Platform Agnostic Streaming Data Application Performance Models

Clayton J. Faber
Tom Plano
Samatha Kodali
Zhili Xiao
Abhishek Dwaraki
Jeremy D. Buhler
Roger D. Chamberlain
Anthony M. Cabrera

Clayton J. Faber, Tom Plano, Samatha Kodali, Zhili Xiao, Abhishek Dwaraki, Jeremy D. Buhler, Roger D. Chamberlain, and Anthony M. Cabrera, “Platform Agnostic Streaming Data Application Performance Models,” in *Proc. of IEEE/ACM Workshop on Redefining Scalability for Diversely Heterogeneous Architectures* (RSDHA), November 2021.
DOI: 10.1109/RSDHA54838.2021.00008

McKelvey School of Engineering
Washington University in St. Louis

Architectures and Performance Group
Oak Ridge National Laboratory

Platform Agnostic Streaming Data Application Performance Models

Clayton J. Faber
Tom Plano
Samatha Kodali
Zhili Xiao
Abhishek Dwaraki
Jeremy D. Buhler
Roger D. Chamberlain
Dept. of Computer Science and Engineering
Washington Univ. in St. Louis
St. Louis, MO, USA
{cfaber,planot,kodalis,xiaozhili,adwaraki,jbuhler,roger}@wustl.edu

Anthony M. Cabrera
Architectures and Performance Group
Oak Ridge National Laboratory
Oak Ridge, TN, USA
cabreraam@ornl.gov

***Abstract*—The mapping of computational needs onto execution resources is, by and large, a manual task, and users are frequently guided simply by intuition and past experiences. We present a queueing theory based performance model for streaming data applications that takes steps towards a better understanding of resource mapping decisions, thereby assisting application developers to make good mapping choices. The performance model (and associated cost model) are agnostic to the specific properties of the compute resource and application, simply characterizing them by their achievable data throughput. We illustrate the model with a pair of applications, one chosen from the field of computational biology and the second is a classic machine learning problem.**

I. Introduction

Over the last few decades data volume has exploded at an enormous rate, and in many cases the availability of the data and compute resources to process the data are physically separated. This necessitates data movement, in effect data streaming, which can be expensive in its own right. These kinds of applications come in a variety of different implementation flavors targeting a wide range of compute systems, from heterogeneous systems that include computational accelerators to massive server clusters and everything in between (i.e., exploiting both vertical and horizontal scaling). Recently, we have seen the inclusion of edge computing in the mix, where live IoT sensor data has pre-processing performed at the edge and subsequent processing performed on servers in the cloud.

The computational resources available are often quite diverse. On the edge, power is often quite limited, so the compute capability can be small. With the advent of computational accelerators (e.g., GPUs, FPGAs, TPUs, etc.), different portions of the application can take advantage of non-traditional architectures. However, this adds the additional complexity of deciding what tasks get assigned to what compute engines.

One often overlooked step in the execution of these applications is the amount of compute time spent on data pre-processing, both in terms of data transformation and in the movement of data from its origin to the primary computation location. As noted by Malicevic et al. [1] there can often be a disconnect between the primary algorithm running time and the overall execution time. Even with improvements to the overall algorithm, the pre-processing time can completely dominate, effectively squandering any algorithmic improvements. Malicevic et al. specifically point to graph algorithms where pre-processing is done upfront; however, in a streaming application data arrives as it is available to different compute nodes, which might include both forms of horizontal and vertical scaling. One can readily imagine a scenario where in the middle of an execution of a streaming algorithm the application is starved for data, resulting in wasted resources.

In this work we present a model that seeks to predict the end-to-end performance of a given data streaming application, especially when the data resides apart from the compute node(s). 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. Using a pair of example applications, we will demonstrate how a streaming data application can be accelerated using a combination of FPGAs, GPUs, and solutions for networking data between computational resources. Along with this we will discuss potential future work using the model as a guiding hand for research avenues moving forward.

II. Background and Related Work

A. Streaming Data Applications

Streaming data applications have been a target of study for a considerable time, well over twenty years [2]. Examples of development platforms for the streaming paradigm include Auto-Pipe [3], Brook [4], Raftlib [5], StreamIt [6], and Streams-

C [7]. In addition, each of these development platforms supports (or has been extended to support) computational accelerators, either FPGAs or GPUs.

Figure 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 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 are not being shared with other users in the cloud or other applications. 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.

