A Warp-synchronous Implementation for Multiple-length Multiplication on the GPU

Takumi Honda, Yasuaki Ito, Koji Nakano
Department of Information Engineering,
Hiroshima University
Kagamiyama 1-4-1, Higashi-Hiroshima, 739-8527 Japan
Email: {honda, yasuaki, nakano}@cs.hiroshima-u.ac.jp

***Abstract*—If we process large-integers on the computer, they are represented by multiple-length integer. Multiple-length multiplication is widely used in areas such as scientific computation and cryptography processing. However, the computation cost is very high since CPU does not support a multiple-length integer. In this paper, we present a GPU implementation of bulk multiple-length multiplications. The idea of our GPU implementation is to adopt warp-synchronous programming. 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, 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. The experimental results show that our GPU implementation on NVIDIA GeForce GTX 980 attains a speed-up factor of 62 for 1024-bit multiple-length multiplication over the single CPU implementation.**

***Keywords*-Multiple-length multiplication; GPU; GPGPU; Parallel processing; warp-synchronous programming**

I. Introduction

Recent Graphics Processing Units (GPUs), which have a lot of processing units, can be used for general purpose parallel computation. Since GPUs have very high memory bandwidth, the performance of GPUs greatly depends on memory access. CUDA (Compute Unified Device Architecture) [1] is the architecture for general purpose parallel computation on GPUs. Using CUDA, we can develop parallel algorithms to be implemented in GPUs. Therefore, many studies have been devoted to implement parallel algorithms using CUDA [2], [3], [4], [5].

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 number 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 [6], and computational science [7]. Since multiple-length numbers of size thousands to several tens of thousands bits are used in such applications, the acceleration of the computation of their multiplications is in great demand. Also, considering practical cases, a large number of multiplications are usually computed. Therefore, in this work, we target at the computation for many multiple-length multiplications of such size.

Main contribution of this paper is to present an implementation of multiple-length multiplication optimized for CUDA-enabled GPUs. The idea of our GPU implementation is to adopt warp-synchronous programming. 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, 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 multiplication method as a sub-routine. The experimental results show that our GPU implementation on NVIDIA GeForce GTX 980 attains a speed-up factor of 62 for 1024-bit multiple-length multiplication over the single CPU implementation.

In sequential implementation, we can utilize software libraries that support multiple-length arithmetic operations such as GMP (GNU Multiple Precision Arithmetic Library) [8]. A sequential CPU implementation with this library is used to compare the performance of our proposed GPU implementation. On the other hand, there are GPU implementations to accelerate multiple-length multiplications. In paper [9], [10], GPU implementations of very large

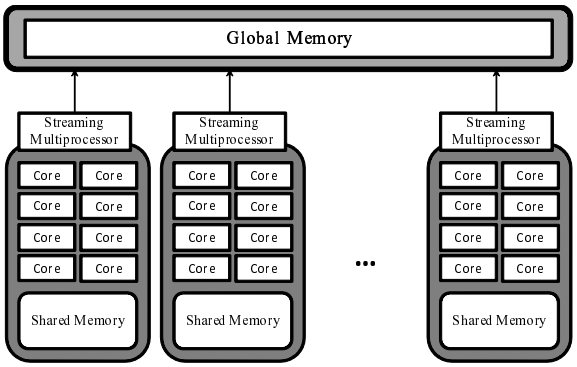


Figure 1: CUDA hardware architecture

integer multiplications using FFT are shown. Zhao *et al.* proposed multiple-length multiplication on the GPU as one of library functions [11]. This implementation is based on School method that is naive multiplication. Kitano *et al.* proposed a GPU implementation of parallel multiple-length multiplication also based on School method. In the implementation, load of each thread is equalized by reordering the computation of partial products.

The rest of this paper is organized as follows. Section II provides an overview of the GPU architecture. Section III describes multiple-length multiplication methods. This section also by reviewing the warp shuffle functions considered in this work. In Section IV, our GPU implementation of multiple-length multiplication using warp synchronize programming is proposed. Experimental results are shown in Section V. Finally, Section VI concludes the paper.

