C2CU : A CUDA C Program Generator for Bulk Execution of a Sequential Algorithm

Daisuke Takafuji, Koji Nakano, and Yasuaki Ito

Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan

Abstract. A sequential algorithm is oblivious if an address accessed at each time does not depend on input data. Many important tasks including matrix computation, signal processing, sorting, dynamic programming, and encryption/decryption can be performed by oblivious sequential algorithms. Bulk execution of a sequential algorithm is to execute it for many independent inputs in turn or in parallel. The main contribution of this paper is to develop a tool that generates a CUDA C program for the bulk execution of an oblivious sequential algorithm. More specifically, our tool automatically converts a C language program describing an oblivious sequential algorithm into a CUDA C program that performs the bulk execution of the C language program. Generated C programs can be executed in CUDA-enabled GPUs. We have implemented CUDA C programs for the bulk execution of bitonic sorting algorithm, Floyd-Warshall algorithm, and Montgomery modulo multiplication. Our implementations running on GeForce GTX Titan for the bulk execution can be 199 times faster for bitonic sort, 54 times faster for Floyd-Warshall algorithm, and 78 times faster for Montgomery modulo multiplication, over the implementations on a single Intel Xeon CPU.

Keywords: GPGPU, CUDA, bulk execution, oblivious algorithms, Floyd-Warshall algorithm, Montgomery modulo multiplication

1 Introduction

A Graphics Processing Unit (GPU) is a specialized circuit designed to accelerate computation for building and manipulating images [1–3]. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many application developers [1, 4–7]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [8], the computing engine for NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs. In many cases, GPUs are more efficient than multicore processors [9], since they have hundreds of processor cores and very high memory bandwidth.

CUDA uses two types of memories in the NVIDIA GPUs: *the shared memory* and *the global memory* [8]. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-48 Kbytes. The global memory is implemented as an off-chip DRAM, and thus, it has large capacity, say, 1.5-6 Gbytes, but its access latency is very long. The efficient usage of the shared memory and the global memory is a key for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *the bank conflict* of the shared memory access and *the coalescing* of the global memory access [6, 9, 10]. The address space of the shared memory is mapped into several physical memory banks. If two or more threads access the same memory bank at the same time, the access requests are processed in turn. Hence, to maximize the memory access performance, CUDA threads should access the distinct memory banks to avoid the bank conflicts of the memory accesses. To maximize the bandwidth between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed at the same time. Thus, CUDA threads should perform coalesced access and avoid stride access when they access the global memory. However, it is not an easy task for CUDA developers to design efficient parallel algorithms that does not perform stride memory access.

The bulk execution of a sequential algorithm is to execute it for many independent inputs in turn or in parallel. For example, suppose that we have p arrays $b_0, b_1, \ldots b_{p-1}$ of n points each. We can execute the Fourier transform of each b_j $(0 \leq j \leq p-1)$ by executing the FFT algorithm for n points on a single CPU in turn or on a parallel machine in parallel. The bulk execution of an FFT is frequently used in the area of image processing and signal processing. Further, the bulk execution is widely used in many applications. For example, plain text is partitioned into substrings with the same size when we encrypt it. The substrings are encrypted in turn to obtain encrypted text.

Intuitively, a sequential algorithm is *oblivious* if an address accessed at each time unit is independent of the input. For example, the prefix-sums of an array b of size n can be computed by executing $b[i] \leftarrow b[i]+b[i-1]$ for all i $(1 \leq i \leq n-1)$ in turn. This prefix-sum algorithm is oblivious because the address accessed at each time unit is independent of the values stored in b. The readers may think that the oblivious memory access is too restricted, and most useful algorithms are not oblivious. However, many important and complicated tasks including many matrix computations, signal processing, sorting, dynamic programming, and encryption/decryption can be performed by oblivious sequential algorithms.

In our previous paper [11], we have introduced an algorithmic technique performing the bulk execution of a sequential algorithm on the GPU and evaluated the performance using the Unified Memory Machine (UMM). The UMM is a theoretical parallel computing machine used to evaluate the performance of the computation on the GPU. The resulting implementation on the UMM performs the bulk execution for p independent inputs in $O(\frac{pt}{w} + lt)$ time units using p threads on the UMM if a sequential algorithm is oblivious, where w is the number of threads in a warp, l is the global memory access latency, and t is the running time of a sequential algorithm. It also proved that this implementation

is time optimal. Further, it implemented the prefix-sum algorithm and the dynamic programming algorithm using this algorithmic technique and obtained a speedup factor of 150 over the sequential computation by a single CPU. However, developers need to write CUDA C programs for the bulk execution of a sequential algorithm. Since it needs deep knowledge of CUDA programming and GPU architecture to optimize CUDA C programs, it is not an easy task to write efficient CUDA C programs for the bulk execution.

