In the second kernel, one block consisting of one warp is assigned to every 32 multiplications. Each thread computes one multiplication using the values obtained in the first kernel. Namely, one thread computes the values of $C_0, ..., C_4$ in serial and then, the final result of the multiplication is computed. In the second kernel, each thread runs for individual multiplications, but warp divergence does not occur since all threads execute the same instructions. In the above two kernels, each block is composed of only one warp so that the occupancy is increased by reducing the number of warps in a block. Moreover, all the data stored in the global memory is arranged so that the memory access can be performed with coalesced access, without going into detail. For $k \geq 2$, the

Algorithm 3 Sum-rotate multiplication using a warp

Input: $A = (a_{w-1}, ..., a_1, a_0)$, $B = (b_{w-1}, ..., b_1, b_0)$

Output: $C = AB$

1: $i \leftarrow id(= 0, 1, ..., w-1)$

2: $u \leftarrow 0, v \leftarrow 0, c' \leftarrow 0$

3: $a \leftarrow a_i$

4: $b \leftarrow b_i$

5: **for** $j \leftarrow 0$ **to** $w - 1$ **do**

6: $b' \leftarrow$`shfl`(b, j) ▷ Broadcast b from thread j

7: $\{t, u, v\} \leftarrow a \cdot b' + \{u, v\}$

8: $v \leftarrow$`shfl`$(v, (i + 1)\%w)$ ▷ Right circular shift v

9: $c' \leftarrow$`shfl`$(c', (i + 1)\%w)$ ▷ Right circular shift c'

10: **if** $id = w - 1$ **then**

11: $c' \leftarrow v$

12: $v \leftarrow 0$

13: **end if**

14: $\{u, v\} \leftarrow \{t, u\} + v$

15: **end for**

16: **while** `any`$(u) \neq 0$ **do**

17: $u \leftarrow$ `shfl`$(u, (i + w - 1)\%w)$ ▷ Left circular shift u

18: $\{u, v\} \leftarrow u + v$

19: **end while**

20: $c_i \leftarrow c'$

21: $c_{i+w} \leftarrow v$

GPU implementation performs Toom-k multiplication in a same way.

Let us evaluate the number of multiplications and memory access in the above methods that are important factors for the computing time. Table 3.2 shows a comparison of the number of multiplications and memory read/write for multiplying two w-word numbers in our implementation. Regarding as the number of multiplication, School and Comba methods are the largest among them. In Toom-k, when k is larger, the number of multiplications is smaller. Although every memory access is performed by coalesced access, memory access time is much longer than the other instructions such as arithmetic operations including multiplication and addition. Therefore, the computing time does not always become shorter for larger k. Hence, we should find the best parameter k that minimizes the computing time for the bit size of numbers.

3.3.3 Optimization of the code for arithmetic

As mentioned in the above, arithmetic with larger integers having more than 64 bits is necessary. In C language, however, there is no efficient way of doing such arithmetic because C language does not support operations with the carry bits. To optimize the arithmetic with more than 64-bit integers, therefore, a part of the code is written in PTX [20] that is an assembly language for NVIDIA GPUs and can be used as inline assembler in CUDA C language. PTX supports arithmetic operations with the carry bits.

3.4 Experimental results

The main purpose of this section is to show the experimental results. In order to evaluate the computing time for multiple-length multiplication, we have used NVIDIA GeForce

Table 3.2: The number of multiplications and memory read/write for multiplying two w-word numbers in our implementation

method	multiplication	memory read	memory write
School	w^2	$2w^2$	$w^2 + w$
Comba	w^2	$2w^2 - 2$	$2w$
Toom-2	$\frac{3}{4}w^2$	$\frac{3}{2}w^2 + \frac{17}{2}w - 2$	$\frac{15}{2}w + 4$
Toom-3	$\frac{5}{9}w^2 + \frac{44}{3}w + 21$	$\frac{10}{9}w^2 + \frac{82}{3}w + 32$	$\frac{44}{3}w + 36$
Toom-4	$\frac{7}{16}w^2 + 26w + 55$	$\frac{7}{8}w^2 + 41w + 76$	$17w + 60$
Toom-5	$\frac{9}{25}w^2 + \frac{188}{5}w + 105$	$\frac{18}{25}w^2 + 54w + 136$	$\frac{92}{5}w + 84$
Toom-6	$\frac{11}{36}w^2 + \frac{148}{3}w + 171$	$\frac{11}{18}w^2 + \frac{200}{3}w + 212$	$\frac{58}{3}w + 108$
Toom-7	$\frac{13}{49}w^2 + \frac{428}{7}w + 253$	$\frac{26}{49}w^2 + \frac{554}{7}w + 304$	$20w + 132$
Toom-8	$\frac{15}{64}w^2 + 73w + 351$	$\frac{15}{32}w^2 + \frac{183}{2}w + 412$	$\frac{41}{2}w + 156$
Toom-9	$\frac{17}{81}w^2 + \frac{764}{9}w + 465$	$\frac{34}{81}w^2 + \frac{934}{9}w + 536$	$\frac{188}{3}w + 180$

Table 3.3: The computing time of GPU implementations for 100000 1024-bit multiplications

method	execution unit for one multiplication	time[ms]
School	single thread	8.43
Comba	single thread	5.06
parallel	one warp with shared memory	1.73
column-based	one warp with warp shuffle functions	1.43
parallel	one warp with shared memory	1.43
sum-rotate	one warp with warp shuffle functions	0.82

GTX 980, which has 2048 cores running 1.216GHz [6]. The source program of the GPU implementation is compiled by nvcc version 6.5.13 with -O2 and -arch=sm_50 options. In the following, the computing time is average of 10 times execution and the computing time of the GPU does not include data transfer time between the main memory in the CPU and the device memory in the GPU. The reason that data transfer time between the main memory in the CPU and the device memory in the GPU is excluded is that in practical case this execution of multiplication is performed before and/or after some process on the GPU. Therefore, input and output data is not always located in the main memory.

First, we evaluate the performance of sum-rotate multiplication based on warp-synchronous programming technique. We have implemented the single thread implementation such that each thread computes one multiplication. This implementation is based on the idea proposed in [26]. In the implementation, there is no warp divergence

since all threads execute the same instructions, that is, this implementation is also based on warp-synchronous programming technique. In addition, to evaluate the effect of the use of warp shuffle function, we have implemented a multiplication method with the shared memory instead of the warp shuffle function. Table 3.3 shows the computing time when 100000 multiple-length multiplications of 1024 bits are computed. In the above implementations, every block has 32 threads, that is, one warp. According to the table, we can find that one warp implementation is faster than the single thread implementation. For data communication within threads, use of warp shuffle functions is more effective than that of shared memory.

Table 3.3 also shows the computing time of the sequential School method and parallel column-based multiplication with shared memory. These two methods are mainly used in the existing methods proposed by Zhao [31] and Kitano [16]. As a result, the proposed method is 10.2 and 2.1 times faster than the existing two methods, respectively.

Figure 3.5 shows the computing time of the GPU implementations of Comba, Toom-k ($k = 2, ..., 9$) when 10240 multiplications are computed. Recall that each implementation uses sum-rotate multiplication with warp shuffle functions as a sub-routine for 1024-bit multiplication. Although the graph may be unclear, Comba method is faster for less than 10240 bits. For 10240 or more bits, Toom-k is faster than Comba method. Also, when the number of bits is larger, Toom-k for larger k ($k \leq 8$) is faster. However, Toom-9 is not the fastest for any number of bits because the overhead such as memory access cannot be ignored even though the number of multiplication is less than the other methods from Table 3.2.

