Supporting Iteration in a Heterogeneous Dataflow Engine

Jon Currey, Simon Baker, 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. These systems sacrifice generality for simplicity: while many workloads are easily expressed, important idioms like iteration and recursion are difficult to express and support efficiently. We consider the problem of extending a dataflow engine to support data-dependent iteration in a heterogeneous environment, where architectural diversity introduces data migration and scheduling challenges that complicate the problem.

We propose constructs that enable a dataflow engine to efficiently support data-dependent control flow in a heterogeneous environment, implement them in a prototype system called IDEA, and use them to implement a variant of *optical flow*, a well-studied computer vision algorithm. Optical flow relies heavily on nested loops, making it difficult to express without explicit support for iteration. We demonstrate that IDEA enables up to 18× speedup over sequential and 32% speedup over a GPU implementation using synchronous host-based control.

1 Introduction

This paper addresses programmability challenges in parallel applications running in the presence of heterogeneous potentially distributed compute resources, focusing primarily on GPUs. Compute fabric of this form is increasingly common: GPU-based super-computers are the norm [1], and cluster-as-service systems like EC2 make GPUs available [2]. Compute-bound algorithms are abundant, even at cluster scale, making acceleration with specialized hardware an attractive approach. However, while their toolchains are increasingly rich [26], programming these systems remains the province of expert programmers: indeed, the implementation of a well-tuned benchmark on distributed, heterogeneous hardware remains a publishable result [30]. The need for tools that make parallel heterogeneous systems accessible for non-expert programmers is urgent. Achieving this goal involves challenges at many layers of the technology stack including front-end programming tools and runtimes. In this paper we focus primarily on the runtime.

Developing parallel applications is characterized by a host of well-known difficulties such as synchronization, consistency, load-balancing, and resource management. Architectural heterogeneity introduces additional challenges such as discrete, non-coherent memory spaces, and diverse ISAs and compilers. Our previous work with PTask [28] demonstrated that a dataflow execution engine can make the runtime, rather than the programmer responsible for addressing many of these problems. However, PTask, like many dataflow systems such as MapReduce [10] and Dryad [18] , trades generality for simplicity by admitting only acyclic dataflow graphs. Expressing iteration in a dataflow engine requires support for cyclic or recursive graph structures: many data-parallel applications that rely heavily on looping constructs cannot be expressed as dataflow without it. Support for iteration in dataflow systems is a challenging problem, evinced by the wealth of active research in the area [7, 23, 11, 22, 25, 24]. The root cause of this challenge is the lack of centralized control that imperative languages depend upon to implement such abstractions.

This paper considers IDEA (**I**terative **D**ataflow **E**ngine for **A**ccelerators), a runtime for heterogeneous systems comprised of CPUs and GPUs. IDEA supports a set of primitives that enable a dataflow system to efficiently support algorithms with iterative, data-dependent control flow. We have used these primitives to implement a broad range of applications, including *k*means clustering, PageRank, deep neural networks and decision tree learning, and consider this to be reasonable evidence of their broad applicability. In this paper we focus on their use to implement *optical flow*, an embarrassingly parallel computer vision algorithm whose heavy use of nested loops makes it challenging to express and execute in a dataflow system. We use optical flow as a case study to examine the control flow primitives supported in

IDEA, show that naïve control-flow support (e.g. explicit loop-unrolling) is insufficient, and show that on a system with CPUs and GPUs, the primitives in IDEA enable a dataflow implementation of optical flow that outperforms a sequential CPU implementation by 18× and a GPU-based implementation based on traditional host-based control flow by up to 32%.

2 Motivation

We consider the problem of computing optical flow. Optical flow is the apparent motion of brightness patterns in a sequence of 2D images and is a common building block for image processing and computer vision algorithms, e.g. removing rolling shutter wobble from video [5]. Optical flow algorithms are well-studied [6]: Most approaches use iterative optimization of an energy function, either fixing the number of iterations, when low latency or deterministic performance is the primary objective, or iterating until the error in the motion drops below a threshold, when quality is paramount. Our implementation is a variation of the Horn-Schunck [17] algorithm which uses a pyramid of successively smaller rescaled versions of the input images. An optical flow value is calculated first for the smallest images (the top level of the pyramid) and the result for each level is used to seed the calculation for the next level. This approach allows the optimization at the lower levels of the pyramid to converge in fewer iterations than if Horn-Schunck was applied only to full size images, leading to improved performance.

