SKMD: Single Kernel on Multiple Devices for Transparent CPU-GPU Collaboration

JANGHAENG LEE and MEHRZAD SAMADI, University of Michigan
YONGJUN PARK, Hongik University
SCOTT MAHLKE, University of Michigan

9

Heterogeneous computing on CPUs and GPUs has traditionally used fixed roles for each device: the GPU handles data parallel work by taking advantage of its massive number of cores while the CPU handles non data-parallel work, such as the sequential code or data transfer management. This work distribution can be a poor solution as it underutilizes the CPU, has difficulty generalizing beyond the single CPU-GPU combination, and may waste a large fraction of time transferring data. Further, CPUs are performance competitive with GPUs on many workloads, thus simply partitioning work based on the fixed roles may be a poor choice. In this article, we present the single-kernel multiple devices (SKMD) system, a framework that transparently orchestrates collaborative execution of a single data-parallel kernel across multiple asymmetric CPUs and GPUs. The programmer is responsible for developing a single data-parallel kernel in OpenCL, while the system automatically partitions the workload across an arbitrary set of devices, generates kernels to execute the partial workloads, and efficiently merges the partial outputs together. The goal is performance improvement by maximally utilizing all available resources to execute the kernel. SKMD handles the difficult challenges of exposed data transfer costs and the performance variations GPUs have with respect to input size. On real hardware, SKMD achieves 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.

CCS Concepts: • **Software and its engineering** → **Runtime environments;** *Incremental compilers*; • **Computing methodologies** → *Parallel programming languages;*

Additional Key Words and Phrases: Compiler, runtime, CPU, GPU, collaboration, optimization

ACM Reference Format:
Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. 2015. SKMD: Single kernel on multiple devices for transparent CPU-GPU collaboration. ACM Trans. Comput. Syst. 33, 3, Article 9 (August 2015), 27 pages.
DOI: http://dx.doi.org/10.1145/2798725

1. INTRODUCTION

Heterogeneous computing that combines traditional processors (CPUs) with graphic processing units (GPUs) has become the standard in most systems from cell phones to servers. GPUs achieve higher performance by providing a massively parallel

This research was supported by National Science Foundation grants CNS-0964478 and SHF-1217917, and the Defense Advanced Research Projects Agency under the Power Efficiency Revolution for Embedded Computing Technologies (PERFECT) program. This article extends an earlier version that appeared in the Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques. (PACT'13) [Lee et al. 2013].
Authors' addresses: J. Lee, M. Samadi, and S. Mahlke, 2260 Hayward St, Computer Science and Engineering Department, University of Michigan, Ann Arbor, MI, USA; emails: {jhaeng, mehrzads, mahlke}@umich.edu; Y. Park, P501, 94, Wausan-ro, Mapo-gu, Department of Electronic and Electrical Engineering, Hongik University, Seoul, Korea; email: yongjun.park@hongik.ac.kr.

DOI: http://dx.doi.org/10.1145/2798725

architecture with hundreds of relatively simple cores while exposing parallelism to the programmer. By leveraging new programming models, such as OpenCL [KHRONOS 2014] and CUDA [NVIDIA 2014a], programmers are able to effectively develop highly threaded data-parallel kernels to execute on the GPUs. 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 [Lee et al. 2010; Stratton et al. 2008; Diamos et al. 2010; Gummaraju et al. 2010; Karrenberg and Hack 2011; Hormati et al. 2011; Fung et al. 2007].

More recently, systems are configured with several different types of processing devices, such as CPUs with integrated GPUs and multiple discrete GPUs 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 [Diamos and Yalamanchili 2008]. This approach requires programmer effort to ensure there are no interkernel 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 [Luk et al. 2009; Kim et al. 2011; Kessler et al. 2012]. Luk et al. [2009] proposed the Qilin system, which 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. [2011] 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 architectures, 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 1, a system may have several GPUs that have different performance and memory bandwidth characteristics. In addition, some computing devices, such as CPUs or integrated GPUs, can share

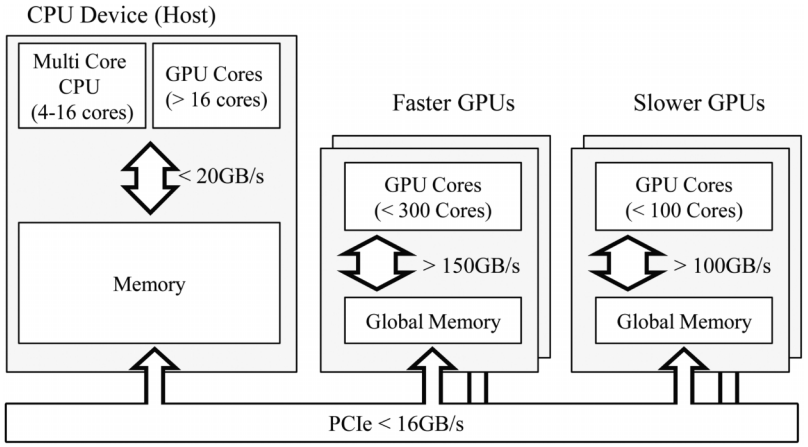


Fig. 1. Physical OpenCL computing devices with different performances, memory spaces, and bandwidths.

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, in which 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 article, we propose single kernel multiple devices (SKMDs), 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 article makes the following contributions:

—The SKMD runtime system, which 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

2. BACKGROUND

This section briefly describes the OpenCL programming and execution model, then discusses memory consistency of OpenCL to understand semantics of partitioning on a single data-parallel kernel.

