Addressing Memory Bottlenecks for Emerging Applications

by

Animesh Jain

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2018

Doctoral Committee:

Assistant Professor Lingjia Tang, Co-Chair
Assistant Professor Jason Mars, Co-Chair
Assistant Professor Hun Seok Kim
Professor Scott Mahlke

Animesh Jain

anijain@umich.edu

ORCID iD: 0000-0001-6777-9168

To my parents, Chandan Jain and Chandresh Kumar Jain

ACKNOWLEDGEMENTS

Looking back at my journey of pursuing the doctoral degree, I believe that I have come a long way from where I started, not just in terms of technical strength but also in terms of becoming a more humble and resilient person. And this would not have been possible without the guidance of my advisors – Lingjia and Jason. Lingjia, your knack for seeing right through the problem and questioning every step from a logical angle taught me the importance of having an absolute clarity in mind. You inculcated a sense of technical rigor and sound reasoning that is going to be extremely helpful going forward. Similarly, Jason, your constant motivation and drive to achieve big in life taught me the determination that is of the utmost importance for becoming a better engineer.

Apart from advisors, I also owe thanks to my collaborators over past five years. Scott, I learned a lot from our discussions that showed me how to technically approach a problem and find surprises early in the stage. Joel, thanks for being an amazing mentor and showing that it is possible to be extremely smart and humble at the same time. Amar and Gena, my internship would not have been such a fun learning experience if not for you. Gilles, I really enjoyed our weekly meetings and thanks for honing my architecture skills over time. Hun Seok, your tough questions have helped me make my thesis more comprehensive and impactful.

I also want to especially thank Mike. Mike, you taught me how to research, how to write papers and how to handle rejections. Thanks for believing in me and spending time on honing my skills. It was my pleasure to work and learn from you. Parker,

I am going to miss the intense technical discussions we had. I learned a lot from those conversations. I also want to thank my lab colleagues - Ram, Chang-Hong, Yunqi, Shih-Chieh, Md, Johann, Yiping and Matt - for an amazing clarity-lab time. Thanks for being there for sharing the frustation of failures as well as happiness of few well-deserved successes.

Obviously, this would not have been possible without the help of my dear friends - Vimal, Siva, Mohit, Tanvi, Jasjit, Vaibhav, Aanjaneya, Akshitha, Biruk and many others. Vimal and Siva, these five years would have been utterly dull if not for you. Thanks for bringing pet Snow, a bundle of joy, into my life and for being there for me always. Mohit, I had an amazing time as your flatmate and I am going to miss the good food. Jasjit, thanks for making those frantic courses and Duderstadt library fun. I will always cherish all these memories.

Lastly, I want to thank my family - Chandan Jain, Chandresh Kumar Jain and Jagrat Jain. Thanks for believing in me and showing me the value of hard work in life. Whatever I am today, it is all because of your teachings and your sacrifices to give me an amazing childhood.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

There has been a recent emergence of applications from the domain of machine learning, data mining, numerical analysis and image processing. These applications are becoming the primary algorithms driving many important user-facing applications and becoming pervasive in our daily lives. Due to their increasing usage in both mobile and datacenter workloads, it is necessary to understand the software and hardware demands of these applications, and design techniques to match their growing needs.

This dissertation studies the performance bottlenecks that arise when we try to improve the performance of these applications on current hardware systems. We observe that most of these applications are data-intensive, i.e., they operate on a large amount of data. Consequently, these applications put significant pressure on the memory. Interestingly, we notice that this pressure is not just limited to one memory structure. Instead, different applications stress different levels of the memory hierarchy. For example, training Deep Neural Networks (DNN), an emerging machine learning approach, is currently limited by the size of the GPU main memory. On the other spectrum, improving DNN inference on CPUs is bottlenecked by Physical Register File (PRF) bandwidth. Concretely, this dissertation tackles four such memory bottlenecks for these emerging applications across the memory hierarchy (off-chip memory, on-chip memory and physical register file), presenting hardware and software techniques to address these bottlenecks and improve the performance of the emerging applications.

For on-chip memory, we present two scenarios where emerging applications per-

form at a sub-optimal performance. First, many applications have a large number of marginal bits that do not contribute to the application accuracy, wasting unnecessary space and transfer costs. We present ACME, an asymmetric compute-memory paradigm, that removes marginal bits from the memory hierarchy while performing the computation in full precision. Second, we tackle the contention in shared caches for these emerging applications that arise in datacenters where multiple applications can share the same cache capacity. We present ShapeShifter, a runtime system that continuously monitors the runtime environment, detects changes in the cache availability and dynamically recompiles the application on the fly to efficiently utilize the cache capacity.

For physical register file, we observe that DNN inference on CPUs is primarily limited by the PRF bandwidth. Increasing the number of compute units in CPU requires increasing the read ports in the PRF. In this case, PRF quickly reaches a point where latency could no longer be met. To solve this problem, we present LEDL, locality extensions for deep learning on CPUs, that entails a rearchitected FMA and PRF design tailored for the heavy data reuse inherent in DNN inference.

Finally, a significant challenge facing both the researchers and industry practitioners is that as the DNNs grow deeper and larger, the DNN training is limited by the size of the GPU main memory, restricting the size of the networks which GPUs can train. To tackle this challenge, we first identify the primary contributors to this heavy memory footprint, finding that the feature maps (intermediate layer outputs) are the heaviest contributors in training as opposed to the weights in inference. Then, we present Gist, a runtime system, that uses three efficient data encoding techniques to reduce the footprint of DNN training.

CHAPTER I

Introduction

The computing industry is witnessing an emergence of applications from the domains of machine learning, numerical analysis, data mining and image processing, that act as key processing applications in both mobile and datacenter workloads [114, 37, 74]. These applications form the core computation of many important workloads that have become critical for the computing industry in the last decade. For example, amongst machine learning algorithms, Deep Neural Networks (DNNs) have recently emerged as a primary computational component in user-facing applications that include analyzing text, decoding speech, recognizing images and searching the web, among others [75, 92, 155, 148, 89, 162, 170, 59, 87, 91]. DNNs are heavily driving the development of intelligent personal assistants (like Apple Siri, Google Now, Microsoft Cortana etc) and autonomous vehicle driving research [148, 89]. Similarly, social media and social networking service providers like Facebook and Twitter, and online retail industries like Amazon, analyse the user data and perform data mining to extract insights, for example, to identify the trends in content sharing or shopping patterns [50, 134]. As we observe this transformation at the frontier of how applications work, it is incumbent upon us, as computer engineers, to recognize the changing nature of applications and to design systems and solutions that can address the needs of these applications.

A common characteristics of these applications is that they work on large amount of data. For example, DNN are trained via iterating over large training dataset. Similary, the data mining to extract insights also query a large dataset collected over long period of time. As a result, one of the main processing bottlenecks among these data-intensive applications is the memory subsystem, where capacity and bandwidth can be critical factors in determining application performance. These applications put stress on different levels of the memory subsystem, on both off-chip and on-chip structures, requiring innovative and alternative computing techniques to match the increasing pressure on the memory subsystem.

This dissertation studies the characteristics of such emerging applications, investigates their implications on different levels of memory hierarchy, identifying the sources of performance bottlenecks, and presenting techniques to address those bottlenecks. In this Chapter, we present the characteristics of these applications, followed by how they pressurize the memory subsystem and brief overview of the techniques developed to mitigate such memory bottlenecks.

1.1 Characteristics of Emerging Applications

The emerging applications have three primary characteristics as follows:

1. **Data Intensive** – The applications process a large amount of data, with memory footprint sometimes reaching in multiple Gigabytes (GBs). For example, training a DNN requires can have working size of tens of GBs easily for upcoming networks [128]. Similarly, Facebook and Twitter have large amount of user data to mine before they can extract any useful insights [50, 134].

2. **Heavy Data Reuse** – Unlike streaming data intensive applications, these applications have large amount of data reuse, i.e., a data element is reused multiple times before it can be discarded by an application. Therefore, majority

of these applications are amenable to cache tiling, a compiler optimization that restructures the application loop structure into tiles (small subsets of the working set that fit into the cache), to take advantage of the heavy data reuse and improve the effectiveness of the cache for the computation [58, 156, 151].

3. **Error Tolerance** – Some of the most prominent emerging applications in datacenter and server workloads are driven by models that consume imprecise and noisy inputs. In addition, the application outputs are also estimates. These factors make the applications highly error tolerant, i.e, the application accuracy is retained or suffers little loss even after removing large number of bits from the input data elements or performing computation with very low precision [136, 16, 51, 78, 95].

1.2 Pressure on Memory Subsystem

As the emerging applications are date-intensive, they put significant stress on the memory subsystem. We observe that the performance bottlenecks in the memory subsystem are not limited to a particular memory structure. Instead, the applications puts stress on different parts of the memory hierarchy for different scenarios. This section briefly discusses these scenarios and the performance bottlenecks across the complete memory stack.

1.2.1 On-chip Memory

Error Tolerant Applications. An inherent characteristic of the emerging applications is the presence of a large number of *marginal* bits - the bits that do not contribute significantly to the application accuracy. These bits, though do not affect application accuracy, waste cache capacity and bandwidth, and energy in transferring these bits between memory structures. However, due to the limited number of

available floating-point datatypes and the dramatic HW/SW stack changes needed to support more flexible floating-point formats, these marginal bits persist, resulting in wastage of cache space, where many more data elements could be stored if the marginal bits were removed. Because of the costs associated with moving and storing the marginal bits, we are not able to effectively utilize cache capacity and bandwidth.

Contention in Shared Caches. As discussed in Section 1.1, many emerging applications have large amount of data reuse and are amenable to a compiler optimization known as *cache tiling*. As a statically parameterized optimization, cache tiling requires that the compiler control both the size and shape of the tiles used in the computation, which is intimately linked to cache size [38, 28, 113]. However, this class of optimization was conceptualized before the multicore era, which has introduced numerous additional dynamic factors that affect application runtime environment, i.e, the cache size available to an application changes at runtime due to different sources of dynamism.

The static assumptions used to aggressively tune the tiling parameters can be easily broken by sources of post-deployment dynamism. Therefore, a pre-deployment best tile can result in sub-optimal performance across different runtime environments. The advent of highly dynamic multicore/multiprocessor environments necessitates the rethinking of how cache tiling should be applied and deployed for these emerging applications in commercial and production contexts.

1.2.2 Physical Register File

A key focus of recent work in our community has been on devising increasingly sophisticated acceleration devices for deep neural network (DNN) computation. Yet, despite the promise of substantial improvements in performance and energy consumption offered by these approaches, due to the cost and complexity of overhauling compute infrastructure and programming model, the questions arises as to what can

be done, if anything, to evolve conventional CPUs to accommodate efficient deep neural network computation.

In this part of research, we focus on the challenging problem of identifying and alleviating the performance bottlenecks for convolution layer computation for conventional CPU platforms. By performing a detailed study of a range of convolution based DNN applications on a modern CPU microarchitecture, we observe that that designing a *physical register file* (PRF) capable of feeding computational units is the primary barrier that prevents the addition of more compute units in the CPU, limiting the performance improvements that can be achieved by CPU on convolution layers. Therefore, we need to craft a solution that can efficiently utilize the PRF bandwidth to improve the DNN performance on CPUs.

1.2.3 Off-chip Memory

Modern deep neural networks (DNNs) training process typically relies on GPUs to train complex hundred-layer deep networks. The DNN training process has large compute and memory requirements and primarily relies on modern GPUs as the compute platform. A significant problem facing both researchers and industry practitioners is that, as the networks get deeper and larger, the available GPU main memory becomes a primary bottleneck, limiting the size of networks it can train and the amount of input data GPUs can process in parallel [128, 32] (For GPU main memory, GDDR5/GDDR5X, the first order concern is bandwidth as many GPU applications are bandwidth-bound. It is hard to get both high bandwidth and high density DRAM-based memory at low cost [100]). Therefore, DNN training process requires innovative solutions to fit larger and deeper models in the GPU main memory, facilitating the machine learning research to train more accurate deeper DNNs.

1.3 Addressing Memory Bottlenecks

The goal of this dissertation is to alleviate the memory bottlenecks of these emerging applications. This section gives a brief introduction of the techniques for addressing the memory bottlenecks discussed in Section 1.2.

1.3.1 ACME - An Asymmetric Compute-Memory Extension

Applications need different floating point datatypes, as evidenced by the ubiquitous support at the both the architecture and language levels for `double`, `float` and (occasionally) `half` types. In this research, we argue that the existing space of 3 options is nowhere near rich enough to capture the needs of application code, leaving many applications using datatypes that are larger than necessary, along with the associated costs in moving and storing those datatypes. Thus, these applications use up unnecessary space in the memory subsystem in terms of cache capacity and memory bandwidth, resulting in substantial application performance degradation and unnecessary energy dissipation.

Supporting arbitrary precision floating-point types in both memory and computation is highly impractical, as it is extremely invasive, requiring major changes to the functional units, pipeline, datapaths and so forth to support using arbitrary precision in the compute substrate. Moreover, prior work in cache compression have been effective at addressing this problem in the integer and fixed-point domains, but cache compression has been shown to achieve negligible compression ratios for floating-point data because of the lack of value-level replication in floating-point data [14, 121, 142, 141, 138]. Thus, this problem remains unaddressed for floating point datatypes.

The goal of this work is to take advantage of this opportunity, reducing the pressure on the memory subsystem by enabling *concise storage* – a storage paradigm where the data elements are stripped of their marginal bits, removing the movement

and storage costs associated with those bits in the memory subsystem. However, several challenges emerge in designing an approach that enables concise storage:

1. **Flexibility** – different applications need different numbers of bits to achieve satisfactory accuracy. Therefore, the design of a concise storage approach needs to have the flexibility to capture the wide spectrum of design points required by different applications and design objectives.

2. **Highly Concise Storage** – the approach should be able to identify as many marginal bits as possible, and avoid storing those bits throughout the memory subsystem while still delivering high-quality computational results.

3. **All Memory Levels** – techniques focused on a particular level of cache, or those focused solely on DRAM, only alleviate pressure on part of the memory subsystem. A better solution should reduce the burden of marginal bits throughout all levels of memory.

4. **Modular** – the approach should reuse as much existing compiler, architectural and micro-architectural infrastructure so that it can be easily built into those infrastructures. It should also be backward compatible and should have minimal impact on exact applications.

To address these challenges and enable concise storage throughout the memory hierarchy, this work motivates and describes ACME, an *asymmetric compute-memory extension* for conventional architectures. In ACME, data can be treated *asymmetrically*; computation is done on conventional 32-bit IEEE 754 single precision [120] values – while data is stripped of its marginal bits before being used in the memory hierarchy. ACME includes a simple ISA extension that can be leveraged by the programmer and compiler, adding two new instruction classes to the ISA to operate on concise data – *load-concise* and *store-concise* – to perform conversions between concise and single precision format via three small additional micro-architectural units.

The asymmetric approach significantly increases the ability to achieve concise storage with small precision loss.

1.3.2 ShapeShifter - Continuous Cache Tiling

As discussed in Section 1.2, many emerging applications have heavy data reuse, making them suitable for the compiler optimization known as cache tiling. However, the static assumptions used to aggressively tune the tiling parameters can be easily broken by sources of post-deployment dynamism. We observe substantial performance loss in the presence of different sources of dynamism due to the mismatch in availability of architectural resources compiler assumed at compile time versus actual availability at runtime. The focus of this research is to develop a dynamic end-to-end solution that is capable of detecting such opportunities and retiling the application tiles suited to its current runtime environment.

To design a cache tiling solution that can encompass these numerous factors, two main challenges emerge:

(i) *the solution should be* ***accurate****, generating tiles that are customized to take full advantage of cache and delivering significant performance benefits, and*

(ii) *monitoring application code for tiling opportunities and rewriting application code to introduce new tiles must be* ***low-overhead****, such that the overhead of those activities does not outweigh the benefits of the improved tiles.*

A key insight of this work is to use a rapidly and dynamically constructed environment- and application-specific *black box* model for predicting the performance of a host of tiling options within the immediate environment. This paper introduces **continuous shape shifting** with *ShapeShifter*, an end-to-end dynamic compilation infrastructure that enables continuous shape shifting and aggressively rewrites running applications

in response to runtime dynamism. ShapeShifter uses a lightweight monitoring infrastructure to examine the running applications and the runtime environment to look for opportunities to tile and re-tile the applications in response to changes in the runtime environment. Upon identifying a suitable tile shape based on the dynamically constructed model, ShapeShifter rewrites and re-tiles the application leveraging a low-overhead dynamic compilation capability to divert execution into the aggressively tiled code with near-zero overhead.

1.3.3 LEDL - Locality Extensions for Deep Learning

This work focuses on the challenging problem of identifying and alleviating the performance bottlenecks for convolution layer computation for conventional CPU platforms. Looking at modern CPU offerings, it is clear that they offer substantially fewer raw floating point operations per second (FLOPS) than their GPU counterparts. However, CPUs are an indispensable part of the design of any system, meaning they are a well understood part of conventional system design practices while offering the benefit of a seamless, familiar programming model and software stack. Moreover, CPU designs have a long history of incorporating hardware and ISA support for specialized domain-specific operations, evidenced by the near-universal support for cryptography, virtualization, security and multimedia operations in modern CPU offerings [10, 4, 11, 5, 6]. Thus, alongside designing dual-device acceleration platforms, it remains an important objective to design CPU hardware that can perform all the non-accelerable tasks for which CPUs are essential while also serving as an energy-efficient fabric for convolution layer computation.

Unfortunately, despite the large body of work in our community on accelerating DNNs [34, 117, 31, 105, 66, 128, 13, 127, 68, 97], there is little understanding in the literature of the interplay among the factors involved in improving CPU performance on convolution layers. Simply increasing raw FLOPS by continuing down the path of

scaling vector widths, such as in the progression from SSE to AVX to AVX2 among x86 platforms [109, 53], is unlikely to continue for two reasons. First, the AVX2 vector width of 512 bits spans a full cache line, and thus longer vectors would necessarily touch multiple cache lines per vector register load, introducing significant performance penalties or substantial microarchitectural workarounds. Second, leveraging larger vector widths puts the onus on programmers and compilers to find additional sources of SIMD parallelism, an extremely difficult task even for current vector widths that remains an active, open area of research in the compiler community [17, 79, 116]. Thus, it is clear that improving the computational capability of CPUs for convolution layers requires an alternative approach, yet it remains unclear what that approach is.

In this research, we perform a detailed characterization of the issues involved in improving CPU performance for convolution layers. We find first that scaling the read bandwidth of the physical register file (PRF) is one of the key constraints needed to deliver additional data to increasingly capable compute units. Second, we find that harnessing increasingly capable compute units requires crafting a solution that spans both hardware and software to take full advantage of the data reuse present in the core of the CNN computation. Building on this insight, we design Locality Extensions for Deep Learning (LEDL). LEDL is a technique that spans both hardware and software, consisting of a novel set of microarchitectural and ISA extensions to increase the computational capabilities of modern CPUs for CNNs. We present the design in detail, which in hardware includes a handful of architecturally visible remote registers that reside within the VFMA units in the CPU and a set of inter-VFMA links that allow data to be passed between units directly. In software, LEDL's automatic code generator, ACG, is carefully designed to generate code that is robust to different microarchitectural implementations while taking full advantage of the reuse opportunities exhibited by convolution layer computation and aggressive prefetching mechanisms within modern CPUs.

1.3.4 *Gist* - Efficient Data Encoding for DNN training

The availability of large datasets and powerful computing resources has enabled a new breed of deep neural networks (DNNs) to solve hitherto hard problems such as image classification, translation, and speech processing [75, 77, 88, 90, 160]. These DNNs are *trained* by repeatedly iterating over datasets. This DNN training process has large compute and memory requirements and primarily relies on modern GPUs as the compute platform. Unfortunately, as DNN models are getting larger and deeper, the size of available GPU main memory quickly becomes the primary bottleneck.

Many researchers have recognized this shortcoming and proposed approaches to reduce the memory footprint of DNN training to train larger and deeper DNNs on GPUs. However, prior approaches are not able to simultaneously achieve all of the following three desirable properties: (i) *provide high memory footprint reduction, (ii) low performance overhead, and (iii) minimal effect on training accuracy.* Most prior works propose efficient techniques to reduce the memory footprint in DNN *inference* with an emphasis on reducing model size (also referred to as weights) [67, 71, 98, 72, 69, 66]. However for DNN training, weights are only a small fraction of total memory footprint. In training, intermediate computed values (usually called *feature maps*) need to be stored/stashed in the forward training pass so that they can be used later in the backward pass of training. These feature maps are the primary contributor to the significant increase in memory footprint in DNN training compared to inference. This important factor renders prior efforts, that target weights for memory footprint reduction, ineffective for training. State-of-the-art memory footprint reduction approaches for training copy data structures back and forth between CPU and GPU memory but pay a performance cost in doing so [128]. Finally, approaches that explore lower precision computations for DNN training, primarily in the context of ASICs and FPGAs, either do not target feature maps (and hence are unable to achieve high memory footprint reduction) or, when used aggressively, result in reduced training

accuracy [44, 63, 39].

The key insight of this work is in acknowledging that a feature map typically has two uses in the computation timeline and that these uses are spread far apart temporally. Its first use is in the forward pass and second is much later in the backward pass. Despite these uses being spread far apart, the feature map is still stashed in single precision format (32-bits) when they are unused between these accesses. We find that we can store the feature map data with efficient encodings that result in a much smaller footprint between the two temporal uses. Furthermore, we propose that if we take layer types and interactions into account, we can enable highly efficient *layer-specific encodings* – these opportunities are missed if we limit ourselves to a layer-agnostic view. Using these key insights, we plan to design two layer-specific lossless encodings and one lossy encoding that are *fast, efficient in reducing memory footprint, and have minimal effect on training accuracy.*

Our first lossless encoding, Binarize, specifically targets ReLU layers followed by a pooling layer. We observe that the ReLU output, that has to be stashed for its backward pass, can be encoded using just 1-bit values because ReLU's backward pass calculation only needs to know whether ReLU output is zero or non-zero, leading to 32× compression for these ReLU outputs. Our second lossless encoding, Sparse Storage and Dense Compute (SSDC), specifically targets ReLU followed by convolution layer. We observe that ReLU outputs have high sparsity that can be exploited to reduce memory footprint of stashed ReLU outputs. SSDC facilitates storage in memory-space efficient sparse format but performs computation in dense format, retaining the performance benefits of highly optimized cuDNN dense computation, while achieving high reduction in memory footprint. Finally, in the lossy domain, our key insight of representing the stashed feature maps in smaller format *only* between the two temporal uses lets us be very aggressive with precision reduction without any loss in accuracy. Our third lossy encoding based on this insight, Delayed Precision

Reduction (DPR), delays precision reduction to the point where values are not longer needed in the forward pass, leading to significant bit savings (as small as 8 bits).

Utilizing all these encodings, we present *Gist*, a runtime system, that specifically targets feature maps to reduce the training memory footprint. It will perform a static analysis on the DNN execution graph, identifies the applicable encodings, and creates a new execution graph with relevant encode and decode functions inserted. *Gist* also performs a static liveness analysis on the affected feature maps and newly generated encoded representations to assist the DNN framework's memory allocator, CNTK [145] in our case, to achieve an efficient memory allocation strategy.

1.4 Contributions

The specific contributions of this dissertation are as follows.

1. **Asymmetric Compute Memory Extension** – This dissertation presents ACME, a novel asymmetric compute memory paradigm, where the marginal bits are removed from the memory subsystem while the computation still happens in full precision. We present the hardware and software techniques to realize this asymmetric architecture, enabling us to efficiently utilize the cache capacity and bandwidth.

2. **Continuous Loop Tiling** – This dissertation presents ShapeShifter, a runtime system that continuously monitors, detects if an application runtime environment has changed and requires a new tiling strategy, and dynamically retiles the application code tailored for the runtime environment. This enables cache tiling, a hitherto static compiler optimization, to adapt to the current datacenters having high degree of dynamism in post multi-core era.

3. **Rearchitected FMA and PRF design for DNN inference on CPUs** – This dissertation tackles the challenging problem of improving DNN interference

on CPUs, finding that PRF is the primary bottleneck in increasing the raw compute capability of CPUs. We present LEDL, a rearchitected FMA design that exploits the heavy data reuse inherent in DNN computations within and across the FMA units, along with an Automatic Code Generator (ACG) that generates the code tailored to the number of compute units available in the processor.

4. **Efficient Encodings to reduce DNN training memory footprint** – A significant challenge facing the researchers and industry practitioners is that, as the networks grow deeper and larger, the available GPU main memory becomes a primary bottleneck, limiting the size of DNNs it can train. This dissertation investigates the data structures that contribute heavily to this memory footprint and presents *Gist*, a runtime system that uses three efficient data encodings to significantly reduce the memory footprint of the primary memory consumers in DNN training.

CHAPTER II

Background and Related Work

In this Chapter, we survey the related literature and provide the background to the topics covered in this dissertation. This includes the current state-of-the-art in removing marginal bits from the memory, dynamically retiling the applications for the applications having heavy data reuse, hardware and software techniques for improving DNN inference performance and reducing memory footprint of DNN training.

2.1 Removing Marginal Bits

One common way to reduce application cache footprint is using cache compression. The majority of cache compression techniques strives to reduce value replication in the memory subsystem [14, 142, 141, 121]. However, these cache compression techniques are limited to integer benchmarks. Prior work shows that floating point data do not show redundancy to the same degree as integer benchmarks [14, 138]. Our work, focusing on floating-point data, is orthogonal and can be applied in conjunction with cache compression.

There have been significant advances in using emerging memory technology as approximate storage to trade-off storage accuracy for performance and energy savings [177, 125, 99, 136]. Our approach is different from these works because we focus on concisely representing the data elements in traditionally designed memory. Dopple-

ganger maps approximately similar cache lines to one physical cache line, resulting in increased effective cache size [138]. Load-value approximation approximates the value of a load on a cache miss [139].

Others have proposed techniques to reduce DRAM energy consumption by adjusting DRAM refresh interval [108, 126, 159]. These techniques are specific to DRAM and focus on energy savings. These techniques divide DRAM into critical and non-critical partitions and reduce the refresh interval of non-critical portions to get energy savings. The refresh interval of critical portion is kept unchanged but the non-critical refresh interval can be lowered in order to get energy savings. Our work achieves concise storage throughout the memory hierarchy and reduces DRAM accesses by fitting more elements in the caches.