II. GPU Implementation

We briefly explain CUDA architecture that we will use. Figure 1 illustrates the CUDA hardware architecture. CUDA uses three types of memories in the NVIDIA GPUs: *the global memory*, *the shared memory*, and *the registers* [12]. 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, 16-64 Kbytes. The registers in CUDA are placed on each core in the multiprocessor and the fastest memory, that is, no latency is necessary. However, the size of the registers is the smallest during them. The efficient usage of the global memory and the shared memory is a key for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *the coalescing* of the global memory access and *the bank conflict* of the shared memory access [13], [14]. To maximize the bandwidth between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed in the same time. Thus, threads should perform coalescing access when they access to the global memory.

CUDA parallel programming model has a hierarchy of thread groups called *grid*, *block* and *thread*. A single grid is organized by multiple blocks, each of which has equal number of threads. The blocks are allocated to streaming processors such that all threads in a block are executed by the same streaming processor in parallel. All threads can access to the global memory. However, as we can see in Figure 1, threads in a block can access to the shared memory of the streaming processor to which the block is allocated. Since blocks are arranged to multiple streaming processors, threads in different blocks cannot share data in shared memories. Also, the registers are only accessible by a thread, that is, the registers cannot be shared by multiple threads.

CUDA C extends C language by allowing the programmer to define C functions, called *kernels*. By invoking a kernel, all blocks in the grid are allocated in streaming processors, and threads in each block are executed by processor cores in a single streaming processor. In the execution, threads in a block are split into groups of thread called *warps*. A wrap 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. 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. When one warp is paused or stalled, other warps can be executed to hide latencies and keep the hardware busy.

There is a metric, called *occupancy*, related to the number of active warps on a streaming processor. The occupancy is the ratio of the number of active warps per streaming processor to the maximum number of possible active warps. It is important in determining how effectively the hardware is kept busy. The occupancy depends on the number of registers, the numbers of threads and blocks, and the size of shard memory used in a block. Namely, utilizing too many resources per thread or block may limit the occupancy. To obtain good performance with the GPUs, the occupancy should be considered.

The kernel calls terminates, when threads in all blocks finish the computation. Since all threads in a single block are executed by a single streaming processor, the barrier synchronization of them can be done by calling CUDA C

`syncthreds()` 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 terminates, 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.

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 [12]. 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(a)). 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(`$x, (id+1)\%w$`)`. The function call performs data transfer like right circular shift between threads as illustrated in Figure 2(b). In the similar way, the `shfl(`$x, (id+w-1)\%w$`)` allows to transfer data like left right circular shift (Figure 2(c)). 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.

Warp synchronous programming [15] is a parallel programming model such that one warp is used as an execution unit. The characteristic of this model is that any synchronous operations are not necessary. Usually, it is costly to synchronize execution of threads and communicate within threads. In our GPU implementation shown in the following sections, we adopt warp synchronous programming. Also, inter-thread communication is performed by warp shuffle functions without accessing shared memory.

III. 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 d be the number of b-bit words. Therefore, $d = \lceil \frac{R}{r} \rceil$. For example, a 1024-bit integer consists of 32 words. Next, we will introduce several multiplication methods for such multiple-length numbers.