2.1. OpenCL Programming Model and Execution Model

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, that is, *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 multiprocessors (SMs) by quickly switching context of a work group to another using a hardware scheduler.

2.2. Memory Consistency and Multidevice Execution

The OpenCL programming model uses relaxed memory consistency model 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 collaborative execution of a single kernel even on separate devices that use different address spaces. By simply assigning a subset of work groups to each device exclusively, partial results would appear interleaved in their address spaces. The final result can be made when the partial results are properly merged.

Another approach for multidevice execution is that several GPUs directly access the host application's address space with the support from the device driver and operating systems [NVIDIA 2014a]. However, on-demand accesses have the following limitations:

—The host application must secure contiguous physical memory space so that GPUs can access the memory directly without the OS's memory management. This may cause slowdown of the system as it reduces the amount of available physical memory for paging.
—Discrete GPUs are attached to a PCI express (PCIe) bus, which has limited bandwidth and long latency (tens of microseconds). As a result, frequent accesses for a small chunk of data through a PCIe bus will cause significant overhead.

The combination of these factors makes on-demand accesses unfavorable for multi-GPU executions.

3. SKMD SYSTEM

An 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, an SKMD returns an illusion of virtual platform with only one large available device. An 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 2.

The Dynamic compiler has four units: *kernel transformer*, *buffer manager*, *partitioner*, and *performance predictor,* as shown in the gray boxes in Figure 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 merged. In order to merge irregular locations of output from different devices, the kernel transformer generates the *merge* kernel, and the 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 on offline profile data. If the profile information does not exist, the 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, the SKMD launches the actual kernel for each device.

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

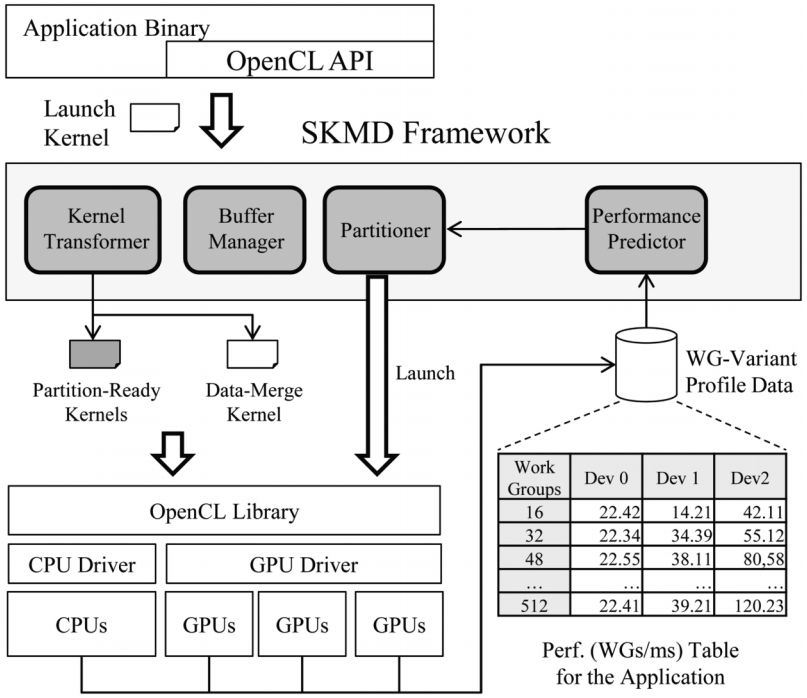


Fig. 2. The SKMD framework consisting of four units: kernel transformer, buffer manager, partitioner, and profile database.

3.1. 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(a) shows three-dimensional ranges, each of which has 8 work groups. Figure 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(c). Based on this idea, the next section discusses how the 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. 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, the 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. After flattening, each work item identifies its work-group index ($flattened\_id$) and determines whether it is allowed to execute the kernel.

The additional code with gray background is lowered to 3 to 9 instructions in PTX and x86-64 ISAs. These additional instructions consist of loading indices and dimension sizes, 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 group [NVIDIA 2014b]. Nonetheless, these instructions can be unnecessary overhead for disabled work groups

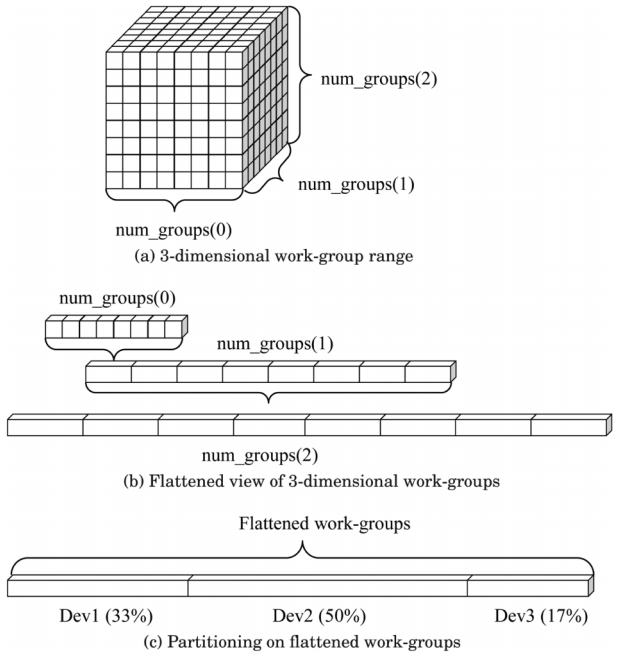


Fig. 3. OpenCL's N-dimensional range.

```
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 }
```

Fig. 4. Partition-ready Blackscholes kernel.

