A GPGPU PROGRAMMING FRAMEWORK BASED ON A SHARED-MEMORY MODEL

Kazuhiko Ohno,* Dai Michiura,* Masaki Matsumoto,* Takahiro Sasaki,* and Toshio Kondo*

Abstract

Although general purpose computation on GPU (GPGPU) seems to be a promising method for high-performance computing, current programming frameworks such as CUDA and OpenCL are difficult and not portable enough. Therefore, we propose a new framework *MESI-CUDA* for easier GPGPU programming. MESI-CUDA provides shared variables which can be accessed from both CPU and GPU. Our compiler translates user's shared-memory-based program into a CUDA program automatically generating the memory allocation and data transfer code. The compiler also overlaps kernel executions and data transfers by optimizing the scheduling. The evaluation results show that programs using MESI-CUDA can achieve the performance close to hand-optimized CUDA programs, largely reducing user's coding cost.

Key Words

Parallel programming language, compiler, GPU, CUDA, virtual-shared memory

1. Introduction

The performance of graphic chips, known as graphics processing units (GPU), is improving rapidly outpacing the Moore's law [1]. Therefore, general purpose computation on graphics processing units (GPGPU) [2] is expected to be a practical platform for high-performance computing. Programming frameworks such as CUDA [3] and OpenCL [4] are provided for the purpose.

However, such frameworks are based on the low-level description of the memory allocation and data transfer. Although such description enables detailed optimization of the program, it also makes programming very difficult. The user must explicitly specify data transfers between the host (CPU) memory and the device (GPU) memory. Furthermore, the GPU has complicated memory hierarchy whose specification differs in different chips. It makes the cost of tuning GPU programs very large and also reduces portability.

Therefore, we are developing a new programming framework named *MESI-CUDA* based on more abstract parallel description. Low-level code, such as the memory management, data transfer, and scheduling of parallel execution, is automatically generated. Our compiler translates a shared-memory style parallel program into a CUDA program, generating code for the data transfer between the host and the device memories. Device-dependent memory hierarchy, such as the global/shared memories, is hidden from the user and their usage is optimized by the compiler. The scheduling of kernel executions and data transfers is also determined by the compiler, overlapping the computation and the transfer. This approach enables easier and portable programming, leaving the device-dependent optimization to the compiler.

This paper is organized as follows: Section 2 gives a brief introduction of the GPU and CUDA and points out the current issue. In Section 3, we discuss related works. Section 4 details MESI-CUDA, and in Section 5, we illustrate its implementation. In Section 6, we show the evaluation results, and in Section 7, we discuss our future works. In Section 8, we state the conclusion.

2. Background

2.1 GPU Architecture

GPU is a collection of streaming multiprocessors (SM), each consisting of certain number of processor cores. Compared with general purpose CPU, these GPU cores are simple without complex features such as branch prediction and execution reordering, but their number is very large. For example, NVIDIA GeForce 9800 GTX and Tesla C2050 have 128 and 448 cores, respectively.

The GPU has hierarchical memories. Each core has registers and a local memory, each SM has a small shared memory which is shared among the cores in the SM, and a GPU has one global memory shared by all SMs. Furthermore, the GPU has constant and texture memories for specific purposes. They are shared by all SMs like the global memory but has some restrictions such as they

* Department of Information Engineering, Mie University, Japan; e-mail: {ohno, michiura, masaki, sasaki, kondo}@arch.info.mie-u.ac.jp

Recommended by Dr. Yifeng Zhu
(DOI: 10.2316/Journal.211.2013.1.211-1053)

```
1 #include <stdio.h>
2 #define N 3200
3 #define S (N*sizeof(int))

4 __global__ void add_array(int *x, int *a){
5   int id = blockDim.x*blockIdx.x + threadIdx.x;
6   a[id] = a[id] + x[id];
7 }

8 __global__ void prod_array(int *d, int *e, int *f){
9   int id = blockDim.x*blockIdx.x + threadIdx.x;
10  f[id] = d[id] * e[id];
11 }

12 int main(){
13   int *ha, *hb, *hc, *hd, *he, *hf;
14   int *da, *db, *dc, *dd, *de, *df;
15   cudaMallocHost(&ha, S);
              ⋮
20   cudaMallocHost(&hf, S);
21   cudaMalloc(&da, S);
              ⋮
26   cudaMalloc(&df, S);
27   cudaStream_t sa, sb;
28   cudaStreamCreate(&sa);
29   cudaStreamCreate(&sb);

30   init_array(ha);
              ⋮
34   init_array(he);
```

```
35   cudaMemcpyAsync(da, ha, S, HtoD, sa);
36   cudaMemcpyAsync(db, hb, S, HtoD, sa);
37   cudaMemcpyAsync(dc, hc, S, HtoD, sa);
38   cudaMemcpyAsync(dd, hd, S, HtoD, sb);
39   cudaMemcpyAsync(de, he, S, HtoD, sb);

40   add_array<<<N/128, 128, 0, sa>>>(db, da);
41   cudaMemcpyAsync(ha, da, S, DtoH, sa);
42   cudaStreamSynchronize(sa);
43   output_array(ha);
44   add_array<<<N/128, 128, 0, sa>>>(dc, da);
45   cudaMemcpyAsync(ha, da, S, DtoH, sa);
46   prod_array<<<N/128, 128, 0, sb>>>(dd, de, df);
47   cudaMemcpyAsync(hf, df, S, DtoH, sb);
48   cudaStreamSynchronize(sa);
49   output_array(ha);
50   cudaStreamSynchronize(sb);
51   output_array(hf);

52   cudaFreeHost(ha);
              ⋮
57   cudaFreeHost(hf);
58   cudaFree(da);
              ⋮
63   cudaFree(df);
64   cudaStreamDestroy(sa);
65   cudaStreamDestroy(sb);
66 }
```

*`HtoD`, `DtoH` are actually `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost`, respectively.

Figure 1. Sample program for CUDA.

are small and read-only from the GPU cores. Physically, the shared memories are fast on-chip memories, whereas the local and global memories are off-chip and have large access latency. The constant/texture memories are also part of the off-chip memory but have small caches which enable better access performance.

GPUs are still in rapid evolution and their features, such as the number of cores and sizes of memories/caches, differ by GPU models. For example, NVIDIA's second generation GPU architecture *Fermi* has additional features compared with the first generation architecture *Tesla*, such as extended shared memory size, increased number of cores per SM, L1/L2 caches for the global memory access [5]. Therefore, a well-tuned program for a certain GPU model may not achieve high performance on other models.

2.2 CUDA Programming

Compute unified device architecture (CUDA) [3], [6], [7] is a GPGPU programming framework provided by NVIDIA, using extended C/C++ and the compiler `nvcc`. Figure 1 is an example of a simple CUDA program. The functions `init_array()` and `output_array()` perform read and write to the argument array, respectively[1]. The additional code required for parallel programming in CUDA is shown in bold font.

[1] This code is simplified as an example and actually cannot expect speedup. Kernel functions should be more compute intensive to hide the data transfer overhead.

In CUDA, the CPU and GPU are called *host* and *device*, respectively. Functions, declared with the `__device__` or `__global__` qualifier, are called *kernel functions* and executed on the device (Fig. 1, *l.* 4–11). Other functions (called *host functions* in this paper) are executed on the host. To start computation on the GPU, any host function can call a `__global__` kernel function specifying the numbers of thread blocks and threads per block. The called function is not executed on the host but the kernel threads which execute it are created. The threads are scheduled by the system and run on assigned GPU cores. Each thread block is executed on a SM, thus the threads in the block are executed by the cores in the SM. Using block/thread IDs to specify the data element to process, data parallel processing can be described easily.

Executing a set of kernel threads created in one kernel function *call* is an instance of parallel execution of the kernel function. However, due to loops and function calls, the same *call* code may be executed more than once, each time creating a set of kernel threads. So in this paper, we call the static code of kernel function *call* as *kernel invocation*, and the parallel execution instance as *kernel execution*.

In Fig. 1, two kernel functions are invoked to compute the sum and the scalar product of arrays. Each kernel function computes the array element corresponding to the ID (Fig. 1, *l.* 5–6, 9–10), thus creating `N` kernel threads for the array size `N` (Fig. 1, *l.* 40, 44, 46) makes parallel computation of the whole arrays.