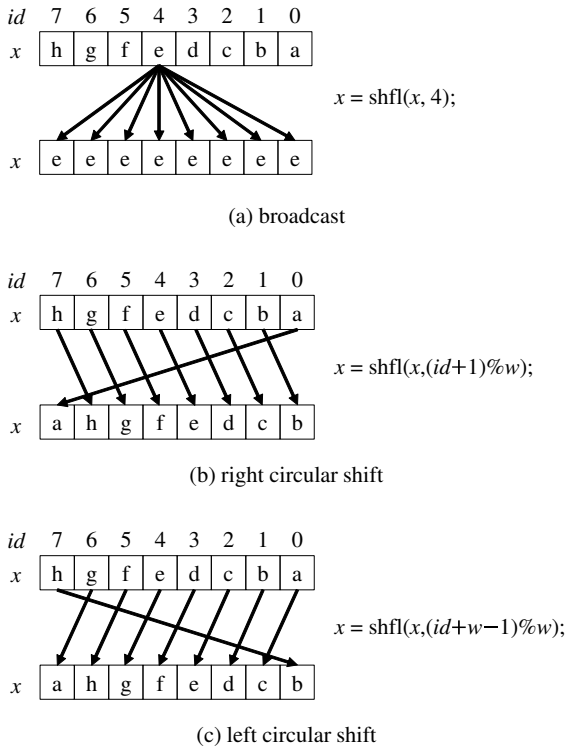
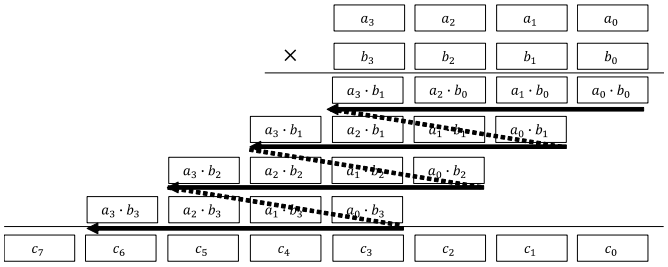


Figure 2: Example of intra-warp data exchange using warp shuffle functions

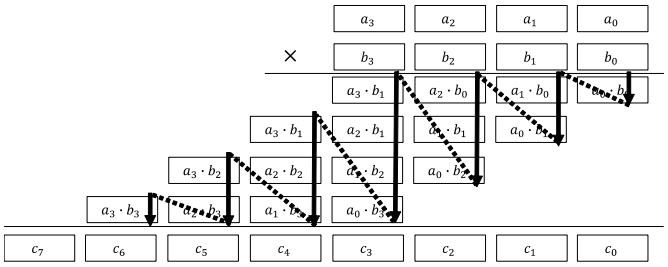
A. Multiple-Length Multiplication

Suppose A and B represent two multi-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(a). 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(a). As illustrated in the figure, calculation of School method is performed 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.

To avoid storing the partial products, *Comba method* [16] 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



(a) School method



(b) Comba method

Figure 3: The order of word-wise multiplication for multiple-length numbers $C = A \cdot B$

same as illustrated in Figure 3. More specifically, calculation of Comba method is performed 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 I 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.

Algorithm 1 School method

Input: $A = (a_{w-1}, \ldots, a_1, a_0), B = (b_{w-1}, \ldots, b_1, b_0)$
Output: $C = AB$

```
1: C ← 0
2: for j = 0 to w − 1 do
3:     {u, v} ← 0
4:     for i = 0 to w − 1 do
5:         {u, v} ← a_i b_j + c_{i+j} + u
6:         c_{i+j} ← v
7:     end for
8:     c_{2w+i} ← u
9: end for
```

Karatsuba method [17] is an algorithm for multiplying two numbers that reduce the number of multiplications compared with School method and Comba method. Let us consider multiple-length multiplication for $C = AB$, where A and B are multiple-length numbers of size R bits each. The two

Algorithm 2 Comba method

Input: $A = (a_{w-1}, \ldots, a_1, a_0), B = (b_{w-1}, \ldots, b_1, b_0)$
Output: $C = AB$

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

numbers A and B are divided into two parts of size $\frac{R}{2}$ bits each such that $A = A_1 \cdot 2^{\frac{R}{2}} + A_0$ and $B = B_1 \cdot 2^{\frac{R}{2}} + B_0$. The product C is computed as follows:

$$\begin{aligned} C &= AB \\ &= (A_1 \cdot 2^{\frac{R}{2}} + A_0)(B_1 \cdot 2^{\frac{R}{2}} + B_0) \\ &= A_1B_1 \cdot 2^R + (A_1B_0 + A_0B_1) \cdot 2^{\frac{R}{2}} + A_0A_0 \end{aligned}$$

