WASHINGTON UNIVERSITY IN ST. LOUIS

McKelvey School of Engineering
Department of Computer Science & Engineering

Dissertation Examination Committee:
Roger Chamberlain, Chair
Jeremy Buhler
Ron Cytron
Chris Gill
Martin Herbordt

Improving and Modeling Heterogeneous Streaming Computation

by
Clayton Faber

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

May 2024
St. Louis, Missouri

Table of Contents

List of Figures

List of Tables

Acknowledgments

When I started my academic career I definitely didn't expect to end up in a PhD program and now that this body of work sits before me I'm still shocked that I ended up here. There were certainly times I didn't believe that I was capable of completing undergrad studies let alone an entire PhD dissertation. I will be forever grateful for the people mentioned here for not giving up on me whether it be in the form of a mentor, a comrade, a friend, a family member, or likely some combination of these traits. The journey was arduous but I knew that I had people at my back for support, thank you, from the bottom of my heart.

To my family for their love and support: Terry Faber, Sara Faber, Libby Esile, Beamer Esile, my nieces: Ruby and Stella Esile, and, of course, all the extended Faber and Schafer clans.

To my committee and mentors that never gave up on me: Jeremy Buhler, Ron Cytron, Chris Gill, Martin Herbordt, Brad Noble, George Engel, Scott Smith, Steve Muren, and Tim York. Also an important dedication to my advisor, Roger Chamberlain, thank you for taking a chance on me from the very start of this journey.

To my friends and comrades, partners in camaraderie and crime: Mark Ostroot, Mitch McKay, Justin Deters, Kat Saurllo, Tom Plano, Kyle Singer, Anthony Cabrera, Chenfeng Zhao, Luke Hunt, Courtney Hunt, Christian Cool, Tyler Mustard, Marion Sudvarg, Steven Harris, and Steven Timcheck.

To those who are gone too soon, I hope I made you proud: Dorine Schafer, John Schafer, Francis Faber, Carol Faber, Sue Schoth, Rose Lewis and Joe Schafer, and Sandie Barrie.

This work was supported by NSF awards CNS-1527510 and CNS-1763503 and a gift from BECS Technology, Inc.

Clayton Faber

Washington University in St. Louis
May 2024

ABSTRACT OF THE DISSERTATION

Improving and Modeling Heterogeneous Streaming Computation

by

Clayton Faber

Doctor of Philosophy in Computer Engineering

Washington University in St. Louis, 2024

Professor Roger Chamberlain, Chair

Data streaming algorithms are a class of problems that deal with moving data through a system while being processed. When implementing these types of algorithms a developer will spend time tuning an implementation for deployment, but if they are using heterogeneous architectures or when nodes of computation are physically separate from one another performance may be lost to unforeseen data movement complications. In this work we aim to alleviate some of these pain points through a combination of programming advice and mathematical models.

One of the pain points often unseen and underappreciated by developers is a type of data streaming task known as data integration, which are tasks that transform one data element form into another form, usually targeting some other step in the overall processing pipeline. When studying how to improve these types of tasks, we implement them across a variety of execution platforms. Of particular interest to us, we give advice on how to implement such tasks on FPGA architectures. Beyond individual data streaming tasks we then utilize mathematical modeling to understand how individual nodes of computation effect the full data flow of a streaming algorithm. Here we apply existing queuing theory models to reason about average performance of the algorithm and also make estimations on the cost of using the system. To speak on absolute bounds of the system we turn to network calculus which allows us to make estimates of latency on data throughput in the system and predictions

of queue bounds on a node for given arrival and service processes. This represents the first known application of network calculus techniques to streaming data applications.

Chapter 1

Introduction

In the world of computer science we often concern ourselves with the idea of algorithms to solve problems: brute force, dynamic programming, machine learning, approximation algorithms, artificial intelligence, and numerous other options. Researchers and developers alike spend a majority of their time creating, analyzing, and testing, all in hopes that their solution might be the prolific application in their field. Time and resources are spent finding an optimal strategy that accounts for both hardware and software implementations. Going further, while a single algorithm may be optimal, they are often one part of a much larger system where information produced by algorithms are utilized by other algorithms, analyzed for metadata, or even stored for later retrieval. The developers in question now have to spend the time implementing and optimizing a single algorithm for deployment but consider a larger system that has interactions with other modules. These types of systems are often considered as *streaming algorithms*, or *streaming data applications*, where data is processed in chunks sequentially and sent downstream for some other purpose and are distinctively different from algorithms that are standalone running on singleton data sets. Streaming algorithms often exist as a service, acting on requests or when a larger data set needs to be broken into chunks for individual work.

When streaming algorithms are implemented by developers they often experience additional problems that can effect performance in ways that a singleton algorithm may not consider. Often the input data exists in a separate location than the target platform where the algorithm is slated for execution, either coming from sensors in the real world or remote data stores. Taking it further, once the data arrives at a target machine it may need to be modified or transformed in some way before feeding it directly to the target. This can often take the form of changing an input format from one representation into another, or a re-arrangement of data in a table. Once the programmer gets around to tackling these problems, they may

find that with all the extra overhead their impressive mountain of an algorithm now looks like a molehill unable to reach projected performance.

For programmers one imperative is to find efficient and effective methods of both transforming the data into a correct format and delivering it to the eventual target computation node. Researchers from EPFL published an example showing that while an algorithm change might show improvement, the small amount of improvement is dwarfed by the actual time spent in the pre-processing step, which increased more than the algorithm time improved [70]. The specific example is a pair of algorithms for Breadth First Search (BFS) on a graph. Previous work [11] had shown that a push-pull approach results in a 3× improved execution time relative to the traditional push algorithm. However, due to the need for a different data layout in the push-pull algorithm, the pre-processing time increased by 2×, and since pre-processing dominates the execution time, this results in a 1.5× increase in overall execution time. This is illustrated in Figure 1.1 (from [70]).

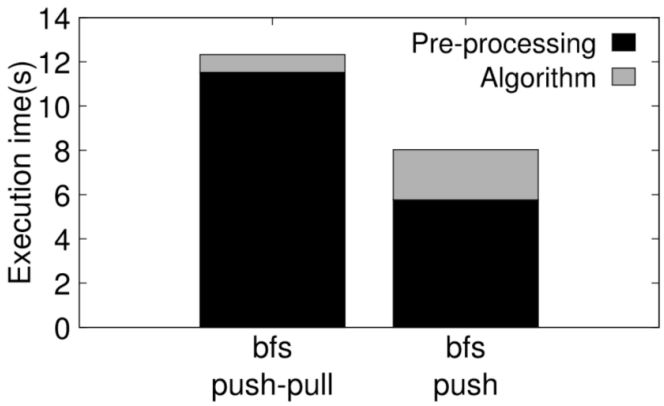


Figure 1.1: Example of the trade-off between pre-processing and algorithm execution time for BFS on the Twitter graph [70].

These pre-processing tasks, often called *data integration* tasks, are often unreported in overall algorithm running time, but are of course necessary and vital for the computation to take place. Often when one wishes to speed up a task, such as a data pre-processing step, hardware accelerators and parallel programming become reasonable targets of investigation, and when this requires data movement from a disparate source to a computation destination the best approach often becomes even more unclear. Therefore, it becomes imperative to understand not only how to efficiently and effectively perform the required computations but also how we move data throughout the system.

Currently in computer architecture there is a focus on how data is moved throughout a system as well as computational accelerators that specialize in a class of computations. Most of this driving force comes from the slowdown in Moore's law and an end to Dennard scaling, as the physical limits of what may be possible with current transistor technology are hit. Through these innovations, domain specific languages and hardware have taken off in the last decade, giving huge performance gains if the problem maps well to these tools. Technologies like systolic architectures such as tensor cores and vector engines such as GPUs map well to specific classes of problems and can be helpful for many algorithms, however a data integration task is not always straightforward in its implementation as each task can be wildly different. Flexible architectures, such as Field Programmable Gate Array (FPGA) accelerators can potentially offer solutions that are tailor made to the task at hand.

FPGAs are a type of accelerator architecture that by their very nature are customized to the computation. Utilizing hardware description languages, higher level languages like C with a high level synthesis compiler, or both one can write a program that performs a data integration task and then actualize the hardware for a task creating a highly specialized compute platform. FPGAs can also play a role in alleviating some of the pain points of data streaming applications when data has to traverse a network, via direct network access available on some specific platforms. It is reasonable to assume that this will help mitigate bottlenecks in streaming applications associated with the movement of data across a network, however problems still arise with reasoning how these discrete processing algorithm kernels interact at a larger scale.

When considering a data streaming application at a higher level the interactions of data movement and performance using heterogeneous hardware become apparent. While devices can utilize specialized hardware for data movement, the relocation of data to discrete memory zones can impact performance to the point where if not handled properly the benefit can become dwarfed by the data movement. Understanding how to mitigate the effects of data movement can be an involved process when considering a fully working deployment where the effects of one node can interact with the performance on a node further downstream by changing data size or processing time. It is our hypothesis that tools that can be used offline and utilize measurements taken in isolation can help inform a developer how design decisions will effect the streaming data algorithm prior to implementation of the full application online.

Models that can be utilized offline can be of great benefit to any developer, helping them test and refactor designs that expend effort in actual implementation. It is through both a combination of programming guidance and mathematical modeling that one can reason about performance in a way the minimizes the effort of developers when it comes to implementing streaming data applications.

1.1 Contributions

In this work we focus on streaming data applications in multiple execution environments, what their characteristics are, how to model their performance, and how to effectively implement them in heterogeneous environments. Utilizing the platform-independent utility OpenCL we are able to target both CPU, GPU, and FPGA platforms and compare performance across the platforms. Of these platforms, we are particularly interested in FPGA platforms for their performance and give general programming advice for implementors. For our streaming data applications we target data integration tasks which involve the transforming of data from either one format to another or massaging data such as rearrange or removing columns in a CSV file. These data integration tasks are usually under-appreciated in many applications and a piece of a larger system similar to stages in other streaming data applications and while one stage may be performant it is important to understand the system in aggregate.

In our efforts to improve data streaming applications we turn to *queuing theory* to help model the system as a series of servers sending data from one stage to another. This achieves two goals: Giving a general idea on upper-bound performance and allows us to model not only the algorithm stages but also unseen effects related to data movement in the system. The latter of these two is of particular importance when it comes to utilizing heterogeneous hardware in streaming applications due to the movement of memory between zones. This model utilizes the basic *M/M/1* queuing model and can be derived from measurements taken in isolation requiring little overhead on the developers part when testing instead of implementing a fully functioning system. This model also has the added benefit of analyzing value when it comes to implementing parts of algorithms on hardware with a cost. In effect, it is an economic model as well as a performance model.

When looking into how closely the queueing model matches a predicted result with a real-world result we see that the roof-line performance does not tell the whole story. To further our understanding of these systems and in an attempt to build a more accurate model we turned to *network calculus* to help fill the gaps. Network calculus, although originally developed for the analysis of networking equipment, has properties that lend themselves well to performance modeling for heterogeneous streaming application. Utilizing min-plus algebra, network calculus seeks to make guarantees on service provided by individual nodes in a system. This has the property of both retaining each node's isolation but also allowing the concatenation of nodes to potentially simplify any section in the pipeline. In this work we present our adaption of network calculus to a heterogeneous streaming data application and it's evaluation when compared to both simulated and real-world results.

The specific contributions of the dissertation include the following:

- *Analysis of data integration applications across platforms.* Utilizing OpenCL on data integration tasks, a sub-class of streaming data applications, for deployment on multiple types of hardware (CPU, GPU, and FPGA) [22, 37].
- *General programming advice for streaming data applications on FPGA hardware.* We make recommendations for implementing such applications on FPGA hardware utilizing OpenCL High-Level Synthesis [37, 38].
- *Development and analysis of queueing theory models for streaming computations.* In an attempt to reason about the performance of data streaming applications on heterogeneous compute engines we develop a model based on queuing theory and test modeled results with real world data [36].
- *Development and analysis of network calculus models for streaming computations.* We also develop a model based on network calculus to help us reason more about maximum delay through the system and queue sizes between different stages of the data streaming algorithm [35].

Both the queueing models and network calculus models are tested against simulated and real world results in order to test their validity and get a sense of how closely models match the real world. The goals with these models are to require little of developers in terms of data collection and knowledge about the system intricacies.

1.2 Outline

The outline of the dissertation is as follows. Chapter 2 provides background and related work. Chapter 3 describes the streaming data tasks that we investigate and their implementation on a variety of hardware platforms. Chapter 4 presents the queueing theory performance models and their evaluation, and Chapter 5 introduces the network calculus performance models and their evaluation. Chapter 6 gives conclusions and future work.

Chapter 2

Background and Related Work

Streaming data applications have been a target of study for a considerable time, well over twenty years [90]. Examples of development platforms for the streaming paradigm include Auto-Pipe [25], Brook [17], Raftlib [15], StreamIt [92], and Streams-C [41]. In addition, each of these development platforms supports (or has been extended to support) computational accelerators, either FPGAs or GPUs.

Figure 2.1 illustrates an example streaming application with two compute nodes (labeled `Stage A` and `Stage B`). Data outbound from `Stage A` is delivered, as input, to `Stage B` by the run-time system. Common examples are applications in which the input data are not in the appropriate form or format for the computation of interest, so a pre-processing or data integration step is inserted ahead of the computation so as to enable the computation to proceed. In these examples, `Stage A` is the data integration and `Stage B` is the computation of interest. In the models presented here we make the assumption that the asymptotic complexity stays linear for all stages of the streaming application. We also make the assumption that the resources are dedicated to the current task and are not being shared with other users in the cloud or other applications. (Work that relaxes this assumption with simple sharing models for FPGAs and traditional processor cores is described by Beard [12, 13].) This paradigm readily supports the two nodes being executed on distinct execution platforms, whether they be processor cores, FPGAs, GPUs, or some other accelerator, and the data delivery might be via shared memory, PCIe bus, or the network.

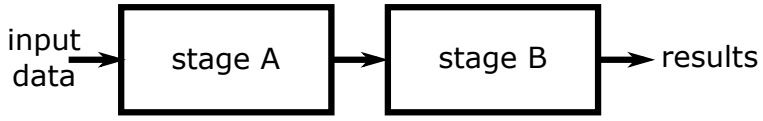


Figure 2.1: Example streaming application.

When modeling the flow of data down the pipeline, it is prudent to explicitly recognize that this data movement might be the primary contributing factor to the overall performance, and as such should be included in the model. This is readily accomplished by inserting an additional node in the pipeline that represents the communication task (see Figure 2.2). By modeling each node as a queueing station (with ingest rate λ and service rate μ), the resulting queueing network is shown in Figure 2.3 [12]. Prior works by Choi et al. [29] and Gu and Wu [43] make use of similar models, however, these works are more concerned with the online scheduling of tasks whereas our focus is on a more static analysis that a programmer may use to reason about how an application could be distributed in a platform agnostic way.

Figure 2.2: Example streaming application flow model.

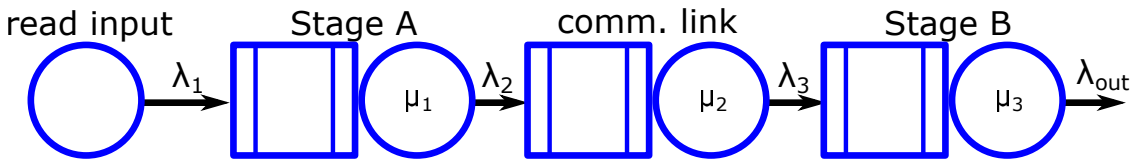


Figure 2.3: Example streaming application queueing network model.

While this model is useful, it only deals in averages and can't speak to things like performance bounds and data latency through a system. To reason about these metrics we turn to network calculus, a theory of systems designed to reason about network bounds [33, 33]. Developed to reason about ATM networks, network calculus borrows concepts from circuit theory and utilizes min-plus algebra to estimate data as it moves through a system. This model retains the same benefits of queuing theory where nodes are separable and can be considered in isolation but also has the benefit of node concatenation, where nodes can be consolidated into a single node for simplicity. Network calculus models can give us absolute bounds on service provided by nodes as well as end-to-end delay on data through the system. The specifics of how the network calculus model are used is expanded upon in Chapter 5.

2.1 Related Work

For the experimental efforts, this work draws from the Data Integration Benchmark Suite (DIBS) [21, 22]. A number of the applications are either members of the benchmark suite, or build on the benchmark suite (e.g., an individual benchmark application that streams data to a downstream data processing application). A number of groups have utilized accelerators for various data integration problems. Fang et al. [39] utilize FPGAs as part of an enterprise ETL operation. Aggarwal [2] explores the use of GPUs for a similar set of tasks. Cabrera and Chamberlain [19, 20] report on the performance of several of the DIBS applications accelerated using FPGAs. Pourhabibi et al. [79] describe Optimus Prime, an ASIC design specifically aimed at data transformations of this type, that is targeted for use as a set of microservices in a distributed environment. Thomas et al. [93] present Fleet, a framework that builds streaming FPGA designs from individual kernels, including data transformation. Fleet aims to automatically parallelise the computation (via replication of the kernels), including the management of data flows to and from external memory units. In addition to data transformation, Fleet is applicable to machine learning applications as well.

