ACCELERATING EMERGING NEURAL WORKLOADS

by

Jacob R. Stevens

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

School of Electrical and Computer Engineering

West Lafayette, Indiana

December 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Anand Raghunathan, Chair
School of Electrical and Computer Engineering

Dr. Vijay Raghunathan
School of Electrical and Computer Engineering

Dr. Kaushik Roy
School of Electrical and Computer Engineering

Dr. Timothy G. Rogers
School of Electrical and Computer Engineering

Approved by:
Dr. Dimitri Peroulis

ACKNOWLEDGMENTS

First, I would like to thank my advisor, Dr. Anand Raghunathan, for his guidance and support throughout the past five years. His insights, curiosity, and drive have shaped me as a researcher; his eye for detail, writing feedback, and emphasis on intuitive explanations have improved my skills as a scientific communicator. Moreover, I am also deeply grateful for his care for my personal life, allowing me the flexibility necessary to complete my PhD journey.

I would also like to thank all of the members of my doctoral advisory committee: Dr. Vijay Raghunathan, Dr. Kaushik Roy, and Dr. Timothy Rogers. Their feedback, comments, and suggestions have helped me to deliver the best possible version of my research and thesis. I am thankful for the time that they dedicated to this end.

Next, I would like to thank the members, past and present, of the Integrated Systems Laboratory (ISL), including Dr. Ashish Ranjan, Dr. Shubham Jain, Dr. Younghoon Kim, Dr. Sanchari Sen, Manik Singhal, Sarada Krithivasan, Vinod Ganesan, Sourjya Roy, Shrihari Sridharan, Reena Elangovan, Amrit Nagarajan, Christin David Bose, Surya Selvam, and Aradhana Mohan Parvathy. I appreciate not only the research discussions and collaborations we have had, but also the conversations and fun.

In terms of other influences on my Purdue computer engineering career, I would like to thank Dr. Mark Johnson and Dr. Matthew Swabey for igniting my passion for computer hardware; those late-night, hours-long SoCET meetings were a truly formative time in my academic development. I would also like to thank my lab partner-in-crime for many of these years, Manik Singhal, as well as Everett P. Berry, for their numerous discussions of various technical interests.

Finally, I must thank my family: my parents, Keith and Katharine Stevens, whose hard work and sacrifice laid the foundation that allowed me to end up where I am today and whose love and support gave me the freedom to pursue my dreams. My sister, Allison Stevens, for believing in me and supporting me. And finally, I want to express my love and gratitude to my wife Julia Stevens, for navigating the up and downs of graduate school with me and without whom I could not have gotten through this journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Due to a combination of algorithmic advances, wide-spread availability of rich data sets, and tremendous growth in compute availability, Deep Neural Networks (DNNs) have seen considerable success in a wide variety of fields, achieving state-of-the art accuracy in a number of perceptual domains, such as text, video and audio processing. Recently, there have been many efforts to extend this success in the perceptual, Euclidean-based domain to non-perceptual tasks, such as task planning or reasoning, as well as to non-Euclidean domains, such as graphs. While several DNN accelerators have been proposed in the past decade, they largely focus on traditional DNN workloads, such as Multi-layer Perceptions (MLPs), Convolutional Neural Networks (CNNs), and Recurrent Neural Networks (RNNs). These accelerators are ill-suited to the unique computational needs of the emerging neural networks. In this dissertation, we aim to fix this gap by proposing novel hardware architectures that are specifically tailored to emerging neural workloads.

First, we consider memory-augmented neural networks (MANNs), a new class of neural networks that exhibits capabilities such as one-shot learning and task planning that are well beyond those of traditional DNNs. MANNs augment a traditional DNN with an external differentiable memory that is used to store dynamic state. This dissertation proposes a novel accelerator that targets the main bottleneck of MANNs: the soft reads and writes to this external memory, each of which requires access to all the memory locations.

We then focus on Transformer networks, which have become very popular for Natural Language Processing (NLP). A key to the success of these networks is a technique called self-attention, which employs a softmax operation. Softmax is poorly supported in modern, matrix multiply-focused accelerators since it accounts for a very small fraction of traditional DNN workloads. We propose a hardware/software co-design approach to realize softmax efficiently by utilize a suite of approximate computing techniques.

Next, we address graph neural networks (GNNs). GNNs are achieving state-of-the-art results in a variety of fields such as physics modeling, chemical synthesis, and electronic design automation. These GNNs are a hybrid between graph processing workloads and DNN workloads; they utilize DNN-based feature extractors to form hidden representations for

each node in a graph and then combine these representations through some form of a graph traversal. As a result, existing hardware specialized for either graph processing workloads or DNN workloads is insufficient. Instead, we design a novel architecture that balances the needs of these two heterogeneous compute patterns. We also propose a novel feature dimension-blocking dataflow to further increase performance by mitigating the memory bottleneck.

Finally, we address the growing difficulty in tightly coupling new DNNs and a hardware platform. Given the extremely large DNN-HW design space consisting of DNN selection, hardware operating condition, and DNN-to-HW mapping, it is infeasible to exhaustively search this space by running each sample on a physical hardware device. This has led to the need for highly accurate, machine learning-based performance models which can *predict* the latency/power/energy even faster than direct execution. We first present a taxonomy to characterize the possible approaches to these performance estimators. Based on the insights from this taxonomy, we present a new performance estimator that combines coarse-grained and fine-grained to achieve superior accuracy with a limited number of training samples. Finally, we propose a flexible framework for creating these DNN-HW performance estimators.

In summary, this dissertation identifies the growing gap between current hardware and new emerging neural networks. We first propose three novel hardware architectures that address this gap for MANNs, Transformers, and GNNs. We then propose a novel hardware-aware DNN estimator and framework to ease addressing this gap for new networks in the future.

1. INTRODUCTION

Deep Neural Networks (DNN) have become increasingly ubiquitous as they continue to improve upon the state-of-the-art results in a wide variety of domains. These DNNs power many popular services, including Apple's Siri, Google Translate, and Amazon's Alexa. The success of DNNs in recent years stems in large part from a virtuous cycle, wherein increases in compute power drive larger and larger models which in turn drive a demand for even more compute power. For example, the famous AlexNet [1] architecture that won the 2012 ImageNet competition handily and kicked off the deep learning revolution was comprised of roughly 60M weights. The recently released Megatron-Turing NLG [2], by contrast, has 530B parameters, a nearly 9,000x increase in model size in just nine years. NVIDIA's K20x, also released in 2012, had a peak single precision performance of 2.90 TFLOPs. Eight years later, NVIDIA's A100 has a peak performance for neural networks of 624 TFLOPs for training and 2496 TFLOPs for inference– an increase of 200x-800x. It is important to note that this increase in performance for executing neural networks resulted from careful hardware/software co-design; the A100 contains features tailored for deep neural networks such as specialized cores, low precision compute units, and sparsity-aware hardware. However, despite this impressive increase in compute, there is still a large gap in the growth of DNN models versus the growth in compute. This is further exacerbated by the emergence of new classes of DNNs– such as memory-augmented neural networks (MANNs), Transformers, and graph neural networks (GNNs)– for which existing commodity hardware is not specialized and thus cannot be executed at peak performance. Attempting to close this gap between the demands of software and the capabilities of hardware, particularly in the face of novel neural network workloads, is the main focus of this thesis.

In the following sections, we discuss the computational challenges of traditional DNNs, the unique challenges posed by emerging DNNs, our proposals to address those unique challenges through the development of novel specialized hardware architectures, and our effort to facilitate hardware/software co-design for future networks and platforms. Finally, we outline the remaining chapters of this dissertation.

1.1 Feed Forward Neural Networks

Feed-forward networks (FFNN), in which layers of neurons are stacked together with strictly forward connections (*i.e.*, no backward or recurrent connections), are an extremely popular class of networks. The two most popular types of FFNNs are multilayer perceptions (MLPs), which consist of only fully-connected layers, and convolutional neural networks (CNNs), which consist of a mix of fully-connected layers and convolutional layers. Convolutional layers exhibit local weight sharing, wherein neurons located near each other in space have their weights tied together in order to reduce the number of parameters needed. While the focus of this thesis is on emerging classes of neural networks specifically, rather than FFNNs, we believe that it is important to highlight the recent trends in FFNN performance and computational demands, as these networks are often used as the backbone of the more complex emerging classes of networks. Thus, these emerging workloads are likely to face a similar trajectory and will suffer from some of the same bottlenecks in addition to the unique bottlenecks posed.

To illustrate the high level of compute and memory needed for modern state-of-the-art FFNNs, we investigate a large set of networks that have achieved Top-5 accuracy over 90% on the challenging ImageNet [3] dataset. While there is a large amount of variability in the metrics of these SOTA networks, a few trends emerge. In Figure 1.1, we see that these networks tend to require roughly 6 GFLOPs per 224x224-sized image input, with networks requiring more FLOPs being less common. In Figure 1.2, we see that SOTA networks tend to have roughly 25M parameters.

Note that although this is already a considerable amount of compute and memory, image recognition CNNs are one of the smallest classes of FFNs thanks to the limited use of fully-connected layers. Other classes of networks, such as those used in recommendation systems like Facebook's Deep Learning Recommendation Model (DLRM) [5], can have an order of magnitude more FLOPs and parameters– for example, DLRM requires 540 M parameters.

To address the large compute and memory requirements for modern FFNNs, a wide array of accelerators with large amounts of compute as well as memory hierarchies specialized to the specific, fixed data access patterns of FFNNs have been proposed [6]–[11]. Further efficiency

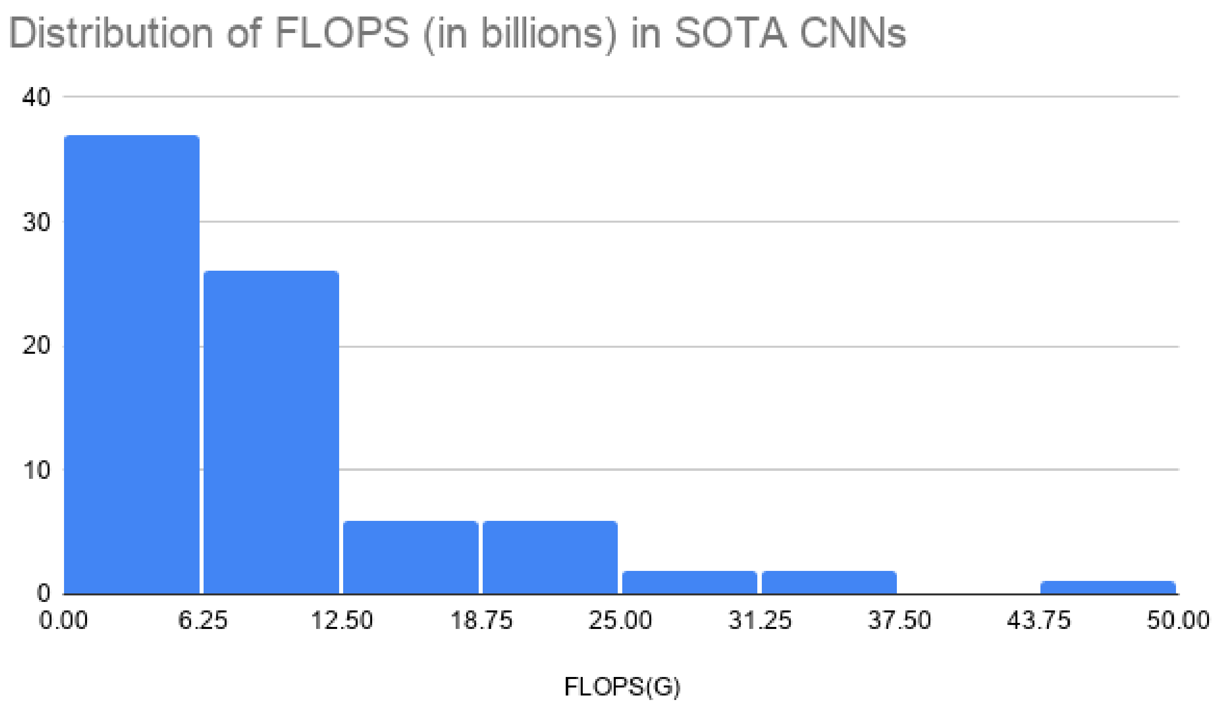


Figure 1.1. SOTA CNNs with Top-5 accuracy on ImageNet tend to require on the order of 6GFLOPs. Data from Computer Vision Leaderboard [4]

gains in hardware can be realized through exploiting sparsity [12]–[18] and low-precision [16], [19]–[22].

1.2 Memory Augmented Neural Networks

Memory-augmented neural networks (MANNs)– which augment a traditional Deep Neural Network (DNN) such as the feed-forward network described above with an external, differentiable memory– are emerging as a promising direction in machine learning. MANNs have been shown to achieve one-shot learning and complex cognitive capabilities that are well beyond those of classical DNNs. However, these increased capabilities come at an increased computational and memory cost; in order to keep accesses to the external memory differentiable– and therefore, allow accesses to be learnable– every single memory location must be read (written) to each time. Further, unlike many DNN workloads, there is extremely little reuse in these so-called *soft reads* and *soft writes* of the external memory. These accesses thus present a unique challenge that results in poor performance of MANNs on modern CPUs, GPUs, and other accelerators. In this thesis, we first provide a detailed

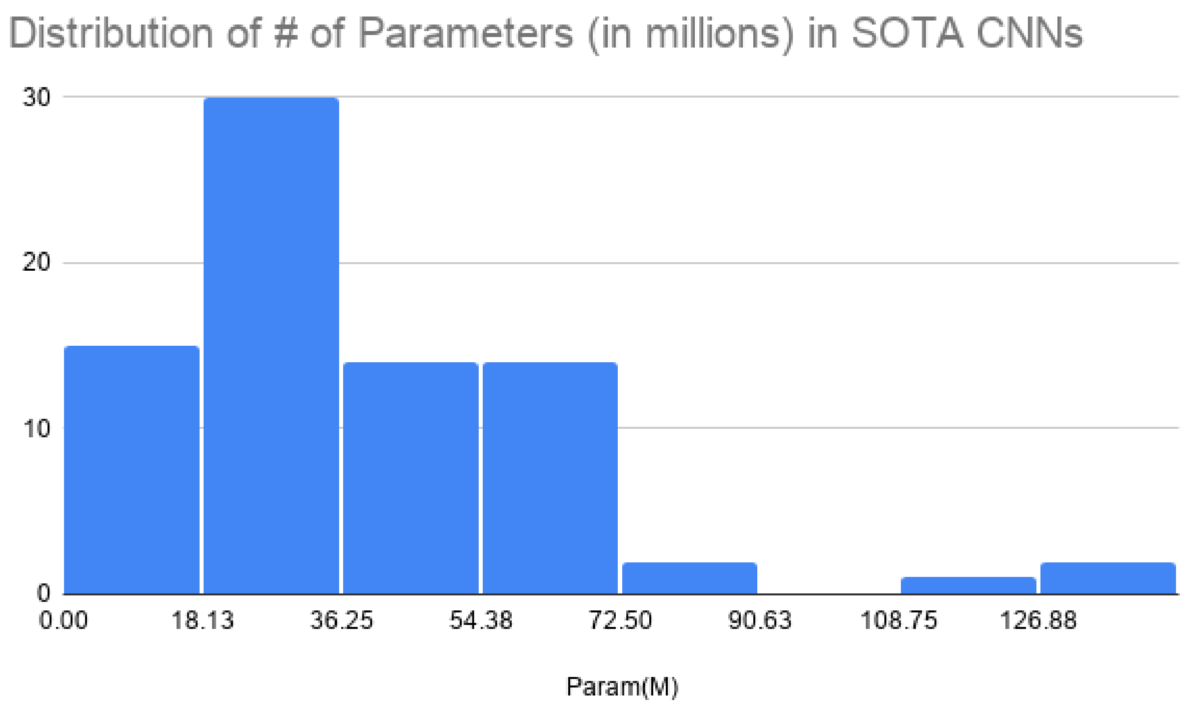


Figure 1.2. SOTA CNNs with Top-5 accuracy on ImageNet tend to require on the order of 25M parameters. Data from Computer Vision Leaderboard [4]

investigation of the computation and memory characteristics of the soft read and soft write kernels, as well as other MANN-specific kernels. We use this analysis to guide our development of MANNA, a specialized hardware inference accelerator for MANNs. MANNA is a memory-centric design that focuses on maximizing performance in an extremely low FLOPS/Byte context. The key architectural features from which MANNA derives efficiency are: (i) investing most of the die area and power in highly banked on-chip memories that provide ample bandwidth rather than large matrix-multiply units that would be underutilized due to the low reuse, (ii) a hardware-assisted transpose mechanism for accommodating the diverse memory access patterns observed in MANNs, (iii) a specialized processing tile that is equipped to handle the nearly-equal mix of MAC and non-MAC computations present in MANNs, and (iv) methods to map MANNs to MANNA that minimize data movement while fully exploiting the little reuse present.

1.3 Transformer Networks

An important subclass of MANNs are Transformer networks, which are currently transforming the field of natural language processing (NLP) through their impressive accuracy on a variety of tasks. The key aspect of Transformers is their use of "self-attention" layers, which consist mainly of matrix multiplies and softmax operations. These self-attention layers are repeated to form a full a network. Due to this heavy use of attention, the softmax operation accounts for a significant fraction of the total run-time of Transformers. This is in contrast to standard DNNs where the softmax operation is used only as the last classification layer and accounts for very little run-time. As a result, DNN accelerators typically do not optimize this operation, leading to a new bottleneck in these Transformer-based networks. To address this, we propose `Softermax`, a hardware-friendly softmax design that exploits approximate computing to enable the softmax logic to be more tightly coupled with the MAC operation, allowing for higher softmax throughput and alleviating the bottleneck. `Softermax` consists of four parts: (i) reducing the overhead of exponentiation by replacing the expensive natural number base with the lower cost base two, (ii) implementing low-precision softmax computations by leveraging the natural resiliency of these networks in contrast to the high-performance scientific computing applications that these special functions typical target, (iii) an online, integer-based normalization calculation which reduces the number of passes through the matrix needed to compute the softmax and (iv) `Softermax`-aware fine-tuning, which virtually eliminates the accuracy degradation that may have been introduced through these approximations.

1.4 Graph Neural Networks

Graph Neural Networks (GNNs) brings the success of deep learning from the Euclidean domain (*e.g.*, images, video, audio, *etc.*) to the non-Euclidean domain (*e.g.*, graphs); GNNs have achieved state-of-the-art results in a wide variety of application domains, such as social network recommendation systems and chemical synthesis. There are two major kernels used across all GNNs. One is a *feature extraction* step, which passes the features of a graph's nodes (and/or edges) through a fully-connected layer in order to obtain a new representation.

The second is an *aggregation* step, which combines the features within a given node's graph neighborhood, thereby incorporating information based on graph structure. These two steps are stacked to form multiple layers of a network. The final node (edge) features created via these layers can then be used for various tasks, such as node-, edge-, or graph-level classification. This heterogeneity in compute kernels introduces new, unique challenges that current computing platforms– CPUs, GPUs, and special purpose deep learning accelerators – do not address. Architectures such as GPUs and DNN accelerators that are typically targeted at dense, regular computations can perform the feature extraction step efficiently, but are ill-suited to the irregular memory access patterns present in the aggregation step. In contrast, architectures specialized for graph analytics workloads can perform the aggregation step efficiently, but are ill-suited for the large amount of regular computation needed to perform the feature extraction step. To address this shortcoming, we propose GNNerator, a graph neural network accelerator that utilizes two heterogeneous compute engines, one optimized for feature extraction and one optimized for feature aggregation. We also propose feature-blocking, a novel dataflow for GNNs that increases the number of nodes that can be held on-chip during GNN processing, helping to address the memory bottleneck inherent in GNNs. We provide hardware support for this dataflow in GNNerator. We develop a simulation framework to evaluate the performance benefits of the proposed architecture, to explore previously unexplored areas in the architecture design space for GNNs, and to analyze the potential bottlenecks for GNN accelerators as GNNs continue to increase in size.

1.5 Learned DNN Performance Estimators

Given the explosion in the number of DNN models proposed, combined with the increasing number of hardware devices on which to run them, it is becoming more and more difficult to optimally map a given DNN to a given hardware platform. This is particularly true for modern edge platforms, which typically integrate multiple heterogeneous hardware IPs and allow for the configuration of the operating condition of those IPs (*i.e.*, number of cores active, operating frequency, *etc*). It is infeasible to run every model directly on physical hardware, due the latency of the networks, the overhead of loading the networks on the device, and the physical number of devices acting as a bottleneck. As a result, there has been recent interest in developing sample efficient predictive *performance estimators*, which can then be used in downstream applications such as hardware-aware neural architecture search.

To this end, we first analyze the design space of DNN performance estimators, synthesizing this information into a design taxonomy. Using the insights from this taxonomy, we propose a novel hardware-aware DNN performance estimator specifically designed for the unique challenges of heterogeneous edge devices, which presents a reasonable trade-off between fine-grained and coarse-grained information. We develop a flexible framework for the rapid development of these performance estimators, and demonstrate the accuracy of the estimators produced on a variety of use cases.

1.6 Thesis outline

The rest of the thesis is organized as follows. Chapter 2 details the related research efforts in accelerating different types of DNNs. Chapter 3 provides the necessary background on DNNs, emerging classes of DNNs, and DNN-HW performance models. Chapter 4 proposes `Manna`, a memory-augmented neural network accelerator. Chapter 5 proposes `Softermax`, a set of hardware/software techniques for accelerating Transformers. Chapter 6 proposes `GNNerator`, a graph neural network accelerator. Chapter 7 describes our performance estimator design taxonomy, our novel hardware-aware performance model designed for heterogeneous edge devices, and our framework for generating these estimators. Finally, Chapter 8 concludes the thesis by summarizing its main contributions.

2. RELATED WORK

Deep neural networks have become a topic of increasing interest for computer system designers, as they have continued to achieve state-of-the-art results in a wide variety of tasks such as image recognition and speech translation through the use of more and more complex networks with greater and greater computational and memory requirements. In order to address these increasing demands of both traditional as well as emerging neural network architectures, there has been a wide array of research efforts. In this chapter, we categorize and discuss a variety of these efforts. Specifically, we first discuss research efforts addressing traditional neural networks that we believe are of particular interest for also addressing emerging neural networks. We next discuss research efforts specifically addressing memory-augmented neural networks (including Transformer-based networks) as well as graph neural networks, and identify the key differences with our work. Finally, we discuss the relation of previous efforts in DNN performance estimators to our proposal.

2.1 Deep Neural Networks

As discussed in Chapter 1, existing hardware and software optimizations for traditional Deep Neural Networks are insufficient for emerging neural workloads such as memory-augmented neural networks and graph neural networks. However, these emerging neural networks still use traditional DNNs as the backbone for some of their computations (*i.e.*, the controller in MANNs and the feature extractors in GNNs). Thus, it is instructive to explore the state of the field in optimizing DNNs.

2.1.1 Algorithmic optimizations

Due to the immense popularity of DNNs in recent years, there is an extremely large body of work on algorithmic optimizations for DNNs. For the sake of brevity, we focus mainly on the families of optimizations that we believe are the most likely to be relevant for MANNs and GNNs: quantization and sparsity optimizations.

Quantization. Originally, computations for DNNs were done using 32-bit floating point representations. However, DNNs are intrinsically quite resilient and can be performed using far fewer bits. [23] proposed using a quantized forward pass with low-precision weights and activations, with a full-precision backwards pass using a straight-through estimator (STE) along with stochastic rounding. A variety of other techniques have been proposed since, including learning various quantization parameters [24], quantization-aware training [25], per-layer mixed-precision quantization [26], and blocked precision scaling [22].

Sparsity. Modern DNN workloads exhibit considerable *sparsity* in various data structures of the network, such as activations and weights. DNN sparsity can either be static or dynamic. Static sparsity results from pruning low-valued (and therefore, insignificant) weights from a network, in a process referred to as network pruning. Network pruning has shown to be very effective in modern CNNs [27] as well as other architecture types such as RNNs [28]. To exploit static sparsity, weights are stored in a sparse format, allowing more of them to fit on-chip and avoiding expensive DRAM accesses. Exploiting dynamic sparsity, which typically results from use of the ReLU activation function, is more difficult in software.

2.1.2 Specialized accelerators

Specialized accelerators for deep neural networks is an extremely robust field of research, for both inference [8], [10], [29], [30] as well as training [6], [9]. DNN accelerators typically fall into two categories: systolic-array based [8] and SIMD-based [6], [9], [10], [29], [30]. Both categories of accelerators typically provide the compute units with access to small register files to store inputs, weights, or partial sums, as well as relatively larger on-chip buffers for an additional level of memory foor the same.

In order to fully exploit the benefits derived from the algorithmic optimizations described above (*i.e.*, enable performance benefits in addition to the compression benefits), many DNN accelerators, including commercial products [16], have also begun to integrate hardware support for quantized and sparse computations.

Quantization. On top of reducing memory storage requirements, quantization also provides performance benefits by enabling low-precision arithmetic, increasing the amount of compute in a given area budget as well as increasing the effective bandwidth. While many accelerators simply directly implement low-precision (typically eight-bit) arithmetic [8], other accelerators propose further specializations, such as precision-scaling computation [19], [22], [31] or compensation support [21] in order to enable efficient computation at even lower precisions.

Sparsity. Like quantization, sparsity also offers benefits beyond reducing memory storage requirements; by skipping the ineffectual computations introduced by sparsity, sparsity-aware accelerators can realize large performance gains. There has been much interest in recent years in techniques to efficiently exploit sparsity while minimizing the area and power overheads [12]–[15], [17], [32]. One key challenge is load balancing [13], as unstructured sparsity can result in uneven distribution of work across processing elements. Another key challenge is efficiently computing intersections of sparse representations [33], often used to determine which computations are actually effective.

2.2 Memory-Augmented Neural Networks

Memory-augmented neural networks utilize a form of self-attention, realized through a so-called external differentiable memory, in order to provided capabilities well beyond traditional DNNs. The accesses to this differentiable memory, as well as the addressing schemes used to determine the accesses, constitute a large fraction of the run-time of MANNs. As a result, both the algorithmic optimizations and specialized accelerators for MANNs target these kernels.

2.2.1 Algorithmic optimizations

Algorithmic optimizations have been proposed to address the large overhead associated with the differentiable memory. [34] uses a K-nearest neighbor scheme to update only the k most relevant memories, instead of all locations. A concurrent work explores a similiar idea, but introduces additional levels of hierarchy [35]. Finally, Q-MANN [36] addresses the differentiable memory bottleneck through the use of quantization. By replacing the cosine similarity with a Hamming distance, Q-MANN is able to perform the soft memory operations using eight-bit fixed-point calculations without a catastrophic loss in accuracy.

2.2.2 Specialized accelerators

Given the memory-intensive nature of MANNs, many of the hardware-specific research efforts have focused on processing-in-memory (PiM), often using emerging devices. [37] replaces the cosine-based similarity computation often used in MANNs with a fixed-point, hardware-friendly approximation, allowing the computation to be executed using efficient ternary content addressable memory (TCAM) look-ups. Similarly, X-MANN [38] also replaces the similarity computation with a lower-cost alternative, allowing the differentiable memory operations to be realized using resistive crossbars.

There have also been efforts outside of PiM-based architectures. [39], a FPGA-based architecture, introduces Bayesian-based inference thresholding, allowing the architecture to stop soft reads early if a certain confidence level is met. MnnFast [40] proposes two key hardware-software co-design strategies for MANNs. Column-based "lazy softmax" allows for greater parallelization of a fused key similarity and soft read kernel, while an embedding cache provides dedicated memory for the memory intensive embedding operation of a MemNN (a variant of MANN) to reduce cache-thrashing of the other data structures.

2.3 Transformer Networks

An important subclass of MANNs that warrant specific attention are Transformer networks

2.3.1 Algorithmic optimizations

Most recent algorithmic works have tried to make Transformers more efficient by targeting their large number of parameters, rather than targeting the softmax operation. This is commonly performed through the creation of smaller networks through techniques such as knowledge distillation [41], inductive biases [42], or approximations [43]. These optimizations are orthogonal to our effort, as `Softermax` can still be used in these smaller, more efficient networks. There have also been a few efforts towards more efficient Transformers through the use of lower-precision computation [41], [44], with the most recent efforts also targeting quantization of the softmax operation [45], [46]. However, as these works are software-only, there are no actual gains in performance from their softmax quantization techniques. This is because full-precision special function units are still used for the exponential and division calculations– in fact, there may actually be a slight performance penalty due to overheads in casting between data types. Our work, in contrast, is able to exploit low-precision in both software and hardware, unlocking actual performance gains.

2.3.2 Specialized accelerators

There have been efforts to provide specialized accelerators for Transformer networks as a whole. A3 [47] provides hardware support for a lower cost attention mechanism , while SpAtten [48] allows for hardware-aware pruning of Transformers. EdgeBERT [49] provides hardware support for voltage-scaled early exit from Transformer networks. There have also been a few recent efforts that offer specialized hardware that target the softmax operation specifically [47], [50]–[52]. These efforts propose various low-precision exponential and division units, such as a variable precision softmax unit generator [52], a group lookup table-based exponential unit [51], a split high-bits/low-bits exponential unit [47], and approximate units [50]. This allows for exploiting low-precision within the special functions, unlike in the software only approaches.

2.4 Graph Neural Networks

There are two main kernels of GNNs with vastly different needs: the feature extraction stage and the aggregation stage. Since software packages for accelerating feature extraction on GPUs are already quite robust, most algorithmic-focused work has been on optimizing for the memory requirements of the aggregation stage. Specialized accelerators have also been proposed to meet the heterogenous requirements of GNNs.

2.4.1 Algorithmic optimizations

As GNNs are still a relatively new area, most of the focus on software optimizations has been on simply providing optimized implementations for existing hardware. For example, the popular DGL framework [53] utilizes fused message passing in order to reduce the memory overhead associated with the aggregation step of GNNs by replacing the defined computations with equivalent Sparse-Dense Matrix operations. Note that this sparsity is due to the underlying graph structure of the input and is orthogonal to the potential sparsity in the feature extraction layers (*i.e.*, fully connected layers). Roc [54] provides a distributed GPU framework, allowing users to scale to extremely large real-world graphs that would be infeasible on a single GPU due to memory constraints. GNNAdvisor [55] proposes a runtime that sets GPU-related parameters based on characteristics of the input graph (*e.g*, node degree) in order to maximize performance. [56] provides detailed insights into the specific bottlenecks when running GNNs on Intel hardware and proposed optimizations for the identified bottleneck, the aggregation stage. More recently, algorithmic optimizations have expanded to include techniques like quantization [57] and eliminating redundant computations [58].

2.4.2 Specialized accelerators

There have been a few recent works directly targeting graph neural networks. Similar to GNNerator, HyGCN [59] proposes two heterogeneous compute engines, one optimized for the feature extraction stage and one for the aggregation stage. Unlike in our work, HyGCN only exploits intranode parallelism, processing a single node's entire feature across all cores before moving on to the next node, while we exploit both intranode and internode parallelism–that is, the parallelism within a node due to feature dimensions and the parallelism between nodes. GNNA [60] uses smaller compute engines as a base element, and connects multiple of these tiles in a NoC. Unlike HyGCN, the engines do not share on-chip storage, but rather communicate over an on-chip cross-bar switch. Finally, EnGN [61] proposes a unified GNN accelerator architecture, in which the feature extraction and aggregation computations are performed using the same register file and compute units. Much of EnGN's performance comes from two sources: dimension-aware stage re-ordering and a degree-aware vertex cache.

The dimension-aware stage re-ordering is a simple software optimization and is indeed actually standard in GNN frameworks such as DGL, while the degree-aware vertex cache is orthogonal to our architecture and can be integrated into GNNERATOR.

2.5 DNN Performance Estimation

One popular method of designing estimators [62]–[65] relies on the composability of DNNs– the fact that DNNs are composed from a relatively small number of types of layers (Convolutional, Fully-Connected, ReLU, *etc.*). These works create estimators for each layer, which are then aggregated together to predict for the entire network. The other main approach [66], [67] predicts the performance of a network directly, instead of on a layer-by-layer basis.

The efforts described above are largely agnostic to the underlying hardware. [68] addresses this by building hardware-aware estimators that generalize across different hardware platforms by using a suite of networks as a ”hardware fingerprinting” set. Each hardware platform is still treated as a fixed platform, however, ignoring the ability to vary the hardware operating condition. Further, these works all ignore the heterogeneity of new edge platforms, wherein the DNN may be partitioned across multiple hardware IPs. Finally, these estimators are created in an ad-hoc manner, with little justification for the chosen estimation strategy, input features, *etc.*

2.6 Thesis contributions

The primary contributions described throughout this thesis are different or complementary to the prior work described above in the following ways.

Specialized accelerators for MANNs. [37] and [38] rely on emerging device technologies that are not widely available. Further, these works are also not an end-to-end solution since they do not accelerate the addressing kernels or the DNN controller. MANNA, by contrast, is a CMOS-based architecture and implements all MANN kernels. The other proposed architectures, MnnFast [40] and [39], are fixed-function architectures that are specialized for a specific class of MANNs, end-to-end memory networks (MemNets), whereas MANNA has sufficient programmability to realize a broad class of MANNs (*e.g.*, NTMs and DNCs from Google DeepMind), which is important given the evolving nature of MANNs. Moreover, since MemNets do not require soft writes, these accelerators are not designed to support non-MAC operations that constitute a key kernel in soft writes. Consequently, these architectures suffer from reduced throughput for such element-wise operations, resulting in inefficient soft writes. Besides, since MemNets do not update the differentiable memory, these efforts store a copy of the memory in its transposed form and do not provide any support for on-chip transpose. For the variants of MANNs that do require soft writes, hardware support is crucial; our ablation study indicates that support for element-wise operations and on-chip transpose lead to speedups of `2.8x` and `1.4x`, respectively.

Software optimizations for Transformers. We are the first to propose the use of a different, more hardware-friendly base in the softmax operation of Transformers. Further, we propose a more efficient, *integer-based* online normalization scheme, mitigating the overheads present in other online normalization methods.

Specialized accelerators for Transformers. Many of the accelerators previously proposed for Transformers [47]–[49] are orthogonal to Softermax, as these efforts do not target the softmax operation specifically. Within the softmax operation specifically, low-precision hardware has been explored before [50]–[52]. None of these efforts, however, use our other approximate computing techniques, such as base substitution and integer-based online normalization. These techniques work in concert with our low-precision hardware to maximize hardware efficiency. Further, our Softermax-aware finetuning process allows for the use of a smaller LUT than in previous efforts.

Software optimizations for GNNs. While we are mostly focused on specialized accelerators, we note that many of the software optimizations proposed for GNNs are either indeed present in our accelerator or are orthogonal and can be incorporated. As a result of our architecture, which consists of an Apply Unit that feeds directly into the Reduce Units, we can fuse nearly all message passing kernels, not just those which map cleanly into sparse-dense matrix operations as in DGL.

Specialized accelerators for GNNs. There have been a few recent accelerators for GNNs proposed: HyGCN [59], EnGN [61], and GNNA [60]. All three of these works require entire features to be resident on-chip at once, reducing the size of the subgraph that can be processed. Further, none of the three have support for operations that modify features through a vector-vector operation before aggregations– for example, by multiplying a feature by a weight for attention-based networks or by an error gradient for training. GNNERATOR supports this through the Apply Engine and its corresponding instructions. This limits the usage of these accelerators in emerging GNN workloads, as well as a base architecture for training hardware.

Predictive performance estimators for DNNs. There have been a number of different performance estimators proposed that use a variety of different techniques (layer-wise vs direct network) and different features (layer details, number of MACs, memory footprint, *etc.*). It is difficult to understand the impacts of these different hyperparameters. To address this, we step back and provided a principled look at the DNN performance estimator design space, providing a taxonomy of the various designs. Further, we provide a flexible framework for quickly generating different performance estimators and comparing them with iso-conditions. Furthermore, unlike previous efforts, we consider the full capabilities of heterogeneous hardware platforms, allowing for predictions across the range of operating conditions of a device, as well as across different DNN-to-IP mapping strategies.

3. BACKGROUND

In this chapter, we provide a brief background on traditional DNNs, as well as the emerging MANN and GNN workloads.

3.1 Deep Neural Networks

In this work, we refer to traditional Deep Neural Networks (DNNs) as encompassing two major families of architectures: feed-forward neural networks (FFNNs) and recurrent neural networks (RNNs).

Feed-forward neural networks. The simplest FFNN is the perceptron [69], which consists of a layer of input and ouputs units called neurons. These neurons are connected in an all-to-all pattern and the importance of these connections are defined by the connection's *weight*. To train the perceptron, the error between the predicted output and the correct, labeled output is computed and used to update the weights in a process referred to as gradient descent. Multilayer perceptrons build upon this by adding multiple *hidden* layers, which are layers of neurons between the input and output layers. A similiar training process is followed for multilayer networks, with the exact updates required for each layer determined in reverse order through a process known as backpropogation. Convolutional layers introduce weight sharing, which builds spatial-based priors into the network. Additional layer types can include pooling layers, which downsample the network, and activation layers.