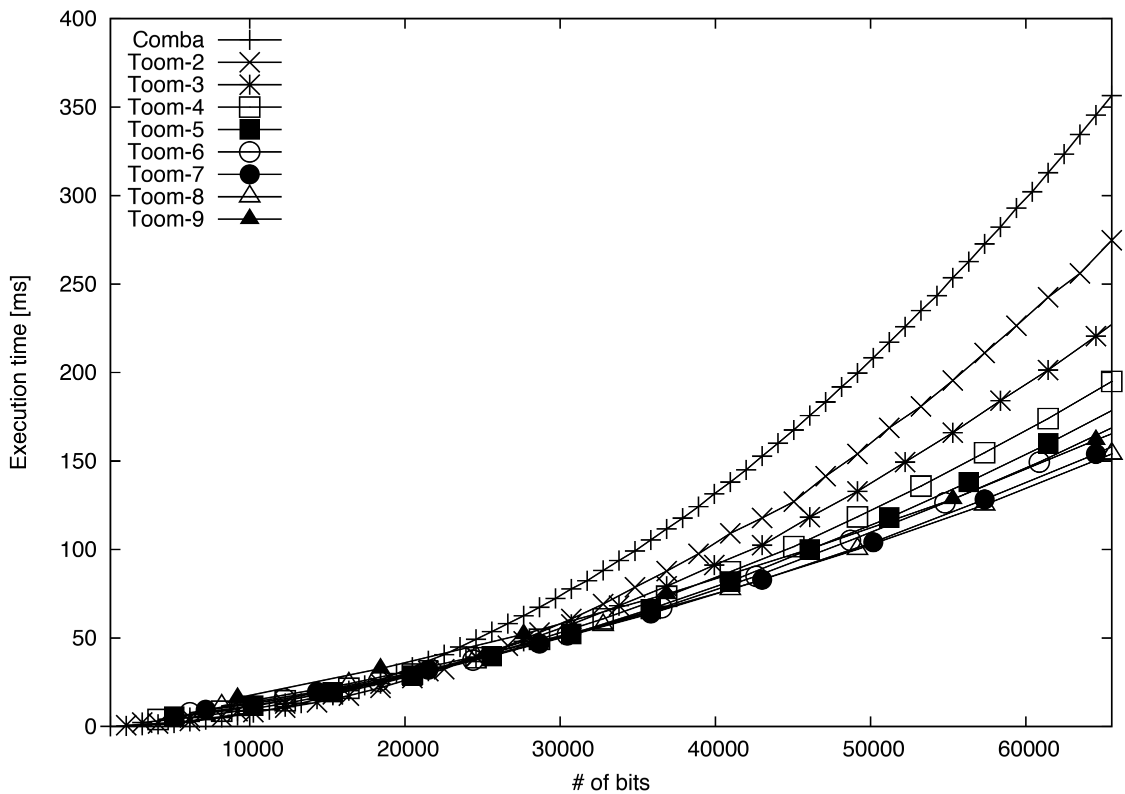

Figure 3.5: The computing time of GPU implementations in milliseconds for 10240 multiple-length multiplications

We have also used Intel PC using Xeon X7460 running on 2.6GHz to compare the performance of the GPU implementation with sequential algorithms on the CPU. In the CPU implementation, we have utilized GMP version 6.1.0. The source program is compiled by gcc version 4.8.3 with -O2 option. Table 3.4 shows the comparison between CPU and GPU implementations for the computing time in milliseconds when 10240 multiple-length multiplications are computed. The table also shows the multiplication methods that have been used. In the CPU implementation, when the GMP library was installed, the best method has been selected according to the execution environment. On the other hand, the best method shown in Figure 3.5 has been used in the GPU implementation. We note that in 64-, 128-, 256-, and 512-bit multiplications, we cannot directly use the sum-rotate multiplication shown in Section 3.3 since 32 threads in one warp compute one 1024-bit multiplication. Therefore, in such smaller size of bits, several multiplications are concurrently computed by one warp to use all the threads in a warp. For example, in 256-bit multiplications, four 256-bit multiplications are simultaneously computed by assigning 8 threads in a warp to each 256-bit multiplication. According to the table, using the proposed GPU implementation, the computing time can be reduced by a factor of 19.65 to 52.14.

Table 3.4 also shows the computing time of GPU implementation using CUMP library version 1.0.1 [19] that supports multiple-length multiplication on the GPU. Since CUMP uses the local memory whose size is limited, 65536-bit multiplication could not be executed. Also, in the GPU implementation with CUMP, each multiplication is computed with School method by one thread sequentially. As a result, the proposed method can compute multiplication 1.6 to 50 times faster than CUMP.

Table 3.4: The comparison between sequential CPU implementation and GPU implementations for the computing time when 10240 multiple-length multiplications are computed

# of bits		64	128	256	512	1024	2048	4096	8192	16384	32768	65536
CPU	time[ms]	0.201	0.214	0.627	1.468	4.858	16.755	54.624	168.431	497.046	1279.736	3330.669
(GMP [11])	method	School	School	School	School	School	Toom-2	Toom-2	Toom-3	Toom-6	Toom-8	Toom-8
GPU 1	time[ms]	0.509	0.515	0.703	0.743	0.874	1.298	2.648	8.008	29.077	107.361	-
(CUMP [19])	method	School	School	School	School	School	School	School	School	School	School	-
GPU 2	time[ms]	0.010	0.012	0.016	0.031	0.093	0.345	1.297	5.024	17.892	57.137	154.015
(proposed method)	method	Comba	Comba	Comba	Comba	Comba	Comba	Comba	Comba	Toom-2	Toom-8	Toom-8
Speed-up (CPU/GPU 2)		19.65	18.32	39.73	47.93	52.14	48.49	42.13	30.10	27.78	22.40	21.63

3.5 Concluding remarks

We have presented a GPU implementation of bulk multiple-length multiplications. The idea of our GPU implementation is to adopt warp-synchronous programming technique. Using this idea, we have proposed Sum-rotate multiplication of two 1024-bit integers. We assign each multiple-length multiplication to one warp that consists of 32 threads. The experimental results show that our GPU implementation on NVIDIA GeForce GTX 980 attains a speed-up factor of 52 for 1024-bit multiple-length multiplications over the single CPU implementation using GNU multiple precision arithmetic library. Moreover, we use this 1024-bit multiple-length multiplication for larger size of bits as a sub-routine. The GPU implementation attains a speedup factor of 21 for 65536-bit multiple-length multiplication.

Chapter 4

GPU implementations of exhaustive verification of the Collatz conjecture

The main contribution of this chapter is to further accelerate the exhaustive verification for the Collatz conjecture using a GPU. The ideas of our GPU implementation are

- a GPU-CPU cooperative approach,
- efficient memory access for the global memory and the shared memory, and
- optimization of the code for arithmetic with larger integers.

4.1 The Collatz conjecture

The Collatz conjecture is a well-known unsolved conjecture in mathematics [17, 23, 25, 30]. Consider the following operations on an arbitrary positive number;

even operation: if the number is even, divide it by two, and

odd operation: if the number is odd, triple it and add one.

The Collatz conjecture asserts that, starting from any positive number, repeated iteration of the operations eventually produces the value 1. For example, starting from 3, we have the following sequence to produce 1;

$$3 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1.$$

The exhaustive verification of the Collatz conjecture is to perform the repeated operations for numbers from 1 to the infinite as Algorithm 4. Clearly, if the Collatz conjecture

Algorithm 4 Exhaustive verification algorithm of Collatz conjecture

```
1:  for m ← 1 to +∞ do
2:    n ← m
3:    while n > 1 do
4:      if n is even then
5:        n ← n/2
6:      else
7:        n ← 3n + 1
8:      end if
9:    end while
10: end for
```

is not true, then the while-loop in the program above never terminates for a counter example m. Working projects for the Collatz conjecture are currently checking 61-bit numbers [23] and 72-bit numbers [1]. The project in [1] only shows the number of odd/even operations until the 72-bit number reaches 1. On the other hand, regarding the mathematical interest for the Collatz conjecture, not only whether numbers converge to 1, called *convergence*, but also the number of the odd/even operations until a number reaches 1, called *delay*, interests the working project in [23]. Let $D(n)$ denote a delay

of a positive integer n. For example, starting from 3, 1 is produced by 2 odd operations and 5 even operations, that is, $D(3) = 2 + 5 = 7$. In [23], delay is used to compute a *delay record*. A delay record is defined such that a positive integer n is a delay record if for all positive integers m ($m < n$) we have $D(m) < D(n)$. For example, 3 is a delay record since $D(1) = 0$, $D(2) = 1$ and $D(3) = 7$.

