Scheduling Dataflow Execution Across Multiple Accelerators

Jon Currey, Adam Eversole, and Christopher J. Rossbach
Microsoft Research

ABSTRACT

Dataflow execution engines such as MapReduce, DryadLINQ and PTask have enjoyed success because they simplify development for a class of important parallel applications. Expressing the computation as a dataflow graph allows the runtime, and not the programmer, to own problems such as synchronization, data movement and scheduling - leveraging dynamic information to inform strategy and policy in a way that is impossible for a programmer who must work only with a static view. While this vision enjoys considerable intuitive appeal, the degree to which dataflow engines can implement performance profitable policies in the general case remains under-evaluated.

We consider the problem of scheduling in a dataflow engine on a platform with multiple GPUs. In this setting, the cost of moving data from one accelerator to another must be weighed against the benefit of parallelism exposed by performing computations on multiple accelerators. An ideal runtime would automatically discover an optimal or near-optimal partitioning of computation across the available resources with little or no user input. The wealth of dynamic and static information available to a scheduler in this context makes the policy space for scheduling dauntingly large. We present and evaluate a number of approaches to scheduling and partitioning dataflow graphs across available accelerators. We show that simple throughput- or locality-preserving policies do not always do well. For the workloads we consider, an optimal static partitioner operating on a global view of the dataflow graph is an attractive solution - either matching or out-performing a hand-optimized schedule. Application-level knowledge of the graph can be leveraged to achieve best-case performance.

1. INTRODUCTION

This paper addresses the problem of scheduling computations expressed as dataflow graphs on systems which have multiple compute accelerators. We focus on GPU accelerators. Compute fabric of this form is increasingly common: GPU-based super-computers are the norm [1], and cluster-as-service systems like EC2 make systems with potentially many GPUs widely available [2]. Compute-bound algorithms are abundant, even at cluster scale, making acceleration with specialized hardware an attractive and viable approach.

Despite the rapid evolution of front-end programming tools and runtimes for such systems [14, 44, 35, 16, 22, 36, 43, 12], programming them remains a significant challenge. Writing *correct* code for even a single GPU requires familiarity with different programming and execution models, while writing *performant* code still generally requires considerable systems- and architectural-level expertise. Developing code that can effectively utilize multiple, potentially diverse accelerators is still very much an expert's game.

Dataflow execution engines purport to solve many of the challenges introduced by heterogeneity and multiplicity [39, 38, 28, 20]. Because computations are expressed as graphs of vertices representing computation, connected by edges representing communication, a runtime has a complete view of available parallelism, and can implement scheduling and communication policies based on a dynamic view of the system. The separation of concerns simplifies programming, and enables applications to benefit from richer scheduling and communication policies than are available to a programmer with only a static view. The wealth of dynamic information available to a scheduler does much to reinforce the intuition that automated, optimal or near-optimal scheduling must be attainable. However, our experience building such schedulers [38, 19, 39] suggests that the size of the policy space makes the task of dynamically finding performant scheduling policies a significant challenge, since objective functions may work at cross purposes. For example, a policy that preserves locality by attempting to schedule work where its data are fresh may do so at the cost of poor utilization.

In this paper, we consider a range of scheduling policies in a multi-GPU environment for two real-world dataflow workloads. We find that simple locality- or progress-preserving policies that have been effective in previous systems [38, 39] fail to deliver attractive performance for these workloads, and find that scheduling policies that consider a global view of the graph are necessary for such workloads. We find that while strategies that search for an optimal partition based on a simplified view of the graph achieve the best overall *automated* performance, outperforming programmer-crafted partitions, application-level knowledge that allows the runtime to duplicate or re-write the graph is necessary to achieve best-case performance for one of the workloads.

The contributions of this paper are:

1. Proposal, analysis, and evaluations of a range of scheduling policies for dataflow engines based on graph partitioning and graph replication.

2. Evidence that greedy policies based on traditional objective functions such as locality preservation are insufficient in such contexts, and can even result in degraded performance.