Another application we will use is the Basic Local Alignment Search Tool (BLAST) [5, 6]. BLAST is among the most widely used software in bioinformatics. It scans a DNA or protein sequence, the *query*, against a *database* of other sequences to determine which members of the database are most similar to the query under a biologically motivated score equivalent to a weighted string edit distance. In this work, we focus on BLASTN, the variant of BLAST that compares a DNA query to a database of other DNA sequences, such as a genome, a metagenome, or a reference such as GenBank NR.

A representative subset of previous implementations of all or portions of BLAST on accelerators include CAAD BLAST [69], Mercury BLAST [49, 55, 57], RC-BLAST [71], and Tree-BLAST [47] on FPGAs and cuBLASTP [103], GPU-BLAST [98], and Mercury BLAST [67, 77] on GPUs. We utilize the GPU-accelerated Mercury BLAST of Plano and Buhler [77] in our experimental work.

We will apply some of our modeling efforts to an ML application.

Machine learning has long benefited from acceleration. The TensorFlow framework [1] regularly utilizes GPUs, and now is supported by specialized hardware [52]. Zhang et al. [102]

describe a general approach to deploying machine learning applications using convolutional neural networks on FPGAs. Geng et al. [40] use a cluster of FPGAs for ML training, and Li et al. [64] investigate how to partition inference on an FPGA cluster. Sharma et al. [86] start from a domain-specific expression of the ML problem and compile to an FPGA deployment. Liu et al. [66] combine the use of GPUs and FPGAs on a machine learning problem, ultimately concluding that for their problem, GPUs were best suited for the training and FPGAs were best for inference. Shahid and Mushtaq [85] review multiple generations of TPUs on ML problems, comparing them to GPUs and FPGAs, and Reuther et al. [80] survey a wide range of machine learning accelerators.

Our final set of applications include compression and encryption, both of which have a long history of hardware acceleration.

Early work in hardware acceleration of compression algorithms was performed by Huang et al. [48], Rigler et al. [81], and Salama et al. [82]. An LZ4 compression (the one we explore) was reported by Bartík et al. [9]. Early work in FPGA-based acceleration of the AES algorithm includes the efforts of Chodowiec and Gaj [28], Good and Benaissa [42], and Zambreno et al. [101]. Work in this area is sufficiently mature that these functions are now available in libraries provided by the FPGA manufacturers.

The notion of using a high-level language to describe an algorithm to be deployed in hardware has a long history. Streams-C [41] has its origins in streaming computations. ROCCC [97] was an early system that focused on analysis of loops. AutoPilot [104] is the system that eventually evolved into Xilinx's commercial offering. LegUp [23] has a focus on identifying and accelerating a portion of an application that is amenable to FPGA deployment. Cong et al. [31] provided a comprehensive review and vision approximately one decade ago.

There has been significant work recently in the area of optimizing the performance of arbitrary applications implemented on FPGAs using HLS. Examples include the empirically driven approach of Sanaullah et al. [83], constraining the application set to a specific domain (e.g., CNNs) by Sohrabizadeh et al. [89], the use of multi-level intermediate compiler representations by Ye et al. [100], and the exploitation of an affine type system for compile-time analysis [72]. Our work can be considered to be within the scope of this body of work.

A recent review describes applications that exploit more than one accelerator [24] which will be important for the types of tasks we would like to perform utilizing multiple compute

resources on nodes. To this end, we need ways to evaluate such systems and how data flows between them. Queueing theory has long since been used for the design and evaluation for computer systems [3, 53]. The origins of our model, built on queuing theory to measure performance, is described in Chapter 4. It relies on prior work by Dor et al. [34], Padmanabhan et al. [73], Beard and Chamberlain [12], Timcheck and Buhler [94], and Plano and Buhler [78].

Other applications of queueing theory for modeling streaming systems include a queueing network model for a family of cyclic SPMD applications executing on MIMD platforms by Cremonesi and Gennaro [32]. They validate their model on the high-performance applications of a PDE solver and a quantum chemical reaction dynamics code. Another example is a long-lived transaction (LLT) processing system for database management systems (DBMS) modeled by Liang and Tripathi [65]. Here, the performance of the overall transaction processing system being modeled includes the effects of data locking, resource contention, and failure recovery.

Tolosana-Calasanz et al. [95] combine queueing theory models and feedback control mechanisms to provision cloud resources to process data streaming from high data rate sensors. Dor et al. [34] developed an early queueing theory model of BLAST. Finally, PalunčIć et al. [74] survey the development of queueing models for cognitive radio applications.

The above queueing network models make the simplifying assumption that individual queueing station analyses are separable [10]. However, if individual queues fill, in a streaming computation that invokes backpressure on the upstream nodes, which is neglected in a separable analysis. Modeling this form of backpressure has been addressed for $M/M/1$ queueing systems by Perros and Altiok [76] and extended to Coxian distributions by Krishnamurthy and Chamberlain [56]. However, generalizing to arbitrary arrival and service distributions is a challenge. As a result, we turn to network calculus as an approach to understanding the performance of the system under these conditions.

The application of network calculus is widespread in classical networking systems [7, 8, 60, 84]. These applications, however, are mostly concerned with extensions to other networking models, such as network firewalls [99] and job scheduling [63]. Network calculus also has two sub-branches; one that deals with systems that behave in a stochastic manner, called stochastic network calculus [51], and the other dealing with hard real-time deadlines, known as real-time network calculus [91]. In this particular work we use the standard, deterministic,

network calculus and this is, as far as we know, the first application of these models to streaming computations that specifically target heterogeneous architectures.

Chapter 3

Streaming Data Tasks for Heterogeneous Compute

As discussed prior, streaming algorithms are often split up into multiple sections where data is processed and passed onto the next node for completion. Pre-processing algorithms are a wide range of streaming algorithms that are present not only in streaming environments but they are also a component in algorithms that are singleton in nature. These data integration tasks, as mentioned in the introduction, are often not thought about and can become bottlenecks in a full data streaming application when looking at the end to end performance. Unfortunately, it can be hard to reason about improvements for these types of application due to the differences of data structures and implementations. In prior work, effort was expended to categorize and analyse these applications into a set of data integration applications [22] and in this work we specifically want to look at their implementation on heterogeneous platforms, like FPGAs. FPGAs have the wonderful property of being flexible and can be tailor made to suit an an application, but it can often be a bit daunting to program for them as their programming tools can be difficult to utilize. In this work we want to discover what properties and best practices should influence a programmer when implementing data integration applications on an FPGA.

3.1 Data Integration Tasks

The set of all data integration tasks have the defining feature of transforming one data record into another data record, but their individual implementations can be quite varied and difficult to reason about. Prior work of defining the space of data integration tasks can be found in the Data Integration Benchmark Suite (DIBS) [21, 22]. Here we spent time to define categories of data integration along with analysing their single threaded performance

and implementation. As shown in Figure 3.1 the applications cross a wide range of target fields and potential data types. At their core most data integration applications take an individual record and transform it into an equivalent record, each record being independent of each other. This type of processing gives rise to a streaming algorithm implementation as each record is processed once, and kept in-order as to preserve the integrity of the data set. These applications were programmed initially in a simple singled threaded manner, but if we want to improve and migrate them to faster, more specialized pieces of hardware it becomes imperative to think about creative ways to move to a parallel implementation.

	Data Integration Tasks		
Domain	**Parsing/Cleansing**	**Transformation**	**Aggregation**
Computational Biology	Separate bases and meta-data Handle non-A,T,G,C bases	fa→2bit 2-bit→fa	Track total size
Image Processing	Parse FITS tags	fits→tiff idx→tiff optdigits→tiff unipen→tiff	Pixel statistics Histogram Taking log of pixels
Enterprise	Adjust non-ASCII characters	ebcdic→txt fix→float	Count number of elements
Internet of Things	Tokenize input	tstcsv→csv gotrackcsv→csv plt→csv	Running total of file size
Graph Processing	Parse edge list	edgelist→csr	Get total vertex/edge count Compute vertex edge degree

Figure 3.1: Data Integration Benchmark Suite application classification [22].

3.2 Parallel Implementation

When thinking about accelerating applications in the data integration benchmark suite two things come to mind: what kind of programming model one wants to use and what kind of hardware will be the eventual target. In DIBS most of the statistical analysis showed that a large portion of the programs had low branch entropy and low temporal locality meaning that, as expected, in a majority of the apps the data is transformed with minimal amount of divergence in the processing path and is then stored to the final memory buffer.

When looking at the instruction mix of the x86 implementation, shown in Figure 3.2, we also see that the percentages of instruction mixes can vary from one data integration app to

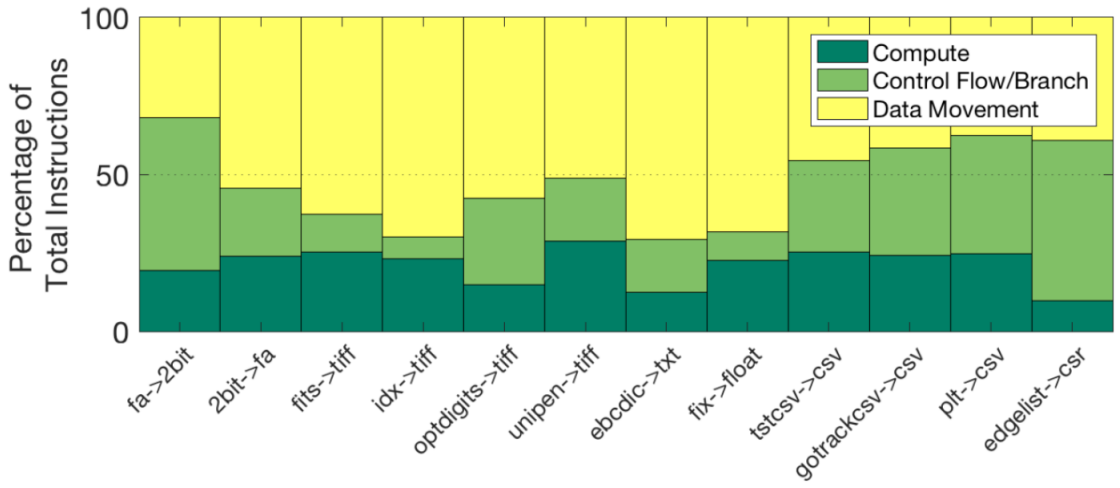


Figure 3.2: Instruction Mix of DIBS Applications on x86 [22].

another, meaning that one application suited for one hardware target might not be the best fit elsewhere. With these considerations in mind one can either follow a traditional approach of parallelizing via CPU multi-treading or rather pursuing an FPGA implementation due to it's ability to actualize custom architecture that fits an individual data integration application efficiently.

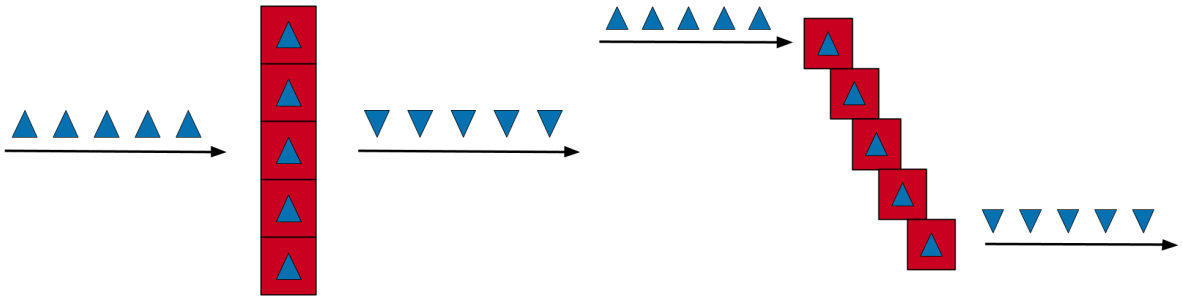
Ideally we want to have an implementation that is easy to move between different platforms, to empirically test and see what platforms are a fit for data integration. The original implementation of DIBS has single threaded implementations of each app coded in the C language compiled for the x86 platform which we use as a starting point for our measurements. While there are many different ways of programming for a multi-threaded implementation ideally we would like to keep as much of the core data integration as similar as possible. When one programs for an FPGA device often the common way is to utilize hardware description languages (HDL) such as VHDL and Verilog to directly program, utilizing modules and IP blocks to actualize hardware. While this is a reasonable approach, most HDLs require a programmer to worry about computation at a cycle by cycle granularity and it can be a burden to layout a system in this way. Both major manufactures of FPGAs (Xilinx and Intel) offer High Level Synthesis (HLS) programming tools that allows a programmer to write code in a higher level language and utilize the tools to parse and create a representation in Verilog which is then used to generate hardware. HLS tools require programming in the C/C++ languages with some restrictions and along with these, C derivatives such as OpenCL can be directly used by the tools and allow a developer to use a model that is inherently parallel.

We took four of these data integration apps and implemented them in OpenCL, a language designed for multi-platform execution targeting CPU, GPU, and FPGA systems. Part of the advantage of taking an OpenCL approach is that we could start with a base computation kernel that looks highly similar to the original C single threaded implementation. From there we took a straightforward implementation, essentially a 1 to 1 copy from the original C code with minor edits for each platform necessary for execution. While it was relatively simple to get a straightforward implementation running well on the CPU it was a bit unclear what would and would not work on FPGA implementation.

3.3 Multi-Target OpenCL

In the original OpenCL programming model one writes a "kernel" of computation to be executed on some set of compute units that map to either multi-core CPUs or GPU cores. An individual device has its own set of device memory and execution queue which is managed by a host system, usually the main operating system. When programming a kernel one utilizes OpenCL specific calls to grab a work item identification and articulate how said work is to be done in isolation from other work items. Memory is both allocated and written to the device specific memory via host-side library calls. At the time of enqueuing the kernel execution a number specifying how many work items are to be computed before the kernel is to be considered done and when complete the host is in charge of reading device memory into host side buffers. This programming model for OpenCL is referred to as Multi-Work Item (MWI) and is the typical target for multi-core style systems.

When OpenCL is used in the HLS space the designers of the tools added another programming model called Single-Work Item (SWI) which, along with MWI, allows for styles of execution on the FPGA. A SWI kernel is very similar to the MWI however the difference is instead of using OpenCL API calls to get an ID for work, the amount of work is built into the kernel similar to how one would program a for loop. With this style of execution a kernel is instead queued onto the FPGA with a work size of one, allowing either the amount of work to be hard coded into the kernel or adjustable through kernel arguments. The HLS tools use this type of programming model to implement pipeline parallelism, implementing the hardware needed to perform each instruction and, when the pipeline is full, complete one execution of the main kernel loop within one clock cycle. An illustration of both the MWI



(a) Multi-work item (MWI) model. (b) Single-work item (SWI) model.

Figure 3.3: Illustration of the MWI and SWI OpenCL programming models.

and SWI models is given in Figure 3.3. It may seem on the surface that this would make an even easier porting effort from our single threaded data integration task as one would just have to import the main loop from the code and make the proper API calls for host side execution, however, HLS tools offer a large range of flexibility in optimizations one may add and these are what we want to take into consideration when looking at programmer best practices.

Our experience has not been consistent with the guidance of the FPGA manufacturers with respect to MWI and SWI implementations. We have experience with a number of applications for which the MWI implementation performs better than the SWI implementation. Figure 3.4 shows the speedup of the MWI implementation relative to the SWI implementation for four applications (data from [18, 20, 45]). These four applications can be split into two categories: compute intensive applications in the form of a standard matrix-matrix multiplication and streaming computations drawn from DIBS. To factor out issues with I/O, the input and output data reside in main memory. For the matrix-matrix multiply applications, the MWI implementation outperforms the SWI implementation by more than two orders of magnitude, and the performance of MWI over SWI is more than one order of magnitude for the two applications chosen from DIBS. The measured data throughput for each of the applications is shown in Table 3.1.

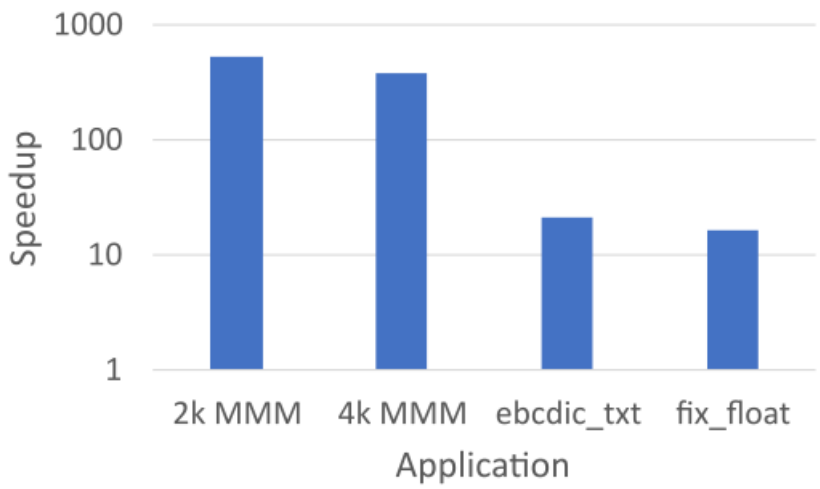


Figure 3.4: Speedup of MWI over SWI implementations. Applications are 2k×2k and 4k×4k matrix-matrix multiply plus data integration applications `ebcdic_txt` and `fix_float` from DIBS.