4.2 Related work

There are several researches for accelerating the exhaustive verification of the Collatz conjecture. It is known [2, 12, 13, 14] that series of even and odd operations for n can be done in one step by computing $n \leftarrow B[n_L] \cdot n_H + C[n_L]$ for appropriate tables B and C, where the concatenation of n_H and n_L corresponds to n.

In [2, 12, 13, 14], FPGA implementations have been proposed to repeat the operations of the Collatz conjecture. These implementations perform the even and odd operations for some fixed size of bits of interim numbers. However, in [2], the implementation ignores the overflow. Hence, if there exists a counter example number m for the Collatz conjecture such that, infinitely large numbers are generated by the operations from m, their implementation may fail to detect it. On the other hand, in [12], the implementation can verify the conjecture for up to 23-bit numbers. This is not sufficient because a working project for the Collatz conjecture is currently checking 61-bit numbers [23].

In [13], a software-hardware cooperative approach to verify the Collatz conjecture for 64-bit numbers n has been shown. This approach supports almost infinitely large interim numbers m. The idea is to perform the while-loop for interim values with up to 78 bits using a coprocessor embedded in an FPGA. If an interim value m has more than

78 bits, the original value n is reported to the host PC. The host PC performs the verification for such n using a quite large number of bits by software. This software-hardware cooperative approach makes sense, because the hardware implementation on the FPGA is fast and low power consumption, but the number of bits for the operation is fixed, and the software implementation on the PC is relatively slow and high power consumption, but the number of bits for the operation is quite large. Additionally, in [14], an efficient implementation of a coprocessor that performs the exhaustive search to verify the Collatz conjecture using embedded DSP slices on a Xilinx FPGA has been proposed. By effective use of embedded DSP slices instead of multipliers used in [13], the coprocessor can perform the exhaustive verification faster than the above FPGA implementations.

4.3 Accelerating the verification of the Collatz conjecture

The main purpose of this section is to introduce algorithms for accelerating the verification for the convergence and the delay of the Collatz conjecture. The basic ideas of acceleration are shown in [17, 30] and the details of them are shown, as follows.

4.3.1 Verification algorithm for the convergence

In the verification of the convergence, we use the following techniques. The first technique is to terminate the operations before the iteration produces 1. Suppose that we have already verified that the Collatz conjecture is true for numbers less than n, and we are now in position to verify it for number n. Clearly, if we repeatedly execute the operations for n until the value is 1, then we can confirm that the conjecture is true for

n. Instead, if the value becomes n' for some n' less than n, then we can guarantee that the conjecture is true for n because it has been proved to be true for n'. Thus, it is not necessary to repeat this operation until the value is 1, and we can terminate the iteration when, for the first time, the value is less than n.

The second technique is to perform several operations in one step. Consider that we want to perform the operations for n and let n_L and n_H be the least significant two bits and the remaining bits of n. In other words, $n = 4n_H + n_L$ holds. Clearly, the value of n_L is either $(00)_2$, $(01)_2$, $(10)_2$, or $(11)_2$. We can perform the several operations for n based on n_L as follows:

$n_L = (00)_2$: Since two even operations are applied, the resulting number is n_H.

$n_L = (01)_2$: First, odd operation is applied and the resulting number is $(4n_H + 1) \cdot 3 + 1 = 12n_H + 4$. After that, two even operations are applied, and we have $3n_H + 1$.

$n_L = (10)_2$: First, even operation is performed and we have $2n_H + 1$. Second, odd operation is applied and we have $(2n_H + 1) \cdot 3 + 1 = 6n_H + 4$. Finally, by even operation, the value is $3n_H + 2$.

$n_L = (11)_2$: First, odd operation is applied and we have $(4n_H + 3) \cdot 3 + 1 = 12n_H + 10$. Second, by even operation, the value is $6n_H + 5$. Again, odd operation is performed and we have $(6n_H + 5) \cdot 3 + 1 = 18n_H + 16$. Finally, by even operation, we have $9n_H + 8$.

For example, if $n_L = (11)_2$ then we can obtain $9n_H + 8$ by applying 4 operations, odd, even, odd, and even operations in turn. Let B and C be tables as follows:

	B	C
$(00)_2$	1	0
$(01)_2$	3	1
$(10)_2$	3	2
$(11)_2$	9	8

Using these tables, we can perform the following table operation, which emulates several odd and even operations:

table operation For least significant two bits n_L and the remaining most significant bits n_H of the value, the new value is $B[n_L] \cdot n_H + C[n_L]$.

Let us extend the table operation for least significant two bits to d bits. For an integer $n \geq 2^d$, let n_L and n_H be the least significant d bits and the remaining bits, respectively. Namely, $n = 2^d n_H + n_L$. We call d is *the base bits*. Suppose that, the even or odd operations are repeatedly performed on $n = 2^d n_H + n_L$. We use two integers b and c such that $n = b \cdot n_H + c$ to denote the current value of n. Initially, $b = 2^d$ and $c = n_L$. We repeatedly perform the following rules for b and c;

even rule: If both b and c are even, then divide them by two, and

odd rule: If b is even and c is odd, then triple b, and triple c and add one.

These two rules are applied until no more rules can be applied, that is, until b is odd. It should be clear that, even and odd rules correspond to even and odd operations of the Collatz conjecture. If i even rules and j odd rules applied, then the value of b is $2^{d-i}3^j$. Thus, exactly d even rules are applied until the termination. After the termination, we can determine the value of elements in tables B and C such that $B[n_L] = b$ and $C[n_L] = c$. Using tables B and C, we can perform the table operation for d bits n_L, which involves

d even operations and zero or more odd operations. In this way, we can accelerate the operation of the Collatz conjecture. In this chapter, we have implemented for various numbers of bits of n_L.

The third technique to accelerate the verification of the Collatz conjecture is to skip numbers n such that we can guarantee that the resulting number is less than n after the table operation. For example, suppose we are using two bit table and $n_H > 0$. If $n_L = (00)_2$ then the resulting value is n_H, which is less than n. Thus, we can skip the table operation for n if $n_L = (00)_2$. If $n_L = (01)_2$ then the resulting value is $3n_H + 1$, which is always less than $n = 4n_H+1$, and we can skip the table operation. Similarly, if $n_L = (10)_2$ then we can skip the table operation. On the other hand $n_L = (11)_2$ then the resulting value is $9n_H + 8$, which is always larger than n. Therefore, the Collatz conjecture is guaranteed to be true whenever $n_L \neq (11)_2$, because it has been verified true for numbers less than n. Consequently, we need to execute the table operation for number n such that $n_L = (11)_2$. We can extend this idea for general case. For least significant d bits n_L, we say that n_L is not *mandatory* if the value of b is less than 2^d at some moment while even and odd rules are repeatedly applied. We can skip the verification for non-mandatory n_L. The reason is as follows: Consider that for number n, we are applying even and odd rules. Initially, $b = 2^d$ and $c \leq 2^d - 1$ hold. Thus, while even and odd rules are applied, $b > c$ always holds. Suppose that $b \leq 2^d - 1$ holds at some moment while the rules are applied. Then, the current value of n is $bn_H + c < bn_H + b \leq (2^d - 1)n_H + b < 2^d n_H \leq n$. It follows that, the value is less than n when the corresponding even and odd operations are applied. Therefore, we can omit the verification for numbers that have no mandatory least significant bits. We note that this technique cannot be applied to the computation for the delay because every number has its own value of the delay and cannot be skipped.

