# Orchestrating Multiple Data-Parallel Kernels on Multiple Devices

Janghaeng Lee
University of Michigan, Ann Arbor
jhaeng@umich.edu

Mehrzad Samadi
University of Michigan, Ann Arbor
mehrzads@umich.edu

Scott Mahlke
University of Michigan, Ann Arbor
mahlke@umich.edu

## Abstract

Traditionally, programmers and software tools have focused on mapping a single data-parallel kernel onto a heterogeneous computing system consisting of multiple general-purpose processors (CPUS) and graphics processing units (GPUs). These methodologies break down as application complexity grows to contain multiple communicating data-parallel kernels. This paper introduces MKMD, an automatic system for mapping multiple kernels across multiple computing devices in a seamless manner. MKMD is a two phased approach that combines coarse grain scheduling of indivisible kernels followed by opportunistic fine-grained workgroup-level partitioning to exploit idle resources. During this process, MKMD considers kernel dependencies and the underlying systems along with the execution time model built with a few sets of profile data. With the scheduling decision, MKMD transparently manages the order of executions and data transfers for each device. On a real machine with one CPU and two different GPUs, MKMD achieves a mean speedup of 1.89x compared to the in-order execution on the fastest device for a set of applications with multiple kernels. 52% of this speedup comes from the coarse-grained scheduling and the other 48% is the result of the fine-grained partitioning.

## 1. INTRODUCTION

Over the past decade, heterogeneous computer systems that combine multicore processors (CPUs) with graphics processing units (GPUs) have emerged as the dominant platform for general-purpose computing. New programming models, such as OpenCL [15] and CUDA [25], enable programmers to efficiently develop data-parallel *kernels* to execute on GPUs. As more application domains focus on exploiting the computational power of GPUs, the complexity of the applications being mapped onto heterogeneous systems has increased. Applications will grow from a single kernel surrounded by the corresponding setup code, to a multitude of communicating data parallel kernels with interspersed CPU code that require exploiting all processing resources (CPUs and GPUs) to achieve the desired performance level.

Unfortunately, applications with several data parallel kernels are difficult to efficiently map onto multiple CPUs and GPUs for three main reasons. First, the mapping decision must be made depending on the number of available computing devices, being aware of their performance capability. Second, kernel execution time is varied by the input size, so kernels must be mapped considering the input size of each kernel. However, programmers cannot determine the input size that will be used for the real execution. Third, it is hard to fully utilize computing resources due to kernel dependencies. If no kernel can be executed in parallel due to dependencies, kernels should be mapped to a single device in serial resulting in other devices being idle.

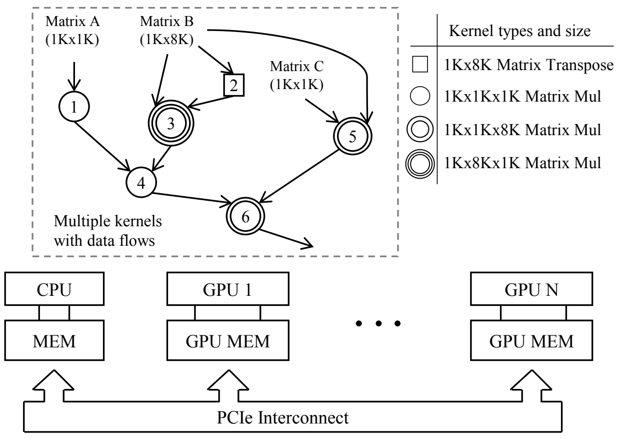


Figure 1: A kernel graph for solving a matrix equation, $A^2BB^TCB$, consisting of six kernels. The system is equipped with different computing devices with separated physical memory. Devices are connected through PCI express (PCIe) interconnect. Each kernel has different amount of computation, and each device has different performance.

To explain these observations, Figure 1 illustrates an example application with multiple kernels for solving a matrix equation, $A^2BB^TCB$. In the equation, $A$ and $C$ are $1K \times 1K$ matrices, and $B$ is a $1K \times 8K$ matrix. The target system has different devices each of which shows different performance on different kernels. First, kernels 1, 2, and 5 in Figure 1 are not dependent on each other, thus they can be executed in parallel on separate devices if there are enough devices. However, it is difficult for programmers to allocate resources efficiently as they do not know the target system at compile time.

Next, kernels 1 and 5 in Figure 1 are the same code, but have different computation cost due to the input size. For this reason, even though programmers target a specific system, they cannot statically decide which kernel should be mapped to a faster device.

Last, kernel 6 in Figure 1 must be executed alone after all other kernels are finished, which leaves the other devices idle. However, the performance can be further improved by splitting kernel 6 into sub-kernels, and mapping them to all devices.

To address these challenges, this paper proposes *MKMD*, or **multiple kernels on multiple devices**, a runtime system that combines temporal scheduling of multi-kernels along with spatial partitioning of data/computation across multiple computing devices. The objective of MKMD is to complete all kernels and CPU code in the least time. To achieve this goal, MKMD proposes a two-phase scheduling approach considering the expected kernel execution time, data transfer cost, and available bandwidth of the interconnect. The first phase is coarse-grain scheduling, which constructs a kernel graph and schedules at a kernel granularity maximizing the resource utilization. This phase assumes kernels must be entirely executed by a single device. The second phase is fine-grain partitioning, which reschedules kernels at the work-group (thread-block) granularity by spatially partitioning kernels into sub-kernels across available computing devices. In this manner, this phase removes idle computing periods on devices to reduce kernel execution time.

As MKMD schedules kernels before their execution, it must be aware of the execution time for each kernel on each device for the given input sizes. Offline profiling can be used for the execution time estimation, but profiling all combinations of kernels, devices and different input sizes is time-consuming and often infeasible. In order to estimate the execution time with a few sets of offline profile data, MKMD builds a regression model for each kernel on each device, and uses the model for the different input sizes.

With MKMD, programmers are only responsible for enqueuing data-parallel kernels to MKMD without worrying about mapping kernels to target devices or splitting a kernel into sub-kernels. The contributions of this paper are as follows:

- Input-variant performance estimation methodology that is specialized for data-parallel kernels.
- Mapping the list of data-parallel kernels to a task scheduling problem where the goal is to assign kernels to compute devices cognizant of execution capabilities and data transfer times.
- A fine-grain kernel partitioning algorithm that identifies idle time slots and splits kernel execution across multiple idle devices.

## 2. BACKGROUND

For MKMD, OpenCL is chosen as the input language because both CPU and GPU vendors support OpenCL, while the language supports a variety of different architectures with explicit support for multiple communicating kernels. For a better understanding of scheduling kernels at finer (work-group) granularity, this section briefly discusses the background on OpenCL execution model and OpenCL kernel decomposition.

### 2.1 OpenCL Execution Model

In OpenCL, the basic unit of execution is a single *work-item* which corresponds to a thread. A group of work-items executing the same code are weaved together to form a *work-group*. These work-groups are combined to form a unit of execution called *NDRange, N-Dimensional Range*, where each NDRange is scheduled by a command queue. For execution, the OpenCL program assumes that the underlying devices consist of a number of compute units (CUs) which are further split into processing elements (PEs). When executing a kernel, work-groups are mapped to CUs, and work-items are assigned to PEs. In order to launch a kernel, a programmer must define the number of work-groups in NDRange, and the number of work-items in a work-group. In real hardware, since the number of actual cores are limited, CUs and PEs are virtualized by the hardware scheduler or OpenCL drivers.

