# VIRTUALIZING DATA PARALLEL SYSTEMS FOR PORTABILITY, PRODUCTIVITY, AND PERFORMANCE

by

Janghaeng Lee

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2015

Doctoral Committee:

Professor Scott Mahlke, Chair
Nathan Clark, Virtu Financial
Associate Professor Kevin Pipe
Assistant Professor Lingjia Tang
Associate Professor Thomas Wenisch

To my family

# ACKNOWLEDGEMENTS

I would like to express my first and foremost gratitude to my research advisor, Professor Scott Mahlke, for his continuous support of this research. I consider myself truly lucky to have worked with him these past years. He has shown incredible patience, served as an excellent mentor from immense knowledges, encouraged me all the time, and guided pathways to success in this field.

My sincere gratitude also goes to my thesis committee, Prof. Thomas Wenisch, Prof. Kevin Pipe, Prof. Lingia Tang, and Nathan Clark for providing excellent comments and suggestions that helped me to make this work more valuable. I am grateful to Nathan Clark, the former advisor at Georgia Tech, who brought me to this area and enlightened me the first sight of research in this field. Also, I thank to Neungsoo Park who led me out into the graduate study in the United States.

It was very fortunate to be a member of Compilers Creating Custom Processors (CCCP) research group. I specially thank to Mehrzad Samadi, who is a great collaborator throughout years, providing significant helps in this work. It would not possible to have my thesis in the current shape without his support. I would also like thank a number of other students and alumni in the CCCP research group: Shantanu Gupta, Yongjun Park, Hyoun Kyu Cho, Ankit Sethia, Gaurav Chadha, Anoushe Jamshidi, Daya Khudia, Andrew Lukefahr, Shruti

Padmanabha, Jason Jong Kyu Park, John Kloosterman, Babak Zamirai, and Jiecao Yu. All the people in the group are tightly bound together helping each other, and they made my PhD life much more enjoyable. In particular, Mehrzad, Ankit and Daya gave funny-and-stupid jokes all the time although I do not care, Jason Jong Kyu helped me when I encountered with math problems, and Shantanu played a great role as a mentor during my internship at Intel, giving me an opportunity to work on a great project.

My special thanks are extended to my Korean friends who made my time in Ann Arbor more enjoyable. Especially, I thank to Jason Jong Kyu Park and Eugene Kim as I could have tasty meals every dinner.

Finally and most importantly, my family deserves endless gratitude. My father, Jungsik Lee, and my mother, Sunmee Kim, always gave me the unconditional love and support. Whatever I am, I stand here because of them. I appreciate my brothers, Jihaeng and Jang-wook. I have unforgettable childhood memories with them. They are always supportive and encouraging all the time.

# TABLE OF CONTENTS

# LIST OF FIGURES

**Figure**

# LIST OF TABLES

**Table**

# ABSTRACT

VIRTUALIZING DATA PARALLEL SYSTEMS FOR PORTABILITY,
PRODUCTIVITY, AND PERFORMANCE

by

Janghaeng Lee

Chair: Scott Mahlke

Computer systems equipped with graphics processing units (GPUs) have become increasingly common over the last decade. In order to utilize the highly data parallel architecture of GPUs for general purpose applications, new programming models such as OpenCL and CUDA were introduced, showing that data parallel kernels on GPUs can achieve speedups by several orders of magnitude. With this success, applications from a variety of domains have been converted to use several complicated OpenCL/CUDA data parallel kernels to benefit from data parallel systems. Simultaneously, the software industry has experienced a massive growth in the amount of data to process, demanding more powerful workhorses for data parallel computation. Consequently, additional parallel computing devices such as extra GPUs and co-processors are attached to the system, expecting more performance and capability to process larger data.

However, these programming models expose hardware details to programmers, such as

the number of computing devices, interconnects, and physical memory size of each device. This degrades productivity in the software development process as programmers must manually split the workload with regard to hardware characteristics. This process is tedious and prone to errors, and most importantly, it is hard to maximize the performance at compile time as programmers do not know the runtime behaviors that can affect the performance such as input size and device availability. Therefore, applications lack portability as they may fail to run due to limited physical memory or experience suboptimal performance across different systems.

To cope with these challenges, this thesis proposes a dynamic compiler framework that provides the OpenCL applications with an abstraction layer for physical devices. This abstraction layer virtualizes physical devices and memory sub-systems, and transparently orchestrates the execution of multiple data parallel kernels on multiple computing devices. The framework significantly improves productivity as it provides hardware portability, allowing programmers to write an OpenCL program without being concerned of the target devices. Our framework also maximizes performance by balancing the data parallel workload considering factors like kernel dependencies, device performance variation on workloads of different sizes, the data transfer cost over the interconnect between devices, and physical memory limits on each device.

# CHAPTER I

# Introduction

Over the past decade, heterogeneous computer systems that combine multicore processors (CPUs) with graphics processing units (GPUs) have emerged as the dominant platform. The advent of new programming models such as OpenCL [37] and CUDA [58] makes it possible to utilize GPUs for processing massive data in parallel for general purpose applications. By leveraging these programming models, programmers can develop data parallel kernels for GPUs that achieve speedup of 100-300x in optimistic cases [55], and speedup of 2.5x in pessimistic cases [46].

As a result of these new ways to use massive data parallel hardware, applications from a variety of domains have been converted to OpenCL/CUDA programs. Meanwhile, the industry for large-scale data-intensive applications has grown rapidly, and now requires higher performance on data parallel processing of much larger sets of data [8, 59, 16]. Because hardware vendors cannot meet these demands with a single computing device (CPU or GPU including off-chip memories), they configured systems with additional GPUs, expecting applications to benefit from the additional devices.

Now the burden of improving performance and handling large data sets on increased

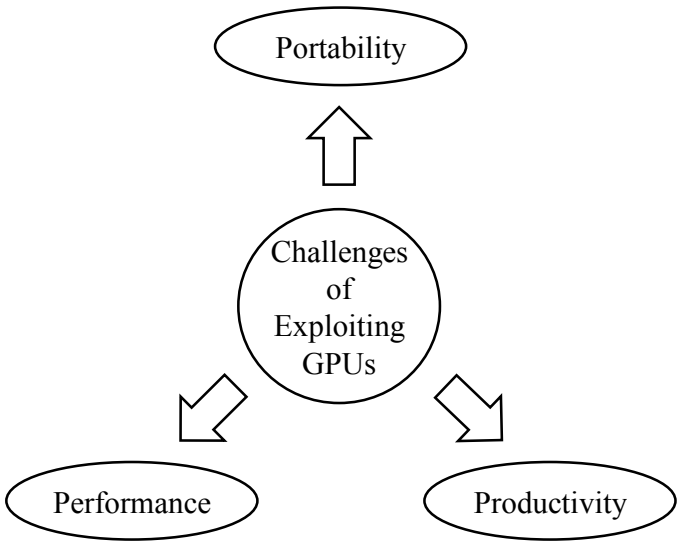



**Figure 1.1:** Challenges in exploiting multiple GPUs for large data sets. Because the programming model exposes the hardware details, programmers must consider portability, productivity, and performance when they write the data parallel kernels with large data on multiple devices.

computing devices must come from the software layers, e.g. applications, libraries, compilers and operating systems (OSs). In traditional software, the benefit from improved hardwares came for free as OSs virtualize underlying hardwares by providing the illusion of private computing resources to an application. Therefore, application programmers do not have to consider target hardware such as processor types and memory sub-systems for optimizing an application.

For data parallel software, although the OpenCL/CUDA programming model alleviated part of the complexity by providing unified processing interfaces, it still exposes hardware details to programmers, such as the number of processing elements, the size of off-chip memory, and interconnects between computing devices. Due to the absence of the virtualization layer between the hardware and OpenCL/CUDA program, we found three main challenges in using multiple GPUs: portability, productivity, and performance as shown in Figure 1.1.

## 1.1 Challenges of Using Multiple GPUs

**Portability:** Different GPUs have different architectural specifications, e.g. the number of cores, the number of registers, maximum number of threads per processors, and the size of global memory. As a result, a data-parallel kernel optimized for a specific GPU is not guaranteed to be optimal for other GPUs. In the worst case, the execution may fail on other GPUs if there are not enough resources, such as physical memory. Most importantly, OpenCL/CUDA programming model exposes the computing devices of the system, so application programmers must write the code to list up available devices and pick a GPU to process data parallel kernels. Also, if a program is written to use a single GPU, simply attaching new additional GPUs does not bring any performance improvement.

**Productivity:** In order to make data parallel kernels with large data sets run on multiple GPUs, the programmer must restructure their code to operate within the limited physical memory space of a GPU by following several steps: 1) manually divide working sets to create a set of partial workloads; 2) change the kernel if the algorithm depends on the size of the working set; 3) transfer the working set back and forth between the application and GPUs; and, 4) merge the partial outputs from different GPUs into the application's memory address space. This process requires a deep analysis of the memory access pattern of the target kernel's working set, and substantial code is necessary to facilitate communication management between the application and GPU.

In addition to these efforts, if an application consists of multiple data parallel kernels, programmers must analyze the workload of each kernel to map kernels properly into the devices. During this process, they must also consider dependencies and communication

cost between kernels.

This process is tedious and prone to errors and may fail if programmers are unable to statically determine the working set due to indirect array accesses.

**Performance:** To maximize the performance of data parallel kernels on multiple GPUs, statically determining where to execute or how much of the workload to assign to a device cannot be determined without knowing what resources will be available at the time of execution. For example, if the fastest GPUs are busy with another data parallel kernel, the application should select alternative GPUs as a computing device instead of waiting for the fastest GPU to be free. In addition, interconnects, such as Peripheral Component Interconnect Express (PCIe), must also be considered for the cost of data transfer because GPUs have separate memories that use different address spaces. If PCIe bus bandwidth is saturated by transferring another kernel's data, workload distribution over multiple GPUs must be different from the case where the bus is idle. With these dynamic behaviors, it is hard for programmers to statically decide a workload distribution that maximizes the performance.

Obviously, shifting all the burden of solving these issues on programmers is an unachievable goal. Instead, it is desirable to push as much of these responsibilities as possible to an additional abstraction layer of software that provides a seamless adaptation of application to hardware.

## 1.2 Contributions

In this thesis, we propose a dynamic compiler framework that significantly improves portability, productivity, and performance for multiple OpenCL kernels on multiple heterogeneous devices. This is accomplished by virtualizing computing units (CPU cores or GPU's streaming multi-processors), physical memory of GPUs, and the interconnect between devices. With the information of the underlying system, the framework takes multiple kernels from the applications, and schedules them on the physical devices considering kernel dependencies. The framework is fully transparent to OpenCL applications, thus the only responsibility for programmers is to write OpenCL kernels assuming that there is a single data parallel device.

The remainder of this chapter describes different frameworks that are specifically designed to virtualize multiple devices and physical memory space, and to schedule multiple kernels for improving portability, productivity, and performance.

### 1.2.1 SKMD

In order to improve portability, productivity, and performance of OpenCL kernels on multiple devices, we propose *Single Kernel Multiple Device* (SKMD) [44], a dynamic system that transparently orchestrates the execution of a single kernel across asymmetric heterogeneous devices regardless of memory access pattern. SKMD transparently partitions an OpenCL kernel across multiple devices being aware of the transfer cost and performance variation on the workload, launches partitioned kernels for each devices, and merges the partial results into the final output automatically. For partitioning, performance for each de-

vice is predicted through a linear regression model which is trained offline. Using the performance prediction model, partitioning decision is made using *steepest ascent hill climbing heuristic* in order to minimize the execution time. SKMD is fully transparent to the applications, so it provides the illusion of a single device by virtualizing physical devices. As a result, SKMD not only eliminates the tedious process of re-engineering applications when the hardware changes, but also makes efficient partitioning decisions based on application characteristics, input sizes, and the underlying hardware. More details of SKMD are discussed in Chapter III.

### 1.2.2 VAST

Although SKMD provides an abstraction layer for computing units and interconnects, it does not virtualize memory space of GPUs, exposing the physical memory of GPUs to the programmer. Without virtual address space on GPUs, SKMD cannot handle an application with large data that exceeds the physical memory of GPUs, so OpenCL applications are not fully portable to GPUs even with SKMD. For larger data sets, programmers must manually split the working set of the application to make data fit into physical memory, and manage the data transfer between the application and GPUs. This is still a huge burden for programmers. To increase the programming productivity, we present *Virtual Address Space for Throughput processors* (VAST) [43], a runtime system that provides programmers with an illusion of a virtual memory space for OpenCL devices. With VAST, the programmer can develop a data parallel kernel in OpenCL without concern for physical memory space limitations. In order to virtualize the memory space for GPUs, VAST uses a *inspector-executor* model, which inspects memory footprints of each thread before the real

execution, and efficiently extracts required working set of a subset of threads so as to not exceed the physical memory of GPUs. The extracted data is reorganized into contiguous memory space, and page tables are created for the address translation. Later, a subset of threads are executed accessing data through software address translation, and the execution of a subset of threads is repeated until all the threads finish their executions. Because VAST is able to process regardless of the type of kernels and fully transparent to the applications, it significantly improves portability and productivity of OpenCL applications. In Chapter IV, VAST is discussed in more detail.

### 1.2.3 MKMD

As more applications are converted to utilize the highly data parallel architectures, applications are composed of several data parallel kernels communicating one another. Consequently, it is critical to map data parallel kernels properly onto multiple data parallel hardware in order to maximize the performance. However, it is difficult to manually map several data parallel kernels onto several computing devices because programmers must consider many factors like input size, type of data parallel kernels, kernel dependencies, the number of computing devices, and the interconnect between devices.

While SKMD and VAST virtualize computing resources and address space of GPUs, they focus only on a single kernel, so their executions can be suboptimal in terms of multiple data parallel kernels. For example, if there are two kernels that are independent each other, it could be better mapping kernels onto different devices separately rather than applying SKMD for each kernel.

To tackle this challenge, this thesis proposes *Multiple Kernels on Multiple Device*

(MKMD), a runtime system that does temporal scheduling of multiple kernels along with spatial partitioning across multiple devices. To achieve this goal, MKMD proposes a two-phase scheduling. The first phase builds a kernel graph and schedules at a kernel granularity maximizing the resource utilization. In this phase, an entire kenrel is executed by a single device. After than, the second phase reschedules kernels at the work-group granularity by spatially splitting kernels into sub-kernels considering temporal available computing resources. As a result of this phase, idle time slots on devices are removed. Further details of MKMD is described in Chapter V.

# CHAPTER II

# Background

The OpenCL programming model uses a single-instruction multiple thread (SIMT) model that enables implementation of general purpose programs on heterogeneous CPU/GPU systems. An OpenCL program consists of a host code segment that controls one or more OpenCL devices. Unlike the CUDA programming model, *devices* in OpenCL can refer only to both CPUs and GPUs whereas *devices* in CUDA usually refer to GPUs. The host code contains the sequential code sections of the program, which are run on the CPUs, and a parallel code is dynamically loaded into a program's segment. The parallel code section, i.e. *kernel*, can be compiled at runtime if the target device cannot be recognized at compile time, or if a kernel runs on multiple devices.

The OpenCL programming model assumes that underlying devices consist of multiple compute units (CUs) which are further divided into processing elements (PEs). The OpenCL execution model consists of a three level hierarchy. The basic unit of execution is a single *work-item*. A group of work-items executing the same code are stitched together to form a *work-group*. Once again, these work-groups are combined to form parallel segments called *NDRange, N-Dimensional Range*, where each NDRange is scheduled by a command

queue. Work-items in a work-group are synchronized together through an explicit barrier operation. When executing a kernel, work-groups are mapped to CUs, and work-items are assigned to PEs. In real hardware, since the number of cores are limited, CUs and PEs are virtualized by the hardware scheduler or OpenCL drivers. For example, NVIDIA devices virtualize an unlimited number of CUs on physical streaming multi-processors (SMs) by quickly switching context of a work-group to another using a hardware scheduler.

For scheduling work-groups, devices do not have to consider the execution order of work-groups because the programming model relies on the relaxed memory consistency model. The OpenCL programming model uses relaxed memory consistency for *local* memory within a work-group and for *global* memory within a kernel's workspace, *NDRange*. Each work-item in the same work-group sees the same view of local memory only at a synchronization point where a *barrier* appears. Likewise, every work-group in the same kernel is guaranteed to see the same view of the global memory only at the end of kernel execution, which is another synchronization point. This means that the ordering of execution is not guaranteed across work-groups in a kernel, but only guaranteed across synchronization points.

Based on this memory consistency model, an OpenCL kernel can be executed in parallel at work-group granularity without concern of the execution order. If a kernel executes a subset of work-groups instead of the entire NDRange, the result at the end of kernel execution would be incomplete. However, if the rest of the work-groups are executed after all, it would correctly complete regardless of type of application. This feature enables scheduling a subset of work-groups by software even on separate devices that use different address spaces. By simply assigning subsets of work-groups to several devices exclusively,

partial results would appear interleaved in their address spaces. The final result can be made when the partial results are properly merged.

# CHAPTER III

# SKMD: Single Kernel Execution on Multiple Devices

## 3.1 Introduction

Heterogeneous computer systems with traditional processors (CPUs) and graphic processing units (GPUs) have become the standard in most systems from cell phones to servers. GPUs achieve higher performance by providing a massively parallel architecture with hundreds of relatively simple cores while exposing parallelism to the programmer. Programmers are able to effectively develop highly threaded data-parallel kernels to execute on the GPUs using OpenCL or CUDA. Meanwhile, CPUs also provide affordable performance on data-parallel applications armed with higher clock-frequency, low memory access latency, an efficient cache hierarchy, single-instruction multiple-data (SIMD) units, and multiple cores. With these hardware characteristics, many studies have been done to improve the performance of data-parallel kernels on both CPUs and GPUs [46, 75, 12, 24, 33, 26, 17].

More recently, systems are configured with several different types of processing devices, such as CPUs with integrated GPUs and multiple discrete GPUs or data parallel coprocessors for higher performance. However, as most data-parallel applications are written

to target a single device, other devices will likely be idle, which results in underutilization of the available computing resources. One solution to improve the utilization is to asynchronously execute data-parallel kernels on both CPUs and GPUs, which enables each device to work on an independent kernel [13]. Unfortunately, this approach requires programmer effort to ensure there are no inter-kernel data dependences. In spite of this effort, if dependences cannot be eliminated, but several kernels are dependent on a heavy kernel, the default execution model of one kernel at a time must be used.

To alleviate this problem, several prior works have proposed the idea of splitting threads of a single data-parallel kernel across multiple devices [49, 38, 36]. Luk et al. [49] proposed the Qilin system that automatically partitions threads to CPUs and GPUs by providing new APIs. However, Qilin only works for two devices (one CPU and one GPU), and the applicable data parallel kernels are limited by usage of the APIs, which requires access locations of all threads to be analyzed statically. Kim et al. [38] proposed the illusion of a single compute device image for multiple equivalent GPUs. Although they improved the portability by using OpenCL as their input language, their work also puts several constraints on the types of kernels in order to benefit from multiple equivalent GPUs. For example, the access locations of each thread must have regular patterns, and the number of threads must be a multiple of the number of GPUs.

Despite individual successes, the majority of data parallel kernels still cannot benefit from multiple computing devices due to strict limitations on the underlying hardware and the type of data-parallel kernels. As hardware systems are configured with more than two computing devices and more scientific applications have been converted to more complicated OpenCL/CUDA data-parallel kernels in order to benefit from heterogeneous archi-

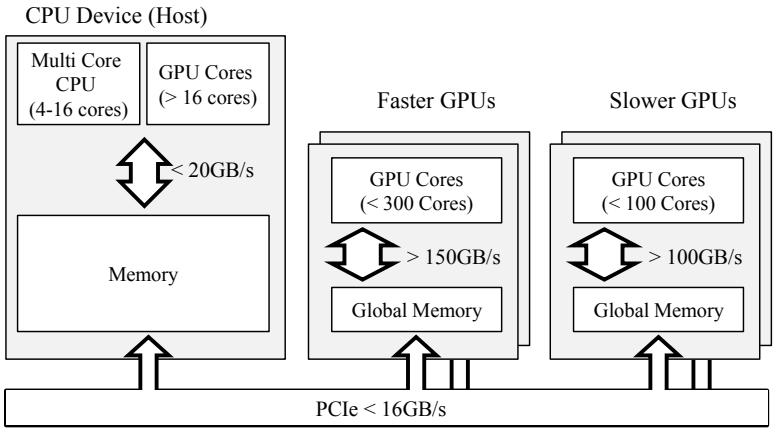



**Figure 3.1:** Physical OpenCL computing devices with different performances, memory spaces, and bandwidths.

tectures, these limitations become more significant. To overcome these limitations, we have identified three central challenges that must be solved to effectively utilize multiple computing devices:

**Challenge 1: Data-parallel kernels with irregular memory access patterns are hard to partition over multiple devices.** Memory read/write locations of adjacent threads may not be contiguous, or the access location of each thread may depend on control flow or input data. This kind of data-parallel kernel discourages partitioning over multiple devices because the irregular locations of input data must be properly distributed over multiple devices before execution, and output data must be gathered correctly after execution.

**Challenge 2: The partitioning decision becomes more complicated when systems are equipped with several types of devices.** As shown in Figure 3.1, a system may have several GPUs which have different performance and memory bandwidth characteristics. In addition, some computing devices, such as CPUs or integrated GPUs, can share the memory space with the host program while external GPUs cannot because they are physically separated. In this case, the partitioning decision must be made very carefully with regard

to the cost of data transfer in addition to the performance of each device.

**Challenge 3: The performance of a GPU is often not constant to the amount of data that it operates upon, and this variation will affect the partitioning decision.** This problem is more significant for memory-bound kernels where each thread spends most of its time on memory accesses. For this type of kernel, GPUs hide memory access latency by switching context to other groups of threads. With fewer threads, more memory latency is exposed that often leads to disproportionately worse performance. This behavior makes the partitioning decisions more complex since the partitioner must consider the performance variation of GPUs.

In this dissertation, we propose SKMD (Single Kernel Multiple Devices), a dynamic system that transparently orchestrates the execution of a single kernel across asymmetric heterogeneous devices regardless of memory access pattern. SKMD transparently partitions an OpenCL kernel across multiple devices being aware of the transfer cost and performance variation on the workload, launches parallel kernels, and merges the partial results into the final output automatically. This dynamic system not only eliminates the tedious process of re-engineering applications when the hardware changes, but also makes efficient partitioning decisions based on application characteristics, input sizes, and the underlying hardware.

The challenge for transparent collaborative execution is threefold: 1) generating kernels that execute a partial workload; 2) deciding how to partition the workload accounting for transfer cost and performance variation; and, 3) efficiently merging irregular partial outputs. To solve these problems, this dissertation makes the following contributions:

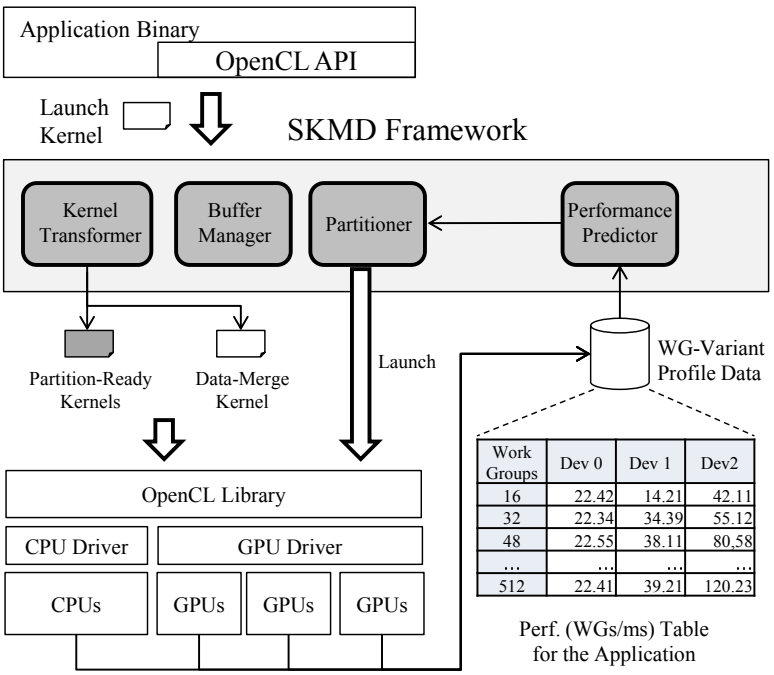



**Figure 3.2:** The SKMD framework consisting of four units: Kernel Transformer, Buffer Manager, Partitioner, and Profile Database.

- The SKMD runtime system that accomplishes transparent collaborative execution of a data-parallel kernel.

- A code transformation methodology that distributes data and merges results in a seamless and efficient manner regardless of the data access pattern.

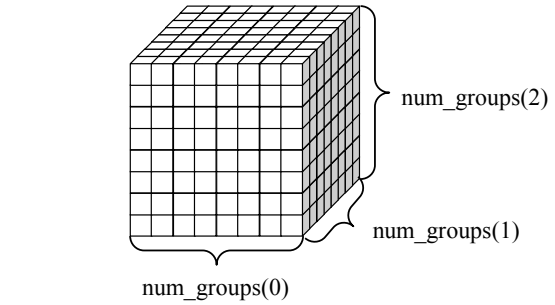
- A performance prediction model that accurately predicts the execution time of OpenCL kernels based on offline profile data.

- A partitioning algorithm that balances the workload among multiple asymmetric CPUs and GPUs considering the performance variation of each device.

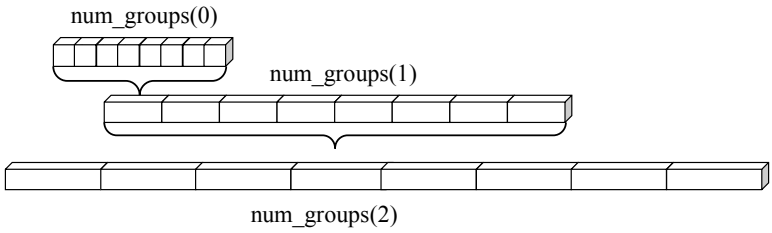
## 3.2 SKMD System

SKMD is an abstraction layer located between applications and the OpenCL library. Since OpenCL supports both CPUs and GPUs as computing devices, it is selected as the language for SKMD. The SKMD layer hooks into every OpenCL application programming interface (API) including querying-platform APIs. For querying-platform APIs, SKMD returns an illusion of virtual platform with only one large available device. SKMD maintains all information such as device buffer size, kernel name, and kernel arguments in an internal mapping table, and does not pass them to the real OpenCL libraries but returns a fake value (e.g. $CL\_SUCCESS$) immediately to the application until the kernel launch ($clEnqueueNDRangeKernel$) is requested. The framework consists of a profiler to collect performance metrics for each device by varying the number of work-groups, and a dynamic compiler to transform and execute the data-parallel kernel on several devices as shown in Figure 4.2.

