LightSpeed: A Framework to Profile and Evaluate Inference Accelerators at Scale

by

Christian Williams

S.B. Computer Science and Engineering
Massachusetts Institute of Technology (2022)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

© 2023 Christian Williams. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Christian Williams
Department of Electrical Engineering and Computer Science
May 12, 2023

Certified by: Manya Ghobadi
Associate Professor
Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

LightSpeed: A Framework to Profile and Evaluate Inference Accelerators at Scale

by

Christian Williams

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2023, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The massive growth of machine learning-based applications, and the end of Moore's law, created a pressing need to build highly efficient computing platforms from the ground up. Consequently, researchers and practitioners have been developing highly innovative cutting-edge architectures to meet today's exponentially increasing demands for machine learning services.

However, evaluating the performance gains of newly developed machine learning systems at scale is extremely challenging. Existing evaluation platforms are often specialized to a specific hardware target, such as GPUs, making them less amenable to novel designs. Moreover, evaluating the performance of a newly designed system at scale requires careful consideration of workload and traffic patterns.

To address the above challenges, I introduce LightSpeed, a framework to profile and evaluate inference accelerators at scale. LightSpeed is an event-based simulator that enables users to compare the performance of their system to best-in-class accelerators at scale. LightSpeed profiles the computation and communication requirements of real-world deep neural networks through accurate measurements on hardware. It then simulates the service time of inference requests under a variety of accelerators and scheduling algorithms.

Thesis Supervisor: Manya Ghobadi
Title: Associate Professor

Acknowledgments

First and foremost, thank you to Manya Ghobadi, whose sponsorship, guidance, and support were vital to the completion of my work. I would also like to thank Zhizhen Zhong, my post-doctorate advisor, and mentor, who worked closely with me in all aspects of this project, from high-level vision to the finest implementation details. Finally, I would like to acknowledge Homa Esfahanizadeh, for her guidance in simulator development, and Mingran Yang, for her help in both evaluations and simulation.

This thesis was supported in part by ARPA-E ENLITENED PINE DE-AR0000843, DARPA FastNICs 4202290027, NSF SHF-2107244, NSF ASCENT-2023468, NSF CAREER-2144766, NSF PPoSS-2217099, NSF CNS-2211382, and Sloan fellowship FG-2022-18504.

Contents

1 Introduction **13**

2 Background and Related Work **17**

3 LightSpeed Simulator Design **21**

3.1 Motivation 21

3.2 Simulator Design 21

3.2.1 Neural Network Basics 22

3.2.2 Abstractions and High-Level Design 23

3.2.3 Event Queue 25

3.2.4 Calculating Event Start and Completion Times 30

4 Measurements and Profiling **33**

4.1 Motivation 33

4.2 NVIDIA Triton Setup 34

4.2.1 Motivation for Using NVIDIA Inference Triton Server 34

4.2.2 How NVIDIA Triton Works 34

4.3 Google Cloud Setup 35

4.3.1 Google v3 TPU 35

4.3.2 Measurements on Google VM 36

4.4 Profiling Evaluations 37

5 Simulation Evaluations **41**

5.1 Simulated Systems and Workloads 41

5.1.1 Lightning 41
5.1.2 NVIDIA A100 and P4 GPUs 42
5.1.3 NVIDIA A100X DPU 43
5.1.4 Microsoft Brainwave 43
5.1.5 Simulated DNN Models 44
5.2 Inference Serve-Time Simulations 46
5.2.1 Inference Serve Times 46

6 Conclusion 49

List of Figures

3-1 Simple feed-forward neural network [22]. 22
3-2 High-level Request object design 24
3-3 High-level Processor representation 25
3-4 Sample snapshot of the Event Queue 26
3-5 Snapshots of the Event Queue through time 27
3-6 Sample min-heap queue sorted by event start time 28

4-1 End-to-End Latency Measurements on Hardware 38
4-2 End-to-End Latency for Experimental Models on NVIDIA P4 39
4-3 Datapath Latency Versus End-to-End Latency on NVIDIA P4 40

5-1 Simulated inference serve times 47
5-2 Average core utilization across simulations 48

List of Tables

5.1 Hardware parameters used in LightSpeed 44
5.2 DNN models used in LightSpeed 46

Chapter 1

Introduction

New machine-learning models are being developed at a breakneck pace, causing similarly unprecedented growth in data center needs and demands. For example, ChatGPT [28] is hosted on Microsoft Azure and serves over 10 million live inference queries per day [17]. In general, the market for data center accelerators is projected to grow at a compounded rate of nearly 25% over the next 10 years [18], due in large part to the rapid growth of machine learning applications that demand significant computational resources.

Traditionally, this demand is met using GPUs, which benefit from massive parallelization in computation. Among the most popular hardware devices for serving inference requests are GPUs, such as NVIDIA's A100 GPU and P4 GPU. The high computing throughput of host-based accelerators makes them popular platforms for offline deep neural network (DNN) training and inference. But using GPUs to serve online inference traffic requires transferring packets from the NIC across the PCI-e and kernel through several latency bottlenecks [21, 34]. As a result, GPUs often suffer from low utilization [7, 1, 37, 8, 11].

To make strides in this space, many research groups develop application-specific hardware or ASICs (Application-Specific Integrated Circuits). A prime example of this is Google's TPU [10], which does not perform general computation, and instead focuses on performing matrix multiplication at high speeds. The hardware architecture of a TPU varies greatly from a traditional GPU as a result of being

application-specific. Other, more experimental solutions, such as quantum computing [13] and photonic computing [32] for machine learning, feature architectures that are even more unique as compared to existing solutions. Such architectures might feature computing using qubits, light modulators, or other highly distinct computation hardware and architectures.

However, building complex and experimental hardware architectures for machine learning often takes years of research and development. During this time, it is challenging to evaluate the effectiveness of new hardware architecture at scale, since research often spends a significant amount of time in the prototyping phase. Because of this long development cycle, there is a distinct need for simulation tools to accurately project the performance of novel machine learning accelerators at scale, to either anticipate unexpected challenges or obstacles in the development cycle or validate the current research trajectory and time invested in the work.

As a result of the significant hardware diversity between experimental hardware solutions, however, existing simulation tools often do not provide a sufficiently scaled, yet accurate, one-size-fits-all simulation platform for different varieties of hardware accelerators. This requires researchers of such hardware accelerator prototypes to build their own simulators from the ground up, which can be difficult and time-consuming.

Fortunately, among most hardware accelerators, even the most distinct, there exist a set of common attraction elements. The first of these is a processing unit. Processing units are modular components that take computation input (e.g., matrices) and output a result. Beyond this, many processing units are often parallelized, to the constraints of what the hardware is capable of running in parallel. Finally, in the case of simulating inference requests, there are certain inherent properties of multiplication within a neural network that must be followed, particularly the input-output dependency between the different layers of the network.

To this end, I introduce LightSpeed, a generic, all-purpose machine-learning simulation framework that leverages the above commonalities of machine-learning accelerators to profile and evaluate these systems at scale. LightSpeed's simulator

instantiates general machine learning accelerators, with adjustable parameters that can be tailored to the characteristics of a particular hardware. LightSpeed simulates inference requests at scale, allowing the end user to calculate metrics for hundreds of processed inference requests. Moreover, LightSpeed establishes a methodology for running real-world experiments with existing hardware, to serve as a litmus and comparison point to simulated results.

In this thesis, I present the LightSpeed framework, go into depth with respect to the design of the simulator, discuss real-world experiments that were run on existing hardware, and leverage the results of these experiments, including vital metrics measured like datapath latency, to compare an existing experimental machine-learning accelerator prototype to best-in-class hardware in simulation. With this work, I hope to provide a useful toolkit for systems for machine learning researchers. LightSpeed establishes a starting point for these researchers to progress in their development phase with additional confidence and insight.

The rest of this thesis is organized as follows. Chapter 2 discusses related work, with a particular focus on existing DNN workload simulators. Chapter 3 features the simulator design. It overviews how inference accelerators are modeled in the simulation, as well as inference requests, from the inter-layer level down to individual matrix multiplies. It then discusses the queueing mechanism of the simulator, which is an important component of the design. Chapter 4 details the measurement and profiling efforts for the industry accelerators that served as relevant comparison points for the experimental accelerators, including focused efforts with profiling the NVIDIA P4 GPU using NVIDIA Triton [27, 26], which was critical to establishing GPU datapath latency. Finally, Chapter 5 provides the evaluation, which involves the data and results from both the measurements and profiling, as well as the simulations.

Chapter 2

Background and Related Work

Of course, simulation is extremely common in networking, and there exists significant prior work in the space. Cloud computing services like Microsoft Azure and Google Cloud, which host thousands of NVIDIA A100 GPUs, handle millions of DNN inference requests a day for popular models like ChatGPT, DLRM, VGG-19, and more. As a result, there is a significant need to develop simulators to project the performance of accelerators that seek to improve the runtime of popular DNN workloads.