in GPUs. To estimate this overhead, *VectorAdd* was tested with NVIDIA GTX 760 by enabling only 1 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-group. 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. [2010] proposed. In their work, the driver transforms a kernel to be wrapped by

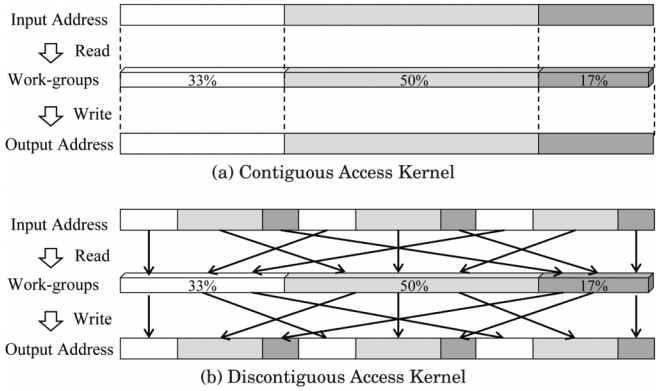


Fig. 5. Different memory access patterns of kernels.

N-nested loops in order for CPUs to execute N-dimensional work items in a work group. This is 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. This leaves CPU execution inefficient for the *Partition-ready* kernel, however, 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, thus the SKMD system does not transform a kernel but the driver selectively iterates over work groups. Through this loop-independent code motion, SKMD eliminates the overhead of checking code within the kernel code.

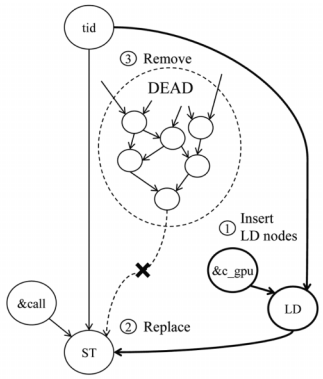
Merge Kernel: Another challenge of collaborative execution of a single data-parallel kernel is that several computing devices may use different address spaces, thus the results from each device *must* be merged after execution. Some kernels have contiguous memory accesses, called, *contiguous* kernel, in which each of the threads writes the result in contiguous locations, as shown in Figure 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 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, thus each device has to keep the write location for merging and selectively copies the data afterward.

To solve this problem, the 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 multicore CPUs. The 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

```
1  __kernel void Blackscholes(
2          __global float *call,
3          __global float *put,
4          __global float *price,
5          __global float *strike,
6          float r,
7          float v)
8  {
9      int tid = get_global_id(1) *
10         get_global_size(0) + get_global_id(0);
11     float c, p;
12     BlackScholesBody(&c, &p,
13                      price[tid],
14                      strike[tid],
15                      r, v);
16     call[tid] = c;
17     put[tid] = p;
18 }
```

(a) Blackscholes Kernel



(b) Data Flow Graph

```
1  __kernel void Blackscholes_Merge(
2          __global float *call,
3          __global float *put,
4          __global float *c_gpu,
5          __global float *p_gpu,
6          int GPU_from, int GPU_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 flat_id = idx + idy * size_x;
12     // check whether to execute
13     if (flat_id < GPU_from || flat_id > GPU_to)
14         return;
15     int tid = get_global_id(1) *
16               get_global_size(0) + get_global_id(0);
17     // computation code is removed by DCE
18     call[tid] = c_gpu[tid];
19     put[tid] = p_gpu[tid];
20 }
```

(c) Merge Kernel for Blackscholes

Fig. 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 the GPU's partial results, and then proceed with dead code elimination (b). As a result, the merge kernel does not have computational part (c).

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 6(a) shows the original Blackscholes kernel that generates two output arrays (`call` and `put`). For the merge kernel shown in Figure 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 dataflow 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 the *partition-ready* kernel as shown at Line 13 in Figure 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 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 and 20 of Figure 6(c). Finally, it marks store instruction as *live* and proceeds with *dead code elimination* using the mark-sweep algorithm [Torczon and Cooper 2011] to remove all computation code. As a result of this transformation, all computation code, Lines 11 through 15 in Figure 6(a), are removed. Note that every function call is inlined before the transformation in order to avoid expensive interprocedural analysis.

Clearly, the 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 (>20GB/s with DDR-3) regardless of application. That is, 67MB 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 a large amount of data, merging in the host can be a bottleneck. In this case, the SKMD does not partition a kernel across multiple devices as the partitioning algorithm considers merging cost, which is discussed in detail in Section 3.4.

3.2. Buffer Management

In the SKMD framework, the buffer manager is in charge of transferring I/O back and forth between the host and external devices. Since the main idea of the 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 6GB/s), it becomes critical to reduce the amount of transferring I/O for external GPU devices.

To determine if it is safe to transfer partial data to GPU devices, the buffer manager checks whether the kernel is a contiguous kernel by analyzing index space of each work group. For index space analysis, the buffer manager uses dataflow analysis focusing on the index operand of store and load instructions, which is represented as *tid* in Figure 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 delineated earlier, the buffer manager gives up optimizing data transfer and defines it as a discontiguous kernel. In this case, the entire I/O 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.

3.3. Performance Prediction

After the kernel transformations, the 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, the 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 I/O, value of scalar parameters, and NDRange information. However, the 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.

