MERCATOR: a GPGPU Framework for Irregular Streaming Applications

Stephen V. Cole and Jeremy Buhler
Washington University
St. Louis, MO, USA
{svcole,jbuhler}@wustl.edu

***Abstract*—GPUs have a natural affinity for streaming applications exhibiting consistent, predictable dataflow. However, many high-impact *irregular* streaming applications, including sequence pattern matching, decision-tree and decision-cascade evaluation, and large-scale graph processing, exhibit unpredictable dataflow due to data-dependent filtering or expansion of the data stream. Existing GPU frameworks do not support arbitrary irregular streaming dataflow tasks, and developing application-specific optimized implementations for such tasks requires expert GPU knowledge.**

We introduce MERCATOR, a lightweight framework supporting modular CUDA streaming application development for irregular applications. A developer can use MERCATOR to decompose an irregular application for the GPU without explicitly remapping work to threads at runtime. MERCATOR applications are efficiently parallelized on the GPU through a combination of replication across blocks and queueing between nodes to accommodate irregularity. We quantify the performance impact of MERCATOR's support for irregularity and illustrate its utility by implementing a biological sequence comparison pipeline similar to the well-known NCBI BLASTN algorithm.

MERCATOR code is available by request to the first author.

Keywords – streaming dataflow; parallel computing; SIMD; GPU; irregular computation

I. Introduction

With the exponential increase of available data across all disciplines in the last decade has come a corresponding need to process that data, leading to a rise in popularity of the streaming computing paradigm and of the use of GPUs as wide-SIMD streaming-data multiprocessors [1], [2]. We use *streaming computing* to denote a method of processing data that has the following characteristics:

- The input data set is assumed to be of unbounded size, either because it is finite but huge or because new inputs are continuously produced (e.g. when processing a live video or sensor data stream in real time).
- Each input item must be processed by performing computations on the item's data.
- Each input item may generate zero or more output items when processed.
- Performance comes from optimizing total throughput (input items consumed per unit time) as opposed to latency per item.

Work supported by NSF CNS-1500173 and by Exegy, Inc.

The size and type of a "data item" are application-specific and therefore programmer-defined. The computation to be performed is described by a graph consisting of compute nodes connected by dataflow edges. Application execution streams input data through the graph to produce outputs.

Streaming computing on GPUs has been used to accelerate applications with regular, predictable memory access and computation patterns. However, many high-impact applications exhibit behaviors that are obstacles to performance optimization on a GPU [3]–[5]. These "irregular" applications have become targets of optimization with the rise of GPGPU computing.

Some authors, such as Zhang [3] and Burtscher et al. [4], define irregular applications in terms of operations directly tied to GPU performance, such as control divergence and irregular memory access patterns. Others, such as Pingali et al. [5], define them by the types of data structures on which they operate, e.g., arbitrary graphs rather than dense matrices. We define irregular streaming applications by the property that *they include computational steps that produce a variable, data-dependent number of outputs per input.* Examples of high-impact irregular streaming applications from scientific and engineering domains include biological sequence alignment [6], network packet filtering (as NFA matching [7]), telescope data processing [8], and big graph algorithms [9], [10].

With wide SIMD, lightweight threads, and low-cost thread-context switching, GPUs allow considerable flexibility in the way application work is assigned to threads. However, irregular applications are challenging to map efficiently onto a GPU because data-dependent filtering or replication of items creates an unpredictable data wavefront items ready for further processing. This wavefront, which may comprise items stored sparsely at different stages of the computation, must be efficiently mapped to SIMD threads to maximize *occupancy*, that is, to provide work to as many threads (which are actually lanes in wide SIMD instructions) as possible at all times. Straightforward implementations of irregular applications on a GPU are therefore prone to load imbalance and reduced occupancy, while more sophisticated implementations require advanced use of, e.g., SIMD parallel scatter/gather or atomic operations to redistribute work efficiently among threads.

In this work, we describe MERCATOR, a framework to transparently support optimizations applicable to irregular streaming dataflow on NVIDIA GPUs. Given a graph specifying a streaming application's topology and CUDA code

978-1-5386-3250-5/17 $31.00 © 2017 IEEE
DOI 10.1109/HPCS.2017.111

for computations to be performed in each node, MERCATOR automatically generates CUDA code to manage the application's execution on the GPU, including all data movement between nodes and a scheduler for running the application to completion. MERCATOR's principal technique to mitigate the impact of irregularity is to gather and queue data items between nodes. We implement queues transparently to the application developer, using SIMD-parallel operations to minimize their overhead. Queueing supports applications with directed cycles in their dataflow graphs and exposes opportunities for optimization, such as concurrent execution of compute nodes with the same code.

To characterize MERCATOR's behavior, we first use synthetic application kernels to quantify performance impacts of its support for irregular execution, then demonstrate implementation of a more complex application from the domain of biological sequence comparison.

The rest of the paper is organized as follows. Section II describes related work. Section III describes the MERCATOR application model, while Section IV extends this model to the GPU. Section V presents the MERCATOR framework and its programmer interface. The remaining sections present results and future work.

II. Related Work

Other streaming data-flow models. Past work in streaming computing has focused on exploiting *task parallelism*, in which an application is broken into 'tasks' that may be run in parallel by independent heavyweight threads on distributed systems, subject to the data dependencies among tasks. Many Models of Computation (MoCs), beginning with Kahn process networks [11] and continuing with models that place various restrictions on data flow rates and node execution characteristics, have been designed to model task parallelism, with tasks represented as the compute nodes of a dataflow graph.

One of the most restrictive yet best-studied MoCs for streaming computation is the Synchronous Data Flow (SDF) model [12], in which the number of data items produced and consumed by each node's firing is known *a priori* for each application. SDF models many digital signal processing applications, such as audio filters, video encoders/decoders, and software-defined radios. SDF applications can be optimally scheduled at compile time for a uniprocessor or multiprocessor system [13], and frameworks such as StreamIt allow programmers to develop SDF applications for GPUs [14], [15]. In contrast to SDF applications, the irregular applications we seek to support allow *data-dependent* production and consumption rates at each computation stage, making compile-time load balancing impossible.