There are a few existing simulators in the space of simulating DNN inferences on hardware accelerators. In general, these simulators are highly capable but do not suffice for our specific purposes.

Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling

Accel-Sim [15], an extension of the original GPGPU-Sim [2], is a detailed simulation model of contemporary NVIDIA GPUs running CUDA and/or OpenCL workloads. Accel-Sim updates GP-GPU sim in that it increases GPGPU-Sim's accuracy and configurability. It also emphasizes a new frontend, to improve ease of use in simulating different architectures as compared to the previous design. However, this simulator does not provide the level of extensibility and flexibility to simulate some experimental hardware outside of GPUs, while LightSpeed is a more general-purpose system for a variety of accelerators, even those in the experimental stages.

STONNE: A Simulation Tool for Neural Networks Engines

STONNE [20] is a modular simulation framework that takes general DNN frameworks, flexibly models an accelerator device and performs end-to-end simulations of DNN inference. There is a special focus on introducing support for sparse models, which many accelerators aim to leverage in their design. However, while this simulator is flexible in its ability to simulate a wide variety of accelerator architectures, it still does not seem to easily support expressly unique datapath designs, such as potential analog computing designs to run inference. Moreover, it does not simulate inference requests at scale, simulating many requests at once or heavy workloads.

Timeloop: A Systematic Approach to DNN Accelerator Evaluation

Timeloop [30] is a DNN accelerator evaluator whose motivation sits close to the basis of this work, which is to create a broadly applicable DNN workload simulator. Timeloop, however, employs its own custom mapper that automatically defines how workloads are deployed and distributed across an accelerator's resources. Timeloop does this in order to provide an optimized comparison across accelerators. However, some hardware is especially fast, and researchers might have concerns that the runtime of this mapspace optimization may not be feasible to run in real-time, especially if the compute of their experimental hardware is so fast that the scheduling is actually a bottleneck. For this reason, I build LightSpeed as a simulator with a simple, constant runtime scheduler using round-robin (which may not be optimized) with the additional flexibility of introducing custom scheduling algorithms if desired.

Sparseloop: An Analytical Approach To Sparse Tensor Accelerator Modeling

Sparseloop [36] is an extension of Timeloop that aims to improve support for accelerators that wish to leverage model sparsity in their acceleration schema specifically. Sparseloop cites a lack of modeling support for sparse tensor accelerators, which makes efficient accelerator design space exploration difficult. While this is a useful

extension of Timeloop, the earlier concerns with respect to fast and feasible runtime scheduling remain.

HSIM-DNN: Hardware Simulator for Computation-, Storage- and Power-Efficient Deep Neural Networks

HSIM-DNN [35] is another DNN training and inference accelerator simulator that aims to provide additional insight for hardware design. HSIM-DNN seeks to provide accurate models for storage and power efficiency as well as computation for DNN workloads. It features a block-circulant matrix-based representation to model DNN weights. However, this work emphasizes a special focus on hardware designs that integrate with FPGAs or ASICs, which is not always the case with some accelerators, which may not use FPGA at all, or where FPGAs are not the key acceleration component of the design.

Neurophox: Photonic Simulation Framework

Neurophox [29] specializes in simulating optical neural networks (ONNs), which are a specific, and unique experimental innovation in machine learning accelerators. It has a specific focus on scalable ONNs, which use reconfigurable nanophotonic processors, which are essentially two-by-two feed-forward networks for matrix multiplication. While this simulator does well in this domain and makes significant strides in photonic simulations, it is not extensible to all varieties of machine-learning accelerators.

PIMulator-NN: An Event-Driven, Cross-Level Simulation Framework for Processing-In-Memory-Based Neural Network Accelerators

PIMulator-NN [38] is another neural network acceleration framework that is similar to this work due to its event-driven approach. It focuses on "processing-in-memory" architectures, where the processor and memory component are integrated, rather than having the processor separate load from main memory. It also provides additional functionality for analog computation. However, PIMulator while applies a special

focus to analog computing and processing-in-memory, it is not generally or widely applicable to any variety of experimental accelerators.

Chapter 3

LightSpeed Simulator Design

3.1 Motivation

An event-based simulator is a type of simulation that models a system as an evolving sequence of events. LightSpeed's goal is to simulate a generic machine learning accelerator (e.g. a GPU, a CPU, or a new hardware) that is processing deep neural network (DNN) inference requests. At the highest level, the notable 'events' in the simulation are inference request arrivals and inference request completions. By accurately simulating the arrival and completion times for all inference requests handled by a processor, LightSpeed calculates request latency and outputs how quickly a given processor specification (such as an NVIDIA A100 GPU) is completing requests.

3.2 Simulator Design

The following section describes the basic architecture of a neural network (i.e., the structure of an inference request). The goal of this section is to enable the reader to better understand what exactly is being simulated, which is necessary to understand the high-level design of the simulator.

3.2.1 Neural Network Basics

A neural network [9] is a variety of machine learning model that is inspired by the human brain. Nodes in the network, called neurons, are connected to one another and pass information from layer to layer. A feed-forward neural network is a simple kind of neural network that processes input data exclusively in the forward direction, from the input layer, through the hidden layers, and to the output layer. Each layer will transform the data until the output layer gives the result of the inference. We may refer to a neural network as a DNN, or a deep neural network, which is simply a very large variety of neural networks, often with many layers and connections. Figure 3-1 shows an example of a simple neural network with this general architecture.

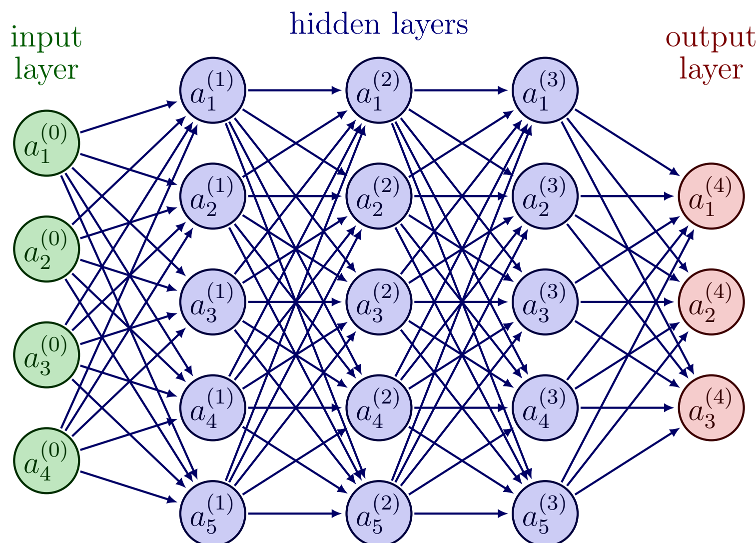

Figure 3-1: Simple feed-forward neural network [22].

If we consider a basic neural network, like one that classifies images, we can think of the input layer can be thought of as the layer that processes that initial image, which is often represented as an array of values from 0 to 255. The earlier "transformation" of data as it is passed through the network is essentially matrix multiplication, particularly in a feed-forward neural network. Each neuron has a vector of "weights," which is essentially just a list of numbers a corresponding vector in the input data will be multiplied by. We can also think of the vectors for each

neuron together as a matrix of weights. After the input vector is multiplied by the weight matrix, the result is passed through an "activation function," a nonlinear transformation of the data [9]. The output of each layer is passed through as an input to the next layer until the output layer produces the final prediction.

In this work, I focus on the simulation of these linear components, or matrix multiplication, during model inference. For large neural networks or DNNs, there can be millions of matrix multiplies in a single inference. It is these matrix multiplies we want to simulate, in order to get an accurate idea of just how much time an accelerator spends on matrix multiplication during inference, and thus its inference latency.

3.2.2 Abstractions and High-Level Design

With any simulator, the goal is to simulate what would happen in reality as closely as possible. Throughout developing the LightSpeed simulator, I continued to add functionality and constraints to achieve this goal. However, making certain abstractions was necessary, especially within the scope of this project. In the following section, I will detail the high-level design of the simulator, as well as address and justify all assumptions made.

As seen in the previous section, most of the computation that occurs in an inference request is matrix multiplication, thereby the bulk of LightSpeed is to simulate matrix multiplication operations. As a result, each inference request is abstracted as essentially a batch of matrix multiplications, which are further subdivided into vector-vector products.

To consider an inference request "completed," all of the constituent vector-vector products of a request must be computed by the processing cores of the architecture that is being simulated. I abstract a processor as a collection of parallel computing cores, each of which is capable of processing one vector-vector product at a time. Aside from cores, I establish multiple other fundamental data structures, described below.

