Accelerating Deep Learning on Heterogenous Architectures

Avinash Nandakumar
Sophia Shao, Ed.
Borivoje Nikolic, Ed.

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-100
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-100.html

May 13, 2022

Acknowledgement

Thank you to Prof. Sophia Shao for her advice, guidance, and mentorship throughout my time researching at ADEPT and Hasan Genc for being the best graduate student mentor ever. Next, thanks to Prof. Bora Nikolic and Prof. Dan Garcia for sparking my interest in computer architecture through CS61C and for mentoring me throughout being a GSI on course sta. Also, thanks to Apple’s Hardware Technologies for sponsoring me in this master’s program. Thank you to Sameer, Nikhil, Adi, Callum, Pranav, Mark, Ronan, Roast Squad and all my friends and roommates who made my Berkeley experience what it is and this thesis possible. Finally, biggest thank you to my loving Mom, hard working Dad, and caring sister for always supporting and believing in me. Go Bears!

Accelerating Deep Learning on Heterogenous Architectures

by

Avinash Kumar Nandakumar

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Sophia Shao, Chair
Professor Borivoje Nikolic, Second Reader

Spring 2022

The thesis of Avinash Kumar Nandakumar, titled Accelerating Deep Learning on Heterogenous Architectures, is approved:

Chair	[illegible]	Date	5/11/2022
	[illegible]	Date	5/13/2022
		Date	

University of California, Berkeley

Accelerating Deep Learning on Heterogenous Architectures

Abstract

Accelerating Deep Learning on Heterogenous Architectures

by

Avinash Kumar Nandakumar

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Sophia Shao, Chair

The growth of machine learning workloads, specifically deep neural networks (DNNs), in both warehouse scale computing (WSC) and on-edge mobile computing has driven a huge demand in different types of accelerators. This project focuses on exploring the different levels of parallelism when running deep learning inferences on heterogeneous architectures and characterization of coordinating unique accelerators with varying workloads. We have implemented an accelerated depthwise convolution kernel on a vector accelerator and explored the design space of executing MobileNetv2 in different configurations on an architecture consisting of both a systolic and vector accelerator. This work examines shared resource contention at the memory level with this given architecture and analyzes the effects of model pipelining and batch parallelism. Through layer by layer performance and cache analysis we examine the best parameters and configurations to execute MobileNetv2 inference, observing a 1.4x and 3.5x speedup over a naively accelerated baseline on single core and multi core SoCs.

Contents

List of Figures

List of Tables

Acknowledgments

Thank you to Prof. Sophia Shao for her advice, guidance, and mentorship throughout my time researching at ADEPT and Hasan Genc for being the best graduate student mentor ever. Next, thanks to Prof. Bora Nikolic and Prof. Dan Garcia for sparking my interest in computer architecture through CS61C and for mentoring me throughout being a GSI on course staff. Also, thanks to Apple's Hardware Technologies for sponsoring me in this master's program. Thank you to Sameer, Nikhil, Adi, Callum, Pranav, Mark, Ronan, Roast Squad and all my friends and roommates who made my Berkeley experience what it is and this thesis possible. Finally, biggest thank you to my loving Mom, hard working Dad, and caring sister for always supporting and believing in me. Go Bears!

Chapter 1

Introduction

The recent growth of machine learning workloads, specifically deep neural networks (DNNs), to perform tasks like object detection in autonomous vehicles or path finding in robotics or even natural language processing on mobile computing devices has significantly influenced the evolution of the computing hardware. With the end of Denard scaling and move towards domain-specific architectures, mapping these complicated deep neural network (DNN) workloads on to specialized hardware while achieving minimal latency and high throughput has not been trivial. There are many challenges to first designing accelerators that are both performant and versatile enough for these new intricate and demanding workloads, in addition to then successfully exposing the architecture such that programmers are able to fully utilize and take advantage of the hardware. This is often advertised as software-hardware co-design, yet very rarely achieved because of the complexity of the computing stack and trade offs between programmer usability and system performance.

One specific area of high growth, due to this phenomenon, is in the accelerator architecture space. Modern day SoCs are starting to incorporate more and more domain specific accelerators in their architecture that are used in a wide variety of ways from encoding/decoding data to supporting high-end camera modules or even accelerated compression of data. These accelerators all try to address a specific niche in software workload while the overall CPU, GPU, and newly added neural engines try to meet the demands of generic workloads. While there is much focus in both academia and industry in making faster and more versatile novel accelerator architectures that meet the growing demands of "Software

2.0”, there is relatively less work in understanding and investigating how current accelerators work and scale together.

In contrast, moving from single-core to multi-core CPUs in the last few decades, much work has gone into studying the performance and scaling of workloads as computation resources increase [10]. In addition, significant studies exist on how to effectively expose thread level and application level parallelism to the programmer [8]. This body of work is a great starting point when it comes to accelerators but calls for a deeper dive into studying how to best broadly abstract, implement, and coordinate accelerators in modern SoCs.

This work focuses on starting the exploration into using multiple unique Berkeley-developed accelerators together on a single SoC and examining the complexities and challenges that arise. This work is motivated by accelerating a few different kernels of a specific DNN workload and analyzing its overall performance. Then this work will move into exploring how running multiple accelerators in parallel can exploit workload level parallelism and improve performance but attract resource contention problems while doing so. Finally, this work will conclude by examining the performance and constraints in optimizing a single dependent DNN workload on two unique accelerators through pipelined execution of the system while being constrained by the workload.

1.1 Thesis Organization

Chapter 2 provides an overview of the motivating software workload, MobileNetV2, and a deep dive into depthwise convolutions, in addition to defining the various levels of DNN parallelism. Chapter 3 reviews our accelerator architecture configuration and baseline performance of MobileNet. Chapter 4 describes the Hwacha programming model and accelerated kernel performance and compared with the baseline CPU and Gemmini results. Chapter 5 examines resource contention within the context of multi accelerator usecases and proposes potential solutions. Chapter 6 focuses on increasing utilization while maintaining synchronization in a singular dependent workload.

1.2 Project History

This thesis leverages a lot of tools and preexisting work done through ADEPT Lab at Berkeley. Tools like FireSim, Chipyard, Gemmini, and Rocket are heavily relied upon to make this study possible and have had numerous contributors in their development. Most of this study has been developed through individual contribution by myself but reference some closely related projects and ideas. Specifically, the use of OnnxRuntime was made possible through another project contributed to and lead by Pranav Prakash. In addition, references to CALM and solutions for shared resource contention in Chapter 5 were developed in a closely related project lead by Seah Kim and assisted by myself. Finally, ideas and findings were assisted by Hasan Genc through close mentorship and guidance.

Chapter 2

Motivating Workload

In order to justify the need for using heterogeneous accelerators for a singular DNN workload, this work will first describe the motivating DNN that will be primarily used throughout this work, MobileNetV2 [20]. This work will then characterize MobileNet's architecture and discuss the mapping of specific layers within such as normal convolution and depthwise convolution. In addition, this work will broadly define and describe the levels of parallelism within DNNs that this work will look to exploit further in this work.

2.1 MobileNetV2

MobileNets [11] in general are light weight DNNs used in mobile and embedded applications for a wide variety of applications from image recognition to object detection to even geo-localization. These DNNs are popular in these contexts because of their smaller memory footprint in terms of its parameters and efficient latencies even when executed with modest compute, available on edge devices like smartphones. While there has been a general trend of increasing complexity and size to reach high levels of accuracy when developing DNNs in the past few decades, this opposite approach out of Google aimed to pair high accuracy with small yet latency efficient networks.

Choices in DNN architecture such as number of parameters, network depth, and precision type are critically important as they all significantly impact the performance of the underlying hardware and should be considered when tuning for performance. More impor-

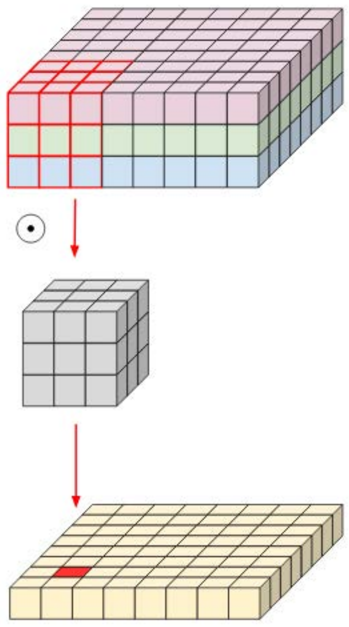


Figure 2.1: Graphical representation of a normal convolution [18]

tantly, however, is the choice of layers in the DNN architecture which dictate the latency and throughput of any targeted accelerator. Depending on the layer, or kernel, attempting to offload computation to accelerators may sometimes even slowdown the overall execution of the network due to accelerator overhead or data communication costs among many different variables. This shows that programmers must think carefully about how to accelerate a specific workload given both the available configuration of hardware accelerators and software workload.

2.2 Depthwise Separable Convolutions

The MobileNet network architecture is heavily based on a special type of kernel known as the depthwise separable convolution. This type of convolution, which reduces complexity, has steadily grown in popularity being found in many modern DNNs outside of MobileNets most notably in the Xception DNN [4]. According to the researchers in the MobileNet paper, depthwise separable convolutions can result in between 8 to 9 times less computation with only a small drop in accuracy. This almost magnitude of reduction in computation guarantees less overall operations in the network but not necessarily an increase in performance, which will be evident in Chapter 3.

What is a Depthwise Separable Convolution?

The depthwise separable convolution kernel can be broken down into two distinct parts (1) a depthwise convolution and (2) a 1x1 pointwise convolution. The goal of this two-part layer is to decrease the overall computation while still maintaining the richness of the activations gained from a normal convolution layer. By having a depthwise convolution followed by a pointwise convolution, the depthwise separable convolution splits the filtering and accumulation of a single convolution into two separate operations. This is beneficial in terms of reducing the number of operations to be executed but can lead to performance degradation when mapped naively.

Mathematically the depthwise convolution is computed as follows:

$$G_{k,l,m} = \sum_{i,j} K_{i,j,m} * F_{k+i-1,l+j-1,m}$$

where K is the depthwise convolutional kernel of size D_K x D_K x M where m_{th} filter in K is applied to the m_{th} channel in F to produce the m_{th} cahnnel of the filted output feature map G [12].

The separable part of the depthwise separable convolution is a 1x1 pointwise convolution that comes after the depthwise convolution. This 1x1 pointwise convolution is needed to generate the new features of this entire layer. However, this work moving forward will focus only the depthwise convolution stage of the depthwise separable convolution.