The main contribution of this paper is to present a tool, *C2CU*, that converts a sequential C program into a CUDA C program with no stride memory access. More specifically, a sequential program written by C programming language is given to C2CU. C2CU converts it into a CUDA C program that performs the bulk execution of a sequential program on CUDA-enabled GPUs. The CUDA C program thus obtained performs no stride global memory access of GPUs. Hence, even developers with few knowledge of CUDA C programming and GPU architecture can automatically generate a CUDA C program for the bulk execution. Once they write a C program for a sequential algorithm, they can obtain a CUDA C program for the bulk execution using our tool C2CU.

To see the performance of CUDA C programs generated by our C2CU converter, we have measured the running time of the bulk execution of three oblivious sequential algorithms: bitonic sort [12, 13], Floyd-Warshall algorithm [14–16], and Montgomery modulo multiplication [17–19]. For this purpose, we first have written sequential algorithms for these three algorithms by C programming language. We then have converted them into CUDA C programs using our C2CU converter. CUDA C programs thus obtained have been executed on GeForce GTX Titan. They run 199 times faster for bitonic sort, 54 times faster for Floyd-Warshall algorithm, and 78 times faster for Montgomery modulo multiplication, over the implementations on a single Intel Xeon CPU.

2 The bulk execution of sequential algorithms on the UMM

The main purpose of this section is to review the bulk execution of sequential algorithms on the Unified Memory Machine(UMM). Please see [11] for the details.

Intuitively, a sequential algorithm is *oblivious* if an address accessed in each time unit is independent of the input. More specifically, there exists a function $a : \{0, 1, \ldots, t-1\} \rightarrow \mathcal{N}$, where t is the running time of the algorithm and $\mathcal{N}$ is a set of all non-negative integers such that, for any input of the algorithm, it accesses address $a(i)$ or does not access the memory at each time i ($0 \leq i \leq t-1$). In other words, at each time i ($0 \leq i \leq t-1$), it never accesses an address other than $a(i)$.

Let us see an example of oblivious algorithms. Suppose that an array b of n integers are given. The prefix-sum computation is a task to store each i-th prefix-sum $b[0]+b[1]+\cdots+b[i]$ in $b[i]$. Let r be a register variable. The following algorithm computes the prefix-sum of n numbers.

[**Algorithm Prefix-sums**]
$r \leftarrow 0$
for $i \leftarrow 0$ to $n-1$ do
  $r \leftarrow r + b[i]$
  $b[i] \leftarrow r$

Since $b[0]$, $b[1]$, ..., $b[n-1]$ are added to r in turn, the prefix-sums are stored in b correctly when this algorithm terminates. Let us see the address accessed in each time unit to confirm that this algorithm is oblivious. For simplicity, we ignore access to registers and local computation such as addition and we assume that such operations can be done in zero time unit. Clearly, memory access operations performed in this algorithm are: read $b[0]$, write $b[0]$, read $b[1]$, write $b[1]$, ..., read $b[n-1]$, and write $b[n-1]$. Hence, the memory access function a is $a(2i) = a(2i+1) = i$ for all i $(0 \leq i \leq n-1)$, and thus, this algorithm is oblivious.

Suppose that we need to execute a sequential algorithm for many independent inputs on a single CPU in turn or on a parallel machine at the same time. We call such computation the *bulk execution*. For example, suppose that we have p arrays $b_0, b_1, \ldots, b_{p-1}$ of size n each on the UMM. The goal of the bulk execution of the prefix-sums is to execute the prefix-sums of every b_j $(0 \leq j \leq p-1)$ on the UMM in parallel. We use p threads and each thread j $(0 \leq j \leq p-1)$ executes the prefix-sums of b_i by Algorithm Prefix-sums. Let r_j $(0 \leq j \leq p-1)$ be a register of thread j. The prefix-sums can be computed in parallel by the following algorithm:

[**Parallel Algorithm Prefix-sums**]
for $j \leftarrow 0$ to $p-1$ do in parallel
  $r_j \leftarrow 0$
  for $i \leftarrow 0$ to $n-1$ do
    $r_j \leftarrow r_j + b_j[i]$
    $b_j[i] \leftarrow r_j$

In our previous paper [11], we have evaluated the running time of the bulk execution of the prefix-sums algorithm for column-wise arrangement on the Unified Memory Machine (UMM) [20, 21]. The UMM captures the essence of the global memory access of CUDA-enabled GPUs. The UMM has three parameters: the number p of threads, width w, and memory access latency l. Each thread is a Random Access Machine (RAM) [22], which can execute fundamental operations in a time unit. Threads are executed in SIMD [23] fashion, and run on the same program and work on the different data. The p threads are partitioned into $\frac{p}{w}$ groups of w threads each called *warp*. The $\frac{p}{w}$ warps are dispatched for the memory access in turn, and w threads in a dispatched warp send the memory access requests to the memory banks (MBs) through the memory management unit (MMU). We do not discuss the architecture of the MMU, but we can think that it is a multistage interconnection network in which the memory access requests are moved to destination memory banks in a pipeline fashion. Note that the UMM with width w has w memory banks and each warp has w threads.