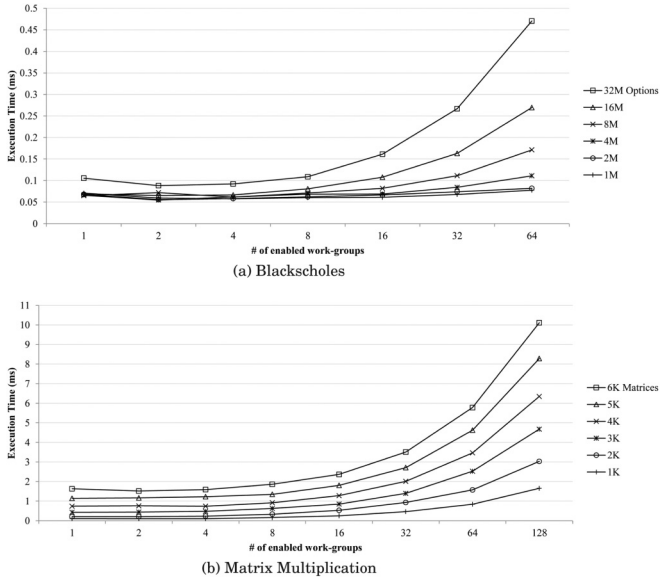


Fig. 7. Execution time varied by applications, input size, and the number of enabled work-groups. Depending on the application and input size, the number of enabled work-groups impacts on the execution time differently.

Figure 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, the SKMD utilizes a linear regression analysis model [Montgomery et al. 2001] using the following equation:

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

For the 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 variables. Those properties are the size of each global argument, 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 β_i and independent variables x_i, but the execution time in SKMD may not be represented as a simple combination of β_i and x_i as described in Equation (1). Figure 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 7(a). To handle these cases, transformations are applied to independent variables, as shown in Equation (2).

$$y = \beta_0 + \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{m} \beta_k f_k(x_i, x_j) + \epsilon \tag{2}$$

This equation is still a linear regression model since y is linear in the coefficients β_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 regarding the 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 at which the linearity appears is varied by application characteristics, as shown in Figure 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 converge from 0 to 1, the tan^{-1} function is divided by π; then 0.5 is added, as shown in Equation (3), where x is the number of work groups.

$$g(x) = \frac{tan^{-1}(a(x-b))}{\pi} + 0.5 \tag{3}$$

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 the 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.

To this end, a complete transformed function can be represented as Equation (4), where x_i is the number of work groups and x_j is another independent variable.

$$f(x_i, x_j) = x_i g(x_i) h(x_j) \tag{4}$$

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 the output buffer. Note that the SKMD tries various time complexity functions for $h(x_j)$, then the linear regression solver will eventually find the best transformed function by assigning a meaningful coefficient.

3.4. Transfer Cost and Performance Variation-Aware Partitioning

Once the performance model for each device is ready, the SKMD makes a decision regarding 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 [Garey and Johnson 1990] and a common problem in load-balancing parallel systems [Lee et al. 2010].

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 important, the performance of devices can vary as the number of work groups assigned to devices changes. To illustrate, Figure 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.

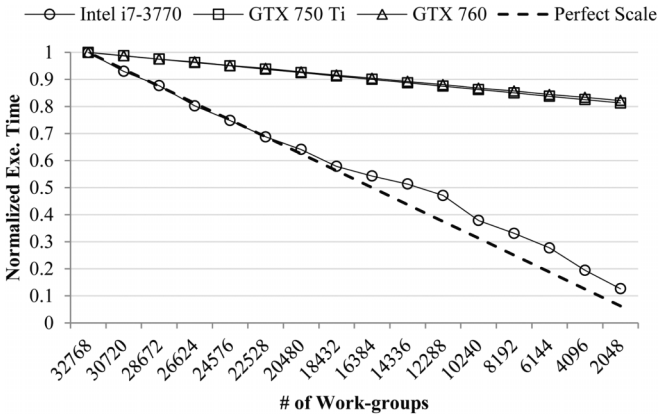


Fig. 8. Performance impact on VectorAdd varying the number of work groups. The execution time of GPUs does not scale down in spite of reduced number of work groups.

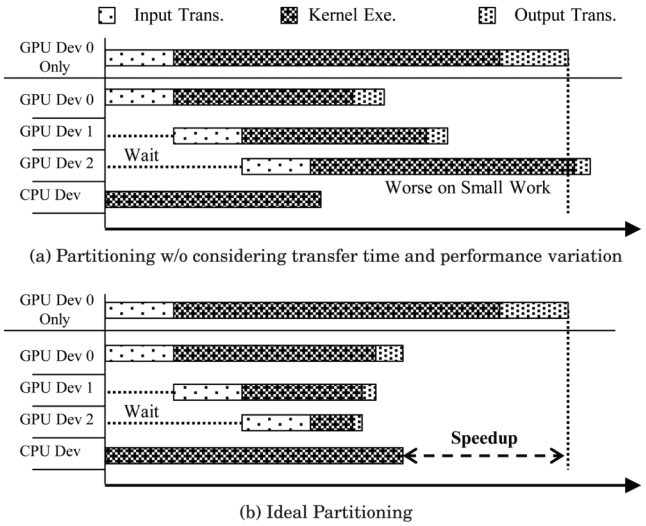


Fig. 9. Comparison of linear partitioning and ideal partitioning.

To illustrate, the example shown in Figure 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 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 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 the SKMD performs partitioning at runtime, thus the algorithm must be executed very quickly so as not to

overwhelm potential benefits from collaborative execution. This restriction prohibits the exact time-consuming integer programming solutions [Kudlur and Mahlke 2008].

To perform partitioning at runtime, the SKMD utilizes a decision tree heuristic [Quinlan 1986]. For our system, the SKMD uses a top-down induction tree, in which 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.