Table 1
CUDA Data Transfer Functions

`cudaMemcpy(`d`, `h`, `b`, cudaMemcpyHostToDevice);`	Download Transfer
`cudaMemcpy(`h`, `d`, `b`, cudaMemcpyDeviceToHost);`	Readback Transfer
`cudaMemcpyAsync(`d`, `h`, `b`, cudaMemcpyHostToDevice, `s`);`	Asynchronous Download Transfer
`cudaMemcpyAsync(`h`, `d`, `b`, cudaMemcpyDeviceToHost, `s`);`	Asynchronous Readback Transfer

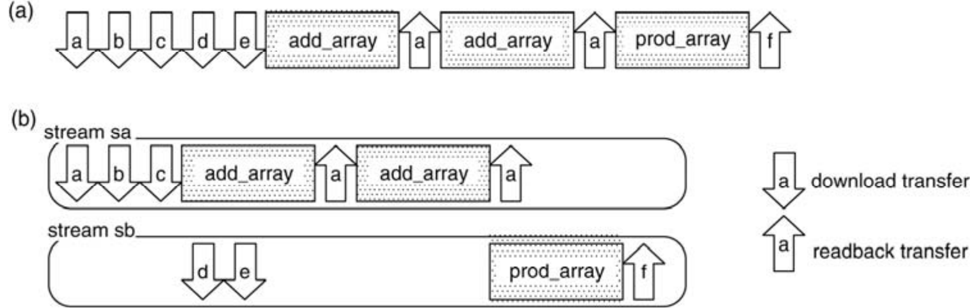


Figure 2. Overlapping execution/transfer using streams: (a) non-overlapping execution and (b) overlapping execution using streams.

The local variables in kernel functions are compiled as registers/local memory accesses and are private for each thread. Variables declared with the `__shared__` qualifier in kernel functions are allocated in the shared memory and shared among threads in the block. While these memories can be used only by the GPU, the GPU global memory, physically on the off-chip memory (called *device memory*), is used to share values between all threads and host/kernel functions. In CUDA programming, explicit data transfers between the host and the device memories are required. Data transfers from host to device and device to host are called *download* and *readback* transfers, respectively. The memory areas on the device used as sources/destinations of transfers are allocated/deallocated using CUDA functions (Fig. 1, *l.* 21–26, 58–63).

CUDA also enables asynchronous data transfer, overlapping kernel executions and data transfers. Kernel executions which have dependencies and the related data transfers can be bundled using a *stream*. On each stream, the executions and transfers are performed sequentially in the assignment order. While a kernel execution assigned with a stream is underway, any data transfer assigned with other streams can be performed in parallel. Table 1 shows CUDA data transfer functions. The parameters d, h are pointers to the device/host memories, respectively, b (bytes) is the transfer size, and s is the stream to assign the transfer. To use asynchronous data transfers, streams must be created and destroyed (Fig. 1, *l.* 27–29, 64–65) and the source/destination memory areas on the host must be allocated/deallocated using CUDA functions to page-lock them (Fig. 1, *l.* 15–20, 52–57). Then, each kernel invocation and asynchronous transfer function call must specify a stream to assign (Fig. 1, *l.* 35–41, 44–47). Furthermore, a stream synchronization is necessary before reading transferred data on the host to assure asynchronous readback transfer is completed (Fig. 1, *l.* 42, 48, 50). Some GPU models, such as Tesla C2050, have dual DMA channels and bi-directional data transfers can also be overlapped.

Figure 2 illustrates how kernel executions and data transfers are overlapped. The code in Fig. 1 assigns `add_array()` and `prod_array()` to different streams. Therefore, the download transfer for the latter can be overlapped with the execution of the former, and the readback transfer for the former can be overlapped with the execution of the latter.

CUDA provides some features to simplify data transfers. CUDA 4.0 supports *Unified Virtual Addressing*, which can map the host/device memories into a single address space. Although the transfer directions can be omitted, explicit data transfer is still necessary. Another feature, called *mapped memory*, enables direct access to the page-locked host memory from kernel functions. It does not require explicit transfers and can be overlapped without using streams. However, multiple accesses to the same data will decline the performance because an implicit transfer occurs for each access.

2.3 CUDA Programming Issue

As described in Section 2.2, one important issue in CUDA programming is that the low-level description is required. Although such description enables the user device-dependent optimization, it also increases the programming difficulty. In addition to the cost of sequential programming, CUDA programming also requires the cost of (1) computation parallelization, (2) data allocation, (3) data transfer and often (4) computation/transfer scheduling.

To parallelize the computation, the target computation must be appropriately divided and assigned to the CPU core(s) and GPU cores. Although the total performance of the GPU is higher than the CPU, it is inefficient on many code patterns such as frequent conditional branches. Thus, SIMD algorithms are required for the efficient execution on many GPU cores. As for the data allocation, data accessed by the CPU/GPU must be allocated in the host/device memories, because the CPU and GPU do not share the memory. Moreover, the decision of each data allocation on GPU also influences the program performance, because GPU has several memory storages whose sizes and access latencies largely differ. To share any data between the CPU and the GPU, explicit data transfers are required. To reduce the transfer overhead, the number/size of transfers should be considered. Furthermore, the computation/transfer scheduling may also dominate the performance. Although CUDA streams can overlap the computation and the transfer, the user must determine their bundling and ordering considering inter-kernel data dependencies. To improve the efficiency, the time of each kernel execution and transfer should also be considered.

CUDA programmer must write code for these features. To obtain satisfying performance, such code should be tuned considering the target platform's specifications. Furthermore, such hand-optimization will be required again to run the program on another GPU of different specifications. Automatic generation of such code will be the answer to this problem, because it reduces the programming cost and porting to another GPU only requires recompiling for the target GPU.

3. Related Work

As a major GPGPU framework, OpenCL [4] is also available. It is similar to CUDA but is based on more generic programming model for heterogeneous multi-core platforms, supporting the Cell processor, DSPs, and FPGAs. However, like CUDA, the low-level optimization by the user is needed for obtaining high performance.

To lower the difficulty and improve the portability of GPGPU, many researches have proposed various schemes providing more abstract programming models to the user hiding the low-level optimization inside the compilers and runtime systems. One of the simple and desirable approaches is the auto-parallelizing compiler, which automatically generate CUDA code from a sequential program. For the programs of regular control/data structures and data access patterns, such as array processing using definite loops, satisfying result is obtained [8]. However, like conventional auto-parallelizing compilers for generic multiprocessors, it is often not practical if the program has irregular factors.

More pragmatic approaches give some kind of parallel description to the system. Some researches propose translators from other major parallel languages [9] or templates [10] to CUDA. Other researches are based on the conventional frameworks like CUDA but the code for some features is automatically generated [11], [12]. Our framework adopts the latter approach, but we currently focus on controlling multiple invocations of kernel functions, whereas most existing works focus on inside of a single invocation.

Thrust [13] is an another approach of the higher level programming interface to CUDA, providing some generic classes as a template library similar to STL. Common computations, such as sort/reduce on vectors, can be easily and efficiently executed on the GPU by calling the library methods. Although the data transfer is hidden under assignments and accesses to the data elements, explicit data allocations in both host/device memories are needed. Another disadvantage is that library approach cannot introduce a global optimization. The library code can be pretuned within each method, but the further optimization is left to `nvcc`.

MESI-CUDA provides a shared-memory model over physically distributed memories of the CPU and GPU, which can be regarded as a kind of virtual-shared memory (VSM) [14], [15]. While the most of the existing VSM systems are hardware/OS-based, our VSM should be implemented in the language level, because GPUs do not have either hardware support for VSM or OS running on the GPU cores. CUBA [16] provides a shared-memory model on CPU and co-processors and is similar to MESI-CUDA. However, it is implemented as a cache of the host memory in the co-processor memory and requires the hardware support. On the other hand, MESI-CUDA implements a shared-memory model in the software level, by automatically generating code required for its behaviour.