For the purposes of this paper, it is sufficient to understand that the number of required pyramid levels grows slowly with the image size and that each pyramid level features two loops of nested iteration: an inner loop iteratively solves a linear system required for each iteration of the outer loop, which refines the flow estimate. Both the inner and outer loop can be implemented to terminate early based on data-dependent convergence tests.

Expressing this algorithm as dataflow without support for cycles in the graph, recursive graph structures, or other direct support for iteration is problematic. Data-dependent loop termination is impossible to express in such systems. While programmer-controlled loop unrolling could be used to implement a fixed number of iterations, the nested iterations will cause the size of the unrolled graph to blow up, for example, increasing the size of the graph by over 7× for a typical input size (720HD video). With many algorithms having iteration counts in the tens, hundreds or even thousands, unrolling loops is clearly an untenable approach.

```
void Laplacian(flow, result) {
  copyToGPU(flow);
  invokeKernel(Laplacian);
  copyFromGPU(result);
}
void LinearSystem(flowX, flowY, ..., result) {
  copyToGPU(flowX);
  copyToGPU(flowY);
  // ...
  invokeKernel(LinearSystem);
  copyFromGPU(result);
}
for(...level...) {
  // ...
  Laplacian(flowX[level], flowLapX[level]));
  Laplacian(flowY[level], flowLapY[level]));
  LinearSystem(flowLapX[level], flowLapY[level],...);
  //...
}
```

Figure 1: Pseudo-code for an excerpt of a GPU-offloaded implementation of *optical flow*.

2.1 Dataflow for optical flow

Dataflow is an attractive programming and execution model for this workload on a heterogeneous system for a number of reasons. First, the graph eliminates the need for control to be orchestrated centrally, simplifying scheduling. Second, the graph *only* expresses tasks and producer-consumer relationships among them so the programmer does not write code for communication, synchronization, and data movement. Finally, the graph expresses all the inherent parallelism in the workload, enabling the runtime to exploit it.

Consider the pseudo-code in Figure 1, which represents an excerpt from a traditional implementation of GPU-offload for our optical algorithm. The code computes a Laplacian over flow values in the X and Y dimensions, both of which results are consumed by a linear system solver computation. Two points are most salient. First, control flow is orchestrated synchronously on the CPU, which coordinates execution on the GPU. Second, because most GPUs have private memory spaces, an implementation based on any modern GPU framework (e.g. CUDA [26]) will feature the explicit communication code that dominates the code. The naïve composition of calls results in multiple needless round trips between CPU and GPU memory for the intermediate results produced by `Laplacian`. Moreover, the fact that the `Laplacian` computations for the X and Y dimensions can be computed in parallel is not expressed by the sequential code. These shortcomings have obvious so-

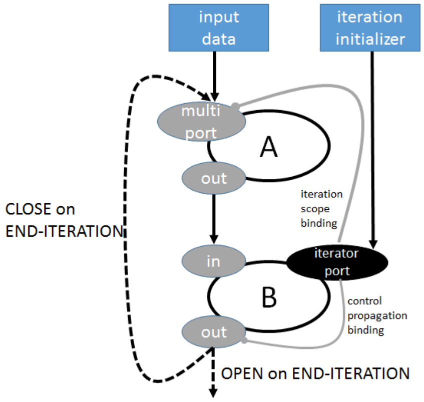

Figure 2: A dataflow graph using our proposed control flow constructs to perform an iterative computation.

lutions but they require programmer intervention. A dataflow framework avoids the coupling of algorithm, data movement, and scheduling that are difficult to avoid in a traditional implementation.

3 Design

This section considers the design of our system, called IDEA which extends a basic DAG-based dataflow execution engine with constructs that can be composed to express iterative structures and data-dependent control flow.

3.1 Base Dataflow System

IDEA re-implements and extends the abstractions proposed in PTask [28], with the caveat that IDEA is entirely in user-mode. We adopt the nomenclature from PTask, a *token model* [9] dataflow system: computations, or nodes in the graph are *tasks*, whose inputs and outputs manifest as *ports*. Ports are connected by *channels*, and data moves through channels discretized into chunks called *datablocks*. The programmer codes to an API to construct graphs from these objects, and drives the computation by pushing and pulling datablocks to and from channels. Tasks execute when a datablock is available at all of its input ports. Following the example of PTask, a unique thread manages each task in the graph, allowing IDEA to overlap data movement and computation for different tasks. We use the PTask approach to scheduling tasks.

3.2 Control Flow Constructs

Our base system, like most dataflow engines, admits only computations whose graphs are acyclic (DAGs).