Recurrent-neural networks. Recurrent Neural Networks extend FFNNs by introducing a recurrent connection, thus introducing memory to the network by allowing dynamic state to persist across iterations of the network. However, RNNs can be difficult to train because the error gradients through the RNN tend to either explode or vanish when using backpropagation through time (BPTT). In practice, this means that RNNs tend to either be susceptible to noisy input or have difficulty learning long-term dependencies [70]. To overcome this limitation, [71] proposed Long Short-Term Memory networks (LSTMs). LSTMs build upon a basic recurrent unit by adding a memory cell (C) that stores state. LSTMs also add a forget gate (f), an input gate (i), and an output gate (o) that control the retaining, addition, and outputting of this stored state, respectively. Later, [72] proposed a Gated Recurrent

Unit (GRU) that is similar to LSTMs, but slightly simplified. An additional memory cell is not used; instead, the flow of information of the hidden state is directly modulated through the use of gates. The GRU retains analogs to the input and forget gates, but eliminates the output gate, exposing the entire state instead. RNNs and its variants have resulted in recent success in some domains, such as neural machine translation [73], [74] and speech recognition [75], [76]. However, there exists a fundamental constraint on the use of RNNs for storing dynamic state. The dynamic state is intrinsically coupled to the design of the network. As a result, the amount of information that can be stored in the network cannot be increased without increasing the size of the underlying network, thereby increasing the total number of parameters that must be trained.

3.2 Memory-Augmented Neural Networks

To address the aforementioned scaling problem, memory-augmented neural networks (MANNs) have been proposed, wherein the dynamic state is explicitly decoupled from the neural network in an external memory. Several variations of MANNs have been proposed [77]–[83] to tackle a diverse variety of tasks such as question answering, route planning, scene understanding, and language transduction. These MANNs have also been used as building blocks for goal-directed agents in reinforcement learning settings [84].

3.2.1 Neural Turing Machines Overview

While several variants of MANNs exist in the literature, we will focus on DeepMind's Neural Turing Machine (NTM) [81] for our exposition. In this section, we provide a detailed description of NTMs.

As shown in Figure 3.1, NTMs are composed of three main components: a neural network-based controller, a differentiable external memory, and the read and write heads that control the interaction between the two. To ensure the external memory is differentiable, NTMs use soft read and write operations. These soft operations differ from traditional read and write operations in that the soft operations require access to all the locations in memory.

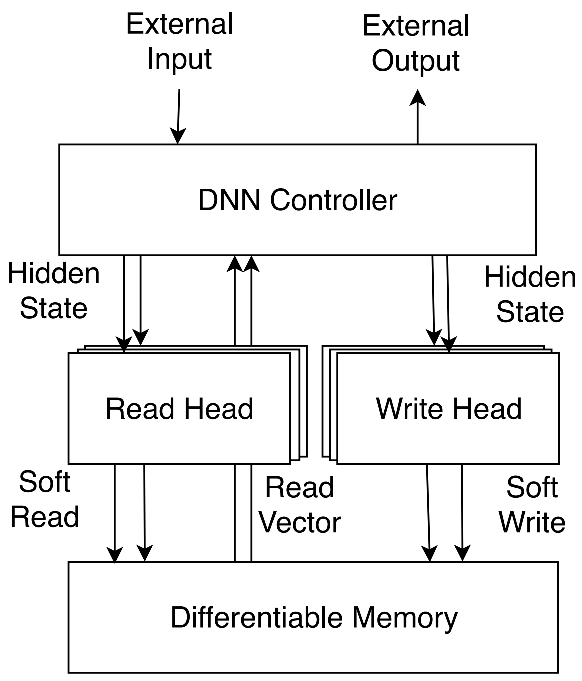


Figure 3.1. A high-level representation of the Neural Turing Machine

Attention mechanisms are used to determine how the values of all of the different locations are combined (read) or updated (write).

Controller. NTMs employ a standard DNN as a controller that generates both the final output vector as well as the hidden state vector for the read and write heads. A wide range of DNNs, *viz.*, MLPs of varying depths, RNNs, CNNs, *etc.*, can be utilized as the controller. Even with a feedforward network-based controller, NTMs are intrinsically sequential algorithms. At each time step t, the controller network receives as inputs an external input vector and the read vectors ($\mathbf{r}_h^{t-1}$) from each read head (h) corresponding to the previous time step $t-1$. The controller then produces an output vector that is sampled at the end of execution of the NTM, and a hidden state vector that is used by the read/write heads to interact with the external memory state at the same time step. Throughout this thesis, we define $\mathbf{M}^t$ as the differentiable external memory M at time t. $\mathbf{M}^t$ is composed of M_N row vectors, where each row vector $\mathbf{M}^t(\mathrm{i})$ consists of M_M words (dimensions). Thus, $\mathbf{M}^t$ comprises of M_N rows and M_M columns.

Soft Read. A soft read is a weighted summation over all the locations in $\mathbf{M}^t$ at time t, producing one read vector, $\mathbf{r}_h^t$, for each read head h. This can be formulated as a vector-matrix multiplication between the transpose of the weight column vector $\mathbf{w}_h^t$ and $\mathbf{M}^t$ for each read head h, as shown in Equation 3.1. The column vector w_h^t reflects the contribution of the corresponding rows to the final soft read vector, and is obtained by the attention mechanism described below.

$$\mathbf{r}_h^t = \mathbf{w}_h^{t\,T}\mathbf{M}^t, \forall\ h \in H_r \tag{3.1}$$

Soft Write. A soft write involves two steps: an *erase* and an *add* operation shown in Equations 3.2 and 3.3. In the erase step, an erase row vector $\mathbf{e}_h^t$ produced by the write head h is multiplied with the corresponding scalar from the weight column vector $\mathbf{w}_h^t$. The resulting vector is next subtracted from $\mathbf{1}$ (a vector of length M_M containing all 1s). Finally, the subtracted vector is multiplied in an element-wise fashion with $\mathbf{M}^t(\mathrm{i})$, resulting in a modified external memory $\mathbf{M}^t$. After erasing, the modified external memory is then updated by adding a similarly weighted add vector $\mathbf{a}_h^t$. The erase and add steps are repeated for each row in $\mathbf{M}$, as well as for each write head h, in order to complete the soft write operation.

$$\mathbf{M}^t(\mathrm{i}) = \mathbf{M}^t(\mathrm{i}) \odot [\mathbf{1} - \mathbf{w}_h^t(\mathrm{i}) \cdot \mathbf{e}_h^t] \tag{3.2}$$

$$\mathbf{M}^{t+1}(\mathrm{i}) = \mathbf{M}^t(\mathrm{i}) + \mathbf{w}_h^t(\mathrm{i}) \cdot \mathbf{a}_h^t, \forall \mathrm{i} \in M_N,\ \forall\ h \in H_w \tag{3.3}$$

Read/Write Heads. Each read and write head h consists of a weight matrix W_h. In each head, the hidden state from the controller is multiplied by the weight matrix to produce a set of vectors and scalars (*e.g.*, the key vector, erase vector, *etc.*) that are used as inputs to the attention mechanisms present in NTMs.

Attention Mechanism. To determine the weights for soft read and write operations, NTMs use a series of attention mechanisms that consume the output of the read and write heads. First, content-based weighting produces a weight vector $\mathbf{w_c}_h^t$ of length M_N for each read and write head h. Each element of the content weight column vector for a given head is based on the similarity between the corresponding row of $\mathbf{M}^t$ and the key row vector $\mathbf{k}_h^t$ emitted by that head (Equation 3.4).

$$Sim[\mathbf{k}_h^t, \mathbf{M}^t(\mathrm{i})] = \frac{\mathbf{k}_h^t \cdot \mathbf{M}^t(\mathrm{i})}{||\mathbf{k}_h^t|| \cdot ||\mathbf{M}^t(\mathrm{i})||} \tag{3.4}$$

$${\mathbf{w_c}}_h^t(\mathrm{i}) = \frac{\exp(\beta_h^t Sim[\mathbf{k}_h^t, \mathbf{M}^t(\mathrm{i})])}{\sum_{\mathrm{j}}^{M_N} \exp(\beta_h^t Sim[\mathbf{k}_h^t, \mathbf{M}^t(\mathrm{j})])} \tag{3.5}$$

After obtaining the similarity for all M_N memory vectors in $\mathbf{M^t}$, the similarities are then amplified (attenuated) by a scalar β_h^t and a softmax function is applied to obtain ${\mathbf{w_c}}_h^t$ (Equation 3.5). Next, a scalar interpolation gate g_h^t is used to blend the current weighting with the weighting produced at the previous time step of the algorithm (Equation 3.6). The result of the interpolation is then convolved with a rotation vector $\mathbf{s}_h^t$ (Equation 3.7). Finally, the result of this convolution is sharpened by a scalar λ_h^t and normalized in order to combat the blurring that may occur as a result of the shifting (Equation 3.8). This produces the final weight vector $\mathbf{w}_h^t$ corresponding to each read and write head key vector, which are then used in the soft read and write operations.

$${\mathbf{w_g}}_h^t(\mathrm{i}) = g_h^t {\mathbf{w_c}}_h^t(\mathrm{i}) + (1 - g_h^t){\mathbf{w_c}}_h^{t-1}(\mathrm{i}) \tag{3.6}$$

$${\mathbf{w_s}}_h^t(\mathrm{i}) = \sum_{\mathrm{j}}^{M_N-1} {\mathbf{w_g}}_h^t(\mathrm{j})\mathbf{s}_h^t(\mathrm{i}-\mathrm{j}) \tag{3.7}$$

$$\mathbf{w}_h^t(\mathrm{i}) = \frac{{\mathbf{w_s}}_h^t(\mathrm{i})^{\lambda_h^t}}{\sum_{\mathrm{j}}^{M_N} {\mathbf{w_s}}_h^t(\mathrm{j})^{\lambda_h^t}} \tag{3.8}$$

3.2.2 Neural Turing Machines Computational Behavior

For the following experiments, we use the `copy` task from the original NTM paper [81] as an illustrative example and execute it on an NVIDIA Turing GPU and an Intel Skylake Xeon CPU.

Table 3.1. Summary of kernels in Neural Turing Machines

Kernel Name	Key Primitive	Mem. Accesses	FLOPs/Byte	Reduction
Key Similarity	Vector-Matrix Mul.	$\mathcal{O}(\mathbf{M_N} \cdot \mathbf{M_M} \cdot (\mathbf{H_r} + \mathbf{H_w}))$	$H_w + H_r$	Row-wise
Content-based Weighting	Normalization	$\mathcal{O}(M_N \cdot (H_r + H_w))$	3	-
Location Interpolation	El.-wise Mul/Add/Sub	$\mathcal{O}(M_N \cdot (H_r + H_w))$	2	-
Shift Weighting	Circular Conv.	$\mathcal{O}(M_N \cdot (H_r + H_w))$	S	-
Weight Sharpening	Normalization	$\mathcal{O}(M_N \cdot (H_r + H_w))$	3	-
Soft Read	Vector-Matrix Mul.	$\mathcal{O}(\mathbf{M_N} \cdot \mathbf{M_M} \cdot \mathbf{H_r})$	H_r	Column-wise
Soft Write	El.-wise Mul/Add/Sub	$\mathcal{O}(\mathbf{M_N} \cdot \mathbf{M_M} \cdot \mathbf{H_w})$	H_w	-

Memory access characteristics. Table 3.1 categorizes the various NTM kernels depending on their computational primitives and identifies the number of memory accesses associated with each kernel. There are three distinct groups of kernels: the controller kernel that consists of a classical DNN, the addressing kernels that determine how the external memory is accessed, and the kernels that actually access the external memory. The addressing kernels– content-based weighting, location interpolation, shift weighting, and weight sharpening– involve $\mathcal{O}(M_N)$ memory accesses per head, since they are used to create and modify the weight column vectors $\mathbf{w}_h$. These kernels also have a relatively low FLOPs/Byte ratio (two or three).

On the other hand, the access kernels– key similarity, soft write, and soft read– are extremely memory-intensive kernels with many more memory accesses ($\mathcal{O}(M_N \cdot M_M)$) since each of these kernels involve accessing every single element in the differentiable memory at least once for each of the read (write) heads. Furthermore, these kernels exhibit very little reuse, accessing the external memory in a streaming fashion. Note that the only opportunity for reuse in such kernels is across multiple heads, which is usually less than a factor of five. This combination of large memory footprint and extremely low FLOPs/Byte ratio cannot be mitigated through the standard techniques for DNNs, *viz.*, compression and batching. Since the differentiable memory content is dynamic, the memory footprint cannot be reduced by using weight compression techniques [85]. Moreover, the external memory is unique to each input sequence to the NTM, and therefore it cannot be shared across input batches similar to RNNs and MLPs. Thus, accelerating NTMs requires designing hardware specifically for low FLOPs/Byte ratios. It is also important to note that while both key similarity and soft reads involve vector-matrix multiplications, the external memory access patterns for the constituent dot products are fundamentally different. Specifically, dot products for key similarity are performed across the rows, while dot products for soft read are performed across the columns. Therefore, it is also important to enable efficient access along both rows and columns.

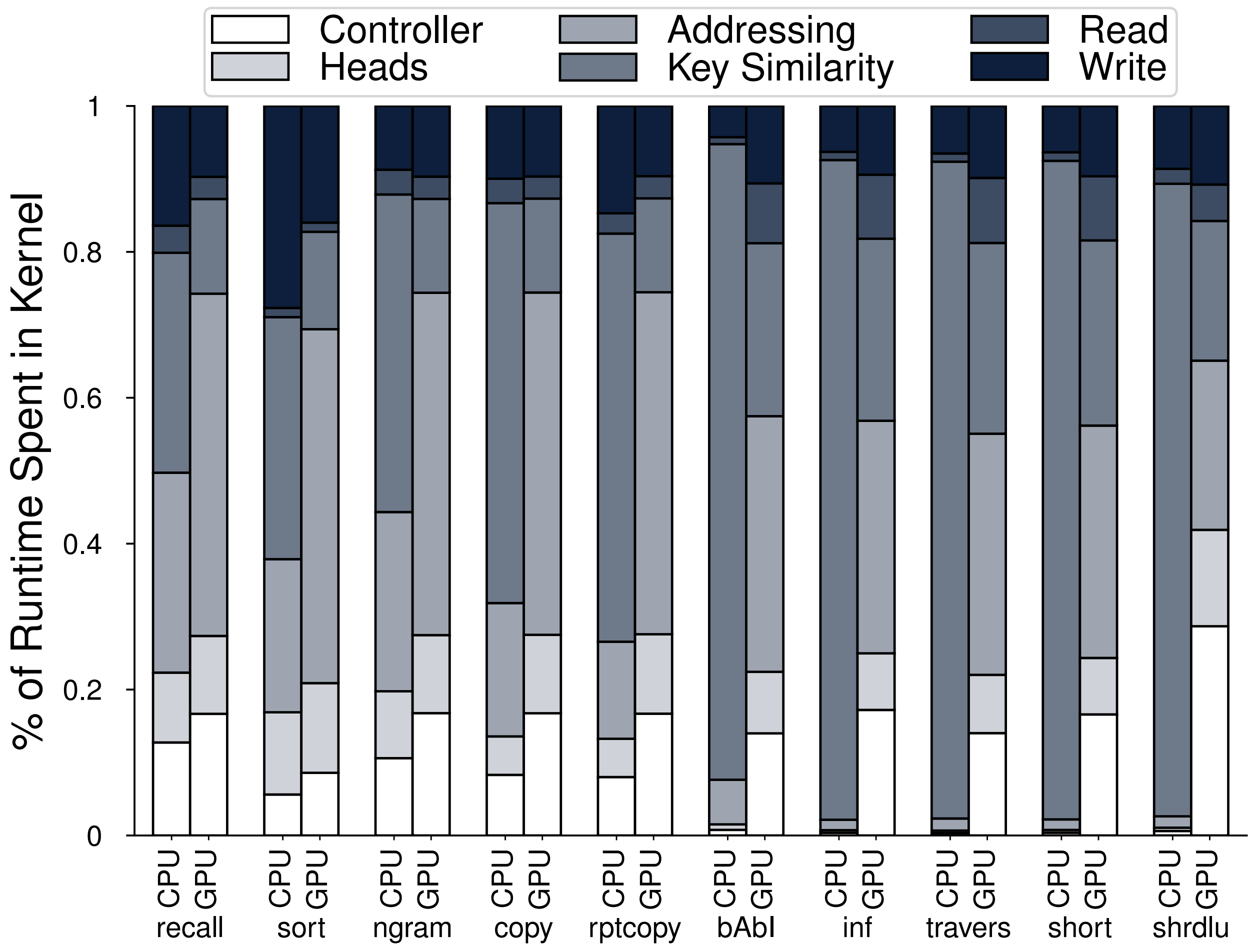


Figure 3.2. Runtime breakdown of different NTM kernels

Kernel breakdown. Figure 3.2 presents a breakdown of the runtime spent during inference in different MANN kernels across a suite of ten benchmarks. These benchmarks use differentiable memories that have been scaled to be larger, but still fit in a reasonably-sized on-chip memory (40MiB). For clarity, the read and write head kernels have been grouped together, as have all of the addressing kernels. As shown in the figure, the non-controller kernels, *viz.*, heads, addressing, key similarity, soft reads, and soft writes, dominate the runtime, making up roughly 80% of the total. This is particularly true for benchmarks that require higher differentiable memory capacity (`bAbI`, `inference`, `traversal`, `shortest`, and `shrdlu`). This behavior is expected, since MANNs decouple the dynamic state from the network itself, allowing the amount of dynamic state stored to grow dramatically without significantly increasing the size of the controller network. Hence, efficiently realizing these kernels is key to high performance in MANNs.

Interestingly, the kernels that dominate the total runtime within these runtime-intensive kernels vary depending on the platform. For the CPU platforms, the dominant kernels, particularly at larger differentiable memory sizes, are key similarity, soft write, and soft read. This is because these kernels are extremely memory-intensive compared to the remaining kernels as shown in Table 3.1. However, on GPUs, the vector-only addressing kernels represent an unexpectedly large portion of the runtime, comparable to the memory-intensive access kernels. This is because the memory-heavy access kernels (*e.g.*, key similarity) are large enough to fully utilize the GPU. The addressing kernels, on the other hand, are much smaller, resulting in so-called "narrow tasks" that exhibit poor GPU utilization [86] and thus poor performance due to the GPU kernel call overheads. CPUs, on the other hand, have fewer cores and thus exhibit good utilization even for limited parallelism in the addressing kernels. In order to efficiently perform all of the NTM kernels, an accelerator therefore must be able to efficiently execute both the large, extremely parallel matrix operations as well as the comparatively smaller vector operations.

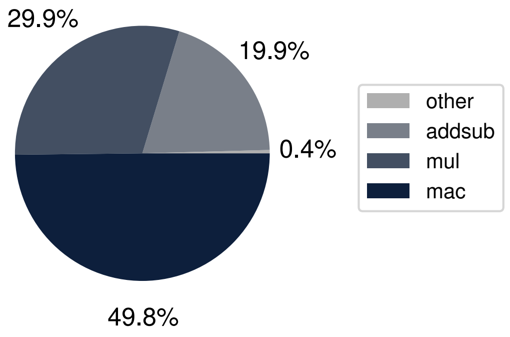


Figure 3.3. Relative mix of operations in runtime-intensive NTM kernels

Operation breakdown. Finally, we analyze the relative mix of operations that constitute the runtime-intensive kernels by analytically modeling the number of operations of each type that would be executed when running the `copy` benchmark. As shown in Figure 3.3, we found that, unlike traditional DNN kernels, the non-controller kernels are roughly equally comprised (49.8% each) of fused multiply-and-accumulate operations and element-wise vector operations, specifically multiplication, addition, and subtraction. Thus, an accelerator for MANNs cannot solely emphasize MAC or dot-product performance but rather a high throughput is required on a wider variety of operations.

In summary, we make several observations for desiderata in a MANN accelerator. First, as demonstrated by Figure 3.2, MANN-specific kernels dominate the runtime and therefore a MANN-specific accelerator is warranted. In particular, such an accelerator should: (i) balance the on-chip compute and memory resources for extremely low FLOPs/Byte workloads to maintain high utilization; (ii) provide support for efficient dot products across both rows and columns of the external memory; and (iii) accommodate the higher use of non-MAC compute elements.

3.3 Transformer-based Networks

Transformer-based networks can be considered a sub-type of MANNs. Specifically, these networks also utilize an analogue of the *soft attention* reading mechanism found in MANNs (though they do not emply soft write). However, it is useful to examine these networks in depth.

3.3.1 Transformers

A Transformer network is a deep neural network (DNN) that consists of one or more embedding layers, followed by multiple Transformer layers, and finally a task-specific final layer that is added when fine-tuning for the given task. The Transformer layer is the main algorithmic innovation in Transformer networks; for brevity, we will mainly focus on this layer, although our experimental evaluations are performed on complete networks. Transformer layers consist of a multi-headed attention block followed by a feed-forward block, as shown in Figure 3.4.

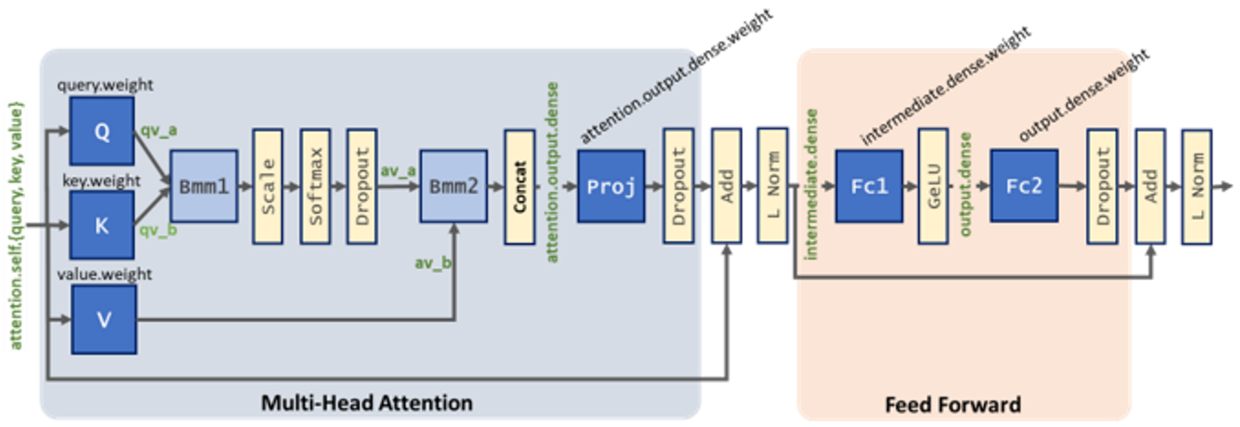


Figure 3.4. A Transformer layer consists of a multi-headed self attention block and a feedforward block. Of particular note is the use of softmax as a crucial operation in self-attention.

The attention block applies three linear transformations to the input vector in order to obtain three new matrices: the query matrix (Q), the key matrix (K), and the value matrix (V). The query matrix and key matrix are then multiplied together and the result is scaled to the number of feature dimensions. This process can be thought of as an analogue to the similarity operation found in generic MANNs, depicted in Equation 3.4. Softmax is then applied to the resultant matrix (this is the weighting operation of MANNs described in Equation 3.5), followed by dropout, resulting in the self-attention matrix A. This matrix is finally multiplied by the value matrix V, as in the soft read of MANNs described in Equation 3.1. These operations can be repeated multiple times in parallel, resulting in multi-headed attention. The results of each head are concatenated together before being passed through multiple fully-connected layers in the feed forward stage. After both the multi-headed attention stage and feed forward stage, there is a set of dropout-add-norm layers.

We note that, in addition to the typical matrix multiplications, *each* Transformer layer also contains less common operations such as softmax and dropout. As shown in Figure 5.1, these non-matrix multiply operations begin to become larger bottlenecks, especially as recent Transformer-based networks have moved towards increasing sequence lengths, with the GPT family of networks moving from a sequence length of 512 to 2048 in the most recent version [87].

3.3.2 Softmax Bottlenecks

Existing hardware architectures for DL, which have been designed based on the characteristics of DNNs such as CNNs, MLPs and LSTMs, have largely neglected the softmax operation. This is because softmax is typically used only as the last layer in these networks in order to generate the final probabilities used in classification tasks, and thus represents only a small fraction of computation time and energy. However, this is no longer true for Transformer networks, which use softmax as a key component of the attention mechanism. For these networks, softmax can become a significant bottleneck, as shown in Figure 5.1. The softmax operation is inefficient in current hardware for two main reasons. First, softmax requires the use of the exponential function. Exponential functions tend to require large look-up table (LUTs) to compute the result through the use of Taylor expansions. This is particularly true for general-purpose hardware such as CPUs and GPGPUs, which cater to exponential computations with high accuracy requirements due to their use in various scientific computing applications. This large area and power overhead makes it difficult to instantiate a large number of these units. Second, in order to improve training stability, deep neural networks typically use a numerically stable softmax, which subtracts the max of the vector on which softmax is being performed in order to ensure that the result does not blow up to infinity. However, this stability comes at a cost, as calculating the max introduces an additional pass through the vector, incurring latency and memory overheads.

3.4 Graph Neural Networks

Graph neural networks (GNNs) have attracted significant interest in the machine learning community, but have not yet been explored in detail by computing system designers. In this section, we will provide the necessary background for understanding graph neural networks, followed by a brief introduction into GNNs themselves. A more complete introduction can be found in recent survey papers [88], [89].

3.4.1 Graph Definition

A graph G is a data structure defined by two unordered sets: its vertices, V, and the edges, E, between those vertices. We will denote the i-th node as $\mathbf{v}_\mathrm{i}$. The directed edge between nodes i and j is defined as $\mathbf{e}_\mathrm{i,j}$, where i is the source node and j is the destination. Therefore, we will refer to the source node of a given edge as $\mathbf{e}_\mathrm{i}$ and the destination node of a given edge as $\mathbf{e}_\mathrm{j}$. Further, each node and edge may have one or more features associated with them, referred to as $\mathbf{h}_\mathrm{i}$ and $\mathbf{h}_\mathrm{i,j}$, respectively.

3.4.2 Graph Sharding

Many real-world graphs are too large to fit into any given level of the memory hierarchy (last-level cache, main memory, etc). This severely degrades performance of graph processing algorithms, as it makes it extremely difficult to exploit the limited spatial locality available. To address this, graphs are often broken into smaller pieces, such that each subgraph can fit in the given level of memory, in a process referred to as *graph sharding* [90]–[92]. Similar to [92], we adopt a two-dimensional sharding paradigm, as depicted in Figure 3.5. In this paradigm, a graph's edge list is divided in shards such that each shard contains a maximum of n^2 edges. A shard is then referred to by the tuple (U, V), such that shard (U, V) contains the edges whose source nodes are in the set $[U \cdot n, (U + 1) \cdot n)$ and whose destination nodes are in the set $[V \cdot n, (V + 1) \cdot n)$.

For the rest of this thesis, we will refer to a sharded graph as graph that has undergone this preprocessing step. Note that in some cases, a graph may actually fit entirely in the given level of the memory hierarchy and thus the sharded graph consists of only one graph. To maintain consistency, we will still refer to this special case as a sharded graph.

3.4.3 Graph Neural Networks

Graph neural networks (GNNs) are a widely diverse family of networks. Many popular GNNs [93]–[97] consist of two distinct stages: a *feature extraction* stage, and an *aggregation* stage. In the feature extraction stage, the nodes' (edges') features are passed through a fully-

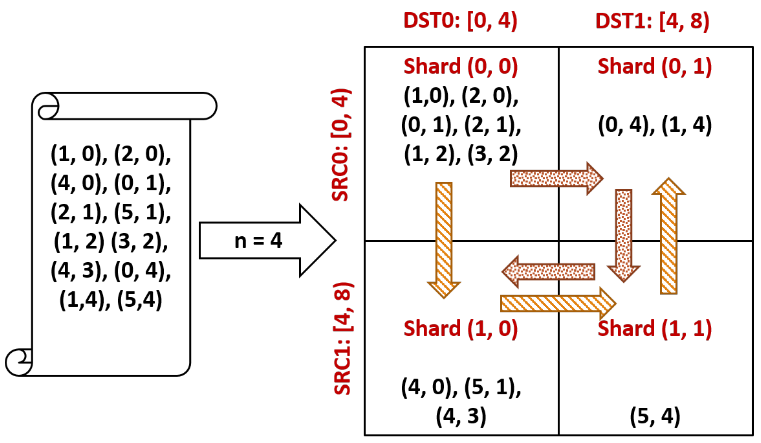


Figure 3.5. Visualization of a two-dimensional graph sharding algorithm that divides an edge list into shards such that Shard (U, V) is the set of all edges such that e.$u \in [U \cdot n, (U + 1) \cdot n)$ and e.$v \in [V \cdot n, (V + 1) \cdot n)$. The resulting subgraphs can then be processed in either a source-stationary (dotted arrow) or destination-stationary (slanted arrow) manner.

connected linear layer. In the aggregation stage, a node aggregates feature vectors from its neighbors and then uses this aggregated feature vector to update its own feature.

These two stages can be combined arbitrarily to make up a single GNN layer. A full GNN can then be constructed by stacking these layers. By stacking layers, a GNN incorporates information from nodes that are increasingly far away from the original node. For example, a single layer GNN only considers a node's neighbors, while a two layer GNN will consider nodes in the two-hop neighborhood. Notice, however, that this results in an exponential increase as the depth increase.

We provide a brief overview of some popular GNNs below in order to further illustrate this concept.

Graph Convolutional Networks. The Graph Convolutional Network (GCN) [98] is an extremely popular early GNN. GCNs apply the local weight sharing found in traditional convolutional neural network to graphs such that every node (edge) feature shares the same weights. As shown in Equation (3.9), GCN first applies an aggregation stage, where the features of a node's neighbors are summed together. Note that the features are also normalized

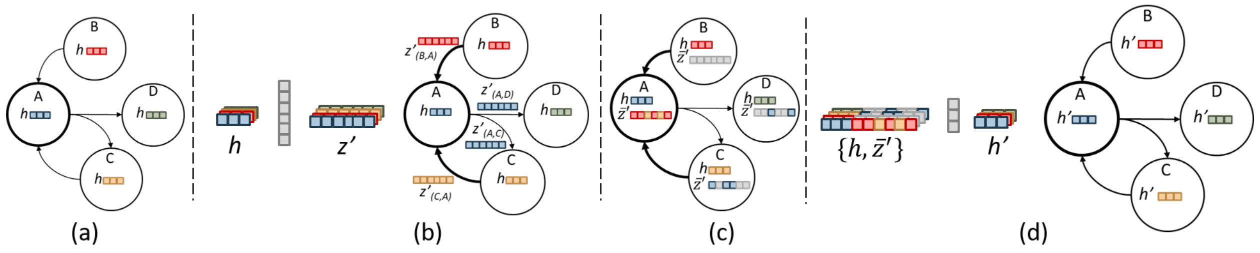


Figure 3.6. Forward pass of one layer in a GraphSagePool network, applied to a sample graph with four nodes (a). For clarity, we focus only on node A. First, in a feature extraction stage, each edge's source node feature h is passed through a fully-connected layer to obtain the feature z (b). Note that the nodes are not concatenated, but rather are analogous to different samples in a minibatch. Next, each node *aggregates* information from its *incoming* edges using a symmetric function such as max to obtain an aggregated feature vector $\bar{z}$ (c). Finally, (d) the aggregated feature and the original node feature are combined and passed through another linear layer to obtain an updated node feature, h. This process can be repeated for multiple GNN layers.

both by the degree of the source node u as well as the destination node v. The aggregated node feature $\bar{z}$ is then passed through a linear layer to obtain the updated node feature, h_u.

$$\begin{gathered}\bar{\mathbf{z}} = \frac{1}{\sqrt{|N(u)|}} \sum \{ \frac{1}{\sqrt{|N(v)|}} \cdot \mathbf{h_v} | \forall v \in N(u) \cup u \} \\ \mathbf{h_u} = \sigma(\mathbf{W} \cdot \bar{\mathbf{z}})\end{gathered} \tag{3.9}$$

Graphsage. The popular Graphsage network [99] is similar to the GCN, but adds a ResNet-like skip connection [100] in the aggregation stage; the node's original feature h_u is combined (concatenation or addition) with the aggregated node feature before being passed through the linear layer. Graphsage commonly uses a mean-based aggregator that is nearly identical to the aggregator used in GCN, as seen in Equation (3.10).

$$\begin{gathered}\bar{\mathbf{z}} = \frac{1}{|N(u)|} \sum \{ \mathbf{h_v} | \forall v \in N(u) \cup u \} \\ \mathbf{h_u} = \sigma(\mathbf{W} \cdot (\bar{\mathbf{z}} \cup \mathbf{h_u}))\end{gathered} \tag{3.10}$$

GraphsagePool. The GraphsagePool variant of Graphsage replaces the mean-based aggregator with a symmetric, trainable aggregator, as in Equation (3.11). Specifically, each node's feature is fed through a linear layer, defined by W_{pool}, as shown in Figure 3.6b. The resulting features are then aggregated using an element-wise pooling operation (typically, max) like in Figure 3.6c. Finally, this aggregated feature is concatenated with original node feature and passed through another linear layer as shown in Figure 3.6d.

$$\begin{gathered}\mathbf{z} = \{\sigma(\mathbf{W_{pool}} \cdot \mathbf{h_v}) | \forall v \in N(u) \cup u\} \\ \bar{\mathbf{z}} = max(\mathbf{z}) \\ \mathbf{h_u} = \sigma(\mathbf{W} \cdot (\bar{\mathbf{z}} \cup \mathbf{h_u}))\end{gathered} \tag{3.11}$$

3.4.4 GNN Computational Behavior

Feature Extraction Stage. The feature extraction stage is usually implemented as a fully-connected layer, in which a vertex (edge)'s feature vector is multiplied by a weight vector, producing a new feature vector– *i.e.*, is a matrix-vector multiplication. However, in order to allow GNNs to generalize to different sized graphs, GNNs typically shared the weight matrix across the vertices (edges), meaning that a batch of feature vectors can be processed together with the same weight matrix, resulting in a matrix-matrix multiplication. Thus, the feature extraction stage is characterized primarily as a dense, regular computation with high data reuse of weights. A compute engine for the feature extraction stage, therefore, should have ample compute, with enough internal bandwidth to feed that compute while taking advantage of the large amount of weight reuse.

Aggregation Stage. The aggregation stage, by contrast, is a highly sparse, irregular kernel, much like many other graph-processing workloads. However, there is one key difference between the aggregation stage and other graph processing kernels. In traditional graph processing applications, the graph data tends to be quite small, on the order of a few bytes. In a GNN's aggregation stage, by contrast, the features being communicated between vertices can be much larger, on the order of up to thousands of bytes. This has two implications. First, the large dimension size of the features compared to a traditional graph analytics workloads means that there is more ample opportunity for *intra*-node parallelism, as the dimensions for a given node can be processed independently in parallel. This is in addition to the *inter*-node parallelism present in both the feature extraction and aggregation stages that results from large amount of nodes that each need to be updated independently in a GNN layer. Secondly, the large dimension size of the features can actually help to reduce the irregular nature of the aggregation stage as compared to other graph-based workloads, as instead of each vertex needed to load one or two bytes per neighbor, each vertex instead must load many bytes, which can result in better coalesced memory accesses. A compute engine for the aggregation stage, therefore, should exploit both the inter- and intra-node parallelism that results from the longer node features, as well as design the memory system around the wider accesses seen in the aggregation stage as compared to other graph accelerators such as Graphicianado [101].

Feature Extraction vs Aggregation Stage. Above, we describe the unique characteristics of the two different kernels present in GNNs, as well as the specific architectural desiderata that results. However, a crucial question is also the relative importance of the two stages, as this would dictate the amount of resources that should be dedicated to them.

To that end, we investigate the relative breakdown in execution time between the aggregation kernels and the feature extraction kernels running on a GNN accelerator design with comparable performance to HyGCN. In particular, we examine this breakdown as a function of the hidden layer's dimension size. As shown in Figure 3.7, the aggregation stages dominate the run time for smaller dimension sizes. This is in line with finding from other GNN accelerators, such as [59], [61]. However, at larger, more production-like networks sizes, this trend reverses and the linear layers dominate the run time. Note that the linear layers always dominates the run time for the GraphSage-Max network; this is a result of the additional linear feature extractor present in the pooling stage. This implies that the needs of the two stages must be carefully balanced, as the bottleneck in GNN processing on a custom architecture depends on the size of the network; smaller networks are bottlenecked by graph processing, whereas larger networks are bottlenecked by fully connected, dense layers.

