Transparent CPU-GPU Collaboration for Data-Parallel Kernels on Heterogeneous Systems

Janghaeng Lee, Mehrzad Samadi, Yongjun Park and Scott Mahlke

Advanced Computer Architecture Laboratory
University of Michigan - Ann Arbor, MI
Email: {jhaeng, mehrzads, yjunpark, mahlke}@umich.edu

***Abstract*— 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. Unfortunately, this work distribution can be a poor solution as it under utilizes 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 paper, 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 29% 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.**

***Index Terms*—GPGPU, OpenCL, Collaboration, Data parallel**

I. 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 architecture with hundreds of relatively simple cores while exposing parallelism to the programmer. By leveraging new programming models, such as OpenCL [13] and CUDA [1], 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 [18], [26], [3], [7], [10], [8], [5].

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 [4]. Unfortunately, applications that launch multiple independent kernels are rare and require programmer effort to ensure there are no inter-kernel data dependences. When dependences cannot be eliminated, 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 [21], [14], [12]. Luk et al. [21] 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. [14] 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

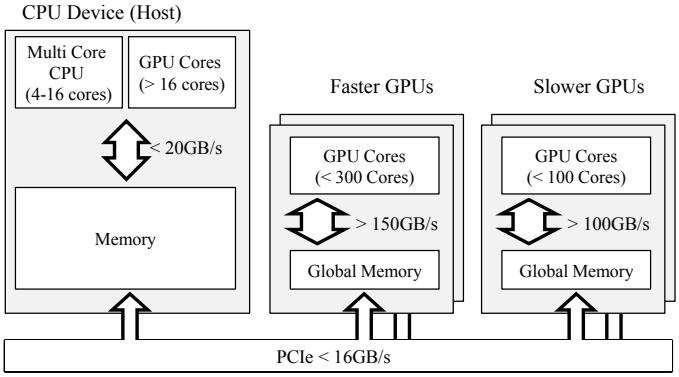


Fig. 1: Physical OpenCL Computing Devices with Different Performances, Memory Spaces, and Bandwidths

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 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 paper, 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 paper makes the following contributions:

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

II. BACKGROUND

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

A. 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 CPUs/GPUs 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 to both CPUs and GPUs whereas *devices* in CUDA usually refer to GPUs. Host code contains the sequential code sections of the program, which is run only 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 devices 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 is further divided into processing elements (PEs). The OpenCL execution model consists of three levels of 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 hardware scheduler.

B. Memory Consistency and Multi-Device 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

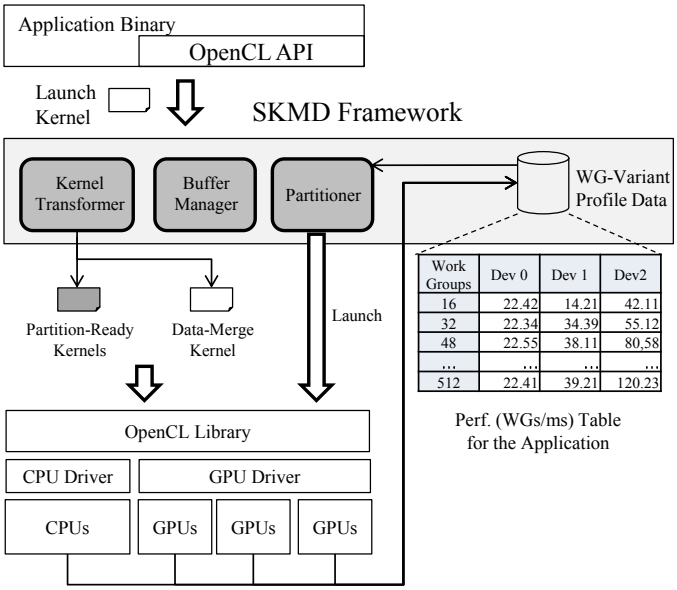


Fig. 2: SKMD Framework consisting of Kernel Transformer, Buffer Manager, Partitioner, and Profile Database