In detail, the partitioner loads the *linear regression equation* for performance prediction for each device. The performance equations for each device arecomputed 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, the 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 the 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 adds 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 the *balancing factor*, which is the difference between the overall execution time in the parent node 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 are no candidates, the partitioner increases $PartitionGranularity$ temporarily to make sure that the slowdown does not come from the performance variance. If

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
```

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 Lines 12 through 46 corresponds to traversing down the decision tree, and `For-Loop` at Lines 16 through 36 corresponds to testing children of a node in the tree.

Table I. Experimental Setup

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 Frequency	3.4 GHz	1.62 GHz	1.28 GHz
Memory	32GB DDR3 (1866)	2GB GDDR 5	2GB GDDR 5
Peak Performance	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		

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 earlier.

3.5. Limitations

As the 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 [Padua and Wolfe 1986]; 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 [KHRONOS 2014]. If an OpenCL compiler can analyze the atomic operations during compilation and detect whether 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 the SKMD system. The main reason is that an SKMD predicts the execution time based on a regression model, as discussed in Section 3.3, 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 (e.g., breath first search), the execution time is unpredictable only with the size of array. Because an SKMD partitions a kernel statically before distributing work groups, it is difficult to partition this kind of kernel optimally across several devices if the execution time is unpredictable. Applications with these semantics were not handled in this article, as an SKMD gives up partitioning if a kernel has array-value-dependent control flows.

4. EVALUATION

The SKMD was evaluated on a real machine that has three different types of computing devices, as shown in Table I. Intel Ivy Bridge has an integrated GPU, but it does not support OpenCL in UNIX-based operating systems, thus the integrated GPU is not considered to be a computing device in our experiments. However, the idea of an SKMD framework is not limited to discrete GPUs. The SKMD was prototyped using Low-Level

Table II. Benchmark Specification

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

VectorAdd, Blackscholes, BinomialOption, and ScanLargeArrays are classified as contiguous kernels, whereas others are defined as discontiguous kernels.

Virtual Machine (LLVM) 3.4 [Lattner and Adve 2004], on top of a Linux system with an NVIDIA driver for GPU execution, and an Intel OpenCL driver for the CPU execution.

Every function call to the OpenCL library was hooked by our custom library that leverages the SKMD's compilation framework. Inside the framework, we used Clang for the OpenCL front end, and LLVM 3.4 incorporated with `libclc` extension was used for the PTX back end [LLVM 2014]. However, the PTX back end 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 the Intel OpenCL driver may cause significant overhead, as discussed in Section 3.1. 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, 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 [AMD 2012] and the NVIDIA SDK [NVIDIA 2012] 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 II. The applications from the benchmark suite were compiled without any modification. In Table II, Histogram and Reduction were marked as *1st round*, because the OpenCL kernels are used for generating intermediate results, and the host applications finalize 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 set of data is divided by the number of work groups; 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 II. 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 II. 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, the SKMD requires an application to use *profiling mode* if there is not enough profile data. With profiling mode, the 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 3.3, the SKMD increases the number of work groups by four (*finer granularity*) until it reaches 16, then increases the granularity as the number of work groups grows. Once profile data is collected, the SKMD performs the linear regression analysis, as discussed in Section 3.3.

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, the SKMD increases partitioning granularity. 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 1*ms*, 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 the case of discontiguous access kernels. Because the CPU resource is shared with the OS 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.

4.1. Results and Analysis

Figure 10(a) shows speedup of the SKMD compared to the fastest single-device execution and the linear partitioning execution, which is similar to prior approaches [Luk et al. 2009; Kim et al. 2011]. In linear partitioning, the number of work groups assigned to each device is 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

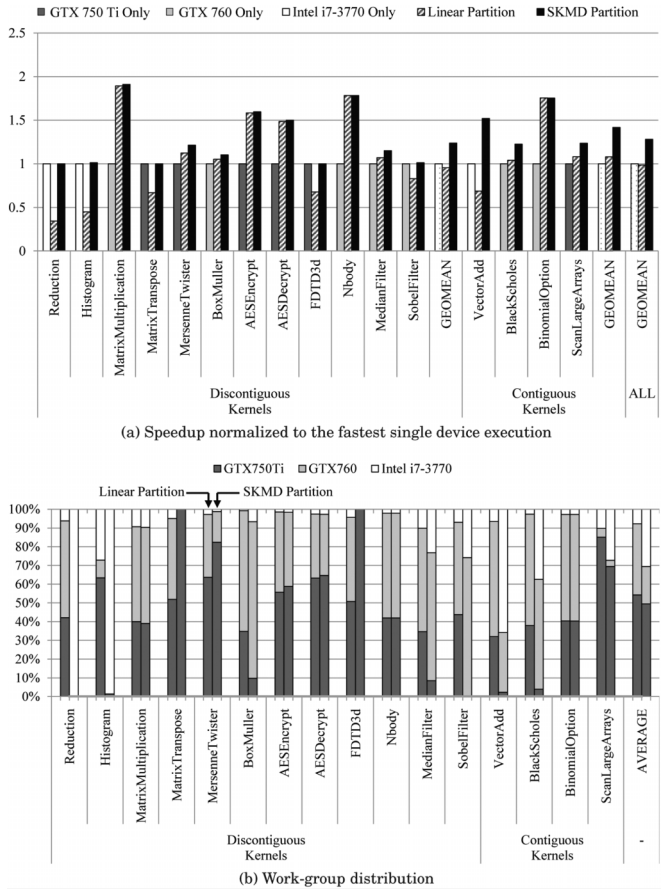