Similar to MESI-CUDA, Rthreads [17] and compile-time virtualization (CTV) [18] provide VSM in the language level without any hardware support. Rthreads provides the shared-memory based POSIX thread (Pthreads) model on distributed-memory environments. For each access to a global variable in the target program, code to emulate a shared variable is statically inserted at compile time. Therefore, Pthreads-like programs can run on distributed environments such as workstation clusters. Similarly, ANVIL, an implementation of CTV, converts concurrent C+Pthreads programs into the code for the target architecture at compile time. Although these schemes are more general than MESI-CUDA, the user needs to explicitly specify low-level synchronizations and mutual exclusions. We adopt a model specific to GPGPU programming, thus such specifications are eliminated.

BSGP [19] adopts *fork and join* parallel computation model based on Pthreads-like long-term threads and global barriers for GPU programming. Although the model quite differs from CUDA, it has similarities to our approach, such as simplifying/optimizing multiple kernel invocations and the thread variables translated to be shared among kernel functions. BSGP code is simple because the compiler generates and optimizes kernel functions, and the GPU memory management is hidden. However, the optimization between the host and the device such as the execution/transfer overlapping is not considered. Mapping different computation model to GPU architecture also causes overheads such as emulating global barriers.

```
 1 #include <stdio.h>
 2 #define N 3200
 3 __global__ int a[N], b[N], c[N], d[N], e[N], f[N];

 4 __global__ void add_array(int *x){
 5   int id = blockDim.x*blockIdx.x + threadIdx.x;
 6   a[id] = a[id] + x[id];
 7 }

 8 __global__ void prod_array(){
 9   int id = blockDim.x*blockIdx.x + threadIdx.x;
10   f[id] = d[id] * e[id];
11 }
```

```
12 int main(){
13   init_array(a);
14   init_array(b);
15   init_array(c);
16   init_array(d);
17   init_array(e);

18   add_array<<<N/128, 128>>>(b);
19   output_array(a);
20   add_array<<<N/128, 128>>>(c);
21   prod_array<<<N/128, 128>>>();
22   output_array(a);
23   output_array(f);
24 }
```

Figure 3. Equivalent program using MESI-CUDA.

4. MESI-CUDA

4.1 Outline

While auto-parallelizing compilers hide all parallel processing factors (1)–(4) in Section 2.3, we provide an explicit but more abstracted parallel programming model; computation parallelization (factor (1)) is left to the user, but other factors (2)–(4) are implicit and automatically resolved by the system. This decision is due to the following two reasons: First, explicitly dividing the computation to the CPU/GPU will be practically necessary to achieve high performance, because the advantages of CPU/GPU cores largely differ. The specification of host/kernel functions in CUDA is simple enough for the purpose and not dependent to specific GPU models. However, scheduling kernel executions/data transfers is device dependent and very difficult, thus it should be hidden in the system. Second, the data allocation/transfer are device dependent and largely influence the performance. Hiding these factors, leaving the optimization to the compiler, the programming difficulty is greatly reduced and the program will be much more portable.

Figure 3 is a MESI-CUDA program equivalent to the CUDA program in Fig. 1. The additional code required for parallel programming is shown in bold font. The code is an explicitly parallel program consisting of host and kernel functions. However, most low-level code, such as allocations/deallocations of memory areas/streams, calls of data transfer functions, and specifications of stream assignments, are eliminated.

4.2 Programming Model

In MESI-CUDA, parallelizing computation is same as CUDA; parallel processing is described as the parallel execution of kernel threads executing same kernel function. The CPU code sets up data, creates kernel threads, and refers to the result after the GPU computation. On the other hand, we provide a shared-memory model; global variables declared with the `__global__` qualifier are regarded as global shared variables and can be accessed from both host/kernel functions. Explicit data transfer is not needed. However, a method for synchronization on shared data is necessary to assure semantically correct execution[2]. Specifying synchronization explicitly, like Pthreads, enables the low-level optimization but increases the programming difficulty. Therefore, we adopt an implicit synchronization model.

In our programming model, a kernel invocation k, which starts a kernel execution e of a kernel function f, is regarded as a kind of subroutine call. Shared variables concerning e are regarded as input/output arguments of this call. If a shared variable s is accessed in e, the accesses to s in host functions, preceding k, are completed before k. Similarly, e and all accesses to s in e are completed before any access to s in host functions, succeeding k, occurs. To support this model, the value of s is transferred to the device memory after the latest write on the host before k. When e is completed, the value in the device memory is transferred back to the host memory. Although the execution of host functions continues in parallel with e, it is blocked if it reads s and is suspended until e finishes and the value of s is transferred back.

Adopting this simple model, programming is much easier without explicitly specifying synchronization. Although our model synchronizes shared variables only before/after each kernel execution, it is sufficient for GPU programming. Kernel threads cannot synchronize with the host during their execution, while thread creation cost is very small. Thus, a GPU is expected to be used as a SIMD co-processor, running simple and fine-grained kernel functions. Furthermore, our model enables strong optimization. To improve the computation/transfer overlap, our compiler can determine synchronization points and change the access points to the shared variables as long as the semantics is not changed.

In CUDA programming, a kernel invocation only means that the corresponding kernel execution will start from now on and the actual start/finish time is not assured. Because MESI-CUDA hides all data transfers from the user, our compiler can optimize the scheduling of kernel executions and data transfers by moving kernel invocations and deciding the location of inserting transfer code.

[2] Mutual exclusion is not considered because read/write direction between the host and the device is usually deterministic in GPU programming. Algorithms causing write conflict between the CPU and the GPU will be avoided by the reason of performance.

4.3 Programming Guideline and Restrictions

Although MESI-CUDA is designed to hide some parts of GPU's low-level architecture, the user still needs some knowledge of GPU to achieve high performance.

Coding kernel functions is the same as CUDA, except they can access shared variables. Hand-optimization using GPU specific functions, such as `__sinf()` and `__expf()`, or using the physical-shared memory is still possible. As for coding host functions, CUDA functions for the memory allocation/deallocation, data transfer, and synchronization, are not needed and should not be called because they may interfere with the code generated by our compiler. However, it is possible to store values in the constant/texture memories before the first kernel invocation.

Using shared variables, the implementation is hidden from the user and the optimization by the system can be expected. Multi-dimensional arrays can be declared as shared variables in MESI-CUDA, whereas the memory areas for data transfer must be allocated using `malloc`-like functions in CUDA. Therefore, array-based coding is encouraged because it is simpler than pointer-based coding in CUDA and also make the compiler's optimization easier. However, the user should be aware that static-shared variables occupy the host/device memories throughout the execution and accessing them may have large latency. Thus unnecessary usage should be avoided.

In addition, the current MESI-CUDA implementation has some restrictions as follows:

1. Conditional branches may cause inefficient code. Although the location of transfer code is optimized considering the branches (Section 5.2.2), it cannot handle dynamic behaviours. For example, if a pointer `p` points either of arrays `a` and `b` according to the execution path and is given as a kernel function argument, the transfer code for `a` and `b` are both generated because the compiler cannot statically determine which array is needed.
2. The compiler uses only the host and global memories to implement shared variables. Thus, other GPU memories, such as the shared/constant/texture memories, are not automatically used.
3. Shared variables must be statically declared and the dynamic allocation is not supported.
4. Multi-threading of the CPU code and multiple GPUs are not supported.

Resolving these restrictions is discussed in Section 7.

5. Implementation

MESI-CUDA is implemented as a source-to-source compiler (Fig. 4). A MESI-CUDA program is translated into CUDA code by the compiler, then compiled into the executable code by the CUDA compiler `nvcc`. On the translation, the MESI-CUDA compiler generates additional CUDA code for the memory management, data transfer, and scheduling, based on the result of static analysis. Furthermore, the scheduling/transfer optimization is also performed on the source level. The lower level optimization is left to `nvcc`.

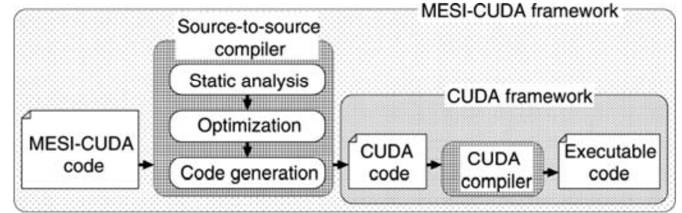


Figure 4. MESI-CUDA compilation flow.

5.1 Program Analysis

Because kernel functions are explicitly defined by the user, the MESI-CUDA compiler only needs to modify the code accessing shared variables and generate data transfer code for them. So we perform static analysis to extract the locations of read/write to the shared variables.