Table 3.1: Measured throughputs for MWI and SWI implementations.

Application	SWI (MiB/s)	MWI (GiB/s)
2k MMM	8	4.2
4k MMM	8	3.1
`ebcdic_txt`	260	5.5
`fix_float`	400	6.5

3.4 Application Targets and Deployment

To explore implementation details we chose the following benchmark programs from DIBS: `fa->2bit`, `gotrackcsv->csv`, `fix->float`, and `idx->tiff`. These have the characteristics of being from different fields and a distinct mix of instruction sets and gave a good overview of how these applications would perform on various architectures. Our initial inquiry into these applications was the impact of a sequential dependency in the data transform and how such a dependency would affect performance and how much effort would be required to improve performance. The applications `fa->2bit` and `fix->float` have databases that are sequential but their data boundaries are knowable at run-time making an easy transition to a parallel implementation. The application `gotrackcsv->csv`, has its records separated by a new line character at un-even boundaries requiring an upfront analysis of the database

before the actual data-integration task can begin. The `idx->tiff` application is similar in that a small amount of metadata processing is required upfront for information on how the images in the database are represented and their size allowing for highly parallel execution after this task. In all instances we consider the total execution time of the data integration task being time spent in the data integration and any metadata processing. In porting these applications over to multi-threaded CPU and FPGA we used an OpenCL implementation with a MWI kernel for the CPU and SWI kernel, at the manufacture's recommendation, for the FPGA.

In this particular experiment we utilized a Xeon E-2256 for CPU deployment and an FPGA from the Hardware Accelerator Research Program (HARP) provided through Intel. The HARP system is a combination of an Arria 10 FPGA and a Xeon Intel CPU connected with a cache coherent bus within the same socket. This specific implementation of a heterogeneous compute FPGA is unique in that the host machine and the compute unit have a unified view of memory meaning that when memory is allocated for the compute device it can accesses it without waiting on a host transfer. It is up to the programmer to setup coherency between the host and device which is handled through shared virtual memory `map` and `unmap` commands in the host code. The general program execution first sets up the host for OpenCL execution creating a device handle and command queue for kernel execution, the data integration tasks are then loaded in as kernels. For the CPU target the kernel source code is read in and compiled in real time for execution but for the FPGA target an HLS compiled binary (bitfile) is used for configuration. When a metadata processing step is required its kernel is queued before the the data integration task and the total queue execution time is included as part of the final throughput numbers. Once the data integration task is completed the results are validated and any stats are reported. For all of our data integration application measurements we do not consider the time it takes for data to be read off the disk, but instead only consider data that is active or already in memory, partially due to the fact that such advancements in data retrievement are not the focus of this research and a large number of techniques both in hardware and software exist to try and alleviate this concern [26, 27].

In Figure 3.5 we display the performance numbers for the data integration apps on both target platforms using OpenCL kernel execution. The kernels listed here were ported with minimal effort to both the multi-threaded CPU and FPGA which can be representative of what one might expect base performance to look like on these given systems. First we see the greatest performance with the FPGA in the `fa->2bit` application, over the

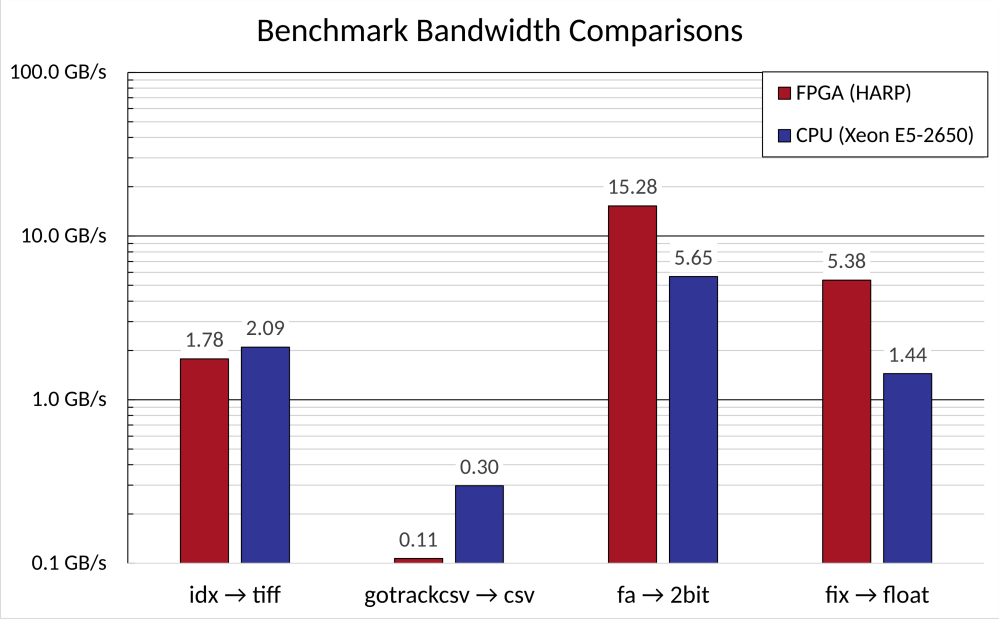


Figure 3.5: Bandwidth comparison numbers for selected DIBS benchmarks running on multicore CPU and HARP FPGA machines.

baseline implementation in the original DIBS this results in a 653× speed up in throughput along with an 2.7× speedup over the multi-threaded version. (Note that the original DIBS single-threaded implementations did not include compiler optimizations, which are enabled in these multi-threaded OpenCL implementations.) Similarly, we see a performance boon with `fix->float` FPGA implementation over the original and multi-threaded versions with a 26× speedup over the single threaded and a 3.7× speedup over multi-threaded. In the `idx->tiff` application the FPGA is not the top performer compared to the multi-threaded implementation, here we have the metadata processing kernel along with the main processing kernel which points to the notion that cooperative processing might be worth considering here as the transient startup of using an accelerator for one metadata step might not be worth the processing time. Finally looking at the worst performer on the FPGA, `gotrackcsv->csv` we have a marginal increase of throughput over the base single-threaded implementation of roughly 10 MB/s. There are a multitude of reasons for why this app has poor performance. Mostly, the performance can be attributed to the control flow present in this particular app and the amount of times it accesses global memory. The app's processing path reads in characters one at a time looking for boundaries noted by the comma character and once a certain amount is seen a branching path is enabled to remove data from the structure. This

causes a very irregular execution pattern for this app and because each line has a variable amount of characters in each line it becomes difficult to determine ahead of time where these boundaries are. Furthermore, a metadata step is required up front to determine where line breaks exist in the database adding more processing time not suited for FPGA computation.

3.5 Improving Kernel Implementations

Building upon naive approaches we wanted to look at HLS features that one could add to a kernel in order to improve performance across all apps. For this approach we attempted to compare both FPGA and CPU performance on one app with a plethora of varying setups, including an OpenMP CPU Implementation, and were performed on the HARP system. In FPGA HLS tools as mentioned prior there are two coding styles for OpenCL kernels: MWI and SWI; with these there are also extra additional options and flags that can be added to the kernel that can affect performance. As it can take hours to compile kernels in multiple configurations we settled on 4 different configurations to observe: A naive SWI kernel with no pipeline flags enabled, SWI kernel with a flag to pipeline as aggressively as possible (our kernel from above), A MWI kernel with the largest work-group size as possible (64 for our tools), and what we call an "over-optimized" kernel which is a a MWI kernel with the largest work-group size as possible, 2 total compute units, and a SIMD work item count of 16. The OverOpt kernel is meant to represent a scenario where a potentially confused programmer might try to do everything under the sun to try and improve a FPGA kernel without giving too much consideration to the hardware that they are targeting. While we expected to see somewhat poor performance we were curious which choice in kernel style would compare to multi-threaded CPU execution.

Figure 3.6 shows the execution time of the `fix->float` application on a log scale with respect to database size also in a log scale. Here on this graph a lower bar indicates better performance. From the outset for a considerably small database of 512Kb we observe the the OverOpt Kernel actually performs quite well and is only beaten out by the OpenMP CPU implementation, however, as the data set size increases this observation will not hold. We observe a breaking point between the multi-threaded CPU and the FPGA MWI and SWI implementations, but along with this we see the OverOpt kernel begins to run away and only outperform a naive implementation and a single threaded implementation. Interestingly, this

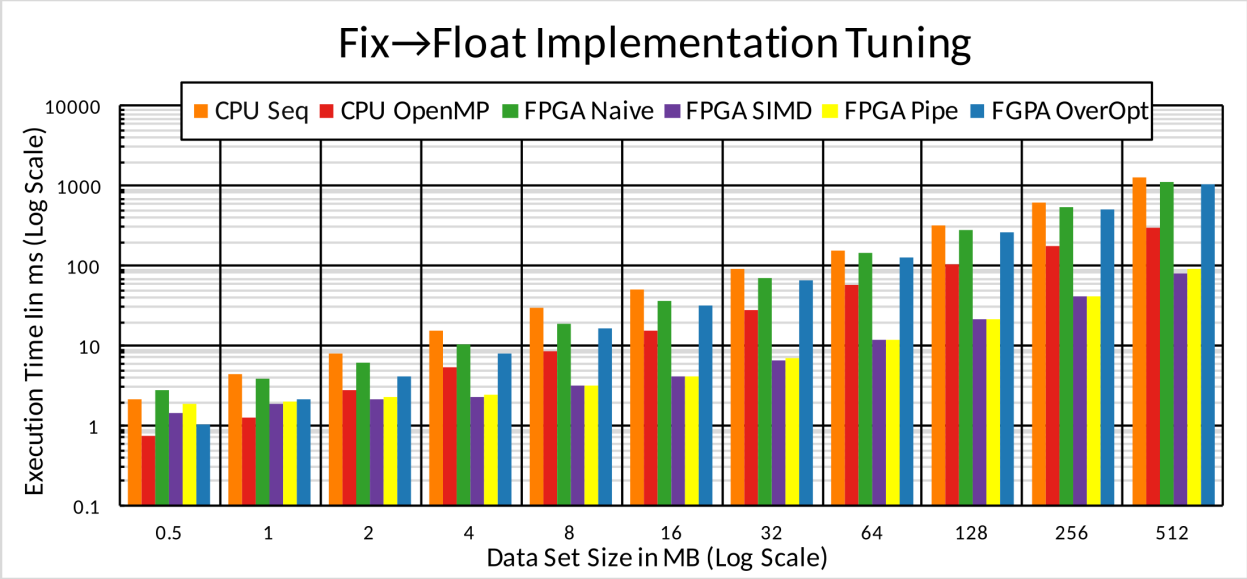


Figure 3.6: Data size vs. execution time on HARP for versions of the `fix->float` kernel.

trend continues as we increase the database size and in fact the performance gap between the kernels with less attributes for the HLS engine and the kernel made in an attempt to improve performance only gets larger. Furthermore, a closer inspection between the MWI model and the SWI model shows that, for this one application, a MWI model actually has a slightly better performance than the SWI contrary to what is suggested by the manufacturer. While this is only one application this test is run on it does paint an interesting picture of what general programming advice we would like to give.

3.6 Programming Advice

Currently through our research we have found a few guidelines that we believe can be applicable across multiple data integration applications implemented on FPGAs. With our examples on the HARP system, three main points come to mind: data transforms with data dependencies, divergent data transforms, data transforms with metadata processing. Data transforms that have both data dependencies and divergent execution paths cause the same issue on FPGA implementation: an inability to pipeline efficiently for unrolled loops. When the HLS tools analyze kernel loops it does so in an attempt to reduce the amount of cycles required for each instruction. When divergence is introduced, such as an `if else` statement,

the compiler will generate an execution path that will take the time to execute the largest path. Data dependencies, similarly, will cause the tools to create loops that have long cycles between each loop iteration, causing execution time to increase significantly. Metadata processing is a different problem in that this is a bit of execution that needs to be done upfront and separate from the main kernel loop. Processing situations where an entire dataset may need to be parsed for metadata can lead to divergent processing if the kernel is looking and marking the location of certain characters. Other metadata situations may only require a small amount of data to process. In our second experiment we can deduce that the transient startup costs of FPGA execution may not outweigh the benefits of specialized execution as a single threaded CPU implementation can beat out a FPGA kernel. While the data integration task measured in the second experiment did not deal with a meta processing task the small database size could be indicative of smaller processing tasks.

Looking further beyond the database size implications in the second task we note the runaway performance as mentioned above. A large portion of the improvements chosen here were meant to be indicative of a programmer trying to squeeze performance out of a HLS kernel. Often when implemented at first blush a programmer will often see terrible performance when it comes to FPGAs. As such a developer might see a large portion of the FPGA be unused and as such try to do reasonable things such as increasing the number of compute units or increasing the number of SIMD work items and the issue arises when the generated hardware does not match what is required to perform the computation efficiently. In this case the extra compute units create more contention for the shared memory bus which is not ideal for our data integration applications as our applications are mostly made of memory movement instructions demonstrated by the instruction count mix shown in Figure 3.2. This is of course exacerbated with the addition of more SIMD work items increasing the memory requests in parallel. Such advice could be applied to data integration tasks on FPGAs that take this form of regular isolated tasks.

The Xilinx Vitis toolchain has attempted to improve SWI implementations with a programming strategy available for both OpenCL and C HLS kernels known as *dataflow optimization*. In the documentation, dataflow optimization is described as a way to implement "task-level pipelining," which allows for code blocks contained in functions to be scheduled in a way to achieve pipeline parallelism, similar to the approach that loop unrolling does within a loop. As shown in Figure 3.7, the idea is to create a collection of tasks or functions that would normally run in sequence and allow the compiler, with the help of `#pragma` directives, to

create hardware that allows for execution of downstream tasks to start before their preceding task has completed. This is achieved through the use of first-in-first-out (FIFO) buffers to pass data elements from one code block to the next. These code blocks could be some type of read from global memory, general compute functionality, then a write back to global memory, which allows the hardware to take advantage of blocked reads and writes. Ideally, this type of coding style, as the name suggests, can work well for data streaming applications such as the previously mentioned data integration applications.

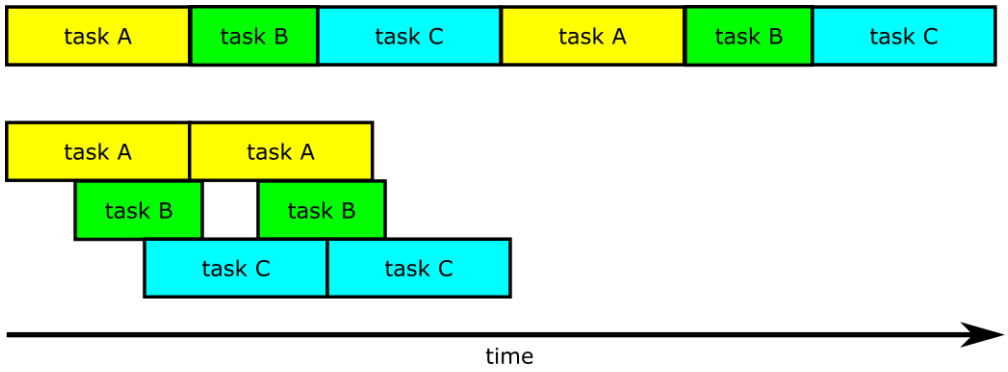


Figure 3.7: Illustration of dataflow execution with 3 tasks. The top diagram illustrates a sequential execution timeline, while the bottom diagram illustrates a pipelined execution timeline.

Using this technique, we have seen success deploying a data integration application as a dataflow kernel, with performance better than the MWI and even the initial SWI implementation. However, it is important to note that in other data integration applications we see that this style of kernel does not perform quite as well as expected. In an experiment implementing a handwriting database from an ASCII picture representation to a bit array representation (the `optidigits` application from DIBS) we find that the best performance is with the original SWI programming model with a throughput of 221 MiB/s. Believing that the dataflow approach would result in better performance, we made a pair of attempts using it. With our first attempt at the dataflow model we observed a slight reduction in throughput to 218 MiB/s. While a reduction of roughly 3% in throughput is arguably negligible, it still gives pause to what implementation strategies developers should choose.

3.7 Conclusion

Streaming data tasks, while being sometimes difficult to implement, can see performance improvements from heterogeneous architecture. The difficulty stems from a need to fully understand the algorithm before deployment but this could be said of almost any performance engineer for any type of algorithm. When it comes to data integration often time emphasis is put on data movement which can often be a pain point for these types of applications. Here we explored implementing data integration tasks on multiple types of hardware utilizing OpenCL and focusing in on FPGAs as an execution platform. While FPGAs show promise in this field it is not without its own caveats and here we have pointed out some of the difficulties one may have when implementing data integration tasks. We have explored two types of programming styles: MWI which takes a thread-based work item approach and SWI which is a deeply pipe-lined approach utilizing for loops. When implementing effective programming strategies it is critical to not only have an understanding of how the algorithm handles memory access, where that memory access is originating from, and how memory hierarchies interact with eachother. With these interactions in mind a developer may spend some time implementing one piece of a data streaming algorithm without focusing on the higher level interactions of other nodes in the process. Stepping back and looking at the larger picture can be somewhat difficult, however, we address these concerns with mathematical models that help a developer reason about performance beyond individual node implementation.