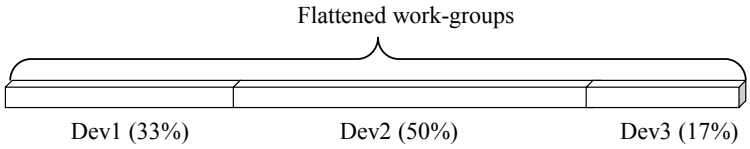
The Dynamic compiler has four units: **kernel transformer**, **buffer manager**, **partitioner**, and **performance predictor** as shown in the grey boxes in Figure 4.2. The kernel transformer changes the original kernel to the *Partition-ready* kernel, which enables the kernel to operate on a subset of work-groups. After kernel transformation, the buffer manager performs static analysis on kernels to determine the memory access pattern of each work-group. If the memory access range of each work-group can be analyzed statically, the buffer manager will transfer only necessary data back and forth from each device once the partitioning decision is made. On the other hand, if the memory access range cannot be analyzed, the entire input should be transferred to each device and the output must be




**(a)** 3-dimensional work-group range




**(b)** Flattened view of 3-dimensional work-groups




**(c)** Partitioning on flattened work-groups

**Figure 3.3:** OpenCL's N-Dimensional range

merged. In order to merge irregular locations of output from different devices, the kernel transformer generates the *Merge* kernel, and SKMD launches it on the CPU.

Once kernel analysis and transformation are done, ranges of work-groups to execute on each device are decided by the partitioner considering the workload performance on each device. To estimate performance, the performance predictor utilizes a *linear regression model* based upon offline profile data. If the profile information does not exist, SKMD executes a dry run with *Partition-ready kernels* varying number of the work-groups for each device in order to collect the data. After the partitioning decision is made, the buffer manager transfers necessary data from the host to external devices, then SKMD launches the actual kernel for each device.

The rest of this section discusses these three components of SKMD: kernel transformation, buffer management, performance prediction, and performance variation-aware partitioning.

### 3.2.1 Kernel Transformation

As OpenCL kernels can launch up to three dimensional work-groups, the kernel transformation flattens N-dimensional work-groups to one dimension to assign balanced work over all devices at a work-group granularity. For example, Figure 3.3(a) shows three dimensional ranges, each of which has 8 work-groups. Figure 3.3(b) shows the flattened view, which has 512 work-groups in a single dimension. Once the SKMD framework has the flattened view of N-dimensional work-groups, it assigns a subset of work-groups in the flattened range as shown in Figure 3.3(c). Based on this idea, the next subsection discuss how SKMD generates the *Partition-ready* and *Merge* kernels.

***Partition-Ready Kernel***: Assigning partial work-groups can be done through the code transformation shown in Figure 4.1. The lines of code with gray background illustrate the generated code by dynamic compiler. As shown in the figure, it adds a parameter `WG_from` and `WG_to` to represent the range of the flattened work-group indices to be computed on a device. In other words, SKMD runs $(WG\_from - WG\_to + 1)$ work-groups and skips the rest on a device. If a kernel launches more than a one dimensional NDRange, flattening code is inserted as shown at line 11 in Figure 4.1. After flattening, each work-item identifies its work-group index ($flattened\_id$) and checks if it is allowed to execute the kernel.

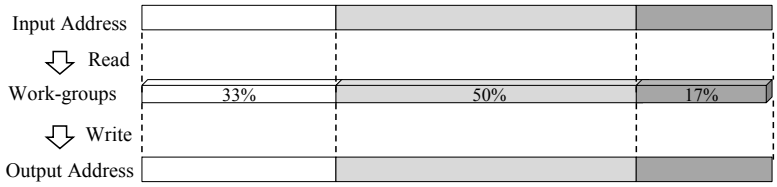
The additional code with gray background is lowered to 3-9 instructions in PTX and x86-64 ISAs. These additional instructions consist of loading indices and dimension sizes,

```
1 __kernel void Blackscholes_CPU(
2         __global float *call,
3         __global float *put,
4         __global float *price,
5           global float *strike, float r, float v,
6         int WG_from, int WG_to)
7 {
8    int idx = get_group_id(0);
9    int idy = get_group_id(1);
10   int size_x = get_num_groups(0);
11   int flattened_id = idx + idy * size_x;
12   // check whether to execute
13   if (flattened_id < WG_from || flattened_id > WG_to)
14       return;
15   int tid = get_global_id(1) * get_global_size(0)
16             + get_global_id(0);
17   float c, p;
18   BlackScholesBody(&c, &p,
19            price[tid], strike[tid], r, v);
20   call[tid] = c;
21   put[tid] = p;
22 }
```

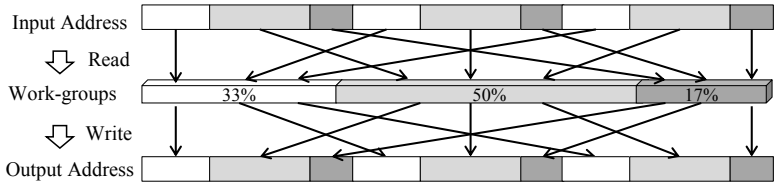
**Figure 3.4:** Partition-ready Blackscholes kernel.

MADDs, comparisons, and branches. For PTX code, however, there is no actual load instruction for indices and sizes, because GPUs maintain special registers for them, and they are available to each work-item and work-groups [60]. Nonetheless, these instructions can be unnecessary overhead for disabled work-groups in GPUs. To estimate this overhead, *VectorAdd* was tested with NVIDIA GTX 760 by enabling only one work-group out of 524,288 work-groups, each of which consists of 256 work-items. As a result, the overhead for this checking code on the GPU is 2.687 ns / work-groups. This overhead can be eliminated if GPU vendors provide interfaces to the software for work-group scheduling, so that the runtime system can run partial work-groups without imposing additional work for disabled work-groups.

On the other hand, for x86 code, the checking code in CPUs may produce significant overhead as the Intel OpenCL driver executes a kernel in the same way that Diamos et al. [12] proposed. In their work, the driver transforms a kernel to be wrapped by $N$-nested loops in order for CPUs to execute $N$-dimensional work-items in a work-group. This is




**(a)** Contiguous Access Kernel




**(b)** Discontiguous Access Kernel

**Figure 3.5:** Different memory access patterns of kernels

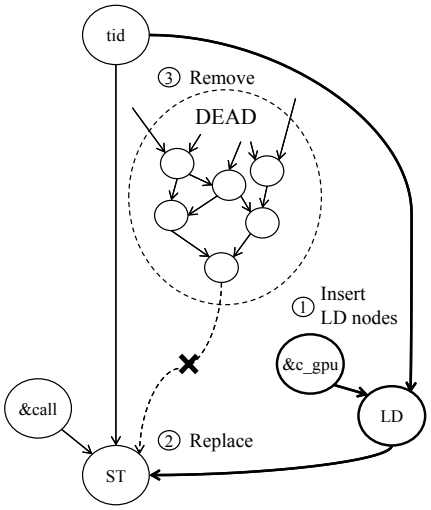
necessary because the context of each work-item in CPUs must be switched by the code, not the hardware. After the transformation, the driver iterates over work-groups distributing them to multiple threads in order to fully utilize multiple CPU cores. Unfortunately, this leaves CPU execution inefficient for the *Partition-ready* kernel as CPUs must execute checking code serially with actual load instructions inside the innermost loop, even though checking whether to execute is independent from inner loops.

To avoid this problem, SKMD is configured with a specialized OpenCL driver for CPU devices. The specialized driver directly takes the range of enabled work-groups, so the SKMD system does not transform a kernel but the driver selectively iterate over work-groups. Through this loop-independent code motion, SKMD eliminates the overhead of checking code within the kernel code.

***Merge Kernel***: Another challenge of collaborative execution of a single data-parallel kernel is that several computing devices may use different address spaces, so the results from each device *must* be merged after execution. Some kernels have contiguous memory accesses called, *Contiguous* kernel, where each of the threads writes the result in contigu-

```
__kernel void Blackscholes(
        __global float *call,
        __global float *put,
        __global float *price,
        __global float *strike,
        float r,
        float v)
{
    int tid = get_global_id(1) *
        get_global_size(0) + get_global_id
    float c, p;
    BlackScholesBody(&c, &p,
                 price[tid],
                 strike[tid],
                 r, v);
    call[tid] = c;
    put[tid] = p;
}
```

**(a)** Blackscholes Kernel




**(b)** Data Flow Graph

```
__kernel void Blackscholes_Merge(
        __global float *call,
          global float *put,
        __global float *c_gpu,
        __global float *p_gpu,
        int GPU_from, int GPU_to)
{
    int idx = get_group_id(0);
    int idy = get_group_id(1);
    int size_x = get_num_groups(0);
    int flat_id = idx + idy * size_x;
    // check whether to execute
    if (flat_id < GPU_from || flat_id > GPU_to)
        return;
    int tid = get_global_id(1) *
              get_global_size(0) + get_global_id(0);
    // computation code is removed by DCE
    call[tid] = c_gpu[tid];
    put[tid] = p_gpu[tid];
}
```

**(c)** Merge Kernel for Blackscholes

**Figure 3.6:** Merge Kernel Transformation Process. Only global output parameters, `call` and `put` in (a), are marked for merging. Using data flow analysis, store values to global output parameters are replaced with GPU's partial results, and then proceed with dead code elimination (b). As a result, the merge kernel does not have computational part (c).

ous locations as shown in Figure 3.5(a). In this case, partial outputs can be merged at lower cost by simply concatenating partial output from the external GPU devices to the host.

On the other hand, for *Discontiguous* kernels that have discontiguous memory accesses, it is difficult to merge partial output. For example, matrix multiplication is usually implemented by assigning a work-group to work on a tile. Because a two dimensional matrix

is flattened to a single dimensional array, writing locations of consecutive work-groups become discontiguous as shown in Figure 3.5(b). Clearly, this type of memory layout can cause significant overhead for merging outputs. The overhead is high because the output cannot be copied at once, so each device has to keep the write location for merging and selectively copies the data afterward.

To solve this problem, SKMD uses a code transformation technique that automatically merges the data without storing memory-write locations and takes full advantage of the data/thread parallelism in multi-core CPUs. SKMD merges the output without storing memory-write locations by reusing the original kernel function for merging partial outputs. In the CPU device, enabled work-items will write their results to the host's memory, while locations for disabled work-items will remain untouched. Instead, the kernels launched in external GPU devices touch those locations in their own address space. Thus, transferring the GPU devices' output to the host and then selectively copying them to the CPU output would complete the final results. In order to selectively copy the external GPU results, SKMD launches the *Merge* kernel to regenerate the addresses that external GPU devices modified in their output.

To illustrate how merge kernel is generated, Figure 3.6(a) shows the original Blackscholes kernel that generates two output arrays (`call` and `put`). For the merge kernel shown in Figure 3.6(c), the dynamic compiler inserts a parameter `GPU_from` and `GPU_to`, as well as two additional parameters, `p_gpu` and `c_gpu`, which are the GPU's partial output arrays (put prices and call prices) transferred to the host's memory. Output parameters of the kernel can be determined by the basic data-flow analysis, checking whether `__global` pointer parameters are used for store. For kernels that copy `__global` pointer parameters

to temporary local variables, SKMD uses alias analysis to keep track of the usage of those pointer variables. The condition for enabled work-group of *Merge* kernel is equivalent to that of *Partition-ready* kernel as shown at line 13 in Figure 3.6(c).

Once the GPU output parameters have been set up, the dynamic compiler follows several steps to transform the kernel as illustrated in Figure 3.6(b). The first step is to match the base of the store instruction to the base of the output parameter from the GPU using use-def chains. After the dynamic compiler gets the corresponding base, it inserts a load instruction with the base and the same offset of the store instruction. Next, it replaces the value of store instruction with the loaded value as shown in lines 19-20 of Figure 3.6(c). Finally, it marks store instruction as *live* and proceeds with *dead code elimination* using the mark-sweep algorithm [78] to remove all computation code. As a result of this transformation, all computation code, lines 11-15 in Figure 3.6(a), are removed. Note that every function call is inlined before the transformation in order to avoid expensive inter-procedural analysis.

Clearly, transformed merge kernel does not contain any computation code, except the calculation of index for the load and store. With this approach, the cost of merging reaches the bandwidth between CPU cores and main memory (>20 GB/s with DDR-3) regardless of application. That is, 67 MBytes of 4K $\times$ 4K single-precision floating point matrix can be merged in a short time ($< 3.9ms$) compared to total execution time ($< 1500ms$). However, if a kernel finishes the execution quickly, but still has to merge large size of data, merging in the host can be a bottleneck. In this case, SKMD does not partition a kernel across multiple devices as the partitioning algorithm considers merging cost, which is discussed in detail in Section 3.2.4.

### 3.2.2 Buffer Management

In the SKMD framework, the buffer manage is in charge of transferring input/output back and forth between the host and external devices. Since the main idea of SKMD is to assign subsets of work-groups to several devices, each device may not require the entire input data. Likewise, each device will generate a subset of the output, so it is desirable to send only updated output back to the host. Considering that the bandwidth of the PCI express channel is relatively low (less than 6 GB/sec), it becomes critical to reduce the amount of transferring input and output for external GPU devices.

To determine if it is safe to transfer partial data to GPU devices, the buffer manager checks if the kernel is a contiguous kernel by analyzing index space of each work-group. For index space analysis, the buffer manager uses data flow analysis focusing on the index operand of store and load instructions, which is represented as $tid$ in Figure 3.6(b). Using use-def chains, the buffer manager computes the function of index. If the function is affine and represented as $a \cdot (W_0 \cdot w_0 + l_0)$, it is defined as a *Contiguous Kernel*. In this equation, $a$ is a constant or an induction variable of loop, and $w_i$, $l_i$, and $W_i$ represents work-group ID, work-item ID, and size of work-group in the i-th dimension, respectively. For this type of kernel, it is safe to transfer a subset of data to each device proportional to assigned work-groups.

On the other hand, if the index space of the kernel cannot be determined statically, or the affine function fails to be recognized as above, the buffer manager gives up optimizing data transfer and defines it as a discontiguous kernel. In this case, the entire input and output will be transferred back and forth between the host and external devices if the kernel

is partitioned and the *Merge* kernel will be launched at the end.

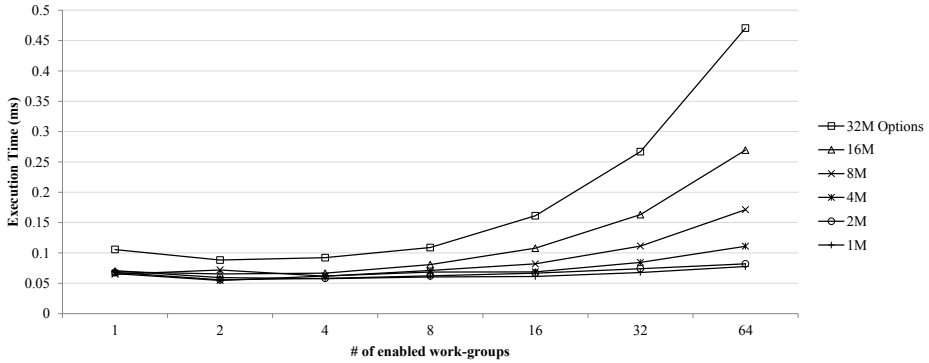
### 3.2.3 Performance Prediction

After the kernel transformations, SKMD statically determines how many work-groups should be assigned across several devices. The goal of the partitioning is to minimize the overall execution time by balancing workload across devices. Therefore, accurate performance prediction for each device is necessary for optimal load balancing. For performance prediction, SKMD relies on offline profile data, which includes the execution time along with the number of partial work-groups and *kernel parameters* such as the size of input, output, value of scalar parameters, and NDRange information. However, SKMD cannot simply rely on the raw profile data because kernel parameters of real execution may be different from those of the profiling execution, and it is unrealistic to profile with all combinations of execution parameters.

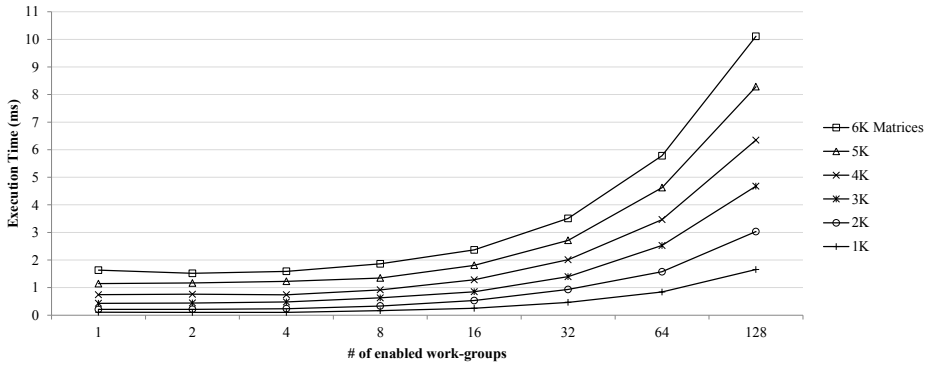
Figure 3.7 illustrates the execution time of Blackscholes (a) and Matrix Multiplication (b) varying the size of input (one of the kernel parameters) and the partial number of work-groups. As shown in the figure, the execution times of each application are dependent both on the size of input and the number of enabled work-groups. In response to this property, SKMD utilizes linear regression analysis model [53] using the equation below.

$$y = \beta_0 + \sum_{i=1}^{n} \beta_i x_i + \epsilon \tag{3.1}$$

For SKMD, the execution time corresponds to $y$, the dependent variable to be predicted, and the properties that can affect the execution time are mapped to $x_i$, independent vari-




**(a)** Blackscholes




**(b)** Matrix Multiplication

**Figure 3.7:** Execution time varied by applications, input size, and the number of enabled work-groups. Depending on the application and input size, the number of enabled work-groups impacts on the execution time differently.

ables. Those properties are the size of each global arguments, values of scalar arguments, the dimension of NDRange, the number of work-groups, and the number of work-items.

A linear regression model requires the dependent variable $y$ to be linear to the combination of coefficients $\beta_i$ and independent variables $x_i$, but the execution time in SKMD may not be represented as a simple combination of $\beta_i$ and $x_i$ as described in Equation 3.1. Figure 3.7 illustrates such case since the execution time is not always linear to the number of work-groups, but becomes linear as the number of work-groups grows. Meanwhile, in some applications, the execution time also may not be linear to the input size when the input size is very small as shown in Figure 3.7(a). To handle these cases, transformations

are applied to independent variables as shown in Equation 3.2.

$$y = \beta_0 + \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{m}\beta_k f_k(x_i, x_j) + \epsilon \tag{3.2}$$

This equation is still a linear regression model since $y$ is linear in the coefficients $\beta_k$, but the only difference is that *transformed independent variables* ($f_k(x_i, x_j)$) are used instead of simple $x_i$. If modeling a linear equation is done by transforming independent variables, the prediction can also be done by plugging transformed variables into the linear equation.

For the transformation, an important observation in SKMD is that the execution time eventually becomes linear to the number of work-groups when the number of work-groups is large, but the point where the linearity appears is varied by application characteristics as shown in Figure 3.7. From this property, the number of work-groups is multiplied by a function that converges from 0 to 1 as the number of work-groups increases. The $tan^{-1}$ function can meet this requirement because it converges to $\frac{\pi}{2}$ from $-\frac{\pi}{2}$. In order to make the $tan^{-1}$ function to converge from 0 to 1, the $tan^{-1}$ function is divided by $\pi$ and then 0.5 is added as shown in Equation 3.3, where $x$ is the number of work-groups.

$$g(x) = \frac{tan^{-1}(a(x-b))}{\pi} + 0.5 \tag{3.3}$$

In this equation, $a$ is an arbitrary number that changes the slope of the $tan^{-1}$ function, and $b$ is another arbitrary number that changes the point that starts to converge. As a result of this function, the linearity to the number of work-groups will grow as the number of work-groups increases. Note that SKMD puts several transformed functions with different $a$ and $b$, so the regression solver will find the best $a$, $b$ values by computing the coefficients.

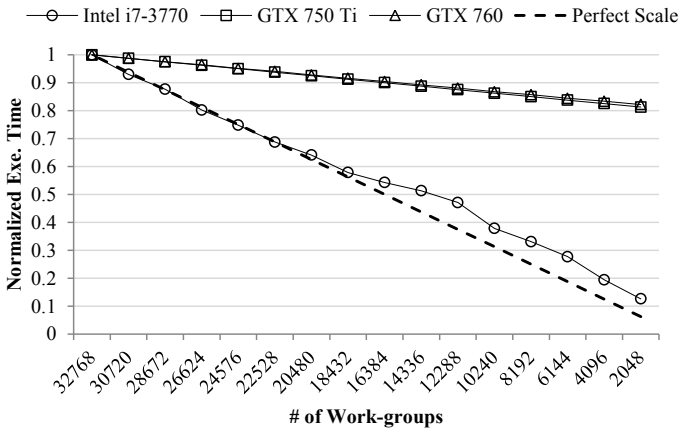



**Figure 3.8:** Performance impact on VectorAdd varying the number of work-groups. The execution time of GPUs do not scale down in spite of reduced number of work-groups.

To this end, a complete transformed function can be represented as Equation 3.4, where $x_i$ is the number of work-groups and $x_j$ is another independent variable.

$$f(x_i, x_j) = x_i g(x_i) h(x_j) \tag{3.4}$$

In this equation, the function $h(x_j)$ is applied for the independent variable $x_j$ because the time complexity of the program may vary. For example, the time complexity of the square matrix multiplication is $O(N^3)$, where $N$ is the number of output elements. In this case, $h(x_j)$ corresponds to ${x_j}^3$, where $x_j$ is the size of output buffer. Note that, SKMD tries various time complexity functions for $h(x_j)$, then the linear regression solver will eventually find the best transformed function by assigning meaningful coefficient.

### 3.2.4 Transfer Cost and Performance Variation-Aware Partitioning

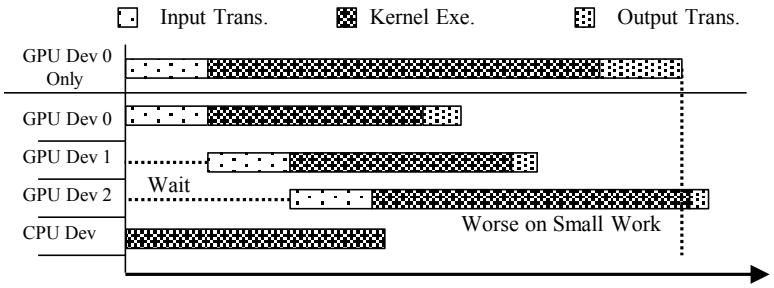
Once the performance model for each device is ready, SKMD makes a decision of how many work-groups should be assigned to each device. The goal of assigning is to minimize the overall execution time by balancing workloads among several devices. This

is an extension of the NP-Hard bin packing problem [18] and a common problem in load balancing parallel systems [45].

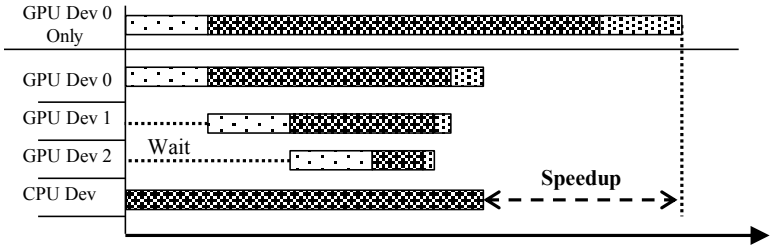
The difference is that it involves more parameters, such as data transfer time between the host and devices, and the cost of merging partial outputs. Most importantly, the performance of devices can vary as the number of work-groups assigned to devices changes. To illustrate, Figure 3.8 shows the relative execution time of the $VectorAdd$ kernel normalized to the time spent executing 32,768 work-groups on three devices. As shown in the figure, execution time does not scale down well as the number of work-groups decreases on discrete GPUs. If the partitioning decision is made without considering transfer cost and performance variance of partitioning, it will be suboptimal or even cause slowdown compared to single-device execution.

To illustrate, the example shown in Figure 3.9 assumes that there are three external GPU devices, each of which has a different performance. If partitioning is done relying only on their maximum performance, partitioned execution may take longer than single device execution for two reasons: 1) serialized data transfer; and 2) decreased performance due to small amount of workload as shown in Figure 3.9(a). In this example, since the CPU device does not have data transfer and GPU device 2 has significant slowdown when it executes a small amount of work, more workload should have been assigned to the CPU device instead of GPU device 2. Figure 3.9(b) shows the ideal case for this example.

Regarding the cost of transfer and performance variance of devices, the partitioning decision becomes a nonlinear integer programming problem. Many heuristics could potentially be used for this problem, however, one limitation is that SKMD performs partitioning at runtime, thus the algorithm must be executed very quickly so as not to overwhelm




**(a)** Partitioning w/o considering transfer time and performance variation




**(b)** Ideal Partitioning

**Figure 3.9:** Comparison of linear partitioning and ideal partitioning

potential benefits from collaborative execution. This restriction prohibits the exact time consuming integer programming solutions [41].

To perform partitioning at runtime, SKMD utilizes a decision tree heuristic [66]. For our system, SKMD uses a top-down induction tree, where the root node is the initial status and all work-groups are assigned to the fastest device based on the estimation. A node in the tree represents a distribution of the work-groups among the devices. A node is branched to its children, and each child differs from the parent in that a fixed number of work-groups are offloaded from the fastest device to another from the parent's partition. For each child, the partitioner estimates the execution time for all devices considering data transfer cost and performance variation of assigned work-groups. The induction is done by a greedy algorithm that chooses a child with the most time difference between offloaded device and offloading device. The partitioner traverses the tree until offloading does not decrease overall execution time.