2. MOTIVATION

We are motivated in this inquiry by our experience building schedulers for the PTask [38] execution engine. PTask (and by extension EDGE [19] and Dandelion [39]) supports a handful of simple scheduling policies that generally allow the runtime to meet the performance expectations of the programmer. Those expectations are twofold. First, the scheduler should provide good aggregate system-wide throughput in the presence of contention, and second, resources should not go idle when there is a workload that can make performance-profitable use of them. In short, the user expects minimal performance loss under contention, and transparent speedup from multiple GPUs under no contention. The first set of expectations are well served by well-known scheduling techniques such as priority, aging, and multi-level feedback queues. The second set of expectations is more nuanced: when tasks are ready and compute resources are available, the cost of moving data to a memory accessible by that compute resource may introduce more latency than would be introduced by waiting, so scheduling based on objective functions that promote utilization alone is untenable. In its default configuration, the PTask scheduler address this concern by using heuristics that strongly prefer to schedule a task where the most of its inputs are the most fresh, while avoiding starvation for tasks waiting on locality-preserving compute resource using aged dynamic priority. PTask allows the programmer to set priority for tasks when additional control to manage contention is required.

In the majority of workloads we have encountered, these simple policies have been sufficient to allow the runtime to deliver transparent speedup when multiple accelerators are present. However, we have encountered a handful of workloads in which this is decidedly not the case, and these simple heuristics not only fail to deliver a speed-up, but in fact harm performance. Our goal in this inquiry is to understand what these workloads have to teach us about dataflow scheduling, and generalize these lessons into more effective policy.

Workloads where current heuristics fail are characterized by large graphs which make heavy use of iteration and other control flow primitives, resulting in densely interconnected graphs with many cycles. A number of issues can arise for greedy schedulers working with such graphs. First, dense interconnection and frequent cycles can easily leave the scheduler in a situation where a previous decision is unrecoverable, and a placement that does not require data movement is impossible. To understand why this is the case, consider a scenario in which a graph has tasks A and B producing inputs for task C; A and B are ready, and GPUs G_0 and G_1 are both available. If inputs for A and B are stale on both GPUs, a naïve scheduler might assign A to G_0 and B to G_1 without violating any locality-preserving objectives, only to find later that it cannot schedule C without requiring data movement for one of the results produced by A and B. Avoiding this kind of problem in the general case requires a scheduler to consider a global view of the graph. In the limit, a scheduler is faced with a multi-variate optimization problem. For complex graphs with cycles, searching the space for an optimal solution likely induces unattractive performance overhead.

The second feature common to workloads where current heuristics fail is that average end-to-end compute latency for all tasks in the graph is close to the latency of a data transfer. This is easily understood with a simple example: consider a graph with two tasks, where task A produces the input of task B. In theory, pipeline parallelism may make it profitable to schedule A and B on different resources despite the additional data transfer latency along the edge from A to B because concurrent execution of A and B improves throughput despite the longer critical path. However, if the compute and transfer latencies are not well balanced, the strategy is ineffective and actually harms performance. In the general case, for a workload with low latency tasks detecting whether a profitable assignment across multiple accelerators even exists is complicated by the fact that transfer latencies are non-linear in the transfer size. To implement good policy for such workloads, a scheduler must not only consider graph structure but must make accurate predications about dynamic compute and transfer latencies.

In the remainder of this paper, we consider approaches to implementation of a scheduler that can better handle the challenges these workloads illustrate. We examine two production workloads for which we know a greedy scheduler produces poor performance: deep belief neural network training (DBN) for speech recognition, and computing optical flow (OptFlow) on a sequence of video images. Coded for PTask, both workloads are expressed as complex graphs with nested iterative control structures and many cycles. We consider a range of designs for which the scheduling algorithm produces a static partition of the graph across multiple accelerators, incorporating increasingly high-fidelity elements in the model on which the partitioner reasons. We are able to evaluate the quality of these partitions against hand-optimized partitions coded by the developers based on domain and application-level expertise. Our goal is to achieve performance close to or better than the hand-coded partitions, while minimizing the impact of the partitioner itself, both in terms of compute latency, and the amount and quality of runtime information required as input.

3. WORKLOADS

3.1 Optical Flow

Optical flow (OptFlow) captures apparent motion of patterns in a sequence of images, and is a common primitive for image processing and computer vision algorithms, e.g. removing rolling shutter wobble from video [9]. Optical flow algorithms are well-studied [10], and generally rely on iterative optimization of an energy function. Our implementation is a variation of the Horn-Schunck [26] algorithm and uses a pyramid of successively smaller scaled versions of input images. Optical flow values are calculated starting from the smallest image versions, using results from each level to seed the next level. Each pyramid level features two loops of nested iteration: an inner loop solves a linear system used by each iteration of an outer loop which refines the flow estimate. The PTask expression of optical flow results in a densely connected graph of 126 tasks and 404 edges, amongst which back edges are frequent. Compute latencies and transfer latencies vary significantly with the pyramid level.