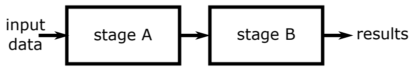

Fig. 1. Example streaming application.

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). By modeling each node as a queueing station (with ingest rate λ and service rate μ), the resulting queueing network is shown in Figure 3 [8]. Prior works by Choi et al. [9] and Gu and Wu [10] 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.

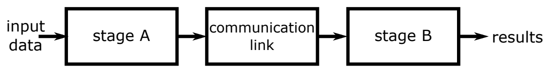

Fig. 2. Example streaming application flow model.

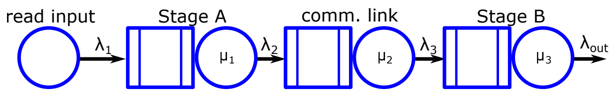

Fig. 3. Example streaming application queueing network model.

In what follows, we will apply the approach above to a pair of streaming data applications. Next, we describe the two streaming data applications, both of which leverage data integration tasks drawn from the Data Integration Benchmark Suite (DIBS) [11].

1) BLAST: BLAST [12], [13], the Basic Local Alignment Search Tool, 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.

The stages of our BLASTN implementation mirror the stages of the NCBI BLASTN computation pipeline, shown in Figure 4, and is built using the Mercator framework on a GPU [14]. 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 [15]. 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 [13], but for BLASTN, that stage takes negligible time compared to the rest of the pipeline [16] and would be implemented on the host processor.

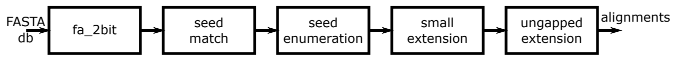

Fig. 4. BLAST application.

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 [17].

2) ML: The Optidigits library is a representation of handwriting data available through the UC Irvine Machine Learning Repository [18]. 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 preprocessing 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 and is practically a solved problem. We utilize off-the-shelf components and models. 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. Section IV-B provides more details about the model.

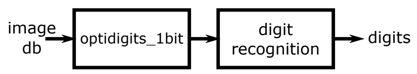

Fig. 5. ML application – handwriting recognition.

B. Related Work

DIBS defines the domain and characteristics of data integration: multi-discipline, low data reuse, and memory dependent. Although DIBS only analyzed these apps using a single-threaded approach, work done in [15], [19], [20] make attempts to accelerate these applications. In these works the Intel Hardware Accelerator Research Program (HARP) system was utilized which consists of an Intel Xeon CPU with an on package Intel FPGA. This allows the FPGA to access the same memory space as the host system and on top of that is last-level cache coherent. While we find this system incredibly interesting and a worthwhile piece of hardware for the future of heterogeneous compute, support by the manufacture has waned and has left the project in an indeterminate state. The tools have not been upgraded since 2016.

A number of groups have utilized accelerators for various data integration problems. Fang et al. [21] utilize FPGAs as part of an enterprise ETL operation. Aggarwal [22] explores the use of GPUs for a similar set of tasks.

A representative subset of previous implementations of all or portions of BLAST on accelerators include CAAD BLAST [23], Mercury BLAST [16], [24], [25], RC-BLAST [26], and TreeBLAST [27] on FPGAs and cuBLASTP [28], GPU-BLAST [29], and Mercury BLAST [17], [30] on GPUs.

Machine learning has long benefited from acceleration. The TensorFlow framework [31] regularly utilizes GPUs. Zhang et al. [32] describe a general approach to deploying machine learning applications using convolutional neural networks on FPGAs. Geng et al. [33] use a cluster of FPGAs for ML training, and Li et al. [34] investigate how to partition inference on an FPGA cluster. Liu et al. [35] 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 [36] review multiple generations of TPUs on ML problems, comparing them to GPUs and FPGAs, and Reuther et al. [37] survey a wide range of machine learning accelerators.

A recent review describes applications that exploit more than one accelerator [38].

III. Model Description

We adopt the approach of Padmanahban et al. [39], Beard and Chamberlain [8], and Timcheck and Buhler [40] to develop an analytic queueing model of each application, beginning with the BLAST application. Starting from the conceptual diagram of Figure 4, 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. The addition of these blocks transforms the diagram of Figure 4 into the expanded diagram of Figure 6.

In Figure 6, 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.