**Algorithm 1** Performance Variation-Aware Partitioning

```
1: Partition[1..k] = 0                                          ▷ Partition result of k devices
2: BaseDev = argmin_{x∈k}{EstDevExeTime(x, TotalWGs)}
3: PrevExeTime = Min{EstDevExeTime(x, TotalWGs)}
4: Partition[BaseDev] = TotalWGs                                ▷ Assign all groups to base device
5: if Contiguous_Kernel then
6:     MinOffloadCnt = PartitionGranularity
7: else
8:     MinOffloadCnt = Cnt_OffsetsMergeCost(BaseDev)
9: end if
10: TolerateCnt = 0
11: OffloadedCnt = 1
12: while (OffloadedCnt > 0 or TolerateCnt < 10) do
13:     OffloadedCnt = 0
14:     CandidateDevs[1..k].TrialCnt = 0
15:     CandidateDevs[1..k].Diff = MAX_VALUE
16:     for i = 1 to k do
17:         if Partition[i] = 0 then
18:             OffloadingTrial = MinOffloadCnt
19:         else
20:             OffloadingTrial = PartitionGranularity
21:         end if
22:         OffloadingTrial *= 2^TolerateCnt
23:         if OffloadingTrial > Partition[BaseDev] then
24:             continue                                        ▷ Skip trial for this device
25:         end if
26:         Partition[BaseDev] -= OffloadingTrial
27:         Partition[i] += OffloadingTrial
28:         DevsTime[1..k] = EstAllDevsTime(Partition)
29:         EstExeTime = Min{DevsTime[0..k − 1]}
30:         if EstExeTime < PrevExeTime then
31:             CandidateDevs[i].TrialCnt = OffloadingTrial
32:             CandidateDevs[i].Diff = DevsTime[BaseDev] - DevsTime[i]
33:         end if
34:         Partition[BaseDev] += OffloadingTrial
35:         Partition[i] -= OffloadingTrial
36:     end for
37:     OffloadDev = argmax_{x∈k}{CandidateDevs[x].Diff}
38:     OffloadedCnt = CandidateDevs[OffloadDev].OffloadingTrial
39:     Partition[OffloadDev] += OffloadedCnt
40:     Partition[BaseDev] -= OffloadedCnt
41:     if OffloadedCnt > 0 then
42:         TolerateCnt = 0
43:     else
44:         TolerateCnt++
45:     end if
46: end while
47: return Partition
```

In detail, the partitioner loads the *linear regression equation* for performance prediction for each device. The performance equations for each device are computed offline using profile data. By using the performance equation, the partitioner initially estimates the execution time for single device execution for all $k$ devices to identify the fastest device for each kernel. The execution time in the algorithm includes the transfer cost, which can be estimated using buffer size allocated by the OpenCL APIs divided by the bandwidth of PCIe.

Before the partitioner offloads work-groups from the fastest device, it determines the granularity of the number of work-groups to offload ($PartitionGranularity$) based on the total number of work-groups ($TotalWGs$). In our framework, we limited the number of induction steps to 2,048, so $PartitionGranularity$ becomes $Ceil(TotalWGs/2,048)$. One more thing to consider in terms of offloading is the number of minimum work-groups ($MinWGs$) that offsets the merge cost as a result of multiple-device execution. If the kernel is a discontiguous kernel, SKMD must merge output at the end. If the fastest device offloads work-groups to another device for the first time, the time reduced from offloading must be greater than the merge cost. The merge cost can be roughly estimated through the size of output buffer divided by the bandwidth between CPUs and the main memory. Note that the merge cost is computed only for a discontiguous kernel, while for a contiguous kernel, it uses default $PartitionGranularity$ for initial offloading. After initial offloading, since the node in the tree contains enough work-groups to offset the merge cost already, the number of work-groups offloaded to the same device can be increased by $PartitionGranularity$.

Once the partitioner has prepared the necessary values for traversing, it starts to traverse

down the decision tree from the root node by offloading $PartitionGranularity$ work-groups to $k$ devices at each step. At each child node, the partitioner estimates the execution time for all devices using the $EstAllDevTime$ function, which considers data transfer, serialization of PCIe transfer, and performance variation as a result of offloading. After the time estimation of all devices at a child node, the partitioner chooses the maximum value among estimated time, and add the merge cost to compute the overall execution time. Then, the partitioner checks if the overall execution time is reduced compared to the parent node. If a child node takes longer, it is not a candidate for the induction. If the overall time of a child node is reduced, the partitioner marks it as a candidate. For each candidate node, the partitioner computes *Balancing Factor*, which is the difference between the overall execution time in parent node's and the time spent in the device that is offloaded from the parent. For the induction, the partitioner selects the candidate node with the highest *Balancing Factor* among all candidates.

If there is no candidates, the partitioner increases $PartitionGranularity$ temporarily to make sure that the slowdown does not come from the performance variance. If there is still no candidate after additional trials, the partitioner stops traversing and returns the status of child node which has the partitioning results. Algorithm 1 shows a high-level description of partitioning algorithm. `While-Loop` presented at Line 12-46 corresponds to traversing down the decision tree, and `For-Loop` at line 16-36 corresponds to testing children of a node in the tree.

Overall, the time complexity of this algorithm is $O(kN)$ where $k$ is the number of devices, and $N$ is the number of total work-groups. Note that $N$ can be reduced to a constant by limiting the number of induction steps as described above.

### 3.2.5 Limitations

As SKMD partitions workloads at a work-group granularity, *global barriers* or *atomic operations* must be handled carefully.

For global barriers, the execution of work-groups should be ordered at synchronization points in the middle of execution. If work-groups are distributed across multiple devices, work-groups in each device must make sure that the other devices reached the same synchronization point. One approach to handle this case is to break down the entire kernel into multiple kernels at the global synchronization point, similar to loop fission [63], then the split kernels are executed in order.

For atomic operations, the value must be updated with atomicity across all the work-items in the NDRange. However, if work-groups are scattered across multiple devices, each device will end up having their own partial atomic values. If the atomic operations are associative and commutative, intermediate atomic values from different devices can be aggregated later in the host. According to OpenCL specification, there are 11 atomic operations [37]. If an OpenCL compiler can analyze the atomic operations during compilation and detect if they are associative and commutative, OpenCL kernels can still benefit from the idea of SKMD by running special aggregation code in the runtime system.

Also, kernels that have irregular behaviors may not benefit from SKMD system. The main reason is that SKMD predicts the execution time based on a regression model as discussed in Section 3.2.3, which builds a model with NDRange information, the size of array parameters, and value of scalar parameters. However, it does not consider the value of array parameters. If control flows of a kernel are heavily dependent on the value of array

| Device | **Intel Core i7-3770 (Ivy Bridge)** | **NVIDIA GTX 760 (Kepler-GK104)** | **NVIDIA GTX 750 Ti (Maxwell-GM107)** |
|---|---|---|---|
| **# of Cores** | 4 (8 Threads) | 1,152 | 640 |
| **Clock Freq.** | 3.4 GHz | 1.62 GHz | 1.28 GHz |
| **Memory** | 32 GB DDR3 (1866) | 2 GB GDDR 5 | 2 GB GDDR 5 |
| **Peak Perf.** | 435.2 GFlops | 2,258 GFlops | 1,306 GFlops |
| **OpenCL Driver** | Intel SDK 2014 (Enhanced) | NVIDIA CUDA SDK 6.0 | |
| **PCIe** | N/A | 3.0 x8 | |
| **OS** | Ubuntu Linux 12.04 LTS | | |

**Table 3.1:** Experimental setup.

(e.g. breath first search), the execution time is unpredictable only with the size of array. Because SKMD partitions a kernel statically before distributing work-groups, it is difficult to partition this kind of kernels optimally across several devices if the execution time is unpredictable. Applications with these semantics were not handled in this disseration, as SKMD gives up partitioning if a kernel has array-value-dependent control flows.

## 3.3 Evaluation

SKMD was evaluated on a real machine that has three different type computing devices as shown in Table 4.1. Intel Ivy Bridge has an integrated GPU but it does not support OpenCL in unix-based operating systems, so the integrated GPU is not considered as a computing device in our experiments. However, the idea of SKMD framework is not limited to discrete GPUs. SKMD was prototyped using Low-Level Virtual Machine (LLVM) 3.4 [42], on top of a Linux system with NVIDIA driver for GPU execution, and Intel OpenCL driver for the CPU execution.

Every function call to the OpenCL library was hooked by our custom library that leverages SKMD's compilation framework. Inside the framework, we used Clang for the OpenCL frontend, and LLVM 3.4 incorporated with `libclc` extension was used for the PTX backend [48]. However, the PTX backend is used only for *Merge* kernels, while *Partition-Ready* kernels were transformed at the source level and then directly fed into the NVIDIA OpenCL driver.

**Enhanced OpenCL driver for CPUs:** For *Partition-Ready* kernels in CPUs, simply transforming a kernel at the source level and passing it to Intel OpenCL driver may cause significant overhead as discussed in Section 3.2.1. This is mainly because checking code for disabled work-groups will be executed for all work-groups within the innermost loop. To address this problem, we implemented an enhanced OpenCL driver that takes the range of enabled work-group directly so that it can selectively iterate over work-groups. In order to keep the aggressive optimizations made by the Intel driver, we used Intel's offline OpenCL compiler that generates optimized LLVM-IRs, and then we reverse-engineered them to implement the enhanced driver that executes the generated IRs for partial work-groups. As a result of the enhanced driver, the overhead for *Partition-Ready* kernels is removed for the CPU.

**Benchmarks:** For the experiments, a set of benchmarks from the AMD SDK [2] and the NVIDIA SDK [56] were used to evaluate SKMD. Some benchmarks that either do not create enough work-groups regardless of input size, or have atomic operations were excluded. Input sizes for each benchmark for the evaluation are shown in Table 3.2. The applications from the benchmark suite were compiled without any modification. In Table 3.2, Histogram and Reduction were marked as *1st round*, because the OpenCL kernels

| Application | Execution Parameters | Buffer Size | | # of Work-groups | Contiguous Access |
|---|---|---|---|---|---|
| | | Input | Output | | |
| AESEncrypt | 4,096×4,096 BMP image | 48 MB | 48 MB | 16,384 | N |
| AESDecrypt | 4,096×4,096 BMP image | 48 MB | 48 MB | 16,384 | N |
| BinomialOption | 524,288 options | 8 MB | 8 MB | 524,288 | Y |
| Blackscholes | 32 million options | 400 MB | 270 MB | 32,768 | Y |
| BoxMuller | 192 million numbers | 768 MB | 768 MB | 256 | N |
| FDTD3d | 3D dimsize=256, Radius=2 | 68 MB | 68 MB | 256 | N |
| Histogram (1st-round) | 67 million numbers | 256 MB | 2 MB | 2,048 | N |
| MatrixMultiplication | 8,192×8,192 matrices | 512 MB | 256 MB | 65,536 | N |
| MatrixTranspose | 8,192×8,192 matrices | 512 MB | 256 MB | 65,536 | N |
| MedianFilter | 7,680×4,320 PPM image | 128 MB | 128 MB | 518,400 | N |
| MersenneTwister | 192 million numbers | 512 KB | 768 MB | 256 | N |
| Nbody | 524,288 particles | 16 MB | 16 MB | 1,024 | N |
| Reduction (1st-round) | 67 million numbers (float) | 256 MB | 65 KB | 16,384 | N |
| ScanLargeArrays | 8 million numbers (float) | 32 MB | 32 MB | 32,768 | Y |
| SobelFilter | 7,680×4,320 PPM image | 128 MB | 128 MB | 518,400 | N |
| VectorAdd | 50 million numbers (float) | 400 MB | 200 MB | 196,608 | Y |

**Table 3.2:** Benchmark specification. VectorAdd, Blackscholes, BinomialOption, and ScanLargeArrays are classified as contiguous kernels, whereas others are defined as discontiguous kernels.

are used for generating intermediate results, and the host applications finalizes the results later.

To explain Histogram in NVIDIA SDK implementation, each work-group consists of 256 work-items, and each work-item has its own 256 bins in the local memory (65,536 bins per work-group). The entire data is divided by the number of work-groups, and these chunks are split again into 256 parts for work-items. Thus, one work-item will increment its own 256 bins by inspecting one part of data. After incrementing the bins, 256 work-items are in charge of aggregating bins in the local memory. For example, work-item 0 in a work-group aggregates $bin$ 0 of all 256 work-items, and work-item 1 gathers $bin$ 1s, and so on. In this manner, bins for every chunk are gathered for each work-group, and these aggregations are done for all work-groups in the OpenCL kernel, which is referred to as *Histogram (1st-round)* in Table 3.2. With the result from the OpenCL kernel, the final aggregation is done in the second round by the host application .

Similarily, Reduction from NVIDIA SDK is implemented without atomic operations. Instead, work-items in a work-group reduce two numbers at the first step, and reach the local barrier. After that, half of them reduce the reduced numbers again until only one work-item remains. As a result, the last work-item will generate the reduced number for the entire work-group. These steps are done for all work-groups in the OpenCL kernel, which is also referred to as *Reduction (1st-round)* in Table 3.2. The reduced numbers for all work-groups are finally reduced in the second round in the host application.

**Methodology:** Before the real execution, offline profiling is performed to collect performance data for each benchmark. For offline profiling, it is important to collect enough data to model linear regression accurately. If the model is computed with too few data, the error rate can be high especially when execution parameters of the real execution differ much from profile-run. For this reason, SKMD requires an application to use *profiling mode* if there is not enough profile data. With profiling mode, SKMD launches the OpenCL kernel on each device several times varying the number of work-groups. Because it is important to catch the point that linearity appears as discussed in Section 3.2.3, SKMD increases the number of work-groups by four (*finer granularity*) until it reaches 16, and then increases the granularity as the number of work-groups grows. Once profile data is collected, SKMD performs the linear regression analysis as discussed in Section 3.2.3.

For the dynamic overheads, we did not consider the cost of kernel analysis and transformation because they can be done during offline profiling, but we measured the partitioning overhead, which is done in the real execution. To reduce the overhead, we forced the height of decision tree used in partitioning algorithm to be within 1,024 steps. In other words, for kernels that launch more than 1,024 work-groups, SKMD increases partitioning granular-

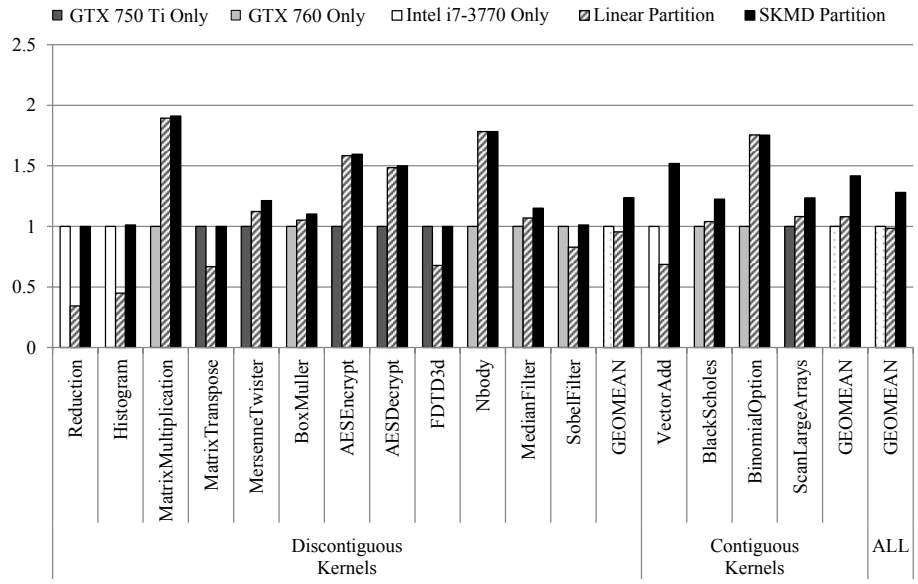
ity. As a result, $1,024 \times 2$ (the number of offloading devices) estimations are done in the worst case. As 2,048 time estimations can be done with less than 100K instructions, the overhead for the partitioning algorithm is observed as less than $1ms$ which is negligible for all benchmarks.

We measured wall clock execution time including the transfer time between host and GPU devices, kernel execution time, and data merging cost in case of discontiguous access kernels. Because the CPU resource is shared with the operating system or other applications, the execution time on the CPU device can vary. Therefore, we ran 1,000 times for each benchmark and selected 100 sets of results that have the least CPU execution time, and used the average of those 100 results for the final result.

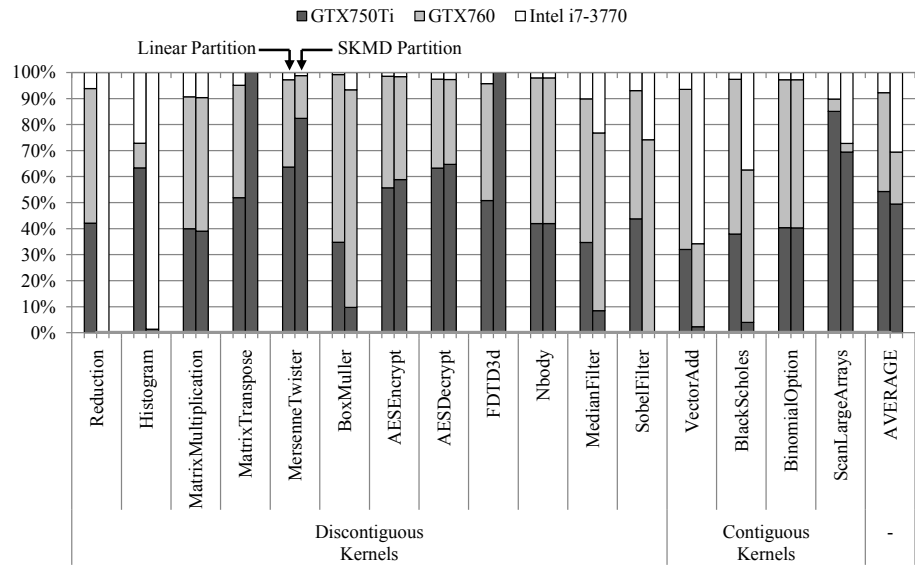
### 3.3.1 Results and Analysis

Figure 3.10(a) shows speedup of SKMD compared to the fastest single device execution and the linear partitioning execution, which is similar to prior approaches [49, 38]. In the linear partitioning, the number of work-groups assigned to each device are proportional to the predicted performance without consideration of the transfer cost. The baseline is different for each benchmark based on its characteristics. For each benchmark, we ran them on all devices and chose the fastest device (including the transfer cost) as the baseline. Three benchmarks, Reduction, Histogram, and VectorAdd used CPU-only execution as their baseline because data transfer cost overwhelms the benefits of executing on GPUs, as they are extremely memory-bound kernels.

As illustrated in Figure 3.10(a), SKMD performs 28% faster than the single device execution on average as it considers the transfer cost and performance variation of each




**(a)** Speedup normalized to the fastest single device execution




**(b)** Work-group distribution

**Figure 3.10:** Speedup and work-group distribution. Each benchmark has different baseline (a), as the fastest device differ by kernels. The fastest device is determined with regard to the execution time and data transfer cost.

device during partitioning. An important point from this result is that the linear partitioning causes slowdown on memory-bound kernels compared to the single device execution. This is mainly because it does not take the transfer cost into account during the partitioning although collaborative execution is not favorable due to the transfer cost.

To illustrate how SKMD partitions work-groups across different devices, Figure 3.10(b) shows the work distribution of all applications. On average, SKMD partitioning, which considers the transfer cost, assigns more workload to the CPU than the linear partitioning. The linear partitioning makes a bad decision for memory-bound applications by assigning less workload to the CPU, although considerable amount of time is spent on transferring the data. On the other hand, SKMD partitioning assigns more workload to the CPU, as the CPU can work more while the data is being transferred to the external GPUs.

**Reduction, Histogram, VectorAdd:** Reduction, Histogram, and VectorAdd are extremely memory-bound kernels so SKMD assigns most of the work to the CPU device. The difference is that Reduction and Histogram are recognized as a discontiguous kernel, so the host program must transfer the entire input to the external GPU device which discourages collaborative execution due to expensive cost of data transfer. On the other hand, VectorAdd which is a contiguous kernel does not require the entire input for the partial execution, so there is still a chance for the CPU to offload work-groups to the GPU devices.

**MatrixMultiplication, AESEncrypt/Decrypt, Nbody, BinomialOption:** These benchmarks are compute-bound kernels where a significant amount of time is spent on computation, not memory accesses. For these benchmarks, the portion of the workload assigned to the GPUs are higher than the CPU because of its massively data-parallel structure. As mentioned earlier, because GTX 760 is a high performance GPU, it executes more work-groups than GTX 750 Ti.

**MatrixTranspose:** MatrixTranspose is a memory-bound kernel but SKMD assigns all of work to GTX 750 Ti despite the expensive cost of data transfer. This is due to the very low performance of the CPU. Since the OpenCL implementation targets GPUs, each work-

group has a local memory to store input in order to avoid un-coalesced global memory accesses among work-items. However, for the CPU execution, having local memory does not benefit from coalesced memory access, but rather produces unnecessary overhead of copying data to additional space. This overhead may not be significant for other benchmarks, but for MatrixTranspose in the CPU, copying input to the scratchpad is another equal amount of work compared to the naive transpose.

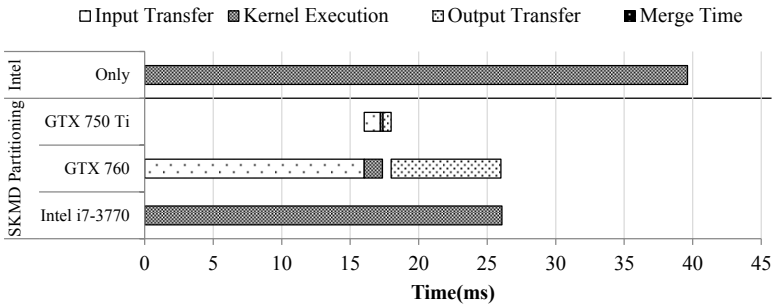
**ScanLargeArrays:** ScanLargeArrays has large memory foot-prints with contiguous memory access patterns. Similar to VectorAdd, it does not have to transfer the entire data back and forth between the CPU and the GPUs. However, it has more computations than VectorAdd that are faster on GPUs, so larger portion of workload is offloaded to the GPUs than VectorAdd.

**Other benchmarks** have considerable amount of computations and large memory foot-prints with discontiguous access patterns. In this case, both compute and transfer costs are proportional to the size of data, so the data transfer time could offset the reduced computation time from the collaborative execution. As a result, the speedup from the collaborative execution is relatively low as shown in Figure 3.10.

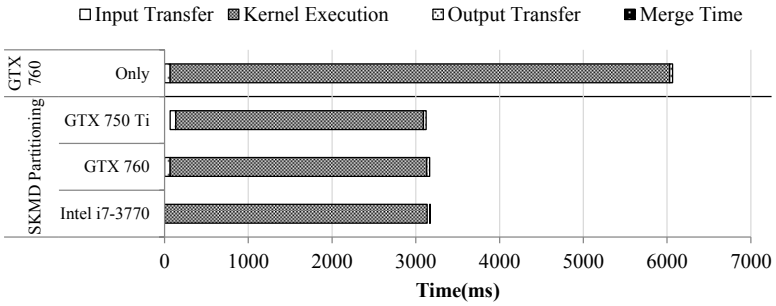
### 3.3.2 Execution Time Break Down

In this section, we show how SKMD transfers data between the CPU and the GPUs, and assigns work-groups to different devices. Figure 3.11 shows the execution time break down of three sample applications: Vector Add, Matrix Multiplication, and Histogram.

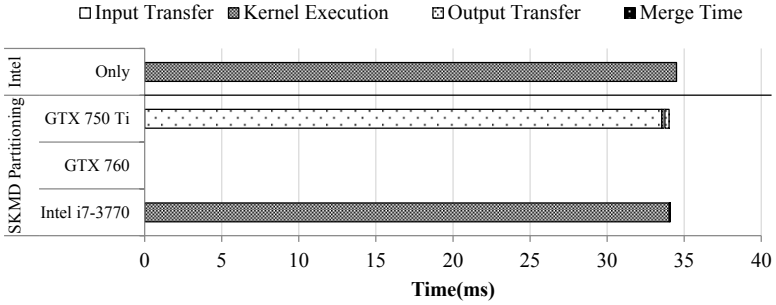
For *VectorAdd*, CPU-only is the baseline because it is an extremely memory-bound kernel. As shown in Figure 3.11(a), SKMD starts the execution on the CPU while trans-




**(a)** VectorAdd




**(b)** MatrixMultiplication




**(c)** Histogram

**Figure 3.11:** Break down of the execution time on each device. The bars on the top is the baseline, which is the fastest single-device execution. SKMD considers the transfer cost, and offloads work-groups in order to balance the workload among the three devices.

ferring huge data to GTX 760 in background. As soon as the data transfer is finished, SKMD launches the kernel on GTX 760, and at the same time, it transfers data needed for the remaining work-groups to GTX 750 Ti and then launches the kernel. The transfer time for GTX 750 Ti is smaller because it is a less powerful GPU for VectorAdd so the size of data assigned to it is smaller. Since VectorAdd has contiguous memory accesses, there is no need to merge the data. After both kernels are done, the buffer manager transfers the

data from the GPUs and simply puts them in the final result. As shown in the figure, the CPU finishes execution almost at the same time as the GPUs finish their data transfer as a result of accurate partitioning.

The baseline of *MatrixMultiplication* is GTX 760-only as shown in Figure 3.11(b). Since Matrix Multiplication takes much more time in computation than VectorAdd, the impact of transferring time is less for this benchmark. However, SKMD transfers the entire input and output back and forth between the host and GPU devices as it is classified as a discontiguous kernel. Similar to VectorAdd benchmark, GTX 760 starts execution first followed by GTX 750 Ti but finishes later than GTX 750 Ti because it has more work-groups to execute due to its higher performance. At the end, the CPU merges all partial results to generate the final output by launching the merge kernel.

*Histogram* shows different behavior from the other cases. The baseline for Histogram is CPU-only execution because it has large input, which incurs large transfer cost for the GPU execution. In terms of execution performance, GTX 750 Ti outperforms than GTX 760 for Histogram as shown in Figure 3.10(b)-*Linear Partition*, which partitions kernels linear to the performance. Therefore, GTX 750 Ti has higher priority for offloading. Also, the output size is much smaller than input size as shown in Table 3.2.

Histogram is categorized as a discontiguous kernel, so SKMD still has to transfer the entire input to the external GPU devices. As shown in Figure 3.11(c), SKMD does not assign any work-groups to GTX 760 after assigning some work-groups to GTX 750 Ti, because serialized input data transfer to GTX 760 would break balanced execution among three devices. As the output size is small, the overhead of merging kernel is negligible for this benchmark.