Although non-shared variables can be basically ignored, we need to detect their aliases to shared variables, caused by pointer variables and parameter/argument assignments of function calls. Currently, our scheme regards every shared variable as *atomic*; any access to an array element causes transfer of the whole array. Thus tracing parameters/arguments and pointer assignments is enough. For example, the access to `x` in Fig. 3, *l.* 6 is actually accesses to the shared variables `b` and `c` because `x` is a parameter of `add_array()` and assigned to these shared variables on creating kernel threads (Fig. 3, *l.* 18, 20).

The conditional branches and loops (`if`/`switch` and `for`/`while` statements) may dynamically change the occurrences of kernel executions and accesses to the shared variables. Therefore, the entry/exit points of each branch/loop block are checked.

More than one kernel invocation may correspond to the same kernel function. In such cases, access of shared variables should be analysed not for each function but for each invocation, because each invocation may have different arguments. Similarly, loops and function calls may cause more than one kernel execution corresponding to the same kernel invocation code on the host. However, we do not distinguish such execution instances because our purpose is generating static code for each kernel invocation.

Because kernel functions cannot call host functions in CUDA, we can make static analysis efficiently as follows: First, we make analysis for each `__global__` function f (and its sub-functions) and generate a list of read/written shared variables. We also identify the read/write to each parameters of f. Second, we make analysis of host functions starting from `main`, obtaining every locations of read/write to shared variables and kernel invocations. For each kernel invocation, the shared variables read/written in the corresponding kernel execution are obtained from the result of the kernel function analysis, by matching invocation arguments with kernel function parameters.

5.2 Optimization

5.2.1 Scheduling

To overlap a kernel execution and a data transfer, they must be assigned to different streams. However, the order of executions/transfers is nondeterministic if they are assigned to different streams. Therefore, kernel executions accessing same shared variables and related data transfers must be assigned to the same stream to assure the execution in the programmed order. For this purpose, our scheduler performs clustering of kernel invocations by their accesses to shared variables and assign a stream to each cluster.

Figure 5 shows the stream assignment for Fig. 3 program. The two `add_array()` invocations share the variable `a`. Thus, these invocations and the transfers of `a`, `b`, `c` are assigned to the stream `_s[0]` to satisfy the ordering. On the other hand, the `prod_array()` invocation does not share any variable with these invocations. Therefore, the invocation and transfers of `d`, `e`, `f` are assigned to the stream `_s[1]`.

As for the scheduling within a stream, the invocations in a cluster have partial order relation; some invocations may not have the ordering constraint each other. The compiler can exchange locations of such invocations and corresponding transfers to maximize the overlapping of executions/transfers.

5.2.2 Data Transfer

Our shared variable is implemented as the corresponding memory areas on both host and device, and the code transferring their values mutually. For all related shared variables s_j in each kernel invocation k_i, data transfer code should be inserted if needed. If s_j is read in the corresponding kernel execution of k_i and the latest write preceding to k_i occurs on the host, a download transfer is required. Similarly, if s_j is written in the corresponding kernel execution of k_i and the earliest read succeeding to k_i occurs on the host, a readback transfer is required. In the latter case, a synchronization is also needed before the read on the host to assure the transfer is completed.

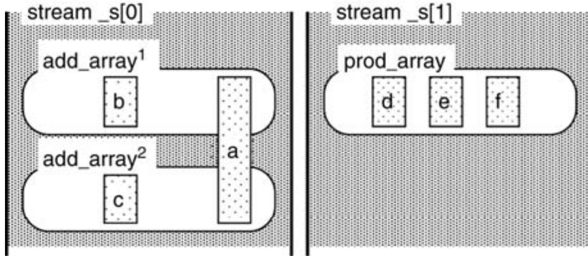


Figure 5. Stream assignment for program in Fig. 3.

To avoid redundant transfers and maximize the execution/transfer overlapping, the locations inserting the transfer code should be optimized. Here, we denote the download/readback transfer and synchronization of s_j for k_i as $dl(k_i, s_j)$, $rb(k_i, s_j)$, and $sync(k_i, s_j)$, respectively. Their locations are noted as $loc(dl(k_i, s_j))$, and so on. Note that $loc()$ may be a set of locations because multiple transfer code can be required due to the branches. The read and write on s_j is denoted as $rd(s_j)$ and $wr(s_j)$, respectively. We also denote the entry/exit points of conditional branches and loops as ben_l/bex_l and len_m/lex_m, respectively.

The download transfer should be issued as earlier as possible to minimize the time k_i is suspended, thus $loc(dl(k_i, s_j))$ is desirable to be right after the latest write to s_j in host functions. Similarly, $loc(rb(k_i, s_j))$ and $loc(sync(k_i, s_j))$ are desirable to be right after k_i and right before the earliest read to s_j in host functions, respectively. However, the locations must be determined considering branches and loops.

When k_i is the reader of s_j, the code is parsed backward from k_i until it finds the first $wr(s_j)$. If any bex_l is found, each branch block is parsed recursively. If ben_l also appears before finding a $wr(s_j)$, such branch is *closed* and can be ignored. If all branches are closed, the latest $wr(s_j)$ is deterministic and $loc(dl(k_i, s_j))$ is right after the $wr(s_j)$ (Fig. 6(a)). If ben_l does not appear before finding a $wr(s_j)$, the latest $wr(s_j)$ is not statically unique. In such cases, the transfer is duplicated and $loc(dl(k_i, s_j))$ is a set of every latest $wr(s_j)$ on each path (Fig. 6(b)). If any branch block does not include a $wr(s_j)$, the transfer is inserted at the beginning of the block, because the preceding $wr(s_j)$ is determined to be the latest at the point (Fig. 6(c)). If ben_l appears without bex_l, it means that a branch occurs after the latest write and the reader (*i.e.*, k_i) is in one of the possible path. In such cases, other branch blocks of the conditional branch l are parsed. If every block has the kernel invocations reading s_j, s_j is deterministically read

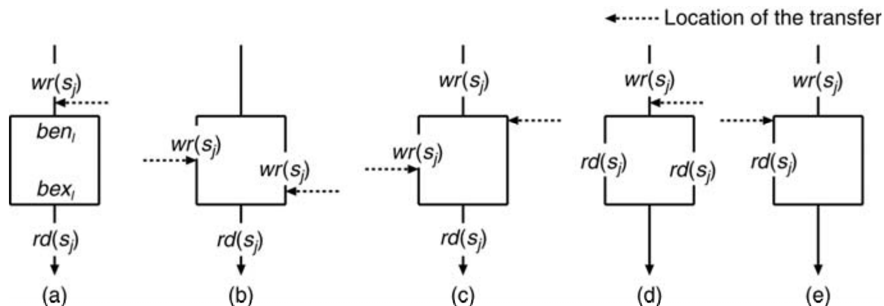


Figure 6. Location of transfer for conditional branches.

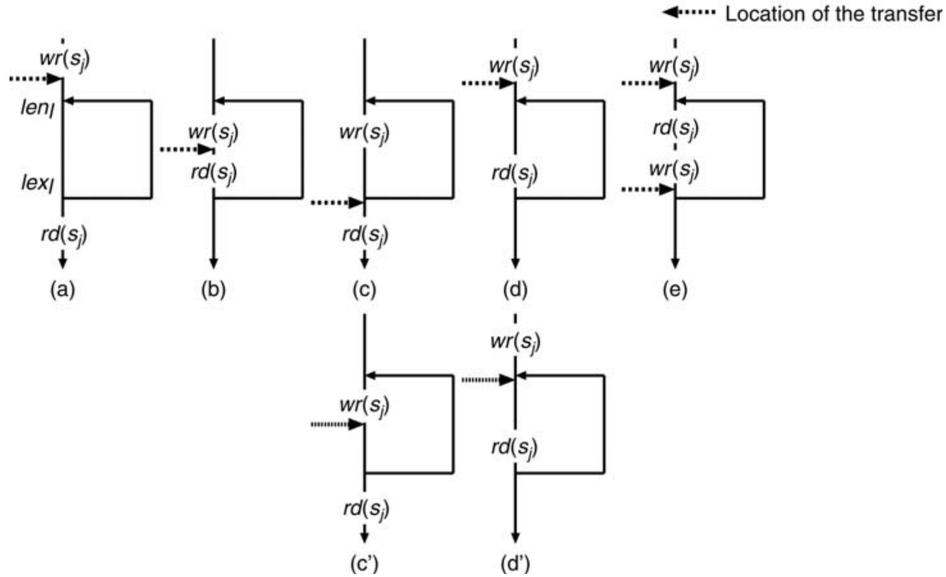


Figure 7. Location of transfer for loops.