MBs constitute a single address space of the memory. A single address space of the memory is mapped to the MBs in an interleaved way such that the word of data of address i is stored in the $(i \bmod w)$-th bank $B[i \bmod w]$, where w is the number of MBs. In the UMM, a single set of address lines from the MMU is connected to the MBs. Hence, the same address value is broadcast to every MB, and the same address of the MBs can be accessed at each time unit. Also, we assume that MBs are accessed in a pipeline fashion with latency l. In other words, if a thread sends a memory access request, it takes at least l time units to complete it. A thread can send a new memory access request only after the completion of the previous memory access request and thus, it can send at most one memory access request in l time units. Let $A[j] = \{j \cdot w, j \cdot w + 1, \ldots, (j + 1) \cdot w - 1\}$ denote the j-th address group. In the UMM, if multiple memory access requests by a warp are destined for different address groups, they are processed separately. Figure 1 illustrates the memory access by two warps $W(0)$ and $W(1)$. Since memory access requests by $W(0)$ are destined for three address groups, they occupy three pipeline stages. On the other hand, those by $W(1)$ are destined for the same bank, they occupy only one stages. Thus it takes $3(\text{stages}) + 1(\text{stage}) + 5(\text{pipeline stages}) - 1 = 8$ time units to complete memory access requests in Figure 1.

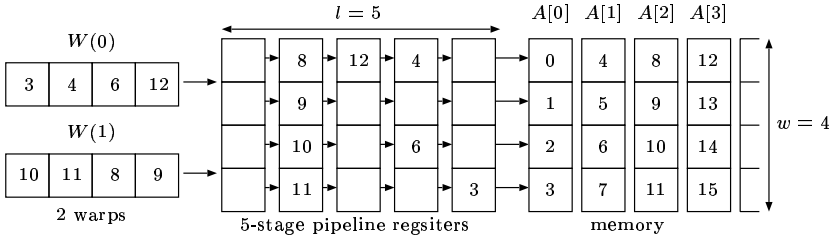


Fig. 1. The memory access of Unified Memory Machine (UMM) with width $w = 4$ and latency $l = 5$

Suppose that each element $b_j[i]$ $(0 \leq i \leq n-1, 0 \leq j \leq p-1)$ is arranged in address $i \cdot p + j$ of the global memory as illustrated in Figure 2. Suppose that the bulk execution of an oblivious algorithm running in t time units is performed for p inputs with column-wise arrangement on the UMM. Clearly, pt memory access operations are performed at all and all memory access operations by all warps are coalesced. Also, each thread on the UMM performs t memory access operations, each of which takes l time units. Thus, we have the following theorem:

Theorem 1 ([11]). *A column-wise oblivious computation of size $n \times p$ runs $O(\frac{pt}{w} + lt)$ time units using p threads on the UMM with width w and latency l, where t is the running time of the corresponding oblivious sequential algorithm.*

Please see [11] for the details of the proof of Theorem 1.

0	1	2	3	4	5	6	7
$b_0[0]$	$b_1[0]$	$b_2[0]$	$b_3[0]$	$b_4[0]$	$b_5[0]$	$b_6[0]$	$b_7[0]$
8	9	10	11	12	13	14	15
$b_0[1]$	$b_1[1]$	$b_2[1]$	$b_3[1]$	$b_4[1]$	$b_5[1]$	$b_6[1]$	$b_7[1]$
16	17	18	19	20	21	22	23
$b_0[2]$	$b_1[2]$	$b_2[2]$	$b_3[2]$	$b_4[2]$	$b_5[2]$	$b_6[2]$	$b_7[2]$
24	25	26	27	28	29	30	31
$b_0[3]$	$b_1[3]$	$b_2[3]$	$b_3[3]$	$b_4[3]$	$b_5[3]$	$b_6[3]$	$b_7[3]$
32	33	34	35	36	37	38	39
$b_0[4]$	$b_1[4]$	$b_2[4]$	$b_3[4]$	$b_4[4]$	$b_5[4]$	$b_6[4]$	$b_7[4]$
40	41	42	43	44	45	46	47
$b_0[5]$	$b_1[5]$	$b_2[5]$	$b_3[5]$	$b_4[5]$	$b_5[5]$	$b_6[5]$	$b_7[5]$

Fig. 2. Column-wise arrangement of $p = 8$ arrays of $n = 6$ elements each

3 Our C2CU converter

The main purpose of this section is to describe C2CU converter, that converts a sequential algorithm written by C programming language into CUDA C program for the bulk execution on CUDA-enabled GPUs.