For least significant d bit number, we use table S to store the mandatory least significant bits. Let s_d be the number of such mandatory least significant bits. Using these tables, we can write a verification algorithm in Algorithm 5. For the benefit of readers, we show tables B, C, and S for 4 base bits in Table 4.1. From $s_4 = 3$, we have 3 mandatory least significant bits out of 16.

Algorithm 5 Verification algorithm for convergence of Collatz conjecture

```
1:  for m_H ← 1 to +∞ do
2:    for i ← 0 to s_d − 1 do
3:      m_L ← S[i]
4:      n ← m ← 2^d m_H + m_L
5:      while n ≥ m do
6:        Let n_L be the least significant d bits and n_H be the remaining bits.
7:        n ← B[n_L] · n_H + C[n_L]
8:      end while
9:    end for
10: end for
```

For the reader's benefit, Table 4.2 shows the necessary word size for each of tables B and C for each base bit. It also shows the expected number of odd/even operations included in one step operation $n \leftarrow B[n_L] \cdot n_H + C[n_L]$. Table 4.3 shows the size of table S. It further shows the ratio of the mandatory numbers over all numbers. Later, we set base bit 18 for tables B and C, and base bit 37 for table S in our proposed GPU implementation. Thus, in our implementation, one operation $n \leftarrow B[n_L] \cdot n_H + C[n_L]$ corresponds to expected 27.0 odd/even operations. Also, we skip approximately 99.3% of non-mandatory numbers.

Table 4.1: Tables B, C, and S for least significant 4 bits.

	B	C	S
$(0000)_2$	1	0	$(0111)_2$
$(0001)_2$	9	1	$(1011)_2$
$(0010)_2$	9	2	$(1111)_2$
$(0011)_2$	9	2	-
$(0100)_2$	3	1	-
$(0101)_2$	3	1	-
$(0110)_2$	9	4	-
$(0111)_2$	27	13	-
$(1000)_2$	3	2	-
$(1001)_2$	27	17	-
$(1010)_2$	3	2	-
$(1011)_2$	27	20	-
$(1100)_2$	9	8	-
$(1101)_2$	9	8	-
$(1110)_2$	27	26	-
$(1111)_2$	81	80	-

Table 4.2: The size of tables B and C

base bit	words	operation
4	16	6.0
5	32	7.5
6	64	9.0
7	128	10.5
8	256	12.0
9	512	13.5
10	1k	15.0
11	2k	16.5
12	4k	18.0
13	8k	19.5
14	16k	21.0
15	32k	22.5
16	64k	24.0
17	128k	25.5
18	256k	27.0

Table 4.3: The size of table S

base bit	words	ratio
3	2	0.2500
4	3	0.1875
5	4	0.1250
6	8	0.1250
7	13	0.1016
8	19	0.0742
9	38	0.0742
10	64	0.0625
11	128	0.0625
12	226	0.0552
13	367	0.0448
14	734	0.0448
15	1295	0.0395
16	2114	0.0323
17	4228	0.0323
18	7495	0.0286
19	14990	0.0286
20	27328	0.0261

base bit	words	ratio
21	46611	0.0222
22	93222	0.0222
23	168807	0.0201
24	286581	0.0171
25	573162	0.0171
26	1037374	0.0155
27	1762293	0.0131
28	3524586	0.0131
29	6385637	0.0119
30	12771274	0.0119
31	23642078	0.0110
32	41347483	0.0096
33	82694966	0.0096
34	151917639	0.0088
35	263841377	0.0077
36	527682754	0.0077
37	967378591	0.0070

4.3.2 Verification algorithm for the delay

In the following, we show an algorithm of counting delay of Collatz conjecture. In the computation of delay, the above idea for convergence that several odd/even operations are skipped by tables B and C can be used. It is necessary to count the number of odd/even operations skipped by applying a table operation. For example, when table operation uses tables B and C for least significant 2 bits, if $n_L = (11)_2$ then 4 operations, odd, even, odd, and even operations are applied in turn. Let J be a table as follows:

	J
$(00)_2$	2
$(01)_2$	3
$(10)_2$	3
$(11)_2$	4

On the other hand, the idea for the convergence that if the value becomes n' for some n' less than n by applying table operations, then we can guarantee that the conjecture is true for n cannot be applied to the computation of the delay because the number of operations needs to be counted until the value is 1. Therefore, in the computation of the delay, we introduce table A that stores the delays which have been pre-computed for all numbers less than t. Each element of table $A[i]$ $(0 \leq i \leq t-1)$ stores the delay of i. Namely, if the value becomes n' for some n' less than t, then we can obtain the delay of n' to refer $A[n']$. After that, we add $A[n']$ to the number of operations necessary to produce n' and the delay of n is obtained. In our GPU implementation, we use table A for $t = 2^{12}$. Algorithm 6 shows an algorithm to count delay of Collatz conjecture using the above ideas.

Algorithm 6 Count algorithm for delay of Collatz conjecture

1: **for** $n \leftarrow t$ **to** $+\infty$ **do**

2: $n' \leftarrow n$

3: $D(n) \leftarrow 0$

4: **while** $n' \geq t$ **do**

5: Let n'_L be the least significant d bits and n'_H be the remaining bits.

6: $D(n) \leftarrow D(n) + J[n'_L]$

7: $n' \leftarrow B[n'_L] \cdot n'_H + C[n'_L]$

8: **end while**

9: $D(n) \leftarrow D(n) + A[n']$

10: **end for**

4.4 GPU implementation

The main purpose of this section is to show GPU implementations of verifying the Collatz conjecture. The ideas of our GPU implementation are

(i) a GPU-CPU cooperative approach,

(ii) efficient memory access for the global memory and the shared memory, and

(iii) optimization of the code for arithmetic with larger integers.

In this section, we explain the details of our GPU implementation of the verification for the convergence using these ideas first. After that, our GPU implementation of the computation for the delay which is an extension of it is provided.

4.4.1 A GPU-CPU cooperative approach

In the following, we show a GPU-CPU cooperative approach that is similar to the idea of a hardware-software cooperative approach in [13]. We assume that 64-bit numbers are verified. This assumption is sufficient because a working project for the Collatz conjecture is currently checking 61-bit numbers [23]. We note that the verified numbers can be extended easily since the interim numbers in the verification can be larger than 64-bit numbers. In the verification of the Collatz conjecture, therefore, arithmetic with larger integers having more than 64 bits is necessary to compute $B[n_L] \cdot n_H + C[n_L]$. Depending on an initial value, the size of the interim value may become very large during the verification. If larger interim value is allowed in the computation on the GPU, the values cannot be stored on the registers, that is, they have to be stored on the global memory whose access latency is very long. In our implementation, the maximum size of interim values is limited to 96 bits, which consists of three 32-bit integers, to perform the computation only on the registers. By limiting the maximum size, the computation can be performed as fixed length computation without overhead caused by arbitrary length computation. Suppose that a thread finds that the interim value is overflow for the initial value m. The thread reports m through the global memory if the overflow is detected. After all the threads finish the verification, the host program checks whether there are overflows or not. If overflows are found, the host verifies the Collatz conjecture for the values using a quite large number of bits by software on the CPU. The number of bits for the verification on the host is only limited by the memory size of the host. Recent PCs have several GBytes memory. Therefore, we can verify a number even if the interim value becomes several thousands bits. In our implementation, the number of bits for the verification on the host is 960 bits. There was no 64-bit number that the

maximum size of the interim value was largar than 960 bits in our experiment.

The reader may think that if the number of overflows is larger, the verification time is longer. However, the number of overflows is small enough for the limitation of 96 bits [14]. Therefore, it is reasonable to perform the verification for overflow numbers on the host. In Section 4.5, we will evaluate the number of overflows and the verification time for them.