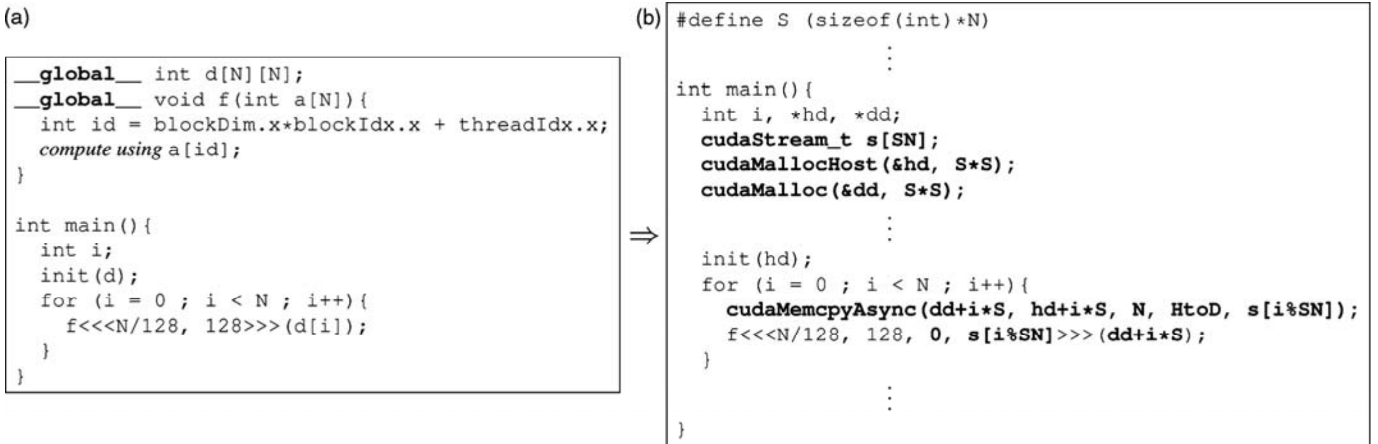


Figure 8. Progressive data transfer optimization: (a) MESI-CUDA code and (b) generated optimized code.

and $loc(dl(k_i, s_j))$ can be determined as described earlier (Fig. 6(d)). If not, $loc(dl(k_i, s_j))$ is a set of the beginning of every block including the kernel invocation reading s_j to avoid unnecessary transfers (Fig. 6(e)).

Loops can be handled similarly. If no len_l or lex_l is found on the backward parsing from k_i to $wr(s_j)$, the loop l is *closed*; k_i and the latest $wr(s_j)$ are both outside the loop (Fig. 7(a)) or both within the loop (Fig. 7(b)). Thus, $loc(dl(k_i, s_j))$ is right after the $wr(s_j)$. If only lex_l appears, the latest $wr(s_j)$ is within the inner loop l, thus $loc(dl(k_i, s_j))$ is right after the loop block l (Fig. 7(c)). If only len_l appears, k_i is within the inner loop l. In this case, $dl(k_i, s_j)$ must be inserted right after the latest $wr(s_j)$ preceding the loop (Fig. 7(d)). However, the existence of $wr(s_j)$, succeeding k_i, must be checked within the loop. If such writes exist, $dl(k_i, s_j)$ must be also inserted right after the latest write before lex_l, because the updated value will be read by k_i on the next iteration (Fig. 7(e)).

We also introduced a progressive data transfer optimization for a simple loop pattern. A computation on n-dimensional arrays is often coded as k-nested loops, each iteration processing m-dimensional sub-arrays, where $m = n - k$ (Fig. 8(a), where $n = 2$, $k = 1$). For such programs, we generate the code which transfers each m-dimensional sub-array in the loop using a constant number of streams in rotation (Fig. 8(b)). Such code can overlap the transfer for the $(i + 1)$th kernel invocation with the ith kernel execution (Fig. 7(d')) and often have better performance than transferring the whole n-dimensional array before the loop (Fig. 7(d)).

As for the readback transfer, $loc(rb(k_i, s_j))$ can be determined similarly with $loc(dl(k_i, s_j))$, whereas $loc(sync(k_i, s_j))$ can be determined right before every possible earliest $rd(s_j)$. The progressive data transfer optimization can also be applied (Fig. 7(c)→(c')).

5.3 Code Generation

5.3.1 Memory Allocation/Deallocation

For each shared variable s_i, the declaration is replaced with the declaration of two pointers: `_host_`s_i and `_dev_`s_i,

```
                   ⋮
 4 __global__ void add_array(int *x, int *_dev_a){
 5   int id = blockDim.x*blockIdx.x + threadIdx.x;
 6   _dev_a[id] = _dev_a[id] + x[id];
 7 }

 8 __global__ void prod_array(int *_dev_d, int *_dev_e, int *_dev_f){
 9   int id = blockDim.x*blockIdx.x + threadIdx.x;
10   _dev_f[id] = _dev_d[id] * _dev_e[id];
11 }

12 int main(){
( 1) int *_host_a, *_dev_a;
                   ⋮
( 6) int *_host_f, *_dev_f;
( 7) cudaMallocHost(&_host_a, N*sizeof(int));
                   ⋮
(12) cudaMallocHost(&_host_f, N*sizeof(int));
(13) cudaMalloc(&_dev_a, N*sizeof(int));
                   ⋮
(18) cudaMalloc(&_dev_f, N*sizeof(int));

13   init_array(_host_a);
                   ⋮
17   init_array(_host_e);

18   add_array<<<N/128, 128>>>(_dev_b, _dev_a);
19   output_array(_host_a);
20   add_array<<<N/128, 128>>>(_dev_c, _dev_a);
21   prod_array<<<N/128, 128>>>(_dev_d, _dev_e, _dev_f);
22   output_array(_host_a);
23   output_array(_host_f);

( 1) cudaFreeHost(_host_a);
                   ⋮
( 6) cudaFreeHost(_host_f);
( 7) cudaFree(_dev_a);
                   ⋮
(12) cudaFree(_dev_f);
24 }
```

Figure 9. Memory allocation/deallocation code.

which point memory areas corresponding to s_i in the host and device memories, respectively. Memory allocation/deallocation code calling CUDA functions is also generated for the pointers.

Figure 9 is the result of generating the memory allocation/deallocation code for Fig. 3 program. The generated code is shown in bold font. For each declaration of the shared variables `a`–`f` (Fig. 3, *l.* 3), pointer variables `_host_a`–`_host_f` and `_dev_a`–`_dev_f` are declared (Fig. 9, *l.* 12(1)–12(6)). Memory allocations/deallocations using CUDA functions are also inserted (Fig. 9, *l.* 12(7)–12(18), 23(1)–23(12)).

All accesses to the shared variables are replaced with the accesses to the corresponding areas in the host/device memories (Fig. 9, *l.* 6, 10, 13–17, 19, 22–23). For each shared variable accessed in kernel executions, the kernel function needs the pointer to the corresponding area on the device memory. The kernel function parameters/invocation arguments are modified to pass such pointers (Fig. 9, *l.* 4, 8, 18, 20–21).

5.3.2 Data Transfer

For each $loc(dl(k_i, s_j))$, $loc(rb(k_i, s_j))$, and $loc(sync(k_i, s_j))$, determined in the transfer optimization phase (Section 5.2.2), the code calling CUDA transfer/synchronization functions is inserted. To overlap kernel executions and data transfers, CUDA asynchronous functions are used. Therefore, the code to create and destroy streams is also inserted. The number of streams is determined in the scheduling phase described in Section 5.2.1. According to the scheduling result, the assigned stream is specified for each transfer/synchronization function and also each kernel invocation.

