Summer 8-15-2017

Efficiently and Transparently Maintaining High SIMD Occupancy in the Presence of Wavefront Irregularity

Stephen V. Cole
Washington University in St. Louis

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering and Applied Science
Department of Computer Science and Engineering

Efficiently and Transparently Maintaining High SIMD Occupancy in the Presence of Wavefront Irregularity

by
Stephen V. Cole

A dissertation presented to
The Graduate School
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

August 2017
Saint Louis, Missouri

Table of Contents

List of Figures

List of Tables

Acknowledgments

I owe a debt of gratitude to many persons for making this work possible through their investment in my personal and professional formation. I will try to acknowledge the most relevant sources of this formation here, going roughly in chronological order.

First in influence as well as time, I am grateful to God: not only for creating me and orchestrating circumstances of my life beyond my control to provide me every opportunity and capacity for education imaginable, but for being a faithful friend throughout my dissertation work process. We kept vigil together on many a late night, which helped me appreciate His constancy as He "never slumbers or sleeps (Psalm 121:4)" while constantly watching over us.

I'm grateful to my parents for modeling the value of hard work while maintaining a sense of humor, the joy of natural curiosity, the satisfaction of acquiring knowledge for its own sake, and the integrity of working for personal standards based on objective reality rather than common beliefs about what is "good enough." I'm grateful to them for *not* modeling math-o-phobia, devotion to career over family, or attachment to things over people. My siblings Jean, Tim, Mary, Caroline, Matt, and Laura have also been extremely supportive of me throughout my life and certainly throughout my grad school work.

Many thanks and boomba heys to Frank Corley for teaching an outstanding class on infinity at Missouri Scholars Academy (MSA) in 2000 that planted the seeds of university culture in me by twisting my mind up in all sorts of stimulating ways that wouldn't happen again for several years, and doing it within a group of people who all enjoyed it as much as I did.

I'm grateful to my professors and fellow students at Benedictine College for a college experience that made teaching at the college level seem attractive. Sr. Linda Herndon, O.S.B., taught me almost everything I learned about CS as an undergrad and always ran fair and well-structured courses while still leaving a little space for our small-classroom antics. Matt Hoeing, Brett Hermann, Andy Gardner, Rosemary Kasten, Chad Johnson, Kristen (Mersinger) Schmitt, Nathan Engert, Mike Carrington, and the rest of the gang provided good late-night lab fellowship. I owe a special shout-out to Matt Hoeing for convincing me to add CS as a second major, which obviously led to bigger things.

I'm grateful to the NSF REU program at the U. of South Carolina, especially John Bowles, Caroline Eastman, Michael Huhns, and Matthew Royal, for my first taste of authentic computer science research.

I'm grateful for my time at the University of California at Riverside, where I learned what grad school is all about and enjoyed it enough to pursue a Ph.D. afterwards. My adviser Neal Young showed me that it's possible to be both a tenacious researcher and a devoted teacher, that it's good to have hobbies outside of work, and that openness and generosity are the best foundation for collaboration both on and off campus. I couldn't have asked for better friends both in and out of the department to enjoy those two years with: Dino Buenviaje was a faithful and supportive friend from Day 1; Edward Fernandez and Casey Czechowski were the first people I met in the CS department and always kept me looking on the bright side of life; Monik Khare was always up for doing something new and challenging as a frequent class-project partner, roommate, and academic sibling; Peter Lonjers challenged my thinking from the ground up in many ways; Nathan Diep modeled the integration of faith and professional life; Viabhov Parulkar, Akshay Morye, Pradyumna Goli, and many others made Friday BBQs well worth the effort; Trini, Jelena, Lorenzo, Diana, Fr. George, and the rest of the Grad Group at the Newman Center provided some perspective on grad school through the eyes of other departments besides CS.

When I decided to pursue a Ph.D., I chose Washington University in part because of its personal, collaborative atmosphere, and I am grateful to have earned my degree in a department that has that kind of culture. Many thanks to my lab group predecessors Arpith Jacob, Joe Lancaster, Michael Hall, Peng Li, Lin Ma, and Jonathan Beard for their support and confidence that I would soon follow in their successful footsteps; to David Lu!!, Zeke Maier, and Justin Wilson for proving that perseverance pays off; to Dan Lazewatsky, Austin Abrams, Kylia Miskell, David Ferry, and the rest of the GSA crew for a good couple of years of working on something productive together besides research; to Dan, David, and James Orr for keeping me sane during the final push with our weekly lunches; to Anthony Cabrera, Clayton Faber, Michael, Peng, Lin, and the rest of the lab meeting group over the years for teaching me interesting things every week and providing extremely helpful feedback on my own research; to Jake Gardner, Stephen Timcheck, Kim Orlando, Edgar Fernandez, and Theron Howe for enjoyable as well as productive summers of REU research; and to many, many other students for good happy hour conversations and friendly hallway hellos. The CS department staff (Myrna Harbison, Kelli Eckman, Sharon Matlock, Monét Demming, Cheryl Newman, Jayme Moehle, Lauren Huffman, Madeline Hawkins) have not only made the logistics of degree progress and student life go smoothly by keeping us above the surface of the iceberg of paperwork and scheduling involved with those things, but have always done so cheerfully and made us feel at home in the department. In my off-campus time, my friends at South Side Youth Ministry, my Emmaus Group, my roommates, and my morning-coffee team of Tim Hunn and Adam Reinagel have been instrumental in my personal formation and valuable reminders of life outside of grad school.

Many thanks to my committee for their time and diligence in guiding my project through the dissertation proposal and defense. Special thanks to Roger Chamberlain for being an almost-co-adviser for several years and for always being willing to quickly distill for me the vast literature on scheduling, performance modeling, or whatever else I needed to understand about distributed processing.

Most importantly for this work, of course, I am grateful to Jeremy for guiding me through my professional formation as my adviser. Jeremy leads first by example, and he is the kind of researcher I would aspire to be: motivated by curiosity, attentive to detail, articulate in writing, concise in coding, and devoted to his family as well as his work. I hope that some of these qualities have rubbed off on me as I now become qualified to mentor others.

I met my wonderful wife Jeannine near the end of my journey at Wash U., and when she married me she knew she was inheriting a sizeable debt of time to my project. She has been exceptionally supportive and encouraging as she has helped me pay this debt by sacrificing evenings and weekends of time together, and our relationship has been a great reminder of what's worth working hard for.

Finally, I am grateful to my research benefactors for funding this work. Woodstocc was developed under National Science Foundation grant CNS-0905368, and Mercator was developed under NSF CISE-1500173 and support from Exegy, Inc.

Stephen V. Cole

Washington University in Saint Louis

August 2017

ABSTRACT OF THE DISSERTATION

Efficiently and Transparently Maintaining High SIMD Occupancy in the Presence of Wavefront Irregularity

by

Stephen V. Cole

Doctor of Philosophy in Computer Science

Washington University in St. Louis, 2017

Professor Jeremy Buhler, Chair

Demand is increasing for high throughput processing of irregular streaming applications; examples of such applications from scientific and engineering domains include biological sequence alignment, network packet filtering, automated face detection, and big graph algorithms. With wide SIMD, lightweight threads, and low-cost thread-context switching, wide-SIMD architectures such as GPUs allow considerable flexibility in the way application work is assigned to threads. However, irregular applications are challenging to map efficiently onto wide SIMD because data-dependent filtering or replication of items creates an unpredictable *data wavefront* of items ready for further processing. Straightforward implementations of irregular applications on a wide-SIMD architecture are prone to load imbalance and reduced occupancy, while more sophisticated implementations require advanced use of parallel GPU operations to redistribute work efficiently among threads.

This dissertation will present strategies for addressing the performance challenges of wavefront-irregular applications on wide-SIMD architectures. These strategies are embodied in a developer framework called Mercator that (1) allows developers to map irregular applications onto GPUs according to the *streaming paradigm* while abstracting from low-level data movement and (2) includes *generalized* techniques for *transparently* overcoming the obstacles to high throughput presented by

wavefront-irregular applications on a GPU. MERCATOR forms the centerpiece of this dissertation, and we present its motivation, performance model, implementation, and extensions in this work.

Chapter 1

Background and Context

1.1 Motivation

With the exponential increase of available data across all disciplines in the last two decades has come a corresponding need to process that data, leading to increasing demand for high-throughput architectures and algorithms. Over the same time frame, Graphics Processing Units (GPUs) and other wide-SIMD architectures have grown in popularity for large-scale general-purpose data processing. Although many of the applications requiring high data throughput have "regular" execution patterns, many do not, and these "irregular" applications are particularly detrimental to wide-SIMD performance. **This dissertation will present strategies for addressing the performance challenges of irregular applications on wide-SIMD architectures.**

Why high-throughput applications? The growth of dataset sizes across diverse domains such as genomics (see Figure 1.1), health care [100], retail [118], and social media (e.g. [50]) in recent years is widely known. In aggregate, the International Data Corporation estimates that the total amount of digital data created and copied annually has been doubling every two years since 2013 and will reach 44 zettabytes by 2020 [49]. This data is a rich target for analysis, yet current hardware and software is having a hard time keeping up: according to one estimate, only 0.5% of generated data is ever analyzed [101]. A great demand therefore exists for large-scale data processing.

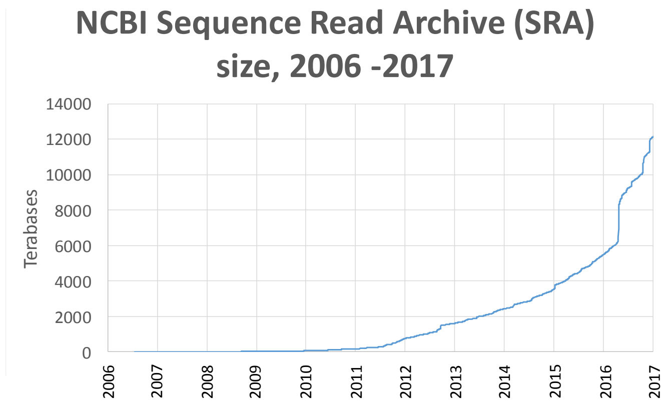
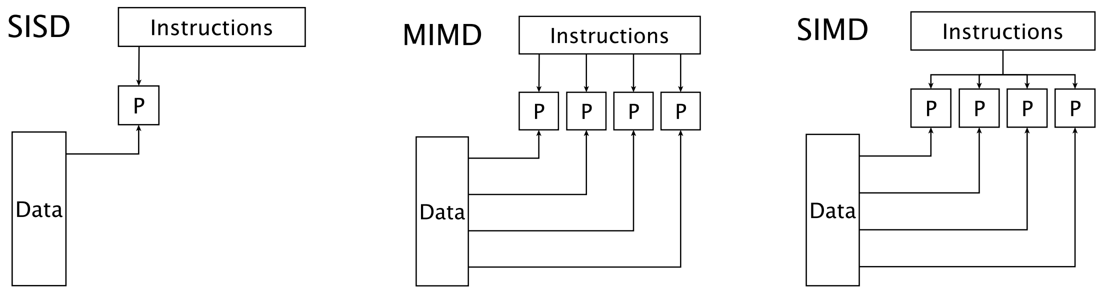


Figure 1.1: Number of DNA bases stored in the NCBI Sequence Read Archive (SRA) database [80], which is a common source of data for bioinformatics applications.

Large datasets may be processed with different optimization targets according to their usage. For example, an application such as a search engine that processes data in real time may optimize for single-input (i.e. single-query) latency, while an application such as retail data analysis that processes data offline may optimize for total throughput. In this dissertation we focus on applications in the latter category, which we refer to as *high-throughput* applications.



(a) Single Instruction, Single Data (SISD).

(b) Multiple Instruction, Multiple Data (MIMD).

(c) Single Instruction, Multiple Data (SIMD).

Figure 1.2: Common processor categories from Flynn's taxonomy, with "P" representing a processing unit and independent edges representing independent streams. SISD models standard von Neumann serial processing, MIMD models independent multicore processing, and SIMD models vector processing.

Why SIMD? The SIMD (Single Instruction, Multiple Data) category in Flynn's classic taxonomy of computer architectures [40], illustrated in Figure 1.2, describes systems that operate on multiple data inputs simultaneously using a single instruction stream. Since these inputs may be thought of as occupying a vector on which instructions operate, SIMD processing is also known as *vector processing* and specialized SIMD instructions are sometimes called *vector instructions.* [1] Each slot in the vector is known as a SIMD *lane*, and the number of lanes– i.e., the number of inputs that can be simultaneously processed– is the SIMD *width.*

This dissertation presents techniques that specifically target SIMD architectures for two reasons. First, SIMD execution is ideally suited to throughput-oriented applications with many independent inputs: at each execution step, each SIMD lane may contain a separate input, exploiting *data parallelism* to yield a theoretical speedup equal to the SIMD width over a serial processor.

Second, SIMD support has become nearly ubiquitous in processors and coprocessors, making our contributions widely applicable. Specialized SIMD-capable processors were originally incorporated into supercomputers made by companies such as Cray and Thinking Machines decades ago [6,107]. However, the rise in demand for throughput-oriented processing of data-parallel applications from domains such as computer graphics as well as advances in processor technology have led to both the widespread integration of SIMD capabilities into commodity chips and the development of wide-SIMD coprocessors. Intel processors and compilers have supported SIMD instructions since the introduction of MMX intrinsics in 1996 [96], and current-generation Intel chips support AVX intrinsics for SIMD widths of 256 bits and growing [71]. Hardware coprocessors designed to accelerate high-throughput applications natively support SIMD widths of 512 bits or more, either through vector instructions as in Intel's Many Integrated Core (MIC) architecture instantiated in the Xeon Phi card [29,51] or through a multithreaded lockstep-execution model as in GPUs [81]. With this growth in popularity, high-throughput processing using wide-SIMD execution is no longer restricted to the realm of supercomputing and SIMD-friendly approaches to data parallelism are widely applicable.

[1] Technically vector processing is one specific hardware implementation of SIMD, of which others such as array processing exist [89]. However, we follow popular convention in taking vector processing as the nominal representative of such implementations and use it interchangeably with SIMD.

Why GPUs? Out of all SIMD-capable architectures, we target GPUs (Graphics Processing Units) running NVIDIA's CUDA/C++ toolkit [86] as the implementation platform for the techniques given in this dissertation. We note, however, that since our contributions rely only on the wide SIMD capability of the target architecture rather than on GPU-specific characteristics, the techniques described here could be applied equally well to other popular wide-SIMD architectures such as AMD graphics cards running OpenCL [34] or Intel's Xeon Phi coprocessors [29].

GPUs were chosen as the implementation platform for this work because they have emerged as a quintessential example of a wide-SIMD architecture due to their support for *general-purpose, cost-efficient, developer-friendly, very wide* data-parallel execution.

GPUs provide support for *general-purpose* parallel programming. Graphics processing using resources that augment a main CPU gradually evolved over the course of decades until the entire 3D graphics pipeline was offloaded to a separate card called a Graphics Processing Unit (GPU) by NVIDIA in 1999 [30]. The first GPUs were optimized and hard-wired for highly parallel graphics operations such as pixel shading, ray tracing, and physics calculations, and they had a "fixed function" graphics pipeline that remained opaque to developers. In 2001 NVIDIA exposed to developers its new "programmable" graphics pipeline, though operations still needed to be framed in terms of traditional graphics pipeline stages such as vertex and pixel shading [111]. In 2007 NVIDIA introduced the CUDA C programming language, which allowed for General-Purpose GPU (GPGPU) programs to be written entirely abstracted from the graphics pipeline [7]. Since then GPGPU applications have proliferated, allowing the benefits of the GPU architecture to be applied to many domains outside of graphics [44]. For a summary of the genesis and development of GPUs see [30].

With their architectural support for *wide* SIMD-parallel execution, GPUs are an ideal candidate for high-throughput processing. GPUs can execute thousands of lightweight threads simultaneously, organized hierarchically into groups having a minimum SIMD width of 32 inputs [84].

With the economies of scale realized on their production for the gaming industry, GPUs provide a *cost-efficient* means of high-throughput processing. Although FPGAs consume significantly less power than GPUs due to the customizability of their circuits, the initial purchase cost of an FPGA currently outweighs the power savings even over a span of several years assuming FPGA dynamic power usage of 10W and GPU usage of 100W per application run [78]. GPU performance per watt also continues to increase with each generation of card [12].

As GPU development toolchains such as CUDA and OpenCL have matured, the barrier to entry for programming applications for GPUs has decreased sharply, making them more *developer-friendly*. Unlike FPGAs, which require development in a hardware description language for optimal performance, CUDA and OpenCL are based on C++ and therefore have syntax and semantics familiar to most application programmers.

GPUs' wide parallelism, low cost, and ease of programming have helped them quickly become a commodity solution for high-throughput processing. As noted at the GPU Technology Conference in 2015, the popularity of CUDA increased by at least a factor of ten between 2008 and 2015 across multiple metrics: toolkit downloads (to 3 million), universities teaching CUDA (to 800), academic papers citing CUDA (to 60,000), and Tesla GPGPU cards shipped (to 450,000) [52].

With their ability to easily integrate into existing systems as peripheral accelerators (GPUs connect to CPUs via PCI-e bus), GPUs also scale. Just as Google achieved massive search and storage throughput by scaling up cheap commodity hardware, many current supercomputing systems achieve their impressive throughput by integrating many GPUs into dedicated clusters rather than investing in additional expensive specialized supercomputers. In 2016, 64 of the supercomputers on the Top500 list used GPUs, with another 35 using other manycore accelerators such as the Xeon Phi [120]. GPUs are also now available as a standard hardware option for cloud processing services such as Amazon EC2 [4] and Google Cloud [43].

Two recent developments indicate that FPGAs may soon gain ground on GPUs in terms of economies of scale and development cost. First, the release of a system by Intel that combines an FPGA and a CPU [82], with an expected future release having both on the same die, will provide FPGAs with the manufacturing momentum of Intel. Second, hardware support and high-level synthesis tools for OpenCL that target FPGAs [131] allow programmers to write FPGA applications in OpenCL rather than hardware description languages. Although OpenCL is not the first software language to gain this support, its architecture-agnostic style enables its code to target both GPUs and FPGAs, thus allowing developers to test high-level code on both architectures with minimal intervention. The impact of these developments is unknown, but they may drive the further evolution of GPU computing by competitive necessity.

Since GPUs will be used as a representative wide-SIMD architecture throughout this dissertation, we now review the relevant aspects of a GPU's architecture and execution.

Processor/core structure: GPUs contain several (8-25) independent *streaming multiprocessors* (SMs). Fixed-sized groups (*warps*) of w threads are bound to a particular SM for scheduling and execution. An SM can be thought of as a w-lane SIMD execution unit, with threads corresponding to individual SIMD lanes. All current NVIDIA GPUs set $w = 32$. A single SM may dynamically context-switch among several active warps.

Execution: GPU code is exposed for execution as one or more *kernels* that are launched from CPU code. The division of work across SMs is specified at launch time in the form of a desired number of *blocks* of work and of compute threads (hence, warps) per block. The warps of a single block are executed by a single SM for the duration of the kernel.

Execution constraints: To distill the salient features of GPU execution behavior and maintain generalization to wide-SIMD platforms, we will constrain an application's mapping to the GPU in the following ways:

- Each application runs within a single kernel call, keeping the GPU block configuration and the assignment of blocks to SMs fixed throughout execution.
- Every thread within a GPU block executes the same code.
- GPU blocks operate on disjoint sets of input items and use independent working memory sets. Hence, no synchronization among blocks is required.

Under these constraints, each block may be viewed as an independent instance of the full application running on its own input data set.

What is an *irregular* application? GPUs have traditionally been best at processing applications with regular execution and memory patterns since their architecture was designed specifically to exploit these characteristics in graphics applications. NVIDIA provides CUDA libraries with highly optimized implementations of these applications from areas such as linear algebra, signal processing, FFT calculation, and random number generation [85]. NVIDIA also provides libraries for newer high-impact applications that require large-scale repetitive computation such as deep neural network training, and its forthcoming Volta-architecture cards are specifically designed to provide high throughput for very regular common machine learning calculations such as strings of fused multiply-adds [87].

However, not all high-impact applications have a regular structure that is naturally amenable to acceleration on a GPU. Many GPU applications exhibit irregularity in various ways: divergent control flow, unbalanced workloads, unpredictable memory accesses, input-dependent dataflow, complicated data structure requirements [20, 97, 134]. Although all of these sources of irregularity are important optimization targets, the kind of irregularity we address in this dissertation is *wavefront irregularity*; that is, irregularity in the computational wavefront of an algorithm as it proceeds. This irregularity (in various forms) is intrinsic to many applications and is particularly detrimental to performance on a GPU because the GPU is a wide-SIMD machine whose lanes must be fully utilized to achieve optimal throughput.

Figure 1.3 shows an example of how wavefront irregularity may arise in a sample application. The application is represented as a Data Flow Graph (DFG), with nodes representing computational work to be performed on input items and edges representing data paths between nodes. If arbitrary filtering (i.e. dropping) of data inputs is possible in the nodes, efficiently mapping data inputs to SIMD lanes for execution becomes nontrivial as the computational wavefront of pending data inputs

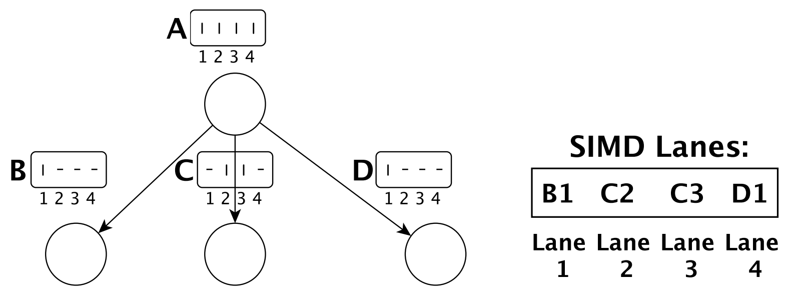


Figure 1.3: Dataflow execution graph showing groups of valid data items (vertical bars) and filtered items (horizontal dashes) during the first few execution steps of an irregular application. All nodes are assumed to execute the same code, so that inputs from any node may execute in SIMD together. The assignment of items to SIMD lanes shown here highlights the irregular-wavefront work assignment challenge; items must be densely gathered from their sparse storage at nodes B, C, and D in order to fill the SIMD lanes.

is unpredictable. Section 1.2.1 lists many applications from diverse domains such as scientific computing [121], modeling and calculation [57,75,112], computer vision [128], and machine learning [16] that exhibit this type of wavefront irregularity.

In addition to the arbitrary filtering of data inputs, another application characteristic that gives rise to wavefront irregularity is having a variable amount of work to do for each data input at each processing step. A variety of big data applications construed as vertex-centric graph processing problems exhibit this characteristic, such as those found in the GraphLab framework [72], the Pannotia benchmark suite [22], and the LonestarGPU suite [20]. In these applications, rounds of computational work are done for each vertex proportional to the number of its incident edges that are 'live' in a given round; since the number of live edges can vary for each round for each vertex, the amount of work to be done in a given round is an irregular wavefront that can lead to low SIMD utilization if not managed efficiently. Chapter 2 will explore in detail a DNA sequence alignment application that also exhibits variable work per data input, though in a slightly different way.

In light of our discussion so far, efficiently mapping wavefront-irregular applications onto GPUs would yield a big payoff: low-cost high-throughput processing of high-impact applications. There is therefore strong incentive to overcome the obstacles to efficient mappings of these applications onto GPUs, and our contributions seek to do just that.

1.2 Research Questions and Direction

The preceding motivation for wavefront irregularity suggests the following research questions, which this dissertation addresses:

1. Is there an execution paradigm that can be used to represent the running of wavefront-irregular applications on a wide-SIMD architecture?
2. Can general techniques be identified for efficiently mapping wavefront-irregular applications onto these architectures?
3. If so, can these techniques be automated so that they are transparent to application developers across many domains?

We answer all these questions in the affirmative. As evidence of our answers this dissertation presents a developer framework called Mercator that (1) allows developers to map irregular applications onto GPUs according to the *streaming paradigm* while abstracting from low-level data movement and (2) includes *generalized* techniques for *transparently* overcoming the obstacles to high throughput presented by wavefront-irregular applications on a GPU.

In the remainder of this chapter, we first present the streaming paradigm and argue that it is appropriate and advantageous for modeling the execution of irregular applications on wide-SIMD architectures, then outline our strategies for efficient transparent handling of irregularity on a GPU within the streaming paradigm. We conclude the chapter by putting this dissertation in the context of previous work and outlining the remainder of the dissertation.

1.2.1 The Streaming Paradigm

Intuitively, the streaming paradigm models the execution of applications represented as Data Flow Graphs (DFGs) such as the one in Figure 1.3 that a large dataset of identical inputs must traverse.

More specifically, we use the term “streaming computing” broadly to indicate 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 stream in real time).
- Each input item must be processed, usually by performing fixed computations on the item’s data.
- Each item may generate zero or more output items when processed.
- Desired performance comes from optimizing total *throughput* as opposed to latency of a single item. Here throughput is defined as the number of input data items consumed per unit time.

The size and type of a “data item” are application-specific and therefore programmer-defined. The computations to be performed on data inputs are grouped into *modules* that are instantiated in one or more nodes of a computation graph, each of which is executed on a physical processor. Data processing consists of “streaming” the data through the pipeline of computation nodes. Streaming computing is often associated with digital signal processing (DSP) applications such as audio/video filters [119], coders/decoders [35], and telescope data filtering [121], though it may also be applied to a variety of other high-throughput applications such as data encryption/decryption [42] and financial simulations [112].

The pipeline of a streaming Monte Carlo simulation of Value At Risk (VaR) for a financial application is shown in Figure 1.4 as an example.

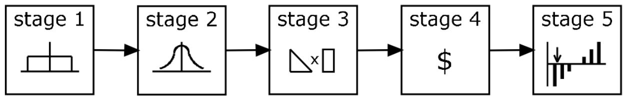


Figure 1.4: Pipeline of a streaming VaR simulation application, taken from Figure 1 in [21]. Stage 1 generates a random number stream, and other stages perform various operations on chunks of data from the stream: transforming their distributions (Stages 2-3), calculating profit/loss values (Stage 4), and sorting results (Stage 5).

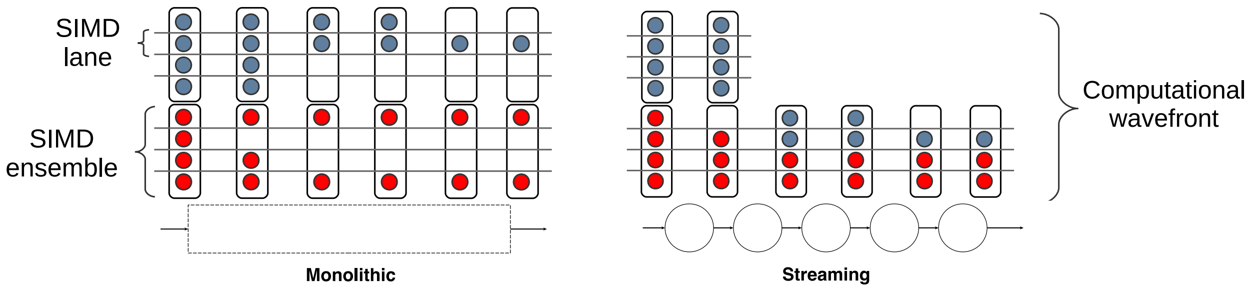


Figure 1.5: Comparison between monolithic application structure and modularized streaming structure for rectification of wavefront irregularity. The streaming paradigm's decomposition of an application into nodes provides convenient checkpoints at which to reorganize data into an efficient execution wavefront of SIMD-width ensembles, thus *isolating* irregularity to the node in which it arises. Meanwhile, a monolithic application may suffer performance degradation due to an early manifestation of irregularity for the remainder of its execution.

Beyond this general description of the streaming paradigm, more specific models of streaming computing may be employed based on assumed characteristics of the input data such as rate and variability of dataflow. We consider only *stateless* streaming applications; i.e., those for which an input's computation does not depend on results from previously processed inputs. Our streaming model of computation (MoC) should reflect these constraints. Chapter 4 will present a new dataflow execution model that captures the instantiation of the stateless streaming paradigm on a GPU architecture as one of our contributions.

Why use the streaming paradigm? With its support for data parallelism, the streaming paradigm naturally facilitates efficient SIMD execution of regular streaming applications: since multiple inputs are queued at each node and each node represents code to be applied to its inputs, the contents of a node's queue may be packed into SIMD lanes, and the node's code applied to them in parallel.

However, the streaming paradigm is also advantageous to use for modeling the execution of *irregular* streaming applications for two reasons. First, the streaming paradigm *surfaces* irregularity through its highly regularized execution model. As Figure 1.3 illustrates, inputs in SIMD lanes that get filtered out during a node's computation create obvious gaps in the input stream when passed downstream for further processing.

Second, the streaming paradigm *isolates* irregularity through the modularity of its execution model. Because applications are decomposed into modules, any irregularity that arises during a node's execution– e.g., filtering or replication of inputs– may be rectified directly following the node's execution at the time its outputs are passed downstream, rather than persisting throughout the remainder of the application's execution. The rectification points, then, are the edges of the streaming DFG: if data movement is well-managed, then any input filtered out by a node will not be passed downstream and the input queue at each node will contain only the still-live items to be processed by that node. These inputs can then be mapped directly to SIMD lanes. Figure 1.5 illustrates this advantage. Our contributions in this dissertation help to accomplish efficient data movement between nodes, leveraging the edges of streaming DFGs to maintain high SIMD utilization in the presence of irregularity.

How must the streaming paradigm be implemented? In order to cover a diverse set of applications with different dataflow paths, our implementation of the streaming paradigm must support a rich set of DFG topologies that include branching and feedback loops. In Chapter 3 we will show how our MERCATOR developer framework efficiently supports such topologies.

In order to realize the potential performance gains of the data parallelism exposed by the streaming paradigm, the overheads of data movement and DFG execution must not outweigh the benefits of modularization. To achieve this, the following opportunities for SIMD parallelism must be exploited with implementations that are efficient for all supported DFG topologies:

- Node execution: SIMD lanes must be packed for high occupancy when executing each DFG node.
- Data movement: Irregularities in dataflow such as filtered or replicated items must be handled along dataflow edges to allow efficient downstream processing.
- Scheduler operations: scheduling and other centralized operations must have minimal runtime overhead.

Our work exploits these opportunities with efficient generalized CUDA implementations of the above operations that remain transparent to developers. Chapter 3 describes these implementations in more detail.

1.3 Related Work

We now place this dissertation in the context of previous work relevant to our three main research questions. Pertinent to our first research question, we first discuss parallel processing that conforms to paradigms other than streaming, then discuss our work's relation to other streaming systems. Pertinent to our second and third research questions, we discuss generalization and automation in conjunction with each approach to parallelism where appropriate, then place our work on the spectrum of parallel code abstraction afterwards.

1.3.1 Non-streaming Approaches to Parallelism

Differing execution paradigms supporting parallelism reflect the distinct characteristics of the data on which they operate. The streaming paradigm is suited to processing an unbounded number of independent data items once each and then discarding them; other paradigms are suited to different execution models. While parallelization strategies associated with these other paradigms are not fully applicable to the streaming context, some of their intuitions and techniques do transfer between paradigms. We present the most relevant of these here.

Work stealing for task DAGs Task-graph applications are represented as Directed Acyclic Graphs (DAGs) whose nodes are individual coarse-grained tasks and whose edges are dependencies between tasks. Rather than processing an arbitrary number of inputs as in the streaming paradigm, a task graph is designed to execute a single application run efficiently, with edges implementing fork-join semantics in the application code and parallelism coming from executing distinct tasks simultaneously.

Since its introduction in 1999, the work stealing strategy developed by Blumofe and Leiserson [15] has gained wide popularity for efficiently executing task-graph applications on multicore processors and distributed systems. Several production-level systems exist for transparently providing work-stealing functionality, including Intel's Cilk Plus [23], based on Blumofe and Leisersons's Cilk [14] and Cilk++ [64], and Intel's Threaded Building Blocks (TBB) [102]. These systems primarily target task parallelism but provide some support for data parallelism via CPU vector instructions. Recent efforts have also produced GPU runtime systems that implement work-stealing techniques (e.g. [11, 122, 123]).

In terms of expressivity, systems that realize the task-graph paradigm support more applications with DAG topologies than our MERCATOR streaming framework because our framework does not currently support 'join' semantics. However, one version of such semantics suitable for the irregular streaming context is provided by Li et al. in [19] and may be incorporated into MERCATOR in the future. At a more fundamental level, the streaming paradigm (and our implementation of it) supports a richer set of application topologies than task-graph execution since it permits cycles in its dataflow graphs rather than restricting them to DAGs.

In terms of support for irregularity, work stealing does support a type of irregularity in that it dynamically load-balances the execution of variable-work tasks across processors at runtime. However, this technique is not transferrable to our framework since it pertains to irregularity at the level of coarse-grained tasks rather than at the level of fine-grained data wavefronts executing in SIMD. One technique we do adopt from the GPU work-stealing systems is the idea of a scheduling framework that runs entirely on the GPU according to the *persistent threads model* [1], allowing blocks to repeatedly pull/push and execute work with no CPU intervention until all work is complete. Our MERCATOR framework provides such a scheduler transparently to developers, allowing individual GPU blocks to repeatedly pull streaming inputs from a common buffer and execute them until the input buffer is empty and the application completes.

Polyhedral analysis At the opposite end of the granularity spectrum from work stealing is polyhedral analysis, which is extremely fine-grained but restricted in its application. Polyhedral analysis

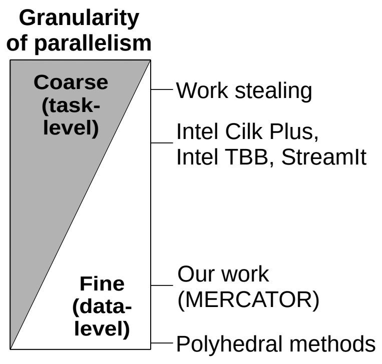


Figure 1.6: Spectrum of parallelism granularity targeted by various systems. The spectrum is represented with overlap between coarse-grained task parallelism and fine-grained data parallelism since many systems support some mix of both. Pure work stealing is designed for coarse task-level parallelism, while polyhedral methods are designed for fine data-level parallelism. Intel Cilk Plus [23], TBB [117], and StreamIt [119] primarily target task-based parallelism but provide some vectorization. Our work targets data-level parallelism that is coarser than polyhedral methods because we seek to parallelize entire functions rather than just tightly nested loops.

identifies dependencies in nested loop structures, and polyhedral techniques re-organize the execution of these loop structures in a way that exposes maximal parallelism. Polyhedral techniques have achieved significant speedups on many parallel architectures, including GPUs [5, 83, 127]. However, polyhedral techniques are designed exclusively for nested loop structures rather than general-purpose programs and do not account for the kinds of irregular filtering and replication our work aims to support.

By conforming to the streaming paradigm, we target parallelism at a level of granularity between that of work stealing and polyhedral techniques. Figure 1.6 illustrates this context.