Figure 3 illustrates the behavior of C2CU converter. A sequential program written by C programming language is converted into a CUDA C program. The converted C program accepts p independent inputs. They are copied to the device memory (global memory) of the GPU. The CUDA device program with p threads is spawned, and each thread executes the sequential program for one input. After all threads terminate, p outputs obtained by all threads are copied to the host memory.

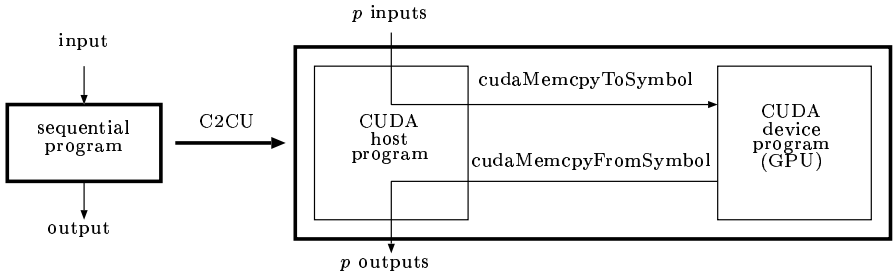


Fig. 3. The behavior of C2CU converter

Let us see how C2CU converter generates CUDA C program using Floyd-Warshall algorithm [14–16] as an example. Floyd-Warshall algorithm is a well

known graph theoretic algorithm that computes the distances of the shortest paths of all pairs of nodes in a directed graph. It uses a 2-dimensional array D of size $n \times n$ for an n-node graph. We assume that, initially, $D[i][j]$ $(0 \leq i, j \leq n-1)$ stores the distance of an edge from node i to j if it exists and $+\infty$ otherwise. Floyd-Warshall algorithm is described as follows:

[**Algorithm Floyd-Warshall**]
for $k \leftarrow 0$ to n do
  for $i \leftarrow 0$ to n do
    for $j \leftarrow 0$ to n do
      if ($D[i][j] > D[i][k] + D[k][j]$)
        $D[i][j] \leftarrow D[i][k] + D[k][j]$

After termination of the algorithm, $D[i][j]$ stores the distance of the shortest path from node i to j. If there is no such path, it stores $+\infty$.

Figure 4 shows a C program for Floyd-Warshall algorithm. It should be clear that this C program computes the all-pairs shortest distance by Floyd-Warshall algorithm. The values of D is updated by calling `update_dist`, although it is not necessary to be a function. The reason is to show our C2CU converter supports function calls. The C program in Figure 4 is a direct implementation of Floyd-Warshall algorithm except that it has a directive `#pragma kernel` in line 22. Most C compilers such as GNU C compiler ignores this directive. Hence, this C program can be compiled correctly, and it computes all-pairs shortest distance in an input graph by Floyd-Warshall algorithm. A directive `#pragma kernel` is used to specify a function for the bulk execution on the GPU. A function call just after directive `#pragma kernel` will be executed on the GPU in the CUDA C program obtained by C2CU.

Figure 5 shows a CUDA C program generated by our tool C2CU from the C program in Figure 4. Users can specify the number p of inputs (i.e. the number p of threads) and the number of threads in each CUDA block, by using options for C2CU. These values are defined as `__P__` $(= p)$ and `__T__` in lines 2 and 3. In Figure 5, they are 2048 and 64, respectively. Thus, 32 CUDA blocks with 64 threads each are spawned by CUDA kernel call `floyd_warshall<<<__B__,__T__>>>()` in line 31. Since the generated CUDA C program accepts p inputs, a 3-dimensional array D of size $N \times N \times p$ allocated in the host memory are used to store them. Also, a 3-dimensional array `__D` of the same size allocated in the device memory (i.e. the global memory of the GPU) are used. In line 30, cudaMemcpyToSymbol is used to copy p inputs stored in D to `__D`. After the bulk execution by CUDA kernel call `floyd_warshall<<<__B__,__T__>>>()` in line 31, cudaMemcpyToSymbol is used to copy `__D`, which stores the resulting values, to D.

