International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 6, Number 2, pages 290–308, July 2016

Dynamic Task Scheduling Scheme for a GPGPU Programming Framework

Kazuhiko Ohno, Rei Yamamoto, Hiroaki Tanaka
Department of Information Engineering, Mie University,
1577 Kurimamachiya-cho, Tsu, Mie, 514-8507, JAPAN

Received: February 15, 2016
Revised: May 6, 2016
Accepted: July 13, 2016
Communicated by Hiroyuki Sato

Abstract

The computational power and the physical memory size of a single GPU device are often insufficient for large-scale problems. Using CUDA, the user must explicitly partition such problems into several tasks repeating the data transfers and kernel executions. To use multiple GPUs, explicit device switching is also needed. Furthermore, low-level hand optimizations such as load balancing and determining task granularity are required to achieve high performance. To handle large-scale problems without any additional user code, we introduce an implicit dynamic task scheduling scheme to our CUDA variation *MESI-CUDA*. MESI-CUDA is designed to abstract the low-level GPU features; virtual shared variables and logical thread mappings hide the complex memory hierarchy and physical characteristics. On the other hand, explicit parallel execution using kernel functions is the same as in CUDA. In our scheme, each kernel invocation in the user code is translated into a *job* submission to the runtime scheduler. The scheduler partitions a job into *tasks* considering the device memory size and dynamically schedules them to the available GPU devices. Thus the user can simply specify kernel invocations independent of the execution environment. The evaluation result shows that our scheme can automatically utilize heterogeneous GPU devices with small overhead.

Keywords: GPGPU, CUDA, parallel programming, compiler, optimization, scheduling

1 Introduction

GPUs (Graphics Processing Units) are widely used for high performance computing. Such usage is called GPGPU (General Purpose computation on GPU). However, the computational power and the physical memory size of a single GPU device are often insufficient for large-scale problems. One solution to such cases is to partition the whole computation on GPU into multiple tasks and execute them in turn on a single GPU or in parallel on multiple GPUs.

Using a standard GPGPU programming framework CUDA (Compute Unified Device Architecture) [1], the user must implement such tasks and control them explicitly specifying GPU thread invocations and data transfers. Device switching is also needed to use multiple GPUs. To achieve high performance, tuning task granularity and load balancing is required. Such hand optimizations are difficult and make the performance environment-dependent. An alternative framework OpenACC [2] hides low-level API calls adopting directive-based approach like OpenMP [3]. However, low-level directives are required for the optimizations. Moreover, the current version does not use multiple GPUs automatically [4].

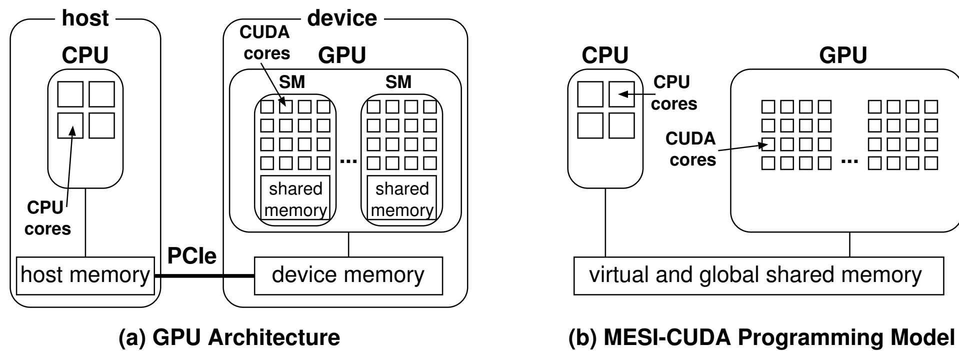


Figure 1: GPU Architecture and Programming Model

To handle large-scale problems without any additional user code, we propose a dynamic scheduling scheme introducing implicit tasks for our CUDA variation *MESI-CUDA* [5, 6, 7, 8]. MESI-CUDA is designed to abstract the low-level GPU features; virtual shared variables and a logical thread mapping scheme hide the complex memory hierarchy and physical characteristics. The API calls for the memory management and data transfers are not needed. The user's thread mapping can be device-independent. On the other hand, explicit parallel execution using kernel functions is the same as in CUDA. The compiler translates user's MESI-CUDA programs to CUDA programs generating low-level code automatically. To achieve high performance without user's specifications, the compiler also makes static analysis and performs the optimizations.

The proposed scheme translates each kernel invocation in the user program into a *job* submission to the runtime scheduler. The job consists of the kernel invocation and the corresponding data transfers. Considering the device memory size, the scheduler dynamically partitions the job to *tasks* and schedules them to the available GPU devices. Thus the user can simply specify kernel invocations without considering the hardware limitations of the execution environment. The user program is expected to be performance portable leaving the optimization to the compiler and runtime system.

This paper is organized as follows: Section 2 gives an introduction of GPU, CUDA, and MESI-CUDA, discussing the current issue. Sections 3, 4 details our scheme and its implementation. Section 5 shows the evaluation results and in Section 6 we discuss the related works. In Section 7, we state the conclusion and future works.

2 Background

2.1 GPU and CUDA

Fig. 1a shows a typical architecture of a NVIDIA GPU card installed on a PC. The PC and the card are called *host* and *device*, respectively.

An NVIDIA GPU is a collection of streaming multiprocessors (SM), which have certain number of CUDA cores. All CUDA cores share a large off-chip *device memory* in the same way that all CPU cores share the main memory (*host memory*). Furthermore, each SM has a small on-chip *shared memory*, which is shared by all CUDA cores in the SM.

CUDA [1, 9, 10] is a GPGPU programming framework for NVIDIA GPUs. It supports C, C++, and Fortran, providing small language extensions and many API functions. Fig. 2 shows a matrix multiplication program using CUDA. The additional code required for parallel programming in CUDA is shown underlined.

Kernel functions, declared with the `__device__` or `__global__` qualifier (Fig. 2 *l*. 5–13), are executed on the device. The other functions (*l*. 14–32), called *host functions* in this paper, are executed on the host. To start computation on the GPU, any host function can invoke a `__global__` kernel function specifying the number of threads (*l*. 26). Then, the created GPU threads execute the kernel function. In this paper, we simply call such GPU threads as *threads*.

```
 1 #define N 4096
 2 #define BX 256
 3 #define S (N*N*sizeof(int))
 4 int ha[N][N], hb[N][N], hc[N][N];
 5 __global__ void matmul(int a[][N], int b[][N], int c[][N]){
 6   int k;
 7   int row = blockDim.y*blockIdx.y+threadIdx.y;
 8   int col = blockDim.x*blockIdx.x+threadIdx.x;
 9   c[row][col] = 0;
10   for(k = 0 ; k < N ; k++){
11     c[row][col] += a[row][k] * b[k][col];
12   }
13 }
14 void init_array(int d[N][N]){...}
15 void output_array(int d[N][N]){...}
16 int main(int argc, char *argv[]){
17   int *da, *db, *dc;
18   dim3 dimGrid(N/BX, N);
19   cudaMalloc(&da, S);
20   cudaMalloc(&db, S);
21   cudaMalloc(&dc, S);
22   init_array(ha);
23   init_array(hb);
24   cudaMemcpy(da, (int*)ha, S, cudaMemcpyHostToDevice);
25   cudaMemcpy(db, (int*)hb, S, cudaMemcpyHostToDevice);
26   matmul<<<dimGrid, BX>>>((int(*)[N])da, (int(*)[N])db, (int(*)[N])dc);
27   cudaMemcpy((int*)hc, dc, S, cudaMemcpyDeviceToHost);
28   output_array(hc);
29   cudaFree(da);
30   cudaFree(db);
31   cudaFree(dc);
32 }
```

Figure 2: CUDA Program of Matrix Multiplication

CUDA uses *grids* and *blocks* for controlling the thread mapping to data and physical resources. A block is a group of threads executed on the same SM, and a grid is a group of blocks of the same size. On the invocation of a kernel function, an *execution configuration* must be specified using an expression of the form `<<<`D_g, D_b`>>>`. D_g and D_b are the sizes of the grid and blocks, respectively. They should be the values of integer or a built-in 3D vector type `dim3`. For example, Fig. 2 program creates a grid of `N/BX` × `N` blocks and each block consists of `BX` threads (Fig. 2 *l*. 18, 26). Using built-in variables to obtain grid/block sizes and block/thread indices, each thread can perform the same computation on the different array element (*l*. 7–12).