Less restrictive MoCs for task-based streaming computing, such as Boolean Data Flow [16], Dynamic Data Flow [17], and Scenario-Aware Data Flow [18], can express applications composed of modules with different running times and work-to-thread mappings, at the cost of weaker scheduling and space requirement guarantees. The Ptolemy Project [19] maintains a framework supporting applications conforming to these MoCs, though they do not target GPUs. However, MERCATOR enables expression of irregular applications in a way that naturally exposes not only coarse-grained task parallelism due to modularization but also fine-grained *data parallelism* enabled by the GPU's wide-SIMD architecture.

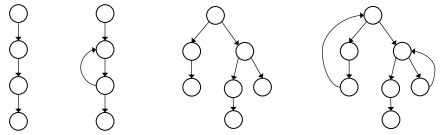

Fig. 1: Example application topologies, including pipelines and trees with or without back edges.

GPU support for irregular applications. Several previous works [20]–[22] have developed support for efficient execution of irregular, task-parallel applications on a GPU. A key component of these works is a task scheduling framework that runs entirely on the GPU, allowing blocks to repeatedly pull/push and execute work with no CPU intervention until all tasks are complete. MERCATOR adopts a related scheduling strategy to minimize host-GPU coordination overhead. However, these existing systems again do not offer abstractions that specifically target the fine-grained data parallelism inherent in streaming dataflow.

The theoretical GPU streaming execution framework presented in [23] uses a channel data structure [24] similar to our queues, with work aggregated by function type and a scheduling strategy designed to maximize SIMD lane occupancy. However, this framework requires more intervention by the application programmer than does MERCATOR to manage dataflow. More importantly, implementing the channel framework would also require hardware changes to the GPU, whereas MERCATOR is entirely software-based.

OpenACC [25], like MERCATOR, allows programmers to augment their code with parallelization suggestions designed to abstract details of GPU data movement and execution from a programmer. However, it primarily targets regular computations and small code snippets and may therefore require frequent host-GPU coordination to manage irregularity between stages of a computation.

Existing approaches to optimizing irregular applications on wide-SIMD architectures share common successful elements but are application-specific in their implementations. In [7], Ren et al. introduce a *stream compaction* step on the CPU that allows for efficient storage and execution of yet-to-be-traversed tree nodes in the presence of arbitrary, non-uniform path cutoffs. Our WOODSTOCC DNA sequence alignment application [26], [27] uses parallel reduction techniques on a GPU similar to Ren's stream compaction to maintain dense work queues of candidate DNA reads, while [28] uses similar techniques to optimize graph-processing algorithms on a GPU. MERCATOR seeks to generalize these enhancements into a development framework for diverse applications.

Domains of application. Irregular streaming dataflow computations arise often as part of high-performance scientific

and engineering computation. Below, we give examples of such applications organized by the *topology* of the dataflow computation, i.e. the connections among its computational operations. Fig. 1 gives examples of these topologies.

Linear pipelines with filtering. A common topology is that of a simple linear pipeline, each of whose stages may discard some of its inputs. Examples include telescope image processing [8], Monte Carlo financial simulation [29], and Viola-Jones face detection [30].

Linear pipeline with data-dependent feedback. Pipelines may be augmented with feedback edges to process each input a data-dependent number of times. Examples include recursive algorithms such as those vectorized in [31], fixed-point GPU algorithms (e.g. points-to analysis) in the LonestarGPU benchmark suite [4], rejection sampling algorithms such as the ziggurat RNG [32], pattern-matching algorithms such as NFA processing [7] used for, e.g., network packet inspection and security scanning, and iterative mathematical functions with data-dependent convergence, such as the Mandelbrot set [33].

Tree with or without feedback. Generalizing a pipeline of filtering stages results in a tree topology. A natural example is a decision tree, as used in e.g. random forest evaluation [34]. Also, a linear pipeline may be replicated to divide heterogeneous inputs into multiple homogeneous streams according to some characteristic (e.g. window size in Viola-Jones). Replicating a pipeline with feedback gives a tree with feedback.

Below, we describe how we efficiently support these application topologies in MERCATOR.

III. MERCATOR Application Model

In this section, we describe a sequential version of the irregular dataflow application model underlying MERCATOR. The next section describes how we augment this model to accommodate SIMD execution on the GPU.

A MERCATOR application consists of a directed graph of compute nodes connected by edges along which data flows. Nodes are assumed to implement relatively "heavy-weight" computations requiring tens to hundreds of milliseconds. Each node has a single input channel, from which it receives a stream of inputs, and may have zero or more output channels, on which it emits streams of outputs. Channels are typed according to their contents; an edge may connect an output channel only to an input channel of the same type. A single designated node with no input channel is the *source*; its input stream is supplied to the GPU by the host processor. A node with no output channels is a *sink*; it may only return data from the GPU to the host.

A node implements a user-defined computation over its input stream. For this section (only), assume that a node processes inputs one at a time. A node with output channels $c_1 \dots c_k$ may, for each input it consumes, generate between zero and n_j outputs on channel c_j. The maximum values n_j are known statically, but the actual number of outputs per input may vary dynamically at runtime. If $n_j = 1$, we say that a node *filters* its input onto channel c_j; that is, each input produces at most one output.

To execute an application, we repeatedly select a node with at least one available input to *fire*, i.e. to consume one of its inputs, perform a computation on that input, and generate any output that may result. An application completes when no node has any input left to process. Which node should fire next is determined by a scheduler that is part of MERCATOR's runtime system.

A. Topologies and Deadlock Avoidance

Inputs to a node are assumed to queue on its input channel until they are consumed. If multiple edges point to a node, data flowing in from all edges is placed on the node's input queue in some arbitrary order. Queues have a fixed, finite sizes.

If a node N can generate up to n outputs from an input on some output channel c, and c is connected to an input channel for node N', then N cannot fire unless the queue of N' has at room for at least n items. If N cannot fire, we say that it is *blocked* by N'. The MERCATOR scheduler detects blocked nodes by inspecting the queues at the heads of their outgoing edges and will not fire a blocked node. Any node other than a sink may sometimes become blocked, depending on the order in which nodes are fired.

Applications with directed cycles can potentially *deadlock*, i.e. reach a state where at least one node has queued inputs but every such node is blocked and so is unable to fire. Whether an application with any particular topology can deadlock depends on queue sizes, filtering behaviors, and the scheduling policy.