Chapter 4

Queueing Models for Streaming Data Computation

4.1 Introduction

While we have spent some time describing the characteristics of data streaming applications in the form of data integration, however, it is important to remember that they do not exist on their own. When including data integration as a step in an overall streaming application the system inherently becomes more complex, and the use of an effective performance model can be used as a concrete way to reason about both the application and its deployment. On top of that there may be scenarios where different instances of computational accelerators may be used to run different portions of the overall algorithm. These accelerators may exist on the same platform or on entirely separate nodes. There can be a large number of potential deployment options and configuration settings that can impact performance, and it can be difficult to pinpoint where time and resources need to be spent and what to budget.

We propose an empirically guided mathematical model to help guide performance choices in the streaming algorithms that seeks to find a throughput roofline for a given application and takes into account data compression/expansion with data integration applications. It also includes cost information (if available), so that the deployment choices can be based on cost-benefit analyses, not just throughput performance.

4.2 Applications

The two applications used to illustrate the models in this chapter are BLAST and an ML handwriting recognition application. We describe each in turn.

4.2.1 BLAST

The stages of our BLASTN implementation mirror the stages of the NCBI BLASTN computation pipeline, shown in Figure 4.1, and is built using the Mercator framework on a GPU [30]. The DNA database to be searched, represented in FASTA format, is first converted to two bits per DNA base. This is a pre-processing step, `fa_2bit`, from DIBS that is implemented on an FPGA [37]. In the next computational stage, `seed match`, each byte-aligned 8-mer (8-base word) of the database is checked to see whether it appears in a hash table (stored in GPU DRAM) constructed from all 8-mers of the query sequence. If the 8-mer at database position p does appear in the table, a third stage, `seed enumeration`, accesses the table to enumerate all positions q at which it appears, generating one or more 8-mer matches (p, q). These matches are passed to the fourth stage, `small extension`, which attempts to extend each match to the left and right by up to 3 bases. If a match (p, q) can be extended to a total length of at least 11, it is passed on to the final stage, `ungapped extension`, which extends the match to the left and right, this time allowing scoring of both matches and mismatches. Our implementation limits ungapped extension to at most a fixed-size window (currently 128 bases) centered on the initial seed match. Only seed matches whose highest-scoring ungapped extension score above a specified threshold are returned for further processing. Our implementation does not presently perform gapped extension [6], but for BLASTN, that stage takes negligible time compared to the rest of the pipeline [55] and would be implemented on the host processor.

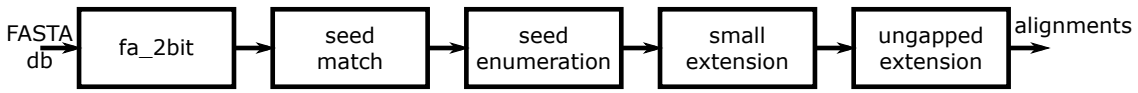


Figure 4.1: BLAST application.

Most stages of BLASTN act as filters over either database positions (seed matching) or matches (small and ungapped extension). Their task is to eliminate inputs that should not

proceed to the next stage. Seed matching in particular is a highly effective filter, eliminating the vast majority of input 8-mers, for query lengths much less than 2^{16} bases. Seed enumeration, in contrast, may produce multiple matches per input position if the same 8-mer occurs at several places in the query. Except for highly repetitive query sequences, this stage produces on average 1-2 matches per input position.

All stages of BLASTN produce a variable number of outputs per input, and most produce zero outputs for the majority of their inputs. On a SIMD processor such as a GPU, executing all stages of BLASTN independently in each thread will result in many threads discarding their inputs and becoming idle early in the computation, resulting in many wasted cycles. The Mercator system therefore inserts queues between each stage to collect and redistribute work among threads before executing the next stage. These queues have limited size, so each stage may need to be executed multiple times; scheduling execution of stages is performed so as to maximize GPU thread occupancy and minimize overhead [77].

4.2.2 ML

The Optidigits library is a representation of handwriting data available through the UC Irvine Machine Learning Repository [4]. This well known data set has a large number of hand written digits ranging from 0 to 9 represented in a 32×32 binary matrix. This data resides in a text file containing all the digits in an ASCII `char` matrix with a corresponding label that identifies what the handwriting raster is supposed to represent. In the original DIBS this database is transformed into a set of tiff images as a potential input to a machine learning application.

In this work we make a change to this application to help lessen the impact of data communications on the overall application throughput. Instead of transforming the ASCII matrices to a tiff format image we make the choice to compact the binary values into integers (one bit per pixel), resulting in an output size of 128 bytes instead of 1.2 kiB per image. This transformation results in no loss of data fidelity, a 10 fold reduction in network usage, and only requires a small pre-processing step of adding a `.png` header and footer before being fed to the downstream ML computation.

Handwritten digit recognition is a classic example of a machine-learning application. The model we utilize is trained on the well-known MNIST handwritten digits dataset, that consists of 60,000 handwritten, grayscale digits in a 32×32 pixel format.

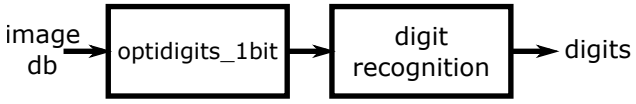


Figure 4.2: ML application – handwriting recognition.

The model is designed with a VGG-like architecture [87] and has two main aspects: a feature extraction front-end and the classifier backend. The feature extraction front-end begins with a single convolutional layer, utilizing a small-sized (3, 3) filter and 32 filters followed by a max-pooling layer. To improve classification accuracy, we then add two additional convolutional layers, each with the same filter size as previously used, but we increase the number of filters in each layer to 64. These layers are again followed by a max-pooling layer. Subsequently, the filter maps are flattened to provide features to the classifier.

Since we are dealing with a multi-class classification task, we require an output layer with ten nodes to predict the probability distribution of an image belonging to each of the classes. This requires the use of a softmax activation function. Between the front-end feature extractor and the classifier, we add a dense layer with 100 nodes to help with feature interpretation.

All layers use a Rectified Linear Activation Function (ReLU) and the 'He' weight initialization scheme, both widely used best practices for this type of problem specifically.

For training, the stochastic gradient descent optimizer is configured with a learning rate of 0.01 and a momentum of 0.9. The categorical cross-entropy loss function will be optimized, suitable for multi-class classification. Each of the images in the Optidigits library is then fed into the model, which subsequently gives us a multi-class probability distribution for each of the digit classes.

4.3 Queueing Theory Model

We develop and extend a model that seeks to predict end-to-end performance of a given data streaming application targeting applications that can be characterized as separable, multi-stage computations. The model is agnostic to the architecture used for the compute engines, supporting processor cores as well as accelerators. The model aims to help a programmer decide where to spend resources to improve the overall running time of a streaming application. In this section we will both describe the model and illustrate its use as a guiding hand for a streaming data application utilizing a combination of FPGAs, GPUs, and solutions for networking data between computational resources. Furthermore it will help us determine effective strategies when it comes to multi-node data streaming applications.

Figure 2.1 illustrates an example streaming application with two compute nodes (labeled `Stage A` and `Stage B`). Data outbound from `Stage A` is delivered, as input, to `Stage B` by the run-time system. For the following description of the model, `Stage A` is the data integration and `Stage B` is the computation of interest. In the model we make the assumption that the asymptotic complexity stays linear for all stages of the streaming application. We also make the assumption that the resources are dedicated to the current task and not shared as it could be in a cloud system scenario. This paradigm readily supports the two nodes being executed on distinct execution platforms, whether they be processor cores, FPGAs, GPUs, or some other accelerator, and the data delivery might be via shared memory, PCIe bus, or the network. When modeling the flow of data down the pipeline, it is prudent to explicitly recognize that this data movement might be the primary contributing factor to the overall performance, and as such should be included in the model by adding more nodes to represent their contribution to the overall data rate. To explain and illustrate the usage of this model we will utilize a pair of applications. First, we combine the data integration task `fa_2bit` running on an FPGA machine feeding data to a BLAST implementation running on a GPU system via a network link. Second we pair the modified data integration task `optidigits_1bit` running on an FPGA feeding data to an ML handwriting recognition implementation executing on a GPU in the same system.

4.3.1 Model Technical details

Our model is derived from previous work by Dor et al. [34], Padmanahban et al. [73], Beard and Chamberlain [12], Timcheck and Buhler [94], and Plano and Buhler [78] to develop an analytic queueing model of a data streaming application, beginning with the BLAST application. Starting from the conceptual diagram of Figure 4.1, additional blocks are added that represent potential performance bottlenecks in the flow of data through the complete application. In our instantiation, the accelerators (both FPGA and GPU) are connected to their host systems via a PCIe bus. In addition, there is a network connection from the system hosting the FPGA to the system hosting the GPU. Adding these blocks into our system we have the full version of the model as shown in Figure 4.3.

In Figure 4.3, the top row represents the system hosting the FPGA, responsible for the `fa_2bit` data transformation. The second row represents the network connection between the two host systems, and the third row represents the system hosting the GPU, responsible for remainder of the comparison pipeline. Note the presence on each host system of the PCIe block both before and after the computation mapped to the respective accelerator. This represents the data transfer both to the accelerator and from the accelerator back to host memory.

We can directly transform this representation into a queueing network by replacing each block (or node) of Figure 4.3 with a queueing station, resulting in the queueing network of Figure 4.4. Each queueing station is comprised of a FIFO queue and its associated server. The service capacity is modeled by a mean service rate μ_i, expressed in bytes/s, that represents the maximum rate at which the server can ingest (process or communicate) data.

Each of the nodes in Figure 4.3 and the corresponding queueing station of Figure 4.4 consumes data from its incoming edge(s) at mean rate λ_i. The nodes implementing communications links will deliver data out at the same rate ($\lambda_{i+1} = \lambda_i$). Computational nodes, however, will have a data volume gain or loss denoted by γ_i, reflecting the notion that either the format of the data has been transformed or (in many cases) the computation is a filter and many input data elements do not generate output. Therefore,

$$\lambda_{i+1} = \gamma_i \lambda_i, \qquad i \geq 1. \tag{4.1}$$

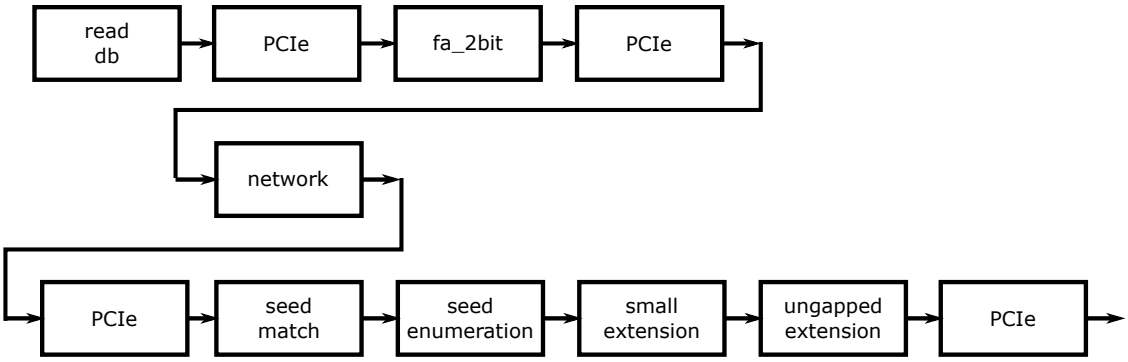


Figure 4.3: Flow graph for BLAST application.

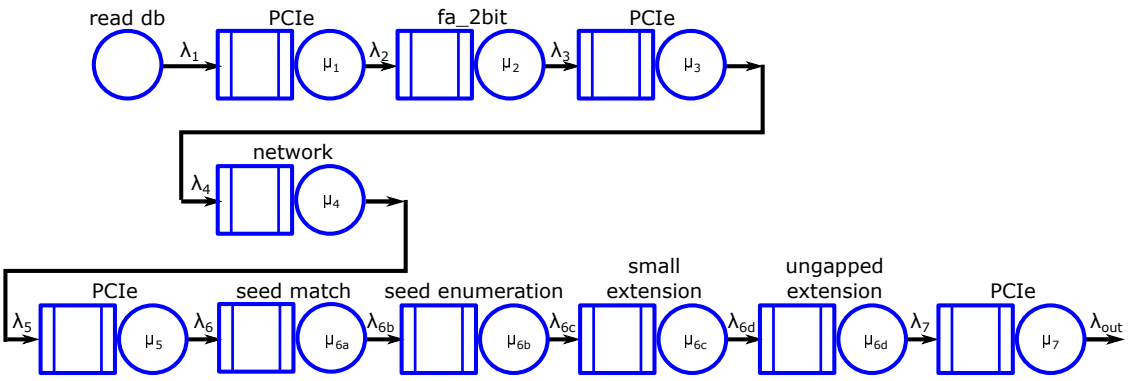


Figure 4.4: Queueing network for BLAST application.

With $\gamma_i = 1$ for all nodes representing communications, the mean data rate into each node is shown below. If we define the cumulative gain up to node i as

$$\Gamma_i = \prod_{k=1}^{i-1} \gamma_k, \qquad i > 1, \tag{4.2}$$

then the mean data rate into each node can be expressed as

$$\lambda_i = \Gamma_i \lambda_1, \qquad i > 1. \tag{4.3}$$

The above description assumes a one-to-one transformation of blocks in Figure 4.3 to queueing stations in Figure 4.4. However, it is difficult to separately measure (and therefore reason about) the distinct blocks in the comparison pipeline executing on the GPU. We will instead merge these blocks in the queueing network model into a single server (and associated queue),

resulting in the queueing network of Figure 4.5. It is this model that we will exploit for the results that are presented below.

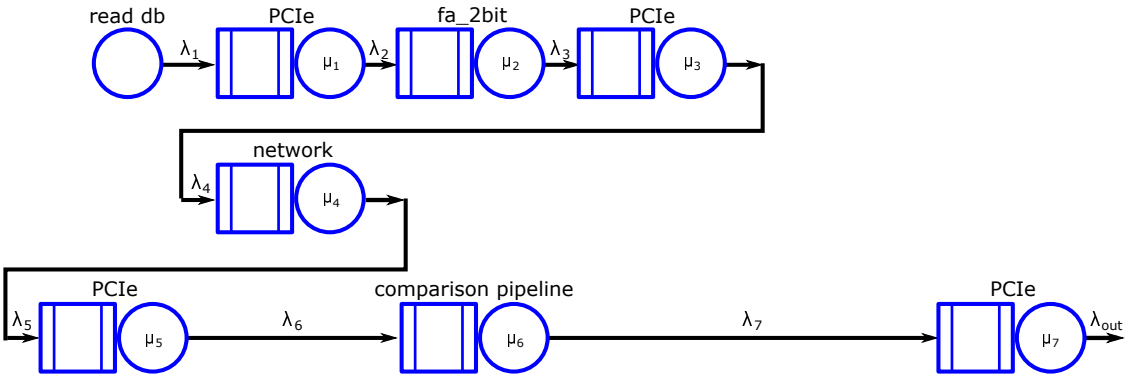


Figure 4.5: Modified queueing network for BLAST application.

In a similar way, our handwriting recognition application is transformed from the initial diagram shown in Figure 4.2 to the flow graph of Figure 4.6, which makes explicit reference to the PCIe bus to and from the FPGA and the PCIe bus to and from the GPU. Figure 4.6 is then transformed in a straightforward way into the queueing network model of Figure 4.7.

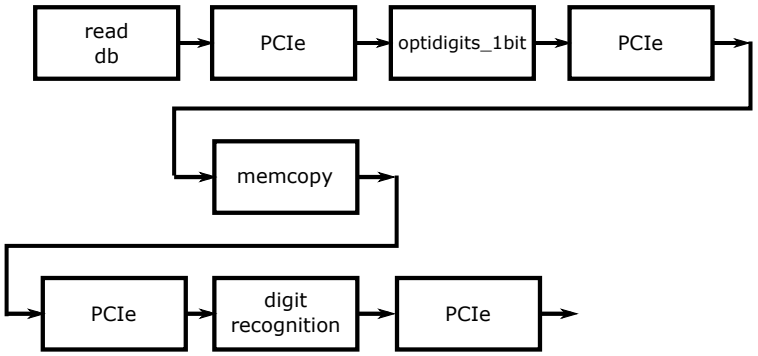


Figure 4.6: Flow graph for ML application.

To simplify the analysis, we will make the assumption that all of the queueing networks are separable, meaning that we can analyze each queueing station independently and then combine their results. This condition holds as long as the physical queues are large enough so that they do not regularly fill (i.e., their probability of filling is low) and/or the network is in the class BCMP [10], both of which are often (but not always) true in these cases.

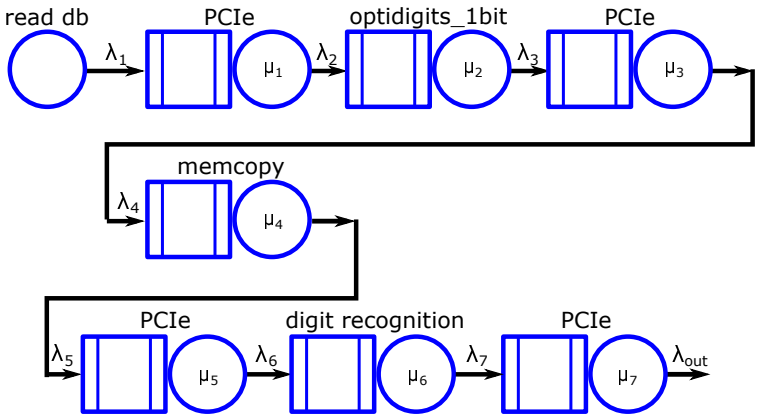


Figure 4.7: Queueing network for ML application.