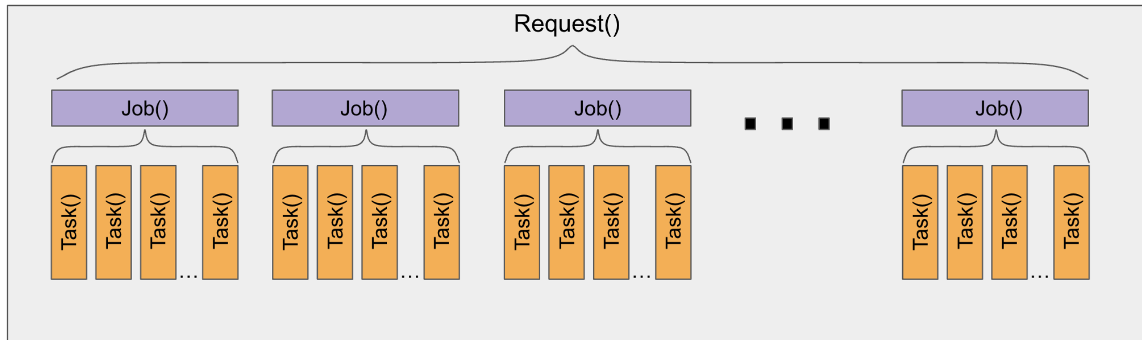

Figure 3-2: High-level Request object design

Data Structures

Figure 3-2 shows the high-level architecture of an inference request in the simulator. I refer to every inference request as a Request, every layer of the request (i.e., a layer of the DNN) as a Job, and every vector-vector product (or VVP, for short) within a DNN layer as a Task.

Why Subdivide into Jobs and Tasks?

The natural question to ask is why to subdivide inference requests at all. I divide DNN requests into Jobs because each layer of a DNN request cannot be computed before the prior layer is finished computing. This is because the outputs of the first layer are also the inputs of the second layer, and so on. We need to ensure that the simulator respects this dependency between DNN layers. Furthermore, some more complex neural networks (not considered in this work) might have more complex dependency structures, which define computation sequence on an intralayer basis. This dependency is captured in a dependency graph, called the directed acyclic graph.

I further divide Jobs into Tasks (vector-vector products) to schedule VVP in the simulator. In reality, some processors and GPUs are capable of doing more complicated subdivisions, but subdividing based on vector-vector products is already highly granular, so I consider this a fair abstraction.

Processors and Cores

To keep the simulator straightforward, every processor is represented by three attributes: its total number of cores, its clock frequency, and its datapath latency. I choose this abstraction because these are the three relevant and analogous components of today's inference accelerators.

Figure 3-3 shows the basic task scheduling procedure. When LIGHTSPEED schedules a Request, it divides it into its individual Tasks, and schedules each Task onto a Core on the Processor, in a round-robin fashion.

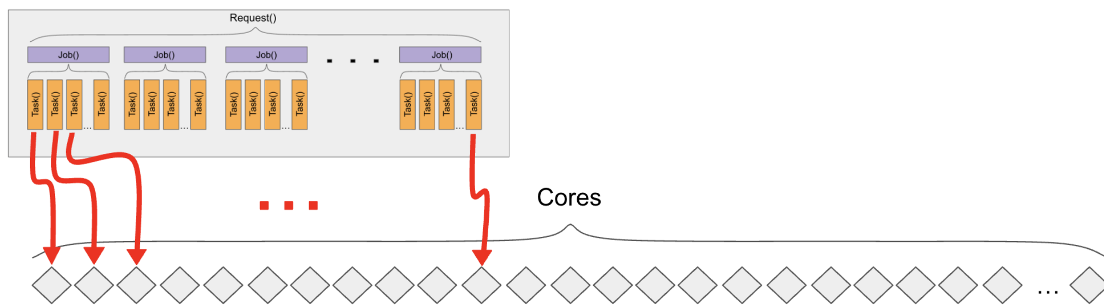

Figure 3-3: High-level Processor representation

Why Event-Based Simulation?

As introduced in the motivation, the simulator is event-based. This means that the process of completing inference requests is represented as a sequence of events, most notably inference request arrival events and inference request completion events (among other relevant events such as job completions, task completions, etc). The main advantage of event-driven simulations is their flexibility to simulate arbitrary events throughout the system.

3.2.3 Event Queue

Naturally, to keep track of arrival and completion events, there needs to be a queueing data structure, where the queue is sorted by the time associated with each event in

the system. The design choices made with this queue are where the bulk of the innovation in the simulator lies (including both the queue architecture and speed and memory optimizations), so I will justify and discuss this design in detail.

Figure 3-4 shows an example snapshot of the simulation EventQueue. The EventQueue is a FIFO (first-in-first-out) queue that keeps track of when all computation starts in the simulation (request start times, task start times, etc). At a high level, it is a Python list of Event objects, which are sorted by the order in which they begin in simulator time.

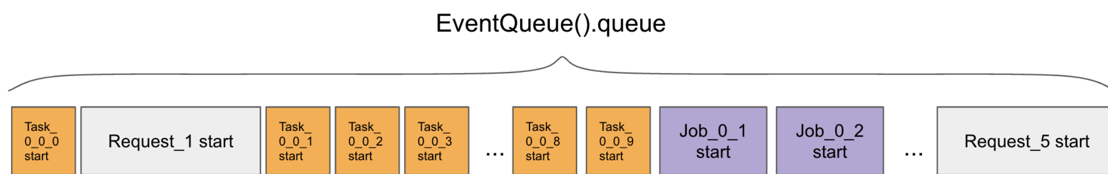

Figure 3-4: Sample snapshot of the Event Queue

The simulator pops the first event of the queue, takes the appropriate action based on the event's type, and proceeds to the next event. This is how the simulator proceeds in time. There are five different possible events in the simulator - RequestStart, RequestFinish, JobStart, JobFinish, and TaskStart. There is no TaskFinish in the simulation because it makes the queue doubly long. Some Requests may consist of tens of millions of Tasks, so this has a significant effect. Moreover, keeping track of TaskFinish events does not produce any especially important metrics for our purposes, so I choose not to represent them.

Populating the Queue

Events are populated in the queue through one of two processes - event *generation* and event *dissolving*. I define event *generation* as the case where an event is simply added to the queue - for example, generating a new inference request (which is represented as a RequestStart object) or generating Finish objects, such as JobFinish or RequestFinish. This is handled by the EventGenerator object, which simply generates

all such Events. Event *dissolving* is the case where an event is replaced by its constituent subevents - for example, when a JobStart event is replaced by the TaskStart events which make up that Job. Figure 3-5 shows an example of how the EventQueue might evolve over time as a result of both event generation and dissolving.

Figure 3-5: Snapshots of the Event Queue through time

In either case, whenever LIGHTSPEED adds event(s) to the queue, it must meet the constraint that the queue is sorted by Event start time. As a result, LIGHTSPEED maintains the Event Queue as a min-heap sorted by start time, which enables us to both quickly pop events off the queue, and quickly re-sort the queue when new events are added during simulation. This architecture becomes especially necessary when handling large requests, which may consist of tens of millions of vector-vector products, and thus tens of millions of Tasks (and tens of millions of Python objects, just for one request). The following section details exactly how popping and merging with this min-heap architecture works.

Min-Heap Queue

A min-heap [3] is a binary tree data structure that maintains the property that each parent node has a value less than or equal to its children nodes. This tree is implemented as an array where each element represents a node in the tree. The root of the tree (i.e. the node with the smallest value) is the first element of the array. Each subsequent element is that element with the next-largest value (which in this case is event start time).

A min-heap maintains two properties - first, the shape property, which is that

each level of the tree is fully filled, except for the last level which may be partially filled from left to right. This is to ensure operations are efficient, and that reaching any node in the tree can be done in the minimum number of iterations. The second property is the heap property, which is the property that each parent node has a value less than or equal to its children nodes, as described earlier. Figure 3-6 shows what a min-heap looks like in the context of the simulation, sorted by event start time.

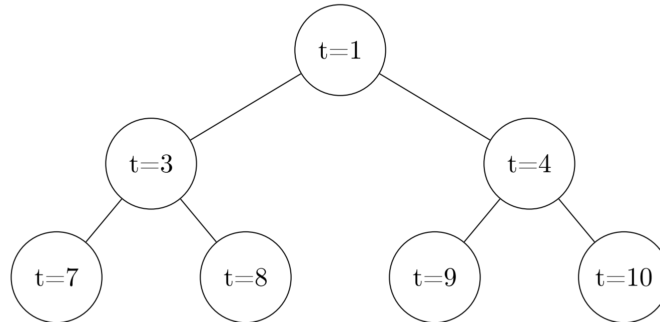

Figure 3-6: Sample min-heap queue sorted by event start time

To remove the smallest element from the heap, or pop the next event of the queue, we first swap it with the last element in the array, which removes it from the tree. Then, we “sift down", where the new root is compared with its children. If the new root has a larger value than one of its children, we swap it with the child that has the smallest value. We continue this process until the new root's value is less than or equal to its children.

To merge a list of events into the queue (i.e. add an event to the queue), LightSpeed first builds another min-heap using the events to be added. Then, it instantiates an empty min-heap and performs a minimum element-wise comparison between the heap of events to add and the heap which is the current event queue, in order to build the new event queue.