Fig. 10. Speedup and work-group distribution. Each benchmark has a different baseline (a), as the fastest device differs by kernels. The fastest device is determined with regard to the execution time and data transfer cost.

(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 10(a), the SKMD performs 28% faster than single-device execution on average as it considers the transfer cost and performance variation of each device during partitioning. An important point from this result is that the linear partitioning causes slowdown on memory-bound kernels compared to single-device execution. This is mainly because it does not take the transfer cost into account during partitioning although collaborative execution is not favorable due to the transfer cost.

To illustrate how the SKMD partitions work groups across different devices, Figure 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 linear partitioning. Linear partitioning makes a bad decision for memory-bound applications by assigning less workload to the CPU, although a 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, thus the SKMD assigns most of the work to the CPU device. The difference is that Reduction and Histogram are recognized as discontiguous kernels, therefore the host program must transfer the entire input to the external GPU device, which discourages collaborative execution due to the high 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 for which a significant amount of time is spent on computation, not memory accesses. For these benchmarks, the portion of the workload assigned to the GPUs is 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 the work to GTX 750 Ti despite the high 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 uncoalesced 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 an 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, thus a larger portion of workload is offloaded to the GPUs than VectorAdd.

Other benchmarks have a considerable amount of computations and large memory footprints with discontiguous access patterns. In this case, both compute and transfer costs are proportional to the size of data, therefore 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 10.

4.2. Execution Time Breakdown

In this section, we show how the SKMD transfers data between the CPU and GPUs, and assigns work groups to different devices. Figure 11 shows the execution time breakdown of three sample applications: VectorAdd, Matrix Multiplication, and Histogram.

For VectorAdd, CPU-only is the baseline because it is an extremely memory-bound kernel. As shown in Figure 11(a), the SKMD starts the execution on the CPU while transferring a huge amount of data to GTX 760 in the background. As soon as the data transfer is finished, the SKMD launches the kernel on GTX 760; 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, thus 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 11(b). Since Matrix Multiplication takes much more time in computation than VectorAdd, the impact of transferring time is less for this benchmark. However, the SKMD transfers

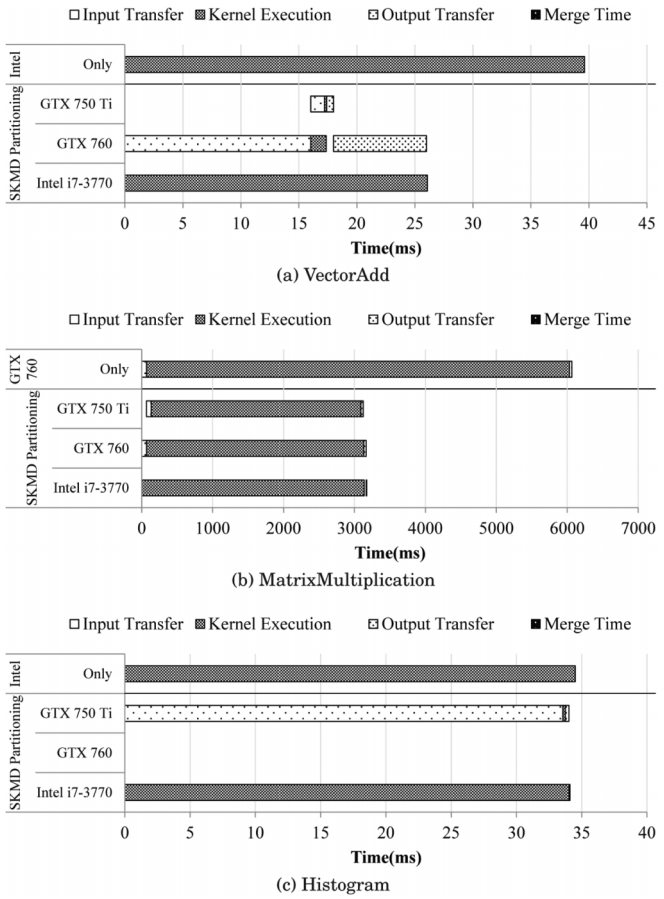


Fig. 11. Breakdown of the execution time on each device. The bars on the top show the baseline, which is the fastest single-device execution. The SKMD considers the transfer cost, and offloads work groups in order to balance the workload among the three devices.

the entire I/O back and forth between the host and GPU devices, as it is classified as a discontiguous kernel. Similar to the 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 a large input, which incurs high transfer cost for the GPU execution. In terms of execution performance, GTX 750 Ti outperforms GTX 760 for Histogram, as shown in Figure 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 II.

Histogram is categorized as a discontiguous kernel, thus the SKMD still has to transfer the entire input to the external GPU devices. As shown in Figure 11(c), the 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 the three devices. As the output size is small, the overhead of the merging kernel is negligible for this benchmark.

Table III. Profile Execution Parameters and Real Execution Parameters for Evaluating Performance Prediction Accuracy

Application	Profile Parameters		Real Parameters (same as Table II)
	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

For each profile, 16 sets of profile data were collected, varying the number of work groups.

4.3. 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 III. With the profiled data from two sets of execution parameters, the 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*in Table III. 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 12(a) shows the L2-Norm errors between predicted time and execution time; Figure 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 in *milliseconds*, while the average error rate shows the difference over the real execution time. For all benchmarks, high error ratios were observed when the SKMD predicts the execution time with a very low number of work groups. This is mainly because the execution time is very short with a low number of work groups. As a result, even small error values can result in high error ratios. For example, if the SKMD predicted the time as 0.011ms, but the real execution took 0.01ms, then the error ratio becomes 10% in spite of only 0.001ms of misprediction. Considering that the execution time for the execution parameters shown in Table III takes more than 10ms for all benchmarks, the errors in prediction time are negligible since the error remains under 0.1ms in most cases, as shown in Figure 12(a).

