International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 7, Number 1, pages 69–85, January 2017

GPU-accelerated Exhaustive Verification of the Collatz Conjecture[1]

Takumi Honda, Yasuaki Ito, and Koji Nakano
Department of Information Engineering, Hiroshima University
Kagamiyama 1-4-1, Higashi-Hiroshima, Hiroshima, 739-8527 JAPAN

Received: April 6, 2016
Revised: June 15, 2016
Accepted: July 4, 2016
Communicated by Jacir L. Bordim

Abstract

The main contribution of this paper is to present an implementation that performs the exhaustive search to verify the Collatz conjecture using a GPU. Consider the following operation 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 m, repeated iteration of the operations eventually produces the value 1. We have implemented it 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 a similar idea, we achieved a speed-up factor of 73.

Keywords: Collatz conjecture, GPGPU, Parallel processing, Exhaustive verification, Coalesced access, Bank conflict

1 Introduction

The Collatz conjecture is a well-known unsolved conjecture in mathematics [9, 15, 17, 20]. Consider the following operation 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 follows:

[1]The preliminary version of this paper has been presented at the 14th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2014) [4].

```
for m ← 1 to ∞ do
  begin
    n ← m
    while(n > 1) do
      if n is even then n ← n/2
      else n ← 3n + 1
  end
```

Clearly, if the Collatz conjecture 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 [15] and 72-bit numbers [1]. The project in [1] 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 [15]. 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 [15], 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 $D(3) = 7$.

The main contribution of this paper is to further accelerate the exhaustive verification for the Collatz conjecture using a GPU (Graphics Processing Unit). Recent GPUs can be utilized for general purpose parallel computation. We can use many processing units connected with an off-chip global memory in GPUs. CUDA (Compute Unified Device Architecture) [12] is an architecture for general purpose parallel computation on GPUs. Using CUDA, we can develop parallel processing programs to be implemented in GPUs. Therefore, many studies have been devoted to implement parallel algorithms using CUDA [3, 8, 10, 16, 18, 19]. 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.

By effective use of a GPU and the above ideas, our new GPU implementation can verify 1.31×10^{12} and 1.01×10^9 64-bit numbers per second for the convergence and the delay, respectively.

This paper is organized as follows. Section 2 provides an overview of related work. Section 3 presents several techniques for accelerating the verification of the Collatz conjecture. In Section 4, we show the GPU and CUDA architectures to understand our idea. Section 5 proposes our new ideas to implement the verification of the Collatz conjecture on the GPU. The experimental results are shown in Section 6. Finally, Section 7 offers concluding remarks.

2 Related work

There are several researches for accelerating the exhaustive verification of the Collatz conjecture. It is known [2, 5, 6, 7] 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, 5, 6, 7], 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 [5], 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 [15].

In [6], a software-hardware cooperative approach to verify the Collatz conjectures 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 [7], 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 [6], the coprocessor can perform the exhaustive verification faster than the above FPGA implementations.

In our conference version of this paper [4], we have shown that the verification for the Collatz conjecture runs 1.80×10^9 64-bit numbers per second. Note that, this implementation of the conference version is slower than this journal version paper, because in this work we used a different GPU that is a state-of-the-art architecture of NVIDIA GPUs and optimized implementations for the GPU. Additionally, we added a GPU acceleration of computation for the delay.

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 [9, 20] and the details of them are shown, as follows.

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 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 paper, 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 hold. 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 1. For the benefit of readers, we show tables B, C, and S for 4 base bits in Table 1. From $s_4 = 3$, we have 3 mandatory least significant bits out of 16.

For the reader's benefit, Table 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 3 shows the size of table S. It further shows the ratio

Algorithm 1 Verification algorithm for convergence of Collatz conjecture

for $m_H \leftarrow 1$ **to** $+\infty$ **do**
  for $i \leftarrow 1$ **to** $s_d - 1$ **do**
    $m_L \leftarrow S[i]$
    $n \leftarrow m \leftarrow 2^d m_H + m_L$
    while $n \geq m$ **do**
      Let n_L be the least significant d bits and n_H be the remaining bits.
      $n \leftarrow B[n_L] \cdot n_H + C[n_L]$
    end while
  end for
end for

Table 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 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

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.

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 2 shows an algorithm to count delay of Collatz conjecture using the above ideas.