To share the data between the CPU and GPU, memory allocations on both memories and data transfers are required. The user must explicitly describe such low-level behaviors calling API functions (*l*. 19–21, 24–25, 27, 29–31). The host-to-device and device-to-host data transfers are called *download* and *readback* transfers, respectively.

If multiple devices are installed on the host, threads can run on the devices in parallel. However, the user must explicitly control each device by switching the target device calling `cudaSetDevice()` and specifying the data transfers and kernel executions individually.

```
 1 #define N 4096
 2 __global__ int ga[N][N], gb[N][N], gc[N][N];
 3 __global__ void matmul(int a[][N], int b[][N], int c[][N]){
 4   int k;
 5   int row = lThreadIdx.y;
 6   int col = lThreadIdx.x;
 7   c[row][col] = 0;
 8   for(k = 0 ; k < N ; k++){
 9     c[row][col] += a[row][k] * b[k][col];
10   }
11 }
12 void init_array(int d[N][N]){...}
13 void output_array(int d[N][N]){...}
14 int main(int argc, char *argv[]){
15   init_array(ga);
16   init_array(gb);
17   matmul<[<N, N>]>(ga, gb, gc);
18   output_array(gc);
19 }
```

Figure 3: MESI-CUDA Matrix Multiplication

2.2 MESI-CUDA

CUDA API directly reflects the complex GPU architecture. Although such low-level API enables hand-tuning considering hardware specifications, it is difficult and not performance portable. Therefore we are developing an easier programming framework *MESI-CUDA* [5, 6, 7, 8].

MESI-CUDA adopts a virtual shared memory model that all cores in both CPU and GPU share a single global memory (Fig. 1b). Actually, only global variables defined with the `__global__` qualifier are shared. To avoid confusion with variables defined with the `__shared__` qualifier, we call our shared variables as *virtual shared variables* or *VS variables*. The values of VS variables are made logically consistent on each kernel invocation, thus explicit synchronizations and mutual exclusions are not needed.

On the other hand, we do not hide explicit parallelization using host and kernel functions. We regard the difference of characteristics between CPU and CUDA cores is not negligible on HPC programming. However, we hide SMs by introducing a logical specification of thread mapping [8] and mapping optimization by the compiler.

Using MESI-CUDA, Fig. 2 program can be simplified as shown in Fig. 3. The additional code required for parallel programming in MESI-CUDA is shown underlined. The arrays for 2D matrices can be defined as VS variables (Fig. 3 *l.* 2) thus they can be accessed from both host and kernel functions (*l.* 7, 9, 15-18). The number of threads can be simply specified as `N`×`N` (*l.* 17). Thus, the user can concentrate on parallel algorithm without low-level API functions.

The MESI-CUDA compiler is implemented as a translator to CUDA code. The low-level code such as the memory management and data transfers is automatically generated. The definitions of VS variables are replaced with the memory allocations and deallocations of the same size on both host and device memories. For the VS variables accessed in the kernel functions, download and readback transfer code is inserted before and after each kernel invocation. Logical thread mappings are also converted to the block-based mapping specifications of CUDA.

To achieve high-performance without user's hand-tuning, the compiler performs optimizations based on the static analysis. For this purpose, we have developed automatic optimization schemes such as overlapping thread executions and data transfers [5], explicit cache using shared memories [7], and thread mappings improving the device memory accesses [8].

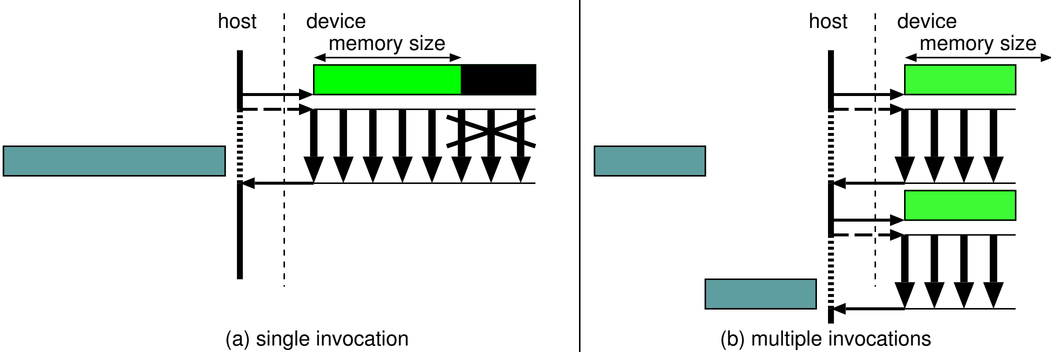


Figure 4: Partitioned Kernel Execution on a Single GPU

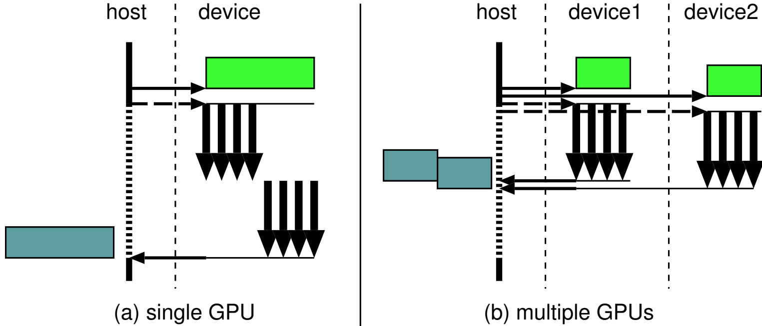


Figure 5: Partitioned Kernel Execution on Multiple GPUs

2.3 Current Issue

Since GPUs do not support virtual memory, the amount of available memory space in each kernel is limited to the size of device memory, which is currently 12GB at maximum. Therefore, large-scale computation cannot be executed in a single kernel invocation if the total size of input/output arrays exceeds the device memory size (Fig. 4a). In most cases, the accessed ranges of the arrays in each thread are limited. Thus the computation can be partitioned into multiple kernel invocations reducing the transfer for each array to the size of accessed range (Fig. 4b).

The computational power of a GPU device may also be insufficient for large-scale problems. Although CUDA enables to create huge number of threads, concurrent or parallel execution of threads and blocks is limited by the physical resources, such as the number of SMs and the sizes of shared memories and register files (Fig. 5a). If multiple GPUs are available, partitioning such a huge kernel invocation into multiple invocations on the devices will reduce the total execution time (Fig. 5b).