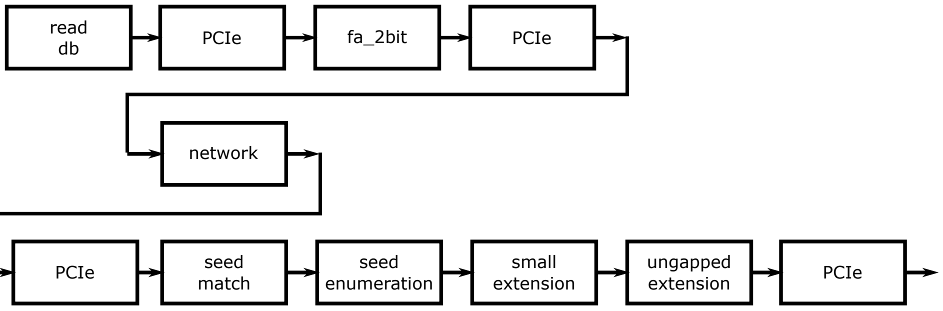

Fig. 6. Flow graph for BLAST application.

We can directly transform the diagram of Figure 6 into a queueing network by replacing each block (or node) of Figure 6 with a queueing station, resulting in the queueing network of Figure 7. 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 6 and the corresponding queueing station of Figure 7 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,

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

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

$$\Gamma_i = \prod_{k=1}^{i-1} \gamma_k, \qquad i > 1, \tag{2}$$

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

$$\lambda_i = \Gamma_i \lambda_1, \qquad i > 1. \tag{3}$$

The above description assumes a one-to-one transformation of blocks in Figure 6 to queueing stations in Figure 7. 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 8. It is this model that we will exploit for the results that are presented below.

In a similar way, our handwriting recognition application is transformed from the initial diagram shown in Figure 5 to the flow graph of Figure 9, which makes explicit reference to the PCIe bus to and from the FPGA and the PCIe bus to and from the GPU. Figure 9 is then transformed in a straightforward way into the queueing network model of Figure 10.

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 [41], both of which are often (but not always) true in these cases.

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 emprirically 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 [8]. An example of a topology of this type is illustrated in Figure 11, in which the transformed FASTA data is delivered to multiple GPU instances for parallel execution. The queueing network for this configuration is illustrated in Figure 12. For the tandem topology of Figure 8, the slowest node in the graph will establish the effective throughput limit.

Denoting the overall throughput by the ingest rate at the first node, we have

$$Tput = \lambda_1. \tag{4}$$

In addition to the performance achieved, we are also interested in the cost effectiveness of the deployment. Since we are using the AWS cloud for our empirical measurements with BLAST, this is straightforward to assess, as all of the costs are explicitly known.

If c_r is the cost per unit time for resource r, the total cost is just the sum of utilized resources.

$$C = \sum_{r \in R} c_r \tag{5}$$

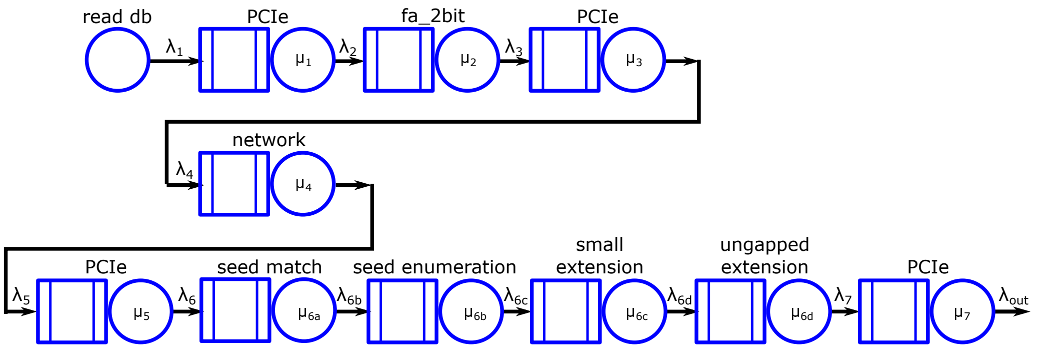

Fig. 7. Queueing network for BLAST application.

