Task Mapping in Heterogeneous Embedded Systems for Fast Completion Time

Husheng Zhou and Cong Liu
Department of Computer Science, The University of Texas at Dallas
{husheng.zhou, cong}@utdallas.edu

ABSTRACT

Graphics processing units are being widely used in embedded systems as they can achieve high performance and energy efficiency. In such systems, the problem of computation and data mapping for multiple applications while minimizing the completion time is quite challenging due to a large size of the policy space, including heterogeneous application characteristics, complex application structure, data communication costs, and data partitioning. To achieve fast competition time, a fine-grain mapping framework that explores a set of critical factors is needed for heterogeneous embedded systems. In this paper, we consider this mapping problem by presenting a theoretical framework that yields an optimal integer programming solution. Moreover, based upon several interesting measurements-based case studies, we design three practical mapping algorithms with low time complexity, each of which explores a specific set of factors that may affect the completion time performance. We evaluated the proposed algorithms by implementing them on a real heterogeneous system and using a large set of popular benchmarks for evaluation. Experimental results demonstrate that our proposed algorithms can achieve up to 30% faster completion time compared to the state-of-the-art mapping techniques, and can perform consistently well across different workloads.

Categories and Subject Descriptors

D.4.7 [**Organization and Design**]: Real-time systems and embedded systems; C.1.3 [**Other Architecture Styles**]: Heterogeneous (hybrid) systems

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

GPU, heterogeneous scheduling

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ESWEEK'14, October 12-17, 2014, New Delhi, India.
Copyright 2014 ACM 978-1-4503-3052-7/14/10 ...$15.00.
http://dx.doi.org/10.1145/2656045.2656074

1. INTRODUCTION

Graphics processing units (GPUs) are now commonly used as co-processors in many embedded systems to accelerate general-purpose applications. They are particularly capable of executing data-parallel applications, due to their highly multi-threaded architecture and high-bandwidth memory. Various embedded system domains can benefit high performance and better energy efficiency from utilizing GPUs. For example, GPUs can efficiently perform matrix operations such as factorization on large data sets and multi-dimensional FFTs and convolutions. Such operations are often seen in many embedded applications including signal processing, imaging and video processing. By leveraging new programming models, such as CUDA [24] and OpenCL [10], programmers can effectively develop highly data-parallel kernels to execute such applications on GPUs.

By providing heterogeneous processing elements with different performance characteristics in the same system, heterogeneous CPU/GPU architectures are expected to provide more flexibility for better performance compared to homogeneous systems. Fast completion time is an imperative performance metric that needs to be optimized in most embedded systems. For example, in a driver-assisted and autonomous vehicle, the video streaming and sensor data processing tasks need to be completed in a rapid manner. In order to minimize the completion time for running a set of workloads, the step that maps computations to processing elements is critical. In this paper, we consider the mapping problem in a heterogeneous system containing multiple CPUs and GPUs. Our goal is to minimize the completion time.

This mapping problem is quite challenging due to a large size of the policy space. First of all, applications may demonstrate (sometimes significantly) different performance characteristics when executed on GPUs than CPUs. The mapping algorithm thus needs to consider such heterogeneity when making prioritization and mapping decisions. Moreover, most real world workloads are implemented using rather complex kernel graphs, where a kernel graph contains a number of data- or logical- dependent kernels. The precedence constraints among kernels require the mapping algorithm to consider: *(i)* the kernel graph structure and *(ii)* different data transfer costs among kernels if executed on different processors. Furthermore, for data-intensive kernels, data partitioning techniques need to be incorporated into the mapping algorithm because partitioning a kernel into threads that can be run on multiple devices in parallel improves the overall utilization.

Without considering the above-mentioned factors, map-

ping algorithms are unlikely to perform consistently well across different workloads. Prior work on heterogeneous CPU/GPU systems has focused on new programming models and API extensions for supporting multiple heterogeneous devices [23, 3, 12], automating the mapping processor [5, 15, 16], enabling CPU and GPU sharing [20]. Different mapping heuristics have been designed and applied in these work. However, since the fine-grain mapping problem is not the major focus of these work, the existing mapping heuristics make simplified mapping decisions based upon a limited set of metrics (e.g., data locality or execution time).

To better understand this mapping problem, in this paper, we first present a theoretical framework that yields an optimal integer programming (IP) solution. To the best of our knowledge, this is the first work that provides theoretical understanding of the mapping problem in the context of data-dependent kernel graphs and heterogeneous devices. We then show that practical mapping algorithms considering several critical factors may also perform consistently well across different workloads. Specifically, motivated by a number of measurements-based case studies, we design three mapping algorithms, each of which explores a specific set of factors that may affect the completion time performance.

We evaluated the proposed algorithms by implementing them on a real heterogeneous system containing a four-core CPU and two discrete GPUs with different performance characteristics. Extensive experiments were conducted using a set of popular benchmarks and workloads, such as cholesky factorization, MonteCarlo. Experimental results demonstrate that our proposed algorithms can achieve much faster completion time (up to 30% improvement) compared to the state-of-the-art mapping techniques. By testing workloads with varying characteristics, experiments show that the completion time performance under our mapping algorithms is also consistent.

The rest of this paper is organized as follows. Sec. 6 describes related work. Sec. 2 presents the system model and our theoretical framework. Sec. 3 describes the motivational measurements-based case studies. Sec. 4 presents the proposed practical mapping algorithms. Sec. 5 discusses our implementation methodology and experimental results. Sec. 7 concludes.

2. SYSTEM MODELING AND MIP FORMULATION

In this section we give out a list of notations and definitions to help us better formalize the proposed problem, and then we propose a MIP (Mixed Integer Programming) Formulation to get optimal mapping in theory.

2.1 System Model

Let us consider the problem of mapping n independent applications $\Gamma = \{\tau_1, \tau_2, \tau_3, ..., \tau_n\}$ onto m processors $M = \{M_1, M_2, ..., M_m\}$. Each processor is either a CPU or a GPU.

Each application τ_i is composed by serial instructions and kernels, where kernels represent computation operations. Kernels of each application are chained together according to the computation logic, they may have dependencies since data flows from one kernel to another. That is, τ_i is modeled as a kernel graph that contains z_i connected kernels $\{\tau_i^1, \tau_i^2, ..., \tau_i^{z_i}\}$. Let N denote the total number of kernels

Table 1: Notation Summary.