In summary, a GNN accelerator should be designed with following characteristics in mind. First, in order to efficiently compute the feature extraction stage, the architecture should contain a large compute array for performing matrix-matrix multiplications, optimized for a large amount of weight reuse. Second, the graph engine should be optimized for both the abundant inter- and intra-node parallelism in GNNs as well as the uniquely wide graph

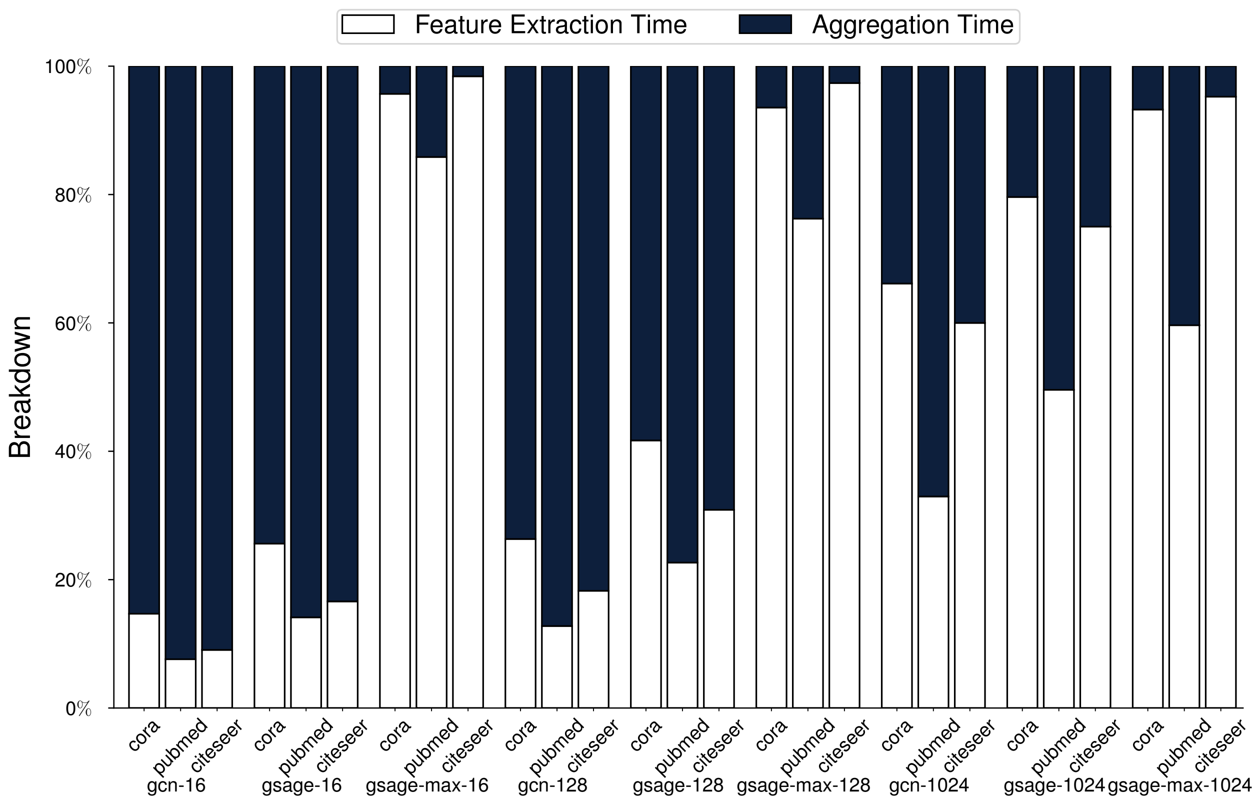


Figure 3.7. At lower hidden dimension sizes, the aggregation stages dominate the run time over the feature extraction stages. However, at the large dimension size of 1024, this trend reverse. Feature extraction always dominates the run time for the GraphSage-Max network, due to the additional linear layer present before aggregation

property accesses present in GNNs compared to other graph workloads. Finally, the needs of the two compute engines must be balanced, as which stage is the bottleneck is dependent on the size of the GNN being processed.

3.5 DNN Performance Estimators for Edge Devices

DNN performance estimators have been a recent area of interest. These estimators provide a low-cost way to search the space of potential neural network candidates– and hardware platforms– for the best combination of software and hardware. This is of particular interest as developers look to deploying DNNs in the real-world using heterogeneous edge devices.

3.5.1 Heterogeneous Edge Devices

There is a vast array of heterogeneous edge devices deployed in the real world today running neural workloads [102]. The processing platforms in these edge devices consist of one or more *hardware IPs*, such as CPUs, GPUs, DSPs and specialized DNN accelerators. For example, the NVIDIA Jetson TX2 platform [103] has a six-core big.LITTLE ARM CPU with two high performance Denver cores and four energy-efficient ARM Cortex cores, as well as a Pascal GPU. Two key design choices exposed by these platforms are heterogeneous execution and hardware operating conditions.

Heterogeneous execution. A DNN may be partitioned such that different portions execute on different IPs within a heterogeneous edge platform. This helps fully utilize the processing resources within the platform. Further, the unique characteristics of different parts of a DNN may benefit from the hardware heterogeneity.

Hardware operating conditions. Most edge platforms support multiple hardware operating conditions, including the number of active CPU cores, the CPU cores' frequencies, the memory controller's frequency, and the GPU's frequency.

As shown in Figure 7.1, the developer of DNNs for edge platforms is presented with a dramatically expanded design space. For example, the Jetson TX2 provides over 500 unique hardware operating conditions, adding two orders of magnitude to the already large search space of DNNs available for a given task. The search space expands even more with heterogeneous mapping; even for the simple case where the network is split into two parts that execute on two different IPs, mapping networks such as MobileNetV3 [104] will further increase the search space by two orders of magnitude.

3.5.2 DNN Performance Estimators

Given the large design space available when deploying DNNs on heterogeneous edge platforms, it is critical to have a fast way to determine metrics of interest (latency, power, energy, *etc.*) for a given DNN, heterogeneous mapping, and hardware operating condition. Unfortunately, even running on actual hardware is too slow to evaluate this large design space. This is in large part due to the overheads involved in the software stack for changing hardware voltage/frequency, loading new models, *etc.* Further, running on physical devices is constrained by the limited number of devices available to run in parallel [105]. Learned performance estimators, which use a machine learning algorithm to generate the estimate, have emerged as promising technique. However, due to the aforementioned constraints, an important consideration in the design of learned estimators is that they should be sample-efficient, i.e., possible to create with limited training data [106].

Learned estimators can be useful in a variety of applications. We focus on three important use cases: hardware mapping and configuration for a fixed network, selection of network variants, and the design of novel networks.

Hardware Mapping and Configuration Search. This corresponds to the scenario wherein a system designer has a specific DNN that they desire to use, and need to search for the most efficient mapping and configuration of a hardware platform to optimize the deployment of that DNN.

Network Variant-Hardware Configuration Co-Design. Network variants come up in a variety of contexts. For example, transfer-learning– in which an off-the-shelf network is adapted for a given task– is frequently used when deploying DNNs. By using hardware-aware performance estimators, designers can also consider the impact of the hardware mapping and operating conditions when searching for the best network variant.

Novel Network-Hardware Configuration Co-Design. This use case includes hardware-aware neural architecture search (NAS), where a system designer is searching for the optimal DNN topology to deploy to an edge device with the optimal hardware configuration. The universe of possible networks is large and it is infeasible to execute each one directly on hardware to search this space.

4. MANNA: AN ACCELERATOR FOR MEMORY-AUGMENTED NEURAL NETWORKS

4.1 Introduction

Deep Neural Networks (DNNs) have kindled a renewed interest in the field of machine learning over the past decade. Algorithmic advances, availability of larger datasets, and tremendous growth in compute capabilities have led to DNNs achieving state-of-the-art accuracy in a wide variety of perceptual domains, including video, audio, and text processing [75], [100], [107]. Despite the success and widespread deployment of DNNs, many machine learning tasks such as task planning, reasoning *etc.*, as well as one-shot learning (the ability to learn from one or very few examples), remain open challenges for them.

Memory-augmented neural networks (MANNs) promise to address some of these challenges by providing the network with access to dynamic (readable and writeable) state [77], [79], [81], [82]. Unlike recurrent neural networks (RNNs), wherein the dynamic state is intrinsically embodied within the topology of the network itself, in MANNs the dynamic state is decoupled from the neural network. This decoupled dynamic state is realized through a *differentiable external memory* that is accessed by read and write heads using *soft reads and writes*. A soft read is a read that is comprised of a weighted sum over all of the memory locations in the external memory. Similarly, a soft write updates every element in the external memory, with the update to a given location being proportional to that location's similarity to a key vector. Crucially, these soft read and write operations are continuous (in contrast to the discrete nature of reads and writes to single locations), allowing the external memory to be differentiated and the entire MANN to be trained end-to-end with stochastic gradient-descent algorithms. Several variants of MANNs have been proposed, such as the Neural Turing Machine (NTM) [81], Differentiable Neural Computer (DNC) [82], Dynamic Memory Networks [79], and End-to-End Memory Networks [77]. These recent efforts from Google DeepMind, Facebook AI Research, and others have demonstrated the ability of MANNs to solve entirely new classes of problems that are well beyond the capabilities of classical DNNs. For example, DeepMind's Differentiable Neural Computer was trained to efficiently navigate the complex routes that make up the London Underground subway network [82].

The enhanced capabilities of MANNs come at a high computational cost. As mentioned above, the soft nature of the reads and writes results in accesses to *all the memory locations* for each operation in order to keep these operations differentiable. This introduces a serious performance bottleneck, which is exacerbated when dealing with real-world problems requiring thousands to millions of memory locations [82].

Existing neural network accelerators are ill-suited to address this bottleneck for three main reasons. First, existing neural network accelerators are designed primarily for (i) CNNs, which have high FLOPs/Byte ratios and are compute-bound, enabling the use of large arrays of multiply-and-accumulate (MAC) units, or (ii) fully-connected multilayer perceptrons (MLPs) and RNNs, which are memory-bound but whose FLOPs/Byte ratio can be increased considerably by batching inputs and reusing weights across the inputs in a batch.

However, this paradigm of designing for compute-bound networks (*i.e.*, high FLOPs/Byte ratios) and using batching to improve the ratios even for memory-bound networks *cannot be applied to MANNs.* The external memory in MANNs represents dynamic state akin to the activations in a traditional DNN and is unique to each input. Therefore, it cannot be shared across a batch, unlike the weights of an MLP or RNN. Thus, an accelerator designed for compute-bound networks, allocating large amounts of die area to compute units, is highly inefficient for memory-bound MANNs. Instead, a MANN accelerator needs to be memory-centric, focusing not on high theoretical throughput but rather on maintaining high utilization of the available compute units through highly banked on-chip memories.

Second, many existing neural network accelerators are also ill-equipped to cater to the heterogeneity of memory access patterns found in MANNs. During inference in DNNs, there is only one access pattern per data structure – for example, a set of activations may be accessed, but not the transpose of those activations. During inference for MANNs, however, the external memory is accessed in both a column-wise as well as a row-wise pattern–that is, both the external memory and the transpose of the external memory are required. Further, the external memory is quite large and can be updated on every time step, making it impractical to efficiently maintain two copies. Therefore, a MANN accelerator must be able to perform efficient transpose operations on-chip.

Finally, all existing neural network accelerators are designed for efficiently executing dot product operations, as matrix-multiply kernels dominate the runtime of current DNNs, including CNNs, MLPs, and RNNs. In MANNs, however, MAC operations are not the dominant operation; non-reductive element-wise addition, subtraction, and multiplication operations make up a roughly equal portion of the operations in MANNs. As a result, a MANN accelerator must be designed to perform non-reductive element-wise operations equally as efficiently as dot products.

To address these shortcomings of existing neural network accelerators, we propose `Manna`, an accelerator developed from the ground up to efficiently execute MANNs. `Manna` is a memory-centric design that focuses on maximizing on-chip storage and bandwidth and uses just enough compute to match that bandwidth in order to eschew the die area and power wasted by underutilized compute elements. `Manna` also utilizes a hardware-based transpose mechanism to efficiently perform the row-wise and column-wise memory accesses required. Finally, `Manna` is provisioned with an array of processing tiles, each of which are composed of specialized units (called eMACs) designed to efficiently perform the unique mix of operations present in MANNs.

We also propose a compiler that efficiently maps kernels to `Manna`, exploiting the little available reuse in order to reduce memory transfers and maximize bandwidth utilization.

In summary, we make the following contributions:

- We provide a detailed investigation of the computational characteristics of MANNs, a promising emerging class of DNNs.
- We propose `Manna`, an end-to-end, CMOS-based accelerator that is able to efficiently perform all kernels used in memory-augmented neural networks. We provision `Manna` with a memory-to-compute resource allocation reflective of the low FLOPs/Byte inherent to MANNs.
- We implement a hardware-based transpose mechanism to efficiently execute both vector-matrix and vector-transposed matrix multiplications.

- We provision Manna with specialized units (called eMACs), which are designed for the unique mix of operations that are used in MANNs.
- We develop an ISA and a compiler that maps MANNs to Manna so as to minimize data transfers and maximize the utilization of on-chip bandwidth.
- We develop an architectural simulator and synthesize RTL implementations of key components in order to demonstrate the benefits of Manna. Our experiments indicate that Manna achieves average speedups of 39x (24x) and average energy improvements of 122x (86x) over an NVIDIA 1080-Ti (2080-Ti) GPU with a Pascal (Turing) architecture.

The rest of the chapter is organized as follows. Section 4.2 describes the proposed Manna architecture and provides insights into how various kernels are executed on Manna. Section 4.3 details the ISA as well as the proposed compiler. Section 4.4 outlines the experimental setup used to evaluate the proposed architecture. Finally, Section 4.5 presents the results of our experiments and Section 4.6 concludes the chapter.

4.2 Manna Architecture

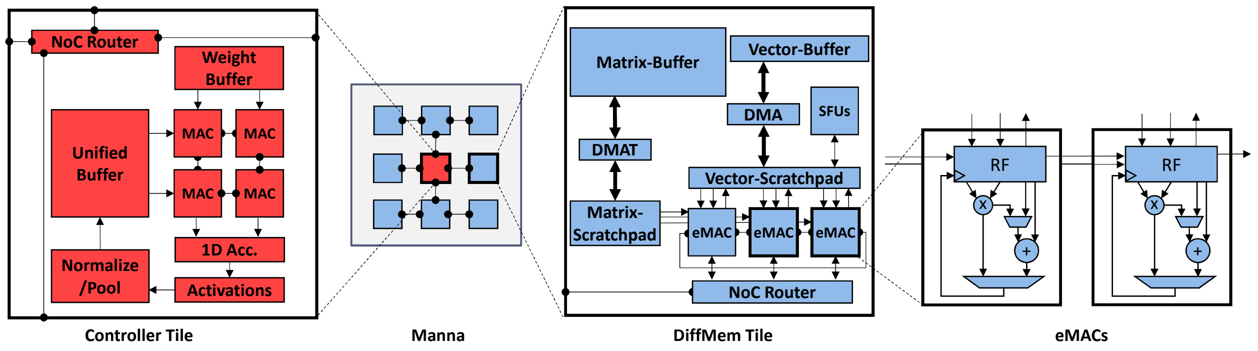


Figure 4.1. Overview of Manna

4.2.1 Overview

Manna is a memory-centric, highly parallel CMOS-based architecture designed explicitly for memory-augmented neural networks. Figure 4.1 shows the organization of Manna, whose components are described in the following subsections. The building blocks of MANNA are *DiffMem tiles* that execute the MANN-specific kernels (*i.e.*, read and write head operations, key similarity, addressing, soft read, and soft write) which control the external differentiable memory (hence, the name DiffMem tiles). The entire external memory is partitioned and distributed across the DiffMem tiles. DiffMem tiles are designed to cater to the low FLOPS/Byte ratio observed in MANN kernels. As a result, the majority of die area of the DiffMem Tiles (and MANNA in general) is dedicated to highly banked on-chip memories. This is in dramatic contrast with other modern architectures used to execute classical DNNs, as seen in Table 4.1. The DiffMem tiles are then provisioned with just enough processing elements to match that on-chip memory bandwidth. We also specialize MANNA's processing elements to be tailored to the unique mix of operations found in MANN kernels, which exhibit a higher share of non-MAC element-wise operations than the traditional MAC-centric DNN workloads. Finally, we provision MANNA with specialized hardware to facilitate efficient transpose operations to accommodate the heterogeneity in memory access patterns seen in MANNs.

MANNA also includes *Controller tiles*, which are used to execute the DNN-based controler. We tailor the interconnect topology between tiles to the specific communication patterns that manifest in MANNs.

	Xeon[108]	**2080-Ti**	**TPU** [8]	**Manna**
Memory	25	10	60	90
Compute	75	90	40	10

Table 4.1. Manna prioritizes high bandwidth on-chip memory over compute when allocating die area

4.2.2 DiffMem Tiles

DiffMem tiles are used to perform the MANN-specific kernels: read (write) heads, key similarity, addressing, soft reads, and soft writes. Each tile is provisioned with element-wise or multiply-and-accumulate units (eMACs). These eMACs are used to perform element-wise addition, element-wise subtraction, element-wise multiplication, and fused multiply-

and-accumulate. Each eMAC also consists of a small register file (RF) that is used to temporarily store inputs and intermediate outputs. The eMACs are connected to a double-buffered scratchpad (`Matrix-Scratchpad`) such that each word of the memory's output is directly connected to the corresponding eMAC unit. The eMACs are also connected to another double-buffered scratchpad (`Vector-Scratchpad`). The `Vector-Scratchpad` either broadcasts a shared value to all eMAC units or unicasts each word of the scratchpad's output to the corresponding eMAC unit. Additionally, there are lateral connections between neighboring eMACs, which enable efficient transpose operations. The `Matrix-Scratchpad` is connected to a larger matrix buffer (`Matrix-Buffer`) through a direct memory access for transpose (`DMAT`) unit that manages data transfers in both regular and transposed form, between the `Matrix-Scratchpad` and the `Matrix-Buffer`. The `Vector-Scratchpad` is similarly connected to a larger vector buffer (`Vector-Buffer`). However, since these memory elements store the vector data structures which do not require any hardware transpose support, a regular direct memory access (DMA) unit is used. The `Vector-Scratchpad` is also interfaced to a set of Special Function Units (SFUs). The SFUs contain a scalar power function unit, an accumulator, and the logic needed to support various activation functions such as sigmoid, ReLu, *etc.* Each DiffMem tile also has a small instruction memory and control unit used to execute the tile's program. Finally, it also includes an NoC router to perform reduce and broadcast operations between tiles.

4.2.3 Controller Tiles

The Controller tile is used to execute the DNN controller. Since a wide variety of DNN topologies may be used for the controller, the Controller tile utilizes a more traditional, systolic array-based DNN accelerator architecture. A controller tile consists of a two-dimensional systolic array-based matrix multiplication unit. The matrix multiplication unit is connected to a `Weight Buffer` that supplies the weights, and a `Unified Buffer` that provides the activations. These activations are then accumulated in a one dimensional `Accumulation Unit`, the output of which is connected to the activation units (*e.g.*, ReLu), followed by the normalization and pooling units. The outputs are finally stored in the `Unified Buffer` to be used as inputs to the next layer.

4.2.4 Implementing MANNs on Manna

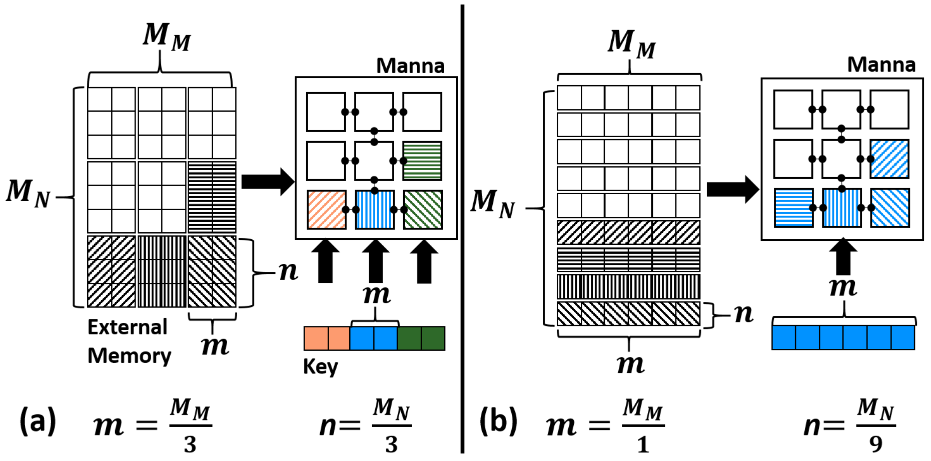
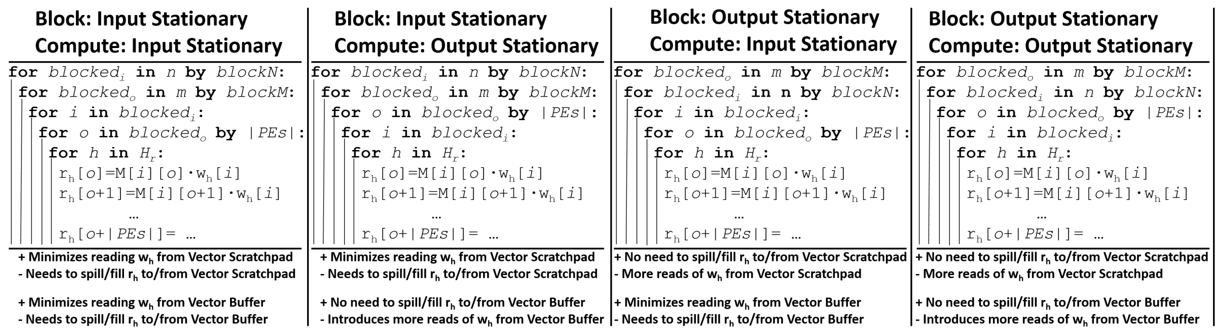


Figure 4.2. Each DiffMem tile receives a unique, independent section of the external memory. If $MDistrib \neq NumTiles$, as in (a) where $MDistrib = 3$, there will be $MDistrib$ sets of tiles in which each tile in the set receive copies of the same slice of the key vector. If $MDistrib = 1$, as in (b), every tile will receive a copy of the whole key vector

Distributing MANN kernels. A key feature of Manna is that the entire differentiable external memory is partitioned and distributed across the DiffMem tiles. This is achieved by dividing the differentiable memory into independent sections by selecting the number of tiles ($NDistrib$) to distribute the rows across, which results in each tile receiving $n = \frac{M_N}{NDistrib}$ rows of $\mathbf{M}$. The number of tiles to distribute the columns across is then $MDistrib = \frac{NumTiles}{NDistrib}$, resulting in each tile receiving $m = \frac{M_M}{MDistrib}$ columns of $\mathbf{M}$. Therefore, each physical tile is responsible for a logical (n,m) tile of the differentiable external memory. The tile is also responsible for the corresponding slices of the various vectors (*e.g.*, key). Figure 4.2 illustrates this distribution.

The choice of $NDistrib$ and $MDistrib$ affects the performance of the Manna accelerator, since it impacts the communication patterns and degree of parallelization that can be achieved in executing the MANN kernels. For example, consider the content weighting kernel



Block: Input Stationary
Compute: Input Stationary

```
for blocked_i in n by blockN:
  for blocked_o in m by blockM:
    for i in blocked_i:
      for o in blocked_o by |PEs|:
        for h in H_r:
          r_h[o]=M[i][o]·w_h[i]
          r_h[o+1]=M[i][o+1]·w_h[i]
                ...
          r_h[o+|PEs|]= ...
```

+ Minimizes reading w_h from Vector Scratchpad
- Needs to spill/fill r_h to/from Vector Scratchpad

+ Minimizes reading w_h from Vector Buffer
- Needs to spill/fill r_h to/from Vector Buffer

Block: Input Stationary
Compute: Output Stationary

```
for blocked_i in n by blockN:
  for blocked_o in m by blockM:
    for o in blocked_o by |PEs|:
      for i in blocked_i:
        for h in H_r:
          r_h[o]=M[i][o]·w_h[i]
          r_h[o+1]=M[i][o+1]·w_h[i]
                ...
          r_h[o+|PEs|]= ...
```

+ Minimizes reading w_h from Vector Scratchpad
- Needs to spill/fill r_h to/from Vector Scratchpad

+ No need to spill/fill r_h to/from Vector Buffer
- Introduces more reads of w_h from Vector Buffer

Block: Output Stationary
Compute: Input Stationary

```
for blocked_o in m by blockM:
  for blocked_i in n by blockN:
    for i in blocked_i:
      for o in blocked_o by |PEs|:
        for h in H_r:
          r_h[o]=M[i][o]·w_h[i]
          r_h[o+1]=M[i][o+1]·w_h[i]
                ...
          r_h[o+|PEs|]= ...
```

+ No need to spill/fill r_h to/from Vector Scratchpad
- More reads of w_h from Vector Scratchpad

+ Minimizes reading w_h from Vector Buffer
- Needs to spill/fill r_h to/from Vector Buffer

Block: Output Stationary
Compute: Output Stationary

```
for blocked_o in m by blockM:
  for blocked_i in n by blockN:
    for o in blocked_o by |PEs|:
      for i in blocked_i:
        for h in H_r:
          r_h[o]=M[i][o]·w_h[i]
          r_h[o+1]=M[i][o+1]·w_h[i]
                ...
          r_h[o+|PEs|]= ...
```

+ No need to spill/fill r_h to/from Vector Scratchpad
- More reads of w_h from Vector Scratchpad

+ No need to spill/fill r_h to/from Vector Buffer
- Introduces more reads of w_h from Vector Buffer

Figure 4.3. Overview of possible loop orderings for soft read

which is a softmax operation. In order to obtain the content weights, the content similarities must be normalized. To achieve this, the total sum of the similarities must first be reduced across the $NDistrib$ tiles that partition the weight vector. The final reduced result must then be broadcast back to those tiles so that the final result can be used in a map operation (as described below) to normalize each element in the weight vector. Notice that Equation 3.5 that describes the content weighting– as with all of the other addressing kernels– is only dependent on M_N. Therefore, if $MDistrib > 1$, there will be a subset of tiles for which no useful computation can be performed. This is due to the fact multiple tiles would have the same slice of the weight vector. To maximize the amount of work that can be done in parallel, thereby maintaining high utilization of the available compute resources, we use the simple heuristic that we will force $MDistrib = 1$ and therefore $NDistrib = NumTiles$.

Realizing Map Operations. Many of the operations that make up the addressing kernels can be envisioned as map operations applied to a given vector. For example, as part of the content weighting kernel, every element in the $\mathbf{w}$ vector is multiplied by the same scalar β. These kernels are executed as follows. First, the value that is to be mapped to the vector (*e.g.*, the accumulated value for normalizing) is broadcast to the eMACs and stored in their RFs. Subsequently, for each element in the vector that the tile is processing, the values are unicast to the eMAC units such that a different element of the vector is computed in each eMAC unit. In this way, all eMACs in each tile can be fully utilized for executing map operations in parallel, with minimal overhead.

Realizing Vector-Matrix Multiplication. As discussed above, each tile executes a portion of the vector-matrix multiplication in parallel. We store the tile's entire portion of the differentiable memory in a large, highly-banked on-chip memory, the `Matrix-Buffer`. Vectors are stored in a comparatively much smaller on-chip memory, the `Vector-Buffer`. We further break the smaller vector-matrix multiplications into blocks, such that each block consists of a vector-multiplication between *blockN* elements of the vector and (*blockN* x *blockM*) elements of the matrix, resulting in an output partial sum vector of length *blockM*. The portion of the vector and matrix needed for the current block is then brought into the `Vector-Scratchpad` and the `Matrix-Scratchpad`, respectively. To compute the vector-matrix multiplications of a given block, data is fed from the two scratchpads into the eMAC units such that one element from the `Vector-Scratchpad` is broadcast to all of the eMAC units. Unlike the case of map operations, for vector-matrix multiplication, each eMAC unit receives a unique memory word from the `Matrix-Scratchpad`, *i.e.*, vector-matrix multiplication is realized by multiplying one element of the input vector with multiple elements from the matrix to compute a set of partial sums. The eMACs then read the next element from the vector, continuing to hold the partial sums resident in the RF. This strategy is referred to as *output stationary*, since a subset of the output vector of the vector-matrix multiplication is completely finished before moving on to the next subset. This strategy does not require storage for partial sums in the `Vector-Scratchpad`, but may result in re-reading the same element from the input vector multiple times if the number of eMAC units is lower than the block size. Alternatively, the element from the input vector can be kept resident and the partial sums spilled to the `Vector-Scratchpad`. This strategy is referred to as *input stationary* and results in minimizing the number of reads of the input vector, but introduces spills (fills) to (from) the `Vector-Scratchpad` since the partial sums are not kept stationary. Note that, while the choice between input stationary and output stationary impacts the number of transfers between the `Vector-Scratchpad` and the eMACs, the total number of transfers between the `Matrix-Scratchpad` and the eMACs remains the same in both cases. In order to maximize reuse (*e.g.*, in the case of multiple heads), the RF is provisioned with enough capacity to allow the eMACs to store the temporary value (input or output) from multiple heads, allowing the eMAC to reuse the element from the `Matrix-Scratchpad` for each head without needing to refetch the input (output) element when moving to the next element from the `Matrix-Scratchpad`.

This loop, which computes the vector-matrix multiplication for a given block, is referred to as the compute loop. Once the multiplication is finished for a given block, we must determine the ordering of the block loop– that is, which block of the vector-multiplication will be executed next. Similar to the compute loop, this can be done in an output-stationary or input-stationary manner. The choice between output-stationary and input stationary determines the number of data transfers between the `Vector-Scratchpad` and the `Vector-Buffer`. The number of transfers between the `Matrix-Scratchpad` and the `Matrix-Buffer` is invariant, since each element in the `Matrix-Buffer` is brought into the `Matrix-Scratchpad` exactly once. Figure 4.3 summarizes the algorithms that result from these different loop orderings and their impact on data transfers.

After each tile has completed computing its portion of the larger vector-matrix multiplication, these partial sums must be reduced across the $NDistrib$ tiles in order to obtain the final result (*e.g.*, the final read vector).

Realizing Vector-Transposed Matrix Multiplication. As discussed, the differentiable memory is accessed in both row and column directions. Unfortunately, due to the size of the external memory, as well as the frequency of updates to the memory, it is infeasible to keep an updated copy of both the external memory and its transpose. This means that it is impossible to map the external memory to the physical `Matrix-Buffer` in such as manner as to result in efficient memory accesses for both soft read and key similarity, as illustrated in Figure 4.4. Therefore, to maintain high utilization of the on-chip bandwidth, we propose an efficient, hardware-based transpose mechanism, outlined in Figure 4.5.

The proposed transpose mechanism draws inspiration from a well-known software technique in the GPGPU domain [109]. This technique relies on splitting the vector-matrix multiplication into blocks, as outlined above. We first fill the `Matrix-Scratchpad` with the entire working set for the current block using efficient reads that utilize the full bandwidth between the `Matrix-Scratchpad` and the `Matrix-Buffer` ❶. Once the `Matrix-Scratchpad` is efficiently filled with a block, we begin computing on that block. Notice that since the

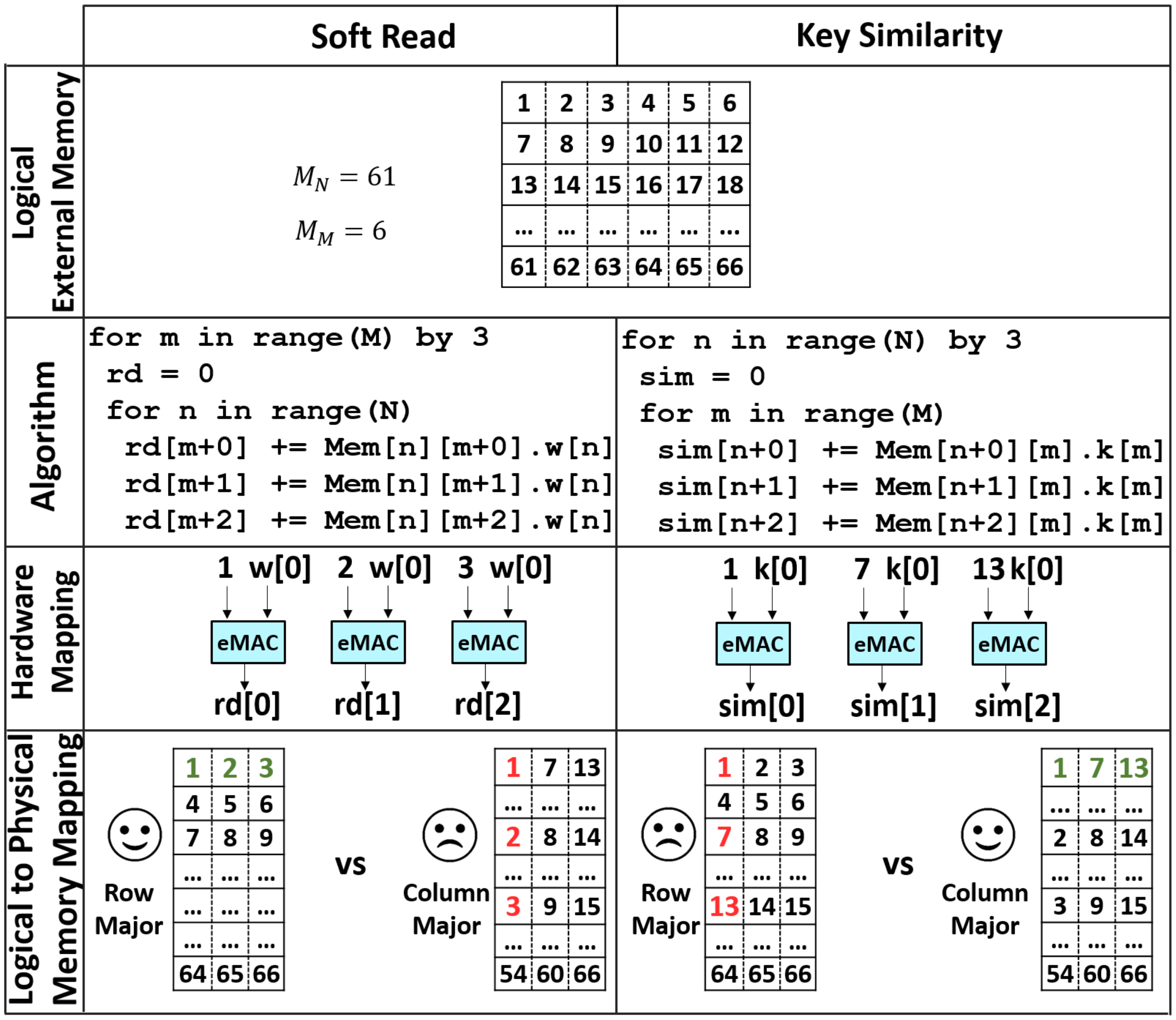


Figure 4.4. Logical-to-physical mapping of the differentiable memory always results in good memory accesses for one kernel, but poor memory accesses for the other

`Matrix-Scratchpad` is double buffered, this sequence can be pipelined, filling the other half of the scratchpad while the first half is being consumed ❷.

This approach fully utilizes the bandwidth between the `Matrix-Buffer` and the `Matrix-Scratchpad`, but there still exists a key inefficiency– there would be many bank conflicts when the eMACs try to read from the `Matrix-Scratchpad` since we have not done anything to fundamentally change the layout of the data yet. To avoid these conflicts, the DMAT unit augments the memory transfers between the `Matrix-Buffer` and the `Matrix-Scratchpad` such that a space is padded between every memory transfer ❸. Doing so offsets the elements

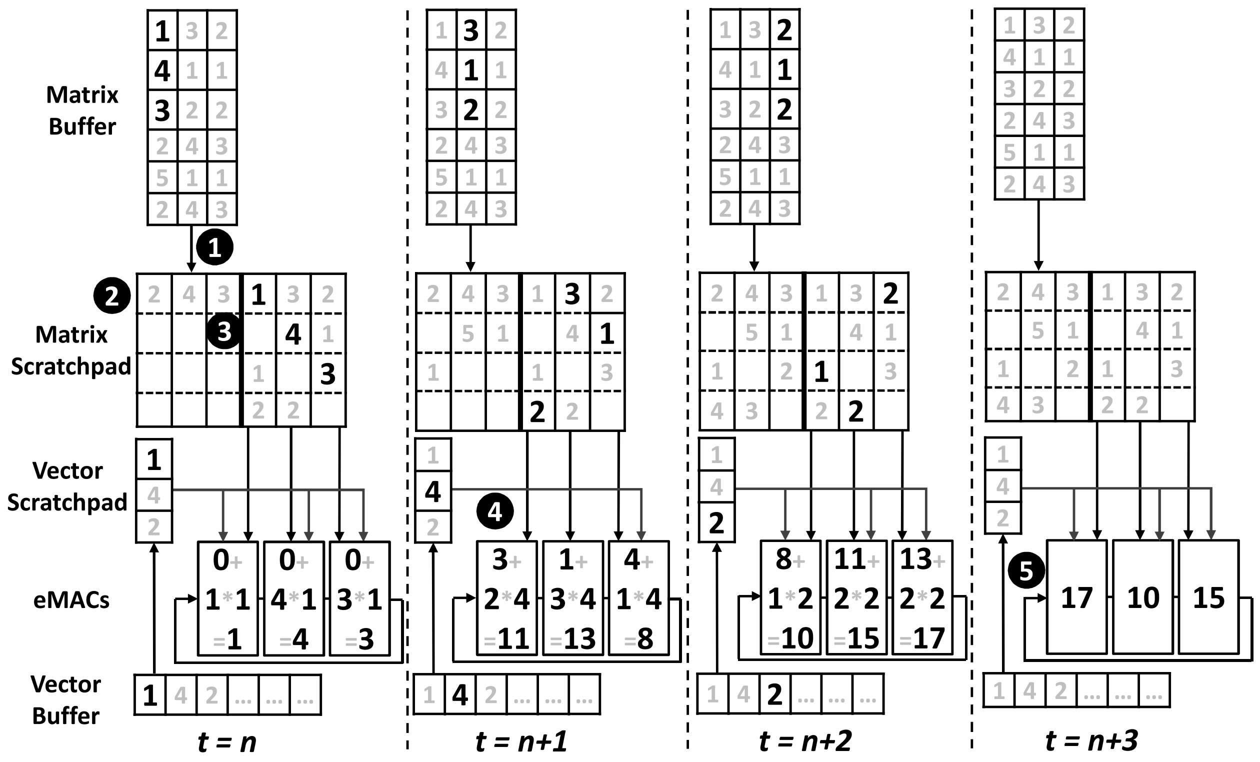


Figure 4.5. Padding the transfer from buffer to scratchpad and moving the partial sums laterally between eMACs allows for conflict-free direct access to banks

in the `Matrix-Scratchpad` such that all elements to be read from the scratchpad at a given time are located in different banks, eliminating bank conflicts.