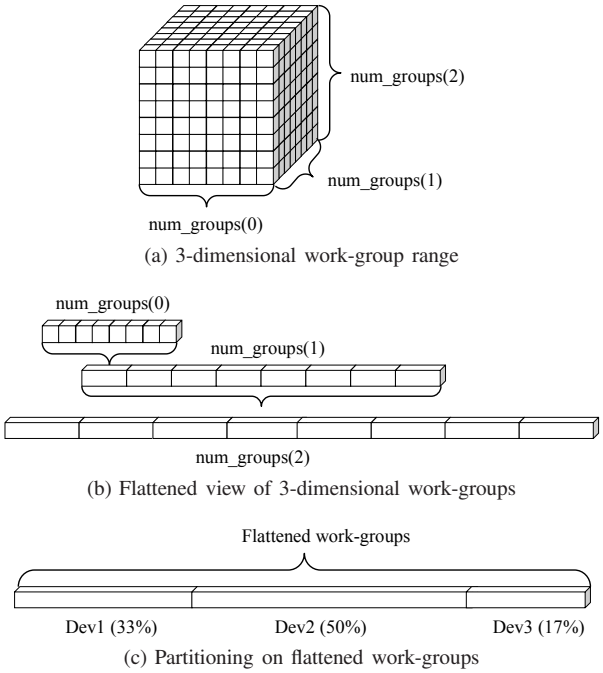


Fig. 3: OpenCL's N-Dimensional Range

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 the 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 in 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. Final results can be made when the partial results are merged properly.

One limitation of distributing data-parallel workloads to multiple devices at work-group granularity is that it must handle *global barriers* or *atomic operations* carefully because the execution of work-groups should be ordered at those points in the middle of execution. In this paper, we do not consider applications with these semantics since it will produce significant overhead on multi-device execution.

III. 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 API including querying-platform APIs. For querying-platform APIs, SKMD returns an illusion of virtual platform with only one large available device. When other APIs were hooked, SKMD system maintains all information such as device buffer size, kernel name, and kernel arguments in an internal mapping table, and does not pass them to real OpenCL libraries but returns fake value (e.g. $CL\_SUCCESS$) immediately to the application until 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 three units: **kernel transformer**, **buffer manager** and **partitioner** as shown in grey boxes in Figure 2. The kernel transformer changes the original kernel to *Partition-Ready* kernel, which enables the kernel to work only on a subset of work-groups. After kernel transformation, the buffer manager performs static analysis on kernels to determine memory access pattern of each work-group. If 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 partitioning decision has been made. On the other hand, if memory access range cannot be analyzed, entire input should be transferred to each device and output must be merged. In order to merge irregular locations of output from different devices, the kernel transformer generates *Merge* kernel, and SKMD launches it on CPU device.