Figure 10 is the result of generating data transfer code for Fig. 3 program. The generated code is shown in bold font. First, the creation/destruction code for two streams, assigned in the scheduling phase, is inserted (Fig. 10, *l.* 12(19)–12(22), 23(13)–23(14)). Then, the transfer and synchronization code is inserted for each related shared variable of every kernel invocations. For example, `b` is

```
                         ⋮
12 int main(){
                         ⋮
(19) cudaStream_t _s[2];
(20) int _i;
(21) for (_i = 0 ; _i < 2 ; _i++)
(22)   cudaStreamCreate(&_s[_i]);
13   init_array(_host_a);
( 1) cudaMemcpyAsync(_dev_a, _host_a, N*sizeof(int), HtoD, _s[0]);
14   init_array(_host_b);
( 1) cudaMemcpyAsync(_dev_b, _host_b, N*sizeof(int), HtoD, _s[0]);
15   init_array(_host_c);
( 1) cudaMemcpyAsync(_dev_c, _host_c, N*sizeof(int), HtoD, _s[0]);
16   init_array(_host_d);
( 1) cudaMemcpyAsync(_dev_d, _host_d, N*sizeof(int), HtoD, _s[1]);
17   init_array(_host_e);
( 1) cudaMemcpyAsync(_dev_e, _host_e, N*sizeof(int), HtoD, _s[1]);

18   add_array<<<N/128, 128, 0, _s[0]>>>(_dev_b, _dev_a);
( 1) cudaMemcpyAsync(_host_a, _dev_a, N*sizeof(int), DtoH, _s[0]);
( 2) cudaStreamSynchronize(_s[0]);
19   output_array(_host_a);
20   add_array<<<N/128, 128, 0, _s[0]>>>(_dev_c, _dev_a);
( 1)  cudaMemcpyAsync(_host_a, _dev_a, N*sizeof(int), DtoH, _s[0]);
21   prod_array<<<N/128, 128, 0, _s[1]>>>(_dev_d, _dev_e, _dev_f);
( 1) cudaMemcpyAsync(_host_f, _dev_f, N*sizeof(int), DtoH,_s[1]);
( 2) cudaStreamSynchronize(_s[0]);
22   output_array(_host_a);
( 1) cudaStreamSynchronize(_s[1]);
23   output_array(_host_f);

                         ⋮
(13) for (_i = 0 ; _i < 2 ; _i++)
(14)   cudaStreamDestroy(_s[_i]);
24 }
```

Figure 10. Data transfer code.

written in `init_array()` (Fig. 10, *l.* 14) and read in the first `add_array()` invocation (Fig. 10, *l.* 18). Thus, the download transfer from `_host_b` to `_dev_b` is inserted right after the `init_array()` call (Fig. 10, *l.* 14(1)), but the readback transfer of `b` is not inserted because it is not read on the host after the kernel execution. Similarly, the download transfer of `a` is inserted right after it is written on the host (Fig. 10, *l.* 13(1)). Two readback transfers are also required for `a` because it is read on the host twice (Fig. 10, *l.* 19, 22). Therefore, the transfer code is inserted right after `add_array()` invocations (Fig. 10, *l.* 18(1), 20(1)), and the synchronization of the assigned stream is inserted right before the `output_array()` calls (Fig. 10, *l.* 18(2), 21(2)). A download transfer before the second `add_array()` invocation is not needed because `a` is not changed on the host after the first `add_array()` invocation. Specifications of assigned streams are also inserted as the arguments of kernel invocations and transfer functions; `add_array()` and related transfers use `_s[0]` (Fig. 10, *l.* 13(1), 14(1), 15(1), 18, 18(1–2), 20, 20(1), 21(2)), while `prod_array()` and related transfers use `_s[1]` (Fig. 10 *l.* 16(1), 17(1), 21, 21(1), 22(1)).

6. Evaluation

We evaluated MESI-CUDA using four programs in Table 2: two simple benchmarks (`bsearch`, `matmul`) and two application cores (`raytrace`, `poisson`). We executed sequential/hand-optimized CUDA/MESI-CUDA versions of each program on a PC (core i7 930 2.8 GHz, 6 GB Memory) with Tesla C2050/C1060 (Fermi/Tesla architectures, respectively). Each GPU program uses only the global memory. However, we also tried the following faster versions of `matmul`, `bsearch`, and `raytrace`, which will be noted with a postfix `-m`.

`matmul-m` Computes a sub-matrix of size 16^2 using threads in the same block, copying required values into the shared memory to reduce the global memory accesses [6], [7].

`bsearch-m` Copies data of index $i \times N/2048$ ($i = 0, 1, \ldots, 2048$) into the shared memory, thus the first 11 ($=\log_2 2048$) steps of every search can avoid the global memory access.

`raytrace-m` Stores constant data, such as the properties of primitive objects, in the constant memory.

6.1 Code Size

To estimate the coding cost using each programming framework, we compared the lines and file sizes (bytes) of each source code, excluding indents, blank lines and comments. The results are shown in Figs. 11 and 12.

The values in Figs. 11 and 12 are not a strict estimation of the coding cost, because they may vary by the coding

Table 2
Evaluation Programs

`matmul`	Computes matrix multiplication of size $N \times N$. ($N = 1024$)
`bsearch`	finds locations of M random keys in N sorted values using binary search. ($N = 256\,\mathrm{M}$, $M = 128\,\mathrm{K}$)
`raytrace`	Renders an image of size $X \times Y$ using ray tracing algorithm. ($X = 3200$, $Y = 2400$)
`poisson`	Poisson equation solver kernel using a point-Jacobi method [20]. The size of the grid is 128^3.

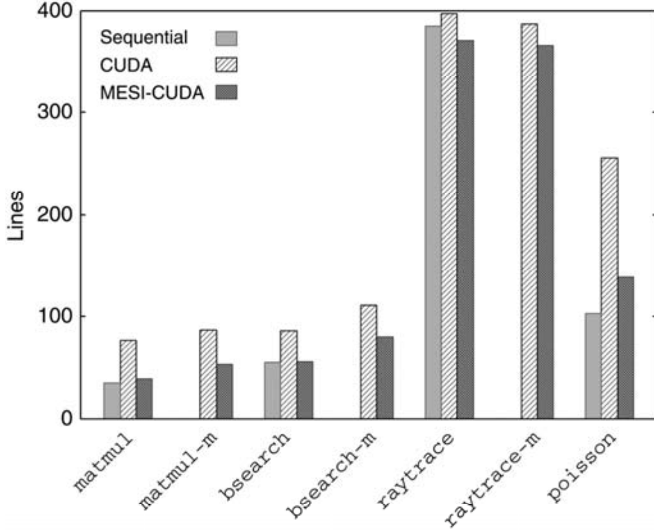


Figure 11. Source code lines.

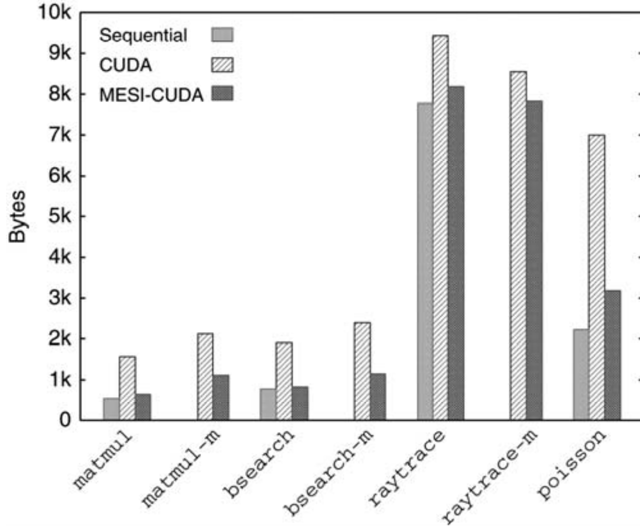


Figure 12. Source code file size.

style and each code line/byte does not require the same coding effort. However, the results show that CUDA programming requires much more additional code for parallelizing sequential computation into kernel functions and invoking them, allocating/deallocating memory areas/streams, and inserting data transfers/synchronizations. Using MESI-CUDA, such additional code is largely reduced because only computation parallelization is needed[3]. For `poisson`, which has 3 kernel functions and 14 multi-dimensional arrays accessed on the GPU, the code size of the MESI-CUDA version is approximately 50% of the CUDA version. And even for `raytrace`, which has only 1 kernel function accessing a single two-dimensional array, the code size is reduced to approximately 90%.