### 2.2 OpenCL Kernel Decomposition

OpenCL uses a relaxed memory consistency model for *global* memory within a kernel's workspace, an NDRange. Within a kernel, the execution order of work-groups does not affect the output until all work-groups reach a global synchronization point, which is a new kernel invocation. This memory consistency enables CUs to be virtualized by the hardware scheduler or OpenCL drivers because they execute the work-groups in an arbitrary order.

```
__kernel void sample(__global int *Output, ...) {
    int tid = get_global_id(0);
    [COMPUTE CODE]
    Output[tid] = compute_value;
}
```

(a) Original code

```
__kernel void sample(__global int *Output, ...
        int exe_range_from, int exe_range_to) {
    int tid = get_global_id(0);
    int gid = get_group_id(0);
    if ( gid < exe_range_from || gid > exe_range_to )
        return;
    [COMPUTE CODE]
    Output[tid] = compute_value;
}
```

(b) Code transformation for sub-kernel

```
__kernel void sample(__global int *Output, ...
        int merge_exe_from, int merge_exe_to,
        __global int *Out_for_merge) {
    int tid = get_global_id(0);
    int gid = get_group_id(0);
    if ( gid < merge_exe_from || gid > merge_exe_to )
        return;
    ~~[COMPUTE CODE]~~  // REMOVED
    Output[tid] = Out_for_merge[tid];
}
```

(c) Code transformation for merging output

Figure 2: Code transformation for sub-kernel execution and merging output

Motivated by this property, several previous works proposed code transformation techniques in order to decompose an OpenCL kernel into several *sub-kernels*, which execute a subset of work-groups in a predefined order [20, 29]. To launch a sub-kernel they linearize N-dimensional work-groups in the original kernel, and identify the work-group by the linearized index after the kernel invocation as shown in Figure 2(b). If the linearized index is not identified to execute, the work-group immediately finishes, and the hardware scheduler will schedule it out.

If sub-kernels are executed on different devices, the results on each device must be properly merged. For merging outputs from different devices, [29] proposed a way that checks differences between two outputs and merges the values if they are different. Another approach was proposed by [20], which transforms a kernel to generate the addresses of outputs for the executed work-groups, and selectively copies them based on generated addresses as shown in Figure 2(c). MKMD follows the second approach, because it is faster as it does not perform comparisons of memory values.

## 3. MKMD OVERVIEW

MKMD is a runtime library that is compatible with OpenCL APIs as illustrated in Figure 3. Since MKMD is transparent to OpenCL applications by providing the illusion of a single virtual device, programmers can build an algorithm without concern for mapping multiple kernels to several devices.

Instead, MKMD makes a scheduling decision by estimating the execution time of each kernel on the underlying devices for the given input size. In addition, each kernel can be decomposed into several sub-kernels for scheduling at work-group granularity. MKMD also predicts the execution time of sub-kernels with a partial number of work-groups. In order to estimate the execution time, MKMD operates in two different modes, *profiling mode*, and *execution mode*, as shown in Figure 3.

In *profiling mode*, MKMD collects offline profile data by executing the kernels with various input sizes and different numbers of work-groups. As profiling the execution time for all possible in-

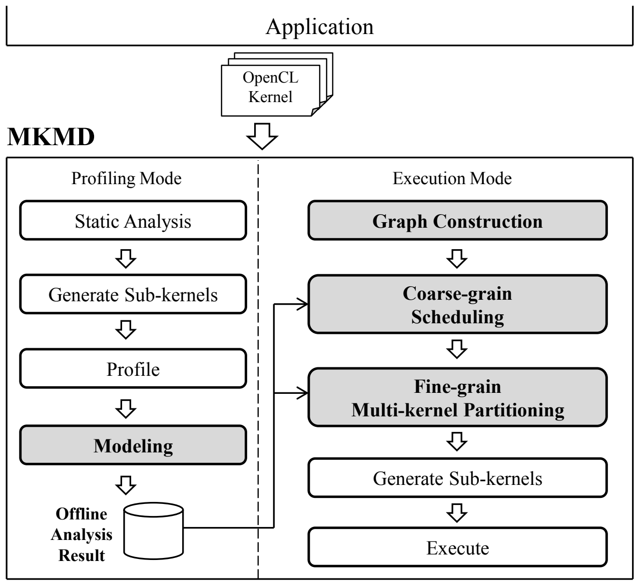


Figure 3: MKMD workflow that operates in profiling mode and execution mode. In profiling mode, MKMD builds a mathematical model with a set of profile data for the execution time prediction. In execution mode, MKMD predicts the execution time of kernels on various input sizes using the model, and schedules kernels based on the predicted time.

put sizes and numbers of work-groups is unrealistic, MKMD profiles kernels on each devices with a set of few representative inputs and work-groups, and performs a regression analysis to construct a mathematical model for a wide range of input and work-group sizes. In order to facilitate the regression analysis, MKMD statically analyzes kernels to approximate the computational complexity. Details of the modeling are discussed in Section 4.

Once the offline analysis is done, MKMD can be run in *execution mode*, which follows five steps as shown in Figure 3; 1) Kernel graph construction; 2) Coarse-grain scheduling; 3) Fine-grain multi-kernel partitioning; 4) Sub-kernel generation; and 5) Execution.

For the kernel graph construction, MKMD analyzes the parameters of each kernel, determines the data (buffer) flow between kernels, and then constructs the graph. In the next step, MKMD performs coarse-grain scheduling, which assigns kernels to the devices considering kernel dependencies, predicted execution time, and buffer transfer cost between the devices. After coarse-grain scheduling, MKMD runs fine-grain partitioning to improve the scheduling results, in which the scheduler could have left some devices idle for certain amount of time due to insufficient kernel-level parallelism. In order to utilize those idle devices, MKMD decomposes a kernel into a set of *sub-kernels* at work-group granularity, offloads them to available devices, and adjusts the scheduling results.

After the scheduling decision is made, MKMD executes kernels by generating the actual OpenCL commands for each device, which include both kernel executions and data transfers. The details of MKMD scheduling and partitioning are discussed in Section 5.

## 4. EXECUTION TIME MODELING

In order for MKMD to schedule kernels before it runs kernels, it must be aware of the execution time for each kernel on each device for the given input sizes. One way to estimate the kernel execution

```
__kernel void square_matmul(__global float *C,
    __global float *A, __global float *B, int N) {
  int i = get_global_id(0);
  int j = get_global_id(1);
  for (int k = 0; k < N; ++k)
    tmp += A[i * N + k] * B[k * N + j];
  C[i * N + j] = tmp;
}
```

(a) Matrix multiplication

```
__kernel void vadd(__global float *C,
    __global float *A, __global float *B, int N) {
  int i = get_global_id(0);
  int T = get_global_size(0);
  for (int k = i; k < N; k += T)
    C[k] = A[k] + B[k];
}
```

(b) Vector addition

Figure 4: Upper bounds of trip count. The upper bounds are statically determined as $N$ for (a), and $\frac{N}{T}$ for (b)

time is to refer to the offline profile data, which is gathered by varying the combination of the number of work-groups, input size, and device. However, it is often impractical to profile for every possible input size.

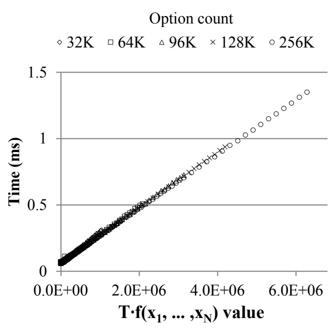
To avoid a large number of profiling, another approach is to build a model to predict the execution time for the given input size. To build such a model, the relation between computational cost and a given input must be analyzed first. Prior research has examined experimental algorithmics in order to analyze the asymptotic cost of programs using representative input sets [5, 22, 27, 34]. The intuition behind these works is that the asymptotic cost can be inferred from several executions with different input size by extrapolating the trend of the result. [22] showed that a large number of input sets may be required as there are some cases where cost functions are hard to be discovered. To improve the accuracy on these cases, [34] narrowed down the domains to specific data structures, and [5] applied regression analysis techniques.