This process is efficient for large merges, such as merging a list of Tasks, but less efficient for situations when merging a few or just one event, just as generating a single request. However, since merging Tasks constitutes the vast majority of the overhead for this process, and merging just one event is still relatively fast regardless, this remains the procedure for all such merges into the event queue.

Job Generation Schema

One important characteristic of the simulator is that LIGHTSPEED generates Jobs one at a time after encountering a Request in the queue, rather than all at once (which is technically feasible given the simulator design). We make this design choice for simulator efficiency purposes. In order to generate a Job, it has to be assigned a start time, to add it to the queue - thus, the completion time of the previous Job must be known, since there may be a DAG dependency that defines when the following Job can begin computation. Establishing the completion time of a Job requires generating all of its constituent Tasks, such that the completion time of its final Task can be accurately calculated (details of Event start and completion time calculations are further discussed in Section 3.2.4.

However, in large DNNs, even one Job may consist of millions of Tasks. As a result, maintaining tens of millions of Tasks in memory for an entire Request both slows down the simulation and introduces memory constraints.

Fortunately, LIGHTSPEED does not have to generate all of the Jobs for a given Request at once. Since the computation for a Job cannot begin until the computation for the previous Job completes, LIGHTSPEED delays generating a Job until the completion of the previous Job. This is why one Job at a time is generated, rather than all of them at once, as in Figure 3-5.

Core Queues

Aside from the primary Event Queue, each Core in the simulator also has its own queue, which maintains the order in which the Core handles Tasks (vector-vector products). This is necessary because the completion time of Task t_0 constitutes the start time of Task t_1, where t_0 and t_1 are Tasks scheduled to the same Core (in general, although this can change slightly based on DAG dependencies, which will be discussed in the following section). As a result, individual queues for each core are vital to maintaining information about when Tasks start and complete, and thus when Jobs and Requests start and complete.

3.2.4 Calculating Event Start and Completion Times

In order to establish when an Event begins its computation, LightSpeed precalcluates its parameters based on known values maintained in the simulator state. By design, the start or finish times of any Task, Job, or Request are known at the moment that the Event is generated.

For example, the simulation time at which a Task begins computation is the *maximum* of:

1. The finish time of the previous Task scheduled to the same core.
2. The finish time of the previous Job of the same Request (to obey the DAG).
3. The time at which the Task itself was generated (in case the processor is idle).

This property follows from simply reasoning about the start time of a Task, and what could possibly delay its calculation. In the simplest case, a Task is scheduled to an idle Core and immediately begins calculation, in which case the start time of the Task is equivalent to its generation time (plus any potential datapath latency - I will discuss how datapath latency is addressed in a later section). This is represented by Item 3. If the Core is not idle, and there are no DAG dependencies, then this Task will begin when the previous Task ends, i.e. Item 1. In the case that there are DAG dependencies, for example when the Task could potentially begin computation before the Tasks from the previous Job are completed, this Task will then begin computation at the moment the last Task of the previous Job is completed (Item 2), assuming this is the maximum time of the three scenarios.

In reality, the simulator only generates one Job at a time. As a result, it is actually impossible for a Task to be generated before the completion of the previous Job within the same Request. So while this is still a constraint of Task start times, it is not necessary to consider at runtime due to the design of the simulator.

The completion time of a Task is simply the start time plus the size of the Task. This reveals a couple of important abstractions within the simulator. First, the size of a Task is defined as the number of multiplications that it represents. For example,

if the Task represents a vector-vector product between two size 5 vectors, this would be a Task of size 5. Moreover, one unit of time in the simulator is equivalent to one multiplication. This is a useful baseline because this means the simulation runs at 1GHz (or one multiply per nanosecond) which can then be scaled up or down based on the real clock frequency of the processor being simulated. As a result, the simulator runs on the nanosecond scale.

Calculating the start and completion times of Jobs and Requests is much simpler. The start time of a Job is simply the minimum start time of its constituent Tasks, and the start time of a Request is the start time of its first Job. Conversely, the completion time of a Job is the maximum completion time of its constituent Tasks, and the completion time of a Request is the completion time of its final Job. LightSpeed calculates these values, and then generates JobFinish and RequestFinish Events and adds them to the Event Queue to represent them in the simulation.

Chapter 4

Measurements and Profiling

4.1 Motivation

In order to make the inference request simulations as accurate and close to a real-world setting as possible, it is vital to profile the hardware that is being simulated, to get accurate metrics for characteristics like datapath latency, which are used directly in simulation. Accurately representing the datapath latency is critical because for some experimental hardware, much of the innovation is in the datapath, and therefore the overall end-to-end inference latency. As a result, this must be represented in simulation, to demonstrate whatever gains might be made as a result of improving in this domain.

Moreover, results from measurements and profiling are useful to broadly cross-validate the results of real-world experiments with simulated results. To this end, we have access to NVIDIA P4 GPUs as well as Google's v3 TPU, which are best-in-class for model inference. I set up a series of experiments using a combination of NVIDIA's Triton Inference Server and Google Cloud Computing platform to evaluate each of these processors on popular ML models (i.e. AlexNet, VGG-19, etc.)

4.2 NVIDIA Triton Setup

This section is dedicated to a brief overview of NVIDIA's Triton Inference Server, how it works, and how I leverage it to measure various inference metrics in a real-world setting.

4.2.1 Motivation for Using NVIDIA Inference Triton Server

NVIDIA Triton Inference Server [27] is an open-source inference serving software that helps standardize model deployment and execution and delivers fast and scalable AI in production [27]. For the purpose of this work, I use it to evaluate model performance for one model at a time, sending inference requests from a client machine to a server machine. I use NVIDIA Triton instead of simply running models on the hardware for multiple reasons.

First and foremost, NVIDIA Triton has access to hardware-level code for NVIDIA GPUs. As a result, Triton is able to distinguish between which part of an inference request is network or datapath latency time, and which part is actually compute time on the hardware. This means Triton not only gives precise measurements with respect to inference latency but also gives a number of other useful and informative metrics about the end-to-end latency for an inference request. I am interested in these metrics because most inference today is done in massive data centers, which are handling millions of latency requests every day.

4.2.2 How NVIDIA Triton Works

Using NVIDIA Triton first requires defining and configuring the desired models and inference data. Models are then downloaded within your preferred framework (I use PyTorch) and configured in a Triton-specific format. After choosing a model, that model and its corresponding inference data (for example, a batch of images in the case of a vision model like VGG-19) are then loaded by Triton, which starts serving inference requests.

When a client sends a request, Triton first preprocesses the input data, such as

matrix resizing or normalization. Then Triton passes the preprocessed data to the model for inference. The output of the model is then post-processed, such as applying a softmax or converting to a human-readable format, and then returned to the client.

For these experiments, I configured LeNet [19], AlexNet [16], VGG-11 [12], VGG-16 [4], VGG-19 [14], and two binary neural networks (BNNs) from a networking paper called N3IC [34] on a machine with an NVIDIA P4 GPU.

Triton also enables variable concurrency and batch size for inference requests. In inference, batch size essentially defines how many individual inferences are made within a single inference request. For example, if passing one size 64 vector through a model constitutes one inference, then passing a $128x64$ matrix through that same model would be a batch size of 128. Concurrency, however, defines how many inference requests are sent side-by-side along the networking datapath. For this work, I run all experiments with concurrency 1, and test batch sizes ranging from 1 to 128 for a subset of those models. For all other models, I also run experiments at batch size 1 for simplicity.

4.3 Google Cloud Setup

4.3.1 Google v3 TPU

I run experiments on Google's TPU (Tensor Processing Unit) [10] as an additional benchmark. A Google TPU is a specialized processor designed to accelerate machine learning workloads. It is also specifically optimized for performing large-scale matrix operations.

Some useful specification details about the TPU we use, TPU v3 [10]:

- The TPU v3 has two tensor cores, which are specialized hardware units for matrix multiplication. Tensor cores are highly optimized for inference workloads and compute at 123 teraflops (1.23×10^{14} operations per second).
- 1.6 terabytes per second (TB/s) of memory bandwidth, for quick memory access to large amounts of data.

- TPU v3 uses a high-speed interconnect which connects multiple TPUs together to form a 'TPU pod'. This interconnect has 6.4 terabits per second (TB/s) of bisection bandwidth across a pod, which enables TPUs in the pod to communicate efficiently.

- 32 GB of high-bandwidth memory (HBM), which supports large models that can't fit onto the memory of just one GPU.

Overall, the TPU v3 is a highly powerful machine learning workload accelerator with strong performance metrics, memory capabilities, and an interconnect which make it well-suited for handling inference requests at scale. This makes it both a useful and necessary comparison point for LightSpeed.

NVIDIA T4 on Google VM

Google Cloud also provides quick and convenient to other NVIDIA GPUs, such as NVIDIA T4, which is another one of NVIDIA's GPUs that is specialized for inference. I also took measurements on one of Google's NVIDIA T4 GPUs, to provide a more complete picture for the latency experiments.