Once kernel analysis and transformation are done, range of work-groups to execute for each device is decided by the partitioner considering performance effect of adjusting workload to each device. If the profile information does not exist, SKMD executes a dry run with *Partition-Ready kernels* varying number of work-groups for each device in order to

```
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

collect the data. After partitioning decision has been made, the buffer manager transfers necessary data from the host to external devices, then SKMD launches the actual kernel for each device.

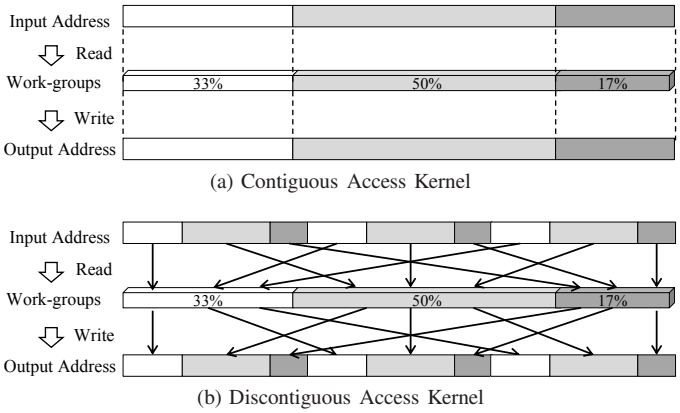
Rest of this section discusses these three components of SKMD: kernel transformation, buffer management, and performance variation-aware partitioning.

A. Kernel Transformation

As OpenCL kernels can launch up to three dimensional work-groups, the kernel transformation flattens N-dimensional work-groups to one dimensional work-groups to assign balanced work over all devices in a work-group granularity. For example, Figure 3(a) shows three dimensional ranges, each of which has 8 work-groups. Figure 3(b) shows flattened view of the work-groups, which has 512 work-groups in a single dimension. Once SKMD framework has flattened view of N-dimensional work-groups, it assigns a subset of work-groups in flattened range as shown in Figure 3(c). Based on this idea, next subsection discuss how SKMD generates *Partition-Ready kernel* and *Merge kernel*.

Partition-Ready Kernel: Assigning partial work-groups can be done through code transformation as shown in Figure 4. Lines of code with gray background in the figure illustrates 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 flattened work-group indices to be computed on a device. In other words, SKMD runs $(WG\_from - WG\_to + 1)$ work-groups of the kernel and bails out the rest of the work-groups on a device. If a kernel launches more than one dimensional NDRange, flattening code is inserted as shown at line 11 in Figure 4. After flattening the work-group index, each work-item identifies its work-group index ($flattened\_id$) and checks if it is allowed to execute the kernel.

Additional code with gray background is lowered to 3-9 instructions in PTX ISAs and x86-64 ISAs. These additional instructions consist of loading index and dimension size, MADD, comparison, and branch. For PTX code, however, there is no actual load instruction for indices and sizes,



(a) Contiguous Access Kernel

(b) Discontiguous Access Kernel

Fig. 5: Different Memory Access Patterns of Kernels

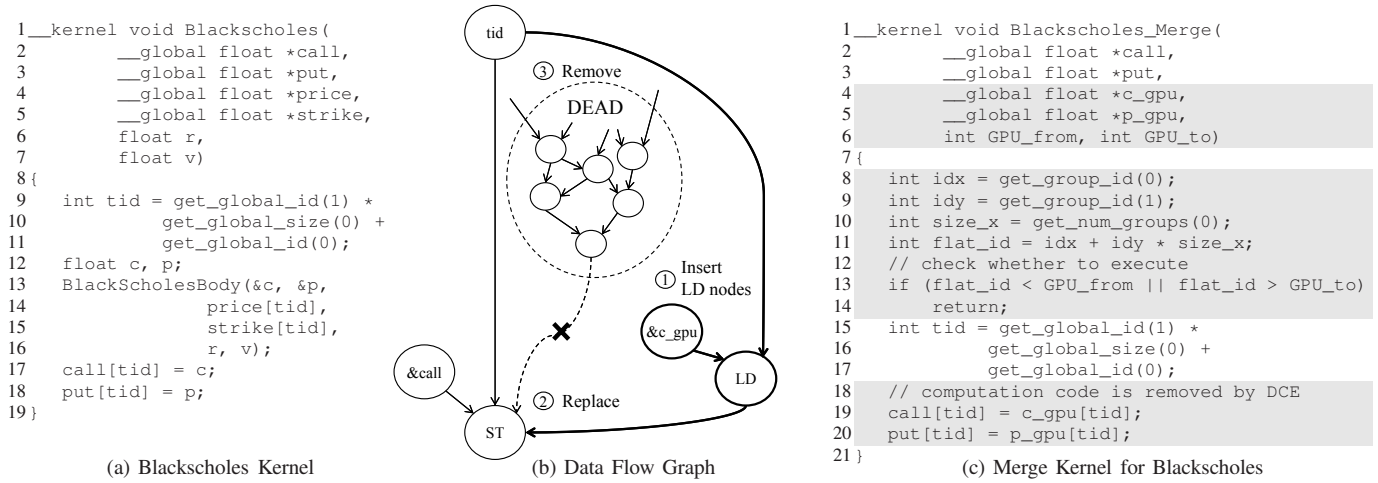
because GPUs maintain special registers for them, and they are available to each work-item and work-groups [22]. The overhead of checking code becomes more invisible in case of a memory-bound kernel that hides the time for computation code.

However, for x86 code, load instructions for the indices and sizes appears, and the checking code in CPUs may produce significant overhead as Intel OpenCL driver executes a kernel in the same way that Gregory et al. [3] 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 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 very inefficient for *Partition-Ready kernel* as CPUs must execute checking code serially inside the innermost loop, even though checking whether to execute is independent from inner loops.

To avoid this problem, SKMD system is configured with the enhanced OpenCL driver for CPU devices. The enhanced driver takes the range of enabled work-group directly from the SKMD system, so 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 for CPU devices.

Merge Kernel: Another challenge of collaborative execution of a single data-parallel kernel is that several computing devices may use different address spaces, so results from each device *must* be merged after execution. Some kernels have contiguous memory accesses, so called *Contiguous Kernel*, where each of threads writes the result in contiguous locations as shown in Figure 5(a). In this case, partial output 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*, which have discontiguous memory accesses, it is difficult to merge partial output. For example, matrix multiplication kernel is usually implemented assigning a work-group to work on a tile. Since two dimensional matrix is flattened to a single dimensional



(a) Blackscholes Kernel (b) Data Flow Graph (c) Merge Kernel for Blackscholes

Fig. 6: Merge Kernel Transformation Process