Although previous studies showed that empirical analyses can provide accurate cost of a program, they mainly target legacy sequential applications on conventional processors. As traditional software has dynamic behaviors and faces difficulties in static analysis due to complex data structures, asymptotic analyses may require considerable amount of profile data.

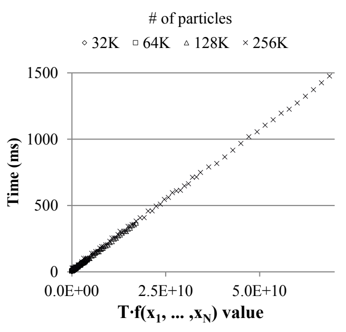
However, the asymptotic cost of OpenCL kernels can be statically analyzed in many cases due to restrictions of the programming model and their deterministic properties. First, OpenCL does not allow recursive calls, so expensive inter-procedural analysis can be avoided by inlining function calls. Second, it prohibits system calls and double pointers (pointers of pointers), which make kernels more deterministic and easy to analyze. Third, the number of work-items and the input/output size of the kernel is predefined before kernel launch, thus the upper bound of the loop can be statically determined for many cases [11, 12].

Based on these observations, this work investigates the potential of static analysis on OpenCL kernels, proposes an efficient methodology that requires a few input sets for modeling the execution time, and evaluates the accuracy of proposed approach. Note that if the cost function cannot be analyzed statically, it can also be modeled using more profile data [5].

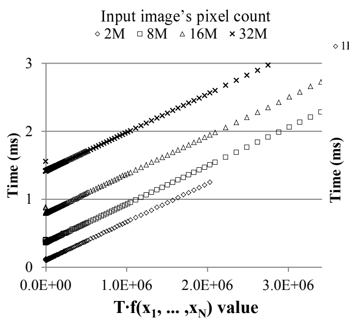
To illustrate the static cost analysis, Figure 4 shows two simple code examples. In Figure 4(a), the upper bound of the loop trip count is $N$, which is passed by the host program. Because the kernel code is executed by the number of work-items defined by the host program, the asymptotic cost of the kernel is $O(TN)$, where $T$ is the number of work-items.



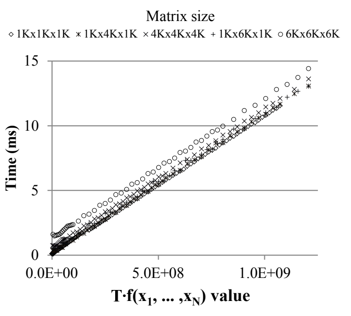
(a) Blackscholes



(b) N-body



(c) Sobel filter



(d) Matrix multiplication

Figure 5: Scalability of execution time on NVIDIA GTX760 varying input sizes and the number of enabled work-items ($T$). The execution time is linear to the value of cost function $Tf(x_1, ..., x_N)$.

In contrast, some kernels are launched with a fixed number of work-items, but each work-item iterates until all input elements are properly handled as shown in Figure 4(b). In this case, the upper bound of the loop trip count is analyzed as $\frac{N}{T}$. Because the code will also be executed by $T$ work-items, the asymptotic cost of this kernel becomes as $O(N)$.

Since the asymptotic cost can be regarded as the dynamic instruction count in the worst case, the estimated dynamic instruction count can be defined as,

$$c \times Tf(x_1, ..., x_N) \tag{1}$$

where $c$ is an estimated coefficient, $T$ is the number of work-items, and $x_i$ is a variable that can affect the trip count of a loop in the kernel. Consequently, the estimated execution time on a device, $Time_{est}$ can be defined as

$$Time_{est} = \frac{CPI}{Freq_{clock}} \times c \times Tf(x_1, ..., x_N) \tag{2}$$

In Equation 2, $CPI$ (cycles per instruction) and $Freq_{clock}$ differ by device properties (e.g. the number of cores and memory hierarchies), while the value of $Tf(x_1, ..., x_N)$ can be determined statically at compile time. Therefore, the new coefficient, $\frac{CPI}{Freq_{clock}} \times c$, is estimated using a regression analysis.

To explain, Figure 5 illustrates the execution time of several benchmarks from NVIDIA SDK [26], varying input sizes and the number of work-items. Each legend represents the execution with different input parameters. The X-axis in the figure is the value of the cost function, $Tf(x_1, ..., x_N)$, varying the number of work-items, $T$, with the technique discussed in Section 2.

As shown in the figure, for the same input size, the execution time is linear to the value of the cost function. Also, the slopes, $\frac{CPI}{Freq_{clock}} \times c$, are equivalent regardless of the input size. Although Figure 5 (a) and (b) show the same initial cost over different input sizes, (c) and (d) show different initial cost despite the same value of the cost function. The reason is that the number of work-items ($T$) is controlled by the software methodology as discussed in Section 2, which activates the entire number of work-items first, and selectively disables work-items by exiting work-items immediately. In other words, the fixed cost increases as the total number of work-items grows.

| **Kernel** | $f(x_1, ..., x_N)$ | **Avg. Error (%)** | |
|---|---|---|---|
| | | **20 profiles** | **40 profiles** |
| Blackscholes | $8^{th}Arg$ | 2.12 | 1.27 |
| N-body | $\frac{8^{th}Arg}{LocalSize(0)}$ | 1.72 | 1.23 |
| MatrixMul | $6^{th}Arg$ | 1.4 | 0.9 |
| FDTD3d | $6^{th}Arg$ | 1.98 | 1.45 |
| SobelFilter | 1 | 1.18 | 0.97 |
| MedianFilter | 1 | 1.06 | 0.98 |
| K-Means | $4^{th}Arg * 5^{th}Arg$ | 1.42 | 1.18 |

Table 1: Execution time estimation on NVIDIA GTX 760. The cost functions, $f(x_1, ..., x_N)$, were statically analyzed. For example, $8^{th}Arg$ means that the value of the $8^{th}$ argument is the trip count of a loop in the kernel. $LocalSize(0)$ means the work-item count per work-group in the first dimension, while the constant 1 means that a loop was not found in the kernel.

Because Blackscholes in NVIDIA SDK uses the fixed number of work-items similar to the example shown in Figure 4(b), and N-body runs with a relatively small number of work-items, the initial cost is similar regardless of the input size. On the other hand, the other benchmarks, (c) and (d), have different initial cost because they increase the number of work-items as the input size grows. Note that the initial cost is linear to the number of work-item, but it still can be different across devices. Therefore, initial cost must also be considered during regression analysis, and the final equation for the execution time can be expressed as,

$$y = \beta_1 Tf(x_1, ..., x_N) + \beta_2 T + \epsilon \tag{3}$$

With Equation 3, the tuple, $<y, Tf(x_1, ..., x_N), T>$, is recorded for each profile-run, and then $\beta_1$, $\beta_2$, and $\epsilon$ are modeled through the regression analysis. Once the values of $\beta_1$, $\beta_2$, and $\epsilon$ are modeled, the execution time $\hat{y}$ can be estimated in runtime by putting the real value of the tuple.

In order to evaluate the accuracy of the execution time model, OpenCL applications from NVIDIA SDK and Rodinia [3] were used. Table 1 shows the estimation result for a subset of applications from two benchmark suites on NVIDIA GTX 760. The execution time models were constructed with 20 and 40 sets varying input sizes and work-group sizes. The sizes of input for the profiling were more than 100 MB. The second column of Table 1 describes the cost function, $f(x_1, ..., x_N)$, which is analyzed at compile time. To compute the average error rate, 100 executions were performed with random inputs and work-group sizes, and the estimated time was compared with the observed time.

As shown in Table 1, the average performance prediction error with random input remained under 3% with 20 profiling sets, and under 2% with 40 profiling sets.