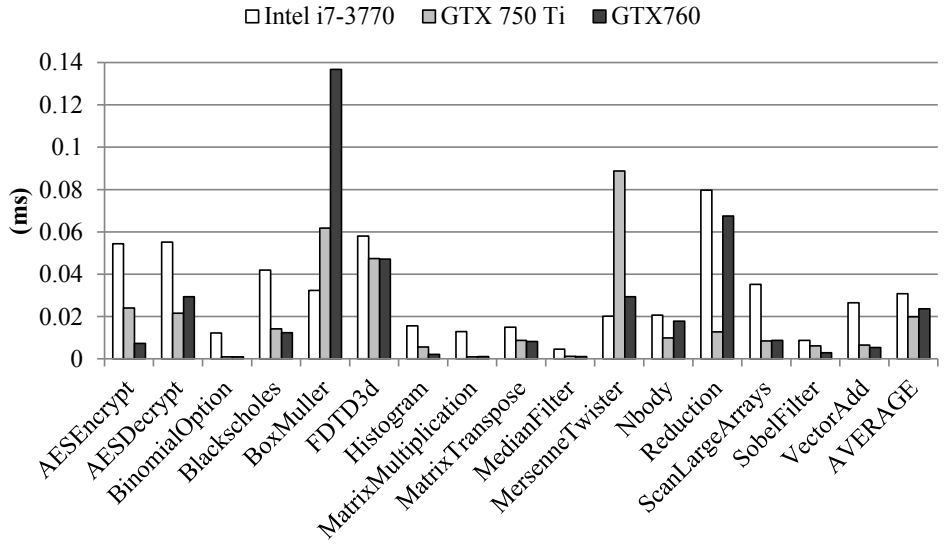
| Application | Profile Parameters | | Real Parameters (same as Table 3.2) |
|---|---|---|---|
| | Profile 1 | Profile 2 | |
| AESEncrypt | 1,024×1,024 BMP image | 2,048×2,048 BMP image | 4,096×4,096 BMP image |
| AESDecrypt | 1,024×1,024 BMP image | 2,048×2,048 BMP image | 4,096×4,096 BMP image |
| BinomialOption | 16,384 options | 65,536 options | 524,288 options |
| Blackscholes | 1 million options | 8 million options | 32 million options |
| BoxMuller | 8 million numbers | 32 million options | 192 million options |
| FDTD3d | 3D dimsize=64, Radius=1 | 3D dimsize=128, Radius=2 | 3D dimsize=256, Radius=2 |
| Histogram (1st-round) | 4 million numbers | 16 million numbers | 67 million numbers |
| MatrixMultiplication | 1,024×1,024 matrices | 2,048×2,048 matrices | 8,192×8,192 matrices |
| MatrixTranspose | 1,024×1,024 matrices | 2,048×2,048 matrices | 8,192×8,192 matrices |
| MedianFilter | 1,920×1,080 PPM image | 3,840×2,160 PPM image | 7,680×4,320 PPM image |
| MersenneTwister | 8 million numbers | 64 million numbers | 192 million numbers |
| Nbody | 65,536 particles | 131,072 particles | 524,288 particles |
| Reduction (1st-round) | 8 million numbers | 34 million numbers | 67 million numbers |
| ScanLargeArrays | 500,000 numbers | 1 million numbers | 8 million numbers |
| SobelFilter | 1,920×1,080 PPM image | 3,840×2,160 PPM image | 7,680×4,320 PPM image |
| VectorAdd | 8 million numbers (float) | 16 million numbers | 50 million numbers |

**Table 3.3:** Profile execution parameters and real execution parameters for evaluating performance prediction accuracy. For each profile, 16 profile data was collected varying the number of work-groups.

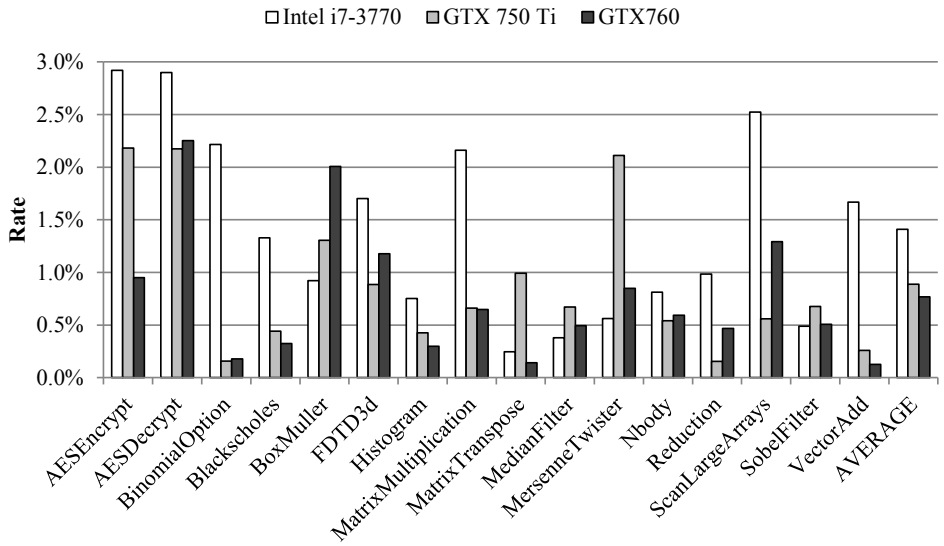
### 3.3.3 Performance Prediction Accuracy

To evaluate the accuracy of performance prediction on the CPU and the GPUs, we profiled the applications with two sets of execution parameters as shown in *Profile Parameters* of Table 3.3. With the profiled data from two sets of execution parameters, SKMD performed a linear regression analysis to get the coefficients. After the computation of the coefficients, we ran the applications with the real execution parameters as shown in *Real Parameters Table 3.3*. For the real execution, we randomly picked the number of partial work-groups 128 times, and compared the real execution time with the predicted execution time.

Figure 3.12(a) shows the L2-Norm errors between predicted time and execution time, and Figure 3.12(b) shows the average error rate for each benchmark. The L2-Norm error means Euclidean distance between two time vectors, thus it represents the amount of error




**(a)** L2-Norm




**(b)** Average Error Rate

**Figure 3.12:** Performance prediction accuracy. L2-Norm error (a) shows Euclidean distance between the real execution time and the predicted execution time in milliseconds. Average error rate (b) shows the average percentage of errors in predictions.

in *milliseconds*, while the average error rate shows the difference over the real execution time. For all benchmarks, high error ratios were observed when SKMD predicts the execution time with a very few number of work-groups. This is mainly because the execution time is very short with a few number of work-groups. As a result, even small error values can result in high error ratios. For example, if SKMD predicted the time as 0.011 ms, but

the real execution took 0.01 ms, then the error ratio becomes 10% in spite of only 0.001 ms of misprediction. Considering that the execution time for the execution parameters shown in Table 3.3 takes more than 10 ms for all benchmarks, the errors in prediction time are negligible since the error remains under 0.1 ms in most cases as shown in Figure 3.12(a).

## 3.4 Related Work

A significant focus has been on the execution of data-parallel applications on CPUs. [46] examined several data parallel applications to show that CPUs can have comparable performance to GPUs, if it takes full advantage of multi-cores with single instruction multiple data (SIMD) units. There has also been some work on efficient execution of OpenCL/CUDA applications on CPUs. [75] proposed a source-to-source compiler that translates a CUDA program into a standard C program using loop-fission technique to eliminate *synchronization*. Similarly, [12] developed the Ocelot, a runtime system that dynamically transforms OpenCL/CUDA kernels for CPU execution. [24] also performed a similar study, but approached in a light-weight thread (LWT) execution model. In a similar fashion, [33] has focused on more efficient execution of OpenCL applications using whole-function vectorization. All of these prior works are focusing on performance improvement on CPUs to show CPUs can perform as good as GPUs for some applications but none of them deals with collaborative execution with GPUs.

Performance modeling of GPUs for a certain set of applications have been studied for several years [31, 25]. [25] proposed an analytical model for a GPU architecture with awareness of memory-level and thread-level parallelism. However, this model relies only

on static information of GPU architectures and applications, such as the number of registers, the size of memory on the device and those numbers required by the application. Also, this study was based on relatively simple GPU architectures compared to contemporary GPU architectures, which is much harder to predict the performance statically. Meanwhile, Jia et al. proposed a GPU performance prediction method based on a linear regression model, but the work used the prediction model for GPU space exploration varying GPU architectures. On the other hand, our work described the linear regression model that fits for various execution parameters in order to optimize the performance.

Dynamic decision of execution on heterogeneous systems with CPUs and GPUs has been studied in the past [13, 47, 49, 7, 36]. Harmony [13] reasons about the whole program by building a data dependency graph and then scheduling independent kernels to run in parallel. However our approach is different from prior works in that our system is working on finer granularity (work-groups) rather than function or task level. MERGE [47] is a predicate dispatch-based library system for managing map-reduction applications on heterogeneous systems. [49] proposed the Qilin that automatically partitions threads to one CPUs and one GPUs by providing new APIs that abstract away two different programming models, Intel Thread Building Block and CUDA. [38] also proposed a framework that distributes workload of an OpenCL kernel to multiple equivalent GPUs for specific types of data-parallel kernels. Delite [7] and is a compilation framework that takes a program written in OptiML and converts it into C++/CUDA program. Then runtime system manages execution between CPU and GPUs. While this work is limited to domain-specific languages, SKMD provides more generality as it supports a variety of OpenCL applications. The PEPPHER proposed by [36] improved the performance by tuning the execution

strategy on a heterogeneous system based on their performance prediction model. Our approach differs from prior work in that our system supports more than two different types of devices and considers data transfer cost and performance variance during partitioning. Also, our approach does not rely on additional programming extensions or APIs.

In the mean time, a series of works have been done for virtualizing GPU resources [67, 34, 68, 79, 76, 43]. PTask [67] provides APIs that work with OS abstraction layers to manage compute tasks on GPUs by using a data-ow programming model. Dandelion [68] also proposes a compiler/runtime framework that takes C# sources with newer APIs, and converts them to CUDA code, and runtime manages execution between CPUs and GPUs using PTask [67]. [34] proposed Gdev that manages GPU resources in the OS level, so GPUs can be treated as first class computing resources in multi-tasking systems. SKMD is different from these prior works as SKMD does not require programmers to use additional APIs or language extensions, but it is transparent to OpenCL applications, which can be further optimize parallel kernels by utilizing local memories. [79, 76] also proposed virtualization layers for GPUs which take over the control of GPU memory space from applications without changing APIs. Through these techniques, GPUs can access the data in the host directly on page faults. Similarly, NVIDIA recently offered Unified Virtual Address to provide abstract view of unified memory system in separate physical memory [58]. Main purpose of this idea is removing the burden of managing multiple memory spaces [35], but it still leaves work distribution between devices as programmer's responsibility. On the other hand, SKMD focuses on balancing workloads across multiple computing devices and transfers the entire working-sets at once in order to avoid high overhead from frequent PCIe bus transactions for page fault handling.

## 3.5 Conclusion

In this chapter, we presented SKMD, a framework that transparently manages collaborative execution on CPUs and GPUs of a single OpenCL kernel. SKMD leverages assigning a subset of data-parallel workload over multiple CPUs and GPUs so as to increase overall performance. As a part of the exploration, this chapter introduced several techniques that transparently enable a kernel to work on a partial workload and efficiently merge results from separate devices. In order to distribute a balanced workload, this chapter also presented an accurate performance prediction model and an efficient methodology for balancing workload between CPUs and GPUs being aware of data transfer cost and performance variance depending on the type of device. By experimenting with OpenCL applications on a real hardware, we showed that SKMD yields a geometric means of 28% speedup on a machine with one CPU and two different GPUs as compared to the fastest device-only execution.

# CHAPTER IV

# VAST: Virtualizing Address Space for Throughput Processors

## 4.1 Introduction

Graphics processing units (GPUs) have emerged as the computational workhorse of throughput oriented applications because of their low cost, high performance, and wide availability [8]. Modern GPUs achieve several tera floating point operations (FLOPS) of peak performance while costing a few hundred dollars. With CUDA or OpenCL, programmers can develop data parallel kernels for GPUs that achieve speedups of 4-100x over traditional processors (CPUs) [69].

As a result, software in various domains has been connverted to exploit GPU's computing resources, many of which work on large data sets. For example, galaxy formation simulation in space research computes physical forces among hundreds of million particles [74]; economical analysis combines thousands of factors in tens of dimensions and then computes on it [15]; and numerous fields perform data minings from a huge amount of data [80],

However, discrete GPUs have a critical limitation: the entire data to process *must* reside in GPU memory before execution. When the data size exceeds the physical memory of the GPU, the application should be developed to let the GPU to access the host memory directly through the peripheral component interconnect express (PCIe) bus for every access, which has much lower bandwidth and higher latency than GPU's memory. Otherwise, the execution must be fallen back to the CPU which supports nearly arbitrary data sizes through virtual addressing. Smart programmers can overcome this restriction by applying a *divide-and-conquer* approach. In this case, the programmer explicitly divides the workload into smaller chunks and executes a series of kernels each processing a subset of the data. This approach is not a panacea, however. Even in the simple case where the kernel operates on contiguous data chunks, explicit data management is tedious, requiring manual buffer allocations and data transfer to/from the host. For non-contiguous data, divide-and-conquer is more complex. Lastly, the size of chunks may change across GPUs with differing amounts of physical memory.

A natural question is why do GPUs not support virtual addressing to eliminate this problem as CPUs have done for many decades [71, 5]. Through a combination of hardware (e.g., translation look-aside buffers) and operating system support (e.g., page tables), CPUs provide the appearance of a nearly infinite address space to facilitate processing large data sets. However, virtualizing the address space of discrete GPUs is difficult for the following reasons:

- Discrete GPUs have separate memories that use different address spaces, and each transaction between the host and GPU is expensive.

- GPUs do not interact with the operating systems for memory management, but instead directly access their physical memory.

- Execution on GPUs is non-preemptive, therefore all data must be present in the physical memory before execution.

The combination of these factors makes it difficult to execute kernels on GPUs whose total memory footprint exceeds the physical memory size on the GPU without programmer intervention. As the size of data to process keeps increasing, this limitation will become more significant.

To tackle these challenges, we present *Virtual Address Space for Throughput processors* (VAST), a run-time software system that provides the programmer with the illusion of a virtual memory space for commodity OpenCL devices. VAST transparently divides and transfers working sets based on the available physical memory space on the target GPU. To virtualize the memory space, VAST adopts a *look-ahead page table (LPT)*, a new type of page table that contains a list of virtual pages that will be accessed by specific ranges of the OpenCL workload. LPT differs from conventional page tables used in operating systems in that the LPT is filled up before the pages are actually accessed. With LPT, VAST decomposes the working set into individual *page frames* for execution of the partial workload. Page frames are packed into a contiguous buffer (*frame buffer*) that resides in the GPU's physical memory and LPT represents the mapping of an OpenCL buffer from the CPU's virtual space into the GPU's physical space (the frame buffer). Instead of transferring the entire data to the GPU, VAST transfers the LPT and frame buffer for each partial workload. At the same time, VAST transforms the kernel to access memory through

the LPT (e.g., software address translation).

The VAST runtime system significantly improves GPU portability as it can utilize any GPU for larger sized workloads. The challenges of VAST are four fold: dividing an arbitrary workload based on the available physical memory size, efficiently generating the LPT and frame buffer, transforming the kernel to use the relocated and packed data, and avoiding replicated transfers due to reuse of data across partial workloads. To address these issues, this dissertation makes following contributions:

- A code transformation methodology that quickly inspects memory access locations of an OpenCL kernel in order to generate LPTs.
- A novel technique that partitions an OpenCL workload into partial workloads based on the physical memory constraints of a GPU and packs the corresponding data into a frame buffer using LPTs.
- Kernel transformation techniques to access data out of the frame buffer using LPTs.
- A technique to avoid replicated data transfers to the GPU.
- A comprehensive performance evaluation of VAST on real hardware consisting of an Intel Core i7 3770 CPU and an NVIDIA GTX 750 Ti GPU.

The rest of the chapter is organized as follows. Section 4.2 discusses the OpenCL execution model and opportunities for virtualizing GPU memory space. Section 4.3 explains the overview of VAST, and then the implementation and optimizations are discussed in Section 4.4 and Section 4.5 respectively. The experimental results of using VAST for various OpenCL applications are presented in Section 4.6. Section 4.7 discusses the related

```
__kernel void
matrixMul(__global float* C,
          __global float* A, __global float* B,
          int wA, int wB,
          int range_from, int range_to)
{
    int gid_x   = get_group_id(0);
    int gid_y   = get_group_id(1);
    int size_x  = get_num_groups(0);
    int flat_id = gid_x + gid_y * size_x;
    // check whether to execute
    if (flat_id < range_from || flat_id > range_to)
      return;

    int idx = get_global_id(0);
    int idy = get_global_id(1);

    float value = 0;
    for (int k = 0; k < wA; ++k) {
      value += A[idy * wA + k] * B[k * wB + idx];
    }
    C[idy * wA + idx] = value;
}
```

**Figure 4.1:** The code transformation for partial execution of an OpenCL kernel. The kernel takes two additional arguments for the work-group range to execute, and grey backgrounded code is also inserted at the beginning of the kernel to check if the work-group is to be executed. The work-groups out of the range will terminate the execution immediately.

work in this area. And finally, we conclude in Section 4.8.

## 4.2 Motivation

As discussed in Chapter II, an OpenCL kernel can be executed in parallel at a work-group granularity without concern of the execution order. Through the code transformation, the OpenCL host program can control the number of executed work-groups as shown in Figure 4.1. By inserting checking code at the beginning of the kernel, every work-item checks if the work-group it belongs to is supposed to execute. If it should not, the work-group terminates the execution immediately and the GPU will schedule it out. In this way, we can limit the total amount of memory accessed by those work-groups to a predefined amount (the GPU's physical memory size). This memory is reorganized into page frames

and packed into a frame buffer by VAST. A look-ahead page table (LPT) is then used to map between the original CPU address to the page frame address for the GPU. We refer to this model as *partial execution* and the goal of VAST is to automate the decomposition and to restrict the kernel as necessary to realize software-managed paging.

## 4.3 VAST System Overview

VAST is an abstraction layer located between an application and the OpenCL library. The VAST layer overloads all OpenCL APIs including device-querying functions. By overloading device-querying APIs, VAST provides the application with the illusion of a virtual GPU device that has very large amount of memory. With the virtual device, programmers can allocate buffers as much as they need without concern for the physical memory size.

In order to virtualize address space of OpenCL kernels, actual data must be rearranged into page frames with the look-ahead page table (LPT), which is filled with a list of pointers to the corresponding page frames that will be accessed during the execution. In addition, the OpenCL kernel must access data through the LPT and frame buffer (address translation), as similar to the conventional program accessing data through TLBs and the operating system's page table. After kernel execution, the output buffer for the LPT and page frames must be recovered to original memory space on the host.

The rest of this section describes execution flow of VAST system and illustrates timeline of VAST execution.

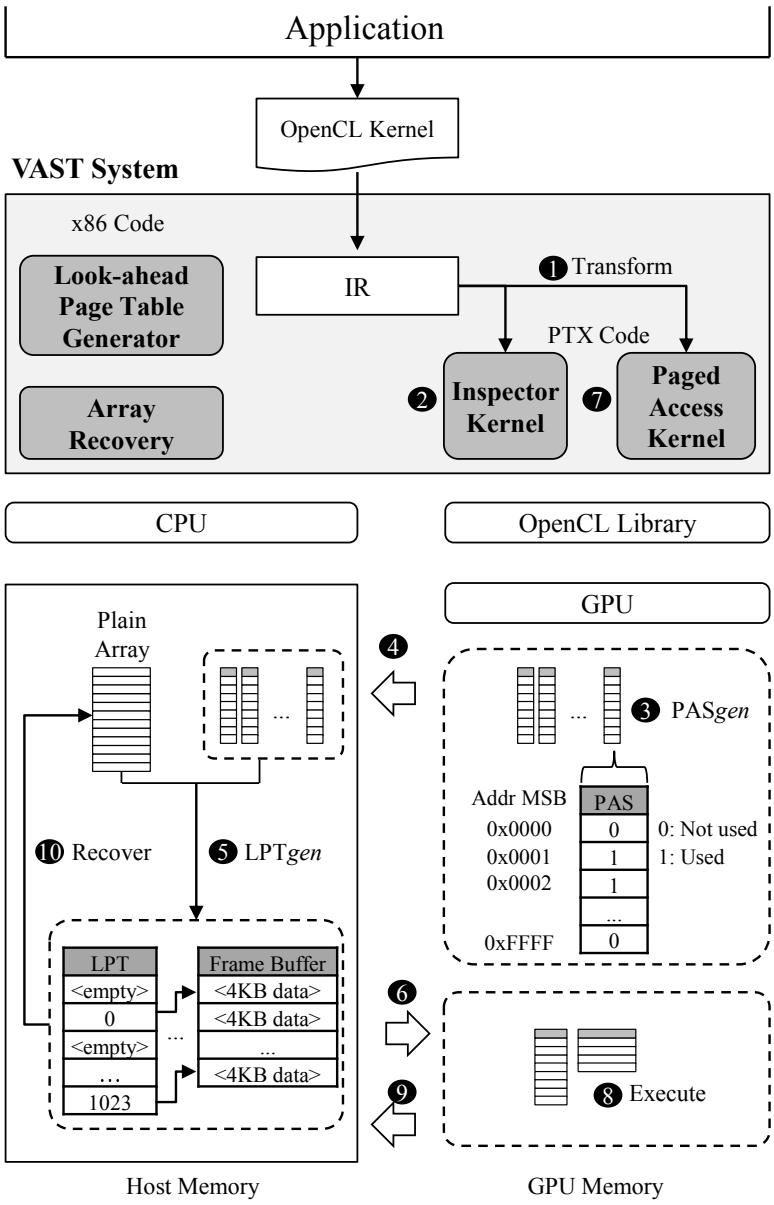



**Figure 4.2:** The VAST system located between applications and OpenCL library. VAST takes an OpenCL kernel and transforms it into the inspector kernel and the paged access kernel. At kernel launch, the GPU generates PASs ($PASgen$) by launching the inspector kernel, then transfers them to the host to create LPT and frame buffer ($LPTgen$). Next, LPT and frame buffer are transferred to the GPU in order to execute the paged access kernel.

### 4.3.1 VAST System Execution Flow

Figure 4.2 illustrates sequences of VAST system operations. First, VAST compiles an OpenCL kernel into intermediate representations (IRs), and then generates a GPU binary

(PTX) for the ***Inspector Kernel***, and the ***Paged Access Kernel*** as shown in Figure 4.2 ❶.

After a compilation request, the application will request the kernel launch. On this request, VAST first launches the *Inspector Kernel* ❷ that only inspects usage of global arguments (arguments with `__global` keyword) and fills up the *Page Accessed Sets* (PASs) ❸. PAS contains boolean values that represent whether a work-group has accessed each page. Note that each work-group has its own PAS for each global argument of the kernel, and the size of PAS is fixed as $Ceil(\frac{AllocSize}{PageSize})$ where $AllocSize$ is actual OpenCL buffer allocation size from the application. For example, 4GB of OpenCL buffer will require 1MB of PAS with 4KB pages. In order to reduce the number of PASs, one PAS can be shared among several work-groups. In this case, one PAS represents the pages accessed by a subset of work-groups.

Next, VAST transfers PASs from the GPU to the host ❹, and then the host fills up the LPT and frame buffers using the PASs until the size of frame buffer reaches the available GPU memory size ❺. After that, VAST allocates the actual buffers for the LPT and frame buffer on the GPU, and transfers them to the GPU device ❻. At this point, VAST knows how many work-groups should be executed as it has generated the LPT and frame buffer for the specific range of work-groups. In order to execute specific range of work-groups of the kernel, the *Paged Access Kernel* also takes additional parameters for the ranges of work-groups to execute as discussed in Section 4.4.2 of Chapter III. Once VAST finishes the transfer, it launches the *Paged Access Kernel* ❼, which accesses the memory through the LPT.

During execution, the frame buffer in the device memory will be updated ❽, and after the execution, VAST will transfer them back to the host memory space ❾. Finally, VAST

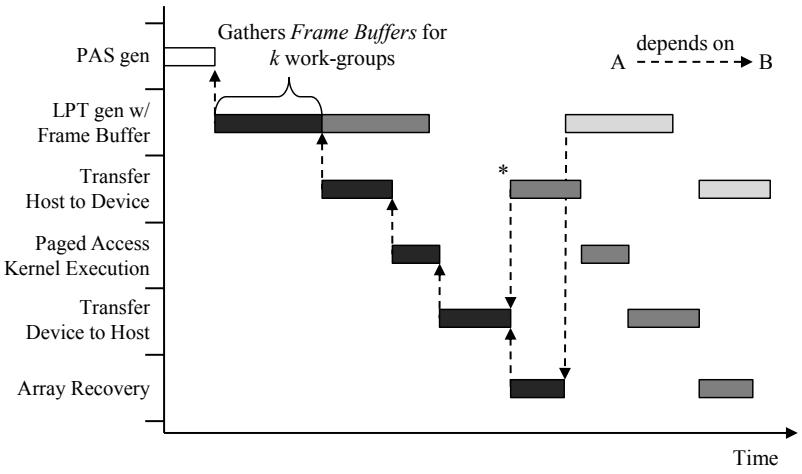



**Figure 4.3:** Execution timeline for VAST system. Only PAS generation and the first LPT generation cost is exposed. Other LPT generations and array recoveries are overlapped from data transfer and kernel execution. With double buffering, the second LPT generation starts immediately after the first LPT generation.

recovers arrays using the LPT and modified frame buffer ❿, and repeats the steps from LPT generation ❺ to recovery ❿ until all work-groups finish their execution.

### 4.3.2 VAST Execution Timeline

Figure 4.3 visualizes the timeline of VAST execution. As shown, the cost of generating LPTs and frame buffers can be hidden by overlapping them with data transfer and kernel execution. Thus, the only exposed cost is PAS generation and the first LPT generation. Mind that VAST allocates LPTs and frame buffers twice on the host for double buffering. With double buffering, the next LPT generation can proceed right after the previous LPT generation, as shown in Figure 4.3.

One important point from Figure 4.3 is that transferring the second working set to the GPU (marked as $^*$) starts after retrieving the first partial result back to the host, not after the kernel execution. The main reason is that some pages written in the first partial execution can also be written by the work-groups in the second partial execution. Similar to data for-

warding in a CPU pipeline datapath, VAST forwards only written pages from the previous execution to the next execution. Details of forwarding shared pages will be discussed in Section 4.4.4.

Another feature of VAST is that it avoids duplicated data transfer. To illustrate, if page frames of a global argument during the partial execution are identical to that of the next execution, VAST skips LPT and frame buffer generation as well as buffer transfer for the next execution.

## 4.4 Implementation

This section discusses the implementation of VAST system including design of PAS, kernel transformations, and how to create LPT and frame buffers using PASs. Also, the ways to handle shared pages and to avoid duplicated data transfers are discussed.

### 4.4.1 The Design of Page Accessed Set

As described in Section 4.3.1, the first step for VAST is launching the inspector kernel on the GPU in order to generate *Page Accessed Sets* (PASs). PAS represents a list of pages accessed by a set of work-groups during execution. Therefore, each entry of PAS contains a boolean value as shown in Figure 4.4. Later, these PASs will be used on the host to generate *Look-ahead Page Table* (LPT), which contains pointers to the page frames.