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 output. 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 novel code transformation technique that automatically merges the data without storing memory-write locations and takes full advantage of 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 output. In the CPU device, enabled work-items will write their results in 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 6(a) shows the original Blackscholes kernel that generates two output arrays. For 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 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 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.

Once GPU output parameters have been setup, 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 GPU using use-def chains. After the dynamic compiler gets 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 loaded value as shown in lines 19-20 of Figure 6(c). Finally, it marks store instruction as *live* and proceeds with *dead code elimination* using the mark-sweep algorithm [27] to remove all computation code. As a result of this transformation, computation code, line 12-16 in Figure 6(a), are removed. Note that the transformation is done in IR level where every function call is inlined in general, as OpenCL adopts SIMT execution model that does not allow threads to execute different instructions at a time.

Clearly, transformed merge kernel does not contain any computation code, except the calculation of index for load and store. With this approach, the cost of merging reaches the bandwidth between CPU cores and main memory ($<$20 GB/s) regardless of applications. That is, 4 MBytes of 1K $\times$ 1K single-precision floating point matrix can be merged in negligible time($< 0.2ms$.)

B. Buffer Management

In SKMD framework, the buffer management unit 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 subset of work-groups to several devices, each device may not require the entire input data. Likewise, each device will generate subset of output, so it is desirable to send only updated output back to the host. Considering that the bandwidth of 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 index operand of the store and load instructions, which is described as tid in Figure 6(b). Using use-def chains

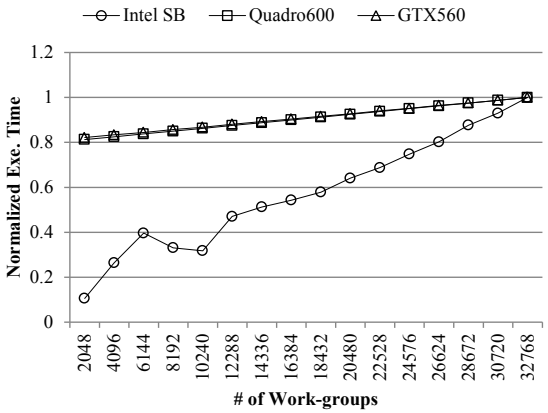


Fig. 7: Performance Impact by Work-group Size

from tid node, 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 i-th dimension respectively. For this type of kernel, it is safe to transfer subset of data to each device proportional to assigned work-groups.

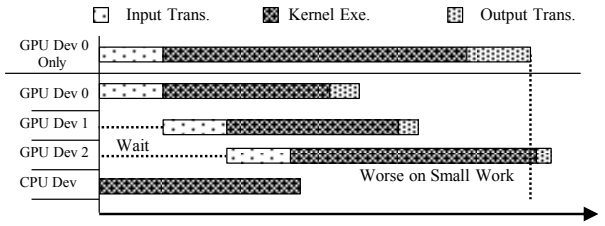
On the other hand, if index space of the kernel cannot be determined statically, or the affine function fails to be recognized as above, the buffer manager gives up optimizing data transfer and defines it as discontiguous kernel. In this case, entire input and output will be transferred back and forth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

C. Transfer Cost and Perf. Variation-Aware Partitioning

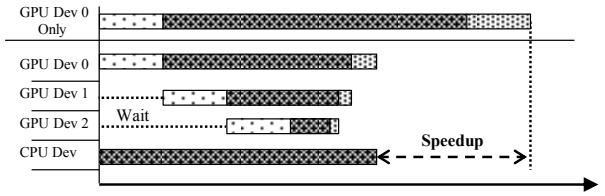
Once the kernel transformation is done, SKMD determines how many work-groups should be assigned to the underlying devices. The goal of the partitioning is to minimize the overall execution time by balancing workloads among several devices. This is an extension of the NP-Hard bin packing problem [6] and a common problem in load balancing parallel systems [17].

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 the device changes. To illustrate, Figure 7 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 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 8 assumes that there are three external GPU devices, each of which has 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



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



(b) Ideal Partitioning

Fig. 8: Comparison of Linear Partitioning and Ideal Partitioning

of workload as shown in Figure 8(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 8(b) describes the ideal case for this example.

Regarding the cost of transfer and performance variance of devices, 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 potential benefits from collaborative execution. This restriction prohibits the exact time consuming integer programming solutions [15].