Graph-based processing The graph-processing paradigm of execution is designed for iteratively processing inputs representing graph elements. Many high-throughput applications of recent interest and the datasets they operate on are naturally represented using a (large) graph structure; examples include analysis of social network connections [125] and ranking of website relevance [93]. To make analyzing these graphs tractable, paradigms of graph processing typically operate locally on each graph element, proceeding for a fixed number of rounds or until convergence. The elements processed during each round may be vertices, in which case the paradigm is called *vertex-centric* graph processing or Think Like A Vertex (TLAV) processing [76], or edges, in which case the paradigm is called *edge-centric* graph processing [105]. The TLAV paradigm in particular has proven useful as an execution model for high-impact applications, inspiring systems such as Pregel [74], which Google uses to compute its PageRank algorithm; Apache Giraph [8], which Facebook uses to analyze its network; and GraphLab [73], which directly supports many parallel machine learning algorithms.

Closely related to the graph processing paradigm in practice is the operator formulation paradigm [97] instantiated in the LonestarGPU system [20]. This system characterizes algorithms by the data structures they require and the operations they perform on them. Although this theoretically allows higher expressivity than a pure graph processing approach, in practice the data structures analyzed are usually sparse graphs, trees, and sets, and the system processes them in a vertex-centric way.

Unlike task graph execution and polyhedral analysis, the graph processing paradigm does not necessarily parallelize at a different native granularity than our streaming-paradigm system. Indeed, graph processing systems have been developed that incorporate both coarse-grained task-level parallelism and fine-grained SIMD data parallelism [54]. However, these paradigms are designed to operate on input sets with fundamentally different characteristics: the graph processing paradigm operates on a fixed set of inputs iteratively, while the streaming paradigm operates on a dynamic set of inputs once each. These paradigms have hitherto been non-overlapping in their application coverage. Although we are not aware of any graph-based systems that support traditional streaming applications, we believe that with an appropriate topology and vertex data storage strategy, the streaming paradigm as realized in our Mercator framework shows promise for supporting some graph-based applications. We explore this possibility in Chapter 5.

Application- or paradigm-specific techniques To conclude our discussion of alternative approaches to parallelism, we briefly mention existing approaches to optimizing irregular applications on wide-SIMD architectures. These approaches share common successful elements but are either application-specific in their implementations or are designed for execution paradigms other than streaming. For example, in [103], Ren et al. introduce a *stream compaction* step for a tree-traversal application 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 [25, 28] uses parallel reduction techniques on a GPU similar to Ren's stream compaction to maintain dense work queues of candidate DNA reads. Our work seeks to generalize these enhancements into a development framework for diverse applications. Khorasani et al. [54] provide this type of GPU framework, but for applications conforming to the graph-based execution paradigm. We seek to develop a generalized framework for streaming applications instead.

1.3.2 Context Within the Streaming Paradigm

Past work in streaming computing has focused primarily on exploiting task-level parallelism by executing independent heavyweight threads on distributed systems. Many Models of Computation (MoCs), beginning with Kahn process networks [53] 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 [62], 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 [61]. As the quintessential SDF framework, different implementations of the StreamIt system allow programmers to transparently apply SDF scheduling for streaming applications on a CPU [119] or GPU [46, 124]. 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 [18], Dynamic Data Flow [94], and Scenario-Aware Data Flow [114], 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 [36] maintains a framework supporting applications conforming to these MoCs, though they only target CPUs. However, since we explicitly target wide-SIMD architectures, we seek to enable expression of irregular applications in a way that naturally exposes not only coarse-grained task parallelism due to modularization but also the fine-grained data parallelism enabled by these architectures.

The Auto-Pipe execution system [41] directly supports modular streaming applications and abstracts data movement from developers with an interface that was influential in Mercator's own design. However, Auto-Pipe was designed to target heterogeneous architectures and optimize workload distribution and communication among heavyweight processors, so its infrastructure is not suitable for optimizing irregular applications targeting self-contained wide-SIMD machines as we seek to do.

One recent work [9] presents a methodology that supports SIMD execution of applications requiring stateful actors but leaves important application-specific components of the methodology to developers. In contrast, we do not require the complexity of supporting stateful actors, and we seek to supply all expert-level parallel algorithms required for the efficient execution of streaming applications.

1.3.3 Level of Developer Abstraction

We conclude our discussion of related work by placing our Mercator framework in the context of other parallel systems regarding the level of abstraction they provide developers from details of implementing their parallelism strategies. Parallel code is notoriously challenging to write and debug, and its efficiency is architecture-dependent. Much work has therefore been done to automate the process of parallel code development and abstract the developer from low-level implementation details. This leads to a tradeoff for programmers between ease of development and control over semantics and optimization for parallel code.

At one end of the spectrum, languages such as CUDA and OpenCL and language features such as vector intrinsics give the developer maximum control over the mapping of their application onto parallel hardware, at the cost of acquiring expert-level knowledge to master the language. Scores of efficient GPU implementations of applications testify to the effectiveness of low-level development in a language such as CUDA, but the proliferation of publications presenting clever GPU application mappings testifies to difficulty of developing these implementations (for a sample of such implementations see [48]). At the other end of the spectrum, directive-based programming allows novice developers to target parallel hardware such as GPUs with minimal code change, at the cost of control over mapping details. We now discuss directive-based programming and related techniques in more detail, then place Mercator on the ease-control spectrum.

Directive-based programming In directive-based programming, a developer provides explicit hints to a compiler about how to parallelize specific code sections via special `pragma` statements and the compiler translates the hints into architecture-specific optimizations when it generates its output. The preeminent framework for directive-based programming for multicore CPU processing is OpenMP [32], which allows a developer to suggest optimizations such as loop unrolling. OpenMP also allows more complex specifications of parallelism, such as differentiation between fine-grained data parallelism and coarse-grained task parallelism, that facilitate processing across SIMD lanes as well as across CPU cores. OpenACC [91] has emerged as an analogue to OpenMP in the hardware accelerator domain and is currently compatible with NVIDIA and AMD GPUs as well as the Intel Xeon Phi. OpenACC directives may be used to mark code that should be launched on a GPU as a kernel in addition to suggesting similar optimizations to OpenMP.

Directive-based frameworks such as OpenACC successfully insulate the developer from low-level implementation details of their parallel code; indeed, all developer code is written in `Fortran/C/C++` rather than an accelerator language such as CUDA or OpenCL, and the only interaction the developer has with the hardware accelerator is via the OpenACC API. However, this extreme abstraction comes at the cost of support for complex parallelism. Directive-based frameworks are designed to optimize regular computations such as nested loops rather than general tasks, and their data movement strategies are designed to optimize data transfers for these tightly structured code blocks. In contrast, the strategies we present in this dissertation address the challenges of wavefront-irregular computation, which may occur in arbitrary code structures across large blocks. In addition, our strategies maximize the amount of application logic packed into a single kernel, which obviates repeated kernel calls and data transfers that may be necessary in a host-controlled directive-based framework. Overall, then, directive-based frameworks such as OpenACC provide greater portability than our Mercator framework while Mercator supports more nuanced parallelism.

Parallel libraries In addition to directive-based frameworks, many architecture-specific libraries have been written to optimize common computations and expose them to a developer via an API in CPU code. For example, the Thrust template library [47] for CUDA allows host-side API calls to create and manipulate data structures on the GPU with operations such as sorting and reduction. Like directive-based frameworks, libraries such as Thrust primarily target a specific type of parallelism over a short code span and lack strategies for mitigating wavefront irregularity and GPU-persistent applications.

Mercator On the spectrum of ease of programming vs. implementation control, our Mercator framework lies between directive-based programming and systems such as Cilk Plus on the ease-of-programming end. Unlike directive-based programming, Mercator expects programmers to parallelize entire functions and requires them to explicitly interact with framework components directly in their application code rather than in compiler hints. However, unlike the other frameworks discussed in this section such as Cilk Plus, Intel TBB, and StreamIt, Mercator abstracts programmers from all semantics of parallelism, allowing them to write code as though for a single SIMD lane and managing the parallelism transparently. Figure 1.7 shows Mercator's place on the ease vs. control spectrum of parallelism support.

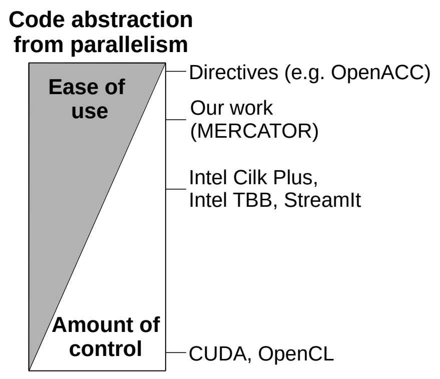

Code abstraction from parallelism
Ease of use
Amount of control
Directives (e.g. OpenACC)
Our work (MERCATOR)
Intel Cilk Plus, Intel TBB, StreamIt
CUDA, OpenCL

Figure 1.7: Spectrum of developer abstraction from implementation of parallelism. The spectrum is represented with overlap between ease of programming and implementation control since a tradeoff exists between these components. Directive-based tools such as OpenACC [90] require no direct code changes but provide no access to implementation details, while full languages such as CUDA [31] and OpenCL [45] allow fully customized parallel implementations but require significant programmer intervention. Cilk Plus, TBB, and StreamIt allow some programmer control over parallelism with added language keywords while hiding implementation details of those keywords. We seek to abstract programmers from parallelism entirely, but our Mercator system does come with its own API with which programmers must interact.

1.3.4 Road Map and Contributions

To address our main research questions, this dissertation makes the following contributions:

a. A novel Model of Computation (MoC) for streaming computing called the Function-Centric Model that more fittingly expresses the parallel data movement and execution capabilities of a wide-SIMD machine than previous models;

b. A developer framework called Mercator that transparently instantiates the FC Model, allowing developers to write module function code and automatically generating all other code for data movement and execution of the application on a GPU, employing remapping between node executions to maintain high SIMD occupancy;

c. A scheduling strategy for Mercator that guides applications toward high throughput by maximizing SIMD occupancy during execution.

The remainder of this dissertation is organized as follows: we discuss in Chapter 2 typical application characteristics that can generate an irregular SIMD computation wavefront. We use DNA sequence alignment as a motivating application and present our strategies for mitigating the irregularity of the DNA sequence alignment application. These strategies are implemented in our Woodstocc tool [25, 28] and were the inspiration for creating the Mercator framework. We end that chapter with a discussion of the salient features of the tool and the algorithm's irregularity that are relevant to a generalized framework for executing other irregular streaming applications.

We then introduce our Mercator framework, which provides developers with transparent implementations of our strategies. We first discuss Mercator's user interface, basic execution model, and performance results in Chapter 3, drawing material from [26]. We then discuss its queueing model and scheduling strategy implementation details, including the operation of its scheduler, in Chapter 4.

We conclude with a discussion of future work in Chapter 5, highlighting extensions of Mercator to support synchronization and input reductions for graph-processing applications.

Orthogonal concerns Optimizing programs for efficient GPU execution is a complex problem with many facets beyond the scope of this work. In this dissertation, we focus on optimization challenges inherent in *any* wide-SIMD architecture, and we therefore do not study optimizations in the following areas:

- Other kinds of irregularity besides wavefront irregularity, e.g., irregular memory accesses and code divergence. We attempt to minimize these kinds of irregularity in our system implementations, but we make no attempt to do so automatically nor to explore general strategies for doing so.
- GPU memory layout. GPUs have a complex memory hierarchy, and clever use of it can significantly reduce memory access costs. For simplicity and portability to other architectures, we store all application-specific data in the GPU's global memory, which is accessible to all threads on the device.

- Performance-optimal but sub-maximal SIMD occupancy. Although previous work has shown that maximum SIMD occupancy does not necessarily yield maximal throughput [129], the ability to execute with maximal occupancy is an important input to any throughput optimization problem. We therefore pursue techniques to achieve high occupancy without proving that the achieved occupancy is optimal.
- Other GPU configuration decisions, such as grid dimensions of the kernel launch.

Chapter 2

Wavefront Irregularity Case Study: Woodstocc

We now describe a DNA sequence alignment tool for GPUs called Woodstocc, which addresses multiple sources of wavefront irregularity inherent in a straightforward SIMD execution of its underlying algorithm. We present the context and contributions of Woodstocc to give a comprehensive view of an irregular application "in the wild" and to show the benefits of a GPU implementation that deliberately mitigates irregularity to achieve high SIMD utilization. This implementation conforms to the streaming paradigm of execution and exploits its separation of modules to eliminate the gaps in its computational wavefront caused by input-dependent filtering inside the modules as inputs flow downstream. In so doing, it employs techniques to address the three challenges mentioned at the end of Chapter 1: efficient execution of module code, efficient data movement, and a scheduler capable of throughput-optimizing module execution decisions.

In addition to wavefront irregularity from filtering, Woodstocc also addresses irregularity that arises as a result of alternating granularities of operation and load imbalance between GPU blocks. Although its implementation is application-specific, the techniques it employs to mitigate irregularity generalize to a wider class of streaming applications. Following our presentation of Woodstocc, we will discuss these generalizations, a subset of which form the core of the Mercator system to be presented in Chapter 3.

WOODSTOCC was originally published and presented at the International Symposium for Parallel and Distributed Computing (ISPDC) [25], and a follow-up version of the conference paper with the more extensive performance optimizations and results included in this chapter was published as a technical report [28].

2.1 Overview

Short-read DNA sequence alignment is a compute-intensive application of strong interest to the biological community. In this application, a large number of short DNA strings ("reads") are matched against an index generated from a long DNA reference string to locate areas of correspondence. Although all reads are processed with the same algorithm, each one requires a different amount of computation to compare against the reference. This input-dependent processing behavior is a hallmark of algorithms that exhibit wavefront irregularity.

The WOODSTOCC ("WOODSTOCC Offers Optimal Dynamic programming - Suffix Trie alignment while optimizing OCCupancy") tool seeks to ameliorate the fundamental irregularity of short-read alignment in order to extract SIMD parallelism suited to a GPU. We first describe algorithmic transformations of short-read alignment, applicable to any architecture, that partly regularize it and expose SIMD parallelism. We then present our CUDA implementation of WOODSTOCC for NVIDIA GPUs, which exploits several techniques to maximize occupancy and minimize wavefront irregularity. Finally, we empirically study the performance of WOODSTOCC, showing that it indeed boosts occupancy and suggesting generalizations of its techniques for other irregular streaming applications.

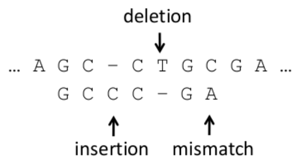


Figure 2.1: Alignment of a read `GCCGA` to a longer reference sequence with three differences.

2.2 Background

2.2.1 Problem Definition

Woodstocc focuses on a variant of DNA sequence alignment known as the *short-read alignment problem*, which is motivated by the development over the last decade of experimental techniques for sampling a very large number of short substrings, or *reads*, from a long DNA sequence such as a genome [110]. This inexact matching problem is as follows:

Given a single DNA reference sequence of length n, and many DNA reads of common length $m \ll n$, find for each read all starting positions in the reference where the entire read matches with no more than k differences.

Allowable differences include substitution, insertion, or deletion of individual characters. Figure 2.1 shows a read alignment to part of a reference with three differences. Typically, the reference length n is tens of millions to billions of characters, while the read length m ranges from a few tens to 200 characters. A sequencing experiment generates tens to hundreds of millions of reads that must be aligned to the reference.

2.2.2 Related Work

The computational demands of short-read alignment have led to several algorithmic strategies for rapidly matching many reads against a common reference. Methods using a hash-based index of the reference [66,106] have largely been superseded by tools that construct a tree-based index, such as a *suffix trie* (see Figure 2.2). Due to the high storage cost to represent such tries explicitly, they

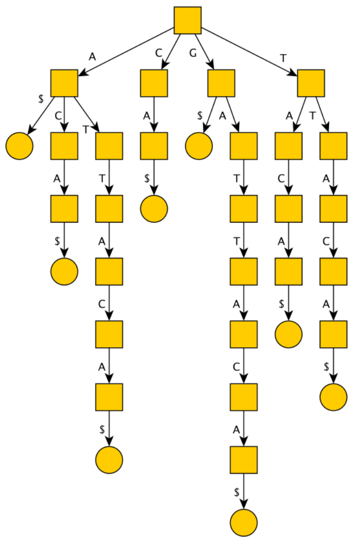


Figure 2.2: Suffix trie for string `GATTACA$`. Each path from root to leaf is labeled with a suffix of the string.

are instead represented *virtually*, using a compact data structure such as the FM-index [39] that allows the trie to be reconstructed on the fly during alignment. FM-index-based aligners represent the current state of the art in fast short-read alignment software [59, 65, 68].

Our approach to short-read alignment emphasizes *completeness* of results. Most short-read aligners use trie-search heuristics that can quickly locate at least one match to a given read in the reference with up to k differences, if any exist, but may not find all such matches. Incomplete results are useful for read-mapping applications in which matches to multiple sites are simply discarded, but completeness is desirable for more precise analyses such as genome rearrangement history [2] and interspecies read alignment. Among CPU-based aligners, the BWA software [65] in "`-N` mode" is one of the few that produce complete output, so we use this tool as our CPU baseline for performance comparison.

Like Woodstocc, some short-read alignment tools use a dynamic programming (DP) algorithm such as Smith-Waterman [113] as part of their search algorithm. However, these tools are either optimized for aligning much longer sequences than a typical short read [58, 67] or use DP only to filter putative matches obtained by an incomplete trie-search heuristic [37, 60].

Modern short-read alignment methods have previously been ported to GPUs. SARUMAN [13] uses a hash-based index on the GPU, but its storage requirements restrict its use to microbial-sized genomes (around 10^6 characters). MUMmerGPU [109] searches an explicit representation of the suffix tree (a compressed trie), which is copied to the GPU a piece at a time; however, it does not allow for differences between reads and reference. BarraCUDA and CUSHAW [55, 70] implement BWA-like implicit trie search on the GPU, but they are limited either to incomplete trie search heuristics or to finding alignments without insertion and deletion of characters.

In contrast to previous GPU-based short-read aligners, Woodstocc combines virtual trie traversal with dynamic programming to guarantee completeness of its results. Its approach to regularizing GPU-based computation shares some similarities with BarraCUDA's worklist strategy, but it is quite different in detail to accommodate the needs of both dynamic programming and trie traversal.

2.3 Alignment Strategy

This section describes how Woodstocc organizes its alignment computation at a high level to expose regular structure. We exploit the highly structured nature of dynamic programming alignment to allow computations for many reads to proceed in lockstep, which is crucial to a SIMD implementation.

2.3.1 Core Algorithm

Conceptually, we attempt to align a given read starting at every character in the reference by aligning it to each labeled path in the reference's (virtual) suffix trie. The trie is traversed depth-first beginning at the root. Each traversal step descends by one node, or equivalently one character, along an edge and computes one row of a dynamic programming matrix (a "DP row"). When the computation reaches node x of the trie, the DP row contains the least number of differences in an optimal alignment between each prefix of the read and all occurrences in the reference of the

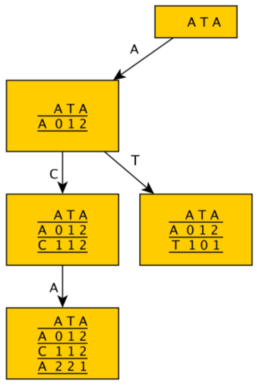


Figure 2.3: Alignment of read `ATA` against a fragment of the reference trie of Figure 2.2. Each internal node shows the dynamic programming matrix (set of DP rows) between the read and the string labeling the path to that node. Each downward step computes one more row of the matrix.

substring labeling the path from the root to x. The set of leaves below the current node indicate the reference locations at which this substring occurs. As shown in Figure 2.3, when the trie branches, the work done to compute the DP row at the branch point can be reused on each of the branching paths; hence, the total number of rows computed equals the number of nodes reached by the traversal.

Because the number of differences allowed in the alignment is *a priori* limited to k, traversal down a given path may be truncated once it is clear that all alignments of the read to that path must have more than k differences, i.e. when all entries of a DP row become $> k$. Truncating alignments in this way limits the depth traversed down any path from the root to $O(m)$ and so greatly curtails the cost of alignment overall. We also note that limiting the number of differences to k permits the use of *banded alignment*, which computes only $2k + 1$ cells per DP row regardless of the read length m. In all the work described for this chapter, we set $k = 3$.

Because storage of the complete suffix trie, or even a more efficient suffix tree, would be prohibitively large for genomes of 10^8 or more characters, the trie is stored implicitly using the FM-index. Each step down a path in the trie involves a computation to "discover" the current node's children, as described in [65]. A discovery computation performs several random accesses to the FM-index data structures in memory, which are comparable in size to the original reference sequence.

Algorithm	Time			
	Total(s)	Traversal (%)	DP calculation (%)	Other (%)
BWA -N	634	N/A	N/A	N/A
Naive aligner	860	51	19	30
Batched aligner	454	17	63	20

Table 2.1: Cost to align 10^5 reads of length 47 against human chromosome 1 on one core of 2.6 GHz Intel Core i5 CPU. For Woodstocc algorithms, time spent in each portion of the algorithm is profiled.

2.3.2 Read Batching

Table 2.1 compares the running time of the above algorithm ("naive aligner") to that of BWA v0.6.0 (using the `-N` option to ensure completeness of output) when sequentially aligning a list of 10^5 reads of length $m = 47$ against human chromosome 1 ($n = 2.5 \times 10^8$), on a single core of a 2.6 GHz Intel Core i5 CPU. The majority of the algorithm's time is spent in the discovery computations of virtual trie traversal; overall, it is slower than BWA.

To reduce the cost of trie traversal and lay the groundwork for the parallelization to follow, we implement *read batching.* In one traversal of the suffix trie, we simultaneously perform dynamic programming alignment computations for many different reads, maintaining separate DP row data structures for each. Each time a node is discovered by traversal, the DP rows for all reads are updated as the traversal moves down to this node. This process is conceptually equivalent to doing a single traversal of the trie and then reusing all the discovered nodes to perform dynamic programming for each read. Because the discovery computation is expensive, performing it once per node instead of N times for N reads substantially reduces computation at the cost of maintaining N simultaneous DP rows, each of size $O(k)$. Figure 2.4 shows a snapshot of a trie traversal employing read batching.

An important detail of read batching is that some, but not all, reads in a batch may have their alignment computation truncated on a given path due to having no alignment with $\leq k$ differences. We track the set of "live" reads in a batch during traversal (i.e. those that have not yet been truncated) and perform dynamic programming at each step only for live reads. In this way, the total DP work performed matches that of the naive implementation, while the traversal work is

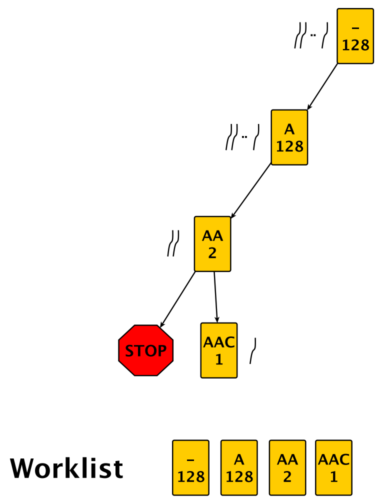


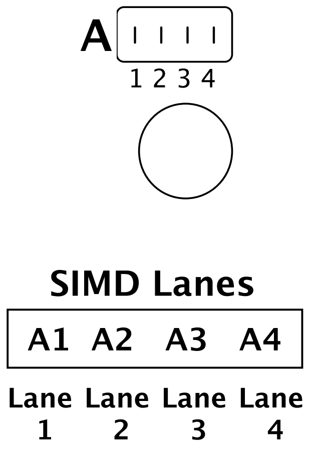
Figure 2.4: Snapshot of an example trie after several steps of batched traversal. Trie nodes are labeled by the substring they represent and the number of reads live at the node. A worklist holds trie nodes with pending extensions. The 'STOP' node indicates a traversal path where all reads became idle and the path was pruned.

much less. As shown in the last row of Table 2.1, batching all reads into a single group greatly reduces time spent in traversal and nearly halves overall running time.

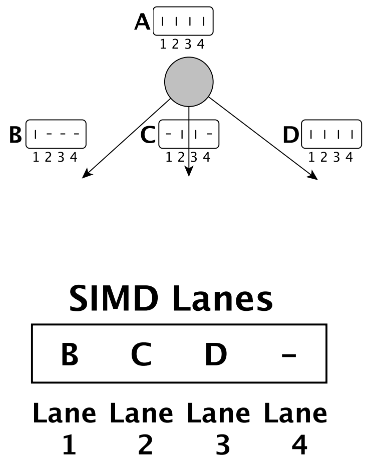
Implications for Parallelization

Dynamic programming with read batching introduces a high degree of regularity to read alignment. In particular, all live reads' DP computations following a given traversal step can be performed in lockstep using SIMD parallelism. This regularity is in contrast to multithreaded CPU versions of BWA and related aligners, which use one independently executing thread per CPU core, each processing a separate stream of reads. Batching is particularly attractive for devices with a wide-SIMD programming model, such as GPUs.

However, parallelizing the alignment of each read in a batched implementation also introduces two sources of wavefront irregularity: the *granularity difference* between trie traversal and dynamic programming operations, and the uneven distribution of *idle reads*. The granularity difference arises because trie traversal need be done only once per trie node while dynamic programming must be done for each read still live at that node. Efficiently executing both interleaving operations in SIMD



(a) **Item granularity**. When executing DP at the root node, SIMD lanes (GPU threads) are filled with individual reads.



(b) **Group granularity**. After executing DP at the root node, the set of live reads for each candidate extension must discover its corresponding child node. SIMD lanes (GPU threads) are therefore filled with read groups.

Figure 2.5: Dataflow execution graph showing groups of live reads (vertical bars) and idled reads (horizontal dashes) during the first few execution steps of a batched read alignment, which requires group- and item-wise operations. Nodes in the virtual suffix trie are represented as circles, and batches of reads at each node are enclosed and labeled with a capital letter. (a) and (b) together illustrate the multiple-granularity work assignment challenge.

requires intelligently remapping threads to work items on the fly to avoid both idle threads during the trie traversal step and insufficient threads to handle the workload in the dynamic programming step. Figure 2.5 illustrates the challenge of keeping SIMD lanes (equivalently, GPU threads) occupied in the presence of the granularity difference.

The problem of idle reads arises as candidate reads “drop out” during the course of the alignment. Traversal down a given path in the trie continues until no reads in the current batch are live; however, some reads may become idle before others, and no further progress is made on those reads until the traversal returns to a node at which they are live. If reads are statically batched and assigned to compute resources (e.g. vector slots for SIMD instructions, or GPU threads), resources will be left idle whenever their reads become idle, limiting parallelism. Figure 2.6 illustrates the challenge of keeping SIMD lanes occupied in the presence of idled reads.

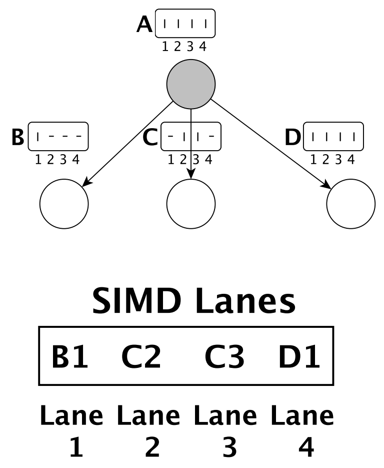


Figure 2.6: Dataflow execution graph from Figure 2.5 showing the next step of DP execution. Idled reads force (sparse) remaining live reads to be gathered from multiple batches to fill SIMD lanes.

Depth	Liveness	Depth	Liveness
0-3	1	8	0.014
4	0.377	9	0.005
5	0.191	10	0.002
6	0.086	11	0.001
7	0.036		

Table 2.2: Average fraction of live reads at traversal depths 0-11 in the batched implementation, $k = 3$.

To assess the impact of idle reads, we measured the proportion of live reads encountered by the batched implementation at a range of depths in the trie (averaged over all nodes reached at that depth). These results are given in Table 2.2. All reads remain live near the top of the trie, in particular for the first k levels where no alignment accumulates enough differences to be ruled out. Thereafter, a substantial fraction of reads are truncated at each level; after just a few levels, most reads at a given node are idle.

A wide-SIMD architecture such as a GPU encourages highly regular parallelism, yet the fundamentally irregular nature of when reads' alignments are truncated in trie traversal demands that we eliminate or mitigate idle reads. Dealing with these conflicting design pressures informs our GPU implementation, which we describe next.

2.4 GPU Implementation

We now describe Woodstocc, the implementation of our aligner on a GPU using CUDA. We first describe at a high level our mapping of the alignment algorithm onto the GPU. We then identify a flexible mapping strategy that maintains high utilization of the GPU's parallelism throughout the various stages of the algorithm despite the irregular aspects of the application's execution.

2.4.1 Application Mapping to GPU

Woodstocc divides the set of reads to be aligned across multiple blocks on the GPU. Each block operates autonomously and runs a CUDA kernel that implements virtual suffix trie alignment on a disjoint subset of reads. All blocks share a common FM-index data structure for virtual trie traversal, which occupies the majority of global memory. However, each block independently performs DP and traversal of the virtual suffix tree to align its subset of reads. As each block discovers alignments of its reads to the reference with at most k differences, it aggregates them into a per-block global memory buffer that is copied back to the host when the block completes.

The relatively short duration and high data volume of each DP and trie traversal calculation step in our algorithm precludes mapping strategies that require frequent control transfers between CPU and GPU. In particular, we cannot put either trie traversal or DP alone on the GPU but must implement the entire algorithm. We chose to implement the algorithm as a single, monolithic kernel to enable data transfer between algorithm stages in per-block shared memory. The principal innovation in Woodstocc's approach is in how it organizes this application kernel, which we describe next.

2.4.2 Kernel Organization to Maximize Occupancy

Woodstocc is designed to map the threads of each block to the work of the application kernel so as to maximally exploit the SIMD parallelism of the GPU. In particular, we seek to maximize

GPU *occupancy* in two senses: first, all threads within a block should be kept busy doing useful work (thread occupancy); and second, there should be enough available blocks of work that no SM is idle for lack of a block to work on (block occupancy).

In a traditional, regular GPU application such as matrix multiplication, a straightforward static mapping of threads onto work units for the duration of a kernel suffices to maintain high occupancy. However, the irregular nature of short read alignment demands a more nuanced approach. Batching is an effective algorithmic optimization, but when few reads remain live, thread occupancy within a block suffers greatly. This challenge extends to block occupancy as well – some blocks may finish much sooner than others, leaving SMs partially or completely idle while the last few blocks finish.

Figure 2.7 illustrates the structure of the Woodstocc GPU kernel. The kernel repeatedly executes a main loop that performs alternating DP and trie traversal steps. Two key features of this kernel are the use of a *worklist* and associated *pull stage*, which ameliorate the problem of idle reads in DP, and *dynamic remapping*, which recomputes the mapping of work to threads in a block on the fly, entirely within the GPU, to enable differently shaped computations to run efficiently in a single kernel. These two features directly address the two types of wavefront irregularity described in Section 2.3.2.

Eliminating idle reads

Recall that the CPU version of our alignment algorithm works on a single batch of live reads in lockstep until the batch is exhausted. The algorithm therefore works on live reads from only a single trie node at a time. In contrast, Woodstocc aggregates live read batches from multiple trie nodes into a common worklist. On each loop iteration, the kernel pulls and simultaneously computes on as many batches from this worklist as are needed to occupy (nearly) all threads in a block.

The worklist is maintained as follows. Starting with the root node, nodes with live reads are added to the worklist as they are uncovered by traversal. Each queued node requires a DP calculation for

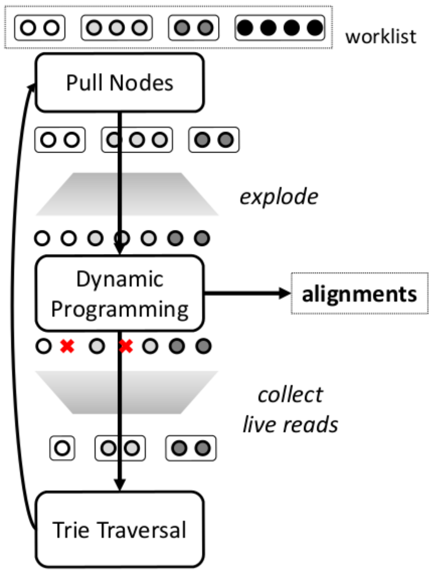


Figure 2.7: Stages of the Woodstocc GPU kernel main loop. Trie nodes with nonempty batches of live reads (circles) are queued on a worklist. The first stage pulls enough nodes to assign a live read to most threads in the block. Nodes are "exploded" into their read sets for dynamic programming, after which the remaining live reads are collected into new, smaller read batches. Finally, nodes are subjected to trie traversal, which generates more work for the worklist. The output of the kernel, read-reference alignments, is generated by the dynamic programming stage.

each of its live reads, as well as a trie traversal step to discover and enqueue any of its child nodes with nonempty live read batches. Nodes become idle when all reads associated with them are idle, or when they have no more children to expose; such nodes are not enqueued. The kernel's main loop runs until the worklist is empty, at which time every possible live node in the trie has been processed.

To utilize the worklist effectively, we add a "pull" stage to the main loop prior to DP and trie traversal. In this stage, the kernel computes a progressive sum of batch sizes for the first m nodes on the worklist, for each m from 1 up to the number of threads per block. Each node may have a different-sized batch of live reads. The kernel then dequeues the maximum number of nodes such that the sum of their batch sizes fits within the number of threads for the block. For efficiency, the progressive sum is computed using a parallel scan operation in time logarithmic in the list size. The DP calculation then devotes one GPU thread to each live read from the dequeued batches.

Because a loop iteration works on either all or none of a node's batch of live reads, the worklist does not guarantee that we always have a live read for every thread in a block. However, we anticipate that most threads will have DP work to do on every loop iteration, leaving very few threads idle. This mitigates the wavefront irregularity that would otherwise result from the variation in live-read counts across nodes.

Dynamic remapping

An important feature of the Woodstocc kernel is that it combines operations at different granularities. Dynamic programming uses one thread per live read, while the pull and trie traversal stages of each loop work with entire nodes, rather than their component live reads. Woodstocc dynamically reassigns work to threads within each loop iteration of the kernel to match the granularity of the computation required by each computation stage to the granularity of the GPU's parallelism.

The pull stage of the computation assigns one GPU thread to each node pulled from the worklist. After some initial work, we must "explode" the nodes to assign one live read per GPU thread for DP. The reads from all dequeued nodes are assigned to consecutive threads, but each read must also know which node it came from. To compute each read's node efficiently in parallel, we use a parallel binary search [38] on the progressive sum of the number of live reads in each node, which was computed in the pull stage.

After DP, we know whether each read remains live, but we must then regroup *only* the live reads for each node into new, usually smaller batches. For efficiency, this operation is also parallelized, using a segmented parallel scan to map the live reads for each node into a dense array stored with the node. Finally, trie traversal is performed at node granularity, and nodes are enqueued onto the worklist for future processing. These operations mitigate the wavefront irregularity that could result from having a mixed idle and live read set present at each node during the pull stage.

2.4.3 Decoupling Traversal from DP

A straightforward implementation of the Woodstocc main loop executes each of its three stages once per main loop of the kernel. However, we anticipate that trie traversal, a major part of the computation, will typically not utilize all available threads. First, the number of nodes dequeued by the pull stage may be much less than the number of threads, since each node may have multiple live reads. Second, some nodes' read batches may be empty after the DP step, in which case those nodes are discarded without the need for trie traversal.