CUDA kernel call `floyd_warshall<<<__B__,__T__>>>()` in line 31 invokes `__B__` CUDA blocks with `__T__` threads each. Thus, `__P__` $(= p)$ threads execute Floyd-Warshall algorithm on the CUDA-enabled GPU. Since `blockDim.x` is the number `__B__` of threads in a CUDA block and `blockIdx.x` and `threadIdx.x` take values in $[0, \texttt{\_\_B\_\_} - 1]$ and $[0, \texttt{\_\_T\_\_} - 1]$, respectively, `__id__` in line 15 takes value from 0 to $p-1$. Hence device function `update_dist(i,j,k,__id__)` is executed for `__id__` in $[0, p-1]$ on the GPU in parallel. The reader should

```
 1: #define N 1024
 2: float D[N][N];
 3: void update_dist(int i, int j, int k){
 4:   if( D[i][j] > D[i][k] + D[k][j] ) {
 5:     D[i][j] = D[i][k] + D[k][j];
 6:   }
 7: }
 8:
 9: void floyd_warshall(){
10:   int i,j,k;
11:   for(k=0;k<N;k++) {
12:     for(i=0;i<N;i++) {
13:       for(j=0;j<N;j++) {
14:         update_dist(i,j,k);
15:       }
16:     }
17:   }
18: }
19:
20: int main(int argc, char *argv[]){
21:   input_array();
22: #pragma kernel
23:   floyd_warshall();
24:   ...
```

Fig. 4. A C program of the Floyd-Warshall algorithm

have no difficulty to confirm that CUDA C program in Figure 5 executes Floyd-Warshall algorithm for p inputs in parallel.

Let us see how C2CU converts a C program into a CUDA C program for general cases and confirm that the generated CUDA C programs performs coalesced memory access. If an original C program uses d dimensional array a of size $s_1 \times s_2 \times \cdots \times s_d$, a CUDA C program generated by C2CU uses $d+1$ dimensional array a of size $s_1 \times s_2 \times \cdots \times s_d \times p$. If the original C program accesses $a[i_1][i_2]\cdots[i_d]$ then each thread with ID id of the corresponding CUDA C program accesses $a[i_1][i_2]\cdots[i_d]$[__id__]. Since $a[i_1][i_2]\cdots[i_d][0]$, $a[i_1][i_2]\cdots[i_d][1]$, ..., $a[i_1][i_2]\cdots[i_d][p-1]$ are allocated in consecutive addresses, these memory accesses by p threads are coalesced.

4 Experiment results

The main purpose of this section is to show experimental results on GeForce GTX Titan. GeForce GTX Titan has 14 streaming multiprocessors with 192 cores each. Hence, it can run 2688 threads in parallel. Note that, a single kernel call to GeForce GTX Titan can run more than 2688 threads in a time sharing manner using CUDA [8] parallel programming platform. All input and output

```
 1: #define N 1024
 2: #define __P__ 2048
 3: #define __T__ 64
 4: #define __B__  __P__/__T__
 5: float D[N][N][__P__];
 6: __device__ float __D[N][N][__P__];
 7:
 8: __device__ void update_dist(int i, int j, int k, int __id__){
 9:   if( __D[i][j][__id__] > __D[i][k][__id__] + __D[k][j][__id__] ) {
10:     __D[i][j][__id__] = __D[i][k][__id__] + __D[k][j][__id__];
11:   }
12: }
13:
14: __global__ void floyd_warshall(){
15:   int __id__ = blockIdx.x * blockDim.x + threadIdx.x;
16:   int i,j,k;
17:   for(k=0;k<N;k++) {
18:     for(i=0;i<N;i++) {
19:       for(j=0;j<N;j++) {
20:         update_dist(i,j,k,__id__);
21:       }
22:     }
23:   }
24: }
25:
26: int main(int argc, char *argv[])
27: {
28:   input_array();
29: #pragma kernel
30:   cudaMemcpyToSymbol(__D, D, sizeof(float)*N*N*__P__, 0);
31:   floyd_warshall<<<__B__,__T__>>>();
32:   cudaMemcpyFromSymbol(D, __D, sizeof(float)*N*N*__P__, 0);
33:   ...
```

Fig. 5. A CUDA program for the bulk execution of Floyd-Warshall algorithm generated by C2CU

data are stored in the global memory of the GPU and we do not use the shared memory of the streaming multiprocessors.

We have used three sequential algorithms as follows:

- bitonic sort [12, 13],
- Floyd-Warshall algorithm [14–16], and
- Montgomery modulo multiplication [17–19].

Bitonic sort is a well-known parallel sorting algorithm developed by K.E. Batcher [12]. It can be described as a sorting network with comparators as illustrated in Figure 6. Since elements compare-exchanged in each stage are fixed, bitonic sort can be written as an oblivious sequential algorithm.