Our initial interest is in the performance that is achievable in this model. Fortunately, that is straightforward to determine in queueing network models of this type. The service rate at each queueing station establishes the flow capacity at the input to that station (i.e., $\lambda_i < \mu_i$).

Note, as this model allows for empty queues it is required that the data rate be strictly less than the service rate at each node. With knowledge of the service rates, μ_i, (which can be measured empirically in isolation) and the data volume gains, γ_i, (also empririically determined) one expresses the ingest rate at the source, λ_1, as the solution to a flow maximization problem over the graph with individual flow constraints given by the relevant service rates. For arbitrary directed acyclic graph topologies, an efficient solution to this flow maximization problem is given by [12].

We denote the overall throughput by the ingest rate at the first node,

$$Tput = \lambda_1. \tag{4.4}$$

Finally, in addition to the performance achieved, we are also interested in the cost effectiveness of the deployment. This can be any number of metrics a user may desire as long as it is explicitly known. For our example the monetary value for the time spent on a cloud machine from amazon AWS is used but a measurement of energy consumption If c_r is the cost per unit time for resource r, the total cost is just the sum of utilized resources.

$$C = \sum_{r \in R} c_r \tag{4.5}$$

where R is the set of resources (i.e., AWS instances) utilized.

The cost-performance is simply the ratio of the cost to the throughput, $C/Tput$. With C having units of cost/unit time and $Tput$ having units of bytes/unit time, the cost-performance will have units cost/byte.

4.4 Model in Practice

For our implementations of BLAST and ML we target heterogeneous hardware for both the data transformation and final data application.

To test the effectiveness of this model we take empirical measurements of a heterogeneous, multi-node implementation of BLAST running in the Amazon AWS cloud and ML running on dedicated hardware. Table 4.1 gives parameters of the hardware used in these two scenarios.

Table 4.1: AWS EC2 Instances used for BLAST and ORNL and WU machines used for ML.

Machine	CPU	Memory	Accelerator	Cost
AWS F1.2xlarge	8× Intel Xeon E5-2686 v4 @ 2.3 GHz	122 GiB	Virtex UltraScale+ VU9P with 64 GiB	$1.65/hr
AWS g4dn.xlarge	4× Intex Xeon Platinum 8259CL @ 2.5 GHz	16 GiB	Nvidia Tesla T4 with 16 GiB	$0.526/hr
ORNL	24× Intel Xeon Skylake @ 2.1 GHz	94 GiB	UltraScale+ XCU250 with 64 GiB Nvidia Tesla P100 with 12 GiB	N/A

4.4.1 BLAST Implementation Specifics

The stages of our BLAST implementation mirror the stages of the NCBI BLASTN computation pipeline and is built using the Mercator framework on a GPU [30]. The DNA database to be searched, represented in FASTA format, is first converted to two bits per DNA base utilizing an FPGA for the data integration task. The task is run on a f1.xlarge instance which utilizes a Xilinx Virtex UltraScale+ VU9P card. The card is programmed using the Vitis HLS tools utilizing OpenCL HLS using the dataflow programming model. The interface to the global memory uses separate buses for inputs and outputs working on `uint16` and `char16` vector data types respectively. The database is split into chunks where each chunk is processed in-order and sent to a GPU machine running the BLAST algorithm.

The GPU machine running the BLAST application is an AWS g4dn.xlarge machine which utilizes a Nvidia Tesla T4. As the host program receives data chunks over the network they

are packaged and queued up as input data for the BLAST execution on the GPU. Each GPU execution is launched asynchronously with the CPU host program so it is free to perform other tasks while the GPU completes its execution.

4.4.2 ML Implementation Specifics

The initial stage of our ML application, `optidigits_1bit`, is implemented on a Xilinx UltrasScale+ XCU250 FPGA board installed in a Xeon Skylake host that also has an Nvidia Tesla P100 GPU. The `digit recognition` stage of the ML application is executed on the Nvidia GPU. This system was made available to us by Oak Ridge National Laboratory.

4.4.3 Network Connection

In streaming applications similar to the ones presented here we would ideally like to have a large memory storage easily accessible by all compute nodes as data becomes available. However, this is easier said than done. Unfortunately the network facilitating data transfer between multiple machines is typically far slower than the internal memory buses for a machine. As this is often a critical task we give consideration to three different utilities to facilitate data movement across the network. In Table 4.4, μ_4^B refers to three different types of network utilities and their throughput as measured on the AWS EC2 machines listed in Table 4.1 and the internal virtual private cloud (VPC) network in the US West region. Secure Shell Copy (`scp`) is a ubiquitous way to move files via the terminal in Linux systems. The measured throughput is the result of a file copy from one system to another, however, along with being slowest this comes with two major drawbacks. One, this writes a file to disk requiring it to be loaded into the program space for use, needing extra time and resources. Two, the overhead of the secure shell protocol is substantial.

Apache Kafka [54] is a protocol designed for streaming applications using a subscription model. The protocol is designed for both small, short latency messages and longer bulk style messaging. In our tests we observed that although Kafka performs marginally better than `scp` the overhead of the Kafka system results in poor throughput.

In an attempt to solve poor performance from off-the-shelf solutions we implemented our own multi-threaded TCP socket solution using the Boost ASIO libraries. This solution creates a server queue to hold data as it is ready to be sent to the following client node. When the server has data to send, it immediately sends it to the client. The client thread then places said data on a queue for the eventual host program to consume when compute resources are available. This solution far outperforms our other two explored solutions, resulting in more than 2× the speed of `scp` and more than 1.5× the speed of Kafka.

4.4.4 Empirical Measurements

Here we show our measured values that are used as input parameters for our proposed model in Tables 4.2, 4.3, and 4.4. For the most part, the data volume gain figures are from first principles (e.g., packing 4 ASCII characters into a single byte is a reduction in data volume by a factor of four). The sole exception is the gain in the BLAST comparison pipeline, γ_6^B, which will depend upon the combination of query and database. In our experimental cases (as is true for typical usage of the BLAST application [55]), the output data volume is quite small, so the impact on performance is negligible. The reported value is the mean over our experimental runs.

Table 4.2: Data Volume Gain at each Queueing Server (BLAST).

Queueing station	Symbol	Expression	Value	Symbol	Expression	Value
PCIe to FPGA	γ_1	λ_2/λ_1	1			
`fa_2bit`	γ_2^B	λ_3/λ_2	0.25	Γ_2	γ_1	1
PCIe from FPGA	γ_3	λ_4/λ_3	1	Γ_3^B	$\gamma_1\gamma_2^B$	0.25
Network	γ_4	λ_5/λ_4	1	Γ_4^B	$\gamma_1\gamma_2^B\gamma_3$	0.25
PCIe to GPU	γ_5	λ_6/λ_5	1	Γ_5^B	$\prod_{i=1}^{4}\gamma_i$	0.25
`comparison pipeline`	γ_6^B	λ_7/λ_6	4.9×10^{-6}	Γ_6^B	$\prod_{i=1}^{5}\gamma_i$	0.25
PCIe from GPU	γ_7	λ_{out}/λ_7	1	Γ_7^B	$\prod_{i=1}^{6}\gamma_i$	1.2×10^{-6}

Contrasting this, the service rates shown in Table 4.4 are primarily empirically measured, in isolation, without the rest of the application pipeline executing. In this way, we can determine the capacity of that particular pipeline stage. A few rates have been reported in the literature, these are each noted in the table.

Table 4.3: Data Volume Gain at each Queueing Server (ML).

Queueing station	Symbol	Expression	Value	Symbol	Expression	Value
PCIe to FPGA	γ_1	λ_2/λ_1	1			
`optidigits_bit`	γ_2^M	λ_3/λ_2	0.125	Γ_2	γ_1	1
PCIe from FPGA	γ_3	λ_4/λ_3	1	Γ_3^M	$\gamma_1\gamma_2^M$	0.125
Memcopy	γ_4	λ_5/λ_4	1	Γ_4^M	$\gamma_1\gamma_2^M\gamma_3$	0.125
PCIe to GPU	γ_5	λ_6/λ_5	1	Γ_5^M	$\prod_{i=1}^{4}\gamma_i$	0.125
`digit recognition`	γ_6^M	λ_7/λ_6	0.03125	Γ_6^M	$\prod_{i=1}^{5}\gamma_i$	0.125
PCIe from GPU	γ_7	λ_{out}/λ_7	1	Γ_7^M	$\prod_{i=1}^{6}\gamma_i$	3.9×10^{-3}

4.5 Performance Results

4.5.1 BLAST

Table 4.5 shows the normalized service rates for the BLAST implementation of Figure 4.5 (using the Boost ASIO libraries for networking) and the maximum achievable throughput based on Equation (4.7). When just looking at performance, we can ignore Equation (4.5) and focus on Equation (4.4). To ensure that the throughput is achievable at each queueing station i, it is sufficient to have $\lambda_i < \mu_i$. However, we find it more convenient to re-normalize all the individual flow rates λ_i to the ingest rate at stage 1, λ_1. To enable this, we define a normalized service rate, $\hat{\mu}_i = \mu_i/\Gamma_i$, which represents the service rate achievable at station i in units of the ingest rate at the beginning of the pipeline. For all downstream stations the flow constraint can be then expressed as

$$\lambda_1 < \hat{\mu}_i, \qquad i > 1. \tag{4.6}$$

With this normalization, the maximum ingest rate is simply the minimum normalized service rate,

$$\lambda_1 < \min_i \hat{\mu}_i \tag{4.7}$$

and the pipeline stage that determines that value is the bottleneck stage. (Note, for $\hat{\mu_7}$, the data reduction is sufficient such that the normalized service time will not be a limiting factor.) For this application, the GPU-deployed `comparison pipeline` is the rate-limiting stage.

Table 4.4: Capacity (Service Rate) of each Queueing Server.

Queueing station	Symbol	Value
PCIe to FPGA	μ_1	1.1 GB/s
`fa_2bit` (FPGA)	μ_2^B	1.2 GB/s
`fa_2bit` (HARPv2)	μ_2^B	15.3 GB/s
`fa_2bit` (CPU)	μ_2^B	23.4 MB/s (Note 1)
`optidigits_1bit` (FPGA)	μ_2^M	250 MB/s
`optidigits_1bit` (CPU)	μ_2^B	133 MB/s (Note 1)
PCIe from FPGA	μ_3	940 MB/s
Network (`scp`)	μ_4^B	127 MB/s
Network (Kafka)	μ_4^B	178 MB/s
Network (Boost ASIO)	μ_4^B	277 MB/s
Memcopy	μ_4^M	1.3 GB/s
PCIe to GPU	μ_5	6.3 GB/s
`comparison pipeline` (T4)	μ_6^B	137 MB/s
`digit recognition` (CPU)	μ_6^M	70 kB/s
`digit recognition` (GPU)	μ_6^M	90 kB/s
PCIe from GPU	μ_7	6.6 GB/s
`gapped extension`	μ_8	48.9 KB/s (Note 2)

Notes: (1) from [22], (2) from [55].

Table 4.5: BLAST Figure 4.5 Modeled Performance.

$\hat{\mu_2^B}$ GB/s	$\hat{\mu_3}$ GB/s	$\hat{\mu_4^B}$ GB/s	$\hat{\mu_5}$ GB/s	$\hat{\mu_6^B}$ GB/s	$\hat{\mu_7}$ GB/s	λ_1 GB/s
1.2	3.8	1.1	25	0.5	> 100	0.5

For the complete application, the empirical data rate that is achieved is 355 MB/s, a bit below the predicted 500 MB/s. This is not surprising for a pair of reasons. First, Equation (4.7) gives an upper bound on throughput, not a nominal predicted value. Second, there are any number of additional overheads in the execution of the complete pipeline that will have the effect of decreasing the achievable throughput. However, despite the room for improvement in an individual implementation the model can help us where and how to parallelize once the ceiling of the original implementation has been reached.

4.5.2 ML

Similar to the approach we used for BLAST, we can normalize each of the service rates in Figure 4.7 to the ingest rate. These normalized values are shown in Table 4.6. Again, the

GPU is the limiting factor, in this case by quite a bit (the next lowest rate constraint is over two orders-of-magnitude larger). The empirical measurements of the full application are indistinguishable from the isolated GPU measurements, which is not terribly surprising given the performance capabilities of each stage of the computation. Here, replicating the digit recognition stage on multiple instances is clearly going to benefit performance.

Table 4.6: ML Figure 4.7 Modeled Performance.

$\hat{\mu_2^M}$	$\hat{\mu_3}$	$\hat{\mu_4^M}$	$\hat{\mu_5}$	$\hat{\mu_6^M}$	$\hat{\mu_7}$	λ_1
MB/s	GB/s	GB/s	GB/s	MB/s	GB/s	MB/s
250	7.5	10	50	0.72	> 100	0.72

4.6 Utilizing the Model for Implementation Decisions

4.6.1 Alternative Topology

Regardless of how the actualized program performed versus the model, we can still use it to reason about how performance should be improved. For our example application of BLAST, our normalized data rates point to the GPU implementation being the bottle neck for this system. Utilizing flow analysis we can predict where the bottleneck will be and also how to scale nodes by duplication as they are separable. In Figure 4.8 we display a possible topology that can be utilized for our particular setup. From our measurements we determine that the actual machine running BLAST is the bottleneck and not the network as would typically expected. By multiplying the number of GPU instances by 3 and assume that they run independent of each other a possible service rate of the BLAST application could be 411 MB/s which would then create a bottleneck at the network layer. Ideally, a programmer should be able to see that focusing on an implementation that can take advantage of running on multiple systems will net the most performance versus trying to improve pieces that might be out of their control. Beyond this, finding ways to eliminate nodes might also become a viable option. A machine that, for instance, has both accelerators residing in the same system can eliminate the network queue entirely from the equation. However, in instances where large compute resources exist separate from the data source one could potentially utilize other means of eliminating service queues from their dataflow application.

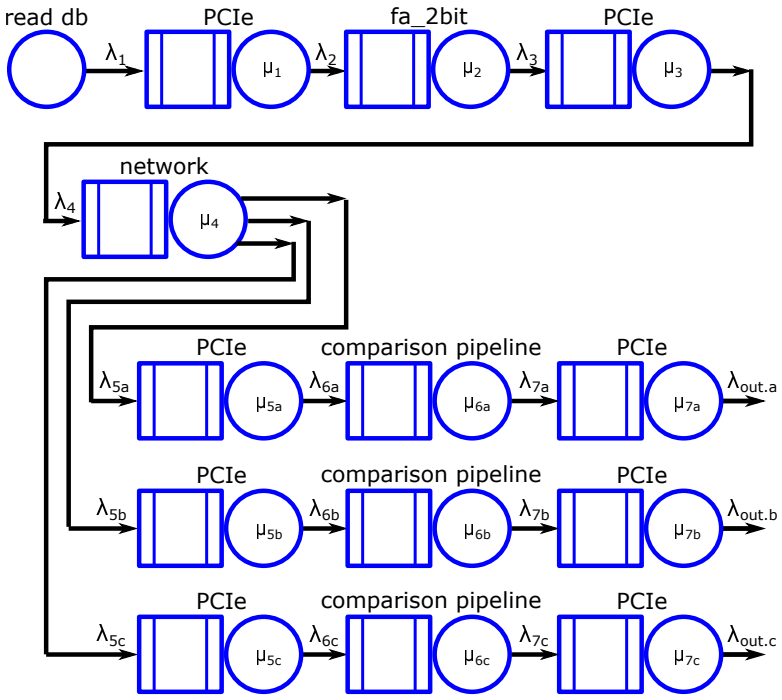


Figure 4.8: Queueing network for multiple BLAST comparison pipelines.

4.6.2 Cost Modeling

Using the fact that our BLAST application is executing on AWS, we have good cost data for our execution platform. (This is often not the case.) With this information, we can derive the cost of running a streaming computation in this way. In Table 4.1 the costs of using an on demand EC2 instance for both the `g4dn` machine and the `F1` instance are listed. Utilizing Equation (4.5) we can now determine what would be a cost per byte on our streaming computation system for BLAST. Given that our BLAST application runs at 355 MB/s and our cost is \$2.176/hr, our total cost per MB is $\$1.70 \times 10^{-6}$, or \$1.70 per TB of data running through the BLAST application.

Although the cost of using an `F1` instance may look rather high compared to the `g4dn` instance we can utilize our model to investigate if deploying the `F1` instance is worthwhile. Replacing the FPGA instance with a free tier CPU instance we can then eliminate the extra PCIe queuing stations and we can further estimate the throughput of the free tier instance via μ_2^B. The modeled performance for the BLAST application with a free instance running `fa_2bit` is shown in Table 4.7.

Table 4.7: BLAST modeled performance utilizing a free tier CPU node.