Recently, research in the field of machine learning has shown that several neural networks require very few bits for storing their input parameters [40, 64, 70]. However, these works are targeted towards deep learning systems. Our work is generic and presents an end-to-end system, tackling challenges that come when converting these memory savings into performance improvements.

There has been research to tune the precision level of an application to tradeoff performance with accuracy. Precimonious [131] and gappa++ [107] provide software precision tuning algorithms to find suitable data types for an application. However, these works are limited to `float` and `double` data types. There has been extensive research in the programming languages field to support approximate computing [137, 24, 26, 135, 16, 51]. Our works uses programmer annotations to identify approximation friendly variables as is done in prior work [135, 52, 169, 168, 26]

2.2 Dynamically Retiling Applications

Prior research in finding the best tile size can be divided into two categories: static techniques that develop a detailed analytic model for a set of host environments and

predict a tile [28, 38, 93, 143], and dynamic techniques that use a model to prune the search space, execute a subset of tiles and choose the one with the least execution time [167, 163, 30, 47].

Static Techniques. This class of methods take an approach of developing detailed white box analytic models for the applications and runtime environments. TSS [113] studies how tiling interacts with several level of caches. Defensive tiling [18] considers tiling strategy in presence of last-level cache interference, with the goal of reducing the number of inclusion victim misses. Coleman and McKinley [38] develop a cache model to find the largest tile that suffers from minimum self-interference misses. These models can deliver useful insights about how applications interact with the runtime environment. However, it is difficult and sometimes intractable for the white box approaches to accurately model the complex set of factors that impact the choice of tiling strategy. ShapeShifter differs from these techniques as it creates a model on-the-fly.

Yuki et al. [172] discuss the limitations of the white box approaches. They use a neural network to statically predict a tile for an application. However, it is a completely static technique unable to adapt to changing runtime environment. In addition, they limit the search to only square tiles to reduce the large training time, leaving a significant performance opportunity on the table.

Dynamic Techniques. Reactive tiling [151] is a combined static and dynamic technique that compiles an application with a fixed set of tiling parameters and inserts mechanisms in the code to switch between this set of tiles at runtime. Reactive tiling focuses only on the scenarios where the cache is resized during the application execution as opposed to ShapeShifter that accounts for a wide range of sources of dynamism. Some prior works mitigate the complexity of searching by resorting to only square tiles (i.e., all tiling parameters must be equal) [163, 172]. Such limitations are fundamental as they exclude valuable tiling configuration possibilities. We observed

that performance difference between the best rectangle tile was 1.11× (up to 1.5×) faster than best square for our test applications. The ATLAS library generator [163] executes a wide range of tiles on the target machine and chooses the one with the best performance. However, the optimized kernels cannot react to sources of dynamism.

2.3 Hardware and Software Techniques for DNN Inference

Accelerators. A significant amount of research has been done on DNN in past few years [117, 34, 31, 105, 66, 175, 84]. Spatial architectures, having distributed compute and memory, have been gaining attention as deep learning accelerators. The Catapult CNN accelerator for FPGAs [117], TPU [84] and Eyeriss [34] are examples of spatial architectures that use or can be configured as systolic arrays to transfer partial sums between the distributed compute elements. DianNao and DaDianNao research present DNN accelerators, focusing on minimizing off-chip as well on-chip data accesses [31, 33].

Our technique inspired from exploiting the inter-unit reuse of data, have some similarities with the systolic dataflow model presented in the spatial architecture DNN research. However, there are substantial differences between the amount of compute and memory in CPUs as compared to spatial architectures. Distributed compute and memory helps spatial architectures divide up the work in a coarse-grained manner where several PEs can compute partial sums for a small subset of inputs in parallel and then transfer these partial sums between the compute elements. This is not possible in CPUs, because there is a centralized PRF and the amount of compute is also limited, preventing coarse-grained division of work. As a result, FMA latency becomes a critical constraint while passing partial sums between the VFMA units on CPUs, resulting in low performance for dataflows employing partial sum transfers. Therefore, instead of passing partial sums, we transfer the input elements between the compute units while maximizing the partial sum usage at PRF.

In addition, there has been research in designing general purpose hardware that can efficiently take advantage of SIMD execution units – Libra [119] and Dyser [56]. These designs, though showing higher flexibility, do not exploit the heavy data reuse inherent in convolution algorithms. In addition, LEDL is targeted for CPUs, aiming at low hardware intrusion techniques that can be implemented in CPUs quickly.

Weight Pruning and Precision Reduction. Convolution layers show high opportunity of pruning weights, substantially reducing the data footprint and the costs associated with the data movements. Research efforts have focused on either achieving this pruning or designing hardware solutions taking advantage of the pruned datasets [34, 13, 66, 68]. In addition, many DNN applications do not require 32 bits of precision, further reducing the weight storage requirements. DNNs retain their accuracy even after converting the data format to 8/16-bit fixed point format [127, 105, 83]. Many insights from these efforts are orthogonal to our work, resulting in additional speedups when applied in conjunction with our work.

Software. On software-focused efforts, there have been an increasing number of efforts in writing aggressively hand-tuned codes for hardware, like Intel MKL and NNPACK for CPUs, Nvidia CuDNN and Nervana Neon for GPUs [1, 49, 35, 2], extracting every last ounce of compute packed on the machines. In addition, there have been efforts to reduce the arithmetic complexity of convolution algorithms [97, 157]. Our code generator stands in a similar category of software efforts with focus on automatic code generation for a given number of CPU VFMA units, instead of hand-tuning it for one hardware design point.

2.4 Reducing Memory Footprint for DNN Training

Our work presents a systematic analysis of breakdown of total memory footprint across different data structures in DNN training, showing that stashed feature maps

and immediately consumed data structures are the major contributors in modern DNN frameworks (as opposed to weights in DNN inference). We present layer-specific lossless encodings, targeting different categories of stashed feature maps, which to our knowledge has not been proposed before. Our lossy encoding is specifically designed for DNN training and stands in stark contrast with the prior body of similar work on DNN inference. Our work presents a unique way of applying precision reduction in which the data in the forward pass is kept in full FP32 format, while only the data that is stashed for the backward pass is represented with fewer bits, resulting in more aggressive bit savings which is unseen in the previous work.

Generic Approaches. vDNN transfers the data between CPU DRAM and GPU memory using smart prefetching analysis [128]. vDNN enables fitting very large networks in the GPU memory, but at the expense of (1) performance cost (11% on average, and up to 25% for Inception, for $vDNN_{all}$ configuration), and (2) energy cost of using PCIe and GPU DRAM bus constantly for the data transfer, and (3) using PCIe, which is a shared critical resource in a distributed training [41], potentially causing performance issues in distributed setting. CDMA, designed on top of vDNN, leverages sparsity to compress the data sent between CPU and GPU [129]. [32, 62] presents memory sharing and inplace optimizations that are implemented in MxNet framework. It also proposes layer re-computation to trade off large memory space with re-computing fast DNN layers. This work is orthogonal and can achieve additional speedup with our encodings.

Encodings. Lossy encodings have been studied rigorously in the domain of DNN inference. These works apply network pruning, quantization, huffman encoding and precision reduction to reduce the model size (*weights*) [67, 85, 71, 98, 72, 69]. Many HW accelerators have been designed employing limited precision and leveraging sparsity to reduce computational and memory requirements [66, 31, 34, 118, 86, 147, 46, 127, 158, 13]. However, these techniques do not apply directly for training as weights

change frequently during the training process, and weights are not a major contributor to total memory footprint.

Most of the other works for DNN training have looked into the reducing precision requirements for *computation*. These works do not focus on reducing memory footprint and, thus, do not optimize memory for stashed feature maps. For example, BUCKWILD! breaks down memory footprint into four categories (DMGC in the paper), but ignore stashed feature maps, as it does not play significant role in computational precision study [44]. Similarly, [63, 39, 65] show that 16-bits dynamic fixed point computation is enough for training small DNNs on CIFAR-10 and do not focus on primary contributor to memory footprint. We share an observation with this work that uniform precision reduction results in severe accuracy losses. These works keep a shadow copy of weights in full precision, which is updated at the end of each minibatch and then quantized for next minibatch, to keep accuracy in check.

CHAPTER III

Concise Loads and Stores: The Case for an Asymmetric Compute-Memory Architecture

One of the main processing bottlenecks among data-intensive applications is the memory subsystem, where capacity and bandwidth can be critical factors in determining application performance. Prior work has made this observation, resulting in a class of techniques focused on the problem of identifying and building systems that take advantage of replication and redundancy across different data elements in the memory hierarchy [14, 121, 142, 141, 138].

This work takes a new approach to addressing the problem, focusing on *marginal bits* – bits within the data representation that add little extra information among elements in a data structure while consuming a significant fraction of the memory and cache resources. Motivating this work is the observation that a number of applications (1) are tolerant to the removal of marginal bits, where the accuracy of results is minimally impacted and (2) stand to benefit significantly in performance and energy when the burden of storing and moving those additional bits is removed.

This opportunity is illustrated in Figure 3.1, which shows the output accuracy of Kmeans across a spectrum of different input bit counts. Figure 3.1(a) uses the "precise" 32-bit single-precision format, while (b), (c) and (d) use input elements represented in 16, 10, and 8 bits, respectively. Note that these experiments simply

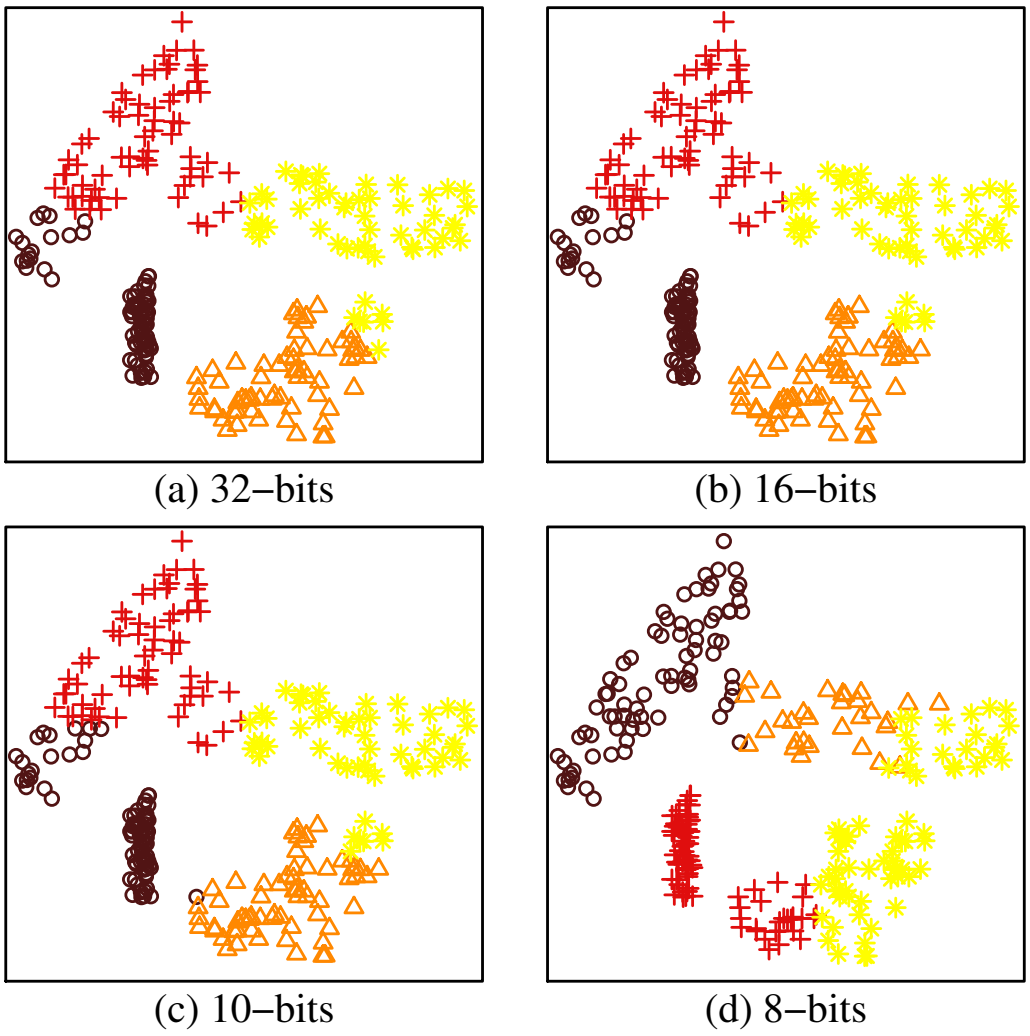


Figure 3.1: Kmeans clustering output when applying a range of different storage formats. Similar accuracy for 32 (precise), 16 and 10 bits but poor accuracy for 8 bits

drop input bits; the computation still happens at 32-bit single precision. We observe that using 16 or 10 bits changes the cluster membership of a few points, but the results remain almost indistinguishable from the exact results. However, further reducing the input representation to 8 bits results in incorrect cluster membership for the majority of points. We have observed a similar trend in numerous applications, where *dropping marginal bits from the input has little impact on application accuracy but can significantly improve performance*. Our further investigation, as we will show in Section 3.1, provided two more insights – 1) storing data with fewer bits while performing computation at full precision removes more marginal bits compared to the approach where fewer bits are used for both memory and compute, and 2) the remaining bits after removing the marginal bits often do not fit neatly into `double`, `float` or `half`, or any other representation that is a multiple of 8.

To enable concise storage throughout the memory hierarchy, this work motivates and describes ACME, an *asymmetric compute-memory extension* for conventional architectures. In ACME, data can be treated *asymmetrically*; computation is done on conventional 32-bit IEEE 754 single precision [120] values – while data is stripped of its marginal bits before being used in the memory hierarchy. ACME includes a simple ISA extension that can be leveraged by the programmer and compiler, adding two new instruction classes to the ISA to operate on concise data – *load-concise* and *store-concise* – to perform conversions between concise and single precision format via three small additional micro-architectural units. The asymmetric approach significantly increases the ability to achieve concise storage with small precision loss.

This asymmetric approach is flexible, allowing the application programmer and compiler make clear choices as to how much space is used to store data. The approach results in highly concise storage, significantly outperforming prior approaches based on leveraging redundancy across data elements or cache lines. The approach impacts all memory levels, converting between concise and full-precision formats at

the boundary of the memory hierarchy, ensuring that data is stored concisely throughout the hierarchy. Finally, the approach is backward compatible and reuses existing hardware, adding three small additional micro-architectural units on top of existing designs to perform address generation for concise data accesses and to perform format conversion between concise and native data formats.

We perform an evaluation of ACME on 10 applications covering a range of data-intensive and compute-intensive applications. We find that the approach is able to achieve speedups that average 1.3× (up to 1.8×) while losing a maximum of 1% end-to-end application accuracy.

3.1 Motivation

In this section, we discuss the limitations of prior work in achieving highly concise storage, and make the case for an asymmetric compute and storage technique.

3.1.1 Limitations of Prior Work

Lossless cache compression techniques [14, 121, 142, 141] focus on removing redundant bits by reducing the incidence of replicated values in last-level caches (LLCs). These approaches are designed to work with fixed-point and integer programs. However, cache compression has been shown to achieve negligible compression ratios for floating-point data because floating-point data lacks the value-level replication that is often found in integer and fixed-point data [14, 138]. Others have explored extending the definition of replication to include softer definitions of replication, treating LLC lines of similar floating point data as replicas [138]. These techniques achieve better compression for floating-point values than lossless compression techniques. However, as we show later in Section 5.4, these softer definitions of replication still leave large numbers of marginal bits in cache. Moreover, the narrow focus of prior work on last-level cache only partially addresses this problem, leaving all data in place in private

caches and DRAM.

In addition, different applications need different numbers of bits to achieve satisfactory accuracy. Therefore, the design of a concise storage approach needs to have the flexibility to capture the wide spectrum of design points required by different applications and design objectives. Current architectural designs that include support for `double`, `float` and (occasionally) `half` precision floating-point configurations are of limited applicability, as they do not capture a rich enough range of options and leave a significant opportunity on the table. Moreover, recent prior work focused on building approximate storage structures also does not provide sufficient flexibility because its approximation settings are built into the hardware at design time [138].

To address the limitations of prior techniques, our approach uses custom-precision floating-point formats. In our approach, each number still has sign, exponent and mantissa fields, however the number of mantissa and exponent bits are not fixed. This makes our approach highly flexible, providing a rich spectrum of design points with different numbers of mantissa and exponent bits to choose from, resulting in a highly concise storage. Our approach is fundamentally different from previous works [138, 14, 121, 142, 141] as it identifies marginal bits by carefully characterizing the impact of bits in the data elements, while the previous works apply softer definitions of data replication across LLC lines, missing the opportunity to remove all the marginal bits.

3.1.2 The Problem with Asymmetry

One might posit that concise storage could be achieved via a system using custom precision formats that is *symmetric* in compute and memory, using a concise format in both memory and compute. However, there are two reasons that make such an approach impractical.

First, it would be extremely invasive and hardware-intensive, requiring major changes to the functional units, pipeline, datapaths and so forth to support using

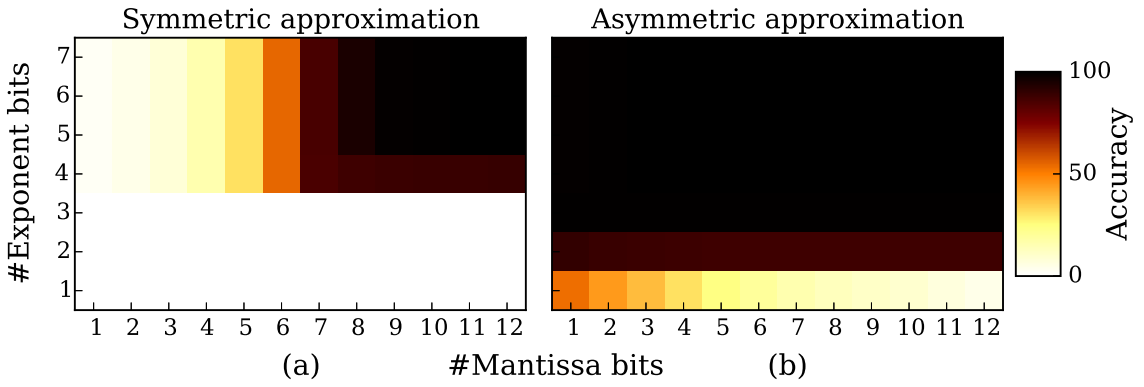


Figure 3.2: Accuracy comparison between (a) symmetric approximation and (b) asymmetric approximation for Kmeans, showing that asymmetry achieves same accuracy with significantly fewer bits as compared to symmetric approach

concise data formats throughout.

Second, we have observed in our experiments that a symmetric approach tends to lose accuracy very quickly as the number of bits in the data format are reduced. This is illustrated in Figure 4.2(a) and (b), which show the result accuracy of running Kmeans using symmetric and asymmetric approaches, respectively, for a range of different exponent and mantissa lengths. The asymmetric approach stores data concisely throughout the memory hierarchy while performing computation at full precision. Each plot shows the accuracy of a range of different formats, where darker colors indicate higher accuracy results. The key observation is that the asymmetric approach can achieve a particular level of accuracy with far fewer bits. Value saturation causes steep dropoffs in accuracy when reducing the number of bits in the symmetric approach (e.g., going from 4 to 3 exponent bits in Figure 2a). Such reductions in the number of bits reduce the range of values supported by the functional units, frequently leading to saturated intermediate and output values and highly inaccurate computation. For example, the symmetric approach requires 15 bits to achieve 99% accuracy, while the asymmetric approach requires just 5. This trend holds true across applications – on average across 10 test applications, we find that the symmetric approach requires 1.7× as many bits as the asymmetric approach to attain 99%

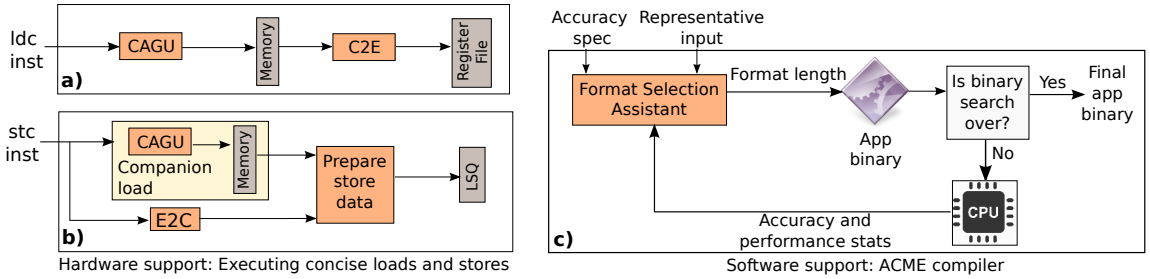


Figure 3.3: ACME system architecture; a) ACME compiler finds a suitable precision for the application and produces concise loads and stores for the annotated variables. b) and c) show execution of these concise loads and stores in hardware.

accuracy.

3.1.3 Bridging the Format Divide

An asymmetric approach has significant benefits over a symmetric approach in terms of hardware simplicity and accuracy, but there remains one main difficulty to solve to enable the asymmetric approach – *bridging the format divide* by converting between precise and concise data formats at the boundary of the memory hierarchy.

An obvious way to perform these conversions to extract precisely formatted data from concise data is to leverage existing software mechanisms such as shifts, masks and other operations. Such an approach would work by loading concise data using conventional memory operations, then convert and distribute it (potentially across multiple registers) by shifting, masking and other bit-level operations. The main difficulty making software conversion approach impractical is that many such operations may be needed per memory operation, introducing significant amounts of additional processing overhead to support concise storage.

While such an approach may reduce capacity and bandwidth requirements in the memory hierarchy, through experimentation (not shown here) we have observed that it significantly undermines the ability of the approach to improve application performance on net, often introducing non-negligible slowdowns due to the cost of converting data every time it is loaded. This suggests that the key to enabling an

effective approach to leveraging an asymmetric compute-memory approach lies in efficiently bridging the format divide.

3.2 Overview of ACME

ACME is designed to address these problems. ACME is based on an asymmetric compute-memory architecture; the data is stored concisely in memory while computation happens on full precision. ACME reduces pressure on the memory subsystem exploiting marginal bits to reduce the cost of storing and moving data.

3.2.1 Challenges

However, there are several challenges in converting these savings in memory storage and bandwidth to performance improvements.

Quick Format Conversion. ACME is based on an asymmetric compute and storage paradigm, resulting in a format divide between compute and memory. Therefore, each concise load requires conversion from the concise format to the single precision format. Similarly, each concise store requires conversion from the single precision format to the concise format to bridge this format divide. These conversions must be fast to extract maximum performance benefit from the concise storage.

Bit-level Interactions in Byte-addressable Memory. Achieving highly concise storage requires storing values of arbitrary length in the memory. This gives rise to situations in which the concise data element might not start at a byte boundary. Since conventional memory subsystem is byte-addressable, ACME needs to support certain bit-level interactions in a byte-addressable memory environment.

Choosing Precision. Different applications have varying accuracy requirements, and thus varying format requirements. Finding a suitable precision requires navigating through a non-trivial search space (`23` mantissa * `8` exponent = `184` for each

variable). Therefore, ACME requires quickly finding the right level of precision for the application.

3.2.2 Key Components

We introduce these components to address the challenges outlined earlier.

Fast Conversion Units. ACME introduces two small additional units, *Concise to Exact (C2E) and Exact to Concise (E2C)*, to bridge the format divide between compute and storage. These units perform format conversions in a single cycle. The C2E unit converts the concise data element into single precision format before writing it into the register file. Similarly, the E2C unit converts the data element format from single precision format to concise format before sending it to memory.

Concise Address Generation Unit. ACME uses a Concise Address Generation Unit (CAGU) to calculate the memory address of concise data elements. Our approach keeps the memory byte-addressable. CAGU generates a byte-level memory address that is closest preceding to the concerned concise data element. It works in concert with the E2C and C2E to access the memory response at a bit-level granularity.

Format Selection Assistant. ACME employs a Format Selection Assistant (FSA) to find an appropriate format for an application. For a specified accuracy target, ACME performs a binary search over the number of exponent and mantissa bits to quickly identify a suitable precision for each approximated variable.

ISA Support. We propose two ISA extensions in the form of *load-concise* (`ldc`) and *store-concise* (`stc`) instructions. These instructions support arbitrary length storage in the memory hierarchy, leveraging the CAGU, E2C and C2E units to realize the asymmetric compute and storage architecture.

3.3 Design and Implementation

ACME is an end-to-end system that stores data concisely by removing marginal bits while performing computations at full precision, an approach that improves performance of memory-intensive applications by increasing effective cache size and effective memory bandwidth. In this section, we describe the details of the ACME system architecture.

3.3.1 System Architecture

Figure 3.3 illustrates a high level overview of ACME. illustrating the hardware support (left) and the software support (right).

Hardware Support. Concise loads and stores are supported in the hardware via the CAGU, E2C and C2E units. For the `ldc` instruction, as shown in Figure 3.3a, the processor sends a load request for the memory address generated by CAGU. The data response is passed through the C2E unit to convert the data element format from concise to single precision, before writing it into the register file. For `stc` instructions (Figure 3.3b), the processor first performs a *companion load* to find the data contents at the requested memory address. In parallel to the companion load, the processor removes marginal bits from the store value using the E2C module, converting the data element format from single precision to concise. The concise data is then inserted at the appropriate location in the companion load response, which is later written to the load-store queue (LSQ).