3.2 Deep Belief Network Training

The deep belief network (DBN) workload trains a neural network for speech recognition [42]. Like OptFlow, the computation features multiple layers: DBN takes the input samples through a forward pass of the network producing a predicted result. The differences in this result and the ground truth are then used in the back propagation phase to update the weight matrices at each layer through Stochastic Gradient Descent. In its most basic form, neural network training is not parallel because a model must be updated for every input sample. Parallelization strategies trade convergence rate for parallelism, operating on batches of input samples concurrently. The PTask expression of DBN has 65 nodes and 200 edges, with back edges at each layer of the network.

4. DESIGN AND IMPLEMENTATION

The design space for schedulers that operate on a global view of the dataflow graph is large. The graph model on which the scheduler operates can attempt to capture a number of dynamic metrics at varying levels of fidelity. At one extreme, brute force search over all possible assignments, based on a model that incorporates a history of measured transfer and compute latencies should be able to find an optimal solution that precisely predicts actual runtime behavior. At the other end, a scheduler can limit traversals to local neighborhoods within a graph and employ very coarse performance modes to heuristically identify good graph cuts or sub-graphs - potentially at some performance cost when heuristics fail. In this paper, we attempt to cover important points in the design space with the following policies.

4.1 Heuristic Partitioning

The heuristic partitioning approach favors a minimal compute budget for partitioning algorithms and instrumentation to collect data informing the partitioner. Rather than search the space of assignments, the heuristic partitioner examines local neighborhoods in the graph attempting to identify structures that can be used to estimate whether separating tasks within that neighborhood will yield multiple cut edges elsewhere in the graph. This partitioner considers the denseness of the connectivity within the local neighborhood, as well as the presence or absence of other runtime level hints such as programmer-configured block pools, which generally are present at points in the graph where communication and allocation latencies are performance-critical. The goal of this heuristic partitioner is to co-schedule logically related sub-graphs without having to traverse the entire graph, and without accepting additional hints from the programmer. While the predictive accuracy of these heuristics varies a great deal, we include this policy because it represents an extreme that strongly favors minimal overhead.

4.2 Optimal Partitioning

This partitioner operates by constructing a weighted model of the graph and applying an optimal partitioning algorithm that finds an assignment across available GPUs that favors utilization (assignments are balanced across all resources) while minimizing communication latency by finding the lowest cost cut that achieves a balanced assignment. We consider both coarse- and fine-grain models of the graph as input to this partitioner. In the coarse case, no attempt is made by the runtime to set edge and vertex weights to reflect actual latencies incurred for computation and communication at those sites in the graph. In the fine-grain case the model weights are set to reflect actual dynamic properties. These two variations represent points in the design space that willingly incur compute overhead for partitioning algorithms, but may or may not invest resources for instrumentation to produce a high fidelity model for the partitioner.

Our current implementation uses an exact combinatorial graph bisection algorithm [21], which is guaranteed to find an optimal solution where one exists. We modify the PTask runtime to generate an abstract model of the graph. In the higher-fidelity variant which models edge and vertex weights, we currently hand-code weights based on our knowledge of the underlying computation. Before putting the graph in the running state, PTask passes the model to an implementation of the optimal bisection algorithm, which determines the assignment of each task to one or other of the partitions. The PTask runtime enforces these assignments using support for mandatory affinity.

4.3 Graph Cloning

The graph cloning approach does not partition the graph, but rather clones it for each available GPU, and partitions the input data across the clones. We include this point in the design space because for some workloads, (streaming, data-parallel, lacking shared mutable state) this is clearly the best policy, and consequently, a scheduler that can correctly identify and exploit the opportunity to apply this strategy is quite attractive. It should be observed that the strategy cannot be safely applied in the general case, since the runtime does not have sufficient information to know when the inputs are completely data-parallel and can be re-ordered arbitrarily. Case in point, this strategy can produce a correct result for the OptFlow workload, but would yield incorrect results if applied to the DBN workload.

While cloning support can easily be subsumed into the PTask runtime, dividing the input data without some involvement of the application is a greater challenge, since application semantics may determine the points at which the data may be divided.