To ensure that deadlock cannot occur, we first restrict the set of permitted graph topologies for MERCATOR applications, then define a scheduling rule for nodes that ensures progress after finite time. Deadlock is impossible if application topology is restricted to a tree rooted at the source, because any acyclic path of blocked nodes terminates on a sink or another unblocked node and hence will unblock after finite time. However, support for non-tree-like topologies is desirable, in particular *feedback loops* for applications that may process items a variable, data-dependent number of times.

A tree can be augmented with *back edges* that point from a node N to one of its ancestors M. A back edge $N \to M$, together with the path $M \rightsquigarrow N$, forms a loop (self-loops with $N = M$ are permitted). MERCATOR requires that

- Loops may not overlap or nest; that is, no node on the forward path $M \rightsquigarrow N$ of a loop may be the target of a back edge, other than the edge $N \to M$.
- For each node in a loop, the output channel that participates in the loop must produce at most one output per input to the node.

These two criteria are easily verified for an application at compile time by a linear-time traversal of its tree.

Even this limited set of topologies permits deadlock under certain schedules. In particular, if all nodes in a loop have full queues, then every node in the loop is blocked by its successor, and the application cannot make progress. However, it can be shown (proof omitted for space reasons) that, if each node's queue can hold at least two items, the following scheduling rule suffices to prevent deadlock: *if M is the head of a loop*

$M \rightsquigarrow N \rightarrow M$, do not fire M's parent node if M has only one free slot in its input queue.

With these restrictions, MERCATOR can support all the types of application topologies shown in Fig. 1. Future work will consider the limits imposed by more complex topologies, such as DAGS and nested loops.

IV. Parallelization on a SIMD Platform

In this section, we describe how MERCATOR applications are parallelized on SIMD hardware, in this case an NVIDIA GPU. We exploit the streaming nature of computations, the wide-SIMD features of the GPU, and code shared between compute nodes of an application to parallelize not only execution of the user's functions but also our supporting infrastructure.

A. Parallel Semantics and Remapping Support

A MERCATOR application is realized as a single GPU kernel. The host processor supplies an input memory buffer containing the source node's input stream and one output buffer per sink node to receive results. The host then calls the application, which transfers the input buffer to the GPU's global (i.e. DRAM) memory, processes its contents, writes any results to global memory, and finally transfers them back to the host-side output buffers. No intermediate host-device control transfers occur during application execution – all nodes, the scheduler, and any supporting code run entirely on the GPU. This *uberkernel* design [21] avoids overhead related to either control or data transfers between host and device.

Coarse-grained Replication. To avoid overhead due to coordination among multiprocessors on the GPU, MERCATOR instantiates an application within one thread block, which runs on a single multiprocessor. To utilize the entire GPU, multiple blocks are launched, each with its own instance of the application. Application instances in different blocks do not share queues or coordinate their execution, except to concurrently acquire inputs from a global input buffer or write to global output buffers. These buffers are respectively read-only and write-only, so access to them is coordinated simply by atomic updates to their head and tail pointers, respectively.

Fine-grained Mapping. Within a single block, we extend the semantics of MERCATOR applications to utilize multiple SIMD threads as follows. Each firing of a compute node now consumes an *ensemble* of one or more input items from its queue. Each item is mapped to a GPU thread, and the node executes on all items of an ensemble in parallel. As in the sequential semantics described previously, each thread may dynamically generate different numbers of outputs per input on each of the node's output channels, up to some predefined maximum number of outputs per channel. During node execution, each thread "pushes" its outputs to the relevant output channel via a call to a function supplied by the MERCATOR runtime. All outputs pushed during a firing are gathered by MERCATOR and forwarded to the appropriate downstream nodes' queues.

The width of an ensemble is the thread width of a block (at most 1024 threads on current NVIDIA GPUs), which may be limited by the user if execution of a node or the entire application would otherwise require too many resources per thread. If a node fires with fewer inputs than the block width, any GPU threads without an associated input remain idle for the duration of the firing.

The use of ensembles together with inter-node queueing is MERCATOR's fundamental tool for realizing irregular applications on the GPU. Each node in an application may have different, dynamic filtering behavior on its inputs. By interposing queues between nodes, MERCATOR can alter the thread-to-data-item mapping between nodes, so that each firing utilizes a contiguous set of threads. If firings are scheduled so that at least a block's width of items is usually available, the block's threads will usually be fully occupied.

B. Efficient Support for Runtime Remapping

Because MERCATOR applications execute entirely on the GPU, remapping of items to threads between nodes must be done in parallel to avoid incurring unacceptable overhead.

SIMD Queue Management. The firing of a MERCATOR node is structured as follows. First, the number of input items to be consumed by the firing is determined. This number is capped both by the number of available inputs on the node's queue and by the number of free output slots on the queues at the heads of its outgoing edges. The blocking rule and deadlock avoidance scheme described for sequential semantics extends straightforwardly to the parallel case – free queue space is simply measured in units of ensembles rather than individual items.

Once the number of inputs to consume is known, MERCATOR repeatedly gathers one ensemble's worth of items from the node's input queue and calls the node's code to process the ensemble. Output items pushed by the node on a given channel are stored temporarily in an output buffer for that channel. If a node executes n threads and may push up to k outputs per input, the buffer has size nk (or a multiple of nk, to support multiple calls between queue updates within one firing). If not all threads in a block push an item at the same time, the output buffer may be sparsely occupied. When enough calls have been made to possibly fill the output buffers, MERCATOR identifies all nonempty slots in each channel's buffer using a SIMD parallel scan and then concurrently compacts the items onto the channel's downstream queue.

Queues and output buffers are stored in global memory, so their size is not strongly constrained. Moreover, because each queue is accessed only within one block, and firings within a block are handled sequentially, there is no need for locking or atomic access to manage a queue's head or tail pointers. Indeed, to minimize global memory accesses, MERCATOR only updates these pointers once per queue per firing, even across repeated calls to the node's code.