4.3.2 Measurements on Google VM

Measuring the latency capabilities of Google's TPU is slightly less straightforward since there is no convenient software like NVIDIA Trition to take complex measurements. For this reason, I constrain the experiments on Google's VM to just inference latency and take measurements simply with Python's time module. Of course, the datapath latency is included in this measurement, but I include the datapath latency in the final measurements for NVIDIA GPUs as well. Moreover, I found that these measurements were consistent with those provided by Google for their hardware, so I consider it a reasonable approximation.

4.4 Profiling Evaluations

The following section provides the results and data from measuring and profiling the NVIDIA GPUs and Google TPU.

The models I choose to deploy for measurements are a collection of vision models (LeNet [19], AlexNet [16], VGG-11 [12], VGG-16 [4], VGG-19 [14]) and two binary neural networks, including a model I refer to as IoT, for network traffic classification, and a model I refer to as Anomaly, a security model for network traffic anomaly detection [34]. I choose vision models because their datasets, which consist of images, are a common inference request input that also sufficiently strains the datapath to get an accurate view of datapath latency. The binary neural networks (BNNs) [34] are small models, which provide a useful comparison point to large vision models like VGG-16 and VGG-19, which furthers the overall goal to get a strong general idea of how GPUs perform on a variety of models, both small and large.

The following plots summarize the results from the experiments that were run on the profiled hardware. Since I focus on GPUs in simulation, I especially focus on Triton experiments with the NVIDIA P4 GPU [26], to get the best possible data to accurately represent GPU datapath latency. Figure 4-1 and Figure 4-2 show the end-to-end latency results for the models I chose to measure. From these measurements, we have a more complete picture of how these accelerators perform on this selection of models in the real world - as a result, we should expect comparable results in simulation. Furthermore, Figure 4-3 separates datapath latency measurements from end-to-end latency - from these results, I found that datapath latency for a GPU (specifically, P4) is around 1549 μs. Therefore, this is the value I use moving forward in simulations.