5. EVALUATION

We evaluate the ability of these policies to deliver speedup for the OptFlow and DBN workloads when executed on 2-4 GPUs. Two systems are considered: one with four NVIDIA Tesla GPUs, a representative server-class system; the other with four NVIDIA GTX GPUs, a representative desktop-class system. Platform and configuration details are shown in Table 1. While subtle differences between the two GPU architectures are numerous, we consider both systems because the server class system supports fast path (peer-to-peer) communication between GPUs, while the desktop class system does not. The Tesla system can be configured either with all 4 GPUs in PCI slots associated with just one of the CPUs, or with 2 GPUs in slots associated with each CPU. The latter configuration trades faster CPU-GPU communication against slower GPU-GPU communication. We experimented with both configurations and find that higher GPU-GPU communication cost hurts performance in a very predictable way for workloads sharing data across GPUs (as ours do). Consequently, we report data only for the former configuration.

Table 2 enumerates the scheduling policies considered.

platform		description
Server	CPU	2x Intel Xeon E5-2680 @ 2.70 GHz
	GPU	4x Tesla K20M, 2496 cores each
	OS	Windows Server 2008 R2
Desktop	CPU	1x Intel Core i7-4770 @ 3.40 GHz
	GPU	4x GTX 780 Ti, 2880 cores each
	OS	Windows 8.1 Enterprise

Table 1: Configuration of Server (Tesla-Based) and Desktop (GTX-Based) Systems. Both platforms use CUDA 5.5.

Scheduling Policy	Description
FIFO	first-in, first-out
Data-Aware	favor locality with aged priority
Hand Partition	hand-optimized
Heuristic Partition	low-fidelity rule-based
Optimal Partition	partition using graph model
Graph Clone	duplicate graph

Table 2: Evaluated Scheduling Policies

We include the FIFO and Data Aware policies from PTask, along with hand-optimized partitions informed by application-level expertise. Graph cloning is applicable only for OptFlow, as the transformation does not preserve correctness for DBN. The DBN dataflow graph contains back-edges that represent shared mutable state that would need to be synchronized across the graph clones to maintain consistency.

The optimal partitioning policy relies on an algorithm that consumes a weighted model of the dataflow graph, and produces a maximally balanced partition with minimal cut cost, which translates to minimization of GPU-GPU data transfer costs. As a result, using this policy requires the runtime to produce such a model dynamically, which in turn introduces the problem of automating the selection of weights. For the DBN workload, the model assigns an equal weight to all nodes and edges in the graph. For OptFlow, we experimented with weighting schemes that approximate the actual relative compute and communication latencies of the workload. While we explored a number of such models, we found that increasing the fidelity of the models beyond a crude strategy that predicts latencies as a linear function of input size did not significantly change performance. In light of this, for OptFlow, we report only data for that scheme (as OptW2) and the uniform weighting scheme used in DBN (as OptW1).

Figure 1 shows speed-up over 1 GPU of the same type for all policies and workloads. Table 3 shows wall-clock time in seconds. All data represent an average over 10 runs, and we elide confidence intervals as the data are quite stable. For the majority of OptFlow experiments, standard deviation as a percentage of the mean is below 3%, and excluding two outliers, the highest value is 4.92%. The two outliers are the hand-optimized policy on 3 and 4 Teslas, with 18.4% and 17.5% respectively. DBN has higher standard deviation, 5-7% being typical, with two outliers at 10.20%, and 11.48%.

The data show that GPU-GPU communication latency is the dominant term in performance. Because the partition directly determines the frequency of such communication, performance is very sensitive to the quality of the partition: small increases in the number of edges cut make the difference between significant performance gains and significant losses. Figure 2 shows the latency in milliseconds of data transfers to and from GPU memory. The times are for an uncontended PCI bus, and therefore represent the best case. We are interested in latency in this context, rather than bandwidth because partitioning across multiple GPUs typically puts such communication on the critical path. Moreover, due to limits on channel capacity, high latency communication becomes a structural hazard, effectively eliminating the system's ability to exploit pipeline parallelism. Two trends in the latency data bear emphasis. First, the latency is non-linear in the transfer size: our workloads use transfer sizes that tend toward the right edge of the graph. Second, the GPU-GPU latency for the GTX system is far greater than for the Tesla system because the GTX system must route GPU-GPU through host memory. The harmful impact of this additional latency is clearly visible in the speed-up data–for example, the OptW1 and OptW2 policies in the OptFlow workload yield reasonable scaling on the Tesla system and performance losses on the GTX system.