Software Support. The ACME compiler allows the programmer to annotate those variables that are amenable to approximation, as is done in prior work [135, 52, 169, 168, 26]. In ACME, these take the form of `#pragma` directives in order to ensure the compatibility of ACME-enabled code with NON-ACME compilers. The ACME compiler takes the annotated application, an accuracy specification and a

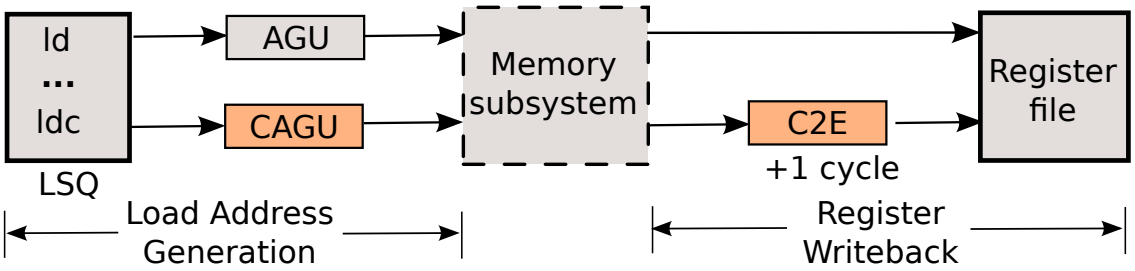


Figure 3.4: Execution of exact and concise loads

representative input dataset as input and generates an application executable that uses `ldc` and `stc` to enable concise storage. As illustrated in Figure 3.3c, the compiler generates `ldc` and `stc` for the annotated variables with the precision information (format length) as instructed by FSA. The resulting executable is then profiled and the accuracy and performance statistics are sent to FSA. FSA uses this information to decide the format length of the next step of binary search. In addition, the ACME compiler provides a `cmemcpy` (concise `memcpy`) function that uses concise memory operations to remove marginal bits from the approximated input variables, after the variables have been initialized.

3.3.2 Hardware Execution

ACME uses `ldc`/`stc` instructions to enable precise computation on concise data elements. These instructions reuse most of the existing processor micro-architecture with the help of three small additional hardware units - CAGU, C2E and E2C.

3.3.2.1 Execution of Concise Loads

Every load instruction (with or without ACME) has 2 steps as shown in Figure 3.4 – i) Load address generation, where Address Generation Unit (AGU) calculates the effective address to be sent to the memory, and ii) Register file writeback, where the data response from the memory is written back into the register file. ACME introduces additional hardware units in both of these steps to bring concise data el-

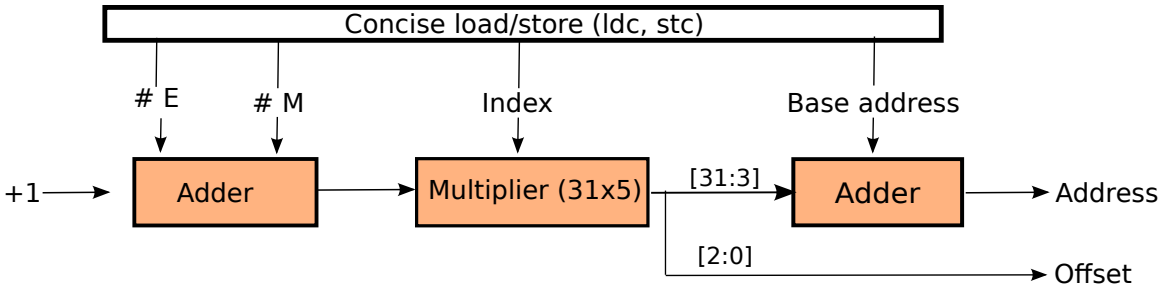


Figure 3.5: Block diagram of Concise Address Generation Unit (CAGU)

ements into the processor and convert them to the single precision format.

Load Address Generation. Conventional processors have dedicated functional units to calculate the effective memory address for loads and stores. These functional units are called Address Generation Units (AGU). By adding dedicated AGUs, memory instructions do not use integer ALUs for address generation, creating opportunities for executing more independent integer instructions in parallel. The compiler encodes the necessary information to perform address generation into the memory instructions while generating the application binary. This information is extracted by the instruction decoder and passed on to the AGUs. For example, in x86, array traversal uses a base register, index register for the array and the data size of each element (in bytes). In this case, the AGU performs the following integer arithmetic operation to generate the effective memory address: `(base_register) + data_size * index_register`.

However unlike conventional loads, ACME requires the capability of storing a data element of any arbitrary length in the memory, breaking the assumption that data elements are byte-aligned. This gives rise to situations where ACME requires bit-level access while the memory is byte-addressable. ACME solves this challenge by introducing the CAGU and C2E unit, allowing bit-level access in the data response of the concise loads while the caches and memory remain byte-addressable. These units thus serve as a transparent layer between the processor and the memory where everything else is byte-addressable by design while ACME has bit-level access in the

data response.

To accomplish this, the CAGU generates a byte-level memory address and a bit-offset to completely specify the address of a concise data element. The byte-level memory address is the closest byte preceding the requested concise data element. The bit-offset is the number of bits that are present between the above byte-address and the concise data element location. As shown in Figure 3.4, the CAGU first sends this byte-level address to the memory. The C2E unit then extracts the relevant bits from the data response using the bit-offset, converts them into the single precision format and stores the final 32-bit value into the register file.

Figure 3.5 illustrates the design of the CAGU. The instruction decoder extracts the precision information (the number of exponent and mantissa bits) from concise loads and stores to calculate the length of the concise format. CAGU multiplies the length of the concise format with the index register. Since the maximum length of a concise data element is 31, the format length can be encoded using 5 bits. Therefore, the above multiplication requires a 32x5 (for the index register and format length, respectively) integer multiplication unit. This intermediate value is the number of bits between the base address and the concise data element. Therefore, masking off the last 3 bits of this value results in a byte-level memory address which is closest byte-level memory address preceding the requested concise data element. Moreover, the least significant 3 bits of the intermediate value form the bit-offset, i.e. the number of bits to ignore in the memory response to get to the requested data.

While sending the concise load request, the CAGU also sends the bit-offset and the precision information along with the request. These are required later by the C2E unit to extract the relevant bits from the data response. If a cache miss happens, then the bit-offset and the packing information gets stored in the MSHR entries. The bit-offset requires 3 bits and the packing information requires $\lceil log_2(8\ exponent \times 23\ mantissa) \rceil = 8$ bits of storage. Therefore, each MSHR entry needs extra 11 bits

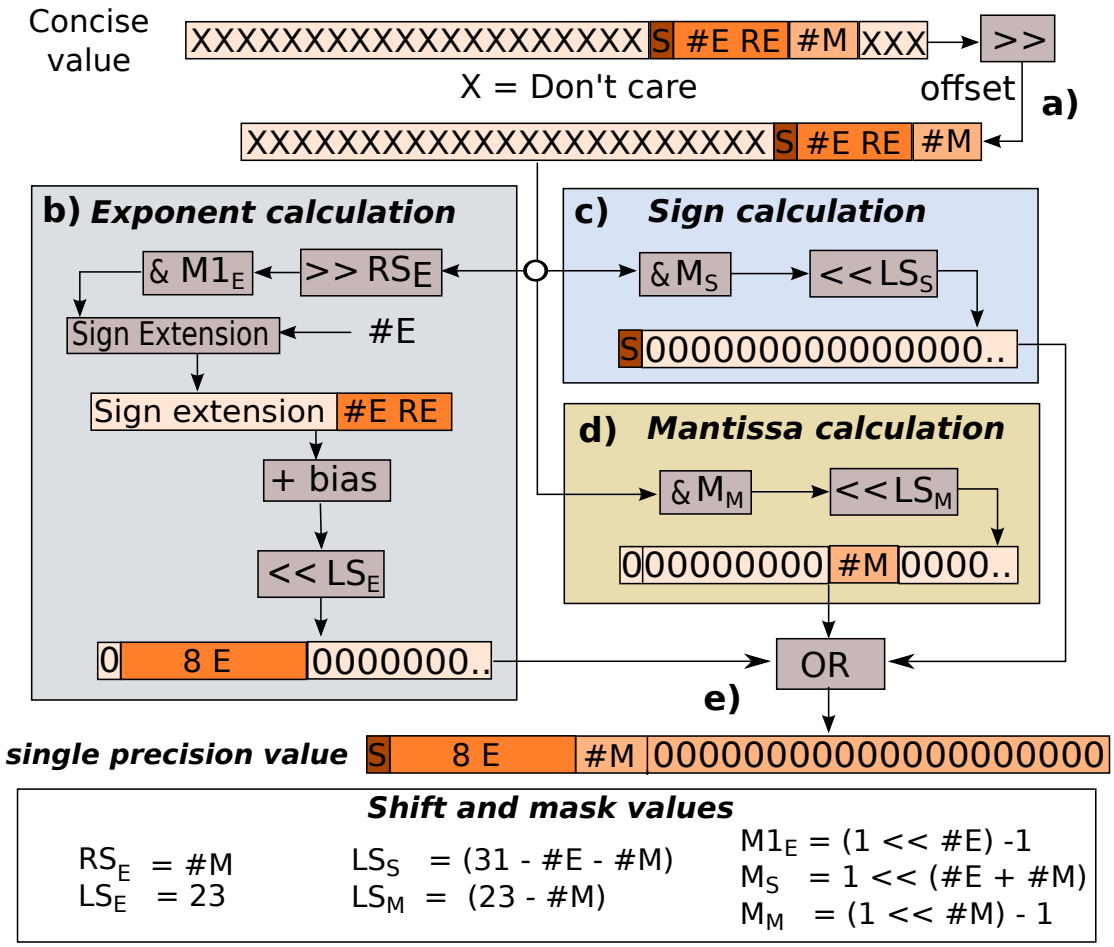


Figure 3.6: Block diagram of Concise to Exact (C2E) unit

of storage.

Note that exact loads also go through AGUs which have their own integer arithmetic units. Therefore, the CAGU does not add to the critical path of the processor for non-concise memory operations. We synthesize and report the timing characteristics of the CAGU in Section 5.4.

Register File Writeback. On receiving a concise load memory response, the C2E unit extracts the relevant bits using the bit-offset and precision information present in the memory response. The data is converted to the single precision format and stored into an intermediate register before being written to the register file. In the next cycle, ACME performs a lookup on the LSQ to find the destination register and

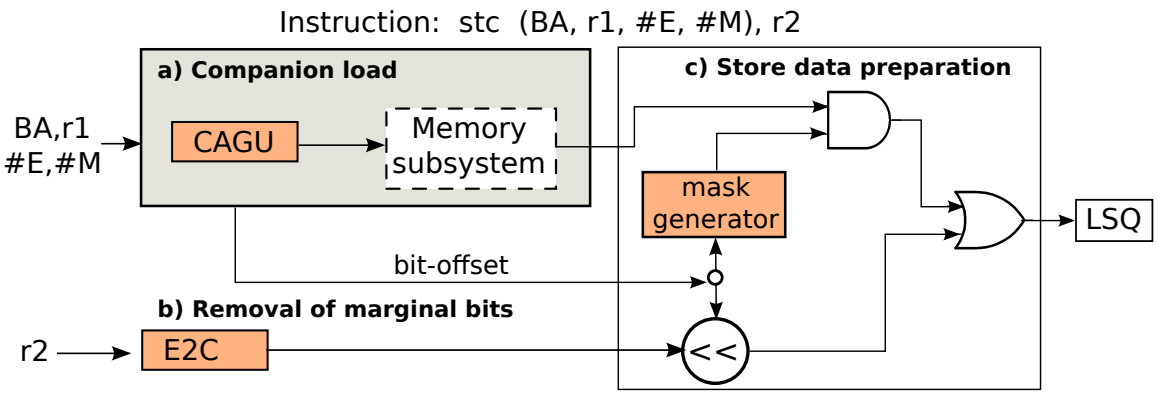


Figure 3.7: Execution of concise stores

performs a writeback into the register file.

Since each concise load performs format conversion from concise to single precision, this conversion has to be fast to provide the maximum performance benefits of concise storage. We introduce a Concise to Exact (C2E) unit to address this challenge. It converts the concise data into the single precision format in a single cycle. Figure 3.6 gives a step-by-step walk-through of this conversion process in the C2E unit. The process can be broken down into 5 steps – a) the C2E unit shifts the data response by bit-offset to align the relevant bits at the end, b) it masks and shifts this value to get the sign bit at the right position, c) similar operations are performed to put mantissa bits at the right position, d) the concise data has a raw (unbiased) exponent. This raw exponent is extracted, sign-extended and a bias of 127 is added to it to calculate the final exponent value. This exponent is then shifted and put at the correct position e) lastly, the C2E unit performs a logical OR operation on the sign, exponent and mantissa portions to get the final value in the IEEE floating-point format. This final value is written to an intermediate register.

In the next cycle, an LSQ look up is performed to find the destination register and the data is written back into the register file. From this point, the data is in single precision format and the computation happens precisely.

3.3.2.2 Execution of Concise Stores

Supporting arbitrary length concise stores in hardware is challenging because concise stores require partial byte modifications, while memory is typically byte-addressable. Concise loads solve this problem by reading the memory first and performing bit manipulations later. However, concise stores need to preserve parts of a byte in memory while modifying another part of the byte.

We solve this problem by performing a *companion load* to the relevant memory location alongside every concise store. This data returned by the companion load is then used to prepare the final store data to be written back to the memory. In our experiments, we have observed that extra companion loads have minimal performance impact, as they are greatly outnumbered by concise and conventional loads.

Concise store execution can be broken down into 3 steps as shown in the Figure 3.7 – a) Performing a companion load, b) removing the marginal bits from the register value, and c) preparing the store value.

Companion Load. Concise stores perform a companion load using the CAGU as shown in Figure 3.7a. This is performed to keep track of the bits (other than the required data element) that need to be preserved at the time of storing in the memory.

Removal of Marginal Bits. In parallel to companion load execution, the register value that needs to be written to the memory is stripped of its marginal bits using the Exact to Concise (E2C) unit. Figure 3.8 gives a step-by-step walk-through of this process – a) The register value is first rounded. This rounded value is used to find b) exponent, c) sign and d) mantissa portions separately which are then e) logically ORed to generate the concise value. Intuitively, these calculations are reverse of C2E calculations described earlier. In case the register value is beyond the range supported by current precision, it is clamped at the format-supported maximum/minimum value, whichever is closer. For representing value 0, we set all the bits in the concise format to 1.

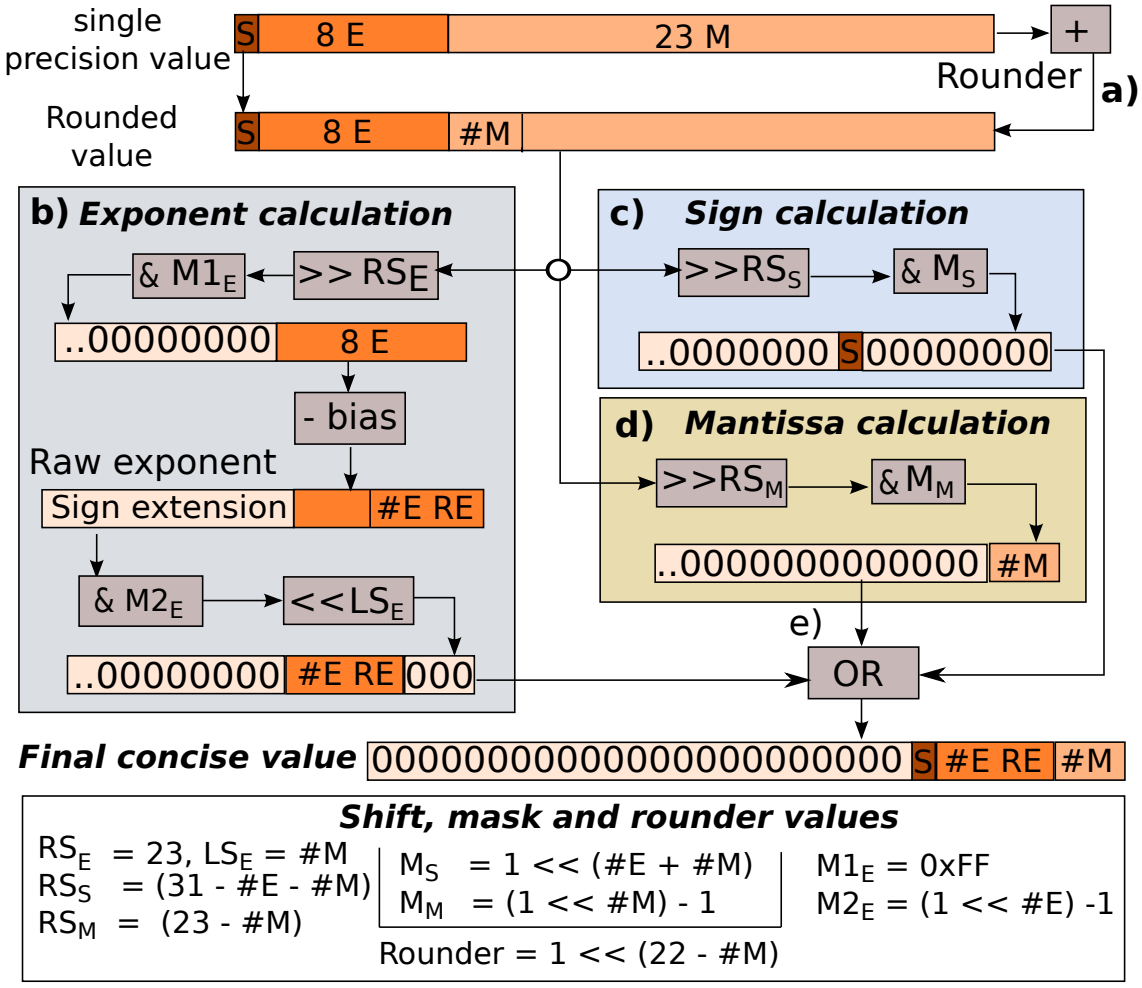


Figure 3.8: Block diagram of Exact to Concise unit

Store Data Preparation. Finally, ACME prepares the store value to be written back to the memory as shown in Figure 3.7c. ACME first left shifts the concise value by the bit-offset and brings it to the right position. The data response from the companion load is masked at the bit-locations that are going to be written by the concise value. These two values are then logically ORed. This value is then written to the LSQ. Finally, the value in the LSQ is written to the memory on instruction commit.

3.3.3 Software Support

ACME provides the flexibility to handle many levels of approximation by adding concise loads and stores; load-concise (`ldc`) and store-concise (`stc`) instructions. The ACME compiler is responsible for generating these concise loads and stores for the annotated variables. These instructions support storage of the concise data in memory with precise computation using the CAGU, E2C and C2E units. In addition, the compiler adds support for a `cmemcpy` function to remove marginal bits from the input dataset in the application code .

ISA extension. We use x86 assembly instruction `movl` to explain the workings of the `ldc` and `stc` instructions, though the idea can be extended to other ISAs as well. Consider the following load and store instructions:

```
movl (%ebx, %esi, 4), %eax ## Load    movl %eax, (%ebx, %esi, 4) ## Store
```

For traversal of an array, these memory operations use i) base address (`%ebx` in this example), ii) index register (`%esi`), and iii) data size (`4`). Since the base address and index are not known at the compile time, the memory address calculation (`%ebx + %esi * 4`) happens in the AGU at runtime.

Concise memory operations differ from their exact counterparts in the data size field. Here, compiler encodes the number of exponent and mantissa bits as instructed by FSA (`23` $\times$ `8` = `184` combinations, `8` bits). For example

```
ldc (%ebx, %esi, #E_#M), %eax
stc %eax, (%ebx, %esi, #E_#M)
```

In hardware, this precision information is extracted by the instruction decoder and passed on to the CAGU to perform memory address calculations.

Concise Memcopy Function. ACME requires a mechanism to remove the marginal bits from the annotated variables. There are several ways to perform this removal – directly converting the input data into concise format while initializing the approximated variables, or performing removal after the initializations are complete. We take the latter approach because it enables us to carefully evaluate the impact of removing marginal bits on the application speedup.

The ACME compiler adds support for a `cmemcpy` function that can be applied in the application code just after the variable initializations complete. All the variables are in IEEE format just after the initialization. The `cmemcpy` function is a simple loop that makes a pass over the annotated array, creating an in-place (smaller) concise copy of the data using concise store operations. In this way, the annotated input data elements are now stored concisely, fitting more elements in the memory hierarchy. In Section 5.4, we experimentally show that the overhead of applying `cmemcpy` is very small (<1% of application execution time).

3.3.4 Format Selection Assistant

ACME uses highly flexible ISA extensions providing a wide spectrum of precision configurations to choose from. Different applications have varying precision requirements, resulting in different number of marginal bits. ACME requires finding out this precision requirement for an application at a specified accuracy target. This requires navigating through a search space of precision configurations consisting of `23` mantissa * `8` exponent = `184` options for each annotated variable.

We introduce a Format Selection Assistant (FSA) to help ACME in quickly finding

this suitable precision level. It takes an application, a set of representative inputs, an error metric and an error bound (i.e., the maximum error an application can tolerate). It generates the minimum number of bits (e.g., number of exponent and mantissa bits for floating point numbers) to represent the input.

Tuning Algorithm. This approach leverages the observation that the accuracy of an application in asymmetric storage and compute will typically monotonically increase with length of exponent and mantissa bits. This enables us to leverage a greedy binary-search based approach to reduce the complexity of the accuracy space exploration. The algorithm is greedy because it finds a suitable precision configuration for the first variable while keeping others exact, then it fixes the precision of first variable and moves on to the second variable while keeping the others exact, and so on. In this way, this algorithm finds suitable precision for each variable one-by-one.

We use the intuition that exponent is typically much more important than mantissa for mathematical operations. Thus, we explore the exponent values first, using the maximum number of mantissa bits. For 32-bit IEEE floating point numbers, we start with 4 bits of exponent with 23 bits of mantissa. Once we determine the number of exponent bits using binary search, we again perform binary search over the length of mantissa bits. For each variable, this will require at most $\lceil log_2(8) \rceil + \lceil log_2(23) \rceil = 8$ executions instead of $8 \times 23 = 184$ executions in exhaustive approach. The Format Selection Assistant (FSA) can also be configured to apply a single format to all concise variables in the application, where the search occurs over 1 variable and the formats of all variables are kept in lockstep throughout the tuning algorithm. This reduces the search space significantly at the cost of some reduction in data conciseness, a tradeoff we evaluate in Section 3.4.3.

The final precision configuration at the end of the binary search is used for approximating the application. In addition to accuracy, this exploration also records performance of different precision configurations. In case the approximation results

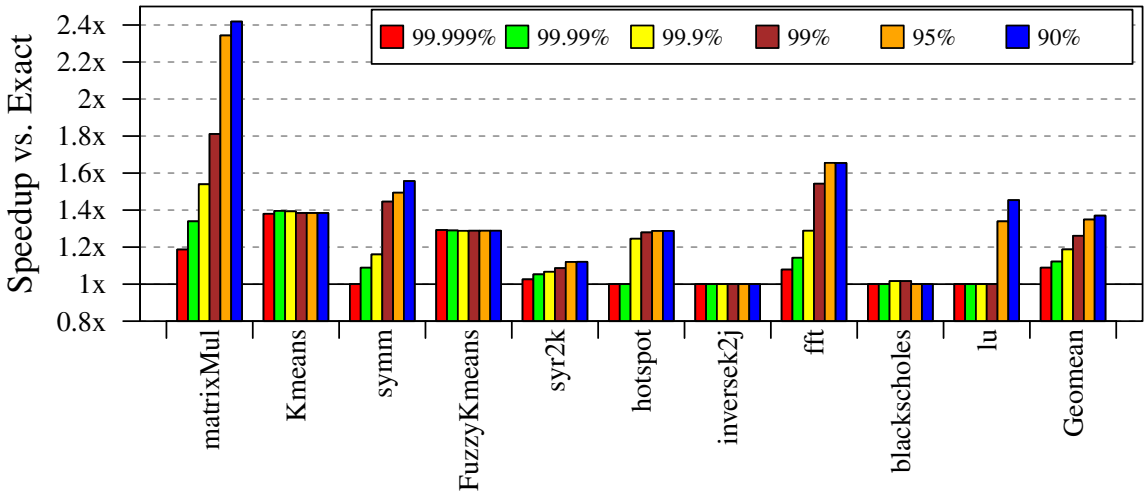


Figure 3.9: ACME performance benefits. ACME achieves good speedup for memory-bound applications

in a performance degradation compared to the exact execution, FSA instructs the compiler to drop the approximation.

3.4 Evaluation

3.4.1 Methodology

Applications. We evaluate ACME across 10 applications. We use matrixMul, symm and syr2k from PolyBench [58], Kmeans, FuzzyKmeans, inversek2j, fft and blackscholes from AxBench [52] and hotspot and lu from Rodinia benchmark suite [29]. These floating point applications are at the core of emerging machine learning and data mining workloads, having a mix of compute-bound and memory-bound applications and thus presenting a wide spectrum of program characteristics for evaluating ACME.

Accuracy Measurement. We use *average relative error* [52, 169, 138] as the error metric for our applications. Average relative error can be calculated using following

equation, where v_i is the exact value and v_i^* is the approximated value.

$$AverageRelativeError = \left[\sum_{i=1}^{N} |v_i - v_i^*|/v_i\right]/N$$

Performance and Energy Measurement. We evaluate the performance of ACME on Gem5 simulator [21]. We extend x86 ISA support in Gem5 by adding load-concise and store-concise instructions. We also add functional and timing models of ACME hardware components, CAGU, C2E and E2C units. A penalty of one cycle is added to concise loads and stores to account for conversion latency as detailed in Section 3.4.5.

Processor	8-wide OoO core, 3.0 GHz 192-entry ROB, 72-entry load queue
Private L1 cache	32 KB, 8-way, 2-cycle, 64 B block
Private L2 cache	256 KB, 8-way, 5-cycle, 64 B block
Shared LLC	2 MB, 16-way, 12-cycle, 64 B block
Main memory	1 GB, 200-cycle latency
L1 prefetcher	Tagged prefetcher
L2 and LLC prefetcher	Stride prefetcher

Table 3.1: Hardware configuration

Table 5.1 lists the specifications of the relevant hardware components that are configured to model an Intel Haswell processor. The applications are simulated for 5 billion instructions or to completion whichever is sooner. For measuring energy, we use McPat [103] and CACTI [115] to calculate the static and dynamic energy of core, caches, DRAM and ACME hardware units.

FSA Testing and Training. We partition the inputs into training and testing sets for all applications. We use the FSA to identify a suitable precision for the application on the training set and then used the same precision on the testing set. We found that the precision obtained from training satisfied the accuracy targets during testing. Unless otherwise noted, our experiments configure the FSA to use a single format for all application variables. The impact of using single- and multi-format configurations is explored in depth in Section 3.4.3.