However, now the eMACs would require a full crossbar structure to access the `Matrix-Scratchpad`, instead of direct connections to each bank as before. This is undesirable, as the crossbar will be expensive in terms of chip area and we want to conserve that area for enabling more on-chip bandwidth. In order to avoid requiring this full crossbar, we utilize lateral connections between the eMACs that forward the partial product from one eMAC to its neighbor 4. This way, each eMAC can still be directly connected to just one bank of the `Matrix-Scratchpad`, with the partial sums shifting to keep the proper alignment. When the product is complete, an additional shift is required to realign the partial sums to their original location 5. However, this additional shift can be pipelined with the beginning of the next set of elements.

NoC Design. Given that we have set $MDistrib = 1$ and $NDistrib = NumberofTiles$, only two inter-tile communication patterns are required: reducing across all of the tiles (*e.g.*, to produce the final read vector) or broadcasting to all the tiles (*e.g.*, the output from the controller). Since these are the only patterns required, we implement a simple, H-tree based NoC with a fixed routing strategy, in turn simplifying the design and requiring only $lg(NumTiles)$ communication steps to complete a given reduction (broadcast). Since the output of the controller network must be broadcast to all the tiles and the controller network must receive the final soft read vectors r_h, the controller tile is chosen as the root node for the H-tree network.

4.3 Manna Software

Given the abundance of literature on ISAs for DNN accelerators and on mapping arbitrary DNNs to target accelerators, here we focus solely on the Diff Mem Tiles for brevity.

4.3.1 Execution Model and ISA

To provide the flexibility to perform a wide variety of MANNs, `Manna` provides an instruction set architecture (ISA) that can be used to implement various proposed MANNs. These instructions are grouped into three categories:

Control. MANN kernels consist entirely of data flows that are easy to generate procedurally. Therefore, we use a set of `control` instructions in order to provide `Manna` with the necessary information to generate the memory accesses needed for a given kernel. `loop` and `end-loop` are used to define the block loop, while `addr-gen` defines the compute loop. The end of the compute loop is inferred from the end of `compute` instructions.

Compute. To enable the execution of a wide variety of MANNs, the key computational primitives of the MANN kernels such as element-wise multiply, vector-matrix multiplication, softmax, *etc.* are expressed as a set of course-grained `compute` instructions. These instructions perform their operations based on the previous `control` instructions.

Communication. Given the execution model outlined in Section 4.2, the communication patterns used in MANNA are extremely well defined and consist solely of reduce and broadcast operations that involve all tiles. Therefore, we provide only two communication instructions– `reduce` and `broadcast`. Since the communication patterns are fixed, these instructions double as synchronization primitives–*i.e.*, when a tile executes a `reduce` operation, the tile will wait until it receives the appropriate message from its neighbor, thereby acting as a fence instruction.

Inst. Type	**Instruction**	**Function**
control	`loop` R_{id}, R_{parent}, R_{len}, R_{stride}	Define a block loop
	`addr_gen` R_{id}, R_{parent}, R_{stride}, R_{base}, `transp`	Define an iterator over a data structure
compute	`el-sub` R_{dst}, R_{src1}, R_{src2}	Elementwise subtraction of two vectors
	`el-mul` R_{dst}, R_{src1}, R_{src2}	Elementwise multiplication of two vectors
communication	`reduce` R_{dst}, R_{src}, R_{addr}, R_{len}	Wait on *src* tile, reduce, send to *dst*

Table 4.2. Summary of ISA.

Table 4.2 provides a summary of MANNA's ISA. To minimize programmer burden, we also provide a compiler that maps a MANN to MANNA, which we detail below.

4.3.2 Compiler

The MANN compiler takes as inputs a description of the target MANN (*e.g.*, number of read/write heads, size of external memory) and a microarchitectural description of the MANNA accelerator (*e.g.*, size of scratchpads, number of tiles). Using this information, the compiler then generates code for MANNA in two phases– mapping and code generation. This process is summarized inFigure 4.6.

Code Mapping

First, a given MANN network is mapped to MANNA by setting the tile size for loop tiling and choosing the ordering of the loops based on the set tile size.

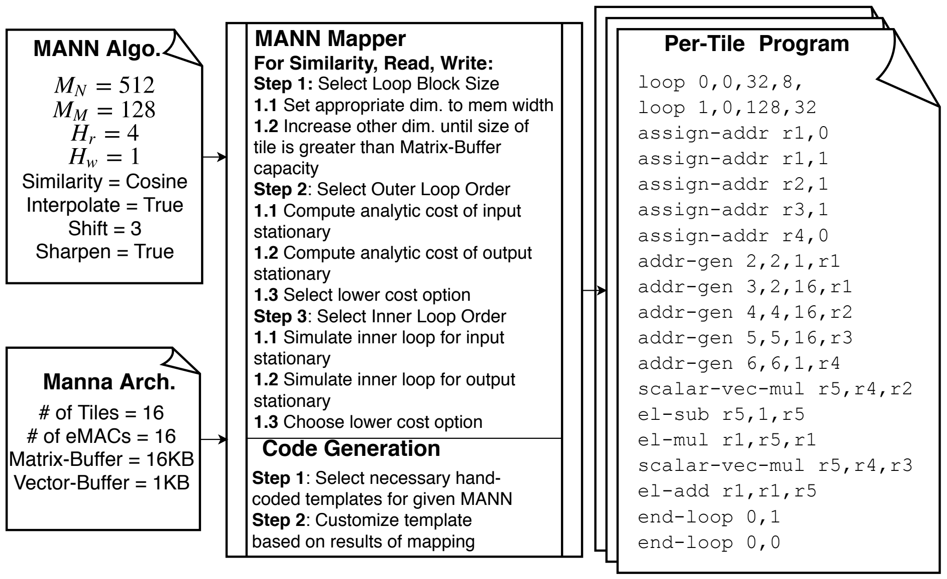
Loop Blocking. As discussed previously, the key similarity, soft read, and soft write kernels are executed in a blocked fashion. A key parameter for optimization, thus, is the size of the tiles for loop blocking. Since the main goal of `MANNA` is to maximize the utilization of on-chip memory bandwidth, we use the following algorithm for setting the loop tile size for each kernel individually. First, we set the M dimension of the block ($blockM$) to match the Matrix Buffer's memory width in order to maximize bandwidth utilization. This is also required by the proposed transpose mechanism used to ensure bank-conflict free execution. We then maximize the N dimension ($blockN$), while still ensuring that the blocked tile can fit in the `Matrix-Scratchpad`. We also account for the additional storage overhead introduced as a result of padding when computing this dimension.

Loop Ordering. As discussed in Section 4.2, for matrix-vector operations, there are two sets of pairs of loops that must be ordered. The order of the outer, block loop determines the number of accesses to the scratchpad, while the order of the inner, compute loop determines the number of accesses to the buffers. Figure 4.3 summarizes these loop orderings along with the advantages and disadvantages of each ordering. In order to select between the four options, the compiler firsts sets the ordering of the block loop, as the accesses to the scratchpad are more expensive and are therefore of a higher priority to optimize. The compiler uses a set of equations to analytically model the cost of the output stationary and input stationary orderings of the block loop and selects the ordering with the lowest cost for each kernel.

Upon determining the outer loop ordering, the compiler next chooses the best ordering of the compute loop for each kernel, choosing the ordering that results in the smallest number of accesses to the buffers.

Code Generation

The code generation phase creates a program for each individual tile. To generate these programs, the compiler utilizes a library of hand-coded assembly routines for each of the MANN kernels. These routines are parameterized based on both the results of the mapping



MANN Algo.
$M_N = 512$
$M_M = 128$
$H_r = 4$
$H_w = 1$
Similarity = Cosine
Interpolate = True
Shift = 3
Sharpen = True
Manna Arch.
# of Tiles = 16
# of eMACs = 16
Matrix-Buffer = 16KB
Vector-Buffer = 1KB
MANN Mapper
For Similarity, Read, Write:
Step 1: Select Loop Block Size
1.1 Set appropriate dim. to mem width
1.2 Increase other dim. until size of tile is greater than Matrix-Buffer capacity
Step 2: Select Outer Loop Order
1.1 Compute analytic cost of input stationary
1.2 Compute analytic cost of output stationary
1.3 Select lower cost option
Step 3: Select Inner Loop Order
1.1 Simulate inner loop for input stationary
1.2 Simulate inner loop for output stationary
1.3 Choose lower cost option
Code Generation
Step 1: Select necessary hand-coded templates for given MANN
Step 2: Customize template based on results of mapping
Per-Tile Program

```
loop 0,0,32,8,
loop 1,0,128,32
assign-addr r1,0
assign-addr r1,1
assign-addr r2,1
assign-addr r3,1
assign-addr r4,0
addr-gen 2,2,1,r1
addr-gen 3,2,16,r1
addr-gen 4,4,16,r2
addr-gen 5,5,16,r3
addr-gen 6,6,1,r4
scalar-vec-mul r5,r4,r2
el-sub r5,1,r5
el-mul r1,r5,r1
scalar-vec-mul r5,r4,r3
el-add r1,r1,r5
end-loop 0,1
end-loop 0,0
```

Figure 4.6. Overview of Phases of `Manna` Compiler

phase and the microarchitectural parameters. The synchronization between tiles is handled via the communication instructions described above. Notice that the output of some kernels are consumed by kernels that use orthogonal resources. For example, the output of the key similarity kernel (Sim) is followed by the softmax of the content-weighting kernel. Since the key similarity kernel is computed using the eMACs, while the softmax is computed using the SFUs, we can begin executing the softmax early, as soon as the outputs of the eMACs are available. Therefore, some of these kernel templates are actually realized as fused kernels.

4.4 Experimental Setup

Simulation infrastructure. We developed a detailed, cycle-level architectural simulator in order to evaluate the execution of the benchmarks on `MANNA`. The simulator models all necessary events that occur in an execution cycle, including compute, memory, and NoC transactions. To simulate the Controller tile, we used the performance simulator from [20], which has been verified against an RTL implementation. To estimate power, we implemented the logic components of `MANNA` in RTL, synthesized them using the 15 `nm` Nangate Open Cell library, and evaluated their power using Synopsys Design Compiler. For the on-chip SRAM memories, we obtained power estimates using CACTI-P [110]. The power consumed by each

Table 4.3. Summary of benchmarks.

Benchmark	DiffMem Dim.	Controller Dim.	# Read/Write Heads
copy	1024x256	1x100	1/1
repeat-copy	512x512	1x100	1/1
recall	1024x64	1x100	1/1
dynamic n-grams	1024x128	1x100	1/1
priority sort	512x128	2x100	1/4
bAbI	4096x1024	1x256	4/1
shortest path	3648x1400	2x256	5/1
graph traversal	5056x1000	3x256	5/1
graph inference	3584x1400	3x256	5/1
mini-shrdlu	1280x4000	2x256	3/1

Table 4.4. Summary of platforms.

Platform	Area (mm^2)	Tech. Node (nm)	Freq. (MHz)	TDP (W)	Memory (MiB)	BW (GB/s)
Pascal GTX 1080-Ti	470	16	1480	250	11.9	484
Turing RTX 2080-Ti	750	12	1500	250	29.5	616
MANNA	40	15	500	16	38	N/A

component was then incorporated into the cycle-level simulators in order to estimate energy consumption.

Benchmarks. Table 4.3 shows the list of 10 benchmarks modeled on the tasks presented in the NTM and DNC papers [81], [82]. These NTM benchmarks use a diverse range of sizes and aspect ratios for the differential memory, with varying numbers of read and write heads. `copy`, `rptcopy`, `recall`, `ngram`, and `sort` are small, algorithmic examples. `bAbI` is a question answering task testing logical reasoning. `travers`, `short`, and `inf` are graph tasks that test the MANN's ability to generate directions, find shortest paths, and infer relationships from structured information, respectively. Finally, `shrdlu` tests a network's ability to understand natural language through the form of dialogues about the state of a synthetic block world. These benchmarks have been scaled up from the original works in order to reflect the size of the external memory needed for real-world applications, as projected in the DNC paper [82].

Manna Configuration. We evaluated a MANNA implementation with sixteen DiffMem tiles and one Controller tile. Each DiffMem tile is provisioned with: 32 eMAC units, a 2 MiB Matrix-Buffer; a 16KiB Matrix-Scratchpad; a 32KiB Vector Buffer, and a 4KiB Vector Scratchpad. The Controller tile is provisioned with an 8x8 matrix-multiply unit, and 5 MiB of on-chip storage for the unified and weight buffers. All of the compute units utilize full FP32 precision, in keeping with current MANN software implementations. The entire chip is clocked at 500 MHz and provides a total of 1.2TB/s of effective bandwidth for accessing the differential memory.

Comparison with GPUs We compare against two GPU platforms: a Pascal GTX 1080-Ti (1080-Ti) and a Turing RTX 2080-Ti (2080-Ti). The 1080-Ti has a total of 11.9 MiB of on-chip memory across the register file, L1, shared memory, and L2, with 484 GB/s memory bandwidth, whereas the 2080-Ti has a total of 29.5 MiB with a 616 GB/s bandwidth. Both platforms executed a Pytorch 1.0 implementation that utilized the highly optimized cuDNN library. These platforms are summarized in Table 4.4.

4.5 Results

In this section, we present the results of our experiments evaluating the benefits of MANNA.

4.5.1 Inference Performance

Figure 4.7 compares the performance of MANNA with the GPU implementations, when executing the inference phase with no batching for 10 NTM benchmarks (ordered by size of the external memory). Overall, MANNA achieves 11x - 184x speedup (average 39x) over the 1080-Ti. The average speedup achieved by MANNA over the 2080-Ti is 24x. There is a substantial difference in speed up for the small benchmarks (recall-rptcopy) as compared to the larger benchmarks (bAbI-shrdlu). To illustrate where the benefits come from, as well as to show why there is such a substantial difference between the small and large benchmarks, we investigate the kernel-specific performance improvements for the addressing kernels, the read and write heads, and soft reads in Figure 4.8.

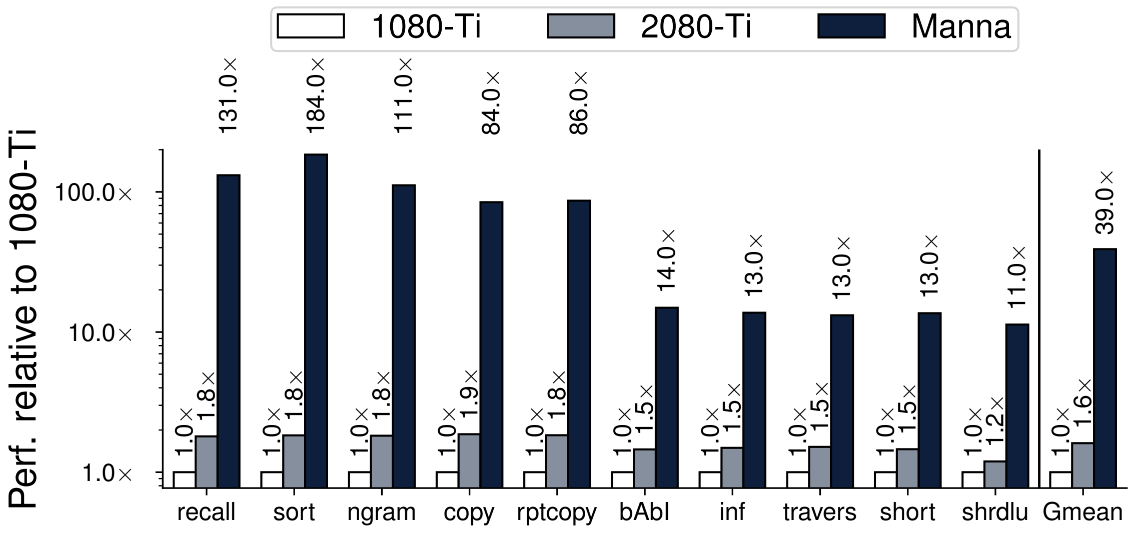


Figure 4.7. Inference performance

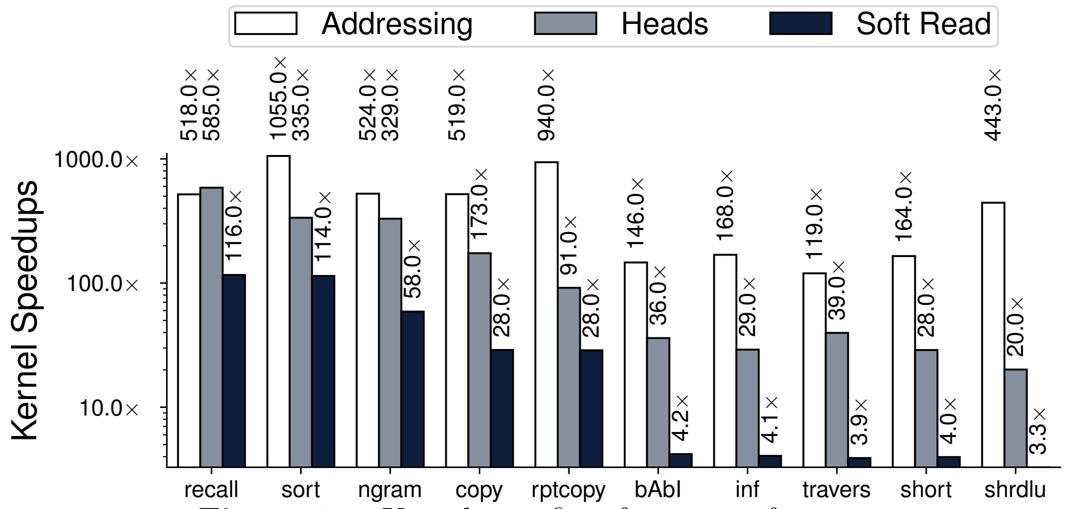


Figure 4.8. Kernel-specific inference performance

MANNA demonstrates significantly higher improvement in performance on the addressing kernels because MANNA is able to fully parallelize these kernels across the available processing elements, while the GPU is severely underutilized on these kernels. Unlike the addressing kernels, the soft read kernel involves a vector-matrix multiplication against the differentiable external memory. Thus, at smaller memory sizes, MANNA exhibits a significant speedup over the GPU platforms, since there is still not enough compute to fully utilize the GPU and mitigate the kernel call overheads. However, as the external memory size increases, the amount of parallelism that can be exposed to the GPU grows rapidly. For the largest benchmarks, MANNA's speedup over the GPUs saturates at around `3x`, at which point the GPUs are highly utilizing all of their cores (Streaming Multiprocessors) and memory bandwidth. We observe a similar trend for the key similarity and soft write kernels. Finally, the head kernels represent a middle ground between these two extremes; the read and write head kernels also involve a vector-matrix multiply, but with a much smaller matrix than the differentiable memory. Correspondingly, MANNA's performance improvement is between the two extremes.

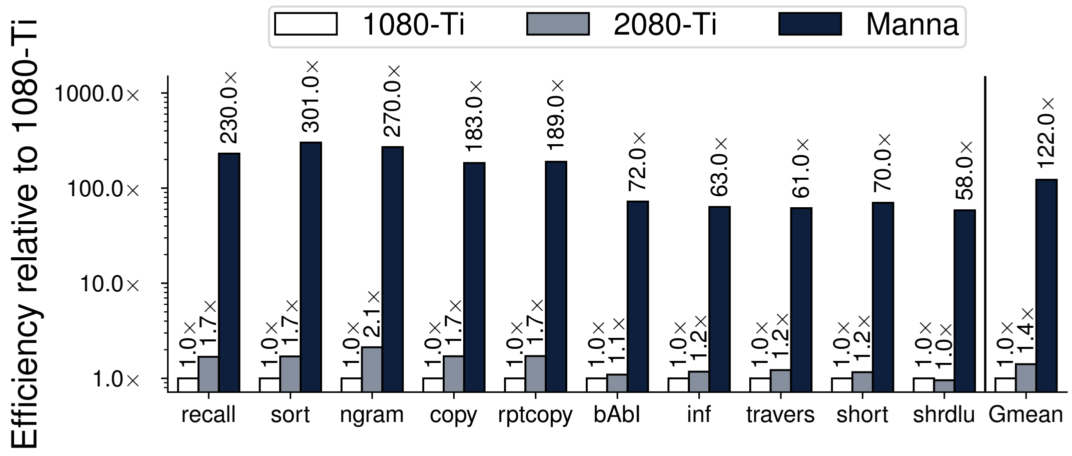


Figure 4.9. Energy efficiency compared to GPU baselines

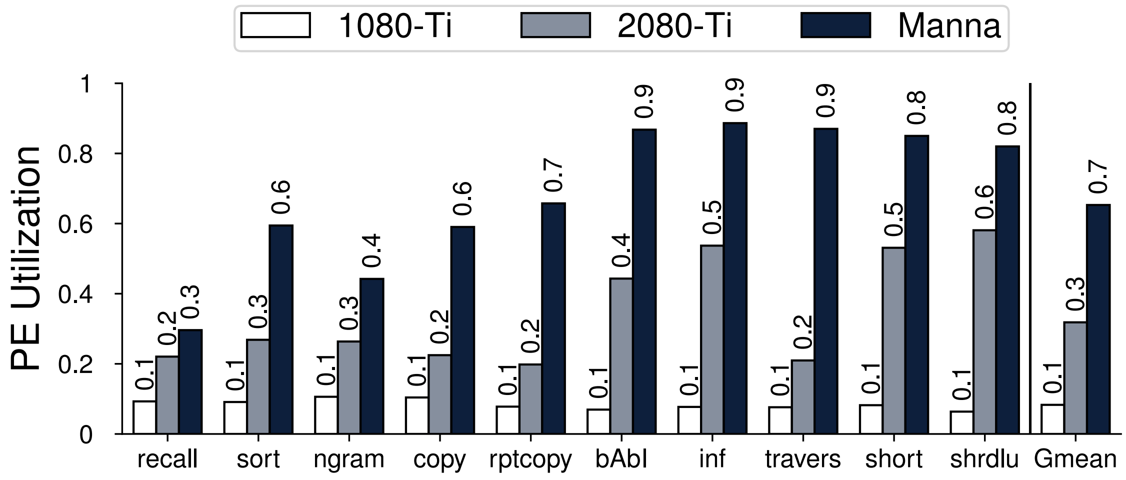


Figure 4.10. PE Utilization for the three platforms

4.5.2 Energy Efficiency

Since MANNs are sequentially evaluated across time steps, to quantify the energy efficiency of the system we consider how many time steps each platform could compute for fixed amount of energy, *i.e.*, steps/J. Based on this measure of energy efficiency, `Manna` shows substantial improvements of `58x-301x` (`122x` on average) over the `1080-Ti` and `86x` average improvement over the `2080-Ti`, as shown in Figure 4.9. A majority of this energy efficiency is derived from the increase in performance demonstrated in Figure 4.7, reducing the amount of time the system needs to be active to compute a time step. Moreover, `Manna` consumes an order of magnitude lower power than GPUs, which further increases its energy efficiency. Finally, as shown in Figure 4.10, `Manna` exhibits much higher (average `7x` over the baseline GPU) utilization of its processing elements, leading to a higher energy efficiency, as the underutilized on-chip resources of the GPU still draw considerable power without improving performance.

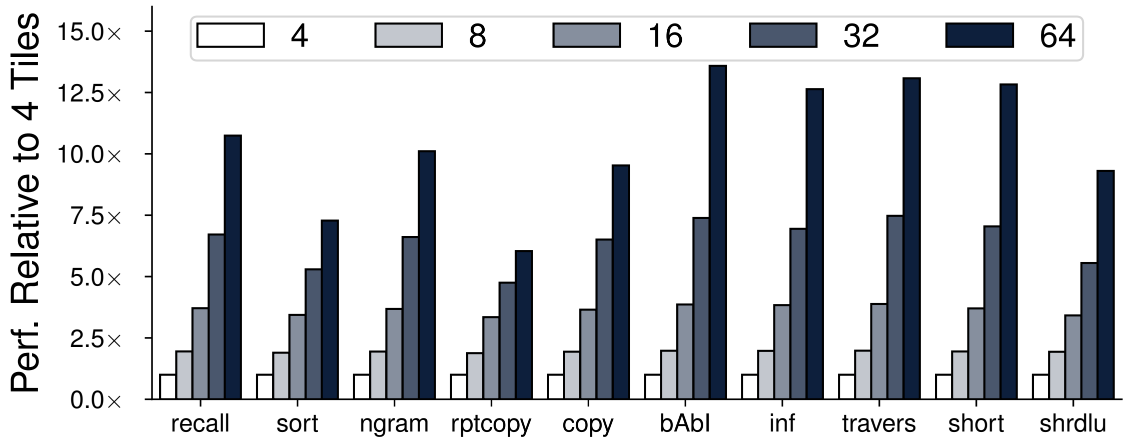


Figure 4.11. `Manna` performance trends with strong scaling

4.5.3 `Manna` Scaling

Strong scaling. First, we perform a strong scaling analysis in which we compare the performance of a small, four tile baseline configuration of `Manna` to designs having more tiles, while keeping the workload fixed. The results of this analysis are shown in Figure 4.11. Although `Manna` is able to scale quite well for many of the larger benchmarks, there are some limitations to the strong scaling of `Manna`. First, by choosing to distribute the external memory across the tiles such that $MDistrib = 1$, we see diminishing returns for smaller N_N, as in the case of smaller benchmarks, or for instances were N_M is similar in size (or greater) than N_N. But even for the large benchmarks where $N_N >> N_M$, we begin to see diminishing returns as the scaling factor increases because MANNA becomes limited by the serial accesses to the SFUs in each tile. Therefore, the speedups achievable through strong scaling are limited without (i) also allowing parallelization across M_M, and (ii) increasing the number of SFUs accordingly.

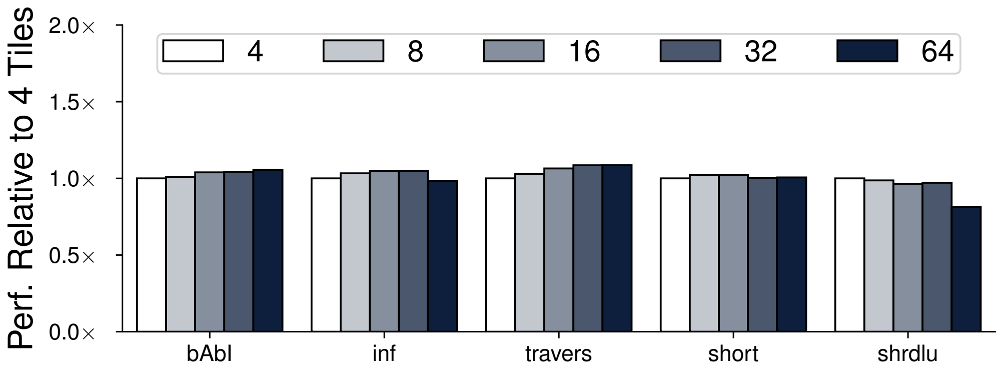


Figure 4.12. MANNA performance trends with weak scaling

Weak scaling. In this analysis, unlike the strong scaling analysis, as we grow the number of DiffMem tiles in MANNA, we correspondingly grow the differential external memory size of the benchmarks used, while maintaining the aspect ratio. So, if we analyze a system with sixteen tiles, then both dimensions of the external memories used in the benchmarks are doubled as compared to those used for the four tile system, resulting in a problem size that is four times larger. As seen in Figure 4.12, MANNA exhibits near-ideal weak scaling, exhibiting very little variability as the number of tiles and problem size are both changed. This is due to MANNA's ability to exploit the embarrassingly parallel nature of many of the MANN kernels, meaning the amount of inter-tile communication required is minimal compared to the amount of memory accesses and compute that each tile performs between communications.

Scaling the Differentiable Memory. Finally, we discuss scaling the differentiable memory to be larger than the capacity of a single MANNA chip. In this case, multiple MANNA chips can be used in a cluster, with the state distributed across them. This scaling method increases the parallelism and compute available proportionally with the capacity of the differential memory. However, in scenarios where scaling out is prohibitive due to cost or space constraints, a High Bandwidth Memory (HBM) can be introduced in the memory hierarchy of a single `Manna` chip without significant degradation in performance. To understand this further, consider the worst-case scenario in which there is no re-use of the differentiable memory. Each tile has 32 eMACs, each of which requires 4 bytes of differentiable memory elements, for a total of 128 bytes each cycle. An HBM2 module can provide 256GB/s of bandwidth. Since `Manna` has a 500MHz clock, this means that `Manna` can receive 512 bytes per clock cycle from HBM2, enough to feed four tiles. Thus, with 4 such memory modules, the 16-tile `Manna` baseline can have enough bandwidth to cater to all its processing elements. However, adding DRAM is not without its drawbacks. Each HBM2 controller will add $\sim 35mm^2$ area [111], leading to an increase in total chip area from $40mm^2$ to $180mm^2$. At a 25W power envelope for each HBM2 module, the TDP of `Manna` will increase to 116W. This design would result in an average `17x` improvement in energy efficiency for `Manna` over the baseline, down from the `122x` improvement for the SRAM-only design.

4.5.4 Ablation Study

We next compare `Manna` to three accelerator variants in order to illustrate how the various architectural features of `Manna` contribute to its performance. The first variant, `MemHeavy`, is a design optimized for low FLOPs/byte that dedicates the vast majority of the die area to large, highly-banked on-chip memories. However, it does not provide hardware support for transpose or element-wise operations. The second variant builds on the first variant by adding support for transpose (`MemHeavy-Transpose`), while the third variant replaces the traditional MAC units with the proposed eMAC units (`MemHeavy-eMAC`). As shown in Figure 4.13, `Manna` achieves `2x-4x` (`3.3x` average) performance improvement over the `MemHeavy` design. Similarly, it achieves `2.3x`, and `1.8x` average speedup over the designs with only hardware-assisted transpose and eMACs, respectively.

4.6 Conclusion

MANNs are a promising direction in machine learning that enable DNNs to achieve cognitive capabilities well beyond those of classical DNNs. In this chapter, we provide a detailed investigation of the memory and computational characteristics of MANNs, as well as their

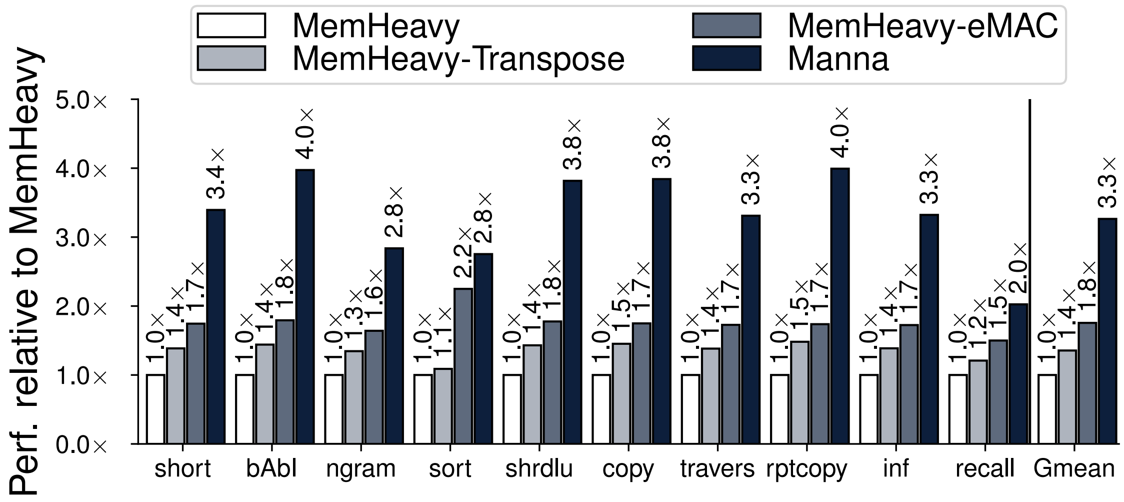


Figure 4.13. Impact of MANNA's architectural features.

impact on the design of an efficient accelerator for MANNs. We present MANNA, a specialized hardware accelerator for MANNs. MANNA is better suited to the low FLOPs/Byte ratio inherent to MANNs, allocating most of the on-chip die area to memory. MANNA also utilizes a hardware-assisted transpose mechanism to accomodate the heterogeneity of memory access patterns found in MANNs. We evaluate a 16-tile MANNA configuration and demonstrate significant performance and energy benefits over GPUs, as well as favorable weak and strong scaling.

5. SOFTERMAX: HARDWARE/SOFTWARE CO-DESIGN OF AN EFFICIENT SOFTMAX FOR TRANSFORMERS

5.1 Introduction

Transformer neural networks have recently become an extremely important deep learning (DL) workload, achieving state-of-the-art performance in a number of natural language processing tasks (NLP) [112]. These networks are characterized by their use of Transformer layers, which utilize self-attention, an attention mechanism that relates different symbols within a sequence in order to compute a representation of the sequence.

Based on the success of the self-attention mechanism, the Transformer network and its later variants have quickly come to dominate the field of natural language processing. This success is not just limited to NLP tasks; Transformer-based networks have recently begun to also show tremendous promise on tasks previously dominated by Convolutional Neural Networks (CNNs), such as image recognition [113].

However, the functional performance of Transformers comes at a cost. These networks are quite large, spanning hundreds of millions to hundreds of billions of parameters in recent networks such as BERT [114], Megatron [115], GPT-2 [116] and GPT-3 [87]. These networks are continuing to grow in size; OpenAI's GPT-3 [87] has 175B parameters and an input sequence length of 2048 tokens in comparison to GPT-2's 1.5B parameters and sequence length of 1024. Looking beyond the high memory and compute overheads associated with large models, Transformer networks also have a unique mix of computations, as each attention layer of a Transformer network consists of softmax and dropout operations in addition to the standard matrix multiply-based fully-connected layers. As shown in Figure 5.1, these attention operations, particularly the softmax computation, represent a large fraction of runtime, especially at the longer sequence lengths found in more recent state-of-the-art networks.

Previous DL inference accelerators have focused on CNNs, MLPs and LSTMs, which are dominated by matrix-multiply operations. Relatively little attention has been given to the acceleration of softmax, since it contributes a negligible amount to the runtime of these networks. With the growing importance of Transformers (*e.g.*, inference for conversational

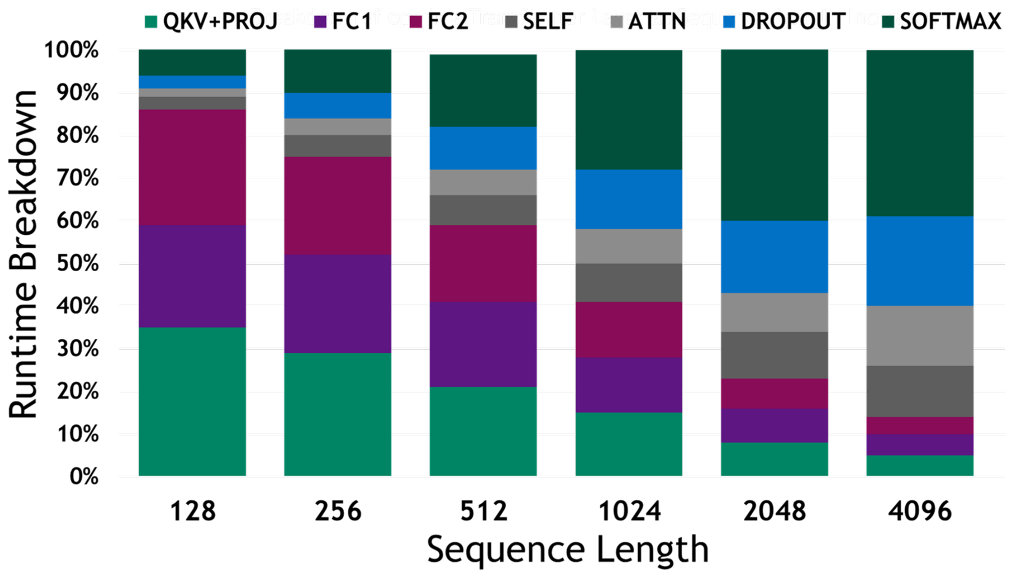


Figure 5.1. Runtime breakdown for BERT-Large on a Volta GPU. Softmax contributes a larger fraction of the run time in Transformers than other DNNs, particularly at the longer sequence lengths seen in state-of-the-art networks

AI), it has become important to improve the performance of the softmax operation, which is the focus of this chapter. To that end, we make the following contributions:

- We propose `Softermax`, a hardware-friendly softmax algorithm that consists of base replacement, low-precision softmax computations, and an online normalization calculation
- Taking advantage of the fine-tuning paradigm of Transformer-based networks, we utilize `Softermax`-aware finetuning to reduce the accuracy loss incurred by our hardware-friendly algorithm while introducing no additional training overhead
- We detail the microarchitecture necessary to implement `Softermax` in an inference accelerator
- We demonstrate that `Softermax` achieves a 2.35x more energy efficient implementation while using 0.90x the area in a 7nm FinFET technology, with negligible impact on network accuracy

The rest of this chapter is organized as follows. In Section 5.2, we detail our algorithmic adaptions to the traditional softmax algorithm, which ease the implementation of a hardware-friendly design. In Section 5.3, we detail the hardware units necessary to implement our redesigned algorithm. In Section 5.4, we describe our experimental setup and in Section 5.5 we evaluate our `Softermax` proposal. Finally, Section 5.6 concludes the chapter.