[3] For `raytrace`, the sequential version requires more code lines than the MESI-CUDA version because it includes the code implementing some CUDA-provided data types such as `float3`.

The kernel functions in `matmul-m` and `bsearch-m` have the additional code for the optimization: copying data between the global memory and the shared memory, and accessing the latter. The same optimization is possible in MESI-CUDA by copying data between our shared variables and the shared memory. For `raytrace-m`, `const` variables accessed only on the GPU are allocated on the constant memory using the `__constant__` qualifier. This description simplifies CUDA code because dynamic memory allocations/data transfers are not needed. The same description is also possible in MESI-CUDA because it does not interfere our implicit memory allocation and data transfer.

6.2 Execution Speedup

Figure 13 shows the speedup of GPU versions, which is the inverse of execution time ratio to the sequential version. Note that the vertical axis is log-scaled, because the speedup largely varies by the programs.

The results show that in most cases, MESI-CUDA can automatically generate optimized data transfer code whose performance is close to the hand-optimized CUDA code. For `matmul` and `raytrace`, MESI-CUDA achieved the same speedup with CUDA versions. These programs compute each row of a two-dimensional array on each kernel execution, thus MESI-CUDA can generated equivalent code with CUDA using the progressive transfer optimization described in Section 5.2.2. For `poisson`, 2 of 3 kernel functions are independent each other and a corresponding data transfer can be overlapped with the execution. Although our compiler currently cannot detect such parallelism and generates non-overlapped code, the slowdown is smaller than 0.5% because the overlapped transfer is not a dominant factor.

For `bsearch` and `bsearch-m` on C2050, MESI-CUDA caused approximately 20% and 45% slowdown, respectively. The CUDA version overlaps the computation and the transfer by equally dividing one-dimensional arrays of M keys/locations into multiple regions. The MESI-CUDA compiler regards the access to each region as the access to the same array and cannot generate overlapping code.

Because the accesses to the global memory is the dominant factor, the performance is largely improved in `matmul-m` and `bsearch-m` by copying multiple-accessed

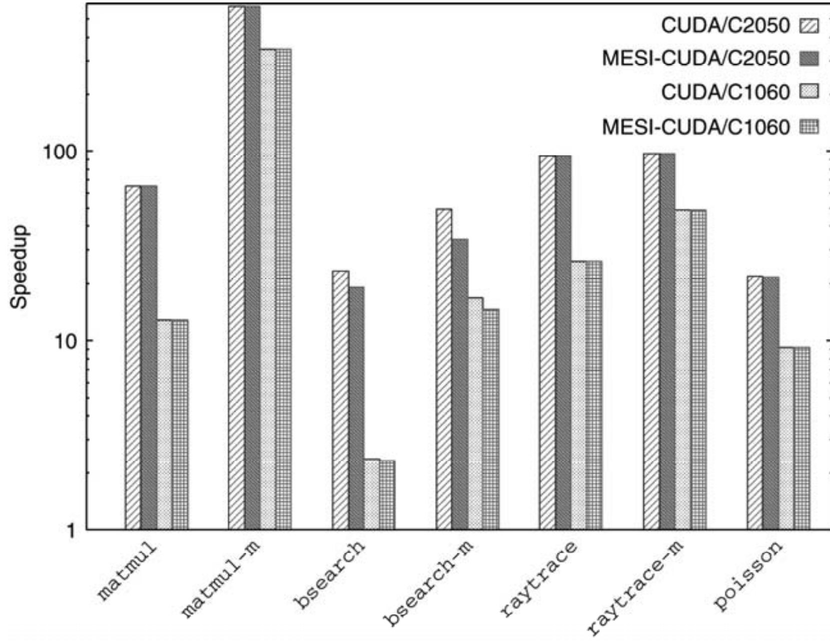


Figure 13. Execution speedup.

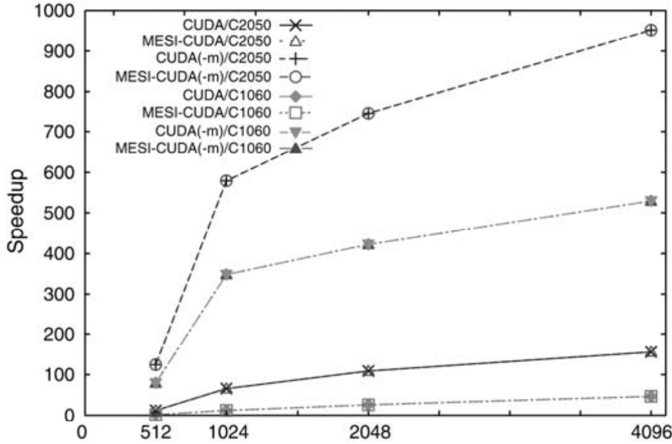


Figure 14. Speedup for different size of `matmul`.

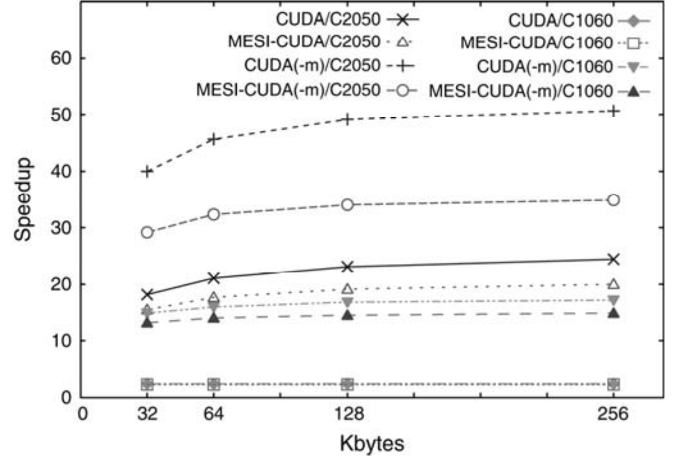


Figure 15. Speedup for different size of `bsearch`.

data into the shared memory. The improvement compared with `matmul`/`bsearch` is larger on C1060, because it does not have the L1/L2 caches and every global memory access causes large latency. For `raytrace-m`, 88% speedup was achieved on C1060, but only 3% on C2050. This is because C2050 has both constant and global memory caches, but C1060 has only the former.

6.3 Scalability

We also executed `matmul` and `bsearch` on different data size. The results are shown in Figs. 14 and 15.

As explained in Section 6.2, the MESI-CUDA compiler generated equivalent code with the CUDA version for `matmul`, thus using MESI-CUDA has no disadvantage on any data size. The results show that utilizing the shared memory has a large benefit achieving notable speedup regardless of data size or GPU architecture. As for `bsearch`/`bsearch-m`, the slowdown ratio of the MESI-CUDA versions to the CUDA versions are approximately constant regardless of data size. Therefore, using MESI-CUDA has the same scalability as using CUDA, although the improvement of code generation is desirable to prevent the slowdown.

7. Future Works

The current MESI-CUDA implementation makes only simple static analysis and the optimization is limited. Therefore, introducing stronger static analysis and enforcing the optimization is our future work. For example, estimating computation/transfer cost will largely improve the scheduling. Lifetime analysis on shared variables will enable recycling the memory areas. Introducing range and pattern analysis on the array accesses will narrow down the

necessary elements and improve data transfer efficiency. A range analysis of array indices can generalize the progressive transfer optimization (Section 5.2.2) to cover the cases like `bsearch` in Section 6.

As for the restrictions shown in Section 4.3, we have future plans as follows:

1. Although MESI-CUDA needs to generate all additional code statically, we can handle the dynamic behaviours of user's code inserting conditional branches in the additional code. For the example in Section 4.3, only the required one of *a* and *b* can be transferred by comparing the pointer *p* with addresses of *a* and *b*. The overhead of executing such test code for each transfer is ignorable compared with the transfer overhead.
2. CUDA-like hand optimization using the shared/ constant/texture memories is possible in MESI-CUDA. However, our goal is hiding the whole GPU memory architecture. Many schemes utilizing such memories have been proposed [8], [9], [11], [12] and possibly adopted to MESI-CUDA. Narrowing down the vital data using static analysis will also help utilizing such small memories.
3. The shared variables are statically declared, but they are compiled into the code dynamically allocating memory areas. Thus their sizes do not need to be constant at compile time. We are planning to provide `cudaMallocGlobal()` for dynamic allocation of a memory area working as a shared variable, and also variable-length arrays like C99 to encourage array-based coding.
4. Static analysis on multi-threaded program is possible by handling concurrent threads like conditional branch blocks. However, mutual exclusions will be required to make the accesses to shared variables MT-safe, which may cause considerable overhead. Nondeterministic scheduling of CPU threads will also disturb our scheduling. Minimizing mutual exclusions using static analysis and generating code which controls CPU thread scheduling may resolve such issues.