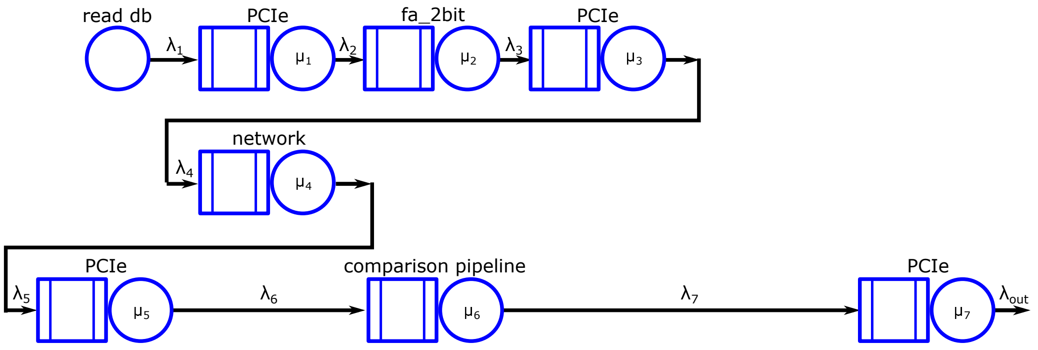

Fig. 8. Modified queueing network for BLAST application.

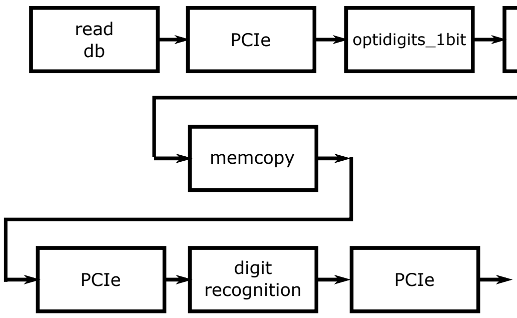

Fig. 9. Flow graph for ML application.

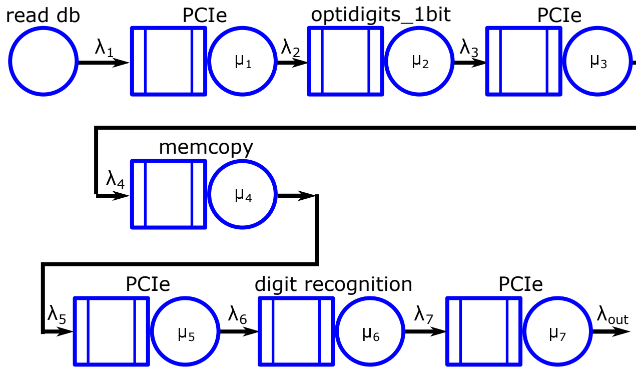

Fig. 10. Queueing network for ML application.

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.

IV. Implementations and Setup

For our implementations of BLAST and ML we target heterogeneous hardware for both the data transformation and final data application. In prior work, we observed that for many data transformations an approach using High Level Synthesis (HLS) targeting FPGA systems has the potential to outperform other forms of parallel programming targeting heterogeneous compute systems [15]. For the second stage, we utilized Mercator for BLAST and Keras and TensorFlow for ML, taking advantage of GPU compute architectures. We execute BLAST in the cloud and ML on dedicated hardware, so as to illustrate the applicability of the model to both. Table I gives parameters of the hardware used in these two scenarios.

A. *BLAST*

The BLAST application is run using hardware from Amazon Web Services (AWS), as it best represents a real world scenario of systems and costs available for immediate purchase in the cloud provider space. The DNA base transformation is run on a f1.xlarge instance which utilizes a Xilinx Virtex

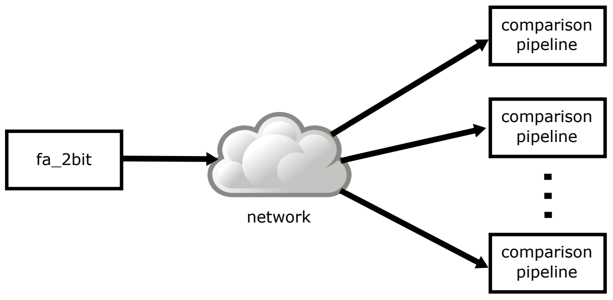

Fig. 11. Multiple comparison pipelines for BLAST.

