GPU-Accelerated Verification of the Collatz Conjecture

Conference Paper · August 2014
DOI: 10.1007/978-3-319-11197-1_37

3 authors, including:

Yasuaki Ito
Hiroshima University
114 PUBLICATIONS **831** CITATIONS
SEE PROFILE

Koji Nakano
Hiroshima University
249 PUBLICATIONS **2,284** CITATIONS
SEE PROFILE

GPU-accelerated Verification of the Collatz Conjecture

Takumi Honda, Yasuaki Ito, and Koji Nakano

Department of Information Engineering, Hiroshima University,
Kagamiyama 1-4-1, Higashi Hiroshima 739-8527, Japan
{honda,yasuaki,nakano}@cs.hiroshima-u.ac.jp

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 and evaluated the performance. The experimental results show that, our GPU implementation can verify 5.01×10^{11} 64-bit numbers per second, while the CPU implementation on Intel Xeon X7460 can verify 1.80×10^9 64-bit numbers per second. Thus, our implementation on the GPU attains a speed-up factor of 278 over the single CPU implementation.

Keywords: Collatz conjecture, GPGPU, Parallel processing, Exhaustive verification

1 Introduction

The Collatz conjecture is a well-known unsolved conjecture in mathematics [8, 19, 22]. 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.

$$3 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

The exhaustive verification of the Collatz conjecture is to perform the repeated operations for numbers from 1 to the infinite as follows:

for $m \leftarrow 1$ 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. A working project for the Collatz conjecture is currently checking 61-bit numbers [17].

There are several researches for accelerating the exhaustive verification of the Collatz conjecture. It is known [1, 3–5] 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 [1, 3, 5], FPGA implementations have been presented 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 [1], 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 [3], 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 [17]. In our previous paper [4], we have shown a software-hardware cooperative approach to verify the Collatz conjectures for 64-bit numbers n. 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 unlimited 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,
- the software implementation on the PC is relatively slow and high power consumption, but the number of bits for the operation is unlimited.

Additionally, in another previous paper of ours [5], we have proposed an efficient implementation of a coprocessor that performs the exhaustive search to verify the Collatz conjecture using embedded DSP slices on a Xilinx FPGA. By effective use of embedded DSP slices instead of multipliers used in [4], the coprocessor can perform the exhaustive verification faster than the above FPGA implementations.

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 [2, 6, 7, 9, 10, 16, 18, 20, 21]. 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. By effective use of a GPU and the above ideas, our new GPU implementation can verify 5.01×10^{11} 64-bit numbers per second. On the other hand, the CPU implementation can verify 1.80×10^9 64-bit numbers per second. As far as we know, the FPGA implementation in [5] has been the fastest implementation. However, our GPU implementation can verify the Collatz conjecture 3.05 times faster the FPGA implementation.

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

2 Accelerating the verification of the Collatz conjecture

The main purpose of this section is to introduce an algorithm for accelerating the verification of the Collatz conjecture. The basic ideas of acceleration are shown in [8, 22].

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, 01, 10, or 11. We can perform the several operations for n based on n_L as follows:

- $n_L = 00$**:** Since two even operations are applied, the resulting number is n_H.
- $n_L = 01$**:** 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 $3n_H + 1$.
- $n_L = 10$**:** 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 $3n_H + 2$.

$n_L = 11$**:** 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 $n_L = 11$ 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	1	0
01	3	1
10	3	2
11	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, that is, $n = 2^d n_H + n_L$. 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 a and b 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.
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 paper [1], we have implemented for various numbers of bits of n_L. Our GPU implementation results show that the performance is well balanced when the number of bits of n_L is 11.

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 $n_L = 00$ then the resulting value is n_H, which is less than n. Thus, we can skip the table operation for n if $n_L = 00$. If $n_L = 01$ 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 $n_L = 10$ then we can skip the table

operation. On the other hand $n_L = 11$ then the resulting value is $9n_H + 8$, which is always larger than n. Therefore, the Collatz conjecture is guaranteed to be true whenever $n_L \neq 11$, because it has been verified true for numbers less than n. Consequently, we need to execute the table operation for number n such that $n_L = 11$.

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.

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 as follows:

```
for m_H ← 1 to +∞ do
  for i ← 0 to s_d − 1 do
    begin
      m_L ← S[i];
      n ← m ← 2^d m_H + m_L;
      while(n ≥ m) do
        begin
          Let n_L be the least significant d bits and
            n_H be the remaining bits.
          n ← B[n_L] · n_H + C[n_L];
        end
    end
```

For the benefit of readers, we show 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 of the mandatory numbers over all numbers. Later, we set base bit 11 for tables B and C, and base bit 32 for table S in our proposed GPU implementation. Thus, one operation $n \leftarrow B[n_L] \cdot n_H + C[n_L]$ corresponds to expected 16.5 odd/even operations. Also, we skip approximately 99.04% of non-mandatory numbers.