Concurrent Execution of Nodes With Identical Code. Multiple nodes in a MERCATOR application may implement the same computation. For example, a decision cascade such as that used by the Viola-Jones face recognition algorithm [30] might consist of a chain of nodes, each implementing the

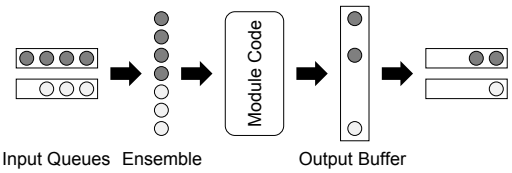

Fig. 2: Schematic of steps in firing a module, in this case combining two nodes. Items are gathered from the nodes' input queues to form an input ensemble, which is processed by the module's code. Any outputs produced by the module are stored in the output buffer, from which they are scattered to their downstream queues. Shading denotes tags indicating the node associated with each item.

same filtering algorithm on its input stream but using different parameter data (i.e. filter coefficients) in each node. All such nodes execute the same code, albeit with distinct, node-specific parameter data. More generally, in decision trees, each node may implement a similar computation, again with different node-specific parameters, that executes on the fraction of the inputs directed to that node. We say that nodes executing the same computation are instances of the same *module type* ("module" for short).

MERCATOR concurrently executes multiple nodes with the same module type. User-supplied code is associated with a module, rather than with the individual nodes of that module type. The scheduler fires not individual nodes but rather entire modules. When a module is fired, the scheduler determines for *each* node of that module type how many inputs may safely be consumed from its queue, using the per-node constraints described above. Inputs are then gathered into ensembles from the input queues for *all* instances of the module, processed concurrently, and finally scattered to the downstream queues appropriate to each instance.

Concurrently executing nodes of the same type may benefit performance. First, the overhead of multiple node firings may be reduced by processing all nodes of a given type in a single firing. Second, if multiple nodes of a given type each have a limited number of queued inputs – perhaps even less than a full ensemble width – concurrent execution could pack these inputs together into ensembles, maximizing thread occupancy.

Runtime Support for Concurrent Execution. To support processing items from multiple nodes' queues in a single ensemble, each item is tagged with a small integer indicating its source queue. Within a module's code, any calls by the developer to MERCATOR's runtime (e.g. to access per-node parameters) pass in these tags to ensure the correct node-specific behavior for each item. Outputs to a channel are pushed to a module-wide output buffer along with their tags to indicate which downstream queue should receive them. Items in the output buffer are scattered to their queues using a mapping from tag to downstream queue computed per channel at compile time and stored in shared memory. Fig. 2 illustrates this process.

Efficient SIMD-parallel scatter and gather across multiple queues is challenging because of the complex index computations required. MERCATOR minimizes the cost of these operations using tuned implementations that exploit parallel primitives such as branch-free binary search [35] for gathering and a customized implementation of segmented parallel scan for scattering. Use of limited shared memory resources for queue index computation is minimized by doing as much work as possible within single *warps* – groups of 32 GPU threads that can transfer data among themselves using register shuffle operations. These computations require less than two bytes of shared memory per thread in a block and almost no shared memory for the special case of a module with a single instance. On our NVIDIA GTX 980Ti GPU, they take only a few thousand cycles per firing, even for a block of 1024 threads and 32 instances of a module.

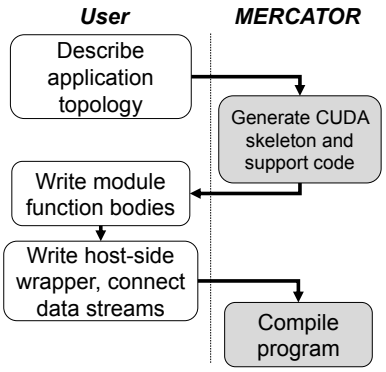

Fig. 3: Workflow for writing a program incorporating a MERCATOR application. MERCATOR generates skeleton and support code from the user's topology and provides APIs called from module bodies and from the host.

V. System Implementation

A. Developer Interface

To develop an application in MERCATOR, a programmer must (1) specify the application topology, (2) provide CUDA code for each module type, and (3) embed the application into a program on the host system. Fig. 3 illustrates the workflow associated with these tasks. Details of the developer interface and MERCATOR API may be found in our system's user manual [36].

Application topology is specified declaratively, as illustrated in Listing 1. The most important declaration, `module`, specifies a module type. Each named module type has a *signature* that specifies its input type and thread width, the types of each of its output channels, and its filtering behavior, in particular the maximum number of outputs anticipated for each input. Data types may be arbitrary C++ base types or structs. The spec then declares each node of the application as an instance of its module type, and finally declares the edges connecting nodes. Special module types are defined for the source node and sink node(s) of an application, whose implementations are supplied by MERCATOR. Nodes may have associated parameter data that is initialized on the host and made available to the GPU at runtime.

MERCATOR converts the developer's spec into a CUDA skeleton for the application along with any code needed to

```
//Filter module type
#pragma mtr module Filter (int[128]->acc<int>:?1)

//Node declarations
#pragma mtr node sourceNode : SOURCE
#pragma mtr node FilterNode : Filter
#pragma mtr node sinkNode : SINK<int>

//Edge declarations
#pragma mtr edge sourceNode::outStream->FilterNode
#pragma mtr edge FilterNode::acc->sinkNode
```

Listing 1: Specification for a simple linear pipeline. One filter node sits between source and sink and processes up to 128 input ints per firing, emitting 0 or 1 ints downstream via an output channel named `acc`. `outStream` is the default output channel name for a source node.

support scheduling and data movement between nodes. The skeleton provides a device function stub for each module type other than a source or sink. The developer then fills in the body of each stub with CUDA code for its computation. If a module type describes multiple nodes in an application, the stub is called with both an input item and a small integer tag in each thread, with the tag indicating the node that is processing the item. The module body calls a MERCATOR-supplied `push` function to emit output on a given channel, passing each output item along with its tag to ensure that it is routed to the correct downstream node. Different nodes of the same module type may access node-specific data using arrays indexed by tag.

Finally, the developer must instantiate the application on the host side. An instance of the application appears to the host as an opaque C++ class, which the developer connects to host-side input and output buffers. Running the application copies its inputs to the GPU, launches its uberkernel on all GPU multiprocessors to process those inputs, and finally copies any outputs back to the host.

B. Infrastructure

MERCATOR's core data structures and runtime system are implemented as a hierarchy of C++/CUDA classes. Module type, module instance, and queue objects are defined by base classes with member functions that delineate the semantics of data movement, scheduling, and module firing in the system. Inherited versions of these classes parameterized by data type implement proper queue storage and appropriately typed connection logic between objects.