We aggressively maximize thread occupancy for trie traversal by dynamically scheduling the stages of the kernel's main loop. In particular, we allow multiple iterations of the pull and DP stages to be executed between calls to trie traversal. Each iteration performs DP on read batches from a different set of nodes, generating work for the traversal stage. We queue up this work (in the form of pending nodes for trie traversal) on a separate worklist until enough traversal work exists to occupy most or all threads in a block; only then do we pull nodes from this list and perform the traversal operation. This strategy leverages the modularity of the streaming paradigm to mitigate the wavefront irregularity caused by operating on data at multiple granularities.

2.4.4 Boosting Block Occupancy via Kernel Slicing

Periodic kernel slicing is a technique for maintaining occupancy that operates at block rather than thread granularity. The purpose of kernel slicing is to maximize the number of *active blocks* available to execute on a Streaming Multiprocessor (SM) at any given time. When a kernel is launched, it does not terminate until all blocks in its grid terminate, so all blocks must wait for the slowest block to finish. If some blocks are much slower to finish than others, the SM spends much of its time underutilized.

To avoid idle blocks, periodic kernel slicing decomposes the execution of a single monolithic kernel instance into many shorter instances called *slices*, effectively pausing the kernel after each slice and assigning any idle blocks a new set of reads to be processed. Kernel slicing exploits the fact that global memory persists between kernel launches within the same CUDA context. A statically allocated global continuation buffer holds all non-persistent block data between slices, so that control can be returned to the CPU between slices via kernel termination and re-launch, with all necessary data persisting on the device across slices. In this way, the maximum time a block could remain idle is the length of a single slice, rather than the execution time of the maximum-latency block in its grid.

We note that an effect similar to periodic kernel slicing can be achieved in CUDA simply by creating a big enough computation grid that the total number of blocks to process exceeds the maximum number of active blocks for the device. For example, the maximum number of active blocks per SM on the GTX Titan device is 16, for a total of $16 \times 14 = 224$ active blocks. As active blocks are retired, CUDA schedules the remaining blocks onto the available SMs. However, this approach fails to scale to very large data sets because the number of blocks that can be created for one kernel execution is limited by the device resources and/or CUDA implementation. Periodic kernel slicing, on the other hand, faces no such limitation because it periodically restarts the kernel after creating new blocks as needed.

To see why slicing is helpful for our computation, consider Figure 2.8, which shows a CDF of measured block finishing times over the course of a kernel instance. This CDF confirms that, while most blocks have a short execution time, a few take dramatically longer to complete. A monolithic kernel does not return control to the host until every block is finished; hence, blocks with low latencies will remain idle for most of the kernel's overall execution time. With slicing enabled, blocks with low latencies can be assigned new work as blocks with long latencies continue their execution, better utilizing the GPU's resources.

Woodstocc incorporates slicing at intervals of 3000 main loops, which introduces negligible overhead according to profile data. Empirical tests showed that this yields a time savings of approximately 5-10% for input sets of size comparable to the one tested in our experiments.

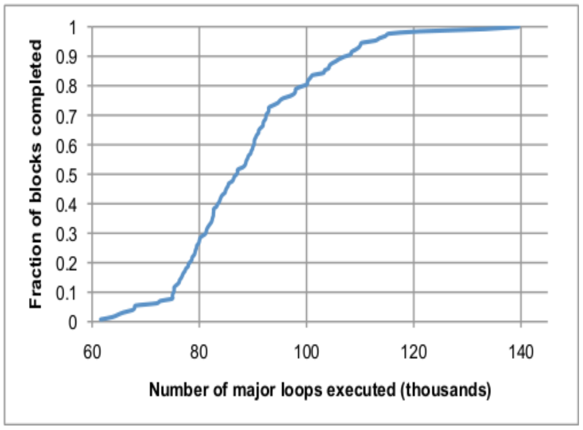


Figure 2.8: Cumulative distribution function of the number of major loop iterations executed by each of 128 blocks when run to completion.

2.5 Results

We now provide empirical measurements to support the utility of our occupancy-boosting techniques for the Woodstocc kernel and to assess its performance.

Measurements were taken on an NVIDIA GeForce GTX Titan GPU with Kepler architecture. This card contains 14 SMs, each of which can maintain up to 64 active warps, operating at 837 MHz with 6.1 GB of global memory. The input set comprised 10^5 length-47 reads and was aligned against human chromosome 1. Unless otherwise specified, the GPU kernel was run with 160 threads per block.

2.5.1 Impact of Worklist and Dynamic Mapping

Table 2.3 illustrates the distribution of work (as measured by GPU cycles) among the three stages of the Woodstocc main loop, as well as the measured thread occupancy for each. The profiled implementation used the worklist and dynamic remapping of work between reads and threads, but not the decoupling of trie traversal from the rest of the loop. Our design successfully boosted the thread occupancy of dynamic programming to around 86%, and the thread occupancy to greater than 70% for two thirds of the application's total runtime. As predicted, the thread occupancy in trie traversal is limited by previous loop stages; it is only about a third.

We also investigated the impact of the number of threads per block and blocks per SM on overall performance. We measured the best performance to come from using 160 threads per block and 6 blocks (30 warps) per SM. While this configuration does not maximize the number of active blocks (16) or active warps (64) per SM supported by the device, our dynamic work mapping technique does ensure high thread occupancy within the active warps present on the device, and optimizing over the thread/block configurations for performance is orthogonal to the strategies explored by WOODSTOCC.

2.5.2 Kernel Stage Decoupling

To probe the effect of stage decoupling, we measured thread occupancy during the third stage of the kernel for more and less aggressive queueing of nodes for trie traversal. Decoupling has the practical effect of growing the size of the main worklist, as it causes more nodes to become simultaneously available for processing. To control queueing, we chose different thresholds for how large this worklist was allowed to grow before traversal was triggered, effectively enforcing a scheduling strategy for the application.

Table 2.4 shows the effect of increasing the threshold from a relatively small value (twice the expected maximum depth of the trie) to a much larger value (20 times this depth). As anticipated, the

Stage	Time (%)	Thread occupancy (%)
Pull from worklist	24	71
DP align	42	86
Trie traversal	34	31

Table 2.3: Relative contribution of each main stage of the kernel to overall runtime, and thread occupancy during each stage measured as the proportion of threads mapped to useful work units. All measurements were taken from aligning a set of 10^5 length-47 reads against human chromosome 1 and averaged across all GPU execution blocks.

occupancy of trie traversal with the more aggressive approach goes to nearly 100%, while the number of distinct calls to traversal drops by two-thirds.

The performance impact of boosting the occupancy of trie traversal from about one-third to 100% is sensitive to the intensity of the traversal stage. An earlier implementation of Woodstocc showed only a 6% reduction in overall runtime. We suspect that because trie traversal is extremely global-memory intensive and exhibits little coalescing of accesses across threads, boosting occupancy did not substantially reduce the number of distinct global memory transactions needed to align all reads to the reference, with correspondingly little overall performance impact. This optimization would likely be more effective for less global-memory intensive kernels.

2.5.3 Overall Performance and Limitations

On an Intel Core i7-950 processor, the best CPU software version of Woodstocc achieves a throughput of approximately 320 reads/sec per CPU core, while the throughput of the software tool BWA (version 0.7.9, modified to find alignments in the forward direction only) achieves throughput of approximately 360 reads/sec, on an input set of 10^5 length-47 reads against human chromosome 1. On the same input set, Woodstocc on an NVIDIA GeForce GTX Titan achieves throughput of approximately 4000 reads/sec – slightly more than a factor of 11 greater than BWA's per-core throughput (see Figure 2.9).

Threshold factor	Stage execution count	Avg thread occupancy
2	93652	34.5%
20	32346	100%

Table 2.4: Stage execution count and average thread occupancy (as a percentage of maximum) during the third kernel stage for two threshold factors. The threshold factor is given as the ratio of the main worklist size to the max possible depth in the search trie. As expected, the third stage is executed far less frequently and its average thread occupancy maximized when the higher threshold factor is used. All measurements were taken from aligning a set of 10^4 length-47 reads against human chromosome 1 and averaged across blocks.

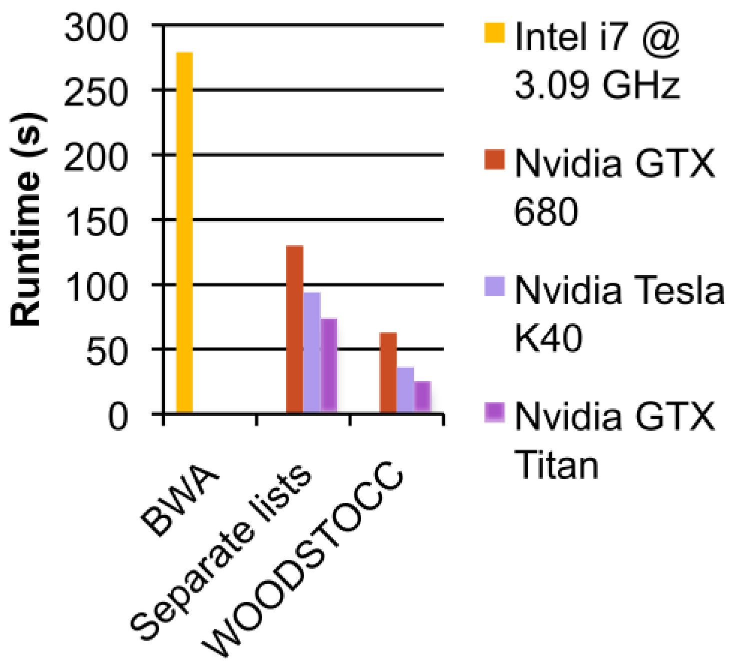


Figure 2.9: Runtime comparison of BWA on a single CPU core and two versions of WOODSTOCC on three NVIDIA GPUs when aligning a dataset of 10^4 length-47 reads against human chromosome 1. The 'Separate lists' version employs a simple data-parallel strategy with independent worklists for each thread, whereas the full version incorporates dynamic thread mapping and kernel slicing.

To isolate the added value of our contributions over a pure data-parallel but task-independent approach, we implemented a version of WOODSTOCC in which threads are mapped to reads 1-to-1 for the duration of the algorithm and each thread stores its own (sequentially processed) worklist. While this version realized approximately a 3× speedup over BWA by itself, adding dynamic thread mapping with the shared worklist further increased performance by 2-3×, and kernel slicing by an additional 10%, for a total speedup of around 6.1×.

Performance of the task-independent version of WOODSTOCC and the full version including worklists and slicing is shown in Figure 2.9. These results show that the benefits of worklist maintenance and slicing far outweigh their overhead, and suggest their general utility as a parallel work management framework versus a pure task-independent framework.

The optimizations of this work were effective in boosting the thread occupancy of all stages of short read alignment, despite the inherent irregularity of the computation. The current bottleneck to further major performance gains is not thread occupancy. The occupancy of the DP stage, which is limited by the need to use reads from a whole number of states, could be further improved with a strategy such as node splitting or bin packing of whole states. However, we anticipate that according to Amdahl's Law the performance improvement from such a change would be marginal, since there is room for only a further 14% increase in occupancy during this stage and it currently accounts for less than half of overall runtime.

2.6 Summary and Relevance

This chapter has presented Woodstocc, a short-read alignment tool for the GPU that extracts regular, SIMD-compatible parallelism from an irregular alignment and trie search computation. Woodstocc employs three strategies– dynamic thread mapping, kernel stage decoupling, and periodic kernel slicing– to boost available GPU parallelism, thus mitigating wavefront irregularity. Examination of our kernel's timing results shows that execution time is dominated by the core algorithmic operations of dynamic programming and trie traversal rather than system overheads, demonstrating the practical benefit of our implementation of these strategies.

Although Woodstocc was built to solve the short-read alignment problem, its key strategies suggest applicability to more general dataflow-based irregular streaming application processing. Indeed, these strategies inspired the creation of our Mercator framework, which comprises a major component of this dissertation. In particular, Mercator generalizes the following aspects of the Woodstocc implementation to reduce wavefront irregularity, transparently supplying them to developers of irregular streaming applications:

- The *worklist* (i.e. queue) for supporting efficient parallel SIMD operations. In Woodstocc, worklists store groups of reads organized by their position in the virutal suffix trie; Mercator queues store input items of user-defined type.

- Efficient worklist *pull* and *push* operations. In Woodstocc these are used for gathering/scattering live reads and their read groups in parallel while maintaining dense storage in the presence of filtering; in Mercator they may be used for gathering/scattering work items in the presence of input-dependent replication as well as filtering.

- A *scheduler* capable of selecting a next module to execute based on application metadata. In Woodstocc the scheduler selects a phase of the algorithm to execute (trie traversal, worklist pull, or DP alignment) based on queue occupancies at each stage; in Mercator the scheduler also selects a module for execution based on dynamic queue statistics.

In addition, future versions of Mercator may generalize the *explosion* and *collection* operations included in the worklist pull/push functionality of Woodstocc to efficiently process irregular application work at multiple granularities. In Woodstocc, this was done to support in the same application both trie traversal work for nodes and DP work for the varying number of live reads they contained. In general, support for multiple granularities could facilitate, for example, processing both vertices and their varying number of incident edges in large-scale graph processing applications.

In Woodstocc, the collection operation was straightforward in the sense that no coordinated work was required over all reads when gathering live reads back into their associated groups. In the general case, however, the collection operation may be nontrivial, requiring a reduction operation over all fine-grained items when they are gathered back into their associated coarse-grained items. For example, in a graph processing application, values from all a vertex's incident edges may need to be compared and reduced (to find, e.g., the minimum value) after one edge-processing step. To be parallelized, such operations would require support for a reduction-by-key on edge values with vertices as keys, which was not required for Woodstocc and is currently not provided by Mercator. Adding support for these reductions will therefore expand Mercator's application scope beyond the functionality requirements of Woodstocc.

Chapter 3

Mercator Overview and Results

We now present the Mercator framework. Mercator builds on the techniques of the Woodstocc application described in the previous chapter, generalizes them for a wider class of applications, and adds a framework to abstract developers from the details of the techniques. Mercator was first published and presented at the Conference on High Performance Computing & Simulation (HPCS) [26].

Mercator is designed to transparently support optimizations applicable to irregular streaming dataflow on wide-SIMD machines, currently NVIDIA GPUs. Given a graph specifying a streaming application's topology and CUDA code 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.

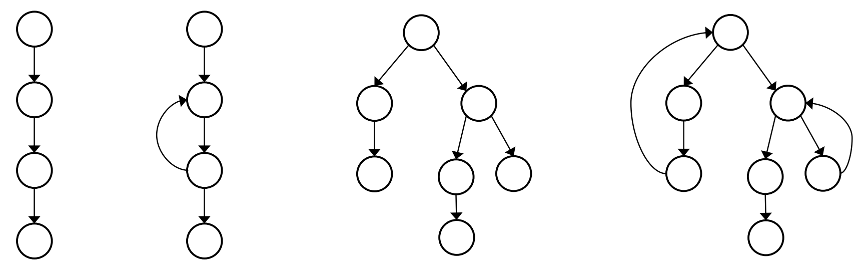


Figure 3.1: Example application topologies, including pipelines and trees with or without back edges.

The rest of the chapter is organized as follows. Section 3.1 describes the Mercator application model, while Section 3.2 extends this model to the GPU. Section 3.3 presents the Mercator framework and its programmer interface. The remaining sections present results and conclusions.

3.1 Mercator Application Model

In this section, we first introduce the application topologies Mercator aims to support. We then 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.

Irregular streaming applications arise in a variety of scientific and engineering computing domains; we list several here, organized by their application topologies. Fig. 3.1 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 [121], Monte Carlo financial simulation [112], and Viola-Jones face detection [128].

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 the recursive algorithm benchmarks vectorized in [104], fixed-point GPU algorithms (e.g. points-to analysis) in the LonestarGPU benchmark suite [20], rejection sampling algorithms such as the ziggurat PRNG [75], pattern-matching algorithms such as NFA processing [103] used for, e.g., network packet inspection and security scanning, and iterative mathematical functions with data-dependent convergence, such as the Mandelbrot set [57].

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 [16]. 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.

We now describe the general model for topology of MERCATOR applications.

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 the remainder of Section 3.1 (only), assume that a node processes inputs one at a time. A node with output channels $c_1 \ldots c_k$ 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.

3.1.1 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 fixed, finite sizes.

If a node N can generate up to k 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 k 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, as described in Section 3.3.2.

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

In fact, a loop of blocked nodes is the *only* way that an application can deadlock, since more complex cycles are not permitted, and (as we argued above) any acyclic chain of blocked nodes terminates on an unblocked node or sink and will unblock after finite time.

We now develop scheduling and queue-size criteria sufficient to ensure freedom from deadlock. In what follows, we assume that an application is free from *livelock*; that is, if a loop $M \rightsquigarrow N \to M$ is not blocked, and the parent W of M does not fire, then after finite time, the total number of queued items in the loop's nodes will decrease (and hence W will eventually be able to fire). Livelock freedom is dependent on the user's code.

Claim: 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 the parent W of M if M has only one free slot in its input queue.*

Proof: The proof proceeds by induction from lower to higher nodes in the application tree.

Let $C = M \rightsquigarrow N \rightarrow M$ be a loop. We have by the IH that no subtree of the application descending from from any node in the loop, other than the loop itself, can deadlock. (In the base case, these subtrees are acyclic.) Moreover, any loops in this subtree are assumed free of livelock, so the head of the subtree (i.e. a node in the loop) cannot block indefinitely on output channels that are not part of loop C. We now argue that C itself cannot have all its nodes simultaneously blocked.

By our scheduling rule, M's input queue becomes full, causing N to block, only if the last item added to this queue came from N. But then N has just fired and consumed an input, creating a free slot in its input queue, so N's parent will become unblocked after finite time. The parent in turn will be able to fire, and so forth, until M's child on C has a free queue slot, and M can fire, unblocking N. Hence, not all nodes in the loop can simultaneously block, and so the loop cannot deadlock. ■

With these restrictions, Mercator can support all the types of application topologies shown in Fig. 3.1. Future work will consider the limits imposed by more complex topologies, such as DAGS and nested loops.

3.2 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.

3.2.1 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 [123] 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.

3.2.2 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.

Further details of MERCATOR's queue management scheme are presented in Chapter 4.

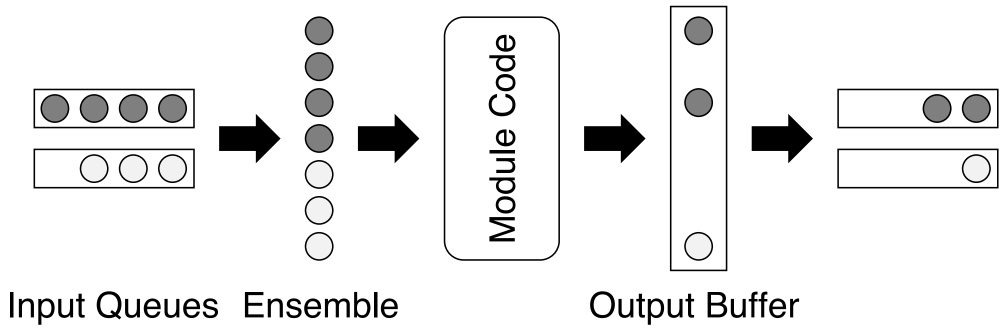


Figure 3.2: Schematic of steps in firing a module, in this case combining two nodes. Items are gathered from the nodes' input queues to form an input ensemble, which is processed by the module's code. Any outputs produced by the module are stored in the output buffer, from which they are scattered to their downstream queues. Shading denotes tags indicating the node associated with each item.

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 [128] might consist of a chain of nodes, each implementing the 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). Figure 3.3 shows a sample dataflow graph of nodes, some of which share a module type.

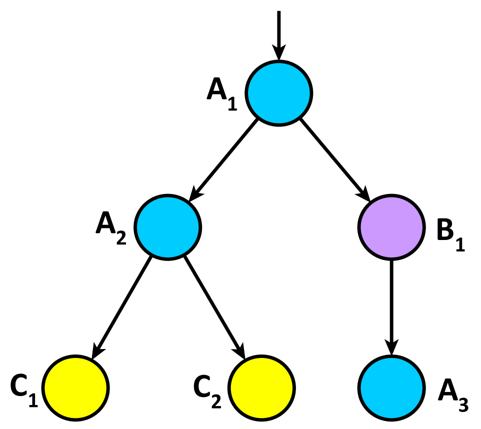


Figure 3.3: Dataflow graph for an example MERCATOR application. Here nodes of the same label and color share a module type, with label subscripts indicating instance indices for nodes of the same module type.

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. 3.2 illustrates this process.

Parallel Gather and Scatter Across Queues. To support efficient concurrent execution, Mercator must be able to efficiently gather inputs from multiple queues and scatter outputs on a channel to multiple queues. We developed fast parallel implementations of these gather and scatter operations tuned to the properties of our SIMD platform. In what follows, we refer to the *warp size* of the GPU, which is the number of threads that actually execute their instructions simultaneously as single wide vector operations. Threads within a single warp can exchange data via efficient register "shuffle" operations without touching memory, and they may be assumed to operate synchronously.

Gathering inputs from multiple queues entails computing two things for each thread of the input ensemble: the queue from which it should read an input item, and the index of that item within its queue. Suppose we have an array A such that $A[i]$ is the number of items to be read from queue i. We first compute $\tilde{A}$ such that $\tilde{A}[i] = \sum_{k=0}^{i-1} A[k]$. The jth thread of the input ensemble then performs a binary search on $\tilde{A}$ to determine the least i_j such that $j \geq A[i_j]$; thread j then reads item $j - A[i_j]$ from queue i_j. Figure 3.4 shows an example of the gather operation.

The exclusive sum can be performed with a parallel scan, while the binary search can be done concurrently without divergent branching in each thread [17]. We limit the number of concurrently executed instances per module *a priori* to the size of a single warp (32 on current NVIDIA GPUs), so the above index arithmetic can be done independently by each warp in a block, using only intra-warp register shuffles and requiring no memory other than the initial read of A.

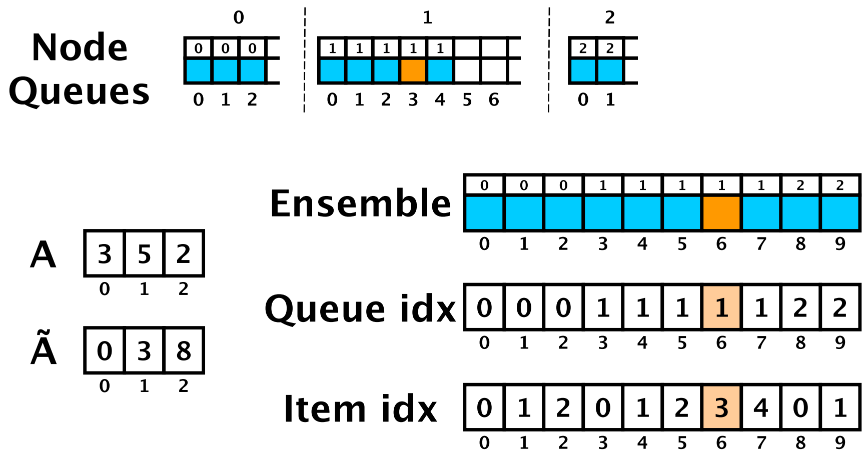


Figure 3.4: Snapshot of a gather calculation for an ensemble of size 10 on a set of three node queues. Input items are marked with their instance (node) tags. Each ensemble thread calculates the queue and item indices of its target input item using the $\tilde{A}$ array as described in the text, then fetches the item from the calculated queue position. Indices calculated for ensemble thread 6 and the target item found at those indices are highlighted in orange.

Scattering a channel's outputs to queues requires slightly more coordination across a block. Each thread reads one entry from the channel's output buffer. The threads of each warp w independently sort their entries by tag and count the number $N_w[i]$ of items with each distinct tag i. Again, the number of instances per module is limited to the warp size, so the array N_w can be computed and stored within the registers of the warp. The warps then coordinate to compute for each warp w $\tilde{N}_w$, such that $\tilde{N}_w[i] = \sum_{k=0}^{w-1} N_k[i]$. Finally, the thread holding the jth item with tag i in warp w looks up the queue $q(i)$ that should receive items with this tag and writes it to entry $\tilde{N}_w[i] + j$ in this queue. Figure 3.5 shows an example of the scatter operation.

The computation of $\tilde{N}_w$ for each warp can be done by concurrent SIMD parallel scans, while all other index computations can be done within one warp. We implement bitonic sort with CUDA shuffle instructions in each warp to accomplish intrawarp sorting and counting without using shared or global memory, while coordinated computation of $\tilde{N}_w$ for up to 1024 threads requires one 16-bit value in shared memory per warp per instance of the module. The mapping q from input tag to output queue uses shared memory proportional to the number of instances of the module.

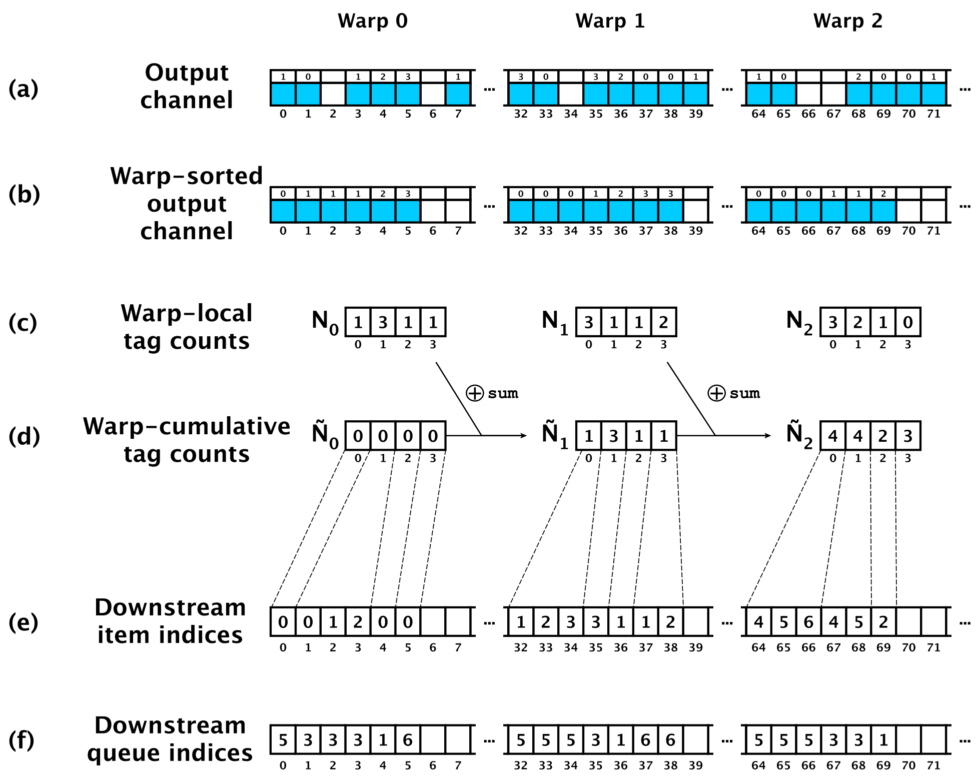


Figure 3.5: Snapshot of three warps' worth of a scatter calculation for an output channel of a module with four nodes. Steps (a)-(f) in the scatter calculation are described individually. (a) Output items in the output channel are marked with their instance (node) tags. It is assumed that the only output items in the channel are the ones shown here so that operations across warps are transparent. (b) Output items are sorted by instance tag within each warp. (c) Counts of output items with each instance tag are tallied by each warp. (d) Cumulative counts of output items with each instance tag are tallied across warps. (e) Indices of output items into their downstream queues are calculated based on the cumulative tag counts and the rank by tag of each item within its warp. Downstream queues are assumed to be initially empty, else an occupancy offset for each queue would be added to the downstream item indices. (f) Downstream queue indices for each output item are looked up in the application topology based on the items' node tags. The queue index values shown here are arbitrary but internally consistent; i.e., outputs from the same node have the same downstream queue index.

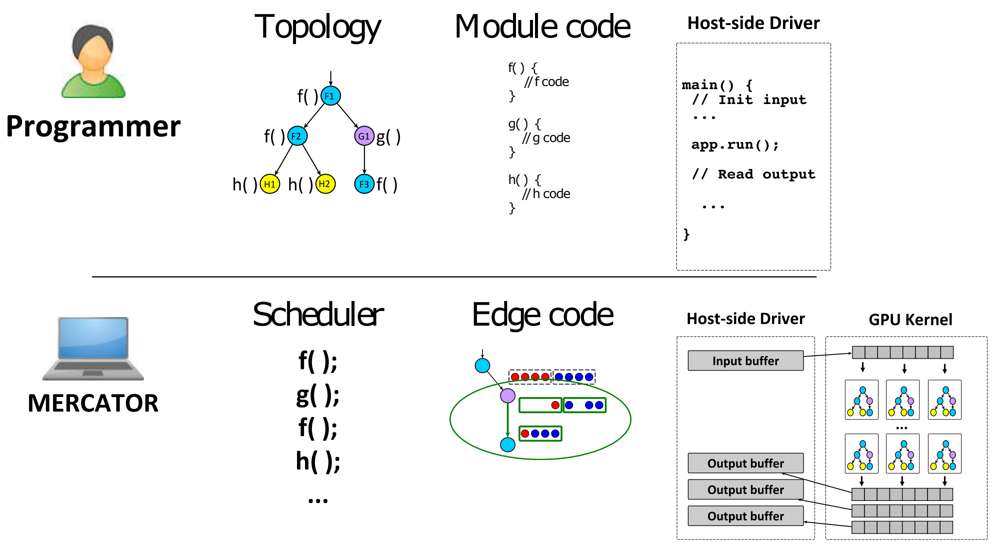


Figure 3.6: Major components of the MERCATOR framework. The programmer provides a topology specification, code for each module function, and a host-side driver file to interact with the MERCATOR API. MERCATOR provides a scheduler, code to accomplish data movement along edges, and all other infrastructure necessary to realize the application as a CUDA kernel.

3.3 System Implementation

MERCATOR's implementation comprises a developer interface and infrastructure code. Figure 3.6 shows the key constituents of these two components.

3.3.1 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. 3.7 illustrates the workflow associated with these tasks. Details of the developer interface and MERCATOR API may be found in our system's user manual [24], which is included in this dissertation as Appendix A.

Application topology is specified declaratively, as illustrated in Listing 3.1. The most important declaration, `module`, specifies a module type. Each named module type has a *signature* that specifies

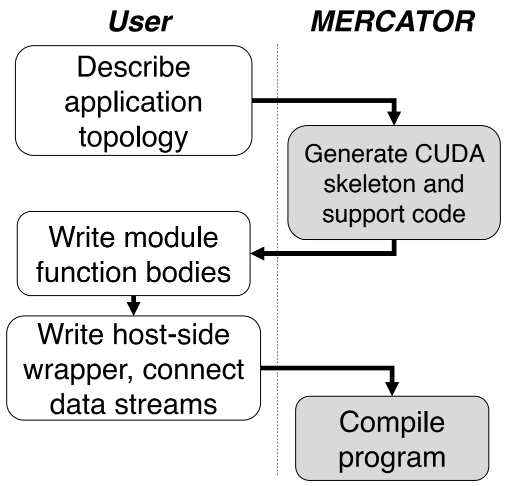

User
MERCATOR
Describe application topology
Generate CUDA skeleton and support code
Write module function bodies
Write host-side wrapper, connect data streams
Compile program

Figure 3.7: 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.

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 specification 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, module types, and the application as a whole may have associated parameter data that is initialized on the host and made available to the GPU at runtime.

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

Listing 3.1: 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.

```
//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
```

the same module type may access node-specific data using arrays indexed by tag. Module-specific and global application data may be accessed directly.

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.

3.3.2 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 [99] first parses the developer's app spec file and extracts the parameters for each module type, node, and queue from the `module`,

```
void markCycles()  {
  // start recursive process
  markCyclesRec(sourceNode);
}

void markCyclesRec(ModuleInstance* node)
{
  // mark node as visited (preorder)
  node->set_visited(true);

  // loop over children
  for (each child of node)
  {
    if (child->is_visited())  // edge to child is a back edge
    {
      // set cycle indicators
      node->set_isCycleTail();
      child->set_isCycleHead();
      e->set_isBackEdge();  // e is edge from node to child
    }
    else // edge is not a back edge
    {
      // recursively process child
      markCyclesRec(child);

      // mark this node as predecessor if child
      //   is a cycle head node (but not along a back edge)
      if (child->isCycleHead())
        node->set_isCyclePredecessor();
    }
  }
}
```

Figure 3.8: Pseudocode of application graph traversal to mark cycle components: heads, tails, head predecessors, and back edges.

```
void verifyCycles()  {
  // start recursive process
  verifyCyclesRec(sourceNode);
}

bool verifyCyclesRec(ModuleInstance* node)
{
  // indicator for whether cycle is pending in traversal
  bool inCycle = false;

  // loop over children
  for (each child of node)
  {
    // recursively process child
    bool childInCycle;
    if (child was not reached from back edge)
      childInCycle = verifyCyclesRec(child);

    // if child passes on a pending cycle to this (parent) node
    if (childInCycle)
    {
      // if node was already in a pending cycle, FAIL
      if (inCycle)
        // FAIL
      else
      {
        // visit: mark node as being in a pending cycle now
        inCycle = true;

        // check production rate of edge along cycle; must be <= 1
        if (e->productionRate > 1) // e is edge to child
          // FAIL
      }
    }
  }  // end loop over children

  // visit: account for node being head or tail of cycle
  if (node->isCycleHead())  // head node closes the cycle
    inCycle = false;

  if (node->isCycleTail())  // tail node opens a cycle
  {
    if (inCycle)  // can't open a cycle if one is already pending
      // FAIL
    else
      inCycle = true;
  }
}
```

Figure 3.9: Pseudocode of postorder graph traversal to verify cycle validity according to MERCATOR's topology constraints: overlapping and nested cycles are not permitted, and all production rates along a cycle's edges must be ≤ 1. Overlapping or nested cycles are implicated if a node participates in more than one pending cycle in a bottom-up traversal of the graph. Note that a single node acting as head of one cycle and tail of another is permitted, as its head cycle is closed before its tail cycle is opened.

`node`, and `edge` declarations respectively. The front end then infers the application topology and checks for type compatibility and conformance to Mercator's topological constraints. Specifically, each output edge is checked to ensure its type is identical to the input type of its downstream node; node and edge declarations are checked to ensure that any referenced topology objects therein have been validly declared elsewhere; and cycles are detected and validated. Cycle validation consists of ensuring the three conditions necessary for deadlock avoidance introduced in Section 3.1.1: cycle head nodes are marked as requiring sufficient queue space to prevent deadlock; cycle nodes are checked to ensure all production rates along the cycle are ≤ 1; and application topology is examined to ensure that no nested or overlapping cycles are present. These type and topology checks are performed using linear-time traversals of the application's dataflow graph (see Figures 3.8 and 3.9). 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 specification 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.