5.2 `Softermax` Software

The `Softermax` algorithm, described in Figure 5.2, is comprised of four enhancements over the standard softmax. First, we propose switching the base used in the exponential calculation from Euler's number, e, to 2. Second, we replace full precision computations with fixed-point, low precision calculations– including for exponentiation and division. Next, we use a hardware-friendly online normalization scheme to avoid an additional explicit pass to calculate the max. Finally, we propose `Softermax`-aware finetuning, wherein a non-`Softermax` based pretrained model is fine-tuned for a specific task, using `Softermax` in place of the standard softmax layer, in order to account for errors introduced by our first two techniques. We note that this fine-tuning for downstream tasks is already required for Transformer-based networks, so we are not introducing additional overheads; we are simply making the required fine-tuning `Softermax`-aware. In summary, a Transformer-based model is first *pre-trained* using the standard, high-precision softmax, *fine-tuned* for a specific downstream task in a `Softermax`-aware manner to mitigate accuracy loss, and deployed for *inference* using `Softermax`.

5.2.1 Base Replacement

Softmax is used in DNNs for three reasons. First, it produces a probability distribution which is useful for classification as well as in self-attention, where a probability can be thought of as a weighting. Second, it is differentiable and therefore can be used with gradient descent-based optimization. Finally, the use of the natural exponential provides a non-linear function wherein small differences in input values are exaggerated, leading to much higher probabilities for higher scores. We note that the use of the natural exponential results in overheads for both specialized and general-purpose hardware. In specialized hardware, it is usually cheaper to implement the natural exponential using a power of two compute unit coupled with a

```
1. $m_0 \leftarrow -\infty$
2. for $k \leftarrow 1, V$ do
3. $m_k \leftarrow \max(m_{k-1}, x_k)$
4. end for
5. $d \leftarrow 0$
6. for j $\leftarrow 1, V$ do
7.     $d \leftarrow d + 2^{x_j - m_V}$
8. end for
9. for i $\leftarrow 1, V$ do
10.    $y_i \leftarrow \frac{2^{x_j - m_v}}{d}$
11. end for
```

(1)

```
1. $m_0 \leftarrow -\infty$
2. $d \leftarrow 0$
3. for j $\leftarrow 1, V$ do
4.     $m_j \leftarrow \max(m_{j-1}, x_j)$
5.     $d \leftarrow d \cdot 2^{m_{j-1} - m_j} + 2^{x_j - m_j}$
6.     $y_i \leftarrow 2^{x_j - m_j}$
7. end for
8. for i $\leftarrow 1, V$ do
9.     $y_i \leftarrow \frac{y_i \cdot 2^{m_i - m_V}}{d}$
10. end for
```

(2)

```
1. $m_0 \leftarrow -\infty$
2. $d \leftarrow 0$
3. for j $\leftarrow 1, V$ do
4.     $m_j \leftarrow \mathrm{IntMax}(m_{j-1}, x_j)$
5.     $d \leftarrow d \gg (m_j - m_{j-1}) + 2^{x_j - m_j}$
6.     $y_i \leftarrow 2^{x_j - m_j}$
7. end for
8. for i $\leftarrow 1, V$ do
9.     $y_i \leftarrow \frac{y_i \gg (m_V - m_i)}{d}$
10. end for
```

(3)

Figure 5.2. The algorithmic changes proposed in `Softermax` consist of: (1) replacing e^x with a low-precision implementation of 2^x, (2) replacing an explicit pass to calculate the max with an online version, and (3) replacing the maximum function with an integer-based version to simplify the renormalization calculations.

dedicated hardware multiplier for performing the base e to base two conversion. Similarly, in general-purpose hardware, there is software overhead to perform this conversion. We further note that a base two exponential still satisfies the three desirable properties of softmax listed above, while allowing for a more hardware-friendly implementation. Therefore, `Softermax` uses a base-two based softmax. We demonstrate that this substitution leads to negligible loss in accuracy when `Softermax`-aware fine-tuning is performed.

5.2.2 Low-precision Softmax Operations

Even with the above base replacement, softmax now consists of a power of two calculation, an accumulation, and a division. With 32-bit single-precision floating point, these units can be expensive in terms of area and power. Part of the reason these units are so expensive is the need for highly accurate results, particularly in general purpose computing platforms that cater to a wide range of applications. DNNs, by contrast, are quite resilient to errors introduced through low precision computing. With this in mind, we propose performing all of the softmax compute operations - exponential, accumulation, and division - in low precision. We note that this is only possible through custom hardware, as commod-

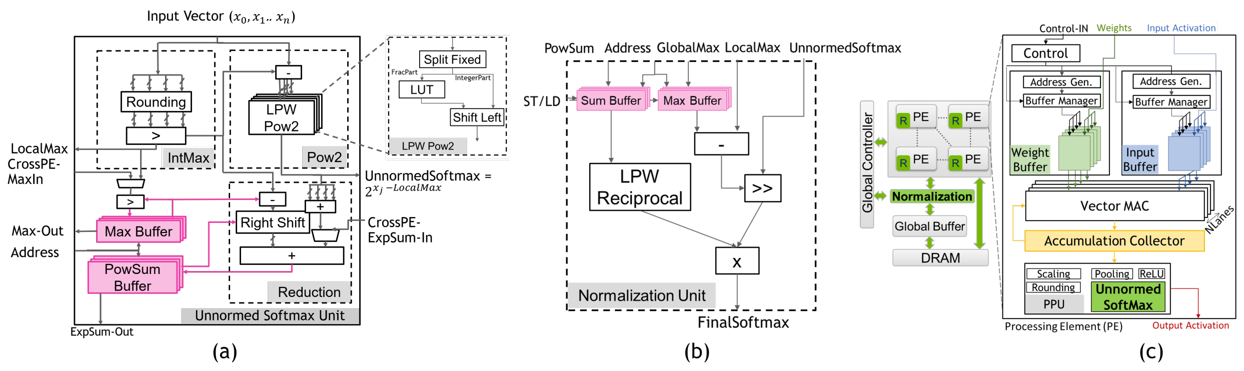


Figure 5.3. (a) The Unnormed Softmax Unit determines the local max, performs the power of 2 calculation using the current max, and accumulates the denominator. (b) The Normalization Unit performs the renormalization of the numerator, as well as the final division of the numerator by the accumulated sum. (c) In an example accelerator [29], the Unnormed Softmax can be integrated into the post-processing vector unit on a per PE basis, while the Normalization Unit can be shared across multiple PEs and integrated between the PEs and the Global Buffer

ity hardware platforms do not support low precision special function units like power and division. Again, Softermax-aware fine-tuning minimizes accuracy loss as a result of these low-precision operations.

5.2.3 Hardware-friendly Online Normalization

In the DNN context, softmax is typically used in a "numerically stable" version, wherein the input vector is preprocessed by subtracting the maximum of the vector from every element. This helps with training stability, but at the cost of introducing an additional pass through the input vector. Prior work [117] has proposed an "online normalizer" calculation, where the max is calculated continuously along with the normalization value (*i.e*, the summation for the denominator). In this case, the maximum of the vector is not subtracted before applying the exponential; rather, it is the maximum of the vector *up to this point*. Thus, the current running sum must be renormalized when a new max is found. To see why this is the case in a concrete example, assume we are processing the vector [2,1,3]. For the first element, the running max is two and thus the running sum will be $d = 2^{2-2} = 1$. For the second element, the running max is still two and thus the running sum will be

$d = 1 + 2^{1-2} = 1.5$. For the final element, however, we encounter a new maximum value of three. We cannot simply add to the existing sum as is (*i.e.*, $d \neq 1.5 + 2^{3-3} = 2.5$) since the previous accumulations were computed with a different point of reference for the numerically stable exponentiation. We must instead renormalize the running sum to account for this by multiplying by $2^{OldMax-NewMax}$, so that $d = 1.5 \times 2^{2-3} + 2^{3-3} = 0.75 + 1 = 1.75$. Note that this result is the same as if we had computed the accumulation using the true global maximum from the start: $d = 2^{2-3} + 2^{1-3} + 2^{3-3} = 1.75$. For a mathematically rigorous proof of this algorithm, we refer the reader to [117].

We propose a simple co-design modification in order to make the online calculation more hardware-friendly. Specifically, we note that, unlike the original online normalization algorithm, we use a base two implementation. With this in mind, we switch the max function with an integer max function, ensuring that the difference between maxes is always an integer. Since the renormalization operation is multiplying by $2^{OldMax-NewMax}$ and $OldMax - NewMax$ is guaranteed to be an integer, the renormalization hardware can therefore be realized simply using a shifter.

5.3 `Softermax` Hardware

We propose two compute units, *viz.* the Unnormed Softmax unit and the Normalization unit, to realize `Softermax` (Figure 5.3). The Unnormed Softmax unit calculates the local maximum, performs the exponential, and accumulates the denominator (lines 4-6 in the final algorithm in Figure 5.2). The Normalization Unit performs the final renormalization of the numerator and the division (lines 9-10). We provide greater detail of the implementation of these units in the subsections below. We also detail how these units may be integrated in an existing DNN inference accelerator.

5.3.1 Unnormed Softmax Unit

The Unnormed Softmax unit, shown in Figure 5.3(a), consists of three subunits: the IntMax unit, the Power of Two unit, and the Reduction unit.

IntMax Unit: The IntMax Unit operates on a slice of an output vector. It applies a ceiling function to each element in parallel before finding the max of the vector slice, thereby implementing an IntMax instead of simply max.

Power of Two Unit: Power of two is implemented by decomposing the fixed point input into integer and fractional parts. The fractional part is implemented as a linear piece-wise function (LPW) applied to the range [0, 1), defined below.

$$x_{scaled} = frac(x << 2) // \ 4 \ segments \ in \ LPW$$

$$lpw = m_{lut}[int(x_{scaled})] * frac(x_{scaled}) + c_{lut}[int(x_{scaled})]$$

In our implementation, we use four segments in the LPW equation, while modern general purpose hardware typically utilize between 64-128 entries, a considerable overhead. To account for the four segments, our implementation (described above) requires a shift left by two (multiplication by four) in the first line The fractional part of this scaled value ($frac(x_{scaled})$) is then multiplied by the output of the m LUT. We note that, in cases where the input has less than two decimal places (*i.e.*, the input has two or fewer fractional bits), the fractional part will always be zero. Hence, the m LUT is unused and the LPW implementation of the fractional part is simply:

$$lpw = c_{lut}[int(x_{scaled})]$$

Either way, the output of the LPW is then shifted by the integer part to obtain the final result.

Reduction Unit: The Reduction unit receives the $UnnormedSoftmax$ from the power of two unit and reduces it using a summation tree. It also reads from buffers in case the output vector is larger than can be computed in one slice. In this case, the current largest max for the row is read from the buffer along with the current running sum for that row. The current largest max is compared to the $LocalMax$ found and the running sum is renormalized as needed using a shifter when there is a difference between the local max and the current determined max for a row, as required by the online normalization algorithm. The renormalized running sum and the local sum are then added together to obtain the new running sum for the row.

5.3.2 Normalization Unit

The Normalization Unit performs the renormalization of the numerator as well as the division to obtain the final result. As a result of the integer max used in `Softermax`, it is guaranteed that the difference between the local maximum and the global maximum is an integer. Therefore, due to the base change used in `Softermax`, the renormalization of the numerator can be implemented simply using a shifter. The division is implemented using a linear piece-wise reciprocal unit, followed by an integer multiplier.

5.3.3 Accelerator Integration

The UnnormedSoftmax Unit is designed to integrate into tensor processing hardware, present in GPU tensor cores [118], TPUs [8], or other dedicated DNN inference accelerators. For example, using the MAGNet architecture [29] as a baseline, the unit can be integrated into its post-processing unit (PPU), which performs operations such as pooling and ReLu. Ideally, the UnnormedSoftmax Unit should be sized such that it matches the MAC throughput, fully exploiting the low-overhead hardware enabled by Softermax. The Normalization Unit can be introduced between the processing tile and global memory to complete the softmax off of the critical path.

5.4 Experimental Setup

Software setup: To evaluate the impact of `Softermax` on accuracy, we modified the PyTorch Huggingface library [119]. Our Huggingface implementation uses a 99.999% percentile calibrator to generate the scale factors which are then used to to perform quantization-aware finetuning with 8-bit weights and activations [120]. We further augment our implementation with custom forward/backward passes for each of the fixed point softmax operations: power of 2, reciprocal, etc. The forward passes faithfully implement the fixed point, low precision computations, while the backward passes use the straight through estimator (STE) for the conversions to/from fixed point. The bitwidths for each operation are summarized in Table 5.1. We note that the input and output of our `Softermax` are 8-bits each, allowing for easy integration into existing 8-bit integer vector MAC datapaths, such as those found in modern GPU tensor cores and DNN inference accelerators.

Table 5.1. Summary of `Softermax` Bitwidths, Q(Int., Frac.)

Inp.	**LocalMax**	**Unnormed**	**PowSum**	**Recip.**	**Outp.**
Q(6,2)	Q(6,2)	Q(1,15)	Q(10, 6)	Q(1,7)	Q(1, 7)

Hardware setup: To quantify the area and power overheads associated with `Softermax`, we implemented the proposed hardware units using high-level synthesis and integrated the units into an existing accelerator, MAGNet [29]. We compare our `Softermax` implementation to a standard 16-bit floating point precision softmax implementation that uses Synopsys DesignWare components[121]. We note that this represents an optimistic baseline already, as current state of the art accelerators [118] use a full 32-bit precision implementation.

5.5 Evaluation

We split our evaluation of `Softermax` into two parts. First, we demonstrate that our proposals for making softmax more hardware-friendly do not have a negative impact on accuracy. Second, we demonstrate that these proposals do in fact result in more efficient hardware implementations.

5.5.1 Impact on Accuracy

We use the setup described in the previous section to analyze the impact of our proposed `Softermax` implementation.

Under these conditions, we find that `Softermax` results in negligible loss in accuracy. Table 5.3 details the accuracy results on the BERT-Base and BERT-Large networks, across the SQuAD and GLUE tasks. To summarize, `Softermax` results in negligible impact on accuracy; the worst drop in accuracy is under 0.5%, while the average accuracy actually

Table 5.2. Experimental Setup

Design Tools	
HLS compiler:	Mentor Graphics Catapult HLS
Verilog simulator:	Synopsys VCS
Logic synthesis:	Synopsys Design Compiler Graphical
Power analysis:	Synopsys PT-PX
Design Parameters	
Weight/Activation precision:	8 bits
Accumulation precision:	24 bits
VectorSize:	16 | 32
NLanes:	16 | 32
Input Buffer Size:	16KB | 32KB
Weight Buffer Size:	32KB | 128KB
Accumulation Collector Size:	6KB | 12KB
Technology Node:	TSMC 7nm
Supply Voltage:	0.67V

Table 5.3. Accuracy results for `Softermax` vs an eight-bit quantized baseline

		SQuAD	**RTE**	**CoLA**	**MRPC**	**QNLI**	**QQP**	**SST-2**	**STS-B**	**MNLI**
BERT	**Baseline**	86.28	62.45	53.65	84.31	90.77	90.71	92.09	87.86	83.27
Base	**Softermax**	85.86	64.26	56.76	84.07	90.41	90.83	92.20	87.78	83.80
BERT	**Baseline**	89.40	65.70	59.58	86.03	92.09	91.24	92.89	89.39	85.87
Large	**Softermax**	89.46	69.68	60.10	86.27	91.76	90.90	92.66	89.55	85.74

increases 0.9% and 0.7% across all tasks for BERT-Base and BERT-Large, respectively. This validates the use of `Softermax` as an acceptable replacement for full-precision, numerically-stable softmax in Transformer-based networks.

5.5.2 Impact on Hardware Efficiency

Compute Unit Level Analysis. We first compare our proposed `Softermax` implementation to a DesignWare-based baseline [121] in isolation– *i.e.*, not integrated into the PE of a DNN accelerator. Specifically, we compare the Unnormed Softmax unit, as implemented using the techniques outlined in `Softermax`, to a unit in which the requisite units (max, exponential, accumulation) are implemented using DesignWare components. We consider a softmax workload with a sequence length of 384, as used in SQuAD dataset. We see that `Softermax` offers a much more efficient implementation, resulting in an Unnormed Softmax unit that is `4x` smaller and `9.53x` more energy efficient as demonstrated in Table 5.4. Similarly, the Normalization Unit is much more efficient than the baseline, resulting in a unit that is `1.54x` smaller and `2.53x` more energy efficient.

Table 5.4. Softermax comparison to DesignWare-based softmax baseline for SQUAD dataset

	Area (um^2)	**Energy (uJ)**
Unnormed Softmax Unit	0.25x	0.10x
Normalization Unit	0.65x	0.39x
Full PE	0.90x	0.43x

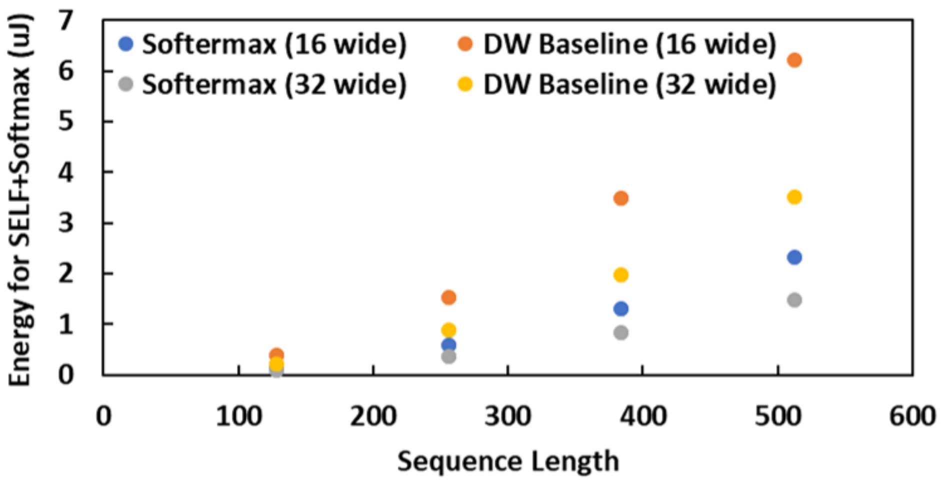


Figure 5.4. Energy consumption of a `Softermax`-based PE vs a DesignWare baseline for SELF+Softmax as the sequence length increases. We evaluate 16- and 32-wide implementations.

PE-level Analysis. Next, we compare `Softermax` to the DesignWare-based baseline when integrated into the PE of a DNN accelerator. Specifically, we integrated the Unnormed Softmax unit into a 32-wide MAGNet PE, with parameters as described in Table 5.2. As shown in Table 5.4, we see that `Softermax` still offers a more efficient solution even when integrated into a PE and thus accounting for other, non-softmax related components such as the MAC units and various scratchpads; `Softermax` is `1.11x` more area efficient and `2.35x` more energy efficient than the baseline.

Sequence Length Sweep. Finally, we perform a sweep of the sequence length of the input vector, evaluating both 32-wide as well as 16-wide PE configurations. As shown in Figure 5.4, `Softermax` scales much better than the baseline, both starting from a lower baseline and having a more shallow slope. This is crucially important in Transformer-based networks, which are trending towards longer and longer sequence lengths.

5.6 Conclusion

Transformer networks are an important emerging class of deep learning workloads, which differ computationally from other DNNs through their extensive use of softmax computations. Implementations of Transformers on current general-purpose and specialized hardware platforms are therefore limited by the time and energy of softmax operations. To address this challenge, we proposed `Softermax`, a set of software and hardware optimizations to softmax operations in Transformer networks. Our implementation of `Softermax` indicates that it can lead to 4x area and 9.53x energy improvements over conventional softmax units, translating to 1.11x area improvement and 2.35x energy improvement for softmax computations within a state-of-the-art DNN accelerator, with no loss in accuracy compared to the quantized baseline.

6. GNNERATOR: A HARDWARE/SOFTWARE FRAMEWORK FOR ACCELERATING GRAPH NEURAL NETWORKS

6.1 Introduction

Deep Neural Networks (DNNs) have established state-of-the-art accuracy in a wide range of domains, such as image recognition [1], [100], [122], [123], language translation [73], [124], natural language processing [114], [125], and voice recognition [75], [126]. For the most part, these advances in the state-of-the-art have been limited to inputs from the Euclidean domain, such as images, speech, and text. Recently, however, there has been research in expanding the success of deep learning to non-Euclidean domains, such as point clouds, graphs, manifold, *etc.*, resulting in the emerging field of *geometric deep learning* [127]. Of particular interest is the application of deep neural networks to graph-based representations. These graph neural networks (GNNs) are used in a wide variety of applications such as physics modeling [128], [129], chemical synthesis [130]–[132], recommendation systems [133], medical diagnosis [134] even electronic design automation (EDA) problems [135], [136]. Recently, GNNs have moved from the research lab to production systems [93], [137].

These GNNs consist of two main stages: the *feature extraction* stage and the *aggregation* stage. The feature extraction stage is essentially a traditional deep neural network– specifically, a fully-connected layer. In the aggregation stage, each vertex in a graph aggregates features from its neighbors into a new feature representation, resulting in sparse, random memory accesses. Thus, the aggregation stage is similar to a traditional graph processing algorithm. However, there is a crucial difference between the aggregation stage in GNNs and other traditional graph-processing workloads such as PageRank; the node (edge) features being access in traditional workloads tend to be a few bytes and of a static length, whereas the features in GNNs can be thousands of bytes long, and are of a dynamic length, since the feature extraction stage may project the features into a different dimension space. These stages are then combined in order to form a single GNN layer, and multiple GNN layers are stacked to form the full GNN.

As a result of their unique computational characteristics, which are a combination of both dense computations as well as graph-based computations, existing hardware architectures are not well suited for GNNs. Neural network accelerators such as Eyeriss [10] or Google's TPU [8] do not have support for graph operations. Instead, the graph operations such as feature aggregation must be performed by casting the operation as a sparse-dense matrix operation, a popular optimization for high performance graph analytics [138]. This is undesirable for two main reasons. First, existing DNN accelerators are optimized for the sparsity levels found in traditional DNNs, which tends to be under 85% for weight sparsity and 60% for activation sparsity [12]. Graph sparsity, on the other, can be vastly more sparse; the popular Amazon shopping dataset is 99.998% sparse. Further, realizing the graph operations as sparse-dense matrix operations limits the operations to those than can be expressed as such, which artificially constrains the space of future GNN research. Finally, the hardware and software for these neural network accelerators are not optimized for the memory patterns present in the graph-based aggregation step, leading to poor utilization of on-chip memories.

Graph analytics accelerators such as [139], [140] are also ill-suited for GNNs. First, these accelerators simply lack support for the dense, compute-heavy aspect of GNNs found in the feature extraction stage. Surprisingly, these accelerators are also not well-suited for the graph-based aggregation stage of GNNs. This is because graph analytics accelerators tend to be optimized for small graph features, as is typical in traditional graph workloads such as PageRank. For example, Graphicianado [101] is optimized for supporting graph features of a few bytes at most; GNN graph features, by contrast, can be thousands of bytes.

Thus, there have been recent efforts to design graph neural network-specific accelerators. HyGCN [59] and GNNA [60] use heterogenous compute units for the feature extraction and aggregation stages, which are then tightly coupled together. EnGN [61] proposes a unified architecture that performs the two stages using the same register file and compute units. These previous works mainly focus on small GNNs, with small hidden dimensions and only a single layer, which are unrepresentative of the sizes of networks being currently proposed and used in production. Further, these works do not consider many parts of the GNN architecture design space, such as graph communication styles.

To address these shortcomings, we propose GNNERATOR a new <u>G</u>raph <u>N</u>eural <u>N</u>etwork accel<u>erator</u>. GNNERATOR consists of a Graph Engine and a Dense Engine, provisioned to efficiently execute the aggregation and feature extraction stages, respectivel. GNNERATOR also contains a Controller that orchestrates the fine-grained pipelining between the two engines. The Dense Engine is a systolic-array based compute engine that is optimized for the large amounts of weight-reuse present in GNNs. The Graph Engine consists of multiple Graph Processing Elements that operate on different nodes of a graph, each of each has multiple lanes that execute multiple feature dimensions in parallel. Thus, the Graph Engine can exploit both inter- and intra-node parallelism.

We develop a well-defined execution model and a compiler framework to realize GNNs on the proposed architecture using this execution model. We evaluate GNNERATOR on a suite of benchmarks using a GPU baseline, as well as HyGCN. We also analyze the impacts of larger networks, as well as explore different aspects of the GNN accelerator design space.

In summary, we provide the following contributions:

- We design GNNERATOR, a programmable graph neural network accelerator that exploits both the intra-node parallelism as as well as the inter-node parallelism abundant in GNN workloads

- We propose a novel, feature-dimension blocking dataflow for GNNs and provide hardware support for this dataflow in the GNNERATOR accelerator

- We detail the ISA, execution model, and compiler necessary for efficiently executing GNNs on GNNERATOR

- We develop a simulation framework that implements the above architecture and software proposals. We use this framework to demonstrate the benefits of GNNERATOR. Our experiments indicate that GNNERATOR achieves an average speedup of `8x` over an NVIDIA 2080-Ti GPU and an average speedup of `3.15x` over a state-of-the-art GNN accelerator, HyGCN

- We provide a detailed study of the impact of various, previously unexplored GNN architectural parameters such as DNN dataflow and graph communication patterns, as well as the potential bottlenecks for GNN accelerators as GNN continue to increase in size (*i.e.*, depth and hidden layer size).

The rest of the chapter is organized as follows. Section 6.2 describes the proposed GNNERATOR architecture, as well as its ISA. Section 6.3 details how GNNs are mapped to and executed on GNNERATOR. Section 6.4 explains the proposed GNNERATOR compiler and runtime environment, including memory layout optimizations used for efficient usage of the on-chip memories. Section 6.5 outlines the simulation framework used to evaluate the proposed architecture. Finally, Section 6.6 presents the results of our experiments and Section 6.7 concludes the chapter.

6.2 GNNerator Architecture

GNNERATOR consists of two different compute engines, the Dense Engine and the Graph Engine. The Dense Engine is used for performing the dense, regular feature extraction steps of a GNN, while the Graph Engine performs the graph-based aggregation steps. The interoperation of these engines is controlled by the `GNNerator Controller`.

6.2.1 Dense Engine Overview

The Dense Engine is a systolic array-based DNN accelerator architecture that consists of a two-dimensional systolic array-based matrix multiplication unit, an activation unit, and input, weight, and activation buffers. All on-chip buffers are double-buffered. The input and weight buffers feed the systolic array. The output of the two-dimensional systolic array is connected to a one-dimensional activation unit, which performs any required activation operations (*e.g.*, ReLu). The results from the activation unit are stored in the output buffer, where they can either be transferred out to DRAM or to the input buffer to be reused as input to the systolic array. The Dense Engine can be configured as using an input-, output-, or weight- stationary dataflow using the `DNN Stationality` hyperparameter. In order to support fine-grain pipelining of the feature extraction and aggregation stages, the Dense Engine also contains connections with the Graph Engine that are used to communicate the current state of the respective computing engines.

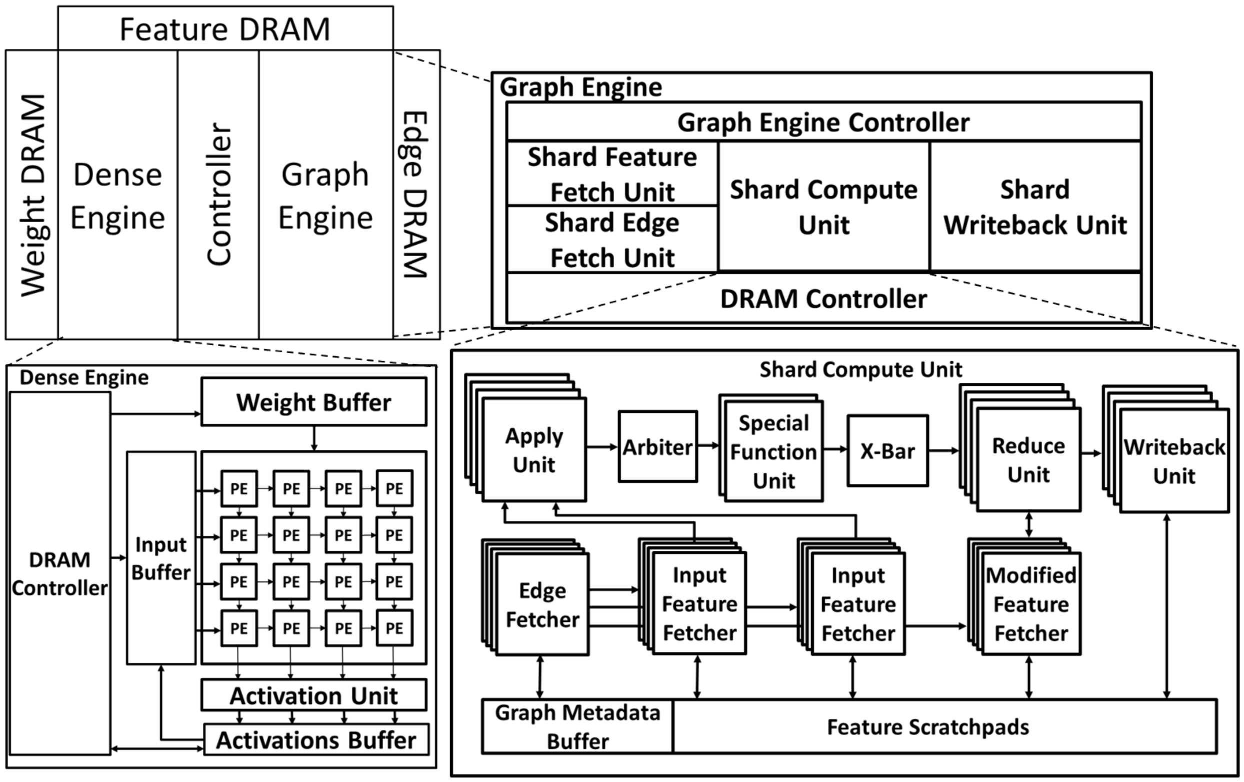


Figure 6.1. GNNERATOR (top left) consists of two heterogenous compute engines, a Dense Engine and a Graph Engine, that share feature storage. The Dense Engine (bottom left) is a systolic-array based architecture. The Graph Engine (top right) consists of a set of fetch and writeback units that orchestrate off-chip data movement of the graph shards as well as a compute unit. The compute unit (bottom right) contains units that orchestrate on-chip data movement as well as units that compute on that data.

6.2.2 Graph Engine Overview

The Graph Engine is tailored for the irregular workload of graph processing. It consists of four major units– the `Shard Feature Fetch`, the `Shard Edge Fetch`, the `Shard Compute`, and the `Shard Writeback Units`– as well as a `Graph Engine Controller` which orchestrates the processing pipeline between the four major units.

The Shard Edge Fetch and Feature Fetch Units work in parallel to load the edge data and feature data, respectively, required for computing a given graph shard from main memory to the on-chip scratchpads. After a shard is loaded, this computation is performed by the Shard Compute Unit, which steps through the edges associated with the given graph shard and performs the given computations. Finally, the Shard Writeback Unit stores output data from the on-chip scratchpads to main memory. As in the Dense Engine, all of the on-chip buffers in the Graph Engine are also double-buffered, enabling the pipelining of the above computations, such that the next shard is being prefetched while the current shard is being executed.

Internally, the Shard Edge Fetch, Shard Feature Fetch, and Shard Writeback Units are quite similar, consisting mainly of an automatic address generator that steps through the necessary addresses for performing the requisite loads (stores). The Shard Compute Unit, on the other hand, is more complex.

It contains an `Edge Fetcher`, which steps through the associated edges for the current graph shard. For each processed edge, the Edge Fetcher distributes the required edge information (*e.g*, source node ID) to the `Feature Fetcher Units`, as well as the `Writeback Unit`. These units use this information in order to generate read (write) accesses to the on-chip scratchpad, similar to the Shard level units. Note that there are three different Feature Fetch Units. These fetch units feed the actual compute units. The `Apply Unit` performs binary operations such as addition or multiplication, the `Special Function Unit` performs complex unary operations such as exponentiation or ReLu, and the `Reduce` unit performs an aggregation operation, such as maximum or summation. Each of these compute units are vectorized in order to exploit *intra-node parellelism* (*i.e.*, the different dimensions of a given node's feature).

In order to exploit *inter-node parallelism*, the Shard Compute Unit contains multiple copies of the set of units described above, referred to collectively as a `Graph Processing Element` (GPE). Each GPE (Edge Fetcher, Input Feature Fetcher, Modified Feature Fetcher, Writeback, Apply Unit and Reduce Unit) are assigned to a subset of the edges for a given graph shard. Since the Special Function Unit contains functional units that are area- and compute-intensive, the number of these units may be less than the other units. To accommodate this, a simple round-robin `Arbiter` is used to grant access to the limited number of units and a crossbar is used to deliver the results to the appropriate Reduce Unit. In order to support the push-style dataflow, wherein a source nodes *pushes* its contribution to all of its neighbors, the Reduce Units may also contain an atomic lock to ensure that only one

Reduce Unit is modifying the aggregated feature for a given destination at one time. This lock is implemented as a simple CAM-structure.

In addition to the units described above, each GPE also contains three sets of memories: a graph metadata memory (*i.e.*, edge list), an input feature memory, and a modified feature memory. Every GPE can access the feature memories in the other GPEs through a broadcasting crossbar, which avoids stalls associated with reading the same feature. However, atomicity is still ensured as described above. The edge memories are private to each GPE. The amount of *total* on-chip capacity for each type of memory is equally distributed across the number of GPEs, with one SRAM for each GPE.

6.2.3 GNNerator Controller

The `GNNerator Controller` coordinates the interaction between the Dense Engine and the Graph Engine. This coordination is mainly dependent on which of the engines is the producer and which is the consumer for a given computation– that is, if the computation is a feature extraction followed by an aggregation or the other way around.

Dense first. If the feature extraction is first, then the Dense Engine must run ahead of the Graph Engine. Hence, the `GNNerator Controller` reads the state of the Dense Engine and stalls the Graph Engine until the source nodes for the current shard of the Graph Engine have been processed by the Dense Engine.

Graph first. If aggregation is first, then the Graph Engine must run ahead of the Dense Engine. Hence, the `GNNerator Controller` reads the state of the Graph Engine and stalls the Dense Engine until the Graph Engine is *done* with a set of destination nodes (*i.e.*, the Graph Engine has gone completed a full column of the shard grid).

6.2.4 ISA

In order to ensure the wide variety of emerging GNNs can be executed on GNNerator, we design a custom CISC-based ISA for GNNerator, with two different classes of instructions: dense instructions and graph instructions. The dense instructions are used to perform course-grained matrix operations such as matrix-multiply or matrix-add. Other operations common to DNNs and which are applied in a dense manner, such as ReLU or softmax, are also performed using the dense instructions. The graph instructions can be divided into two categories: compute and memory instructions.

Graph memory instructions. A graph memory instruction consists of nine fields: `memory_type`, `shard`, `aggregate`, `feature_type`, `sram_base`, and `dram_base`. There are two memory types: off-chip memory instruction and on-chip memory instructions. Off-chip memory instructions are used to program the automatic address generators within the Shard Edge Fetch, Shard Feature Fetch, and Shard Writeback Units, while on-chip memory instructions are used to program the automatic address generators within the Shard Compute Unit. The `shard` field indicates either the graph should be accessed in a sharded manner (such as when aggregating features) or in a streaming fashion (such as when applying a normalization to *all* nodes without any edge information). The `feature_type` and `aggregate` fields indicate what type of feature is being loaded and how it being used, which has implications on what address should be generated. For example, a node that is being aggregated must be written back and reloaded. Finally, the bases indicate where in memory the features begin.