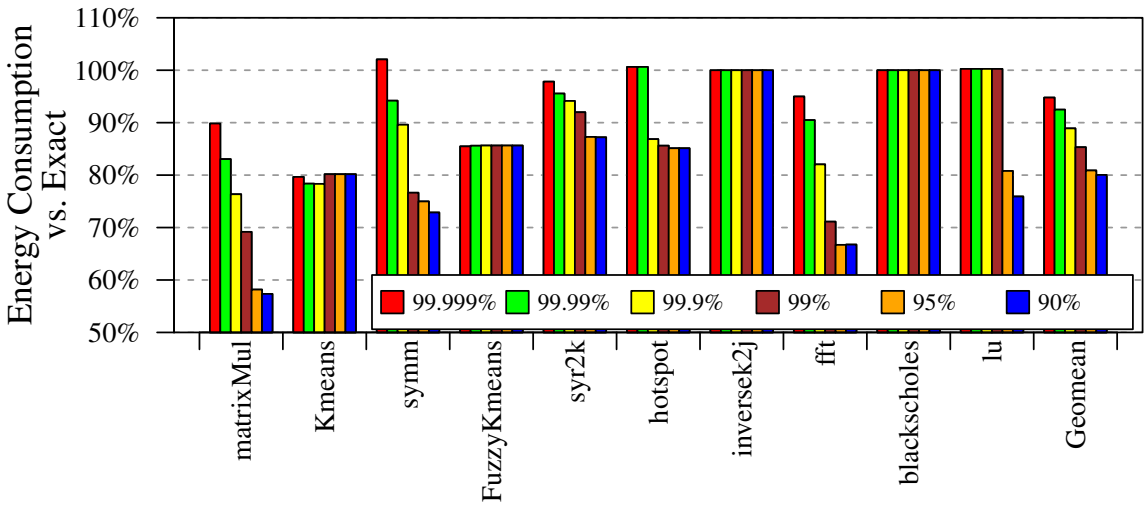


Figure 3.10: ACME energy benefits. ACME provides significant energy savings for memory-bound applications

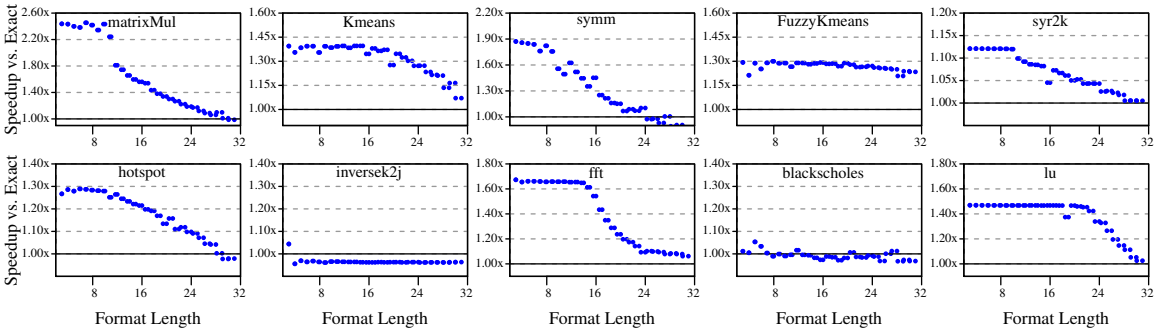


Figure 3.11: ACME performance study with varying format length. Smaller length yields less cache and memory pressure, resulting in higher application speedup

3.4.2 Performance and Energy Benefits

In this section, we evaluate ACME performance and energy tradeoffs for six accuracy targets. For each accuracy target, the number of exponent and mantissa bits is determined by the FSA. We use this precision configuration to find the speedup and energy savings compared to the exact execution.

Performance Accuracy Tradeoff. The performance-accuracy tradeoffs are shown in Figure 3.9. The figure shows the speedup of ACME against exact execution carried out on a non-ACME hardware for six accuracy targets. We observe significant speedup for applications that can benefit from larger caches. This occurs because ACME removes marginal bits from the memory subsystem, fitting more data elements into the lower level of memory closer to the processor. As one might expect, speedup goes up with looser accuracy constraints. Nevertheless, ACME gets speedup of 10% while attaining 99.999% accuracy. This is possible because some applications have large number of marginal bits whose contribution to the application accuracy is minute. For compute-bound applications, the FSA chooses exact execution, as reducing the data representation size has minimal impact on performance. For an accuracy target of 99%, ACME achieves a speedup of 1.8x for matMul, with an average of 1.3x for the whole application suite.

Energy Accuracy Tradeoff. Figure 3.10 presents the energy-accuracy tradeoffs of the same experiment. The figure shows the total energy consumed during the ACME execution compared to exact execution for six accuracy targets. Again, we observe that memory-bound applications consume lower energy compared to the exact execution. There are 2 reasons for this improvement. First, the application finishes sooner, leading to reduced static energy, and second, ACME reduces the number of DRAM requests leading to lower dynamic DRAM energy. ACME hardware components are small and consume minimal amount of energy. For an accuracy target of 99%, ACME reduces the energy consumption to 85% energy of the non-ACME

hardware on average.

Impact of Format Length. We next carry out a detailed performance evaluation of ACME with varying number of bits. The experimental setup consists of executing an application with different format lengths (number of bits used to represent a data element). For a particular format length, we can have different configurations of exponent and mantissa bits. The graph presents the one with the highest accuracy. The results of this experiment are presented in Figure 3.11.

ACME is able to achieve significant speedup for all memory-bound applications with small format lengths. Due to increased effective memory capacity and bandwidth, we observe higher speedup for smaller format lengths. These improvements outweigh the clock-cycle penalty of the C2E unit. With larger format lengths, the benefit of storing data concisely diminishes and extra clock cycle penalty by C2E becomes more prominent. For example for application symm, ACME achieves good speedup for small format lengths that use <16 bits but shows slight performance degradation for larger format lengths >24 bits.

We also observe that a few of the data points do not follow the speedup trend. For example, Kmeans at length = 20 and syr2k at length = 16 . This happens because mapping of data elements to physical cache lines changes with format length. A particular strided-access pattern for a certain format length can cause relatively more conflict misses than the adjacent format lengths. We observe abrupt increase in the number of misses for a certain cache for such format lengths. This is a well-studied cache effect [130].

Finally, as expected ACME does not improve performance for compute-bound applications: blackscholes and inversek2j. Blackscholes has minimal performance degradation because it has good ILP to keep its pipeline busy hiding the cycle penalty induced by the C2E unit. This is not the case in inversek2j, where the C2E penalty delays execution of dependent instructions, resulting in higher degradation. However,

we note that the FSA recommends not using concise types for these applications, and thus these applications do not slow down when compiled with ACME compiler.

The experiment demonstrates ACME's ability to improve the memory behavior of applications resulting in significant speedup and energy improvements for applications sensitive to cache and memory performance.

3.4.3 Format Selection Assistant

In this section, we show details of FSA-chosen concise format for different accuracy targets, shown in Figure 6.12. The figure shows the breakdown between the number of exponent and mantissa bits. We always keep the sign bit in the concise format.

We make 2 key observations from these results. First, the same application has different number of marginal bits for different accuracy targets. For example, matrix-Mul needs 8 bits for 90% accuracy but 24 bits for 99.999% accuracy. Second, different applications have different number of marginal bits for the same accuracy target. For example, Kmeans achieves 99% accuracy with just 5 bits whereas lu needs all 32 bits to achieve 99% accuracy. The results effectively demonstrates the need of designing a *flexible* approximation approach in order to get the desired accuracy targets.

For compute-bound applications, blackscholes and inverske2j, FSA chooses exact 32-bit representation for all accuracy targets.

Comparison to Oracle. We next compare performance achieved by the FSA configuration against an oracle system that finds the best precision configuration for the application by performing an exhaustive search over all the representation formats. The findings of this experiment are shown in Figure 3.13. For most of the applications, the accuracy increases and performance decreases with increasing the number of exponent and mantissa bits. Therefore, the greedy binary-search heuristic achieves performance close to the oracle in most cases. But as explained previously, some precision configurations result in relatively more conflict misses which results

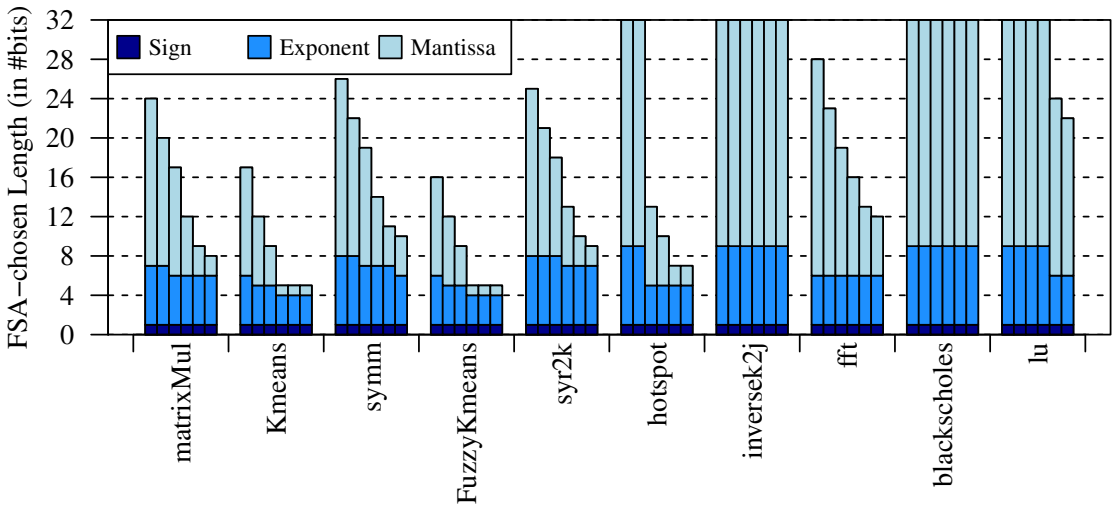


Figure 3.12: Breakdown of FSA chosen representation length for six accuracy targets - (left to right) 99.999%, 99,99%, 99.9%, 99%, 95%, and 90%

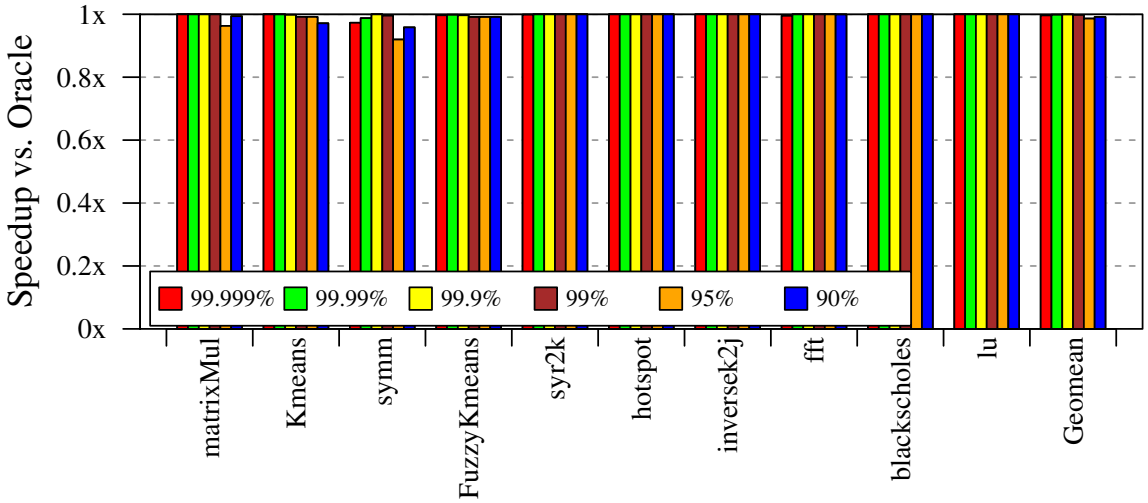


Figure 3.13: Comparison of FSA-chosen configuration performance against an oracle format selector. The FSA achieves > 98% of oracle performance on average

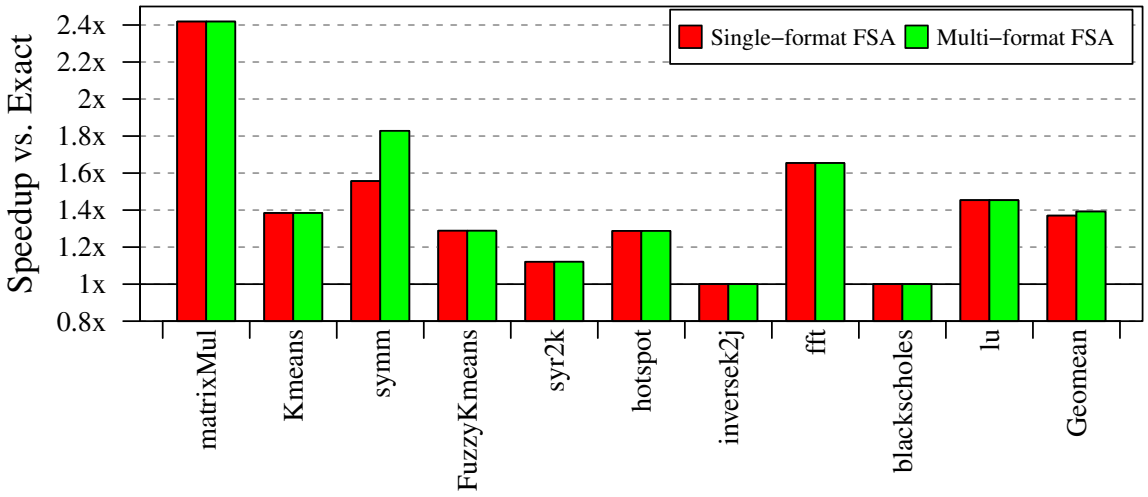


Figure 3.14: Comparison of speedup between single-format FSA and multi-format FSA

in sub-optimal performance compared to the oracle. Overall, FSA is able to achieve $> 98\%$ of the optimal speedup for all accuracy targets.

Different Formats Across Variables. The ACME hardware and compiler support using different formats among the different variables in an application. However, using different formats increases the complexity of the FSA tuning algorithm and thus increases compilation time. Here we evaluate the impact on performance of using a *multi-format* approach in the FSA. We allow the FSA to select formats among all applications in both multi-format and single-format modes at a 90% accuracy target, presenting our findings in Figure 3.14.

We observe that multi-format FSA precision settings provide minimal performance benefit on most applications. Kmeans, FuzzyKmeans and lu have only one variable suitable to approximation, and thus do not see any additional performance benefit when using multi-format mode in the FSA. For the compute-bound applications blackscholes and inversek2j, the FSA chooses exact execution in both multi-format and single-format mode. For 4 the remaining 5 applications that have multiple variables and are not compute bound, we observe negligible performance improvements when using the multi-format FSA. This occurs because, while the working set size

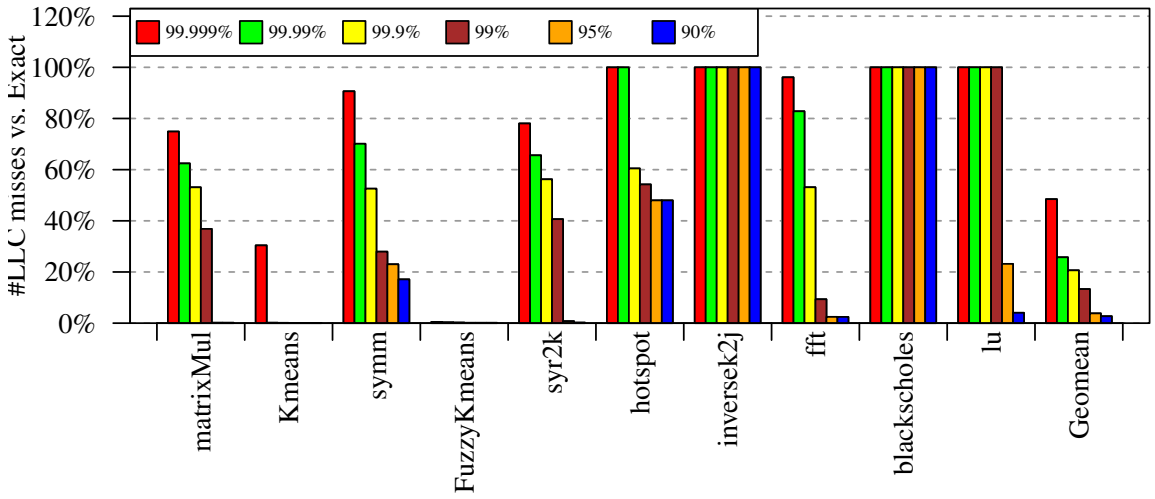


Figure 3.15: ACME reduces #off-chip memory requests which is a major source of speedup

may be somewhat improved by using the multi-format FSA, it often fails to reduce the footprint by enough to fit the application working set into a closer cache level. The single case where we observe a significant preformance improvement is for symm, where such a reduction occurs.

3.4.4 Memory Behavior

ACME achieves concise storage by removing the marginal bits throughout the memory subsystem. This results in an increase in effective capacity and bandwidth, improving performance. A major source of speedup comes from reduction in LLC misses. LLC misses are expensive as processor has to wait for DRAM to satisfy the miss. In this section, we perform experiments to understand how ACME impacts memory behavior.

LLC Miss Reduction. We compare the LLC Misses for FSA-chosen configuration for six accuracy targets against exact execution, presented in Figure 3.15. As expected, ACME brings down the number of LLC misses substantially, which is one of the major causes of performance improvement with ACME. On average, ACME reduces the number of LLC misses by 85% at an accuracy target of 99%.

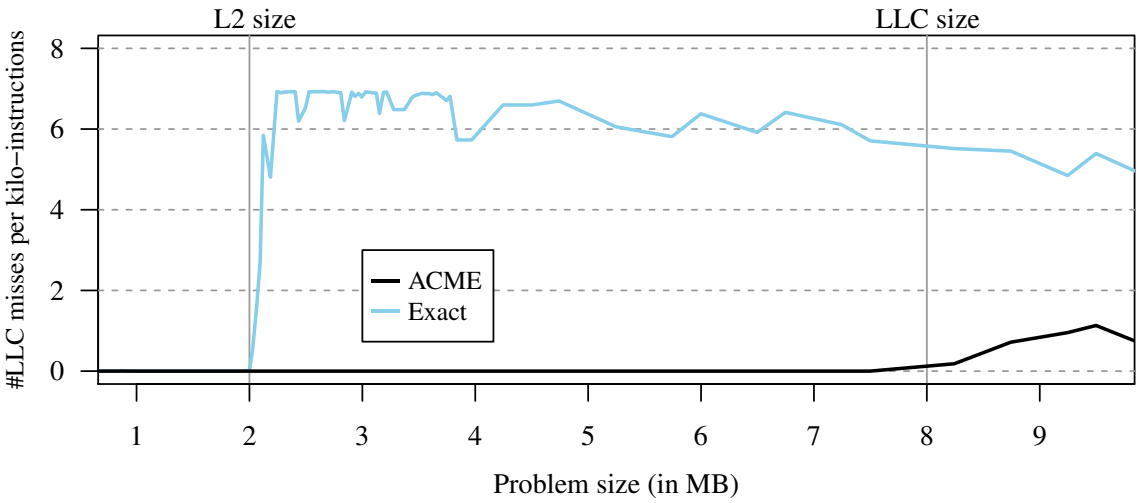


Figure 3.16: LLC misses when varying matrixMul working set size with exact and ACME execution

Impact of Working Set Size. In this experiment, we perform a detailed study on matrixMul with varying working set problem sizes. The experimental setup consists of running exact and ACME version of matrixMul with different problem sizes and then measuring the effect on IPC and LLC misses. The format length chosen for the concise storage is 8 bits which enables us to fit 4 times as many elements in memory-subsystem as compared to exact. The results of this study are shown in Figure 3.16 where the problem size varies from 1 MB to 9 MB.

When the the problem size is less than 2 MB (the size of our LLC), both the exact and approximate data fits into LLC. Therefore, the number of exact and approximate LLC misses are similar resulting in similar performance for exact and ACME execution. However, as the exact problem size goes beyond 2 MB, we start seeing larger number of exact LLC misses. ACME is still able to fit the data in LLC because it is using only 8 bits to represent the input elements. It is only for configurations larger than 8 MB that ACME begins to introduce increasing numbers of LLC misses.

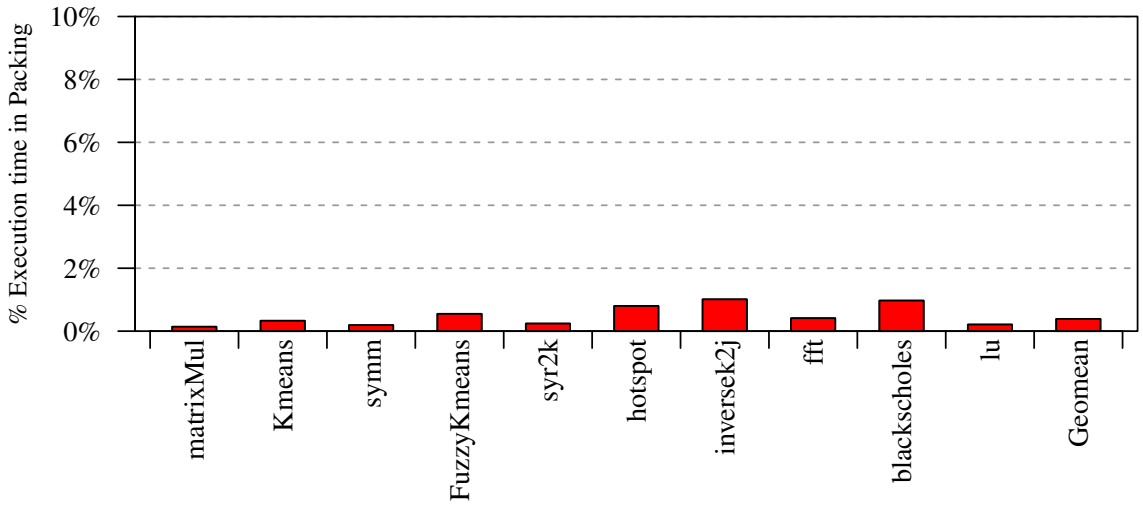


Figure 3.17: Overhead of cmemcpy function. The function consumes minute portion of total application execution time

3.4.5 System Overheads

In this section, we discuss the overhead associated with different components of ACME. Note that all these overheads are already included in other parts of the evaluation.

Packing Overhead. ACME compiler adds a `cmemcpy` in the application code to represent the input elements more concisely. Figure 3.17 shows the portion of application execution time spent in `cmemcpy` function. We see that this overhead is <1% in all the applications. Our hardware implementation removes the marginal bits by performing complex conversions quickly in the hardware, resulting in a minimal overhead. We also implemented a software implementation of store-concise instruction and used it for the `cmemcpy` function. However, we observed as much as 10% overhead with the software implementation, resulting in reduced performance improvements.

Hardware Overhead. In this section, we discuss area, power and frequency numbers for the additional hardware components. We implement CAGU, C2E and E2C unit in Verilog and synthesize it using ARM Artisan IBM SOI 45 nm library. The area, power and frequency of the C2E unit is 0.0034 mm^2, 9.41 mW and 2.78 GHz

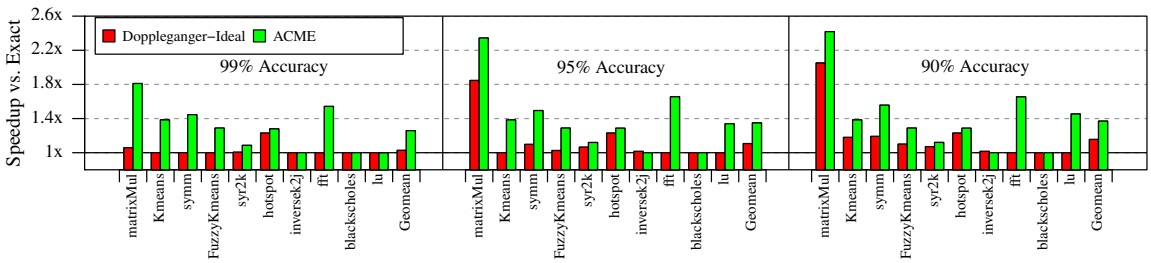


Figure 3.18: ACME vs Doppleganger-Ideal; ACME achieves higher concise storage throughout the memory hierarchy resulting in better application speedup

respectively. Similarly, the numbers are 0.0023 mm^2, 4.23 mW and 2.78 GHz respectively for the E2C unit, and 0.0044 mm^2, 12.7 mW and 2.22 GHz respectively for CAGU. Our baseline is a mainstream core-i7 Haswell processor that operates at a frequency of 3.0 GHz and consumes 177 mm^2 of die area. We see that the additional overhead of ACME units is minimal: 0.0052% area overhead and <0.1% power overhead. By using technology scaling trends [110] to project the frequency for hardware components for 22nm, we find that ACME units can operate at the target frequency of 3 GHz at 22nm. This study shows that additional hardware components are fast and consume minimal area and power.

3.4.6 Comparison to Prior Work

In this section, we compare ACME against a state-of-the-art approximate computing cache technique; Doppleganger [138]. Doppleganger increases effective LLC capacity by finding LLC lines that are similar. Approximately similar cache lines are mapped to single line resulting in increase in effective cache size.

Doppleganger finds approximately similar cache lines by encoding the range and average of the values present in the cache lines. This encoding takes form of an N-bit hash map. Two cache lines are treated approximately similar if they produce same map value. Lower the value of N, higher is the compression ratio at the expense of higher application error. Doppleganger builds this N-bit hash function into the hardware at design time, preventing any accuracy knob. Consequently, Doppleganger

might not be able to satisfy an accuracy target with a N-bit hash function. We create an idealized version of Doppleganger, Doppleganger-Ideal, where it is not restricted by a fixed value of N. Instead, it finds the minimum value of this N for each application and accuracy target separately. This lets us measure the approximate similarity in the application which is equivalent to the magnitude by which the effective LLC size is increased. To simulate this effective increase in LLC size for Doppleganger-Ideal, we increase the actual size of LLC as per the measured similarity without increasing the cache latency.