Queue sizing Queue storage capacities for module instances are based on production rates of upstream nodes. In principle, every queue could be sized as small as 1 slot with the exception of cycle head nodes, which require at least 2 slots for deadlock avoidance. To facilitate SIMD parallelism, however, a queue sizing strategy should allow at least one full ensemble's worth of items to be fired by each node without being blocked by a downstream queue. Nonuniform production rates across nodes prevent minimum queue sizes from being constant throughout the dataflow graph; for example, if a node produces two outputs per input, the queues of its downstream node(s) must be large enough to accommodate two full ensembles of items to allow a single full-ensemble firing.

The queue space required by each node to accommodate a full-ensemble upstream firing may be calculated with one linear graph traversal starting from the source node. At each step, the local production rate of a node is multiplied by the cumulative production rate of all prior nodes on its path from the source. This new cumulative production rate is stored in order to be accessed by downstream nodes, then multiplied by the size of an ensemble to obtain the final queue size for the node. To ensure sufficient queue space for head nodes of cycles to avoid deadlock, the production rate of their upstream nodes is considered to be the maximum of their non-cycle predecessor's production rate and the cycle tail node's rate.

To increase the likelihood of being able to fire one or more full ensembles' worth of items, we heuristically scale each queue size from the SIMD-minimum described above by some small integer K (currently 4). In general, this strategy allows K full ensembles of items to propagate through the graph starting at the source, with space for at least K non-blocked full ensemble's worth of items to fire at each node. As an alternative heuristic, queue sizes could be increased individually based on production rates and degrees of irregularity of nearby nodes.

3.4 Experimental Evaluation

3.4.1 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.

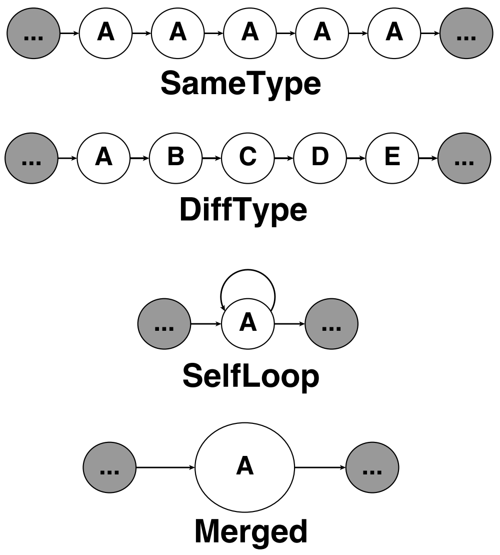


Figure 3.10: Topologies of single-pipeline synthetic applications. Label inside each node indicates its module type. Gray nodes are sources/sinks.

Synthetic Applications We implemented two sets of synthetic applications, shown in Figs. 3.10 and 3.11. 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 [88]; 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.

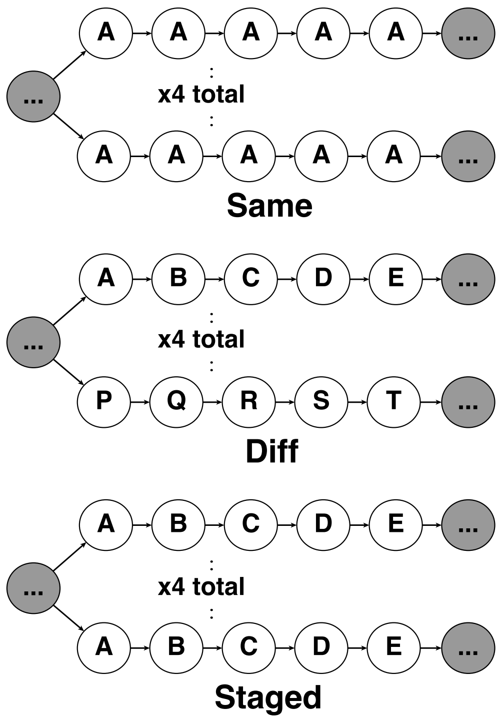


Figure 3.11: Topologies of multi-pipeline synthetic applications. Node labels are as in Fig. 3.10.

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 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.

BLASTN Sequence Comparison Pipeline Our "BLASTN" application implements the ungapped kernel of the NCBI BLAST algorithm for sequence comparison [3]. 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.

3.4.2 Results and Discussion

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 Figure 3.12. The workload used in these experiments was a Black-Scholes financial simulation representative of compute-bound algorithms. To abstract our results from the particular workload chosen, workloads in Figure 3.12 are expressed in terms of microseconds of execution time per input ensemble per compute node when run on a baseline input set having one item per GPU thread. In practice the time required to process each input ensemble remains constant up to the number of ensembles in a GPU blocks' worth of threads, then increases with input stream size until it plateaus at slightly under $2\times$ the baseline time for full-scale input sets ($1.88\times$ for 32K inputs, $1.95\times$ for 256K+ inputs).

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 its overhead for workloads requiring at least 40 μs of execution time per node firing. In some cases, the threshold to see a benefit from MERCATOR was much lower (e.g. < 8 μs for a filtering rate of 0.5).

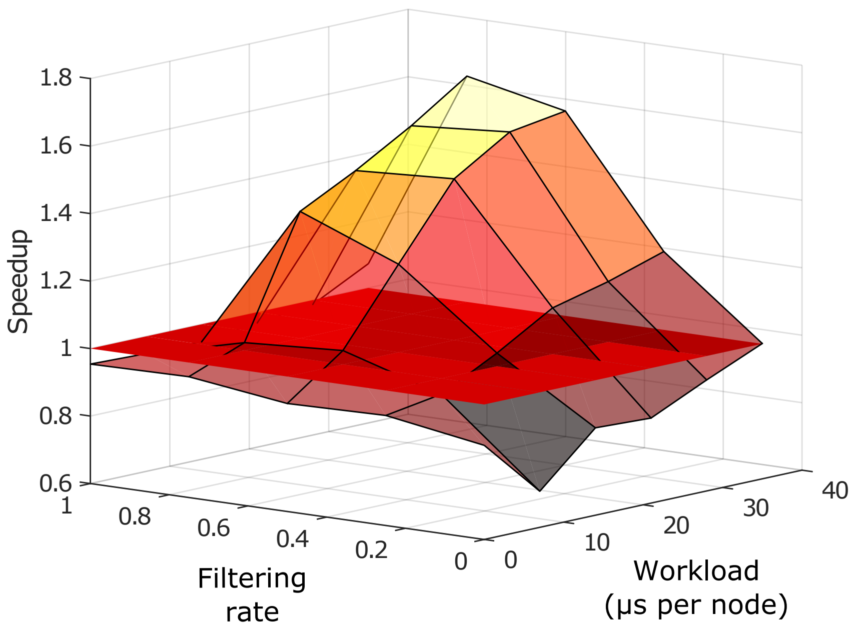


Figure 3.12: Speedup of `DiffType` over `Merged` topology for varying filtering rates and workloads. Filtering rates shown were applied to each stage of the application; rate 0 discards no inputs, while rate 1 discards all inputs. Workloads are shown in terms of approximate average microseconds of execution time per input ensemble per compute node when run with a baseline input set containing a single item per GPU thread. Black-Scholes financial simulation was the algorithm used in the application, with workloads ranging from 0 to 100K simulation rounds per compute node. Wherever the speedup is greater than 1 (i.e. above the plane), the benefits of Mercator outweigh its costs for this application.

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 [20] and recursive [104] algorithms.

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× 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. 3.13 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×, 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 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. Occupancy measurements confirm this hypothesis for all intermediate filtering rates; for example, with a filtering rate of 0.5 the average thread occupancy per firing of all working modules in `Diff` was 0.50, compared to 0.85 for the single working module in `Same`. The `Staged` application was comparable in execution time to the `Same` application to within 3%, suggesting that horizontal concurrency across pipelines (as in `Staged`),

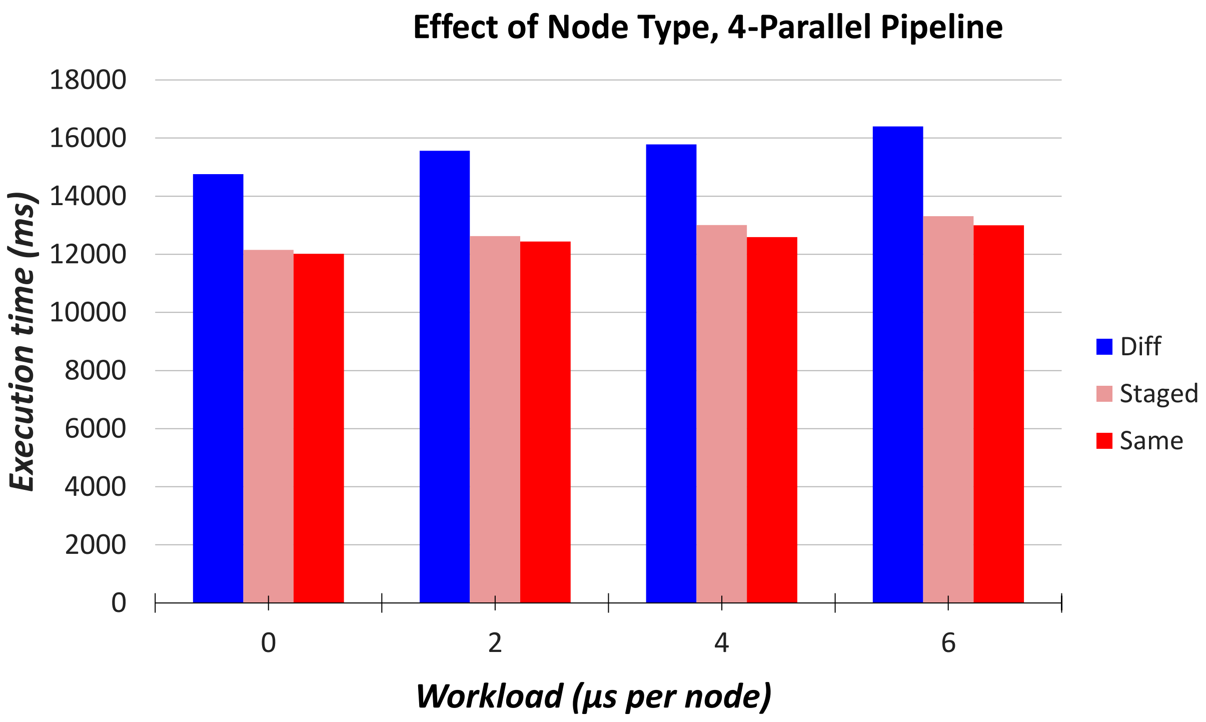


Figure 3.13: Execution time of 10^6 inputs streamed through the parallel pipeline applications in Fig. 3.11 under various workloads with a per-node filtering rate of 0.5. Workloads, which range from 0 to 60K iterations of Black-Scholes financial simulation, are shown in terms of approximate execution time per ensemble per node of the `Staged` application when run with a baseline input set containing one input item per GPU thread.

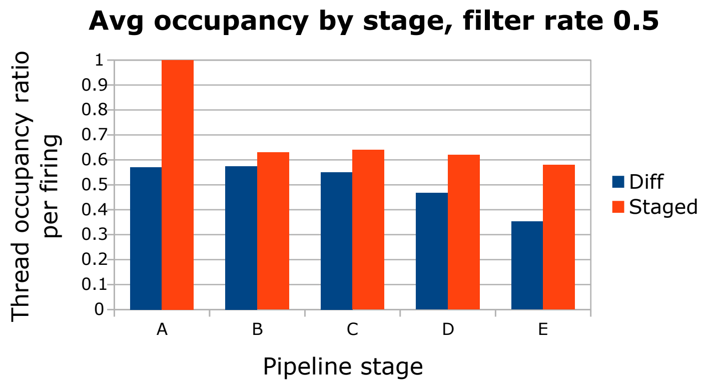


Figure 3.14: Average thread occupancy per firing of modules at different pipeline stages for `Staged` vs. `Diff` using a per-node filtering rate of 0.5. The thread occupancy of `Staged` is higher at all stages, enabling higher throughput.

rather than vertical concurrency within each pipeline, conferred most of the benefit. Figure 3.14 shows thread occupancies per pipeline stage for the `Diff` and `Staged` applications.

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

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. 3.15 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× to 1.78× 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. Figure 3.16 shows the per-stage occupancies of MERCATOR

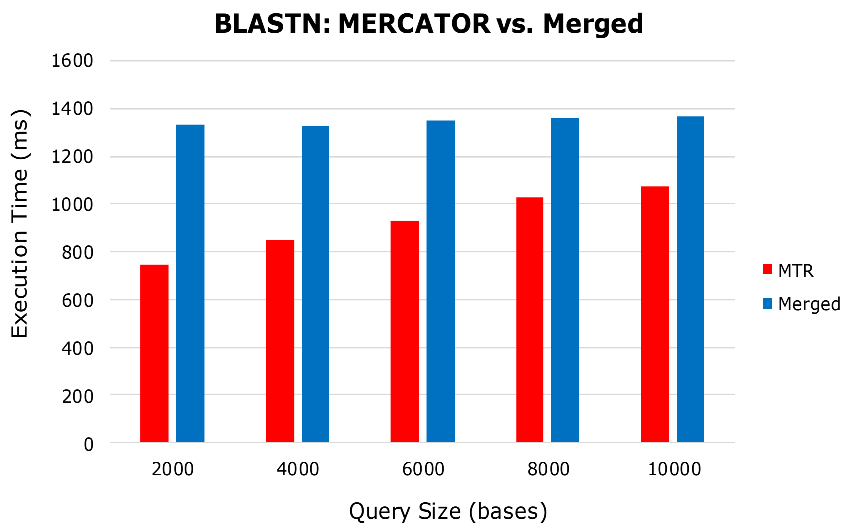


Figure 3.15: Execution time of BLASTN application comparing a chicken genome fragment (query) against human chromosome 1 (database) using MERCATOR vs. the merged topology.

BLASTN and the virtual occupancies of the Merged implementation, measured as the ratio of live inputs to total inputs at each stage.

3.5 Conclusion

In this chapter we have described our MERCATOR framework for 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.

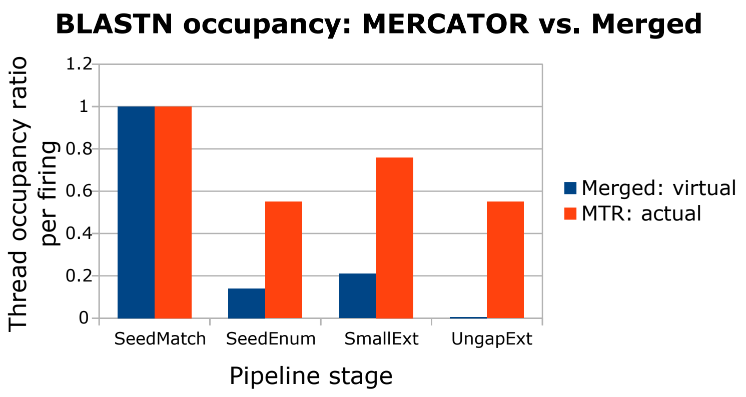


Figure 3.16: Average thread occupancy at all pipeline stages of BLASTN for MERCATOR and `Merged` implementations using a query size of 10 Kbases, which produced the highest occupancies for the Merged implementation. Per-module thread reassignment allows MERCATOR to maintain high occupancy at all stages, whereas early-stage filtering leaves most threads idle in later stages of Merged.

Our description of the MERCATOR system implementation in this chapter was deliberately brief in order to present the implementation in the context of the overall framework. In reality the MERCATOR implementation contains crucial components that must overcome significant SIMD-programming challenges and present interesting optimization opportunities. In order to avoid a performance bottleneck from these components, they must have efficient SIMD-friendly implementations, and to maximize the performance as well as the functionality of the system, the optimization opportunities must be exploited.

One crucial component is the queue management system. Many overlapping concerns inform our queue management strategy: the ability to merge queues of nodes of the same module type for increased occupancy; the need for efficient SIMD 'pull' and 'push' operations, including pushes involving parallel compaction to regularize input-dependent node outputs along dataflow edges; the need to avoid deadlock; the need for efficient interaction with other system components such as the programmer interface and the scheduler.

Closely related to the queue management strategy because of their close interaction is the scheduler. The scheduler selects a module to fire at each scheduling step, which we assume for now to occur at the granularity of a single module firing. Any nontrivial scheduler must make scheduling decisions based on queue states; for example, the scheduler should not schedule an empty queue for execution. In order to avoid becoming a performance bottleneck, then, the scheduler must interface with queues in a SIMD-efficient way when making scheduling decisions no matter what specific (nontrivial) scheduling strategy is employed.

In a deadlock-free system, any valid scheduling strategy– i.e., one that respects queue capacities and chooses a feasible nonempty module to fire at each firing step if such a module exists– is guaranteed to run a given set of inputs to completion. However, to maximize performance, the scheduler should choose a firing order that is efficient in addition to being valid. The Mercator execution model therefore presents an optimization opportunity in the scheduling strategy.

In the next chapter we will consider the challenges of queue management and scheduling strategies in the Mercator system in more detail.

Chapter 4

Mercator Dataflow Model and Scheduler

In the previous chapter we presented our Mercator framework for irregular streaming dataflow application development on GPUs. In this chapter we explore the challenges of queue management and scheduling strategies in the Mercator system in more detail. We first establish the theoretical foundations of our approach to these challenges by describing a Model of Computation (MoC) and execution, the Function-Centric (FC) model, that is more suited to a wide-SIMD multiprocessor platform than previous models. We then present our queue management strategy derived from the FC model and the scheduler operations that interface with these queues. Finally, we discuss scheduling in systems consistent with the FC model and present a scheduling strategy for Mercator that guides execution towards maximum SIMD utilization based on these insights. In practice this strategy boosts Mercator utilization to near-maximal for the benchmark applications considered in Chapter 3 over the naive strategy used for the experiments presented in that chapter.

The FC model we describe next in Sections 4.1–4.2 was first published and presented at the Data Flow Models for Extreme-Scale Computing (DFM) workshop held in conjunction with the Parallel Architectures and Compiler Techniques (PACT) conference [27].

4.1 Model Introduction / Background

To provide guidance to application designers, parallel platforms are often abstracted through prescriptive *parallel execution models* (PXMs), which propose a structured execution strategy conducive to achieving high performance. The closer the match between a PXM and the underlying architecture on which it is realized, the greater its potential to guide developers to high-performing implementations.

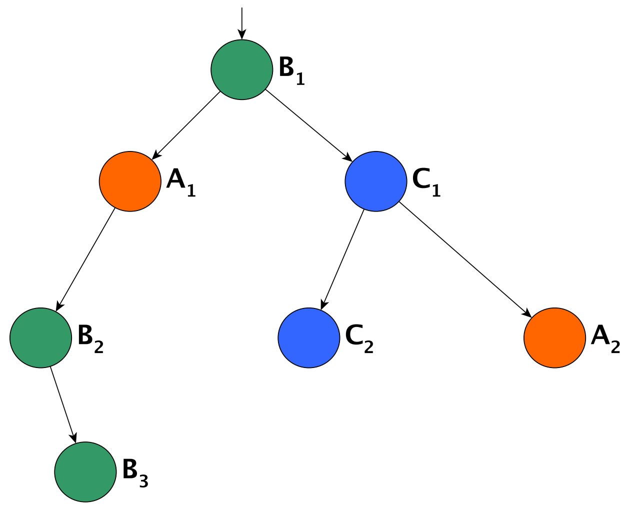
We introduce in these sections the Function-Centric (FC) PXM for mapping streaming computation to wide-SIMD architectures. The FC model keeps the data streams in a computation partitioned *by the function to be applied to the items* throughout an application's execution. This partitioning exposes the maximum potential for the processor to fill SIMD lanes at each execution step. The FC model contrasts with existing streaming computing models, which either partition data more finely and therefore isolate potentially SIMD-executable inputs from each other, or queue data for different computations together and therefore compromise SIMD execution.

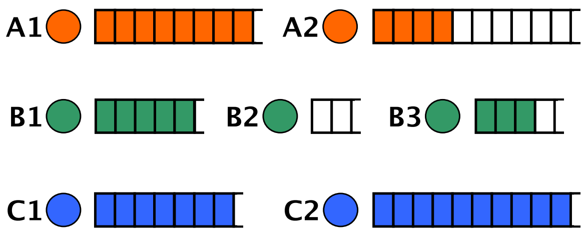
4.1.1 Example Target Platform: GPUs

To make the FC Model concrete, we consider execution on NVIDIA GPUs programmed in CUDA as representative of the broader class of wide-SIMD architectures for which our FC model is appropriate. We refer the reader to Section 1.1 for a brief review of the relevant aspects of a GPU's architecture and execution.

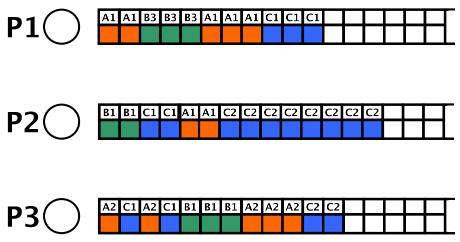
4.1.2 Related Work

Certain classes of streaming application have been optimized for SIMD processing (e.g., tree-traversal algorithms [103] and polyhedral-friendly algorithms [56]). However, to the best of our



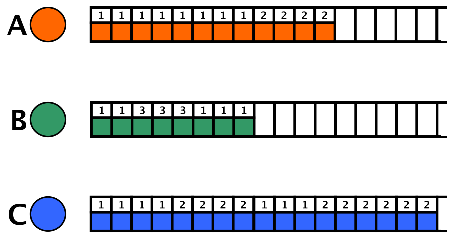


(a) **Data Process Network (DPN) models**. Each function instance (node) has its own worklist holding data to be processed by that node.



P1
A1 A1 B3 B3 B3 A1 A1 A1 C1 C1 C1
P2
B1 B1 C1 C1 A1 A1 C2 C2 C2 C2 C2 C2 C2 C2
P3
A2 C1 A2 C1 B1 B1 B1 A2 A2 A2 C2 C2

(b) **Shared Worklist (SW) models**. Universally accessible per-processor worklists combine input data from all nodes. Each data item on each worklist is tagged with a function ID and its target node's index.



(c) **Function-Centric (FC) model**. Each distinct function (i.e., module type) has its own worklist holding only data to be processed by instances of that module type. Each data item on the worklist is tagged with its target node's index.

Figure 4.1: Dataflow graph for a streaming application composed of three functions (A, B, C), each instantiated at multiple nodes, and work queueing strategies under three streaming PXMs. The FC model resembles a modified DPN model in which worklists for nodes representing the same function have been merged.

knowledge, no general prescriptive PXM has been proposed targeting streaming SIMD computation. Although dozens of streaming applications have been ported to GPU architectures (see [48] for a sample), few implementations follow a general PXM, and the optimizations they use to achieve high throughput are therefore application-specific. We mention general frameworks that do conform to an existing PXM below.

Existing PXMs for streaming processing belong to one of two broad categories based on their work queueing schemes: Dataflow Process Network models and Shared Worklist models.

Dataflow Process Network (DPN) models Dataflow Process Networks [63] encompass both Kahn Process Networks [53] and Dataflow Models (e.g. [62]). In a DPN, as shown in Figure 4.1a, each edge is realized as a typed FIFO queue storing work to be performed by its downstream node, and nodes communicate only via their queues. Modular computing frameworks such as Auto-Pipe [41], RaftLib [10], and Ptolemy [36] are DPN-based, as are the GPU-based StreamIt frameworks presented in [124] and [46].

Shared Worklist (SW) models In contrast to the well-defined topology and typed queues of a DPN, the Shared Worklist model for general task-based execution, illustrated in Figure 4.1b, does not distinguish work by graph topology or data type. Instead, individual "tasks," each of which encapsulates a particular function to be applied to particular data, are placed on a global worklist or per-processor worklists. Strategies such as work stealing and work donation [15] are used to balance computation across resources. GPU frameworks such as those in [122] and [11] are based on the Shared Worklist model.

The Codelet PXM [115, 135] contains components that operate under each of the DPN and SW models. Its *codelets* are event-driven and atomically scheduled, as for nodes in the DPN model, while its *Threaded Procedures* (TPs) pull coarse-grained tasks from a common worklist and employ work-stealing techniques, as in the SW model.

Inadequacy of Existing PXMs While both DPN-based and Shared Worklist PXMs can be used to structure applications whose nodes internally implement SIMD computations, the models themselves do not optimally group data items that could be processed in parallel as a SIMD ensemble. A DPN system maintains separate, isolated queues even for nodes that execute identical code on their input streams. In contrast, an SW system mixes data requiring different forms of processing in a common worklist. We seek an ideal middle ground: a model that keeps separate those data items requiring different kinds of processing yet groups items requiring identical processing, thereby making it easier to form ensembles of items that can fill the lanes of a wide-SIMD architecture.

4.2 Function-Centric (FC) Model

To address the shortcomings of existing streaming PXMs on wide-SIMD architectures, we propose the *Function-Centric* or FC model. By organizing an application's pending work into a form that exposes maximum SIMD-friendly parallelism, the FC model automatically capitalizes on opportunities for wide-SIMD execution in a way that models targeting more general multiprocessor architectures do not.

4.2.1 Structure

The FC model abstracts a modular streaming application program as a dataflow graph whose components are defined as follows. These components were first introduced in Section 3.1 in the context of Mercator's execution model, which is based on the FC model.

(1) Each node (*module instance*) in the application's DFG represents some program function (*module* or *module type*) that operates on streaming data. Multiple instances of the same module type exist in a DFG if multiple nodes execute an identical function on their inputs. Modules may be parametrized; for example, a filtering module may have many instances that implement the same filter but with different threshold values. In such cases, the functional code defining the module type

is still constant across all instances of that type, while the instance to which each item is targeted is passed as a data parameter to the module type's code.

(2) Each edge in the application's DFG represents a buffer for data inputs awaiting execution by its downstream node. Crucially, in the FC model the input buffers of all instances of a given module type are realized as a single worklist. Because all instances execute the same stream of instructions on their inputs, each worklist is composed of exactly those items that require identical processing and hence can be processed in parallel in SIMD fashion!

Figure 4.1c illustrates the FC model's worklist strategy, while Table 4.1 highlights key differences between the FC, DPN, and SW models.

Table 4.1: Characteristic features of different PXMs.

Feature	**DPN**	**SW**	**FC**
Worklist scope	module instance	global or processor	module type
Worklist composition	function-specific	mixed-function	function-specific
Communication	local	global	global
Typed worklist?	yes	no	yes

4.2.2 Execution

In a GPU application organized after the FC model, a block runs the application on its input stream by executing a series of *module firing* steps. A single firing first chooses a module type having at least one input item pending execution by one of its instances. All warps in the block then pull data items from that module type's worklist, forming an ensemble of items requiring identical processing. This ensemble is then processed in SIMD fashion by the module type's code. A firing may generate outputs that are placed on one or more other worklists for processing by later firings.

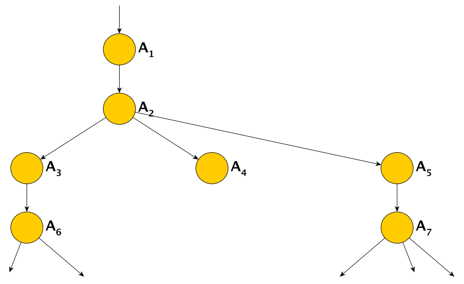
Execution of a full application is simply a sequence of firings, each of which processes one or more pending items from some module's worklist, until the application terminates. To preclude situations in which some firing sequences may lead to deadlock with bounded worklist storage, the FC model restricts dynamic data-rate applications to those whose DFG topologies contain only single-input nodes (i.e. trees). General graph topologies can be supported safely if all modules' data production rates are fixed, as in Synchronous Data Flow (SDF) applications.

A performance-critical consequence of the FC model is that parallel SIMD execution of DFG nodes is implicitly contained in the module firings, since inputs to multiple DFG nodes of the same module type are processed simultaneously during a firing. In other words, the model automatically induces SIMD execution even without explicit coding by the application programmer. Indeed, a module's code could in principle be specified as a single-threaded function that is transparently replicated across all lanes of a SIMD processor. The dynamic scheduling capability of the FC model affords it more flexibility than a classical event-driven Data Flow execution model: an FC scheduler *may* always choose to fire modules as soon as their inputs are available, preserving the integrity of Synchronous Data Flow behavior; however, it may also choose to wait to fire an module until a certain number of inputs have been queued on its worklist in order to maximally fill SIMD lanes upon firing, increasing execution parallelism over that realized by an event-driven model. We consider a scheduling strategy that employs this technique in Section 4.5.

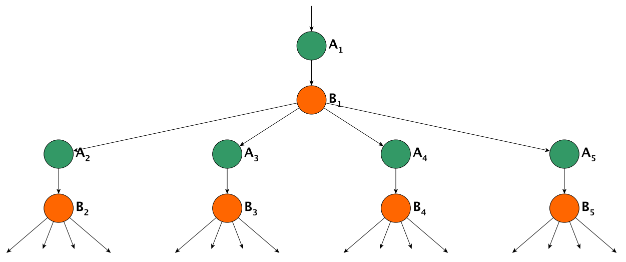
4.2.3 Example Applications

To demonstrate the utility of the FC model, we now show how two high-impact streaming applications may be implemented in conformity to the model. DFGs of these applications are shown in Figure 4.2.

In *random-forest evaluation* (Figure 4.2a), input items (feature vectors) are streamed through a filter cascade. At each level of the cascade, a critical value is computed from an input item's data and compared to a threshold; the result of this comparison determines the output path of the item. Each node of the filter cascade is a module instance parametrized by its threshold value, with nodes representing the same discriminator function having the same module type. When implemented in conformity to the FC model, all nodes of the same module type will share a worklist, and inputs to all of these nodes can be processed in SIMD fashion.



(a) **Random forest classifier**. Classifier-specific topology with module instances operating on items. Since all instances are of the same module type, the FC model merges data inputs to all nodes into a single worklist, enabling every module firing to execute with the maximum possible SIMD width.



(b) **Woodstocc**. Alternating layers of module instances of two different types implement dynamic-programming and tree-traversal calculations, respectively. In this case, the FC model creates two separate worklists, one for each type, and each module firing processes work from one of the two worklists in SIMD.

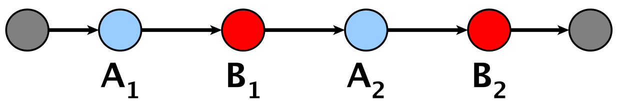
Figure 4.2: Dataflow graphs of example applications. Only the top few levels of the graphs are shown; in practice, they may extend to many more levels depending on the inputs to be processed.

A second application is *DNA short-reads alignment* (Figure 4.2b), which performs approximate string matching of many short DNA strings against a common long reference string. The search is performed one character at a time, first computing a row of a dynamic programming matrix for each successive reference character, then computing the next character to align based on a virtual tree traversal. The two functions of dynamic programming calculation and tree traversal are the module types of the application. Details of these two functions may be found in Section 2.3.1.

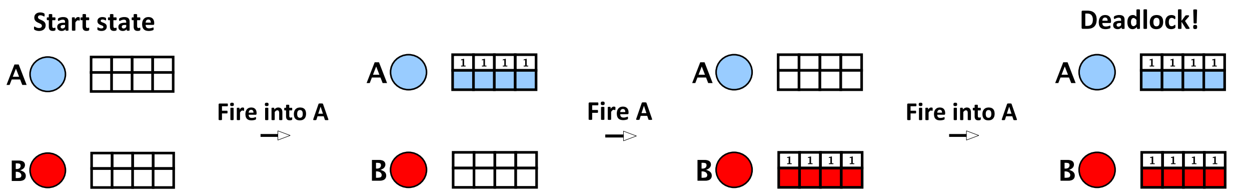
Our Woodstocc application [25] implements short-reads alignments in conformity to the FC model, and it therefore contains one worklist corresponding to each function type. Empirical testing on NVIDIA GPUs [28] revealed a performance improvement of $> 50\%$ with the introduction of these worklists, which are themselves managed using SIMD operations, compared to a sequential data management scheme.

4.3 Mercator and the FC Model

Since Mercator is designed to run on a wide-SIMD architecture, we base its data storage and execution strategy on the FC model to achieve high utilization. Mercator's queue management strategy exploits the FC model's crucial insight that aggregating module instance inputs by module type implicitly exposes maximum SIMD parallelism. However, Mercator's implementation does not precisely follow the abstraction of the FC model due to its inherent potential for deadlock and its lack of topology support for cycles. We discuss these concerns and reveal them to be identical in the next subsection.



(a) Example DFG.

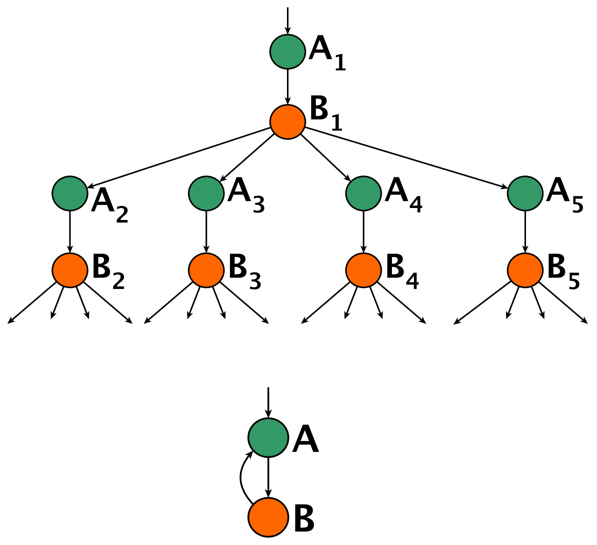


(b) Execution trace leading to deadlock.

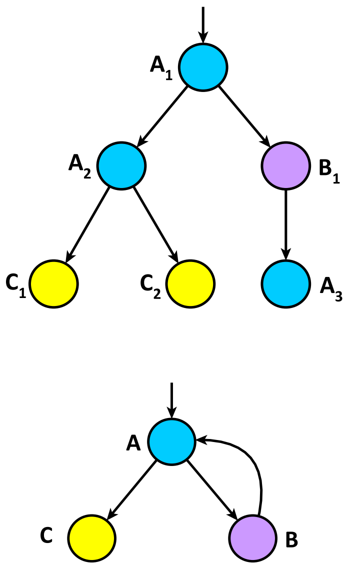
Figure 4.3: Example DFG and execution trace leading to deadlock, with queues shown in the same format as in Figure 4.1. In the deadlocked state, none of the inputs at $\mathbf{B_1}$ can fire since **A**'s queue is full, but none of the inputs at $\mathbf{A_1}$ can fire since **B**'s queue is full. The ability of the upstream node $\mathbf{A_1}$ to effectively block a downstream node $\mathbf{B_1}$ exposes a latent cycle in application dataflow.

4.3.1 Merged DFGs and Deadlock Potential in the FC Model

As it is presented in Section 4.2, the FC model contains the potential for deadlock if implemented precisely. To see why, consider the FC model execution trace of a particular DFG shown in Figure 4.3. Merging queues by module instance in an application merges their corresponding nodes in the application's dataflow graph. This induces a *merged DFG* on the graph that represents the graph's topology with respect to dataflow between queues (see Figure 4.4 for examples). If any two instances of a module type share a common path from the root, then those instances are part of a



(a) **Woodstocc**. Sample application from Figure 4.2b.