4.4.2 Efficient memory access for the GPU memory

To make memory access for the GPU memory efficient, we perform the broadcast access as possible using the following technique. In our GPU implementation, we arrange initial values verified by threads in a block such that the least significant bits of them are identical. More specifically, the data format of initial values is shown in Figure 4.1. In the figure, *thread_ID* denotes a thread index within a block, *block_ID* denotes a block index within a kernel, and M is a constant. In each block, $S[block_ID]$ and M are common values for threads and each thread in a block verifies the Collatz conjecture for $2^8 (= 256)$ initial values. Namely, threads in a block concurrently verify the conjecture for values that are identical except *thread_ID*. Using this arrangement, until the bits depending on the *thread_ID* are included into n_L, threads in a block can refer the identical address of tables B and C at the same time. For each iteration of the while-loop in Algorithms 5 and 6 in Section 4.3, the interim value is divided into the least significant d bits and the remaining bits, that is, the value is d-bit-right-shifted. Therefore, using the data format in Figure 4.1, threads can refer the same address $\lfloor \frac{8+(45-b)+b}{d} \rfloor = \lfloor \frac{53}{d} \rfloor$ times for each verification. For example, when $d = 11$, threads can refer the same address at least 4 times for each initial value.

Since compared with the global memory, the access time of the shared memory is faster, but the size of the shared memory is much smaller, it is important to find the optimal size of base bits for tables B and C and the optimal location in which these tables are stored in the global memory or the shared memory. Therefore, we evaluate the computing time for various cases to find the optimal ones beforehand. On the other hand, since a value in table S is read only once for each 256 numbers to be verified, compared with the total time of the computation, the access time of table S is small enough to be ignored.

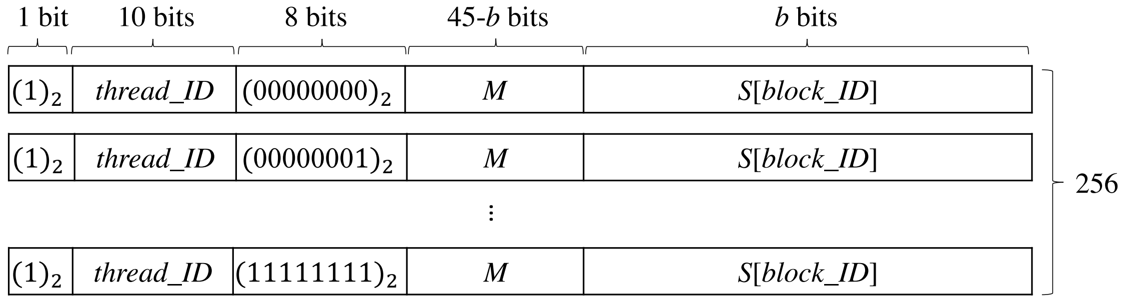

Figure 4.1: The data format of 64-bit numbers verified by each thread in a block, where *thread_ID* denotes a thread index within a block, *block_ID* denotes a block index within a kernel, and M is a constant.

4.4.3 Optimization of the code for arithmetic with larger integers

As mentioned in the above, arithmetic with larger integers having more than 64 bits is necessary to compute $B[n_L] \cdot n_H + C[n_L]$. In C language, however, there is no efficient way of doing such arithmetic because C language does not support operations with the carry flag bit. In a common way to perform the arithmetic with larger integers, 32-bit operations are performed on 64-bit operations by extending the bit-length. However,

the overhead of type conversion for the extension of the bit-length cannot be ignored. To optimize the arithmetic with larger integers, therefore, a part of the code is written in PTX [20] that is an assembly language for NVIDIA GPUs and can be used as inline assembler in CUDA C language. PTX supports arithmetic operations with the carry flag bit. Concretely, we use *mad* and *madc* that are 32-bit arithmetic operations in PTX to compute $B[n_L] \cdot n_H + C[n_L]$. These operations multiply two 32-bit integers and add one 32-bit integer excluding and including the carry flag bit, respectively. Applying the optimization of the code, in the preliminary experiment, the result shows that the optimized implementation can verify the Collatz conjecture approximately 1.8 times faster than the non-optimized implementation.

4.4.4 GPU implementation of the computation for the delay

Our GPU implementation of the computation for the delay that counts the number of odd/even operations for a number, is very similar to that for the convergence described in the above. The delay computation shown in Algorithm 6 additionally uses table J and table A is used instead of table S. Also, since the condition of the while-loop is difference, compared with the verification of the convergence, there is an increase on the number of iterations of the while-loop. In the GPU implementation for the delay, it is also important to find the optimal size of base bits for tables B, C, and J and the optimal location in which these tables are stored in the global memory or the shared memory. In addition, the size of table A is also an important factor since the size determines the number of iterations of the while-loop in Algorithm 6. Besides, a value in table A is read once for each number to be verified though a value of table S for the convergence computation is referred once for each 256 numbers. Thus, we evaluate the computing

time for various cases to find the optimal parameters of the tables beforehand.

4.5 Performance evaluation

We have implemented two GPU implementations of verifying the Collatz conjecture. One is for the convergence and the other is for the delay. We use CUDA C with NVIDIA GeForce GTX TITAN X with 3072 processing cores (24 streaming multiprocessors which have 128 processing cores each) running in 1075 MHz and 12 GB memory. For the purpose of estimating the speed up of our GPU implementation, we have also implemented a sequential implementation of verifying the Collatz conjecture using GNU C. In the sequential implementation, we can apply the idea of accelerating the verification described in Section 4.3. For example, in the CPU implementation, the maximum size of interim values is limited to 96 bits, which consists of three 32-bit integers, to avoid the overhead caused by arbitrary length computation just as the GPU implementation. Suppose that when an interim value is overflow for the initial value m, m is stored to the memory as an overflowed value. After all the computation is finished, the program checks whether there are overflows or not. If overflows are found, the verification is performed for the values using a quite large number of bits. We have used in Intel Core i7-4790 running in 3.6GHz and 32GB memory to run the sequential CPU implementation.

For the computation of the convergence and the delay, we have evaluated the computing time of the GPU implementation by verifying the Collatz conjecture for the 64-bit numbers whose data format is shown in Figure 4.1. For this purpose, we have randomly generated integers as a constant M.

Regarding the size of verified numbers, our GPU implementation computes interim

Table 4.4: The number of verified 64-bit numbers ($\times 10^9$) per second for various size of base bit of tables B and C for the convergence

size of bits	10	11	12	13	14	15	16	17	18	19	20
GPU(shared memory)	677	697	763	—	—	—	—	—	—	—	—
GPU(global memory)	566	583	611	698	739	747	872	931	983	371	160
CPU	3.70	3.92	4.29	4.02	3.75	3.03	3.08	2.85	3.07	3.12	2.05

values using 96 bits. On the other hand, the verification on the host also supports 960-bit numbers. This is sufficient at the present time because working projects for the Collatz conjecture are currently checking 61-bit numbers [23] and 72-bit numbers [1].

4.5.1 Performance for the verification of the convergence

To find the optimal size of bits for tables B and C, we evaluated the computing time of the verification for the convergence in the GPU and CPU implementations for 2^{50} and 2^{35} 64-bit numbers, respectively. Table 4.4 shows the number of verified 64-bit numbers per second for various size of base bit of tables B and C when the size of base bit of table S is 32. Note that tables B and C of base bit more than 12 cannot be stored in the shared memory due to the size limitation. According to the table, in the GPU implementation, we can find that the optimal size of bits is 18 when they are stored in the global memory. Also, in the CPU implementation, the optimal size of bits is 12. In the following, we use these parameters in the GPU and CPU implementation.