Current MESI-CUDA has no support for such cases. However, they should also be handled by the compiler and runtime system without any user's specification, hiding low-level and device-dependent features.

3 Proposed Scheme

We propose a new scheme for MESI-CUDA which automatically partitions kernel invocations and execute them on the available GPUs. To keep compatibility with current MESI-CUDA programs, we introduce implicit creation and dynamic scheduling of jobs and tasks.

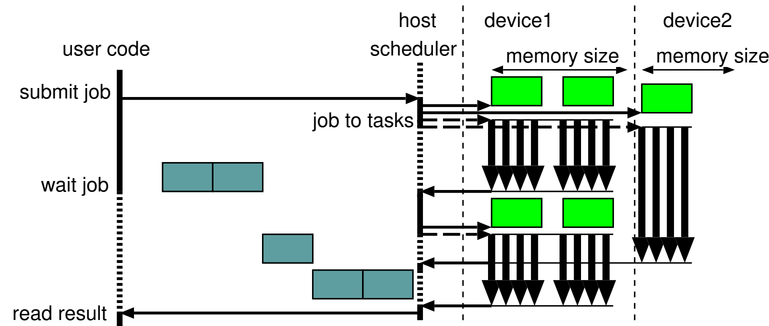


Figure 6: Dynamic Scheduling of Kernel Execution

3.1 Basic Execution Model

A kernel invocation in a MESI-CUDA program is semantically a GPU-accelerated function call. The user expects the kernel is executed immediately and the values of VS variables are updated before the first reference succeeding the invocation. To enable dynamic scheduling without changing this semantics, each kernel invocation and the corresponding data transfers are replaced with a *job* submission to the runtime scheduler. A job consists of the kernel invocation and data transfers of VS variables read or written during the kernel execution. By partitioning the job, the scheduler creates *tasks*. The threads in the original kernel invocation are partitioned into multiple groups and the data transfers are also partitioned or duplicated considering the access ranges of the group. Finally, the tasks are dynamically scheduled to the available GPU devices. A synchronization is performed before the host function accesses any result to ensure all tasks of the job are completed. (Fig. 6).

We expect our target programs satisfy the following assumptions:

A1) VS variables are arrays and all indices are affine expressions of thread indices and loop variables.

A2) Each thread writes to VS variables within consecutive ranges, which are not read/written by any other thread of the same invocation.

A3) Inter-block synchronization functions, such as `atomicAdd()` and `__threadfence()`, are not used.

Assuming A3, A2 must be satisfied for deterministic result.

3.2 Converting Kernel Invocation to Job

A kernel invocation and the corresponding data transfers, submitted as a job, must be performed later by the runtime scheduler. However, our current implementation statically inserts code for them in the host function. Therefore, the compiler is modified to extract the code as callback functions: for each job, functions for the kernel invocation, download transfers, and readback transfers are generated. We call these functions *task functions* because actually they are called when each task of the job is scheduled. The pointers of the corresponding task functions are passed on the job submission so the scheduler can call the functions when needed.

The input values of the kernel function also must be saved for the later execution. The kernel function arguments are usually scalar values because CUDA limits their total size. Thus they can be passed on the submission and stored in the scheduler. On the other hand, VS variables are expected to be large arrays and duplicating them to save the current values is not practical. Therefore, the scheduler controls the progress of the execution in the host function. In our current implementation, the compiler assures that host functions can access the latest values of the VS variables; for each VS variable written in the kernel function, synchronization code is inserted before the earliest read succeeding the kernel invocation. Similarly, our new scheme also inserts synchronization code to prevent overwriting values required on the kernel invocation; for each VS variable read in the kernel function, synchronization code is inserted before the earliest write succeeding the job submission.

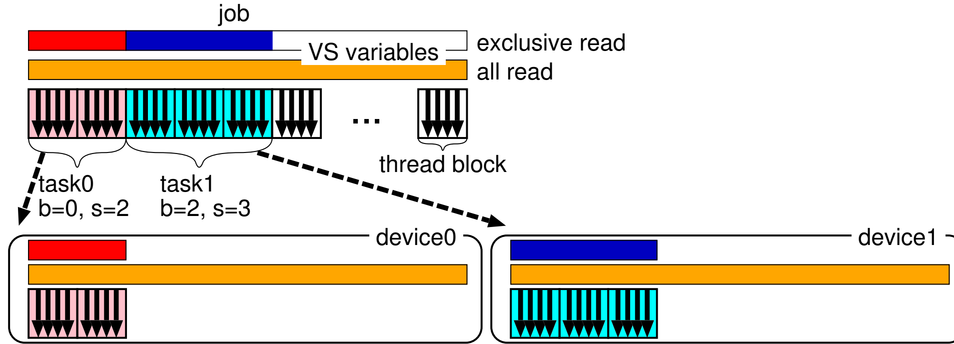


Figure 7: Partitioning Job to Tasks

3.3 Partitioning Job to Tasks

Assuming A2 and A3 in Section 3.1 are satisfied, each thread block of a kernel invocation can be executed independently. Therefore, the thread block can be the unit of job partitioning.

In our scheme, a consecutive segment of thread blocks in the original kernel invocation is assigned to a task (Fig. 7). Adopting a single and consecutive segment, the assignment to each task can be expressed using two integer values b, s, which are the starting block index and the number of blocks of the segment, respectively. In addition, assigning neighboring blocks to a task can expect smaller access ranges in the task because our current scheme optimizes the thread mapping to improve access locality on VS variables [8]. On the other hand, the size of each task, i. e. the number of assigned thread blocks, can be tuned for the optimization. Each task should execute enough blocks to keep the GPU occupancy high, while fewer blocks will reduce the required memory size per task and also provide enough tasks for dynamic load balancing. We do not modify the size of thread blocks because it is automatically determined in the mapping optimization scheme [8] to optimize the execution efficiency in the block. We expect practical programs have enough number of blocks for task partitioning and we regard the disadvantages of disturbing the mapping optimization outweigh the advantages of resizing the block size for the new scheme.

To execute a task, the scheduler must perform the kernel invocation reducing the number of thread blocks to s. The ranges of the corresponding data transfers also must be limited to the access ranges within the task. Assuming A1 and A2 are satisfied, our static analysis can obtain the access ranges of VS variables within a thread block [8]. Therefore, the required range is computed using b and s on the data transfer of each VS variable v.

The memory allocations on the device memory and the array accesses in kernel functions are also modified. On the device memory, the size allocated for v is reduced to the maximum size of access range in the tasks. Although the code in kernel functions expect original block indices, adjusting them requires only constant offsets because consecutive blocks are assigned to a task.

3.4 Optimization

3.4.1 Issues and Strategies

Introducing implicit tasks and dynamic scheduling enables utilizing GPU resources without additional user's specifications. However, it may also cause redundant data transfers and make the execution inefficient.