(b) **Synthetic application**.

Figure 4.4: Merged (bottom) and original (top) dataflow graphs of sample applications.

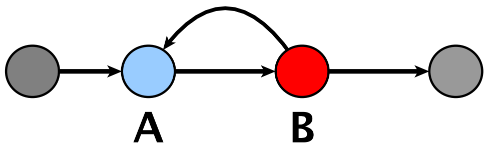


Figure 4.5: Merged DFG for the application shown in Figure 4.3. The cycle in the merged DFG, which reveals true application dataflow, implies potential for deadlock.

common cycle in the merged DFG, implying that a dataflow cycle exists in the graph. Since dataflow cycles imply deadlock potential without special management, module instances on a common path introduce the possibility of deadlock in the FC model. Figure 4.5 illustrates the merged DFG of the topology shown in Figure 4.3, revealing the cycle in its dataflow and thus its potential for deadlock.

4.3.2 Deadlock Avoidance with Mercator Queues

As described in Section 3.1.1, Mercator prevents deadlock by reserving space in the queue of a cycle's head node for outputs from its cycle predecessor and placing constraints on the topology and

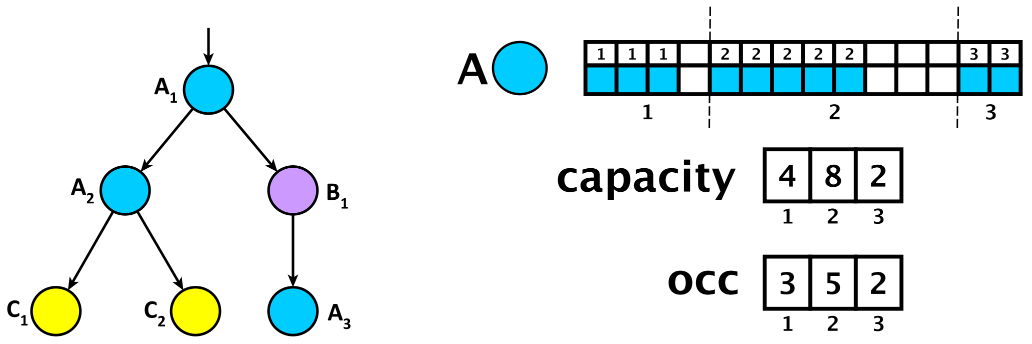


Figure 4.6: Queue storage for an example module with three instances. Contiguous space for holding input items is partitioned by instance and partition metadata such as capacity and occupancy are stored with the queue. Partitions are not uniformly sized; each capacity depends on the maximum number of outputs generated per firing of its upstream node along its upstream (i.e. feeding) edge.

dataflow of cycle nodes. Since this strategy requires managing queue space for a cycle's head node differently than for other nodes of its module type, Mercator maintains queue space for all module instances separately for simplicity. Rather than merging queues of module instances completely, as the FC model dictates, it merges them *conceptually*, maintaining per-type arrays to hold metadata such as occupancy for all instances of the type. This strategy suppresses the full benefit of merging queues, requiring an extra step to gather inputs from the separate instance queues of a module to pack SIMD lanes for the module's firing. However, in exchange for this performance loss, we guarantee system integrity by preventing deadlock. The performance loss is mitigated by the dense storage of instance metadata in per-type arrays, which facilitates efficiency in the parallel gather operation.

To simplify memory management and attempt to recover some of the performance loss of maintaining separate queues for each module instance, we choose as an implementation decision to allocate contiguous queue storage for all instances of a module, creating a single module queue but partitioning it by instance. This scheme creates the potential for coalesced memory accesses when gathering module inputs from multiple instances in parallel.

Figure 4.6 illustrates Mercator's queue storage scheme.

In addition to preventing deadlock, MERCATOR's partitioned queue storage scheme has the benefit of supporting simple cycles in an application's DFG, a possibility we demonstrated with the `SelfLoop` benchmark in Chapter 3. We propose leveraging this capability to extend MERCATOR support to traditionally non-streaming application classes such as recursive and iterative applications in Chapter 5.

4.4 MERCATOR Scheduler: Efficient Internal Execution

Having established MERCATOR's queueing strategy, we now consider the MERCATOR scheduler's interaction with its queues. Recall that MERCATOR's scheduler behaves as presented in Section 3.1, choosing a single module to fire at each firing step until all inputs are processed.

Although the scheduler's interaction with queues depends to some extent on its scheduling policy, any deterministic scheduler must at the very least choose some module with a non-blocked instance at each firing step in order to make progress. We therefore consider the operation of determining which modules have a non-blocked instance at each firing step– i.e. the set of *feasibly fireable* modules– to be fundamental to the scheduler and require that this operation be performed efficiently in SIMD in order to avoid becoming a performance bottleneck. Without increasing complexity, we also calculate *how many* inputs to each module instance are non-blocked at each firing step; i.e., how many inputs are feasibly fireable. This information is used by MERCATOR's scheduling algorithm, which is discussed later in this section. We now describe the requirements of the feasible firing calculation and its implementation in light of our queue management strategy.

4.4.1 Feasible Fireability Requirements

Intuitively, the feasible fireability of each input is determined by queue occupancies: the scheduler must respect queue capacities and not schedule any inputs for execution whose outputs could overflow– or rather, be blocked by– their immediate downstream queues.

To calculate how many of its inputs a module may fire at a given step while respecting downstream queue occupancies, we first consider how to calculate the number of inputs feasibly fireable with respect to a single output edge of one instance of the module. The number of feasibly fireable inputs with respect to an output edge is upper-bounded by two factors: the number of inputs available to be consumed, and the number of slots available on the output edge's downstream queue. Each output edge feeds a downstream queue which has some capacity, say c, and some occupancy occ, and each output edge has an associated number of output items γ produced as a cohort per input processed by the module. The number of slots available on the downstream queue is therefore $c - occ$, and these downstream queue slots can accommodate the firing of $\left\lfloor \frac{c-occ}{\gamma} \right\rfloor$ inputs (since each input can produce up to γ outputs). The minimum of the upper bounds– that is, the minimum of the number of inputs accommodated by downstream queue slots and the number of inputs actually available to be consumed– is the number of feasibly fireable inputs with respect to the output edge.

Once the number of feasibly fireable inputs with respect to each output edge is known, the number of feasibly fireable inputs for each instance may be calculated as the minimum over all its edges of the feasible fireability count per edge. This follows from the fact that a blocked edge blocks its entire instance.

Finally, the number of feasibly fireable items for the entire module may be calculated as the sum of feasibly fireable items over all its instances. Figure 4.7 shows pseudocode for a sequential version of the feasibility calculation.

To perform the calculation, the scheduler must know the maximum number of output items produced per input along each output edge of a given module as well as the occupancy of the module queue and of all queue partitions downstream of any instance of the module. The user-provided specification for the application immediately provides the former, and the queue storage strategy exposes queue occupancies via each module's queue partition metadata.

```
calculateFeasibleInputs()
{
  for all modules  {
      for all instances  {
          for all output edges  {
              // calculate feasible firing count for edge
              //   c: capacity of downstream queue
              //   occ: current occupancy of downstream queue
              //   gamma: outputs per input produced along this edge
              //   numAvailable: inputs present in this module's queue
              numFeasible = min ( floor( (c - occ) / gamma ), numAvailable )
          }
          // store min over output edges in numFireableByPartn[]
      }
      // store sum over instances in totalFireable for module
   }
}
```

Figure 4.7: Pseudocode for sequential version of scheduler scan to calculate the number of feasibly fireable elements per instance of each module.

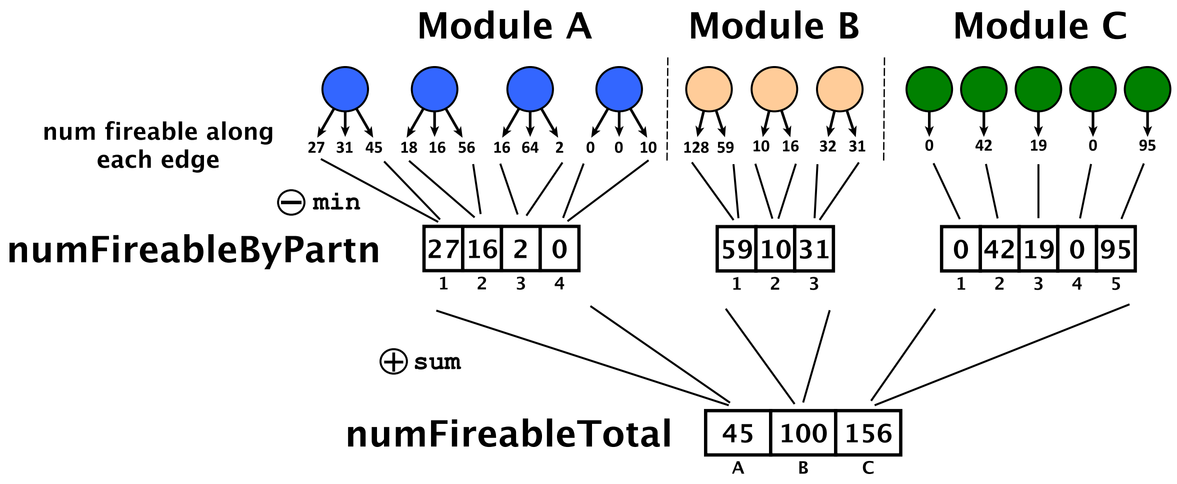


Figure 4.8: Diagram of scheduler scans to calculate the number of feasibly fireable elements for each instance of each module and total per module for an example topology (DFG not shown). The calculation may be accomplished by two parallel segmented scans: a min-scan to calculate the number of feasible inputs per instance from the output edges of the instance (stored in `numFireableByPartn[] for each module`, and a sum-scan to gather the total number of feasible inputs per module from the instances of the module (stored in `numFireableTotal` for each module). Counts of fireable elements for each output edge are based on input queue occupancies, downstream queue occupancies, and the number of outputs per input generated along the edge, with only the final value of the number fireable shown here.

4.4.2 Feasible Fireability Calculation

In order to prevent the scheduler from becoming a performance bottleneck, the feasible firing calculation shown in Figure 4.7 must be efficient, which implies it should be executed in parallel to the extent possible. Parallelism exists at multiple granularities in the calculation: data must be examined for each downstream queue of each instance of each application module. This nested parallelism may be exploited to accomplish the calculation with successive parallel segmented scans, as shown in Figure 4.8. The indices used in these scans by each thread (i.e. with which module, instance, and edge each thread is associated) may be computed statically before execution because the application graph's topology is known *a priori* and is constant across its execution.

In practice we find that the fireability calculation takes a small percentage of overall runtime ($< 1\%$) for our synthetic benchmarks presented in Chapter 3 using a simplified strategy parallelizing only one level of the calculation hierarchy. For the BLASTN application experiments presented in Chapter 3, scheduler operations accounted for just under 20% of runtime. We analyze this profiling result in Section 4.5, showing that the majority of the difference in scheduling time ratios between the synthetic benchmarks and BLASTN is accounted for by the difference in the computation load of their module functions rather than a difference in scheduler operation time. BLASTN's timing profile suggests that lowering the cost of scheduler operations would further increase the performance advantage of MERCATOR for BLASTN and other applications with similar module function profiles, and future versions of MERCATOR will therefore employ the hierarchical segmented scans presented in this section.

4.5 MERCATOR Scheduler: Firing Strategy

In the previous section we considered the efficient internal implementation of an essential scheduling operation given MERCATOR's queue storage scheme. In this section we consider MERCATOR's scheduling strategy. This strategy consists of two parts: a *global policy* for choosing one from among all the feasibly fireable modules at each execution step, and a *local policy* for selecting items from the chosen module's queue for execution. Both of these parts work together to achieve high utilization during execution.

The scheduling problem for parallel execution systems has been well-studied in many contexts. However, we are unaware of any theoretically-grounded strategy targeting throughput that is applicable to Mercator's execution model. In contrast to other execution models, for which scheduler optimization is a complex problem, Mercator's execution model suggests a straightforward yet efficient strategy, which we propose below.

The goal of our scheduling strategy is to maximize SIMD utilization within each module's firings, which we argue maximizes total execution throughput as follows:

Assumption: Application execution time consists of module execution time only, with no data movement or scheduling overhead. The assumption implies that minimizing total module execution time will maximize application throughput.

Claim 1: Minimizing the number of firings of each module maximizes total execution throughput for Mercator applications.

Proof: We first argue that local module throughput is maximized when a module's total number of firings is minimized, then that maximizing local throughput for each module is sufficient to maximize total execution throughput for an application.

Assuming a given module's firing time is consistent, its throughput may be measured in terms of firing counts rather than units of time and calculated as

$$\text{throughput} = \frac{\text{num inputs processed}}{\text{num firings}}.$$

To see when this value is maximized, we note that Mercator's execution is work-conserving with respect to the grouping of items into ensembles: each input item follows the same datapath, produces the same number of output items, and is identically processed regardless of which ensemble it is a part of during each module firing. This property implies that the number of inputs processed by each module over the course of an application's execution remains constant for any specific grouping of items into ensembles. The module's local throughput is therefore maximized by minimizing its total number of firings.

Since in MERCATOR's execution model, modules execute *sequentially* with respect to each other (though in SIMD within themselves), total execution time is simply the sum of all module execution times. Maximizing local throughput for each module is therefore sufficient to maximize overall execution throughput.

Claim 2: The number of firings of a module is minimized when its average SIMD utilization is maximized.

Proof: The number of module firings required to process a given number of inputs may be calculated as

$$\text{num firings} = \left\lceil \frac{\text{num inputs}}{(\text{average SIMD utilization}) \times (\text{SIMD firing width})} \right\rceil .$$

Since the number of inputs and the SIMD firing width are constant, average SIMD utilization determines the number of firings, and the number of firings is minimized when SIMD utilization is maximized.

Conclusion: Combining Claims 1 and 2, we have shown that execution throughput of MERCATOR applications is maximized when the individual SIMD utilizations of modules are maximized.

This conclusion motivates our scheduling strategy, which may be summed up by the rule that we should always fire full SIMD ensembles unless absolutely necessary. Specifically, the MERCATOR scheduler operates as follows:

1. *Global policy: max-feasible choice.* When choosing a module to fire at each scheduling step, choose the source module if possible; otherwise choose the module with the greatest number of feasibly fireable items across all its instances.

2. *Local policy: lazy firing.* When firing a module in the *main phase* of application execution, only process as many items from its input queue as will pack full SIMD ensembles, leaving any *residual items* in the queue. When firing a module in the *tail phase* of application execution, process all input items.

Here we intend an intuitive definition of the main and tail phases of execution, with the tail phase comprising a small fraction of overall operations at the end of execution and the main phase comprising all other operations. Dividing the application's execution into a main phase and a tail phase allows us to take advantage of our second insight while still preventing starvation and guaranteeing termination. Ideally, each module would execute in main-phase mode until the very last time it fires; in other words, it would always process full SIMD ensembles when firing except perhaps for one non-full ensemble on its last firing containing the final residual items on its queue. However, given the possibility of dataflow cycles and the inherently data-dependent execution traces of MERCATOR applications, it is difficult in practice to determine when a module is firing for the last time.

As a heuristic, we consider an application to be in tail mode when all items have been cleared from its input buffer. This makes the relative duration of tail mode operation dependent on module queue sizes, but it also allows relative tail mode duration to shrink with increasing problem input size, which aligns well with the streaming paradigm. In practice we find that our lazy firing policy increases average SIMD occupancy per firing and reduces the total number of firings over an application's execution, suggesting that the tail mode heuristic is effective. Results of implementing the lazy firing policy are presented in the next subsection.

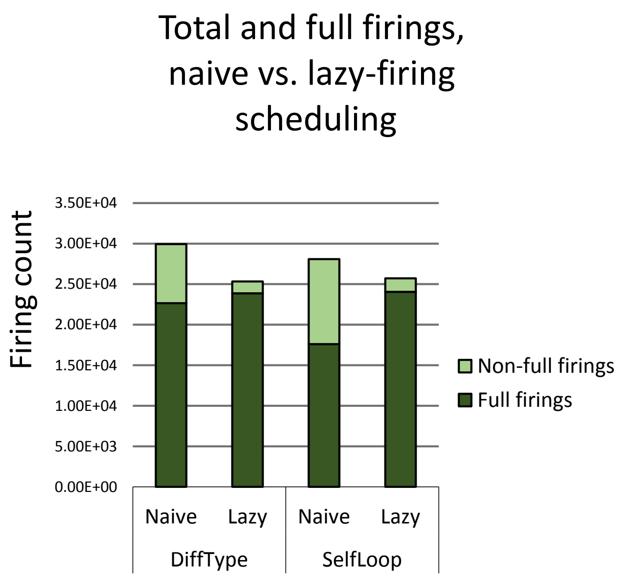
Giving priority to the source module in our first firing rule ensures that inputs will be offloaded from the input buffer and injected into the application's DFG as soon as possible, preventing bubbles in the system waiting for inputs. Choosing the module with the most feasibly-fireable input items minimizes the possibility of selecting a module with less than a single ensemble's worth of items in the main phase. With appropriately sized queues and a preference for firing the source module, the possibility of selecting a module with less than a single ensemble's worth of items in the main phase is eliminated.

4.5.1 Mercator Scheduler: Results

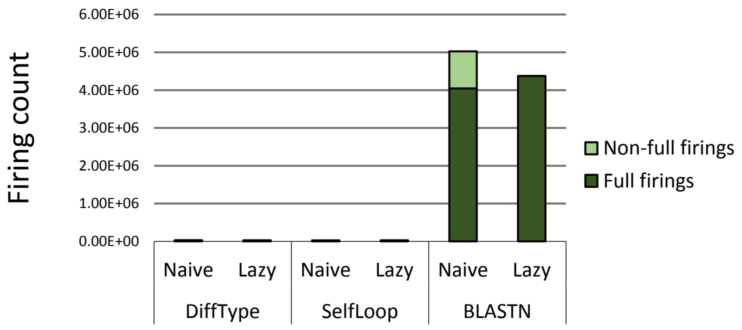
To measure the effectiveness of Mercator's "lazy firing" policy, we compared execution statistics of selected benchmarks from Chapter 3 under two local scheduling policies: the *lazy firing* policy presented in the previous subsection and the *naive* policy used to generate the results presented in Chapter 3. The naive scheduler uses our max-feasible choice global policy, but locally always chooses all input items on a module's queue for execution when the module fires. We present and discuss these statistics below, beginning with overall occupancy and timing results.

Global execution behavior Figure 4.9 shows firing counts for two representative synthetic benchmarks and the BLASTN application, with firings broken down into "full firings"– those having only full ensembles– and "non-full firings," those having some residual items. For all applications, introducing the lazy firing policy substantially increased the number of full firings and decreased the total number of firings, achieving its goal of guiding the application's execution toward maximum utilization.

Despite the consistent increase in full firings and decrease in total firings across applications, the effect of the lazy firing policy on overall runtime was application-dependent (see Figure 4.10). For the synthetic benchmarks, the effect was as expected: overall runtime decreased with the reduction in total firings. However, the runtime of BLASTN increased under the lazy firing policy by 7.3% despite a decrease in its number of firings and its near-maximal full-firing rate of 99.98%. Figure 4.11 shows overall firing counts and execution times for all four of our simple synthetic benchmarks and BLASTN under the lazy-firing and naive scheduling policies.



(a) **Synthetic benchmarks**. Selected benchmarks from Chapter 3.



(b) **Synthetic benchmarks and BLASTN**. BLASTN exhibits the same firing count trends as the synthetic benchmarks, with approximately two orders of magnitude more total firings.

Figure 4.9: Firing statistics for selected synthetic benchmarks and BLASTN. For each application, the number of full firings increased while the total number of firings decreased with the introduction of the lazy firing strategy.

The counterintuitive runtime result observed for BLASTN suggests that SIMD utilization during module firing is not the only significant factor contributing to runtime. One possible source of increased runtime despite the decreased firing count is an increased firing time per module. Although our performance model assumes that firing time per module is constant regardless of SIMD occupancy, firing time may increase at high occupancies due to exacerbated memory inefficiencies such as noncoalesced accesses. Indeed, we noted in Section 3.4.2 a difference in time required to process

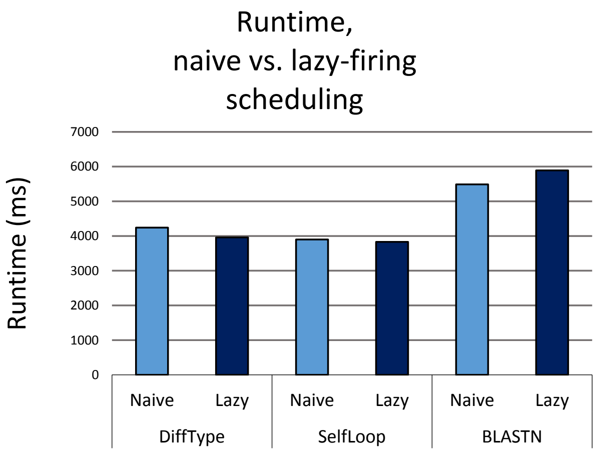


Figure 4.10: Timing results for applications highlighted in Figure 4.9b. Although the total number of firings decreased for all applications with the introduction of the lazy firing policy, the total runtime of the BLASTN benchmark increased.

each input ensemble depending on the overall size of the input stream. Another possible source of increased runtime is scheduler overhead. We further investigate these causes in our discussion of component-wise time results below, but regardless of the cause, the difference in runtime effects between the synthetic benchmarks and BLASTN implies a performance tradeoff between increased utilization and resulting overhead that is controlled by the choice of scheduler. A systematic study of this tradeoff is reserved for future work but enabled by the high occupancy achieved by the lazy firing schedule.

We now investigate the scheduling performance results further by analyzing total runtime in terms of its constituent components.

Components of global execution To better understand the impact of introducing the lazy firing policy, we decompose total execution time into the time spent in three non-overlapping MERCATOR components under the naive and lazy-firing scheduling policies: gathering and executing inputs (i.e. module firing time), scattering outputs from output channels to downstream queues, and scheduler operations. Figure 4.12 shows these decompositions for the `DiffType` synthetic benchmark and for BLASTN.

Benchmark	Local scheduling strategy	Total firings	Full firings	Execution time (ms)
DiffType	Naive	29937	75.67%	4242
	Lazy firing	25338	94.21%	3960
SelfLoop	Naive	28084	62.69%	3900
	Lazy firing	25710	93.56%	3831
SameType	Naive	20055	62.76%	4330
	Lazy firing	8066	93.24%	4061
Merged	Naive	17366	96.37%	5828
	Lazy firing	17088	97.94%	5961
BLASTN	Naive	5024973	80.55%	5488
	Lazy firing	4374771	99.98%	5891

Figure 4.11: Firing counts and execution times for five applications under two local scheduling policies: the *naive* policy that always fires the maximum number of available items, and the *lazy firing* policy that only fires full SIMD ensembles until the tail phase of execution is reached. 'Full firings' are those with full ensembles only; i.e., no residual items. Benchmarks were run with an experimental setup as described in Section 3.4. Synthetic benchmarks were run with a filtering rate of 0.5 and a workload requiring 200 ms of computation per thread per node. BLASTN was run with a query size of 10 Kbases.

We first note that each of the three components consistently signaled either a cost or benefit from lazy firing for both benchmarks. Of the three, the only one directly dependent on SIMD firing occupancy is the gather/execute component– all GPU threads cooperate to accomplish the scatter and scheduling operations in Mercator, so their occupancy is constant regardless of scheduling policy. It is therefore the gather/execute component whose runtime is reduced by the introduction of the lazy firing policy, and its consistent reduction is evidence that the decreased total firing count outweighed any per-module firing cost increases for each benchmark.

The slight increase in time spent on scheduling operations for each benchmark suggests that lazy firing incurs a small implementation overhead relative to the naive policy. More surprising than this overhead, which is to be expected since lazy firing strictly adds instructions to the naive scheduler to manage the residual-item boundary, is the increase in scatter time when using lazy firing. Since the scheduling policy has no direct influence on the scatter procedure, we suspect the increase in runtime to be an indirect effect of increased SIMD firing utilization. For example, with more inputs

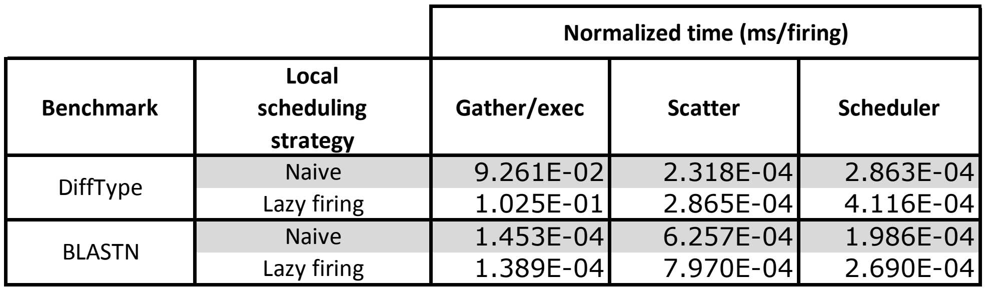
		Time (ms)				
Benchmark	Local scheduling strategy	Gather/exec	Scatter	Scheduler	Gather/exec + Scatter + Scheduler	Total execution
DiffType	Naive	2772.60	6.94	8.57	2788.11	2792.25
	Lazy firing	2597.77	7.26	10.43	2615.46	2619.20
	Lazy firing - naïve	-174.83	0.32	1.86	-172.65	-173.05
BLASTN	Naive	730.19	3144.26	997.90	4872.35	5488.00
	Lazy firing	607.75	3486.79	1176.92	5271.46	5893.00
	Lazy firing - naïve	-122.44	342.53	179.02	399.11	405

Figure 4.12: Total execution time (rightmost column) and time taken by various Mercator application components for the `DiffType` and BLASTN benchmarks under the naive and lazy-firing scheduling policies. The influence of introducing the lazy firing policy is shown as the difference in execution time of each component between using lazy firing and the naive policy. A negative-valued difference, indicating that runtime decreased when lazy firing was introduced, signals a benefit from lazy firing, whereas a positive-valued difference signals a cost. The highlighted boxes indicate both that the three components shown account for all but a negligible amount of total runtime, and that `DiffType` saw a performance benefit (decrease in runtime) when lazy firing was introduced while BLASTN saw a performance cost (increase in runtime).

present per firing, more outputs may be produced per firing, each of which must be scattered to a downstream queue. In this case the index lookups and memory writes required for this scatter (described in Section 3.2.2) may cause increased memory contention and hence the observed increase in runtime of the scatter component.

Having examined the effect of introducing lazy firing on each component of execution time, we now turn to analyzing the runtimes of the individual applications depicted in Figure 4.12. Although their overall runtimes are of a similar magnitude, differing by a factor of two, the execution profiles of `DiffType` and BLASTN reveal vastly different proportions of time spent in each runtime component and these differences explain the mismatched effect of the lazy scheduler on the two applications. `DiffType` spends more than 99% of its time in the gather/execute component, so the efficiency improvement in that component resulting from lazy firing far outweighs the costs incurred in other components, leading to an overall performance benefit in reduced application runtime. BLASTN, on the other hand, spends more time in both the scatter and scheduling components than in the gather/execute component, and the costs incurred in these other components outweigh the increased efficiency of gather/execute for a net performance loss. Since one of Mercator's design goals is to minimize the cost of infrastructure operations such as scheduling and the data movement accomplished by the scatter operation, BLASTN's execution profile warrants further analysis.

The runtime breakdown shown in Figure 4.12 seems to indicate *prima facie* that BLASTN spends an inordinate amount of time in the scatter and scheduler operations compared to `DiffType`. However, execution profiles of the two applications normalized by firing count reveal that the runtimes of BLASTN's scatter and scheduler operations are comparable to those of `DiffType`. We examine these normalized profiles now.



		Normalized time (ms/firing)		
Benchmark	**Local scheduling strategy**	**Gather/exec**	**Scatter**	**Scheduler**
DiffType	Naive	9.261E-02	2.318E-04	2.863E-04
	Lazy firing	1.025E-01	2.865E-04	4.116E-04
BLASTN	Naive	1.453E-04	6.257E-04	1.986E-04
	Lazy firing	1.389E-04	7.970E-04	2.690E-04

Figure 4.13: Execution time of the three main runtime components from Figure 4.12, normalized by the number of module firings for each application run.

Normalized component time Figure 4.13 shows per-firing execution times of the three main runtime components introduced above. These normalized times reveal at least two relevant characteristics of the relative execution behavior of `DiffType` and BLASTN latent in the results already discussed.

First, although BLASTN spends significantly more time in scatter and scheduling operations than `DiffType`, the time they spend in those operations *per firing* is comparable– in fact, `DiffType` spends more time per firing in the scheduler than BLASTN does.

Second, although `DiffType` only spends roughly 3× the time of BLASTN in the gather/execute operations, it spends almost 3 full orders of magnitude more time per firing in gather/execute. In other words, the lopsided execution profiles shown in Figure 4.12 are more a result of BLASTN spending a very short time in its `run()` functions relative to `DiffType` than they are of BLASTN spending longer in the scatter and scheduler components of Mercator.

This finding confirms Mercator's ability to confer a performance benefit on an application even if its computational workload is relatively low, since BLASTN's implementation using Mercator queues outperformed an implementation with no queues (results in Section 3.4). In this case it seems that the filtering behavior of BLASTN creates enough wavefront irregularity that Mercator's management of the irregularity to increase utilization outweighs its relatively high management overhead. The execution profiles of `DiffType` and BLASTN thus highlight the impact of an application's computational workload and dataflow characteristics on its performance in Mercator and suggest an investigation into the workload characteristics of a broader range of applications as future work.

4.6 Conclusion

In this chapter we first presented the FC Parallel Execution Model (PXM) on which Mercator's execution model is loosely based. We then exposed a latent potential for deadlock in the FC model and described our module queue storage scheme, which is based on the key insight of the model but modified to facilitate deadlock prevention. Next we detailed the SIMD-efficient feasible fireability operation used by the Mercator scheduler, highlighting its interaction with the queue storage scheme and showing that the scheduler incurs low overhead for our benchmarks. Finally we presented Mercator's scheduling strategy, arguing that it guides executions toward maximum SIMD utilization and showing its effectiveness for our benchmarks.

This chapter concludes our presentation of the Mercator framework. In the next chapter we discuss future work, which includes extensions of Mercator's functionality as well as its application to traditionally non-streaming domains such as graph processing. Crucial to these extensions is

Mercator's data management and execution strategy, as encapsulated in the concept of the merged DFG introduced in this chapter.

Chapter 5

Conclusion and Future Work

In this dissertation we have considered the performance challenge presented by wavefront irregularity when executing irregular streaming applications on wide-SIMD architectures. We examined a representative irregular streaming application in Chapter 2, highlighting the sources of wavefront irregularity in its computation and presenting our application-specific methods for managing it. This examination and our awareness of other applications exhibiting similar wavefront irregularity motivated the generalization of the techniques employed in that application, which the rest of this dissertation has addressed. In Chapter 3 we introduced our Mercator framework, which transparently provides generalized and efficient irregular data management to developers. In Chapter 4 we provided more details of Mercator's implementation, including the dataflow model on which it is based, its SIMD-efficient strategies for implementing the operations required to manage wavefront irregularity, and its scheduling algorithm. Together, Chapters 3 and 4 provide a comprehensive view of the core functionality, interface, and implementation of the Mercator framework that encapsulates the main contributions of this dissertation. In this chapter we build on that core, exploring how Mercator exposes optimization opportunities such as work-to-thread mapping adjustment for its applications and how it may be used to execute applications from traditionally non-streaming paradigms such as graph processing. We provide proofs of concept for each of these research directions and propose further explorations into these areas as well as optimization opportunities within the Mercator framework itself.

We present our ongoing and future work in two main areas: optimization opportunities for the existing MERCATOR infrastructure and extensions to new application classes.

5.1 Optimization Opportunities

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.

Second, MERCATOR currently executes a copy of an application's DFG with a single GPU block in isolation. It may be advantageous in some cases to share datapaths between blocks, allowing modules from the same copy of the DFG to fire simultaneously and hence exposing pipeline parallelism not currently accessible in our execution model. Efficiently managing dataflow in this environment would require load balancing and enforcing consistent data sharing across blocks.

Third, MERCATOR may be augmented to search for optimal thread-work mappings. MERCATOR is designed to exploit the lightweight, dynamic thread-to-work mapping available on wide-SIMD architectures such as GPUs to manage data wavefront irregularity. In addition to this functionality, the ability to tune thread-to-work mappings enables an exploration of mapping options for a given application– indeed, the expanded name of the MERCATOR framework is Mapping enumeERATOR for CUDA– and a search for the optimal mapping. We propose two ways in which the MERCATOR framework can be used to explore thread-to-work mappings: variable module mappings and module fusion/fission. MERCATOR currently supports performing both of these explorations manually, with support for automated exploration being future work.

5.1.1 Variable Thread-to-Work Module Mappings

To support different thread-to-work mappings inside developer module code, Mercator allows a user-specified number of input items to be assigned to each thread, or vice versa, inside the module's `run()` function. For all experiments described in this dissertation so far, the mapping ratio has been one-to-one. However, different mapping ratios may confer performance benefits for particular modules: 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 and coalescing of memory accesses. We plan to explore under what conditions, if any, performance benefits consistently accrue for different mappings.

The modular structure of Mercator applications and the framework's use of dynamic remapping make different mapping options straightforward to expose: developers simply provide the desired inputs-to-threads ratio as part of the mapping specification for each module, and the system provides an appropriate item or item set to the user in the parameters of the `run()` stub function (see the User Manual in Appendix A for details and examples). Although different mappings of a module currently require different developer-provided `run()` function implementations, future versions of Mercator may automate the process of enumerating implementations to match desired mappings given a reference implementation based on a 1:1 mapping. Section 5.2.4 gives an example of an application we have manually implemented under diverse mappings with minimal code changes.

5.1.2 Module Fusion/Fission

Our system validation experiments in Chapter 3 were designed to test whether the benefit of increased utilization outweighed the overhead of Mercator queue management for our benchmark applications. Although we did find a net benefit for most node filtering rates and workloads tested, our experiments revealed a performance tradeoff between overheads and benefits dependent on node filtering rates and workload. We propose extending our simple validation test to explore this tradeoff by searching for the optimal granularity of modularization of an application. Modularization granularity may be adjusted by fusing or splitting nodes in the application's DFG, which corresponds to combining multiple module functions into one or dividing a module function into multiple functions.

Fusing nodes may be advantageous when filtering rates or computation per node are low, as little cost is incurred from low SIMD utilization relative to the overhead of remapping operations. For example, in our benchmark tests, the `Merged` application outperformed the `DiffTypePipe` application when a filtering rate of 0.0 or a low node computation setting was used.

Conversely, splitting a node may be advantageous when filtering rates or computation per node are high, as vacant SIMD lanes incur a high cost that could be recovered by remapping for high utilization partway through the node's computation. Our benchmark tests show this benefit for the `DiffTypePipe` application over the `Merged` application for most filtering rates and computation settings.