Next, we find the optimal size of bits for table S. Figures 4.2 and 4.3 show the number of verified numbers per second for various base bit of table S in the GPU and

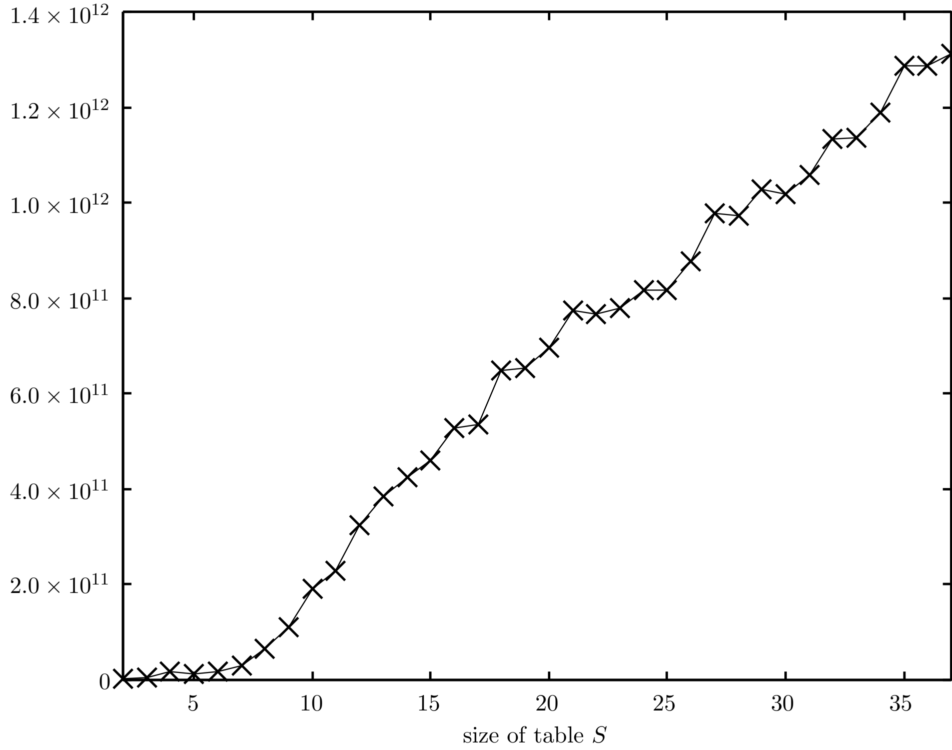

Figure 4.2: The number of verified 64-bit numbers per second for various size of base bit of table S in the GPU implementation for the convergence of the Collatz conjecture

CPU implementations, respectively. According to the both graphs, when the base bit is larger, the number is larger because the number of non-mandatory numbers is larger for larger base bit as shown in Table 4.3. Due to the size limitation, more than 37 bits for table S cannot be stored in the global memory in the GPU and the main memory in the CPU, respectively. For table S with base bit 37, our GPU implementation can verify the convergence for 1.31×10^{12} numbers per second. On the other hand, in the CPU implementation, when the base bit is larger, the verified number per second is also larger. For table S with base bit 37, the CPU implementation can verify the convergence for 5.25×10^9 numbers per second.

We note that in the GPU implementation, the computing time of the verification for

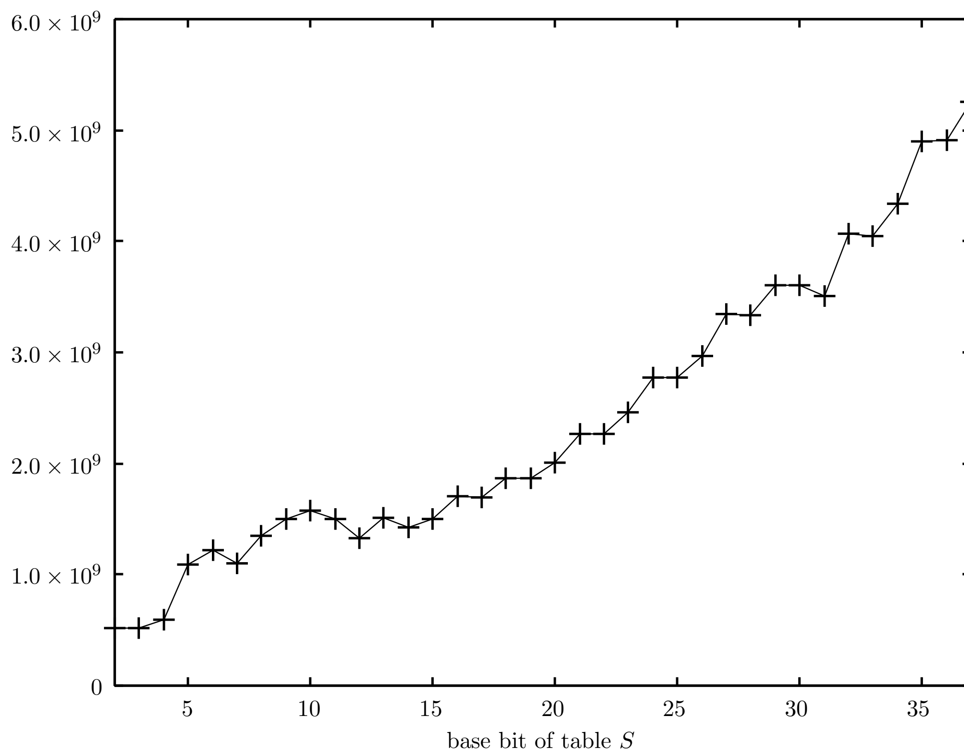

Figure 4.3: The number of verified 64-bit numbers per second for various size of base bit of table S in the CPU implementation for the convergence

overflow numbers by the CPU is included as described in Section 4.4. For example, when the convergence for table S with base bit 37 is verified, 22932 overflow numbers were found, that is, the size of interim values for 22932 numbers became more than 96 bits. After that, the host program verified the conjecture for these numbers using a quite large number of bits by software. The verification time in the CPU was 100 ms including the time of data transfer between the GPU and CPU. Since the total computing time was 853881 ms, the verification time for overflow numbers by the CPU is much shorter. Thus, our GPU implementation for the verification of the convergence of the Collatz conjecture attains speed-up factors of 249 over the CPU implementations.

There are several researches for accelerating the exhaustive verification of the convergence of Collatz conjecture using FPGAs [2, 12, 13, 14]. All of them are implementations and the basic idea of them are using table operation as same as that of our implementations. However, their implementations verify the Collatz conjecture only for the convergence. Also, as far as we know, the FPGA implementation in [14] has been the fastest implementation. However, our GPU implementation can verify the convergence of the Collatz conjecture 7.98 times faster than the FPGA implementation.

4.5.2 Performance for the verification of the delay

For the delay of the Collatz conjecture, we also find the optimal size of bits for tables B, C and J by evaluating the computing time of the GPU and CPU implementations for 2^{30} and 2^{27} 64-bit numbers, respectively. Table 4.5 shows the number of verified 64-bit numbers per second for various size of tables B, C and J when the size of base bit of table A is 25. We note that tables B, C and J of base bit more than 12 cannot be stored in the shared memory due to the size limitation. In the GPU implementation, the

Table 4.5: The number of verified 64-bit numbers ($\times 10^6$) per second for various size of base bit of tables B, C and J for the delay

size of bits	8	9	10	11	12	13	14	15	16	17	18	19	20
GPU (shared memory)	939	951	946	936	922	—	—	—	—	—	—	—	—
GPU (global memory)	924	933	870	851	407	360	383	375	397	413	393	218	134
CPU	9.26	11.2	11.2	12.6	12.4	11.5	12.3	9.05	8.09	7.97	7.85	6.08	2.97

optimal size of bits is 9 when they are stored in the shared memory. On the other hand, the optimal size of bits is 11 in the CPU implementation.