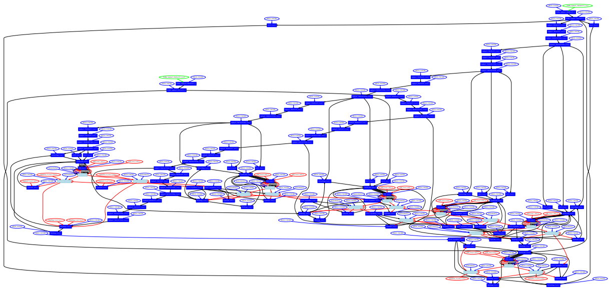

Figure 3: The dataflow graph for our optical flow implementation

This is a fundamental limitation, as expressing iteration in a graph requires, at minimum, support for either cyclic graphs, or for recursion.[1] We avoid the latter approach because recursion requires a dynamic graph structure, which complicates synchronization and optimization significantly. Classical dataflow models suggest specialized nodes such as predicates, “selectors”, and “distributors” [9], which can be composed in a cyclic graph to express arbitrary iteration. We take inspiration from this approach, but avoid expressing these constructs as first-class class nodes in the graph, largely because making light-weight operations like predicate evaluation and routing schedulable entities complicates the scheduler and makes it difficult to preserve locality. For example, a predicate node may be scheduled on a compute resource such that large amounts of data must be moved to execute a handful of instructions: the scheduler can be coded to try to avoid this, but it is simpler if predicates and routing functionality can be piggy-backed onto existing structures. Consequently, IDEA’s control flow constructs extend the functionality of ports and channels. IDEA Uses the following abstractions:

ControlSignals. IDEA graphs carry control signals by annotating datablocks with a control code. The programmer defines arbitrary flow paths for these signals using an API to define a *control propagation pair*, which connects port pairs. Any control code received on a datablock received at the first port, will be propagated to the datablock on the second port. Examples of control signals include `BEGIN/END-STREAM` and `BEGIN/END-ITERATION`.

MultiPort. A MultiPort is a specialized input port that can be connected to multiple (prioritized) input channels. If a datablock is available on any of the input channels, the MultiPort will dequeue it, preferring the highest priority channel if many are ready.

[1] It is commonly argued that programmer-managed loop unrolling allows a programmer to express iteration in a DAG. We address this argument in Section 2.

PredicatedChannel. A PredicatedChannel allows a datablock to pass through it if the predicate holds for the datablock. The API for configuring predicates allows the programmer to define whether the datablock is dropped from the channel or queued for later re-evaluation when the predicate fails. In general, the predicate function is a programmer-supplied callback, but we provide special support common predicates such as open/close on control signals such as `BEGIN/END-ITERATION`.

InitializerChannel. An InitializerChannel provides a pre-defined initial value datablock. InitializerChannels can be predicated similarly to PredicatedChannels: they are always ready, except when the predicate fails. InitializerChannels simplify construction of sub-graphs where the initial iteration of a loop requires an initial value that is difficult to supply through an externally exposed channel.

IteratorPort. An IteratorPort is a port responsible for maintaining iteration state and propagating control signals when iterations begin an end. An IteratorPort maintains a list of ports within its *scope*, which are signaled when iteration state changes. An IteratorPort also propagates `BEGIN/END-ITERATION` control signals along programmer-defined control propagation paths, which in combination with backward/forward PredicatedChannels can conditionally route data either to the top of another iteration, or forward in the graph when a loop completes. IteratorPorts can use callbacks to implement arbitrary iterators, or select from a handful of pre-defined functions, such as integer-valued loop induction variables.

Collectively, these constructs allow us to implement rich control flow constructs and iteration without additional specialized task nodes. Scheduling and resource-management for tasks maintains conceptual simplicity, and routing decisions are always computed locally by construction.

Care must be taken in the configuration of the above constructs to avoid the introduction of unwanted non-determinism. To do this, the inputs to a MultiPort must all be predicated channels, with predicates configured such that at at any given moment at most one channel will allow datablocks to pass through. Note however that non-determinism may be acceptable, and even desirable, in some applications. For example to allow use of a value from one channel until an update is available via another channel.