N	number of total tasks
n	number of applications
m	number of processors
τ_i	i^{th} application
τ_i^z	z^{th} kernel/task of application τ_i
M_i	i^{th} processor (either a CPU or a GPU)
$C_{i,k}^j$	execution time of τ_i^j on processor M_k
$\mathcal{S}(\tau_i^j)$	set of successor kernels/tasks of τ_i^j
$\mathcal{P}(\tau_i^j)$	set of predecessor kernels/tasks of τ_i^j
e_i^{jk}	edge from τ_i^j to τ_i^k
$T_{q\to w}(e_i^{jk})$	time taken to send data from τ_i^j to τ_i^k

of applications in Γ. Each kernel τ_i^j has an execution time of $C_{i,k}^j$ if executed on processor p_k. The execution time ranges from milliseconds to hours, depending on the specific application. Similar to prior work [23], we use the sampling functionality of StarPU [8] to obtain the estimated execution time of a kernel.

Between any two connected kernels is an edge, which implies that precedence constraints exist between these two kernels. If kernel τ_i^j has an outgoing edge e_i^{jk} to kernel τ_i^k, then τ_i^k cannot start execution until it receives the data produced by τ_i^j. Let $\mathcal{P}(\tau_i^j)$ denote the set of predecessor kernels of τ_i^j, i.e., kernels that have outgoing edges to τ_i^j. Similarly, let $\mathcal{S}(\tau_i^j)$ denote the set of successor kernels of τ_i^j, i.e., kernels that have incoming edges to τ_i^j. Let $T_{q\to w}(e_i^{jk})$ denote the time for τ_i^j executed on processor p_q to send its produced data to its successor τ_i^k (connected by edge e_i^{jk}) executed on processor p_w. A summary of important notation is given in Table 1. We use the term *task* to represent a kernel combing with its needed data. For readability, in the rest of this paper, we will use *task* and *kernel* interchangeably.

Definition 1. *We define the depth of a kernel to be the number of kernels on the longest path between this kernel and a kernel of the corresponding kernel graph that has no predecessors. Kernels with no predecessors have a depth of 1. Let $D(\tau_i^z)$ denote depth of kernel τ_i^z in the kernel graph of τ_i.*

Preemptive vs. non-preemptive execution On GPUs, executions are often non-preemptive [14]. That is, once a kernel starts execution on a GPU, it cannot be preempted by other kernels until its completion. On CPUs, executions can be preemptive. However, preemptions may incur significant amount of overheads at runtime such as context switch overhead and migration overhead [4]. To ensure the efficiency, as well as simplify the formalism and algorithms, we thus assume in this paper non-preemptive executions on CPU as well.

2.2 An MIP Formulation

We formalize the problem of minimizing the makespan of a given set of applications executed on m CPU and GPU devices by specifying an integer program with a polynomial number of variables as follows. We first define a new set of variables that are used in the integer program.

Definition 2. *Define*

$$p_w^x(\tau_i^j) = \begin{cases} 1 & \text{if } \tau_i^j \text{ is the } x\text{-th kernel executed by } M_w \\ 0 & \text{otherwise,} \end{cases}$$

for all $1 \leq w \leq m$, $1 \leq x \leq n$, $1 \leq i \leq n$, $1 \leq j \leq z_i$. *For every kernel* τ_i^j, *let* $s_i^j \geq 0$ *denote the starting time of its execution.*

The makespan can be denoted by C_{max}. Then this problem may be formulated as:
minimize C_{max}
subject to:

$$\sum_{q=1}^{m}\sum_{g=1}^{N} p_q^g(\tau_i^j) = 1, \qquad \forall \tau_i^j \in \Gamma \tag{1}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{z_i} p_q^1(\tau_i^j) \leq 1, \forall q = 1, ..., m \tag{2}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{z_i} p_q^g(\tau_i^j) \leq \sum_{i=1}^{n}\sum_{j=1}^{z_i} p_q^{g-1}(\tau_i^j), \forall q = 1, ..., m, \forall g = 2, ..., n \tag{3}$$

$$\begin{aligned} s_i^j \;\geq\; & s_i^k + \sum_{q=1}^{m}\sum_{g=1}^{N} C_{i,q}^k \cdot p_q^g(\tau_i^k) \\ + & \sum_{q=1}^{m}\sum_{g=1}^{N}\sum_{v=1}^{m}\sum_{h=1}^{N} p_{i,q}^g(\tau_i^j) \cdot p_{i,v}^h(\tau_i^k) \cdot T_{q\to v}(e_i^{jk}), \\ & \forall \tau_i^j \in \Gamma, \forall \tau_i^k \in \mathcal{P}(\tau_i^j) \end{aligned} \tag{4}$$

$$\begin{aligned} s_i^j \;\geq\; & s_l^k + C_{l,q}^k - \alpha \cdot \left(2 - \left(p_q^g(\tau_l^k) + \sum_{r=g+1}^{N} p_q^r(\tau_i^j)\right)\right) \\ & \forall \tau_i^j \in \Gamma, \forall \tau_l^k \in \Gamma, \\ & \forall q = 1, 2, ..., m, \forall g = 1, 2, ..., N-1 \end{aligned} \tag{5}$$

$$C_{max} \geq s_i^j + \sum_{q=1}^{m}\sum_{g=1}^{N} C_{i,q}^j \cdot p_q^g(\tau_i^j), \forall \tau_i^j \in \Gamma \tag{6}$$

$$s_i^j \geq 0, \forall \tau_i^j \in \Gamma \tag{7}$$

$$p_q^g(\tau_i^j) \in 0, 1, \forall q = 1, ..., m, \forall g = 1, ..., N, \forall \tau_i^j \in \Gamma \tag{8}$$

where $\alpha >> 0$ is a sufficiently large penalty coefficient.
Integer program description. Eq. (1) ensures that each task is assigned to exactly one processor. Eq. (2) ensures that at most one task will be the first one to be executed by any given processor. If a task is the g^{th} (where $g \geq 2$) assigned to processor M_q, then there must be another assigned as the $(g-1)^{th}$ task of this same processor, as ensured by Eq. (3). Moreover, Eq. (4) ensures that precedence constraints are respected. That is, no task τ_i^j may start execution unless all its predecessors have already completed their execution and τ_i^j has already received the data produced by its predecessors.

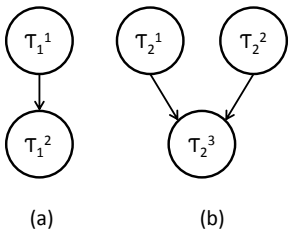

Figure 1: Kernel dependency graph

Eq. (5) defines the sequence of starting times of the set of tasks assigned to the same processor. It expresses the fact that task τ_i^j must start at least $C_{l,q}^k$ time units after the beginning of task τ_l^k, whenever it is executed after task τ_l^k on the same processor M_q, i.e., $p_q^g(\tau_l^k) = \sum_{r=g+1}^{N} p_q^r(\tau_i^j) = 1$ for some $g = 1, 2, ..., N-1$. Eq. (6) defines the constraint on the makespan (i.e., the maximum completion time among all kernels).