In our basic execution model, each task performs download and readback transfers of its read and write access ranges in VS variables. Although it ensures that tasks are independent each other and enables the scheduler to assign any task to any device, unnecessary transfers may occur. First, the same data will be transferred on the execution of each task if tasks reading the same range of a VS variable are assigned to the same device. Fig. 8 shows an example that tasks of job 0 update exclusive ranges of a VS variable `a` and read all elements of `b`. The transfer of `b` for the task 2

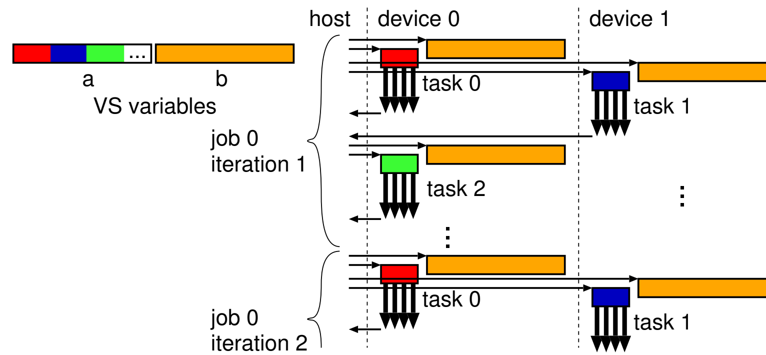


Figure 8: Redundant Data Transfers for Tasks

is redundant because it is already transferred to the device for the task 0. Second, unnecessary transfers between the host and devices occur if a VS variable is iteratively updated by similar jobs. In the Fig. 8 example, the job 0 is submitted multiple times and the task 0 of each job updates the same range of `a`. If the intermediate values are not read on the host and the corresponding tasks are assigned to the same device, the updated range of `a` can be retained on the device memory. However, our execution model forces readback and download transfers on each iteration because the succeeding tasks may run on the different devices.

To suppress such inefficient behaviors, we introduce two techniques: reusing transferred data and fixing task-device mapping. To apply the techniques, we assume the task sizes are fixed for each invocation occurrence in the code.

3.4.2 Data Reusing

To share the transferred data between tasks on the same device, we introduce a dynamic management scheme of VS segments. A VS segment is a partial copy of a VS array variable which contains the array elements within the access range of a task. On the execution of the task, corresponding segments of input VS variables must exist on the device before starting the threads. Instead of allocating and reusing a single area on the device memory for the tasks of a same job, dynamic memory allocation is performed for each segment within the memory capacity.

VS segments can be regarded as caching data from the host memory to each device memories. Therefore, we need a dynamic management scheme to keep the consistency of segments. However, our management problem is much simpler compared with the generic cache coherence problems. First, device memory allocations and data transfers are performed only by the single scheduler thread running on the host. Thus the segments do not require concurrent management; unique management table on the host can be maintained and used by the scheduler thread. Second, detecting device memory accesses at runtime and updating the management table on the host will cause enormous overhead. Therefore, similarly as our other optimization techniques [5, 7, 8], we make static analysis of kernel functions and use the result to identify read/write in the tasks; if the corresponding kernel function has any execution path reading or writing the segment, the task is regarded to read or write the segment, respectively. According to the analysis result, code requesting the scheduler to change the segment state is statically inserted in the task functions.

For each VS variable v_k, a segment table T_k is created. We denote the ith segment of v_k on a device d_j as $s_k(i,j)$. Each segment $s_k(i,j)$ has an entry in T_k which consists of $p_k(i,j)$, a pointer to the allocated memory area, and $f_k(i,j)$, a valid flag of the the segment. On the execution of a task t on a device d_j, data transfers and segment table management are performed as follows:

1. On starting t, for each segment $s_k(i,j)$ accessed in t:

    (a) If $p_k(i,j)$ is null, allocate memory for the segment and store the pointer. $f_k(i,j)$ is initialized as *Invalid*.

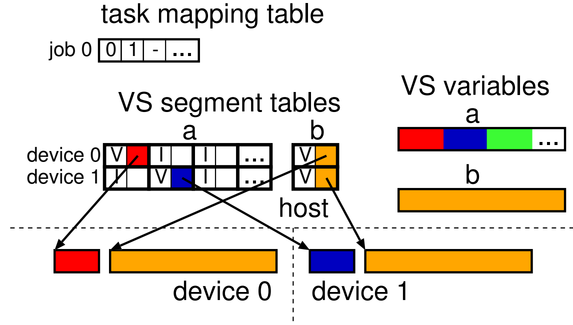


Figure 9: VS Segment Table and Task Mapping Table

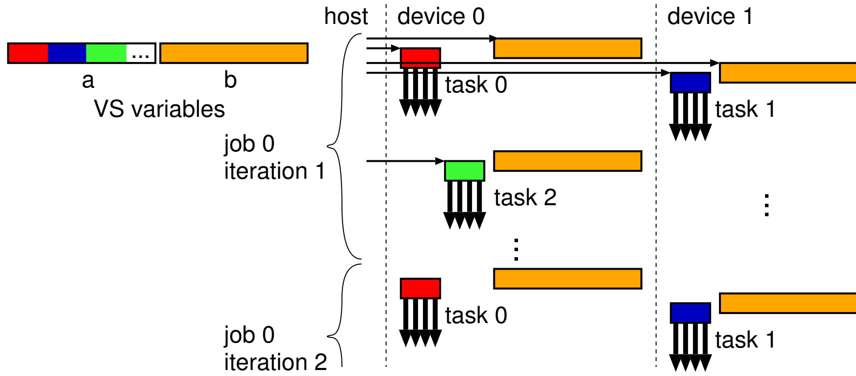


Figure 10: Optimized Data Transfers for Tasks

(b) If $s_k(i,j)$ is read in t and $f_k(i,j)$ is *Invalid*, perform a data transfer and change $f_k(i,j)$ to *Valid*.

(c) If $s_k(i,j)$ is written in t, change all $f_k(i,j')$ to *Invalid* where $j' \neq j$.

Note that if $s_k(i,j)$ is both read and written, i. e. updated, (b) and (c) are both applied.

2. On finishing t, for each segment $s_k(i,j)$ written in t:

   (a) If the segment values are read in the host code, perform a readback transfer.

In this scheme, the segments updated by a task are transferred back to the host only if they are read on the host (2a). Otherwise, the latest values are retained on the device memory. Similarly, the data transfer is skipped if the segment has the latest values (1b). However, in the case that the segment is invalid, a data transfer from the appropriate source is needed; the latest values may be either on the host or other devices. Using static analysis result, the compiler can identify whether the latest write occurs in host functions or kernel functions. Therefore the compiler statically select the transfer code: a download transfer from the host or a device-to-device transfer. For the latter case, the source device is dynamically determined by the runtime system. It checks the segment table and find the device whose corresponding entry is valid.

Fig. 9 illustrates segments and tables when the task 1 of the first job 0 submission has started in Fig. 8. When the task 2 is scheduled, it requires a download transfer for a new segment of `a`. However, the transfer for `b` is not needed because the segment is shared with the preceding task 0. Similarly, the task 0 of the second job 0 submission can reuse the segment of `a`. Therefore, data transfers are reduced as shown in Fig. 10.

3.4.3 Fixed Task-Device Mapping

Dynamic reusing of VS segments can eliminate redundant transfers only if tasks sharing the same segment are assigned to the same device. For example in Fig. 10, the task 0 of second job 0 submission cannot reuse the segment of `a` if it is executed on the device 1.

However, generic task scheduling strategy considering the reusability is not easy. Immediate assignment to an idle device without the required segment may or may not be better than waiting for the device with the segment to be idle. Therefore, we use a simple optimization strategy. The occurrence of a kernel invocation in a loop causes multiple execution of similar kernel threads. In many cases like step simulations, the thread block of the same block index in each execution accesses the same range of VS variables. For each invocation occurrence, the scheduler maintains a task mapping table that records the assigned device of each task. If the invocation occurrence is executed again, the tasks are mapped to the devices in the same way as in the previous execution so that the VS segments on each device can be reused.