To perform partitioning at runtime, SKMD utilizes a decision tree heuristic [25]. For our system, SKMD uses 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 which chooses a child that has 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 *Performance Table* for the application from the profile database. The performance table consists of k columns, each of which contains performance values for one device. Each row of this table has the performance value in certain range of work-groups for each device as shown in Figure 2 By using the performance table, the partitioner initially estimates the execution time for single

Algorithm 1 Performance Variation-Aware Partitioning

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

device execution for all k devices to identify the fastest device for the kernel. The execution time includes the transfer cost, which can be calculated using buffer size allocated by the OpenCL APIs divided by the bandwidth of PCIe.

Before the partitioner offloads the 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 offloaded 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 $PartitionGranularity$.

Device	**Intel Xeon E3-1230 (SandyBridge)**	**NVIDIA GTX 560 (Fermi)**	**NVIDIA Quadro 600 (Fermi)**
# of Cores	4 (8 Threads)	336	96
Clock Freq.	3.2 GHz	1.62 GHz	1.28 GHz
Memory	8 GB DDR3	1 GB GDDR 5	1 GB GDDR 3
Peak Perf.	409 GFlops [9]	1,088 GFlops	245 GFlops
OpenCL Driver	Enhanced Intel SDK 1.5	NVIDIA SDK 4.0	
PCIe	N/A	2.0 x16	
OS	Ubuntu Linux 12.04 LTS		

TABLE I: Experimental Setup

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 13-47 corresponds to traversing down the decision tree, and `For-Loop` at line 17-37 corresponds to testing children of a node in the tree.

Overall, the time complexity of this algorithm is $O(kNlogM)$ where k is the number of devices, and N is the number of total work-groups with M discrete range of performance variance data. Note that N can be reduced to a constant by limiting the number of induction steps as described above, and searching performance data among M ranges can be done in logarithmic time using binary search.

Benchmark	Source	Contiguous Access	Input			Output			Work-Groups
			Type	Size	Buf Size	Type	Size	Buf Size	
AESEncrypt/Decript	AMDSDK	N	Bitmap image	2048×2048	4MB	Bitmap image	2048×2048	4MB	4,096
BinomialOption	AMDSDK	Y	# of Stock Price	16,384	256KB	# of FP numbers	16,384	256KB	16,384
Blackscholes	NVIDIA SDK	Y	# of Stock Price	1,048,576	4MB	# of FP numbers	2,097,152	8MB	1,024
Histogram	AMDSDK	N	# of 8-bit numbers	16 millions	16MB	32-bit integer bins	256	1KB	512
MatrixMultiplication	AMDSDK	N	Matrix size	1,024×1,024	8MB	Matrix size	1,024×1,024	4MB	1,024
MatrixTranspose	AMDSDK	N	Matrix size	1,024×1,024	4MB	Matrix size	1,024×1,024	4MB	1,024
QuasiRandomSequence	AMDSDK	N	# of vectors, dimensions	131,072, 64	8K	# of FP numbers	8,338,608	32MB	64
Reduction(Partial)	AMDSDK	N	# of 32-bit numbers	4,194,304	16MB	# of FP numbers	8,192	32KB	2,048
ScanLargeArrays	AMDSDK	Y	# of FP numbers	1,048,576	4MB	# of FP numbers	1,048,576	4MB	4,096
VectorAdd	NVIDIA SDK	Y	# of FP numbers	4,194,304×2	32MB	# of FP numbers	4,194,304	16MB	16,384

TABLE II: Benchmark Specification

IV. EVALUATION

We tested our framework on a real machine, which has three different type computing devices as shown in Table I. Intel SandyBridge has integrated GPUs but it does not support OpenCL, so 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) 3.1 [16], on top of a Linux system with NVIDIA driver for GPU execution, and Intel OpenCL driver for 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 OpenCL frontend, and LLVM 3.1 incorporated with `libclc` extension was used for PTX backend [20]. However, PTX backend is used only for *Merge* kernels, while *Partition-Ready* kernels were transformed in source level and then directly fed into OpenCL drivers.

For the experiments, a set of benchmarks from the AMDSDK [2] and the NVIDIA SDK [23] were used to evaluate SKMD. Some benchmarks that do not create enough work-groups regardless of input size were excluded. Input size for each benchmark for the evaluation is shown in Table II. The applications from the benchmark suite were compiled without any modification.

Before real execution, we executed profile-run to collect performance variation on the number of work-groups. For each benchmark, 16 discrete ranges of data were collected. For example, Histogram launches 512 work-groups, so performance of each device were measured decreasing 32 work-groups from 512. SKMD system is also capable of online profiling by reducing the profiling time.