The inspector kernel is able to execute on the GPU without transferring the working set because most OpenCL kernels use a combination of work-item index and scalar variables as the index of global array access. Upon this property, VAST passes only scalar and __-

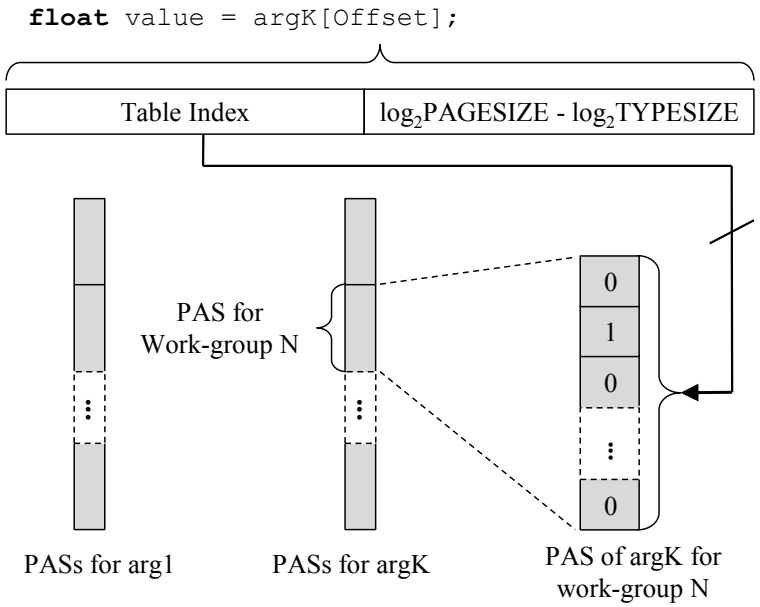



**Figure 4.4:** The design of Page Accessed Set (PAS). Each work-group has its own PAS for each global argument. Each entry of PAS has a boolean value that represents whether corresponding page has been accessed by the work-group.

`constant` arguments to the inspector kernel along with the NDRange information (size of work-groups and work-items) for PAS generation. If global array access depends on the value of a global argument, (e.g. indirect memory access), the inspector code is leveraged on the host, and the host generates PASs directly.

As shown in Figure 4.4, VAST makes use of single level paging because it minimizes both PAS generation time for the inspector kernel and address translation time for the paged access kernel. As a result, with 4KB pages, 4 GB of data can be fit into 1 MB of PAS if each entry uses 1 byte to store a boolean value. However, if PASs are maintained per work-group per global argument, the size of overall PASs can become significant if the kernel is launched with a large number of work-groups. For this reason, VAST makes a set of consecutive work-groups to share one PAS depending on the number of work-groups. The

number of work-groups per PAS is determined statically using the equation below.

$$WORKGROUP\_PER\_PAS = Ceil(\frac{\frac{MEMSIZE}{32}}{\frac{AllocSize}{TOTAL\_WORKGROUP}}) \quad (4.1)$$

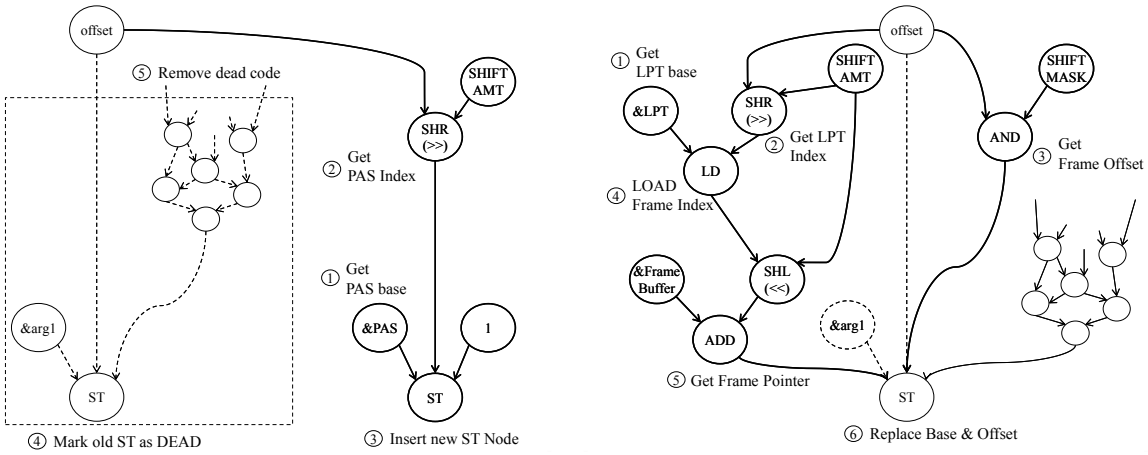
The assumption behind this equation is that one work-group accesses the space of $\frac{AllocSize}{TOTAL\_WORKGROUP}$. With this assumption, VAST computes the number of work-groups not to exceed $\frac{MEMSIZE}{32}$ accesses. This is a rough estimation, and if it appears that the number is too small or large, VAST adjust the number dynamically.

When several work-groups try to modify the shared PAS, atomic operation is not necessary because according to NVIDIA's programming guide [58], if a non-atomic instruction executed by more than one thread writes to the same location in global memory, only one thread performs a write and which thread does it is undefined. Thus, it is safe to share one PAS among several work-groups and work-items because every work-item will try to write the same boolean value (`TRUE`).

### 4.4.2 OpenCL Kernel Transformation

As shown in Figure 4.2 - ❶, VAST transforms each OpenCL kernel into an inspector kernel and paged access kernel. Both kernel transformations only focus on usages of global arguments, but the difference is that the inspector kernel replace entire memory operations with the new stores, while the paged access kernel replaces only the bases and offsets of memory operations as shown in Figure 4.5.

In detail, the first step for the inspector kernel transformation is to inline function calls in the kernel in order to avoid expensive inter-procedural analysis. In general, every function call can be inlined since the OpenCL programming model prohibits recursive calls as it




**(a)** DFG for the inspector kernel **(b)** DFG for the paged access kernel

**Figure 4.5:** Data flow graphs for the kernel transformation. In the inspector kernel (a), all computational code are removed by dead code elimination. In paged access kernel (b), the base and the offset are replaced with new nodes for address translation.

adopts SIMT execution model that does not allow threads to execute different instructions at a time [44]. Next, VAST adds arguments for PASs to the kernel, each of which corresponds to the respective global arguments. Besides, the arguments for each PAS's size are added to the inspector kernel because each work-group finds out their own PAS base by offsetting the PAS argument by $flat\_work\_group\_id \times PAS\_SIZE$ as shown in Figure 4.4. Note that the size of PAS differs by global argument because it depends on the actual OpenCL buffer allocation size from the application.

Once the kernel arguments are setup, VAST performs several steps to transform the kernel as illustrated in Figure 4.5(a). First, VAST finds out load (LD) or store (ST) instruction that accesses global argument using Def-Use (DU) chains. Next, it matches the base of LD/ST to the base of the PAS. After VAST gets the corresponding PAS base, it inserts a node that computes the PAS index using the offset of the LD/ST instruction. The amount of shifting the offset is determined statically by looking at the type of base pointer and the

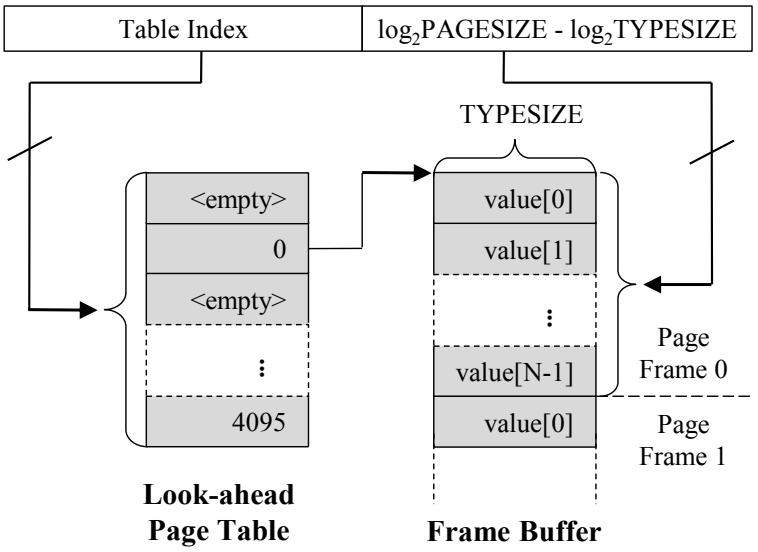



**Figure 4.6:** The design of Look-ahead Page Table (LPT) and frame buffer. One pair of LPT and frame buffer corresponds to one global argument.

page size as shown below.

$$SHIFT\_AMT = log_2 PAGESIZE - Ceil(log_2 TYPESIZE) \tag{4.2}$$

For example, the offset of **`float`** type array will be shifted right 10 bits if the page size is 4,096 bytes for the index of PAS. VAST also supports a custom type such as `struct`, but in the page frame, those array elements will be aligned to the order of magnitude through $Ceil()$ function in the equation. The next step is to insert a new store instruction with the PAS base, and PAS index as shown in Figure 4.5(a). Finally, VAST proceeds with *dead code elimination* (DCE) using the mark-sweep algorithm [78] in order to remove all the computation code as well as the dead control flows. Because computation code and unrelated control flows are removed by DCE, the inspector kernel completes PAS generation very quickly, which is evaluated in Section 4.6.

In the meantime, VAST also generates the *Paged Access Kernel*. Similar to the inspector kernel, VAST also adds additional arguments for *LPT* and *frame buffer* for each global

argument to the paged access kernel. In addition, it inserts two additional arguments for the range of work-groups to execute as well as checking code at the beginning of the kernel for the *partial execution* as discussed in Section 4.4.2 of Chapter III.

After setting up the kernel arguments, VAST only inspects uses of global array access using DU chains as the same as the initial step for the inspector kernel. For each use of global arguments, VAST inserts nodes for *address translation* based on the design of LPT and frame buffer as shown in Figure 4.6. As shown in the figure, an additional `LOAD` is used for querying the target frame index. The `LOAD` address of LPT is likely to be the same among work-items, because OpenCL programmers are encouraged to write the kernel to access the memory coalesced between work-items to fully utilize memory parallelism. Once the nodes for the frame pointer and the frame offset are added, VAST replaces the original base and offset operands of memory access with the frame pointer and the frame offset respectively as shown in Figure 4.5(b).

***Pointer Aliasing*** is also handled in VAST during kernel transformations. As OpenCL kernels can use a pointer variable, aliased global pointers also must be considered for both the inspector kernel and the paged access kernel. An important property of an OpenCL program is that any pointer variable that will alias to a global argument must be declared with the `__global` keyword. As a results, if global pointer arguments do not alias one another, aliases of those can be determined as either *No-Alias* or *Must-Alias* in many cases through basic alias analysis. With this alias result, VAST can keep track of aliased pointers by offsetting the base. If an aliased global pointer uses different type (typecast), VAST uses different $SHIFT\_AMT$ for the aliased pointer during transformation.

**Algorithm 2** PAS Reduction

```
1: k = 1
2: RangeFrom = 1
3: ReducedPAS[1..NumArgs][1..PAS_SIZE] = 0
4: MemFull_PAS[1..N][1..NumArgs][1..PAS_SIZE] = 0
5: for i = 1 to PAS_CNT do
6:     TotalMemUsed = 0
7:     for j = 1 to NUM_ARGS do
8:         MemFull_PAS[k][j] = ReducedPAS[j]
9:         ReducedPAS[j] = OR_REDUCE(ReducedPAS[j], PAS[j][i])
10:        MemUsed = SUM_REDUCE(ReducedPAS[j])×PAGE_SIZE
11:        TotalMemUsed = TotalMemUsed + MemUsed
12:    end for
13:    if TotalMemUsed > MEM_SIZE then
14:        i = i - 1                                   ▷ Roll-back one step
15:        k = k + 1
16:        StoreRange(RangeFrom, i)                    ▷ Store partial execution range
17:        RangeFrom = i + 1
18:        ReducedPAS[1..NumArgs][1..PAS_SIZE] = 0     ▷ Reset
19:    end if
20: end for
21: StoreRange(RangeFrom, PAS_CNT)                     ▷ Store the last range
22: return MemFull_PAS[1..k]                           ▷ return reduced PASs
```

### 4.4.3 Look-ahead Page Table Generation

Once PASs are generated by launching the inspector kernel, those are transferred to the host in order to generate *Look-ahead Page Table* (LPT) and frame buffer. As shown in Figure 4.4 and 4.6, LPT differs from PAS in that the entry of LPT contains a frame pointer (4 bytes), while PAS contains a boolean variable (1 byte). Moreover, LPT stands for one global argument, while PAS is maintained per global argument per work-groups as discussed in previous sections. LPT can be produced multiple times for one global argument because one LPT is for a single sequence of partial execution, which executes for the working set that fits into GPU memory as shown in Figure 4.3.

In order to generate LPT using PASs, VAST follows several steps. Considering that each PAS stores the list of pages accessed by a set of work-groups, the first step is to

reduce several PASs, which will contain the list of pages accessed by *multiple* sets of work-groups. This reduction process is done until the accessed page size in the reduced PASs for all global arguments does not exceed the physical memory size. Algorithm 2 illustrates how VAST reduces the PASs. As illustrated in the algorithm, the outermost loop (line 5-20) iterates over the sets of work-groups, and the PAS used by a set of work-groups are OR-reduced to the PAS used by the next set of work-groups for each global argument (at line 9). As a result, OR-reduced PASs will contain the pages accessed by the multiple sets of work-groups. After OR-reduction, VAST checks the number of accessed pages by executing SUM-reduce on reduced PAS (line 10-11). If the total page size of the reduced PAS exceeds the GPU memory, it stores the range for *partial execution* (line 13-19), and continue reduction until PASs for all the work-groups are processed.

The time complexity of this algorithm is $O(kNM)$, where $k$ is the number of global arguments, $N$ is the number of work-groups, and $M$ is the number of PAS entry count. In general, $k$ is a very small number ($<$5) and N can be reduced by letting several work-groups to share one PAS. Furthermore, reduction operation, which takes $O(M)$, can be further optimized by utilizing SIMD instructions as reducing two entries is independent from reducing other entries. In other words, 16 entries can be reduced at once with the 128-bit SIMD instruction as each PAS entry consists of a single byte. As a result, the entire PAS reduction produce negligible overhead, which is evaluated in Section 4.6.

Once reduced PASs were computed, VAST keeps reduced PASs and the ranges of work-groups for the partial execution. Each reduced PAS that corresponds to the range will represent a list of pages that will be accessed by the range of work-groups to be executed.

With reduced PASs, the next step is to allocate the frame buffer for each global argu-

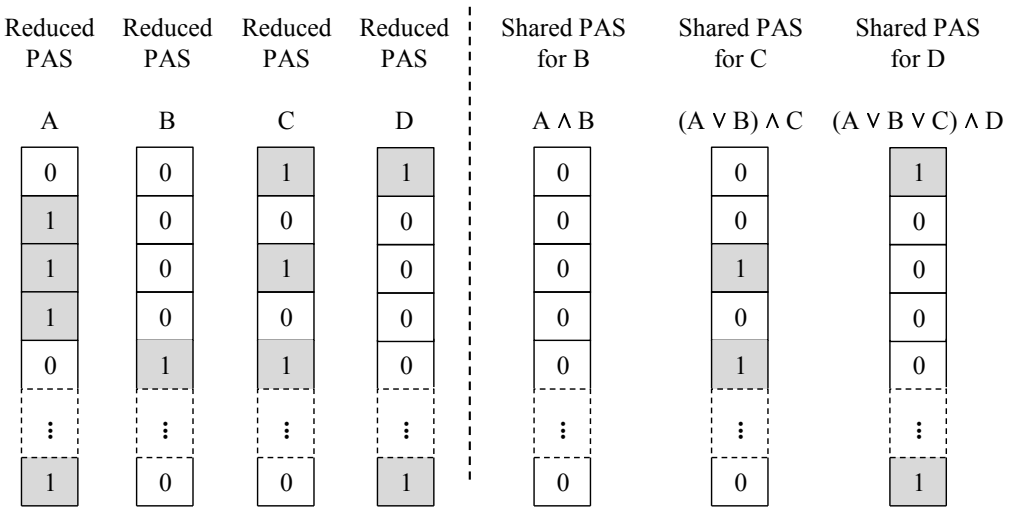



**Figure 4.7:** PAS generation for shared pages (shared PAS). Each reduced PAS is used for a single sequence of partial execution. As the sequence of partial execution increases, the number of logical operations increase for shared PAS as VAST should check pages used in the previous sequences.

ment on the host. At this point, VAST knows how much memory space is required for the frame buffer by multiplying the page size by the number of valid (`TRUE`) entries in the reduced PAS. Finally, VAST starts to fill up the LPT and the frame buffers by iterating entries of the reduced PASs. If the entry value of PAS is `TRUE`, it copies the corresponding page of the plain array to the contiguous memory space, the *frame buffer*. At the same time, the location (index) of the page frame in the frame buffer is stored to the entry of LPT as shown in Figure 4.6. One intelligent feature is that if the corresponding argument is a *Write-Only* argument, VAST fills up the frame buffer only for the shared pages, which will be discussed in detail in the next section.

### 4.4.4 Forwarding Shared Pages

As discussed in Section 4.2, the OpenCL uses a relaxed memory consistency model for *global* memory within NDRange. Thus, any order of work-group execution will produce

the same result without concerning write-conflicts among work-groups. However, in VAST, the granularity of memory transaction between the host and the GPU is a page frame, which is much larger than usual memory write. As a result, *false sharing* of a page may occur among the work-groups with regards to a *writable* frame buffer because more than one work-group can change the same page even though they write different location within the page. Sharing the same pages between work-groups within the same sequence of partial execution is safe because there is no write-conflict within a page as discussed. However, if a page was shared among work-groups in different sequences of partial execution, VAST must transfer up-to-date page frames for each partial execution due to *Write-After-Write* (WAW) dependencies. One option is that VAST waits for the partial output to be arrived from the previous partial execution, and also waits for the partial output to be recovered to the plain array. After that, VAST proceeds with LPT and frame buffer generation in order to transfer up-to-date page frames for the next partial execution.

Obviously, this option brings a serious serialization of the execution, because LPT and frame buffer generation could be overlapped from the kernel execution and transferring data back and forth between the host and the GPU as shown in Figure 4.3. In order to avoid serialization, VAST precomputes shared pages among each sequence of partial execution, and selectively copies the shared pages by looking at the list of shared pages (*Shared PAS*). A shared PAS for partial execution can be precomputed using reduced PASs shown in Figure 4.7. The example in the figure illustrate that there are four reduced PASs, which means the entire kernel execution was decomposed into four sequences of partial execution. At each sequence, every accessed page in the PAS must be compared with the same page in previous sequences of PAS. For example, at the fourth sequence D, VAST performs *OR*-

*reduce* PASs for all previous sequences, A to C, which represents all the pages accessed by the previous sequences of partial execution. After that, VAST *AND-reduce* it with the PAS of the fourth sequence. VAST also keeps these shared PASs only for the *write* arguments, and selectively copies the frame buffer when the partial results are transferred back to the host.

## 4.5 Further Optimization

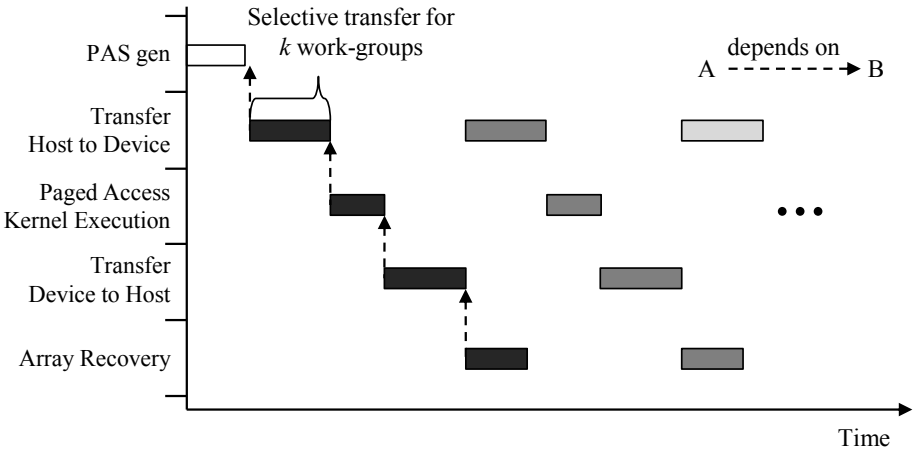
In this section, we further investigate optimization opportunities on VAST.

### 4.5.1 Selective Transfer

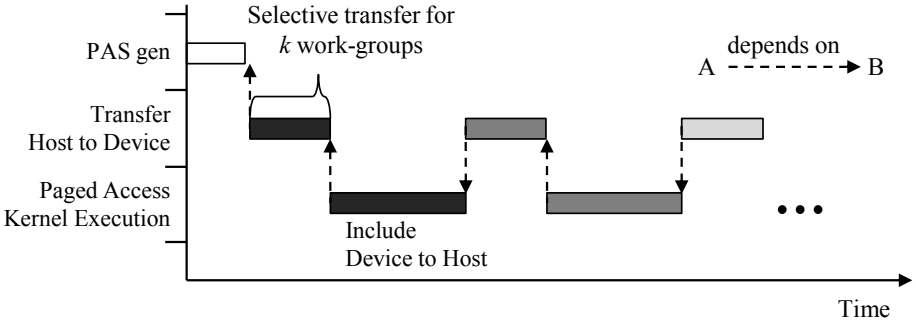
In Figure 4.3, the steps of *LPT/Frame gen.*, *HostToDev*, *DevToHost* and *Recovery* consume the host memory bandwidth. Considering that the bandwidth of the host memory is relatively lower than GPU memory, huge bottlenecks may exist in these stages, so it is important to minimize the usage of host memory. Especially, a significant bottleneck is in *Frame generation* because it copies data scattered accross memory space into another contiguous space within the host memory, which has limited bandwidth.

Strictly speaking, gathering the data into contiguous space does not have to be done in the host because the host can selectively transfer the data to the GPU. In other words, data can be gathered in GPUs directly while the host transfers the data.

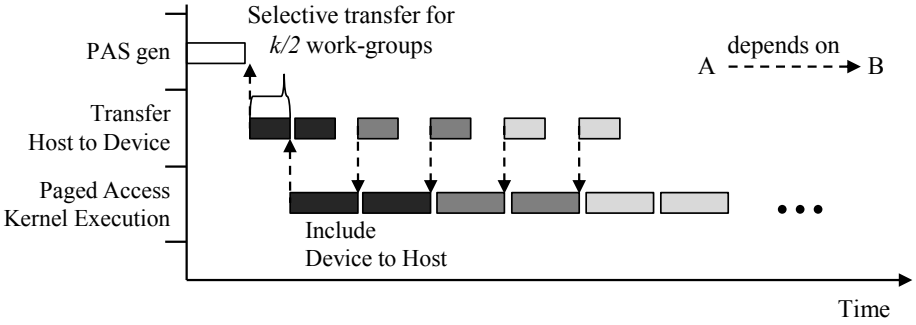
This approach may cause too many transfer requests due to selective transfers at the page granularity. but the cost can be hidden by using asynchronous APIs. As a result of this optimization, *LPT/Frame gen.* will be almost removed as shown in Figure 4.8(a). as




**(a)** Selective input transfer




**(b)** Zero copy memory for output buffers




**(c)** Double buffering

**Figure 4.8:** Execution timeline after optimizations. Selective input transfer removes the cost of frame generation (a). Zero copy memory for output buffer removes the cost of array recovery (b). Double buffering overlaps the input transfer with kernel executioans (c).

the cost LPT generation is almost invisible.

### 4.5.2 Zero Copy Memory

Although using selective transfer reduces the cost of *LPT/Frame gen.*, by removing the cost of *Frame Generation* in the host, a considerable bottleneck may still exist in the host due to *Recovery*.

One option to reduce this cost is to use zero copy memory [51]. Zero copy memory allows GPUs to access the host memory directly through the PCI express bus without allocating buffers on the GPU. The main advantage of zero copy memory is that the execution of a kernel on GPUs is not limited by GPU's physical memory size. However, it can cause serious slowdown from low bandwidth and high latency of PCI express bus, because each work-item in a kernel accesses the host buffer on demand.

For this reason, zero copy memory is known to be beneficial only when the buffers are read from or written to only once during the kernel execution [51]. The key observation in OpenCL/CUDA kernels is that each element of output arrays is written no more than once in general, because multiple work-items writing to the same memory locations will cause undefined behaviors [58]. Thus, multiple writing to the same location is done by the same work-item (thread). This happens when a work-item wants to keep an intermediate result in the array before the final store. In this case, programmers usually use a temporary local variable to reduce the number of memory accesses.

Upon this observation, one optimization that can be applied to VAST is to allocate *output buffers* as zero copy memory. This will eliminates the entire *Recovery* step as shown in Figure 4.8(b), and reduces the time for *PAS gen*, *PAS reduction*, *LPT generation* steps, because page tables and frames for output buffers are not maintained anymore. Meanwhile,

*DevToHost* step will disappear in the execution time line, but the cost will be added to the kernel execution *Kernel Exe.*, because the data will be written to the host memory directly during the kernel execution.

### 4.5.3 Double Buffering

In VAST, one sequence of execution takes workload as much as possible, and the next sequence of execution had to wait until the output buffer arrives from the previous sequence of execution. The main reason of this serialization is that pages for output buffers can be shared between sequences of execution as discussed in Section 4.4.4, so the next sequence of execution must wait until clean copy of pages arrive to the host.

However, by using zero copy memory for output buffers as discussed in Section 4.5.2, the next sequence of execution does not have to wait until the previous sequence finishes execution as each sequence will write the data directly to the host exclusively. Since this restriction is removed, the execution (*Kernel Exe.*) can be overlapped with the data transfer (*HostToDev*) as shown in Figure 4.8(c). In addition, by letting each sequence of execution take fewer amount of workload, the first sequence of execution can start the execution earlier as the amount of data (*HostToDev*) for each sequence decreases.

## 4.6 Evaluation

VAST was prototyped using Clang [10] for OpenCL front-end, and Low-Level Virtual Machine (LLVM) 3.6 [42] for the back-end. Once Clang parses the OpenCL kernel and generates the IRs, LLVM transforms it to the inspector kernel and the paged access kernel.

| Device | **Intel Core i7 - 3770** | **NVIDIA GTX 750 Ti** |
|---|---|---|
| **# of Cores** | 4 (8 Threads) | 640 |
| **Clock Freq.** | 3.2 GHz | 1.02 GHz |
| **Memory (B/W)** | 32 GB DDR3 (12.8 GB/s) | 2 GB GDDR5 (86.4 GB/s) |
| **Peak Perf.** | 435.2 GFlops [27] | 1,306 GFlops |
| **OpenCL Ver.** | Intel SDK 2013 [1] | CUDA SDK 6.0 [58] |
| **PCIe (B/W)** | 3.0 x16 (15.76 GB/s) | |
| **OS** | Ubuntu Linux 12.04 LTS | |

**Table 4.1:** Experimental Setup

After transformation is done, it lowers IRs to PTX binary to launch the kernels in the GPU.

In order to evaluate optimizations discussed in Section 4.5, there are three versions of VAST, each of which uses 4 Kbyte page size.