For example in Fig. 9, the task 0 and 1 of the first job 0 submission have been executed and the task mapping table for the job 0 records their assigned devices. On the execution of the task 2 in the first submission, the device will be dynamically determined because the corresponding entry of the table is invalid. When the task 0 of the second job 0 submission is executed later, it will be assigned to the device 0 according to the table entry.

4 Implementation

4.1 Data Structure

The jobs and tasks are managed using job and task records shown in Fig. 11. A job record contains a job ID, pointers to task functions and an array of union holding kernel invocation arguments. Other fields are used to control job behavior; the number of unfinished blocks, the task size of the job, a flag for fixing task-device mapping, and condition and mutex variables for synchronizing with the user code. A task record contains a task ID, a pointer to the corresponding job record, and the number of assigned blocks.

A unique job ID is assigned to each occurrence of the job submission in the code, not to each job instance. It is used to identify *similar* jobs for reusing task-device mapping. As for the task ID, it is locally unique; regarding tasks in a job as an array of tasks, it is the index of each task and used to identify its data access range.

To implement the optimization techniques in Section 3.4, we also use VS segment tables and task mapping tables. For each VS variable, a 2-dimensional array of size $n_s \times n_d$ is created as the segment table, where n_s and n_d are the numbers of segments and available devices, respectively. As explained in Section 3.4.2, each array element is a struct of a pointer to the allocated memory area and a valid flag. Similarly, for each occurrence of a job submission in the code, an integer array of size n_t is created as the task mapping table where n_t is the number of tasks in the job. Each array element stores the device ID which the corresponding task is assigned on the preceding execution of the job. The number of tables and each size depends on both user programs and execution environments. For example, the numbers of kernel invocation occurrences in the code and available GPU devices are needed. Therefore, the runtime system dynamically creates the tables on demand.

4.2 Code Translation

The compiler first performs the original code translation; the memory management and data transfer code is generated and the logical mappings are converted to the CUDA thread mappings. Then the following procedures are applied to translate it into the code using dynamic scheduling. As an example, Figs. 12, 13 shows the result of code translation on Fig. 3 program. For simplicity, we assume the size of each task is a multiple of `N/_B`, i.e. the computation of a row is not partitioned to multiple tasks.

```
typedef struct _job_record{
  int id;
  void (*malloc_func)(int tid, int ts);
  void (*dl_func)(int tid, int ts);
  void (*rb_func)(int tid, int ts);
  void (*update_func)(int tid, int ts);
  void (*exe_func)(struct _job_record *job, int tid, int ts, cudaStream st);
  union {
    char c; int n; ..., _segtable_t *p;
  } args[ARG_MAX];
  int block_ctr, task_size, fix_device;
  pthread_mutex_t dl_mutex, rb_mutex;
  pthread_cond_t dl_cond, rb_cond;
  struct _job_record *next;
} _job_record_t;

typedef struct _task_record{
  int id;
  _job_record_t *job;
  int block_num;
  struct _task_record *next;
} _task_record_t;
```

Figure 11: Job/Task Records

4.2.1 Initialization and Finalization

At the beginning and end of the program, function calls for initializing and finalizing the runtime scheduler are inserted, respectively (Fig. 12 *l.* 17, 39). To enable reusing VS segments, the usage of the device memory is encapsulated in the runtime system. Pointers for the device memory are modified to point corresponding segment tables (*l.* 15). Each device memory allocation for a VS variable is replaced with the creation of a segment table (*l.* 21–23).

4.2.2 Task Functions

For each occurrence of a kernel invocation in the code, a set of task functions are generated.

The runtime system must allocate memory area for each task on demand because the device memory allocations for VS variables are removed from the initialization code. Required VS segments and their sizes depend on the user code while device memory management is hidden in the runtime system. Thus the runtime system calls a *malloc* task function which demands all VS segments required by the task. In the case of Fig. 3 program, each task accesses exclusive ranges of `ga` and `gc` but accesses all range of `gb`. For the former variables, the compiler generates code specifying multiple segments, each exclusively used by the corresponding task. For the latter variable, the compiler generates code specifying single segment which is shared by all tasks (Fig. 13 *l.* 41–45). The runtime system checks the segment table for each demand and allocate memory of specified size if the segment is invalid.

Download and readback transfers are also extracted as *download* and *readback* task functions (*l.* 46–51, 52–55). The functions compute required range of each transfer using the arguments `tid` (task ID) and `ts` (number of blocks). The addresses of target segments, required for the data transfer, are obtained from the segment table and segment index. Instead of calling `cudaMemcpy()` directly, code requesting the runtime system to transfer the data is inserted. Therefore, the low-level control of the devices is hidden from the task function. It also enables the runtime system to skip

```
 1 #define N 4096
 2 #define _B 256
 3 __global__ void matmul(int _b, int a[][N], int b[][N], int c[][N]){
 4   int k;
 5   int row = (blockIdx.x+_b-_b) / (N / _B);
 6   int col = (blockIdx.x+_b) % (N / _B) * _B + threadIdx.x;
 7   c[row][col] = 0;
 8   for(k = 0 ; k < N ; k++){
 9     c[row][col] += a[row][k] * b[k][col];
10   }
11 }
12 void init_array(int d[N][N]){...}
13 void output_array(int d[N][N]){...}
14 int *_h_ga, *_h_gb, *_h_gc;
15 _segtable_t *_s_ga, *_s_gb, *_s_gc;
16 int main(int argc, char *argv[]){
17   _scheduler_init();
18   cudaMallocHost((void**)&_h_ga, sizeof(int)*N*N);
19   cudaMallocHost((void**)&_h_gb, sizeof(int)*N*N);
20   cudaMallocHost((void**)&_h_gc, sizeof(int)*N*N);
21   _s_ga_ = _scheduler_create_segtable();
22   _s_gb_ = _scheduler_create_segtable();
23   _s_gc_ = _scheduler_create_segtable();
24   init_array((int (*)[N])_h_ga);
25   init_array((int (*)[N])_h_gb);
26   _job_record_t *_job = _scheduler_get_job_record();
27   _job->id = 0;
28   _job->malloc_func = _malloc_matmul_0;
29   _job->dl_func = _dl_matmul_0;
30   _job->rb_func = _rb_matmul_0;
31   _job->exe_func = _exe_matmul_0;
32   _job->args[0].p = _s_ga;
33   _job->args[1].p = _s_gb;
34   _job->args[2].p = _s_gc;
35   _job->block_ctr = N*N/_B;
36   _scheduler_submit(_job);
37   _scheduler_wait_job_end(_job);
38   output_array((int (*)[N])_h_gc);
39   _scheduler_terminate();
40 }
```

Figure 12: Dynamic Scheduling-based Code Generated from Fig. 3 Program

transfers for valid segments.

The kernel invocation in the original code is moved to the *execution* task function (*l*. 56–61) with some modifications. The number of thread blocks are reduced to the task size `ts` and the start block index `tid*bs` is passed to the kernel function. Similar to the transfer task functions, the addresses of segments are obtained and passed as the arguments of the kernel function.