	arch	G	policy	RT		arch	G	policy	RT
DBN	Tesla	1	–	0.68	OF	Tesla	1	–	135.4
		2					2	Clone	68.5
			DA	0.7				DA	140.2
			FIFO	0.7				FIFO	375.0
			Hand	0.6				Hand	130.7
			Heur	0.7				Heur	137.1
			Opt	0.6				Opt	130.6
		3					3	Clone	46.6
			DA	0.73				DA	141.4
			FIFO	0.7				FIFO	408.2
			Hand	0.5				Hand	95.7
			Heur	0.7				Heur	136.7
			Opt	0.5				Opt	85.1
		4					4	Clone	35.9
			DA	0.7				DA	146.3
			FIFO	0.8				FIFO	412.9
			Hand	0.6				Hand	94.7
			Heur	0.8				Heur	136.0
			Opt	0.5				Opt	79.3
	GTX	1	–	0.4		GTX	1	–	67.0
		2					2	Clone	40.1
			DA	1.1				DA	69.7
			FIFO	1.1				FIFO	–
			Hand	0.4				Hand	87.4
			Heur	1.0				Heur	78.4
			Opt	0.4				Opt	77.4
		3					3	Clone	30.8
			DA	1.1				DA	70.8
			FIFO	1.1				FIFO	–
			Hand	0.3				Hand	85.7
			Heur	1.0				Heur	78.8
			Opt	0.3				Opt	76.5
							4	Clone	27.1
			DA	1.1				DA	72.5
			FIFO	1.1				FIFO	–
			Hand	0.4				Hand	86.9
			Heur	1.1				Heur	79.3
			Opt	0.3				Opt	90.3

Table 3: Runtimes in seconds for all configurations and workloads. "OF" is short for OptFlow. "OptW" data for Optical flow are shown for the W1 scheme only. Columns labeled **G** are GPU count, and **RT** is runtime in seconds.

In light of this, for both OptFlow and DBN, the data confirm our expectations: the greedy scheduling policies (FIFO and DA) fail to benefit from additional GPUs because they have no knowledge of graph topology and consequently induce too much communication. FIFO makes no attempt to preserve locality, so dynamic partitions have rampant GPU-GPU communication. The DA policy explicitly attempts to preserve locality, but failure to consider graph topology allows it to make upstream GPU assignments that force GPU-GPU communication at downstream convergence points in the graph. Moreover DA's starvation-prevention mechanism (aged dynamic priority) causes it to occasionally sacrifice locality (unnecessarily) in the name of forward progress. The rate of aging for waiting tasks might be better tuned to lessen this effect, but we believe this is not a viable general solution, as finding good tunings is likely to be effort intensive and workload-dependent.

In contrast to DA and FIFO, the failure of the heuristic partitioner to find useful concurrency in our workloads was unanticipated. The heuristic partitioner attempts to infer

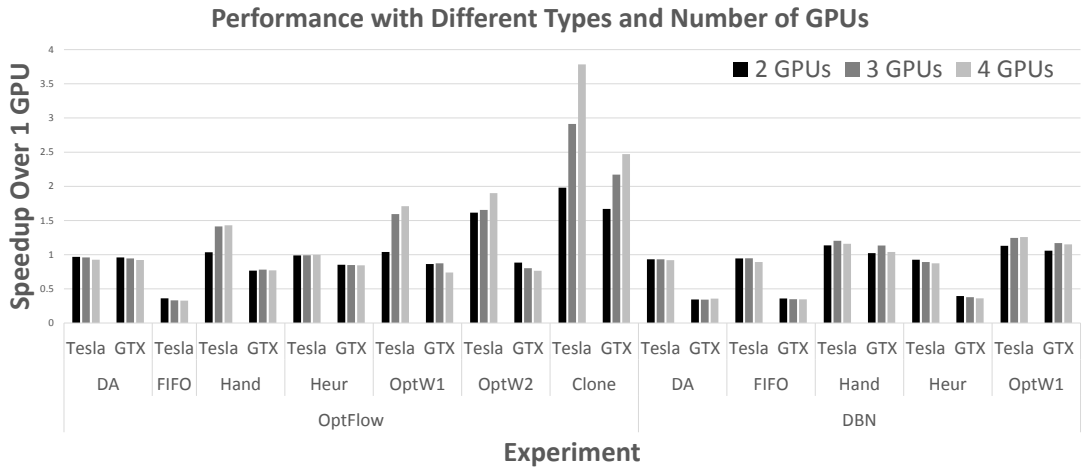

Figure 1: Speedup over 1 GPU under different scheduling policies for Tesla- and GTX-based systems