   On the other hand, supporting multiple GPUs on a host is essentially not difficult. The scheduler can be extended to overlap kernel executions on the different devices. For the direct dependencies on shared variables between kernel invocations on the different devices, device-to-device transfer code should be generated using `cudaMemcpyPeer()` [6].

Because the current MESI-CUDA implementation assumes Tesla/Fermi architectures, supporting the new features of NVIDIA's third generation GPU architecture *Kepler* [21] is also our future work.

Kepler supports kernel invocations on the device to enable dynamic creation of parallel threads, and the hardware scheduling on the device is enforced. While it may reduce the impact of optimizing host-device transfers and their scheduling, hiding and auto-optimizing the memory management will be more important because dynamic scheduling of kernel threads will make the hand-optimization of memory usage more difficult. Expecting kernel invocations in kernel functions and direct dependencies between them, introducing optimization inside kernel functions will be required for the high performance.

Kepler also enforces handling multiple CPUs and GPUs. *Hyper-Q* allows parallel usage from multiple CPU cores to the same GPU. *GPUDirect* supports direct data transfers between GPUs on the different host. Using the latter, GPU clusters will be easily supported in MESI-CUDA. However, supporting multiple CPU will cause the issue similar to supporting multi-threads on a CPU.

To design the detail of these improvements and also verify their effect, we are planning more evaluations of MESI-CUDA using various GPU applications.

8. Conclusion

The hardware potential of GPU is very attractive as a high-performance computing platform. However, current programming frameworks are still difficult and hand-optimization tends to spoil the portability. Hence, we proposed a new framework MESI-CUDA based on a shared-memory model for easier GPGPU programming.

MESI-CUDA provides shared variables accessible from both CPU and GPU with implicit synchronization. The compiler translates user's program into a CUDA program, generating the memory allocation and data transfer code. Its scheduler also maximizes overlapping of kernel executions and data transfers. Thus, the low-level and device-dependent factors are hidden from the user and optimized automatically.

As the results of evaluation, the programs using MESI-CUDA achieved the performance close to the hand-optimized CUDA programs, and the size of source code was much smaller. Although our approach seems promising, we need to introduce many improvements discussed in Section 7 and evaluate their impact for further high performance.

Acknowledgement

This work was supported by JSPS KAKENHI Grant Number 24500060 entitled "GPGPU Programming Scheme based on a Shared-Memory Model and Scheduling Optimization".

References

[1] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, and T.J. Purcell, A survey of general-purpose computation on graphics hardware, *Computer Graphics Forum*, *26*(1), 2007, 80–113.
[2] Gpgpu.org. http://www.gpgpu.org/.
[3] CUDA Zone. http://developer.nvidia.com/category/zone/cuda-zone.
[4] OpenCL. http://www.khronos.org/opencl/.
[5] NVIDIA Corporation. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, 1.1 edition, 2009.
[6] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, 4.2 edition, April 2012.
[7] NVIDIA Corporation. *CUDA C Best Practices Guide*, 4.1 edition, January 2012.
[8] M. Baskaran, J. Ramanujam, and P. Sadayappan, Automatic C-to-CUDA code generation for affine programs. In *Compiler construction*, volume 6011 of *Lecture notes in computer science* (Berlin/Heidelberg: Springer, 2010) 244–263.

[9] S. Lee, S. Min, and R. Eigenmann, OpenMP to GPGPU: a compiler framework for automatic translation and optimization. *SIGPLAN Notations*, *44*, 2009, 101–110.

[10] N. Sundaram, A. Raghunathan, and S.T. Chakradhar, A framework for efficient and scalable execution of domain-specific templates on GPUs, *International Parallel and Distributed Processing Symposium*, *0*, 2009, 1–12.

[11] Y. Yang, P. Xiang, J. Kong, and H. Zhou, A GPGPU compiler for memory optimization and parallelism management, *SIGPLAN Notations*, *45*, 2010, 86–97.

[12] S. Ueng, M. Lathara, S.S. Baghsorkhi, and W.W. Hwu, CUDA-Lite: reducing GPU programming complexity, in *Languages and compilers for parallel computing* (2008), 1–15.

[13] NVIDIA Corporation, *Thrust quick start guide*, March 2012.

[14] J. Protić, M. Tomašević, and V. Milutinović, Distributed shared memory: concepts and systems, *IEEE Parallel and Distributed Technology*, *4*(2), 1996, 63–79.

[15] S. Raina, Virtual shared memory: a survey of techniques and systems. Technical report, University of Bristol, 1992.

[16] I. Gelado *et al.*, CUBA: an architecture for efficient CPU/co-processor data communication, *Proc. 22nd Annual Int. Conf. on Supercomputing*, ICS '08, 2008, 299–308.

[17] B. Dreier, M. Zahn, and T. Ungerer, The Rthreads distributed shared memory system, *Proc. 3rd Int. Conf. on Massively Parallel Computing Systems*, 1998.

[18] I. Gray and N.C. Audsley, Exposing non-standard architectures to embedded software using compile-time virtualization, *Proc. 2009 Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '09, 2009, 147–156.

[19] Q. Hou, K. Zhou, and B. Guo, BSGP: bulk-synchronous GPU programming, in *ACM SIGGRAPH 2008 papers*, SIGGRAPH '08, 2008, 19:1–19:12.

[20] Himeno benchmark. http://accc.riken.jp/HPC_e/himenobmt_e.html.

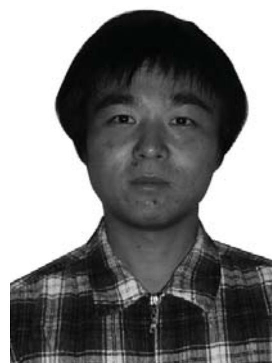
[21] NVIDIA Corporation. *NVIDIA's next generation CUDA compute architecture: Kepler GK110*, 1.0 edition, 2012.

Biographies

Kazuhiko Ohno received the B.S., M.S. and Ph.D. degrees from Kyoto University in 1993, 1995 and 1998, respectively. Since 2003, he has been an associate professor in the Graduate School of Engineering, Mie University. His research interests include the design, implementation and optimization of parallel and distributed programming languages. He is a member of the Information Processing Society of Japan.

Dai Michiura was born in 1987. He received the B.S. and M.S. degrees from the Department of Engineering at Mie University in 2010 and 2012, respectively. He is currently working at JR TOKAI Information System Company. His research interests are parallel processing, heterogeneous environments and HPC environments.

Masaki Matsumoto is a Ph.D. student in the Department of Engineering at Mie University in Japan. He received the B.S. and M.S. degrees from the Department of Engineering at Mie University in 2007 and 2010, respectively. His research interests include parallel computing, task scheduling, grid computing and distributed computing.



Takahiro Sasaki received the B.S., M.S. and Ph.D. degrees from Hiroshima City University in 1998, 2000, and 2003 respectively. He is now assistant professor in the Graduate School of Engineering, Mie University. His research interests are in low-power and high-performance computing, multi-processor architecture, parallel processing and video codec hardware.

Toshio Kondo was born in Nagoya, Japan, on August 28, 1953. He received the B.S. and M.S. and Ph.D. degrees from Nagoya University in 1976, 1978 and 1996, respectively. In 1978, he joined the Musashino Electrical Communication laboratories, Nippon Telegraph and Telephone Corporation (NTT), Tokyo, Japan, and had been engaged in research on SIMD parallel processors, character recognition systems and MPEG encoder LSIs. Since 2000, he has been a professor at Mie University, Mie, Japan. His current research interests include efficient motion estimators for UHDTV video compression, highly parallel processors for object recognition and processor instruction-set extensions for video processing. He is a member of the IEEE Circuits and Systems Society, the Institute of Electronics, Information and Communication Engineers and Information Processing Society of Japan.