Mapping to Hardware

The biggest difference of concern between a normal convolution and a depthwise convolution is properly handling the n number of channels present in both the input of the layer and the filters of the layer. With a normal convolution, the underlying hardware has several different methods to execute the kernel. The most intuitive approach is to implement a direct

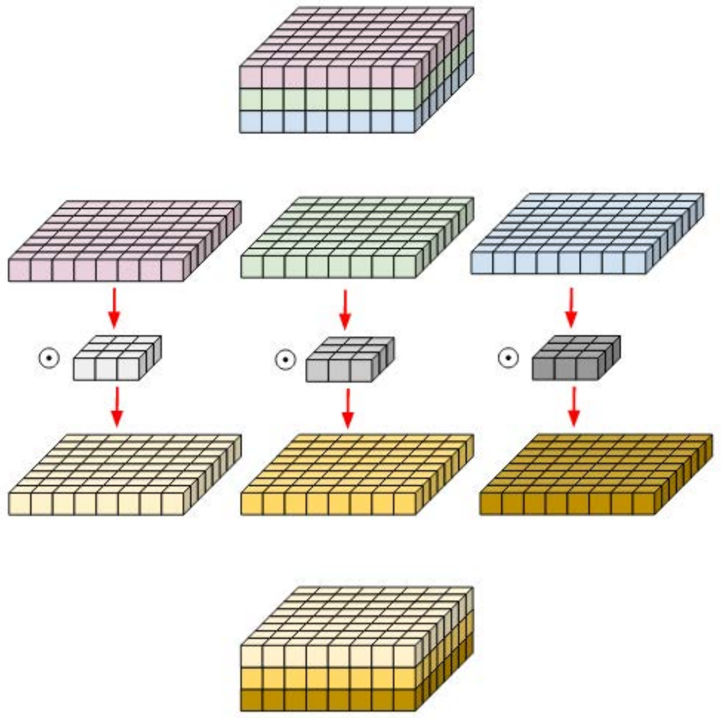


Figure 2.2: Graphical representation of depthwise convolution without tail end pointwise convolution. [3]

convolution kernel which strides across the inputs and filters in a linear fashion multiplying and accumulating as it strides. This kernel can start to be optimized for caches by studying the memory access pattern of the kernel and for the processor by taking advantage of fused multiply add instructions among many other optimizations.

Another approach to executing a normal convolution in hardware is to transform both the inputs and weights into matrices that can then be multiplied in a straight forward fashion. Even though this conversion process, known as im2col, adds overhead to executing a convolution and increases data redundancy within filters, depending on the processing elements available in a machine this method can result in the most efficient performance. This can be attributed to the fact that matrix-matrix multiply operation is heavily studied in research literature and optimized in practice. There are many other different methods in implementing a convolution kernel but these are the two most important to this work.

When moving from mapping convolution to mapping depthwise convolution, both approaches mentioned previously still apply. However, the performance impacts of each are magnified because of the structure of the depthwise convolution and its lack of a single fused multiply add computation. This results in performance degradation when trying to follow the direct convolution approach in mapping this kernel. With the im2col case, because of the increase in filter channel dimension this method also suffers from significant performance reduction because of the eventual size of the matrix multiplication, which will be analyzed in Chapter 3.

2.3 DNN Parallelism

This work, in later chapters, will discuss the performance, obstacles, and potential solutions for various levels of parallelism within DNNs. Specifically, Chapter 5 will focus on workload level parallelism and the resulting resource contention, while Chapter 6 will focus on intra-layer and batch level parallelism within MobileNet. For the sake of clarity and future reference this section will briefly describe and introduce the different layers of parallelism that can be exploited in the context of DNN accelerators and motivation for pursuing each type of parallelism including some related work.

Workload Level Parallelism

Workload level parallelism often describes when multiple unique workloads or multiple inferences of the same workload are colocated onto hardware in order execute parallelly. This is a key space in which previous and current research is focused to provide support for parallel inferences in AR/VR applications where tasks such as segmentation and classification need to be performed in parallel [16]. In addition, the autonomous vehicle space is exploding with tons of novel DNNs and multi-inference environments in on-edge computing with NVIDIA's Drive platform and complete vehicle SoC, AGX Orin [22]. This level of parallelism will drive the work in Chapter 5 where this work will examine how to improve multiple different inferences of unique models that can be executed in parallel.

Intra-Layer Parallelism

Intra-layer parallelism refers to different sequential or parallel layers within a singular workload being mapped to unique accelerators to speed up execution. In this level of parallelism, there are two main factors to consider: (1) parallel independent layers and (2) priority assignment of layers. For example, the first factor is highlighted in DNN workloads like GoogleNet that have multiple independent layers that could be executed in parallel which could result in huge performance gains. Given the right hardware configuration, it is possible to accelerate the performance and reduce the latency of the entire inference by taking advantage of the branching within the DNN architecture and executing these independent layers parallelely.

With regards to the second factor, cleverly assigning priorities to have some specific layers execute on targeted accelerators in a heterogeneous system could boost the performance as well. For example, in the execution of MobileNet using a vector accelerator to execute depthwise convolutions and a systolic accelerator to accelerate normal convolution layers might yield better performance while trading off data movement costs and blocking latency, which will be explored in Chapter 6.

Inter-Layer Parallelism

Finally, inter-layer parallelism often refers to the lowest level of DNN parallelism where each kernel or layer itself can be broken up into smaller tiles of work to be scheduled and executed across multiple homogeneous or heterogeneous accelerators. This would show huge performance benefits for compute bounded kernels and could potentially scale linearly with the number of accelerator cores available in the system. However, data movement and synchronization costs would also grow with the increase in tiling and scheduling, introducing a critical tradeoff. This is a very interesting space that will not be discussed further in this work but will rather be a topic left for future work.

2.4 Summary

This motivating workload with MobileNetV2 and interesting kernel with the depthwise separable convolution sets up the rest of this work to explore the acceleration of this kernel and overall use of multiple hardware accelerators to parallelize and accelerate this workload.

Chapter 3

Architecture and Framework

This section will provide an overview of the tools, frameworks, and architecture configurations that are used and most significant to rest of the work. Most of these open source tools developed through Berkeley ADEPT are available in the Chipyard framework and have enabled countless studies like this to be possible.

3.1 Workflow

From initial stages of design exploration to workload generation to final FPGA based performance simulation of custom system-on-chip (SoC) designs, Berkeley ADEPT's Chipyard [1] framework contain the tools for every step of a full-system design and evaluation. This work's baseline architecture starts with a RISC-V based Rocket [2] core as the CPU attached to Gemmini, a systolic-array based DNN accelerator via RoCC interface. DNN workloads are generated by using a combination of custom software stacks specific to each accelerator along with ONNX Runtime [7] to produce our final linux binaries. In addition, Spike is used for behavioral simulations for accuracy verification and FireSim [15], a cycle-accurate, FPGA-accelerated RTL simulator is used for performance analysis. This work then use data profiling scripts to perform memory cache analysis and workload kernel residency graphs. This is the general workflow from SoC configuration and software generation to RTL simulation and data analysis throughout each study in this work.

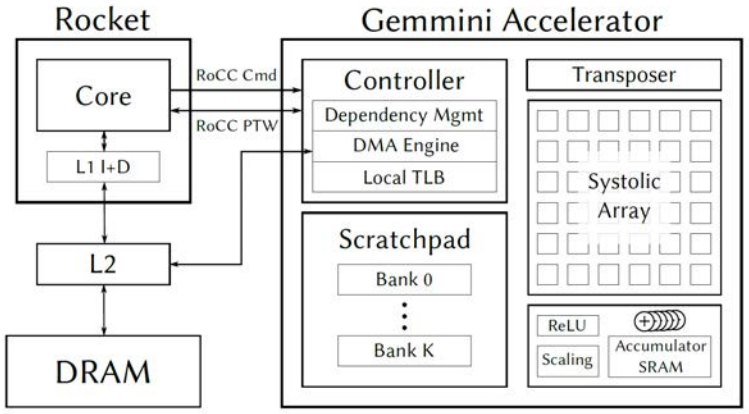


Figure 3.1: Gemmini systolic accelerator system architecture attached to Rocket RISC-V CPU Core [9]

3.2 Gemmini

The Gemmini [9] accelerator is a compile- and runtime configurable systolic array that is made up of individual processesing elements that are connected in a grid like network. This type of DNN accelerator architecture has been widely used in industry and academia to accelerate matrix multiplication and convolution kernels. This work takes advantage of the Gemmini generator, embedded within the Chipyard framework, to produce various instances of this systolic accelerator that interface with our main Rocket RISCV CPU.

Microarchitecture and Memory Model

In addition to the configurable number of processing elements, Gemmini also contains its own DMA engine, local translation lookaside buffer (TLB), and scratchpad. This microarchitecture is important to keep in mind when programming Gemmini because of its memory and overall system implications to our workload. Gemmini's DMA engine directly connects to a shared L2 cache via TileLink [5] crossbar which then connects to the system's main DRAM. Chapter 5 of this work will discuss shared resource contention that occurs especially when connecting multiple accelerators and their memories and propose modifications to this memory model to alleviate this problem.

Parameter	**Value**
Systolic array dimension	16x16
Inner scratchpad size	64KiB
Inner accumulator size	32KiB
Shared L2 size	2MB
Shared L2 banks	4
DRAM bandwidth	16GB/s
Frequency	1GHz

Table 3.1: SoC configurations used in evaluations.

Programming Model

The main CPU and Gemmini accelerator communicate through custom instructions specific to the accelerator. These instructions allow the CPU to offload specific kernels and computations to the accelerator while allowing the CPU to proceed on other non-dependent tasks. These instructions are communicated via Rocket Custom Coprocessor Interface (RoCC) to the Gemmini accelerator for decoding and execution. These custom instructions are predefined and give programmers the ability to move data in/out of Gemmini and compute data at the lowest level. Through the work of the Gemmini project, custom libraries with matrix multiplication and convolution kernels were developed exposing a higher level interface to Gemmini and will be used in this work.

3.3 ONNX Runtime

In addition to the aforementioned tools, this work builds upon previous work [19] to build and integrate ONNX Runtime into Gemmini's software stack. ONNX itself is a DNN model format that is used to describe the representation of a particular DNN regardless of framework. ONNX Runtime on the other hand, at its core, is Microsoft's machine learning runtime engine that is able to consume and execute any ONNX model by assigning and scheduling various hardware providers to software workloads based on graph optimizations. This framework gives us the freedom to first explore any number of different providers executing any type of DNN workloads; and second, create custom backends for each of our unique accelerators. Chapter 4 will go into more detail on our ONNX Runtime integration when discussing our implementation of our accelerated depthwise convolution kernel.