## 5. MKMD SCHEDULING

With a regression model constructed through profiling, MKMD schedules multiple kernels to execute them in the least time. This section discusses how to construct the kernel graph, and how to schedule the kernels in coarse granularity and partition them in finer granularity.

## 5.1 Kernel Graph Construction

In order to launch multiple OpenCL kernels, the application enqueues kernels in a specific order defined by programmer. After enqueuing multiple kernels, the application issues the queue using one of OpenCL APIs, such as *clFlush* or *clFinish*. Upon this issue request, MKMD analyzes the dependencies between kernels to ensure that outputs are available for consuming kernels. Since kernels in the queue are supposed to be executed once, MKMD constructs a directed acyclic graph (DAG), which is called the *kernel graph*, where nodes ($V_i$) and edges ($E_{i,j}$) correspond to the kernels and buffers, respectively.

Each node has the average execution time of the kernel for all devices as a **node weight**. Likewise, each edge contains the buffer transfer time as the **edge weight**, which can be computed by the buffer size divided by the interconnect bandwidth.

For the initial and final buffer transfers between the host program and devices, MKMD also adds a ***source*** node and a ***sink*** node to the graph. The source node has only out-edges that correspond to the initial buffers from the host program, whereas the sink node has only in-edges that correspond to the buffer being transferred to the host. During scheduling, these two nodes are forced to be scheduled in the CPU device, which shares the address space with the host program. Note that the node weights of both source and sink nodes are zero because they do not have actual computation.

## 5.2 Coarse-grain Scheduling

Once the kernel graph is constructed, MKMD schedules a task in kernel granularity using a list scheduling algorithm. The basic idea of list scheduling is to compute priorities of tasks, and make a list of tasks ordered by the priorities. With the list, the scheduler repeatedly selects the task with the highest priority, and assign it to a resource that can accommodate the task.

Many prior researches have utilized list scheduling [23] for certain cases [2, 32, 33]. The way that MKMD schedules in kernel granularity is similar to the HEFT algorithm [33] as MKMD targets heterogeneous OpenCL devices, but uses different metrics due to the interconnect.

For listing the kernels, MKMD traverses down the graph from the source node computing the priority of the node, $P(V_i)$, defined as

$$P(V_i) = \begin{cases} W(V_i) + \max\limits_{V_j \in Succ(V_i)} (W(E_{i,j}) + P(V_j)), & V_i \neq V_{sink} \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

where $W(V_i)$ is a node weight, and $W(E_{i,j})$ is an edge weight from a node to the immediate successors. Because $P(V_i)$ is accumulated with the max value of successors $P(V_j)$ as shown in Equation 4, the list ordered by the priority is topologically ordered, which means that it is guaranteed that all predecessor kernels are scheduled before scheduling a kernel. After the prioritization, MKMD selects the kernel with the highest priority in the list, and schedules it on a device.

The first step for the selected kernel is to find the earliest slot for each device. Note that a kernel cannot be scheduled before predecessors finish, and must wait for the data from predecessors to be transferred if they are scheduled in different devices. Therefore, the **earliest start-able time** of kernel $i$ on device $k$, $EST(V_i, D_k)$, can be defined as

$$EST(V_i, D_k) = \max_{V_j \in Pred(V_i)} \{KT_{end}(V_j) + T_{trans}(V_j, E_{j,i}, k)\} \tag{5}$$

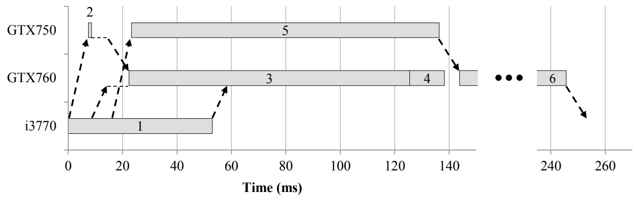


Figure 6: Coarse-grain scheduling result on three heterogeneous devices. Dotted arrows presents the buffer transfer between devices. PCI bus operates in full-duplex, but GTX760 and i3770 experience input and output congestion respectively.

where $KT_{end}(V_j)$ is the scheduled finish time of the predecessor kernel $V_j$, where $T_{trans}(V_j, E_{j,i}, k)$ is the buffer transfer time from the scheduled device of predecessor $V_j$ to device $k$.

Note that if predecessors are scheduled in different devices, buffers cannot be transferred to device $k$ at the same time, but transferred in serial. Thus, $T_{trans}(V_j, E_{j,i}, k)$ is defined as

$$T_{trans}(V_j, E_{j,i}, k) = \begin{cases} \dfrac{W(E_{j,i}) \times BW_{max}}{AvailBW(KD(V_j), k)}, & KD(V_j) \neq k \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

where $W(E_{j,i})$ is the estimated transfer time at full bandwidth, $KD(V_j)$ is the scheduled device of the predecessor $V_j$, and $AvailBW()$ returns the available bandwidth between two devices being aware of the buffer transfer schedule.

Once $EST(V_i, D_k)$ is computed for each device, the next step is to find a device that can finish the kernel in the earliest time. Because $EST(V_i, D_k)$ does not consider if the device $k$ has available time slots in which the execution time of kernel $i$ fits, the **earliest finish-able time** of kernel $i$ on device $k$, $EFT(V_i, D_k)$, can be defined as

$$EFT(V_i, D_k) = AvailEST(V_i, D_k) + T_{exe}(V_i, D_k) \tag{7}$$

where $T_{exe}(V_i, D_k)$ is the estimated execution time of kernel $i$ on device $k$, and $AvailEST()$ returns the available earliest start-able time of device $k$ after $EST(V_i, D_k)$ where $T_{exe}(V_i, D_k)$ fits into.

With $EFT$, the final **schedule device**, **schedule start time**, and **schedule end time** of the kernel are defined as:

$$KD(V_i) = \operatorname*{argmin}_{k \in Devs} \{EFT(V_i, D_k)\} \tag{8}$$

$$KT_{start}(V_i) = AvailEST(V_i, KD(V_i)) \tag{9}$$

$$KT_{end}(V_i) = EFT(V_i, KD(V_i)) \tag{10}$$

Figure 6 shows the scheduling result for the same application shown in Figure 1 on a system with three different devices, Intel i3770, NVIDIA GTX 760, and GTX 750. As shown in Figure 6, the coarse-grain scheduling considers kernel dependencies and the interconnect between devices, but still leaves some devices idle for considerable amounts of time. For example, i3770 is idle from 53 ms, and GTX750 is idle from 138 ms. In order to remove the idle periods from coarse-grain scheduling, MKMD performs fine-grain multi-kernel partitioning on the results, which is discussed in Section 5.3.

## 5.3 Fine-grain Multi-kernel Partitioning

The basic idea of partitioning is to split the kernel into finer granularities, *work-groups*, and then offload some work-groups to the idle devices so that the original device can finish the kernel earlier. As discussed in Section 2.2, an OpenCL kernel can be selectively executed at work-group granularity. Through the transformed kernel, MKMD can decompose a kernel into several *sub-*

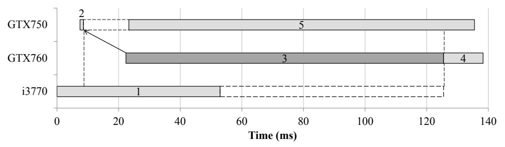


Figure 7: Available compute-time slots (dotted-squares) for partitioning kernel 3. Because kernel 3 depends on kernel 2 (arrow), the lower bound and upper bound of available time slots are the finish time of kernel 2 and 3 respectively.

*kernels*, and distribute them across multiple devices as balanced as possible based on the coarse-grain scheduling result. To achieve this, MKMD follows several steps, prioritization, device availability identification, partitioning, and adjusting successors' schedule.