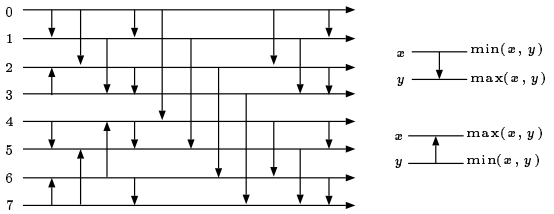


Fig. 6. Bitonic sort for $n = 8$

Montgomery modulo multiplication is used to speed the modulo multiplication $X \cdot Y \cdot 2^{-R} \bmod M$ for R-bit numbers X, Y, and M. The idea of Montgomery modulo multiplication is not to use direct modulo computation, which is very costly in terms of the computing time and hardware resources. By iterative computation of Montgomery modulo multiplication, the modulo exponentiation $P^E \bmod M$ can be computed, which is a key operation for RSA encryption and decryption [24]. Since R is at least 1024 to use Montgomery modulo multiplication for RSA encryption and decryption, addition/multiplication is repeated to perform R-bit addition/multiplication. Figure 7 illustrates how the product $a \cdot b$ of two integers a and b of large bits is computed. Both a and b are partitioned into four integers and the sum of pair-wise products is computed. Using this idea, we can design an oblivious sequential algorithm to compute the product of two integers with large bits in an obvious way. Since Montgomery modulo multiplication repeats computation of the product and the sum of two large integers, it can also be computed by an oblivious sequential algorithm.

We have written a C program for bitonic sort that sorts $n = 32$, 1K (= 1024), and 32K (= 32768) float (32-bit) numbers. We have converted into a CUDA C program for the bulk execution of bitonic sort with parameter $p = 64, 128, \ldots, 4\text{M}$. However, due to the global memory capacity of the GPU, it is executed for up to $p = 128\text{K}$ and $p = 4\text{K}$ when $n = 1\text{K}$ and $n = 32\text{K}$, respectively. The CUDA C program invokes p threads in $\frac{p}{64}$ CUDA blocks with

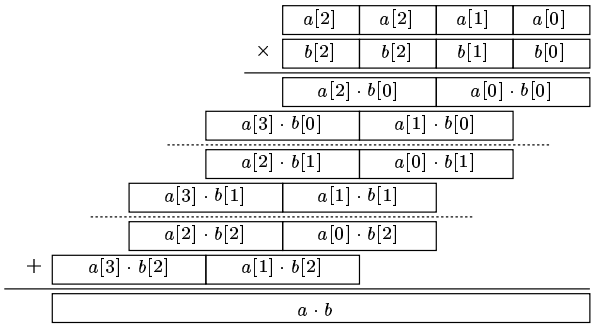
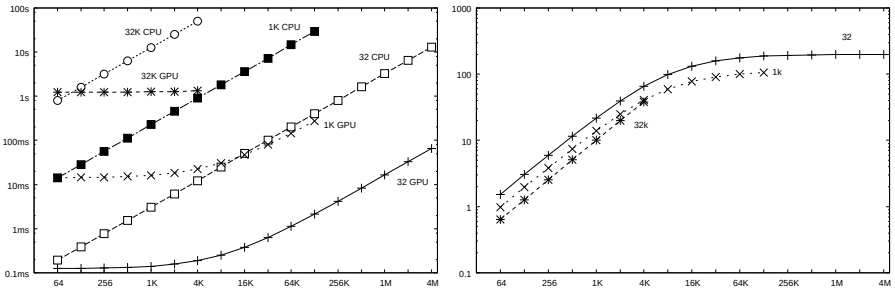


Fig. 7. Multiplication of two integers with large bits

64 threads each to sort p inputs of n numbers each. To see the speedup factor, the original C program is repeatedly executed p times on the Intel Xeon (2.66GHz)

Figure 8 (1) shows the resulting computing time for the bulk execution of bitonic sort. Recall that, from Theorem 1, the bulk execution of a sequential algorithm can be computed in $O(\frac{pt}{w} + lt)$ time units, where p is the total number of threads, l is the memory access latency, and t is the running time of the original sequential algorithm. The bulk execution of bitonic sort for $n = 32$ takes about 0.13ms when $p \leq$1K. Further, the computing time is proportional to p when $p \geq$16K and it runs 65.1ms when $p =$ 4M. Thus, we can think that $O(lt) =$ 0.13ms and $O(\frac{pt}{w}) = (15.5p)$ns. More specifically, the bulk execution of bitonic sort for $n = 32$ and p can be computed in approximately 0.13ms+$(15.5p)$ns. Figure 8 (2) shows the speedup factor of the GPU over the CPU. We can see that the bulk execution of bitonic sort on the GPU can achieve a speedup of factor more than 180 when $n = 32$ and $p \geq$ 128K. Further, when $n = 32$ and $p =$ 4M, the GPU is 199 times faster than the CPU.