By solving the above formulation, we obtain an optimal solution that minimizes the makespan. Unfortunately, solving this integer program (although it has a polynomial number of 0-1 variables) is quite expensive in practice. In the next sections, we report several key observations motivated by measurements-based case studies (Sec. 3), which further motivate our design on several efficient online mapping algorithms that can be applied in practice (Sec. 4).

3. CASE STUDIES: WHAT TO CONSIDER FOR MAKING MAPPING DECISIONS

In this section, we present several measurements-based case studies that motivate the design of our mapping algorithms. We measured the completion time of executing a vector add application τ_1 and a matrix multiplication application τ_2 on a heterogeneous system configured with one Intel Core i7 CPU and NVIDIA GeForce GTX660 GPU. τ_1 can be expressed as $(v_1 + v_2) * \pi$, where v_1 and v_2 are vectors and π is a constant. τ_2 can be expressed as $(a * b) + (c * d)$, where a, b, c, d are four input matrices. These applications are commonly seen in scientific computing. The corresponding kernel graphs are illustrated in Fig. 1. Specifically, τ_1 contains two kernels τ_1^1 τ_1^2, where τ_1^1 is a vector add kernel and τ_1^2 is a vector scale kernel. τ_2 contains three kernels, where τ_2^1 and τ_2^2 are two matrix multiplication kernels, and τ_2^3 is a matrix add kernel. For the generated input data, v_1 and v_2 have a size of 50000 elements each. a, b, c, and d are four matrices with a size of $1024 * 1$, $1 * 1024$, $1024 * 1024$, and $1024 * 1024$, respectively. Through profiling, the execution time of each kernel is listed in Table 2. We have conducted various experiments based upon this system setup and recorded the corresponding mapping sequences and completion times under different strategies. Among the obtained results, we have identified several factors that may significantly affect the mapping performance.
Observation #1: kernel-level mapping or application-level mapping? In this case study, our observation is that for applications that contain multiple dependent kernels, treating kernels as the mapping entity yields better performance than mapping each entire application to a processing unit. Fig. 2(a) shows the schedule of performing application-level mapping. The dash lines in this figure represent the

Table 2: Execution time of kernels

	CPU	GPU
τ_1^1	$5.68 \times 10^2 \mu$s	$4.22 \times 10^2 \mu$s
τ_1^2	$1.52 \times 10^3 \mu$s	$2.42 \times 10^2 \mu$s
τ_2^1	$4.41 \times 10^4 \mu$s	$5.6 \times 10^3 \mu$s
τ_2^2	$8.74 \times 10^2 \mu$s	$8.44 \times 10^2 \mu$s
τ_2^3	$4.40 \times 10^2 \mu$s	$4.20 \times 10^2 \mu$s

final completion time. The (tiny) space among kernel execution blocks represents the delay due to necessary data transfer. Fig. 2(b) shows the schedule of performing kernel-level mapping, in which we can see that the completion time is shortened. The main performance acceleration comes from the parallel executions of multiple kernels on two processing units. Intuitively, for systems that support multiple applications, kernel-level mapping is a better choice because it can better utilize the hardware resources.

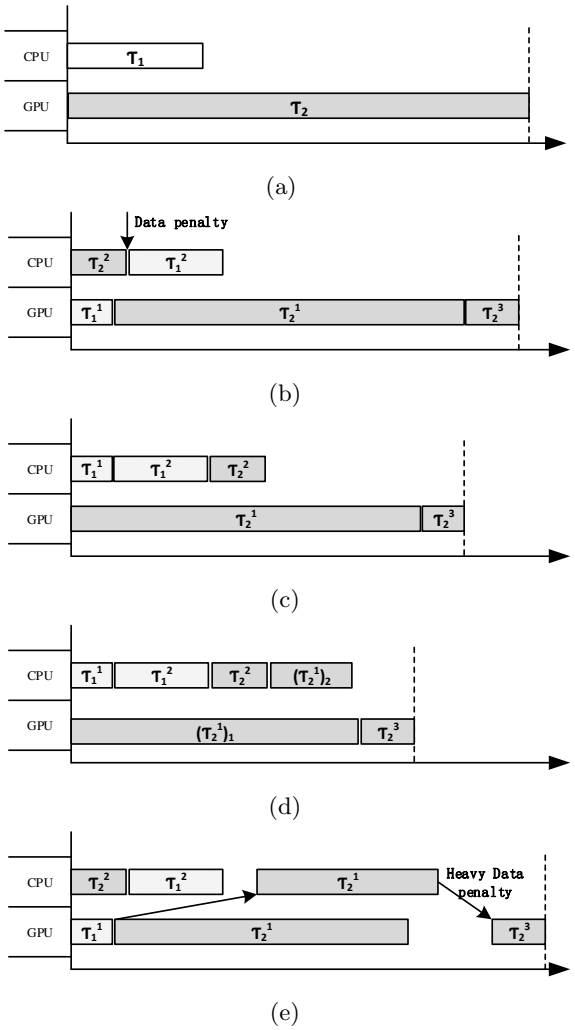

Figure 2: (a) Application level mapping and (b) Kernel level mapping (c) Different map order (d) Data Partition (e) Bad data partition

Observation #2: heterogeneity matters. Fig. 2(c) shows the schedule of a kernel mapping policy with a different kernel ordering scheme than the mapping policy shown in Fig. 2(b). The applied mapping policy considered in this case prioritize kernels by considering the heterogeneity. Intuitively, a kernel that has a faster execution time on a specific type of processor (either CPU or GPU) should preferably be assigned to that type of processor. As shown in Table 2, kernels τ_1^2, and τ_2^1 have much shorter execution times on GPU compared to CPU. Thus, by prioritizing such kernels over other kernels (such as τ_1^1 and τ_2^2), they have higher possibilities to be assigned to their favorite processors, as observed in Fig. 2(c). This case study highlights the fact that for CPU/GPU systems, the heterogeneity reflected by hardware and application characteristics must be considered in the mapping algorithm.