**Prioritization**: For fine-grain multi-kernel partitioning, MKMD must consider the effects of partitioning on the overall scheduling result. To illustrate, in Figure 6, finishing kernel 6 in the earliest time is the objective, but the kernel depends on the results from kernels 4 and 5. Again, kernel 4 is dependent on kernel 3, which is also dependent on kernel 2. Because the earliest scheduled kernel has a higher chance to have larger impact on later kernels as they are scheduled with the consideration of kernel dependencies, MKMD prioritizes the kernels by the order of schedule start time from the coarse-grain scheduling result.

Starting from the kernel with the highest priority, MKMD partitions a kernel by offloading work-groups from the scheduled device to other devices. In Figure 6, MKMD starts from kernel 1, but kernels 1 and 2 cannot be partitioned because of interconnect bandwidth saturation. Thus, kernel 3 becomes the first kernel that will be actually partitioned.

**Device availability identification**: When offloading work-groups to other devices, MKMD must identify the temporal availability of the devices, so MKMD first identifies the *available time slots* for each device. Then, a time slot becomes the basic unit to which work-groups are offloaded. Note that one device can have multiple time slots as it can be idle intermittently.

Available time slots for each device can be easily identified from the scheduling result, but it is important to keep the consistency that the offloaded work-groups cannot be executed before predecessor kernels finish. For this reason, the time slots have lower and upper limits where the lower limit is the latest finish time of predecessor kernels, and the upper limit is the finish time of the kernel to be partitioned. Figure 7 visualizes available time slots for partitioning kernel 3 from the coarse-grain scheduling example in Figure 6. Because kernel 3 depends on kernel 2, the lower bound is the finish time of kernel 2.

**Partitioning**: With available time slots for each device, MKMD partitions a kernel to minimize the schedule length by offloading work-groups to available slots. The problem of minimizing the schedule length is a bin-packing problem when time slots and work-groups are mapped to bins and objects respectively. Bin-packing is a NP-hard problem, but the partitioning must be done quickly because the entire process of MKMD is done in runtime. Therefore, MKMD uses a *hill-climbing greedy heuristic*, which is further discussed in Section 5.4.

**Schedule adjustment**: As a result of partitioning, the schedule length of a kernel can be reduced, and successor kernels can start execution earlier. Therefore, MKMD adjusts the schedules of successor kernels after partitioning. In order to minimize the overhead, MKMD does not change the scheduled device of successor kernels, but only adjusts successors' start time.

Overall, Algorithm 1 shows a high-level description of multi-kernel partitioning. In the algorithm, the first line prioritizes kernels

**Algorithm 1** Multi-kernel partitioning

```
 1: V[1..N] ← kernels ordered by the start time
 2: for i = 1 to N do
 3:     $V_i$ ← V[i]
 4:     Reschedule $V_i$ to EST($V_i$, KD($V_i$))
 5:     LB ← $\max_{V_j \in Pred(V_i)} \{\mathrm{KT}_{end}(V_j)\}$
 6:     UB ← $\mathrm{KT}_{end}(V_i)$
 7:     for $k$ = 1 to $\mathrm{NUM}_{devs}$ do
 8:         $\mathrm{List}_{slot}$[k] ← Available time slots between LB and UB
 9:     end for
10:     Partition $V_i$ to $\mathrm{List}_{slot}[1..\mathrm{NUM}_{devs}]$ by work-groups
11:     if Partitioned then
12:         Create new nodes $\mathrm{SET}_{p \in Partitions} \{V_{i,p}\}$
13:         Update $DAG$ with new nodes
14:         Update $Schedule$ with the partition result
15:     end if
16: end for
```

by the schedule start time, and lines 5-9 compute available time slots for a kernel. After that, a kernel is partitioned into the time slots as shown at line 10, and the kernel graph and the schedule are updated in lines 12-14. For line 10, Section 5.4 explains how to partition a kernel into time slots in detail. As a result of partitioning, the execution time of the kernel will be reduced. This means that the following kernels that were dependent on the partitioned kernel now can be scheduled earlier. For this reason, before partitioning, the algorithm reschedules the kernel to the earliest start-able time (EST) on the same device as shown at line 4 in Algorithm 1.

## 5.4 Partitioning a Kernel to Time Slots

As discussed in Section 5.3, partitioning a kernel across multiple time slots can be reduced to a bin-packing problem as the objective is to minimize the finish time by packing work-groups into time slots. In addition, there are two more challenges to be considered.

The first challenge is that the usage of interconnect bandwidth must be considered when work-groups are offloaded. For example, even if a device has a large available time slot for a specific kernel, offloading work-groups may not be possible if interconnect bandwidth is saturated during the period because the input cannot be transferred.

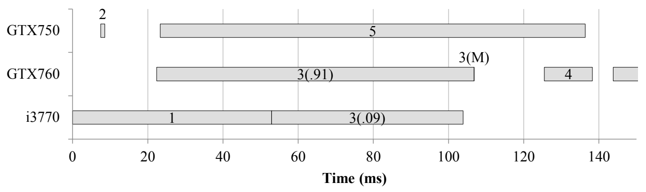
Another challenge is that the cost of merging output may occur if sub-kernels are executed on different physical devices. Because different physical devices use different address spaces, sub-kernels will generate *partial* results in their own address space. Using the methodology discussed in Section 2.2, several partial results can be merged efficiently by executing the *merge-kernel*. Because the merge-kernel is executed for merging two partial results, the cost of merging grows as the number of devices that execute sub-kernels increases.

**Partitioning Heuristic:** To tackle these challenges, the optimal partitioning solution can be found through an exhaustive search, but the overhead will be significant. Since MKMD partitioning is performed at runtime before the execution, MKMD uses a *hill climbing heuristic* to minimize the overhead. The inputs of the partitioning algorithm are the scheduled time slot from coarse-grain scheduling and available time slots for each device.

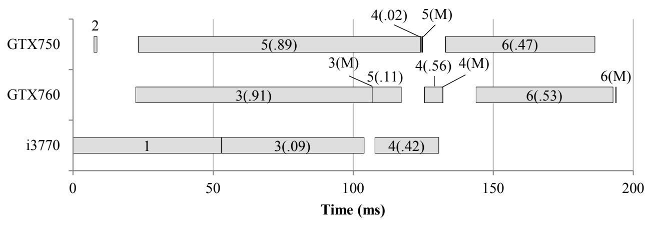
The hill climbing algorithm starts from a coarse-grain scheduling solution, which is the state where all the work-groups are assigned to a scheduled slot. Next, it duplicates the current state to several candidate states. The number of candidates is as many as the number of available time slots. Once candidate states are created,

**Algorithm 2** Kernel partitioning to available time slots

```
 1: Slots[1..S-1] ← all available time slots in List$_{slot}$[1..Num$_{devs}$]
 2: Slots[S] ← Scheduled time slot for kernel v
 3: currState.Slots[1..S] ← Slots[1..S]
 4: W$_{off}$ ← number of work-groups to offload at a time
 5: Time$_{curr}$ ← $\max_{s \in S}$ {currState.Slots[s].FinishTime}
 6: Time$_{prev}$ ← Time$_{curr}$ + 1
 7: while Time$_{curr}$ < Time$_{prev}$ do
 8:     Time$_{prev}$ ← Time$_{curr}$
 9:     nextState[1..S] ← currState
10:     s$_{from}$ ← $\arg\max_{s \in S}$ {currState.Slots[s].FinishTime}
11:     for s = 1 to S do
12:         nextState[s].tryOffload(s$_{from}$, s, W$_{off}$)
13:     end for
14:     s$_{pick}$ ← $\arg\max_{s \in S}$ {nextState,Slots[s].FinishTime}
15:     currState ← nextState[s$_{pick}$]
16:     Time$_{curr}$ ← $\max_{s \in S}$ {currState.Slots[s].FinishTime}
17: end while
18: return currState.Slots
```



(a) Partitioning result for kernel 3



(b) Final scheduling result

Figure 8: Kernel partitioning process. The decimal numbers in a parenthesis shows the ratio of work-groups. The mark (M) is the cost for mering nonlinear outputs.

each candidate attempts to offload a fixed number of work-groups to their available time slot.