$\hat{\mu_2^B}$ GB/s	$\hat{\mu_4^B}$ GB/s	$\hat{\mu_5}$ GB/s	$\hat{\mu_6^M}$ GB/s	$\hat{\mu_7}$ MB/s	λ_1 GB/s
0.0234	1.1	25	0.5	> 100	0.0234

Now the cost of running the BLAST application becomes just the cost of the `g4dn` instance: \$0.526/hr and the $Tput = 0.0234$ GB/s. The cost-performance value of this setup now becomes $\$6.244 \times 10^{-6}$ per MB, over three times the cost of the previous setup that includes the `F1` machine.

4.7 Conclusion

The model presented here is designed to help developers reason about streaming data applications in a hardware agnostic way and better estimate how resources can be allocated in streaming applications. By taking isolated measurements of individual components in a system a higher-level overview of the system can be constructed. Through our measurements we see that we are able to successfully predict roofline performance for a given application.

While this is exciting to see for our results, there are some things this model can not predict. As it stands utilizing our queuing theory model to predict things like latency and queue occupancy is not feasible with just isolated measurements of mean service times. This is because queueing theory requires a fuller picture of the distributions of the arrival processes and service processes before it is capable of analyzing latency and queue occupancy. While collecting histograms of service times might be feasible, the overhead of doing so is substantially higher than measuring moments.

Furthermore, the gap of the roofline model from actual measured performance does not tell the full story on interactions that happen within the system. For example, our GPU implementation of the BLAST comparison pipeline queues up a fairly large chunk of data prior to delivering it to the GPU, and this activity is not well represented in our queueing model.

As we would like to improve prediction without creating a larger burden on the developer by creating new things to test and account for, in the next chapter we turn to network calculus

to cover these concerns. Rather than require the measurement (or approximation) of the distribution of service times, network calculus can instead use bounds on service times, which are not substantially more difficult to acquire than averages. In addition, there are network calculus models that explicitly incorporate intentional buffering prior to initiating a service.

Chapter 5

Network Calculus for Streaming Algorithms

5.1 Introduction

Data movement, as discussed prior, is one of the most critical aspects of streaming algorithm tasks. When trying to discern how a streaming algorithm performs on a high level, how the data moves through the system is paramount to a working model. In the previous chapter we focused on a queuing theory model for roofline prediction, however without further investigation (primarily into the distributions of the arrival and service processes) this model can't make claims about data latency or bounds on performance. Ideally a model for guidance should not burden the developer with additional data gathering (e.g., to empirically measure distributions) when trying to discern how to spend resources, as the data gathering and analysis of it just becomes another burden. Utilizing similar measurements in the queuing theory model we turn to network calculus to try and address these concerns.

5.2 Introduction to Network Calculus

Network calculus is a modeling approach, similar to queuing theory, that is designed to analyze systems that utilize networks of queues and has primarily used to analyze bounds and model performance in networking systems. It relies on the min-plus and max-plus algebras which define a different set of operators compared to normal algebra. In min-plus algebra, addition is replaced by the infimum operator and multiplication is replaced with addition. Similarly in max-plus algebra, addition is replaced by the supremum and, once

again, multiplication is replaced with addition. These two algebras are used in conjunction with the convolution operator to reason about data as it traverses a system.

In network calculus data is modeled as a cumulative function with respect to time to represent the flow in and out of systems. Systems are modeled in a similar fashion with curves representing guarantees on flow into and out of the system known as arrival and service curves respectively. While these flows correspond to the arrival and service processes of queueing theory, they are characterized in queueing theory by mean rates (and distributions of rates).

The discussion below follows the exposition provided by Boudec and Thiran [61].

Consider a data flow, in units of bits, $r(t)$, arriving at a system and let $\alpha(t)$ be a wide-sense increasing function with $\alpha(0) = 0$. The flow is constrained by $\alpha(t)$ and is an arrival curve if and only if for any $0 \leq s \leq t$:

$$r(t) - r(s) \leq \alpha(t - s).$$

Following a similar logic the system offers a service guarantee for an output flow $r^*(t)$. Allow $\beta(t)$ to be a wide-sense increasing function and $\beta(0) = 0$. $\beta(t)$ is a service curve given to the flow $r(t)$ with an output curve $r^*(t)$, defined by:

$$r^*(t) \geq \inf_{s \leq t} \{r(s) + \beta(t - s)\}.$$

Alternatively, this can be written as the min-plus convolution:

$$r^*(t) \geq r(t) \otimes \beta(t).$$

Furthermore, we can define an upper-bound on a service provided defined as:

$$r^*(t) \leq r(t) \otimes \psi(t),$$

where $\psi(t)$ is the maximum (i.e., best case) service curve.

When utilizing a network calculus model it is up to the designer to use appropriate equations to represent arrival and service curves. For arrival curves it is typically standard to model

the data flow using an affine curve known as the leaky bucket arrival curve:

$$\alpha(t) = \begin{cases} R_\alpha(t) + b & \text{if } t > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Here, R_α represents the rate of arrival and b is a burst, that is, how much data can be sent instantaneously. When considering the service curves, these are usually represented as rate latency functions with an associated rate, R_β, and delay, T, associated with them:

$$\beta(t) = \begin{cases} R_\beta(t - T) & \text{if } t > T \\ 0 & \text{otherwise.} \end{cases}$$

By utilizing these models we can begin to reason about bounds on a specific node such as the backlog generated by the flow entering the node, the delay data will experience at a given node, and what is the upper-bound output flow of the node, $\alpha^*(t)$. Figure 5.1 displays a data over time plot of a leaky-bucket arrival curve and two rate latency functions representing both a maximum and normal service curve adapted from [61]. Also in this figure horizontal and vertical lines are included that are meant to represent maximum virtual delay and backlog, respectively. Finally from these two lines we can derive an output flow bound, $\alpha^*(t)$, which, along with the delay and backlog, will be further expanded upon below.

When considering these models it is important to point out that both network calculus and queuing theory as mathematical models are designed to reason about queuing systems and there has been work to explain how one represents network calculus ideas in a queuing theory space [50, 75].

5.3 Network Calculus Modeling

As mentioned prior we want to use network calculus to reason about bounds on a given streaming application that utilizes heterogeneous architectures, however some additional assumptions and modifications to the standard model must be made in order to utilize it properly. Firstly network calculus in its original inception deals with continuous data flows that are bit-by-bit, however in the modern era a majority of network equipment work on

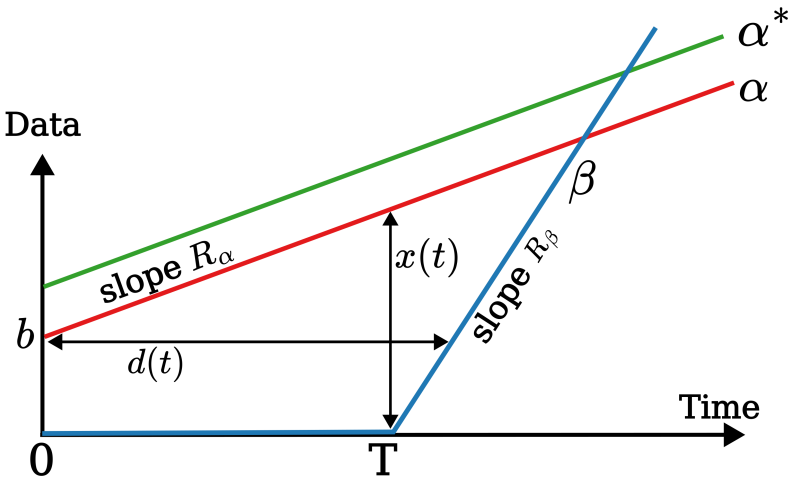


Figure 5.1: Plot of a Leaky Bucket Arrival Curve, α, and a Rate-Latency Service Curve, β, showing the relation of the Backlog, $x(t)$, Virtual Delay, $d(t)$, and Output Flow, α^*, bounds. Adapted from [61].

a per-packet scheme and are similar to jobs flowing through a streaming application. This packetization does indeed have an effect on some of the properties that network calculus models [61] and needs to be accounted for in our final model as well. These adjustments come in the form modifying the arrival and service curves with a variable that describes the the size of the maximum packet l_{max}. Consider a flow $r(t)$ and a packetizer, P^L, the packetized version of the arrival curve, service curve, and maximum service curve are [96]:

$$P^L(r(t)) \leq \alpha(t) + l_{max}1_{t>0}$$

$$\beta'(t) = [\beta(t) - l_{max}]^+$$

$$\psi'(t) = \psi(t).$$

With these adjustments we can now talk about three important bounds previously mentioned and shown in Figure 5.1, virtual delay and backlog. The virtual delay, $d(t)$, is a measure of the maximum amount of time it takes for a system to output the same amount of data sent to the system. For a leaky bucket arrival curve α and a rate latency service curve β the virtual delay is given by:

$$d(t) \leq T_\beta + \frac{b_\alpha}{R_\beta}.$$

The backlog bound, $x(t)$, is a bound on the maximum amount of data that resides in the server before output is sent, and is calculated as the maximum deviation between α and β, In this example it is calculated as:

$$x(t) \leq b_\alpha + R_\alpha T_\beta.$$

Finally we can make an estimation on the output bound on a system, $\alpha^*(t)$. This is known as the output flow bound. This is found by calculating both a min-plus convolution and a min-plus de-convolution utilizing the arrival curve of the node and both the maximum and normal service curves:

$$\alpha^* = (\alpha \otimes \psi) \oslash \beta.$$

While these bounds are beneficial to have, it is important to know that these bounds assume that $R_\alpha \leq R_\beta$. If $R_\alpha > R_\beta$ it is noted in [61] that the bounds are infinite, which is the same result predicted by queuing theory if the arrival rate is greater than the service rate, resulting in an infinite bound on the queue. Taking this into account, there are three particular scenarios that we are interested in: (1) when $R_\alpha < R_\beta$ or standard operation; (2) when $R_\alpha = R_\beta$; and finally (3) when $R_\alpha > R_\beta$. While the bounds are indeed infinite for backlog and virtual delay over the long run, we hypothesize that we can use values given by the model to understand estimates on required queue size for individual nodes as a job traverses a system implementing a streaming data application.

One important aspect of targeting heterogeneous architectures is the need to gather enough data to make dispatching a job worthwhile. The inherent overheads associated with initiating a computation on an attached accelerator, for example, can motivate the aggregation of a minimum data volume at the input to the accelerator prior to dispatching the job to the accelerator. We call this metric the job ratio. To reflect this in the service curve representations, we have made a modification to how initial delay is calculated at these nodes. For a node n that collects data of size b_n prior to initiation and b_n is larger than the burst rate of the previous node ($b_n > b^*_{n-1}$) then the latency at node n is:

$$T^{tot}_n = T^{tot}_{n-1} + \frac{b_n}{R_{\alpha_{n-1}}} + T_n.$$

Intuitively, total latency is the summation of initial delay of the previous nodes, T_{n-1}^{tot}, the time to collect a job from the previous node, $b_n/R_{\alpha_{n-1}}$, and finally the initial delay of the current node, T_n.

5.4 Modeling and Simulation of BLAST

Network calculus can model the structure of a network similar to streaming data applications with directed graphs. Nodes in previous network calculus models typically represent some type network element like a router or switch, each with their own set of arrival curves and service curves for each element. If we restrict the network calculus model into a directed acyclic graph, the resulting chain can represent computation and/or communication similar to our prior queuing theory model. We therefore hypothesize that network calculus is well suited for capturing this type of data movement and can be a viable tool for understanding the performance implications of data channels in streaming environments.

Given a set of N nodes representing stages of a heterogeneous streaming application, we can define network calculus maximum and normal service curves to represent the guarantees on service at each node. Along with the actual compute nodes we can also define two service curves to represent guarantees on data movement nodes. Any sequence of nodes in a contiguous chain can be concatenated together to find the overall service curve for that subset of nodes. When concatenating nodes in a chain the service curves are defined by the minimum service curves for the selected nodes. Through this method we can create models for intermediate systems by finding service curves for a subset of contiguous nodes or create a node that represents the entire system. To test this model we borrow the BLAST application used in the previous chapter. Figure 5.2 repeats Figure 4.5, simply rearranging the position of the queueing stations to all be in a single line. Figure 5.3 illustrates the setup of this application with nodes representing stages of the streaming data application, augmented with ratios to represent the job ratio at said node. For example, the ratio 1:16 associated with stage D indicates that the network link collects 16 data chunks from its upstream neighbor prior to actually delivering data across the network itself.

To improve the visibility into the performance of the system, we take these nodes and model their execution time in a discrete-event simulator facilitated by the SimPy library [88] in Python3. Each node is a server class with an associated queue represented by a SimPy

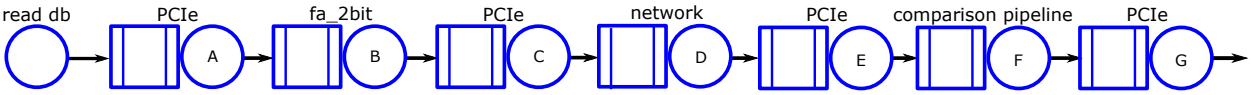
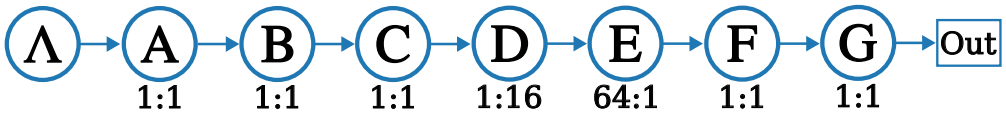


Figure 5.2: BLAST queueing model (from Figure 4.5).



Node	Function	Throughput		
		Average	Minimum	Maximum
Λ	Data Source	N/A	N/A	N/A
A	PCIe link	1.2 GiB/s	1.1 GiB/s	1.2 GiB/s
B	FPGA Computation	1.2 GiB/s	1.2 GiB/s	1.2 GiB/s
C	PCIe link	940 MiB/s	906 MiB/s	958 MiB/s
D	Network Link	277 MiB/s	263 MiB/s	315 MiB/s
E	PCIe link	6.3 GiB/s	6.3 GiB/s	6.2 GiB/s
F	GPU Computation	137 MiB/s	175 MiB/s	89 MiB/s
G	PCIe link	6.6 GiB/s	6.6 GiB/s	6.7 GiB/s

Figure 5.3: Data flow diagram with accompanying node table with names and throughput for BLAST. Nodes represent computations or communications, and the job ratio is shown below each node. Node D decomposes large data blocks from the FPGA for delivery over the network, and Node E composes even larger data blocks for delivery to the GPU. Average, Maximum, and Minimum throughput for each node are also listed, except for Data Source as we assume the source to have infinite throughput (a job will be queued immediately when arriving).

Container which holds the current amount of data in queue for the node. Each server is given a maximum and minimum execution time (derived from empirical measurements of the server in isolation), a data packet size to consume, and data packet size to emit when the execution time has completed. At each time step a server will check if there is enough data in its store to start execution, if so it will then consume that data from its queue and sleep for its given execution time otherwise it will check at the next time step. The time chosen for execution is chosen from a uniform random distribution using the minimum and maximum times as bounds. Again, we normalize the data volumes at each stage referred to the input, as some stages have a natural lossless data compression. In the results section we report all of the following: network calculus predictions on bounds, the results of the original

$M/M/1$ queuing theory model and empirical measured performance (already presented in Chapter 4), and our simulated system performance.

The predictions from our network calculus model and discrete-event simulation are depicted in Figure 5.4. The service curve, represented by $\beta(t)$, corresponds to the lower bound of predicted performance for the entire system. The arrival curve, represented by $\alpha(t)$, corresponds to an upper bound on performance. The output flow bound, represented by $\alpha^*(t)$, is a loose upper bound. The simulated data output is shown by the stairstep curve that stays between the two bounds.

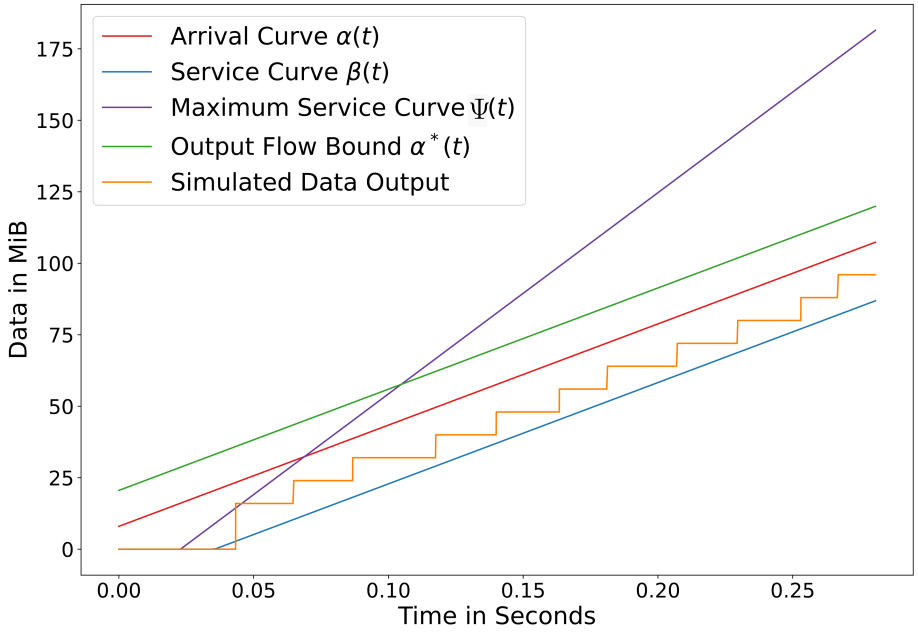


Figure 5.4: Network calculus model results.