4.2.3 Job Submission

Instead of the original kernel invocation, the code submitting a job to the scheduler is inserted. To create a job, a job record is obtained from the scheduler (Fig. 12 *l*. 26) and the code setting

```
41 void _malloc_matmul_0(int tid, int ts){
42   _scheduler_segmalloc(_s_ga, tid, ts*_B*sizeof(int));
43   _scheduler_segmalloc(_s_gb, 0, sizeof(int)*N*N);
44   _scheduler_segmalloc(_s_gc, tid, ts*_B*sizeof(int));
45 }
46 void _dl_matmul_0(int tid, int ts){
47   void *d_ga = _scheduler_segtable2addr(_s_ga, tid);
48   void *d_gb = _scheduler_segtable2addr(_s_gb, 0);
49   _scheduler_memcpy_h2d(d_ga, _h_ga+tid*ts*_B, ts*_B*sizeof(int));
50   _scheduler_memcpy_h2d(d_gb, _h_gb, sizeof(int)*N*N);
51 }
52 void _rb_matmul_0(int tid, int ts){
53   void *d_gc = _scheduler_segtable2addr(_s_gc, tid);
54   _scheduler_memcpy_d2h(_h_gc+tid*ts*_B, d_gc, ts*_B*sizeof(int));
55 }
56 void _exe_matmul_0(_job_record_t *job, int tid, int ts, cudaStream_t st){
57   void *d_ga = _scheduler_segtable2addr(job->args[0].p, tid);
58   void *d_gb = _scheduler_segtable2addr(job->args[1].p, 0);
59   void *d_gc = _scheduler_segtable2addr(job->args[2].p, tid);
60   matmul<<<ts, _B, 0, st>>>
         (tid*ts, (int (*)[N])d_ga, (int (*)[N])d_gb, (int (*)[N])d_gc);
61 }
```

Figure 13: Task Functions Code Generated from Fig. 3 Program

the record members is inserted. First, the pointers to the task functions of the job are assigned (l. 28–31). Second, the argument values of the kernel function are saved in the job record (l. 32–34). Because the number and types of arguments differ in each kernel function, the job record has an array of union. On the job creation, the arguments are assigned to the array using appropriate union members. The number of arguments and each type are not needed to be saved because the arguments are handled by the execution task function and the scheduler does not directly access them. Third, the total number of thread blocks is stored in `block_ctr` for checking the completion of all tasks. (l. 35). And finally, the record is passed to the scheduler to submit the job (l. 36).

After submitting a job, the scheduler asynchronously executes it as multiple tasks. Therefore, the user code continues its execution. However, it must wait for the job to be finished before accessing the result. For this purpose, synchronization code is inserted (l. 37) which makes the caller suspended until the specified job is completed.

4.2.4 Kernel Functions

To adjust block indices, the starting block index `_b` is passed to the kernel function. Each occurrence of `blockIdx.x` is replaced with `blockIdx.x+_b` (Fig. 12 l. 5–6). For index expressions of `a` and `c`, another offset `-_b` is required because they are truncated on the device memory. In this code, `row` only occurs in the index expressions of `a` and `c`, while `col` only occurs in the index expressions of `b`. Thus the offset `-_b` is added to `row` (l. 5).

4.3 Scheduler

The runtime scheduler accepts job submissions from the user code, watches available devices, and assigns a task when an idle device is detected. To enable asynchronous dynamic scheduling, `_scheduler_init()` (Fig. 12 l. 17) creates a scheduler thread on the host. On the initialization, the scheduler checks available devices and allocates CUDA streams for each device. The streams are used for the concurrent execution of data transfers and kernel invocations on the devices.

Table 1: Evaluation Programs

`matmul`	matrix multiplication of Fig. 3 (`N`=8192)
`ep`	The Embarrassingly Parallel in NPB [11] (class D)
`hotspot`	2D transient thermal simulation [12] (size:8192^2, 1000 steps)

Table 2: Evaluation Environments

	CPU	Memory	GPUs
host 1	Xeon E5-2630 2.4GHz×2	32GB	Tesla K80
host 2	Core i7-4820K 3.7GHz	16GB	GeForce GTX TITAN, GeForce GTX 680

The scheduler thread watches the job queue and devices. When `_scheduler_submit()` is called in the host function (l. 36), the job record is enqueued to the job queue. The scheduler thread dequeues a job, create tasks for each idle device. To execute each task, the scheduler calls corresponding task functions. Because each function issues asynchronous data transfers and a kernel invocation to the specified stream, download transfers, a kernel invocation, and readback transfers are booked to be executed sequentially.

When a kernel execution is finished on a device, the booked readback transfers are automatically executed by the stream. However, the scheduler must check if all tasks of a job are completed and notify it to the user thread because the user code waits for the completion of submitted jobs. The completion order of tasks is nondeterministic on multiple GPUs. Therefore, the block counter `block_ctr` of the job record is used to detect the completion of all tasks of the job; the number of blocks in each task, stored in the `block_num` member of the task record, is subtracted from the block counter when the task is finished. If the counter value reaches zero, all blocks of the job is completed. Then the scheduler sends a signal to the condition variable of the job record, which wakes the user thread if it is suspended calling `_scheduler_wait_job_end()`(l. 37).

5 Evaluation

We evaluated our scheme using three benchmark programs shown in Table 1. As the evaluation environments, we used two hosts shown in Table 2. Tesla K80 has two same GPUs on the board and is recognized as two devices by the software. Thus the host 1 can be used as a homogeneous multi-GPU environment. On the other hand, two different GPU devices are installed on the host 2 and can be used as a heterogeneous multi-GPU environment. In the following sections, we use the notations 'K80(2)' and 'TITAN+680' for these hosts when both devices are enabled. If only a single GPU is enabled, we use the notations 'K80(1)', 'TITAN', and '680', corresponding to the available device.

5.1 Speedup using Proposed Scheme

We compared the execution time of each benchmark program using the current MESI-CUDA implementation ('original') and the new implementation adopting the proposed scheme ('proposed'). The results are shown in Fig. 14.

To execute the original versions on multi-GPU environments, we manually made minimum modifications; thread blocks are statically partitioned in equal and each original kernel invocation is replaced with two invocations for each device. Thus the workloads are statically scheduled without considering the performance of devices. For the proposed versions, the original kernel invocations are converted into jobs, partitioned into tasks, and dynamically scheduled using *master-worker* load balancing. On single-GPU environments, the original versions are executed without the modifications while the proposed versions always perform dynamic task scheduling.

On single GPU, applying our scheme to `matmul` slightly increased the execution time (Fig. 14a). Compared with the original version which invokes the kernel only once, the proposed version invokes the kernel for each task. However, the result shows such overhead is negligible. On K80(2), the proposed version caused similar slowdown. The simple static scheduling of the original version can achieve best performance to execute uniform computations on homogeneous multi-GPUs thus the proposed version has no advantage in this case. However, the result shows the overhead of dynamic scheduling on multi-GPUs is also negligible. On the other hand, the execution time of the proposed version was reduced to 76% of the original on TITAN+680, due to the dynamic load balancing.

Applying our scheme to `ep`, the execution time was reduced on two single GPU environments K80(1) and 680 (Fig. 14b). To handle large problem space, the original version invokes the kernel multiple times. As explained in Section 4.3, our implementation of dynamic scheduling uses CUDA streams. Therefore, the kernel executions and data transfers can be overlapped in the proposed version. Because the execution time largely differs on TITAN and 680, the original version on TITAN+680 is slower than on single TITAN. Due to the dynamic load balancing, the execution time of the proposed version on TITAN+680 is 40% of the original and 90% of the execution time on single TITAN.