An example of how these constructs can be used to orchestrate a simple iterative computation is shown in Figure 2. In this figure, white ovals represent tasks, gray ovals represent ports, and the black oval represents an IteratorPort. Solid arrows represent channels, and dashed arrows represent PredicatedChannels. The computation executes a loop whose body is `A()` followed by `B()`. The MultiPort input to A is added to the *scope* of the IteratorPort on B, and the port labeled *out* on B is set up with a control propagation path from the IteratorPort (represented by a gray arc in the Figure). The PredicatedChannels going backward and forward from the *out* port are configured with `CLOSE/OPEN-on-END-ITERATION` respectively. Because the IteratorPort will set the control signals on blocks leaving the *out* port, only datablocks annotated with an `END-ITERATION` control code can flow forward, and only those not so annotated can flow backward. The exposed PredicatedChannel entering A accepts only blocks annotate with `BEGIN-ITERATION`. When an iteration state datablock is pushed into the IteratorPort it signals all objects in its scope to annotate the next block received with a `BEGIN-ITERATION` control code. Consequently, datablocks follow the cyclical path until a loop completes, after which a new datablock can be consumed from the forward PredicatedChannel at the top of the graph.[2]

4 Implementation

We implemented the optical flow algorithm in IDEA using the control flow constructs described in Section 3. In particular, the pattern described in Figure 2 was used to implement the two nested loops within each level of the image pyramid. Figure 3 shows the dataflow graph of our implementation. 18 distinct kernels are used, comprising approximately 1700 lines of HLSL code. Graph construction and driver code comprise about 1600 lines of C++. In this realization, graph construction is via a low-level API, where each task, port and channel must be added explicitly. Higher-level tools that allow more concise expression and preclude mis-configration would be advantageous and we are actively investigating them.

The inner loop, whose body contains 2 tasks with 8 parameters each, has a dataflow analog constructed using 2 InitializerChannels, 10 PredicatedChannels and a single IteratorPort to manage the iteration state. The InitializerChannels are set to be open initially and to re-open on `END-ITERATION`, since we want them to open then to seed a new iteration. Each instance of the outer loop, whose body contains requires 20 tasks (including the two for the inner loop),

[2] We note that a limitation of this design is that execution of different iterations cannot be pipelined, which compromises some of the available concurrency. Extending this design to pipeline the execution of multiple iterations is possible, but at a significant cost in increased complexity.

uses 6 predicated InitializerChannels (similarly configured) 6 PredicatedChannels for backward flow to the top of the loop, along with 2 PredicatedChannels for data flowing to the next pyramid level. The IteratorPort for these outer loops must be bound to 4 MultiPorts on 3 parallel tasks.

We note that the pyramid levels could be implemented as a third, outer loop, which would reduce the size of the graph. However, our GPU kernels are implemented in DirectCompute, which requires that the number of GPU threads required to execute the kernel be fixed at the time of kernel compilation. We also choose to process a fixed number of pixels per kernel thread. Hence we cannot reuse a task at different levels of the image pyramid and the pyramid levels must be unrolled in our implementation. Each level of the pyramid adds 20 tasks to the graph, so the the total graph size is 136 vertices in the 720HD video case, requiring 5 pyramid levels, and 176 vertices in the 4K video case (7 pyramid levels).

5 Evaluation

We evaluate our implementation of optical flow by comparing its performance against a sequential CPU-only implementation and a GPU-based implementation that uses the same GPU kernels and a sequential driver program with synchronous control flow. Measurements were taken on a Windows 7 x64 machine with a 2.67 GHz 4-core Intel Xeon (W3520) with 12GB RAM, and an NVIDIA GTX 580 GPU (512 cores, 3GB memory).

We use input image sequences of various dimensions, representing common video sizes: VGA (640x480), 720HD (1280x720), 1080HD (1920x1080) and 4K 'Ultra HD' (4096x2160). Figure 5 reports speedup over the CPU implementation for 3 and 5 outer and inner iterations at each level of the pyramid respectively; data are the average over 5 runs. Results for other iteration counts are similar. Both GPU-based implementations outperform the CPU-based implementation, by margin that increases with image size. The IDEA-based implementation consistently outperforms the synchronous control GPU implementation. IDEA's relative performance peaks for 720HD and 1080HD video (32% and 17% speedup, respectively), because setup and communication costs are dominant at the smaller image size (only 7% for VGA) and GPU execution latency dominates at 4K Ultra HD, (only 2%). These benefits are made possible by the asynchrony in the IDEA dataflow system; without our proposed abstractions, the workload could not be coded as dataflow.

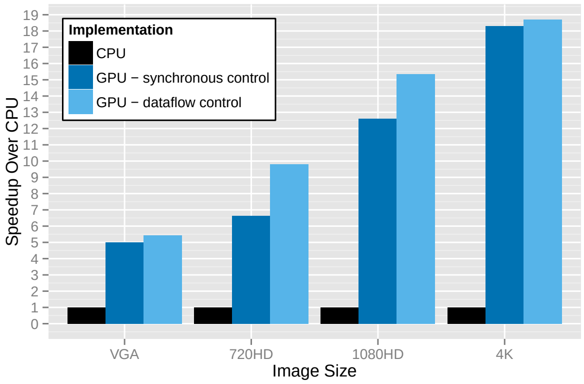

Figure 4: The speedup over CPU of GPU without and with flow control, at different image sizes

6 Related work

Heterogeneous Compute Enginges. Heterogeneous compute engines such as PTask [28], StarPU [4], MATE-CG [19], Mars [14] and others [27, 8] address the same programmability problem addressed IDEA. IDEA could benefit from some of the resource-management techniques used in these systems. IDEA's control flow support expands the set of applications that can target such systems.