Chapter 4

Accelerating the Depthwise Convolution

This section will discuss the results of the baseline performance of a MobileNet inference and use it as motivation to accelerate the depthwise convolution kernel. Then this section will describe our vector accelerator, Hwacha, its unique programming model and the effects of it on our accelerated kernel. Finally this section will conclude with the integration of the accelerated kernel into the ONNX Runtime backend and our performance results when using Hwacha compared with our baseline data.

4.1 MobileNet Baseline

As discussed in Chapter 3, the baseline architecture configuration consists of a singlecore Rocket RISCV CPU attached to a single Gemmini systolic array accelerator. The existing framework inside of the Gemmini software stack is able to run MobileNet in three different configurations to see where the accelerator's performance degrades the most.

1. CPU Baseline: All convolution kernels are run on the CPU making no use of Gemmini

2. Gemmini Matmul: Non depthwise convolution layers are accelerated using Gemmini
3. Gemmini Conv: All convolution layers are accelerated using Gemmini

Cycle Breakdown	**CPU Baseline**	**%**	**Gemmini Matmul**	**%**	**Gemmini Conv**	**%**
Matmul Cycles	1.0×10^{10} c	68%	4.5×10^{7} c	0%	4.3×10^{7} c	13%
Im2Col Cycles	2.7×10^{7} c	0%	2.7×10^{7} c	0%	0 c	0%
Conv Cycles	0 c	0%	0 c	0%	5.1×10^{6} c	1%
Depthwise Conv Cycles	4.6×10^{9} c	30%	4.6×10^{9} c	98%	2.6×10^{8} c	83%
Total Cycles	1.5×10^{10} c	100%	4.6×10^{9} c	100%	3.1×10^{8} c	100%

Table 4.1: Baseline Gemmini data showing the three different configurations and its individual break down of cycles spent with different kernel operations.

As seen in Table 4.1 these three different approaches yield wildly different performances in terms of cycles spent in each kernel. As expected the non-accelerated, CPU only configuration performs the worst in terms of overall cycles and in each individual kernel. In the second configuration, Gemmini Matmul, on an initial glance, the data shows a massive speed up, around 225x, achieved by Gemmini when accelerating all the non depthwise convolutions. However, upon close observation, the total execution cycles has only decreased by a factor of 3x. This is precisely due to the depthwise separable convolution layers within MobileNet that dominate the CPU when not accelerated. Note that both the CPU Baseline and Gemmini matmul implementations use im2col to do matrix multiplies which is seen in the constant number of cycles spent in both configurations.

The final data point, Gemmini Conv, shows the performance when all convolution layers, including the depthwise convolution layers, are accelerated using Gemmini. Impressively, the data shows an overall speedup of around 48x; however, when looking at the breakdown of where Gemmini spends the most cycles it is noted that the depthwise convolution still dominates the performance whether run on CPU or an accelerator.

Not Good At Everything

Gemmini, a systolic based accelerator, is highly optimal for normal convolutions because of its processing element structures and interconnects. In addition, the scratchpad and accu-

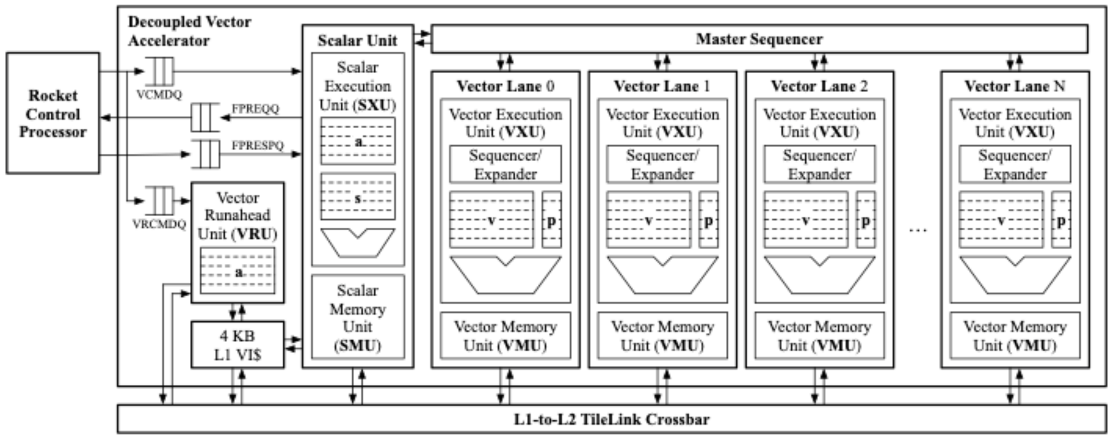


Figure 4.1: Hwacha decoupled vector accelerator block diagram [17]

mulator units help Gemmini execute matrix multiplications with high efficiency and speed. However, when Gemmini is repurposed to execute depthwise convolutions it is suboptimal because of the kernel itself and its mapping on to Gemmini. When depthwise convolutions are run on Gemmini, each layer/channel of activations and filters is an independent operation since there is no accumulation across channels. Each input and output pixel is mapped to a row in the scratchpad and accumulator which is highly inefficient for memory. Gemmini normally when executing standard layers improves in efficiency as the compute workload is increased when layers have higher channel sizes and more parameters. However, with depthwise convolution, with higher channel sizes, since Gemmini treats each individual channel as an independent operation, the execution lowers utilization and is highly inefficient.

4.2 Hwacha

Hwacha is a decoupled vector-fetch accelerator that attaches to the main Rocket RISC-V based CPU via RoCC. Hwacha excels at accelerating data parallel workloads as it features parallel vector lanes each with its own execution and memory unit. To optimize data throughput, programmers are able to configure the length and partitioning of Hwacha's vector register file. This allows the programmer to decide how to partition the vector data registers depending on how many different pieces of data will be used in a specific kernel and the precision of the data itself.

Programming Model

Hwacha at its core is built like a traditional vector accelerator except for some unique differences in its programming model and interface. One popular way of programming accelerators in general, is to embed custom accelerator instructions into a program's overall set of instructions. When the program running on the CPU reaches a custom instruction it cannot execute, it forwards it to the right accelerator to execute and return. Another, also popular, programming model, SIMT, is to treat the parallel-data as multiple different threads such that each thread executes a piece of the overall kernel. Finally, one last SIMD approach to programming traditional vector accelerators is to use vector specific intrinsic instructions that change based on the data type, vector length, and available hardware support.

Hwacha's unique programming model tries to take the best of each of the previous mentioned methods in that it decouples work into a control thread and worker thread. When programming a sample kernel, the non-vectorized code (e.g. stripmine for loops, calculating addresses, moving pointers) can be decoupled from the actual meat of the kernel where the computation lies. In this programming model, the control thread executes all of the non-vectorized code and then launches a worker thread to Hwacha when it reaches a `vector fetch` instruction. This critical control instruction is responsible for communicating from the CPU to Hwacha where the vectorized portion of the code is and what to execute. When called, Hwacha will fetch these instructions, decoupled from the control thread CPU and execute them, allowing the control thread to runahead and process more non-vectorized code that is independent of the executing computation.

4.3 Accelerated Kernel

This section describes the accelerated Hwacha depthwise convolution kernel in detail; however, the algorithm figures are summarized for brevity.

Algorithm 1 Depthwise Convolution Worker Thread

```
setvcfg(0, 1, 4, 1)
count, consumed = 0, setvlen(channels)
for b in batch_size do
    for out_y in output_height do
        for out_x in output_width do
            consumed = setvlen(channels - count)
            va0 ← $output\_ptr$ + offset
            for filter_y in filter_height do
                for filter_x in filter_width do
                    va1 ← $input\_ptr$ + offset
                    va2 ← $filter\_ptr$ + offset
                    vector fetch mac
                end for
            end for

            va0 ← $output\_ptr$ + offset
            vs1 ← $real\_multiplier$
            vector fetch scale
            count += consumed
        end for
    end for
end for
```

Because depthwise convolution has data parallelism between the inputs and filters across all of the channels for a given layer, this kernel's implementation programs Hwacha to configure its vector length to be the `channel_size` of a given layer. This allows us to exploit instruction and data level parallelism when striding through this kernel. As the kernel iterates through the batch, output height, and output width, for each filter height and filter width the kernel will load an input vector and filter vector of length `channel_size` into Hwacha to multiply and accumulate.

In Algorithm 1, the first `setvcfg` control instruction configures Hwacha's vector register file to have 1 single-precision vector, 4 half-precision vectors and 1 predicate register. Then, when calling the `setvlen` command with `channel_size` as the argument, Hwacha will try to configure and partition its vector register file such that each vector declared with the previous `setvcfg` command will be of length `channels`. If Hwacha is unable to meet these constraints the `setvlen` call will return with the actual length of each of the vectors. This unique Hwacha feature allows programmers to be able to easily tradeoff precision for throughput within any kernel. In this particular instance, since executing a quantized `int8` inference of MobileNet, the vector register file can be optimized by using half-precision vectors for activations and filter weights and a single-precision vector for outputs to guard against multiplication overflow.

Algorithm 2 Depthwise Convolution Vector Fetch Blocks

```
.globl mac
mac:
    vpset vp0
    vlb vv1, va1 # input
    vlb vv2, va2 # filter
    vlh vv3, va0 # output
    vmul vv1, vv1, vv2
    vadd vv1, vv1, vv3
    vsh vv1, va0
    vstop

.globl scale
scale:
    vpset vp0
    vlh vv0, va0 # output
    vfcvt.s.w vv0, vv0
    vfmul.s vv0, vv0, vs1
    vfmin.s vv0, vv0, vs2
    vfmax.s vv0, vv0, vs3
    vfcvt.w.s vv0, vv0
    vsb vv0, va1 #store to output
    vstop
```

Once iterating through each batch, output height, and output width, the code can then for each index of the filter, set up Hwacha to do a multiply and accumulate. To clarify, as mentioned previously depthwise convolution does not accumulate across channels like normal convolution, but instead accumulation here is referring to the filter accumulation across the width and height of each filter. After setting up the Hwacha address registers pointing to the input, filter, and output the `vector fetch` instruction is finally called to decouple our worker thread and let Hwacha fetch and execute the `mac` vector block, described in Algorithm 2. The last step in the loop iteration after accumulating across filter height and width is to scale the activations such that they fall in the range of `int8`. The `scale` vector block rescales the output vector by using Hwacha instructions for type conversion and stores the final partial output from the accelerator into memory. The `count` and `consumed` variables ensure that the kernel performs correctly even when the `channel_size` is larger than the actual configured vector length inside Hwacha.