In School method and Comba method, there are 4 multiplications $A_0 \times B_0$, $A_1 \times B_0$, $A_0 \times B_1$, and $A_1 \times B_1$. On the other hand, in Karatsuba method, the sum of two multiplications in the second term $A_1B_0 + A_0B_1$ is modified as follows:

$$A_1B_0 + A_0B_1 = (A_1 + A_0)(B_1 + B_0) - A_1B_1 - A_0B_0$$

In this deformation, computing two products A_1B_1 and A_0B_0 beforehand, there are three multiplications $A_1 \times B_1$, $A_0 \times B_0$ and $(A_1 + A_0) \times (B_1 + B_0)$, though the number of addition/subtraction is increased. Thus, Karatsuba method can reduce the number of multiplications from four to three. This idea to the partial products can be applied recursively. Therefore, in Table— I, the number of multiplications and memory access is shown when Karatsuba method is applied once and then Comba method is used for smaller size of multiplications. Also, "Karatsuba^2" in the table represents a method such that Karatsuba method is applied twice and then Comba method is used. According to the table, when Karatsuba method is used, the number of multiplications is reduced. On the other hand, the number of memory access is increased. In the GPU, the latency of memory access is much longer than that of 32/64 bits multiplication. Therefore, in our implementation, Karatsuba method is not used recursively.

Table I: The number of multiplications and memory read/write for multiplying two w-word numbers

method	multiplication	memory read	memory write
School	w^2	$2w^2$	w^2+w
Comba	w^2	$2w^2-2$	$2w$
Karatsuba	$\frac{3}{4}w^2$	$\frac{3}{2}w^2+\frac{17}{2}w-2$	$\frac{15}{2}w+4$
Karatsuba^2	$\frac{9}{16}w^2$	$\frac{9}{4}w^2+20w-4$	$\frac{31}{2}w+14$

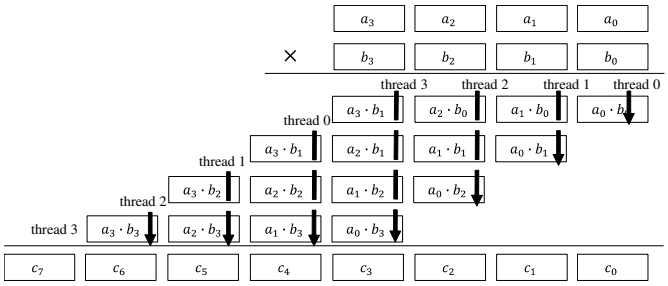


Figure 4: Parallel column-based multiplication

IV. Parallel multiple-length multiplication for the GPU

This section presents the main contribution of this work. We adopt warp-synchronous programming to the proposed parallel multiple-length multiplication. In the following, w threads, that 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 any use 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 Section II. More specifically, data exchange methods, broadcast and right/left circular shift, as shown in Figure 2 using warp shuffle function `shfl()` are utilized. The details of the parallel algorithm are presented next.

In the proposed approach, a product $C = (c_{2w-1}, \ldots, c_1, c_0)$ of two w-word numbers $A = (a_{w-1}, \ldots .a_1, a_0)$ and $B = (b_{w-1}, \ldots, b_1, b_0)$ 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 4. 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 Section II, 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 column. More specifically, when w threads (thread 0, thread 1, ..., thread $w-1$) are launched, thread i computes partial products $a_ib_0, a_ib_1, \ldots, a_ib_{w-1}$ for each column as illustrated in Figure 5. Using this assignment of threads, almost all operations are the same between threads, that is, warp divergence can be avoided mostly.

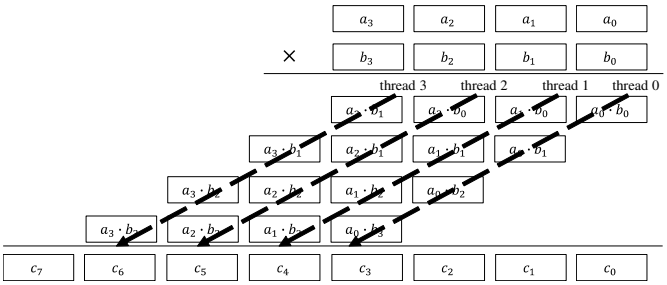


Figure 5: Sum-rotate multiplication

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 Section II. in each row, thread 0 obtains the final product of c_j. According to Figure 5, a thread assigned to the lowest digits can obtain the lower words of the final product $c_0, \ldots, c_{w-1}$ for each row. On the other hand, the upper words of $c_w, \ldots, c_{2w-1}$ 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 5. 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 5, 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 $c_0, \ldots, c_{w-1}$, respectively. At that time, the upper digits $c_w, \ldots, c_{2w-1}$ 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 [12]. 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 coalescing access in lines 20 and 21.