For online profiling, reducing profiling time is important as the overhead is added to the initial execution time and fully exposed to the user. In order to optimize profiling time, both the number of runs per kernel and the number of work-groups executed per run can be scaled back. The performance table would be filled in with a combination of actual values and values obtained by extrapolation. Many kernels have highly predictable behavior, so substantially less profiling is necessary in practice. For kernels that do not have data dependent control flow, the profiling time can be further reduced by using random input, whichever is in the devices' memory. This eliminates the time for input transfer. With these optimization methods, profiling 3 devices can be done in less than 2 seconds for most benchmarks. In our experiment, we considered the profiling as offline in order to focus on the benefit of SKMD system.

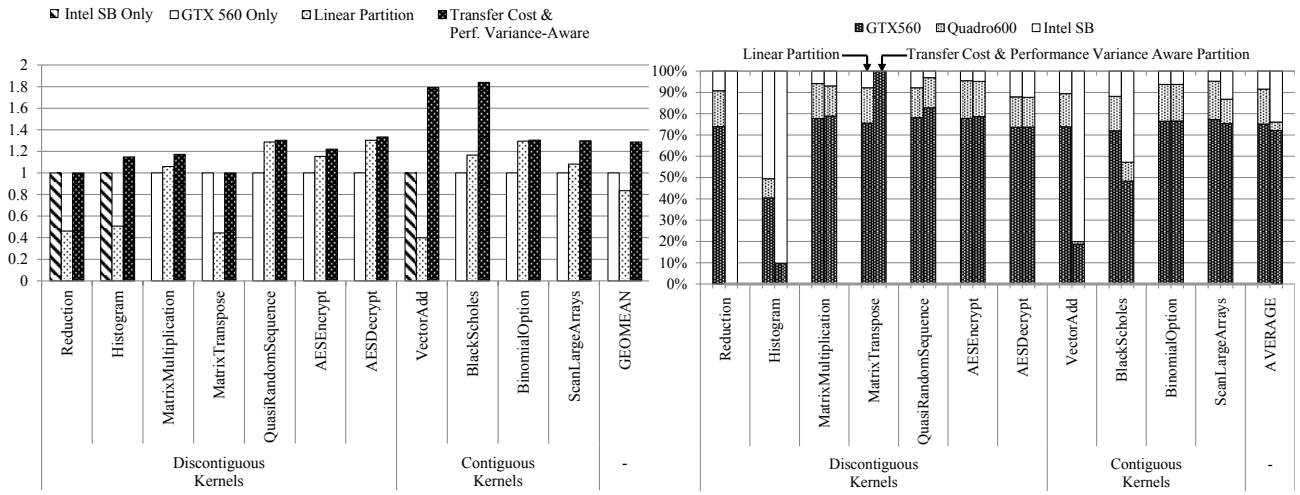
For other dynamic overheads, we did not consider the cost of kernel analysis and transformation because they can be done during offline profiling, but we measured the partitioning overhead, which is done in real execution. To reduce the overhead, we forced the height of decision tree in partitioning algorithm not to exceed 2,048 steps. In other words, for kernels that launch more than 2,048 work-groups, SKMD increases partitioning granularity. As a result, $3 \times 2,048 \times log16$ estimations are done in worst case so the overhead for partitioning algorithm becomes less than $0.1ms$ which is negligible for every benchmark we tested.

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

A. Results and Analysis

Figure 9(a) shows speedup of SKMD framework compared to single device execution and linear partitioning execution, which is similar to prior approaches [21], [14]. In linear partitioning scheme, the number of work-groups are assigned to each device is proportional to its performance without considering 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 as the baseline. Three benchmarks, Reduction, Histogram, and VectorAdd, used CPU-only execution as their baseline because data transfer cost overwhelms the benefits from executing on GPUs, as they are memory-bound kernels.

As illustrated in Figure 9(a), SKMD performs 29% faster than single device execution on average as it considers transfer cost and performance variance of each device during partitioning. Through performance variation-aware partitioning, it brings 6% performance gains out of 29%. An important point from this result is that the linear partitioning causes huge 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.



(a) Speedup normalized to the fastest single device execution

(b) Work-group distribution

Fig. 9: Speedup and Work-group Distribution

To illustrate how SKMD partitions work-groups across different devices, Figure 9(b) shows work distribution of all applications on the system with the CPU and two different GPUs. As the performance of NVIDIA Quadro 600 is relatively lower than GTX 560, the number of work-groups assigned to Quadro 600 is smaller than to GTX 560 on average.

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

Histogram: For histogram, we examined 256-bins histogram that allocates 256 bins multiplied by the number of work-items in the shared memory. Each thread has its own 256 bins in the shared memory, and accesses consecutive address in its working-set by incrementing the counter for corresponding bin. Considering that GPUs have very limited number of banks in the shared local memory (32 banks for tested GPUs) and 32 cores in GPUs shares the local memory, the probability of bank conflicts on random values of array becomes $(1 - \frac{B!}{B^N})$, where B is the number of banks and N is the number of cores. This means almost 100% probability of conflicts in the shared memory banks causes serialization, making GPU execution unfavorable. Moreover, histogram is highly memory bounded kernel, so significant amount of time will be consumed for data transfer if it is executed on external GPU device. For this reason, most of work are assigned to CPU device on both linear and proposed partitioning strategies.

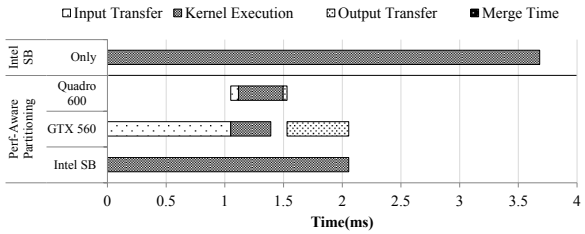
MatrixMultiplication, QuasiRandomSequence, AESEncrypt/Decrypt: These benchmarks are compute-intensive kernels where significant amount of time is spent in computation, not memory accesses. For these benchmarks, the portions of work amount assigned to GPUs are higher than CPU because of its massively data-parallel structure. As it is mentioned earlier, because of GTX 560 high performance GPU executes more work-groups than Quadro 600.