A C++ front end incorporating the ROSE compiler infrastructure [37] first parses the developer's app spec file and extracts the parameters for each module type, node, and queue from the `module`, `node`, and `edge` declarations respectively. The front end then infers the application topology and checks for type compatibility and conformance to MERCATOR's topological restrictions. Finally, a codegen engine produces developer-facing CUDA function stubs for each module, along with code to create the necessary application objects as members of the MERCATOR infrastructure classes. The data types and other properties of modules specified in the app spec determine the signatures of the stub functions and application objects.

To form a working application, the developer compiles together the application skeleton with user-supplied function bodies for each module type, the system-supplied runtime supporting code (including host-GPU communication code and the module scheduler), and the host-side instantiation code using the regular CUDA toolchain.

VI. Design of Performance Experiments

We measured the performance of MERCATOR applications running on an NVIDIA GTX 980Ti GPU under CUDA 8.0. All applications were launched on the GPU using 176 blocks with 128 threads per block, thereby allocating 32 active warps to each multiprocessor. We measured the time to process a stream of inputs by book-ending the GPU kernel invocation with calls to the CUDA event API.

We implemented two types of application: a suite of small synthetic apps designed to measure the overhead and performance impact of MERCATOR's runtime support, and a more complex application implementing a pipeline for DNA sequence comparison.

A. Synthetic Applications

We implemented two sets of synthetic applications, shown in Figs. 4 and 5. The first set measured the impact of MERCATOR on single pipelines, while the second focused on tree-like application topologies composed of four copies of a pipeline fed from a common source node.

Each synthetic application included a source and one or more sinks, plus a collection of intermediate "working nodes." Each working node performed a configurable amount of compute-bound work on behalf of each of its input items and passed on a configurable fraction of those items to the next node downstream, discarding the remainder. The work performed at a node was computation of financial pricing options according to the Black-Scholes algorithm [38]; work per item was varied by controlling the number of Black-Scholes iterations performed.

Synthetic applications were tested on a stream of 10^6 items, each 48 bytes in size. Each item included state used by the Black-Scholes computation, plus a randomly chosen integer identifier. The average filtering rate of each node was controlled by passing only those items with identifiers in a certain range downstream.

Four single-pipeline designs were tested. `DiffType` consisted of five working nodes, each with the same code but declared with different module types to prevent concurrent execution. The `SameType` design was similar to `DiffType`, except that all nodes were declared to have the same module type to permit concurrent execution. In the `SelfLoop` design, the five working nodes were replaced by a single node with a self-loop. Each item carried a counter to ensure that it was processed at most five times. Filtering behavior in each pass around the loop was as for the linear pipeline. Finally, the `Merged` design concatenated the computations in the

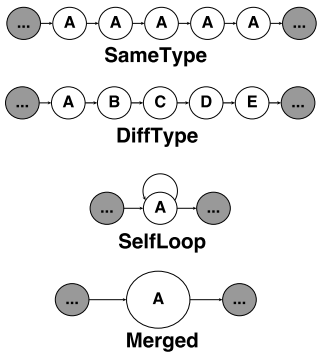

Fig. 4: Topologies of single-pipeline synthetic applications. Label inside each node indicates its module type. Gray nodes are sources/sinks.

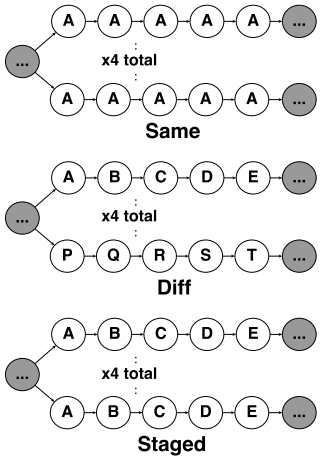

Fig. 5: Topologies of multi-pipeline synthetic applications. Node labels are as in Fig. 4.

five working nodes into a single "supernode." No queueing was performed between compute stages within the supernode. Hence, if an input was filtered early in the pipeline, its thread remained idle for the remainder of that call to the supernode. `Merged` represents a "control" against which the impact of MERCATOR's queueing infrastructure can be compared, while the three alternate designs exercise different subsets of MERCATOR's features.

We tested three multi-pipeline designs to further explore the impact of MERCATOR's concurrent execution of nodes with the same module type. The `Same` and `Diff` multi-pipeline designs again differed in that the former permitted concurrency across all working nodes, while the latter forbid any concurrency. The last design, `Staged`, allowed concurrency across nodes at the same stage in different pipelines, but not across different stages of the same pipeline.

B. BLASTN Sequence Comparison Pipeline

Our "BLASTN" application implements the ungapped kernel of the NCBI BLAST algorithm for sequence comparison [6]. The application compares a DNA database to a fixed DNA *query sequence* to find regions of similarity between the two. The query is stored in GPU global memory, along with a hash table containing its substrings of some fixed length k ($k = 8$ in our implementation, matching the behavior of NCBI BLASTN). BLASTN executes in a pipeline of four stages, each of which we mapped to one application node. First, each length-k substring of the database is compared to the hash table. Second, if the substring occurs in the table, each of its occurrences (up to 16 per database position) is enumerated. Third, each occurrence is extended with additional matches to the left and right to see if it is part of a matching substring of length at least 11. Fourth, matches of length 11 (up to 16 per database position) are further extended via dynamic programming over a fixed-size window. If the resulting inexact match's score exceeds a threshold, it is returned to the host.

Each stage of this pipeline discards a large fraction of its input. We note that the pipeline was implemented in MERCATOR by two undergraduate students with no prior GPU or bioinformatics experience.

We tested BLASTN by comparing a database consisting of human chromosome 1, comprising 225 million DNA bases, to queries drawn from the chicken genome. Query sizes varied between 2,000 and 10,000 bases. To measure the impact of MERCATOR's remapping optimizations, we compared our implementation to one in which all pipeline stages were merged into a single node, analogous to the `Merged` synthetic application. In the merged implementation, threads whose input was filtered out early in the pipeline remained idle until all remaining threads in the current ensemble finished.