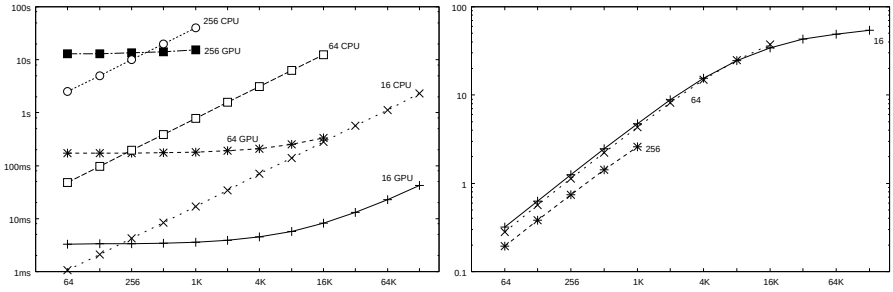


(1) The computing time (2) GPU/CPU speedup factor

Fig. 8. The computing time (ms) of bitonic sort on CPU and GPU, and the speedup for $n = 32$, 1K, 32K, and $p = 64$, 128, ..., 4M.

We have written a C program for Floyd-Warshall algorithm for graphs with $n = 16, 64$, and 256 nodes. We use float (32-bit) numbers to store the length of each edge. The C program is converted into a CUDA C program using C2CU with parameters $p = 16, 64$, and 256. However, due to the global memory capacity of the GPU, it is executed for up to $p = 16$K and $p = 1$K when $n = 64$ and $n = 256$, respectively.

Figure 9 (1) shows the resulting computing time for the bulk execution of Floyd-Warshall algorithm. We will verify $O(\frac{pt}{w}+lt)$ time units shown in Theorem 1. The bulk execution of Floyd-Warshall algorithm for $n = 16$ takes about 3.4ms when $p \leq 512$. Also, the computing time is proportional to p when $p \geq 4$K and it runs 42.6ms when $p = 128$K. Thus, we can think that $O(ln^3) = 3.4$ms and $O(\frac{pn^3}{w}) = (325p)$ns. More specifically, the bulk execution of the Floyd-Warshall algorithm for $n = 32$ and p can be computed in approximately 3.4ms+(325)ns. Figure 9 (2) shows the speedup factor of the GPU over the CPU. We can see that the bulk execution on the GPU can achieve a speedup of factor more than 30 when $n = 16$ and $p \geq 8$K. Further, when $n = 16$ and $p = 128$K, the GPU is 54 times faster than the CPU.



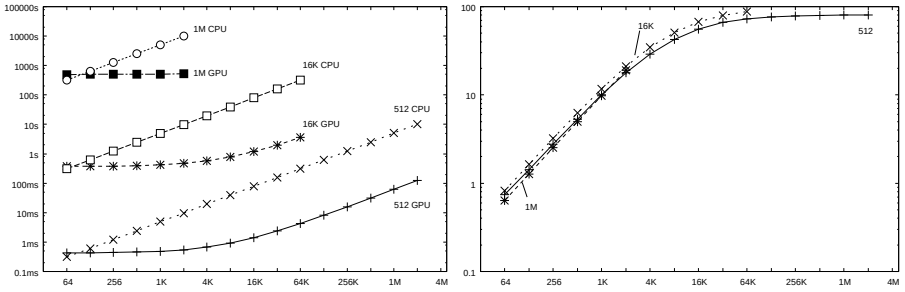
(1) The computing time (2) GPU/CPU speedup factor

Fig. 9. The computing time (ms) of the Floyd-Warshall algorithm on CPU, and GPU and the speedup for $n = 16, 64, 256$, and $p = 64, 128, \ldots, 128$K

Finally, we have written a C program for Montgomery modulo multiplication for $n = 512$, 16K (= 16384), and 1M (= 1048576) bits. We use C2CU to convert it into a CUDA C program with parameter $p = 64, 128, \ldots, 2$M. However, due to the global memory capacity, it is executed for up to $p = 64$K and $p = 2$K when $n = 16$K and $n = 1$M, respectively.

Figure 10 (1) shows the resulting computing time for the bulk execution of the Montgomery modulo multiplication. Again, we will verify $O(\frac{pt}{w} + lt)$ time units shown in Theorem 1. The bulk execution of the algorithm for $n = 512$ takes about 0.45ms when $p \leq 512$. Also, the computing time is proportional to p when $p \geq 128$K and it runs 124ms when $p = 2$M. Thus, we can think that $O(ln^2) = 0.45$ms and $O(\frac{pn^2}{w}) = (59.1p)$ns. More specifically, the bulk execution

of the algorithm for $n = 512$ can be computed in approximately 124ms+$(5.9p)$ns. Figure 9 (2) shows the speedup factor of GPU computation using the GPU over the CPU. We can see that the GPU can achieve a speedup of factor more than 70 when $n = 512$ and $p \geq 32$K. Further, when $n = 512$ and $p = 2$M, the GPU is 78 times faster than the CPU.