The comparison between ACME and Doppleganger-Ideal is presented in Figure 3.18. Doppleganger-Ideal shows speedup for some applications for 90% and 95% accuracy but its speedup drops significantly for 99% accuracy. We see that ACME performs better than Doppleganger-Ideal in all the applications, except inversek2j for accuracy target of 90%. There are 2 reasons for this performance difference. First, ACME achieves more concise storage compared to Doppleganger-Ideal. Doppleganger is limited by finding redundancy across cache blocks. ACME, instead, finds the bits that marginally contribute to the accuracy and removes them from the data representation. Second, ACME achieves concise storage throughout the memory hierarchy, compared to Doppleganger-Ideal which operates only on the LLC.

3.5 Summary

This paper introduces a novel asymmetric compute-memory extension to conventional architectures, ACME, that decouples the format of data in the memory hierarchy from the format of data in the compute subsystem. ACME significantly reduces the cost of storing and moving bits throughout the memory hierarchy improving application performance. We add two instructions to the ISA - *concise-loads* and *concise-stores* which are supported in hardware vis three small functional units. Our results show that ACME achieves 1.3× speedup (up to 1.8×) while maintaining

99% accuracy, or 1.1× speedup while maintaining 99.999% accuracy, while incurring negligible area and power overheads; 0.005% area and 0.1% power to a conventional modern architecture.

CHAPTER IV

Continuous Shape Shifting: Enabling Loop Co-optimization via Near-Free Dynamic Code Rewriting

The class of loop optimizations that reshape the iteration space for cache locality and reuse are traditionally static compiler optimizations [164, 165, 22, 93]. With a specification of microarchitectural design and cache topology in a processor, the computation in an application's nested loops is restructured with strip mine and interchange passes into *tiles* – small subsets of the working set that fit into the cache – to take advantage of data reuse and improve the effectiveness of the cache for the computation. As a statically parameterized optimization, tiling requires that the compiler control both the size and shape of the tiles used in the computation. The choice of these parameters is intimately linked to the characteristics of architectural resources available to the application as it runs [38, 28, 113]. However, this class of optimization was conceptualized before the multicore era, which has introduced numerous additional dynamic factors that affect application runtime environments.

The advent of highly dynamic multicore/multiprocessor environments necessitates the rethinking of how cache tiling should be applied and deployed in commercial and production contexts. The static assumptions used to aggressively tune the tiling

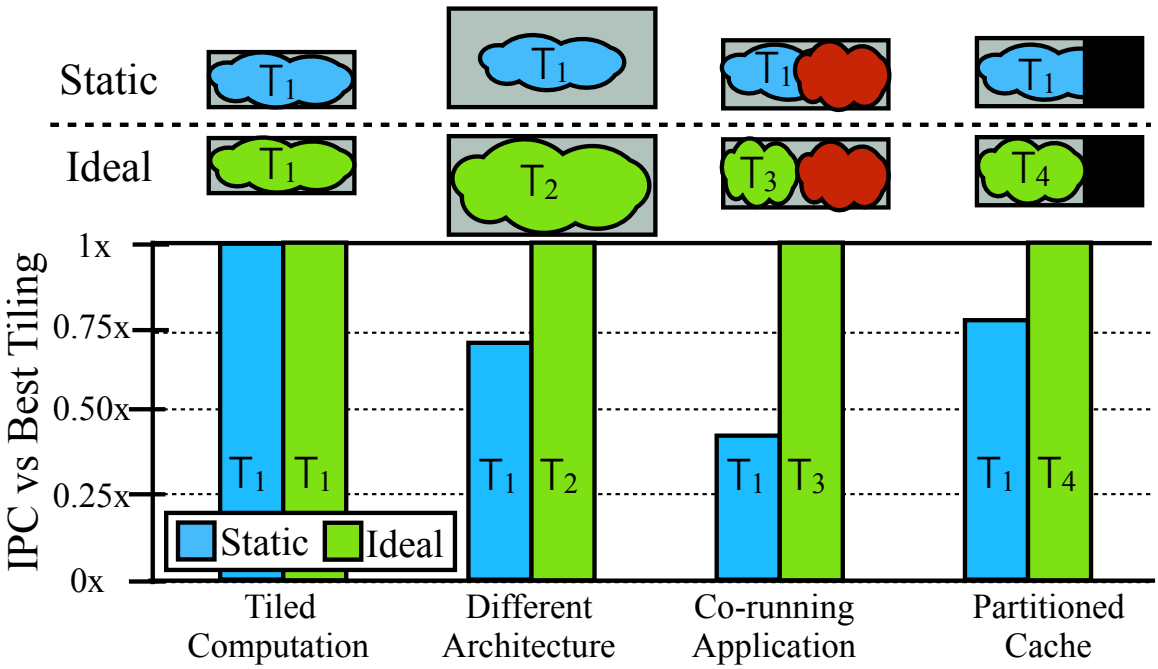


Figure 4.1: The optimal tiling for one runtime environment can perform poorly in other environments

parameters can be easily broken by sources of post-deployment dynamism. This concept is illustrated in Figure 4.1, which compares two tiling approaches. First, an approach that aggressively tiles for one runtime environment achieves excellent performance in that environment, but may perform poorly in other environments. Second an ideal approach that aggressively tiles for each runtime environment. The figure shows that pre-deployment best tile can result in sub-optimal performance across different runtime environments.

Although there has been some prior work addressing particular challenges that arise from dynamism [18, 151, 156], these works use white-box approaches to target particular sources of inefficiency. Realizing a holistic approach that continuously adapts to numerous, varied sources of post-deployment dynamism requires a black box approach and remains an open problem. In particular, three sources of dynamism must be addressed to realize a loop iteration space specialization that is deployable in modern commercial and production environments. These sources of dynamism

include:

1. **Co-runner Dynamism.** Cloud and datacenter operators routinely co-run applications to improve server utilization [112, 45, 176] and multi-program workloads have become a norm on desktop and mobile platforms [54]. The co-runners an application faces will vary in number and character.

2. **Microarchitectural Flexibility.** Processor design has evolved significantly since the original conceptualization of cache tiling. Now, microarchitectural parameters may change over the course of an application run. For instance, cache way-gating [55], processor power capping [43] and cache partitioning [124, 140] may be used to slow, constrain, or shut down architectural resources in order to limit power consumption or provide performance isolation.

3. **Microarchitectural Diversity.** Datacenters operators typically house numerous architectural implementations [20, 111, 7], and heterogeneous architectures, for example ARM big.LITTLE, are becoming common because of their energy efficiency. Moreover, the target platform for commercial off the shelf (COTS) software is rarely known ahead of deployment, and each platform may have different cache configurations and microarchitectural implementations.

Each of these sources of dynamism impacts the availability of important architectural resources to the application, significantly affecting how cache tiling should be aggressively employed. The set of factors impacting the choice of cache tiling parameters is broad, have complex interactions, and may change many times over the course of a single application run. Handling this myriad factors therefore demands a novel, dynamic solution that can quickly and seamlessly change the tile structure to reflect changes to an application's runtime environment.

A key insight of this work is to use a rapidly and dynamically constructed environment- and application-specific *black box* model for predicting the performance of a host of

tiling options within the immediate environment. This paper introduces **continuous shape shifting** with *ShapeShifter*, an end-to-end dynamic compilation infrastructure that enables continuous shape shifting and aggressively rewrites running applications in response to runtime dynamism. ShapeShifter uses a lightweight monitoring infrastructure to examine the running applications and the runtime environment to look for opportunities to tile and re-tile the applications in response to changes in the runtime environment. Upon identifying a suitable tile shape based on the dynamically constructed model, ShapeShifter rewrites and re-tiles the application leveraging a low-overhead dynamic compilation capability to divert execution into the aggressively tiled code with near-zero overhead.

In addition to continuous shape shifting, we propose a *co-optimization* algorithm to perform retiling of multiple co-runners simultaneously. It is a challenging problem to find suitable tile shapes for multiple co-runners because optimizing a tile shape for a co-runner can change the optimal tile for an already optimized co-runner. We observed that cache interference often has little to do with tile shape, i.e., different tile shapes of the same tile size produce similar amount of interference to other co-runners. This observation can be leveraged to design an approach that quickly finds suitable tile shapes. This is the first work to consider the effect of this dynamic interference in the presence of multiple co-runners.

We evaluate ShapeShifter on real systems within a spectrum of runtime environments spanning several architectural platforms, showing that by aggressively retiling application tiles, we are able to achieve an average of 10-40% performance improvement (up to 2.4×) over an oracle that aggressively tiles for a single runtime environment.

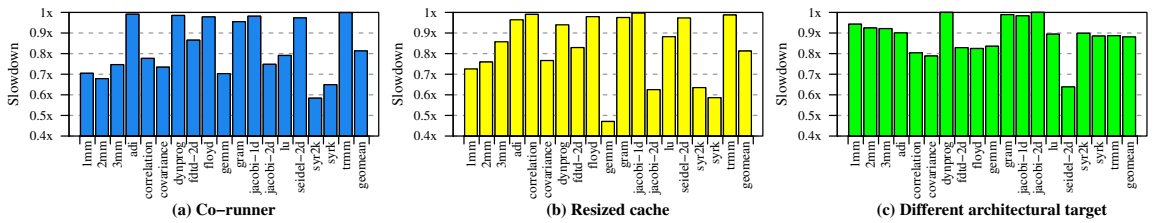


Figure 4.2: Suboptimal performance if the application code is not retiled to the application runtime environment

4.1 Motivation

In this section, we investigate the opportunity available in the presence of a solution that can aggressively re-tile application code in the context of three common sources of post-deployment dynamism.

4.1.1 Opportunity Analysis

The efficacy of a cache tile depends heavily on the runtime environment, as there are numerous factors in the runtime environment that can impact the availability of cache and other microarchitectural resources. This study focuses on three such sources of dynamism – the impact of co-running with other applications, the impact of changing the amount of cache available to the application, and the impact of microarchitectural diversity. Our baseline is an approach we call *StaticBest* that exhaustively runs a large space of tiling parameters to determine the best tiling configuration for a runtime environment that (1) has no-co-running applications, (2) is for a commodity server processor (AMD Opteron), and (3) for which the application has full use of the 16-way L2 cache. We evaluate on a host of applications from Polybench [123].

Co-runner Dynamism. We first evaluate the efficacy of StaticBest when the assumption that the application has no co-runners proves to be untrue (Figure 4.2(a)). For this comparison, we run the applications again against a cache pressure microbenchmark while employing each of the tiling parameterizations used to find Stat-

icBest, then selecting the best performing tiling, which we term AggressiveBest. The difference in performance between StaticBest to AggressiveBest can be interpreted as the slowdown resulting from a failure to tailor the tiling approach to its runtime environment. Figure 4.2(a) illustrates the resulting slowdown, which is over 19% on average, and up to 41% for syr2k.

Microarchitectural Flexibility. We next evaluate the efficacy of StaticBest when the assumption that the full L2 cache is available to the application is violated. We use Bulldozer's way-locking feature to lock half the ways of the 16-way L2 cache, effectively reducing the cache size by half. The resulting slowdown if the applications are not re-tiled to respond to this microarchitectural change is illustrated in Figure 4.2(b). In this case, up to a 52% slowdown over the optimized tile is observed for gemm, with an average of 19% across all applications.

Microarchitectural Diversity. Finally, we evaluate the efficacy of the optimized tile if the assumption that the target architecture is an AMD Bulldozer is violated. To do this, we find AggressiveBest when running the applications on an Intel Haswell server and compare the resulting performance to StaticBest on the Haswell server. Like the previous cases, a significant performance opportunity is left unexploited if applications are not re-tiled to reflect this different runtime environment. The maximum resulting slowdown is 37% for seidel-2d and averages 12% across applications.

4.1.2 Limitations of Prior Work

We compare ShapeShifter to the most relevant previous work in Table 1 [151, 18, 156]. Both Defensive Tiling [18] and Dynamic Selection of Tile Sizes [156] do not retile multiple co-runners simultaneously. It is a necessary and challenging problem to solve as optimizing the tile for one application can change the best tile for an already optimized co-runner. ShapeShifter has the capability of retiling multiple co-running applications. We provide insights as to how tile shape and size affects

Table 4.1: Comparison between ShapeShifter and prior retiling works

	Defensive Tiling[18]	**Reactive Tiling**[151]	**Dynamic Selection**[156]	**Shape-Shifter**
Retiling multiple co-runners				✓
Rectangular tiles		✓	✓	✓
Black-box model approach		✓	✓	✓
Compilation flexibility				✓
Real system evaluation	✓		✓	✓
Handles co-runner presence	✓			✓
Handles cache-partioning		✓		✓
Handles platform changes				✓

the interference between applications. We then present an algorithm built upon that insight to find suitable tiles for co-runners simultaneously. Reactive tiling [151] strives to find the best tiling parameters in the presence of cache partitioning. However, it is evaluated on simulators, which have limited ability to capture industry-standard proprietary features such as prefetcher designs and cache replacement policies. In addition, ShapeShifter supports dynamic compilation providing the opportunity to use wide range of compiler optimizations suitable to the runtime environment.

4.2 System Overview

This section describes the design of ShapeShifter, a dynamic compilation infrastructure that takes advantage of the opportunity to aggressively re-tile running application code to reflect the runtime environment. We discuss the main challenges in designing such an infrastructure, and give an overview of how ShapeShifter overcomes these challenges.

4.2.1 Challenges

Accuracy. Realizing a tiling approach that is *universal*, capable of identifying the right tile among a broad range of runtime environments, is a challenging problem. Existing solutions using detailed cache and memory access pattern models are designed to focus on a narrow range of the possible runtime environments. Thus, designing a

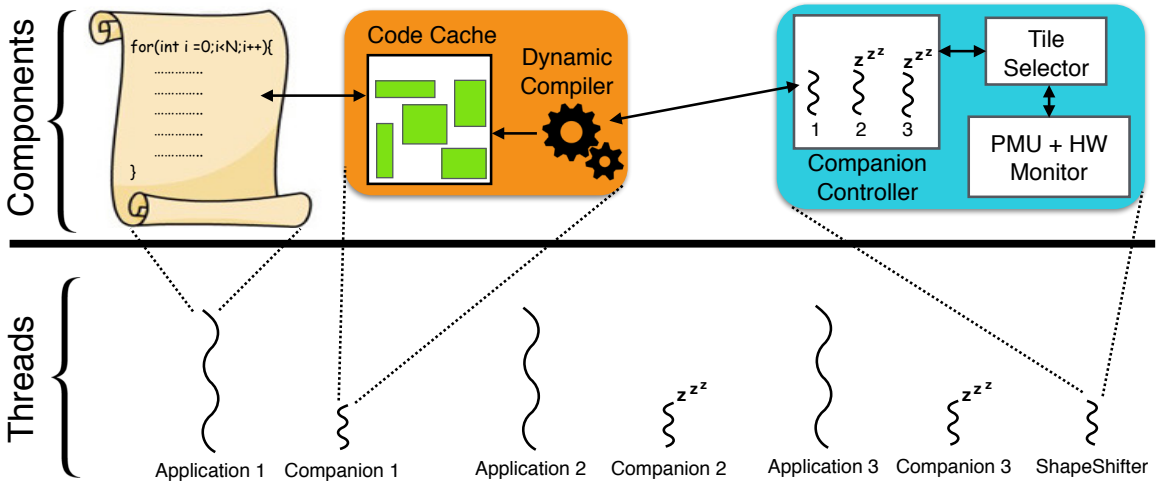


Figure 4.3: Dynamic compilation and monitoring infrastructure

new mechanism capable of reasoning about cache tiling and correctly identifying the most suitable tiling parameters among many runtime environments is necessary for solving this problem.

Overhead. A dynamic re-tiling system must be left in place continuously throughout application execution, available to monitor the application and environment, and able to take steps to exploit re-tiling opportunities as they arise. Having such a capability that is low overhead is a challenging problem. Classic virtualization-based monitoring and dynamic compilation infrastructures are ill-suited to this task, as the overhead introduced by those infrastructures can easily outweigh benefits of the optimizations themselves.

4.2.2 ShapeShifter System Architecture

ShapeShifter is an end-to-end dynamic system continuously monitoring the running application and runtime environment and looking for opportunities to re-tile the application code. The runtime environment can change because of arrival/departure of co-runners, architectural policy changes and platform changes.

To achieve this dynamic capability, ShapeShifter spawns a runtime thread for each application as soon as it starts execution, as shown in Figure 4.3. This thread,

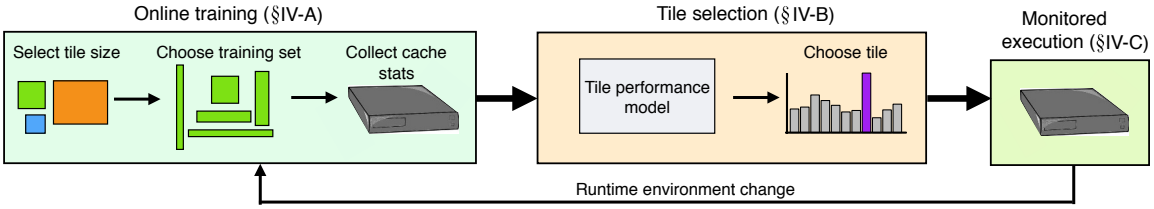


Figure 4.4: Tile selection in ShapeShifter is accomplished by running a small training set of tiling parameters, which is used to model the IPC of a large space of tiling then to select the tiling with the highest IPC

referred to as *Companion thread*, provides a dynamic compilation capability to its application. ShapeShifter continuously looks for tiling opportunities by using a *Runtime Environment monitor*. When an opportunity is identified, it triggers a *Tile Generator* module that accurately predicts a suitable tile for the current runtime environment. Finally, ShapeShifter instructs Companion thread to introduce the new tiling strategy into the application code.

Companion threads, Tile Generator and Runtime Environment Monitor work in tandem to achieve continuous shape shifting. Here we provide an overview of these components.

Companion Thread. Companion threads provide Dynamic Compilation infrastructure inspired by protean code to introduce re-tiled code into the running application [96]. The key difference between protean code and other traditional heavyweight dynamic compilation infrastructures is that protean code runs asynchronously to the application, without stalling the application progress. The application continues running the old code variant and switches to the new code variant only when protean code has lazily stitched it into the running application. Therefore, protean code incurs low overhead on the application performance.

Companion threads are woken up only when a tiling opportunity is detected, as illustrated in Figure 4.3. Because of its minimal interaction with the application, it provides a low-overhead dynamic compilation solution to achieve continuous shape shifting.

Runtime Environment Monitor. One of the key capabilities of ShapeShifter is to detect opportunities to re-tile running application code. This capability takes the form of a *Runtime Environment Monitor* (REM), a lightweight process that occasionally polls the machine state via hardware event monitors and model specific registers (MSRs). It collects performance and cache statistics counters that are used to guide tiling decisions. MSRs often expose useful information about microarchitectural state. This information helps in constructing a view of the application runtime environment. For example, the AMD Bulldozer platform support way-locking in the L2 cache, and MSRs expose the number of L2 cache ways available at any given time. By monitoring the relevant machine state, ShapeShifter can detect changes in the architectural policies at any time.

Tile Generator. Tile Generator is responsible for predicting a suitable tile for the application current runtime environment. It uses the performance and cache statistics collected by REM to generate an online black box linear model. Using this model, ShapeShifter predicts a tile that is optimized for the current runtime conditions. It instructs the Companion thread to generate the corresponding tiled variant and stitch it into the application code.

4.3 ShapeShifter Design and Implementation

In this section, we provide description of ShapeShifter runtime system. The different components of ShapeShifter – Companion threads, Tile Generator and Runtime Environment Monitor – work hand-in-hand to identify and take advantage of tiling opportunities. Figure 4.4 gives an overview of the ShapeShifter runtime system. Whenever REM detects a tiling opportunity, Tile Generator starts constructing an application- and environment-specific tiling performance model. It instructs the Companion thread to stitch a handful of different tile parameterization codes into the application, where each is run for short time. REM collects the performance and

cache statistics, referred to as Training data, while these tiles execute. This training data is used by Tile Generator to construct a tiling performance model on the fly. Tile Generator then selects the tiling with the highest modeled IPC and invokes Companion thread to introduce that tiling into the application. We show in Section 5.4 that this tile generation process is highly accurate, choosing tiling strategies close to optimal-tiling performance.

We divide the above process into three parts: Online training (§4.3.1), Tile Generation (§4.3.2), and Monitored execution (§4.3.3). We now describe these three steps in detail.

4.3.1 Online Training

Online training is triggered when the REM detects a change in the application runtime environment. In this step, the REM collects training data with the help of the Tile Generator and Companion threads. This training data is then later used to develop a tiling performance model. The process can be further broken down into 2 steps. First, finding a suitable tile size for the application and second, collecting training data.

4.3.1.1 Tile Size Selection

Both tile size and shape are important tile characteristics that impact the performance of a tiled loop nest. Tile size defines the working set of the application. A working set larger than the targeted cache size slows down the application because some memory requests take longer to finish as they have to go to lower and slower levels of memory hierarchy. On the other hand, a working set much smaller than the cache size does not utilize the data reuse efficiently. This step tackles the problem of finding a suitable tile size for the application, whereas the problem of finding a suitable tile shape is solved by the black box model described in Section 4.3.2.2.

On detecting a change in the application runtime environment, the REM reads in current cache size using software visible registers and MSRs. This information is passed on to Tile Generator that instructs the Companion thread to generate a tile variant consuming a certain portion of the available private cache size. Companion thread executes this tile variant while REM collects the performance and cache statistics during the tile execution. This process is repeated with reduced tile size until the private cache miss rate is below a certain threshold (<2% in our case). This low cache miss rate signifies that the working set of the application now fits in the cache. This produces a tile variant whose tile size is tuned for the application current runtime environment.

4.3.1.2 Collection of Training Data

On finding a suitable tile size, ShapeShifter starts collecting training data to help generate a tiling performance model. In this step, Tile Generator generates a set of training tiles, Companion threads executes these training tiles one by one for short duration while REM collects the performance and cache statistics for each tile. Algorithm 1 provides an overview of this whole process.

Algorithm 1 Online training

```
1: Input: TileSize
2: Output: TrainingData
3: function GetTrainingData(TileSize)
4:     GenTrainingSet(TileSize)                          ▷ Tile Generator
5:     for (i in 1:size(TrainingSet)) do
6:         GenTiledVariant()                             ▷ Dynamic Compiler
7:         DispatchTileToApp()                           ▷ Dynamic Compiler
8:         RunTheTile()
9:         data = CollectPerfMonData()                   ▷ REM
10:        TrainingData = TrainingData + data
11:    end for
12:    return TrainingData
13: end function
```

Tile Generator first generates a set of training tiles using the tile size identified in

the previous step but with varying tile shapes. There is a broad range of tile shapes to choose from. We classify the tile shapes in 3 categories: Broad tiles – tiles with large number of rows but few columns, Narrow tiles – tiles with few rows but large number of columns, and Intermediate tiles – tiles that are neither broad not narrow. As shown in Figure 4.4, Tile Generator chooses only a subset of these tiles to profile the application. In total, the training set consists of 5 versions of application - 2 broad-tiled, 2 narrow-tiled, 1 intermediate-tiled.

Tile Generator instructs the Companion thread to introduce training set into the application code. These tiles are then executed one by one while REM collects performance counters during their execution. Specifically, REM collects this information for each tile in the training set: a) number of retired instructions, and b) number of execution cycles . This creates a training database which is later used to develop a tiling performance model.

4.3.2 Tile Generator

The training data is now used to develop a model and identify a suitable tiling strategy for the application runtime environment. The runtime environment can change because of various factors like arrival of co-runners, microarchitectural policy changes and platform changes. Figure 4.5 gives an overview of tile generation. Tile Generator uses the training data collected by REM and creates an online black box model for the current application and runtime environment. The black-box model does not assume any prior knowledge of the application and architecture, and is completely created on the fly using the training data. Since the model has to capture only the current application and runtime environment, a relatively simple model can suffice to achieve high prediction accuracy. This is in contrast to traditional white-box models that are quite complex because they are designed to handle a wide variety of cases.

4.3.2.1 Black-box Development

ShapeShifter uses the online training data to develop a black box model. Our goal is to generate a model that takes a tile T_i as an input and predicts its corresponding performance IPC_i. Tile Generator uses this model to identify a suitable tilting strategy for the application in its runtime environment.

We first define a tile T_i. It consists of three parameters – t_i^1, t_i^2, t_i^3 as we focus on widely used three-dimensional tiling [151, 18, 156, 172]. Thus, a tile T_i can be represented as

$$T_i =< t_i^1, t_i^2, t_i^3 > \tag{4.1}$$

The online training data has five training tile parameters and their observed IPC. Tile Generator develops a linear model between these training tile parameters and their corresponding IPC. It applies a linear curve fitting method on these five data points. This model can be formalized in the following manner.

$$f(T_i) \Longrightarrow IPC_i \tag{4.2}$$

where IPC_i is the modeled IPC for the application.

Note that the model is obtained by applying a linear curve fitting method on just five data points. The overhead of generating a linear model with so few points is minimal. Also note that this model only captures current application runtime environment. Therefore this model needs to be updated if a new runtime environment is encountered.

4.3.2.2 Tile Shape Selection

This step uses the black box model represented by Equation 4.2 and predicts a suitable tile shape for the application in its current runtime environment. As shown in

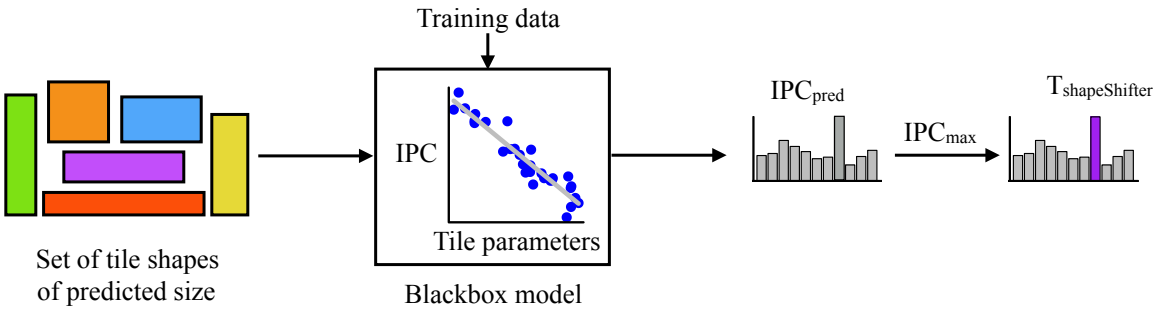


Figure 4.5: ShapeShifter tile shape selection - Tile Generator applies the blackbox model on a set of tiles and chooses the tile with the highest predicted IPC

Figure 4.5, Tile Generator applies the black box model on a large span of tile shapes consisting a mix of broad, narrow and intermediate tiles. Note that ShapeShifter is not executing these tiles, it is just applying the black box model to predict the IPC of each of the available tile shapes. The tile with the maximum predicted IPC is chosen as the tile for further execution. This tile is referred to as $T_{shapeShifter}$.