- **VAST**: VAST with selective transfers.
- **VAST+ZC**: VAST with selective transfers and zero copy memory for output buffers.
- **VAST+ZCDB**: VAST with all optimizations, selective transfers, zero copy memory for output buffers, and double buffering.

For the experiments, we used a real machine configured as shown in Table 4.1.

**Benchmarks and working sets**: For the evaluation, we used the OpenCL benchmarks from NVIDIA Computing SDK 4.2 [56]. We bailed out very simple benchmarks such as *BandwidthTest*, *InlinePTX*, *VectorAdd*, etc. Since the benchmark suite is for evaluating NVIDIA GPUs, some benchmarks are unable to take large inputs. For these benchmarks, we only modified them to take large inputs. The list of applications, execution parameters, and working set size are shown in Table 4.9.

**CPU-Baseline**: We used the OpenCL execution on the CPU device as one of our

baselines under the assumption that OpenCL kernels are fallen back to the CPU device (OpenCL's logical device) when the working set size exceeds the GPU memory. This is possible because the CPU device, which shares the address space with the host, has 32 GB of memory, and thus it can execute on a large working set. For the CPU execution, we used the Intel OpenCL library [1], which fully utilizes all cores with simultaneous multi threading (SMT) and SIMD instructions (SSE and AVX). In addition, the Intel OpenCL library allows a kernel to access the host's memory directly if the buffer was created with the *CL_MEM_USE_HOST_PTR* flag. Therefore, our CPU-baseline does not include data transfer time.

**GPU-Baseline**: In addition to the CPU-baseline, we also used normal GPU execution as another baseline. Normal execution means that the host application transfers input to the GPU at once, executes an OpenCL kernel, and gets the output back to the host. Because this execution model cannot proceed with the data that exceed the GPU memory, we used different execution parameters as shown in Table 4.9(b). The comparison with the GPU-baseline will evaluate the overhead of address translation.

**GPU-ZC**: For a large amount of data, programmers can develop OpenCL/CUDA kernels to use zero copy memory, in which kernels access the data in the host memory directly through the PCIe bus. In order to show the overhead of using zero copy memory and the performance comparison with VAST, we also used the moderate workload as defined in Table 4.9(b) for GPU-ZC.

| Application | Execution Parameters | OpenCL Kernel | Buffer Size | | | # of Workgroups |
|---|---|---|---|---|---|---|
| | | | Input | Output | Total | |
| BlackScholes | 256 million options | BlackScholes | 3.0 GB | 2.0 GB | 5.0 GB | 1,024 |
| FDTD3d | 3D dimsize=1024, Radius=2 | FiniteDifferences | 4.0 GB | 4.0 GB | 8.0 GB | 4,096 |
| MatrixMul | 20,480×20,480 matrices | MatrixMul | 3.1 GB | 1.6 GB | 4.7 GB | 409,600 |
| MedianFilter | 30,720×17,820 PPM image | ckMedian | 2.0 GB | 2.0 GB | 4.0 GB | 8,294,400 |
| MersenneTwister | 1.15 billion numbers | MersenneTwister | 1 MB | 4.3 GB | 4.3 GB | 512 |
| | | BoxMuller | 4.3 GB | | 4.3 GB | 512 |
| Nbody | 41 million particles | IntegrateBodies_MT | 1.3 GB | 1.3 GB | 2.6 GB | 81,920 |
| Reduction | 940 million numbers | Reduce6 | 3.5 GB | 64 KB | 3.5 GB | 16,384 |
| SobelFilter | 30,720×17,820 PPM image | ckSobel | 2.0 GB | 2.0 GB | 4.0 GB | 8,294,400 |

**(a)** Execution parameters with more than 2GB of data

| Application | Execution Parameters | OpenCL Kernel | Buffer Size | | | # of Workgroups |
|---|---|---|---|---|---|---|
| | | | Input | Output | Total | |
| BlackScholes | 64 million options | BlackScholes | 768 MB | 512 MB | 1,280 MB | 1,024 |
| FDTD3d | X-dim=768, YZ-dim=512, Rad=2 | FiniteDifferences | 784 MB | 784 MB | 1,568 MB | 1,536 |
| MatrixMul | 8,192×8,192 matrices | MatrixMul | 512 MB | 256 MB | 768 MB | 65,536 |
| MedianFilter | 11,520×6,480 PPM image | ckMedian | 285 MB | 285 MB | 570 MB | 1,166,400 |
| MersenneTwister | 384 million numbers | MersenneTwister | 0.5 MB | 1,465 MB | 1,465 MB | 256 |
| | | BoxMuller | 1,465 MB | | 1,465 MB | 256 |
| Nbody | 262,144 particles | IntegrateBodies_MT | 8 MB | 8 MB | 16 MB | 512 |
| Reduction | 235 million numbers | Reduce6 | 896 MB | 0.1 MB | 896 MB | 16,384 |
| SobelFilter | 11,520×6,480 PPM image | ckSobel | 285 MB | 285 MB | 570 MB | 1,166,400 |

**(b)** Execution parameters for less than 2GB of data

**Figure 4.9:** Benchmark specifications. For the speedup over the CPU-baseline, data size more than GPU memory is used (a). For the comparison with normal GPU execution, data size less than GPU memory is used (b).

### 4.6.1 Results

**Comparison with the CPU-baseline**: First, we evaluated the speedup of three versions of VAST over the CPU-baseline. The time measured in VAST involves the time for PAS/LPT generation, and buffer transfer time.

As shown in Figure 4.10, all versions of VAST performs better than the CPU execution on average. Especially, VAST+ZC and VAST+ZCDB performed better than CPU on every benchmark, showing geometric means of 3.2x and 3.46x speedup, respectively.

The reason why Blackscholes, MersenneTwister, and BoxMuller show a huge performance gap between VAST and VAST+ZC is that those applications are generating rela-

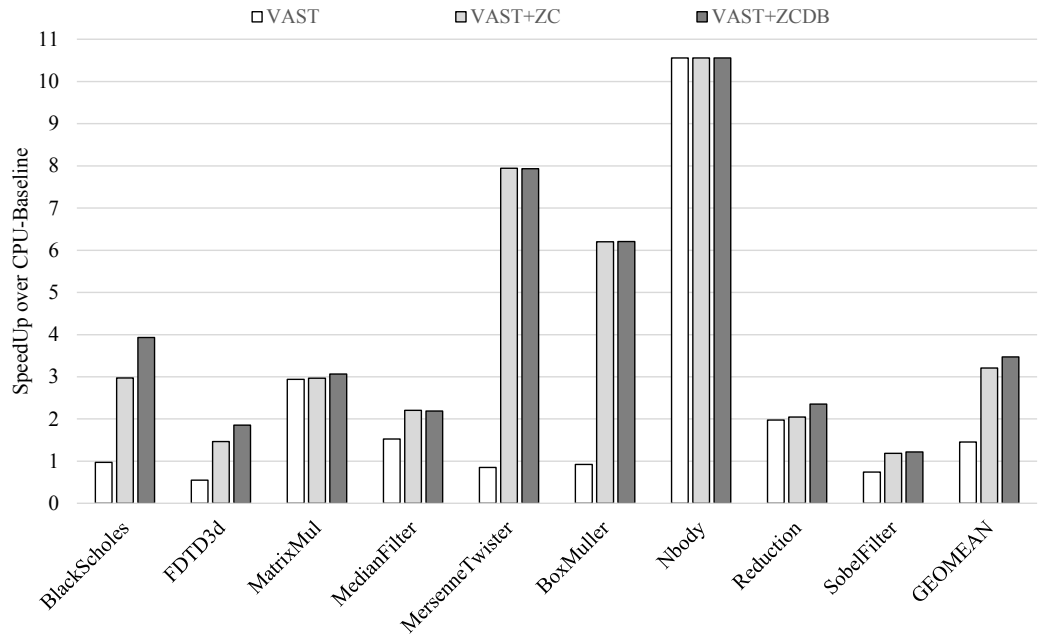



**Figure 4.10:** Speedup of VAST over Intel OpenCL execution with 4 KB page frame size. VAST does selective transfers for input buffers. VAST+ZC uses selective transfers for input buffers and zero copy memory for output buffers. VAST+ZCDB uses all optimization techniques discussed in Section 4.5.

tively large amount of data compared to computations. Particularly, MersenneTwister is a random number generator, which take seeds as an input (few bytes of data), while the outputs are a huge number of random numbers as shown in Table 3.2. Without zero copy, a significant amount of time will be spent in *Recovery* step, which is done in the host memory.

In contrast, MatrixMul and Nbody do not show observable differences between VAST and VAST+ZC because they are very compute intensive benchmarks, which means that the portion of array recovery is almost negligible compared to the kernel execution time.

**Comparison with the GPU-baseline**: Conceptually, VAST is activated only when the data size is more than GPU's physical memory. However, in order to investigate the cost of virtualizing the address space, we forced to trigger VAST execution in spite of small data size defined in Table 4.9(b), and compared three versions of VAST with the GPU-baseline.

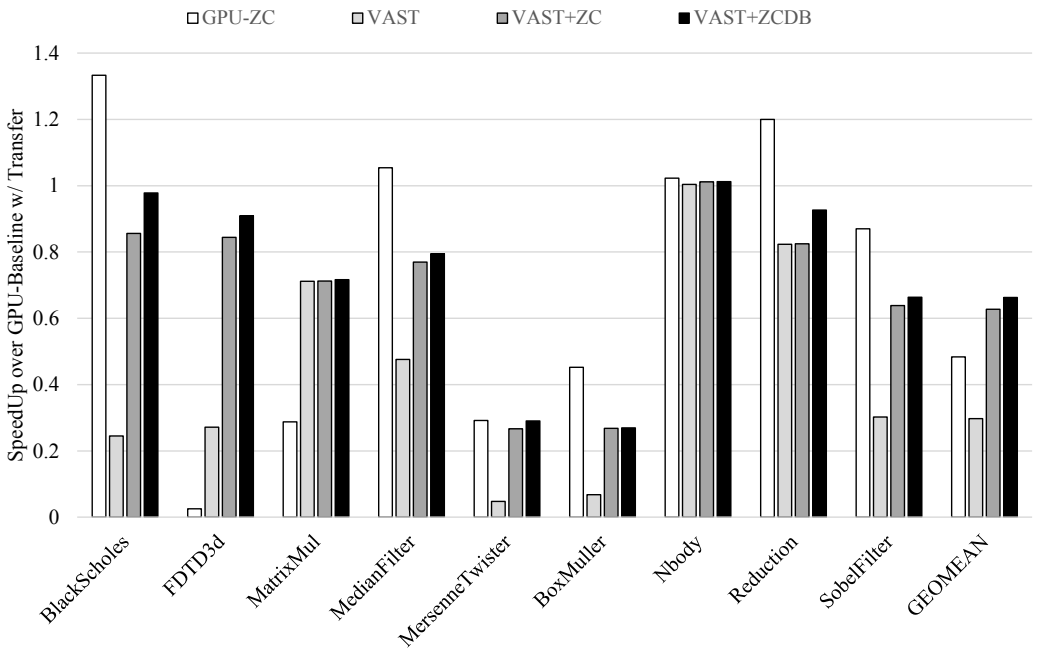



**Figure 4.11:** Speedup of VAST over the GPU-baseline. The performance was measured using small workloads that fits into GPU memory. GPU-ZC is the execution using zero copy memory for all buffers.

With moderate working set sizes, Figure 4.11 shows the performance of GPU-ZC and three versions of VAST normalized to the GPU-baseline. These number includes data transfer for both VAST and the GPU-baseline.

As shown in the figure, VAST with all optimizations achieves 66% performance of the GPU-baseline performance on average, while GPU-ZC that uses zero copy memory for all buffers gets only 48% of GPU-baseline performance. The gap between VAST and GPU-baseline generally comes from the exposed cost of generating PAS and page lookups during the kernel execution, which is not necessary for the GPU-baseline.

Some benchmarks, such as BlackScholes, MedianFilter, and Reduction, perform better with GPU-ZC than with GPU-baseline. The main reason for this is that those are streaming kernels where buffers are read from or written to only once during the kernel execution. As a result, high latency of the PCIe bus is amortized by prefetching the data. as well as the

kernel execution can start the execution without copying the entire data so the execution time can be overlapped with the data transfer. Considering that these benchmarks are very memory intensive in which the communication cost is larger than the execution time, most of the execution time can be hidden.

However, GPU-ZC shows very poor performance on FDTD3d and MatrixMul because one array element is accessed by multiple work-items, which results in multiple PCIe transactions for each work-item.

For VAST, similar to the comparison with the CPU-baseline, memory-intensive benchmarks, such as BlackScholes, FDTD3d, MersenneTwister, and BoxMuller show huge gaps between VAST and VAST+ZC. The reason why MersenneTwister and BoxMuller show poor performance compared to the GPU-baseline is that they generate large output array (1.46 GB), and every time it generate a single random number, it writes back to the host through the PCIe bus that has high latency. Blackscholes is also memory intensive, but achieves good performance because it involves more computations than MersenneTwister and BoxMuller. Also, the output size of Blackscholes is much less than those two benchmarks.

On the other hand, Nbody on VAST shows slightly better performance than the GPU-baseline, even though VAST has overhead for PAS generation and paged access kernel. According to the implementation of Nbody in NVIDIA Computing SDK, the entire input data is accessed by every work-item, but the start locations of global memory accesses among work-items are different in order to avoid all multiprocessors trying to read the same memory locations at once. This implementation can cause scattered accesses. For example, the working set for the last work-group starts to access the tail of the array, and

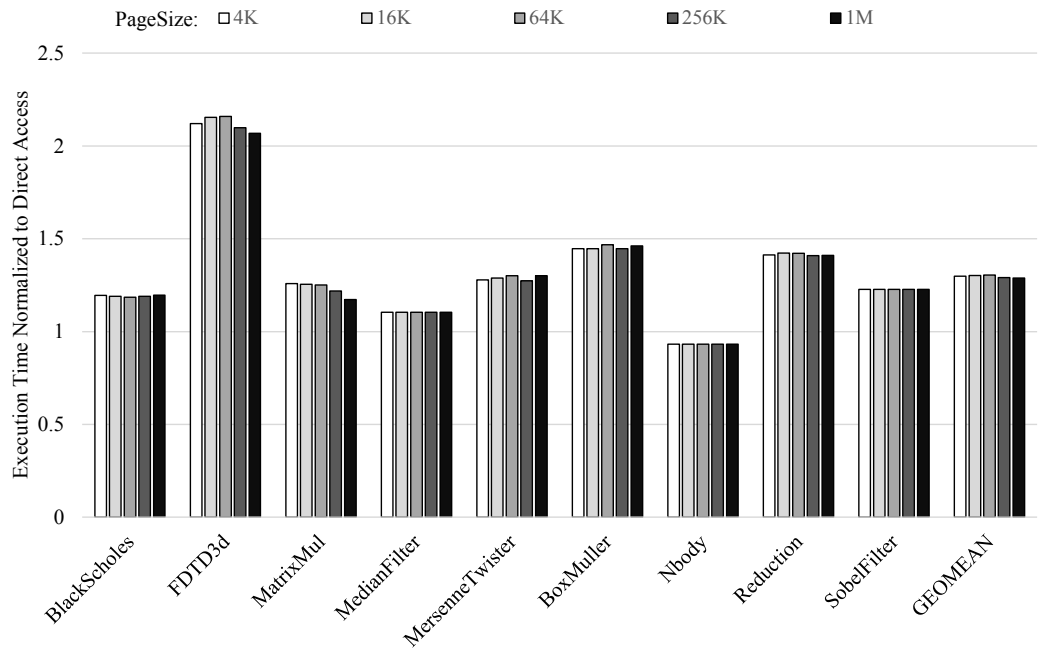



**Figure 4.12:** Paged access kernel execution time normalized to the original kernel execution time. Working set size for each benchmark is less than 2 GB. Paged-access kernel execution does not use zero copy memory.

then accesses the head of the array. In VAST, scattered memory locations are gathered into contiguous page frames, resulting in speedup from more memory level parallelism. The overhead and benefit of paged accesses are discussed in detail in Section 4.6.2.

### 4.6.2 Page Lookup Overhead

As discussed in Section 4.3 and 4.4, if the working set size exceeds GPU's memory size, VAST transforms the OpenCL kernels to make them access the data through LPT. As a result, one additional memory access occurs for each global memory access for page lookups. In the worst case, VAST could experience large slowdown due to doubled memory accesses for looking up pages. However, a realistic lookup overhead will not be significant because the size of LPT is small enough to fit in the GPU's cache and all the work-items in a work-group have a higher probability of looking up the same page, taking advantage of

memory level parallelism. In order to evaluate accurate page lookup overhead, we forced VAST to execute paged access kernels on a small working set, and compared the time spent in the GPU for the kernel execution. For this comparison, we did not use VAST+ZC and VAST+ZCDB because the kernel execution time includes the data transfer for the output when zero copy memory is enabled. We also differentiated page sizes from 4KB to 1MB in order to observe the performance impact from varied page sizes. For the overhead estimation, we chose our working set size as approximately 1.6GB to not exceed the physical memory size on the GPU.

Figure 4.12 illustrates paged access kernel execution time, normalized to normal kernel execution time. As shown in the figure, page lookups bring 29.4% overhead on average with various page sizes. However, this results is based on VAST without zero copy memory, the actual page lookup overhead can be reduced if zero copy memory is applied.

A majority of the benchmarks shows less than 25% page lookup overhead, but FDTD3d shows considerable overhead. FDTD3d is a 3-dimensional stencil benchmark in which 3-dimensional adjacent input elements must be accessed to compute one output element. In this case, working sets of adjacent work-items are not contiguous in linear array space, but far away one another. As a result, VAST suffer from noticeable overhead as adjacent work-items will lookup different pages which causes separate memory transactions for lookups.

In the meantime, VAST shows slightly better performance on Nbody in spite of additional page lookups for every memory access. This is because memory access for the page lookup affects L2 cache in a positive way for fetching actual data from page frames. To explain, Figure 4.13 shows the performance counters collected on Nbody with 256K particles. As shown in the figure, Paged-access kernel has approximately 60 million more

instructions due to page lookups (a), which result in 20 million more load transactions (b) and more register pressure (d). However, Paged-access kernel experiences higher IPC (f), because it gets more L2 hit rate (i). L2 hit rate of paged-access kernel comes from less cache thrashing, because it requires more registers per thread, which results in fewer number of threads scheduled on a streaming multi-processor at a time.

## 4.7 Related Work

A variety of research has been done to virtualize OpenCL devices as OpenCL programming models expose hardware details, such as the number of processing elements, the number of registers, and the physical memory size in each level of the hierarchy.

While a series of previous works tried to virtualize GPU hardwares in order to alleviate the efforts in optimizing performance [26, 81, 70], other types of researches focus on virtualizing multiple GPUs to distribute the workload automatically [49, 38, 39, 44, 40]. Although they split data parallel workloads into several parts to distribute them over multiple computing devices, they either limit the data parallel kernels to have specific memory access patterns [38, 39, 49, 40], or distribute the entire data to each device assuming the entire working set fits into each GPU memory [44]. On the contrary, VAST fully virtualizes the GPU memory that also supports kernels with indirect memory access pattern.

In the mean time, numerous works have proposed automatic CPU-GPU communication in order to remove explicit memory management [29, 28]. CGCM [29] uses compile-time type-inference to statically determine the types of data structures and transfer them between CPU and GPU memories, DyManD [28] employs run-time libraries to allocate

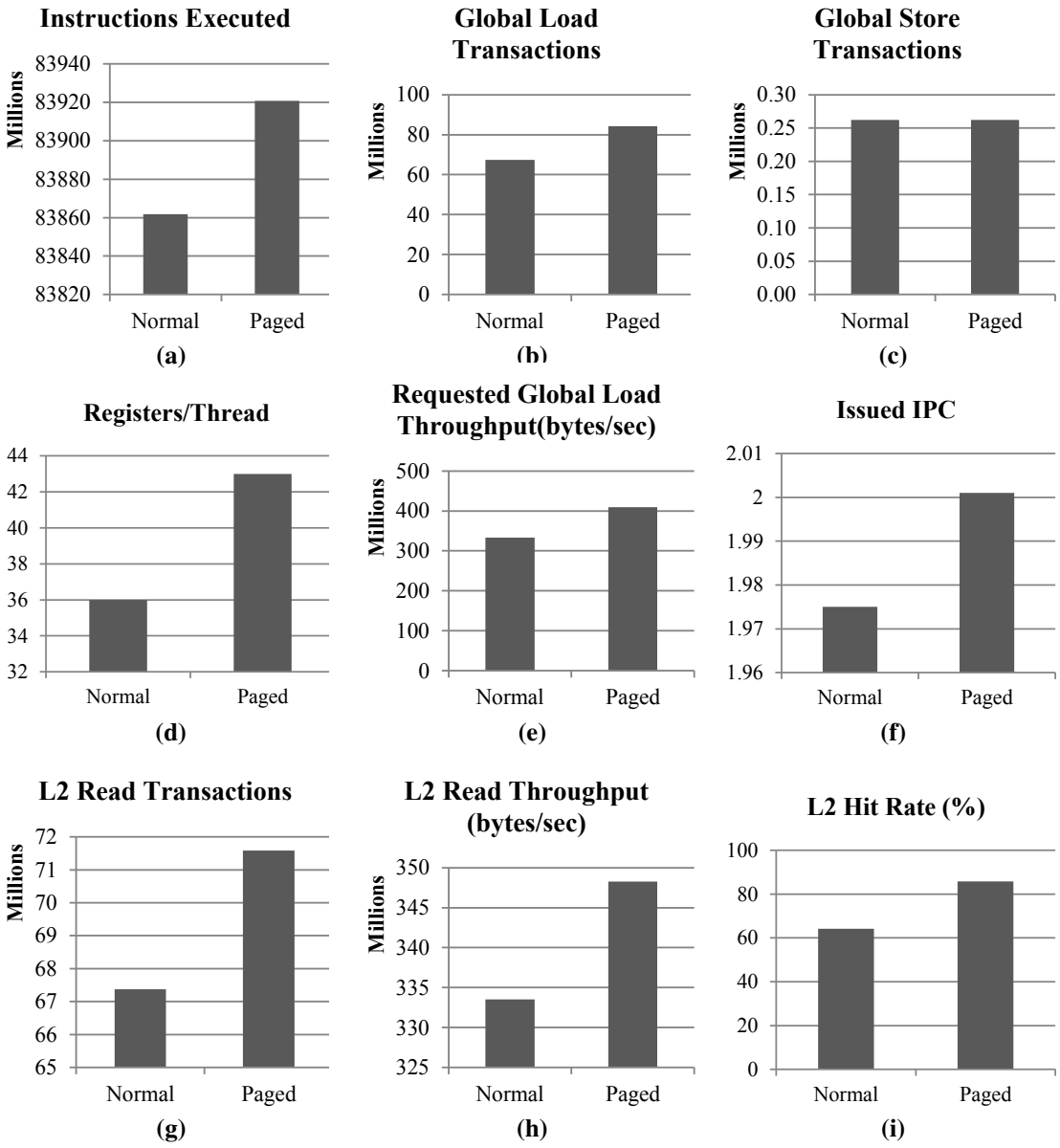



**Figure 4.13:** Performance counters collected on N-body with 256K particles. Paged-access kernel has approximately 60 million more instructions (a), and 20 million more load transactions for page lookups (b). However, Paged-access kernel experiences higher IPC (f), because it gets more L2 hit rate (i).

data-structures in CPU and GPU memories. However, they only manage communications between kernels in cyclic patterns, assuming the entire working set fits into GPU memory. Larger data sets require explicit programmer management. More recently, OpenACC [62] proposed new programming standards that consist of a set of compiler directives at a higher

level, which hides hardware details from programmers. Despite these efforts, programmers must explicitly mange buffer transfer due to physical memory size on the GPU.

NVIDIA introduced Unified Memory in CUDA [58]. NVIDIA's Unified Memory abstracts away data transfer management between the host and GPU by allocating memory spaces in both the host and GPU and managing coherenece at the OS's page granularity. As a result, programmers do not have to copy the data back and forth between the host and GPU, but the runtime system transparently migrates the data between the host and GPU depending on whether pages are clean or dirty on each side. Although Unified Memmory provides programmers with a single virtualized space from seperate physical memories of the host and GPU, it still physically allocates the space in GPU that exactly correspond to the host memory. In other words, Unified Memory failes if the application tries to allocate the space larger than the physical memory size of GPU.

Inspector-executor (IE) model, which is applied to VAST, has been also adopted in the field of distributed systems [4, 52, 72]. In their works, the inspector code is used for identifying precise loop-carried dependencies for irregular access pattern in order to distribute loop iterations over multiple nodes. Although some of their works perform data compaction to reduce communication cost between nodes, the target system extracts the compacted data assuming that the target node has a plenty of memory space supported by virtual address space. On the other hand, VAST makes use of the inspector to generate LPT in order to provide virtual memory space on GPUs, while the executer is transformed from the original code to access through LPT.

Recently, there has been research projects to virtualize GPU memory [30, 65]. RSVM [30] is a region based virtual memory running on both CPU and GPU. In this work, program-

mers must define regions as the basic data unit abstractions. Then, RSVM manages these regions and transfers them if necessary. However, VAST does not need the programmer to think about access patterns of threads and divide the memory manually, thus, VAST is fully automatic and transparent. Pichai et al. [65] explored address translations on GPUs by putting translation look-aside buffers (TLBs) in the shader cores. Their work allows a unified virtual/physical memory space among CPUs and GPUs. However, they only target integrated CPU/GPU systems that share physical memory while VAST handles systems with descrete GPUs.

## 4.8 Conclusion

In this chapter, we presented the VAST system that provides the programmer with an illusion of large memory space for OpenCL devices. Virtualizing memory space for the GPU is done by automatically partitioning the OpenCL's NDRange into subsets of work-groups, and efficiently splitting the working set into several chunks required by a subset of work-groups based on available physical memory on the GPU. With the subsets of work-groups and divided working set, VAST performs partial execution multiple times consecutively.

To support these procedures, we introduced a new type of page table, LPT, which has addresses of page frames that will be accessed by a subset of work-groups on the GPU, and we introduced code transformation techniques to generate LPT efficiently and to make OpenCL kernels access memories through LPT.

Our experiments showed that NVIDIA GTX 750 Ti GPU with 2 GB of memory successfully executed for more than 2 GB of working set regardless of memory access pattern.

In addition, VAST achieved 3.46x speedup over CPU execution, which is a realistic alternative for large data computation.