Table 1. Tables B, C, and S for least significant 4 bits.

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

Table 3. The size of tables 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

base bit	words	ratio
18	7495	0.0286
19	14990	0.0286
20	27328	0.0261
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

3 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 is implemented as an off-chip DRAM of the GPU, and has large capacity, say, 1.5-6 Gbytes, but its access latency is very long. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-48 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 [11, 13]. 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. 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 processors such that all threads in a block are executed by the same streaming processor

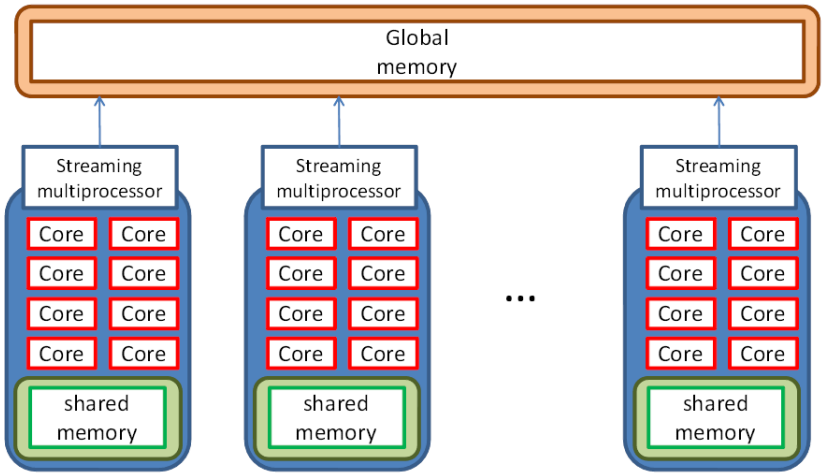


Fig. 1. CUDA hardware architecture

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*. 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 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.

4 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.

The details of our GPU implementation using these ideas are described, as follows.

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 [4]. 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 [17]. 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, therefore, 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. 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 unlimited number of bits by software on the CPU.

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 [5]. Therefore, it is reasonable to perform the verification for overflow numbers on the host. In Section 5, we will evaluate the number of overflows and the verification time for them.

4.2 Efficient memory access for the global memory and the shared memory

In order to reduce the global memory access, all the contents of tables B and C stored in the global memory are cached on the shared memory. Threads in each

block load the contents of the tables B and C to the shared memory at first. In this case, threads read them from the global memory with the coalescing access. After that, each thread verifies assigned numbers. In our implementation, we use tables B and C with base bit 11. Each entry of tables B and C is stored as a 32-bit integer. According to Table 2, the number of words of tables B and C is 2048 and the total size of these tables is 16 Kbytes. Since the maximum size of the shared memory is 48 Kbytes, tables B and C with base bit 12 can be stored on the shared memory. However, since for that case, the size of utilized shared memory is too much, the occupancy is decreased, that is, the performance becomes lower. Thus, we use tables B and C with base bit 11 and cache them on the shared memory.

In addition, we consider efficient memory access of cached tables B and C on the shared memory. Recall that the address of the reference for tables B and C is always determined by the value of n_L which is the least significant bits of interim value n. If interim values have distinct least significant bits, the access for tables B and C becomes distinct. This occurs the bank conflict of the shared memory. To reduce this bank conflict, we utilize the broadcast access, that does not occur the bank conflict. 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 them is shown in Fig. 2. 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. Using this arrangement, threads in a block concurrently verify the conjecture for values that are identical except *thread_ID*. Since the values are not exactly identical, we cannot avoid the bank conflict for all the access. However, 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 the algorithm in Section 2, 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 Fig. 2, 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 $d = 11$, that is the optimal parameter in our experiment, threads can refer the same address at least 4 times for each initial value.