Next, we find the optimal size of bits for table A. Figures 4.4 and 4.5 show the number of verified numbers per second for various size of table A in the GPU and CPU implementations for 2^{30} and 2^{27} 64-bit numbers, respectively. Unlike the table S in the convergence, when the size of table A is larger, the verified numbers is not larger. This is because the memory access time to the table cannot be ignored since the number of access for the delay is 256 times more than that for the convergence as described in Section 4.4. Therefore, in the delay computation, the size of table A affects a trade-off between the hit ratio of the cache memory and the number of iterations of the while-loop. When the size of table A is larger, the hit ratio of the cache memory is lower and the number of iterations of the while-loop is less. On the other hand, when the size of the table is smaller, the hit ratio of the cache memory is higher and the number of iterations of the while-loop is more. This trade-off also exists in the CPU implementation. According to the graph, we can find a peak and it shows a well-balanced trade-off point between them. Thus, we can find that the optimal size of table A in the GPU and CPU implementations is 2^{12} and 2^{23}, respectively.

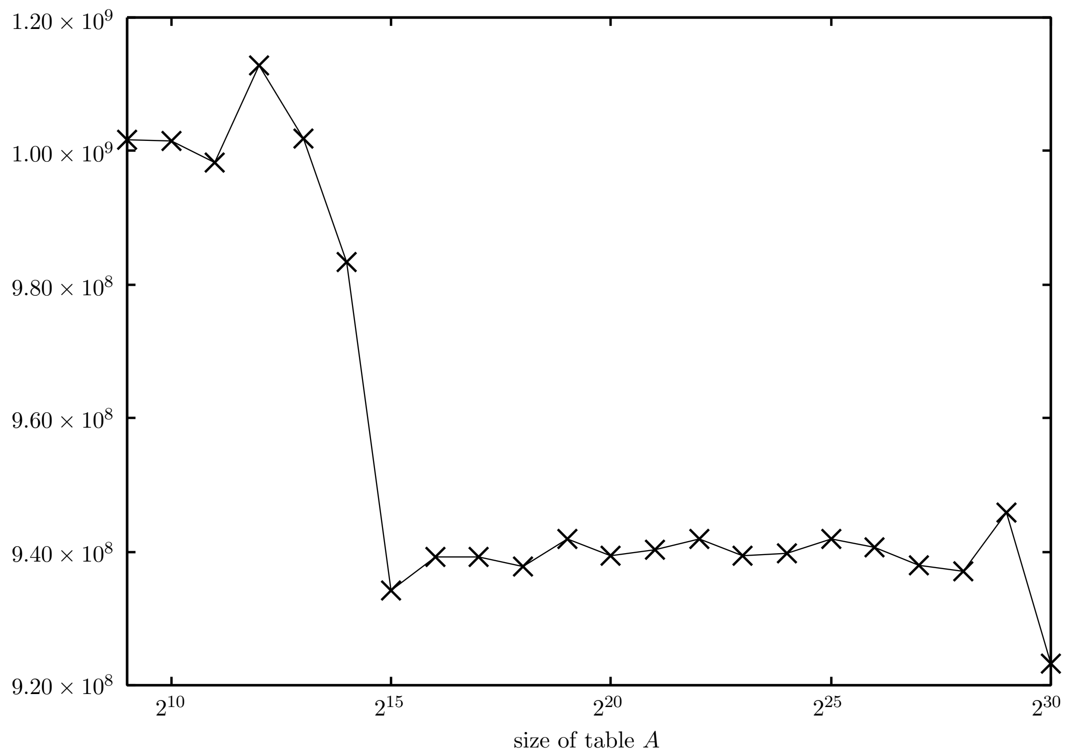

Figure 4.4: The number of verified 64-bit numbers per second for various size of base bit of table A in the GPU implementation for the delay

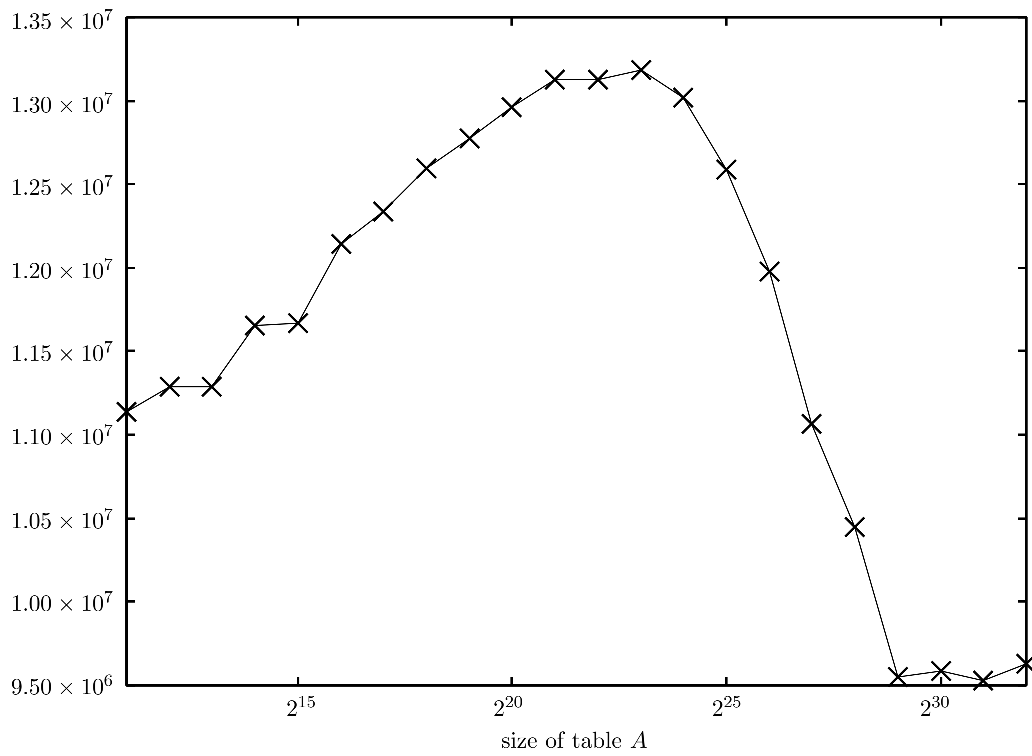

Figure 4.5: The number of verified 64-bit numbers per second for various size of base bit of table A in the CPU implementation for the delay

Using the above optimal parameters, we evaluated the computing time of the GPU and CPU implementations for 2^{30} 64-bit numbers. The results show that our GPU implementation can compute the delay for 1.01×10^9 numbers per second. It includes the computing time of the verification for overflow numbers by the CPU. On the other hand, in the CPU implementation, the CPU implementation can compute the delay for 1.39×10^7 numbers per second. Thus, our GPU implementation for the computation of the delay of the Collatz conjecture attains speed-up factors of 73 over the CPU implementation.

4.6 Concluding remarks

We have presented GPU implementations that perform the exhaustive search to verify the Collatz conjecture for the convergence and the delay. In our GPU implementation, we have considered programming issues of the GPU architecture such as the coalescing of the global memory, the shared memory bank conflict, and the occupancy of the multicore processors. We have implemented them on NVIDIA GeForce GTX TITAN X. The experimental results show that they can verify 1.31×10^{12} and 1.01×10^9 64-bit numbers per second for the convergence and the delay, respectively. On the other hand, the sequential CPU implementations verify 5.25×10^9 and 1.39×10^7 64-bit numbers per second for the convergence and the delay, respectively. Thus, our GPU implementations attain a speed-up factor of at most 249.

Chapter 5

Conclusion

In this dissertation, we proposed GPU implementations for bulk execution of multiple-length multiplication and exhaustive verification of the Collatz conjecture. We considered programming issues of the GPU architecture including warp divergence, coalesced access of the global memory, bank conflict of the shared memory, etc.