# CHAPTER V

# MKMD: Multiple Kernel Execution on Multiple Devices

## 5.1 Introduction

As the amount of data to process keeps growing, data parallel hardware such as graphics processing units (GPUs) and data parallel coprocessors have been intensively focused on. With OpenCL and CUDA, more application domains focus on exploiting the computational power of GPUs, and the complexity of the applications being mapped onto data parallel 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

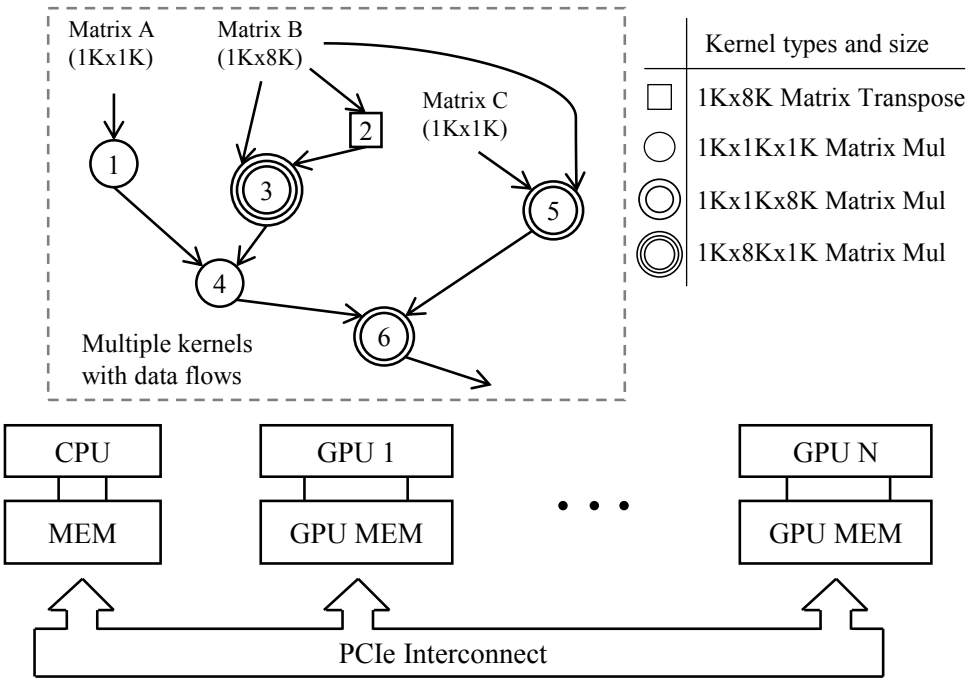



**Figure 5.1:** A kernel graph for solving a matrix equation, $A^2BB^TCB$, consisting of six kernels. The system is equipped with different computing devices with separated physical memory. Devices are connected through PCI express (PCIe) interconnect. Each kernel has different amount of computation, and each device has different performance.

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.

To explain these observations, Figure 5.1 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 5.1 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 5.1 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 5.1 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 dissertation 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 dissertation 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.

## 5.2 MKMD Overview

MKMD is a runtime library that is compatible with OpenCL APIs as illustrated in Figure 5.2. 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*

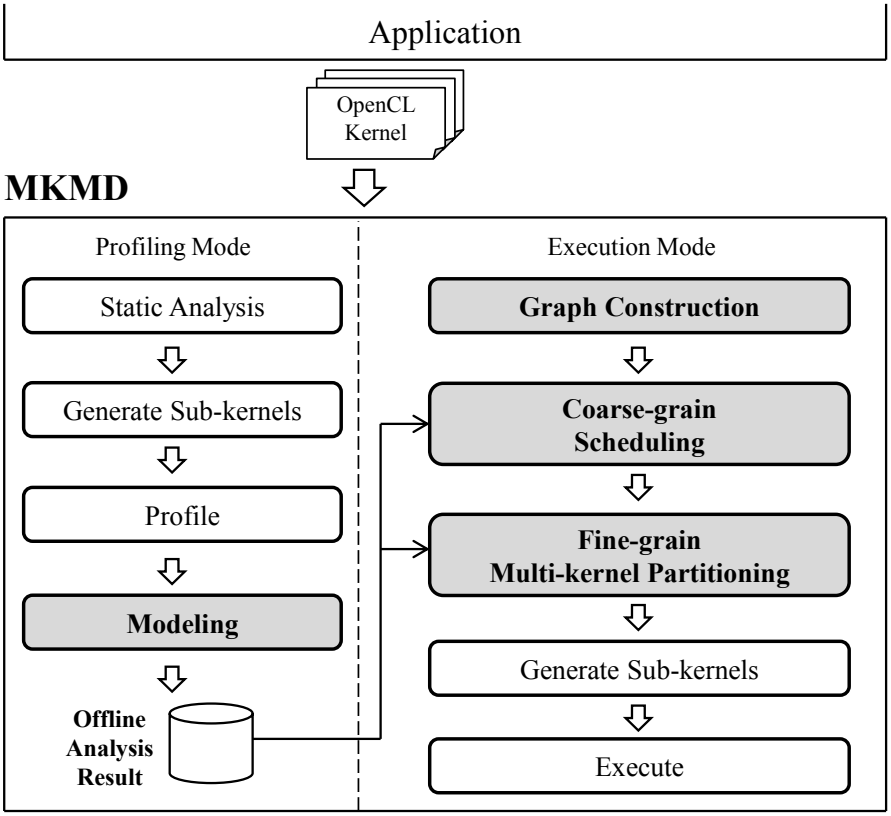



**Figure 5.2:** MKMD workflow that operates in profiling mode and execution mode. In profiling mode, MKMD builds a mathematical model with a set of profile data for the execution time prediction. In execution mode, MKMD predicts the execution time of kernels on various input sizes using the model, and schedules kernels based on the predicted time.

*mode*, and *execution mode*, as shown in Figure 5.2.

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 input 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 5.3.

Once the offline analysis is done, MKMD can be run in *execution mode*, which follows five steps as shown in Figure 5.2; 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.

## 5.3 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 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 works have examined experimental algorithmics in order to analyze the asymptotic cost of programs using representative input sets [61, 50, 82, 11]. 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. [50] 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, [82] narrowed down the domains to specific data structures, and [11] 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 [22, 23].

Based on these observations, this work investigates the potential of static analysis on

```
1  __kernel void square_matmul(__global float *C,
2      __global float *A, __global float *B, int N) {
3    int i = get_global_id(0);
4    int j = get_global_id(1);
5    for (int k = 0; k < N; ++k)
6      tmp += A[i * N + k] * B[k * N + j];
7    C[i * N + j] = tmp;
8  }
```

**(a)** Matrix multiplication

```
1  __kernel void vadd(__global float *C,
2      __global float *A, __global float *B, int N) {
3    int i = get_global_id(0);
4    int T = get_global_size(0);
5    for (int k = i; k < N; k += T)
6      C[k] = A[k] + B[k];
7  }
```

**(b)** Vector addition

**Figure 5.3:** Upper bounds of trip count. The upper bounds are statically determined as $N$ for (a), and $\frac{N}{T}$ for (b)

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, which was discussed in Section 3.2.3.

To illustrate the static cost analysis, Figure 5.3 shows two simple code examples. In Figure 5.3(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.

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 5.3(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

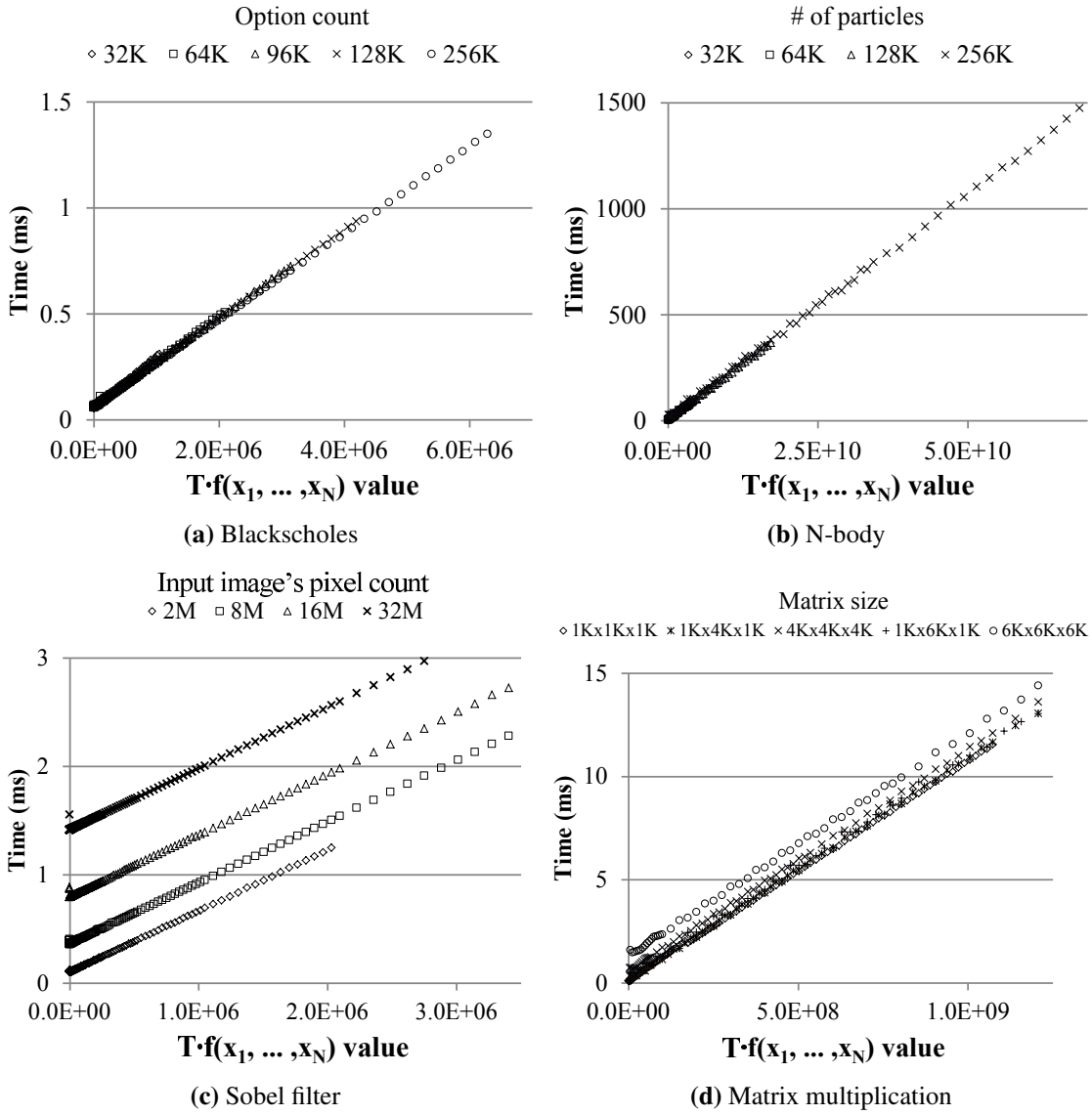



**Figure 5.4:** Scalability of execution time on NVIDIA GTX760 varying input sizes and the number of enabled work-items ($T$). The execution time is linear to the value of cost function $Tf(x_1, ..., x_N)$.

case, the estimated dynamic instruction count can be defined as,

$$c \times Tf(x_1, ..., x_N) \tag{5.1}$$

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

$$Time_{est} = \frac{CPI}{Freq_{clock}} \times c \times Tf(x_1, ..., x_N) \tag{5.2}$$

In Equation 5.2, $CPI$ (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 5.4 illustrates the execution time of several benchmarks from NVIDIA SDK [56], 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 3.2.1.

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 5.4 (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 3.2.1, 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.

Because Blackscholes in NVIDIA SDK uses the fixed number of work-items similar to the example shown in Figure 5.3(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

| 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 5.1:** 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.

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,

$$y = \beta_1 T f(x_1, ..., x_N) + \beta_2 T + \epsilon \tag{5.3}$$

With Equation 5.3, 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 $f(x_1, ..., x_N)$ and $T$.

In order to evaluate the accuracy of the execution time model, OpenCL applications from NVIDIA SDK and Rodinia [9] were used. Table 5.1 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 5.1 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 5.1, the average performance prediction error with random input remained under 3% with 20 profiling sets, and under 2% with 40 profiling sets.

## 5.4 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.

### 5.4.1 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.

### 5.4.2 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 [54] for certain cases [73, 6, 77]. The way that MKMD schedules in kernel granularity is similar to the HEFT algorithm [77] 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

$$P(V_i) = \begin{cases} W(V_i) + \max\limits_{V_j \in Succ(V_i)} (W(E_{i,j}) + P(V_j)), & V_i \neq V_{sink} \\ 0, & \text{otherwise} \end{cases} \tag{5.4}$$

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 5.4, 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

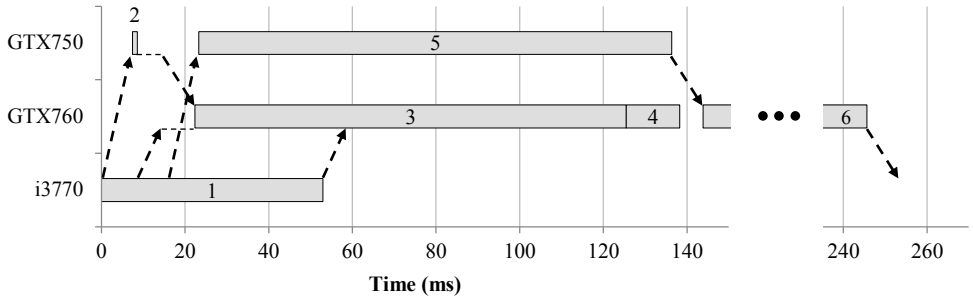



**Figure 5.5:** Coarse-grain scheduling result on three heterogeneous devices. Dotted arrows presents the buffer transfer between devices. PCI bus operates in full-duplex, but GTX760 and i3770 experience input and output congestion respectively.

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

$$EST(V_i, D_k) = \max_{V_j \in Pred(V_i)} \{KT_{end}(V_j) + T_{trans}(V_j, E_{j,i}, k)\} \tag{5.5}$$

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

$$T_{trans}(V_j, E_{j,i}, k) = \begin{cases} \dfrac{W(E_{j,i}) \times BW_{max}}{AvailBW(KD(V_j), k)}, & KD(V_j) \neq k \\ 0, & \text{otherwise} \end{cases} \tag{5.6}$$

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

$$EFT(V_i, D_k) = AvailEST(V_i, D_k) + T_{exe}(V_i, D_k) \tag{5.7}$$

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:

$$KD(V_i) = \underset{k \in Devs}{\operatorname{argmin}}\{EFT(V_i, D_k)\} \tag{5.8}$$

$$KT_{start}(V_i) = AvailEST(V_i, KD(V_i)) \tag{5.9}$$

$$KT_{end}(V_i) = EFT(V_i, KD(V_i)) \tag{5.10}$$

Figure 5.5 shows the scheduling result for the same application shown in Figure 5.1 on a system with three different devices, Intel i3770, NVIDIA GTX 760, and GTX 750. As shown in Figure 5.5, 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 5.4.3.

### 5.4.3 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

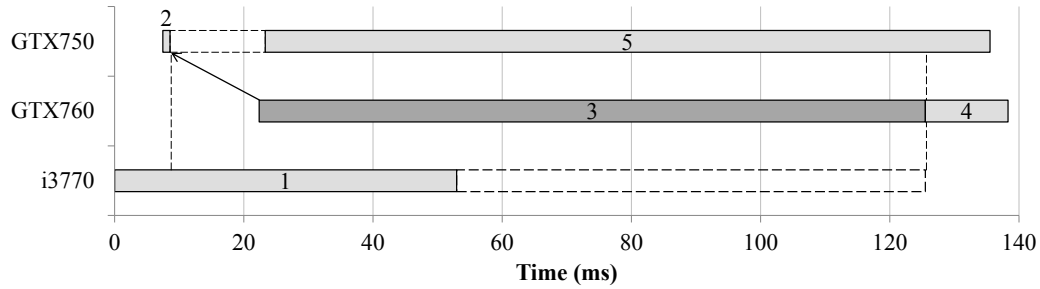



**Figure 5.6:** Available compute-time slots (dotted-squares) for partitioning kernel 3. Because kernel 3 depends on kernel 2 (arrow), the lower bound and upper bound of available time slots are the finish time of kernel 2 and 3 respectively.

the kernel earlier. As discussed in Section 3.2.1, an OpenCL kernel can be selectively executed at work-group granularity. Through the transformed kernel, MKMD can decompose a kernel into several *sub-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 5.5, 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 5.5, MKMD starts