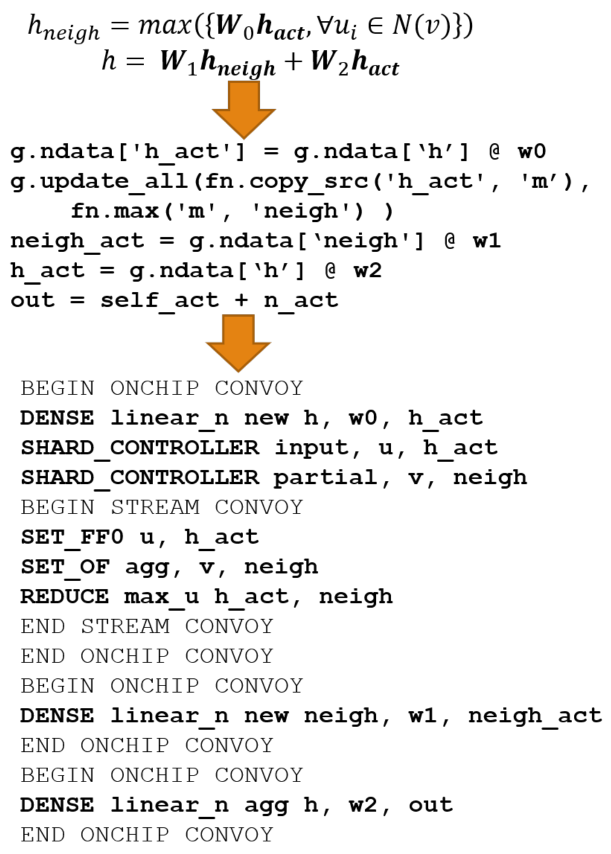
Graph compute instructions. A graph compute instruction is broken up into four fields: `unit`, `op`, `src0`, `src1`. `unit` specifies which compute unit the instruction is associated with (*e.g.*, the `Reduction Unit`) and `op` specifies the operation that unit should perform (*e.g.*, sum). The source field(s) indicate either the input should be read from the associated `Fetcher` or if it should be read from the internal register. Note that there is not destination field, as each unit always feeds its output directly to its successor and the addresses are determined by the address generators using the graph memory instructions.

6.3 GNNerator Execution Model

In this section, we describe the execution model adopted by GNNerator.

6.3.1 Graph Execution Model Overview

In order to demonstrate the execution model, we use Figure 6.2 as an illustrative example, showing a simple GNN network defined in pseudocode, using the Deep Graph Library (DGL)



$$h_{neigh} = max(\{\boldsymbol{W}_0 \boldsymbol{h_{act}}, \forall u_i \in N(v)\})$$
$$h = \boldsymbol{W}_1 \boldsymbol{h_{neigh}} + \boldsymbol{W}_2 \boldsymbol{h_{act}}$$

```
g.ndata['h_act'] = g.ndata['h'] @ w0
g.update_all(fn.copy_src('h_act', 'm'),
    fn.max('m', 'neigh') )
neigh_act = g.ndata['neigh'] @ w1
h_act = g.ndata['h'] @ w2
out = self_act + n_act
```

```
BEGIN ONCHIP CONVOY
DENSE linear_n new h, w0, h_act
SHARD_CONTROLLER input, u, h_act
SHARD_CONTROLLER partial, v, neigh
BEGIN STREAM CONVOY
SET_FF0 u, h_act
SET_OF agg, v, neigh
REDUCE max_u h_act, neigh
END STREAM CONVOY
END ONCHIP CONVOY
BEGIN ONCHIP CONVOY
DENSE linear_n new neigh, w1, neigh_act
END ONCHIP CONVOY
BEGIN ONCHIP CONVOY
DENSE linear_n agg h, w2, out
END ONCHIP CONVOY
```

Figure 6.2. A GraphSage-based network defined mathematically (top), using the popular Deep Graph Library (DGL) framework, and in GNNerator machine code. The bold machine code indicates compute and memory instructions, while the other code contains pseudo-instructions that provide instructions to the compiler.

framework, and GNNerator code. Notice that the GNNerator code contains instructions that are not defined in Section 4.3.1. These pseudoinstructions are compiler directives, which are used to define *on-chip convoys* and *stream convoys*, the two basic units of computation in the GNNerator execution model.

On-chip convoy. An on-chip convoy encompasses one pass through a sharded graph. This pass defines a ”shard program” made of multiple stream convoys.

Stream convoy. An stream convoy defines the longest chain of instructions executed together in the `Shard Compute Unit` such that the output of one instruction is consumed by the following instruction. Thus, a stream convoy has a maximum size of three instructions–

the case where the output of the `Apply Unit` is consumed by the SFU which is in turn consumed by the Reduce Unit and an explicit minimum size of one. Note that a stream convoy actually always has an implicit size of three instructions, where a `nop` instruction is used for units that are unused.

For each shard on-chip convoy, the Graph Engine Controller sets the internal automatic address generation units within the various shard fetch/writeback units based on the `SHARD_CONTROLLER` instructions. For example, in Figure 6.2, we set feature `h` as an input source node feature. As a result, when GNNerator is processing shard (U, V), it will ensure that feature `h` is loaded from on-chip for nodes $[U \cdot n : (U+1) \cdot n)$. Similarly, we set feature `neigh` as a partial output for destination nodes. Partial outputs indicate that this feature is used in a reduction operation and must be written back/reloaded as needed when the shard controller moves to a new set of destination nodes (*i.e.*, going from shard (U, V) to shard $(U, V+1)$.

Within each shard on-chip convoy, the chain of instructions defined by the stream convoy(s) is applied to a shard before moving on to the next shard. Similar to the `SHARD_CONTROLLER` instruction, the `SET_*F` instructions set the internal automatic address generation units of the associated Feature Fetchers within the Shard Compute Unit. These fetchers combine this information with the edge information provided by the Edge Fetcher stepping through the shard's edge list in order to supply the desired inputs to the compute units, which then apply the GNNerator compute instructions contained within the stream convoy. In the working example, we can see that `Feature Fetcher 0` is programmed to load the feature `h` for the given source nodes (thus the Edge Fetcher provides it with the source ID from an edge) and the `Modified Feature Fetcher` is programmed to store the output feature `neigh` which represents the aggregated feature. Note that in the PUSH dataflow, this feature may also have to reload the output feature as well. The inputs are first provided to the Apply Unit because `Feature Fetcher 0` is connected to that unit. Since there is no `APPLY` instruction– nor `SFU` instruction)– these units simply perform `nop` instructions in order to pass the input to the Reduce Unit.

Shard Order In order to compute a given sharded graph, GNNerator must step through the two-dimensional grid depicted in Figure 3.5. This can be done in a source-major or

Table 6.1. Analytical description of the cost of different ways to traverse the two-dimensional sharded graph grid

	Read Cost	Write Cost
SRC Stationary	$S * I + (S-1) * S - S + 1$	$S^2 - S + 1$
DST Stationary	$(S^2 - S + 1) * I$	S

destination-major manner, each with different costs. When traversing in a source-major fashion (*i.e.*, across a row of the shard grid), a set of *source* vertices and their corresponding feature(s) are loaded on-chip and remain on-chip for the entire row. The destination vertices, however, must be written back and reloaded as we move from shard to shard. Conversely, when traversing in a destination-major fashion, a set of *destination* vertices and their corresponding feature(s) are loaded on-chip and remain on-chip until they are done aggregating, while the source features must be reloaded as we move from shard to shard. Note that when we move to a new set of source vertices, we do not have to write-back like we do in the destination case. Further, notice that off-chip transfers can be saved by navigating this grid in an S-pattern. Assuming this S-pattern, we show the read and write costs associated with the two different orders in Table 6.1, where S is the number of shards and I is the maximum number of input features required to be on-chip at one time. Note that there is an additional factor, the number of nodes per shard, but this factor cancels out across the different orders. With these costs, assuming equal costs for reads and writes, we can analytically determine the best ordering; destination stationary is best when there is only one input feature, the ordering are equal at two input features, and row stationary is better if there are more than two input features.

Feature Dimension Blocking. To minimize data transfers (*i.e.*, minimize S in the equations in Table 6.1), we would like to maximize the number of nodes that can be held on-chip at one time– that is, we would the shards to be as large as possible. However, large shards are difficult to fit on-chip in the context of GNNs, since each node is associated with one or more feature, each of each can be of a high dimension. Fortunately, the data storage requirement imposed by high dimension features can be reduced by exploiting the insight that these dimensions are treated *independently* during the graph processing phase of GNNs. As

```
Algorithm 1: Standard Shard-based Algorithm
  Input: Sharded Graph $G$; Width of Shard Grid $W$, Height of Shard Grid $H$,
         Hidden Dimension Size $D$
  Output: Updated graph $G$
1 for $w$ in range $(W)$ do
2     for $h$ in range $(H)$ do
3         for $d$ in range $(D)$ do
4             $G_{agg}.Shard(h,w)[d] = \textbf{Aggregate}(G.Shard(h,w)[d])$
5     $G.Shard(:,w)[\,:\,] = \textbf{FeatureExtract}(G_{agg}.Shard(:,w)[\,:\,])$
```

```
Algorithm 2: Dimension-blocking Algorithm
  Input: Sharded Graph $G$; Width of Shard Grid $W$, Height of Shard Grid $H$,
         Hidden Dimension Size $D$, Dimension Block Size $B$
  Output: Updated graph $G$
1 for $d$ in range $(D)$ by $B$ do
2     for $w$ in range $(W)$ do
3         for $h$ in range $(H)$ do
4             for $b$ in range($B$) do
5                 $dim = d + b$
6                 $G_{agg}.Shard(h,w)[dim] = \textbf{Aggregate}(G.Shard(h,w)[dim])$
7         $G.Shard(:,w)[\,:\,] = \textbf{FeatureExtract}(G_{agg}.Shard(:,w)[d:d+B], G.Shard(:,w)[\,:\,])$
```

a result, the graph processing computations can be *blocked* such that only a portion of the dimensions are required on chip at one time. In this execution model, the graph metadata (*i.e.*, edge list) is loaded once. An on-chip convoy is then executed for the entire sharded graph for a block of dimensions– say, $[0, blockD]$). After this finishes execution, we move onto the next block of dimensions– $[blockD, 2 \cdot blockD)$– and re-execute the on-chip convoy for the next block, while retaining the same graph metadata on chip. This way, we can reduce the number of data transfers necessary, at the expense of additional memory accesses to the on-chip edge list, since it must be repeated multiple times to process the full feature, as well as additional memory accesses in the Dense Engine, as it must reload the partial sums when moving on to the next set of $blockD$ dimensions.

6.3.2 Dense Execution Model Overview

The `GNNERATOR` execution model for the dense engine (*i.e*, feature extraction) is comparatively much more simple; it simply serially executes the set of dense instructions contained within the on-chip convoy. The interaction between the Dense Engineand the Graph Engineare then handled by the `GNNerator Controller` as described in Section 6.2.3.

6.3.3 GNN Accelerator Design Space

In addition to the above, the `GNNERATOR` execution model can be used to explore a variety of the hyperparameters that defines the design space for GNN accelerators.

Edge Dataflows. In the *pull* edge dataflow, the edge list is assumed to be sorted such that all edges with a common *destination* node are sequential in a GPE's edge list memory. That destination node is then pinned to the given PE until it has finished aggregating contributions from all of its neighbors. Thus, the `Shard Edge Feature` determines which GPE's graph metadata memory to store an edge in by simply taking the modulo of the destination node's ID with respect to the number of GPEs. Since the assigned GPE contains all edges with the same destination, a GPE always only has to access its own modified feature memory, which eliminates any possible conflicts and the need for an atomic lock. However, the input feature for a given edge may be located in a different GPE (*i.e.*, the source node and destination node map to different GPEs because $\mathbf{e}_{\text{i}} \bmod \#GPEs \neq \mathbf{e}_{\text{j}} \bmod \#GPEs$) and so each GPE must be able to read every other GPE's input feature. In the *push* edge dataflow, on the other hand, the edge list is assumed to be sorted such that all edges with a common *source* node are sequential in a GPE's edge list memory. That source node is then pinned to the given PE until it has finished pushing its contributions to all of its neighbors. Thus, the `Shard Edge Feature` determines which GPE's graph metadata memory to store an edge in by simply taking the modulo of the source node's ID with respect to the number of GPEs. Unlike in the pull edge flow, in the push edge flow, a GPE must be able to access the modified feature memory of every other GPE since the output feature associated with the edge's destination node is not guaranteed to be in the same GPE's modified feature memory (again, when $\mathbf{e}_{\text{i}} \bmod \#GPEs \neq \mathbf{e}_{\text{j}} \bmod \#GPEs$). Counter-intuitively, each GPE must also be able to access the input feature memory of every other GPE, as the input feature for a given edge *may* still be located in a different GPE, *despite* the edges being assigned to GPE's based on the source node ID. This is because, unlike aggregation, which is always performed on the destination node's feature, an input feature can be associated with *either* a source node or a destination node.

6.4 Programming GNNerator

To program `GNNerator`, we develop a compiler and a runtime environment.

6.4.1 Compiler

The compiler takes as input a GNNerator program, which can be obtained from a graph neural network framework such as Deep Graph Libary (DGL). This program, consisting of GNNerator ISA pneumonics as well as pseudoinstructions indicting off- and on-chip convoys, defines the graph neural network to be executed. A description of the `GNNerator` micro-architectural configuration is also provided as an input to the compiler. Using this information, the compiler first shards the graph into small subgraphs to be executed.

This is done by first analyzing the code to determine the maximum number of node features that must be resident on-chip for the given network. In the example code in Figure 6.2, this would be one input node feature and one output node feature: `h_act` and `neigh`. The maximum number of node features in turn sets the maximum number of nodes that can be on-chip, as there must be enough memory for each node to have that number of features on-chip.

However, this only ensures that the nodes can fit on-chip, but we must also make sure there is space for the edges as well. To do, we attempt to shard the graph based on that optimistic nodes per shard. When attempting to shard the graph, we determine the maximum number of edges that would be required to support that many nodes per shard. Note that this completely dependent on the graph structure and cannot be determined a priori. If the maximum number of edges required fits on-chip, the graph sharding processes is complete. Otherwise, the maximum number of edges is used to estimate the connectivity of the graph,

Algorithm 3: Algorithm to determine number of nodes that can be kept on-chip

```
Input: Graph $G$; GNNerator configuration $params$; number of input features $ifeat$;
       number of output features $ofeat$
Output: Number of nodes per shard $N_{shard}$
1  $N_{shard}^{agg} = params.omembytes/(params.blockD \cdot ofeat)$
2  $N_{shard}^{inp} = params.imembytes/(params.blockD \cdot ifeat)$
3  $E_{max} = params.E_{max}$
   /* First, try the largest possible $N_{shard}$                    */
4  $N_{shard} = min(N_{shard}^{agg}, N_{shard}^{inp}, |G.V|)$
5  $E_{shard} = COUNT\_EDGES(G, params, N_{shard})$
6  if $E_{shard} < E_{max}$ then
7      return $N_{shard}$
   /* Try based on sampled connectivity                               */
8  $c = E_{shard}/N_{shard}^2$
9  $N_{shard} = \sqrt{E_{max}/c}$
10 $E_{shard} = COUNT\_EDGES(G, params, N_{shard})$
11 if $E_{shard} < E_{max}$ then
12     return $N_{shard}$
   /* Default to worst case                                           */
13 $N_{shard} = \sqrt{E_{max}}$
14 return $N_{shard}$
```

which is then used to estimate a new number of nodes per shard. We again attempt to shard the graph, determining the maximum number of edges that would be required to support that many nodes per shard. If the edges fit on-chip, we terminate the sharding process. Finally, if the sampled connectivity attempt does not work, we assume a worst case situation, wherein the nodes per shard is set based on assuming that the network is fully connected, thus guaranteeing the graph shards produced will fit on-chip. This process is summarized in Equation 3.

After sharding the graph, the compiler produces the final machine code, using the parameters determine during the graph sharding process to populate the final symbols, *e.g.,* the location in on-chip memory of each of the node features.

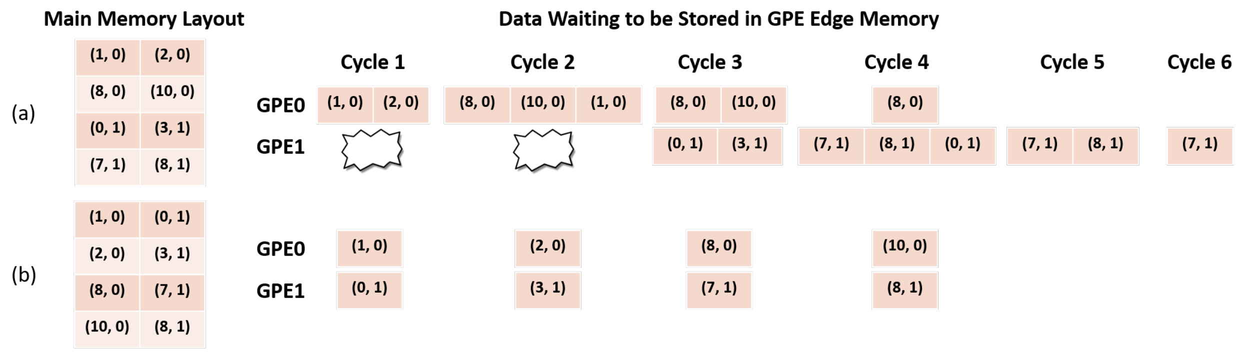


Figure 6.3. In this example, the main memory has a width of two edges. (a) Using a naive edge memory layout where the edges are sorted based on the destination node results in idle cycles for GPE1 while GPE0 is being loaded, as well as requires enough space to buffer three edges at once. (b) Using a interwoven memory layout eliminates the idle cycles and reduces the number of edges that must be buffered to only one edge.

Edge Layout Optimization

As discussed in Section 6.2, the `Shard Edge Fetcher` reads a sharded edge list from main memory and distributes these edges to the private on-chip edge memories of each GPE. In order to realize an efficient ”pull”-based (”push”-based) dataflow, all of the edges with the same destination (source) should be consecutive in the on-chip edge memories, as described in Section 6.2. However, storing edges naively in a consecutive fashion like that results in inefficient memory transfers. There are two main reasons for this. First, as discussed in Section 6.2, each GPE contains an edge memory and edges are distributed to these edge memories such that all edges with the same source (destination) are stored in the same edge memory. If these edges are consecutive in main memory, then usually only one edge memory will be used at a time while the other edge memories are idle, waiting for the set of edges that map to them. Further, even the accesses to the active memory are inefficient. This is because the DRAM width is often much larger than the memory width of the on-chip edge memories, which have a width wide enough to allow one edge access per GPE per cycle. As a result, if all of the edges loaded in a DRAM transaction are destined for the same GPE's on-chip memory, there will be many conflicts, As a result, the architecture would require more on-chip memory, such as queues, to store the stalled DRAM transactions as they are processed. To avoid this, the compiler has an optimization pass after the sharding step, wherein the edges are rearranged in order to pack as many edges from *different* GPEs into a DRAM transaction as possible. This is accomplished by first forming # GPE different edge lists, where each edge list consists of the edges to be assigned to that GPE. These edge lists are then recombined into a single edge list by storing one edge from each GPE in a round-robin fashion. This results in both more on-chip edge memories being active at one time as well as reducing the required on-chip edge storage, as shown in the visual comparison between the unoptimized and optimized cases in Figure 6.3.

Feature Memory Layout Optimization

For the same reasons, feature memories should not be laid out sequentially in memory– that is, it is inefficient to store an entire feature for node 0, followed by the same feature for node 1, and so on. This would again result in only one GPE's on-chip memory being accessed at one time and increase the amount of on-chip storage needed to buffer DRAM transactions. We use a similar round robin strategy to optimize the memory layout, with the additional constraint that the features are divided into chunks of `# of Lanes` in order to match the width of the on-chip scratchpads.

6.5 Methodology

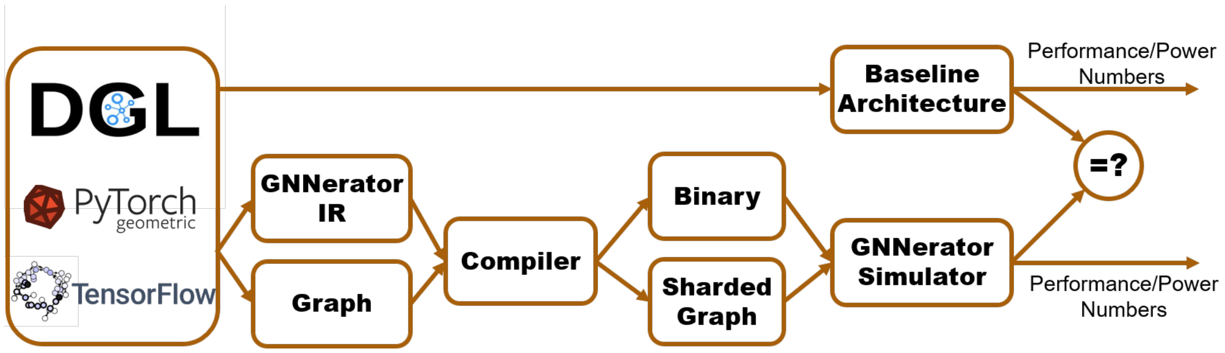


Figure 6.4. An overview of the simulation framework used

Simulation infrastructure. In order to provide a simulation infrastructure for evaluating GNNerator we developed both a cycle-level simulator as well as a prototype compiler and runtime. We developed the cycle-level simulator of GNNerator using the PyMTL3 framework [141]. We implement cycle-level models of all of the Graph Engine and GNNerator Controller modules shown in Figure 6.1 and integrate the cycle-accurate SCALe-Sim simulator for the Dense Engine. In order to run a given GNN, the GNN is first transformed into a GNNerator intermediate representation. This IR, along with the graph(s) to be used in inference and a configuration file specifying the desired GNNerator hardware and software parameters, are then compiled into a binary and pre-processed graph. The GNNerator simulator then ingests these inputs, and produces both a functional output, which can be compared to a golden model running on a commodity platform, as well an estimate of the performance and power of the architecture candidate. This produced is summarized in Figure 6.4.

Table 6.2. Summary of Graph Datasets

Dataset	Vertices	Edges	Feature Dim.	Size
CORA	2708	10556	1433	15.6 MB
CITESEER	3327	9104	3703	49 MB
PUBMED	19717	88648	500	40.5 MB

Table 6.3. Summary of Graph Neural Networks

Network	Hidden Layers	Hidden Dimension Size
GCN [98]	1	128
Graphsage [99]	1	128
GraphsagePool [99]	1	128

Benchmarks. In Table 6.2, we summarize the input graph datasets used in our experiments. These datasets represent standard graph datasets used in GCNs. Note that although the total number of vertices may be small, most of the datasets cannot fit on-chip in either platform due to the large feature dimension sizes. We run each of these input graph datasets on the three graph neural network architectures outlined in Table6.3. We run these benchmarks using the Deep Graph Library (DGL) [53] with the PyTorch backend.

Platforms. For our experiments, we use an implementation of GNNerator that has a Graph Engine with 32 GPEs, each with 32 lanes. Thus, the Graph Engine can process thirty-two nodes at one time, with thirty-two dimensions of each node being processed. The Graph

Table 6.4. Summary of Compute Platforms

	GPU	GNNerator	HyGCN
Compute	4352 cores @ 1.5 Ghz	32x32 Graph, 64x64 Dense @ 1GHz	32x16 Graph, 32x128 Dense @ 1GHz
Peak Compute	13 TFLOPs	10 TFLOPs 2 for Graph, 8 for Dense	9TFLOPs (1 for Graph, 8 for Dense)
On-chip Memory	29.5 MiB	30 MiB 24 MiB Graph, 6 MiB Dense	24 MiB
Off-chip Memory	616 GB/s	256 GB/s	256 GB/s

Engine contains 24 MiB of on-chip memory, distributed equally between the Input Feature, Modified Feature, and Edge scratchpads. The Dense Engine consists of a systolic array of size 64x64, resulting in sixty-four nodes and sixty-four dimensions per node being processed together. It contains 6 MiB of on-chip memory, equally distributed between the input, output, and weight buffers. We use a Turing-based RTX 2080-Ti as our GPU baseline. The 2080-Ti contains 68 Streaming Multiprocessors (SMs), contributing a total of 4352 CUDA cores. Counting both the register file as well as the shared memory, the 2080-Ti has 29.5MiB of on-chip storage. As a result, the peak compute and on-chip storage capacities of `GNNerator` and the GPU baseline are reasonably balanced, at 10 TFLOPs with 30 MiB memory and 13 TFLOPs and 29.5 MiB, respectively. This is summarized in Table 4.4.

6.6 Evaluation

In this section, we present the results of our experiments evaluating the benefits of `GNNerator`.

6.6.1 Performance

Overall Performance. Figure 6.5 shows the normalized speedup with respect to the 2080-Ti.

We consider two variations of `GNNerator`: the standard baseline `GNNerator` uses the proposed novel dimension-blocking scheme described in Section 6.3 with *blockD* set equal to the width of the Dense Engine (*i.e.*, 64), while `GNNerator-full` does not use dimension blocking. `GNNerator-full` demonstrates speed up of `0.7-37x` over the GPU baseline, while `GNNerator` has a speedup of `1.7-37x`. `GNNerator`'s additional performance improvement through the

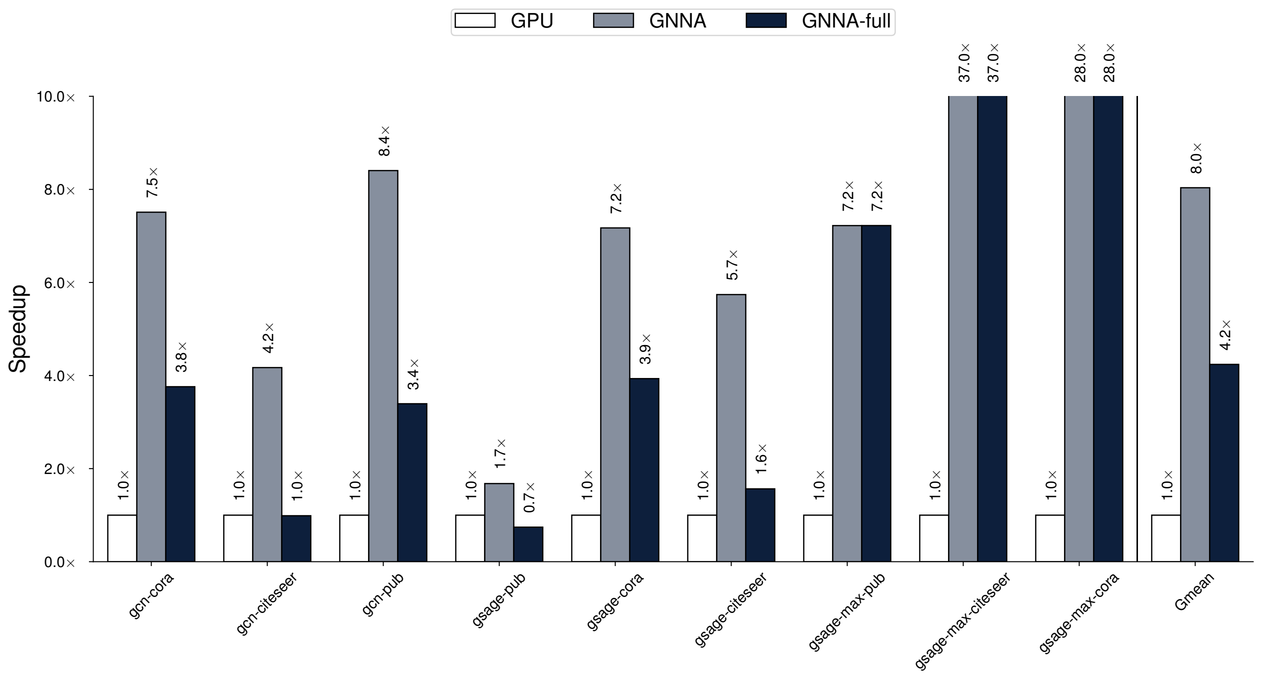


Figure 6.5. GNNerator achieves an `8x` speed up over the 2080-Ti baseline. Roughly half of this speedup results from the specialized architecture (GNNerator-full) and the other half comes from the novel dimension blocking dataflow (GNNerator).

use of dimension-blocking stems from two main sources. First, dimension-blocking allows for more nodes' features to be held on-chip, reducing the memory bottleneck of transferring these features on- and off-chip. Second, dimension-blocking reduces the amount of time the Dense Engine must wait for the Graph Engine to finish aggregating a node's neighborhood, as the Graph Engine only has to aggregate a small fraction of the dimensions before the Dense Engine can begin.

Table 6.5. Speedups of HyGCN and GNNerator relative to a V100

	Cora	**Citeseer**	**Pubmed**
HyGCN	2.1x	0.9x	0.8x
GNNerator-full	3.8x	0.7x	0.8x
GNNerator	8.0x	2.9x	1.8x

HyGCN Comparison. Finally, we compare GNNerator to HyGCN. We chose HyGCN as the comparison platform, as it is the most similar architecture to GNNerator. HyGCN used Pytorch-Geometric running on a V100 as baseline. Thus, in order to provide a fair comparison, we use the V100 run times obtained in DGL's whitepaper [53], in which they provide statistics for both DGL and PyG running on a V100, in order to generate the baseline used in Table 6.5. As demonstrated in the table, GNNerator-full and HyGCN are quite similar in performance, with comparable performance improvements over the GPU baseline. When utilizing dimension-blocking, however, GNNerator consistently greatly outperforms HyGCN, with an average speedup over HyGCN of `3.15x`.

6.6.2 Scaling

In addition to the baseline network configurations outlined in Section 6.5, we examine the effects of scaling to larger networks. Specifically, we examine the effect of increasing the hidden layer size, as well as increasing the number of layers. We also investigate a larger GNNerator design.

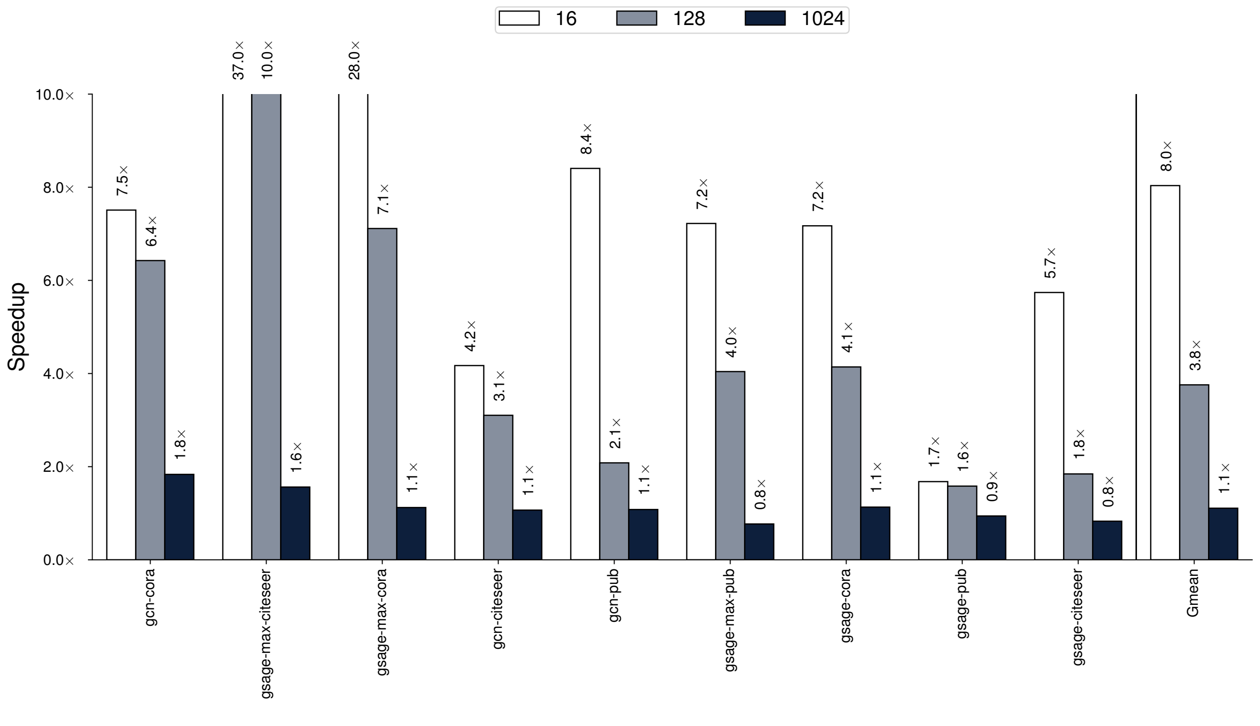


Figure 6.6. GNNerator's speedup reduces from an average of `8x` to 1.1x as the hidden layer size increases from a dimension size of 16 to 1024

Scaling Hidden Layer Size. First, we hold the number of layers constant at one and increase the hidden layer size. As seen in Figure 6.6, as the hidden dimension size increases, the performance gap between GNNerator and the GPU baseline decreases.

To examine why we obtained this result, we refer to Figure 3.7, which demonstrates that as the hidden layer size increases, the run time becomes dominated more and more by the linear layers. The GPU is particularly well suited to the linear layers, since it has over `2x` the bandwidth and `1.5x` the compute available for DNN execution. As a result, the GNNerator's speedup over the GPU decreases as the workload becomes more "dense"-dominated– that is, as the hidden layer size increases. Note that even at small hidden dimension sizes, GNNerator still sees large speedups over the GPU baseline for the Graphsage-Max network. This is because the GPU cannot efficiently perform the extremely tall but narrow matrix multiplies that results from the small hidden dimension sizes.

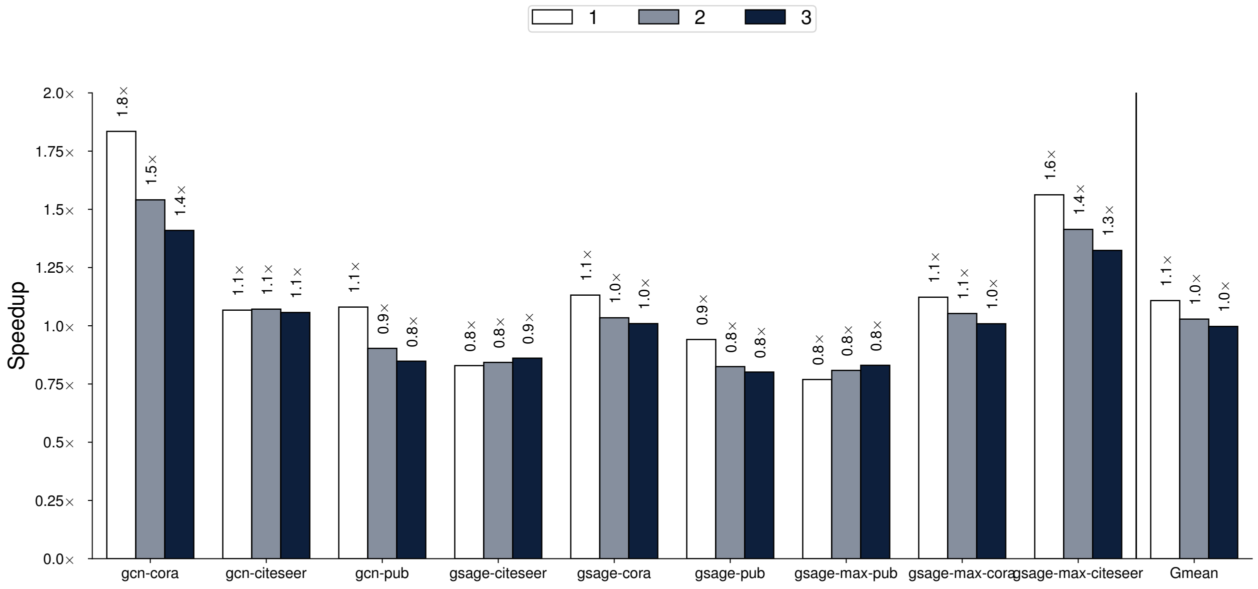


Figure 6.7. As the number of hidden layers with a large dimension size increases, the benefits of a specialized accelerator over a GPU continues to diminish, with no performance benefits at three layers.

Scaling Number of Hidden Layers. We also scale the network size by the number of hidden layers. Consistent with the approach taken in most of the literature, each hidden layer contains the same number of hidden dimensions. In our experiments, we set this hidden dimension size to 1024 dimensions. Generally speaking, as the number of hidden layers increases, the performance benefits of a specialized architecture decreases. This is consistent with the idea that as GNNs become larger and deeper, they tend to become more GPU-friendly.

Scaling Up `GNNerator`. Next, we examine the question of where to invest additional hardware resources in order to maximize the performance return on that investment. We present three possible versions of a "next-generation" `GNNerator`. One version doubles the amount of on-chip memory in the Graph Engine, allowing for more larger shards to be held on-chip. Another version doubles both the height and width of the Dense Engine, increasing the compute available for the linear layers.The final version doubles the bandwidth available for the shared feature memory DRAM. Note that these versions were obtained by investigating the bottlenecks of each engine; since the Graph Engine was rarely stalled by the compute units and the Dense Engine was not memory-constrained, we do not investigate these versions.

We find that the best investment is to increase the size of the Dense Engine, resulting in an average speedup of `1.8x` over the baseline design. This is driven primarily by large speedups for the larger hidden dimension sizes, as seen in Figure 6.8. Increasing the feature memory bandwidth results in a modest speedup of `1.2x`, driven primarily by larger speeds for small to moderate hidden dimension sizes. Increasing the on-chip memory for the Graph Engine does not result in a noticeable speedup across all of the datasets. However, it does result in a modest average speedup of `1.16x` for the largest dataset considered, Pubmed.