Dataflow. IDEA extends our own previous work with PTask [28]. PTask supports dataflow programming at the OS-interface, and in particular, accepts only computations that can be expressed as DAGs. IDEA is a user-mode runtime that addresses PTask's limitation by supporting abstractions that enable cyclical graphs. Applications written in StreamIt [29] and DirectShow [21] are expressed as graphs of nodes which send and receive items to each other over channels. StreamIt and DirectShow are programming models with dedicated compiler support, while IDEA is a runtime only system that extends the set of expressible graphs.

Sponge [16],and other languages [20] such as LUSTRE [13] , along with Lime [3] Flextream [15] propose language and/or compiler support for SDF model dataflow graphs. IDEA is not an SDF system and by avoiding dedicated nodes to represent control flow constructs, the abstractions we propose for IDEA are a departure from previous methods for supporting control flow within a dataflow environment.

Dryad [18] is a graph-based fault-tolerant programming model for distributed parallel execution in data center. IDEA's control flow support allows a developer to write applications that cannot be easily expressed in Dryad. Extending dataflow systems to support iteration [7, 11, 22, 25] or incremental iterative computation [23, 12, 24] is an active research area. IDEA's approach of piggybacking control flow constructs on existing graph structures is unique in this space.

7 Conclusion

Many data parallel algorithms like optical flow require data-dependent control flow that is difficult to implement efficiently in existing dataflow systems. We have proposed control flow abstractions for IDEA that enable a performant implementation at reasonably programming complexity.

We show that with these abstractions it is possible to achieve the performance benefits of dataflow (over a traditional synchronous approach) for workloads that previously could not be implemented in such frameworks.

Using these abstractions correctly proved to be challenging, with much time spent debugging misconfigured graphs. This fits with the general complexity of manually designing dataflow graphs for non-trivial applications, and confirms the need for higher-level tools that either allow more concise graph expression or generate the graph from an entirely different representation of the application.

As future work we plan to create such tools as well as extend IDEA to support pipelining multiple outstanding iterations, and investigate distributed execution.

References

[1] Top 500 supercomputer sites. 2013.

[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*. ACM, 2010.

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

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

[6] 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.

[7] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, Sept. 2010.

[8] B. Catanzaro, N. Sundaram, and K. Keutzer. A map reduce framework for programming graphics processors. In *WSTMS*, 2008.

[9] A. L. Davis and R. M. Keller. Data flow program graphs. *IEEE Computer*, 15(2):26–41, 1982.

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

[11] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *HPDC '10*. ACM, 2010.

[12] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. *VLDB*, 2012.

[13] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[14] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a mapreduce framework on graphics processors. In *PACT*. ACM, 2008.

[15] 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*. IEEE Computer Society, 2009.

[16] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. Sponge: portable stream programming on graphics engines. In *ASPLOS*, 2011.

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

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

[19] W. Jiang and G. Agrawal. Mate-cg: A map reduce-like framework for accelerating data-intensive computations on heterogeneous clusters. *PDPS*, 0, 2012.

[20] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36:24–35, January 1987.

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

[22] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*. ACM, 2010.

[23] F. McSherry, R. Isaacs, M. Isard, and D. G. Murray. Composable Incremental and Iterative Data-Parallel Computation with Naiad. 2012.

[24] S. R. Mihaylov, Z. G. Ives, and S. Guha. Rex: recursive, delta-based data-centric computation. *Proc. VLDB Endow.*, 5(11):1280–1291, July 2012.

[25] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: a universal execution engine for distributed data-flow computing. In *NSDI*, 2011.

[26] NVIDIA. *NVIDIA CUDA Programming Guide*, 2011.

[27] V. T. Ravi, M. Becchi, W. Jiang, G. Agrawal, and S. Chakradhar. Scheduling concurrent applications on a cluster of cpu-gpu nodes. In *CCGRID*. IEEE Computer Society, 2012.

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

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

[30] K. Ueno and T. Suzumura. Highly scalable graph search for the graph500 benchmark. In *HPDC*, 2012.