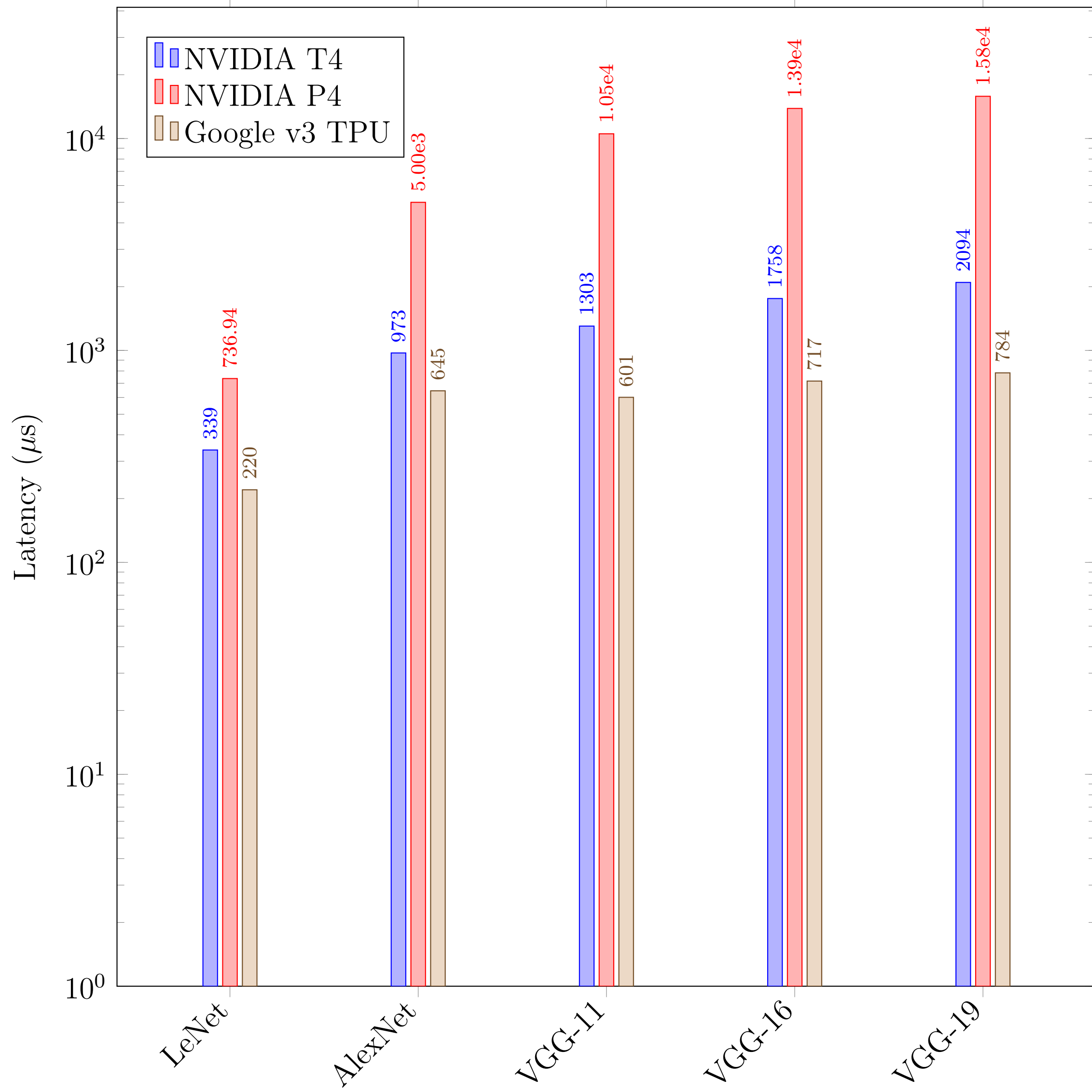

Figure 4-1: End-to-End Latency Measurements on Hardware

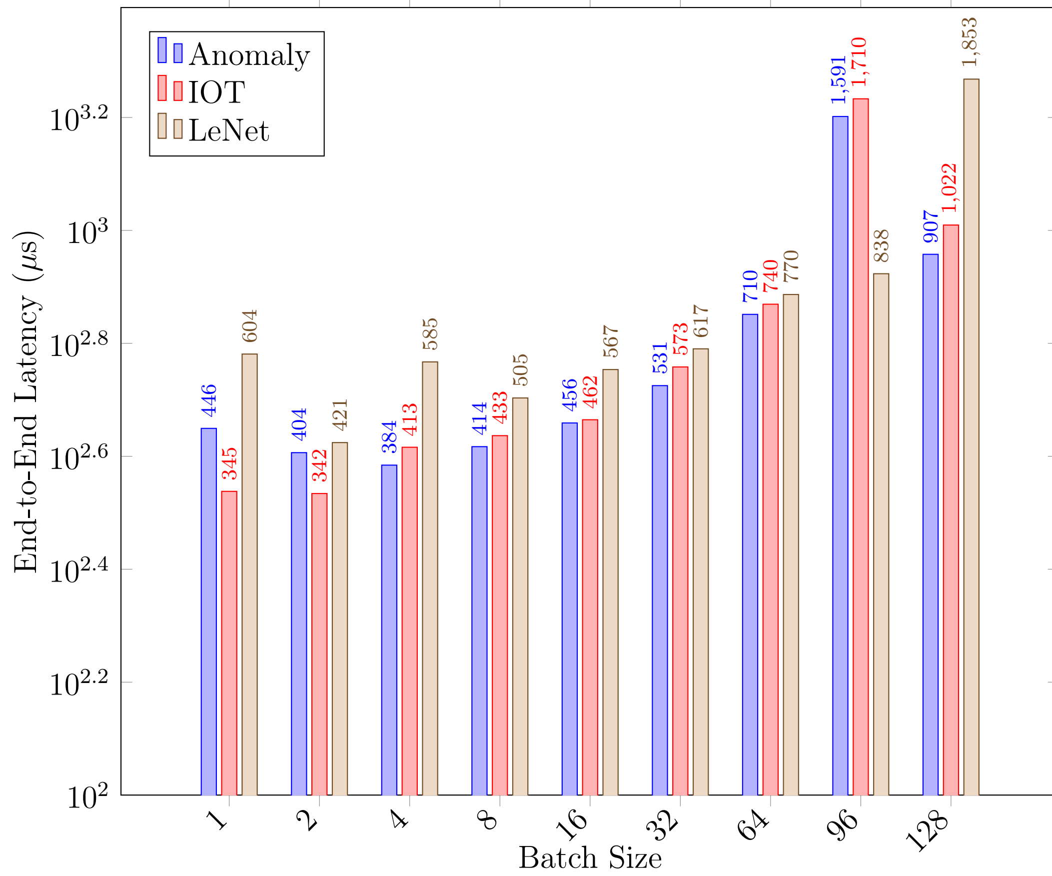

Figure 4-2: End-to-End Latency for Experimental Models on NVIDIA P4

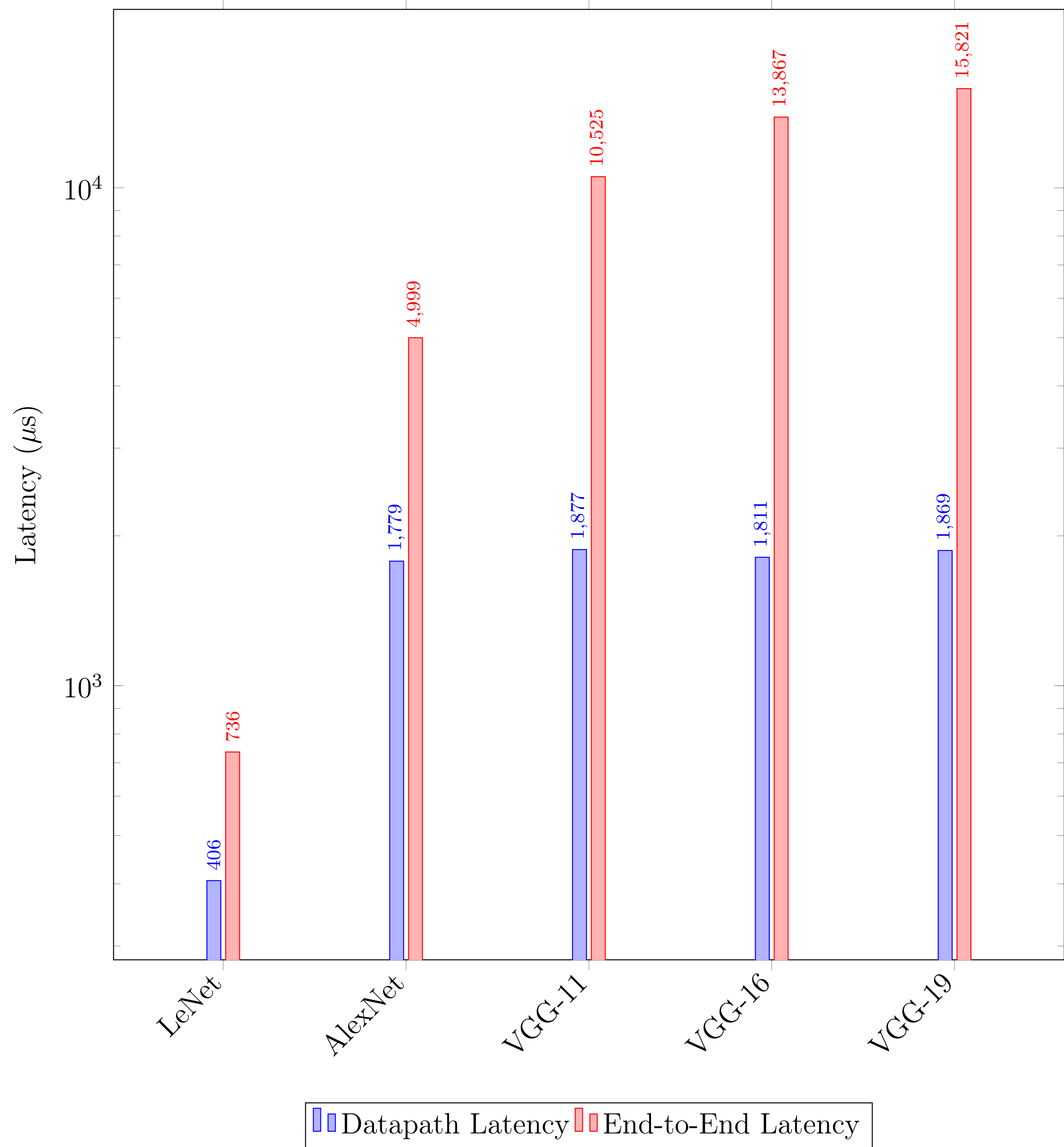

Figure 4-3: Datapath Latency Versus End-to-End Latency on NVIDIA P4

Chapter 5

Simulation Evaluations

To evaluate LightSpeed, I run several simulations comparing state-of-the-art inference accelerators using real-world DNN models. In this section, I begin by detailing each inference accelerator that I use for evaluation. I then discuss their features such as the number of cores, clock frequency, datapath latency, and other relevant characteristics of the hardware. Then, I outline the models which are supported in the simulation. The simulation is highly customizable, and additional models can be supplemented relatively easily. Finally, I discuss simulation results, specifically the average inference latencies across different DNNs and hardware architectures.

5.1 Simulated Systems and Workloads

5.1.1 Lightning

Lightning [39] is a novel machine-learning inference accelerator that leverages photonic computing to perform fast and energy-efficient multiplication in the analog domain. A Lightning system has several hardware components, including optical modulators, digital-to-analog converters, analog-to-digital converters, photodetectors, and a Xilinx FPGA. Lightning's key concept is to feed voltages into the optical modulators, each of which encodes a number from 0 to 255 (i.e., an 8-bit fixed-point number). It then applies the output from the first modulator to that of the second, and the

resultant output is equivalent to the pairwise multiplication of each element. From this signal, Lightning is able to read a multiplication result.

LightSpeed models a Lightning accelerator with 200 cores - this is because we expect a prototype with 1 photonic core and 200 wavelengths (which computes equivalently to 200 cores) to be a realistic goal for later-stage prototypes. Moreover, I simulate Lightning with compute frequency of 100 GHz. The Lightning paper mentions that it has average datapath latency of 344 ns, so this is what is used in simulation [39].

I compare the performance of Lightning to the following real-world hardware. This hardware is the best-in-class for machine learning inference. In the following sections, I will overview each architecture, as well as detail and justify how each architecture is modeled and approximated in the simulator.

5.1.2 NVIDIA A100 and P4 GPUs

NVIDIA A100 GPU

NVIDIA's A100 GPU [23] is a flagship compute platform, and is used in most data centers today for machine learning applications, in both training and inference. It has two specialized core architectures, CUDA cores and Tensor cores. These cores are specialized for math operations and matrix multiplication. CUDA cores, in simple terms, are capable of performing eight multiplications in parallel. Tensor Cores are able to achieve approximately 3× performance across multiplications. The A100 GPU has 6912 CUDA cores and 432 Tensor Cores. It also has a clock frequency of 1.41 GHz, when overclocked [24].

In LightSpeed's simulation for the A100, I represent the architecture using the following parameters:

- The 6912 CUDA cores and 432 Tensor Cores are represented as 6912×8 and 432×3 cores, for 56,592 cores in total.
- Compute is simulated at the 1.41 GHz overclock frequency.

- The measurements gathered in Chapter 4 give an average datapath latency of 1549 μs for GPUs. This value is used to represent the datapath latency in simulation.

NVIDIA P4 GPU

NVIDIA's P4 GPU [26] is their compute platform which is specialized for inference and energy efficiency. Some data centers may feature this GPU if they desire fast, cost-effective, and energy-efficient inference. The P4 GPU features 2560 CUDA cores, and has a clock frequency of 1.114 GHz when overclocked.

In LightSpeed, the 2560 CUDA cores are represented as 2560 $\times$ 8 cores, for 20,480 cores in total, along with the 1.114 GHz clock frequency and GPU datapath latency of 1549 μs.

5.1.3 NVIDIA A100X DPU

The NVIDIA A100X DPU is an extension of the NVIDIA A100 GPU, which connects directly to the network via an integrated PCIe switch. This creates a dedicated datapath between the network and GPU that greatly reduces datapath latency [25]. Simulating this hardware can be desirable for comparison against prototypes that seek to make gains in the datapath latency, as is the case for Lightning.

In LightSpeed, the A100X DPU is represented the same as the A100 GPU in terms of the number of cores and clock frequency. For datapath latency, measurements were not available at the time of writing for the A100X DPU. However, the datapath latency of the DPU is known to be greatly reduced as to be mostly negligible, so to give the hardware the benefit of the doubt in simulation compared to Lightning, the aforementioned clock frequency of 344 ns from Lightning is used.

5.1.4 Microsoft Brainwave

Microsoft's Brainwave [7] is another variety of machine learning inference accelerators. It is an FPGA-based smartNIC that leverages massive parallelism and also

massively reduces datapath latency by sitting on the network datapath. However, since Brainwave leverages FPGAs, which have a relatively slow clock frequency, and doesn't compensate compared to a system like Lightning, it computes at a 600 MHz clock frequency. Brainwave operates with a computational parallelism equivalent to 96,000 cores. Since we also lack datapath latency experimental data as in the case of DPU, I use a similarly generous 344 ns in simulation. The above metrics are the values used in LightSpeed.

Table 5.1 summarizes the above simulation parameters for each evaluation platform.