MatrixTranspose: MatrixTranspose is a memory-bound kernel but SKMD assigns all works to GPUs despite expensive cost of data transfer. This is due to very low performance of the CPUs. Since the OpenCL implementation targets GPU device, each work-group has local memory for storing input in order to avoid un-coalesced global memory access among work-items. However, for CPU execution, having local memory does not have benefits from coalesced memory access, but produces unnecessary the overhead of copying data to additional space. This overhead may not be significant, but in MatrixTranspose, copying input to the scratchpad is another equal amount of work compared to naively transposing data. Since this is a memory-bound kernel, SKMD does not assign any work-groups to Quadro 600 neither in order to avoid another expensive data transfer.

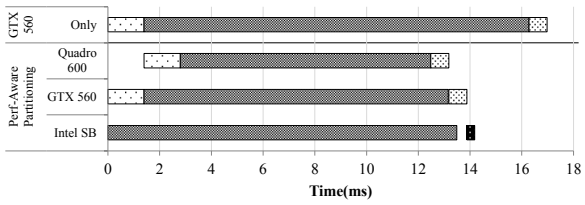
B. Execution Time Break Down

In this section, we show how SKMD transfers data between CPU and GPU and assigns work-groups to different processors. Figure 10 shows the execution time break down of three sample applications: Vector Add, Matrix Multiplication, and Histogram.

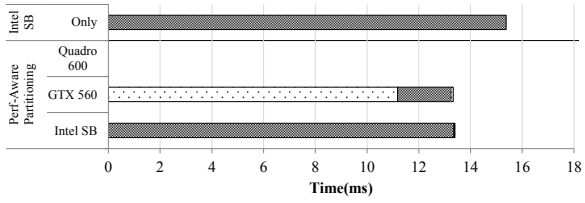
For *VectorAdd* kernel, CPU-only is the baseline for the reason discussed in Section IV-A. As shown in Figure 10(a), SKMD starts the execution on the CPU while transferring huge data to GTX 560 in background. As soon as data transfer is finished, SKMD launches the kernel on GTX 560, and at the same time, it transfers data needed for remaining work-groups to the Quadro 600 and then launches the kernel. Transfer time for Quadro is smaller because it is a less powerful GPU so the size of data the SKMD assigns 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 GPUs and simply puts them



(a) VectorAdd



(b) MatrixMultiplication



(c) Histogram256

Fig. 10: Break down Execution Time on Each Device

in the final result. As shown in the figure, CPU finishes its execution almost at the same time GPU devices finish their data transfer as a result of accurate partitioning.

The baseline of *MatrixMultiplication* is GTX 560-only as shown in Figure 10(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 analyzed as a discontiguous kernel. Similar to VectorAdd benchmark, GTX 560 starts execution first followed by Quadro 600 but finishes later than Quadro 600 because it has more work-groups to execute due to its higher performance. At the end, the CPU device merges all partial results to generate the final output by launching the merge kernel.

Histogram shows different behavior from the other cases. The baseline is CPU-only for this benchmark because it has large input size. However, output size is extremely smaller than input size as shown in Table II. In addition, Histogram is categorized as a discontiguous kernel, so SKMD still need to transfer the entire input to external GPU devices. As shown in Figure 10(c), SKMD does not assign any work-groups to Quadro 600 after assigning some work-groups to GTX 560. Since serialized input data transfer to Quadro 600 would break balanced execution among three devices. As output-size is small, the overhead of merging kernel is negligible for this benchmark.

V. RELATED WORK

A significant focus has been on the execution of data-parallel applications on CPUs. Lee et al. [18] 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. Stratton et al. [26] proposed a source-to-source compiler that translates a CUDA program into a standard C program using loop-fission technique to eliminate *synchronization*. Similarly, Diamos et al. [3] developed the Ocelot, a runtime system that dynamically transforms OpenCL/CUDA kernels for CPU execution. Gummaraju et al. [7] also performed a similar study, but approached in a light-weight thread (LWT) execution model. In a similar fashion, Karrenberg et al. [10] 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 CPU can perform as good as GPUs for some applications but none of them deals with collaborative execution with GPUs.

NVIDIA recently offered Unified Virtual Address to provide abstract view of unified memory system in separate physical memory [24]. Main purpose of this idea is removing the burden of managing multiple memory spaces [11], but still leaves work distribution between devices as programmer's responsibility.