While candidates offload the work-groups, they estimate the execution time considering available interconnect bandwidth, amount of buffer transfer, and additional cost for merging outputs. During the estimation, MKMD also checks if the execution time for offloaded work-groups fits in the time slot, or if the finish time of the slot is later than the upper bound. In this case, the candidate is disqualified.

Among qualified candidates, the algorithm picks the candidate who finishes the kernel in the earliest time. The current state is updated with the picked candidate state, and the algorithm repeats the process of finding candidates until no candidate state is found.

Algorithm 2 describes the procedure of partitioning. In the algorithm, line 9 corresponds to duplicating the current state to several candidate states, and in lines 11-13, each candidate state tries offloading a fixed number of work-groups to a different time slot. While the algorithm tries to offload in line 12, it considers current status of the interconnect, and the merge cost in case of kernel decomposition. After the offloading trials, in line 14, the algorithm picks the candidate state which finishes the earliest time.

Note that the trip count of the while loop in lines 7-17 can be controlled by defining $W_{off}$ as the total number of work-groups divided by the trip count. In MKMD, the trip count of the while loop is limited to 100 to reduce the time complexity. In other words, each iteration tries to offload 1% of work-groups, and reaching 100 iterations means that all work-groups are offloaded to other time slots. Therefore, in most cases, the while loop stops iterating before it reaches the limit of 100. As the while loop is reduced to a constant, the final time complexity of the partitioning algorithm is $O(S)$, where $S$ is the number of time slots.

As a result of the algorithm, kernel 3 in Figure 7 is partitioned as shown in Figure 8(a). After partitioning the rest of kernels, the final scheduling result is illustrated in Figure 8(b).

| Device | **Intel Core i7 3770** | **NVIDIA GTX 760** | **NVIDIA GTX 750 Ti** |
|---|---|---|---|
| **# of Cores** | 4 (8 Threads) | 1152 | 640 |
| **Clock Freq.** | 3.2 GHz | 0.98 GHz | 1.02 GHz |
| **Memory (B/W)** | 32 GB DDR3 (12.8 GB/s) | 2 GB GDDR5 (192 GB/s) | 2 GB GDDR5 (88 GB/s) |
| **Peak Perf.** | 435.2 GFlops [13] | 2,258 GFlops | 1,306 GFlops |
| **OpenCL Ver.** | Intel SDK 2013 | CUDA SDK 5.5 | |
| **PCIe (B/W)** | - | 3.0 x8 x2 (7.88 GB/s) | |
| **OS** | Ubuntu Linux 12.04 LTS | | |

Table 2: Experimental Setup

### 5.5 Overhead and Limitations

The coarse-grain scheduling costs $O(V^2K)$, where $V$ is the number of kernels and $K$ is the number of device. After the coarse-grain scheduling, partitioning is performed for each kernel at the cost of $O(VS)$, where $S$ is the number of time slots. Because the number of time slots can not exceed $V \times K$, the partitioning algorithm is in proportion to to $V^2K$ as well. Therefore, the entire cost of MKMD scheduling algorithm is $O(V^2K)$, which is evaluated in Section 6.

Because MKMD makes a scheduling decision of multiple kernels based on the execution time model before it runs a kernel, it has two main limitations.

First, the scheduling decisions can be suboptimal for irregular applications, because they are hard to model the execution time. For example, if the trip count of a loop is varied by work-groups and it is dependent on the value of input array, it is difficult to build a model to predict the entire execution time.

Second, the scheduling decision is made assuming that all underlying devices are exclusive to the application until it finishes the execution. However, if other applications occupy the hardware resources in the middle of the execution, the scheduling result is not optimal anymore because available resources are changed.

## 6. EVALUATION

**Implementation**: MKMD was prototyped as a library, and it overloads OpenCL API calls from the application through dynamic linker redirection. Inside MKMD, it uses the Clang [4] for the OpenCL front-end, and the Low-Level Virtual Machine (LLVM) 3.6 [19] for the back-end. For execution of partial work-groups, LLVM transforms the kernel to a sub-kernel by adding a checking code to the beginning. Taking the range of linearized work-group indices as parameters, the checking code filters out the work-groups that are not in the range. Once the kernels are built, MKMD can operate in profiling mode for building a regression model. For each parameter, MKMD executes kernels multiple times with different numbers of work-groups.

| Name | Equation | Domain |
|---|---|---|
| **Algebraic Bernoulli (ABE)** | $A^TX + XA - XBB^TX$ | System Theory |
| **Biconjugate gradient stabilized (BiCGSTAB)** | iterative method with 11 operations | Linear Systems |
| **Triple commutator** | $ABC + BCA + CAB$ $-BAC - ACB - CBA$ | Mathematics |
| **Generalized Algebraic Bernoulli (GABE)** | $A^TXE + E^TXA$ $-E^TXGXE$ | System Theory |
| **Reachability Gramian** | $AP + PA^T + BB^T$ | Control Theory |
| **Jacobi** | $D^{-1}(L + U)x + D^{-1}b$ | Linear Systems |
| **Continuous Lyapunov** | $AX + XA^T + Q$ | Control Theory |
| **Continuous Algebraic Riccati (CARE)** | $A^TX + XA$ $-XBR^{-1}B^TX + Q$ | Control Theory |
| **Stein** | $AXA^T - X$ | Probability |
| **Singular value decomposition (SVD)** | $U\Sigma V^T$ | Signal Processing |
| **Sylvester** | $AX + XB - C$ | Mathematics |

Table 3: Benchmark Specification

In execution mode, MKMD takes the list of OpenCL commands from the application, and performs scheduling as discussed in Section 5.

**Baseline**: For the experiments, we configured a real machine as shown in Table 2. The baseline of our experiment is in-order OpenCL execution on a single device assuming that the programmer picks the fastest device, GTX 760 in our experimental setup, and simply enqueues the OpenCL commands to that device. We also compared MKMD with the coarse-grain-only (*Coarse-Only*) algorithm, excluding the fine-grain multi-kernel partitioning. Scheduling assumes that initial status is where the host has initial inputs and the final status is that the final output is gathered to the host. Based on these statuses, kernels will be scheduled.

**Benchmarks**: In order to evaluate MKMD for more complex kernel graphs, we used linear algebra equations found in various scientific domains as our benchmarks. For each linear algebra kernel, we used the OpenCL implementation from NVIDIA SDK [26]. The equations and their domains are listed in Table 3. The sizes of vectors and matrices used in the equations are $4K$ and $4K \times 4K$, respectively.

## 6.1 Results

First, we measured the speedup of MKMD over in-order execution. As shown in Figure 9(a), MKMD performs better than in-order executions on every benchmark. The difference between Coarse-Only and MKMD is the performance gain from fine-grain kernel partitioning as discussed in Section 5.3. In geometric mean, MKMD brings 89% performance improvement over in-order single device execution. Among 89% performance improvement, approximately half comes from the coarse-grain scheduling by assigning kernels out of order across multiple devices, and the other half comes from the fine-grain multi-kernel partitioning by splitting the kernels into several sub-kernels and assigning them to the idle devices.

For BiCGSTAB, both coarse-only and MKMD scheduling do not show much speedup as shown in Figure 9(a). The reason is that it is composed of many matrix-vector multiplications, which are fairly memory-intensive and run much faster on GPUs. As a result, Coarse-Only scheduling assigns most of kernels to a single GPU in order to execute quickly and to avoid multiple data transfers. Even multi-kernel partitioning cannot help reducing the execution time, as the kernel execution time is relatively small compared to buffer transfer time. Thus, MKMD shows the same speedup as Coarse-Only scheduling.