Algorithm 3 Sum-rotate multiplication using a warp

Input: $A = (a_{w-1}, \ldots, a_1, a_0), B = (b_{w-1}, \ldots, b_1, b_0)$
Output: $C = AB$

1: $i \leftarrow id\ (= 0, 1, \ldots, w-1)$
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: $\quad b' \leftarrow \texttt{shfl}(b_j, j)$ ▷ Broadcast b_j from thread j
7: $\quad \{t, u, v\} \leftarrow a \cdot b' + \{u, v\}$
8: $\quad v \leftarrow \texttt{shfl}(v, (i+1)\%w)$ ▷ Right circular shift v
9: $\quad c' \leftarrow \texttt{shfl}(c', (i+1)\%w)$ ▷ Right circular shift c'
10: $\quad$ **if** $id = w-1$ **then**
11: $\quad\quad c' \leftarrow v$
12: $\quad\quad v \leftarrow 0$
13: $\quad$ **end if**
14: $\quad \{u, v\} \leftarrow \{t, u\} + v$
15: **end for**
16: **while** $\texttt{any}(u) \neq 0$ **do** ▷ Loop for carry propagation
17: $\quad u \leftarrow \texttt{shfl}(u, (i+w-1)\%w)$ ▷ Left circular shift u
18: $\quad \{u, v\} \leftarrow u + v$
19: **end while**
20: $c_i \leftarrow c'$
21: $c_{i+w} \leftarrow v$

V. 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 GTX 980, which has 2048 cores running on 1.216MHz [18]. 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.

First, we evaluate the performance of the multiplication methods on the GPU. We have also implemented the single thread implementation such that each thread computes one multiplication. This implementation is based on the idea proposed in [19]. 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. 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 II shows the computing time when 100000 multiple-length multiplications are computed. Note that "Karatsuba^2" in the table denotes a multiplication method such that Karatsuba method is recursively applied twice. 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. Regarding multiplication methods in the one warp implementation with warp shuffle functions, for no more than 8192 bits, Comba method is faster and for more, Karatsuba method is faster than the other methods. Karatsuba^2 method is slower since the overhead such as the number of additional additions cannot be ignored. According to the results, the best configuration for the size of operands is selected such that Comba method is used for 1024 to 8192 bits and Karatsuba method is used for 16384 to 32768 bits.

We have also used Intel PC using Xeon X7460 running on 2.6GHz to evaluate the implementation by sequential algorithms. In the CPU implementation, we have utilized GMP version 4.1.4. Table III shows the comparison between CPU and GPU implementations for the computing time in milliseconds when 100000 multiple-length multiplications are computed. The best configuration in the above has been used in the GPU implementation. Using the proposed GPU implementation, the computing time can be reduced by a factor of 18.71 to 62.88.

Table III: The comparison between CPU and GPU implementations for the computing time in milliseconds when 100000 multiple-length multiplications are computed

# of bits	1024	2048	4096	8192	16384	32768
CPU	95.34	315.65	1031.40	3254.28	10505.87	30928.49
GPU	1.52	7.50	26.16	103.00	435.23	1653.45
Speed-up	62.88	42.10	39.43	31.59	24.14	18.71

VI. Conclusion

In this paper, we have presented a GPU implementation of bulk multiple-length multiplications. The idea of our GPU implementation is to adopt warp-synchronous programming. Using this idea, we have proposed Sum-rotate multiplication