5. RELATED WORK

A significant focus has been on the execution of data-parallel applications on CPUs. Lee et al. [2010] examined several data-parallel applications to show that CPUs can

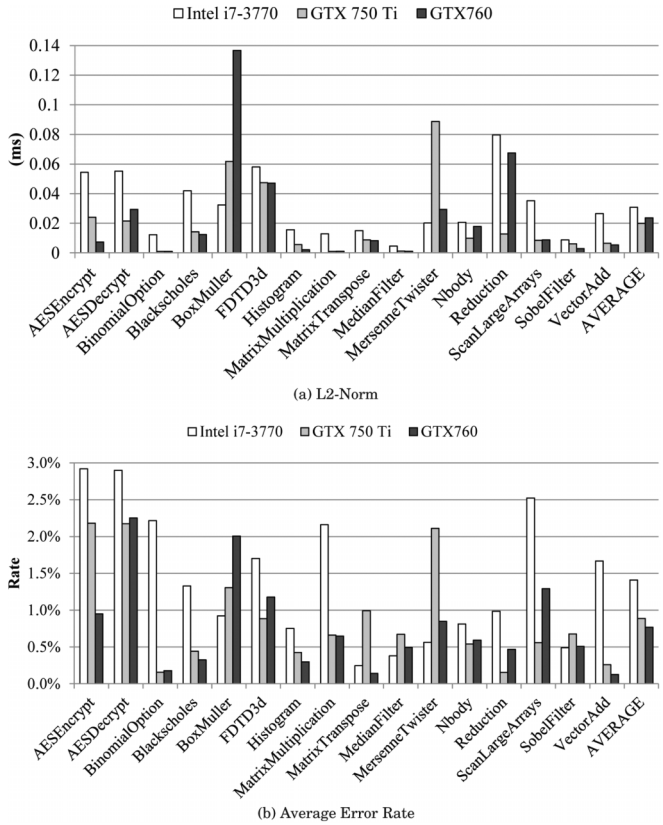


Fig. 12. Performance prediction accuracy. L2-Norm error (a) shows Euclidean distance between the real execution time and predicted execution time in milliseconds. Average error rate (b) shows the average percentage of errors in predictions.

have comparable performance to GPUs if they take full advantage of multicores with SIMD units. There has also been some work on efficient execution of OpenCL/CUDA applications on CPUs. Stratton et al. [2008] proposed a source-to-source compiler that translates a CUDA program into a standard C program using a loop-fission technique to eliminate *synchronization*. Similarly, Diamos et al. [2010] developed the Ocelot, a runtime system that dynamically transforms OpenCL/CUDA kernels for CPU execution. Gummaraju et al. [2010] performed a similar study, but approached it in a lightweight thread (LWT) execution model. In a similar fashion, Karrenberg and Hack [2011] have focused on more efficient execution of OpenCL applications using whole-function vectorization. All of these works are focusing on performance improvement on CPUs to show that CPUs can perform as well as GPUs for some applications, but none deals with collaborative execution with GPUs.

Performance modeling of GPUs for a certain set of applications has been studied for several years [Jia et al. 2012; Hong and Kim 2009]. Hong and Kim [2009] 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 makes it much harder to predict the performance statically. Meanwhile, Jia et al. [2012] 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 [Diamos and Yalamanchili 2008; Linderman et al. 2008; Luk et al. 2009; Brown et al. 2011; Kessler et al. 2012]. Harmony [Diamos and Yalamanchili 2008] 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 [Linderman et al. 2008] is a predicate dispatch-based library system for managing map-reduction applications on heterogeneous systems. Luk et al. [2009] proposed Qilin, which automatically partitions threads to one CPU and one GPU by providing new APIs that abstract away two different programming models, Intel Thread Building Block and CUDA. Kim et al. [2011] also proposed a framework that distributes workload of an OpenCL kernel to multiple equivalent GPUs for specific types of data-parallel kernels. Delite [Brown et al. 2011] is a compilation framework that takes a program written in OptiML and converts it into a C++/CUDA program. Then runtime system manages execution between the CPU and GPUs. While this work is limited to domain-specific languages, an SKMD provides more generality as it supports a variety of OpenCL applications. The PEPPHER proposed by Kessler et al. [2012] 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 studies has been done for virtualizing GPU resources [Rossbach et al. 2011; Kato et al. 2012; Rossbach et al. 2013; Wang et al. 2014; Suzuki et al. 2014; Lee et al. 2014]. PTask [Rossbach et al. 2011] provides APIs that work with OS abstraction layers to manage compute tasks on GPUs by using a data-ow programming model. Dandelion [Rossbach et al. 2013] 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 [Rossbach et al. 2011]. Kato et al. [2012] proposed Gdev, which manages GPU resources in the OS level, so that GPUs can be treated as first-class computing resources in multitasking systems. The SKMD is different from these prior works as it does not require programmers to use additional APIs or language extensions, but it is transparent to OpenCL applications, which can further optimize parallel kernels by utilizing local memories. Wang et al. [2014] and Suzuki et al. [2014] 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 an abstract view of unified memory system in separate physical memory [NVIDIA 2014a]. The main purpose of this idea is removing the burden of managing multiple memory spaces [Keckler et al. 2011], but it still leaves work distribution between devices as the programmer's responsibility. On the other hand, the 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.

6. CONCLUSION

In this article, we presented the SKMD, a framework that transparently manages collaborative execution on CPUs and GPUs of a single OpenCL kernel. The SKMD leverages assigning a subset of data-parallel workload over multiple CPUs and GPUs to increase overall performance. As a part of the exploration, this article 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, we 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 real hardware, we showed that the 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.

ACKNOWLEDGMENTS

Much gratitude goes to the anonymous referees who provided excellent feedback.

REFERENCES

AMD. 2012. Accelerated Parallel Processing (APP) SDK. http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/.

Kevin J. Brown, Arvind K. Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. A heterogeneous parallel framework for domain-specific languages. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*. 89–100.

Gregory Diamos, Andrew Kerr, Sudhakar Yalamanchili, and Nathan Clark. 2010. Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. 353–364.