The reason why the Coarse-Only also shows less speedups on Stein and SVD is that there are not many kernels that can be run in parallel. Therefore, the Coarse-Only execution is similar to in-order execution. On the other hand, MKMD achieves 1.9x speedup for both benchmarks taking advantage of fine-grain kernel partitioning.

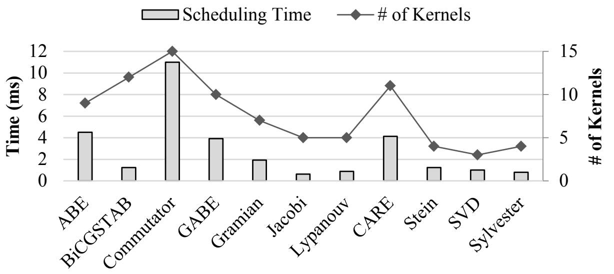


Figure 10: MKMD scheduling overhead.

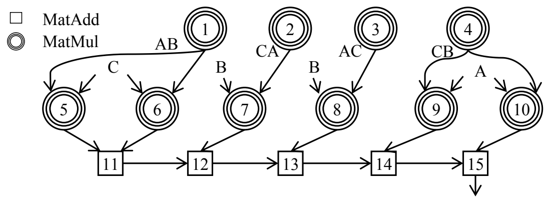


Figure 11: Kernel graph for triple commutator.

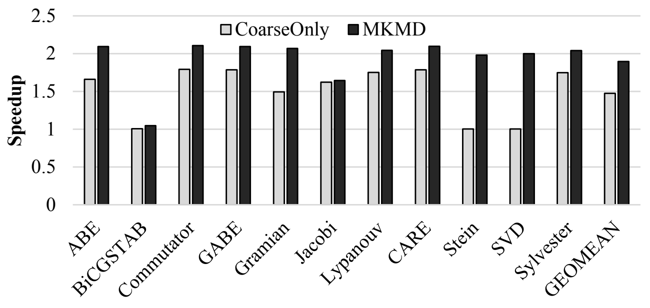
Figure 9(b) shows the average idle time of devices normalized to the entire execution time, or the ratio of device underutilization. As shown in the figure, in geometric mean, MKMD utilizes the devices 94% of the time, while in-order execution makes use of the devices 30% of the time.

For some benchmarks, such as ABE and commutator, device underutilization of MKMD is low. This shows that MKMD utilizes all available resources to improve performance. The detailed behavior of the devices on commutator is discussed Section 6.2.

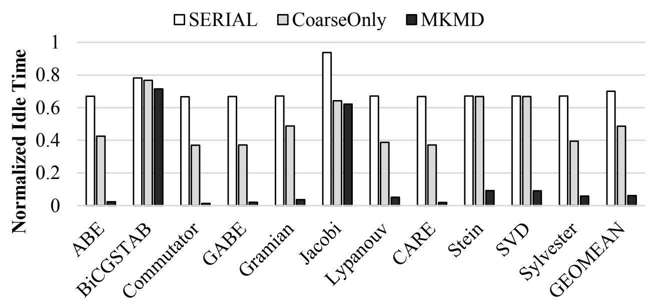
Figure 10 illustrates scheduling overhead for each benchmark. As shown in the figure, the absolute time of scheduling overhead is less than 10 msec for all benchmarks. In terms of the overhead ratio normalized to the entire execution time, BiCGSTAB and Jacobi have 6.4% and 4.2%, respectively, and the other benchmarks have less than 0.1%. The main reason why BiCGSTAB and Jacobi have relatively large overhead is that they finish in a very short time (less than 70 msec). As discussed in Section 5.3, the scheduling overhead is not relative to the input size or kernel execution time, but relative to the number of kernels and devices, and Figure 10 shows such pattern. Nonetheless, the overhead of those two benchmarks does not overwhelm the performance improvement from MKMD, as BiCGSTAB and Jacobi get speedups of 1.12x.

## 6.2 Case Study

This section further investigates the behavior of MKMD on triple commutator because it is composed of many compute-intensive kernels. The kernel graph of triple commutator is built as shown in Figure 11, and the execution timeline is depicted in Figure 12. While MKMD performs coarse-grain scheduling, the kernel with the highest priority is kernel 1, the next is 2, and so on according to the Equation 4. Therefore, kernel 1 will be scheduled on the device which can finish it at earliest, which is GTX 760. Next, for kernel 2, the scheduler will assign it to GTX 750 as shown in Figure 12(a), because there is no dependency between kernels 1 and 2. While the scheduler assigns several matrix multiplication kernels to the GPUs, it does not assign a single kernel to the CPU (i3770), which leaves it idle as shown in Figure 12(a). The reason is that assigning the entire kernel to the CPU will increase the schedule

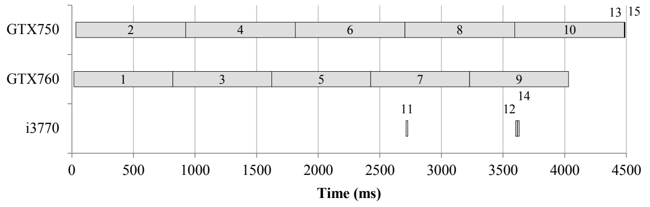


(a) Speedup over in-order OpenCL executions

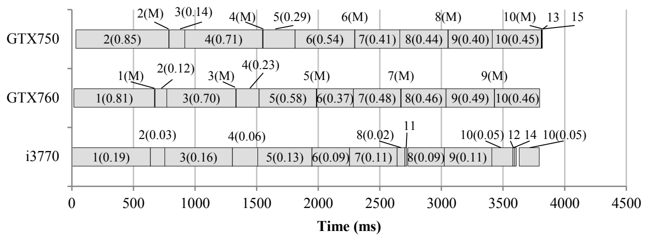


(b) Average device idle time normalized to the entire execution time

Figure 9: (a) Speedup of MKMD over in-order executions, and (b) the average device idle time normalized to the finish time.



(a) Coarse-grain schedule only



(b) MKMD

Figure 12: Execution timeline for triple commutator. Because matrix computation is too expensive on i3770, (a) the coarse-grain scheduler does not schedule any matrix multiplication kernel on it while GPUs take more than 4 kernels. With MKMD, (b) all devices are almost fully utilized.

length more than assigning it to GPUs even if several kernels are already assigned to them.

Based on coarse-grain scheduling results in Figure 12(a), MKMD starts multi-kernel partitioning as discussed in Section 5.3. With prioritization by the start time, kernel 1 will be partitioned first, and kernels scheduled later will be adjusted after partitioning. For this reason, kernel 1 utilizes the CPU more (15% of work-groups) than kernel 2 does (6% of work-groups) as shown in Figure 12(b). In the end, MKMD almost fully utilizes all three devices as shown in Figure 12 by splitting kernels into sub-kernels, executing them out of order without breaking the consistency.

## 7. RELATED WORK

As the systems become more heterogeneous, programming several data parallel kernels for heterogeneous devices has become extremely difficult.

Research has been done for task scheduling on heterogeneous processors or distributed systems using various programming languages [9, 28]. Using StreamIt [9], [8] proposed a compiler framework that refines stream graph of StreamIt program to a multi-core CPUs. Kudlur et al. also proposed a way to map StreamIt languages to distributed shared memory systems [18]. However, the usage of StreamIt language is strictly limited to certain cases, and the programmer must explicitly define the communication graph even for data parallel tasks. [28] proposed a set of compiler directives at a higher level, which hides hardware details from programmers. Despite these efforts, programmers still must know the underlying devices to explicitly schedule data parallel code and manage the buffer transfer between devices.