VII. Results and Discussion

A. Core Functionality: Benefit vs. Overhead

The MERCATOR framework seeks to improve thread occupancy of applications by performing dynamic work-to-thread remapping at node boundaries. To be worthwhile, the performance boost from increased occupancy must outweigh the overhead due to queueing, remapping, and scheduling. We measured the tradeoff between increased occupancy and overhead by comparing the running times of the `Merged` and `DiffType` topologies across filtering rates and workloads, as shown in Fig. 6.

We observe that `DiffType` outperforms `Merged` (speedup > 1) for most cases tested, indicating that the overhead of remapping in MERCATOR was less than the performance improvement due to elimination of idle threads. Filtering rates between the two extremes of 0 (no stage discards any inputs) and 1 (the first stage discards all inputs) favored `DiffType`, with rates of 0.5 and 0.75 showing over 1.5x speedup for sufficient workloads. Only when irregularity is entirely absent from the application (rate 0) or when all work occurs in the first node (rate 1) is the lower-overhead `Merged` implementation consistently superior.

Moreover, the performance advantage of `DiffType` over `Merged` increases with increasing workload at each pipeline stage, since more work per item in later stages causes idle threads in the `Merged` implementation to remain idle for longer. For all cases tested, the benefit of remapping exceeded

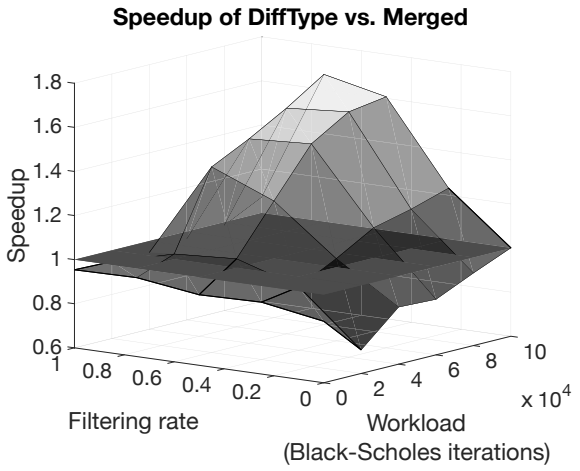

Fig. 6: Speedup of `DiffType` over `Merged` topology for varying filtering rates and workloads. Filtering rates shown were applied to each stage of the application; rate 0 discards no inputs, while rate 1 discards all inputs. Wherever the speedup is greater than 1 (i.e. above the plane), the benefits of MERCATOR outweigh its costs for this application.

its overhead for workloads requiring at least 250ms of execution time per node firing. In some cases, the threshold to see a benefit from MERCATOR was much lower (e.g. < 75ms for a filtering rate of 0.5).

To ensure that the functionality of `SelfLoop` does not incur an unacceptable cost, we compared its performance to that of `SameType` and `DiffType`. The comparison with `SameType` is appropriate since a node with a self-loop may always be unrolled into multiple distinct nodes of the same module type. Across all our experiments, the execution time of `SelfLoop` was lower than that of `DiffType` by an average of 3.6% and was very close to that of `SameType` (average of 1.3% faster). Hence, support for loops does not appear to incur significant overhead in MERCATOR and may be advantageous for implementing applications that require a variable number of identical execution rounds, such as fixed-point [4] and recursive [31] algorithms.

B. Concurrent Execution of Nodes with Same Module Type

Our first test of MERCATOR's ability to concurrently execute nodes of the same type is the single-pipeline `SameType` application. `SameType`, which concurrently executes all its working nodes, achieves a modest speedup of $1.05\times$ compared to `DiffType`, which must fire each node separately.

The replicated-pipeline applications make more nodes available for concurrent firing and offer opportunities for both horizontal and vertical concurrency. Fig. 7 compares the execution time of these three applications under varying per-item workloads using a uniform per-node filtering rate of 0.5.

Concurrently executing nodes of the same module type in these topologies resulted in speedups of 1.23 to $1.26\times$, depending on per-node workload, over a baseline, `Diff`, in which each node's inputs were executed separately. Scheduling overhead in all experiments was less than 2%, so the difference

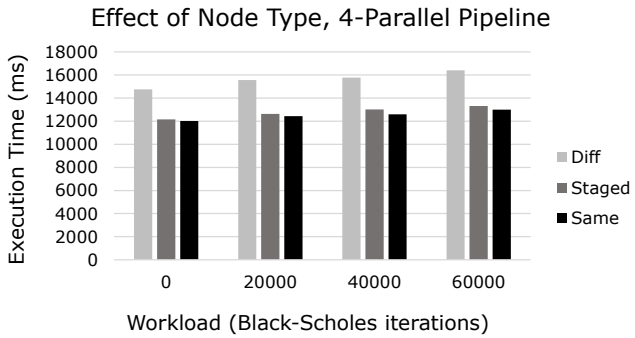

Fig. 7: Execution time of 10^6 inputs streamed through the parallel pipeline applications in Fig. 5 under various workloads with a per-node filtering rate of 0.5.

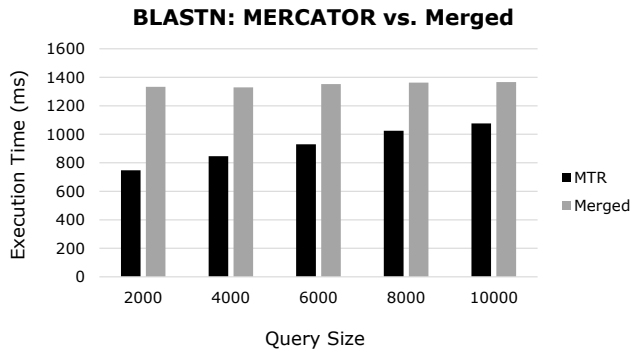

Fig. 8: Execution time of BLASTN application comparing a chicken genome fragment (query) against human chromosome 1 (database) using MERCATOR vs. the merged topology. Query size is measured in bases.

in performance is likely attributable to improved occupancy due to concurrency, rather than merely a change in the number of distinct modules to be scheduled. The `Staged` application was comparable in execution time to the `Same` application to within 3%, suggesting that horizontal concurrency across pipelines (as in `Staged`), rather than vertical concurrency within each pipeline, conferred most of the benefit.

We conclude that concurrent execution can indeed confer performance benefits in executing MERCATOR applications.