Gregory F. Diamos and Sudhakar Yalamanchili. 2008. Harmony: An execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*. 197–200.

Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. 2007. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proceedings of the 40th Annual International Symposium on Microarchitecture*. 407–420.

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

Jayanth Gummaraju, Laurent Morichetti, Michael Houston, Ben Sander, Benedict R. Gaster, and Bixia Zheng. 2010. Twin Peaks: A software platform for heterogeneous computing on general-purpose and graphics processors. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. 205–216.

Sunpyo Hong and Hyesoon Kim. 2009. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*. 152–163.

Amir H. Hormati, Mehrzad Samadi, Mark Woh, Trevor Mudge, and Scott Mahlke. 2011. Sponge: Portable stream programming on graphics engines. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems*. 381–392.

Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. 2012. Stargazer: Automated regression-based GPU design space exploration. In *Proceedings of the 2012 IEEE Symposium on Performance Analysis of Systems and Software*. 2–13.

Ralf Karrenberg and Sebastian Hack. 2011. Whole-function vectorization. In *Proceedings of the 2011 International Symposium on Code Generation and Optimization*.

Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. 2012. Gdev: First-class GPU resource management in the operating system. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'12)*. 401–412.

Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. 2011. GPUs and the future of parallel computing. *IEEE Micro* 31, 5, 7–17.

Christoph Kessler, Usman Dastgeer, Samuel Thibault, Raymond Namyst, Andrew Richards, Uwe Dolinsky, Siegfried Benkner, Jesper Larsson Traff, and Sabri Pllana. 2012. Programmability and performance portability aspects of heterogeneous multi-/manycore systems. In *Proceedings of the 2012 Design, Automation and Test in Europe*. 1403–1408.

KHRONOS. 2014. OpenCL—The open standard for parallel programming of heterogeneous systems. http://www.khronos.org.

Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. 2011. Achieving a single compute device image in OpenCL for multiple GPUs. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 277–288.

Manjunath Kudlur and Scott Mahlke. 2008. Orchestrating the execution of stream programs on multicore platforms. In *Proceedings of the '08 Conference on Programming Language Design and Implementation*. 114–124.

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*. 75–86.

Janghaeng Lee, Mehrzad Samadi, and Scott Mahlke. 2014. VAST: The illusion of a large memory space for GPUs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*. 443–454.

Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. 2013. Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. 245–256.

Janghaeng Lee, Haicheng Wu, Madhumitha Ravichandran, and Nathan Clark. 2010. Thread tailor: Dynamically weaving threads together for efficient, adaptive parallel applications. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*. 270–279.

Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. 2010. Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*. 451–460.

Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. 2008. Merge: A programming model for heterogeneous multi-core systems. In *13th International Conference on Architectural Support for Programming Languages and Operating Systems*. 287–296.

LLVM. 2014. libclc. Retrieved July 23, 2015 from http://libclc.llvm.org.

Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. 2009. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual International Symposium on Microarchitecture*. 45–55.

Douglas C. Montgomery, Elizabeth A. Peck, and G. Geoffrey Vining. 2001. *Introduction to linear regression analysis* (3rd ed.). Wiley, New York, NY.

NVIDIA. 2012. CUDA Toolkit 4.2. Retrieved July 23, 2015 from https://developer.nvidia.com/cuda-toolkit-42-archive.

NVIDIA. 2014a. *CUDA C Programming Guide*. Retrieved July 23, 2015 from http://docs.nvidia.com/cuda.

NVIDIA. 2014b. PTX: Parallel Thread Execution ISA. Retrieved July 23, 2015 from http://docs.nvidia.com/cuda/parallel-thread-execution.

David A. Padua and Michael J. Wolfe. 1986. Advanced compiler optimizations for supercomputers. *Communications of the ACM* 29, 12, 1184–1201.

J. R. Quinlan. 1986. Induction of decision trees. *Journal of Machine Learning* 1, 1, 81–106.

Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. 2011. PTask: Operating system abstractions to manage GPUs as compute devices. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. 233–248.

Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. 2013. Dandelion: A compiler and runtime for heterogeneous systems. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. 49–68.

John A. Stratton, Sam S. Stone, and Wen-Mei W. Hwu. 2008. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 16–30.

Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. 2014. GPUvm: Why not virtualizing GPUs at the hypervisor?. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'14)*. 109–120.

Linda Torczon and Keith Cooper. 2011. *Engineering A Compiler* (2nd ed.). Morgan Kaufmann Publishers Inc., Burlington, MA.

Kaibo Wang, Xiaoning Ding, Rubao Lee, Shinpei Kato, and Xiaodong Zhang. 2014. GDM: Device memory management for GPGPU computing. In *2014 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. 533–545.

Received July 2014; revised February 2015; accepted June 2015