MERCATOR inherently supports module fusion and fission, but such operations must currently be performed by the developer via specifying a new application with the desired topology. Because manually choosing an optimal granularity of modularization requires either substantial timing analysis or intuition for an application's dataflow characteristics, automating this choice would be a helpful aid to optimization.

Since module fission involves splitting a user-defined `run()` function into multiple functions across module boundaries, automating the fission process for arbitrary modules poses a significant challenge. Sophisticated code analysis and transformation may be required to find feasible split points within the `run()` function (e.g. splitting inside a loop is not possible) and ensure consistency of the original module's data (e.g. any function-local variables must be copied between modules). Functions with certain properties may be amenable to automated fission, but further study would be required to identify such function classes and determine the feasibility of automatic fission.

Module fusion holds more promise for automation than fission for some graph topologies. For example, in the case of a linear pipeline with no back edges, contiguous modules may be merged by concatenating their function code with no other effect on application dataflow. Automating this process would still require appropriate data management to ensure that data previously passed between module functions via Mercator queues would be connected appropriately between the fused components of the new `run()` function via variables, but this task would seem to require substantially less code analysis and transformation than automatic fission.

In the case of more complex topologies, it may not be possible to fuse a given pair of modules without altering application dataflow. For example, straightforward fusion of a downstream module to an upstream module with more than one output edge in its merged DFG may create a new data path originating at the downstream module.

To proceed with a study of automated module fission/fusion, we propose first exploring automated fusion for simple graph topologies, then expanding the study to more complex topologies for fusion or simple function semantics for fission.

5.2 Extensions to New Application Classes

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 applications such as graph-processing algorithms that execute iteratively on a subset of fixed inputs rather than once each on arbitrarily many streaming inputs.

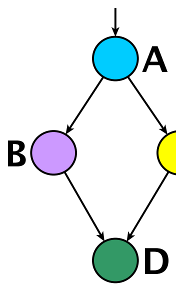


Figure 5.1: DFG of an example application requiring join semantics. Assume module **D** is the head of a true join over modules **B** and **C**, meaning it needs some number of inputs from both **B** and **C** in order to fire. If all data rates in this application are static, appropriate firing rules and queue sizes can be determined at compile time to ensure a stable execution with no deadlock. To see why this DFG represents a problem for Mercator if dataflow is irregular, suppose **A** and **C** fire repeatedly until they are blocked by downstream queues, but **B** produces no output. Now **D** is blocked waiting for output from **B**, **A** and **C** are blocked by downstream queues, and **B** is blocked waiting for output from **A**. In other words, the system is deadlocked.

5.2.1 Applications with "join" Semantics

Join semantics in the presence of filtering in Mercator's execution model are not straightforward. Here we distinguish between true join semantics as provided by, e.g., SDF application frameworks, and allowing multiple output edges to feed a single input queue, as Mercator does in the case of DFG cycles. In the latter case, the downstream module is indifferent to which output edge the items on its input queue arrive from, and input items are processed in a single stream stored in the module's input queue. In the case of true join semantics, by contrast, a downstream module that is the head of a join requires some number of inputs from *each* of its upstream join edges simultaneously in order to fire. For an application with static data rates, this requirement poses no problem because dataflow can be solved at compile time and therefore is known at runtime. For an application with irregular dataflow, however, data-dependent filtering at an upstream node may prevent a downstream join node from making progress as it waits for required input from the upstream edge that never arrives, causing deadlock in the system. Figure 5.1 illustrates this challenge.

While consistent behavior can be defined in such cases [19], the required semantics for doing so would have to be implemented in Mercator. Meanwhile 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 propose initially extending Mercator to support application topologies in which certain subgraphs are free of irregularity but do contain joins. In this case each such subgraph could be analyzed and scheduled as a unit relative to the rest of the application.

5.2.2 Iterative Applications

Up to this point our discussion of Mercator has focused primarily on the modeling and execution of traditional streaming applications, albeit irregular ones. In the following subsections we consider the practicality of supporting another important class of applications in Mercator: those that operate iteratively, executing some function(s) on their inputs for a variable, possibly data-dependent number of rounds. Examples of iterative applications include graph processing algorithms such as Google's PageRank [93], recursive algorithms such as those found in [104], and any algorithm conforming to the Bulk Synchronous Parallel (BSP) model of computation [126]. Iterative applications would seem to be fundamentally incompatible with the streaming paradigm Mercator is designed to instantiate: iterative applications apply the same function(s) to a given (possibly fixed) set of inputs multiple times, whereas traditional streaming applications apply a set of functions once each to a large set of inputs. However, by exploiting two key features of the Mercator framework– its support for dataflow cycles and its developer interface for storing global application data– iterative applications may be conceptually represented in Mercator's model of computation. The synchronization and parallel reduction requirements of some iterative applications currently prevent them from being implemented as a standalone Mercator application; however, these obstacles may be overcome through the use of multiple application executions, and future versions of Mercator will support the necessary operations inside a standalone application.

We first describe how Mercator may be used to represent iterative applications by considering the general Bulk Synchronous Parallel iterative model, then show how some specific categories of iterative applications may be modeled in Mercator.

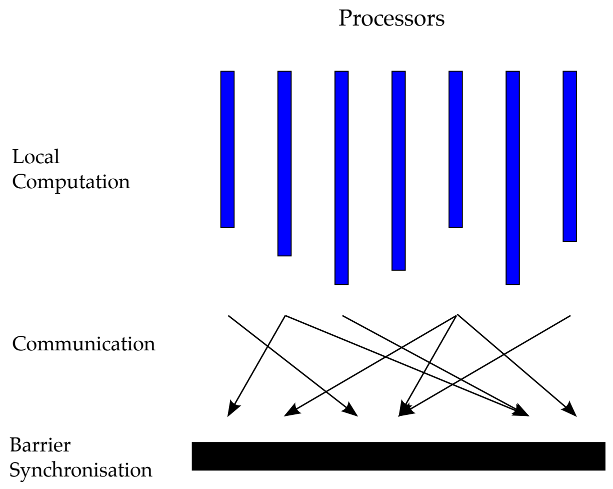


Figure 5.2: Processing steps for a single round in the Bulk Synchronous Parallel (BSP) model (image from [130]).

5.2.3 Bulk Synchronous Parallel (BSP) Applications

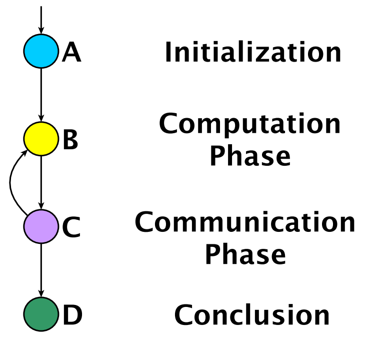
BSP is a classic model of iterative parallel computation dating to 1990 and forming the basis of such big-data processing systems as Google Pregel [74] and Apache Giraph [8]. In the BSP model, processing proceeds in rounds called "supersteps." Each superstep consists of two phases: a computation phase, in which processors operate on input elements assigned to them for the superstep, and a communication phase, in which processors pass messages to each other. During this communication phase, processors may exchange information computed during the computation phase and decide which (if any) input items to process during the next superstep. All processors synchronize at the conclusion of each superstep, and processing terminates when no processor needs to execute another superstep. This may happen as a result of either finishing a set number of iterations or reaching a fixed-point defined by local or global criteria. Figure 5.2 illustrates the BSP model.

To enable execution of traditional BSP applications in Mercator, we translate the salient features of the BSP model into the Mercator model. We first present a straightforward translation, then discuss its limitations in light of BSP's reduction and synchronization requirements. We then suggest an alternative formulation feasible in the current version of Mercator.

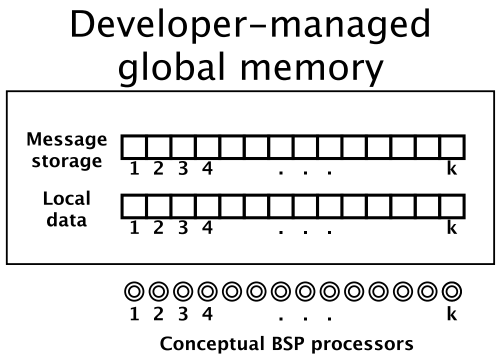
Both the computation phase and the communication phase of BSP correspond naturally to module execution steps in Mercator since in these phases all processors apply the same function(s) to their assigned inputs. We therefore represent each of these phases as module(s) in a merged DFG suitable for Mercator execution. We model the communication phase of BSP using Mercator's developer-accessible global memory: rather than the message-passing paradigm of communication employed by BSP, we employ the shared-memory paradigm via this global memory, with each BSP processing unit having some reserved message space in the memory. We model the iterative dataflow of BSP using cycles: modules represent the computation and communication phases of operation, and connecting these modules into a cycle allows inputs to traverse the computation and communication phases multiple times. Mercator's support for irregularity allows the number of iterations executed for a given input to be data-dependent, as inputs may be filtered out after any number of cycles. This filtering strategy transforms the temporal irregularity of execution across cycles in BSP into the wavefront irregularity Mercator is designed to manage efficiently. Figure 5.3 illustrates our desired strategy for representing BSP applications in Mercator.

Two operations fundamental to the BSP model are especially challenging to support in Mercator: the synchronization barrier at the end of each superstep and the (possible) reduction over inputs at the end of a superstep to check for convergence. We discuss our current strategies for integrating these operations with Mercator applications and future prospects of Mercator's support for each operation in turn.

Superstep synchronization barrier The synchronization barrier required at the end of each superstep cannot be directly modeled using Mercator's dataflow semantics: the synchronization is not a barrier with respect to individual thread executions, which CUDA supports at the block granularity at which Mercator executes, but is rather a barrier with respect to application dataflow, which neither CUDA nor Mercator supports. In place of a barrier on the GPU, a round-trip to



(a) **Merged DFG**. Modules represent the computation and communication phases of the BSP formulation of the application, with initialization and conclusion code executing before and after the main iterative logic of the application respectively. A dataflow cycle involving the computation and communication modules enables inputs to traverse multiple iterations of those phases during execution. The computation and communication phases are represented by a single module each, but in practice may comprise several modules strung together depending on the number of functions applied to input data during those phases.



(b) **Data storage**. Space is allocated in global developer-managed memory for message storage and local data for each processor in the original BSP formulation of the application. The message storage space provides equivalent functionality to the message passing of BSP, but in a shared-memory environment.

Figure 5.3: Representation of a generic BSP application in Mercator. A cycle models iterative dataflow, and global memory facilitates communication between conceptual BSP processors. Synchronization requirements after the communication phase currently prevent this representation from being fully implemented in a standalone Mercator application.

the host CPU guaranteeing dataflow synchronization may be inserted at the end of each superstep by omitting the back edge in the application's merged DFG and invoking the application multiple times from the host. No data copies between host and device are necessary between invocations since global memory (holding the global application data) persists between kernel calls in a CUDA context. This strategy thus enables BSP applications to be correctly implemented as Mercator applications and realize the benefits of Mercator parallelization inside the computation and communication stages at the cost of one control transfer between host and device per superstep. Figure 5.4 illustrates the BSP dataflow modified to be fully compatible with Mercator.

Future implementations of BSP algorithms in Mercator may make use of developer-managed global memory to simulate a dataflow synchronization barrier, thus obviating the return trip to the

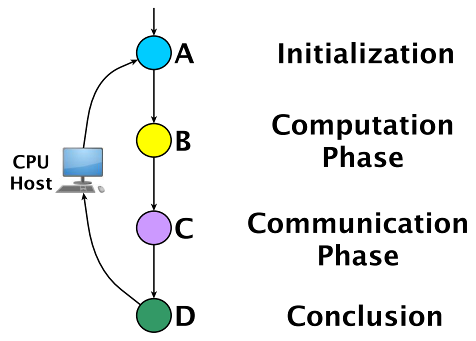


Figure 5.4: Representation of the generic BSP application from Figure 5.3 modified to be fully feasible in the Mercator framework. A round trip to the CPU host has replaced the dataflow back edge, allowing synchronization between supersteps. Although this round trip forms a loop over the entire DFG since the host must invoke the entire Mercator application, the initialization and conclusion modules may be rendered null during the main operation of the application via conditional code in their `run()` functions, causing the loop to effectively be over only the computation and communication phases, as desired.

host after each superstep and enabling the full execution of BSP applications in Mercator according to the merged DFG in Figure 5.3a. The goal of dataflow barriers in iterative applications is to enforce strict iterative execution across all inputs; that is, to ensure that no inputs are processed in a given round until all inputs have been processed in the previous round. Therefore, any method that enforces strict iterative execution suffices as a substitute for the hard synchronization barrier at the end of each round of iterative processing. We conjecture that such a method may be developed for applications whose DFGs consist of a single loop, which would include BSP applications. Loosely, the method would enforce strict iterative execution by reserving space in global memory for each input to be iteratively processed and using a threadsafe input-marking scheme to induce a monotonically increasing round-count ordering on input queues similar to BFS search ordering in graphs. Round-count barriers would be respected when pulling inputs from an input queue for execution, so that only items from the same round would be executed simultaneously. In this way, all items from a single round would be guaranteed to have completed execution by the module before any items from the next round execute, thus providing the functionality of a dataflow synchronization barrier. We propose to explore this method of simulating superstep synchronization as future work.

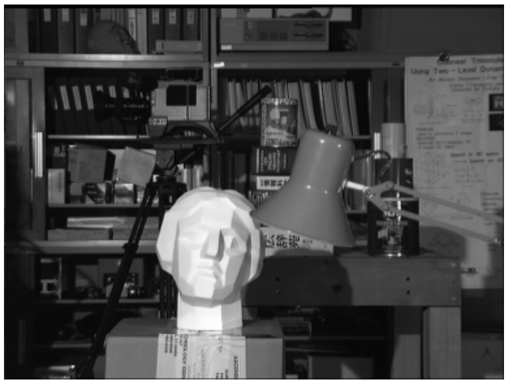
Reductions In addition to the synchronization requirement, some BSP applications require a reduction over multiple data items– possibly from multiple processors– at the end of each superstep to determine whether another superstep is required. Developer-facing reductions are not currently available in Mercator due to their fundamental execution synchronization requirements, so these reductions must be computed on the host during the dataflow synchronization stage of the algorithm.

Future versions of Mercator will provide developer-facing reductions in the form of special *reduction modules*. Similar to source and sink modules in the current version of Mercator, these reduction modules will be available for insertion into an application's DFG, with Mercator's user specification allowing details of the reduction's operation to be provided by the developer and implementations of reduction modules being supplied as part of Mercator's infrastructure.

5.2.4 Case study: Loopy Belief Propagation for Stereo Image Depth Inference

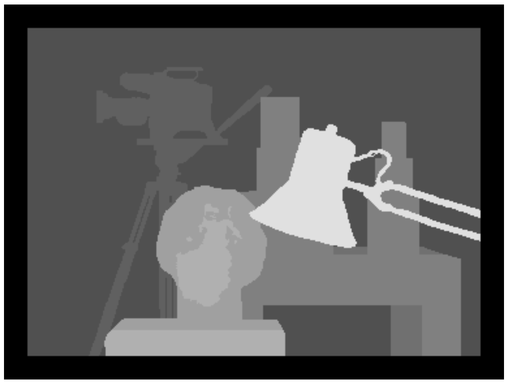
As a proof-of-concept for processing BSP applications using Mercator, we present our implementation of a Loopy Belief Propagation (LBP) algorithm for depth inference in stereo images. LBP is an algorithm for approximately inferring posterior probabilities in Bayesian network models [77, 95] that has been successfully applied to many signal-processing and computer vision problems [98]. LBP simplifies the problem of maximizing joint probability estimates ("beliefs") over an entire network by making local estimates at each node and propagating information about these local beliefs through the network in a series of rounds, refining the beliefs in each round until termination or convergence.

Although many variations of LBP exist, we implement a simple synchronous version that conforms to the BSP model, and we apply it to the problem of inferring depth from stereo images. This particular application of LBP has been studied and implemented with optimized algorithms (e.g. [116, 132]); for simplicity our implementation and experimental setup are based on a straightforward tutorial version of the algorithm [69].

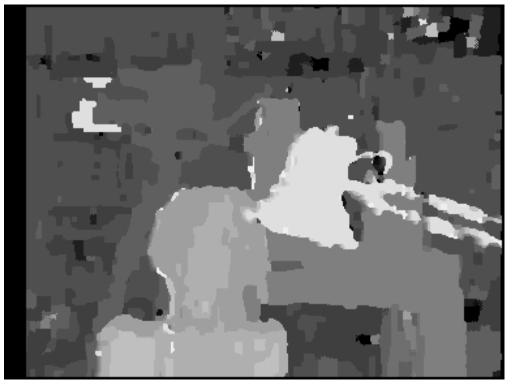


(a) **Left-hand input image**.

(b) **Right-hand input image**.



(c) **Ground truth depth**. Depth values for all pixels in the left-hand input image are represented inversely by pixel intensity: foreground objects are light while background objects are dark.



(d) **Calculated depth**. Depth map calculated by MERCATOR LBP application.

Figure 5.5: Sample stereo input images and depth-map output image produced by a MERCATOR implementation of Loopy Belief Propagation (LBP) vs. ground truth output. Input images are taken from [108].

Problem setup The problem of stereo vision depth inference as we study it may be stated as follows: given two images of the same scene taken from cameras at some small horizontal offset from each other, calculate the depth of all pixels in one image relative to its camera. Depth in stereo inference problems is assumed to take on one of some small set of discrete values at each pixel. Figure 5.5 shows inputs and two independently calculated outputs for an example problem instance.

The LBP strategy for solving this problem is based on two assumptions about "corresponding" pixels in the two images, meaning pixels that represent the same point in the scene rather than pixels at identical coordinates in their respective images. First, corresponding pixels will be found in their respective images at a horizontal coordinate offset inversely proportional to their depth-from-camera in the scene due to the parallax effect. For example, corresponding pixels from foreground objects will have a greater shift in their respective images since objects appear to move faster across our field of vision when they are closer, and corresponding pixels from background objects will have a lesser shift because objects appear to move more slowly across our field of vision when they are farther away. This assumption is based on the nature of stereo vision and is not specific to LBP techniques. We use this assumption to calculate depths immediately from corresponding pixels' offset values, so that the main LBP algorithm is run to compute estimates of offset values between corresponding pixels rather than depths directly. Second, neighboring pixels in an image will most of the time have a similar depth setting in the final solution, since most pixels in a projected image lie inside object boundaries rather than on the boundaries themselves. LBP explicitly exploits this assumption by incorporating neighboring pixels' beliefs into each pixel's estimation of its own depth.

LBP strategy The LBP algorithm for stereo depth estimation computes, as mentioned above, an estimate of the corresponding pixel's horizontal offset value for each pixel in one of the stereo image inputs. It maintains a vector of beliefs over all possible offset values throughout the algorithm, selecting the offset with maximum probability as its final estimate when the algorithm terminates. The algorithm proceeds as follows:

1. Each pixel initializes a belief vector holding probability estimates of its corresponding pixel's offset value for all possible values (16 in our case).
2. Each pixel examines messages from its 4 primary neighbors containing weighted beliefs about their true offset values. These messages are initially set to indicate uniform probability of all offset values.
3. Each pixel computes messages for each of its primary neighbors and sends them.

4. Each pixel incorporates the messages just received into a weighted sum that also includes terms that capture the algorithm's two main assumptions mentioned above. This sum is used to update the pixel's current belief vector about its own offset value.

5. An "energy calculation" is performed for each pixel using its beliefs, and the individual pixel energies are summed to form a global energy value for the current iteration.

6. The energy value is used to check the algorithm for convergence: if the energy does not change by more than some threshold amount between iterations, convergence is assumed.

7. Steps 2-6 above are repeated until convergence, after which the final depth estimates are calculated from the belief vector and output at the conclusion of the algorithm.

Implementation in Mercator Our LBP for stereo depth inference algorithm conforms to the BSP paradigm described above, and so may be modeled and executed in the Mercator framework in the same way. In this case each pixel represents a streaming input, and all information about pixels, including their relevant messages, is stored in developer-facing global memory. The merged DFG of the application follows the pattern of Figure 5.3a, with Step 1 of our LBP algorithm corresponding to the initialization module, Steps 2,3,5, and 6 corresponding the the communication phase, Step 4 corresponding to the computation phase, and Step 7 corresponding to the conclusion module.

Since no irregular data movement happens between the computation and communication phases of our algorithm, we implement all core algorithm logic– i.e., Steps 2-6 – as a single Mercator module. As in the case of the generic BSP procedure, our LBP algorithm requires a global synchronization operation at the end of its communication phase, which we currently implement as a return trip to the host. In addition to the synchronization operation, our LBP algorithm also requires a reduction over all pixels to perform the global energy calculation in Step 5, which we currently also perform on the host. Figure 5.6 illustrates our LBP application's merged DFG.

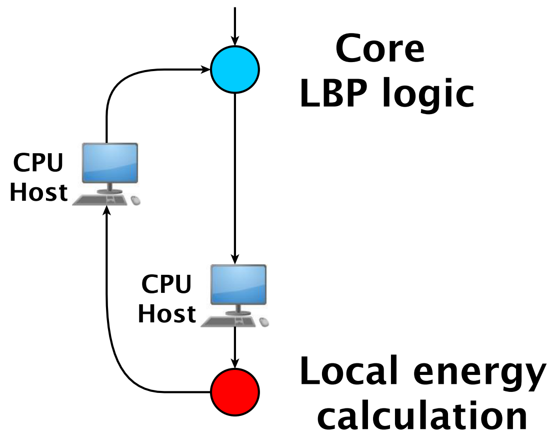


Figure 5.6: DFG of algorithm for stereo depth inference using LBP. Calculation of local energy for each pixel is implemented as a separate Mercator application from the core logic of the algorithm since a synchronization barrier is required between the main logic and the energy calculation. The reduction (sum) of all local energies is computed on the host after the calculation of local energies has completed, as is the convergence check and the setting of final offset values at the conclusion of the application.

Results The LBP algorithm for depth inference in stereo vision was implemented as described above by Edgar Flores (Samford University) and Theron Howe (Washington University). Figure 5.5d shows the depth map produced by the application when run on the input image pair also in Figure 5.5. This depth map was validated as identical to the output map produced by a CPU implementation of the stereo vision algorithm, confirming the correct translation of the algorithm into Mercator. Although neither the CPU nor the Mercator versions of the application were optimized, the Mercator version averaged a 48% speedup over the CPU version (45s vs. 67s), demonstrating the practicality of the Mercator implementation.

To explore the optimization opportunities facilitated by Mercator for the LBP application, we implemented a version of the application using each of three different module mapping configurations for the computation phase of execution: assigning 4 pixels to each thread, assigning 1 pixel to each thread, and assigning 16 threads to each pixel. The first two mappings are straightforward; the third involves cooperation (though not synchronization) among the 16 threads assigned to a given pixel in computing the necessary messages and belief updates for each possible offset value for that pixel. We found a substantial difference in runtime between the different mappings: relative to the reference 1:1 pixel-to-thread mapping, the 4:1 mapping saw a 14% slowdown and the 1:16 mapping saw a 119% slowdown. Although further investigation is required to discern the causes of the slowdowns, the substantial difference in runtime between the mappings indicates significantly different execution characteristics which may be exploited for speedup in other applications.

5.2.5 Graph Processing Applications

In Section 1.3.1 we described the vertex-centric or Think Like A Vertex (TLAV) paradigm [76] for processing large-scale graph applications, which is used by many systems today (e.g. [8,20,22,73,74]). In its general structure, TLAV processing closely resembles BSP processing: graphs are processed 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. It may use this result to queue one or more vertices for further processing in the next round.

TLAV processing may be accomplished in much the same way as BSP processing in Mercator. However, TLAV processing does have two important distinguishing characteristics relevant to its Mercator implementation: first, the amount of work required per round for each vertex is proportional the number of its incident edges, and second, the reduction required at the end of a processing round is also over incident edges for each vertex rather than the whole graph. These two characteristics add an additional layer of irregularity to graph processing workloads: 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 TLAV workloads within a computation round, 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.

To support the per-vertex reductions of TLAV processing in Mercator, the principal extension required is a *reduce-by-key* operation analogous to MapReduce [33]. 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 3.2.2. 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 [25], and an example of the explosion/reduction operations performed for that application is shown in Figure 2.7. The challenge in extending explosion and reduction to the general case is to do so while preserving the generality and transparency of Mercator's support for streaming, irregular dataflow. We discuss this challenge further in Section 5.2.7 below.

5.2.6 Applications with Dynamic Dataflow

In contrast to BSP and graph processing applications, which have partial support in Mercator but require extensions to be fully implemented in the framework, a class of iterative applications that is currently fully supported in the framework is those with dynamic rather than static dataflow.

Applications with dynamic dataflow– that is, whose DFGs are not known at compile time but unfold at runtime– are challenging to model in streaming systems since per-node resource requirements and dataflow paths are unknown until runtime. However, by managing application dataflow at the granularity of modules rather than nodes and supporting cycles, both of which Mercator does, dynamic-dataflow applications that execute iteratively may be modeled and executed in the framework. In Mercator's execution model, arbitrarily long and data-dependent flow paths may be captured in a finite dataflow graph provided the structure of the dataflow is regular with respect to the modules, as it is in iterative applications. For example, the standard DFG of our Woodstocc

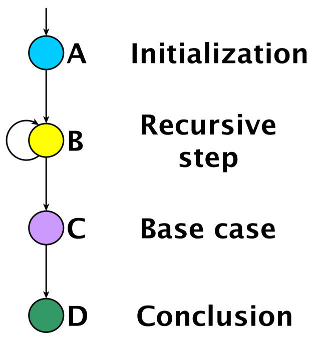


Figure 5.7: Merged DFG of a general tail-recursive application in MERCATOR.

application shown in Figure 4.2b contains a data-dependent number of levels, making it impossible to model before runtime if dataflow is managed at the granularity of nodes (as in DPN systems). However, because of the structure inherent in the DFG, the merged DFG of WOODSTOCC shown in Figure 4.4a contains only a few nodes, making data management straightforward when done at module granularity as MERCATOR does.

The dataflow of tail-recursive applications may be modeled in a merged DFG in much the same way as the dataflow of WOODSTOCC, as shown in Figure 5.7. We anticipate modeling and executing tail-recursive applications in MERCATOR according to this dataflow pattern.

5.2.7 Applications with Multiple Granularities of Operation

Most of the work in this dissertation has focused on addressing wavefront irregularity that arises as the result of input-dependent dataflow in an application. Another source of wavefront irregularity alluded to in our discussion of TLAV processing above is that which arises from multiple granularities of operation being required in the same application. In the case of TLAV applications, operations over vertices and operations over incident edges of each vertex represent two separate granularities at which SIMD execution is performed. In the case of our WOODSTOCC application, operations over candidate DNA read matches and operations over virtual suffix trie nodes represented separate execution granularities, with reads being (irregularly) grouped by suffix trie node.

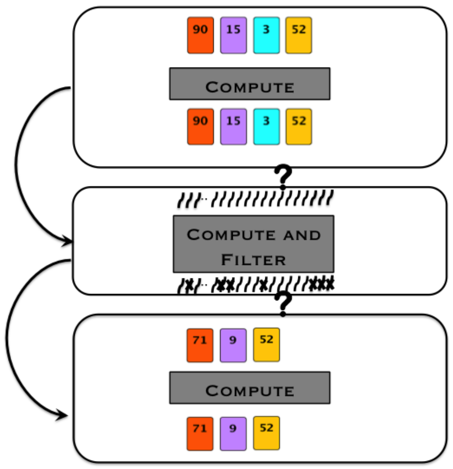


Figure 5.8: Example dataflow between three application module instances, one of which may filter data items. The top and bottom nodes operate on pods of data items, represented as rectangles labeled with their cardinalities. In practice these could represent, e.g., vertices with their number of incident edges. The middle node operates on individual data items. The mismatch in computation granularity between the modules requires one or more of them to internally implement data transformation operations such as pod explosion/repacking, and makes the data interface boundaries between them *a priori* ambiguous, as indicated by the '?'s.

Items and pods In general, individual fine-grained input *items* are grouped into *pods*, with some application modules operating on individual items and some operating on pods. Assuming module queues hold work units at their native granularities, supporting pods introduces a datatype mismatch between module queues, as some queues will store individual items and some will store pods. This mismatch must be rectified at some point during execution to expose work to each module at its native granularity. Executing an application with multiple granularities of computation efficiently, then, requires dynamic thread remapping to maintain utilization when operating on both pods and items, and careful data management to expose work appropriately but keep all items grouped into their respective pods throughout execution. Figure 5.8 illustrates these challenges.

Explosion and collection modules We propose implementing generalized versions of the *explode* and *collect* operations used in our Woodstocc application to address the irregularity caused by multiple granularities of computation. These operations will be implemented as two special infrastructure modules in the MERCATOR framework and will operate as follows:

1. **Explode module**: given a queue of pods and a target number of data items to extract, this module will operate in two steps. (1) *pull work:* Mark the maximum number of consecutive pods whose cumulative number of items is less than the target number. (2) *explode indices:* Given the marked set of pods with counts of how many (densely-stored valid) data items are in each, calculate the target pod and item indices for each worker thread.

2. **Collect module**: given a worklist of items associated with pods and an indicator for which pod each item is associated with, this module will operate in two steps: (1) *collect indices:* Given an additional validity indicator variable for each item, densely store the valid items back in their pods. (2) *push work:* Given a set of pods, add the pods with at least one valid item back to the queue.

Figure 5.9 shows the application modules from Figure 5.8 augmented with appropriate infrastructure module instances to solve the mismatched granularity problem, and Figure 5.10 isolates those instances and describes their operations in more detail.

We developed efficient concurrency-safe implementations of the explosion and collection operations for Woodstocc using parallel-scan primitives, and we anticipate that these implementations will generalize well.

For scheduling and execution, explode and collect module instances will be inserted into the application's (conceptual) dataflow graph between the nodes they connect and will be treated identically to the original application modules. To illustrate this concept, Figure 5.11 shows an example application DFG annotated with the native execution granularity of each of its modules, and Figure 5.12 shows the example dataflow graph and possible worklist states for this application augmented with explode and collect modules.

The infrastructure modules we have described are novel in the context of existing data structures used in streaming application frameworks; we refer the interested reader to Section 5.2.8 as an extended footnote for details.

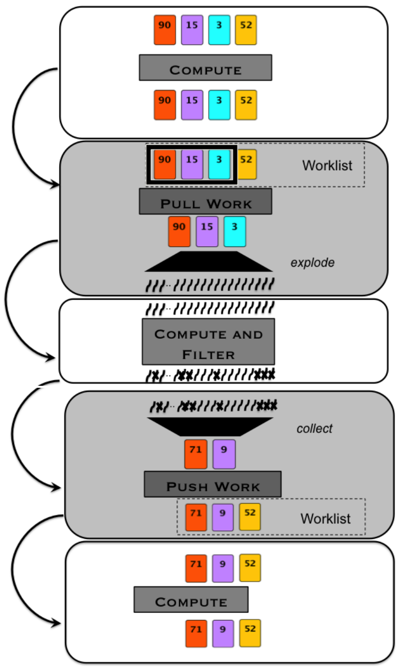


Figure 5.9: Dataflow between the three application module instances from Figure 5.8 (white), but with special infrastructure module instances (grey) inserted between them to implement item explosion and repacking. These infrastructure modules allow the application modules to abstract from data transformations and operate at their native granularities.

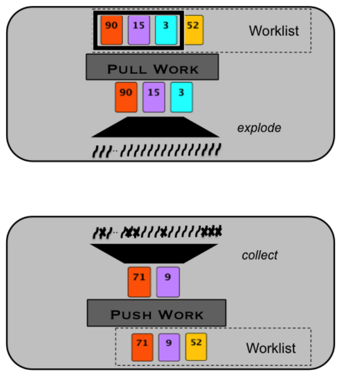


Figure 5.10: Infrastructure module instances isolated from Figure 5.9, with operations shown assuming 128 threads assigned to operate on data items at one per item. The 'PULL WORK' phase selects the maximum number of pods that may be pulled from the head of the queue which collectively contain fewer than 128 items. In the '*explode*' phase, each of the 128 threads self-discovers which pod contains its corresponding data item and the index of that item within the pod. This information will be exposed to the downstream application module, allowing its threads to directly operate on data items. In the '*collect*' phase, non-filtered items are stored compactly within their pods. The 'PUSH WORK' phase pushes pods containing unfiltered items back onto the queue.

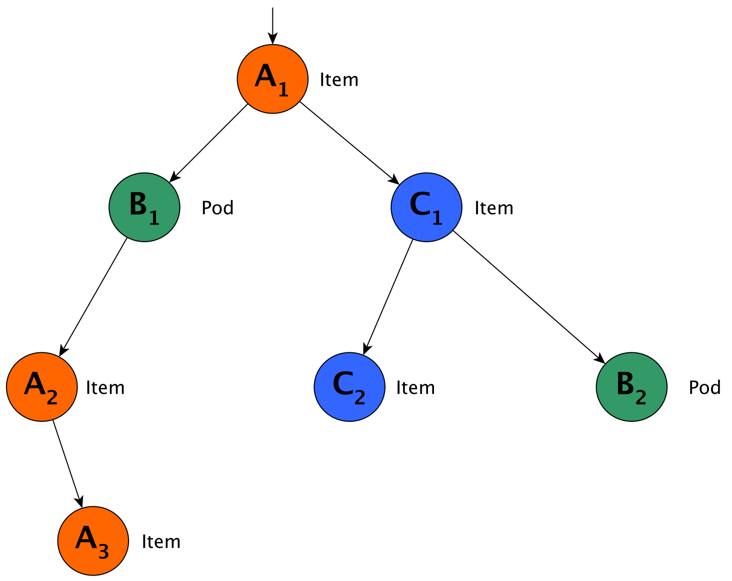


Figure 5.11: Dataflow graph for an example streaming application composed of three modules A, B, and C, each multiply instantiated. Nodes have been annotated to show the native execution granularity of their modules (item or pod).

5.2.8 Footnote: Infrastructure Module Novelty

Structures for holding streaming data in existing GPU frameworks do not fully support the operations necessary for efficient data item filtering and explosion/repacking.

The general task parallel system of Tzeng et al. described in [122] uses a global work queue with locks to allow safe access by many warp-sized workers. While this system does implicitly support the notion of operating on data at multiple granularities by assuming warps will cooperate on a task's work as a SIMD unit, the queue itself does not support the semantics of pod unpacking/repacking, nor does it allow for pushes and pulls of more than a single task at a time.

In [11], Belviranli et al. introduce a task parallel system that uses a central scheduling kernel to manage a (lock-free) worklist in parallel. However, as in [122], each warp-sized worker is assigned a single task at a time, with no mechanisms for adjusting assignments based on work granularity within a task or tasks of varying size.