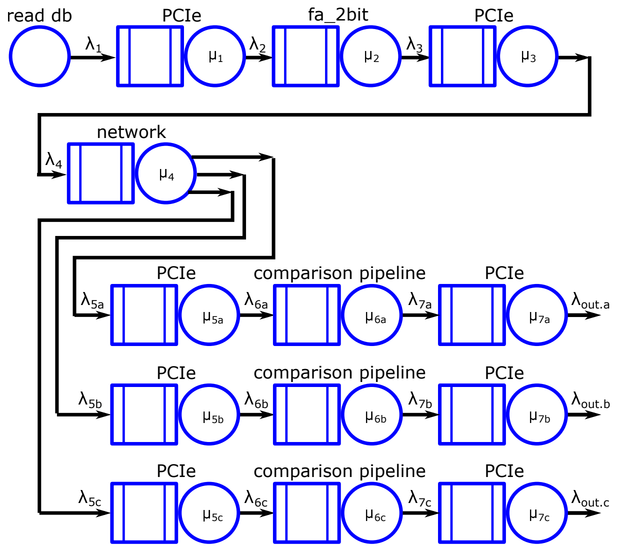

Fig. 12. Queueing network for multiple BLAST comparison pipelines.

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. Block RAM buffers are used to facilitate the movement of the data from kernels designed to access global memory and the actual compute pipeline.

The Mercator system is a framework designed to ease the implementation of irregular multi-stage streaming computations on GPUs. Irregular streaming computations of this nature do not have a simple one to one relationship between the input and output characteristics of a node in the computation pipeline, but rather, may exhibit expansion or filtering behavior on their inputs.

For this work, we use the Mercator system to implement a version of the NCBI-BLAST genomic matching algorithm. As noted in Figure 6, this algorithm is implemented as a four node irregular pipeline on a GPU [17]. The general computation pattern is one where a host CPU packages up a genomic database (or chunk of a database) in a 2 bits per base format along with a query string. Each are copied to the GPU before the Mercator uber-kernel is launched in asynchronous mode. The host is then free to do other work until the Mercator kernel returns with its results.

B. Handwriting Recognition

For the second application, we develop a convolutional neural network (CNN), consisting of multiple layers to recognize the digits in the Optidigits library. This can be considered as a case of transfer learning, since each image in our dataset is evaluated by a model trained on a slightly different dataset.

The model is designed with a VGG-like architecture [42] 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.

C. Network and Intermediate Links

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. In this work we investigate three different candidate utilities to facilitate data movement across a network. In Table IV, μ_4^B refers to three different types of network utilities and their throughput as measured on the AWS EC2 machines listed in Table I 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 [43] 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 $2\times$ the speed of `scp` and more than $1.5\times$ the speed of Kafka.

V. Results

A. Model Parameters

Tables II, III, and IV show the values that are input parameters for the model. 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 [16]), the output data volume is quite small, so the impact on performance is negligible. The reported value is the mean over our experimental runs.

Contrasting this, the service rates shown in Table IV 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.

B. End-to-end Performance Predictions

1) BLAST: The first BLAST execution we will consider is the one represented in Figure 8. Here, the data integration (`fa_2bit`) happens on an AWS FPGA instance and the comparison pipeline happens on an AWS GPU instance.

When just looking at performance, we can ignore Equation (5) and focus on Equation (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 renormalize 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 represent 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

$$\lambda_1 < \hat{\mu_i}, \qquad i > 1. \tag{6}$$

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

$$\lambda_1 < \min_i \hat{\mu_i} \tag{7}$$

and the pipeline stage that determines that value is the bottleneck stage.

Table V shows the normalized service rates for the BLAST implementation of Figure 8 (using the Boost ASIO libraries for networking) and the maximum achievable throughput based on Equation (7). (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.

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 (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. For example, the model assumes that the PCIe transfer is concurrent with the GPU computation, however, in our application there is time spent collecting data before transferring across the PCIe bus. None the less, we are encouraged to see the level of agreement that does exist between the model and the empirical measurement.

From here we can derive the cost of running a streaming computation in this way. In Table I the costs of using an on demand EC2 instance for both the `g4dn` machine and the `F1` instance are listed. Utilizing Equation (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.

We are now in a position to consider alternatives and make predictions about their performance. For example, if we use Kafka for the network link instead of the Boost ASIO libraries. In this case, $\widehat{\mu_4^B}$ is 700 MB/s, which does not impact the predicted upper bound for throughput. This implies that one could use either networking approach without having any significant impact on the performance.

The model also points us towards alternatives that could improve the performance. If we transition to the BLAST implementation of Figure 12, the input rate at each of the three downstream GPUs is 1/3 the previous value (e.g., $\lambda_{5a} = 0.33\lambda_4$). This makes $\widehat{\mu_{6a}^B} = 3\widehat{\mu_6^B}$, and the model now shows the network as the limiting term.