Integration with ONNX Runtime

As a part of this work, the ONNX Runtime backend has been extended by providing Hwacha support for this particular depthwise convolution kernel. Enabling Hwacha as a provider in the ONNX Runtime framework allows us to take advantage of the overall framework for testing and performance analysis of this kernel and Hwacha. With advanced profiling and versatility to run parameter sweeps of different comparisons, ONNX Runtime wraps and abstracts the software and hardware stack while still allowing for studies like this. In addition to performance studies, because of this integration there is on going work trying to bring up a taped out chip that has Hwacha using ONNX Runtime generated binaries to test. Chapter 6 will discuss dual running and threading features enabled by ONNX Runtime for this work.

4.4 Performance Results

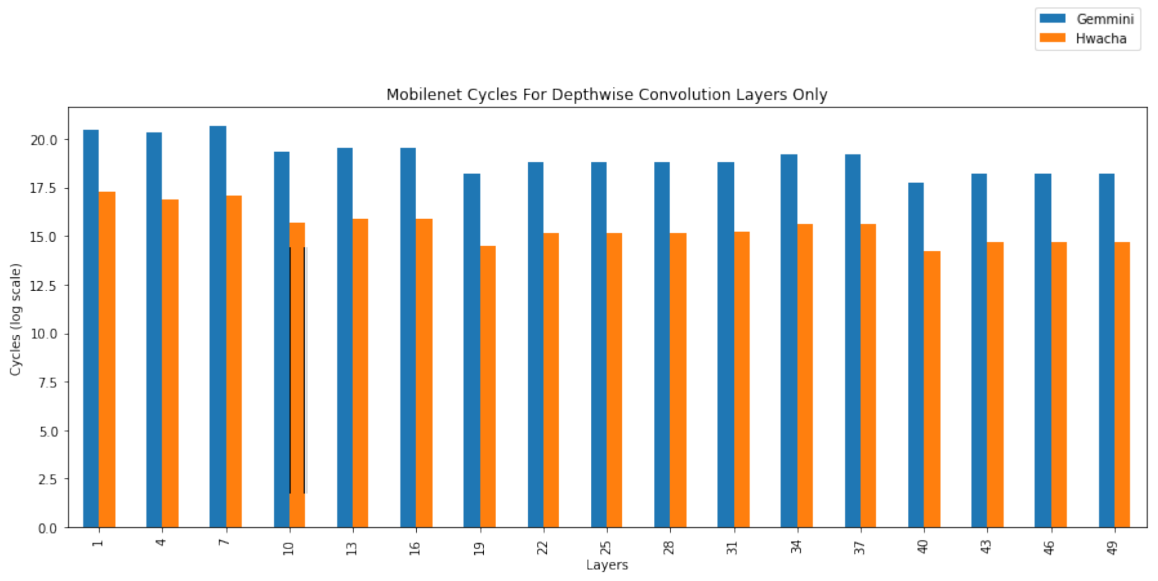


Figure 4.2: Performance comparison between depthwise convolution executed on Gemmini versus Hwacha in a log scale. On average shows around

Chapter 5

Shared Resource Contention

This section will build off of the accelerated depthwise convolution kernel described in the previous chapter and focus on the specific issue of shared resource contention when running multiple accelerators parallelly. As mentioned in Chapter 2, this section will mainly focus on workload level parallelism while Chapter 6, will focus on intra-layer and batch level parallelism. This section will start with some related background work done to address resource contention and move on to the setup, configuration, and results. Finally, this section will conclude with a few different potential solutions and reference a closely developed work that addresses this exact issue.

5.1 Background and Related Work

Previous works examining memory bottle necks of DNN accelerator level parallelism have shown the wide gap in performance and have tried to address them in many different ways. While some papers tend to study a specific pairing of use cases with one ML workload and background host CPU tasks, it is important to properly explore all parts of the accelerator parallelism space. [24] This usecase of one ML workload paired with host CPU tasks mimic a homogeneous server in a warehouse scale computing environment where clients are continuously requesting the execution of some ML workload. However, this is completely different from the edge computing design point where the SoC has numerous wildly different accelerators available to use and the CPU acts more like a scheduler and synchronizer between

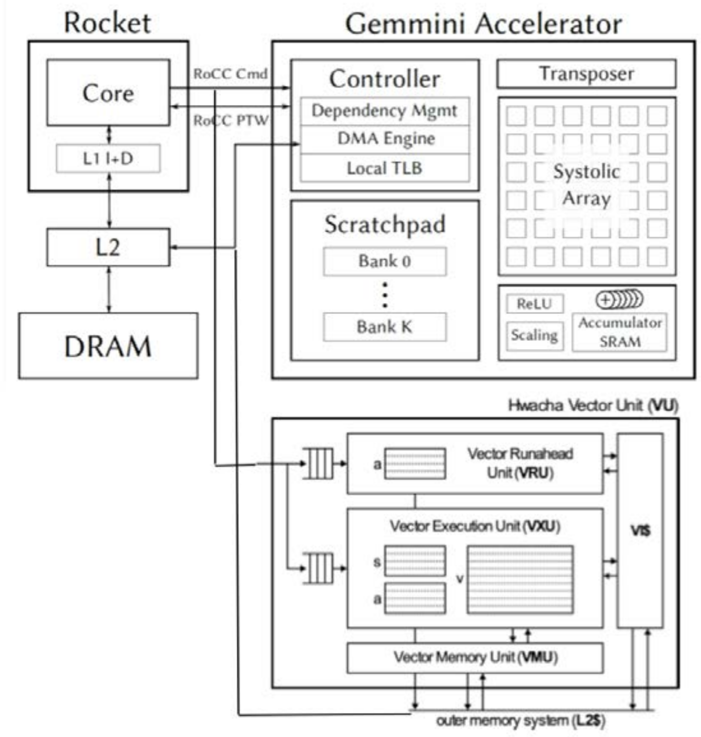


Figure 5.1: Heterogenous system architecture with Hwacha vector accelerator and Gemmini systolic accelerator attached to Rocket RISC-V CPU Core

the accelerators. Since this is a very wide design space, this work will focus on the resource contention in the later context.

When it comes to solutions, one common approach is to do static resource partitioning on the shared resources that accelerators tend to contend for. However, this method has shown to create memory back-pressure in low priority tasks and is untested with more than one high priority ML workload. Another approach is to use a runtime module that has a complex scheduling algorithm that helps alleviate the memory requests by monitoring vital memory statistics. [16] However, even these hardware and software solutions are tied closely with the workload and system architecture making it limited to implement in other systems.

5.2 Setup and Configuration

Architecture

In this work, a heterogenous architecture configuration consisting of both Gemmini and Hwacha is used to study the performance impacts of executing DNN workloads on a multi-accelerator system. Hwacha and Gemmini are both attached to the same RISC-V Rocket CPU and have their own vector register files and scratchpads/accumulators (respectively) but will still contend for L2 cache space in the memory subsystem. In this baseline configuration there is a 2MB L2 cache with 4 bank ports that is later varied in the study to see cache size impacts.

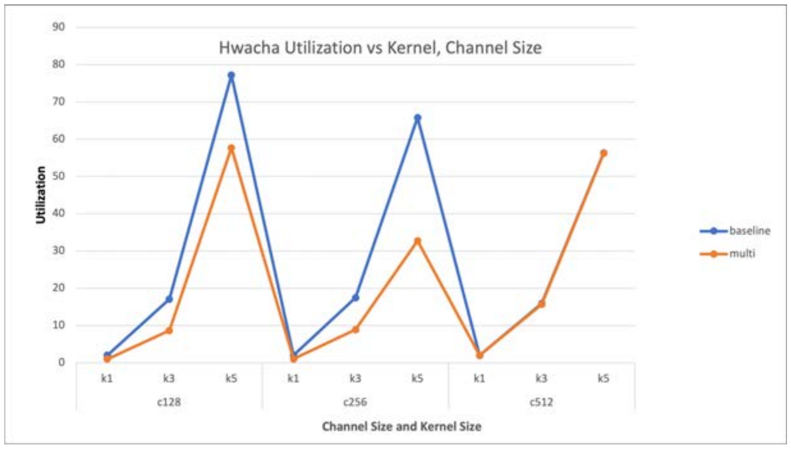
Workloads

To occupy the systolic array accelerator, Gemmini, this work primarily relies on a quantized ResNet50 model, which is a deep and heavy DNN workload. It is expected that this ResNet50 inference should occupy and highly utilize the Gemmini accelerator with its numerous convolution layers, also while running quantize, batchnorm, and other unsupported kernels on the Rocket core.

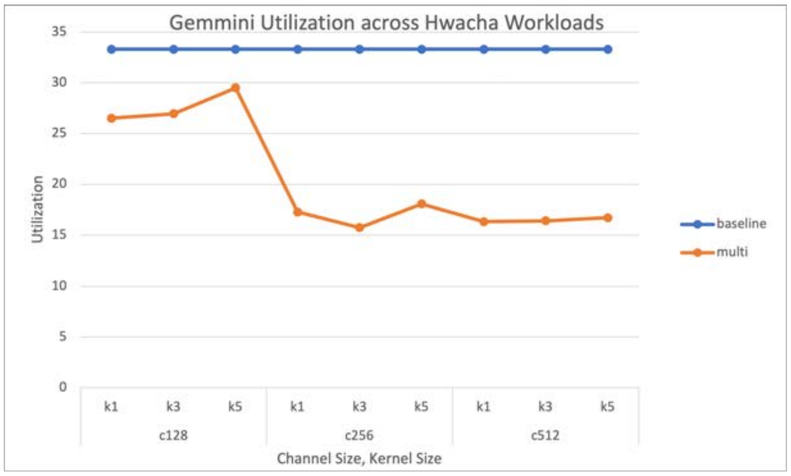
Meanwhile for Hwacha, because of it's low throughput and performance for integer GEMM operations, synthetic DNN workloads were created that consist of back to back depthwise convolution kernels that would highly utilize Hwacha. This design choice was made, following guidance from previous work, so that we could realistically simulate workload level parallelism environments. Both DNN workloads, ResNet50 and synthetic depthwise convolutions, are quantized inferences that use the `int8` type.