Throughput predictions from the various models and experiments are presented in Table 5.1. As is apparent, the network calculus throughput predictions align well with both the discrete-event simulation results and the empirical results reported in Chapter 4.

While these throughput results are clearly of interest, they haven't yet demonstrated the power of network calculus, since they are merely confirming the conclusions from previous models. Additional information we can glean from the network calculus model include the following:

Table 5.1: Streaming data application throughput.

Source	Value
Network calculus upper bound	704 MiB/s
Network calculus lower bound	350 MiB/s
Discrete-event simulation model	353 MiB/s
Queueing theory prediction	500 MiB/s
Measured throughput	355 MiB/s

1. The maximum virtual delay, d, through the system is modeled to be 46.9 ms.

2. The maximum data occupancy resident in the entire system, x, (or backlog bound) is modeled as 20.6 MiB.

Points (1) and (2) above are corroborated by the discrete-event simulation model.

Further capabilities of the network calculus models include the ability to analyze any desired subset of the streaming application separate from the rest of the application. For example, the contributions of the data occupancy bounds that are due to each node in Figure 5.3 can be determined analytically, which can assist a developer in allocating buffers.

5.5 Bump in the Wire Streaming Algorithms

Utilizing network calculus we can model other data streaming applications that utilize other heterogeneous technology (and validate the model predictions using our simulation tool). One that is of particular interest to us is the utilization of what is known as "bump in the wire" communication [16]. Figure 5.5 shows the traditional interconnect for an FPGA accelerator, and Figure 5.6 shows the bump in the wire configuration.

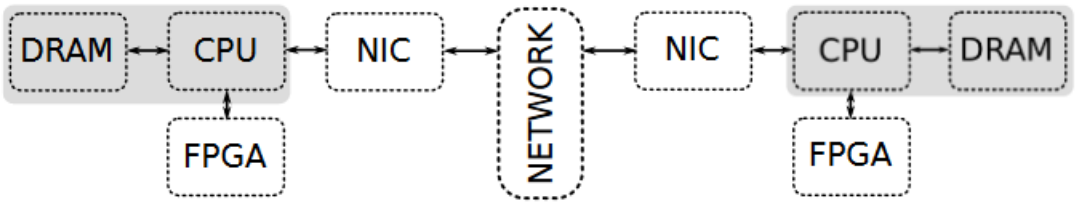


Figure 5.5: Traditional FPGA accelerator [59].

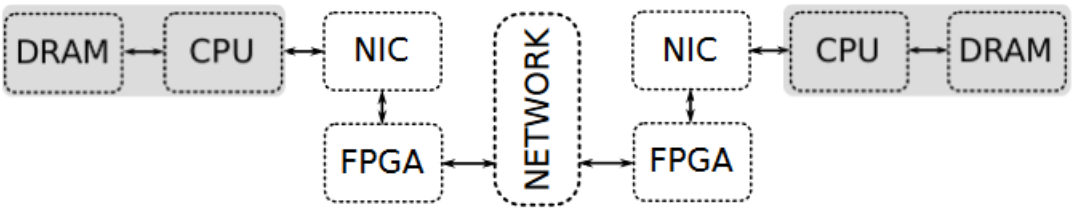


Figure 5.6: Bump in the wire FPGA accelerator [59].

This style of deployment particularly relies on a network connection to pass data to a new node without having to be moved out of the heterogeneous memory pool back to CPU host memory. There are many heterogeneous architectures that implement bump in the wire tech that are primarily deployed in FPGAs, utilizing custom compute alongside network communication. In this scenario instead of a specific streaming algorithm we want to look at adding functionality to a network connection, tasks usually not associated with a specialized algorithm but still desireable in many implementations. Tasks like security and/or compression are often afterthoughts when considering an application development, frequently to be considered essential when it comes to deployment. These algorithms, depending on their implementation, can be considered a type of streaming data application by compressing/encrypting data blocks in chunks and then decompressing/decrypting at the destination. Two FPGAs can be used in conjunction as a source and destination though a network to offload the entirety of this computation from the endpoint CPUs freeing them up for other processes.

Figures 5.7 and 5.8 show the source end flow graphs of the scenario described above, with Figure 5.7 indicating the data flow if the FPGA were installed in the system in the traditional way and Figure 5.8 indicating the data flow in a bump in the wire configuration. Note that the benefit of the bump in the wire configuration is that data no longer need to flow across the PCIe bus to move from the FPGA to the network.

Figure 5.7: Example flow graph for FPGA accelerated compression/encryption using a traditional FPGA interconnection.

The FPGA manufacturer Xilinx maintains a set of of multi-purpose libraries with HLS implementations of various algorithms in the form of function primitives or fully implemented

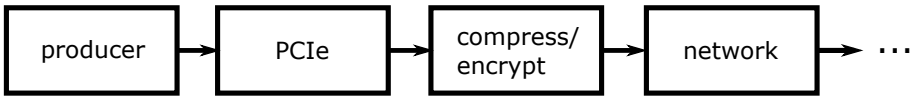


Figure 5.8: Example flow graph for FPGA accelerated compression/encryption using a bump in the wire configuration.

kernels. They are designed to get any developer up and running with implementations of algorithms ranging from image analysis, data analytics, and graph problems to name a few. Within the Vitis libraries are implementations of various data compression and cryptography libraries that we wish to investigate for a bump in the wire implementation. Within the two categories one can find a plethora of various compression and cryptography methods. Here, we have decided to target the LZ4 compression and AES cryptography algorithms which are ubiquitous in their respective spaces. The AES algorithm already exists as a streaming algorithm; no matter the size of the data target, it is broken into 128 bit blocks and each one is encrypted/decrypted in order. In the case of LZ4 textual compression/decompression, a target file or stream of data may need to be chunked and then run through the kernel in order for it to be considered streaming. In the Vitis libraries implementation, a streaming version of the LZ4 algorithm is implemented utilizing stream channels so data can be passed from one kernel to the next in a FIFO. It is important to note, of course, that the effectiveness of compression is dependent on the amount of repeated patterns in the target data and chunked data may reduce similarity for the overall dataset which in turn will reduce the effectiveness of compression.

In this application the amount of compression that the data will experience will effect how much data a downstream node will see until it is decompressed. To account for this in a network calculus model we will want to again normalize data in terms of the input but we want to make note of the possible compression ratio achieved by LZ4. Service curves after compression will then take two forms: one that considers the worst case scenario, a compression ratio of 1.0, and the other being the largest observed compression ratio. As these compression ratios effect how much data is truly going though the individual nodes, the lower bound service curve corresponds to a compression ratio of 1.0 and the maximum service curve will correspond to the maximum compression ratio. In addition, because the data is normalized to the input data volume, the throughput reported by the maximum service curve would be the baseline measured maximum service curve multiplied by the

compression ratio which is then removed from downstream maximum service curves after decompression.

The target platform for this application is the Open Cloud Testbed (OCT) [44, 62] which deploys machines equipped with network capable Xilinx Alveo U280 FPGA cards, which can be targeted by Vitis implementations. Similar to the queuing theory model, we will test each stage in isolation and measure performance in isolation. Compression is implemented via a streaming LZ4 kernel and encryption is provided by a 256-bit CBC AES kernel, both available in the Vitis libraries. Finally the third kernel, the network communication kernel, is a demo implementation of a TCP stack and CMAC kernels that facilitate network communication between two FPGA cards [46, 68]. While Figure 5.8 illustrates the notion of the bump in the wire configuration, Figure 5.9 shows the flow graph of the application we actually model. The measured throughput for each stage are shown in Table 5.2.

Figure 5.9: Actual flow graph for FPGA accelerated compression/encryption using the bump in the wire configuration.

Table 5.2: Listing of functions and their associated throughputs. The compression rates listed here are normalized with respect to their observed compression ratios: 2.2× Average, 1.0× Minimum, and 5.3× Maximum.

	Throughput					
Function	Average		Minimum		Maximum	
Compress	2662	MiB/s	1181	MiB/s	6386	MiB/s
Encrypt	68	MiB/s	56	MiB/s	75	MiB/s
Network	10	GiB/s	10	GiB/s	10	GiB/s
Decrypt	90	MiB/s	77	MiB/s	113	MiB/s
Decompress	1495	MiB/s	1426	MiB/s	1543	MiB/s
PCIe link	11	GiB/s	11	GiB/s	11	GiB/s

The resulting simulation and network calculus model can be seen in Figure 5.10. Like in the previous model we combine all stages of the pipeline to create a single node for our network calculus model to determine latency and backlog bounds. Here we have removed the maximum service curve $\psi(t)$ as it skews the overall graph and is indicative of the maximum observed throughput and also the maximum observed compression. Again we see the simulation curve is below the potential maximum output bound for this system. The quantitative predictions are shown in Table 5.3.

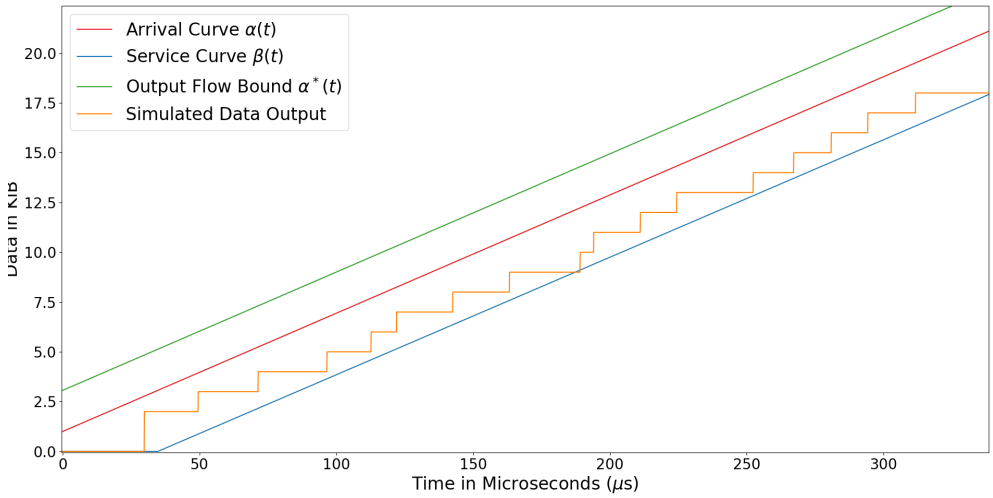


Figure 5.10: Network calculus model for our bump in the wire application.

Table 5.3: Streaming data application throughput.

Source	Value
Network calculus upper bound	313 MiB/s
Network calculus lower bound	59 MiB/s
Discrete-event simulation model	61 MiB/s
Queueing theory prediction	151 MiB/s

1. The maximum virtual delay, d, through the system is modeled to be 38 μs.

2. The maximum data occupancy resident in the entire system, x, (or backlog bound) is modeled as 3 KiB.

Points (1) and (2) above are corroborated by the discrete-event simulation model.

One important thing to note about the simulation is that it is not modeling the breaking of individual chunks of the encrypted AES output and sending them through the network node in individual packets. For ease of simulation we instead assume that data will be gathered at maximum in 1 KiB normalized chunks and then sent over the network. Another simulation shortfall is the lack of a structure to simulate overlapping stream channels which would be utilized in an FPGA deployment to transport data to downstream kernels. Furthermore we would like to corroborate these simulated results with an actual deployment. In the future we would like to show the power of network calculus as a tool to help make decisions about deployment when considering applications that have an arrival rate greater than what can

be provided by the service of the system and when arrival rates need to be changed to accommodate queues that are at risk of overflowing.

Chapter 6

Conclusions and Future Work

Although it can be difficult to reason about the performance of streaming data applications, through general guidance and mathematical modeling reasonable estimates can be made about how an application will behave. The work presented here is intended to help a developer in their quest as they attempt to hit performance goals in systems where disparate compute resources are incorporated into the available design options.

6.1 Conclusions

Streaming applications on heterogeneous architectures can be somewhat convoluted to implement because of memory that travels across host and device boundaries, on top of that programming for such devices can be difficult at first blush. In Chapter 3 we explored implementing data integration tasks, an often underappreciated task, on heterogeneous hardware, specifically targeting FPGAs utilizing OpenCL utilizing two major programming styles available to the platform. When implementing these tasks we found that it was it was important to understand how memory access is handled in an algorithm and how the device handles memory hierarchies.

At a higher level a streaming data application rarely exist in isolation, and understanding how the entire application performs is just as important. Chapter 4 and Chapter 5 utilize queuing theory and network calculus, respectively, to help reason about performance for streaming data applications, each with their own advantages. With queuing theory one can reason about roofline performance for the entire flow and identify nodes that restrict the flow, with the implication that these nodes might require more attention and resources. The model also allows us to analyze the cost effectiveness of a given stage. Network calculus allows us to reason about the performance bounds of a given system as well as important

stats like the amount of data in the system and end-to-end delay of a streaming computation. These two models have the advantage of being hardware agnostic and allow for evaluation of components in isolation allowing a developer to concentrate on individual nodes before worrying about implementing them in a full flow.

6.2 Future Work

We would of course like to improve the quality of programming advice given for streaming data applications for FPGA deployments. When looking at how to effectively compute data integration tasks we found that metadata processing can play an important role in some applications and ideally should also be a target of concern for data integration applications.

With meta data processing we've come across two different types, small up front operations to determine database attributes and larger operations that depend on full database scans to find important characters like record boundaries that are not spaced in a regular manner. It is our thought that finding ways to reason about metadata processing can be important for more than just data integration tasks, but could potentially be impactful for task that can utilize information for more efficient processing.

When it comes to modeling, further use of our network calculus model is of interest to us. In our current model we are strictly bound by the property $R_\alpha \leq R_\beta$ which for our purposes we set the arrival curve of the system to be the rate of R_β, of course when $R_\alpha > R_\beta$ the backlog bound and therefore the queues will grow without bound. While this is fine in a modeled environment, it is unpractical in the real world. In network calculus there already exists a concept of a a variable bit rate (VBR) which defines an arrival curve constrained by two leaky buckets. Such a concept may be useful when the data rate arriving at a system needs to backoff on the rate of data being sent to a particular system to allow for a downstream server to catch up on its own backlog bound. How exactly a system makes such a decision would be another extension of the application of network calculus into streaming algorithms.

Pushing the space in which network network calculus can be useful beyond streaming data applications, FPGA and similar ASIC circuits could potentially use similar models for reasoning about queue size when passing data from one compute kernel to the next. It is unknown if this model has improvements over current HLS queue sizing techniques.

Another area of potential improvement is an investigation of lightweight methods for extracting the performance properties of individual nodes in isolation. There has been work focused on learning these properties in the midst of a full application, both on traditional processors [14] and FPGA-based systems [58], however, the techniques for executing nodes in isolation still require substantial manual effort. This could be improved.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. *arXiv:1603.04467*, 2016.

[2] D. Aggarwal. Exploring the possibility of using a GPU while implementing pipelining to reduce the processing time in the ETL process. *International Journal on Recent and Innovation Trends in Computing and Communication*, 5(6):333–337, June 2017.

[3] A. O. Allen. Elements of queuing theory for system design. *IBM Systems Journal*, 14(2):161–187, 1975.

[4] E. Alpaydin and C. Kaynak. Optical Recognition of Handwritten Digits. UCI Machine Learning Repository, 1998.

[5] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[6] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: A new generation of protein database search programs. *Nucleic Acids Research*, 25:3389–402, 1997.

[7] S. Azodolmolky, R. Nejabati, M. Pazouki, P. Wieder, R. Yahyapour, and D. Simeonidou. An analytical model for software defined networking: A network calculus-based approach. In *Proc. of Global Comm. Conf.*, pages 1397–1402. IEEE, 2013.

[8] M. Bakhouya, S. Suboh, J. Gaber, and T. El-Ghazawi. Analytical modeling and evaluation of on-chip interconnects using network calculus. In *Proc. of 3rd ACM/IEEE Int'l Symp. on Networks-on-Chip*, pages 74–79. IEEE, 2009.

[9] M. Bartík, S. Ubik, and P. Kubalik. LZ4 compression algorithm on FPGA. In *Proc. of IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, pages 179–182. IEEE, 2015.

[10] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios. Open, closed, and mixed networks of queues with different classes of customers. *Journal of the ACM*, 22(2):248–260, 1975.

[11] S. Beamer, K. Asanovic, and D. Patterson. Direction-optimizing breadth-first search. In *Proc. of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2012.

[12] J. Beard and R. Chamberlain. Analysis of a simple approach to modeling performance for streaming data applications. In *Proc. of Int'l Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 345–349. IEEE, Aug. 2013.

[13] J. C. Beard. *Online Modeling and Tuning of Parallel Stream Processing Systems*. PhD thesis, Dept. of Computer Science and Engineering, Washington University in St. Louis, Aug. 2015.

[14] J. C. Beard and R. D. Chamberlain. Run time approximation of non-blocking service rates for streaming systems. In *Proc. of IEEE 17th International Conference on High Performance Computing and Communications (HPCC)*, pages 792–797, Aug. 2015.

[15] J. C. Beard, P. Li, and R. D. Chamberlain. Raftlib: A C++ template library for high performance stream parallel processing. *The International Journal of High Performance Computing Applications*, 31(5):391–404, 2017.