2) ML: Similar to the approach we used for BLAST, we can normalize each of the service rates in Figure 10 to the ingest rate. These normalized values are shown in Table VI. 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). Here, replicating the digit recognition stage on multiple instances is clearly going to benefit performance.

TABLE I
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
WU	8× AMD Ryzen7 3700X @3.6 GHz	32 GiB	Nvidia GeForce RTX 2080Ti with 11 GiB	N/A

TABLE II
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}

TABLE III
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}

VI. CONCLUSIONS AND FUTURE WORK

A. Conclusions

In this work we present a hardware agnostic model that can be used to better estimate how resources are allocated in streaming applications, especially when data resides apart from compute nodes. The model is flexible to meet a variety of situations where data needs to be moved and computed in a variety of different scenarios utilizing both horizontal scaling with specialized compute or vertical scaling with multi-node systems. In our tests, we looked to utilize off the shelf hardware and algorithms in an attempt to replicate real-world scenarios that one may encounter when tackling problems of where and how to compute data. With this model we are able to predict what is the theoretical upper bound on total throughput and a prediction of what would potentially be pain points and opportunities for improvement for both BLAST and a classic machine learning application. The model relies on empirical measurement of the throughput of individual components, rather than any specifics of the execution engine's architecture, which makes it particularly suitable for use with computational accelerators of any form.

B. Future Work

Here, we illustrated the use of the model on a pair of applications, one using rented equipment and the other using owned equipment. Clearly, this should be expanded to additional applications and equipment configurations (e.g., embedded and edge systems).

One example is the deployment of a robust feedback mechanism for a catoptric (mirror) surface [44], [45]. Over 600 mirrors are used to redirect sunlight into interior spaces, increasing the use of natural light for illumination. Each of these mirrors are deployed under pan-tilt control, and without proper feedback, if too many of them point sunlight at the same target, it can overheat. We are designing a computational pipeline that acquires images of the mirrors on a Raspberry Pi edge device, uses ML techniques to ascertain the actual position of each mirror, and uses that information to re-target mirrors as appropriate to maintain safety. Models such as those described here can be quite beneficial in helping assess what computational resources need to be included in the solution, as well as how should the image data be moved from its source to wherever the ML computation is to be performed.

Furthermore, there are a wide variety of technologies and even software that can be explored beyond just simple host to device PCIe memory transfers and socket programming.

TABLE IV
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 (Note 1)
`fa_2bit` (CPU)	μ_2^B	23.4 MB/s (Note 2)
`optidigits_1bit` (FPGA)	μ_2^M	250 MB/s
`optidigits_1bit` (CPU)	μ_2^B	133 MB/s (Note 2)
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 3)

Notes: (1) from [15], (2) from [11], (3) from [16].

TABLE V
BLAST FIGURE 8 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

One popular way to harness the strengths of FPGA compute is to utilize cards and/or chips that have DMA buses or network interfaces that can pull and interpret data on the fly, none of which are in the systems we tested on. New Nvida GPUs also have a wide range of data transfer options ranging from GPUDirect RDMA allowing transfers between device cards, NVLink connections for incredibly fast transfers between equipped GPUs, and Nvidia BlueField DPU cards which combine both compute and networking capabilities. As previously mentioned, the HARP system certainly provides a opportunity to utilize cache coherent memory with a FPGA and clearly warrants further study as most standard market cards make use of I/O bus communications (e.g., PCIe bus). From a software approach, it is often a question of what, if any, trade-offs will there be between performance and programmability.

Further refinement of the model is needed. In these measurements we did not address the variability of the network in the cloud. Although these are dedicated resources, the data center could have a highly congested network depending on the time of day. Further work would seek to measure this variability and make use of it in the model. The dedicated resource model can also be expanded into measurement of the variability of shared resources which is currently not accounted for. The model could also be expanded if one wanted to allocate streams and potentially resources for separate applications that

TABLE VI
ML FIGURE 10 MODELED PERFORMANCE.

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

eventually coalesced at one node as some final computation point. Further refinement could be included with how the computation models are handled for each node, for example, if the device and host share the same memory space versus a traditional dedicated device memory. Supporting shared compute resources is also of interest.

ACKNOWLEDGMENT

This research was supported in part by the following sources: National Science Foundation (NSF) under grant CNS-1763503, Defense Advanced Research Projects Agency (DARPA) Microsystems Technology Office (MTO) Domain-Specific System-on-Chip Program, and the US Department of Energy (DOE) Advanced Scientific Computing Research (ASCR) program.