5.3 Performance Results

With the discussion of the results of this study, it should be noted that the `baseline` case in all the results was conducted with the same heterogeneous architecture configuration (1 Rocket attached to Gemmini + Hwacha) but without utilizing one of the accelerators, while the `multi` case utilizes both accelerators. In addition, utilization in the following results



(a)



(b)

Figure 5.2: In 5.2a Hwacha's utilization degradation is shown in the baseline case versus the multi use case when only Hwacha is executing versus both Gemmini and Hwacha are being utilized (respectively). In 5.2b Gemmini's utilization degradation is shown with the same configurations as the previous figure.

are calculated as $\frac{IdealCycles}{ActualCycles}$ where *Ideal Cycles* comes from the $\frac{\text{\# of MAC operations in the workload}}{\text{Max MACs per Cycle}}$ where Gemmini's and Hwacha's max throughput for integer MACs is assumed to be `256` and `8`, respectively.

Overall Utilization

To analyze performance and utilization at a very high level, two parameters in the synthetic depthwise convolution workload assigned to Hwacha were chosen to be varied: kernel size and channel size. This allows for the observation of how each accelerator responds to changing workloads. Increasing channel size is expected to produce more data movement through the entire system inducing more resource contention and the opposite for kernel size as less data will be involved with larger kernel dimensions.

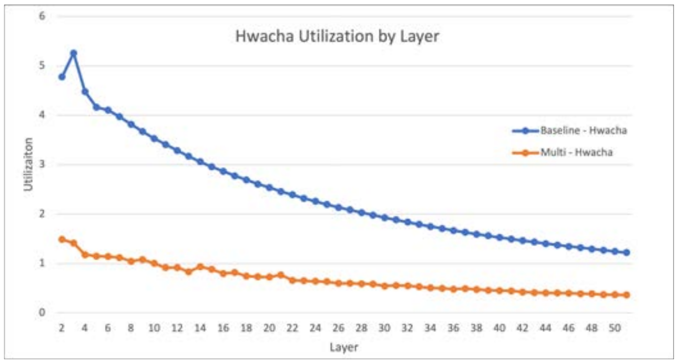
As Figure 5.2a shows, there is almost a 2x degradation in utilization of Hwacha when Gemmini is executing its workload (Resnet50) in parallel. Another interesting observation is the saturation of the performance when the channel size is increased to 512. The baseline and multi utilization statistics look almost identical since it reaches a saturation point of memory and compute for Hwacha for this particular workload.

Next, the same sort of performance impact is noticed when looking at Gemmini's utilization. As seen in Figure 5.2b, the baseline ResNet50 workload overall utilizes Gemmini around 33%, but in the `multi` case when Hwacha is executing it's own independent workload, Gemmini's utilization significantly worsens as Hwacha's workload gets larger, indicating competition between the accelerators over cache lines in the shared L2.

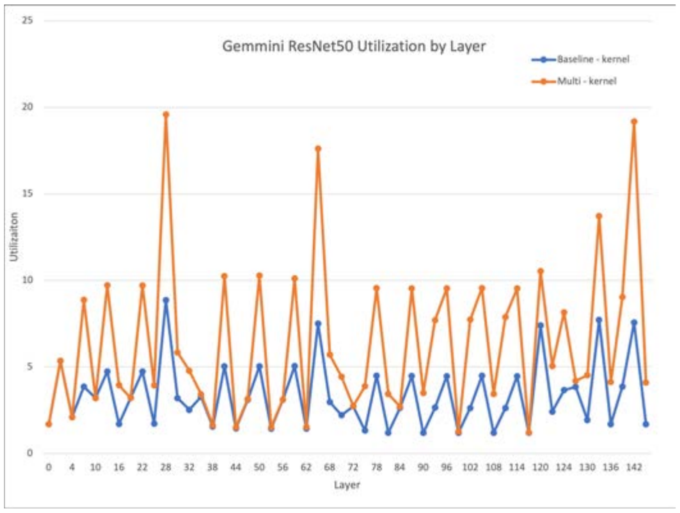
Layer by Layer Utilization

Looking closer at a single data point shown above, specifically the case where channel size is 128 and kernel size is 1, we can see detailed behavior of each accelerators layer by layer performance in Figure 5.3a. Hwacha's performance shows both baseline and multi utilization numbers degrading as the accelerator gets deeper and deeper into the workload. To show the extent of the degradation, when compared to the baseline Hwacha utilization, there is only one layer out of 50 in the multi case where the utilization of Hwacha is slightly more than worst case utilization in the baseline. In addition, some irregular changes were noticed in utilization in the multi case versus the smooth regular change in the baseline, which can be explained by Gemmini's execution.

Looking at the layer by layer analysis of Gemmini's ResNet50 execution in Figure 5.3b a pattern of degradation of utilization at very regular intervals can be seen. The multi case



(a)



(b)

Figure 5.3: In 5.3a Hwacha's utilization degradation of executing back to back depthwise convolution kernels with channel size 128 and kernel size 1 is shown in the baseline case versus the multi use case when only Hwacha is executing versus both Gemmini and Hwacha are being utilized (respectively). In 5.3b Patterned layer by layer utilization degradation of Gemmini systolic accelerator when running ResNet50 with Hwacha being utilized by depthwise convolution workload is shown.

follows the utilization of the baseline almost exactly except for very specific layers, indicative of some memory contention. This could be caused by bank conflicts from competition over the TileLink memory ports that serve both accelerators with data.

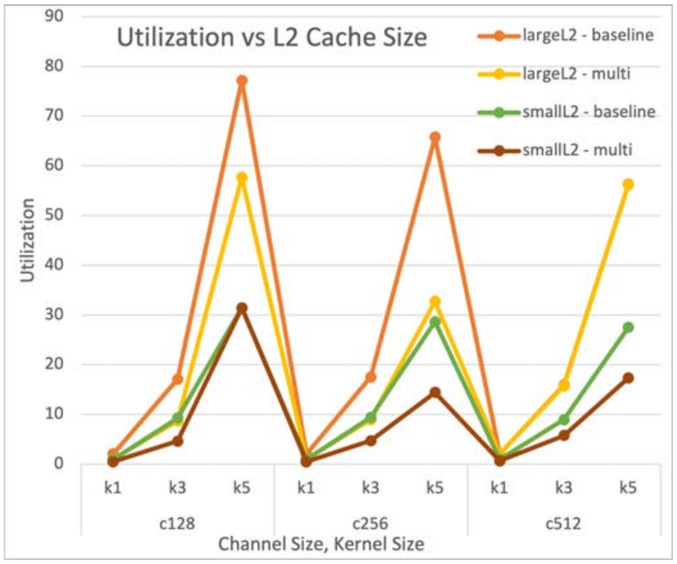


Figure 5.4: Hwacha utilization of baseline and multi cases varied by L2 Cache Size

L2 Cache Variance

Another parameter of consideration is the size of the L2 cache to see its impact on performance. As expected, lowering the L2 to a 512KB, 1 Bank configuration lowered the memory throughput overall and bottle necked both accelerators. As shown in Figure 5.4, the baseline with the smaller cache configuration has a lot lower utilization than even the multi case with the large cache. Looking closely, there is less performance degradation due to resource contention in the smaller cache configuration, since cache resources are already throttling the accelerators and diminishing there performance.

5.4 Solution Space

This work proposes two solutions to alleviate resource contention of memory resources with multi accelerator configurations: (1) Accelerator Aware Execution and (2) Priority Memory Arbitrator. This section will describe both potential solutions and then describe a closely related and developed work that implements a combination of ideas from the potential solutions.

Accelerator Aware Execution

This potential solution requires each or one of the accelerators in the configuration to be able to stall memory requests when detecting other accelerators using the TileLink ports. This solution has potential for deadlock and sequential execution in worst case scenarios; however, has a tremendous upside in reducing bank conflicts between accelerators which contributes to a significant amount of the utilization loss. The hardware implementation of accelerator aware memory execution could also be as simple as a flag with the TileLink port and another input into the accelerator. This however constrains the hardware design of each accelerator added to the SoC and requires each to synchronize and coordinate execution in some way.

Priority Memory Arbitrator

Another potential solution is a priority memory arbitrator that sits in front of the TileLink crossbar to facilitate memory requests between accelerators. In the current configuration, Gemmini and Hwacha communicate with a 2MB, 4 bank L2 cache through the TileLink ports and crossbars. Serious performance and utilization degradation discussed above is caused by the round-robin handling of memory requests within the TileLink crossbar. As exemplified in previous literature, assigning priorities per workloads could boost memory performance and alleviate the pressure and dependency on the memory system for workload performance as well.

Contention-Aware Accelerator Architecture for Multi-Tenacy Execution (CALM)

This work implements a light-weight, contention-aware multi-tenancy architecture for DNN accelerators that was tested specifically on a multi Gemmini configuration with a variety of DNNs. This novel solution, CALM, dynamically manipulates memory access rates based on latency targets such that co-located applications get the resources they demand without significantly starving their co-runners. In implementation, CALM increases service-level agreement satisfaction rate up to 8.6x (2.3x overall), while also increasing system throughput by 1.11x, fairness by 1.38x, and performance variation by 8x.

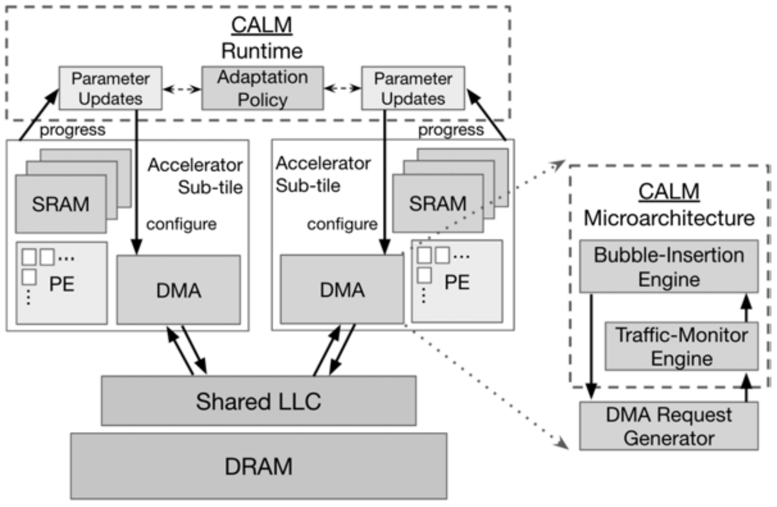


Figure 5.5: CALM-augmented DNN accelerators on an SoC to support dynamic contention-aware execution

CALM consists of 1) a light-weight hardware monitoring and regulation engine to track and control the amount of memory traffic injected into the shared resources and 2) an intelligent runtime system to dynamically manage the contentiousness of workloads based on user-specified adaptation policies. In particular, CALM leverages the regularity of DNN operators and hardware to manipulate the memory access rates using two adaptation policies: balanced and guaranteed policy. These two policies determine the priorities of the co-located DNN workloads. Using the the traffic-monitor engine to track the realtime memory access rate at which DMA issues load reques to shared caches and DRAM during the monitored time window, and the bubble-insertion engine that inserts "bubbles" that prevents the DMA from sending any further memory requests, the CALM microarchitecture facilitates the shared use of the memory subsystem preventing contention and starvation.