Rather than programming languages, many prior works proposed ways to alleviate the efforts in programming data parallel kernels on multiple heterogeneous devices [16, 17, 20, 21, 29]. [21] proposed Qilin system that automatically partitions threads to one CPU and one GPU by providing new APIs that abstract away two different programming models, Intel Thread Building Blocks and CUDA. However, they do not consider multiple kernels, and the number of devices is limited to two. [17] proposed a similar runtime system that distributes OpenCL workloads over multiple heterogeneous devices with the performance prediction based on an artificial neural network. However, they limited the type of OpenCL kernels to have regular memory access pattern. [7, 14, 16, 20, 29] proposed runtime systems that can distribute any type of kernels to several devices. Nonetheless, all these works only focus on optimizing a single OpenCL kernel for multiple devices, not considering the interaction between multiple kernels.

Research for virtualizing GPU resources has been done [30, 31]. PTask [30] proposes APIs that work with operating systems to manage tasks on GPUs by using a data-ow programming model. Dandelion [31] also proposes a compiler/runtime framework that works on C# sources with newer APIs. In this work, a compiler converts C# to CUDA, and the runtime framework manages execution between CPUs and GPUs using PTask [30]. While these works target C# code and require program modification to use additional APIs, MKMD transparently works on multiple OpenCL kernels without program modification.

For scheduling multiple data parallel kernels on heterogeneous devices, [6] proposed the Harmony system, which schedules data parallel kernels considering the performance of device. [1] proposed StarPU system, which also schedules multiple data parallel kernels on heterogeneous devices. [10] dynamically assigns kernel to devices of a heterogeneous system based on historical runtime data. However, all of these works schedule kernels at a kernel granularity, which can cause devices to idle for a considerable amount of time as evaluated in Section 6. [24] proposed Hyper-Q that supports multiple kernels on heterogeneous architectures, but it only considers multiple kernels on a single device, and requires programmers to identify the order of kernel execution.

## 8. CONCLUSION

As applications become more complex, programs commonly execute multiple data parallel kernels. In the meantime, the complexity of underlying hardware continues to increase with a wider variety of computation accelerators. In order to maximally utilize the underlying resources for applications with multiple data parallel kernels, this paper presented MKMD, a runtime framework that automatically builds a dependence graph from the OpenCL command queue, and schedules kernels out of order considering the costs of data transfer and execution time on each device. Execution time estimates are adaptive to input size using a regression model that is driven by a small number of profiling runs. MKMD combines coarse-grain kernel scheduling with fine-grain kernel partitioning to densely make use of all available time slots among devices. For a system with three different computing devices, MKMD achieves a mean 1.89x speedup over in order execution on the fastest device for a set of multi-kernel benchmarks.

## References

[1] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice & Experience*, 23(2):187–198, Feb. 2011.

[2] D. Bernstein and M. Rodeh. Global Instruction Scheduling for Superscalar Machines. In *Proc. of the '91 Conference on Programming Language Design and Implementation*, pages 241–255, 1991.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, , J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. of the IEEE Symposium on Workload Characterization*, pages 44–54, 2009.

[4] Clang. A C language family frontend for LLVM, 2014. http://clang.llvm.org.

[5] E. Coppa, C. Demetrescu, and I. Finocchi. Input-sensitive profiling. In *Proc. of the '12 Conference on Programming Language Design and Implementation*, pages 89–98, 2012.

[6] G. F. Diamos and S. Yalamanchili. Harmony: an execution model and runtime for heterogeneous many core systems. In *Proc. of the 17th international symposium on High performance distributed computing*, pages 197–200, 2008.

[7] T. Diop, S. Gurfinkel, J. Anderson, and N. E. Jerger. DistCL: A framework for the distributed execution of OpenCL kernels. In *IEEE 21st International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 556–566, 2013.

[8] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 151–162, 2006.

[9] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, Oct. 2002.

[10] C. Gregg, M. Boyer, K. Hazelwood, and K. Skadron. Dynamic heterogeneous scheduling decisions using historical runtime data. In *2nd Workshop on Applications for Multi- and Many-Core Processors*, 2011.

[11] S. Gulwani, K. K. Mehra, and T. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *Conference Record of the 38th Annual ACM Symposium on Principles of Programming Languages*, pages 127–139, New York, NY, USA, 2009. ACM.

[12] S. Gulwani and F. Zuleger. The Reachability-bound Problem. In *Proc. of the '10 Conference on Programming Language Design and Implementation*, pages 292–304, New York, NY, USA, 2010. ACM.

[13] Intel. Intel Core i7-3700 Desktop Processor Series, 2012. http://download.intel.com/support/processors/corei7/sb/core_i7-3700_d.pdf.

[14] R. Kaleem, R. Barik, T. Shpeisman, B. T. Lewis, C. Hu, and K. Pingali. Adaptive Heterogeneous Scheduling for Integrated GPUs. In *Proc. of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, pages 151–162, 2014.

[15] KHRONOS. OpenCL - the open standard for parallel programming of heterogeneous systems, 2014.

[16] J. Kim, H. Kim, J. H. Lee, and J. Lee. Achieving a single compute device image in OpenCL for multiple GPUs. In *Proc. of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 277–288, 2011.

[17] K. Kofler, I. Grasso, B. Cosenza, and T. Fahringer. An Automatic Input-sensitive Approach for Heterogeneous Task Partitioning. In *Proc. of the 2013 International Conference on Supercomputing*, pages 149–160, 2013.

[18] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proc. of the '08 Conference on Programming Language Design and Implementation*, pages 114–124, June 2008.

[19] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of the 2004 International Symposium on Code Generation and Optimization*, pages 75–86, 2004.

[20] J. Lee, M. Samadi, Y. Park, and S. Mahlke. Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems. In *Proc. of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 245–256, 2013.

[21] C.-K. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proc. of the 42nd Annual International Symposium on Microarchitecture*, pages 45–55, 2009.

[22] C. McGeoch, P. Sanders, R. Fleischer, P. R. Cohen, and D. Precup. *Experimental Algorithmics*. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[23] S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[24] NVIDIA. NVIDIA's next generation CUDA compute architecture: Kepler GK110, 2012. www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf.

[25] NVIDIA. *CUDA C Programming Guide*, 2014. http://docs.nvidia.com/cuda.

[26] NVIDIA. Sharing a GPU between MPI processes: Multi-process service (MPS) overview, 2014. http://docs.nvidia.com/deploy/mps/index.html.

[27] D. Ofelt and J. L. Hennessy. Efficient Performance Prediction for Modern Microprocessors. In *2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer System*, pages 229–239, New York, NY, USA, 2000. ACM.

[28] OpenACC. Directives for accelerators, 2014. http://www.openacc.org.

[29] P. Pandit and R. Govindarajan. Fluidic Kernels: Cooperative Execution of OpenCL Programs on Multiple Heterogeneous Devices. In *Proc. of the 2014 International Symposium on Code Generation and Optimization*, pages 273–283, 2014.

[30] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating System Abstractions to Manage GPUs As Compute Devices. In *Proc. of the 23rd ACM Symposium on Operating Systems Principles*, pages 233–248, 2011.

[31] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: A Compiler and Runtime for Heterogeneous Systems. In *Proc. of the 24th ACM Symposium on Operating Systems Principles*, pages 49–68, 2013.

[32] G. Shobaki and K. Wilken. Optimal Superblock Scheduling Using Enumeration. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 283–293, Washington, DC, USA, 2004. IEEE Computer Society.

[33] H. Topcuouglu, S. Hariri, and M. you Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, Mar. 2002.

[34] D. Zaparanuks and M. Hauswirth. Algorithmic profiling. In *Proc. of the '12 Conference on Programming Language Design and Implementation*, pages 67–76, 2012.