Table II: The computing time of GPU implementations in milliseconds for 100000 multiple-length multiplications

execution unit	multiplication method	# of bits 1024	2048	4096	8192	16384	32768
single thread	Comba	5.06	38.96	165.56	684.61	2806.77	11411.26
	Karatsuba	6.59	21.62	80.22	320.46	1288.00	5333.75
	Karatsuba^2	5.62	16.54	58.02	219.59	867.14	3896.67
one warp (32 threads) with shared memory	Comba	1.99	8.28	32.16	125.84	501.04	2078.12
	Karatsuba	—	15.64	45.56	150.34	560.37	2057.02
	Karatsuba^2	—	—	60.24	164.61	589.34	2331.08
one warp (32 threads) with warp shuffle functions	Comba	**1.52**	**7.50**	**26.16**	**103.00**	440.38	1729.08
	Karatsuba	—	12.08	35.48	118.73	**435.23**	**1653.45**
	Karatsuba^2	—	—	53.38	144.79	490.27	1819.33

of two 1024-bit numbers. 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 62 for 1024-bit multiple-length multiplication over the single CPU implementation using GNU multiple precision arithmetic library.

REFERENCES

[1] NVIDIA Corporation, "CUDA ZONE." http://www.nvidia.com/page/home.html.

[2] J. Diaz, C. Muñoz-Caro, and A. Niño, "A survey of parallel programming models and tools in the multi and many-core era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, pp. 1369–1386, August 2012.

[3] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, "Implementations of parallel computation of Euclidean distance map in multicore processors and GPUs," in *Proceedings of International Conference on Networking and Computing*, pp. 120–127, 2010.

[4] K. Ogawa, Y. Ito, and K. Nakano, "Efficient Canny edge detection using a GPU," in *International Workshop on Advances in Networking and Computing*, pp. 279–280, Nov. 2010.

[5] Z. Wei and J. JaJa, "Optimization of linked list prefix computations on multithreaded GPUs using CUDA," in *Proceedings of International Parallel and Distributed Processing Symposium*, 2010.

[6] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*. Chapman & Hall/CRC Cryptography and Network Security Series, CRC Press, 2nd ed., 2014.

[7] L. M. Vandersypen, M. Steffen, G. Breyta, C. S. Yannoni, M. H. Sherwood, and I. L. Chuang, "Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance," *Nature*, vol. 414, no. 6566, pp. 883–887, 2001.

[8] T. Granlund, "GNU MP: The GNU multiple precision arithmetic library." http://gmplib.org/.

[9] N. Emmart and C. C. Weems, "High precision integer multiplication with a GPU using Strassen's algorithm with multiple FFT sizes," *Parallel Processing Letters*, vol. 21, no. 3, pp. 359–375, 2011.

[10] H. Bantikyan, "Big integer multiplication with CUDA FFT (cuFFT) library," *International Journal of Innovative Research in Computer and Communication Engineering*, vol. 2, no. 11, pp. 6317–6325, 2014.

[11] K. Zhao and X. Chu, "GPUMP: A multiple-precision integer library for GPUs," in *Proc. of 2010 IEEE 10th International Conference on Computer and Information Technology*, pp. 1164–1168, 2010.

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

[13] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, "Implementations of a parallel algorithm for computing Euclidean distance map in multicore processors and GPUs," *International Journal of Networking and Computing*, vol. 1, pp. 260–276, July 2011.

[14] NVIDIA Corporation, *CUDA C Best Practice Guide Version 7.0*, 2015.

[15] NVIDIA Corporation, *Tuning CUDA Applications for Kepler Version 7.0*, 2015.

[16] P. G. Comba, "Exponentiation cryptosystems on the IBM PC," *IBM Systems Journal*, vol. 29, no. 4, pp. 526–538, 1990.

[17] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," *Doklady Akademii Nauk SSSR*, vol. 145, no. 2, pp. 293–294, 1962.

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

[19] D. Takafuji, K. Nakano, and Y. Ito, "C2CU: CUDA C program generator for bulk execution of a sequential algorithm," in *Proc. of International Conference on Algorithms and Architectures for Parallel Processing*, pp. 178–191, 2014.