Figure 3.7 helps to explain why increasing the Dense Engine's compute results in the largest speedups, particularly for large hidden dimension sizes. It also helps to explain why the increase in feature memory bandwidth does not also result in large speedups at large hidden dimension sizes like it does for smaller sizes; at the larger sizes, the run time is dominated more by the Dense Engine, which is not helped by an increase in the feautre memory bandwidth, except for when first loading (storing) the initial input features (output features).

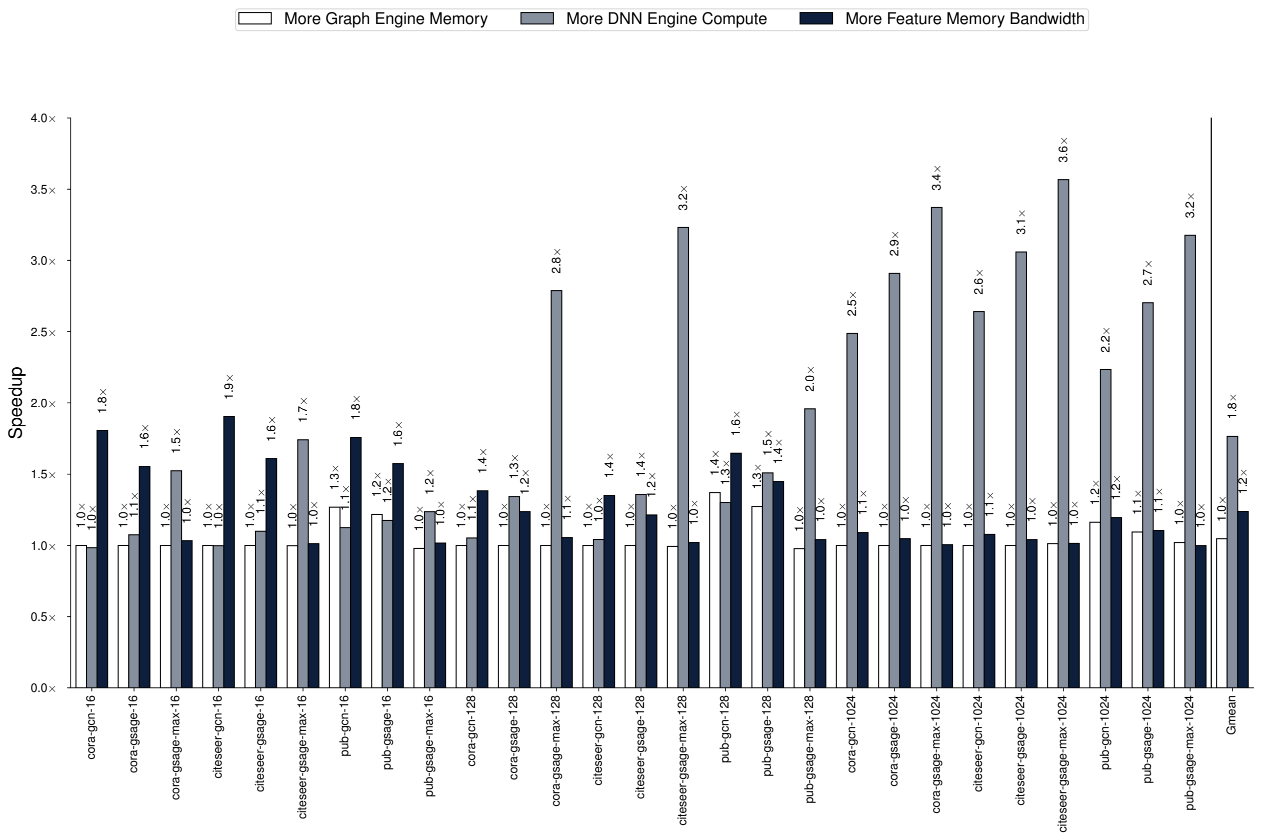


Figure 6.8. Adding more DNN Engine Compute results in the largest speedups relative to the baseline GNNERATOR architecture, with speeds up of `1.1-3.6x`.

6.6.3 Architecture Design Space Exploration

In this section, we explore key architectural parameters in the design of GNN accelerators: the Dense Engine dataflow, the Graph Engine dataflow, and the level of coupling between the two engines.

DNN dataflow. The dataflow of the DNN Engine is an important parameter that has been largely ignored in the literature on GNN accelerators to date: there is no reference at all to it at all in [59]–[61]. However, large gains in performance (`2.1x` on average) can be achieved through careful selection of dataflow. Specifically, output- and weight-stationary DNN dataflows provide substantial speed-ups over the input-stationary dataflow.

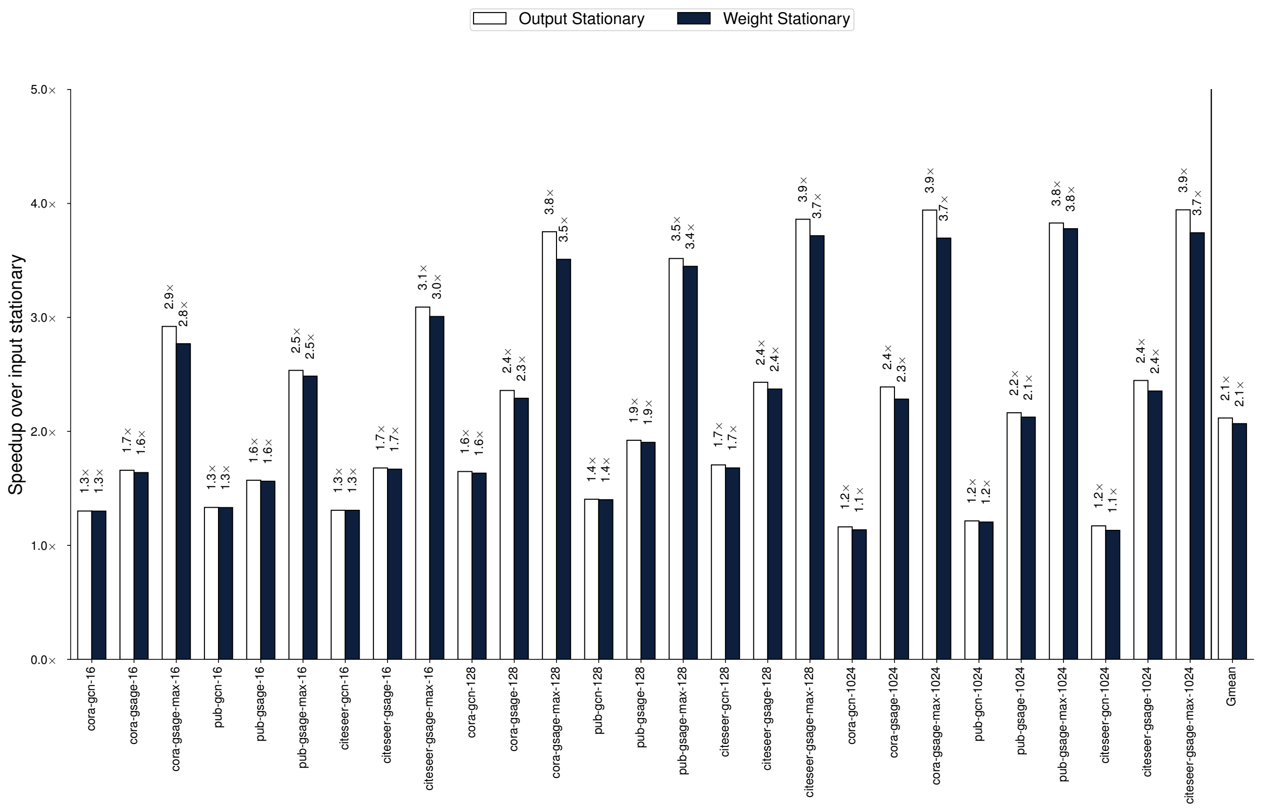


Figure 6.9. Output- and weight-stationary DNN dataflows greatly outperform the input-stationary dataflow, particularly for pooling-based benchmarks.

This is particularly true for the pooling-based benchmarks, *i.e.* the Graphsage-Max network, as well as for larger hidden dimensions, due to the fact that the DNN Engine is responsible for a larger fraction of the run time in the network, and so an increase in performance of the DNN Engine translate to a larger end-to-end increase in performance. The finding that input stationary does not perform as well as the other dataflows is not surprising; [142] two general heuristics for DNN workloads: (1) output stationary tends to be higher performer than the other dataflows and (2) input stationary performs worse when the input matrix is much larger than the weight matrix, which is typically the case in these networks. Thus, these heurisitics seem to hold well for GNN workloads as well.

Push vs Pull Dataflow. The style of communication–either push or pull-based communication– used in traversing a graph algorithm can often result in considerable variations in the runtime of the algorithm [143]. To this end, we compare a push-based communication pattern with the pull-based communication pattern used in GNNerator as well as other proposed GNN accelerators.]

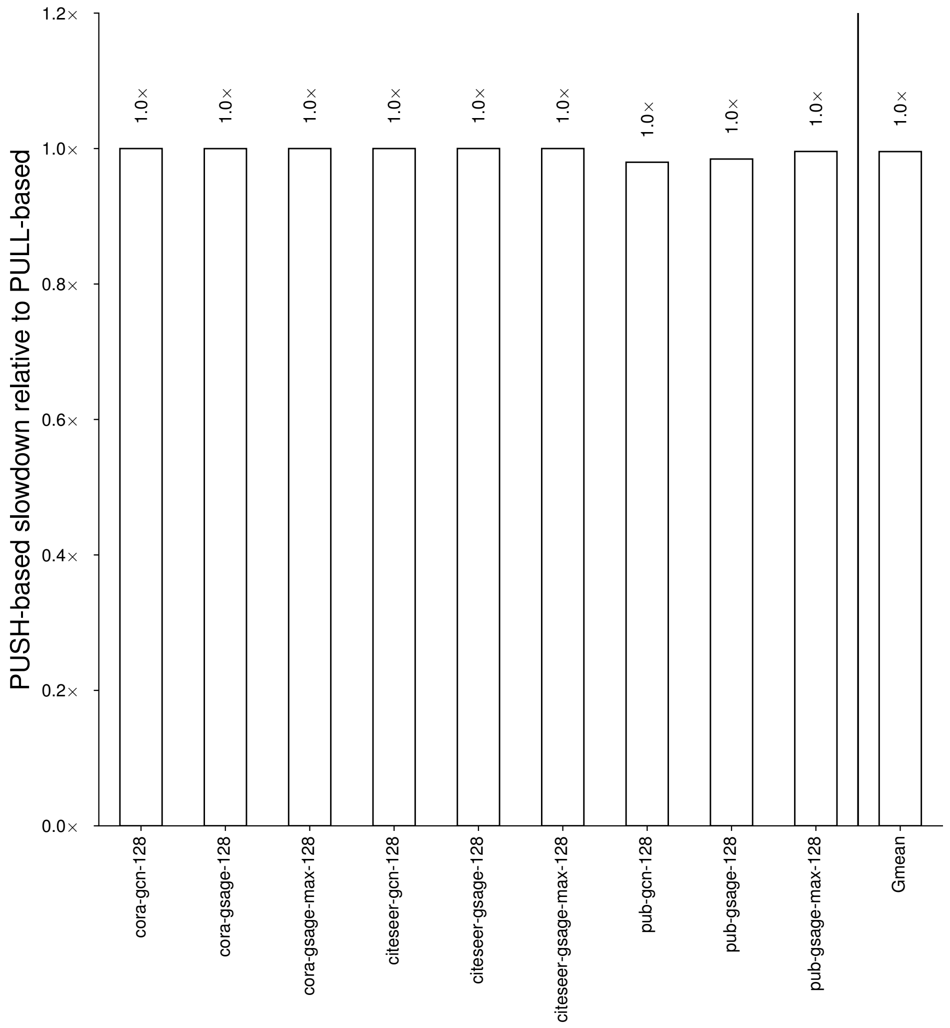


Figure 6.10. Using a push-based edge dataflow results in a negligible slow down compared to a pull-based baseline

As seen in Figure 6.10, there is a negligible slowdown seen when processing in a push-based style. Note that although Figure 6.10 only shows results for a hidden layer size of 128, this holds across all hidden layers sizes tested. However, at the same time, as demonstrated in Figure 6.11, a push-based communication results in a considerable (average `1.3x`) increase in the amount of time GNNerator is stalled for compute, due to stalls for the lock required in push-based communication as described in Section 6.2. This discrepancy is explained by the fact that GNNerator is rarely stalled for compute to begin with, and so the increase does

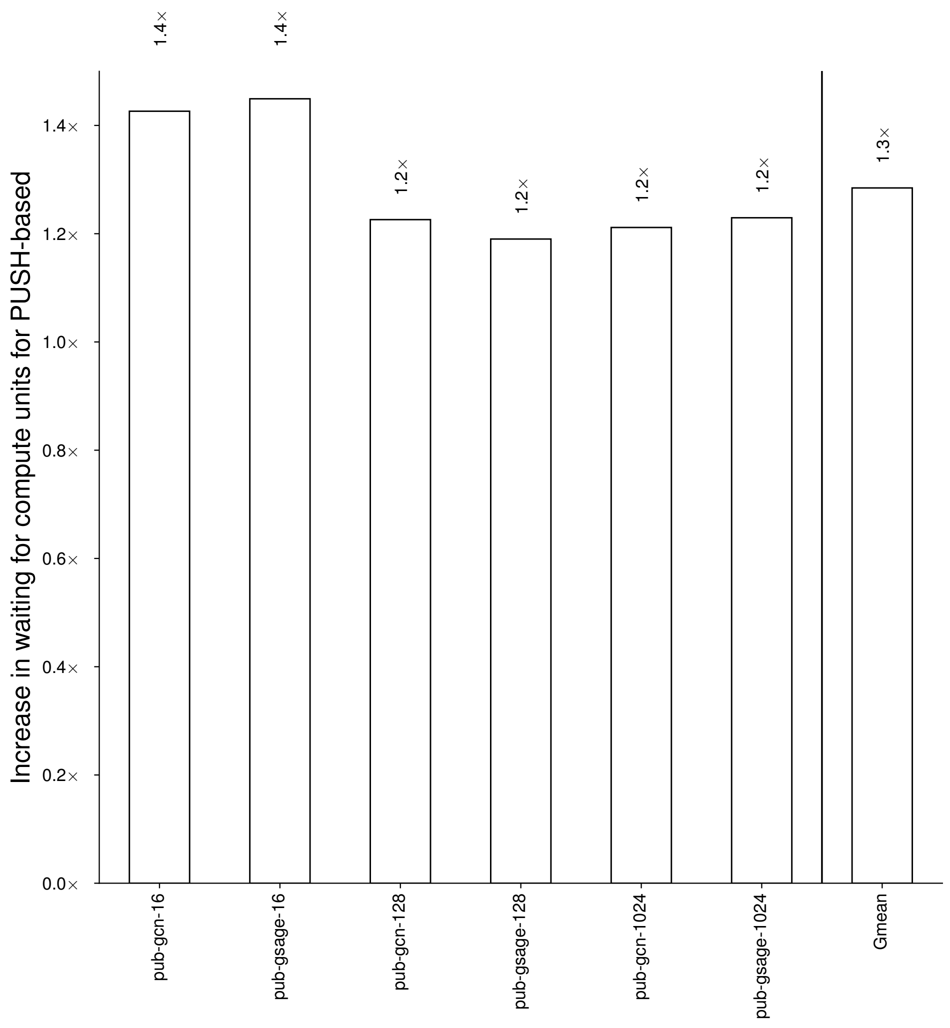


Figure 6.11. For workloads where there are compute stalls in the Graph Engine, these stalls increase under a push-based dataflow

not affect the total run time much; GNNerator's Graph Engine is much more often stalled waiting either for the Dense Engine compute or for features to be loaded from memory.

6.7 Conclusion

GNNs are a promising new area of machine learning that aims to bring the success of deep learning in non-Euclidean domains to graph-based problems. In this work, we detail the limitations of current hardware to efficient perform GNNs and propose GNNerator, a specialized hardware accelerator for GNNs that is able to exploit the abundant intra- and inter-node parallelism inherent in GNNs. GNNerator utilizes feature dimension-blocking, a novel GNN dataflow that allows for processing more nodes in a graph on-chip at one time. We evaluate GNNerator on a suite of benchmarks and demonstrate significant performance benefits over GPUs, as well as a recently proposed GNN accelerator.

7. HARDWARE-AWARE PERFORMANCE ESTIMATORS FOR HETEROGENEOUS EDGE SYSTEMS

7.1 Introduction

Deep Neural Networks (DNNs) have established themselves as the state-of-the-art for a variety of machine learning problems, with applications including computer vision, natural language processing, web search and product recommendation. This success has led to an explosion in the number of DNNs proposed for a given problem (e.g., image classification), each of which has its own unique topology, memory footprint, arithmetic intensity, *etc.* Given the increasing ubiquity as well as the energy-intensive nature of DNN workloads, there is tremendous interest in finding the most efficient mapping of a DNN to a given platform. This is especially important in the context of energy, size and cost constrained edge processing platforms. However, this goal is complicated by the diverse nature of edge platforms; DNNs are being deployed on thousands of different edge computing platforms [102], each of which may consist of a heterogeneous set of hardware IPs, including CPUs, GPUs, DSPs, and dedicated deep learning accelerators (DLAs).

Moreover, many heterogeneous edge platforms offer multiple *hardware operating conditions*, allowing the designer to choose the number of active cores, core frequencies, memory controller frequency, *etc.* This offers an expanded design space, as shown in Figure 7.1, allowing the designer to select from a wider range of design points which may offer a better trade-off between energy consumption and processing speed.

Given such a large design space (combination of choice of DNN, hardware operating condition and DNN-to-HW mapping), it is infeasible to exhaustively determine the optimal solution even by running each combination on the physical device. This motivates the use of *estimators*, which can be used to *predict* latency, power, energy, *etc.* [1]. Performance estimators have several applications, including determining the best DNN for a given platform, determining the best mapping and hardware configuration, and hardware-aware neural architecture search (hardware-aware NAS) [106], [144].

[1]↑For brevity, we refer to latency/power/energy estimators generically as performance estimators.

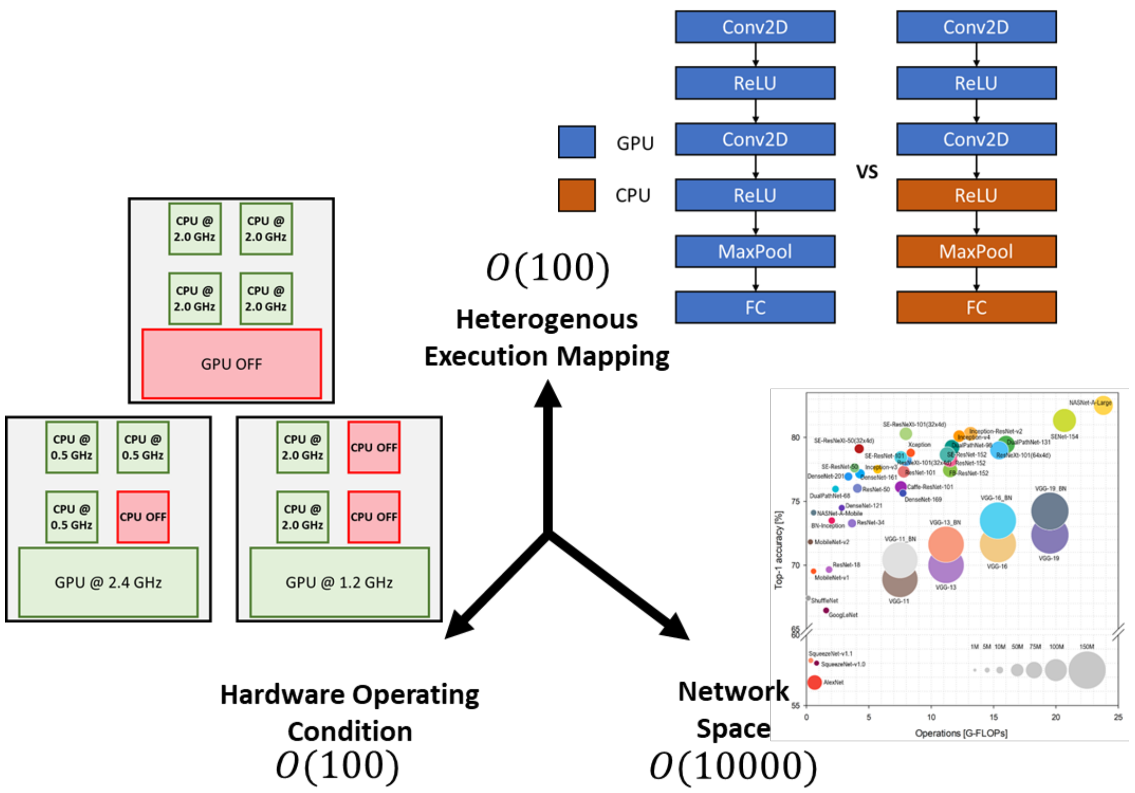


Figure 7.1. Considering both the hardware operating conditions and heterogeneity of edge hardware platforms expands the DNN-HW search space by orders of magnitude. Current estimators do not capture these new dimensions.

In this chapter, we focus on *learned performance estimators* that use machine learning to produce the estimates. As noted in [105], [106], it is crucial for these learned estimators to be able to be created from a limited amount of training data, given the bottleneck of running on physical devices. There have been a few recent efforts[63], [66]–[68] to produce such estimators. However, to date, these efforts have not considered the configurability and heterogeneity of modern edge devices. Moreover, there has not been a systematic characterization and evaluation of the different alternative strategies to the design of learned performance estimations.

To address the aforementioned challenges, we first step back and analyze different approaches to creating learned estimators, providing a new taxonomy for the design space to guide principled design decisions. Based on this analysis, we propose a new DNN performance estimator for heterogeneous, configurable edge devices that combines fine-grained layer-wise information and coarse-grained network-level information to achieve superior accuracy with a limited number of training samples. We implement a comprehensive and flexible framework based on our proposed taxonomy and use it to create and evaluate various DNN performance estimators.

In summary, we make the following contributions:

- We define a performance estimation taxonomy to characterize the design space of learned estimators for DNNs on heterogeneous edge platforms.
- Based on the insights from this taxonomy, we design a novel performance estimator that meets the unique needs of edge devices by combining fine- and coarse-grained information.
- We detail a comprehensive framework for creating DNN-HW performance estimators that consider both heterogeneity and hardware configurability.
- We demonstrate high accuracy on a variety of use cases, with errors between 3.3%-21.8%. The proposed estimators are two orders of magnitude faster than even native execution, enabling them to be used for large-scale design space exploration.

7.2 Learned Performance Estimators for DNNs: A Taxonomy

There is a wide variety of approaches one can adopt to create learned DNN performance estimators. We identify two main dimensions to consider in the design of these estimators: the hierarchy of the estimator and the granularity of the input features.

7.2.1 Estimation Hierarchy

The first dimension to consider is the depth of the hierarchy used when constructing a DNN estimator. For simplicity, we consider two options in our discussion: one-level (*i.e.*, flat) estimation and two-level estimation.

One-level Estimators predict the target metric for the entire network directly. Thus, they take a flat, coarse-grained approach.

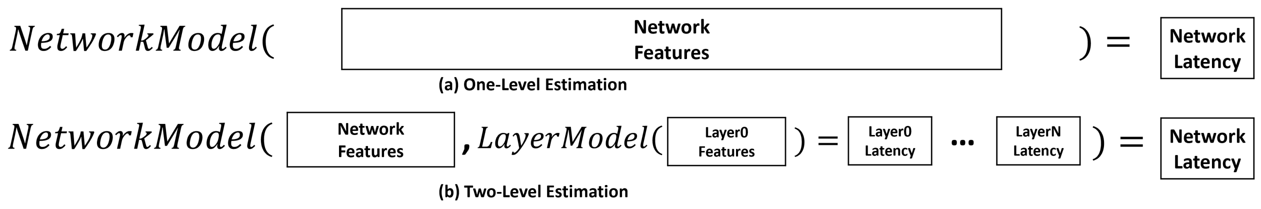


Figure 7.2. Examples of (a) one-level and (b) two-level estimators

Two-level Estimators first make predictions on a layer-by-layer basis using *layer-level performance estimators*, resulting in a prediction for each layer. These layer-level estimates are then incorporated into a *network-level estimator* (hence, the name two-level).

Two-level estimators provide a benefit by controlling complexity through a divide-and-conquer approach. This can be quite beneficial, as even just a naive summation layer times and energies tends to be a good first-order estimate that can be refined by the network-level estimator. However, this information can also bias the estimator in an incorrect direction– for example, due to hardware effects present in executing a network as a whole that are not present in running the layers in isolation.

Two-level estimators raise the question of how to design layer-level performance estimators, which we address next.

7.2.2 Layer-level Performance Estimators

A key characteristic of DNN workloads is that each DNN is comprised of a stack of different layers, wherein each layer is mostly executed sequentially. There is a relatively limited number of types of these layers (convolutional, fully-connected, pooling, *etc.*), though each layer has a number of parameters associated with it that define it's computational and memory footprint. With this in mind, it is useful to develop parameterized layer-level performance estimators for each of these layer types.

Inputs to the layer-level estimators can be the layer's parameters, which define the mathematical operation of the layer: number of input features, number of kernels, input feature size, kernel size, stride, *etc.* Other examples include values which are derived from those parameters combined with knowledge of the layer type: number of operations, activation and weight memory footprint, *etc.* These features are then used as input to a machine learning model to produce the desired estimation.

Hardware-aware Estimation. It is important for performance estimators to consider hardware operating conditions such as varying number of cores, core frequencies, *etc.* To account for this, we concatenate the hardware representation directly to the layer representation. We note that these values are intended to capture the behavior of a layer running on a specific IP; therefore, each IP will have its own estimator, as opposed to a shared estimator *across* IPs. These estimators are referred to as *Layer-IP estimators.* A Layer-IP estimator for a CPU is summarized in Figure 7.3.

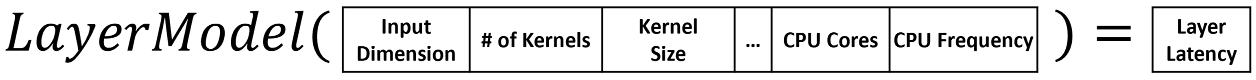


Figure 7.3. Representation of a Layer-IP Estimator

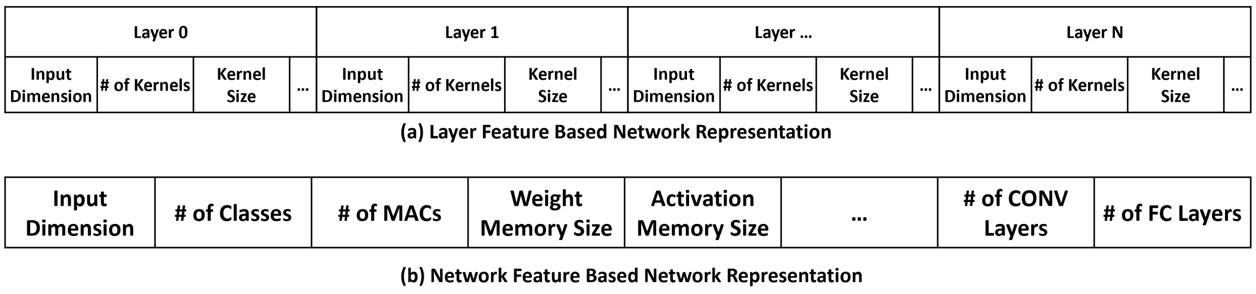
7.2.3 Layer-level vs Network-level Features

A second key dimension in the design of the top-level network performance estimator is the granularity of features to use. In this regard, there is a spectrum of choices ranging from using only layer-level features to using only network-level features.

Layer-level. The features for every layer are concatenated and used as inputs to the performance estimator. This can offer greater insight to the workings of a network, as it provides fine-grained information about each layer. However, this greatly increases the total number of features used by the network estimator, making it more difficult to train, particularly in a setting where limited training data is available. In other words, layer-level features can offer more fine-grained knowledge but will require more training samples due to the greater number of features– an important trade off for heterogeneous edge platforms, where it is difficult to generate large training sets for each device.

Network-level. The information from the layers are aggregated to produce features that summarize the network as a whole: total number of operations for the entire network, total number of weights, *etc.* This helps to reduce the total number of input features to the estimator, making it easier to train with limited data. However, there is loss of information, which may make the predictions less accurate.

These two different strategies can be visualized in Figure 7.4.



Layer 0				Layer 1				Layer …				Layer N			
Input Dimension	# of Kernels	Kernel Size	…	Input Dimension	# of Kernels	Kernel Size	…	Input Dimension	# of Kernels	Kernel Size	…	Input Dimension	# of Kernels	Kernel Size	…

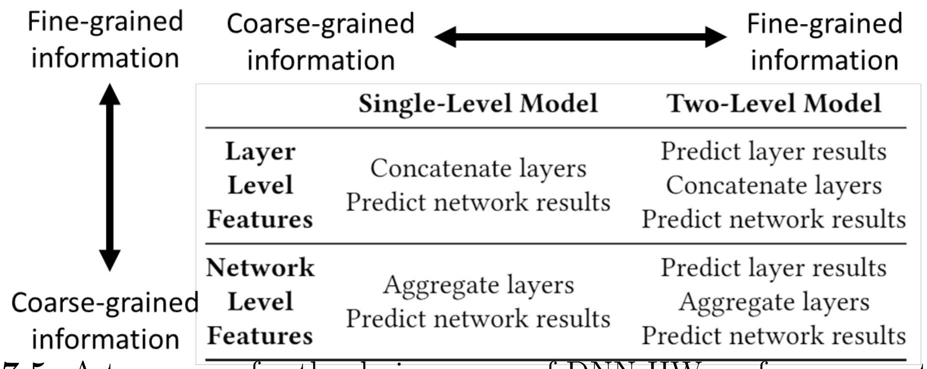
(a) Layer Feature Based Network Representation

Input Dimension	# of Classes	# of MACs	Weight Memory Size	Activation Memory Size	…	# of CONV Layers	# of FC Layers

(b) Network Feature Based Network Representation

Figure 7.4. Examples of (a) layer-based features and (b) network-based features for a network estimator

Based on these two axes, we can define a taxonomy for learned performance estimators as shown in Figure 7.5.


Fine-grained information
Coarse-grained information
Coarse-grained information
Fine-grained information

	Single-Level Model	Two-Level Model
Layer Level Features	Concatenate layers Predict network results	Predict layer results Concatenate layers Predict network results
Network Level Features	Aggregate layers Predict network results	Predict layer results Aggregate layers Predict network results

Figure 7.5. A taxonomy for the design space of DNN-HW performance estimators.

7.3 Performance Estimators for Heterogeneous Edge Devices

Based on the taxonomy proposed in Section 7.2, we now turn specifically to the design of a performance estimator for the unique challenges of heterogeneous edge devices. This estimator must be aware of the different hardware operating conditions of the edge device, support heterogeneous execution of a DNN on a multiple different hardware IPs, and be sample efficient to limit the cost of running on many different physical platforms.

Hardware-aware Estimation. As in hardware-aware estimation for layers, we again concatenate the hardware representation to the network representation. We note that this representation is applied across the entire network; that is, the hardware configuration is for the network as a whole, not on a per-layer basis.

Heterogeneity. We employ two strategies for exposing heterogeneity to the estimator: layer-level and network-level. At the layer-level, we employ a one-hot encoding for each layer, indicating on which IP that layer is being executed. At the network-level, we count the total number of layers executed on each IP and use that as a feature.

Network feature-based, Two-level Estimators. We propose network feature-based, two-level estimators as a promising trade-off between fine- and course-grained representations. As discussed above, network feature-based estimators summarize all of the layer information into a single representation, thus providing a course-grained representation of the network as a whole. Two-level estimators, in contrast, explicitly model the performance of every layer in a given network, thereby offering a representation of the network that uses much more fine-grained information. Thus, network feature-based, two-level estimators represent a favorable design point between detail and model complexity, allowing for fine-grained information without the need for a large number of training samples. This is ideal for the heterogeneous edge device use case. Layer feature-based, one-level estimators may appear to have a similar beneficial trade-off. However, the use of layer features as the fine-grained aspect of the model results in many more input features to the estimator, as each layer can have many features, while in two-level estimators there is only one predicted value per layer. We note that this represents a novel estimator design; to date, existing latency/power estimators have been limited to: layer feature-based, one-level estimators [68], network feature-based, one-level estimators [66], and layer feature-based, two-level estimators [63], [64].

7.4 Estimation Framework

In this section, we describe an automated framework to create different types of learned performance estimators, which consists of Workload Generation, Workload Execution, and Estimator Generation. Due to space limitations, we only present the framework in terms of generating a network estimator for a given hardware platform, but the overall framework is similar for generating layer estimators, which can then be used in the DNN-Hardware latency (power) estimators. This framework is depicted in Figure 7.6.

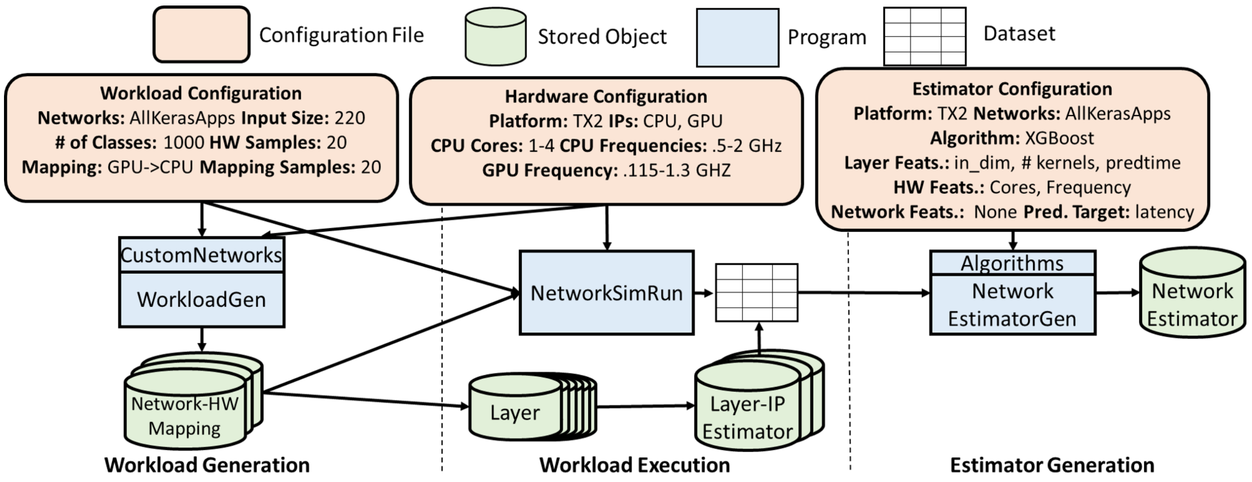

Configuration File
Stored Object
Program
Dataset
Workload Configuration
Networks: AllKerasApps Input Size: 220
# of Classes: 1000 HW Samples: 20
Mapping: GPU->CPU Mapping Samples: 20
Hardware Configuration
Platform: TX2 IPs: CPU, GPU
CPU Cores: 1-4 CPU Frequencies: .5-2 GHz
GPU Frequency: .115-1.3 GHZ
Estimator Configuration
Platform: TX2 Networks: AllKerasApps
Algorithm: XGBoost
Layer Feats.: in_dim, # kernels, predtime
HW Feats.: Cores, Frequency
Network Feats.: None Pred. Target: latency
CustomNetworks
WorkloadGen
NetworkSimRun
Algorithms
Network EstimatorGen
Network Estimator
Network-HW Mapping
Layer
Layer-IP Estimator
Workload Generation
Workload Execution
Estimator Generation

Figure 7.6. Overview of the framework's three stages. Workload Generation determines what networks to consider, as well as the hardware configurations and mappings. Workload Execution executes those choices on the target platform. Estimator Generations offers a flexible way to explore different estimation strategies.

7.4.1 Workload Generation

Workload Generation allows the user to generate examples for the estimator to tailor it to their specific use case by defining two inputs: the Hardware Configuration and the Workload Configuration.

Hardware Configuration. The Hardware Configuration details: the platform's name, hardware IPs that the platform contains, the hardware knobs which are accessible, and the allowed values for those hardware knobs.

Workload Configuration. The Workload Configuration defines the sample workloads that will be generated for the estimator, specifying a number of parameters: networks to use, the size of the input to the network, and the number of classes. Users also specify how to map the network to the hardware platform, *i.e.* all on one IP or heterogeneously. Finally, the user specifies how many different hardware operating conditions to sample.

Using these inputs, the Workload Generation stage creates a set of networks to be executed in the next stage. These networks also contain a list of mappings that define on which hardware IP each layer should be executed. Each of these mappings are then associated with a list of hardware operating conditions that should be executed.