4 GPU and CUDA architectures

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* [14]. The global memory

Table 3: The size of table S

base bit	words	ratio	base bit	words	ratio
3	2	0.2500	21	46611	0.0222
4	3	0.1875	22	93222	0.0222
5	4	0.1250	23	168807	0.0201
6	8	0.1250	24	286581	0.0171
7	13	0.1016	25	573162	0.0171
8	19	0.0742	26	1037374	0.0155
9	38	0.0742	27	1762293	0.0131
10	64	0.0625	28	3524586	0.0131
11	128	0.0625	29	6385637	0.0119
12	226	0.0552	30	12771274	0.0119
13	367	0.0448	31	23642078	0.0110
14	734	0.0448	32	41347483	0.0096
15	1295	0.0395	33	82694966	0.0096
16	2114	0.0323	34	151917639	0.0088
17	4228	0.0323	35	263841377	0.0077
18	7495	0.0286	36	527682754	0.0077
19	14990	0.0286	37	967378591	0.0070
20	27328	0.0261			

Algorithm 2 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: $n' \leftarrow B[n'_L] \cdot n'_H + C[n'_L]$
7: $D(n) \leftarrow D(n) + J[n'_L]$
8: **end while**
9: $D(n) \leftarrow D(n) + A[n']$
10: **end for**

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, 16-96 Kbytes. The registers in CUDA are placed on each core in the streaming 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 [11]. 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 coalesced access when they access to the global memory. Also, CUDA supports broadcast access to the shared memory without the bank conflict [14]. The broadcast access is a shared memory request such that two or more threads refer the same address. Thus, in our GPU implementation, to make memory access efficient, we perform the coalescing and the broadcast access for the reference to tables B and C stored in the global memory and the shared memory as possible, respectively.

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 multiprocessors such that all threads in a block are executed by the same streaming multiprocessor in parallel. All threads can access to the global memory.

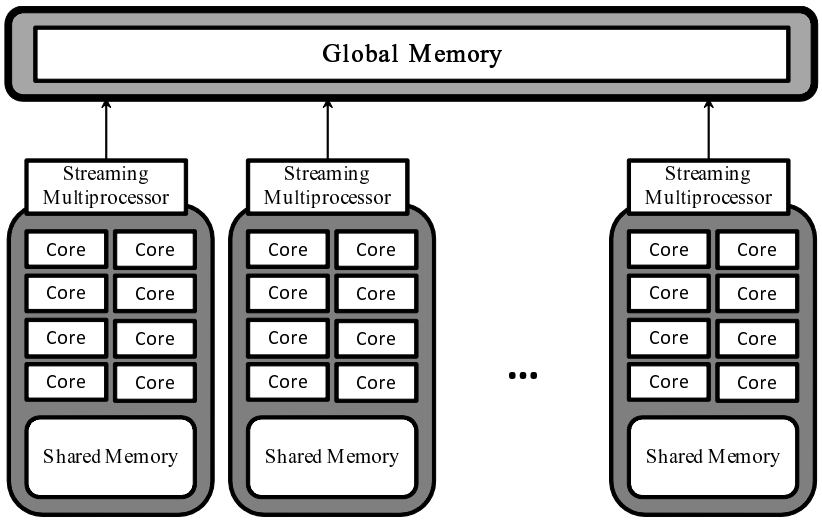

Figure 1: CUDA hardware architecture

However, as we can see in Figure 1, threads in a block can access to the shared memory of the streaming multiprocessor to which the block is allocated. Since blocks are arranged to multiple streaming multiprocessors, 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 multiprocessors, and threads in each block are executed by processor cores in a single streaming multiprocessor. In the execution, threads in a block are split into groups of threads called *warps*. 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. 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 multiprocessor. The occupancy is the ratio of the number of active warps per streaming multiprocessor 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 shared 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.

As we have mentioned, the coalesced access to the global memory is a key issue to accelerate the computation. As illustrated in Figure 2, when threads access to continuous locations in a row of a two-dimensional array (*horizontal access*), the continuous locations in address space of the global memory are accessed in the same time (*coalesced access*). However, if threads access to continuous locations in a column (*vertical access*), the distant locations are accessed in 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. In addition, when all threads in a warp read a word from the same address, the memory access, that is broadcast access, is performed as the coalescing. In our GPU implementation, broadcast access to the global memory is used as possible.