C. Behavior of BLASTN

We conclude by analyzing the performance of our MERCATOR BLASTN implementation vs. a Merged implementation that does not implement remapping between stages. Fig. 8 shows the execution time of these implementations for biologically relevant input query sizes from 2,000 to 10,000 DNA bases. MERCATOR achieved speedups ranging from $1.27\times$ to $1.78\times$ over the merged implementation.

Whereas MERCATOR's execution time improves with decreasing query size, that of the merged version is almost invariant to input set size and always exceeds MERCATOR's worst time. Smaller queries decrease the rate at which 8-mers in the database hit in the query hash table, as well as the number of hits per 8-mer. As this rate decreases, MERCATOR's remapping optimizations maintain full thread occupancy, while

the merged implementation suffers the overhead of idle threads during extension and dynamic programming.

VIII. Conclusion and Future Work

We have described MERCATOR, a platform for implementing irregular streaming dataflow application development on GPUs. MERCATOR supports efficient remapping of work to SIMD threads within an application. Once an application has been divided into nodes with internally regular computations, this support is provided between nodes transparently to the application developer. Both in our synthetic benchmarks and for the BLASTN application, MERCATOR's remapping support offered benefits for overall throughput that substantially exceeded its runtime overhead. MERCATOR's ability to specify and support a large set of practically important application topologies, together with the efficiency of its remapping primitives, offer robust remapping support independent of a particular application domain.

Improvements to MERCATOR Infrastructure. We foresee several opportunities to improve MERCATOR's runtime and remapping support. First, because remapping is transparent to the application developer, we are free to optimize the underlying remapping primitives. For example, managing queues through atomic updates may sometimes be more efficient than our current scanning and compaction approach. Moreover, for certain combinations of node computational cost and filtering rate, it may be advantageous to eliminate remapping between two nodes altogether, instead paying the cost of lower thread occupancy to eliminate remapping overhead. These decisions should be automated and should be guided by profiling of runtime filtering behavior.

Second, it is straightforward to extend MERCATOR's per-module stubs to provide more than one input to each thread, or to dedicate several threads to one input. Multiple inputs per thread offers the possibility of memory latency hiding through loop unrolling, while multiple threads per input allows for parallelization in the processing of each item. We plan to explore how to most efficiently support different thread-to-item ratios and what performance benefits may accrue from varying these ratios.

Third, MERCATOR uses a simple scheduling heuristic that fires the module with the most queued inputs, subject to limitations imposed by the available queue space of the module's downstream nodes. We plan to develop more sophisticated scheduling strategies that provably maximize throughput by, e.g., minimizing the extent to which any one node in an application becomes a bottleneck.

Extension to Other Irregular Streaming Computations. To extend MERCATOR to other high-impact domains, we plan to support two important classes of computations: those with nodes requiring simultaneous inputs from multiple upstream nodes in order to fire (which are supported by "join" semantics in SDF applications), and graph-processing applications.

Join semantics in the presence of filtering are not straightforward, because input items may be discarded on some but not all paths prior to reaching the join point. While consistent behavior can be defined in such cases [39], a large number of practical join-containing applications exhibit static data rates and so fit within the simpler SDF framework. Examples include JPEG compression, MP3 decoding, and AES encryption. We will initially extend MERCATOR to support application topologies in which certain subgraphs are free of irregularity but do contain joins. Each such subgraph can be analyzed and scheduled as a unit relative to the rest of the application.

Graph-processing applications exhibit behaviors beyond the current limits of MERCATOR's streaming execution model, yet they are strong candidates for optimization of irregular data flow. Many graph applications, including ones found in the Pannotia benchmark suite [9] and the LonestarGPU suite [4], process graphs in a series of vertex-centric rounds converging to a fixed point. In one round, each vertex does some computational work on each of its incident edges, reduces the results associated with these edges, and finally computes on the result of this reduction. Each vertex may have a different degree and so may require a different amount of work in a round.

To support efficient SIMD processing of graph computations, we must schedule the per-edge work for edges of many vertices simultaneously. "Exploding" a stream of vertices into a stream of all their edges would regularize much of a computation's degree-dependent irregularity. MERCATOR could even model per-edge computations in which only a dynamically determined subset of edges contribute to the result for a vertex. However, the stream of per-edge results must be *reduced* to a single value per vertex at the end of a round.

The principal extension needed to support graph computation in MERCATOR is a *reduce-by-key* operation analogous to MapReduce [40]. In this case, the key is the vertex associated with each adjacent edge. MERCATOR already implements efficient SIMD reduce-by-key algorithms to support concurrent processing of inputs to multiple nodes within a single module, as described in Section IV-B. However, the temporal scope of a reduction is much smaller than the entire computation – a vertex must be queued for further processing as soon as all of its edges have been processed and their results reduced. To express the limited scope of each reduction, we plan to define explosion/reduction operation pairs that span a limited subgraph of the application. Within this subgraph, we process individual edges; at its boundary, the edges are gathered back into their vertices. We previously prototyped this style of explosion/reduction in our WOODSTOCC DNA sequence aligner [26]. The challenge is to generalize these operations while preserving the generality and transparency of MERCATOR's support for streaming, irregular dataflow.

Acknowledgment

Kim Orlando (St. Mary's College) and Stephen Timcheck (University of Akron) implemented our BLASTN application.

References

[1] C. P. Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: a survey on big data," *Information Sciences*, vol. 275, pp. 314–347, 2014.

[2] NVIDIA, "GPU impact by domain," https://www.nvidia.com/object/gpu-applications-domain.html, 2016, accessed 2016-11-22.

[3] T. Zhang, G. Chen, W. Shu, and M.-Y. Wu, "Microarchitectural Characterization of Irregular Applications on GPGPUs," *SIGMETRICS Perform. Eval. Rev.*, vol. 42, no. 2, pp. 27–29, Sep. 2014.

[4] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in *IEEE Int'l Symp. Workload Characterization*, 2012, pp. 141–151.

[5] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem *et al.*, "The Tao of Parallelism in Algorithms," in *Proc. 32nd ACM SIGPLAN Conf. Programming Language Design and Implementation*, New York, NY, USA, 2011, pp. 12–25.

[6] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *J. Molecular Biology*, vol. 215, no. 3, pp. 403–410, 1990.