**Algorithm 3** 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 ∈ Pred(V_i)} {KT_end(V_j)}
6:     UB ← KT_end(V_i)
7:     for k = 1 to NUM_devs do
8:         List_slot[k] ← Available time slots between LB and UB
9:     end for
10:    Partition V_i to List_slot[1..NUM_devs] by work-groups
11:    if Partitioned then
12:        Create new nodes SET_{p ∈ Partitions} {V_{i,p}}
13:        Update DAG with new nodes
14:        Update Schedule with the partition result
15:    end if
16: end for
```

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 5.6 visualizes available time slots for partitioning kernel 3 from the coarse-grain scheduling example in Figure 5.5. 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. This is a job-shop scheduling problem which minimizes the makespan of the entire schedule. Job-shop problem is an NP-complete 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 5.4.4.

**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 3 shows a high-level description of multi-kernel partitioning. In the algorithm, the first line prioritizes kernels 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 5.4.4 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 3.

### 5.4.4 Partitioning a Kernel to Time Slots

As discussed in Section 5.4.3, 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 3.2.1, 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

**Algorithm 4** Kernel partitioning to available time slots

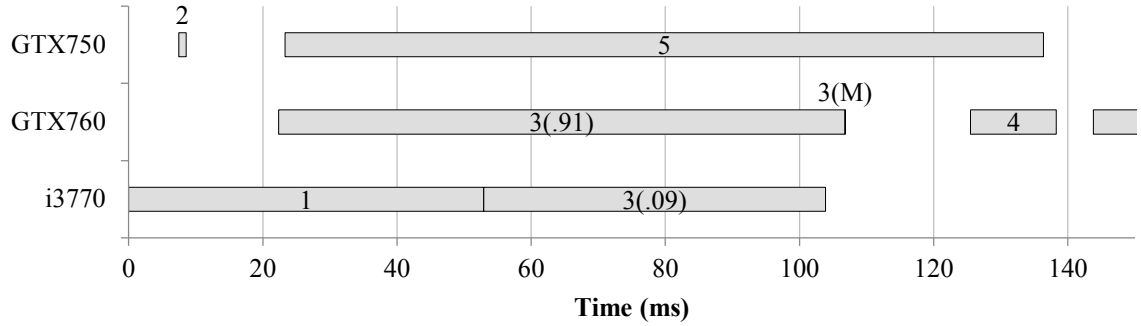
```
1:  Slots[1..S-1] ← all available time slots in List_slot[1..Num_devs]
2:  Slots[S] ← Scheduled time slot for kernel v
3:  currState.Slots[1..S] ← Slots[1..S]
4:  W_off ← number of work-groups to offload at a time
5:  Time_curr ← max_{s∈S}{currState.Slots[s].FinishTime}
6:  Time_prev ← Time_curr + 1
7:  while Time_curr < Time_prev do
8:      Time_prev ← Time_curr
9:      nextState[1..S] ← currState
10:     s_from ← argmax_{s∈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 ← argmax_{s∈S}{nextState,Slots[s].FinishTime}
15:     currState ← nextState[s_pick]
16:     Time_curr ← max_{s∈S}{currState.Slots[s].FinishTime}
17: end while
18: return currState.Slots
```

number of available time slots. Once candidate states are created, each candidate attempts to offload a fixed number of work-groups to their available time slot.

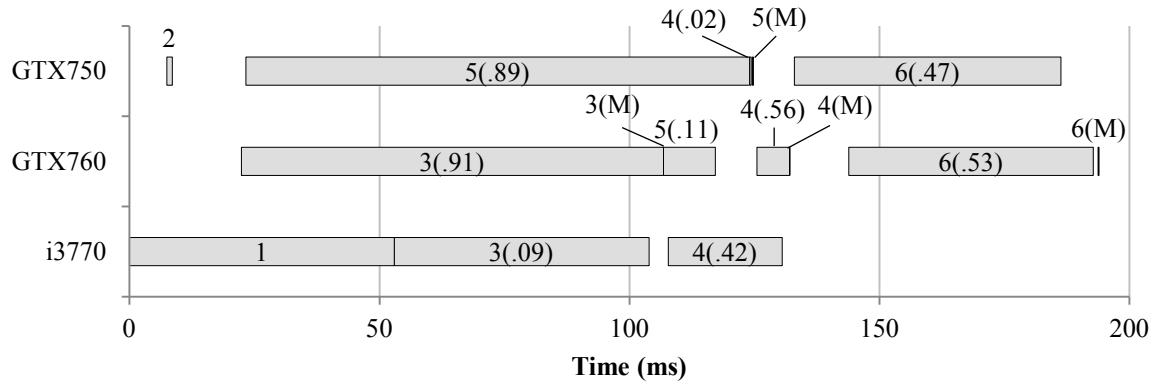
While candidates offload the work-groups, they estimate the execution time considering available interconnect bandwidth, amount of buffer transfer, and additional cost for merging outputs. During the estimation, MKMD also checks if the execution time for offloaded work-groups fits in the time slot, or 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 4 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.




**(a)** Partitioning result for kernel 3




**(b)** Final scheduling result

**Figure 5.7:** 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.

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 5.6 is partitioned as shown in Figure 5.7(a). After partitioning the rest of kernels, the final scheduling result is illustrated in Figure 5.7(b).

### 5.4.5 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 5.5.

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.

| 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 (86.4 GB/s) |
| **Peak Perf.** | 435.2 GFlops [27] | 2,258 GFlops | 1,306 GFlops |
| **OpenCL Ver.** | Intel SDK 2013 | CUDA SDK 6.0 | |
| **PCIe (B/W)** | - | 3.0 x8 x2 (7.88 GB/s) | |
| **OS** | Ubuntu Linux 12.04 LTS | | |

**Table 5.2:** Experimental Setup

## 5.5 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 [10] for the OpenCL front-end, and the Low-Level Virtual Machine (LLVM) 3.6 [42] 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.

In execution mode, MKMD takes the list of OpenCL commands from the application, and performs scheduling as discussed in Section 5.4.

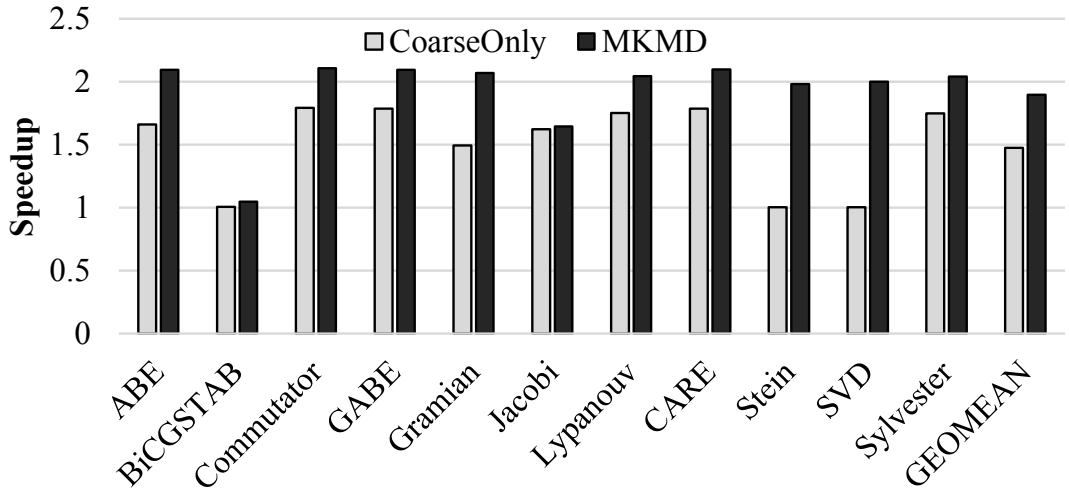
**Baseline**: For the experiments, we configured a real machine as shown in Table 5.2. The baseline of our experiment is in-order OpenCL execution on a single device assuming that the programmer picks the fastest device, GTX 760 in our experimental setup, and

| 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 |

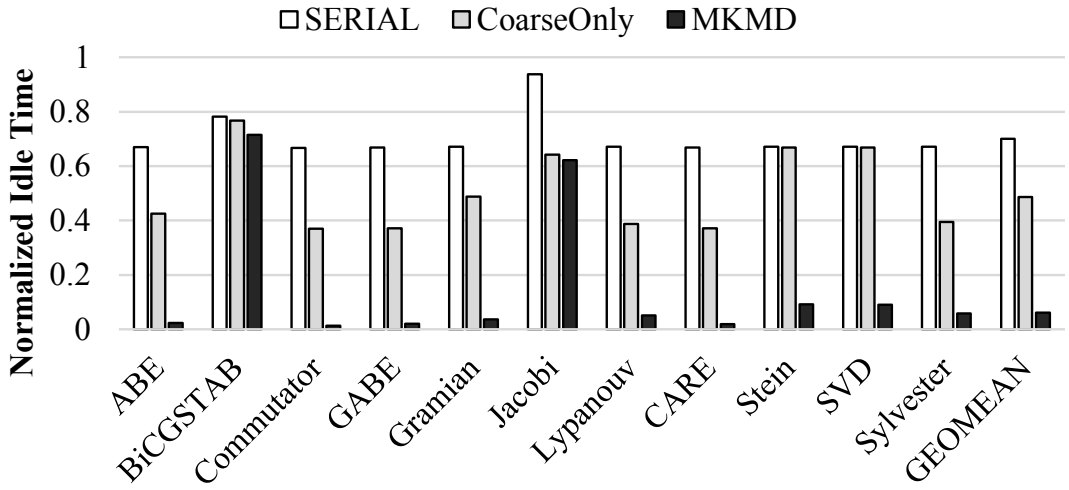
**Table 5.3:** Benchmark Specification

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 [56]. The equations and their domains are listed in Table 5.3. The sizes of vectors and matrices used in the equations are $4K$ and $4K \times 4K$, respectively.




**(a)** Speedup over in-order OpenCL executions




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

**Figure 5.8:** (a) Speedup of MKMD over in-order executions, and (b) the average device idle time normalized to the finish time.

### 5.5.1 Results

First, we measured the speedup of MKMD over in-order execution. As shown in Figure 5.8(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 5.4.3. In geometric mean, MKMD brings 89% performance improvement over in-order single device execution. Among 89% performance

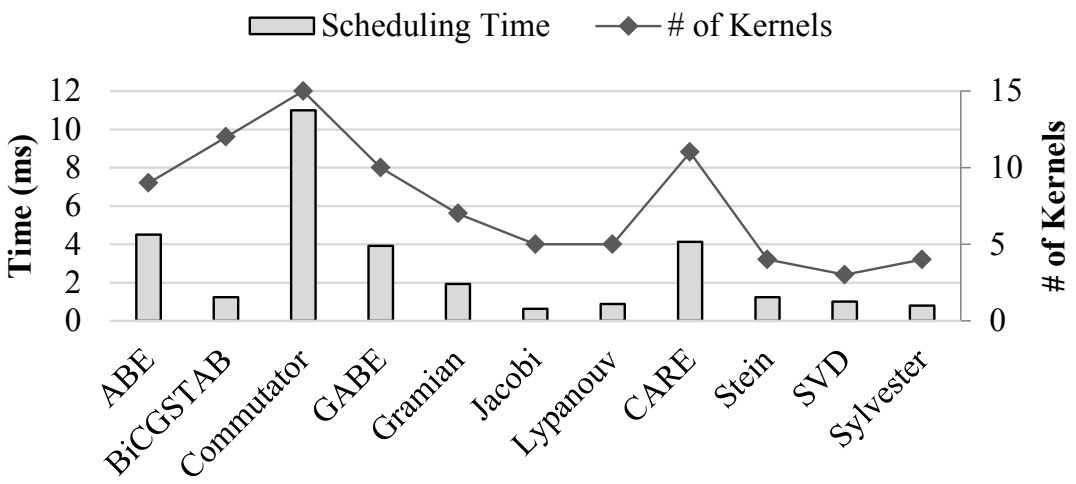



**Figure 5.9:** MKMD scheduling overhead.

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 5.8(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 5.8(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 5.5.3.

Figure 5.9 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 12.8% and 2.9%, respectively, and the other benchmarks have less than 0.2%. The main reason why BiCGSTAB and Jacobi have relatively large overhead is that they finish in a very short time (less than 35 msec). As discussed in Section 5.4.3, 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 5.9 shows such pattern.

### 5.5.2 Sensitivity to Profiles

This section examines the relation between the number of profiles and execution time estimation errors, and shows how the number of profiles affects the final scheduling result.

In order to measure the error rate of models from different number of profiles, we built models using 4, 10, 16, 20, 30, 40, and 80 profiles. For each set of profiling, we used two different input sizes, 4K and 8K for vectors, and 1K×1K and 2K×2K for matrices. Based on these two inputs, we varied the number of work-groups with an increment of $\frac{\#ofprofiles}{2}$.

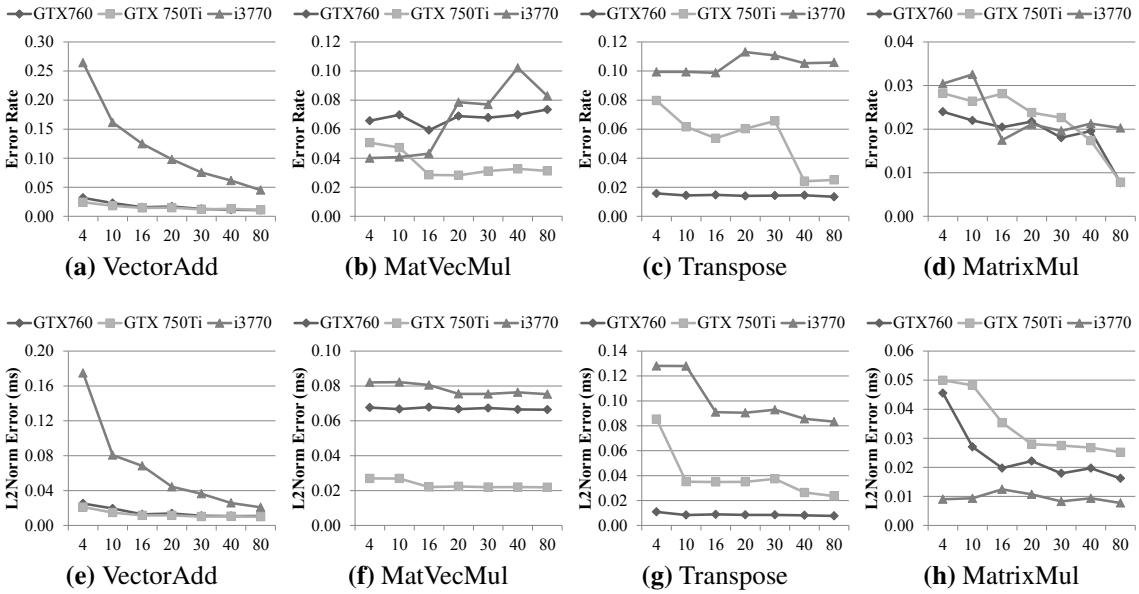



**Figure 5.10:** Error rates and L2-norm error in milliseconds varying the number of profiles for the execution time modeling. CPU has relatively high error rates on memory-intensive kernels as shown in (a), (b), and (c), but the execution time of these kernels is trivial as they do not have much computation. As a result, the absolute error (L2-norm) in time is also small as illustrated in (d), (e), (f), and (g).

Once we built the models with different number of profiles, we ran 100 executions with random input sizes ranging from 2K to 8K for vectors, and 1K×1K to 4K×4K for matrices. Also, random number of work-groups were used for various workloads. With these random workloads, we compared the real execution time with estimated execution time.

Figure 5.10 shows error rates and L2-norm error in milliseconds. L2-norm error is Euclidean distance between the real execution time and estimated execution time. As shown in the figure, memory intensive kernels, such as VectorAdd, MatVecMul, and Transpose, have relatively high error rates on CPU. However, the executions of those kernels finish in a very short time as they do not have much computation. Thus, the absolute error in millisecond becomes less than 0.2 ms even for the model with 4 profiles.

In order to examine the sensitivity of MKMD to errors, we ran the benchmarks on

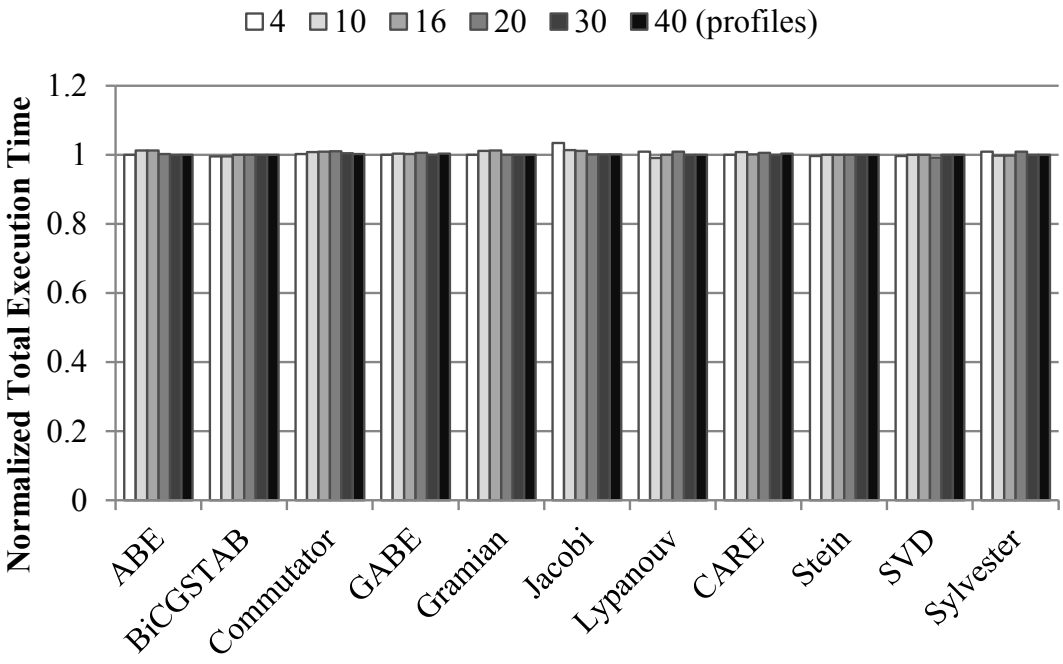



**Figure 5.11:** MKMD total execution time with different timing models varying the number of profiles. The baseline is the execution time scheduled with the model from 80 profiles. This result shows that the entire scheduling time is not sensitive to the number of profiles.

MKMD framework with various execution time models from different number of profiles. Figure 5.11 shows the entire execution time of MKMD with different models varying the number of profiles, normalized to the execution time with the model from 80 profiles. As shown in the figure, the overall performance of MKMD is not sensitive to the number of profiles which affects error rates and L2-norm errors. One of reasons is that the absolute error in time (L2-norm) is negligible as shown in Figure 5.10, which results in similar scheduling decisions for compute-intensive benchmarks. For memory-intensive benchmarks, such as BiCGSTAB and Jacobi, MKMD also produces similar schedules being insensitive to the errors, because the scheduling decision is dominated by data transfers, not computation for memory-intensive kernels.

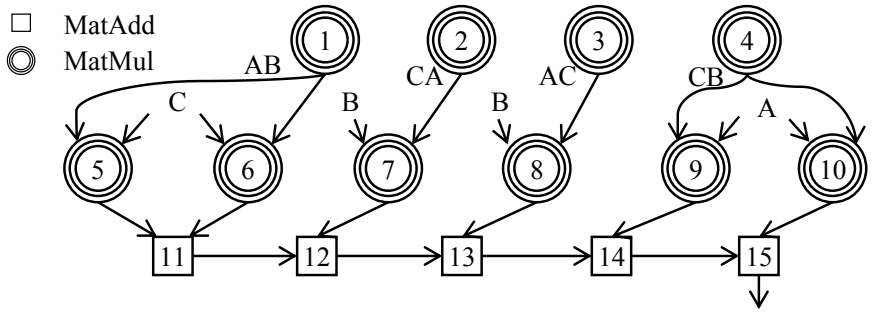
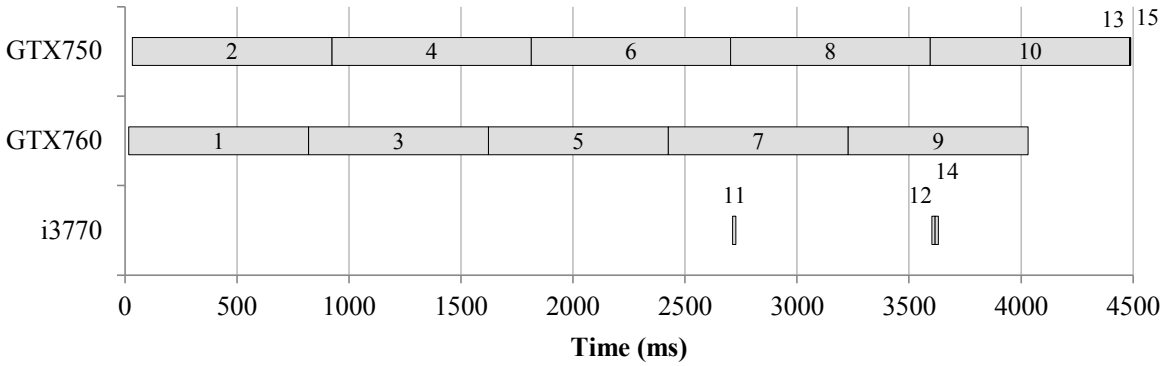



**Figure 5.12:** Kernel graph for triple commutator.

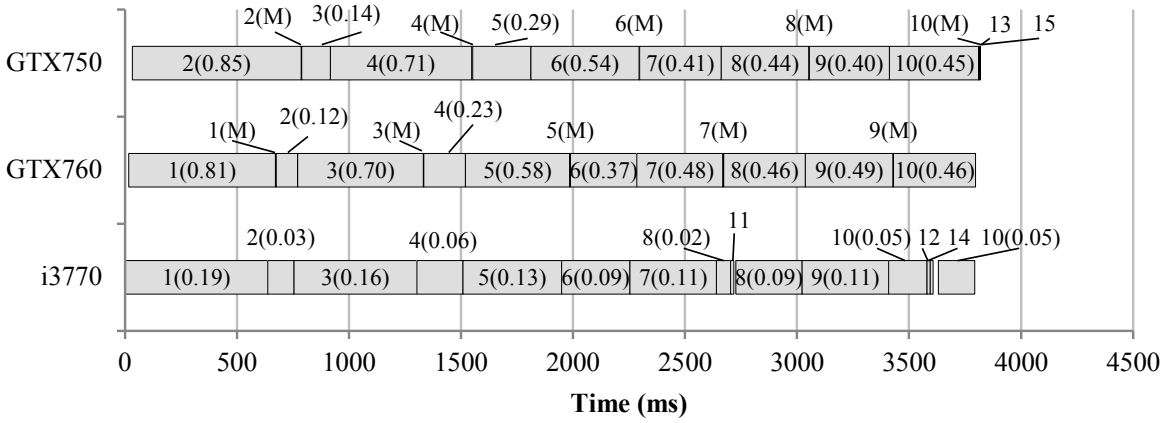
### 5.5.3 Case Study

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 5.12, and the execution timeline is depicted in Figure 5.13. 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 5.4. 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 5.13(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 5.13(a). The reason is that assigning the entire kernel to the CPU will increase the schedule length more than assigning it to GPUs even if several kernels are already assigned to them.

Based on coarse-grain scheduling results in Figure 5.13(a), MKMD starts multi-kernel partitioning as discussed in Section 5.4.3. 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%




**(a)** Coarse-grain schedule only




**(b)** MKMD

**Figure 5.13:** 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.

of work-groups) as shown in Figure 5.13(b). In the end, MKMD almost fully utilizes all three devices as shown in Figure 5.13 by splitting kernels into sub-kernels, executing them out of order without breaking the consistency.

## 5.6 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 [20, 62]. Using StreamIt [20], [19] pro-

posed 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 [41]. 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. [62] 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 [49, 38, 40, 44, 64]. [49] 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. [40] 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. [38, 44, 14, 64, 32] 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 [67, 68]. PTask [67] proposes APIs that work with operating systems to manage tasks on GPUs by using a data-ow programming model. Dandelion [68] 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 [67]. 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, [13] proposed the Harmony system, which schedules data parallel kernels considering the performance of device. [3] proposed StarPU system, which also schedules multiple data parallel kernels on heterogeneous devices. [21] 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 5.5. [57] 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.

## 5.7 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 chapter 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.

# CHAPTER VI

# Conclusion

## 6.1 Summary

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 have grown 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.

In this thesis, we showed three dynamic compiler frameworks that virtualize data parallel computing devices for portability, productivity, and performance of OpenCL applications. In Chapter III, we showed SKMD that virtualizes computing units and interconnects, and transparently orchestrates the execution of a single OpenCL kernel on multiple devices. With the experiments on a real machine with Intel i3770, NVIDIA GTX 750 Ti, and NVIDIA GTX 760, SKMD showed that it transparently utilizes all the underlying devices achieving an average speedup of 28% on a system with one multicore CPU and two asymmetric GPUs compared to a fastest device execution strategy for a set of popular OpenCL

kernels.

Next, in Chapter IV, we showed another framework, VAST that virtualizes the memory space of one physical computing device such as GPUs. While SKMD partitions a kernel for multiple devices, VAST splits a kernel for multiple executions for a single device, but executes sub-kernels multiple times with subsets of data. The experiments with NVIDIA GTX 750 Ti showed that VAST successfully executes kernels that have memory foot prints larger than GPU's physical memory.

Last, in Chapter V, we showed MKMD that orchestrates the execution of multiple data parallel kernels on multiple devices. While SKMD and VAST focus on virtualizing the hardware resources for the execution of a single data parallel kernel, MKMD provides temporal schedules for multiple kernels considering inter-kernel dependencies, and makes better spatial partitioning decisions across multiple devices with regards to the kernel dependencies and the physical interconnect among devices. Through the experiments on the real machine with Intel i3770, NVIDIA GTX 750 Ti, and NVIDIA GTX 760, we showed that MKMD achieves a mean 1.89x speedup over in order execution on the fastest device for a set of multi-kernel benchmarks.

In summary, this thesis addressed important issues of portability, productivity, and performance for emerging data parallel applications on data parallel platforms. By virtualizing computing units, memory space, and the interconnect of data parallel systems, the proposed frameworks significantly improved productivity, portability and performance of data parallel applications on multiple heterogeneous devices.

## 6.2 Future Directions

As data to process grows continuously and the need for the computing power keeps increasing. In response, many data parallel systems are scaled to a cluster consisting of multiple nodes, each of which has several data parallel devices.

A natural future direction to extend this work is to make the framework scalable to a cluster. The scheduling and partitioning problems become more complicated as the interconnect between the node is different from the one inside the node. In addition, latency of communication between nodes should be considered.

Although the works presented in this dissertation showed that the frameworks successfully virtualize data parallel platforms for OpenCL applications, it schedules and partitions multiple kernels on devices based on a static approach, which makes all the decisions before it actually launch kernels. As a result, it required considerable amount of offline profile data to model the execution time of kernels on each device. Also, it made an assumption that all hardware resources are exclusive to the application. However, it is general that multiple application shares hardware resources, and thus some hardware may no be available to the application even though it was available at the time of scheduling and partitioning.

To address this issue, the extension of this work could be the investigation of dynamic scheduling and partitioning, which are more adaptive to dynamic status of the system. Also, as the scheduling and partitioning decision is made focusing on the system status, a dynamic approach does not have to model the execution time. However, a dynamic approach may have worse performance because the decision should be made with narrower scope.

While this dissertation mainly targets the OpenCL runtime, more opportunities can be found throughout the software stack in order to improve performance, portability, and productivity of data parallel kernels. For example, if multiple processes contend for a limited number of processing devices, scheduling at OS level can improve the system performance. On the other hand, exploration of simplifying programming models can be a possible extension of this work to further improve portability and productivity of data parallel applications on parallel hardware.

For several years, OpenCL and CUDA have been improved to provide with more functionalities. One recent improvement in OpenCL and CUDA is sub-device [37] and Hyper-Q [58], which enables splitting a device into several small devices so that multiple kernels can run concurrently on one physical device. Another future direction of this work would be the investigation of utilizing t functionality along with MKMD as it can improve responsiveness of a data parallel kernel among multiple kernels.

# BIBLIOGRAPHY

# BIBLIOGRAPHY


[1] Intel(r) sdk for opencl applications 2013, 2013. http://software.intel.com/en-us/vcsource/tools/opencl-sdk. 75, 76

[2] AMD. Accelerated Parallel Processing (APP) SDK, 2012. http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk. 37

[3] 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. 120

[4] A. Basumallik and R. Eigenmann. Optimizing irregular shared-memory applications for distributed-memory systems. In *Proc. of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 119–128, 2006. 85

[5] A. Bensoussan, C. T. Clingen, and R. C. Daley. The Multics virtual memory: concepts and design. *Communications of the ACM*, 15(5):308–318, May 1972. 53

[6] 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. 100

[7] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *Proc. of the 20th International Conference on Parallel Architectures and Compilation Techniques*, pages 89–100, 2011. 49

[8] J. Canny and H. Zhao. Big Data Analytics with Small Footprint: Squaring the Cloud. In *Proc. of the 19th International Conference on Knowledge Discovery and Data Mining*, pages 95–103, 2013. 1, 52

[9] 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. 98

[10] Clang. A C language family frontend for LLVM, 2014. http://clang.llvm.org. 74, 110

[11] 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. 94

[12] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proc. of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 353–364, Sept. 2010. 12, 20, 48

[13] 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. 13, 49, 120

[14] 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. 119

[15] L. Einav and J. Levin. The data revolution and economic analysis, 2013. 52

[16] J. Fung and S. Mann. Computer vision signal processing on graphics processing units. In *Proc. of the 2004 IEEE International Conference on Acoustics Speech and Signal Processing*, pages 805–808, 2004. 1

[17] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, pages 407–420, 2007. 12

[18] M. Garey and D. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990. 30

[19] 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. 118

[20] 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. 118

[21] 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. 120

[22] 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. 94

[23] 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. 94

[24] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. Zheng. Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In *Proc. of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 205–216, Sept. 2010. 12, 48

[25] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proc. of the 36th Annual International Symposium on Computer Architecture*, pages 152–163, 2009. 48

[26] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. Sponge: portable

stream programming on graphics engines. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 381–392, 2011. 12, 83

[27] Intel. Intel Core i7-3700 Desktop Processor Series, 2012. http://download.intel.com/support/processors/corei7/sb/core_i7-3700_d.pdf. 75, 110

[28] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August. Dynamically managed data for CPU-GPU architectures. In *Proc. of the 2012 International Symposium on Code Generation and Optimization*, pages 165–174, 2012. 83

[29] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. Automatic CPU-GPU communication management and optimization. In *Proc. of the '11 Conference on Programming Language Design and Implementation*, pages 142–151, 2011. 83

[30] F. Ji, H. Lin, and X. Ma. RSVM: A region-based software virtual memory for GPU. In *Proc. of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 269–278, 2013. 85

[31] W. Jia, K. A. Shaw, and M. Martonosi. Stargazer: Automated Regression-based GPU Design Space Exploration. In *Proc. of the 2012 IEEE Symposium on Performance Analysis of Systems and Software*, pages 2–13, 2012. 48

[32] 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. 119

[33] R. Karrenberg and S. Hack. Whole-Function Vectorization. In *Proc. of the 2011 International Symposium on Code Generation and Optimization*, Apr. 2011. 12, 48

[34] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: First-Class GPU Resource Management in the Operating System. In *Proc. of the USENIX Annual Technical Conference (USENIX ATC'12)*, pages 401–412, 2012. 50

[35] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31(5):7–17, 2011. 50

[36] C. Kessler, U. Dastgeer, S. Thibault, R. Namyst, A. Richards, U. Dolinsky, S. Benkner, J. L. Traff, and S. Pllana. Programmability and performance portability aspects of heterogeneous multi-/manycore systems. In *Proc. of the 2012 Design, Automation and Test in Europe*, pages 1403–1408, Mar. 2012. 13, 49

[37] KHRONOS. OpenCL - the open standard for parallel programming of heterogeneous systems, 2014. 1, 35, 125

[38] 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. 13, 40, 49, 83, 119

[39] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. SnuCL: an OpenCL framework for

heterogeneous CPU/GPU clusters. In *Proc. of the 2012 International Conference on Supercomputing*, pages 341–352, 2012. 83

[40] 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. 83, 119

[41] 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. 31, 119

[42] 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. 36, 74, 110

[43] J. Lee, M. Samadi, and S. Mahlke. VAST: the illusion of a large memory space for GPUs. In *Proc. of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, pages 443–454, 2014. 6, 50

[44] 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. 5, 64, 83, 119

[45] J. Lee, H. Wu, M. Ravichandran, and N. Clark. Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. In *Proc. of the 37th*

*Annual International Symposium on Computer Architecture*, pages 270–279, 2010. 30

[46] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proc. of the 37th Annual International Symposium on Computer Architecture*, pages 451–460, 2010. 1, 12, 48

[47] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: a programming model for heterogeneous multi-core systems. In *13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 287–296, 2008. 49

[48] LLVM. libclc, 2012. http://libclc.llvm.org. 37

[49] 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. 13, 40, 49, 83, 119

[50] C. McGeoch, P. Sanders, R. Fleischer, P. R. Cohen, and D. Precup. *Experimental Algorithmics*. Springer-Verlag New York, Inc., New York, NY, USA, 2002. 94

[51] W. mei W. Hwu. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann Publishers Inc., 2011. 73

[52] S.-J. Min and R. Eigenmann. Optimizing irregular shared-memory applications for

clusters. In *Proc. of the 2008 International Conference on Supercomputing*, pages 256–265, 2008. 85

[53] D. Montgomery, E. Peck, and G. Vining. *Introduction to linear regression analysis*. Wiley series in probability and statistics. Wiley, New York, NY [u.a.], 3. ed edition, 2001. 26

[54] S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997. 100

[55] NVIDIA. GPUs Are Only Up To 14 Times Faster than CPUs says Intel, 2010. http://blogs.nvidia.com/ntersect/2010/06/gpus-are-only-up-to-14-times-faster-than-cpus-says-intel.html. 1

[56] NVIDIA. CUDA Toolkit 4.2, 2012. https://developer.nvidia.com/cuda-toolkit-42-archive. 37, 75, 97, 111

[57] NVIDIA. NVIDIA's next generation CUDA compute architecture: Kepler GK110, 2012. www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf. 120

[58] NVIDIA. *CUDA C Programming Guide*, 2014. http://docs.nvidia.com/cuda. 1, 50, 63, 73, 75, 85, 125

[59] NVIDIA. NVIDIA GPU Computing SDK, 2014. http://developer.nvidia.com/gpu-computing-sdk. 1

[60] NVIDIA. Ptx: Parallel thread execution isa, 2014. http://docs.nvidia.com/cuda/parallel-thread-execution. 20

[61] 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. 94

[62] OpenACC. Directives for accelerators, 2014. http://www.openacc.org. 84, 118, 119

[63] D. A. Padua and M. J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12):1184–1201, Dec. 1986. 35

[64] 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. 119

[65] B. Pichai, L. Hsu, and A. Bhattacharjee. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 743–758, New York, NY, USA, 2014. ACM. 85, 86

[66] J. R. Quinlan. Induction of decision trees. *Journal of Machine learning*, 1(1):81–106, Mar. 1986. 31

[67] 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. 50, 119, 120

[68] 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. 50, 119

[69] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82, 2008. 52

[70] M. Samadi, A. Hormati, M. Mehrara, J. Lee, and S. Mahlke. Adaptive input-aware compilation for graphics engines. In *Proc. of the '12 Conference on Programming Language Design and Implementation*, pages 13–22, 2012. 83

[71] D. Sayre. Is automatic "folding" of programs efficient enough to displace manual? *Communications of the ACM*, 12(12):656–660, Dec. 1969. 53

[72] S. Sharma, R. Ponnusamy, B. Moon, H. Yuan-Shin, R. Das, and J. Saltz. Run-time and compile-time support for adaptive irregular problems. pages 97–106, 1994. 85

[73] 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. 100

[74] J. Silk and G. A. Mamon. The current status of galaxy formation. *Research in Astronomy and Astrophysics*, 12(8):917–946, 2012. 52

[75] J. A. Stratton, S. S. Stone, and W.-M. W. Hwu. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In *Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 16–30, 2008. 12, 48

[76] Y. Suzuki, S. Kato, H. Yamada, and K. Kono. GPUvm: Why Not Virtualizing GPUs at the Hypervisor? In *Proc. of the USENIX Annual Technical Conference (USENIX ATC'14)*, pages 109–120, June 2014. 50

[77] 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. 100

[78] L. Torczon and K. Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., 2nd edition, 2011. 24, 65

[79] K. Wang, X. Ding, R. Lee, S. Kato, and X. Zhang. GDM: Device Memory Management for Gpgpu Computing. In *2014 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer System*, pages 533–545, 2014. 50

[80] X. Wu, X. Zhu, G.-Q. Wu, and W. Ding. Data Mining with Big Data. *IEEE Transactions on Knowledge and Data Engineering*, 26(1):97–107, Jan. 2014. 52

[81] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU compiler for memory optimization and parallelism management. In *Proc. of the '10 Conference on Programming Language Design and Implementation*, pages 86–97, 2010. 83

[82] D. Zaparanuks and M. Hauswirth. Algorithmic profiling. In *Proc. of the '12 Conference on Programming Language Design and Implementation*, pages 67–76, 2012. 94