Just as the global memory is divided into several partitions, shared memory is also divided into 32 equally-sized modules of 32-(or 64-)bit width, called banks (Figure 3). In the shared memory, the successive 32-(or 64-)bit words are assigned to successive banks. To achieve maximum throughput,

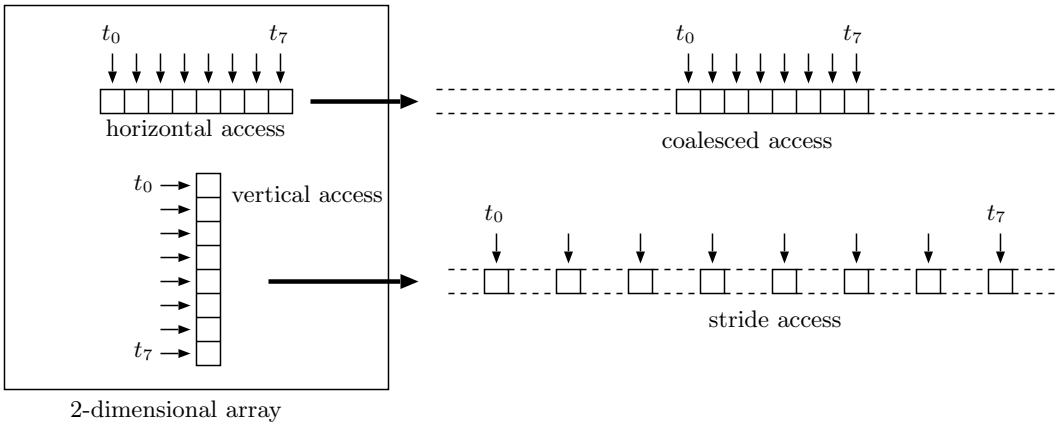

Figure 2: Coalesced and stride access

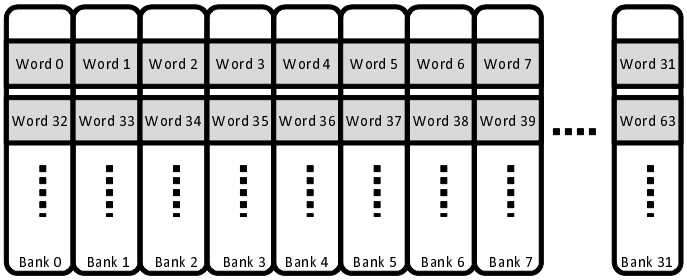

Figure 3: The structure of the shared memory

either concurrent threads of a thread block should access different banks or when all threads access the same address, the value is only read once and broadcasted to all threads, otherwise, bank conflicts will occur. In practice, the shared memory can be used as a cache to hide the access latency of the global memory. Therefore, for the shared memory access in our GPU implementation, the broadcast access is used as possible.

5 GPU implementation

The main purpose of this section is to show a GPU implementation 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.

5.1 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 [6]. In this paper, 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 [15]. 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. The number of bits supported by our implementation on the host is 960 bits. Also, there was no 64-bit number that the maximum size of the interim value was larger 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 [7]. Therefore, it is reasonable to perform the verification for overflow numbers on the host. In Section 6, we will evaluate the number of overflows and the verification time for them.

5.2 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. 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 1 and 2 in Section 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, 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.

5.3 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

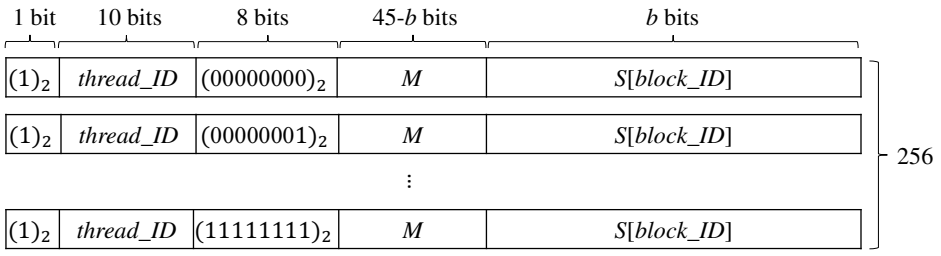

Figure 4: The data format of 64-bit numbers verified by each thread in a block, where *thread_ID* denotes a thread index within a block, *block_ID* denotes a block index within a kernel, and M is a constant.

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 [13] 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.

5.4 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 2 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 2. 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.

6 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 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.

Table 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

We have used in Intel Core i7-4790 running in 3.66GHz 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. For this purpose, we have randomly generated integers as a constant M.

Regarding the size of verified numbers, our GPU implementation computes interim 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 [15] and 72-bit numbers [1].