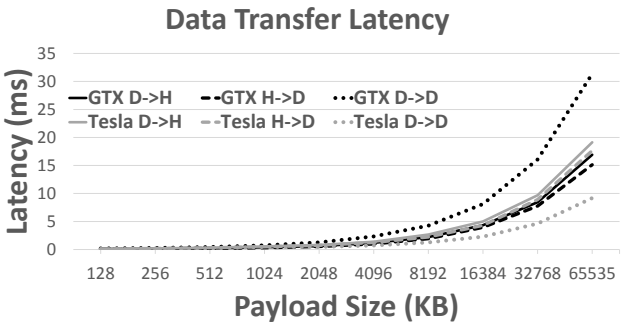

Figure 2: The latency in milliseconds of transfers to or from GPU memory, for different payload sizes. The time is shown for Host to Device (H->D), Device to Host (D->H) and Device to Device (D->D) transfer.

related sub-graphs based on the presence or absence of common programmer-defined structures. Close examination of the partitions produced by this policy reveals that this is a very challenging inference problem, in a context where an approximate answer is simply insufficient. A small number of poor assignments is sufficient to overwhelm performance for this policy, particularly on the GTX system where such mistakes are more costly. For both workloads, on multiple Teslas, the policy yields performance slightly under that of 1 GPU, while performance is marred on multiple GTXs, with losses of 20% and 65% for OptFlow and DBN respectively. In light of this general trend, we focus the remaining discussion on the Tesla-based system.

For DBN, the hand-optimized partition provided by the original developers of the workload obtains speed-ups between 13% and 20% on the Tesla system, with maximal performance at 3 GPUs. The optimal partitioner improves over this marginally, with speed-ups ranging from 13% to 25%. The optimal partitioner can beat the hand-optimized partition because the the latter is based on sub-graphs that are known to the programmer to be functionally related, which translates to coarseness in the partition. Specifically, each layer in the neural network trainer is assigned *in toto* to a single GPU. Examination of the optimal policy's generated partition reveals that it splits one of these layers across GPUs, in a way which reduces overall communication.

For OptFlow, the hand-optimized policy achieves more significant speed-ups of 41% on 3 GPUs and 43% on 4 GPUs, but the optimal partitioner still is able to improve over it, with OptW2 achieving speed-ups of 65% on 3 GPUs and 90% on 4 GPUs. The mean speed-up over the hand-optimized policy is 1.1× for OptW1 and 1.34× for OptW2. The very high variation in the runtimes for the hand-optimized policy on 3 and 4 Teslas is a factor in this. On occasion the hand-optimized policy does as well as the optimal partitioner, but it is not consistent. The causes of this variation are under investigation. One possible factor is that the hand-optimized partitioner does not explicitly consider the balance of graph vertices across the available GPUs. It should be noted that there are many potential hand-partition strategies for this (and most) workloads. Exploring that space in an exhaustive manner is typically not viable for the application developer. The fact that the optimal partitioner is able to select partitions that give good performance with predictability for both workloads is promising.

While the data show that well-chosen policy can enable some scaling for these workloads, the OptFlow data for the cloning policy show, predictably, that multi-GPU scaling for a truly data-parallel workload is significantly easier than for workloads with shared state. Cloning enables near-perfect scaling on the Tesla system, while topping out at 2.5× on the GTX system. The latter is bottlenecked by the copy-back of the final results from GPU to CPU memory. Factoring out copy-back costs yields near-perfect scaling on the GTX system as well.

Cloning is not a general solution in the absence of information about its safety from the programmer. On the other hand, in systems such as Delite or Dandelion, where graphs and GPU code are produced by the compiler from a limited set of operators, there may be hope that the compiler can help by re-writing graphs to enable cloning (for example, consider GroupBy-Aggregate re-writes that enable data parallel execution across multiple machines in a cluster). On

the other hand in cases where cloning is impossible, policies such the optimal partitioner can help provide some scaling without programmer involvement. Ultimately, we conclude there is reason to believe that transparently supporting policies that scale well across *d*iverse workloads remains a significant challenge, and the best policy is workload dependent.We take this as evidence that further exploration of the design and policy space is very much in order.

6. RELATED WORK

GPUs and Dataflow. StreamIt [47] and DirectShow [33] support graph-based parallelism. OmpSs [15], Hydra [49], PTask [38], and IDEA [19] all provide a graph-based dataflow programming models for offloading tasks across heterogeneous devices. Liquid Metal [27] and Lime [3] are programming platforms for heterogeneous targets such as systems comprising CPUs and FPGAs. Lime's filters, and I/O containers allow a computation to be expressed as a pipeline. Flextream [25] is a compilation framework for synchronous dataflow models that dynamically adapts applications to FPGA, GPU, or CPU target architectures, and Flextream applications are represented as a graph. Some of the policies we consider in our evaluation are supported by these systems; to our knowledge none of these systems support the graph partitioning approach we consider here.