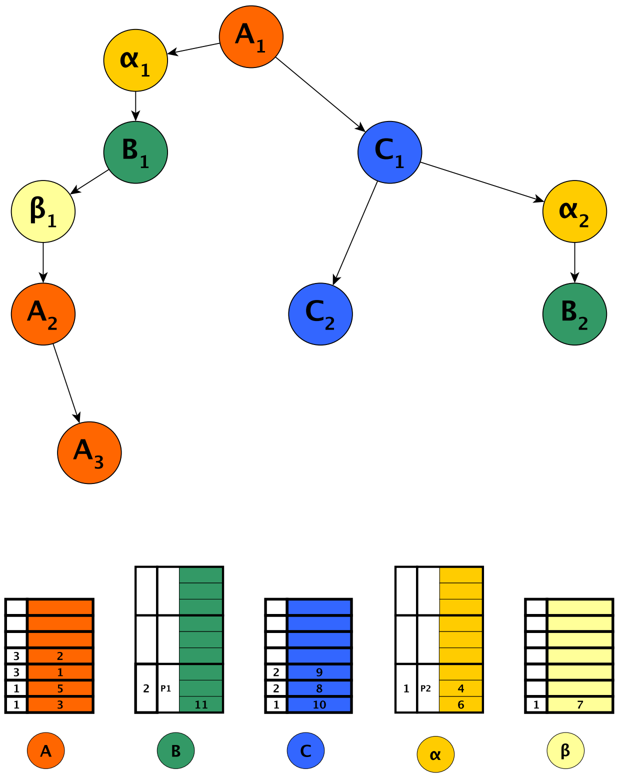


Figure 5.12: Dataflow graph from Figure 5.11 augmented with explode and collect nodes (Greek labels) to manage data between standard application nodes. For example, nodes of type α collect data items into their constituent pods, and nodes of type β explode items out of pods for individual processing. Data granularity is reflected in the queue storage shown: individual items are represented by index number, and pods of items are represented by 'P' followed by a pod number. Queues for infrastructure modules are handled identically to those for application modules by the Mercator execution system.

Synchronous Data Flow systems such as the adaptations of StreamIt for GPUs created by Udupa [124] and Hagiescu [46] support filtering but at a known and constant data rate, and so their data structures need not support operations such as sparse vector compaction which are crucial for performance in the presence of arbitrary filtering.

Nasre et al.'s framework [79] for executing morph algorithms does support the addition or deletion of arbitrary task graph nodes (i.e., filtering) from per-thread queues, but no mention is made of pod operations, and indeed, frequent additions and deletions of work cause degraded performance due to fragmentation.

OpenACC [91] automates GPU memory allocation and data movement between host and device but does not provide any specific data structures such as a worklist or task queue. CUDA-NP [133] is designed to facilitate work at multiple granularities ("dynamic parallelism") and adds automated data communication between 'master' and 'slave' threads, but does not introduce any special support for filtering or pods.

Data structures supporting compaction of sparse vector elements appear in the work of Ren et al. [103], which introduces a framework for executing irregular tree traversals with arbitrary filtering on a SIMD CPU architecture. However, these structures do not directly support pod operations.

The graph processing system introduced by Khorasani et al. in [54] does employ explosion and compaction techniques for graph problems very similar to our WOODSTOCC operations. However, this system conforms to the the vertex-centric graph processing paradigm and is not generalizable to a broader class of applications as MERCATOR is.

Data management in the GPU execution model proposed by Orr et al. in [92] shares many characteristics with MERCATOR's queue management system: it organizes data queues by function, as our modified FC model does, and pulls items off the queues in SIMD parallel. However, it requires hardware changes in order to be implemented and does not support pod operations.

Although none of the systems described in these related works efficiently addresses both generalized pod operations and arbitrary filtering, their techniques for solving one or the other of these problems provide possible future enhancements for our explode and collect modules.

Appendix A

Mercator User Manual

In Chapter 3 we presented an overview of the Mercator framework, including the user interface, but did not present details of the interface. To give a full picture of the framework, we now present the User Manual in its entirety. The latest version of the manual may be found at `http://sbs.wustl.edu/pubs/MercatorManual.pdf`, and Mercator source code may be obtained from the author upon request (svcole@wustl.edu) .

A.1 Introduction

Welcome to the Mercator user manual! Mercator is a CUDA/C++ system designed to assist you in writing efficient CUDA applications by automatically generating significant portions of the GPU-side application code. We hope you find it helpful; please feel free to contact the authors with any questions or feedback.

A.1.1 Intended Use Case: Modular Irregular Streaming Applications

Mercator facilitates the development of applications that are

- *streaming*: they take many small data items as inputs, process them once, and output or discard them;

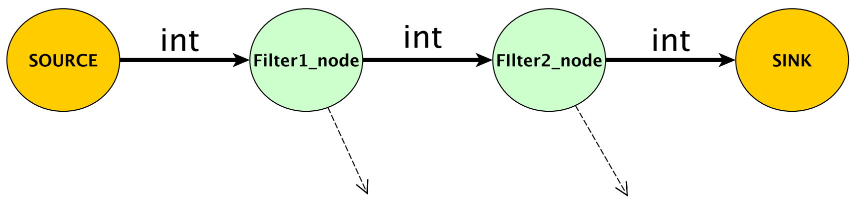


Figure A.1: Example Data Flow Graph of an application with two filtering nodes that each accept integers and output integers. Dashed lines indicate that input items may be dropped by the filtering nodes, which makes the data flow of this application irregular. The `SOURCE` and `SINK` nodes shown represent input generation and output collection in the application.

- *modular*: they may be described by a (constrained) Data Flow Graph of processing steps (i.e., nodes) connected by directed data flow edges; and
- *irregular*: any node may filter an input item or replace it with new output item(s) to send downstream, and the data flow triggered by any given input is unknown at compile time.

mtr's support for irregularity distinguishes it from systems that process Synchronous Data Flow (SDF) applications, in which the I/O rate(s) of each node are known *a priori*.

Figure A.1 shows the DFG of a simple filtering application.

A.1.2 Mercator Application Terminology

We use the following terminology to describe the topology of a Mercator application:

- A *module type* (or just *module*) is an entity whose type is defined by its function (code).
- A *module instance* is one occurrence of a module in an application's DFG (i.e. one *node*).
- An *edge* represents data flow between two module instances. Note that due to their support for irregular data flow, the exact data movement cardinalities of edges are not necessarily known at compile time.

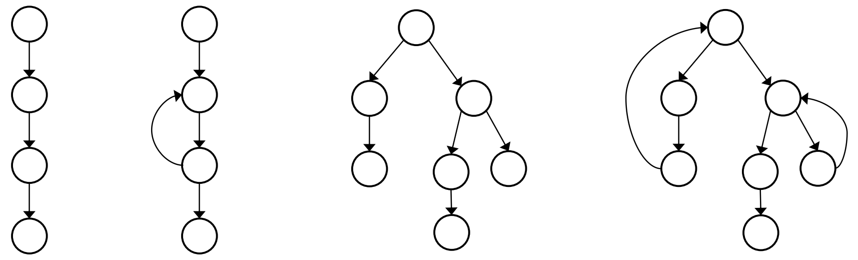


Figure A.2: Examples of valid MERCATOR application topologies, including pipelines and trees with or without limited back edges.

- Data inputs flow into the application from a single `source node`, and outputs may be emitted to one or more `sink nodes`.

MERCATOR supports any modular streaming application whose topology obeys the following constraints:

1. Each instance may have multiple output edges but only a single input edge (not counting any *back edges*; see below).
2. While input items may trigger the generation of multiple output items from a node, a maximum number of possible output items generated for each input item is known at compile time.
3. Back edges may exist between a node and one of its ancestors, creating a loop. Self-loops are permitted; overlapping or nested loops are not. 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.

Examples of valid application topologies are shown in Figure A.2.

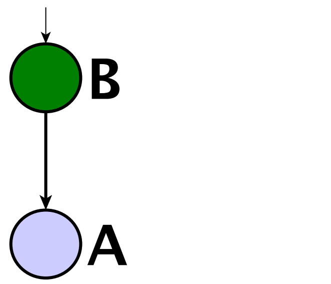
We use the following terminology to describe additional aspects of MERCATOR:

- Any code generated by MERCATOR is considered *system* code, while any code supplied by an application programmer (the *user* of the system) is considered *user* code.

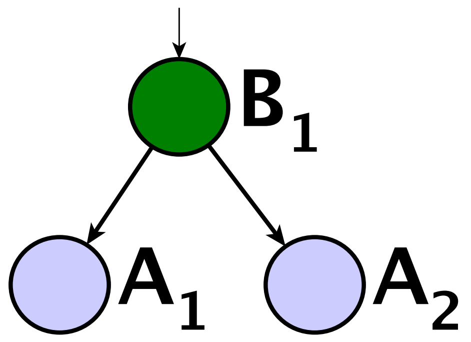
- User-supplied *parameter data* may be associated with MERCATOR DFG objects at three granularities: a global set of data, module-type-specific data, and node-specific data. This allows nodes, module types, and the overall application to maintain state during execution.

When implemented in CUDA, the code for each module type is a separate function, with module instances of the same type sharing function code but operating on different input sets with potentially different parameter values. This sharing of code between instances is important in the CUDA context where execution happens in wide SIMD. In fact, one key contribution of MERCATOR's execution model is to group work by module type rather than by node to take maximal advantage of CUDA's wide-SIMD execution.

Example DFGs labeled with MERCATOR module types and instances are shown in Figure A.3.



(a) DFG with two instances, each of a different module type indicated by its label and color.



(b) DFG with three instances of two module types. Subscripts in the type label distinguish instances of the same type.

Figure A.3: Example DFGs illustrating MERCATOR topology elements. `SOURCE` and `SINK` nodes have been omitted for clarity. Note that when executing the graph in (b), MERCATOR may fire instances A_1 and A_2 simultaneously since they are of the same module type.

A.1.3 System Overview and Basic Workflow

Given the terminology in the previous section, the application support provided by MERCATOR may be stated as follows: the MERCATOR *system* supplies CUDA code for a DFG's *edges* and provides an interface between the *edges* and *nodes*, enabling a *user* to generate a fully-functional CUDA implementation of a DFG by supplying code for its *module types* only. We refer to this implementation as a 'MERCATOR application' or just 'application,' even though it is not a full runnable CUDA program without host-side driver code that launches the device-side MERCATOR application. MERCATOR provides wrapper functions that perform these kernel calls and exposes the wrapper functions to the user via a CPU-side (host-side) API.

The user interacts with a MERCATOR application via this host-side API, which in addition to the function calls that initialize and run the MERCATOR application contains functions for providing inputs and gathering outputs from it. A MERCATOR application is therefore deployed in a fully user-defined context.

In order to generate edge code, MERCATOR must have information about an application's topology, which the user provides to the system in a well-defined specification format. In order to maintain a consistent interface between nodes and edges, MERCATOR provides the user with function headers for each module type in the application.

An overview of the MERCATOR workflow is shown in Figure A.4. Each phase of the workflow is described in the system documentation that follows.

A.2 MERCATOR System Documentation

In the following instructions, we highlight the user's responsibilities and the interface between the user and the MERCATOR system.

A.2.1 Initial Input

We begin with the user's initial input to the system, which consists of `#pragma` statements specifying the application's name, topology, module type requirements, and user-defined parameter data types.

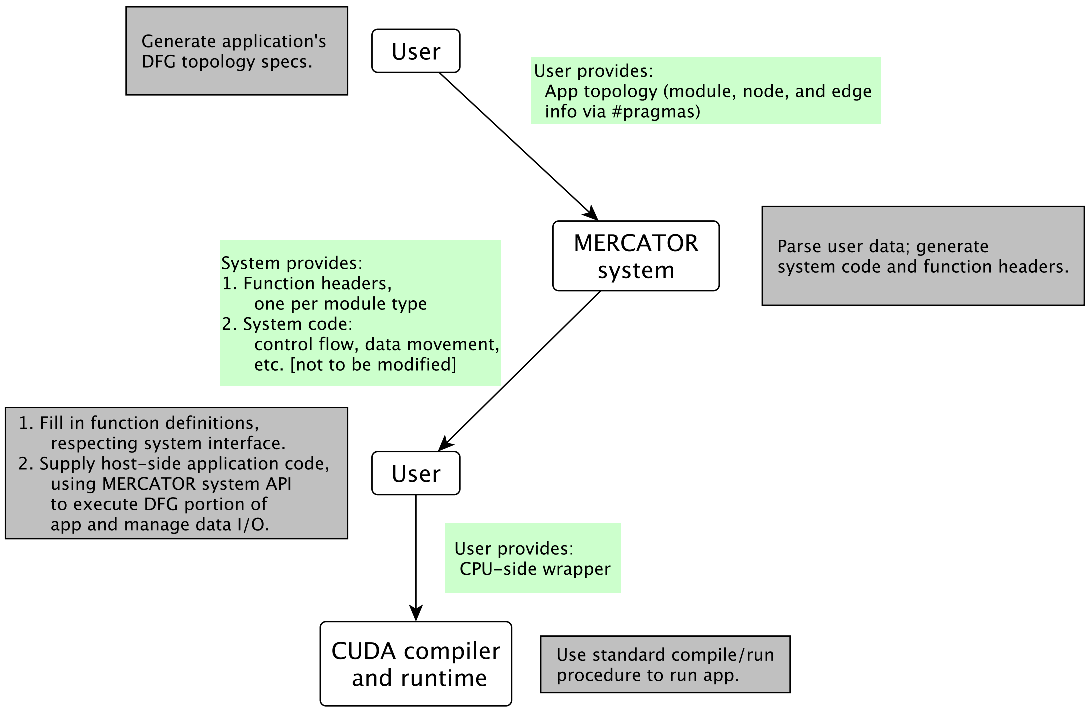

Generate application's DFG topology specs.
User
User provides: App topology (module, node, and edge info via #pragmas)
MERCATOR system
Parse user data; generate system code and function headers.
System provides: 1. Function headers, one per module type 2. System code: control flow, data movement, etc. [not to be modified]
1. Fill in function definitions, respecting system interface. 2. Supply host-side application code, using MERCATOR system API to execute DFG portion of app and manage data I/O.
User
User provides: CPU-side wrapper
CUDA compiler and runtime
Use standard compile/run procedure to run app.

Figure A.4: Workflow for writing a program incorporating a MERCATOR application.

Application name The name of an application is provided by the user in a single *app specification* statement that must conform to the following EBNF grammar. Note that here and in all following specifications, we use the shorthand `<typename>` to stand in for the set of the names of all CUDA-valid types, including user-defined types.

```
   <module-spec>  =  #pragma mtr application <appname>
                     { "<" <parameter-type> ">" }
       <appname>  =  <string>
<parameter-type>  =  <typename>
```

Here `<appname>` is a string representing the name of the application and `<parameter-type>` is the type name of (optional) parameter data to be associated globally with this application.

Module type specification Information about each module type in the application– which may be multiply instantiated in its Data Flow Graph– is provided by the user as a *module specification*. Specifications for user-defined module types must conform to the following EBNF grammar.

```
      <module-spec>  =  "#pragma mtr module" <module-name>
                         { "<" <parameter-type> ">" } "(" <input-stream> "->"
                        <output-stream>{"," <output-stream> } "|"
                        <elements> ":" <threads> ")"
     <input-stream>  =  <io-type> <num-inputs>
    <output-stream>  =  <stream-name> "<" <io-type> ">"
                        [":" <outputs-per-input>]
   <parameter-type>  =  <typename>
       <num-inputs>  =  "[" <int> "]"
<outputs-per-input>  =  <int> | ("?" [<int>])
      <module-name>  =  <string>
      <stream-name>  =  <string>
          <io-type>  =  <typename>
         <elements>  =  <int>
          <threads>  =  <int>
```

Explanation:

The specification consists of three main parts: a description of the data type and range of item cardinalities for each input and output stream, and the work assignment ratio. Notes on specific symbols follow.

`<parameter-type>` : type name of (optional) parameter data to be associated with this module type.

`<num-inputs>` : maximum number of input elements executed simultaneously upon each module firing. This does not limit the total number of input elements executed per module firing, only the number executed simultaneously in SIMD.

`<outputs-per-input>` : number of output elements produced per input element. A “`?`” indicates that the number of outputs per input is *a priori* unknown but bounded above by the integer specified in `<outputs-per-input>`. The absence of a “`?`” indicates a fixed known data rate. If no integer is provided, the number of outputs per input is assumed to be 1; for example, “`?`” is equivalent to “`?1`”.

`<module-name>` : chosen by user.

`<stream-name>` : chosen by user. Required to disambiguate output streams.

`<elements> : <threads>` : ratio of input elements to threads for processing.

`<num-instances>` : total number of instances of this module type in application's dataflow graph

When more output elements are produced than inputs consumed per firing, outputs are produced on a per-element basis; that is, each input element leads to the production of (up to) a fixed uniform number of output elements. This uniform bound is the integer specified in `<outputs-per-input>`.

The current version of MERCATOR only supports an `<elements>:<threads>` ratio with `<elements>` = 1. In other words, multiple threads may be assigned the same input item during execution of a module, but one thread may not be assigned multiple input items.

Examples:

1. Vector addition module:

```
struct VectorPair {
 Vector a, b;
 };
#pragma mtr module addVec(VectorPair[128] ->
outStream<Vector> | 1:1)
```

   Interpretation: The module processes up to 128 input elements per firing, each input element being a struct containing two vectors and each output element being a single vector. The absence of a specified number of outputs per input indicates that each input produces a single output, no matter how many inputs are processed by a given firing. The module assigns one thread to each input element, which implies that the entire element-wise sum of the two vectors is computed by a single thread.

2. Vector addition module, different mapping ratio:

```
#pragma mtr module addVec(VectorPair[128] ->
          outStream<Vector> | 1:4)
```

   Interpretation: Here we assume the same definition of a `VectorPair` as above. This specification is identical to that of the previous example except that the module implementation assigns four threads to process each input vector pair. In this case, the user may arbitrarily divide the work of computing the vector sum among these four threads in the module function definition.

3. Filtering module:

```
#pragma mtr module evalThreshold<StateInfoStruct>(myItem_t[128] ->
          outStream<myItem_t> : ?  | 1:1)
```

Interpretation: The module processes up to 128 input elements per firing, each element being of a user-defined type, and produces either 0 or 1 output elements per input element. The module has associated with it a parameter data object of type `StateInfoStruct`.

4. Cloning module, single output stream:

```
#pragma mtr module cloneItem(myItem_t[128] ->
        outStream<myItem_t> : 4 | 1:1)
```

   Interpretation: The module processes up to 128 input elements per firing, each element being of a user-defined type, and produces 4 output elements per input element.

5. Cloning module, variable output production:

```
#pragma mtr module cloneItem(myItem_t[128] ->
        outStream<myItem_t> : ?4 | 1:1)
```

   Interpretation: Identical to previous example except that each input element produces between 0 and 4 output elements. In this case the MERCATOR infrastructure allocates space for the maximum number of output elements (i.e. 512) for each firing, and the user's module function definition is responsible for respecting this limit.

6. Module that combines cloning and filtering, multiple output streams:

```
#pragma mtr module cloneFilterItem(myItem_t[128] -> outStream1<myItem_t>,
              outStream2<myItem_t>, outStream3<myItem_t> :?,
                    outStream4<myItem_t> :2   | 1:1)
```

   Interpretation: The module's output behavior varies by stream: it sends a single element to `outStream1` and `outStream2`, sends 0 or 1 elements to `outStream3`, and sends 2 elements to `outStream4` per input element per firing.

Special types: `SOURCE` and `SINK` In addition to the application module types designed by the user, Mercator provides two module types that must be included in an application: `SOURCE` and `SINK`. The single `SOURCE` module type serves as the data entry point into the application, taking input from a user-facing input buffer (to be discussed later) and feeding it to the conceptual root node of the application's DFG. The `SOURCE` module type is therefore always instantiated exactly once, and its input type and (single) output type always match the input type of the DFG's conceptual root node.

The `SINK` module type serves as a data exit point from an application, taking final output from user-defined nodes and passing it to user-facing output buffers (to be discussed later). Since the application's DFG may contain multiple nodes that produce final output, and each output stream may be of a different type, the `SINK` module type is parameterized by this output type and describes a category of types rather than a monolithic type. Each `SINK` type may be multiply instantiated, and its input and output type(s) will all match the final output type it is designed to pass through.

Although the implementation of `SOURCE` and `SINK` module types is part of Mercator's system code, the user must include `SOURCE` and `SINK` modules in the input specifications so that the system knows their input and output stream information and so that they may be appropriately included in the application topology. Since their functionality is limited, however, the amount of information required from their specification is considerably less than from user-defined module types, and their specifications are therefore simpler.

`SOURCE` and `SINK` module types must conform to the following EBNF grammar:

```
<source-module-spec>  =  "#pragma mtr module " ( "SOURCE" | "SINK" )
                          "<" <io-type> ">"
           <io-type>  =  <typename>
```

Here `<io-type>` is the input or output type processed by the module. As stated above, there can be only one `SOURCE` module specified for an application, but there may be multiple `SINK` modules specified if an application produces outputs of multiple types.

Examples:

1. `SOURCE` module:

```
#pragma mtr module SOURCE<int>
```

   Interpretation: The `SOURCE` module, and hence the conceptual root of the application's DFG, accepts inputs of type int.

2. `SINK` modules:

```
#pragma mtr module SINK<int>                    (A.1)
#pragma mtr module SINK<MyItem>                 (A.2)
```

   Interpretation: Two types of outputs may be produced by the application: integers and the user-defined type MyItem.

Node specification Information about each module instance (i.e., node) in the application is provided by the user as a *node specification*. Node specifications must conform to the following EBNF grammar:

```
         <node-spec>  =  "#pragma mtr node " <node-name>
                         { "<" <parameter-type> ">" } ":" <module-name>
         <node-name>  =  <string>
    <parameter-type>  =  <typename>
  <module-type-name>  =  <module-name>
```

Explanation:

The specification consists of two parts: a user-chosen name for the module instance, and the name of the module type of which it is an instance.

`<node-name>` : user-chosen name for this node that will be used to identify it throughout the application.

`<parameter-type>` : type name of (optional) parameter data to be associated with this node.

`<module-type-name>` : name of the module type of which this node is an instance; must match the `<module-name>` of some previously-declared module specification.

Example:

In the following example, the relevant module specifications are given, followed by the node specifications describing their instantiations.

```
filterStage1(myItem_t -> fs1outStream<myItem_t> :  ?  | 1:1)

filterStage2(myItem_t[128] -> fs2outStream<myItem_t> :  ?  | 1:1)

node1<ThresholdStruct> :  filterStage1

node2<ThresholdStruct> :  filterStage1

node3<ThresholdStruct> :  filterStage2
```

Interpretation: `node1` and `node2` are instances of the module type `filterStage1`, and `node3` is an instance of the module type `filterStage2`. Each node has an associated parameter data object of type `ThresholdStruct`.

`SOURCE` and `SINK` nodes The specification of `SOURCE` and `SINK` nodes must obey the following EBNF grammar:

```
<source-node-spec>  =  "#pragma mtr node " <node-name> ":SOURCE"
  <sink-node-spec>  =  "#pragma mtr node " <node-name> ":SINK<"<io-type>">"
       <node-name>  =  <string>
         <io-type>  =  <typename>
```

Explanation:

Module instances (nodes) of the `SOURCE` and `SINK` module types are specified in the same way as nodes of user-defined module types, with the keyword '`SOURCE`' being used as the (unique) `SOURCE` module type, and the (possibly not unique) `SINK` module type being specified similarly but with its output type included.

Example:

In the following example, the relevant module specifications are given, followed by the node specifications describing their instantiations.

```
#pragma mtr module SOURCE

#pragma mtr module SINK<int>

#pragma mtr module SINK<MyItem>

...

#pragma mtr node sourceNode :  SOURCE

#pragma mtr node resultNode1 :  SINK<MyItem>

#pragma mtr node resultNode2 :  SINK<MyItem>

#pragma mtr node resultNode3 :  SINK<MyItem>

#pragma mtr node resultNode4 :  SINK<int>
```

Interpretation: `sourceNode` is the single source node, the nodes `resultNode1-3` produce application results of type `MyItem`, and the node `resultNode4` produces application results of type `int`. The specifications of all user-defined module types and nodes has been omitted in this example.

Edge specification An *edge specification* describes a dataflow relationship between two nodes; i.e., it represents an edge in the application's Data Flow Graph. Edge specifications must conform to the following EBNF grammar:

```
    <edge-spec>  =  <us-node> "::" <output-stream>
                    "->" <ds-node>
      <us-node>  =  <node-name>
      <ds-node>  =  <node-name>
<output-stream>  =  <stream-name>
```

Explanation:

`<us-node>` : upstream node in connection; must match the `<node-name>` in a previously-declared node specification.

`<ds-node>` : downstream node in connection; must match the `<node-name>` in a previously-declared node specification.

`<output-stream>` : stream of `<us-node>` to be connected to input of `<ds-node>`; must match a `<stream-name>` in the specification of the module type of which `<us-node>` is an instance.

As a result of our single-input assumption for modules and prohibition of nested back-edges, each module may appear as the `<ds-module>` in only a single edge specification, except in the case of a back edge, in which it appears as the `<ds-module>` exactly twice: once with its parent node and once with its back-edge origin.

Note that since nodes of `SOURCE` and `SINK` module types are named in the same way as those of user-defined module types, edge specifications containing them need not have a distinct form. However, a user is expected to include the following `SOURCE` and `SINK` edge specifications in every application:

1. A single edge specification that explicitly connects the `SOURCE` node to the conceptual DFG root node.
2. An edge specification that explicitly connects each user-defined node that produces final application output to its associated `SINK` node.

Examples

In the following examples, the relevant module and node specifications are given, followed by the edge specification(s) describing the dataflow between the nodes.

1. Connection between two filtering modules:

```
filterStage1(myItem_t -> fs1outStream<myItem_t> :  ?  | 1:1)

filterStage2(myItem_t[128] -> fs2outStream<myItem_t> :  ?  | 1:1)

node1 :  filterStage1

node2 :  filterStage2

node1::fs1outStream -> node2
```

Interpretation: `node1` and `node2` are instances of module types `filterStage1` and `filterStage2` respectively. The elements output by `node1` on its `fs1outStream` stream are placed on the input worklist of `node2`. *Note:* It is assumed that `filterStage1` and `filterStage2` instantiate different filter code, rather than the same filter code parameterized differently (e.g., a simple threshold filter with different threshold values). An example of such a topology exists in the Viola-Jones filter cascade, in which different filters assay different features of a candidate face image.

2. Connection requiring named streams for disambiguation:

```
splitItems(myItem_t[128] -> siOutStream1<int>:2,siOutStream2<int>:3 | 1:1)

filterItems(int[128] -> fiOutStream<int>:?  | 1:1)

node1 :  splitItems

node2 :  filterItems

node1::siOutStream1 -> node2
```

Interpretation: The elements in the `siOutStream1` output stream of `node1` are placed on the input worklist of `node2`. *Note:* In this case naming module output streams enables

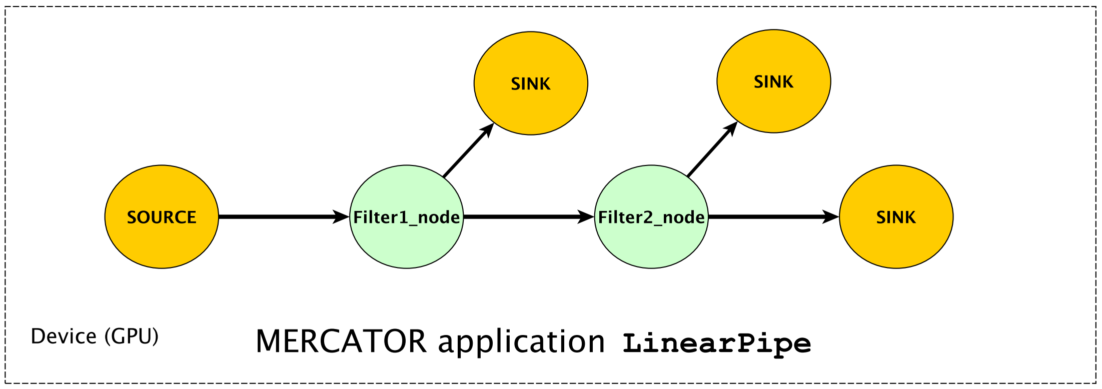


Figure A.5: Diagram showing the DFG of the Mercator filtering application named `LinearPipe` described in the text. Note the existence of multiple `SINK` nodes.

disambiguation; both `siOutStream1` and `siOutStream2` have the same type, so inferring which stream to use to connect `node1` and `node2` would not be possible without a stream ID.

Example: `LinearPipe` application

The set of module type, node, and edge specifications for an example Mercator application are shown in the Appendix in Listing A.1. This application, which we refer to as "`LinearPipe`," implements the linear pipeline of filter nodes shown in the examples in Section 1 of this manual, with the additional property that all nodes in the pipeline may produce final output. This allows the application to expose elements that it filters out as well as those that pass through the filters at each stage. A diagram of the application's topology is shown in Figure A.5. This application will serve as a running example for other pieces of user-generated code in the sections that follow.

A.2.2 System Feedback

Based on the user's initial input, the Mercator system will generate and return CUDA code of two types: system infrastructure code and module function headers.

The system infrastructure code is not intended to be modified by the user, but along with the user's function definitions, it constitutes an essential part of the Mercator application to be integrated into the user's final program code.

Module function headers

Mercator provides a set of module function headers written in CUDA that implicitly contain the user's interface to the Mercator infrastructure code. One header is produced for each module type, and the function bodies are to be filled in by the user. The details of writing function definitions are presented in the next section. Module function headers have the following format:

```
void run (
              <input-type> inputItem,
              int nodeIdx)
```

Explanation:

`<input-type>` : type of elements in module's input stream; determined by module specification previously given by the user.

`inputItem` : input element to be operated on by user code.

`nodeIdx` : an internally-generated tag for the current node being executed; used for API function calls in module definition.

Since Mercator's scheduling strategy involves firing multiple instances of the same module simultaneously, the `<instance-type>` as well as the `inputItem` may vary across threads executing the same function call.

A.2.3 Full Program

Module run function definitions If there are more threads present on the GPU device than there are items to be operated on, Mercator ensures that the "extra" threads do not execute the `run()` function call. This ensures that each thread that executes the function will have a valid input item passed in through the `inputItem` parameter; however, it also implies that to avoid undefined behavior, module definitions must be written with no CUDA synchronization calls such as `__syncthreads()`.

The remaining paragraphs in this section describe Mercator API functions accessible to the user inside `run()` functions.

Accessing user-defined parameter data

Within the `run()` function, users may access parameter data using the following API function calls:

- Application level: `get_appUserData()`
- Module type level: `this->get_userData()`
- Module instance level: `this->get_nodeUserData(nodeIdx)`

Here '`nodeIdx`' is the tag given to the user in the `run()` function header and is passed to the `get_nodeUserData` function verbatim.

Writing output

Output may be written from within the `run()` function using the following API call:

```
node->push(item, nodeIdx, Out::streamName)
```

where `item` is the output item to be written, `nodeIdx` is the tag parameter of the `run()` function, and `streamName` is the name given to the desired output stream by the user in the initial input specification. Here we reiterate that although many threads may be writing to the same output stream, no synchronization code should be included in the `run()` function; MERCATOR's system code ensures the integrity of the output streams.

Multiple threads per input element

When multiple threads are assigned to each input element (as dictated by the `<elements>:<threads>` ratio in the module type input specification), the following API call may be used to access a thread's offset relative to all threads assigned to the same input element:

```
eltOffset(threadIdx.x)
```

This call will return an `int`, which can be used to process the appropriate partition of a shared input. For example, if many threads are assigned the same input string of which they are intended to process consecutive substrings, each thread may find its starting point relative to the input string with a call to `eltOffset` .

Multiple output elements per input element

When multiple output elements may be written to an output stream for each input consumed, either by a single thread or in the case of multiple threads per input element, an output slot within the output stream must be specified for each `push` operation. This is done with an (optional) third parameter to the function, so that the function call will be of the following form:

```
node->push(item, ‘nodeIdx’, Out::streamName, slotIdx)
```

where `slotIdx` is the desired slot within the output stream relative to all possible outputs generated by the current input.

For example, if each thread writes four outputs to the same stream during each `run()` function execution, those four writes may take the following form:

```
node->push(item, ‘nodeIdx’, Out::streamName, 0) // the ’0’ parameter is optional
node->push(item, ‘nodeIdx’, Out::streamName, 1)
node->push(item, ‘nodeIdx’, Out::streamName, 2)
node->push(item, ‘nodeIdx’, Out::streamName, 3)
```

In the case of multiple threads being assigned to the same input element, the `eltOffset` function may be used to specify an output slot. For example, if multiple threads are assigned to the same input element and each thread writes a single output during each `run()` function execution, that write may take the following form:

```
node->push(item, ‘nodeIdx’, Out::streamName, eltOffset(threadIdx.x))
```

Multiple input elements per thread

When multiple elements are assigned to each thread (as dictated by the `<elements>:<threads>` ratio in the module type input specification), the module `run()` function header changes to give the user access to all items assigned to the calling thread from inside the function. The revised header signature is:

```
void run (
              <input-type> *inputItems,
              unsigned char *nodeIdxs)
```

Items and tags are stored in arrays and may be accessed using array notation; e.g., `inputItem[0]` and `nodeIdxs[0]` hold the first item and tag assigned to the calling thread respectively.

Within the `run()` function, the following API call may be used to access the total number of elements assigned to a given thread:

```
this->itemsPerThread(threadIdx.x)
```

This call will return an `int`, which can be used to loop through the inputs and node tags passed in to the `run()` function. Note that the number of items assigned to a thread may not be the `numThreads` value specified in the module type input specification if the calling thread holds the (non-full) tail end of the set of inputs being executed by the module.

One possible set of function definitions for the headers produced for the `LinearPipe` application from the input specifications shown in Listing A.1 is shown in Listing A.2.

Application data I/O The MERCATOR infrastructure code and the user's module definitions may be incorporated into a host-side driver file via a single `#include` of the module header file.

Note: to pull in all MERCATOR *system header files, the file* `mercator-build-path/backend/mercator.cuh` *should also be included by every user driver file.*

The resulting CUDA code may be submitted to any CUDA compiler (e.g., `nvcc`) for compilation and subsequently executed using any CUDA runtime.

We now turn to the portions of the MERCATOR user interface that reside on the CPU. We begin with the data interface between user code and the MERCATOR application. MERCATOR provides special input and output buffer types to manage host-device data transfers, and it is these buffers with which user code interacts to move data into and out of MERCATOR. These buffer types are parameterized by the type of data they hold.

Input buffers

Input buffers are created with a capacity and, optionally, with an initial data set; the API signature for the input buffer constructor is the following:

```
InputBuffer<io-type>(int capacity, io-type* inputElements)
```

where `inputElements` is an optional parameter.

Note: The inputElements array MUST have been allocated using CUDA's "Managed Memory"– i.e., with a call to `cudaMallocManaged()` rather than the usual `cudaMalloc()`– so that the data will be accessible on both host and device.

If the initial data set is not provided, elements may be added to the buffer one at a time using the API member call `add(nextElement)`.