REFERENCES

[1] 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)*. USENIX Association, Jul. 2017, pp. 631–643.

[2] R. Stephens, "A survey of stream processing," *Acta Informatica*, vol. 34, no. 7, pp. 491–541, 1997.

[3] 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*, vol. 43, no. 3, pp. 42–49, Mar. 2010.

[4] 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*, vol. 23, no. 3, pp. 777–786, Aug. 2004.

[5] 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*, vol. 31, no. 5, pp. 391–404, 2017.

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

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

[8] J. C. Beard and R. D. Chamberlain, "Analysis of a simple approach to modeling performance for streaming data applications," in *Proc. of IEEE Int'l Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Aug. 2013, pp. 345–349.

[9] 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*, 2012, p. 163–172.

[10] 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.

[11] 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*, Apr. 2018, pp. 25–28.

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

[13] 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*, vol. 25, pp. 3389–402, 1997.

[14] 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*, Jul. 2017, pp. 727–736.

[15] 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.

[16] 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*, vol. 49, no. 1, pp. 101–121, 2007.

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

[18] E. Alpaydin and C. Kaynak, "Optical Recognition of Handwritten Digits," UCI Machine Learning Repository, 1998. [Online]. Available: http://archive.ics.uci.edu/ml

[19] A. M. Cabrera and R. D. Chamberlain, "Designing domain specific computing systems," in *Proc. of IEEE 28th Int'l Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2020.

[20] ——, "Design and performance evaluation of optimizations for OpenCL FPGA kernels," in *Proc. of IEEE High-Performance Extreme Computing Conference (HPEC)*, Sep. 2020.

[21] 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*. IEEE, 2017, pp. 55–68.

[22] 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*, vol. 5, no. 6, pp. 333–337, Jun. 2017.

[23] A. Mahram and M. C. Herbordt, "NCBI BLASTP on high-performance reconfigurable computing systems," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 4, pp. 33:1–33:20, Jan. 2015.

[24] A. Jacob, J. Lancaster, J. Buhler, B. Harris, and R. D. Chamberlain, "Mercury BLASTP: Accelerating protein sequence alignment," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 1, no. 2, pp. 1–44, Jun. 2008.

[25] J. Lancaster, J. Buhler, and R. D. Chamberlain, "Acceleration of ungapped extension in Mercury BLAST," *Journal of Microprocessors and Microsystems*, vol. 33, no. 4, pp. 281–289, Jun. 2009.

[26] 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.

[27] M. C. Herbordt, J. Model, B. Sukhwani, Y. Gu, and T. VanCourt, "Single pass streaming BLAST on FPGAs," *Parallel Computing*, vol. 33, no. 10-11, pp. 741–756, 2007.

[28] 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*, 2014, pp. 251–260.

[29] P. D. Vouzis and N. V. Sahinidis, "GPU-BLAST: using graphics processors to accelerate protein sequence alignment," *Bioinformatics*, vol. 27, no. 2, pp. 182–188, 2011.

[30] 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*, Sep. 2011, pp. 522–531.

[31] 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.

[32] 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*, 2015, pp. 161–170.

[33] 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*. IEEE, 2018, pp. 81–84.

[34] 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*, 2021, pp. 126–134.

[35] 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*, vol. 141, pp. 104–111, 2018.

[36] 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.

[37] 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.

[38] R. D. Chamberlain, "Architecturally truly diverse systems: A review," *Future Generation Computer Systems*, vol. 110, pp. 33–44, Sep. 2020.

[39] 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*, Sep. 2011, pp. 227–230.

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

[41] 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*, vol. 22, no. 2, pp. 248–260, 1975.

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

[43] 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.

[44] C. Ahrens, R. D. Chamberlain, S. Mitchell, and A. Barnstorff, "Catoptric surface," in *Proc. of 38th Conference of Association for Computer Aided Design in Architecture (ACADIA)*, Oct. 2018, pp. 216–225.

[45] C. Ahrens, R. Chamberlain, S. Mitchell, A. Barnstorff, and J. Gelbard, "Controlling daylight reflectance with cyber-physical systems," in *Proc. of 24th International Conference on Computer-Aided Architectural Design Research in Asia (CAADRIA)*, vol. 1, Apr. 2019, pp. 433–442.