6.1 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 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 5 and 6 show the number of verified numbers per second for various base bit of table S in the GPU and 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 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 overflow numbers by the CPU is included as described in Section 5. 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, 5, 6, 7]. 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

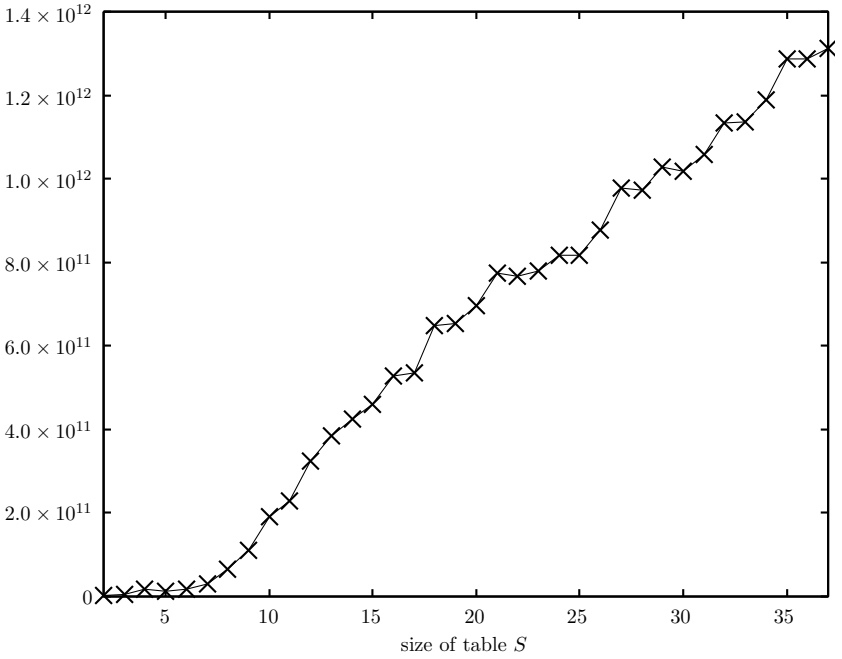

Figure 5: The number of verified 64-bit numbers per second for various size of base bit of table S in the GPU implementation for the convergence of the Collatz conjecture

Table 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

implementation in [7] has been the fastest implementation. However, our GPU implementation can verify the convergence of the Collatz conjecture 7.89 times faster than the FPGA implementation.

6.2 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 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 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 7 and 8 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 5. 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

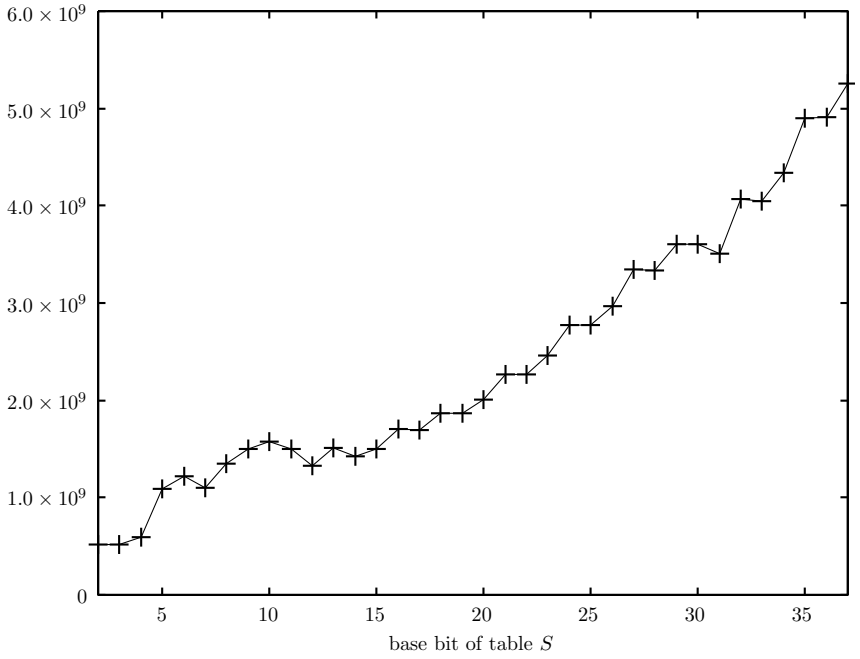

Figure 6: The number of verified 64-bit numbers per second for various size of base bit of table S in the CPU implementation for the convergence

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.

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 implementations.

7 Conclusions

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.