Platforms	Simulated Cores	Clock Frequency	Datapath overhead (μs)
Nvidia A100X DPU	56,592	1.41 GHz	.344
Microsoft Brainwave	96,000	600 MHz	.344
Nvidia A100 GPU	56,592	1.41 GHz	1549
Nvidia P4 GPU	20,480	1.114 GHz	1549

Table 5.1: Hardware parameters used in LightSpeed

5.1.5 Simulated DNN Models

I evaluate six real-world DNN models: VGG16 [4], VGG19 [14], MegatronBERT [5], GPT-2 [31], DLRM [6], and ChatGPT* [28, 17, 18]. Below are additional details regarding model configurations.

VGG16 [4]. VGG-16 is a 16-layer convolutional neural network used for image classification. The bulk of the multiplication is processed in VGG-16 during the convolutional process, where increasingly smaller kernels are slid across the pixel values of an image, resulting in multiplication and summation. LightSpeed calculates the number of multiplications within each vector-vector product and separates each of these vector-vector products on a layer-by-layer basis (Tasks and Jobs respectively). VGG-16 also has 3 dense layers, which are also accounted for.

VGG19 [14]. VGG-19 is essentially the same as VGG-19, except it features 19 layers of convolution, rather than 16. I, therefore, run similar calculations for VGG-19 and account for the same 3 dense layers.

MegatronBERT [5, 33]. Is a combination of the BERT architecture, which is a transformer-based model for natural language processing, and Megatron, which trains transformers at a massive scale. MegatronBERT is a 24-layer model with 1024 hidden features and 16 attention heads, which are essentially subdivisions of matrix multiplication within the transformer model architecture. LightSpeed calculates the number of multiplications within each vector-vector product, for both attention layers and feed-forward layers. I also account for word embedding layers, which encode natural language into language model embedding tables, at the beginning and end of the network.

DLRM [6]. DLRM (Deep Learning Recommendation Model) is a recommendation model that features an architecture that is highly variable in matrix size on a layer-to-layer basis, specifically in its embedding layers. It is useful as a simulation model not only because it is widely used and features another ML use case, but also because it introduces additional vector size variety and stress-testing to the simulation. I analyze the embedding layers of DLRM, as well as its feed-forward layers, and calculate the size of vector-vector products for each layer.

GPT-2 [31]. GPT-2 is one of the earlier iterations of the now widely popular GPT transformer architecture. It is a generative language model, and similar to BERT, it features attention layers, feed-forward layers, and embedding layers at the beginning and end of the network. The largest version of GPT-2 (which LightSpeed simulates) has 48 layers, with a model dimensionality of 1600 (the common dimension which carries throughout the network).

ChatGPT* [28]. ChatGPT is a much larger version of the GPT architecture, which specializes in generated chat output. The finer details of the model architecture are still unknown at the time of writing. However, ChatGPT's number of parameters is known, with 175 billion parameters. Therefore, to approximate the model in simulation, LightSpeed instantiates a new instance of GPT-2, with an increase in the number of layers commensurate with ChatGPT's parameter count (5750 layers, or nearly 120× the number of layers in GPT-2).

Table 5.2 lists each model, as well as the total number of multiplications performed

by each model end-to-end.

DNN	Total # of multiplications	Type
VGG16	1.54×10^{10}	Vision
VGG19	1.96×10^{10}	Vision
MegatronBERT	3.57×10^{9}	Language
GPT2	1.56×10^{9}	Language
ChatGPT*	1.76×10^{11}	Language
DLRM	3.14×10^{9}	Recommendation

Table 5.2: DNN models used in LightSpeed

Request arrivals. I use a Poisson distribution for inference request arrivals and vary the arrival rate between 10 Gbps and 100 Gbps. All models have an equal probability of occurrence.

5.2 Inference Serve-Time Simulations

For simulations, the primary focus is to compare the simulated serve-time performance of Lightning to state-of-the-art benchmarks when serving large-scale DNN inference queries. In these simulations, around 100 inference requests were served, for a total of around 500 inference requests across accelerators. The final results show how each accelerator performs and its percent utilization across these simulations.

5.2.1 Inference Serve Times

I define the inference serve time as the time it takes to respond to a DNN inference query from the moment it arrives at the accelerator. Figure 5-1 compares the average inference serve time of jobs across different DNNs.

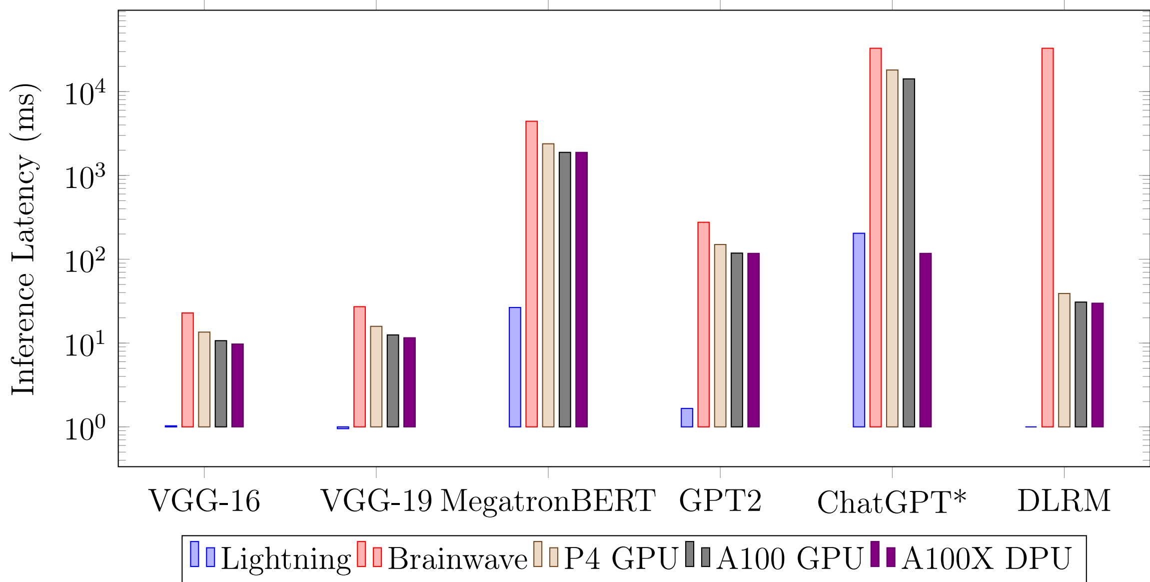

Figure 5-1: Simulated inference serve times

From Figure 5-1, we see that A100X DPU performs the best among the digital hardware platforms (Brainwave, A100 GPU, and P4 GPU) because A100X DPU combines strong compute parallelism with minimal datapath latency to give the best results among existing digital solutions. Although Brainwave has the greatest parallelism, it suffers from low clock frequency because of its FPGA-based implementation, highlighting the importance of clock frequency when handling real-time user-facing inference requests. Finally, the photonic computing system, Lightning, improves the inference serve time compared to all the other digital accelerators because of its high clock frequency and efficient datapath latency.