7.4.2 Workload Execution

The next stage of the our estimation framework is Workload Execution. In this stage, the test cases created in the Workload Generation stage are loaded to the target platform, the platform is configured according to the test case parameters, and the network is then executed and profiled for runtime statistics such as latency, power, *etc.* The results of this stage is saved as a dataset, which contains both a superset of all the static features as well as the dynamic run-time statistics generated by execution.

7.4.3 Estimator Generation

In the Estimator Generation stage, the framework creates a latency (power) estimator whose input features, algorithm, and training/testing test is chosen by the hyperparameters provided in the Estimator Configuration file. The dataset generated by Workload Execution contains essentially a super-set of all possible relevant features that could be used. A critical decision for Estimator Generation, then, is which subset of these features to use. The algorithm can be any of a wide-range of existing regression techniques, such as Lasso, support vector machines, XGBoost, *etc.*. Further, the user can place networks in the training (testing) split, or have them randomly distributed. Finally, the Estimator Configuration specifies what the prediction target should be: latency, power, energy, *etc.*

7.5 Experimental Setup

Hardware Setup. We demonstrate our framework using NVIDIA's Jetson TX2 embedded systems board [103]. Crucially, this platform consists of multiple hardware knobs which can be configured to place the board in different operating conditions. These knobs are summarized in Table 7.2.

Software Setup. For our experiments, we use models from the model zoo provided by the Keras Tensorflow 2.4.1 library. All experiments were performed at a batch size of one using full-precision and eager execution mode.

Dataset. For the GPU, we use twenty-five networks, with four different hardware operating conditions. We sample ten different variations of input dimension/number of classes. For the CPU, we use eight networks, with twenty different hardware operating conditions, and four different variations of input dimension/number of classes. This difference is due to both the larger hardware configuration space for the CPU as well as the greater cost in running these networks on the CPU. These benchmarks are summarized in Table 7.1

Estimators. For layer-level features, we use: input/output dimension, number of kernels, kernel size, stride, number of operations, and memory size. For network-level features, we use: layer count for each layer type, total number of operations, total memory footprint. For two-level estimators, we either include the layer-level time and power estimates as inputs to the network-level estimator, or aggregate these layer predictions to use as inputs. We use the XGBoost algorithm as the machine learning model.

Table 7.1. The networks are obtained from the Keras model zoo. The GPU training set contains all of the networks; the CPU training set contains only the bolded networks.

Networks	DenseNet[**121**|169|201], EfficientNetB[**0**-7] InceptionResNetV2, **InceptionV3**, **Xception** **MobileNet**[V2|V3Large|V3Small] NASNet[Large|**Mobile**] ResNet[**50**|**50V2**|101|101V2|152|152V2]
Input Sizes	$110^2, 150^2, 224^2, 320^2$

Table 7.2. Summary of the range of available values for the hardware knobs present in the TX2 platform.

	TX2
GPU Frequency	[115MHz-1.3GHz]
CPU ARM Cores	[1 - 4]
CPU Frequency	[500MHz-2GHz]

7.6 Evaluation

We evaluate the performance of the estimators generated by our framework under a variety of conditions. While network feature, two-level estimators are our novel proposal, we note that a second contribution of our work is the flexible framework to create performance estimators that embody a wide range of estimation strategies.

7.6.1 Latency Estimators

Hardware Mapping and Configuration Search. This emulates the scenario where the user has a given DNN and is trying to determine the most efficient way to map it to a given hardware platform, while also configuring the platform. We consider the case where layers are distributed to both the CPU and the GPU to execute– *i.e.*, the network is executed heterogeneously. For this experiment, we consider NASNetMobile as the given network and generate multiple random mappings, each with multiple different hardware configurations. As shown in Table 7.3, which summarizes the mean absolute percent error (MAPE) for the produced latency estimators, our framework produces quite accurate estimators across all cases. Specifically, our novel approach of a two-level, network feature-based estimator performs the best, achieving a latency MAPE of 3.3% and 5.1% for the fixed hardware and configurable hardware cases, respectively. Not only are our estimators accurate, they are also much faster than direct execution; the estimator takes an average of 1`ms` per predicted sample vs an average of 500`ms` for execution on the TX2 platform.

Table 7.3. MAPE for latency prediction in a heterogeneous execution case study.

	Layer Feat. One-Level	Network Feat. One-Level	Layer Feat. Two-Level	**Network Feat. Two-Level**
Fixed HW	10.0%	**3.3%**	10.0%	**3.3%**
Configurable HW	6.3%	6.2%	6.5%	**5.1%**

Network Variant-HW Configuration Co-Design. We next consider the performance of the estimators generated by our framework in the network variant use case, wherein the end user is searching through the space of network variants (*e.g.*, transfer-learning) to determine the most efficient network-hardware configuration combination. In this instance, we only consider all CPU or all GPU mappings. Table 7.4 summarizes the MAPE and the Spearman coefficient (an important measure of relative accuracy) of the latency estimators generated for both the GPU and CPU hardware IPs of the TX2 platform. We see that our framework produces accurate estimators, ranging from 5.5%-3.2% error for the GPU and 14.1%-8.9% error for the CPU, with a maximum Spearman coefficient of 99.5% and 97.4%, respectively. In this scenario, our proposed strategy does not perform the best, but is still competitive with the best estimator. We highlight, however, that our estimator does outperform the other methods at smaller training set sizes. In Figure 7.7, we provide a visualization of the increase in Spearman coefficient as a function of the number of training samples. We see that, as expected, layer feature-based estimators require more samples than network feature-based ones, and similarly, one-level estimators require more training data than two-level estimators.

Table 7.4. MAPE and Spearman coefficient for latency prediction in the network variant context.

	Layer Feat. One-Level	Network Feat. One-Level	Layer Feat. Two-Level	**Network Feat. Two-Level**
GPU	5.2% | 99.0%	5.5% | 98.4%	**3.2%** | **99.5%**	5.5% | 99.1%
CPU	**8.9%** | **97.4%**	9.1% | 96.8%	14.1% | 95.4%	13.0% | 95.8%

Novel Network-Hardware Configuration Co-Design. Finally, we consider the performance of the estimators generated by our framework in the novel network use case, wherein the end user is selecting from a large number of *diverse* models and wants to determine the most efficient network-hardware configuration combination. As shown in Table 7.5, our novel approach greatly outperforms the other approaches: by fifteen percentage points for the GPU and eleven percentage points for the CPU. We highlight that this increase in performance is achieved in the most challenging use case, as the testing networks are completely out of the domain of the training networks.

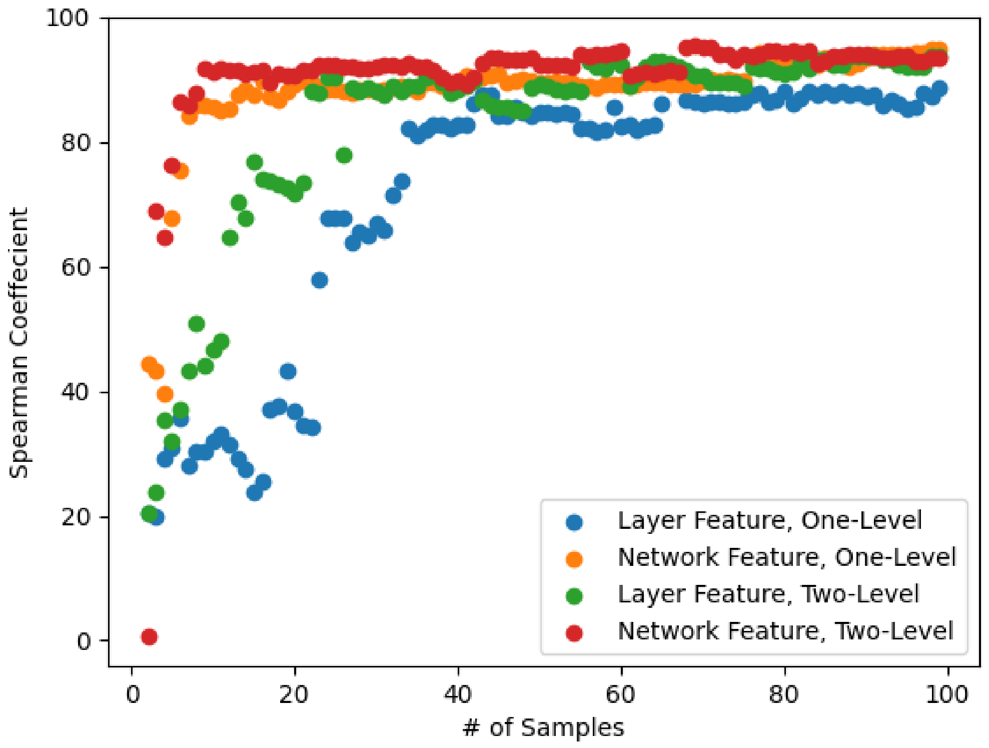


Figure 7.7. Analysis of achieved Spearman coefficient for varying training set sizes. Layer feature-based estimators require more training samples, as do one-level estimators.

Table 7.5. MAPE and Spearman coefficient for latency prediction in the novel network context.

	Layer Feat. One-Level	Network Feat. One-Level	Layer Feat. Two-Level	**Network Feat. Two-Level**
GPU	30.0% | 62.6%	52.6% | 35.1%	39.8% | 57.0%	**15.5%** | **93.6%**
CPU	33.7% | 63.6%	43.9% | 41.1%	55.4% | 63.2%	**21.8%** | **82.0%**

7.6.2 Power Estimators

Figure 7.8 shows the performance of the four different power estimators for the CPU IP for both the network variant and novel network use cases described above. We note that the accuracy is quite high across all strategies, with the maximum error across all estimators and use cases being under 3.5% . Due to space constraints, we only provide results for the CPU here; however, the performance for the GPU power estimators are just as accurate.

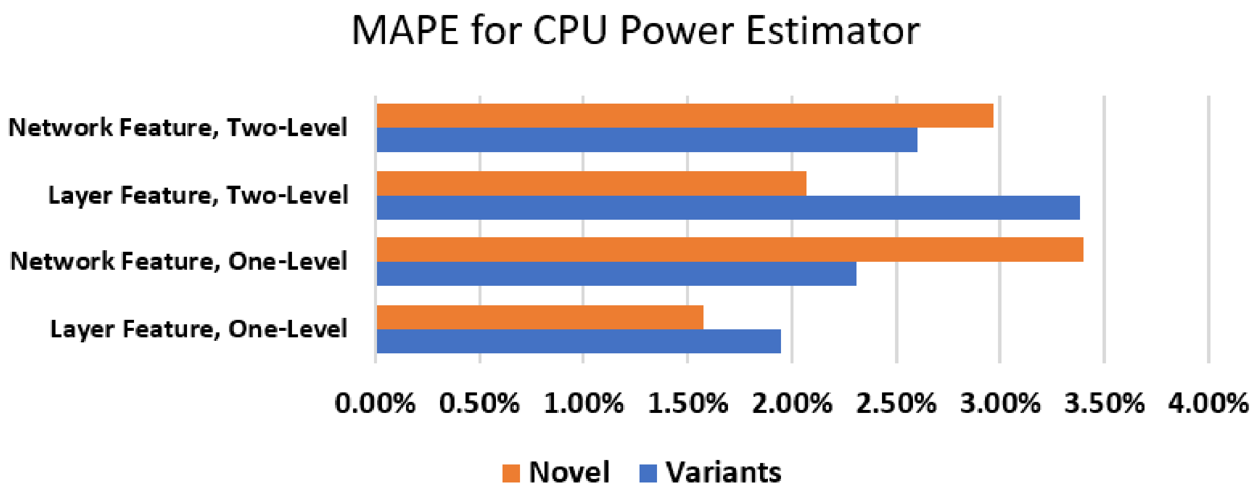


Figure 7.8. All CPU estimators perform quite well for predicting power.

7.7 Conclusion

With the recent explosion in the number of DNNs available to choose from, it is increasingly important for system designers to be able to quickly evaluate the performance of a given DNN on the intended hardware platform. It is infeasible to do so by directly running all of these networks on the physical hardware, which has led to an increased interest in the design of accurate latency (power) estimators, which are able to predict the performance of a network on a given device without actually running it. In this chapter, we first classify the taxonomy of different estimation models in order to provide a principled approach to the design of these models. Using the insights from this taxonomy, we design a novel performance estimator specifically for heterogeneous edge devices. Finally, we present a flexible framework that can be used to generate these hardware-aware performance estimators. We demonstrate this framework's ability to generate accurate estimators for the NVIDIA TX2 board on a variety of different use cases.

8. CONCLUSION

Deep Neural Networks (DNNs) have seen tremendous success in a variety of tasks, achieving state-of-the-art results on many benchmarks– particularly within the perceptual domain; Performance on the ImageNet dataset has long surpassed human-level results. There has recently been many efforts to translate this success in the perceptual domain to other tasks. In particular, computer scientists are interested in new capabilities for DNNs such as one-shot learning and task planning. Furthermore, there have also been efforts to extend this success from the perceptual domain– more specifically, the Euclidean domain– to the non-Euclidean domain (*e.g.*, graphs, manifolds, *etc.*).

As a result, there are emerging neural workloads quickly gaining in popularity which aim to address these new tasks and challenges. These emerging neural workloads exhibit unique computational challenges compared to the previous iteration of DNNs. Memory-augmented neural networks (MANNs)– which augment a traditional DNN with an external, differentiable memory– are one such set of new workloads. MANNs have been shown to achieve one-shot learning and complex cognitive capabilities that are well beyond those of classical DNNs. However, these new capabilities come at a cost; to perform these tasks, MANNs utilize *soft reads and writes* to the differentiable memory, each of which requires access to *all* the memory locations. Further, the self-attention operation required by the soft reads makes heavy use of the softmax operation, particularly in the popular subtype of MANNs known as Transformer-based network. These computational challenges result in poor performance of MANNs on modern CPUs, GPUs, and other accelerators.

Graph Neural Networks (GNNs) are another promising new class of DNNs that apply the success of deep learning to a new domain: graphs. GNNs extract features from the nodes of a graph using a fully-connected layer and aggregate these features using message passing between the nodes. Thus, GNNs combine two distinct computational patterns: dense, regular computations and sparse, irregular computations. Existing hardware acclerators are designed for one type of computation or the other, thus resulting in poor performance in this hybrid workload.

There is a pressing need to design new hardware to address the new challenges resulting from these neural workloads. Further, as the pace of development in DNNs increases, it is important to automate aspects of this tighter coupling between new DNNs and new hardware.

8.1 Thesis Summary

The main contributions of this thesis are as follows:

- This thesis first considers MANNs, in which the main bottlenecks stem from soft reads and writes, wherein every memory location is accessed multiple times as well as addressing mechanisms. To address this, we propose `Manna`, a novel memory-first accelerator that is better-suited to the low FLOPs/Byte ratio inherent to MANNs. `Manna` also utilizes a hardware-assisted transpose mechanism tailored to the memory access patterns found in MANNs.

- Next, we consider Transformer-based networks, an important subclass of MANNs. Transformers retain the weighting and soft read operations found in general MANNs, while eschewing the soft write and addressing kernels, and tend to be deeper than other MANNs (10 layers rather than 3 layers). As a result, Transformers have a unique bottleneck: the softmax operation at the heart of the weighting step of the attention mechanism. We propose `Softermax`, a set of hardware/software co-design techniques to minimize the overhead of the softmax operation: (i) base substitution, (ii) low-precision computations, and (iii) an online, integer-based normalization scheme.

- Next, this thesis explores graph neural networks. GNNs have two main computational patterns: an irregular, graph-based processing step and a regular, DNN-based step. Existing accelerators for graph workloads and DNN workloads are mismatched for this hybrid workload. To address this gap, we propose `GNNerator`, a hybrid architecture with a Graph Engine and a Dense Engine for the two different processing patterns, and is able to exploit the abundant intra- and inter-node parallelism inherent in GNNs.

`GNNerator` also utilizes feature dimension-blocking, a novel GNN dataflow that allows for processing more nodes in a graph on-chip at one time.

- Finally, we turn to accelerating the tighter coupling between neural workloads and the underlying hardware, through the development of accurate performance models which can be used to autonomously search this design space. We first characterize the performance estimator design space taxonomy. We use the insights from this characterization to propose a novel approach tailored to the heterogeneous edge device use case, which balances fine-grained and coarse-grained representations of the network. We propose a framework to predict the performance– both latency and power– of a variety of networks running on edge devices with various hardware operating conditions, and demonstrate reasonable accuracy. This is a crucial step towards automating a tighter coupling between new neural network architectures and the hardware on which they are executed.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[2] *Using deepspeed and megatron to train megatron-turing nlg 530b, the world's largest and most powerful generative language model*, Oct. 2021. [Online]. Available: https://developer.nvidia.com/blog/using-deepspeed-and-megatron-to-train-megatron-turing-nlg-530b-the-worlds-largest-and-most-powerful-generative-language-model/.

[3] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, Ieee, 2009, pp. 248–255.

[4] B. Soo Ko, *Computer vision leaderboard*. [Online]. Available: https://github.com/kobiso/Computer-Vision-Leaderboard.

[5] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini, *et al.*, "Deep learning recommendation model for personalization and recommendation systems," *arXiv preprint arXiv:1906.00091*, 2019.

[6] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14, Salt Lake City, Utah, USA: ACM, 2014, pp. 269–284, ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541967. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541967.

[7] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the International Symposium on Microarchitecture*, ser. MICRO '14, IEEE Computer Society, 2014, pp. 609–622.

[8] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.

[9] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, *et al.*, "Scaledeep: A scalable compute architecture for learning and evaluating deep networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 13–26.

[10] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[11] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss v2: A flexible and high-performance accelerator for emerging deep neural networks," *arXiv preprint arXiv:1807.07928*, 2018.

[12] S. Sen, S. Jain, S. Venkataramani, and A. Raghunathan, "Sparce: Sparsity aware general purpose core extensions to accelerate deep neural networks," *IEEE Transactions on Computers*, 2018.

[13] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. Vijaykumar, "Sparten: A sparse tensor accelerator for convolutional neural networks," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 151–165.

[14] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "Sparch: Efficient architecture for sparse matrix multiplication," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2020, pp. 261–274.

[15] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, IEEE, 2017, pp. 27–40.

[16] "Nvidia a100 tensor core gpu architecture," 2020.

[17] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2020, pp. 58–70.

[18] S. Dave, R. Baghdadi, T. Nowatzki, S. Avancha, A. Shrivastava, and B. Li, "Hardware acceleration of sparse and irregular tensor computations of ml models: A survey and insights," *Proceedings of the IEEE*, vol. 109, no. 10, pp. 1706–1752, 2021.

[19] Y. Umuroglu, L. Rasnayake, and M. Själander, "Bismo: A scalable bit-serial matrix multiplication overlay for reconfigurable computing," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2018, pp. 307–3077.

[20] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural networks," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2018.

[21] S. Jain, S. Venkataramani, V. Srinivasan, J. Choi, P. Chuang, and L. Chang, "Compensated dnn: Energy efficient low-precision deep neural networks by compensating quantization errors," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, IEEE, 2018, pp. 1–6.

[22] R. Elangovan, S. Jain, and A. Raghunathan, "Ax-bxp: Approximate blocked computation for precision-reconfigurable deep neural network acceleration," *arXiv preprint arXiv:2011.13000*, 2020.

[23] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

[24] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, "Pact: Parameterized clipping activation for quantized neural networks," *arXiv preprint arXiv:1805.06085*, 2018.

[25] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," *arXiv preprint arXiv:1806.08342*, 2018.

[26] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "Haq: Hardware-aware automated quantization with mixed precision," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 8612–8620.

[27] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015, pp. 1135–1143.

[28] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, *et al.*, "Ese: Efficient speech recognition engine with sparse lstm on fpga," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 75–84.

[29] R. Venkatesan, Y. S. Shao, M. Wang, J. Clemons, S. Dai, M. Fojtik, B. Keller, A. Klinefelter, N. R. Pinckney, P. Raina, *et al.*, "Magnet: A modular accelerator generator for neural networks.," in *ICCAD*, 2019, pp. 1–8.

[30] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, *et al.*, "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 14–27.

[31] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2016, pp. 1–12.

[32] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," ISCA '16, pp. 1–13, 2016. DOI: 10.1109/ISCA.2016.11. [Online]. Available: https://doi.org/10.1109/ISCA.2016.11.

[33] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "Extensor: An accelerator for sparse tensor algebra," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 319–333.

[34] J. Rae, J. J. Hunt, I. Danihelka, T. Harley, A. W. Senior, G. Wayne, A. Graves, and T. Lillicrap, "Scaling memory-augmented neural networks with sparse reads and writes," in *Advances in Neural Information Processing Systems*, 2016, pp. 3621–3629.

[35] S. Chandar, S. Ahn, H. Larochelle, P. Vincent, G. Tesauro, and Y. Bengio, "Hierarchical memory networks," *arXiv preprint arXiv:1605.07427*, 2016.

[36] S. Park, S. Kim, S. Lee, H. Bae, and S. Yoon, "Quantized memory-augmented neural networks," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[37] A. F. Laguna, M. Niemier, and X. S. Hu, "Design of hardware-friendly memory enhanced neural networks," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2019, pp. 1583–1586.

[38] A. Ranjan, S. Jain, J. R. Stevens, D. Das, B. Kaul, and A. Raghunathan, "X-mann: A crossbar based architecture for memory augmented neural networks," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19, Las Vegas, NV, USA: ACM, 2019, 130:1–130:6, ISBN: 978-1-4503-6725-7. DOI: 10.1145/3316781.3317935. [Online]. Available: http://doi.acm.org/10.1145/3316781.3317935.

[39] S. Park, J. Jang, S. Kim, and S. Yoon, "Energy-efficient inference accelerator for memory-augmented neural networks on an fpga," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2019, pp. 1587–1590.

[40] H. Jang, J. Kim, J.-E. Jo, J. Lee, and J. Kim, "Mnnfast: A fast and scalable system architecture for memory-augmented neural networks," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19, Phoenix, Arizona: ACM, 2019, pp. 250–263, ISBN: 978-1-4503-6669-4. DOI: 10.1145/3307650.3322214. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322214.

[41] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter," *arXiv preprint arXiv:1910.01108*, 2019.

[42] Q. Guo, X. Qiu, P. Liu, *et al.*, "Star-Transformer," in *NAACL-HLT*, 2019.

[43] A. Nagarajan, S. Sen, J. R. Stevens, and A. Raghunathan, "Optimizing Transformers with Approximate Computing for Faster, Smaller and more Accurate NLP Models," *arXiv preprint arXiv:2010.03688*, 2020.

[44] O. Zafrir, G. Boudoukh, P. Izsak, and M. Wasserblat, "Q8BERT: Quantized 8bit BERT," *arXiv preprint arXiv:1910.06188*, 2019.

[45] G. Prato, E. Charlaix, and M. Rezagholizadeh, "Fully quantized transformer for machine translation," in *Proc. of EMNLP*, 2020.

[46] Y. Lin, Y. Li, T. Liu, *et al.*, "Towards fully 8-bit integer inference for the transformer model," in *Proc. of IJCAI*, 2020, pp. 3759–3765.

[47] T. J. Ham, S. J. Jung, S. Kim, *et al.*, "A3: Accelerating attention mechanisms in neural networks with approximation," in *Proc. of HPCA*, IEEE, 2020, pp. 328–341.

[48] H. Wang, Z. Zhang, and S. Han, "Spatten: Efficient sparse attention architecture with cascade token and head pruning," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2021, pp. 97–110.

[49] T. Tambe, C. Hooper, L. Pentecost, T. Jia, E.-Y. Yang, M. Donato, V. Sanh, P. Whatmough, A. M. Rush, D. Brooks, *et al.*, "Edgebert: Sentence-level energy optimizations for latency-aware multi-task nlp inference," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 830–844.

[50] Y. Gao, W. Liu, and F. Lombardi, "Design and implementation of an approximate softmax layer for deep neural networks," in *Proc. of ISCAS*, IEEE, 2020, pp. 1–5.

[51] G. Du, C. Tian, Z. Li, *et al.*, "Efficient softmax hardware architecture for deep neural networks," in *Proc. of GLSVLSI*, 2019, pp. 75–80.

[52] D. Zhu, S. Lu, M. Wang, J. Lin, and Z. Wang, "Efficient precision-adjustable architecture for softmax function in deep learning," *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2020.

[53] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, *et al.*, "Deep graph library: Towards efficient and scalable deep learning on graphs," *arXiv preprint arXiv:1909.01315*, 2019.

[54] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken, "Improving the accuracy, scalability, and performance of graph neural networks with roc," *Proceedings of Machine Learning and Systems (MLSys)*, pp. 187–198, 2020.

[55] Y. Wang, B. Feng, G. Li, S. Li, L. Deng, Y. Xie, and Y. Ding, "Gnnadvisor: An efficient runtime system for gnn acceleration on gpus," *arXiv preprint arXiv:2006.06608*, 2020.

[56] S. Avancha, V. Md, S. Misra, and M. Ramanarayan, "Deep graph library optimizations for intel x86 architecture," *arXiv preprint arXiv:2007.06354*, 2020.

[57] S. A. Tailor, J. Fernandez-Marques, and N. D. Lane, "Degree-quant: Quantization-aware training for graph neural networks," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=NSBrFgJAHg.

[58] Z. Jia, S. Lin, R. Ying, J. You, J. Leskovec, and A. Aiken, "Redundancy-free computation for graph neural networks," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 997–1005.

[59] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Hygcn: A gcn accelerator with hybrid architecture," *arXiv preprint arXiv:2001.02514*, 2020.

[60] A. Auten, M. Tomei, and R. Kumar, "Hardware accleration of graph neural networks," in *2020 57th ACM/ESDA/IEEE Design Automation Conference (DAC)*, IEEE, 2020, pp. 1–6.

[61] L. He, "Engn: A high-throughput and energy-efficient accelerator for large graph neural networks," *arXiv preprint arXiv:1909.00155*, 2019.

[62] H. Qi, E. R. Sparks, and A. Talwalkar, "Paleo: A performance model for deep neural networks," in *Proceedings of the International Conference on Learning Representations*, 2017.

[63] E. Cai, D.-C. Juan, D. Stamoulis, and D. Marculescu, "Neuralpower: Predict and deploy energy-efficient convolutional neural networks," in *Asian Conference on Machine Learning*, PMLR, 2017, pp. 622–637.

[64] M. Wess, M. Ivanov, C. Unger, A. Nookala, A. Wendt, and A. Jantsch, "Annette: Accurate neural network execution time estimation with stacked models," *IEEE Access*, vol. 9, pp. 3545–3556, 2020.

[65] C. F. Rodrigues, G. Riley, and M. Luján, "Energy predictive models for convolutional neural networks on mobile platforms," *arXiv preprint arXiv:2004.05137*, 2020.

[66] H. Bouzidi, H. Ouarnoughi, S. Niar, and A. A. E. Cadi, "Performance prediction for convolutional neural networks on edge gpus," in *Proceedings of the 18th ACM International Conference on Computing Frontiers*, 2021, pp. 54–62.

[67] D. Velasco-Montero, J. Fernández-Berni, R. Carmona-Galán, *et al.*, "Previous: A methodology for prediction of visual inference performance on iot devices," *IEEE Internet of Things Journal*, vol. 7, no. 10, pp. 9227–9240, 2020.

[68] V. Ganesan, S. Selvam, S. Sen, P. Kumar, and A. Raghunathan, "A case for generalizable dnn cost models for mobile devices," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, 2020, pp. 169–180.

[69] F. Rosenblatt, "Perceptron simulation experiments," *Proceedings of the IRE*, vol. 48, no. 3, pp. 301–309, 1960.

[70] Y. Bengio, P. Simard, P. Frasconi, *et al.*, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.

[71] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[72] K. Cho, B. V. Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," *arXiv preprint arXiv:1409.1259*, 2014.

[73] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[74] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[75] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, *et al.*, "Deep speech: Scaling up end-to-end speech recognition," *arXiv preprint arXiv:1412.5567*, 2014.

[76] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2013.

[77] S. Sukhbaatar, J. Weston, R. Fergus, *et al.*, "End-to-end memory networks," in *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, 2015.

[78] J. Weston, S. Chopra, and A. Bordes, "Memory networks," *arXiv preprint arXiv:1410.3916*, 2014.

[79] A. Kumar, O. Irsoy, P. Ondruska, M. Iyyer, J. Bradbury, I. Gulrajani, V. Zhong, R. Paulus, and R. Socher, "Ask me anything: Dynamic memory networks for natural language processing," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2016.

[80] C. Xiong, S. Merity, and R. Socher, "Dynamic memory networks for visual and textual question answering," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2016.

[81] A. Graves, G. Wayne, and I. Danihelka, "Neural turing machines," *arXiv preprint arXiv:1410.5401*, 2014.

[82] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, *et al.*, "Hybrid computing using a neural network with dynamic external memory," *Nature*, vol. 538, no. 7626, p. 471, 2016.

[83] E. Grefenstette, K. M. Hermann, M. Suleyman, and P. Blunsom, "Learning to transduce with unbounded memory," in *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, 2015, pp. 1828–1836.

[84] G. Wayne, C.-C. Hung, D. Amos, M. Mirza, A. Ahuja, A. Grabska-Barwinska, J. Rae, P. Mirowski, J. Z. Leibo, A. Santoro, *et al.*, "Unsupervised predictive memory in a goal-directed agent," *arXiv preprint arXiv:1803.10760*, 2018.

[85] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[86] T. T. Yeh, A. Sabne, P. Sakdhnagool, R. Eigenmann, and T. G. Rogers, "Pagoda: Fine-grained gpu resource virtualization for narrow tasks," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2017. DOI: 10.1145/3018743.3018754.

[87] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[88] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *arXiv preprint arXiv:1901.00596*, 2019.

[89] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, and M. Sun, "Graph neural networks: A review of methods and applications," *arXiv preprint arXiv:1812.08434*, 2018.

[90] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 31–46.

[91] Y. Chi, G. Dai, Y. Wang, G. Sun, G. Li, and H. Yang, "Nxgraph: An efficient graph processing system on a single machine," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, IEEE, 2016, pp. 409–420.

[92] X. Zhu, W. Han, and W. Chen, "Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *USENIX*, 2015, pp. 375–386.

[93] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ACM, 2018, pp. 974–983.

[94] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, JMLR. org, 2017, pp. 1263–1272.

[95] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," *arXiv preprint arXiv:1511.05493*, 2015.

[96] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[97] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *International conference on machine learning*, 2016, pp. 2702–2711.

[98] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” *arXiv preprint arXiv:1609.02907*, 2016.

[99] W. Hamilton, Z. Ying, and J. Leskovec, “Inductive representation learning on large graphs,” in *Advances in neural information processing systems*, 2017, pp. 1024–1034.

[100] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[101] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, “Graphicionado: A high-performance and energy-efficient accelerator for graph analytics,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2016, pp. 1–13.

[102] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, *et al.*, “Machine learning at facebook: Understanding inference at the edge,” in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2019, pp. 331–344.

[103] *Harness ai at the edge with the jetson tx2 developer kit*, Jan. 2021. [Online]. Available: https://developer.nvidia.com/embedded/jetson-tx2-developer-kit.

[104] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, *et al.*, “Searching for mobilenetv3,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 1314–1324.

[105] T.-J. Yang, A. Howard, B. Chen, X. Zhang, A. Go, M. Sandler, V. Sze, and H. Adam, “Netadapt: Platform-aware neural network adaptation for mobile applications,” in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 285–300.

[106] X. Dai, P. Zhang, B. Wu, H. Yin, F. Sun, Y. Wang, M. Dukhan, Y. Hu, Y. Wu, Y. Jia, *et al.*, “Chamnet: Towards efficient network design through platform-aware model adaptation,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 11 398–11 407.

[107] X. Zhang and Y. LeCun, “Text understanding from scratch,” *arXiv preprint arXiv:1502.01710*, 2015.

[108] S. M. Tam, H. Muljono, M. Huang, S. Iyer, K. Royneogi, N. Satti, R. Qureshi, W. Chen, T. Wang, H. Hsieh, *et al.*, “Skylake-sp: A 14nm 28-core xeon® processor,” in *Proceedings of the International Solid-State Circuits Conference (ISSCC)*, IEEE, 2018, pp. 34–36.

[109] G. Ruetsch and P. Micikevicius, "Optimizing matrix transpose in cuda," *Nvidia CUDA SDK Application Note*, vol. 18, 2009.

[110] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, IEEE Press, 2011, pp. 694–701.

[111] AMD, *High-bandwidth memory: Reinventing memory technology.*

[112] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," in *Proc. of NeurIPS*, 2017, pp. 5998–6008.

[113] A. Dosovitskiy, L. Beyer, A. Kolesnikov, *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[114] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. of NAACL-HLT*, 2019, pp. 4171–4189.

[115] M. Shoeybi, M. Patwary, R. Puri, *et al.*, "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[116] A. Radford, J. Wu, R. Child, *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[117] M. Milakov and N. Gimelshein, "Online normalizer calculation for softmax," *arXiv preprint arXiv:1805.02867*, 2018.

[118] "NVIDIA Tesla V100 GPU Architecture," NVIDIA, Tech. Rep., 2017.

[119] T. Wolf *et al.*, "Huggingface's transformers: State-of-the-art natural language processing," *ArXiv*, arXiv–1910, 2019.

[120] H. Wu, P. Judd, X. Zhang, *et al.*, "Integer quantization for deep learning inference: Principles and empirical evaluation," *arXiv preprint arXiv:2004.09602*, 2020.

[121] [Online]. Available: https://www.synopsys.com/designware-ip.html.

[122] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2015.

[123] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, “Learning transferable architectures for scalable image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 8697–8710.

[124] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks,” in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[125] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, “Roberta: A robustly optimized bert pretraining approach,” *arXiv preprint arXiv:1907.11692*, 2019.

[126] Y. He, T. N. Sainath, R. Prabhavalkar, I. McGraw, R. Alvarez, D. Zhao, D. Rybach, A. Kannan, Y. Wu, R. Pang, *et al.*, “Streaming end-to-end speech recognition for mobile devices,” in *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2019, pp. 6381–6385.

[127] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, “Geometric deep learning: Going beyond euclidean data,” *IEEE Signal Processing Magazine*, vol. 34, no. 4, pp. 18–42, 2017.

[128] P. Battaglia, R. Pascanu, M. Lai, D. J. Rezende, *et al.*, “Interaction networks for learning about objects, relations and physics,” in *Advances in neural information processing systems*, 2016, pp. 4502–4510.

[129] M. B. Chang, T. Ullman, A. Torralba, and J. B. Tenenbaum, “A compositional object-based approach to learning physical dynamics,” *arXiv preprint arXiv:1612.00341*, 2016.

[130] Y. Li, O. Vinyals, C. Dyer, R. Pascanu, and P. Battaglia, “Learning deep generative models of graphs,” *arXiv preprint arXiv:1803.03324*, 2018.

[131] N. De Cao and T. Kipf, “Molgan: An implicit generative model for small molecular graphs,” *arXiv preprint arXiv:1805.11973*, 2018.

[132] J. You, B. Liu, Z. Ying, V. Pande, and J. Leskovec, “Graph convolutional policy network for goal-directed molecular graph generation,” in *Advances in Neural Information Processing Systems*, 2018, pp. 6410–6421.

[133] R. v. d. Berg, T. N. Kipf, and M. Welling, “Graph convolutional matrix completion,” *arXiv preprint arXiv:1706.02263*, 2017.

[134] N. G. Ravindra, A. Sehanobish, J. L. Pappalardo, D. A. Hafler, and D. van Dijk, "Disease state prediction from single-cell data using graph attention networks," in *Proceedings of the ACM Conference on Health, Inference, and Learning (CHIL)*, ACM, 2020.

[135] Y. Ma, H. Ren, B. Khailany, H. Sikka, L. Luo, K. Natarajan, and B. Yu, "High performance graph convolutional networks with applications in testability analysis," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ACM, 2019, p. 18.

[136] G. Zhang, H. He, and D. Katabi, "Circuit-gnn: Graph neural networks for distributed circuit design," in *International Conference on Machine Learning*, 2019, pp. 7364–7373.

[137] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou, "Aligraph: A comprehensive graph neural network platform," *arXiv preprint arXiv:1902.08730*, 2019.

[138] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proc. VLDB Endow.*, vol. 8, no. 11, pp. 1214–1225, Jul. 2015, ISSN: 2150-8097. DOI: 10.14778/2809974.2809983. [Online]. Available: https://doi.org/10.14778/2809974.2809983.

[139] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "Graphp: Reducing communication for pim-based graph processing with efficient data partition," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2018, pp. 544–557.

[140] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3, pp. 105–117, 2016.

[141] S. Jiang, B. Ilbeyi, and C. Batten, "Mamba: Closing the performance gap in productive hardware development frameworks," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, IEEE, 2018, pp. 1–6.

[142] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "Scale-sim: Systolic cnn accelerator simulator," *arXiv preprint arXiv:1811.02883*, 2018.

[143] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, "To push or to pull: On reducing communication and synchronization in graph computations," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, 2017, pp. 93–104.

[144] H. Cai, L. Zhu, and S. Han, “ProxylessNAS: Direct neural architecture search on target task and hardware,” in *International Conference on Learning Representations*, 2019. [Online]. Available: https://arxiv.org/pdf/1812.00332.pdf.