Observation #3: data partitioning—is it always beneficial? As seen in Table 2, τ_2^1 is the most computation-intensive kernel. Fig. 2(c) shows that τ_2^3 cannot start execution because τ_2^1 completes late, which causes resource under-utilization and longer completion times. By partitioning the input matrix of τ_2^1 into two slices, we are able to reduce its execution time by running the kernel with partial data on both CPU and GPU in parallel. Let $(\tau_2^1)_1$ and $(\tau_2^1)_2$ denote the resulting two kernels each with half data. The resulting schedule with reduced completion time is shown in Fig. 2(d). However, data partitioning is not free. It incurs additional data transfer overhead because data need to be sent to both $(\tau_2^1)_1$ and $(\tau_2^1)_2$, and the corresponding results need to be merged and then sent to τ_2^3. Since the data size is not very large in this case, the performance gain due to data partitioning overwhelmed the penalty due to additional data transfer. Nevertheless, when we increase the input matrix size for τ_2^1 to 16384 * 16384, the negative impact due to additional data transfer under partitioning becomes obvious, as illustrated in Fig. 2(e). Our observation herein is that data partitioning may be beneficial only when the input data size is reasonably small.

It is clear from these case studies that the completion time performance heavily depends on the mapping algorithm, which needs to consider a number of influential factors including the kernel structure, heterogeneity, kernel prioritization, and data partitioning.

4. PRACTICAL MAPPING ALGORITHMS

In this section, we present three practical online algorithms for mapping tasks in a heterogeneous platform consisting of multiple CPUs and GPUs. Our algorithmic design is motivated by the observations as discussed in Sec. 3. Specifically, the proposed mapping algorithms consider heterogeneity, kernel graph structure, and data partitioning. The first algorithm (we call it the baseline algorithm) mainly factors heterogeneity into making mapping decisions (besides considering traditional factors such as data locality and earliest completion time). The second algorithm considers kernel structure when prioritizing tasks. The third algorithm extends the baseline algorithm by taking advantages of data partitioning. As seen in Sec. 5, these three algorithms yield different performance under different experimental scenarios, depending on specific application characteristics.

4.1 Baseline Algorithm: Heterogeneity Ratio-based Mapping

As discussed in Sec. 3, without considering heterogeneous workload characteristics on CPUs and GPUs, the mapping algorithm is unlikely to efficiently utilize the heterogeneous resources. Our proposed baseline algorithm takes heterogeneity into consideration when making mapping decisions. Before describing the algorithm, we first give several definitions.

Definition 3. *The **favorite ratio** $F_{i,k}^j$ of a task τ_i^j executed on processor M_k is defined to be*

$$F_{i,k}^j = \frac{max_{h=1}^m(C_{i,h}^j)}{C_{i,k}^j} \tag{9}$$

For any task τ_i^j, a larger $F_{i,k}^j$ value implies τ_i^j is more suitable to be executed on M_k. That is, τ_i^j may have a shorter execution time if executed on M_k compared to other processors.

Definition 4. *The **heterogeneity ratio** of a task τ_i^j is defined to be*

$$H_i^j = max_{k=1}^m(F_{i,k}^j) \tag{10}$$

For any task τ_i^j, a large heterogeneity ratio implies that it may be more beneficial to execute τ_i^j on one of its favorite processors M_k where $F_{i,k}^j$ is large.

Example: Considering the example system described in Sec. 3, the *favorite ratio* of τ_1^1 if executed on processor 1 (CPU) is $F_{1,1}^1 = max(C_{1,1}^1, C_{1,2}^1)/C_{1,1}^1 = 5.68/5.68 = 1$, and the favorite ratio of τ_1^1 if executed on processor 2 (GPU) is $F_{1,2}^1 = max(C_{1,1}^1, C_{1,2}^1)/C_{1,2}^1 = 5.68/4.22 = 1.35$. The heterogeneity ratio of τ_1^1 can be calculated by $H_1^1 = max(F_{1,1}^1, F_{1,2}^1) = F_{1,2}^1 = 1.35$.

Definition 5. *Let $MDAC(\tau_i^j, M_q)$ denote the **Max Data Transfer Time** of τ_i^j if τ_i^j is assigned on M_q, which is defined as the maximum time for transferring data from any of τ_i^j's predecessor tasks to τ_i^j. Specifically, $MDTT(\tau_i^j, M_q)$ is given by*

$$MDAC(\tau_i^j, M_q) = \max_{\tau_i^k \in \mathcal{P}(\tau_i^j)} T_{g \to q}(e_i^{kj}) \tag{11}$$

where tau_i^k is executed on M_g.

Definition 6. *Let $EFT(\tau_i^j, M_q)$ denote the **Earliest Finish Time** of τ_i^j if τ_i^j is assigned on M_q. It is defined as:*

$$EFT(\tau_i^j, M_q) = T_{Avail}(M_q) + MDAC(\tau_i^j, M_q) + C_{i,q}^j \tag{12}$$

where $T_{Avail}(M_q)$ is the earliest time at which processor M_q is available,

Our proposed baseline algorithm prioritizes tasks based on their heterogeneity ratio. The intuition is to give tasks with larger heterogeneous ratios higher possibilities to be assigned on their favorite processing units. Compute each task's heterogeneity ratio at runtime may incur a considerable amount of overheads. To avoid such overheads, in our implementation, we maintain a lookup table for each task, which records its historical sampling information. Consider the matrix multiplication kernel as an example, each entry in the lookup table contains data size, average execution time, processing unit to which it is assigned, heterogeneity ratio, hash value, etc. Thus, at runtime, we only need to check the lookup table to figure out the needed information (e.g., heterogeneity ratio). After prioritizing tasks, the algorithm selects the best processing unit for executing each task in turn based on the earliest finish time. The psuedo-code of the algorithm is given in Algorithm. 1.

Algorithm 1 Heterogeneity ratio-based mapping

```
1: function PushTask(Γ)
2:     Sort tasks in the ready queue by largest-heterogeneity-ratio-first
3:     for t_i in ReadyQueue decreased by H_i do
4:         if H(task) < H(t_i) then
5:             continue
6:         end if
7:         InsertBefore(task, t_i, ReadyQueue)
8:     end for
9:     num ← GetAllDeviceLen()
10:    if num < thr then
11:        PushTaskOnDeviceQueue
12:    end if
13: end function
14:
15: function PushTaskOnDeviceQueue
16:    τ_i^j ← PopFront(ReadyQueue)
17:    for M_q in processor set M do
18:        EFT(τ_i^j, M_q) = T_Avail(M_q) + MDAC(τ_i^j, M_q) + C_{τ_i^j, M_q}
19:    end for
20:    Assign τ_i^j to M_q that minimize EFT(τ_i^j, M_q)
21: end function
```