[7] B. Ren, G. Agrawal, J. R. Larus, T. Mytkowicz, T. Poutanen, and W. Schulte, "SIMD parallelization of applications that traverse irregular data structures," in *IEEE Int'l Symp. Code Generation and Optimization*, 2013, pp. 1–10.

[8] E. J. Tyson, J. Buckley, M. A. Franklin, and R. D. Chamberlain, "Acceleration of atmospheric Cherenkov telescope signal processing to real-time speed with the Auto-Pipe design system," *Nuclear Instruments and Methods in Physics Research Sec. A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 595, no. 2, pp. 474–479, 2008.

[9] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular gpgpu graph applications," *2013 IEEE Int'l Symp. Workload Characterization*, pp. 185–195, 2014.

[10] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, 2014.

[11] G. Kahn, "The semantics of a simple language for parallel programming," in *Proc. Int'l Fed. Information Processing Cong.*, vol. 74, 1974, pp. 471–475.

[12] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proc. IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[13] E. A. Lee and D. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Computers*, vol. 100, no. 1, pp. 24–35, 1987.

[14] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil, "Software pipelined execution of stream programs on GPUs," in *Proc. Int'l Symp. Code Generation and Optimization*, 2009, pp. 200–209.

[15] A. Hagiescu, H. P. Huynh, W.-F. Wong, and R. S. Goh, "Automated architecture-aware mapping of streaming applications onto GPUs," in *IEEE Int'l Parallel & Distributed Processing Symp.*, 2011, pp. 467–478.

[16] J. Buck and E. A. Lee, "The token flow model," in *Data Flow Workshop*, 1992, pp. 267–290.

[17] T. M. Parks, "Bounded scheduling of process networks," Ph.D. dissertation, University of California. Berkeley, California, 1995.

[18] S. Stuijk, M. Geilen, B. Theelen, and T. Basten, "Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications," in *Int'l Conf. Embedded Computer Systems (SAMOS)*. IEEE, 2011, pp. 404–411.

[19] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig *et al.*, "Taming heterogeneity-the Ptolemy approach," *Proc. IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[20] M. E. Belviranli, C.-H. Chou, L. N. Bhuyan, and R. Gupta, "A Paradigm Shift in GP-GPU Computing: Task Based Execution of Applications with Dynamic Data Dependencies," in *Proc. 6th Int'l Wkshp. Data Intensive Distributed Computing*, 2014, pp. 29–34.

[21] S. Tzeng, A. Patney, and J. D. Owens, "Task management for irregular-parallel workloads on the GPU," in *Proc. Conf. High Performance Graphics*, 2010, pp. 29–37.

[22] S. Tzeng, B. Lloyd, and J. D. Owens, "A GPU Task-Parallel Model with Dependency Resolution," *IEEE Computer*, vol. 45, no. 8, pp. 34–41, 2012.

[23] M. S. Orr, B. M. Beckmann, S. K. Reinhardt, and D. A. Wood, "Fine-grain task aggregation and coordination on GPUs," in *Proc. 41st Ann. Int'l Symp. Computer Architecuture*, 2014, pp. 181–192.

[24] B. R. Gaster and L. Howes, "Can GPGPU programming be liberated from the data-parallel bottleneck?" *IEEE Computer*, vol. 45, pp. 42–52, 2012.

[25] OpenACC, "OpenACC 2.0a Specification," http://www.openacc.org/sites/default/files/OpenACC%202%200.pdf, Aug. 2013, accessed 2016-11-22.

[26] S. V. Cole, J. R. Gardner, and J. Buhler, "WOODSTOCC: Extracting latent parallelism from a DNA sequence aligner on a GPU," in *Proc. 13th IEEE Int'l Symp. Parallel & Distributed Computing*, 2014.

[27] S. V. Cole, J. R. Gardner, and J. D. Buhler, "WOODSTOCC: Extracting Latent Parallelism from a DNA Sequence Aligner on a GPU," *Washington University Tech Report WUCSE-2015-004*, Sep 2015.

[28] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable SIMD-efficient graph processing on GPUs," in *Proc. 2015 Int'l Conf. Parallel Architecture and Compilation*, Washington, DC, USA, 2015, pp. 39–50.

[29] N. Singla, M. Hall, B. Shands, and R. D. Chamberlain, "Financial monte carlo simulation on architecturally diverse systems," in *Wkshp. High Performance Computational Finance*. IEEE, 2008, pp. 1–7.

[30] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Proc. IEEE Comp. Soc. Conf. Computer Vision and Pattern Recognition*, 2001.

[31] B. Ren, Y. Jo, S. Krishnamoorthy, K. Agrawal, and M. Kulkarni, "Efficient execution of recursive programs on commodity vector hardware," in *ACM SIGPLAN Notices*, vol. 50, no. 6. ACM, 2015, pp. 509–520.

[32] G. Marsaglia, W. W. Tsang *et al.*, "The ziggurat method for generating random variables," *J. Statistical Software*, vol. 5, no. 8, pp. 1–7, 2000.

[33] O. Krzikalla, F. Wende, and M. Höhnerbach, "Dynamic SIMD vector lane scheduling," in *Int'l Conf. High Performance Computing*. Springer, 2016, pp. 354–365.

[34] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct. 2001.

[35] I. Buck and T. Purcell, "A toolkit for computation on GPUs," in *GPU Gems*. Addison-Wesley, 2004.

[36] S. V. Cole and J. D. Buhler, "Mercator user's manual," http://sbs.wustl.edu/pubs/MercatorManual.pdf, 2016.

[37] D. Quinlan and C. Liao, "The ROSE source-to-source compiler infrastructure," in *Cetus Users and Compiler Infrastructure Wkshp., in conjunction with PACT 2011*, Oct. 2011.

[38] NVIDIA, "NVIDIA Compute Unified Device Architecture (CUDA) Code Samples," https://developer.nvidia.com/cuda-code-samples, Sep. 2014, accessed March 24th, 2017.

[39] J. D. Buhler, K. Agrawal, P. Li, and R. D. Chamberlain, "Efficient deadlock avoidance for streaming computation with filtering," in *Proc. 17th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, 2012, pp. 235–246.

[40] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. 6th Symp. Operating Systems Design and Implementation*, 2004, pp. 137–150.