Quincy [29] uses (min-cost flow) graph optimization, but not graph partitioning, and applies the concept in a datacenter context without attempting to accommodate heterogeneous compute.

Scheduling and Execution engines for heterogeneous processors. Scheduling for heterogeneous systems is an active research area: systems such as PTask [38], TimeGraph [32] and others [48] focus on eliminating destructive performance interference in the presence of GPU sharing. Maestro [41] also shares GPUs but focuses on task decomposition, automatic data transfer, and auto-tuning of dynamic execution parameters in some cases. Sun et al. [45] share GPUs using task queuing. Others focus on making sure that both the CPU and GPU can be shared [31], on sharing CPUs from multiple (heterogeneous) computers [8, 11], or on scheduling on multiple (heterogeneous) CPU cores [13, 7]. Several systems [34, 23] automatically choose whether to send jobs to the CPU or GPU [5, 6, 4, 18, 40], others focus on support for scheduling in the presence of heterogeneity in a cluster [15]. Several systems consider support a MapReduce primitive on GPUs, taking care of scheduling the various tasks and moving data in and out of memory [24, 17, 50]. The same abstraction can be extended to a cluster of machines with CPUs and GPUS [30]. This work was later improved with better scheduling [37]. Teodoro et al. describe a runtime that accelerates the analysis of microscopy image data sets on GPU-CPU clusters [46]. The resulting system relies on task-level dataflow to map the application to a heterogeneous platform, and requires support for cyclic structures in the di-graphs that express the image-processing pipeline; To our knowledge none of these systems support the graph partitioning approach we consider here.

7. CONCLUSION

Our examination of multiple points in the design and policy space suggests that using scheduling to realize speedup transparently from multiple accelerators is a significant challenge, and that a diverse set of policies is likely required to address the needs of diverse workloads. Partitioning based on weighted graph models does not address the needs of all workloads, but it can match or even outperform hand-optimized code for some. Even a fairly coarse model can be effective.

Automatically replicating a graph to exploit parallelism is highly effective in some cases, but is not always safe, and supporting this policy in a runtime therefore requires additional hints from the programmer.

Despite the fact that no clear best policy emerged from this investigation, the existence of performance-profitable policies for these workloads, which frustrated previous PTask schedulers, is reason for hope that the goal is realizable, and the results clearly point to avenues for further research toward it.

8. REFERENCES

[1] Top 500 supercomputer sites. 2011.

[2] Amazon. *High Performance Computing on AWS*, 2013.

[3] J. S. Auerbach, D. F. Bacon, P. Cheng, and R. M. Rabbah. Lime: a java-compatible and synthesizable language for heterogeneous architectures. In *OOPSLA*, 2010.

[4] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst. Data-Aware Task Scheduling on Multi-Accelerator based Platforms. In *16th International Conference on Parallel and Distributed Systems*, Shangai, Chine, Dec. 2010.

[5] C. Augonnet and R. Namyst. StarPU: A Unified Runtime System for Heterogeneous Multi-core Architectures.

[6] C. Augonnet, S. Thibault, R. Namyst, and M. Nijhuis. Exploiting the Cell/BE Architecture with the StarPU Unified Runtime System. In *SAMOS '09*, pages 329–339, 2009.

[7] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí. An extension of the starss programming model for platforms with multiple gpus. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 851–862, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] R. M. Badia, J. Labarta, R. Sirvent, J. M. Pï£¡rez, J. M. Cela, and R. Grima. Programming Grid Applications with GRID Superscalar. *Journal of Grid Computing*, 1:2003, 2003.

[9] S. Baker, E. P. Bennett, S. B. Kang, and R. Szeliski. Removing rolling shutter wobble. In *CVPR*, 2010.

[10] S. Baker, D. Scharstein, J. P. Lewis, S. Roth, M. J. Black, and R. Szeliski. A Database and Evaluation Methodology for Optical Flow. *IJCV*, 2011.

[11] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Scheduling strategies for master-slave tasking on heterogeneous processor platforms. 2004.

[12] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[13] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: a programming model for the cell BE architecture. In *SC 2006*.

[14] K. Brown, A. Sujeeth, H. Lee, T. Rompf, H. Chafi, and K. OLUKOTUN. A heterogeneous parallel framework for domain-specific languages. In *20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[15] 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.