Simply applying our scheme to `hotspot` caused drastic slowdown (Fig. 14c). Because it is a step simulation program that iteratively executes similar jobs, redundant data transfers occur as explained in Section 3.4.1. Applying the optimization schemes described in Sections 3.4.2 and 3.4.3, the execution time of the proposed version was improved as shown in Fig. 14d. Compared with the original version, the overhead on single GPU and homogeneous multi-GPUs are negligible. On the heterogeneous multi-GPUs, the execution time is reduced to 89% of the original.

The evaluation result shows that the execution time ratio between different devices largely depends on the target applications. Therefore, appropriate static load balancing for heterogeneous multi-GPUs is difficult even if the execution environment is known at compile time. Our scheme performs dynamic load balancing without any specification for multi-GPUs. Thus it can achieve better performance without user's additional effort.

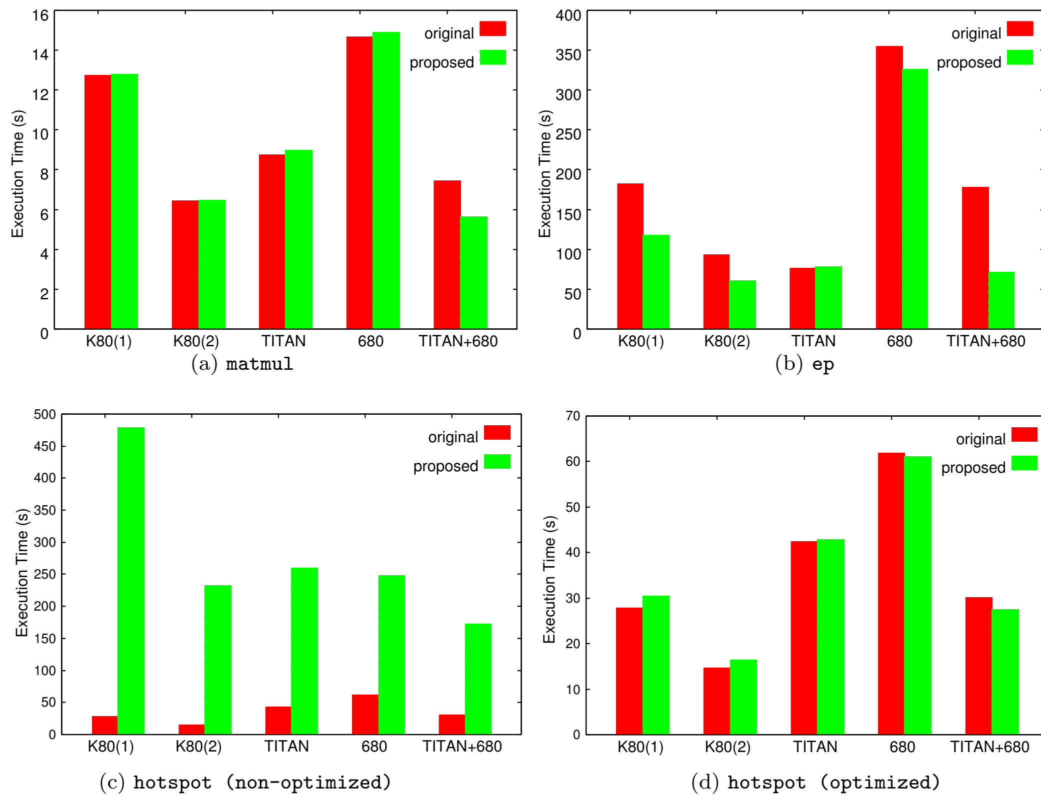
5.2 Impact of Task Granularity

To evaluate the impact of task granularity to the execution time, we compared the execution time of the proposed version changing the number of tasks per job. The result is shown in Fig. 15. The performance of 2 tasks per job is approximately same as the performance of the original version because each device runs only one task of the same size.

The result of `matmul` (Fig. 15a) shows that 8 tasks per job can reduce the execution time on TITAN+680 compared with 2 or 4 tasks per job. As shown in Fig. 14a, TITAN and 680 shows different performance on executing `matmul` tasks and 4 tasks or less cannot balance the load properly. Increasing tasks to 16 or more did not achieve further improvement. The execution time on single GPU and K80(1) are slightly affected by the number of tasks. In this case, 8 tasks would be enough for the load-balancing but the overhead increase is negligible up to 128 tasks.

On the other hand, the execution time of `ep` (Fig. 15b) is much sensitive to the task granularity. The best performance is achieved with 16 tasks per job on TITAN+680, and 64 or more tasks largely increase the execution time. It is notable that the task granularity largely affects the performance even on single GPU and homogeneous multi-GPUs, and the best number of tasks is dependent to the devices. While 680, K80(1), and K80(2) achieve best performance with 64 tasks, increasing number of tasks always declines the performance on TITAN. As shown in Fig. 14b, the proposed scheme improved the performance on 680, K80(1) and K80(2) but not on TITAN. The efficiency of overlapping executions and transfers would have been affected by the task granularity on the former environments. On the other hand, such effect is not achieved on the latter environment thus larger number of tasks simply increased the overhead.

Applying the optimization schemes, the result of `hotspot` (Fig. 15c) shows behaviors similar to `matmul`; 8 tasks per job reduced the execution time on the heterogeneous multi-GPUs. However, the overhead increase of smaller task granularity is not negligible in this case. On all environments, 64 or more tasks should be avoided.



(a) `matmul` (b) `ep`

(c) `hotspot (non-optimized)` (d) `hotspot (optimized)`

Figure 14: Execution Time

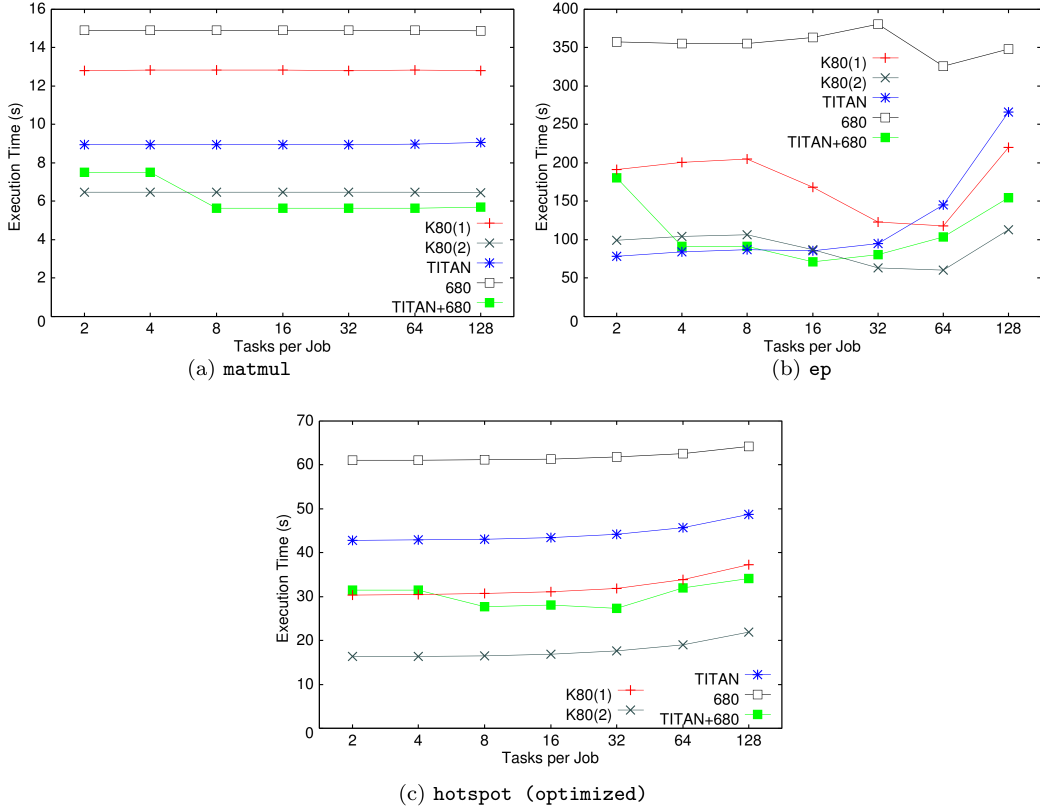
As we mentioned in Section 3.3, the task granularity causes a trade-off. Larger number of tasks reduces required memory per task and enables flexible load-balancing. However, the scheduling overhead will be increased, and the GPU occupancy may be declined because thread blocks per task is reduced. To avoid the latter disadvantages, the number of tasks should be suppressed. The optimal number of partitioning depends on both target program and execution environment. Developing an automatic tuning scheme of the task granularity will be our future work. Especially, granularity-sensitive cases like `ep` need more evaluations and investigations. However, the current evaluation result shows a rough strategy; if enough memory is available, partitioning each job to 8–16 tasks will be sufficient for load-balancing two devices and can minimize the drawbacks.