Pseudo-code description. The *PushTask()* function on Line 1 is in charge of pushing incoming tasks into the ready queue of the scheduler. It first obtains the heterogeneity ratios from the lookup table for each incoming task (Line 2), then insert the tasks into the ready queue by largest-heterogeneity-ratio-first (Lines 3-8). On Line 9, function *GetAllDeviceLen()* gets the total number of assigned tasks in all device queues. If the number is less than a predefined threshold *thr*(Line 10), then the scheduler executes the *PushTaskOnDeviceQueue()* function. In other words, if the total number of tasks that have been assigned to devices is large enough, then the scheduler will stop dispatching tasks in the ready queue to devices. The intuition is to let the ready queue hold most of the unassigned tasks and sort them in order while guaranteeing that processing units have enough tasks residing in their device queues to be executed. Unlike the greedy dispatching approach that assigns ready tasks immediately to devices, our non-greedy approach ensures that tasks entering the ready queue late still have a fairly good chance to be assigned to their favorite processing units. The function *PushTaskOnDeviceQueue* (Lines 15-21) seeks to assign tasks to devices. It first grabs the task with highest heterogeneous ratio (Line 16), then estimates the finish time of this task if assigned to each processor (Lines 17-19), and finally assigns the task to the processor that yields the earliest finish time (Line 20).

Time complexity. This algorithm need to compute the

heterogeneity ratio and do sort insertion that is $O(l^2)$, the assignment phase need $O(l^2 \cdot m)$ time complexity. The total time complexity is $O(l^2 \cdot m)$ where l is the number of tasks and m is the number of processors.

4.2 Kernel Graph Structure Considerations

Our second algorithm improves upon the baseline algorithm by considering the kernel graph structure of each application. As discussed in Sec. 3, our observation is that for many applications, the time taken to transfer data among kernels executed on different devices (which heavily depends on the kernel graph structure) is far from negligible when compared to task execution times. For certain data-intensive applications, the data transfer time is actually the dominant factor in response time performance. Let $T(e_i^{jk})$ to represent the general data transfer cost between two dependent tasks t_i^j and t_i^k. Since $T(e_i^{jk})$ can be decided only after knowing the specific devices to which these two tasks are assigned, we compute the average cost as the estimated data transfer time between t_i^j and t_i^k, which is given by

$$T(e_i^{jk}) = \frac{\sum_{q,w \in M} \left(T_{q \to w}(e_i^{jk}) \right)}{m^2}. \quad (13)$$

Note that if τ_i^j and τ_i^k are assigned to the same device, then $T(e_i^{jk}) = 0$.

The algorithm seeks to assign higher priorities to tasks with larger $rank(\tau_i^j)$ values. $rank(\tau_i^j)$ is defined as:

$$rank(\tau_i^j) = \sum_{M_q \in M} C_{i,q}^j/m + \max_{\tau_i^k \in \mathcal{S}(\tau_i^j)} \Big(T(e_i^{jk}) + \sum_{M_q \in M} C_{i,q}^k/m \Big), \quad (14)$$

where $\sum_{M_q \in M} C_{i,q}^j/m$ denotes the average execution time of task τ_i^j, and the max() term represents the longest time taken to send τ_i^j's data to any of its successor tasks plus this successor's execution time. The intuition behind using $rank(\tau_i^j)$ values is to give pairs of connected kernels that are computation-intensive and/or data-intensive higher possibilities to be assigned to their favorite devices. The pseudocode of this algorithm is given in Algorithm. 2. As seen, the algorithm is identical to our baseline algorithm except that the scheduling priorities tasks using the $rank(\tau_i^j)$ values instead of heterogeneity ratios.

Algorithm 2 Structure rank based heuristics

```
1: function PushTask(task)
2:     rank(task) ← Compute rank of task
3:     for t_i in ReadyQueue decreased by rank(t_i) do
4:         if rank(task) < rank(t_i) then
5:             continue
6:         end if
7:         InsertBefore(task, t_i, ReadyQueue)
8:     end for
9:     num ← GetAllDeviceLen()
10:    if num < thr then
11:        PushTaskOnDeviceQueue
12:    end if
13: end function
```

4.3 Data Partitioning

According to the observation given in Sec. 3, the intuition behind data partitioning is that if a task is data-intensive, then dividing its data into multiple slices would give it a higher chance to utilize more processors. This idea has been proposed and applied in [21], but only under a single kernel scenario. For example, an automated partitioning technique has been proposed in [23] to partition the data of a single kernel such that this kernel can be executed on a CPU and a GPU in parallel. Unlike prior work, our third algorithm considers data partitioning as a sub-component and integrates it into our considered multi-kernel scenario.

Despite of its advantages, data partitioning may also introduce additional data transfer costs, as discussed in Sec. 3. Thus, a mapping algorithm needs to decide whether to apply data partitioning to applications. Our third algorithm extends the baseline heterogeneity ratio-based algorithm by taking data partitioning into account. We apply a historical data profiling technique to decide whether a task needs to be partitioned. In the implementation, we record the historical sampling data and use a non-linear regression-based cost model ($a * D^b + c$) [19] (where a, b, and c are constant coefficients, and D is the data size) to find out the relationship between data size and execution time. Given the data size of a kernel, if the estimated execution time (without applying data partitioning) is larger than a pre-defined threshold, then we partition it into multiple blocks.

5. IMPLEMENTATION AND EVALUATION

In this section, we present the implementation methodology and experimental results used to evaluate the effectiveness of our proposed algorithms.

5.1 Implementation

We implemented our scheduler algorithms on top of the StarPU runtime platform [8] as customized schedulers. To better support our algorithms, we modified part of StarPU's core code. The role of the StarPU scheduler is to dispatch tasks onto different processing units (named "workers" internally). All StarPU scheduling strategies implement task dispatching using queue-based method. Tasks that have received needed data from their predecessors are pushed in a *ReadyQueue*. This *ReadyQueue* is updated at runtime while tasks arrive dynamically. Based upon this dispatching model, our schedulers make mapping decision at runtime for tasks in *ReadyQueue*.