Chapter 6

Pipelined Execution of MobileNet

Instead of running accelerators with different workloads in parallel like the previous chapter, this section will focus on executing a singular, unified workload in MobileNet on multiple, unique accelerators. This section will address the challenges, performance, and optimizations needed when pipelining a DNN workload across a heterogenous architecture by taking advantage of model and batch level parallelism. Finally, this section will include performance results of a multicore architecture running the same workload and discussion of synchronization overhead.

6.1 Introduction and Related Work

When faced with the problem of mapping the execution of a DNN on some particular architecture, many intertwined factors are at play and can lead to unexpected behavior in performance. This widely popular problem in computing today has resulted in many different explorations in solutions. One common approach is to use a runtime module that has a complex scheduling algorithm that helps partition and allocate pieces of workloads to hardware processing units. They can help alleviate memory requests by monitoring vital memory statistics and controlling the flow of workload traffic, acting as a supervisor. Another solution, is to statically partition shared resources that accelerators tend to contend for. This method, however, has shown to create memory back-pressure in low priority tasks and is untested with more than one high priority ML workload. [16]

In the context of inference, some works have explored hardware solutions such as mapping different layers of the network on different FPGA's all together to increase chip utilization; however this proves to be costly in terms of power and communication. [21] Other works have explored running deep learning inference on two different architectures, specifically GPUs and NPUs, but are constrained by there software optimizations specific to the architecture itself. [13]

In the context of training, much research has gone into the space of distributed training across different homogeneous nodes, with intricate runtime or compiled schedulers to assign layers to processing elements [23] [14]. These works are useful starting points for inspiration on exploring the inference side however do not address the unique challenges of mapping inference across heterogeneous accelerator architectures. In addition, many of these hardware and software solutions are tied closely with the workload and system architecture making it limited to implement in other systems.

This work focuses on exploring how accelerators can take advantage of model and batch pipelining to yield performance improvements and have overlapped execution. In addition, this work will examine synchronization costs when comparing inference execution on a multicore heterogenous architecture. These factors, we hope, will help build on and improve the previous works mentioned.

6.2 Single Core Heterogenous Architecture

Batch Level Parallelism

Batch level parallelism refers to increasing performance and throughput of inference by batching images and inputs together into one larger input that can be passed through the entire DNN at the same time. Traditionally, many previous works have leveraged this technique to keep the utilization of accelerators and processing elements high. In addition, at a software level, batch level parallelism has been used to highly parallelize DNN workloads by running individual batches on single cores or nodes. This mostly straight forward approach can lead to large performance gains because of increased hardware throughput and utilization and lower overhead per kernel routine. However, when limited by a singular core and dependent

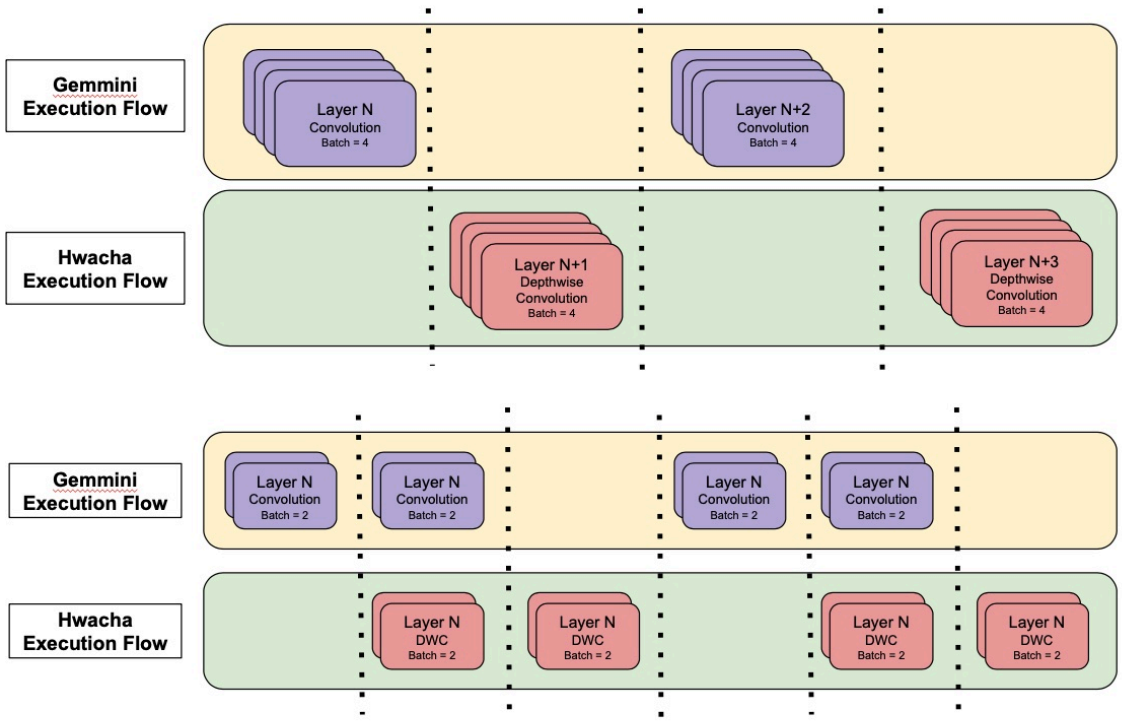

Gemmini Execution Flow
Layer N Convolution Batch = 4
Layer N+2 Convolution Batch = 4
Hwacha Execution Flow
Layer N+1 Depthwise Convolution Batch = 4
Layer N+3 Depthwise Convolution Batch = 4
Gemmini Execution Flow
Layer N Convolution Batch = 2
Layer N Convolution Batch = 2
Layer N Convolution Batch = 2
Layer N Convolution Batch = 2
Hwacha Execution Flow
Layer N DWC Batch = 2
Layer N DWC Batch = 2
Layer N DWC Batch = 2
Layer N DWC Batch = 2

Figure 6.1: Graphical representation of using model and batch pipelining to partition workloads across Gemmini systolic accelerator and Hwacha vector accelerator. The top diagram shows no pipelining, while the bottom image shows pipelining with a batch size of 2 across the two accelerators.

workload, this work looks to explore how to leverage batch sizes to gain the best overall performance.

Model Level Parallelism

Model level parallelism is a combination of inter and intra layer parallelism as described in Chapter 2. Many previous works have also explored this idea in inference and training to increase parallelism across the entire DNN workload which in turn boosts performance. [6] In previous works however, models are usually split across different homogenous cores and machines instead of heterogenous architectures like in this work which adds an extra level of complexity. When splitting across homogenous cores every kernel and piece of workload can be treated independently and scheduled on any hardware. With heterogenous accelerators,

Cycle Breakdown	**Batch = 4**	%	**Batch = 2**	%	**Batch = 1**	%
Normal Conv	4.7×10^7 c	25%	4.0×10^7 c	21%	3.2×10^7 c	17%
Depthwise Conv	1.4×10^8 c	75%	1.2×10^8 c	79%	1.1×10^8 c	83%
Total Cycles	1.9×10^8 c	100%	1.6×10^8 c	100%	1.5×10^8 c	100%

Table 6.1: This table shows the breakdown of cycles spent on normal convolution vs depthwise convolution across different pipeline variations with batch size.

mapping a DNN execution flow becomes complicated because of the lack of flexibility in scheduling and execution.

Performance Results

This study combines model and batch level parallelism to pipeline different layers of the MobileNet workload into our unique accelerators. With the architecture configuration being the same as in the previous studies, 1 Rocket Core + 1 Hwacha + 1 Gemmini, a pipelined workload for both accelerators is built to trade off utilization and throughput with the granularity of pipelining. As shown in Figure 6.1, the objective of partitioning this workload is to overlap the utilization of both accelerators when working on different pieces of data. The pipelined workload is mainly constrained by the dependencies existing in the DNN structure itself and the batch size parameter.

Accelerators can be more utilized when a DNN has more independent layers that each type of accelerator can be a provider for. In our particular case, MobileNet has a singular stream of layers, which makes our window for accelerator overlap very narrow. The batch size parameter allows us to toggle the amount of work per layer per accelerator. By combining the pipelining of batches and layers, performance boosts are expected from greater parallelism even when executing a highly dependent workload.

As seen in Table 6.1, there is an expected trend of reducing the number of cycles as batch size is decreased. In the first case of batch size being 4, there is no overlap between the execution of Gemmini and Hwacha. This scenario is similar to the very first depiction in 6.1 where after executing a normal convolution layer, Gemmini writes all of the output activations to memory and then Hwacha is called to launch its depthwise convolution kernel and execute the layer. As the batch sizes decrease to 2 and 1, there is an introduction of more overlap in execution with the two accelerators. For example, with the batch size of 2, when the MobileNet architecture has a normal convolution back to back with a depthwise convolution, Gemmini will first only compute the normal convolution for 2 images before writing the activations to memory. Then, Hwacha will start executing the next layer, the depthwise convolution kernel, on those 2 images previously outputted by Gemmini. Once initiated, while Hwacha is computing on the first two images, Gemmini can run ahead to the next 2 images to finish computing the previous convolution layer. Extrapolating even further, with a batch size of 1, software pipelining is extended even further and introduces a greater level of parallelism between accelerators. Throughout these configurations, pipelining is not implemented when there is no switch between accelerators, i.e. when there are multiple consecutive Gemmini/Hwacha computed layers.

It is important to also note that when decreasing batch sizes to increase pipelining throughput and accelerator parallelism there is potential that the overall utilization of each accelerator could actually be decreasing. For example, depending on the accelerators processing element configuration, if batch size is decreased enough, the accelerator could be potentially starved of and be underutilized. This tradeoff between accelerator parallelism and utilization is an important design space to consider when tuning workloads and mapping them to architectures.