6 Related Works

AMGE [13] also provides a programming framework which automatically decomposes a kernel invocation for multiple GPUs. Although static analysis and code translation are also used, it is a runtime-based system which allocates arrays to distributed device memories and provides transparent access from the kernel functions. On the other hand, our scheme is mainly compiler-based approach with the minimum runtime system, transforming code for various optimizations. We also support partitioned execution on the same device for the case that the data size exceeds total size of device memories.

Wende et al. proposes a reordering scheme of kernel invocations [14]. As opposed to our scheme, they target concurrent execution of small-scale multiple kernels on a single device. However, similar technique may improve the performance of our scheme when tasks of different jobs can be overlapped.

Chen et al. proposes a task-based dynamic load-balancing scheme for single and multi GPU environments [15]. Their idea is to launch persistent threads on GPUs which executes tasks on



(a) matmul (b) ep

(c) hotspot (optimized)

Figure 15: Impact of Task Granularity

demand. The host code only need to enqueue tasks to the task queue and the task mapping to available devices can be hidden in the runtime system. Although such approach can eliminate the overhead of kernel invocations, it may cause certain inefficiency. The optimal size of thread blocks depends on the usage of registers and shared memories per thread thus it should be tuned for each kernel function. In MESI-CUDA, thread mapping including the block size is determined for each kernel invocation based on the static analysis of user code.

Huynh et al. proposes a framework which maps workloads to multi-GPU environments [16]. Their scheme is based on graph partitioning and has shown scalable performance. However, the scheme targets on their data flow programming language thus applying to CUDA or MESI-CUDA is not easy.

An extension of OpenACC to support multiple GPUs is proposed by Komoda et al. [4]. Because OpenACC generates low-level parallelizing code from a sequential user program, their compiler can utilize multiple GPUs without controlling them in the user code. However, low-level directives specifying local data access and reduction operations are needed.

7 Conclusion and Future Works

Although GPUs are widely used as high performance computing platforms, the computational power and the physical memory size of a single device are often insufficient for large-scale problems. In this paper, we proposed a dynamic scheduling scheme for a CUDA variation MESI-CUDA to solve the issue. Our scheme introduces implicit jobs and tasks. The compiler translates each kernel invocation in the user code into a job submission. The runtime scheduler partitions the job into tasks considering the device memory size and dynamically schedules them to the available devices. Thus large-scale problems can be handled without any additional user code.

The evaluation result shows that the overhead of the proposed scheme is negligible and can improve performance utilizing multiple GPUs. It also shows that our optimization techniques can suppress redundant data transfers caused by the scheme. However, further optimization techniques will be needed for more complicated cases.

Another future work is to support large-scale execution environments which consist of multiple hosts. Currently we are developing proxy processes for remote hosts which control devices on the host and make them available from our scheduler. To support user's host-level parallel programming using OpenMP or MPI, we also need to modify our static analysis and the scheduler implementation to be thread-safe.

Acknowledgment

This work was partially supported by Toyota Riken Scholar.

References

[1] CUDA Zone. `https://developer.nvidia.com/cuda-zone`.

[2] OpenACC Home. `http://www.openacc-standard.org/`.

[3] OpenMP. `http://www.openmp.org/`.

[4] T. Komoda, S. Miwa, H. Nakamura, and N. Maruyama. Integrating multi-GPU execution in an OpenACC compiler. In *Proc. 42nd Intl. Conf. on Parallel Processing*, pages 260–269, 2013.

[5] K. Ohno, D. Michiura, M. Matsumoto, T. Sasaki, and T. Kondo. A GPGPU programming framework based on a shared-memory model. *Parallel and Distributed Computing and Networks*, 3:1–14, 2013.

[6] K. Ohno, M. Matsumoto, T. Kamiya, and T. Maruyama. Supporting dynamic data structures in a shared-memory based GPGPU programming framework. In *Proc. 24th IASTED Intl. Conf. on Parallel and Distributed Computing and Systems*, pages 122–131, 2012.

[7] T. Kamiya, T. Maruyama, K. Ohno, and M. Matsumoto. Compiler-level explicit cache for a GPGPU programming framework. In *Proc. The 2014 Intl. Conf. on Parallel and Distributed Processing Techniques and Applications*, pages 632–638, 2014.

[8] K. Ohno, T. Kamiya, T. Maruyama, and M. Matsumoto. Automatic optimization of thread mapping for a GPGPU programming framework. In *2014 2nd Intl. Symp. on Computing and Networking*, pages 198–204, 2014.

[9] NVIDIA Corporation. *CUDA C Programming Guide*, 2012.

[10] NVIDIA Corporation. *CUDA C Best Practices Guide*, 2012.

[11] NAS parallel benchmarks, NASA advanced supercomputing division. `https://www.nas.nasa.gov/publications/npb.html`.

[12] S. Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. IEEE Intl. Symp. on Workload Characterization*, pages 44–54, 2009.

[13] J. Cabezas et al. Automatic execution of single-GPU computations across multiple GPUs. In *Proc. 23rd Intl. Conf. on Parallel Architectures and Compilation*, pages 467–468, 2014.

[14] F. Wende, F. Cordes, and T. Steinke. On improving the performance of multi-threaded CUDA applications with concurrent kernel execution by kernel reordering. In *2012 Symp. on Application Accelerators in High Performance Computing*, pages 74 – 83, 2012.

[15] Long Chen, O. Villa, S. Krishnamoorthy, and G.R. Gao. Dynamic load balancing on single- and multi-GPU systems. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, 2010.

[16] Huynh Phung Huynh, Andrei Hagiescu, Weng-Fai Wong, and Rick Siow Mong Goh. Scalable framework for mapping streaming applications onto multi-GPU systems. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 1–10, 2012.