StarPU has several pre-defined schedulers, including the eager scheduler, the dm scheduler, and the dmda scheduler. The eager scheduler uses a single FIFO task queue, as illustrated in Fig. 3 (a), from which workers draw tasks to execute. The mapping decision is made only when a worker becomes idle. More complex schedulers such as the dm scheduler maintain one queue for each processing unit, as shown in Fig. 3 (b). A task is immediately dispatched to a specific worker once it is pushed into the *ReadyQueue*. Different from these implementation strategies, our scheduler uses a central priority queue to hold and sort tasks, and dispatch tasks to worker's private queues, as illustrated in Fig. 3 (c). Under our implementation, the proposed schedulers do not immediately dispatch a incoming task to one of the workers' queues. Instead, we set a threshold value (as discussed in Sec. 4.2) to trigger the dispatching action. The central priority queue would dispatch tasks to workers only

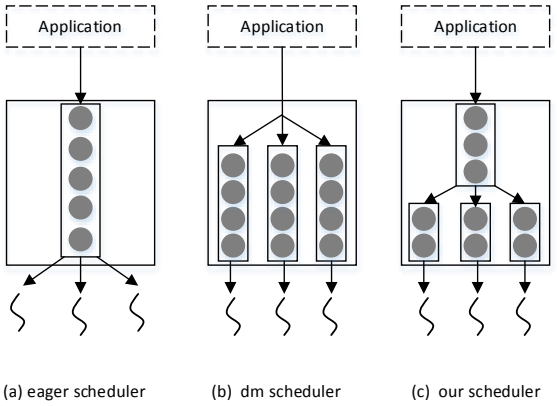

Figure 3: Our scheduler implementation

	CPU	GPU1	GPU2
Architecture	Intel Core i7-4770	NVIDIA GeForce GTX 660	NVIDIA GeForce GT 620
Frequency	3.9 GHz	1033 MHz	700 MHz
Memory	16GB DDR3	2048MB GDDR5	2048MB DDR3
OS	64-bit Linux Ubuntu lucid		

Figure 6: Experimental Hardware Specification

when the total number of tasks residing in workers' queues is less than the pre-defined threshold value. A large threshold value may allow the scheduler to have a better ordering of the *ReadyQueue*. However, when considering multiple application scenarios, the total number of tasks could be large. Since pushing tasks into the *ReadyQueue* may incur overheads, a large threshold value may also reduce the efficiency as such overheads negatively impact the timing performance. Although depending on the specific hardware, the idea behind setting a threshold value is to perform task pushing and task execution in parallel at runtime.

5.2 Experimental Setup

We implemented the proposed algorithms in a real heterogeneous desktop computer consisting of a CPU and two discrete GPUs. The hardware specification is given in Fig. 6. The benchmarks used in the experiments are listed in Fig.4. All benchmarks are rewritten in order to be used on the StarPU runtime platform. Among the benchmarks, Monte-Carlo and Cholesky factorization are considered to be computation -intensive because they have relatively heavier computing workload for processor units and have a relatively high computation-to-communication ratio (i.e., the kernel execution time is far greater than the time to transfer its needed data from another device). On the other hand, VectorAdd and VectorIncrement are consider to be date-intensive because their computing workload is low, but may generate heavy date traffic. To reflect different workload scenarios, we vary the problem scale of each benchmark three problem sizes.

The specific values of the problem sizes generated in the experiments are shown in Fig. 4. Moreover, we test three workload composition scenarios commonly seen in practice, i.e., computation-intensive, data-intensive, and randomly mixed workloads. To generate these composition scenarios, we first generate one instance of each of the seven benchmarks shown in Fig. 4 as the base case. We then generate the computation-intensive workload composition using the base case combined with three instances of each of the two computation-intensive benchmarks (mentioned above). Similarly, the data-intensive workload composition is generated using the base case combined with three instances of each of the two data-intensive benchmarks. The mixed workload composition is generated by creating two instances of each of the seven considered benchmarks. Note that the current StarPU runtime system implementation mainly considers the single application scenario. To support simultaneous execution of multiple applications, in our experiments, we compose all the benchmarks into one single executable file by rewriting and compiling the source codes of the benchmarks using StarPU's SDK.

We compare our proposed mapping algorithms against the best available scheduler of StarPU—the dmdar (deque model data aware) scheduler, which considers the task execution time and the data transfer time when making mapping decisions. It is similar to the classical heterogeneous-earliest-finish-time-first scheduling (HEFT): dmdar schedules each task to a processing unit that provides the minimum finish time, and sorts tasks residing in each worker queue by largest number of available data buffers first. Moreover, we compared our algorithms to the integer programming formulation, which yields an optimal (theoretically) solution. For each experimental setup, we tested two system configurations: one with one CPU and two GPUs, and the other one with one CPU and one GPU (GTX 660). Regarding the evaluation metric, we measured the final completion time for running each entire experiment set. In the following, we denote our baseline mapping algorithm (Sec. 4.1), structure-based mapping algorithm (Sec. 4.2), data partitioning-based mapping algorithm (Sec. 4.3), the dmdar scheduler, and the integer programming solution, as "h-ratio", "d-rank", "ad-part", "dmdar", and "IP", respectively.

5.3 Results

The obtained experimental results comparing our mapping algorithms against dmdar are shown in Fig. 5 (the organization of which is explain in the figure's caption). Each bar plots the speedup achieved by the corresponding algorithm upon a naive CPU-only mapping algorithm which prioritizes workloads by shortest-execution-time-first and maps all workloads only to CPU.

As seen, in most tested scenarios, our proposed mapping algorithms improve upon dmdar. The performance gain varies depending on the workload composition and problem scale. As shown in all six graphs of Fig. 5, when the problem size is small or medium, one or more of our proposed algorithms yield slightly better performance than dmdar. The improvement is not significant in these case because the variances in heterogeneity ratio and structure spawn are small. Thus, the benefit of specifically considering these factors becomes less significant. When the problem size becomes large, the performance improvement achieve under our proposed algorithms becomes more substantial. For example, as seen in Fig. 5(b), for computation-intensive workloads with large

Benchmark	Description	Small Problem Size	Midium Problem Size	Large Problem Size
xgemm	Combined matrix multiplication and addition	1k*1k matrix x 3	4k*4k matrix	8k*8k matrix
cg	Conjugate Gradient	1k*1k matrix and 1k vector	4k*4k matrix and 4k vector	8k*8k matrix and 8k vector
cholesky	Cholesky matrix factorization	1k*1k matrix	1k*1k matrix	4k*4k matrix
increment	Vector incrementation	10k vector	100k vector	1M vector
va	Vector Add	10k vector	100k vector	1Mk vector
pi	Monte Carlo method to compute pi	1k hits per task, 1k tasks	4k hits per task, 1k tasks	8k hits per task, 1k tasks
fblock	3-D assignment	128*128*128 cube	256*256*256 cube	512*512*512 cube