Algorithm 2 Tile shape selection

```
1: Input: TrainingData
2: Input: AvailSet
3: Output: ShapeShifterTile
4: function GetShapeShifterTile(TrainingData)
5:     bbModel = GenBBModel(TrainingData)
6:     predIPC = Apply(bbModel, AvailSet)
7:     ShapeShifterTile = maxIPC(predIPC)
8:     return ShapeShifterTile
9: end function
```

Algorithm 2 gives an overview of this step. This step can be represented in the following manner

$$T_{shapeShifter} : IPC_{shapeShifter} = \max_{i \in avail\_tiles} f(T_i) \tag{4.3}$$

Tile Generator invokes Companion thread to create a new version of application with $T_{shapeShifter}$ parameters. This version is used for execution from now on.

A key point to notice is that the black box model does not have to predict the

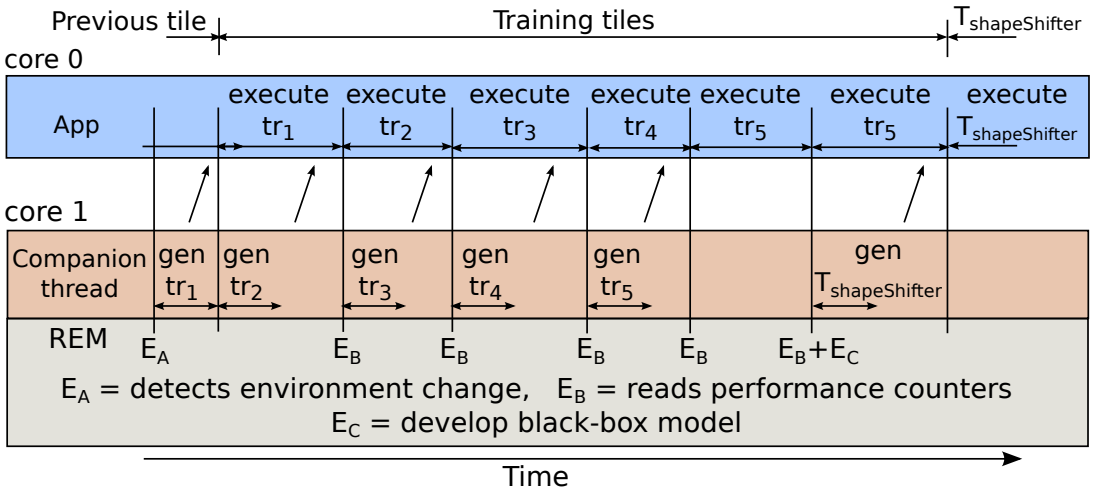


Figure 4.6: REM detects the environment change and wakes up companion threads to start training phase leading to creation of ShapeShifter tile

IPC of each tile accurately. Even ranking the tiles accurately is more than necessary for our purpose. Minimally, ShapeShifter should be able to pick up an acceptable tile when there is a large variation in the IPC of available tile shapes. We show in Section 5.4 that ShapeShifter black box model is highly accurate. It asserts that a simple linear model generated online is sufficient to identify suitable tile parameters across a wide range of runtime environments.

4.3.3 Monitored Execution Phase

REM continuously monitors the runtime environment and triggers online training in the presence of a tiling opportunity. For detecting changes in architectural policies and platform, REM periodically polls MSRs and other software visible registers. In order to detect the presence of a co-runner, ShapeShifter uses a simple technique of monitoring cache misses. Arrival of a co-runner typically increases cache miss rate. ShapeShifter assumes the presence of a software/hardware mechanism that provides an estimate of cache size that should be allocated to each co-runner. In the absence of such a mechanism, ShapeShifter assumes that the co-runners consumes half of the available cache capacity.

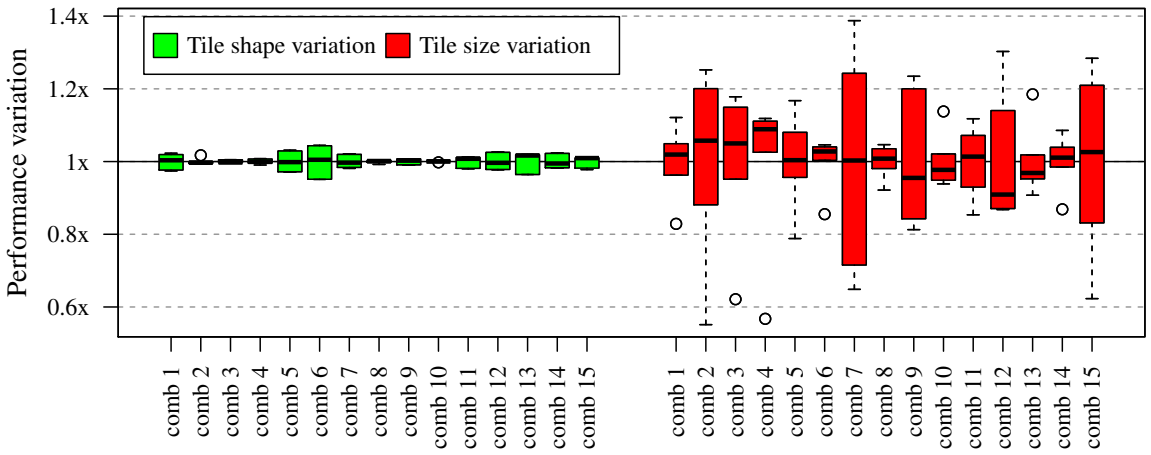


Figure 4.7: Interference caused by different tile shapes is similar whereas different tile sizes exert significantly different amount of cache pressure

In addition, ShapeShifter remembers the tile for a particular application and runtime environment. If the same runtime environment shows up later, ShapeShifter uses the stored tile for the application to avoid unnecessary training overhead.

The entire process is shown in Figure 4.6. In the figure, the application is executing on core 0 while companion threads, REM and Tile Generator are running on core 1. On detecting a change in the runtime environment (event E_A), the Tile Generator starts the training process and instructs Companion thread to generate the training tiles (tr_1-tr_5) and stitch them in the application code. The application runs these training tiles one-by-one while the REM keeps collecting cache and performance statistics for each training tile execution (event E_B). After online training is complete, Tile Generator uses the training data and predicts a suitable tile for the current runtime environment (event E_C). This tile is used for further execution.

In frequently changing environment, REM detects a change while the training is in process. In that case, ShapeShifter finishes the training and discards REM detection for a certain duration.

AMD Bulldozer	**Intel Haswell**	**Intel Atom**
Opteron 6272	Xeon E3-1240v3	Atom 330
16 cores	4 cores	2 cores
2.1 GHz	3.4 GHz	1.6 GHz
48K, 4-way, L1 (private) 2M, 16-way, L2 (shared) 12M, 128-way, L3 (shared) Individual way-locking on L2. Experiments use 16 -way/8-way and 4-way unlocked configurations	32K, 8-way, L1 (private) 256K, 8-way, L2 (private) 8M, 16-way, L3 (shared)	24K, 6-way, L1 (private) 512K, 16-way, L2 (shared)

Table 4.2: Platforms used in the evaluation

4.4 Loop Co-optimization

Applications are often co-run in datacenter operators to improve server utilization [112, 45]. Also, executing multiple programs are common on desktop and mobile platforms [54]. A universal tiling strategy needs to find suitable tile shape and size for all the co-running applications such that the interference between them is minimized. ShapeShifter provides a capability of capturing this interference and adjust tiles of multiple co-runners to their corresponding effective cache size. We refer to this feature as *co-optimization*.

A major hurdle in achieving effective co-optimization is the search space. Tiling for one application itself has a huge search space. Adding co-runners makes the problem intractable. Applying different tiles in one application creates different runtime environments for the co-running applications, and thus, optimizing tiling for one application can change the best tiling strategy for an already optimized application. This makes tiling for multiple co-runners simultaneously a challenging problem.

An insight that can enable a solution to this problem is that the interference caused by co-runners is largely a function of their tile size, that is, different tile shapes of the same size exert similar amount of cache interference. This is illustrated in Figure 4.7. In this experiment, we take 15 pairs of co-runners and study performance variation of the first application when (left) only tile shape of the second application is varied while keeping the tile size same and, (right) tile size of the second application is

varied. This shows that different tile shapes among a tile size result in similar amount of interference, while different tile sizes result in much larger performance variation. This insight gives us a strong foundation to solve the challenging problem of tiling for multiple co-runners.

Algorithm 3 Co-optimization

```
1: Input: Apps
2: function Co-Tiling(Apps)
3:     Initialize TileSize[Apps]
4:     for (app in Apps) do                                   ▷ Optimize size
5:         T_optSize = FindTileSize(app)
6:         DynComp(app, T_optSize)
7:         TileSize[app] = T_optSize
8:     end for
9:     for (app in Apps) do
10:        Ts = TileSize[app]
11:        trainingData = GetTrainingData(Ts)                 ▷ Algo 1
12:        T_shapeShifter = GetShapeShifterTile( trainingData)  ▷ Algo 2
13:        DynComp(app, T_shapeShifter)
14:    end for
15: end function
```

Algorithm 3 gives an overview of our co-optimization. ShapeShifter first identifies a suitable tile size for all the co-runners as described in Section 4.3.1.1. We refer to the tiles after this step as $T_{optsize}$, as the tiles have been optimized for size. Since the interference is dependent heavily on the tile size and does not change significantly with the tile shape, this step creates a stable runtime environment, whereafter the cache interference does not change significantly as the tile size changes. Therefore, ShapeShifter now optimizes the tile shape of all the co-runners one-by-one. Since the cache interference does not change with tile shapes, optimizing tile shape once for all the co-runners results in suitable tiling strategies. We observed that additional optimization on tile shapes resulted in marginal performance improvements. We evaluate co-optimization in Section 4.5.3.

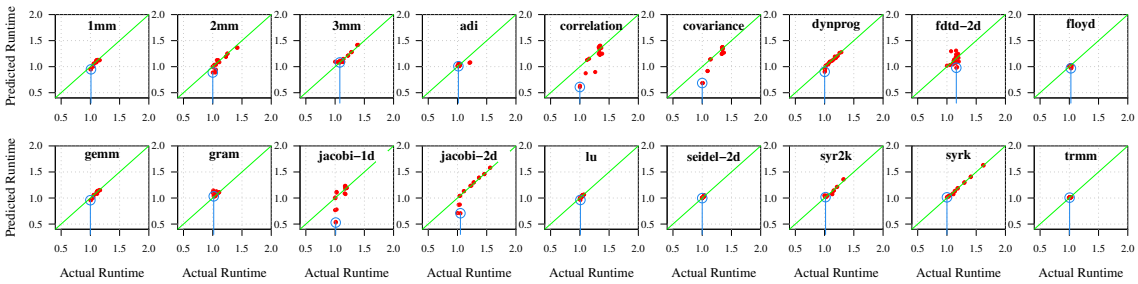


Figure 4.8: Actual runtime of applications vs. the runtime modeled by ShapeShifter's dynamic Tile Generator

4.5 Evaluation

4.5.1 Methodology

Applications. We evaluate ShapeShifter on the Polybench application suite [123, 82, 15, 104, 150, 101], a collection of linear algebra, stencil computation and data mining algorithms.

Implementation. We used Polly [61], a polyhedral optimizer tool that is integrated into LLVM [94] to perform tiling. We integrated Polly with protean code [96] to implement ShapeShifter. Polly performs cache tiling on LLVM intermediate representation while protean code provides the dynamic compilation capability.

Hardware Platforms. Our evaluation encompasses three design points with different microarchitectural and architectural configurations, as summarized in Table 4.2. These platforms are an AMD Bulldozer, an Intel Haswell and an Intel Atom. The AMD Bulldozer allows way-locking on its 16-way L2 cache, preventing a subset of ways in the cache from being accessed by any application. We consider three configurations of way-locked L2 in our evaluation: completely unlocked (all 16 ways are active), half locked (8 ways are available) and mostly locked (4 ways are available).

Baselines. Our baseline is the best performing tile on the largest cache across all the machines. We find this tile by statically running an exhaustive search space on the largest cache in our experimental setup. We term this baseline as *Static Best*

approach to tiling.

4.5.2 Tile Selection Accuracy

This section evaluates the black box modeling technique at the core of the tile selection algorithm. The goal of the model is to map tiling parameters to performance, thus allowing the Tile Generator to choose the tiling strategy with the best performance of the available tiles. For these experiments, we statically compile and perform a run of the application with a host of different tiling strategies, measuring the runtime of each.

The results of this experiment are presented in Figure 4.8, where each plot shows the modeled vs. actual runtime for a particular benchmark, normalized to the runtime of the fastest tiling strategy. Inside each plot, the position of a particular point on the x-axis gives the actual runtime for a single tiling, while its position on the y-axis gives the modeled runtime from the black box model. As a guide, each plot has a line at x=y to show where perfect predictions (modeled runtime equals actual runtime) would reside. Also in each figure is a circled point, showing the tiling strategy chosen by ShapeShifter's tiling selection algorithm, along with a line that illustrates the actual runtime of that point.

Some applications, such as dynprog, result in precise models where the actual and modeled runtimes track each other closely across tiling parameters. However, a precise model is far beyond what is necessary to select a high performance tiling. To make this more clear, we highlight covariance, where the modeled runtime of the tiling chosen by ShapeShifter is 70% of the actual runtime but the the tiling strategy is still the fastest from among the available options. Similarly in jacobi-2d, there are numerous tiling strategies offering similar high performance and ShapeShifter chooses one from among them. This stresses the idea that our models do not need to predict absolute performance precisely.

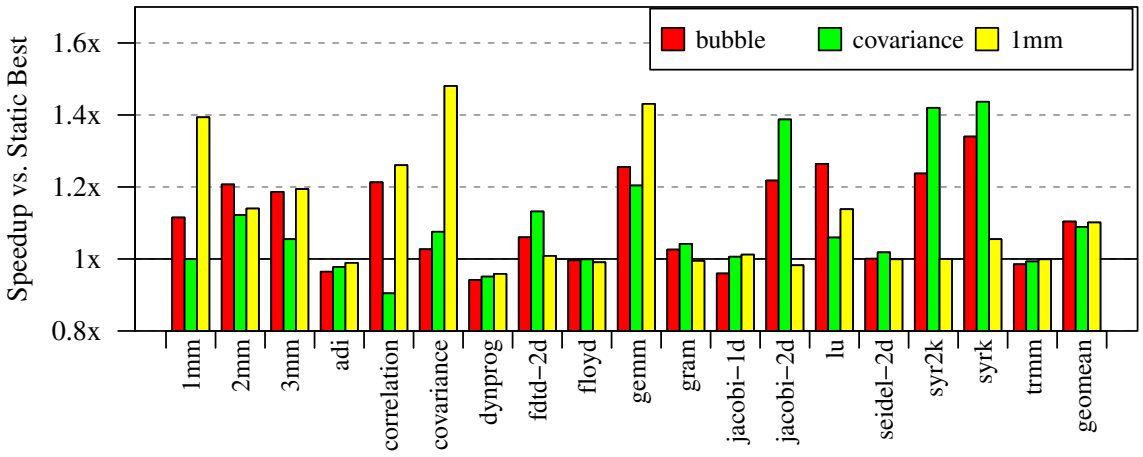


Figure 4.9: ShapeShifter adjusts the tiling strategy of an application in presence of diverse co-runners

4.5.3 Dynamism in Co-runners

In the era of multicore processors, the common case is that multiple applications are run together on a system at the same time. These co-running applications compete for shared resources, which includes caches. In this section, we evaluate how ShapeShifter adapts to runtime environment in the presence of co-runners. These experiments are run on the AMD Bulldozer platform, and we measure co-runners' performance as they run with ShapeShifter. We measure IPC for all the co-running applications and use it compute weighted speedup.

Stable Co-running Workloads. In this set, we conduct three experiments where ShapeShifter is used among 1, 2 or 4 applications.

In the first experiment, we evaluate how co-optimization performs if we limit it to optimize only one co-runner while the other co-runner tile remains unchanged. The results of this experiment are presented in Figure 4.9, which shows the performance improvement ShapeShifter-tiled application normalized to Static Best in the presence of three different co-runners: (1) the bubble, a microbenchmark designed to place pressure on a specific subset of the cache, which we configure to place pressure on half the L2 cache (1MB), (2) covariance from polybench, and (3) 1mm from polybench.

Workload 1	1mm, covariance, gram, lu
Workload 2	jacobi-2d, covariance, correlation, 1mm
Workload 3	correlation, syr2k, syrk, jacobi-2d
Workload 4	gram, lu, jacobi-2d, 1mm
Workload 5	2mm, syr2k, covariance, 1mm
Workload 6	jacobi-2d, 2mm, syrk, correlation
Workload 7	covariance, correlation, 1mm, 2mm
Workload 8	jacobi-2d, 1mm, correlation, covariance
Workload 9	syrk, syr2k, covariance, correlation
Workload 10	1mm, correlation, jacobi-2d, covariance

Table 4.3: Co-runner workloads of 4 applications

These results demonstrate that by re-tiling application code, ShapeShifter is able to achieve sizable speedups over Static Best, achieving performance improvements of up to 1.5× (covariance vs. bubble), and an average improvement of 1.1× on average.

In the second and third experiments, we demonstrate the capability of ShapeShifter co-optimization to accurately select tiling strategies in the presence of two and four co-running applications. The results of this experiment are present in Figure 4.10. Co-optimization works by first finding the right tile size for each co-runner, then optimizing each application tile shape one-by-one. The figure shows step-by-step speedup during this co-optimization process for two and four co-runners across 10 different workloads (Table 4.3). We observe that ShapeShifter co-optimization achieves performance improvement of up to 1.5× , with an average of 1.2× in both the scenarios. We also experimented with running ShapeShifter after all applications have been optimized once. We observed that the additional benefits were negligible, supporting the key insight that different tile shapes of same tile size does not have a large effect.

Dynamically Changing Workloads. In this experiment, we evaluate ShapeShifter co-optimization on a dynamically changing runtime environment that demonstrates how it adapts to the dynamism. In this experiment, at any given time there are 2 co-runners sharing a cache. These co-runners change with time along with the cache allocated to them as shown in Figure 6.17(a). ShapeShifter weighted speedup is com-

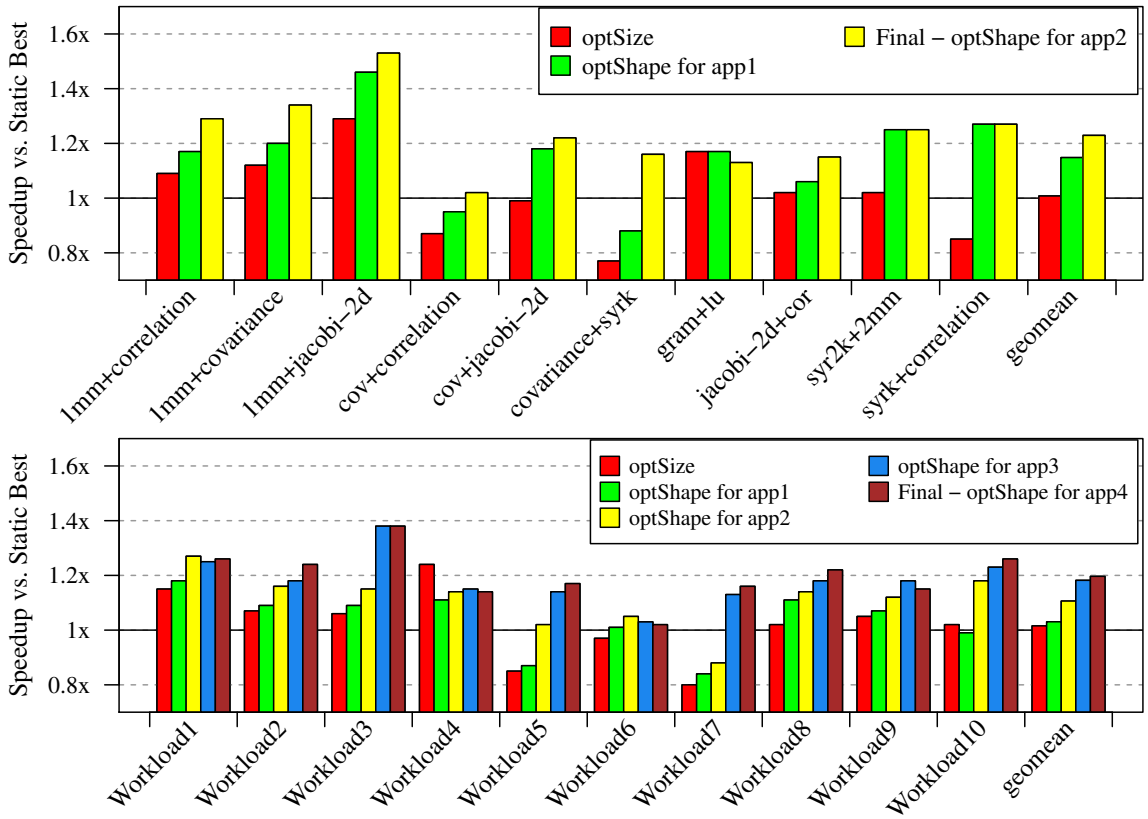


Figure 4.10: ShapeShifter co-optimization retiles multiple co-runners resulting in better cache usage

pared against the weighted speedup obtained by running co-runners aggressively tiled with Static Best strategy.

The result of this experiment are presented in Figure 6.17(b). We observe that ShapeShifter continuously adapts to changing runtime environment, finding a suitable tiling strategy for both the co-runners at different cache allocations. It results in significant speedup as compared to Static Best tiling strategy.

4.5.4 Microarchitectural Factors

In current systems, the architectural/microarchitectural parameters can change during the application execution. Here, we evaluate ShapeShifter on cache resizing and platform changes.

4.5.4.1 Cache Resizing

We begin by exposing applications to diverse situations in which different amount of cache are available. To conduct this experiment, we configure the way-locking feature on the AMD Bulldozer to leave either 8 or 4 ways open, then run the application with ShapeShifter to allow ShapeShifter to realize an aggressive tiling configuration on that microarchitectural configuration. The results of this experiment are presented in Figure 4.12, which is again normalized to application performance when the application is compiled to employ the Static Best tiling configuration.

We see large performance improvements over the Static Best strategy. When 8 ways are available to the application, ShapeShifter achieves performance improvements of 1.2× on average and up to 1.7×. This contrast becomes more stark when only 4 ways are available to the application, where an even larger gap exists between the optimal tiling strategies between the 4-way and 16-way configurations. In this case, ShapeShifter achieves a speedup of 1.4× over Static Best, with a maximum speedup of 2.4× on gemm.