[16] C. Bobda, J. M. Mbongue, P. Chow, M. Ewais, N. Tarafdar, J. C. Vega, K. Eguro, D. Koch, S. Handagala, M. Leeser, M. Herbordt, et al. The future of FPGA acceleration in datacenters and the cloud. *ACM Transactions on Reconfigurable Technology and Systems*, 15(3), 2022.

[17] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, Aug. 2004.

[18] A. M. Cabrera. *Domain Specific Computing in Tightly-Coupled Heterogeneous Systems*. PhD thesis, Washington University in St. Louis, Aug. 2020.

[19] A. M. Cabrera and R. D. Chamberlain. Design and performance evaluation of optimizations for OpenCL FPGA kernels. In *Proc. of High-Performance Extreme Computing Conference (HPEC)*. IEEE, Sept. 2020.

[20] A. M. Cabrera and R. D. Chamberlain. Designing domain specific computing systems. In *Proc. of 28th International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, May 2020.

[21] A. M. Cabrera, C. J. Faber, K. Cepeda, R. Deber, C. Epstein, J. Zheng, R. K. Cytron, and R. D. Chamberlain. Data Integration Benchmark Suite v1. DOI:10.7936/K7NZ8715, Apr. 2018.

[22] A. M. Cabrera, C. J. Faber, K. Cepeda, R. Derber, C. Epstein, J. Zheng, R. K. Cytron, and R. D. Chamberlain. DIBS: A data integration benchmark suite. In *Proc. of ACM/SPIE Int'l Conf. on Performance Engineering Companion*, pages 25–28. ACM, Apr. 2018.

[23] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Proc. of 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 33–36, 2011.

[24] R. D. Chamberlain. Architecturally truly diverse systems: A review. *Future Generation Computer Systems*, 110:33–44, Sept. 2020.

[25] R. D. Chamberlain, M. A. Franklin, E. J. Tyson, J. H. Buckley, J. Buhler, G. Galloway, S. Gayen, M. Hall, B. Shands, and N. Singla. Auto-Pipe: A development environment for streaming applications on architecturally diverse systems. *Computer*, 43(3):42–49, Mar. 2010.

[26] R. D. Chamberlain and B. Shands. Streaming data from disk store to application. In *Proc. of 3rd International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, pages 17–23, Sept. 2005.

[27] R. D. Chamberlain and B. Shands. Direct-attached disk subsystem performance assessment. In *Proc. of 4th International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, Sept. 2007.

[28] P. Chodowiec and K. Gaj. Very compact FPGA implementation of the AES algorithm. In *Proc. of International Workshop on Cryptographic Hardware and Embedded Systems*, pages 319–333. Springer, 2003.

[29] Y. Choi, C.-H. Li, D. D. Silva, A. Bivens, and E. Schenfeld. Adaptive task duplication using on-line bottleneck detection for streaming applications. In *Proc. of 9th Conference on Computing Frontiers*, page 163–172, 2012.

[30] S. V. Cole and J. Buhler. MERCATOR: a GPGPU framework for irregular streaming applications. In *Proc. of 15th Int'l Conf. on High Performance Computing and Simulation*, pages 727–736, July 2017.

[31] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[32] P. Cremonesi and C. Gennaro. Integrated performance models for SPMD applications and MIMD architectures. *IEEE Transactions on Parallel and Distributed Systems*, 13(7):745–757, 2002.

[33] R. Cruz. A calculus for network delay. I. Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, 1991.

[34] R. Dor, J. M. Lancaster, M. A. Franklin, J. Buhler, and R. D. Chamberlain. Using queuing theory to model streaming applications. In *Proc. of 2010 Symposium on Application Accelerators in High Performance Computing*, July 2010.

[35] C. Faber and R. Chamberlain. Application of network calculus models to heterogeneous streaming applications. In *Proc. of 26th IEEE Workshop on Advances in Parallel and Distributed Computational Models (APDCM)*, May 2024.

[36] C. Faber, T. Plano, S. Kodali, Z. Xiao, A. Dwaraki, J. Buhler, R. Chamberlain, and A. Cabrera. Platform agnostic streaming data application performance models. In *Proc. of IEEE/ACM Workshop on Redefining Scalability for Diversely Heterogeneous Architectures*. IEEE, Nov. 2021.

[37] C. J. Faber, A. M. Cabrera, O. Booker, G. Maayan, and R. D. Chamberlain. Data integration tasks on heterogeneous systems using OpenCL. In *Proc. of 7th International Workshop on OpenCL (IWOCL)*, May 2019.

[38] C. J. Faber, S. D. Harris, Z. Xiao, R. D. Chamberlain, and A. M. Cabrera. Challenges designing for FPGAs using high-level synthesis. In *Proc. of IEEE High-Performance Extreme Computing Conference (HPEC)*, Sept. 2022.

[39] Y. Fang, C. Zou, A. J. Elmore, and A. A. Chien. UDP: a programmable accelerator for extract-transform-load workloads and more. In *Proc. of 50th IEEE/ACM International Symposium on Microarchitecture*, pages 55–68. IEEE, 2017.

[40] T. Geng, T. Wang, A. Sanaullah, C. Yang, R. Xu, R. Patel, and M. Herbordt. FPDeep: Acceleration and load balancing of CNN training on FPGA clusters. In *Proc. of IEEE 26th International Symposium on Field-Programmable Custom Computing Machines*, pages 81–84. IEEE, 2018.

[41] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In *Proc. of Symposium on Field-programmable Custom Computing Machines*, pages 49–56. IEEE, 2000.

[42] T. Good and M. Benaissa. AES on FPGA from the fastest to the smallest. In *Proc. of International Workshop on Cryptographic Hardware and Embedded Systems*, pages 427–440. Springer, 2005.

[43] Y. Gu and Q. Wu. Maximizing workflow throughput for streaming applications in distributed environments. In *Proc. of 19th International Conference on Computer Communications and Networks*, 2010.

[44] S. Handagala, M. Herbordt, and M. Leeser. OCT: The open cloud FPGA testbed. In *Proc. of 31st International Conference on Field Programmable Logic and Applications (FPL)*, 2021.

[45] S. Harris, R. D. Chamberlain, and C. Gill. OpenCL performance on the Intel Heterogeneous Architecture Research Platform. In *Proc. of High-Performance Extreme Computing Conference (HPEC)*. IEEE, Sept. 2020.

[46] Z. He, D. Korolija, and G. Alonso. Easynet: 100 Gbps network for HLS. In *Proc. of 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pages 197–203. IEEE Computer Society, Sept. 2021.

[47] M. C. Herbordt, J. Model, B. Sukhwani, Y. Gu, and T. VanCourt. Single pass streaming BLAST on FPGAs. *Parallel Computing*, 33(10-11):741–756, 2007.

[48] W.-J. Huang, N. Saxena, and E. J. McCluskey. A reliable LZ data compressor on reconfigurable coprocessors. In *Proc. of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 249–258. IEEE, 2000.

[49] A. Jacob, J. Lancaster, J. Buhler, B. Harris, and R. D. Chamberlain. Mercury BLASTP: Accelerating protein sequence alignment. *ACM Trans. Reconfigurable Technol. Syst.*, 1(2):1–44, June 2008.

[50] Y. Jiang. Network calculus and queueing theory: Two sides of one coin. In *Proc. of 4th Int'l Conf. on Perf. Eval. Methodologies and Tools*. ICST, 2010.

[51] Y. Jiang and Y. Liu. *Stochastic Network Calculus*. Springer, Berlin, 2008.

[52] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proc. of 44th International Symposium on Computer Architecture*, 2017.

[53] L. Kleinrock. *Queueing Systems, Vol. 1: Theory*. Wiley, New York, NY, USA, 1975.

[54] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *Proc. of 6th Workshop on Networking Meets Databases (NetDB)*, 2011.

[55] P. Krishnamurthy, J. Buhler, R. Chamberlain, M. Franklin, K. Gyang, A. Jacob, and J. Lancaster. Biosequence similarity search on the Mercury system. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 49(1):101–121, 2007.

[56] P. Krishnamurthy and R. D. Chamberlain. Analytic performance models for bounded queueing systems. In *Proc. of Workshop on Advances of Parallel and Distributed Computing Models*, Apr. 2008.

[57] J. Lancaster, J. Buhler, and R. D. Chamberlain. Acceleration of ungapped extension in Mercury BLAST. *Journal of Microprocessors and Microsystems*, 33(4):281–289, June 2009.

[58] J. M. Lancaster, E. F. B. Shands, J. D. Buhler, and R. D. Chamberlain. TimeTrial: A low-impact performance profiler for streaming data applications. In *Proc. of IEEE Int'l Conf. on Application-specific Systems, Architectures and Processors*, Sept. 2011.

[59] J. Lant, J. Navaridas, M. Luján, and J. Goodacre. Toward FPGA-based HPC: Advancing interconnect technologies. *IEEE Micro*, 40(1):25–34, 2019.

[60] J.-Y. Le Boudec. Application of network calculus to guaranteed service networks. *IEEE Transactions on Information theory*, 44(3):1087–1096, 1998.

[61] J.-Y. Le Boudec and P. Thiran. *Network Calculus*. Springer, Berlin, 2003.

[62] M. Leeser, S. Handagala, and M. Zink. FPGAs in the cloud. *Computing in Science & Engineering*, 23(6):72–76, 2021.

[63] M. Li, G. Zhu, and Y. Savaria. Delay bound analysis for heterogeneous multicore systems using network calculus. In *Proc. of 13th Conf. on Industrial Electronics and Applications*, pages 1825–1830. IEEE, 2018.

[64] R. Li, K. Liu, X. Cai, M. Zhao, L. K. John, and Z. Jia. Improving CNN performance on FPGA clusters through topology exploration. In *Proc. of 36th ACM Symposium on Applied Computing*, pages 126–134, 2021.

[65] D. Liang and S. K. Tripathi. Performance analysis of long-lived transaction processing systems with rollbacks and aborts. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):802–815, 1996.

[66] X. Liu, H. A. Ounifi, A. Gherbi, Y. Lemieux, and W. Li. A hybrid GPU-FPGA-based computing platform for machine learning. *Procedia Computer Science*, 141:104–111, 2018.

[67] L. Ma, R. D. Chamberlain, J. D. Buhler, and M. A. Franklin. Bloom filter performance on graphics engines. In *Proc. of 40th International Conference on Parallel Processing*, pages 522–531, Sept. 2011.

[68] S. Mahesh. TCP-network-demo. https://github.com/OCT-FPGA/tcp-network-demo/, 2022. Accessed Dec. 2023.

[69] A. Mahram and M. C. Herbordt. NCBI BLASTP on high-performance reconfigurable computing systems. *ACM Trans. Reconfigurable Technol. Syst.*, 7(4):33:1–33:20, Jan. 2015.

[70] J. Malicevic, B. Lepers, and W. Zwaenepoel. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *Proc. of USENIX Annual Technical Conference (ATC)*, pages 631–643. USENIX Association, July 2017.

[71] K. Muriki, K. D. Underwood, and R. Sass. RC-BLAST: towards a portable, cost-effective open source hardware implementation. In *Proc. of 19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[72] R. Nigam, S. Atapattu, S. Thomas, Z. Li, T. Bauer, Y. Ye, A. Koti, A. Sampson, and Z. Zhang. Predictable accelerator design with time-sensitive affine types. In *Proc. of 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 393–407. ACM, 2020.

[73] S. Padmanabhan, Y. Chen, and R. D. Chamberlain. Optimal design-space exploration of streaming applications. In *Proc. of IEEE Int'l Conf. on Application-specific Systems, Architectures and Processors*, pages 227–230. IEEE, Sept. 2011.

[74] F. PalunčIć, A. S. Alfa, B. T. Maharaj, and H. M. Tsimba. Queueing models for cognitive radio networks: A survey. *IEEE Access*, 6:50801–50823, 2018.

[75] K. Pandit, J. Schmittt, and R. Steinmetz. Network calculus meets queueing theory - a simulation based approach to bounded queues. In *Proc. of 12th Int'l Workshop on Quality of Service*, pages 114–120. IEEE, 2004.

[76] H. G. Perros and T. Altiok. Approximate analysis of open networks of queues with blocking: Tandem configurations. *IEEE Transactions on Software Engineering*, SE-12(3):450–461, 1986.

[77] T. Plano and J. Buhler. Scheduling irregular dataflow pipelines on SIMD architectures. In *Proc. of 6th Wkshp. on Programming Models for SIMD/Vector Processing*, pages 1:1–1:9, Feb. 2020.

[78] T. Plano and J. Buhler. Enabling real-time irregular data-flow pipelines on SIMD devices. In *Proc. of 50th International Conference on Parallel Processing Workshops*, pages 9:1–9:8, Aug. 2021.

[79] A. Pourhabibi, S. Gupta, H. Kassir, M. Sutherland, Z. Tian, M. P. Drumond, B. Falsafi, and C. Koch. Optimus Prime: Accelerating data transformation in servers. In *Proc. of 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1203–1216, 2020.

[80] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner. Survey and benchmarking of machine learning accelerators. In *Proc. of IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019.

[81] S. Rigler, W. Bishop, and A. Kennings. FPGA-based lossless data compression using Huffman and LZ77 algorithms. In *Proc. of Canadian Conference on Electrical and Computer Engineering*, pages 1235–1238. IEEE, 2007.

[82] A. E. Salama, A. H. Khalil, et al. Design and implementation of FPGA-based systolic array for LZ data compression. In *Proc. of IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 3691–3695. IEEE, 2007.

[83] A. Sanaullah, R. Patel, and M. Herbordt. An empirically guided optimization framework for FPGA OpenCL. In *Proc. of International Conference on Field-Programmable Technology (FPT)*, pages 46–53. IEEE, 2018.

[84] J. B. Schmitt and U. Roedig. Sensor network calculus–a framework for worst case analysis. In *Proc. of Int'l Conf. on Distributed Computing in Sensor Systems*, pages 141–154. Springer, 2005.

[85] A. Shahid and M. Mushtaq. A survey comparing specialized hardware and evolution in TPUs for neural networks. In *Proc. of IEEE 23rd International Multitopic Conference (INMIC)*. IEEE, 2020.

[86] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh. From high-level deep neural models to FPGAs. In *Proc. of 49th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016.

[87] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556*, 2014.

[88] SimPy Team. SimPy: Discrete event simulation for Python. https://simpy.readthedocs.io, 2023. Accessed Aug. 2023.

[89] A. Sohrabizadeh, C. H. Yu, M. Gao, and J. Cong. AutoDSE: Enabling software programmers to design efficient FPGA accelerators. *ACM Transactions on Design Automation of Electronic Systems*, 27(4):32:1–32:27, 2022.

[90] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[91] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proc. of Int'l Symp. on Circuits and Systems*, volume 4, pages 101–104. IEEE, 2000.

[92] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. of International Conference on Compiler Construction*, pages 179–196, Apr. 2002.

[93] J. Thomas, P. Hanrahan, and M. Zaharia. Fleet: A framework for massively parallel streaming on FPGAs. In *Proc. of 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 639–651, 2020.

[94] S. Timcheck and J. Buhler. Reducing queuing impact in irregular data streaming applications. In *Proc. of 10th Workshop on Irregular Applications: Architectures and Algorithms*, pages 22–30, Nov. 2020.

[95] R. Tolosana-Calasanz, J. Diaz-Montes, O. F. Rana, and M. Parashar. Feedback-control & queueing theory-based resource management for streaming applications. *IEEE Transactions on Parallel and Distributed Systems*, 28(4):1061–1075, 2016.

[96] A. Van Bemten and W. Kellerer. Network calculus: A comprehensive guide. Technical Report 201603, Technical Univ. of Munich, 2016.

[97] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing modular hardware accelerators in C with ROCCC 2.0. In *Proc. of 18th IEEE International Symposium on Field-programmable Custom Computing Machines*, pages 127–134. IEEE, 2010.

[98] P. D. Vouzis and N. V. Sahinidis. GPU-BLAST: using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, 27(2):182–188, 2011.

[99] Z. Wang, J. Zhang, and T. Huang. Determining delay bounds for a chain of virtual network functions using network calculus. *IEEE Communications Letters*, 25(8):2550–2553, 2021.

[100] H. Ye, H. Jun, H. Jeong, S. Neuendorffer, and D. Chen. ScaleHLS: a scalable high-level synthesis framework with multi-level transformations and optimizations. In *Proc. of 59th ACM/IEEE Design Automation Conference*, pages 1355–1358, 2022.

[101] J. Zambreno, D. Nguyen, and A. Choudhary. Exploring area/delay tradeoffs in an AES FPGA implementation. In *Proc. of International Conference on Field Programmable Logic and Applications (FPL)*, pages 575–585. Springer, 2004.

[102] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proc. of ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170, 2015.

[103] J. Zhang, H. Wang, H. Lin, and W. Feng. cuBLASTP: Fine-grained parallelization of protein sequence search on a GPU. In *Proc. of IEEE 28th International Parallel and Distributed Processing Symposium*, pages 251–260, 2014.

[104] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong. AutoPilot: A platform-based ESL synthesis system. In P. Coussy and A. Morawiec, editors, *High-Level Synthesis: From Algorithm to Digital Circuit*, pages 99–112. Springer, 2008.