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 [15] that is

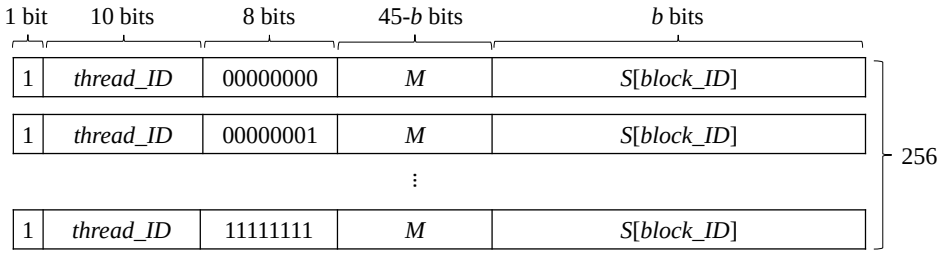


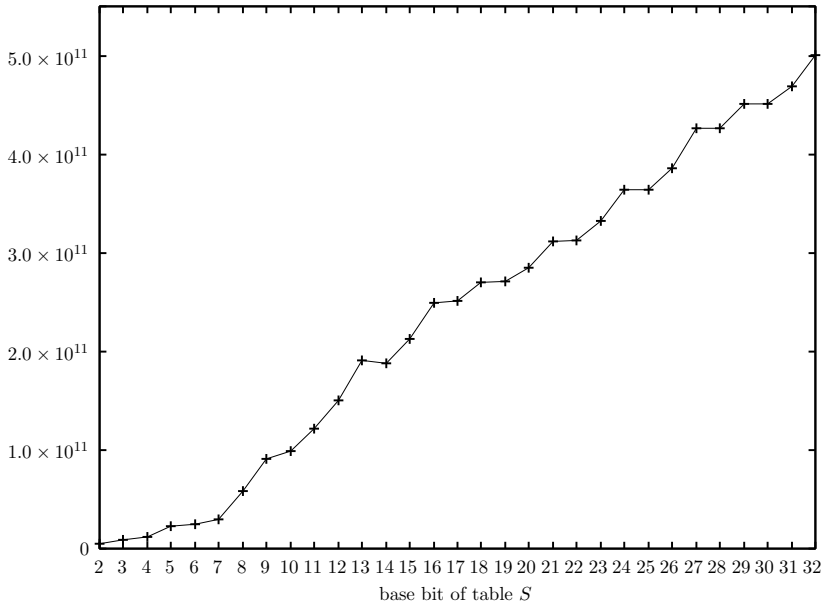
Fig. 2. The data format of 64-bit numbers verified by each thread in a block, where *thread_ID* denotes a thread index within a block, *block_ID* denotes a block index within a kernel, and M is a constant.

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.

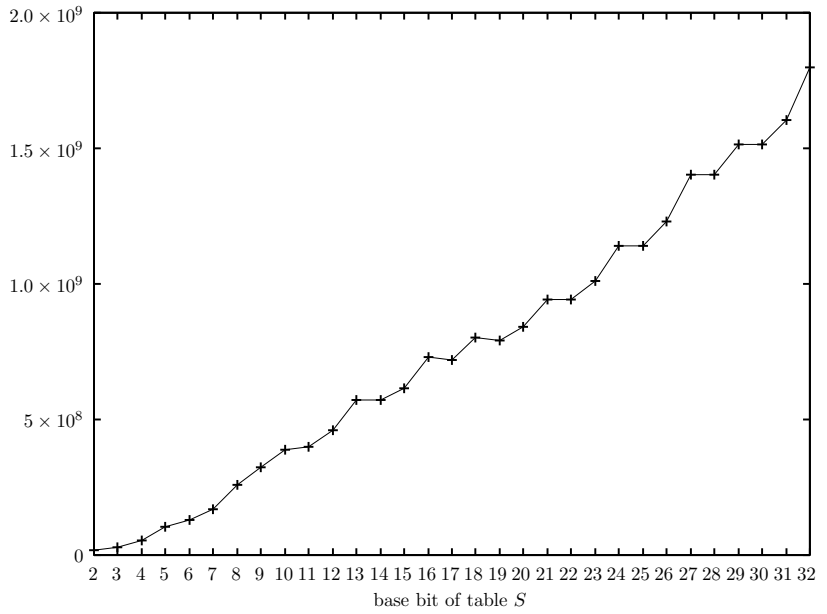
5 Performance Evaluation

We have implemented our GPU implementation of verifying the Collatz conjecture using CUDA C. We have used NVIDIA GeForce GTX TITAN with 2688 processing cores (14 Streaming Multicore processors which have 192 processing cores each) running in 876 MHz and 6 GB memory. For the purpose of estimating the speed up of our GPU implementation, we have also implemented a software approach of verifying the Collatz conjecture using GNU C. In the software implementation, we can apply the idea of accelerating the verification described in Section 2. The difference from the GPU implementation is that the software implementation uses unlimited number operations and verifies the Collatz conjecture serially. Each of them will be compared in the following. We have used in Intel Xeon X7460 running in 2.66GHz and 128GB memory to run the CPU implementation.