Figure 4: Benchmarks used in experiments

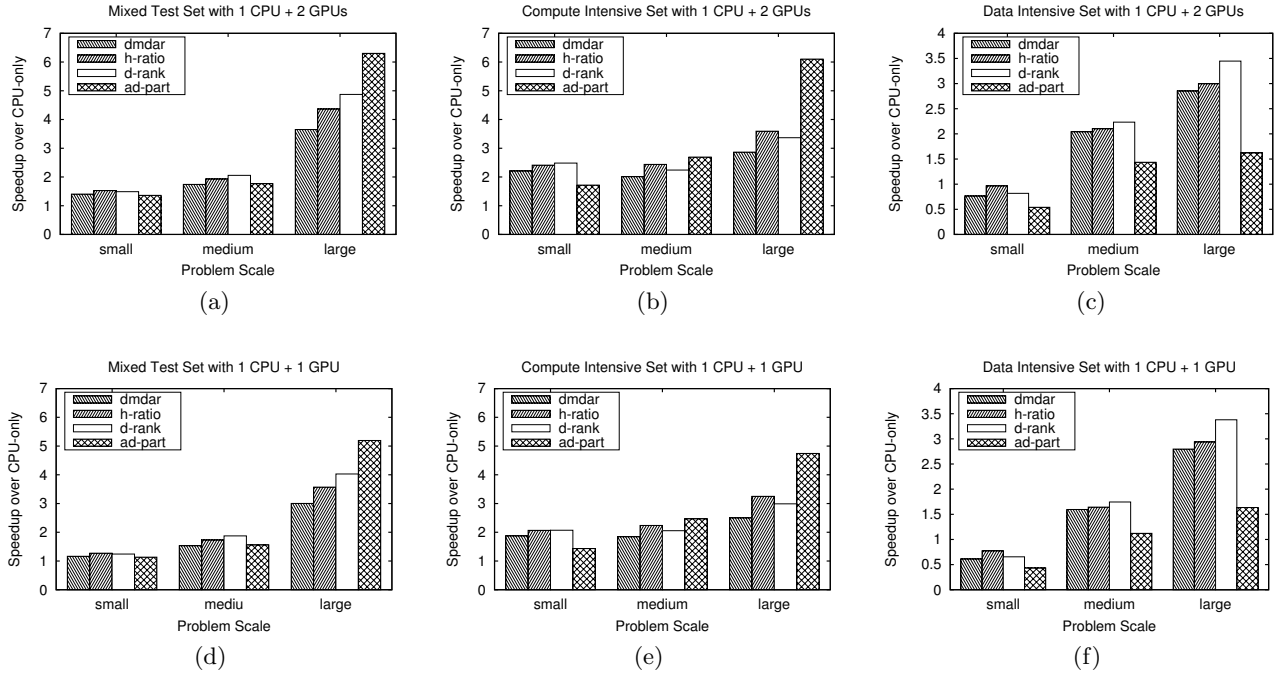

Figure 5: Experimental results on the competition time. In all six graphs, the x-axis denotes the three tested scenarios where problem size scale is varied to be small, medium, and large (according to Fig. 4). The y-axis denotes the speedup each algorithm achieved upon the naive CPU-only mapping algorithm. Graphs in the first (second) row depict the results under the system configuration with one CPU and two GPUs (one CPU and one GPU). In the first (respectively, second and third) column of graphs, mixed (respectively, computation-intensive and data-intensive) workloads are assumed.

problem size, h-ratio, d-rank, and ad-part improve upon dmdar by more than 15%, 10%, and 110%, respectively. In particular, ad-part achieves the best performance in these cases because computation-intensive kernels are divided into parallel threads with partial data. This effectively reduces the time to complete such kernels when multiple processing units become available. Moreover, for computation-intensive kernels, applying data partitioning does not incur much data transfer penalty. Another interesting observation is that when workloads become data-intensive, ad-part yields the worst performance, as shown in Figs. 5(c) and (f). By analyzing the mapping traces of these experiments, we observe that partitioning data-intensive applications may incur significant data transfer time, which negatively impact the completion time performance. Unlike prior work considering single application scenario where data partitioning should be applied in most cases, our results suggest that data partitioning should only be selectively applied, in particularly when workloads become more data-intensive. Figs. 5(d)-(f) show the results under the system configuration with the CPU and only one GPU (removing the less powerful GT 620 GPU). Compared to the case where all three processing units are used (shown in Figs 5(a)-(c)), the observation is that the speedup decreases. This is intuitive because less resources are available in this case.

Comparison against IP. We have also conducted experiments to compare our proposed algorithms against IP. Since solving the IP given in Sec. 2.2 is quite expensive as we experience in these experiments, we choose to only conduct experiments using a relatively small set of applications. Table 3 shows the application set used in the experiments and the results under IP and our algorithm (we select the best result produced under the three algorithms). As seen, our algorithm achieves comparable performance to IP while yielding a much lower runtime complexity.

Table 3: Comparison against IP.

	Exp. set 1 case study × 5	Exp. set 2 va+xgemm+inc × 5	Exp. set 3 xgemm+fblock+pi × 2
IL	318.52 ms	758.19	7176.68
Ours	330.28 ms	868.61	9975.14

6. RELATED WORK

Scheduling algorithms for heterogeneous systems. The general problem of scheduling in heterogeneous systems has received much attention. A number of scheduling heuristics have been proposed for scheduling directed acyclic graph-based (DAG) applications in heterogeneous systems [28, 6, 31, 1, 26, 9]. These algorithms schedule a single DAG (Directed Acyclic Graph) of tasks onto heterogeneous processing units with varying speed for minimizing the completion time. Zhao et.[32] proposed multi-DAG scheduling by merging multiple DAGs into one DAG. However, such algorithms do not specifically target the CPU/GPU platform, and thus ignore several critical factors when making scheduling decisions, including non-preemptivity, data transfer cost among CPUs and GPUs, data partitioning. Moreover, these existing algorithms are mostly greedy in nature and do not provide a theoretical understanding of the mapping problem considered herein. Furthermore, such algorithms use simulation-based evaluation approach and have not been tested in real systems.

Runtime system support and execution engines for heterogeneous CPU/GPU processors. For heterogeneous CPU/GPU platforms, a number of runtime systems have been developed to perform task scheduling. PTask [25] focuses on eliminating performance interference of GPU sharing. TimeGraph [20] and others [29] provides prioritization and isolation capabilities in GPU resource management. Harmony [13] schedules translated CUDA code on various devices. Qilin [23] provides an adaptive mapping to automatically partition tasks on a CPU and a GPU. SKMD [21] transparently translate single OpenCL [10] kernel into variations and execute them on multiple GPUs simultaneously. The aforementionned runtime systems either focus on single kernel or did not consider kernel affinities. Some other runtime systems focus on task dataflow parallelism: OmpSs [7], DirectShow [22], Hydra [30], StreamIt [27], IDEA [11], Liquid Metal [17], Lime [2]. However, these systems do not focus on scheduling multiple graphs onto heterogeneous processors for minimizing the completion time.