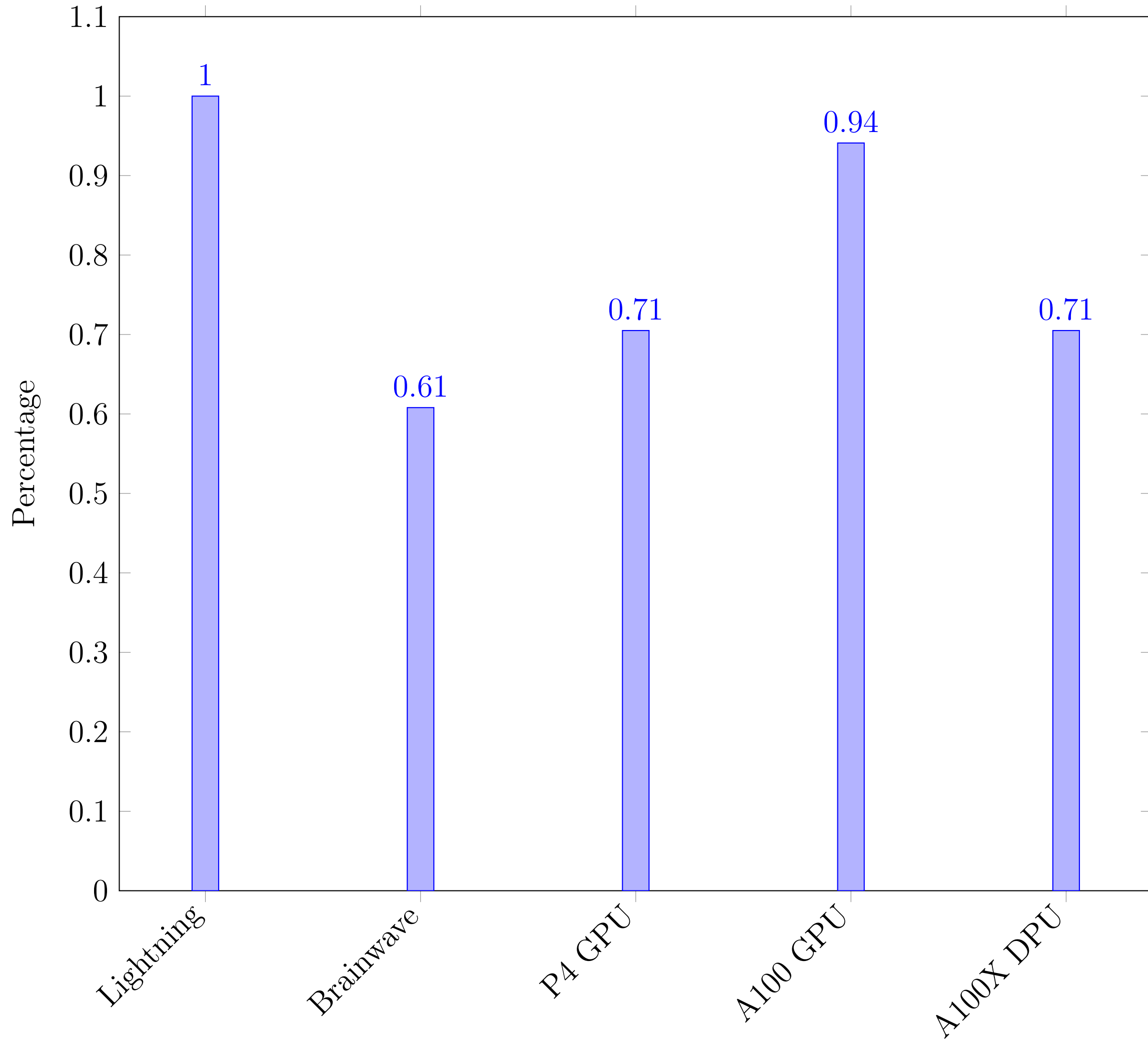

Figure 5-2: Average core utilization across simulations

Core utilization is an important metric in simulation because it not only validates whether simulation results are expected, it also hints at whether an accelerator was used to its full computational capacity, or if more can be done to improve latency times with better scheduling schema. Figure 5-2 shows that while most accelerators are mostly utilized, there is potential for additional utilization for digital accelerators, like Brainwave [7] (even if full utilization is not leveraged in practice). Lightning, however, is fully utilized [39] due to its efficient datapath design.

Chapter 6

Conclusion

LightSpeed is a framework for evaluating and simulating experimental DNN accelerators, inspired by experimental prototypes and hardware. It leverages both simulation and real-world profiling and measurements to provide an accurate and informative evaluation of DNN accelerators. LightSpeed takes into account datapath latency, core parallelism, and clock frequency without being overly demanding or imposing with respect to what can and cannot be simulated. It supports many popular models and architectures, with a straightforward implementation design that enables it to be both flexible and useful. In extensions of this work, the simulation component might take additional steps towards considering model sparsity and a variety of scheduling algorithms, which are currently implemented but were not thoroughly tested and evaluated at the time of writing.

Bibliography

[1] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, Jennifer Hwang, Rebekah Leslie-Hurd, Michael Bye, E.R. Creswick, Matthew Boyd, Mahitha Venigalla, Evan Laforge, Jon Purdy, Purushotham Kamath, Dinesh Maheshwari, Michael Beidler, Geert Rosseel, Omar Ahmad, Gleb Gagarin, Richard Czekalski, Ashay Rane, Sahil Parmar, Jeff Werner, Jim Sproch, Adrian Macias, and Brian Kurtz. Think fast: A tensor streaming processor (tsp) for accelerating deep learning workloads. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 145–158, 2020.

[2] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Gpgpu-sim: a performance analysis framework for execution driven simulation of gpus. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 53–62, 2009.

[3] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT press, 3 edition, 2009.

[4] Abhipraya Kumar Dash. VGG-16 Architecture. `https://iq.opengenus.org/vgg16/`.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[6] Facebook. Deep Learning Recommendation Model for Personalization and Recommendation Systems, 2021. `https://github.com/facebookresearch/dlrm`.

[7] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2018.

[8] Tom Goldstein. How many GPUs does it take to run ChatGPT?, Feb. 2023. https://twitter.com/tomgoldsteincs/status/1600196995389366274?lang=en.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[10] Google. TPU System Architecture, 2023.

[11] Groq. The Challenge of Batch Size 1, 2020. https://groq.com/wp-content/uploads/2020/04/GROQP002_groq_whitepaper_V1-DB-1.pdf.

[12] Sonali Gupta. VGG-11 Architecture. https://iq.opengenus.org/vgg-11/.

[13] Hsin-Yuan Huang Jarrod McClean. Quantum Machine Learning and the Power of Data, 2021. https://ai.googleblog.com/2021/06/quantum-machine-learning-and-power-of.html.

[14] Aakash Kaushik. VGG-19 Architecture. https://iq.opengenus.org/vgg19-architecture/.

[15] Mahmoud Khairy, Jason Shen, Tor M. Aamodt, and Timothy G. Rogers. Accel-sim: An extensible simulation framework for validated gpu modeling. In *The 47th International Symposium on Computer Architecture*, New York, NY, USA, May 2020. ACM.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[17] Shilpa Lama. ChatGPT Review: Everything You Need to Know, 2023. https://beincrypto.com/learn/chatgpt-review/.

[18] Shilpa Lama. ChatGPT Review: Everything You Need to Know, 2023. https://www.globenewswire.com/en/news-release/2023/04/18/2649061/0/en/Data-Center-Accelerator-Market-Is-Expected-to-Reach-USD-130-3-billion-by-2032-Gr html.

[19] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[20] Francisco Muñoz-Martínez, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. Stonne: Enabling cycle-level microarchitectural simulation for dnn inference accelerators. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*, 2021.

[21] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 327–341, New York, NY, USA, 2018. Association for Computing Machinery.

[22] Izaak Neutelings. TikZ.net - Neural Networks, 2021. `https://tikz.net/neural_networks/`.

[23] NVIDIA. Nvidia A100 GPU, 2021. `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf`.

[24] NVIDIA. NVIDIA A100 Tensor Core GPU Architecture, 2021. `https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf`.

[25] NVIDIA. Nvidia converged accelerators, 2022. `https://www.nvidia.com/content/dam/en-zz/Solutions/gtcf21/converged-accelerator/pdf/datasheet.pdf`.

[26] NVIDIA. Nvidia tesla p4 gpu, 2023. `https://images.nvidia.com/content/pdf/tesla/184457-Tesla-P4-Datasheet-NV-Final-Letter-Web.pdf`.

[27] NVIDIA. NVIDIA Triton Inference Server, 2023. `https://developer.nvidia.com/nvidia-triton-inference-server`.

[28] OpenAI. ChatGPT: Optimizing Language Models for Dialogue, 2023. `https://chat.openai.com/chat`.

[29] Sunil Pai, Ben Bartlett, Olav Solgaard, and David A. B. Miller. Matrix optimization on universal unitary photonic devices. *Physical Review Applied*, 11(6):064044, June 2019.

[30] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 304–315, 2019.

[31] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[32] B.J. Shastri, A.N. Tait, and T. et al. Ferreira de Lima. Photonics for artificial intelligence and neuromorphic computing, 2021. `https://www.nature.com/articles/s41566-020-00754-y`.

[33] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[34] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Gianni Antichi, Paolo Costa, Hamed Haddadi, and Roberto Bifulco. Re-architecting traffic analysis with neural network interface cards. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 513–533, Renton, WA, April 2022. USENIX Association.

[35] Mengshu Sun, Pu Zhao, Yanzhi Wang, Naehyuck Chang, and Xue Lin. Hsim-dnn: Hardware simulator for computation-, storage- and power-efficient deep neural networks. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, GLSVLSI '19, page 81–86, New York, NY, USA, 2019. Association for Computing Machinery.

[36] Yannan Nellie Wu, Po-An Tsai, Angshuman Parashar, Vivienne Sze, and Joel S. Emer. Sparseloop: An analytical approach to sparse tensor accelerator modeling, 2023.

[37] Yaqi Zhang, Nathan Zhang, Tian Zhao, Matt Vilim, Muhammad Shahbaz, and Kunle Olukotun. Sara: Scaling a reconfigurable dataflow accelerator. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1041–1054, 2021.

[38] Qilin Zheng, Xingchen Li, Yijin Guan, Zongwei Wang, Yimao Cai, Yiran Chen, Guangyu Sun, and Ru Huang. Pimulator-nn: An event-driven, cross-level simulation framework for processing-in-memory-based neural network accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(12):5464–5475, 2022.

[39] Zhizhen Zhong, Mingran Yang, Christian Williams, Alexander Sludds, Homa Esfahanizadeh, Ryan Hamerly, Dirk Englund, and Manya Ghobadi. Lightning: A reconfigurable photonic-electronic smartnic for fast and energy-efficient inference. In *ACM SIGCOMM 2023 Conference*, SIGCOMM '23, 2023.