Going back to the results in Table 6.1 there is a .13x and .27x gain in overall execution cycles when moving from a non overlapped configuration to a fine-grained batch pipelined configuration. Besides the cycles saved when accelerators are utilized in parallel, based on the layer by layer performance data this work hypothesizes that there are increased cache benefits when pipelining with a finer batch size. In Figure 6.2 a unique pattern can be observed when graphing the cycles taken on all non depthwise convolution layers. In the layer directly after a depthwise convolution, the cycles for the normal convolution actually increase even though there is no difference in software workload across batches; since batches are only pipelined in depthwise convolution layers and normal convolution layers directly before a depthwise convolution layer.

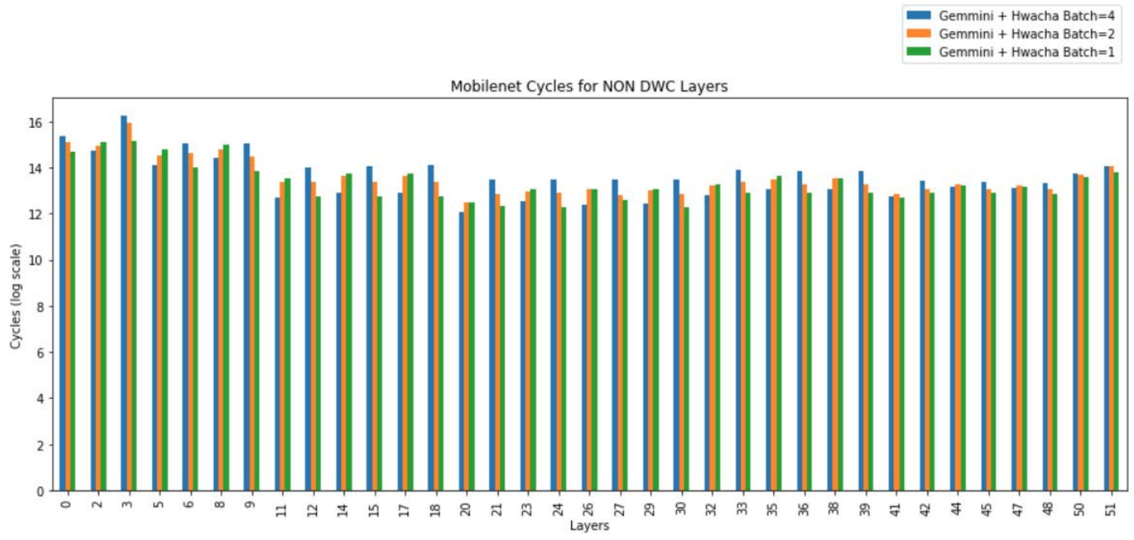


Figure 6.2: Layer by layer performance of MobileNet execution versus batch size of 1,2, and 4 for only non depthwise convolution layers.

For example, looking at layers 5 and 6 in Figure 6.2 we observe that layer 5's cycles steadily increase even though no pipelining takes place since the next layer is a normal convolution. In layer 6 there is a steady trend of decreasing cycles when decreasing batch size due to the pipelining between normal and depthwise convolutions of these layers. In order to understand and isolate this performance pattern, cache traces of these executions were analyzed.

Cache Analysis

Using FireSim's TracerV, we are able to profile our executing workload's cache behavior and compare the workloads average cache miss rate at different time intervals. TracerV samples the hardware counters with a profile interval of 10000 cycles to generate Figure 6.3. In this Figure, it is clear that the finer grained pipelining with the batch size of 1 has a lower average cache miss rate throughout the execution of the workload compared to the batch size of 4, solidifying our previous hypothesis. Compared to the baseline of batch size of 4, there is 101k and 175k less L2 misses in the batch size of 2 and 1 cases, respectively. Batch size of 2 is not included in this figure for clarity purposes but when added averages right in between

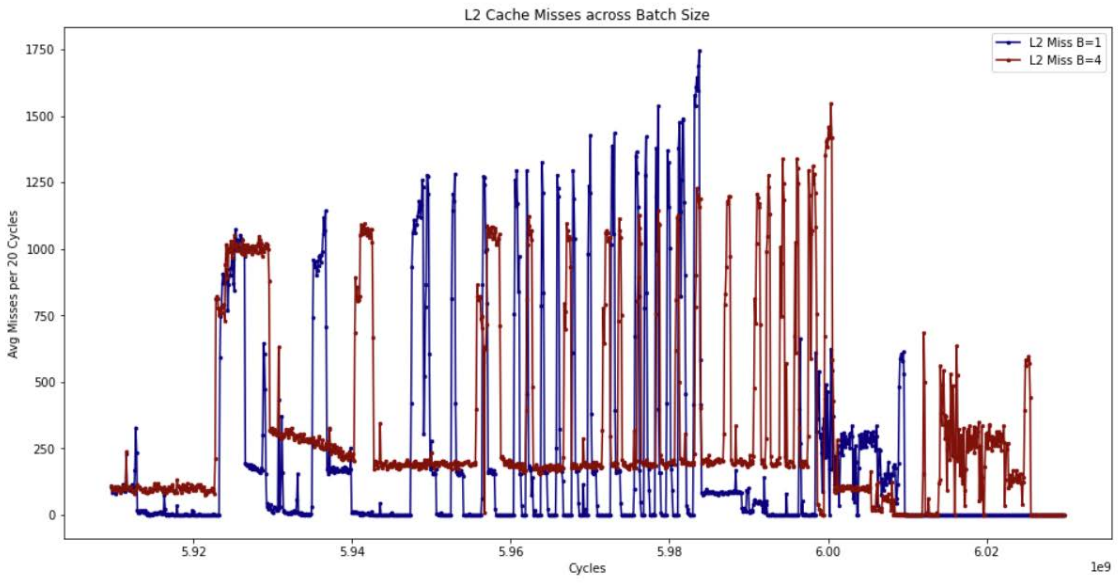


Figure 6.3: L2 cache misses across MobileNet execution with two different batch sizes of 1 and 4.

batch size of 4 and 1 in terms of average L2 cache misses. This proves that the system when having a finer level of pipelining results in better cache locality for both accelerators resulting a less cache misses and better overall performance.

In addition, when combining the layer by layer performance analysis with the cache trace it can be seen in Figure 6.4 how the memory requests correlate to the performance of every layer. The green and red vertical lines correspond to the start and finish of depthwise convolution layers on Hwacha. One overall expected and observed trend is that overall duration of each depthwise convolution layers decrease deeper into the workload because of the network architecture and parameters. Looking closely at the first few depthwise convolutions, it is easily seen that the lower batch size of 1 progresses much more rapidly and maintains a lower average cache miss rate. Interestingly, in between the depthwise convolutions the reverse behavior can be observed where the normal convolutaions have a higher miss rate with the batch size of because of the previous layer only outputting 1 batch at a time. This seems to be the reasoning behind the performance degradation observed in the performance analysis section.

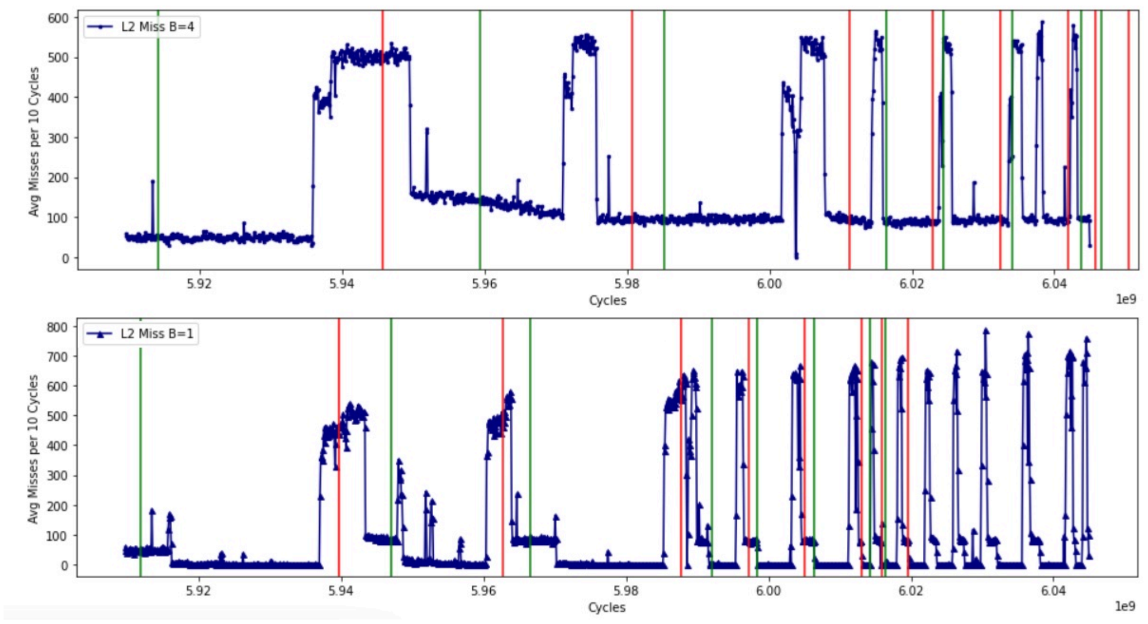


Figure 6.4: The top figure shows the L2 misses for batch size of 4 with green and red lines indicating the start and finish of depthwise convolution layers, respectively. The bottom figure represents L2 misses for batch size of 2 with the same performance lines for depthwise convolutions only.

6.3 Multi Core Heterogenous Architecture

Control Overhead

One of the biggest costs in performance in the previous setup is the overhead in control instructions running on the shared RISC CPU. When looking at a disassembly of the MobileNet inference the CPU is used for coordinating the calls to the normal convolutions via Gemmini and depthwise convolutions via Hwacha. Even though both accelerators have direct access to the L2 cache and can be thought to run ahead individually, it is highly dependent on the kernel and programming model itself.

For example, as mentioned in Chapter 4, with Hwacha's vector fetch programming model, the CPU is still responsible for running the instructions associated with for loops, data pointer manipulation, and setup of all vector registers until the final `vector fetch` call. On the other hand with Gemmini, all of the setup to the covolution kernel call including automatic tiling calculations and buffer checks will also be competing for CPU cycles. This can actually reduce the performance of the overlap between Gemmini and Hwacha because the more CPU dependent the workload is the more stalls will happen because of the CPU being a shared resource.

Configuration

In order to measure the potential for improvement and measure this constraint, a multi core setup is used to run the same workload on. This configuration includes 2 Rocket RISCV CPU cores each with their own Gemmini and Hwacha. Because of Chipyard and FireSim's limitations on producing truly heterogeneous cores with unique accelerators on each core, the CPU is equipped with both Gemmini and Hwacha; however, the software designates one unique core per accelerator and only utilize that accelerator to simulate our desired design.