Dynamic decision of execution on heterogeneous systems with CPUs and GPUs has been studied in the past [4], [19], [21], [12]. Harmony from Diamos et al. [4] 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 [19] is a predicate dispatch-based library system for managing map-reduction applications on heterogeneous systems. Luk et al. [21] 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. Kim et al. [14] also proposed a framework that distributes workload of an OpenCL kernel to multiple equivalent GPUs for specific types of data-parallel kernels. The PEPPHER proposed by Kessler et al. [12] improved the performance by tuning the execution strategy on a heterogeneous system based on their performance prediction model. Our approach differs from from prior work in that our system supports more than two different type of devices and considers data transfer cost and performance variance during partitioning. Also, our approach does not rely on specific programming model and type of data-parallel kernels.

VI. CONCLUSION

In this paper, we presented SKMD, a framework that transparently manages collaborative executions on CPUs and GPUs of a single kernel. SKMD leverages assigning subset of

data-parallel workload over multiple CPUs and GPUs so as to increase overall performance. As a part of the exploration, this paper introduced several techniques that transparently enable a kernel to work on partial workload, and efficiently merge results from separate devices. In order to distribute balanced workload, this paper also presented an efficient methodology for balancing workload between CPUs and GPUs being aware of data transfer cost and performance variance depending on the type of devices. By experimenting OpenCL applications on a real hardware, we showed that SKMD has speedup of 29% on a machine with one CPUs and two different GPUs as compared to the fastest device-only execution.

ACKNOWLEDGMENTS

We thank the anonymous referees who provided excellent feedback. This research was supported by National Science Foundation grants CNS-0964478 and SHF-1217917, and STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

REFERENCES

[1] "NVIDIA CUDA C Programming Guide, version 4.0," 2011.

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

[3] 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*, Sep. 2010, pp. 353–364.

[4] 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*, 2008, pp. 197–200.

[5] 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*, 2007, pp. 407–420.

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

[7] 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*, Sep. 2010, pp. 205–216.

[8] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke, "Sponge: portable stream programming on graphics engines," in *19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 381–392.

[9] Intel, "Intel xeon processor e3-1200 product family," 2012, download.intel.com/support/processors/xeon/sb/xeon_E3-1200.pdf.

[10] R. Karrenberg and S. Hack, "Whole-function vectorization," in *Proc. of the 2011 International Symposium on Code Generation and Optimization*, Apr. 2011.

[11] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the Future of Parallel Computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, 2011.

[12] 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*, Mar. 2012, pp. 1403–1408.

[13] KHRONOS Group, "OpenCL - the open standard for parallel programming of heterogeneous systems," 2010. [Online]. Available: http://www.khronos.org

[14] 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*, 2011, pp. 277–288.

[15] 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*, Jun. 2008, pp. 114–124.

[16] 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*, 2004, pp. 75–86.

[17] 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*, 2010, pp. 270–279.

[18] 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*, 2010, pp. 451–460.

[19] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: a programming model for heterogeneous multi-core systems," in *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008, pp. 287–296.

[20] LLVM, "libclc," 2012, http://libclc.llvm.org.

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

[22] NVidia, "Ptx: Parallel thread execution isa," http://docs.nvidia.com/cuda/parallel-thread-execution/.

[23] Nvidia, "Cuda Zone," 2009, https://developer.nvidia.com/category/zone/cuda-zone.

[24] NVIDIA, "Fermi: Nvidias next generation cuda compute architecture," 2009, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architect - ure_Whitepaper.pdf.

[25] J. R. Quinlan, "Induction of decision trees," *Journal of Machine learning*, vol. 1, no. 1, pp. 81–106, Mar. 1986.

[26] 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*, 2008, pp. 16–30.

[27] L. Torczon and K. Cooper, *Engineering A Compiler*, 2nd ed. Morgan Kaufmann Publishers Inc., 2011.