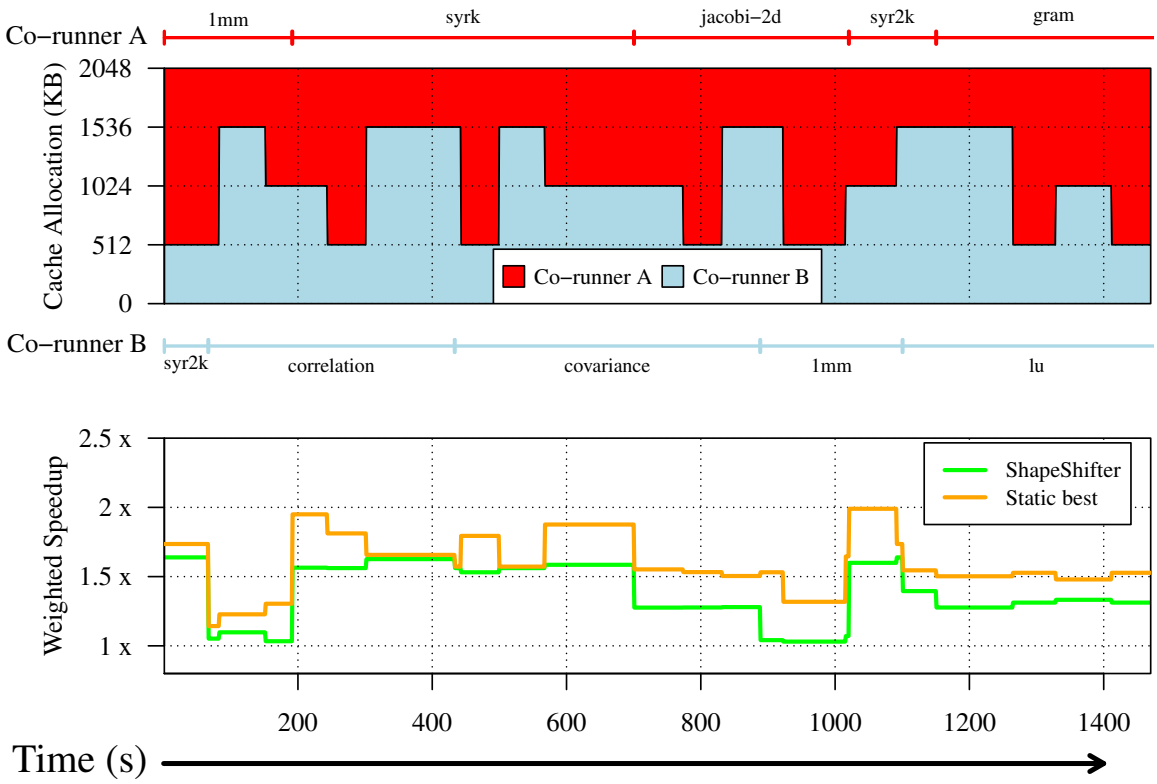


Figure 4.11: ShapeShifter co-optimization continuously adjusts co-runner tiles to changing runtime environment

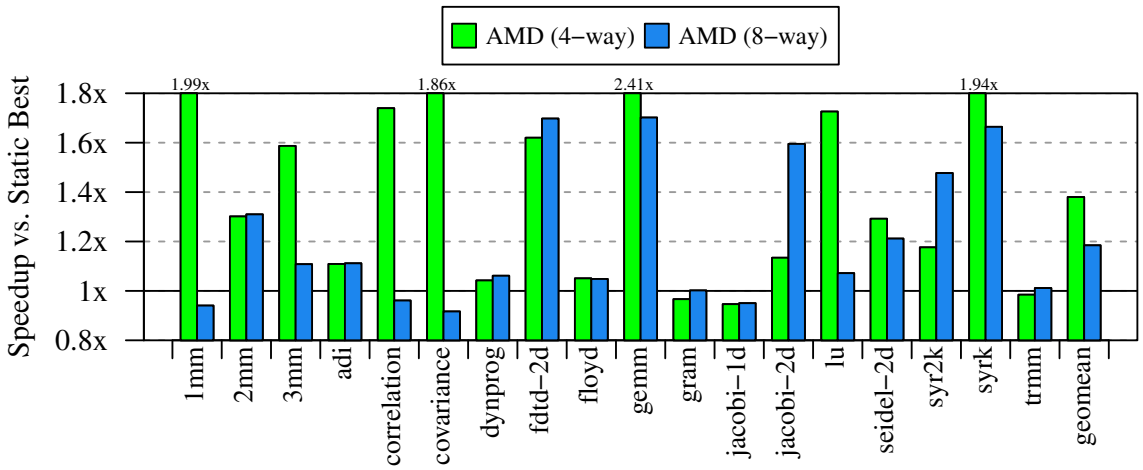


Figure 4.12: ShapeShifter demonstrates significant speedup by retiling for available cache size

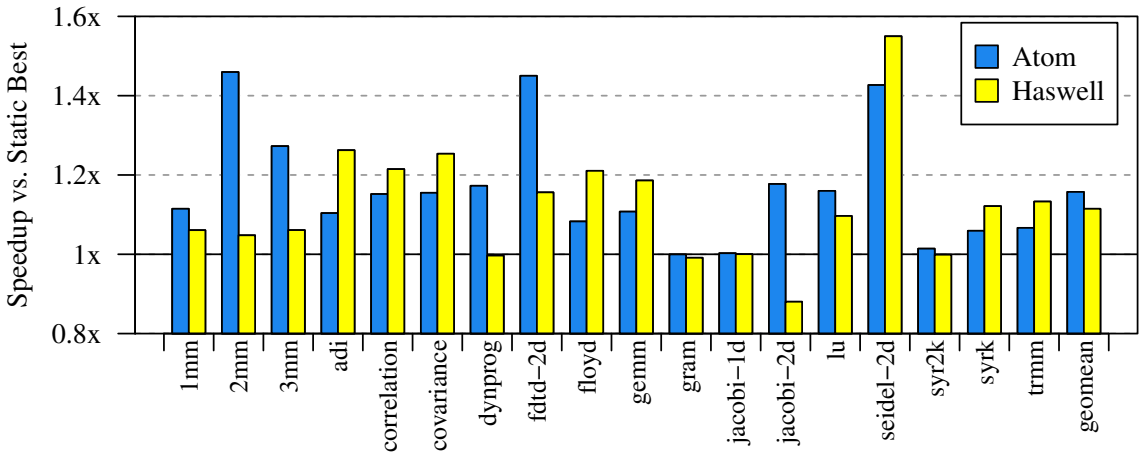


Figure 4.13: ShapeShifter shows sizable speedup by retiling for different microarchitectures

4.5.4.2 Microarchitectural Diversity

We next examine how ShapeShifter deals with significant microarchitectural diversity, applying it on applications running on Intel Haswell and Intel Atom platforms. Re-tiling occurs in ShapeShifter as the application begins execution, arriving at an aggressive tiling strategy for the specific microarchitecture. As a point of comparison, we also measure the performance when running applications that are tiled using the Static Best strategy on the Haswell and Atom.

The results of the experiment are presented in Figure 4.13, phrased as the performance improvement achieved by ShapeShifter over Static Best. These results demonstrate the effectiveness of ShapeShifter at developing aggressive tiling strategies across multiple microarchitectures, with a performance improvement of up to 1.5× when running seidel-2d on the Haswell system, an average performance improvement of 1.1× on Haswell, and a 1.1× average performance improvement on Atom.

4.5.5 Overhead Analysis

We now present the ShapeShifter runtime overhead. Companion threads use a small amount of compute and memory resources to dynamically compile new versions

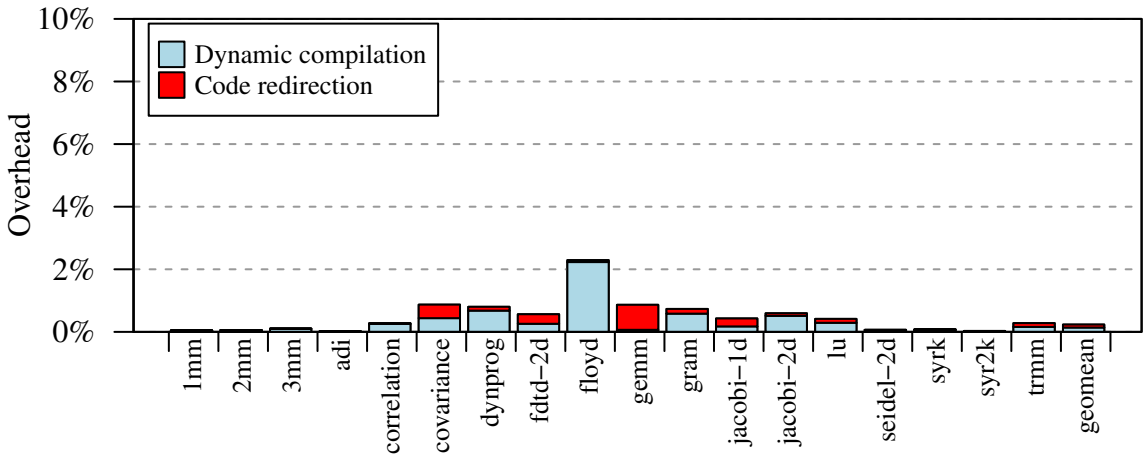


Figure 4.14: Runtime overhead of ShapeShifter dynamic compilation infrastructure

of code. This causes interference to the primary application which can in turn lead to slowdown. We term this slowdown due to interference as *dynamic compilation* overhead. In addition, whenever application is redirected to newly compiled tile, it suffers an I-cache warmup phase. We refer to this overhead as *code redirection* overhead. Finally, there is the online *training* overhead. In this section, we provide quantitative analysis of these overheads.

Dynamic compilation. In order to calculate just the dynamic compilation overhead, we design a stress test experiment where Companion thread continuously generates new tile variants without redirecting the application to the generated code. The associated overhead in this case is the worst case dynamic compilation overhead. Next, we allow application redirection to new tile variants. The difference between the former and latter experiment quantifies the code redirection overhead. We show these overheads in Figure 4.14. We observe that even in the stress testing, the overhead is minimal and less than 1% on average.

In terms of absolute numbers, we found that ShapeShifter takes 136 (336) ms on average with maximum of 430 (990) ms on Intel haswell (AMD Bulldozer) across our benchmark suite while the application is running on other core.

Training. As a part of the tile selection algorithm, ShapeShifter runs a handful of

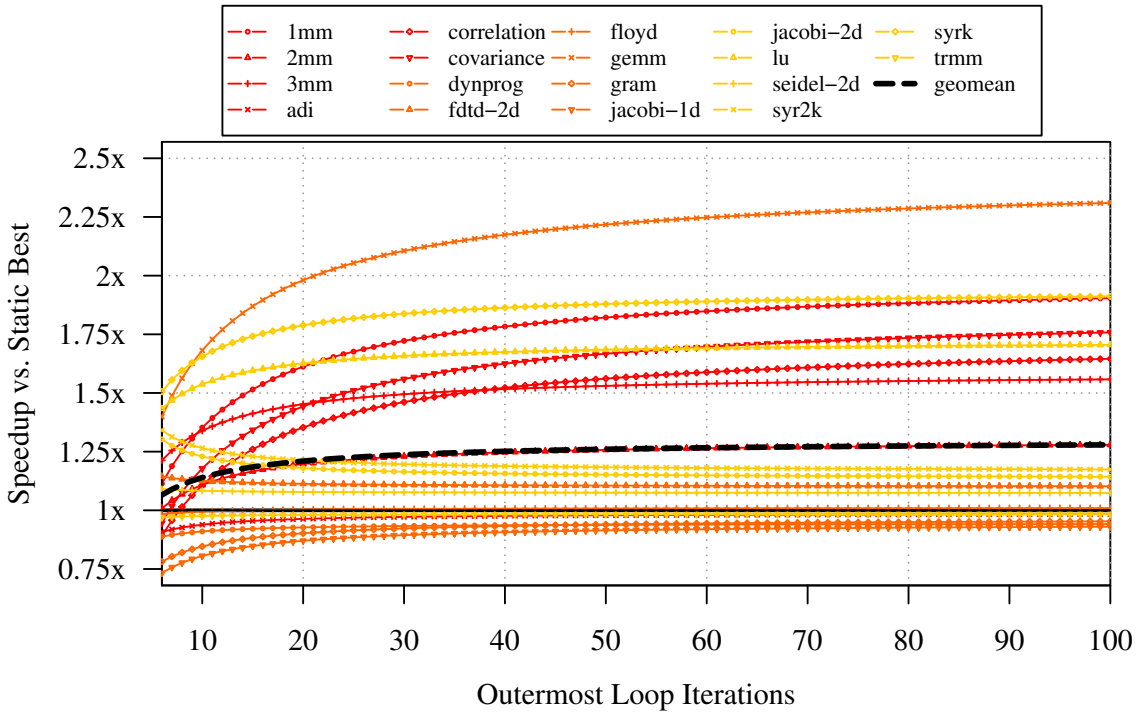


Figure 4.15: Performance benefits of ShapeShifter as a function of the how long the environment remains stable

diverse tiling strategies for training, which can be less performant than the final tile chosen. In this section, we weigh the overhead of that training against the benefit obtained by running an optimized tiling strategy. Our experimental setup is to run each application with ShapeShifter for a number of iterations in a stable environment (the AMD Bulldozer with 8 ways locked), measuring the performance of the application over time as the training and the final selected tiles are run.

The results are presented in Figure 4.15, which presents the performance of each application normalized to the Static Best approach (y-axis) over a number of iterations in the application's algorithm (x-axis). The results show that the performance improvement achievable by ShapeShifter depends on the amount of time the application stays in a stable environment. For example, immediately after training (5 iterations), the average performance improvement over Static Best across applications is 1.08×, while after just 20 iterations, substantially higher performance of 1.2×

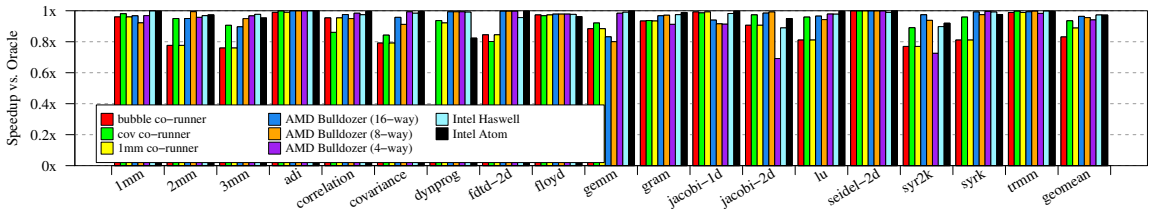


Figure 4.16: Comparison of ShapeShifter against a dynamic oracle, that chooses the ideal tiling strategy with no overhead; ShapeShifter achieves 93% of the performance of the dynamic oracle

is realized.

4.5.6 Comparison to Dynamic Oracle

Our final point of evaluation is to compare the performance achieved by ShapeShifter across a number of different runtime environments to the performance achievable by a dynamic oracle approach to tiling. To execute this experiment, we run each application in the prescribed environment using each of a large set of tiling strategies. Afterward, we choose the best-performing tiling strategy from among them and call this the measured performance of the dynamic oracle.

Figure 4.16 presents the results of this experiment. Across all applications and runtime environments, ShapeShifter achieves 93% of the dynamic oracle's performance on average (no worse than 72%). This demonstrates that ShapeShifter is effective in finding suitable tiling strategies across different runtime environments.

4.5.7 Comparison with Prior Work

We compare ShapeShifter against reactive tiling in this particular scenario of cache re-sizing in Figure 4.17. In this experiment, we generate a time schedule of changing cache sizes where the cache size is chosen randomly between 1x, 1/2x and 1/4x of the cache size during the application run. We observe that ShapeShifter achieves 10% speedup against reactive tiling as reactive tiling is limited by the set of tiles available to it at compile time.

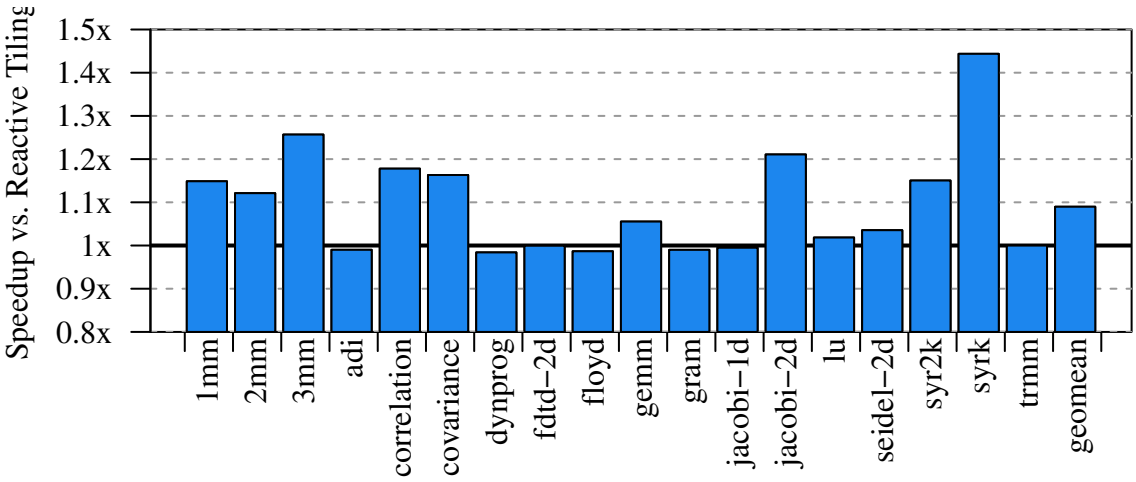


Figure 4.17: Improvement of ShapeShifter over Reactive tiling on a dynamic schedule for all applications

4.6 Summary

This paper introduces ShapeShifter, a dynamic compilation strategy that removes the risks of applying cache tiling by dynamically re-tiling running application code. ShapeShifter is designed to continuously monitor running applications and their runtime environments to find tiling opportunities and pinpoint near-optimal tile sizes. Upon finding such a tiling opportunity, ShapeShifter quickly generates an optimal tiling code for the application, then that code is seamlessly stitched into the running application with near-zero overhead. We evaluate ShapeShifter on real systems amidst three classes of runtime environment changes spanning different co-running applications, platforms, and dynamically shifting architectural resources. Our evaluation shows that ShapeShifter achieves sizable speedups across applications, averaging 1.1-1.4× across different runtime environments.

CHAPTER V

Architectural Support for Convolutional Neural Networks on Modern CPUs

For DNNs based on convolutional layers, GPUs and more specialized accelerators have gained significant traction as the hardware platforms of choice for running convolution computation [31, 128, 35, 33]. This *dual-device acceleration model* that our community has focused on involves adding the GPU or specialized accelerator to a conventional CPU platform, typically over a loosely-coupled interconnect such as PCIe, QPI or NVLink [8, 9, 3].

While the dual-device acceleration model offers a compelling set of performance and energy characteristics, it exposes system designers to two difficult challenges that pose a barrier to its widespread adoption, as observed in some previous research efforts [36]:

1. **Programming Models** – adding a second device to the system adds a second programming model, which can dramatically increase the difficulty of writing and maintaining production code. Recent work shows that the programming models for GPUs and accelerators are non-standard and unfamiliar to many programmers [144]; it is especially problematic for asynchronous accelerator programming models that are error prone and difficult to master [19].

2. **Hardware Complexity** – including a secondary computational device to facilitate acceleration introduces substantial additional complexity in system design, direct hardware purchase costs and maintenance and operational costs [20].

However, CPUs are an indispensable part of the design of any system, meaning they are a well understood part of conventional system design practices while offering the benefit of a seamless, familiar programming model and software stack. Moreover, CPU designs have a long history of incorporating hardware and ISA support for specialized domain-specific operations, evidenced by the near-universal support for cryptography, virtualization, security and multimedia operations in modern CPU offerings [10, 4, 11, 5, 6]. Thus, alongside designing dual-device acceleration platforms, it remains an important objective to design CPU hardware that can perform all the non-accelerable tasks for which CPUs are essential while also serving as an energy-efficient fabric for convolution layer computation.

This paper is the first to undertake a detailed characterization of the issues involved in improving CPU performance for convolution layers. We find first that scaling the read bandwidth of the physical register file (PRF) is one of the key constraints needed to deliver additional data to increasingly capable compute units. Second, we find that harnessing increasingly capable compute units requires crafting a solution that spans both hardware and software to take full advantage of the data reuse present in the core of the CNN computation. Building on this insight, we design Locality Extensions for Deep Learning (LEDL). LEDL is a technique that spans both hardware and software, consisting of a novel set of microarchitectural and ISA extensions to increase the computational capabilities of modern CPUs for CNNs. We present the design in detail, which in hardware includes a handful of architecturally visible remote registers that reside within the VFMA units in the CPU and a set of inter-VFMA links that allow data to be passed between units directly. In software, LEDL's automatic code generator, ACG, is carefully designed to generate code that

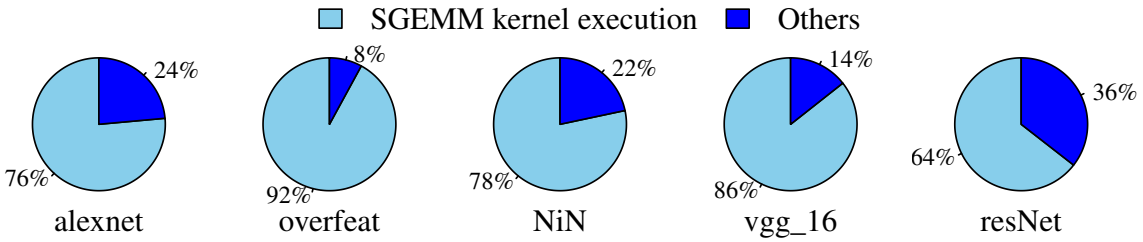


Figure 5.1: SGEMM kernels, on average, contribute to 78% of the total CNN execution time across 5 state-of-the-art CNNs

is robust to different microarchitectural implementations while taking full advantage of the reuse opportunities exhibited by convolution layer computation and aggressive prefetching mechanisms within modern CPUs.

Together, the hardware and software components of LEDL produce a platform design capable of providing substantial performance and energy improvements to convolution layer computation on CPUs. When extending an Intel Haswell server processor design with LEDL, we observe that across 5 state-of-the-art neural networks we achieve performance improvements that average 2× and energy-delay product improvements that average 2.7×.

5.1 Motivation

5.1.1 CNN Computation

Machine learning research has been increasingly focused in recent years on convolution neural networks (CNNs), as CNNs have been shown to outperform the alternatives across a number of different machine learning tasks [48, 153]. It is also evident that convolution layers are becoming more prominent as time goes on, specifically for tasks like object recognition, video analysis, drug discovery and natural language processing [75, 92, 155, 148, 89, 162, 170, 59, 87, 91]. These CNN-driven networks are becoming increasingly larger and deeper. For example, the Alexnet image recognition network had only 5 convolution layers [92], while the recently released ResNet can

have hundreds of convolution layers [75].

CNN Characterization. Beyond making up a large number of layers in modern CNNs, convolution layers consume a large fraction of the computational cycles in the total execution time, an observation that is in line with similar observations made by prior work [31, 34, 117].

Convolution layer computation has a number of implementations that have been explored in the literature and adopted in software packages [1, 35, 49, 97, 157]. We observe that there are two main classes of implementations that appear in high performance implementations: IM2COL + matrix multiplication (IM2COLMM) and winograd transform [97]. While each of these different implementations differs in how broadly applicable they are and in their performance characteristics, the computational kernel underlying all of them is the SGEMM calculation. The importance of SGEMM for CNN computation is illustrated in Figure 5.1, which shows the breakdown of time spent across SGEMM kernel and other computations for the IM2COLMM implementation, showing that a major portion of the time is spent in SGEMM kernels. We observe similar trend for Winograd algorithm as well. In addition, this SGEMM computation has also been used as the underlying implementation of other widely used DNN layers like fully connected and long short term memory layers [84] (we briefly discuss these layers in Section 5.4.6). Thus, the key to increasing the performance of CNN computation on CPUs is to achieve higher performance on the SGEMM calculation.

5.1.2 CPU Bottleneck Identification

The current trend of increasing raw computational capability of the CPUs is to simply scale the vector width of the SIMD units. For example, the Intel x86 SIMD vector width extensions have increased from 128-bits in SSE to 256-bits in AVX2 to 512-bits in AVX-512. However, the vector width scaling trend is unlikely to continue

for two reasons. First, scaling vector width beyond cache line width (512 bits) requires touching multiple cache lines per vector register load, possibly introducing complex microarchitectural workarounds to handle multiple variable-latency memory requests. Second, larger vector widths makes it increasingly difficult for the application developers or compilers to find SIMD parallelism amenable to such large vector widths, which is a difficult problem to solve even for current SIMD widths [17, 79, 116]. Due to these issues with vector width scaling, the only other obvious solution to increase raw CPU FLOPS is to add additional vector math units.

However, adding more vector math units is not enough to achieve higher CPU FLOPS. It is equally necessary to supply data to these vector math units every clock cycle to take advantage of this additional CPU compute. This leads to the question - *which memory-related microarchitectural parameters need to be adjusted to keep vector units busy?*

Bottleneck Analysis. To begin to answer this question, we study five such microarchitectural parameters - L1 cache bandwidth, L1 cache size, number of architectural registers and number of physical registers and Register bandwidth to identify which memory structure(s) should be focused on. In this study, we increase the number of vector math units from 2 (Haswell processor baseline) to 4 and measure the impact of doubling the value of these five parameters in simulation, both in isolation and in conjunction with each other, on the performance of SGEMM kernel. We present our findings in Figure 5.2. There are 2 key observations. First, increasing cache bandwidth and/or size alone (first 16 bars) does not improve SGEMM performance. The reason is that SGEMM employs aggressive register tiling, reusing the data in registers multiple times before going back to caches. Current L1 cache size and bandwidth are sufficient for this usage. Similarly, current Intel machines have enough physical registers for this usage. Second, we observe that both the **number of architectural registers and register bandwidth** have to be increased simultaneously (the right-

most eight bars) to achieve substantial speedup. Register bandwidth is necessary to supply the data to the vector math units every cycle. And, increasing architectural registers is necessary to achieve higher tile size, reusing data multiple times before bringing more in from cache.

5.1.3 Challenges

Energy Consumption. Conventional out-of-order cores use Physical Register File (PRF) for register renaming which helps in extracting more instruction level parallelism. PRF size has been increasing with every new CPU offering, currently set at 168 physical floating-point registers in Haswell processors. This PRF size is large enough to support SGEMM kernel, given we have enough software-visible architectural registers. Therefore, the deciding parameter to keep vector math units busy is PRF bandwidth.

To understand the impact of PRF bandwidth, it is necessary to understand how SGEMM works. All SGEMM kernel operations can be realized using Fused Multiply Add (FMA) instructions. Fortunately, in recent years CPU vendors have introduced vector fused multiply add (VFMA) units in the processor that can be leveraged by SGEMM computation. Each VFMA operation requires 3 vector register reads. Therefore, adding a VFMA unit requires extra three read ports in the PRF, introducing several challenges.

Firstly, the energy per access increases rapidly as the number of PRF read ports increases. Thus, the inclusion of additional read ports to feed a larger number of vector compute units rapidly increases the energy per PRF read, which can quickly turn the PRF a major contributor to the energy consumption of the CPU. Secondly, additional read ports increase the access latency to the PRF, where even a modest number of read ports can begin to constrain clock rate. For instance, a PRF with 14 read ports at 22nm technology node can meet a 2.4GHz clock rate, while a PRF with

15 ports cannot.

Therefore, it is clear that PRF reads are expensive and have to be kept to a minimum to keep the energy-hungry PRF in check. We observe that SGEMM kernel has high amount of data reuse, which if exploited wisely can result in significant reduction in PRF reads. For example, considering multiplication of matrices A and B, first element of A is multiplied to every element in the first row of matrix B, providing opportunity to cut the PRF reads for first element of matrix A. And similarly, first element of matrix B is multiplied to every element in the first column of matrix A. These opportunities for reuse could, alongside register tiling, be leveraged to substantially reduce the number of reads to the PRF.

Code Generation. Another challenge is generating code that can efficiently take advantage of the additional compute in the CPUs. Libraries such as MKL are aggressively tuned to current CPU specifications, and thus these libraries cannot be readily ported to new hardware configurations having more VFMA units without significant additional manual labor. Increasing the number of VFMA units requires handling data movements between the memory, registers and the VFMA units in an effective manner to keep the VFMA units busy. In addition, this interplay changes with the number of architectural registers and VFMA units in the processor, requiring an automatic code generation technique that is robust to different microarchitectural implementations while taking full advantage of reuse opportunities exhibited by SGEMM calculation.

5.2 Overview

This work focuses on devising a set of solutions to the aforementioned physical register file (PRF) energy and performance limitations. This section presents a sketch of the solution components spanning both hardware and software that allow a general purpose CPU design to overcome those limitations.