(1) The computing time (2) GPU/CPU speedup factor

Fig. 10. The computing time (ms) of the Montgomery modulo multiplication on CPU, and GPU. and the speedup for $p = 64, 128, \ldots, 4$M

5 Conclusion

The main contribution of this paper is to develop C2CU converter, which converts a C language program of a sequential algorithm into a CUDA C program for the bulk execution on the GPU. The experimental results show that the generated CUDA C program on GeForce GTX Titan can achieve up to 199 times speed-up over the original C program running on an Intel Xeon CPU. Thus, C2CU is a promising tool to obtain high GPGPU acceleration very easily.

References

1. Hwu, W.W.: GPU Computing Gems Emerald Edition. Morgan Kaufmann (2011)
2. Man, D., Uda, K., Ito, Y., Nakano, K.: A GPU implementation of computing Euclidean distance map with efficient memory access. In: Proc. of International Conference on Networking and Computing. (Dec. 2011) 68–76
3. Uchida, A., Ito, Y., Nakano, K.: Fast and accurate template matching using pixel rearrangement on the GPU. In: Proc. of International Conference on Networking and Computing, IEEE CS Press (Dec. 2011) 153–159
4. Ogawa, K., Ito, Y., Nakano, K.: Efficient Canny edge detection using a GPU. In: Proc. of International Conference on Networking and Computing, IEEE CS Press (Nov. 2010) 279–280

5. Nishida, K., Ito, Y., Nakano, K.: Accelerating the dynamic programming for the matrix chain product on the GPU. In: Proc. of International Conference on Networking and Computing. (Dec. 2011) 320–326
6. Nishida, K., Ito, Y., Nakano, K.: Accelerating the dynamic programming for the optial poygon triangulation on the GPU. In: Proc. of International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP, LNCS 7439). (Sept. 2012) 1–15
7. Uchida, A., Ito, Y., Nakano, K.: An efficient GPU implementation of ant colony optimization for the traveling salesman problem. In: Proc. of International Conference on Networking and Computing, IEEE CS Press (Dec. 2012) 94–102
8. NVIDIA Corporation: NVIDIA CUDA C programming guide version 5.0 (2012)
9. 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) (July 2011) 260–276
10. NVIDIA Corporation: NVIDIA CUDA C best practice guide version 3.1 (2010)
11. Tani, K., Takafuji, D., Nakano, K., Ito, Y.: Bulk execution of oblivious algorithms on the unified memory machine, with gpu implementation. In: Proc. of International Parallel and Distributed Processing Symposium Workshops. (May 2014) 586–595
12. Batcher, K.E.: Sorting networks and their applications. In: Proc. AFIPS Spring Joint Comput. Conf. Volume 32. (1968) 307–314
13. Akl, S.G.: Parallel Sorting Algorithms. Academic Press (1985)
14. Floyd, R.W.: Algorithm 97: Shortest path. Communications of the ACM **5**(6) (June 1962) 345
15. Warshall, S.: A theorem on boolean matrices. Journal of the ACM **9**(1) (Jan. 1962) 11–12
16. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press (1990)
17. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation **44**(170) (1985) 519–521
18. Shigemoto, K., Kawakami, K., Nakano, K.: Accelerating montgomery modulo multiplication for redundant radix-64k number system on the FPGA using dual-port block RAMs. In: Proc. of International Conference on Embedded and Ubiquitous Computing(EUC). (2008) 44–51
19. Bo, S., Kawakami, K., Nakano, K., Ito, Y.: An RSA encryption hardware algorithm using a single DSP block and a single block RAM on the fpga. International Journal of Networking and Computing **1**(2) (July 2011) 277–289
20. Nakano, K.: Simple memory machine models for GPUs. International Journal of Parallel, Emergent and Distributed Systems **29**(1) (2014) 17–37
21. Nakano, K.: Sequential memory access on the unified memory machine with application to the dynamic programming. In: Proc. of International Symposium on Computing and Networking. (Dec. 2013) 85–94
22. Aho, A.V., Ullman, J.D., Hopcroft, J.E.: Data Structures and Algorithms. Addison Wesley (1983)
23. Flynn, M.J.: Some computer organizations and their effectiveness. IEEE Transactions on Computers **C-21** (1972) 948–960
24. Blum, T., Paar, C.: High-radix montgomery modular exponentiation on reconfigurable hardware. IEEE Trans. on Computers **50**(7) (2001) 759–764