StarPU runtime system. The StarPU [8] runtime system provides programmers with a portable interface for dynamically mapping tasks onto heterogeneous processors(CPUs and GPUs). It integrates development tuning and sampling with several pre-defined task scheduling strategies [19] as plugins. These include the eager scheduler that uses the minimum-completion-time-first policy[28], the dm scheduler that performs an HEFT-based scheduling policy, and several variations of the dm scheduler. Among all pre-defined schedulers, the best one is the dmdar (deque model data aware ready) scheduler. The dmdar scheduler similar to the dm scheduler, but taking data transfer time into account and sorting tasks on a per-worker queue basis. Sc_hypervisor [18] is an extension based on StarPU, which supports co-execution of multiple applications each using the StarPU runtime system. It focuses on partitioning approaches, which split computing resources into isolated sets, and then apply existing StarPU schedulers on each set. However, the StarPU runtime system does not focus on designing efficient mapping algorithms to minimize the completion time, but rather contributes in providing a portable interface for programmers to easily utilize GPUs. The StarPU pre-defined schedulers are mainly designed to handle the single application scenario and use simplified criterion to make mapping decisions.

7. CONCLUSION

In this paper, we investigate the problem of mapping multiple applications implemented using kernel graphs in a heterogeneous system consisting of CPUs and GPUs. To achieve fast competition time, we present a fine-grain mapping framework that explores a set of critical factors that are suggested by several measurements-based case studies. We present a theoretical framework that formulates this problem as an integer program and a set of practically efficient mapping algorithms. We implement the proposed algorithms in a real heterogeneous system and conduct extensive experiments using a set of popular benchmarks. Experimental results demonstrate that our proposed algorithms can achieve up to 30% faster completion time compared to the state-of-the-art mapping techniques, and can perform consistently well across different workloads. An interesting future work is to extend the problem space to allow applications to have pre-defined completion time requirements, which is often seen in embedded systems in practice. This would make the problem even more challenging because greedy mapping choices may easily cause applications to miss their timing requirements.

8. REFERENCES

[1] H. Arabnejad and J. G. Barbosa. List scheduling algorithm for heterogeneous systems by an optimistic cost table. *IEEE Trans. Parallel Distrib. Syst.*, 25(3):682–694, Mar. 2014.

[2] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah. Lime: A java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 89–108, New York, NY, USA, 2010. ACM.

[3] C. Augonnet, O. Aumage, N. Furmento, R. Namyst, and S. Thibault. Starpu-mpi: Task programming over clusters of machines enhanced with accelerators. In *Recent Advances in the Message Passing Interface*, pages 298–299. Springer, 2012.

[4] C. Basaran and K.-D. Kang. Supporting preemptive task executions and memory copies in gpgpus. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 287–296. IEEE, 2012.

[5] A. Bhatele and L. V. Kale. Application-specific topology-aware mapping for three dimensional topologies. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

[6] L. F. Bittencourt, R. Sakellariou, and E. R. Madeira. Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 27–34. IEEE, 2010.

[7] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta. Productive cluster programming with ompss. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, Euro-Par'11, pages 555–566, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] R. N. C Augonnet, Samuel Thibault and P. Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, Feb. 2011.

[9] L.-C. Canon, E. Jeannot, R. Sakellariou, and W. Zheng. Comparative evaluation of the robustness of dag scheduling heuristics. In *Grid Computing*, pages 73–84. Springer, 2008.

[10] K. Corporation. The OpenCL Language. www.khronos.org/opencl, 2011.

[11] J. Currey, S. Baker, and C. Rossbach. Supporting iteration in a heterogeneous dataflow engine, 2013.

[12] U. Dastgeer, C. Kessler, S. Thibault, et al. Flexible runtime support for efficient skeleton programming on hybrid systems. In *International conference on Parallel Computing (ParCo)*, 2011.

[13] G. F. Diamos and S. Yalamanchili. Harmony: An execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08, pages 197–200, New York, NY, USA, 2008. ACM.

[14] G. Elliott and J. H. Anderson. Real-World Constraints of GPUs in Real-Time Systems. In *Proceedings of the First International Workshop on Cyber-Physical Systems, Networks, and Applications*, pp. 48-54, 2011.

[15] J. Enmyren and C. W. Kessler. Skepu: a multi-backend skeleton programming library for multi-gpu systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, pages 5–14. ACM, 2010.

[16] D. Grewe and M. F. O'Boyle. A static task partitioning approach for heterogeneous systems using opencl. In *Compiler Construction*, pages 286–305. Springer, 2011.

[17] S. S. Huang, A. Hormati, D. F. Bacon, and R. Rabbah. Liquid metal: Object-oriented programming across the hardware/software boundary. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 76–103, Berlin, Heidelberg, 2008. Springer-Verlag.

[18] A.-E. Hugo, A. Guermouche, P.-A. Wacrenier, and R. Namyst. Composing multiple starpu applications over heterogeneous machines: A supervised approach. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 1050–1059, Washington, DC, USA, 2013. IEEE Computer Society.

[19] N. institute for research in computer science and control. *How To Optimize Performance With StarPU*, 2008.

[20] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[21] J. Lee, M. Samadi, Y. Park, and S. Mahlke. Transparent cpu-gpu collaboration for data-parallel kernels on heterogeneous systems. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 245–256, Piscataway, NJ, USA, 2013. IEEE Press.

[22] M. Linetsky. *Programming Microsoft Directshow*. Wordware Publishing Inc., 2001.

[23] C. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proc. of the 42nd International Symp. on Microarchitecture*, pp. 45-55, 2009.

[24] C. Nvidia. Compute unified device architecture programming guide. 2014.

[25] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. Ptask: Operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 233–248, New York, NY, USA, 2011. ACM.

[26] R. Sakellariou and H. Zhao. A hybrid heuristic for dag scheduling on heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 111. IEEE, 2004.

[27] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, London, UK, UK, 2002. Springer-Verlag.

[28] H. Topcuouglu, S. Hariri, and M.-y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274, Mar. 2002.

[29] U. Verner, A. Schuster, and M. Silberstein. Processing data streams with hard real-time constraints on heterogeneous systems. In *Proceedings of the international conference on Supercomputing*, pages 120–129. ACM, 2011.

[30] Y. Weinsberg, D. Dolev, T. Anker, M. Ben-Yehuda, and P. Wyckoff. Tapping into the fountain of cpus: on operating system support for programmable devices. *ACM SIGOPS Operating Systems Review*, 42(2):179–188, 2008.

[31] H. Zhao and R. Sakellariou. An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm. In *Euro-Par*, pages 189–194, 2003.

[32] H. Zhao and R. Sakellariou. Scheduling multiple dags onto heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 14–pp. IEEE, 2006.