In Chapter 3, we presented a GPU implementation for bulk execution of multiple-length multiplication. We proposed Sum-rotate multiplication which is 1024-bit multiple-length multiplication method. We used one warp to perform 1024-bit multiple-length multiplication in this method and adopted warp-synchronous programming technique to avoid any synchronize operations. Also, inter-thread communication is performed by warp shuffle functions without accessing shared memory. The experimental results show that our GPU implementation on NVIDIA GeForce GTX 980 attains a speed-up factor of 52 for 1024-bit multiple-length multiplications over the sequential CPU implementation. In addition, we use sum-rotate multiplication for larger size of bits as a sub-routine. We also use Toom-Cook method to reduce the number of multiplications. Using Toom-Cook method, our GPU implementation attains a speed-up factor of 21 for

65536-bit multiple-length multiplications over the sequential CPU implementation.

In Chapter 4, we presented GPU implementations to accelerate exhaustive verification of the Collatz conjecture. We used three ideas which are a CPU-GPU cooperative approach, efficient memory access of the GPU memory, and optimization of multiplication. The experimental results show that our GPU implementations on NVIDIA GeForce GTX TITAN X can verify 1.31×10^{12} and 1.01×10^{9} 64-bit numbers per second for the convergence and the delay, respectively. On the other hand, the sequential CPU implementations on Intel Core i7-4790 verify 5.25×10^{9} and 1.39×10^{7} 64-bit numbers per second for the convergence and the delay, respectively. Thus, our GPU implementations for the convergence and the delay attain a speed-up factor of 249 and 73, respectively.

References

[1] BOINC Collatz project. http://boinc.thesonntags.com/collatz/.

[2] FengWei An and Koji Nakano. An architecture for verifying Collatz conjecture using an FPGA. In *Proc. of the International Conference on Applications and Principles of Information Science*, pages 375–378, 2009.

[3] Hovhannes Bantikyan. Big integer multiplication with CUDA FFT (cuFFT) librariy. *International Journal of Innovative Research in Computer and Communication Engineering*, 2(11):6317–6325, 2014.

[4] Paul G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, 1990.

[5] NVIDIA Corporation. CUDA ZONE. https://developer.nvidia.com/cuda-zone.

[6] NVIDIA Corporation. Whitepaper NVIDIA GeForce GTX 980 v1.1, 2014.

[7] Javier Diaz, Camelia Muñoz-Caro, and Alfonso Niño. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1369–1386, August 2012.

[8] Niall Emmart and Charles C. Weems. High precision integer multiplication with a GPU using Strassen's algorithm with multiple FFT sizes. *Parallel Processing Letters*, 21(3):359–375, 2011.

[9] Richard J. Fateman. When is FFT multiplication of arbitrary-precision polynomials practical? Technical report, University of California, Berkeley, 2006.

[10] Martin Fürer. Faster integer multiplication. *SIAM Journal on Computing*, 39(1):979–1005, 2009.

[11] Torbjörn Granlund. GNU MP: The GNU multiple precision arithmetic library. http://gmplib.org/.

[12] Shuichi Ichikawa and Naohiro Kobayashi. Preliminary study of custom computing hardware for the 3x+1 problem. In *Proc. of IEEE TENCON 2004*, pages 387–390, 2004.

[13] Yasuaki Ito and Koji Nakano. A hardware-software cooperative approach for the exhaustive verification of the Collatz conjecture. In *Proc. of International Symposium on Parallel and Distributed Processing with Applications*, pages 63–70, 2009.

[14] Yasuaki Ito and Koji Nakano. Efficient exhaustive verification of the Collatz conjecture using DSP blocks of Xilinx FPGAs. *International Journal of Networking and Computing*, 1(1):49–62, 2011.

[15] Yasuaki Ito and Koji Nakano. A GPU implementation of dynamic programming for the optimal polygon triangulation. *IEICE Transactions on Information and Systems*, E96-D(12):2596–2603, 2013.

[16] Koji Kitano and Noriyuki Fujimoto. Multiple precision integer multiplication on GPUs. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 236–242, 2014.

[17] Jeffrey C. Lagarias. The 3x+1 problem and its generalizations. *The American Mathematical Monthly*, 92(1):3–23, 1985.

[18] Duhu Man, Kenji Uda, Yasuaki Ito, and Koji Nakano. Accelerating computation of Euclidean distance map using the GPU with efficient memory access. *International Journal of Parallel, Emergent and Distributed Systems*, 28(5):383–406, 2013.

[19] Takatoshi Nakayama. The CUDA multiple precision arithmetic library (CUMP) version 1.0.1. https://github.com/skystar0227/CUMP.

[20] NVIDIA Corporation. *Parallel Thread Execution ISA Version 3.2*, 2013.

[21] NVIDIA Corporation. *CUDA C Programming Guide Version 7.0*, 2015.

[22] NVIDIA Corporation. *Tuning CUDA applications for Kepler Version 7.0*, 2015.

[23] Eric Roosendaal. On the 3x + 1 problem. http://www.ericr.nl/wondrous/index.html.

[24] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proc. of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, 2008.

[25] Tomás Oliveira e Silva. Maximum excursion and stopping time record-holders for the $3x + 1$ problem: Computational results. *Mathematics of Computation*, 68(225):371–384, 1999.

[26] Daisuke Takafuji, Koji Nakano, and Yasuaki Ito. A CUDA C program generator for bulk execution of a sequential algorithm. In *Proc. of International Conference on Algorithms and Architectures for Parallel Processing*, pages 178–191, Aug. 2014.

[27] Yuji Takeuchi, Daisuke Takafuji, Yasuaki Ito, and Koji Nakano. ASCII art generation using the local exhaustive search on the GPU. In *Proc. of International Symposium on Computing and Networking*, pages 194–200, 2013.

[28] Andrei L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics Doklady*, (3):714–716, 1963.

[29] Akihiro Uchida, Yasuaki Ito, and Koji Nakano. Accelerating ant colony optimisation for the travelling salesman problem on the GPU. *International Journal of Parallel, Emergent and Distributed Systems*, 29(4):401–420, 2014.

[30] Eric W. Weisstein. Collatz problem. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/CollatzProblem.html.

[31] Kaiyong Zhao and Xiaowen Chu. GPUMP: a multiple-precision integer library for GPUs. In *Proc. of 2010 IEEE 10th International Conference on Computer and Information Technology*, pages 1164–1168, 2010.

Acknowledgment

First and foremost, I would like to show my deepest gratitude to my supervisor, Professor Koji Nakano for his continuous encouragement, advice and support. He is a respectable and resourceful scholar. His knowledge and research experience are in value through the whole period of my study. As a supervisor, he taught me skills and practices that will benefit my future research career.

I shall express sincere appreciation to my thesis committee members, Professor Satoshi Fujita and Associate Professor Yasuaki Ito for reviewing my dissertation.

I would also like to express my thanks to Assistant Professor Daisuke Takafuji for his support and continuous guidance in every stage of my study. My heartiest thanks go to all members of computer system laboratory. They were always kind and very keen to help.

I would express thanks to all the faculty members of the Department of Information Engineering of Hiroshima University.

Last but not least, I wish to express my thanks to my parents who has always supported me.

List of publications

Journals

[J-1] Takumi Honda, Yasuaki Ito, and Koji Nakano, GPU-Accelerated Bulk Execution of Multiple-Length Multiplication with Warp-Synchronous Programming Technique, IEICE Transactions on Information and Systems, Vol.E99-D, No.12, pp.3004–3012, December 2016.

[J-2] Takumi Honda, Yasuaki Ito, and Koji Nakano, GPU-accelerated Exhaustive Verification of the Collatz Conjecture, International Journal of Networking and Computing, Vol.7, No.1, pp.69–85, January 2017.

International Conferences

[C-1] Takumi Honda, Yasuaki Ito, and Koji Nakano, GPU-accelerated Verification of the Collatz Conjecture, Proc. of International Conference on Algorithms and Architectures for Parallel Processing(ICA3PP, LNCS 8630), pp.483–496, August 2014.

[C-2] Takumi Honda, Yasuaki Ito, Koji Nakano, A Warp-synchronous Implementation for Multiple-length Multiplication on the GPU, Proc. of International Symposium on Computing and Networking, pp.96–102, December 2015.