[16] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 47–56, 2011.

[17] B. Catanzaro, N. Sundaram, and K. Keutzer. A map reduce framework for programming graphics processors. In *In Workshop on Software Tools for MultiCore Systems*, 2008.

[18] C. H. Crawford, P. Henning, M. Kistler, and C. Wright. Accelerating computing with the cell broadband engine processor. In *CF 2008*, 2008.

[19] J. Currey, S. Baker, and C. J. Rossbach. Supporting iteration in a heterogeneous dataflow engine. In *SFMA*, 2013.

[20] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.

[21] D. Delling and R. F. F. Werneck. Better bounds for graph bisection. In L. Epstein and P. Ferragina, editors, *ESA*, volume 7501 of *Lecture Notes in Computer Science*, pages 407–418. Springer, 2012.

[22] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: a domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM.

[23] D. Grewe and M. Oï£¡Boyle. A static task partitioning approach for heterogeneous systems using opencl. *Compiler Construction*, 6601:286–305, 2011.

[24] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 260–269, 2008.

[25] A. Hormati, Y. Choi, M. Kudlur, R. M. Rabbah, T. Mudge, and S. A. Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *PACT*, pages 214–223, 2009.

[26] B. K. P. Horn and B. G. Schunck. Determining Optical Flow. *Artificial Intelligence*, 17:185–203, 1981.

[27] S. S. Huang, A. Hormati, D. F. Bacon, and R. M. Rabbah. Liquid metal: Object-oriented programming across the hardware/software boundary. In *ECOOP*, pages 76–103, 2008.

[28] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys 2007*.

[29] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM.

[30] W. Jiang and G. Agrawal. Mate-cg: A map reduce-like framework for accelerating data-intensive computations on heterogeneous clusters. *Parallel and Distributed Processing Symposium, International*, 0:644–655, 2012.

[31] V. J. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *HiPEAC 2009*.

[32] 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*, 2011.

[33] M. Linetsky. *Programming Microsoft Directshow*. Wordware Publishing Inc., Plano, TX, USA, 2001.

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

[35] NVIDIA. *NVIDIA CUDA 5.0 Programming Guide*, 2013.

[36] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[37] V. T. Ravi, M. Becchi, W. Jiang, G. Agrawal, and S. Chakradhar. Scheduling concurrent applications on a cluster of cpu-gpu nodes. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, CCGRID '12, pages 140–147, 2012.

[38] C. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. Ptask: Operating system abstractions to manage gpus as compute devices. In *SOSP*, 2011.

[39] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In M. Kaminsky and M. Dahlin, editors, *SOSP*, pages 49–68. ACM, 2013.

[40] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP 2008*.

[41] K. Spafford, J. S. Meredith, and J. S. Vetter. Maestro: Data orchestration and tuning for opencl devices. In P. D'Ambra, M. R. Guarracino, and D. Talia, editors, *Euro-Par (2)*, volume 6272 of *Lecture Notes in Computer Science*, pages 275–286. Springer, 2010.

[42] H. Su, G. Li, D. Yu, and F. Seide. Error back propagation for sequence training of context-dependent deep networks for conversational speech transcription. In *ICASSP*, pages 6664–6668. IEEE, 2013.

[43] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, H. Chafi, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun. Optiml: An implicitly parallel domain-specific language for machine learning. In L. Getoor and T. Scheffer, editors, *ICML*, pages 609–616. Omnipress, 2011.

[44] A. K. Sujeeth, T. Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky, et al. Composition and reuse with compiled domain-specific languages. In *Proceedings of ECOOP*, 2013.

[45] E. Sun, D. Schaa, R. Bagley, N. Rubin, and D. Kaeli. Enabling task-level scheduling on heterogeneous platforms. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 84–93, 2012.

[46] G. Teodoro, T. Pan, T. Kurc, J. Kong, L. Cooper, N. Podhorszki, S. Klasky, and J. Saltz. High-throughput analysis of large microscopy image datasets on cpu-gpu cluster platforms. 2013.

[47] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *CC 2002*.

[48] 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*, ICS '11, pages 120–129, New York, NY, USA, 2011. ACM.

[49] 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. In *ASPLOS 2008*.

[50] P. Wittek and S. DaráNyi. Accelerating text mining workloads in a mapreduce-based distributed gpu environment. *J. Parallel Distrib. Comput.*, 73(2):198–206, Feb. 2013.