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 Fig. 2. For this purpose, we have 10 randomly generated integers as a constant M. For each generated integer M, the GPU implementation verified the Collatz conjecture. In the GPU and CPU implementation, we use tables B and C with base bit 11 and 12, which are optimal bits obtained by our experiments, respectively.



(a) GPU



(b) CPU

Fig. 3. The number of verified 64-bit numbers per second for various size of base bit of table S

Fig. 3 shows the number of verified numbers per second for various base bit of table S in the GPU and CPU implementations. 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 4. For example, when base bit of table S is 32 for a constant M in the GPU verification, 29764 overflow numbers were found, that is, the size of interim values for 29764 numbers became more than 96 bits. After that, the host program verified the conjecture for these numbers using unlimited number of bits by software. The verification time in the CPU was 65 ms including the time of data transfer between the GPU and CPU. Since the total computing time was 2249144 ms, the verification time for overflow numbers by the CPU is much shorter. 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. For table S with base bit 32, our GPU implementation can verify the Collatz conjecture for 5.01×10^{11} numbers per second. On the other hand, in the CPU implementation, for table S with base bit 32, the CPU implementation can verify the Collatz conjecture for 1.80×10^9 numbers per second. Thus, our GPU implementation attains a speed-up factor of 278 over the CPU implementation. As far as we know, the FPGA implementation in [5] has been the fastest implementation. However, our GPU implementation can verify the Collatz conjecture 3.05 times faster the FPGA implementation.

6 Conclusions

We have presented a GPU implementation that performs the exhaustive search to verify the Collatz conjecture. 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 it on NVIDIA GeForce GTX TITAN. The experimental results show that it can verify 5.01×10^{11} 64-bit numbers per second. On the other hand, the CPU implementation verifies 1.80×10^9 64-bit numbers. Thus, our GPU implementation attains a speed-up factor of 278.

References

1. An, F., Nakano, K.: An architecture for verifying Collatz conjecture using an FPGA. In: Proc. of the International Conference on Applications and Principles of Information Science. pp. 375–378 (2009)
2. Diaz, J., Muñoz-Caro, C., Niño, A.: 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)
3. Ichikawa, S., Kobayashi, N.: Preliminary study of custom computing hardware for the 3x+1 problem. In: Proc. of IEEE TENCON 2004. pp. 387–390 (2004)
4. Ito, Y., Nakano, K.: 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. pp. 63–70 (2009)

5. Ito, Y., Nakano, K.: Efficient exhaustive verification of the Collatz conjecture using DSP blocks of Xilinx FPGAs. International Journal of Networking and Computing 1(1), 49–62 (2011)
6. Ito, Y., Nakano, K.: A GPU implementation of dynamic programming for the optimal polygon triangulation. IEICE Transactions on Information and Systems E96-D(12), 2596–2603 (2013)
7. Ito, Y., Ogawa, K., Nakano, K.: Fast ellipse detection algorithm using Hough transform on the GPU. In: Proc. of International Conference on Networking and Computing. pp. 313–319 (Dec 2011)
8. Lagarias, J.C.: The 3x+1 problem and its generalizations. The American Mathematical Monthly 92(1), 3–23 (1985)
9. Man, D., Nakano, K., Ito, Y.: The approximate string matching on the hierarchical memory machine, with performance evaluation. In: Proc. of the IEEE 7th International Symposium on Embedded Multicore SoCs. pp. 79–94 (2013)
10. Man, D., Uda, K., Ito, Y., Nakano, K.: 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. Man, D., Uda, K., Ueyama, H., Ito, Y., Nakano, K.: 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.: CUDA C Best Practice Guide Version 5.5 (2013)
14. NVIDIA Corp.: CUDA C Programming Guide Version 5.5 (2013)
15. NVIDIA Corp.: Parallel Thread Execution ISA Version 3.2 (2013)
16. Ogawa, K., Ito, Y., Nakano, K.: Efficient Canny edge detection using a GPU. In: International Workshop on Advances in Networking and Computing. pp. 279–280 (Nov 2010)
17. Roosendaal, E.: On the 3x + 1 problem. http://www.ericr.nl/wondrous/index.html
18. Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., mei W. Hwu, W.: 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. pp. 73–82 (2008)
19. Silva, T.O.: Maximum excursion and stopping time record-holders for the $3x + 1$ problem: Computational results. Mathematics of Computation 68(225), 371–384 (1999)
20. Takeuchi, Y., Takafuji, D., Ito, Y., Nakano, K.: ASCII art generation using the local exhaustive search on the GPU. In: Proc. of International Symposium on Computing and Networking. pp. 194–200 (2013)
21. Uchida, A., Ito, Y., Nakano, K.: 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)
22. Weisstein, E.W.: Collatz problem. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/CollatzProblem.html