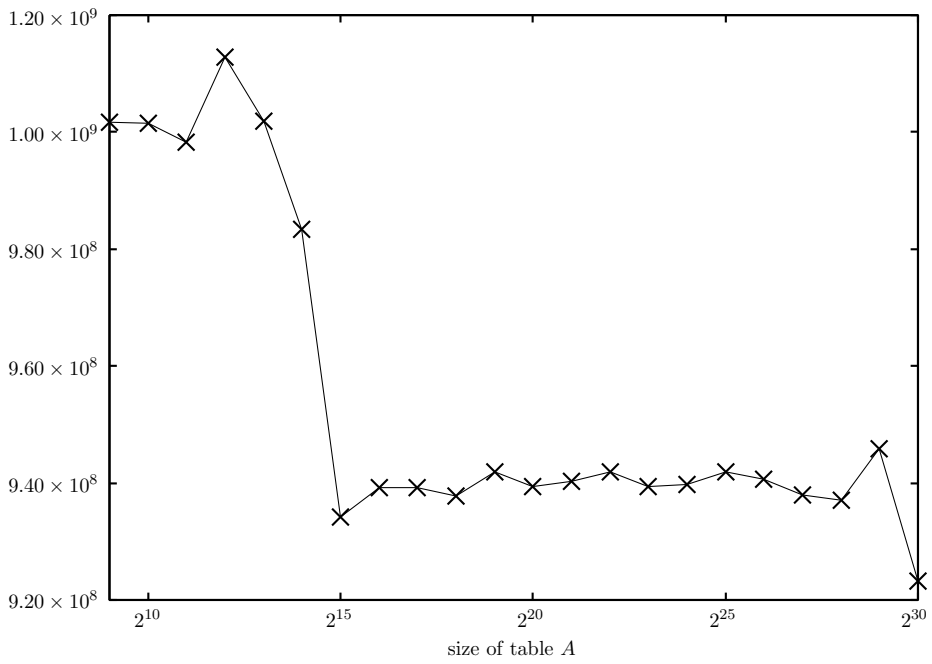

Figure 7: The number of verified 64-bit numbers per second for various size of base bit of table A in the GPU implementation for the delay

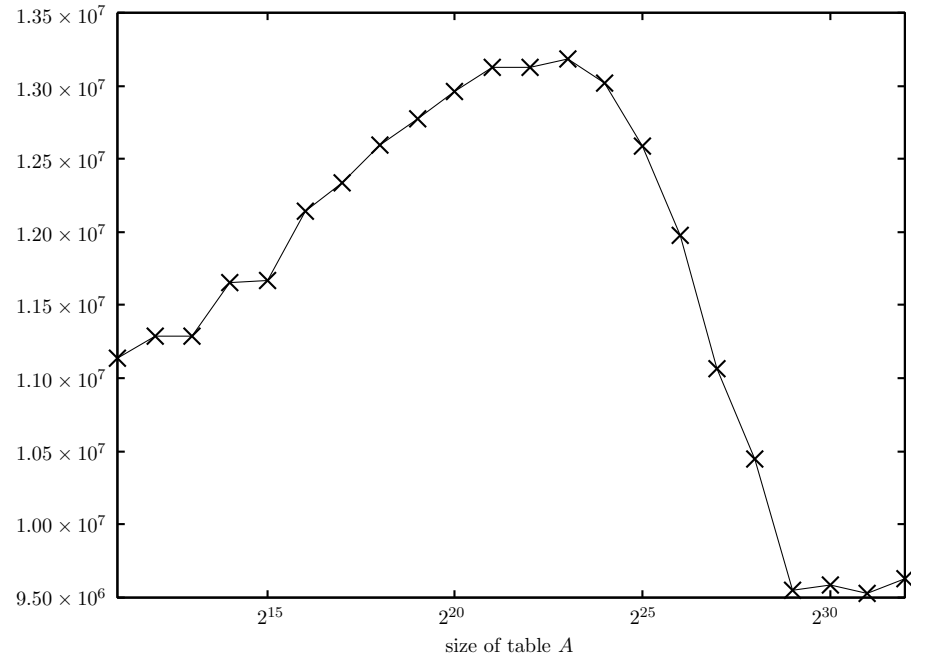

Figure 8: The number of verified 64-bit numbers per second for various size of base bit of table A in the CPU implementation for the delay

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] 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.

[4] Takumi Honda, Yasuaki Ito, and Koji Nakano. GPU-accelerated verification of the Collatz conjecture. In *Proc. of International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP, LNCS 8630)*, pages 483–496, 2014.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] Duhu Man, Kenji Uda, Hironobu Ueyama, Yasuaki Ito, and Koji Nakano. Implementations of a parallel algorithm for computing Euclidean distance map in multicore processors and GPUs. *International Journal of Networking and Computing*, 1(2):260–276, July 2011.

[12] NVIDIA Corp. CUDA ZONE. https://developer.nvidia.com/cuda-zone.

[13] NVIDIA Corp. *Parallel Thread Execution ISA Version 3.2*, 2013.

[14] NVIDIA Corp. *CUDA C Programming Guide Version 7.0*, 2015.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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