Output buffers

Output buffers are created with a capacity; the API for the output buffer constructor is the following:

```
OutputBuffer<io-type>(int capacity)
```

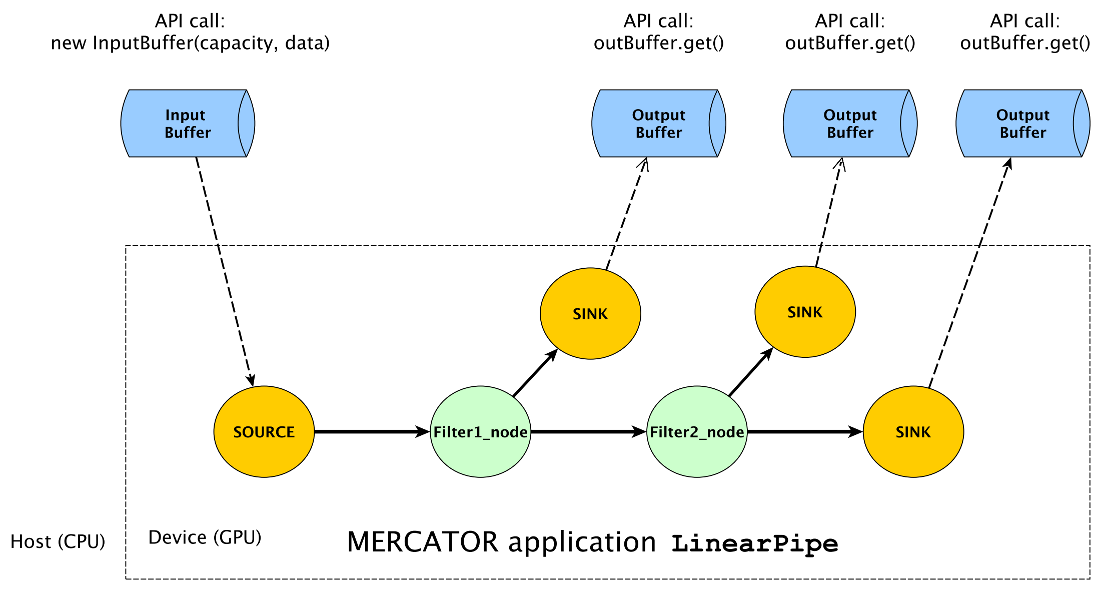


Figure A.6: Diagram showing the `LinearPipe` DFG as well as its data interface with the host via input and output buffers.

The application data stored in an output buffer is exposed the user as an array via the following member API functions:

```
io-type* OutputBuffer::get_data()
int OutputBuffer::size()
```

Data elements may also be retrieved from an output buffer one at a time with the API member call `get()` , which pops an item off the output buffer and returns it.

A diagram of the `LinearPipe` application along with its associated input and output buffers is shown in Figure A.6.

Mercator application hooks To set up and run the Mercator application from the user's host-side program, the user creates an object of the application type given in the input specification, and API member calls on that object perform the necessary application-level operations.

Note: To ensure correct memory allocation using `cudaMallocManaged()`, application objects must be created on the heap with a call to `new()`.

Instantiated objects representing each module type and module instance of the application are provided and exposed to the user; these objects have the names given to each module type and instance in the user specification, and are typed as inner classes of the user-specified application class. These objects are invoked to associate I/O buffers with `SOURCE`/`SINK` nodes and to associate user-provided (non-stream) data with module types and instances.

User-provided Data

The user may associate arbitrary data with a module, an instance, or the entire Mercator application, and this data may be accessed both from the user's host-side code and from within a module's `run()` function. Datatypes for these data must be given in the application specification file at the appropriate granularity.

Note: All user-provided data MUST be accessible in host and device memory. For non-POD data, this may be accomplished directly by the user by allocating all data using CUDA's "Managed Memory"– i.e., with a call to `cudaMallocManaged()` rather than the usual `cudaMalloc()`– or by defining the datatype to extend the `MERCATOR::CudaManaged` class and allocating the data on the heap with a call to `new()`.

User-defined data objects are bound to specific application objects with the following API calls:

- Application level: `appObject->set_userData(myAppData)`
- Module level: `appObject->ModuleTypeName->set_userData(moduleTypeNameData)`

- Node level: `appObject->set_nodeUserData(AppName::Node::NodeName, nodeNameData)`

Here `appObject` is the name of the application object created by the user; `AppName`, `ModuleTypeName`, and `NodeName` must match the names of these components given in the user's input specifications; and `myAppData`, `moduleTypeNameData`, and `nodeNameData` are user-defined data objects. `AppName::Node::NodeName` is an enum value used by Mercator to index the nodes within an application.

Inside each module's `run()` functions, the API calls to access user-defined data are given above in Section A.2.3

Buffer/node association

In order to connect the buffers to the appropriate data entry/exit points of the Mercator application– i.e., the appropriate `SOURCE` and `SINK` nodes– the user must specify which buffer gets associated with which node. This is done with an API call that operates on a node and takes the relevant buffer object as a parameter. The formats of the API member calls to set input and output buffers are the following:

```
sourceNodeObj->set_inBuffer(inBufferObj)
sinkNodeObj->set_outBuffer(outBufferObj)
```

where `sourceNodeObj`/`sinkNodeObj` are the user-supplied names of the source node and a sink node, respectively, and `inBufferObj`/`outBufferObj` are user-created input and output buffer objects. Because an application may include multiple output buffers, multiple calls to `set_outBuffer()` may be made for the same application, but only one call to `set_inBuffer()` should be made since an application includes only a single input buffer.

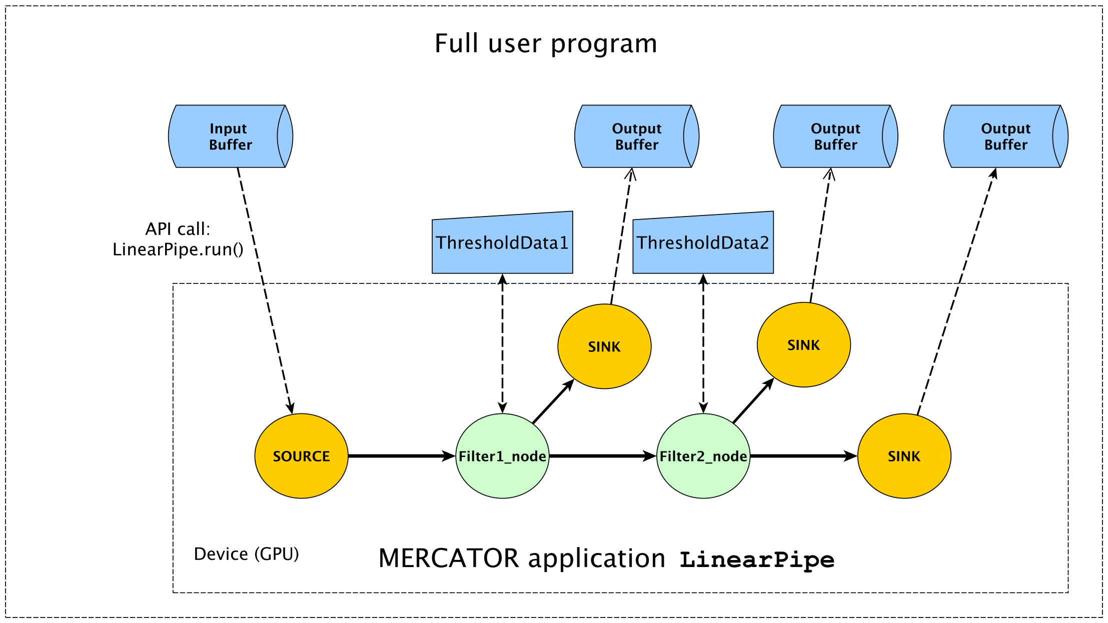


Figure A.7: Diagram showing the `LinearPipe` DFG and its interface with host-side code and data, including I/O buffers and user-defined parameter data accessible from within the modules' `run()` functions. Two of the program's output buffers have been omitted for clarity.

Running the Mercator application

Once input and output buffers have been created, the input buffer has been initialized with data, and buffers have been associated with nodes, the Mercator application may be run from a user program with the API member call `run()`. This blocking function call encapsulates the entire execution of the Mercator application, so that when it returns, the input buffer will be empty and the output buffers will contain all program outputs assuming an error-free execution on the GPU device.

Once the application has been run, data may be gathered from the output buffers using the API calls described above.

The relevant parts of the user program for the `LinearPipe` application are shown in Listing A.3, and a diagram showing the full program layout is shown in Figure A.7.

Restrictions The following restrictions on incorporating Mercator applications into user programs currently exist:

- Single-batch execution: All input and output buffers must fit into the GPU card's global memory simultaneously; multiple rounds of execution and I/O copying are not supported.
- Single-stream execution: While multiple Mercator applications may be invoked in a single user program, each application must be invoked separately. This is automatically enforced by the internal semantics of the Mercator application `run()` function API call, which executes synchronously.

A.2.4 Example Application Code

We show here all code required to create a simple filtering application in Mercator and access it from a user program.

Initial input specifications Listing A.1 shows the set of input specifications for the application.

Listing A.1: Input specification for the "LinearPipe" MERCATOR application. The application contains 2 module types in addition to the `SOURCE` type and a single `SINK` type, and consists of a linear pipeline with shunts to `SINK` nodes off of each node in the pipeline.

```
/*** Application name (with app-level datatype name). ***/
#pragma mtr application LinearPipe<FilterAppData>


/*** Module (i.e., module type) specs. ***/


// Filter 1
// Note the user-defined parameter data of type "Filter1State"
#pragma mtr module Filter1<Filter1State> (
int[10] -> accept<int>: ?4, reject<int>: ?4 | 1 : 1 )
```

```
// Filter 2
#pragma mtr module Filter2 (
int[10] -> accept<int>: ?4, reject<int>: ?4 | 1 : 4 )

// SOURCE Module
#pragma mtr module SOURCE<int>

// SINK Module
#pragma mtr module SINK<int>


/*** Node (i.e., module instance) specs. ***/
#pragma mtr node sourceNode : SOURCE
#pragma mtr node Filter1_node : Filter1

// Note the user-defined parameter data of type "Filter2ThresholdStruct"
#pragma mtr node Filter2_node<Filter2ThresholdStruct> : Filter2
#pragma mtr node sinkNodeReject1 : SINK<int>
#pragma mtr node sinkNodeReject2 : SINK<int>
#pragma mtr node sinkNodeAccept : SINK<int>


/*** Edge specs. ***/

// SOURCE -> Filter1
#pragma mtr edge sourceNode::outStream -> Filter1_node
```

```
// Filter1 -> Filter2
#pragma mtr edge Filter1_node::accept -> Filter2_node


// Filter1 -> Rejects
#pragma mtr edge Filter1_node::reject -> sinkNodeReject1


// Filter2 -> SINK
#pragma mtr edge Filter2_node::accept -> sinkNodeAccept


// Filter2 -> Rejects
#pragma mtr edge Filter2_node::reject -> sinkNodeReject2
```

Module type definitions Example module type definitions are shown in Listing A.2.

Listing A.2: Two sample module function bodies for the module types defined in Listing A.1. `Filter1` conditionally writes four values either to the "`accept`" output stream or the "`reject`" output stream; note its use of `Filter1`'s parameter data, the global application parameter data, and the use of slot indices in the last parameter of the push() function. `Filter2` conditionally writes a single element to the `accept` or `reject` stream. Note its use of the `eltOffset()` function to assign output to the correct slot, since more than one thread is given the same input.

```
////  module Filter1<UserDataExt>(int[10] -> accept<int> : ?4,
//                                            reject<int>: ?4 | 1 : 1)
__device__
void LinearPipe::DerivedModule_Filter1::run(int inputItem,
unsigned char nodeIdx)
{
int stateNum = this->get_userData()->stateNum;
if(stateNum < 6)  {
```

```
int scaler = get_appUserData()->scale;
int outputItem = inputItem * scaler;
node->push(outputItem, nodeIdx, Out::accept, 0);  // '0' optional
node->push(outputItem + 1, nodeIdx, Out::accept, 1);
node->push(outputItem + 2, nodeIdx, Out::accept, 2);
node->push(outputItem + 3, nodeIdx, Out::accept, 3);
}
else {
node->push(inputItem, nodeIdx,     Out::reject, 0);
node->push(inputItem + 1, nodeIdx, Out::reject, 1);
node->push(inputItem + 2, nodeIdx, Out::reject, 2);
node->push(inputItem + 3, nodeIdx, Out::reject, 3);
}
}

////  module Filter2(int[10] -> accept<int>: ?4, reject<int>: ?4 | 1 : 4)
__device__
void LinearPipe::DerivedModule_Filter2::run(int inputItem,
unsigned char nodeIdx)
{
if(inputItem < node->get_userData()->thresh)
node->push(inputItem, nodeIdx, Out::accept, eltOffset(threadIdx.x));
else
node->push(inputItem, nodeIdx, Out::reject, eltOffset(threadIdx.x));
}
```

Driver code Driver code for a user program that invokes the `LinearPipe` application is shown in Listing A.3.

Listing A.3: User code that creates MERCATOR I/O buffers, fills the input buffers with consecutive integers, initializes and invokes the "LinearPipe" application, and reads the results from the output buffers.

```
/**
* User program that acts as test harness
*       for MERCATOR filtering application "LinearPipe."
*
*/

#include <iostream>

#include "../../mercator.cuh"       // MERCATOR system headers
#include "userInput/LinearPipe.cuh"  // headers for run() fcns

int main()
{
////// set up input buffer
const int IN_BUFFER_CAPACITY = 128;

int* inBufferData;
cudaMallocManaged(&inBufferData,
IN_BUFFER_CAPACITY * sizeof(int));
Mercator::InputBuffer<int>* inBuffer =
new Mercator::InputBuffer<int>(inBufferData,
IN_BUFFER_CAPACITY);

////// set up output buffers
// output buffers accommodate 4 outputs/input
const int OUT_BUFFER_CAPACITY1 = 4 * IN_BUFFER_CAPACITY;
const int OUT_BUFFER_CAPACITY2 = 4 * OUT_BUFFER_CAPACITY1;
Mercator::OutputBuffer<int>* outBuffer1 =
new Mercator::OutputBuffer<int>(OUT_BUFFER_CAPACITY1);
Mercator::OutputBuffer<int>* outBuffer2 =
new Mercator::OutputBuffer<int>(OUT_BUFFER_CAPACITY2);
Mercator::OutputBuffer<int>* outBuffer3 =
new Mercator::OutputBuffer<int>(OUT_BUFFER_CAPACITY2);

////// fill input buffer
for(int i=0; i < IN_BUFFER_CAPACITY; ++i)
inBuffer->add(i);

////// create app object
// NB: must be created on heap
LinearPipe* linearPipe = new LinearPipe();

////// set node-, module-, app-level user data
FilterAppData* fullAppData = new FilterAppData();
linearPipe->set_userData(fullAppData);

Filter1State* filter1data = new Filter1State();
linearPipe->Filter1.set_userData(filter1data);

Filter2ThresholdStruct* filter2nodeData =
new Filter2ThresholdStruct(4);
linearPipe->set_nodeUserData(LinearPipe::Node::Filter2_node,
filter2nodeData);
```

```
////// associate buffers with nodes
linearPipe->sourceNode.set_inBuffer(inBuffer);
linearPipe->sinkNodeReject1.set_outBuffer(outBuffer1);
linearPipe->sinkNodeReject2.set_outBuffer(outBuffer2);
linearPipe->sinkNodeAccept.set_outBuffer(outBuffer3);

////// run main function
linearPipe->run();

/////// retrieve output
int* outData1 = outBuffer1->get_data();
int* outData2 = outBuffer2->get_data();
int* outData3 = outBuffer3->get_data();

int outData1Size = outBuffer1->size();
int outData2Size = outBuffer2->size();
int outData3Size = outBuffer3->size();

////// print contents of output buffers
for(int i=0; i < outData1Size; ++i)
printf("Buffer 1 item %d: %d\n", i, outData1[i]);
for(int i=0; i < outData2Size; ++i)
printf("Buffer 2 item %d: %d\n", i, outData2[i]);
for(int i=0; i < outData1Size; ++i)
printf("Buffer 3 item %d: %d\n", i, outData3[i]);

/////// clean up
cudaFree(inBufferData);
```

```
return 0;
}
```

References

[1] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 145–149. ACM, 2009.

[2] C. Alkan, J. M. Kidd, T. Marques-Bonet, G. Aksay, F. Antonacci, F. Hormozdiari, J. O. Kitzman, C. Baker, M. Malig, O. Mutlu, S. C. Sahinalp, R. A. Gibbs, and E. E. Eichler. Personalized copy number and segmental duplication maps using next-generation sequencing. *Nature Genetics*, 41:1061–7, 2009.

[3] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[4] Amazon Web Services. Amazon EC2 P2 Instances. `https://aws.amazon.com/ec2/instance-types/p2/`. Accessed Aug. 18th, 2017.

[5] Mehdi Amini, Béatrice Creusillet, Stéphanie Even, Ronan Keryell, Onig Goubier, Serge Guelton, Janice O. McMahon, François-Xavier Pasquier, Grégoire Péan, Pierre Villalon, and Others. Par4all: From convex array regions to heterogeneous computing. In *IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012*, 2012.

[6] Anonymous. Types of parallel machines. *BYTE*, 19(2):98+, February 1994.

[7] Arun for Beyond3D. NVIDIA CUDA Introduction. `https://www.beyond3d.com/content/articles/12/`. Accessed Aug. 18th, 2017.

[8] Ching Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 11, 2011.

[9] Lee Barford, Shuvra S. Bhattacharyya, and Yanzhou Liu. Data flow algorithms for processors with vector extensions. *Journal of Signal Processing Systems*, 87(1):21–31, Apr 2017.

[10] Jonathan C. Beard, Peng Li, and Roger D. Chamberlain. Raftlib: A C++ template library for high performance stream parallel processing. In *Programming Models and Applications on Multicores and Manycores*, 2015.

[11] Mehmet E. Belviranli, Chih-Hsun Chou, Laxmi N. Bhuyan, and Rajiv 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*, pages 29–34, 2014.

[12] Berten Digital Signal Processing. GPU vs FPGA performance comparison. *Whitepaper*, 2016.

[13] Jochen Blom, Tobias Jakobi, Daniel Doppmeier, Sebastian Jaenicke, Jörn Kalinowski, Jens Stoye, and Alexander Goesmann. Exact and complete short read alignment to microbial genomes using GPU programming. *Bioinformatics*, 27(10):1351–1358, March 2011.

[14] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

[15] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[16] Leo Breiman. Random Forests. *Mach. Learn.*, 45(1):5–32, October 2001.

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

[18] Joseph Buck and Edward A Lee. The token flow model. In *Data Flow Workshop*, pages 267–290. Citeseer, 1992.

[19] Jeremy D. Buhler, Kunal Agrawal, Peng Li, and Roger D. Chamberlain. Efficient deadlock avoidance for streaming computation with filtering. In *PPOPP*, pages 235–246, 2012.

[20] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on GPUs. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 141–151. IEEE, 2012.

[21] R. D. Chamberlain, J. Buhler, M. Franklin, and J. H. Buckley. Application-guided Tool Development for Architecturally Diverse Computation. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 496–501, New York, NY, USA, 2010. ACM.

[22] Shuai Che, Bradford M. Beckmann, Steven K. Reinhardt, and Kevin Skadron. Pannotia: Understanding irregular gpgpu graph applications. *2013 IEEE International Symposium on Workload Characterization (IISWC)*, 00(undefined):185–195, 2014.

[23] Intel Cilk Plus homepage. `http://www.cilkplus.org/`. Accessed Aug. 18th, 2017.

[24] Stephen V. Cole and Jeremy D. Buhler. Mercator user's manual. `http://sbs.wustl.edu/pubs/MercatorManual.pdf`, 2016. Accessed Aug. 18th, 2017.

[25] Stephen V. Cole, Jacob R. Gardner, and Jeremy D. Buhler. WOODSTOCC: Extracting Latent Parallelism from a DNA Sequence Aligner on a GPU. In *IEEE 13th Int'l Symp. Parallel & Distributed Computing*, 2014.

[26] Cole, Stephen V. and Buhler, Jeremy. MERCATOR: a GPGPU framework for Irregular Streaming Applications. In *IEEE 15th International Conference on High Performance Computing & Simulation (HPCS 2017)*, 2017 (to appear).

[27] Cole, Stephen V. and Buhler, Jeremy D. The Function-Centric Model: Supporting SIMD Execution of Streaming Computation. In *Proceedings of the Fifth Annual Workshop on Data Flow Models for Extreme-Scale Computing (DFM 2015)*, PACT '15. IEEE, 2015.

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

[29] Intel Corporation. Intel Xeon Phi Coprocessor System Software Developer's Guide. Technical report, The Intel Corporation, June 2013.

[30] Thomas S. Crow. Evolution of the Graphical Processing Unit. Master's thesis, University of Nevada-Reno, 2005.

[31] CUDA homepage. `http://www.nvidia.com/object/cuda_home_new.html`. Accessed Aug. 18th, 2017.

[32] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[33] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI '04: 6th Symposium on Operating Systems Design and Implementation*, December 2004.

[34] Advanced Micro Devices. AMD APP SDK – A Complete Developer Platform. `http://developer.amd.com/amd-accelerated-parallel-processing-app-sdk/`. Accessed Aug. 18th, 2017.

[35] Matthew Drake, Henry Hoffmann, Rodric Rabbah, and Saman Amarasinghe. Mpeg-2 decoding in a stream programming language. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10–pp. IEEE, 2006.

[36] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity-the Ptolemy approach. *Proc. IEEE*, 91(1):127–144, 2003.

[37] Francisco Fernandes, Paulo G. da Fonseca, LuÃŋs M. Russo, Arlindo L. Oliveira, and Ana T. Freitas. Efficient alignment of pyrosequencing reads for re-sequencing applications. *BMC Bioinformatics*, 12:163, 2011.

[38] Randima Fernando, Eric Haines, and Tim Sweeney. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Addison-Wesley Professional, 2004.

[39] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st Ann. Symp. Foundations of Computer Science*, pages 390–398, 2000.

[40] Michael Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[41] Mark A. Franklin, Eric J. Tyson, James Buckley, Patrick Crowley, and John Maschmeyer. Auto-pipe and the X language: A pipeline design tool and description language. In *Proc. Int'l Parallel & Distributed Processing Symp.*, 2006.

[42] Michalis D Galanis, Paris Kitsos, Giorgos Kostopoulos, Nicolas Sklavos, O Koufopavlou, and Costas E Goutis. Comparison of the hardware architectures and fpga implementations of stream ciphers. In *Electronics, Circuits and Systems, 2004. ICECS 2004. Proceedings of the 2004 11th IEEE International Conference on*, pages 571–574. IEEE, 2004.

[43] Google Cloud. Google Cloud Platform: GPU. `https://cloud.google.com/gpu/`. Accessed Aug. 18th, 2017.

[44] GPGPU.org. GPGPU.org. `http://gpgpu.org/about`. Accessed Aug. 18th, 2017.

[45] Khronos Group. OpenCL homepage. `https://www.khronos.org/opencl/`. Accessed Aug. 18th, 2017.

[46] Andrei Hagiescu, Huynh P. Huynh, Weng-Fai Wong, and Rick S. Goh. Automated architecture-aware mapping of streaming applications onto GPUs. In *IEEE Int'l Parallel & Distributed Processing Symp.*, pages 467–478, 2011.

[47] Jared Hoberock and Nathan Bell. Thrust: A Parallel Template Library. `https://developer.nvidia.com/thrust`, 2010. Accessed Aug. 18th, 2017.

[48] Wen-mei W. Hwu. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, San Francisco, 1st edition, 2011.

[49] International Data Corporation. Executive Summary: the Digital Universe of Opportunities. `https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm`. Accessed Aug. 18th, 2017.

[50] Internet Live Stats. Twitter Usage Statistics. `http://www.internetlivestats.com/twitter-statistics/#rate`, 2013. Accessed Aug. 18th, 2017.

[51] James Jeffers, James Reinders, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann, 2016.

[52] Jen-Hsun Huang. GPU Technology Conference 2015 Keynote Address, Slide 10. `https://www.slideshare.net/NVIDIA/gtc2015-final-published`. Accessed Aug. 18th, 2017.

[53] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proc. Int'l Fed. Information Processing Cong.*, volume 74, pages 471–475, 1974.

[54] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. Scalable simd-efficient graph processing on gpus. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, pages 39–50, Washington, DC, USA, 2015. IEEE Computer Society.

[55] Petr Klus, Simon Lam, Dag Lyberg, Ming Sin Chenug, Graham Pullan, Ian McFarlane, G.S.H. Yeo, and B.Y. Lam. BarraCUDA - a fast short read sequence aligner using graphics processing units. *BMC Research Notes*, 5(1):27+, 2012.

[56] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. When polyhedral transformations meet SIMD code generation. In *Proc. 34th ACM Conf. Programming Language Design and Implementation*, pages 127–138, 2013.

[57] Olaf Krzikalla, Florian Wende, and Markus Höhnerbach. Dynamic simd vector lane scheduling. In *International Conference on High Performance Computing*, pages 354–365. Springer, 2016.

[58] T. W. Lam, W. K. Sung, S. L. Tam, C. K. Wong, and S. M. Yiu. Compressed indexing and local alignment of DNA. *Bioinformatics*, 24(6):791–797, March 2008.

[59] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25+, 2009.

[60] Ben Langmead and Steven L. Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9(4):357–359, April 2012.

[61] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Computers*, 100(1):24–35, 1987.

[62] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proc. IEEE*, 75(9):1235–1245, 1987.

[63] Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proc. IEEE*, 83(5):773–801, 1995.

[64] Charles E. Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[65] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[66] H. Li, J. Ruan, and R. Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Research*, 18(11):1851–1858, November 2008.

[67] Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26(5):589–595, March 2010.

[68] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, August 2009.

[69] Lin Zhang. Belief propagation for stereo tutorial. `http://linzhang-vision.blogspot.com/2014/05/belief-propagation-for-stereo-tutorial.html`, 2014. Accessed Aug. 18th, 2017.

[70] Yongchao Liu, Bertil Schmidt, and Douglas L. Maskell. CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows–Wheeler transform. *Bioinformatics*, 28(14):1830–1837, July 2012.

[71] Chris Lomont. Introduction to intel advanced vector extensions. *Intel White Paper*, pages 1–21, 2011.

[72] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[73] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[74] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[75] George Marsaglia, Wai Wan Tsang, et al. The ziggurat method for generating random variables. *Journal of statistical software*, 5(8):1–7, 2000.

[76] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):25, 2015.

[77] Kevin P Murphy, Yair Weiss, and Michael I Jordan. Loopy belief propagation for approximate inference: An empirical study. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pages 467–475. Morgan Kaufmann Publishers Inc., 1999.

[78] Fahad Bin Muslim, Liang Ma, Mehdi Roozmeh, and Luciano Lavagno. Efficient fpga implementation of opencl high-performance computing applications via high-level synthesis. *IEEE Access*, 5:2747–2762, 2017.

[79] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Morph Algorithms on GPUs. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 147–156, New York, NY, USA, 2013. ACM.

[80] NCBI. Sequence Read Archive. `https://trace.ncbi.nlm.nih.gov/Traces/sra/sra.cgi?` Accessed Aug. 18th, 2017.

[81] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[82] Nicole Hemsoth. Intel Marrying FPGA, Beefy Broadwell for Open Compute Future. `https://www.nextplatform.com/2016/03/14/intel-marrying-fpga-beefy-broadwell-open-compute-future/`. Accessed Aug. 18th, 2017.

[83] Cedric Nugteren, Pieter Custers, and Henk Corporaal. Automatic Skeleton-Based Compilation through Integration with an Algorithm Classification. In *Advanced Parallel Processing Technologies*, pages 184–198. Springer, 2013.

[84] NVIDIA. GP100 Pascal Whitepaper. `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`. Accessed Aug. 18th, 2017.

[85] NVIDIA. GPU-Accelerated Libraries. `https://developer.nvidia.com/gpu-accelerated-libraries`. Accessed Aug. 18th, 2017.

[86] NVIDIA. NVIDIA Compute Unified Device Architecture (CUDA) Toolkit. `https://developer.nvidia.com/cuda-toolkit`. Accessed Aug. 18th, 2017.

[87] NVIDIA. NVIDIA Volta. `https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/`. Accessed Aug. 18th, 2017.

[88] NVIDIA. NVIDIA Compute Unified Device Architecture (CUDA) Code Samples. `https://developer.nvidia.com/cuda-code-samples`, September 2014. Accessed Aug. 18th, 2017.

[89] Onur Mutlu. 18-741 Advanced Computer Architecture Lecture 16: SIMD Processing. `https://www.ece.cmu.edu/~ece447/s14/lib/exe/fetch.php?media=onur-447-spring14-lecture16-simd-afterlecture.pdf`, February 2014. Accessed Aug. 18th, 2017.

[90] OpenACC Homepage. `http://www.openacc-standard.org/`. Accessed Aug. 18th, 2017.

[91] OpenACC. OpenACC 2.0a Specification. `http://www.openacc.org/sites/default/files/OpenACC%202%200.pdf`, August 2013. Accessed Aug. 18th, 2017.

[92] Marc S. Orr, Bradford M. Beckmann, Steven K. Reinhardt, and David A. Wood. Fine-grain task aggregation and coordination on gpus. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 181–192, Piscataway, NJ, USA, 2014. IEEE Press.

[93] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[94] Thomas M Parks. *Bounded scheduling of process networks*. PhD thesis, University of California. Berkeley, California, 1995.

[95] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

[96] Alex Peleg and Uri Weiser. Mmx technology extension to the intel architecture. *IEEE Micro*, 16(4):42–50, August 1996.

[97] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The Tao of Parallelism in Algorithms. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 12–25, New York, NY, USA, 2011. ACM.

[98] Brian Potetz. Efficient belief propagation for vision using linear constraint nodes. In *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*, pages 1–8. IEEE, 2007.

[99] Dan Quinlan and Chunhua Liao. The ROSE Source-to-Source Compiler Infrastructure. In *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT 2011*, October 2011.

[100] Wullianallur Raghupathi and Viju Raghupathi. Big data analytics in healthcare: promise and potential. *Health Information Science and Systems*, 2(1):3, 2014.

[101] Antonio Regalado. The Data Made Me Do It, May 3rd, 2013.

[102] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.

[103] Bin Ren, Gagan Agrawal, James R. Larus, Todd Mytkowicz, Tomi Poutanen, and Wolfram Schulte. SIMD parallelization of applications that traverse irregular data structures. In *IEEE Int'l Symp. Code Generation and Optimization*, pages 1–10, 2013.

[104] Bin Ren, Youngjoon Jo, Sriram Krishnamoorthy, Kunal Agrawal, and Milind Kulkarni. Efficient execution of recursive programs on commodity vector hardware. In *ACM SIGPLAN Notices*, volume 50, pages 509–520. ACM, 2015.

[105] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[106] Stephen M. Rumble, Phil Lacroute, Adrian V. Dalca, Marc Fiume, Arend Sidow, and Michael Brudno. SHRiMP: accurate mapping of short color-space reads. *PLoS Computational Biology*, 5(5):e1000386+, May 2009.

[107] Richard M Russell. The cray-1 computer system. *Communications of the ACM*, 21(1):63–72, 1978.

[108] Daniel Scharstein and Richard Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International journal of computer vision*, 47(1-3):7–42, 2002.

[109] Michael Schatz, Cole Trapnell, Arthur Delcher, and Amitabh Varshney. High-throughput sequence alignment using Graphics Processing Units. *BMC Bioinformatics*, 8(1):474+, 2007.

[110] J. Shendure and H. Ji. Next-generation DNA sequencing. *Nature Biotechnology*, 26:1135–45, 2008.

[111] Anand Lal Shimpi. NVIDIA introduces GeForce FX (NV30). `http://www.anandtech.com/show/1034`. Accessed Aug. 18th, 2017.

[112] Naveen Singla, Michael Hall, Berkley Shands, and Roger D Chamberlain. Financial monte carlo simulation on architecturally diverse systems. In *High Performance Computational Finance, 2008. WHPCF 2008. Workshop on*, pages 1–7. IEEE, 2008.

[113] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, March 1981.

[114] Sander Stuijk, Marc Geilen, Bart Theelen, and Twan Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 404–411. IEEE, 2011.

[115] Joshua Suettlerlein, Stéphane Zuckerman, and Guang R Gao. An implementation of the codelet model. In *European Conference on Parallel Processing*, pages 633–644. Springer, 2013.

[116] Jian Sun, Nan-Ning Zheng, and Heung-Yeung Shum. Stereo matching using belief propagation. *IEEE Transactions on pattern analysis and machine intelligence*, 25(7):787–800, 2003.

[117] Intel Threading Building Blocks homepage. `http://www.threadingbuildingblocks.org/`. Accessed Aug. 18th, 2017.

[118] The Economist. Data, data everywhere, Feb. 25th, 2010.

[119] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *Compiler Construction*, pages 179–196. Springer, 2002.

[120] Top500. Global Supercomputing Capacity Creeps Up as Petascale Systems Blanket Top 100. `https://www.top500.org/news/global-supercomputing-capacity-creeps-up-as-petascale-systems-blanket-top-100/`. Accessed Aug. 18th, 2017.

[121] Eric J. Tyson, James Buckley, Mark A. Franklin, and Roger 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 Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 595(2):474 – 479, 2008.

[122] Stanley Tzeng, Brandon Lloyd, and John D. Owens. A GPU Task-Parallel Model with Dependency Resolution. *IEEE Computer*, 45(8):34–41, 2012.

[123] Stanley Tzeng, Anjul Patney, and John D. Owens. Task management for irregular-parallel workloads on the GPU. In *Proceedings of the Conference on High Performance Graphics*, pages 29–37. Eurographics Association, 2010.

[124] Abhishek Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. Software pipelined execution of stream programs on GPUs. In *Proc. Int'l Symp. Code Generation and Optimization*, pages 200–209, 2009.

[125] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503*, 2011.

[126] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[127] Sven Verdoolaege, Juan C. Juega, Albert Cohen, José I. Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4), January 2013.

[128] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features, 2001.

[129] Vasily Volkov. Better performance at lower occupancy. In *Proceedings of the GPU technology conference, GTC*, volume 10, page 16. San Jose, CA, 2010.

[130] Wikipedia. Bulk synchronous processing workflow diagram. `https://upload.wikimedia.org/wikipedia/en/e/ee/Bsp.wiki.fig1.svg`, 2017. Accessed Aug. 18th, 2017.

[131] Xilinx. OpenCL devices and FPGAs. `https://www.xilinx.com/html_docs/xilinx2016_4/sdaccel_doc/topics/introduction/con-opencl-devices.html`. Accessed Aug. 18th, 2017.

[132] Qingxiong Yang, Liang Wang, Ruigang Yang, Shengnan Wang, Miao Liao, and David Nister. Real-time global stereo matching using hierarchical belief propagation. In *BMVC*, volume 6, pages 989–998, 2006.

[133] Yi Yang and Huiyang Zhou. CUDA-NP: Realizing Nested Thread-level Parallelism in GPGPU Applications. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 93–106, New York, NY, USA, 2014. ACM.

[134] Tao Zhang, Guangshuo Chen, Wei Shu, and Min-You Wu. Microarchitectural Characterization of Irregular Applications on GPGPUs. *SIGMETRICS Perform. Eval. Rev.*, 42(2):27–29, September 2014.

[135] Stéphane Zuckerman, Joshua Suetterlein, Rob Knauerhase, and Guang R. Gao. Using a "codelet" program execution model for exascale machines: Position paper. In *Proc. 1st Int'l Wkshp. Adaptive Self-Tuning Computing Systems for the Exaflop Era*, pages 64–69, 2011.