Using linux pthreads different threads are pinned on to each core, there by selecting which core runs the Hwacha dedicated depthwise convolutions and which core runs the Gemmini normal convolution layers. Specifically, this is done by dynamically detecting which core id the main control thread is first run on and then set affinities of other threads to the opposite core. When working with batch sizes, each pipelined batch is split into a thread of its own to be created and scheduled on the appropriate core (e.g. Hwacha = Core 0, Gemmini = Core 1). This setup tries to see an increased amount of overlap between accelerators executing in parallel which should reduce the overall execution cycles.

Performance Results

Looking at Figure 6.5 it can be seen that with every layer the multi core configuration performs significantly better than the single core configuration for batch sizes of 1 and 2. As expected the speed up in cycles can be attributed to the accelerators overlapping execution and not having to fence and compete over a shared control CPU. In Table 6.2, breaking down the cycles spent in each configuration between normal convolution and depthwise convolution, it can be observed that the speed up of execution actually comes from only the normal convolutions.

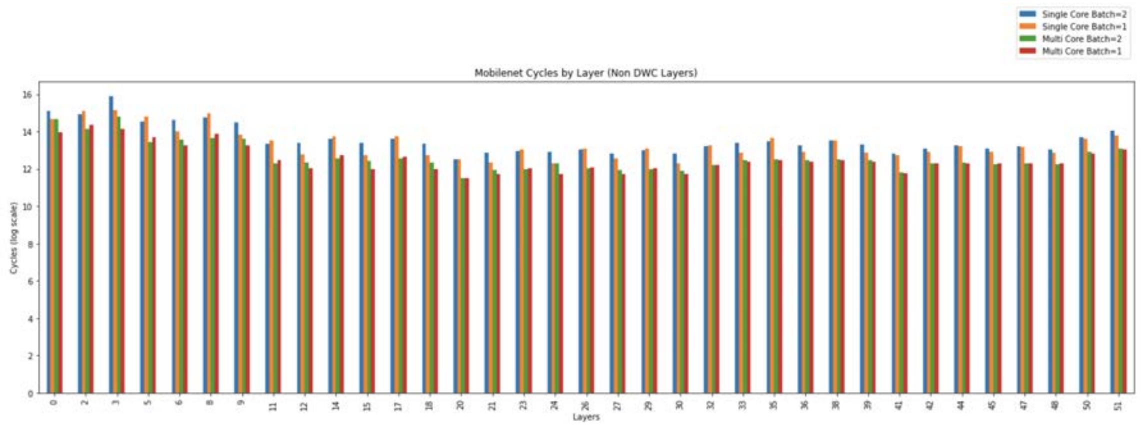


Figure 6.5: Layer by layer performance cycles (log scale) of single core and multi core configurations with batch sizes of 1 and 2.

Cycle Breakdown	**SC Batch = 2**	**MC Batch = 2**	**SC Batch = 1**	**MC Batch = 1**
Normal Conv	4.0×10^7 c	1.6×10^7 c	3.2×10^7 c	1.3×10^7 c
Depthwise Conv	1.2×10^8 c	1.1×10^8 c	1.1×10^8 c	1.2×10^8 c
Total Cycles	1.6×10^8 c	1.3×10^8 c	1.5×10^8 c	1.4×10^8 c

Table 6.2: This table shows the breakdown of cycles spent on normal convolution vs depthwise convolution across the single core vs multi core case, denoted as SC for single core and MC for multi core.

This is due to the fact that in the single core configuration, Hwacha seems to be the blocking accelerator when executing its depthwise convolution. When it is executing on single core, Gemmini is blocked from running ahead and computing the next batch of normal convolutions until the very last few iterations of Hwacha's depthwise convolution kernel, because of the Hwacha kernel running instructions on the CPu, resulting in minimal overlap of accelerator execution. In the multi core case, however, since the Hwacha depthwise kernel is launched as a separate pthread for a different core to execute, the main executing core is free to run ahead and process the next batch of normal convolutions on Gemmini. Interestingly, the difference in performance on the multi core configuration between batch sizes of 1 and 2 is negligible and batch size of 2 performs slightly better. This contradicts the single core configuration as batch size of 1 significantly outperformed the higher batch sizes. This is yet another indication of how different hardware software mapping configurations can lead to conflicting and not intuitive execution performance.

6.4 Conclusion

Future Work and Applications

Future work using different driving DNN workloads and even more heterogenous architectures using several different accelerators could be very interesting. In order to quantify and accurately compare these complex architecture designs, work into defining or adapting a roofline-like model for accelerators could also be a potential project. This model could be used to roughly project the advantages of using different accelerators together for specific workloads.

Moving away from modeling, universal accelerator coordinating hardware or software could be an interesting area to explore based on some of the work done in this research. This work has a few potential solutions unique to systolic and vector accelerators, but developing a generic hardware/software unit that dynamically adapts to differing software workloads would be a huge improvemen in accelerator coordinations and workload execution.

Summary

This work hopes to showcase the potential benefits and complexities that come with the growing accelerator designs on SoCs. By focusing on a few different configurations of specific accelerators and one specific DNN workload, this work is able to achieve up to 3.5x and 2.4x speed up of the execution of MobileNetv2 when compared to some baseline accelerations using just Gemmini. By optimizing and mapping depthwise convolutions to vector accelerators like Hwacha, taking advantage of cache locality through batch and model pipelining, and finally reducing control synchronization overhead this work is able to show various methods of acceleration given a few accelerators. In addition to performance acceleration, this work exposes the many complexities and variables that should be considered when working on software-hardware co-design and mapping. In conclusion, this work starts the exploration in examining the coordination of different accelerators and system impacts and hopes to provide foundation for future studies and analysis.

Bibliography

[1] Alon Amid et al. “Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs”. In: *IEEE Micro* 40.4 (2020), pp. 10–21. DOI: `10.1109/MM.2020.2996616`.

[2] Krste Asanović et al. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Apr. 2016. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html`.

[3] Atul Pandey. *Depth-wise Convolution and Depth-wise Separable Convolution*. [Online; accessed April 5, 2022]. 2018. URL: `https://medium.com/@zurister/depth-wise-convolution-and-depth-wise-separable-convolution-37346565d4ec`.

[4] François Chollet. *Xception: Deep Learning with Depthwise Separable Convolutions*. 2016. DOI: `10.48550/ARXIV.1610.02357`. URL: `https://arxiv.org/abs/1610.02357`.

[5] Henry M Cook, Andrew S Waterman, and Yunsup Lee. *SiFive TileLink Specification*. Tech. rep. `https://www.sifive.com/documentation/tilelink/tilelink-spec/`. SiFive Inc., 2018.

[6] Jeffrey Dean et al. “Large Scale Distributed Deep Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: `https://proceedings.neurips.cc/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf`.

[7] ONNX Runtime developers. *ONNX Runtime*. `https://www.onnxruntime.ai`. Version: x.y.z. 2021.

[8] Krisztián Flautner et al. "Thread Level Parallelism and Interactive Performance of Desktop Applications." In: vol. 28. Dec. 2000, pp. 129–138. DOI: `10.1145/356989.357001`.

[9] Hasan Genc et al. "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration". In: *DAC Design Automation Conference 2021*. 2021, To be published.

[10] N. Holmes. "What Can Computers Do?" In: *Computer* 38.05 (May 1997), p. 11. ISSN: 1558-0814. DOI: `10.1109/MC.1997.10041`.

[11] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. DOI: `10.48550/ARXIV.1704.04861`. URL: `https://arxiv.org/abs/1704.04861`.

[12] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. DOI: `10.48550/ARXIV.1704.04861`. URL: `https://arxiv.org/abs/1704.04861`.

[13] EunJin Jeong et al. "Deep Learning Inference Parallelization on Heterogeneous Processors With TensorRT". In: *IEEE Embedded Systems Letters* 14.1 (2022), pp. 15–18. DOI: `10.1109/LES.2021.3087707`.

[14] Peter H. Jin et al. "How to scale distributed deep learning?" In: *CoRR* abs/1611.04581 (2016). arXiv: `1611.04581`. URL: `http://arxiv.org/abs/1611.04581`.

[15] S. Karandikar et al. "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 2018, pp. 29–42. DOI: `10.1109/ISCA.2018.00014`.

[16] Hyoukjun Kwon et al. "Heterogeneous Dataflow Accelerators for Multi-DNN Workloads". In: *arXiv e-prints*, arXiv:1909.07437 (Sept. 2019), arXiv:1909.07437. arXiv: `1909.07437 [cs.DC]`.

[17] Yunsup Lee. "Decoupled Vector-Fetch Architecture with a Scalarizing Compiler". PhD thesis. EECS Department, University of California, Berkeley, May 2016. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-117.html`.

[18] Atul Pandey. *Depth-wise Convolution and Depth-wise Separable Convolution.* [Online; accessed April 5, 2022]. 2018. URL: `https://medium.com/@zurister/depth-wise-convolution-and-depth-wise-separable-convolution-37346565d4ec`.

[19] Pranav Prakash. "End-to-end Model Inference and Training on Gemmini". MA thesis. EECS Department, University of California, Berkeley, May 2021. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-37.html`.

[20] Mark Sandler et al. "MobileNetV2: Inverted Residuals and Linear Bottlenecks". In: (2018). DOI: `10.48550/ARXIV.1801.04381`. URL: `https://arxiv.org/abs/1801.04381`.

[21] Xuechao Wei et al. "TGPA: Tile-Grained Pipeline Architecture for Low Latency CNN Inference". In: *Proceedings of the International Conference on Computer-Aided Design.* ICCAD '18. San Diego, California: Association for Computing Machinery, 2018. ISBN: 9781450359504. DOI: `10.1145/3240765.3240856`. URL: `https://doi.org/10.1145/3240765.3240856`.

[22] Christian Widerspick, Wolfgang Bauer, and Dietmar Fey. "Latency Measurements for an Emulation Platform on Autonomous Driving Platform NVIDIA Drive PX2". In: *ARCS Workshop 2018; 31th International Conference on Architecture of Computing Systems.* 2018, pp. 1–8.

[23] Lianmin Zheng et al. "Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning". In: *CoRR* abs/2201.12023 (2022). arXiv: `2201.12023`. URL: `https://arxiv.org/abs/2201.12023`.

[24] Haishan Zhu et al. "Kelp: QoS for Accelerators in Machine Learning Platforms". In: *International Symposium on High Performance Computer Architecture.* 2019. URL: `https://ieeexplore.ieee.org/document/8675247`.