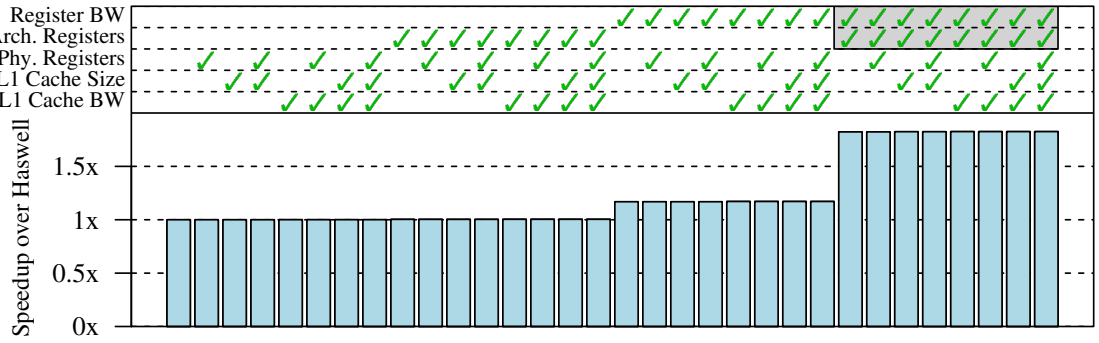


Figure 5.2: Performance impact of doubling five memory-related microarchitectural parameters, when VFMA units are increased to 4; Arch registers and Reg BW are the key factors

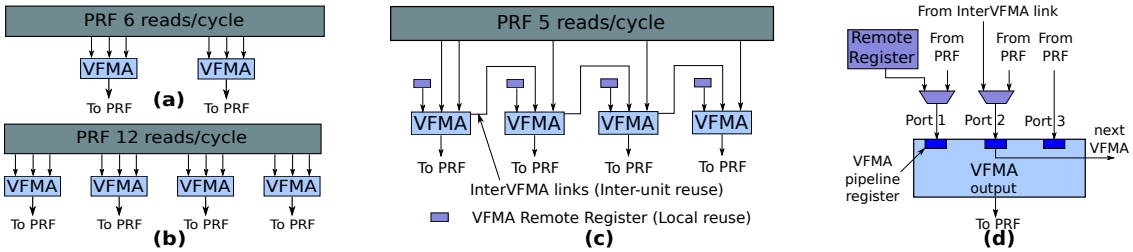


Figure 5.3: Architecture overview; (a) Haswell processor status with 2 VFMA units, (b) Straightforward extension to 4 VFMA units (c) LEDL introduces VFMA remote register and InterVFMA links, and (d) LEDL modifications to VFMA input ports

Hardware. Our solution, Locality Extensions for Deep Learning (LEDL), takes advantage of the substantial data reuse opportunities inherent in the SGEMM calculations to efficiently utilize the scarce PRF bandwidth available on the CPUs. LEDL centers around two key modifications in the CPU microarchitecture to reduce the burden on PRF. First, we add an *architecturally visible register*, VFMA remote register, embedded in each VFMA unit. Second, we add *low-cost unidirectional inter-VFMA links* between the VFMA units, that a VFMA unit can use to pass on the data to the connected VFMA unit. These microarchitectural modifications enable the programmer to reuse the data multiple times, both within and across the VFMA units, instead of reading from the PRF every time, effectively reducing the register reads per cycle while allowing to pack more VFMA units in the CPU.

Software. We introduce Automatic Code Generator (ACG) that automatically generates code for SGEMM calculations suitable for a given number of architectural registers and VFMA units, while maximizing the data reuse. ACG leverages two optimization strategies - *Register tiling and Prefetcher-friendly layout transformation* - to keep compute units busy. These optimization parameters depend on the number of architectural registers and VFMA units. ACG analytically finds a suitable set of optimization parameters that structure the computation in a manner necessary to achieve high data reuse not only within the PRF, but also within and across the VFMA units as facilitated by LEDL microarchitectural additions.

5.3 Design and Implementation

CNN applications have high compute and energy requirements. Improving performance of CNN applications on CPUs requires adding more VFMA units, while keeping the energy-hungry PRF in check. In this section, we present LEDL hardware and software implementation details designed to improve CPU energy efficiently for CNNs.

5.3.1 Hardware Design

5.3.1.1 Energy-Efficient PRF Usage

LEDL's goal is to reduce the burden on PRF, while being able to pack more compute in the CPUs. It utilizes the data reuse inherent in SGEMM calculations to reduce PRF reads, effectively reducing the PRF bandwidth and energy requirements. We achieve this energy-efficient usage of PRF by making minor modifications in the VFMA units.

Figure 6.5 gives an overview of the current state of the PRF and FMA units and our microarchitectural extensions. Figure 6.5(a) shows the status of current Intel Haswell processor design having 2 VFMA units connected to the PRF. Each VFMA unit requires 3 register operands from the PRF and writes 1 register in the PRF, requiring a total of 6 PRF reads per cycle for Intel Haswell. Figure 6.5(b) shows a straightforward extension of Intel Haswell architecture, having 4 VFMA units. This configuration requires PRF bandwidth of 12 register reads per cycle, incurring significantly high energy cost. SGEMM calculations have high data reuse opportunity which can be exploited to reduce the number of PRF reads per cycle substantially. To utilize this data reuse, we extend each VFMA unit to achieve temporal reuse within and across the VFMA units, as shown in Figure 6.5(c). First, each VFMA unit has an architecturally visible register, referred to as VFMA remote register capable of reusing a vector register input locally (at the same unit) across multiple operations. Second, the VFMA units are connected with unidirectional links, referred to as InterVFMA links, adding opportunity of inter-unit reuse across VFMAs.

Local Reuse - VFMA Remote Register. To reuse a data value locally, each VFMA unit is augmented with an architecturally visible register. This register is different from other architectural registers in that it is coupled with a particular VFMA unit. It can be written from the caches or from the other registers like other

architectural registers, but it cannot be written by the VFMA itself. It is used for storing an input value that can be reused multiple times, which would have otherwise come from PRF. Localizing the usage of the remote register to its VFMA unit, while also disallowing the VFMA to update it, results in little hardware overhead. VFMA remote register adds a capability of reducing the PRF bandwidth requirement by a maximum of one-third if the application data-reuse is efficiently utilized.

Inter-unit Reuse - InterVFMA links. Further, LEDL exposes inter-unit reuse capability in VFMA units by connecting them via a unidirectional link, as shown in Figure 6.5(c). The VFMA unit can obtain one of its operands from the InterVFMA link, instead of reading it from PRF. These links help in achieving inter-unit reuse, where an operand can be read just once from the PRF and then can be reused across VFMA units by using InterVFMA links. When coupled with VFMA remote registers, this further cuts down the PRF reads by around one-third by reusing the same value across different VFMA units. Similar to VFMA remote register, InterVFMA links transfer only the input data and do not support transfer of VFMA output to next VFMA input.

VFMA Input Ports. To take advantage of local and inter-unit reuse, we modify VFMA input port design so that it is flexible enough to take inputs from PRF, its Remote Register and InterVFMA link. Figure 6.5(d) shows the implementation details of VFMA ports. Typical VFMA unit has 3 input ports and 1 output port. In current Haswell architecture, each of these input ports is connected to the PRF. We modify input port 1 to take the input from either Remote register or PRF and input port 2 to obtain the input from either InterVFMA link or PRF. Input port 3 is kept unmodified, receiving the operand from the PRF. The VFMA output port is also kept unmodified, writing back the value in the PRF as usual.

5.3.1.2 Instruction Set Architecture

Here, we describe the ISA extensions required to utilize the microarchitectural data reuse capabilities exposed by LEDL. We use x86 operations to explain the workings of these ISA extensions, but the ideas can be applied to other ISAs as well.

Remote Register Instructions. VFMA Remote Registers are architecturally visible registers that can be written by conventional move operations, moving the data from memory or other architectural registers to the remote registers. From a programmer's perspective, these are new registers that are dedicated to the VFMA units. An example of move operation from memory to a VFMA remote register(`%vfma0reg`) is:

```
vmov 0(%rcx), %vfma0reg
```

where `vmov` instruction transfers a vector word from the memory to the VFMA0 Remote Register.

VFMA Instructions. Most of the our ISA extensions are restricted to VFMA instructions. These extensions provide the select signal for the multiplexers in the VFMA input ports shown in Figure 6.5(d), resulting in 4 categories of VFMA operations:

```
vfma <PRF>, <PRF>, <PRF>
vfma <RR>, <PRF>, <PRF>
vfma <PRF>, <LINK>, <PRF>
vfma <RR>, <LINK>, <PRF>
```

where `<RR>`, `<LINK>` and `<PRF>` denote that the operand is read from Remote Register, InterVFMA link and Physical Register File respectively. Note that the first category is the class of VFMA operations currently supported in a conventional microarchitecture, choosing all the operands from PRF.

VFMA and Group ID Tags. To facilitate precise instruction scheduling of VFMA instructions to take advantage of our design (discussed next in Section 5.3.1.3), we

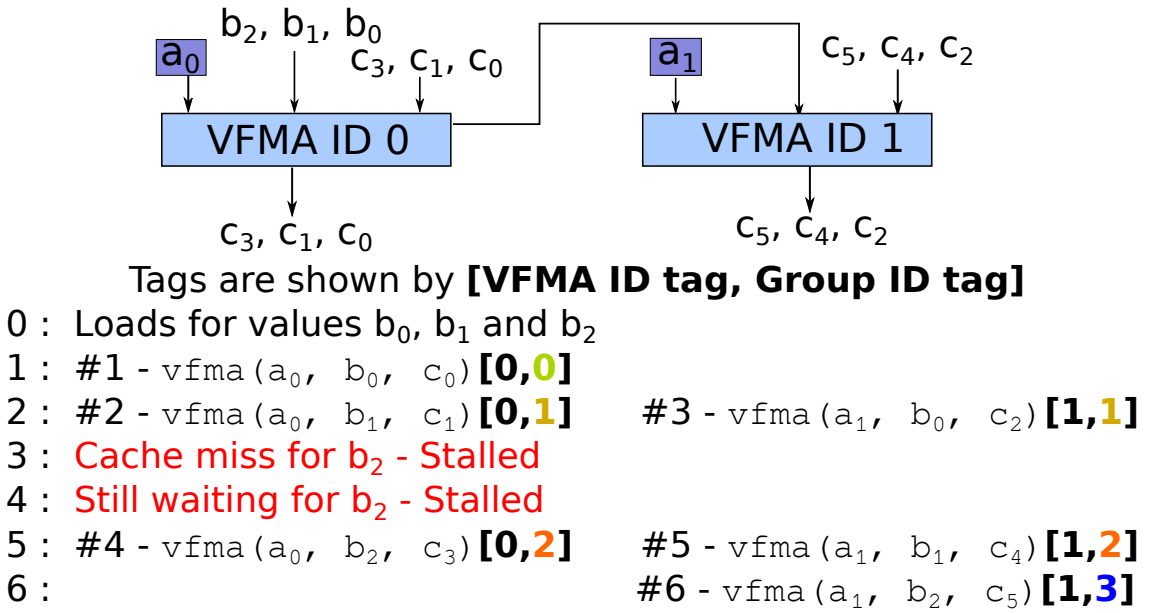


Figure 5.4: Example of leveraging VFMA ID and Group tags for instruction scheduling

add two fields to the VFMA opcode specification. First, each VFMA unit is assigned a tag that can be specified in each instruction. The instruction scheduler extracts this tag from the VFMA instruction opcode and then issues the instruction to the specific VFMA unit as identified by the tag. Second, a Group ID tag provides another layer of precise scheduling capability by informing the instruction scheduler about the instructions that should be issued simultaneously. All the VFMA instructions that have the same Group ID tag must be scheduled simultaneously. This means that every instruction, in the group of VFMA instructions with same Group ID tag, must have its operands ready before the whole group can be issued. This can be seen as introducing a degree of in-orderness to the execution of these groups of instructions, however we show in Section 5.4.2 that this effect has minimal impact on the application performance.

5.3.1.3 Instruction Scheduling

Dynamic Instruction schedulers in CPUs have the responsibility of scheduling ready-to-issue instructions to the functional units as they become available. In current Haswell processors, whenever the dynamic scheduler encounters a ready VFMA instruction, it schedules it on either of the VFMA units, whichever is available.

However, our extensions pose two challenges in the instruction scheduling - (a) The VFMA instructions need to be carefully scheduled to the relevant VFMA units. Since each Remote Register is local to its VFMA, and Inter FMA links are also unidirectional, the operations have to be orchestrated in a certain manner and cannot be scheduled randomly as done by the current dynamic scheduler. (b) In addition, some of the instructions in this pre-defined sequence might not be ready because one of their operands might be waiting for a cache miss to get resolved. To address these challenges, the instruction scheduler takes advantage of the two fields – VFMA ID tag and Group ID tag – included in the VFMA instruction specification.

Figure 5.4 shows the usage of these two tags, using an instruction sequence operating on 2 VFMA units connected via InterVFMA links. VFMA remote registers are already loaded with operands a_0 and a_1. The sequence of operands that go on the InterVFMA links is b_0, b_1 and b_2. The third operand, also the output register, comes from the register file and denoted by c_i. The figure shows the instruction sequence where each instruction has the associated tags in the square brackets [VFMA ID tag, Group ID tag]. In cycles 1 and 2, the VFMA ID tag directs the scheduling of instruction in the corresponding VFMA units. Also Cycle 1 and Cycle 2 have different Group IDs, forcing the instruction scheduler to follow the sequence. Cycle 3 shows an event where operand b_2 is unavailable due to a cache miss. Since instructions 4 and 5 share the same Group ID tag, even though instruction 5 is ready to be issued, instruction scheduler delays its issue until Cycle 5, when instruction 4 operand b_2 is also available. Group ID and VFMA ID tag, therefore, help in achieving precise instruc-

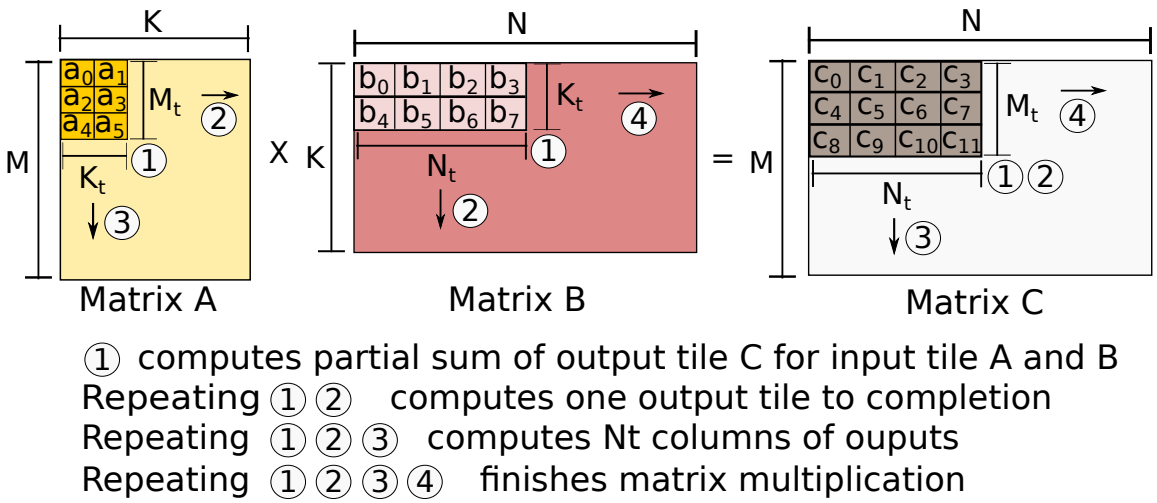


Figure 5.5: Register tiling steps performed by ACG

```
(a)
1  // The output tile is kept in registers - c0, c11
2  vmov b0, r0; vmov b1, r1; vmov b2, r2; vmov b3, r3 // Load tile B
3
4  vbroadcast a0, r4  // Read element from tile A
5  // Calculate partial sum for the first row of output tile
6  vfma r0, r4, c0; vfma r1, r4, c1; vfma r2, r4, c2; vfma r3, r4, c3
7
8  // Perform the same compuation for next output rows
9  vbroadcast a2, r4
10 vfma r0, r4, c4; vfma r1, r4, c5; vfma r2, r4, c6; vfma r3, r4, c7
11 vbroadcast a4, r4
12 vfma r0, r4, c8; vfma r1, r4, c9; vfma r2, r4, c10; vfma r3, r4, c11
13 // Repeat line 4-12 for next row of tile B and next colum of tile A (unroll Kt times)
```

```
(b)
1  // The output tile is kept in registers - c0, c11
2  vmov b0, rr0; vmov b1, rr1; vmov b2, rr2; vmov b3, rr3 // Load tile B
3
4  vbroadcast a0, r4; vbroadcast a2, r5; vbroadcast a4, r6
5
6  // Compute partial sums (Column is VFMA ID tag, row inst have same group ID tag)
7  vfma rr0, r4, c0;
8  vfma rr0, r5, c4; vfma rr1, Link, c1;
9  vfma rr0, r6, c8; vfma rr1, Link, c5; vfma rr2, Link, c2;
10                   vfma rr1, Link, c9; vfma rr2, Link, c6; vfma rr3, Link, c3;
11                                       vfma rr2, Link, c10;vfma rr3, Link, c7;
12                                                           vfma rr3, Link, c11;
13 // Repeat the same steps for next row of tile B and next colum of tile A
```

Figure 5.6: Code generation template for the partial sum output tile calculation for (a) non-LEDL and (b) LEDL hardware

tion scheduling that is necessary to utilize the local and inter-unit reuse capabilities exposed by LEDL.

5.3.2 Code Generation

Increasing VFMA units in the CPU requires an automatic code generator that can generate the code as per the availability of hardware resources, while also maximizing the heavy data reuse exhibited in the SGEMM calculations. Our code generator, ACG, leverages two optimization strategies - Register Tiling and Prefetcher-friendly layout transformation - to maximize data reuse and keep VFMA units busy. Using these optimizations, ACG structures the computation in a manner, where data can be reused within and across the VFMA units. ACG, then, maps the computation to LEDL using the ISA extensions described in Section 5.3.1.2.

5.3.2.1 Register Tiling

SGEMM kernel calls have high data reuse, providing opportunities of achieving high compute to memory ratio. To take advantage of this reuse, it is necessary to perform aggressive vector register tiling in the CPUs. We show later in Section 5.4.8 that by utilizing the registers efficiently, we can achieve upto 5× performance improvements as compared to a software that underutilizes the registers.

The details of our register tiling approach are illustrated in Figure 5.5, showing the steps involved in applying register tiling when multiplying input matrices A and B to produce output matrix C. The tiling is performed for both input and output matrices. As shown in the figure, the input A tile size if $M_t \times K_t$, and the input B tile size is $K_t \times N_t$, resulting in an output tile size of $M_t \times N_t$. The output tile holds the partial sum for the multiplication of A and B input tiles. Structuring SGEMM calculations in this manner not only exposes data reuse in PRF, but also within and across VFMA units, where LEDL can be leveraged to achieve better energy characteristics.

We show the details of the partial output calculation for non-LEDL hardware in ① in Figure 5.5, while the corresponding code template is shown in Figure 6.7(a),where the tiling parameters – M_t, N_t, K_t – are set at (4,24,2). Firstly, first row of the input B tile (N_t elements) is read from the memory into the registers (line 2). These values are reused before moving on to the next row. Now, elements from the first column of the input tile A are read one-by-one and used to compute the partial sums for the first row of output (line 4 - line 12). Note that element A is a scalar, which has to be replicated by vector length (shown as `broadcast` instruction in line 4), as the same value is multiplied to each element in each vector of the current row of input tile B. Once all the elements in the column of A are used, we move to second column of tile A and second row of tile B. This essentially translates into unrolling the loop by K_t times.

Once this partial sum calculation finishes, there are several options to choose

from. We observed that computing an output tile to completion results in the best performance, as it achieves maximum reuse possible for the output matrix. We achieve this by moving the tile horizontally in matrix A and vertically in matrix B, shown in the figure by ②, resulting in the completion of the output tile of elements $M_t \times N_t$. We then move the output tile vertically down shown by ③. Repeating ①, ② and ③ results in the completion of $M \times N_t$ output elements. Finally, we move to the next column, as shown by ④. Repeating ①, ②, ③ and ④ results in the completion of matrix multiplication.

While the underlying basics for performing register tiling using LEDL features remain same, the implementation details change slightly. The corresponding template is shown in Figure 6.7(b) which can be understood in conjunction with Figure 5.7 showing the values that are used within (local reuse) and the values that are used across the VFMA units (inter-unit reuse). The row elements of input tile B are brought into the VFMA remote registers (shown by `rr` in line 2), reusing these operands locally. All the column elements of input tile A are read into the registers before the actual computation starts (line 4). The values of these registers is now passed one by one to the first VFMA register which then transfers the value to the next units using InterVFMA links, enabling inter-unit reuse. While hoisting all the input tile reads to the start increases the register pressure, it results in better performance as it hides the memory latency to large extent.

Identifying Suitable Tiling Parameters. An objective of our code generation step is to find suitable tiling parameters that fit the hardware specifications, while also maximizing the data reuse opportunities. Analyzing the aforementioned template, we can easily find the relationship between the tiling parameters and the number of architectural registers. In addition, we can also calculate compute-to-memory-access ratio (CMAR) which captures data reuse at the PRF. ACG, using these relationships, generates a software variant by choosing an efficient set of tiling parameters that

maximizes data reuse while fitting in available architectural register count.

As we can see from the template, for the software that does not use LEDL capabilities, ① requires 1 register for input tile A, N_t/VL registers for input tile B and $M_t * N_t/VL$ registers for input tile C, where VL refers to the Vector Length; the number of floating point elements that can fit into a vector. For compute-to-memory-access ration (CMAR), the template performs $M_t * N_t/VL$ VFMA operations for every 1 memory read from input tile A and N_t/VL memory reads from input tile B. Therefore, the resulting relationship between tiling parameters and register tiling and CMAR is

$$Arch\ Registers = 1 + N_t/VL + M_t * N_t/VL \tag{5.1}$$

$$CMAR = (M_t * N_t/VL)/(1 + N_t/VL) \tag{5.2}$$

Similarly, the relationships when we leverage LEDL capabilities are

$$Arch\ Registers = M_t + N_t/VL + M_t * N_t/VL \tag{5.3}$$

$$CMAR = (M_t * N_t/VL)/(M_t + N_t/VL) \tag{5.4}$$

Depending on whether the HW supports LEDL, ACG chooses the relevant equations and picks the tile parameters that has the highest compute-to-memory ratio, while also fitting inside the available architectural register file size.

5.3.2.2 Prefetcher-friendly Layout Transformation

We observe that for many convolution layers, even after applying aggressive register tiling, the generated code variants still have low VFMA utilization, sometimes as low as 50%. Upon further investigation, we find that CPU is heavily stalled on cache misses, even though the memory access pattern seems to be predictable for the cache prefetchers. The reason for this slowdown is that the prefetchers are not allowed to

prefetch beyond page boundaries. In convolution layers, the matrices are typically large resulting in stride larger than a page boundary when the the data access pattern jumps to next row of the matrix.

To solve this issue, before starting any SGEMM computations, ACG performs a prefetcher-friendly layout transformation on the input matrices A and B, so that the memory access pattern becomes a continuous back-to-back sequence during the compute part. The overhead of performing this transformation (in the order of $O(M*K+K*N)$) typically gets amortized because of an order of magnitude higher number of FMA operations (in the order of $O(M*N*K)$), where the transformed data is reused multiple times. With this layout transformation, the prefetchers work very efficiently bringing most of the data in L1 caches before it is actually required, resulting in higher compute utilization. ACG uses the tiling parameters to generate the code for layout transformation. The transformation code is same irrespective of whether the code is utilizing LEDL reuse features. Figure 5.8 shows this transformation for input matrices A and B for the example discussed in Section 5.3.2.1. The transformation can be viewed as flattening the 2-dimension matrix into a 1-dimensional matrix, such that every next access is located contiguously in this flattened array.

Interleaving Transformation and Compute. To further reduce the cost of layout transformation, ACG interleaves some portion of compute with the layout transformation. Since layout transformation typically stalls on the memory, the interleaving utilizes the unused VFMA units to complete a small portion of SGEMM calculation in parallel. This technique is particularly useful for the cases where the amount of computation in the SGEMM computation is smaller, where the impact of hiding the overhead of the transformation becomes more visible.

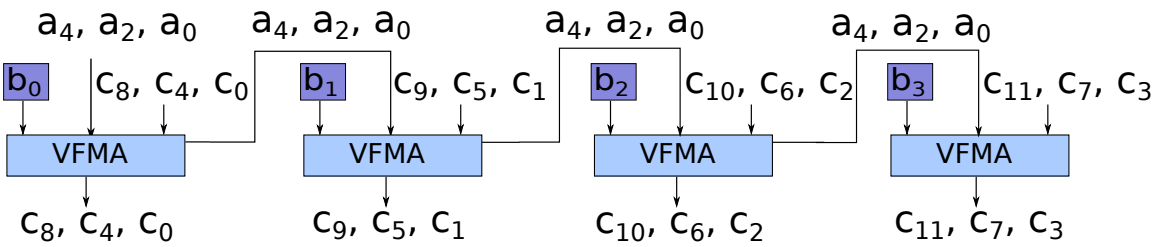


Figure 5.7: Computation and data movement for the LEDL code

5.4 Evaluation

5.4.1 Methodology

Applications. We evaluate our hardware and software mechanisms on 5 state-of-the-art CNN applications – Alexnet, Overfeat, VGG_16, NiN and ResNet [92, 146, 149, 106, 75]. These are medium to large CNNs, presenting a large variation in convolution layer shapes and sizes. The number of convolution layers in the five CNN applications are 5, 5, 13, 12 and 50 respectively. Additionally, we evaluate our hardware on a variety of other widely used DNN layers like Fully Connected and Long short-term memory layers (a type of Recurrent layer). The configuration of these networks is detailed in Section 5.4.6.

Performance and Energy Measurement. We use Snipersim [27] to evaluate the performance impact of LEDL hardware and software mechanisms. We have augmented the Snipersim infrastructure to simulate the vector instruction extensions described in Section 5.3.1.2, along with VFMA Remote Register and InterVFMA link implementations. We took efforts to ensure that Snipersim achieved similar performance statistics in simulation to the characteristics observed on real Haswell processors for the Intel MKL and ACG generated software variants. Our experiments use McPAT infrastructure [103], extended to include the techniques described by Sam et al. [166], to model energy and area consumption. The energy and area measurements used throughout the evaluation include core and all three levels of caches.

Baseline and Hardware Configurations. Our baseline, where not stated otherwise, is derived from a currently available Intel Haswell server processor design whose configuration details are described in Table 5.1. Physical floating point registers in our designs are fixed to 168, similar to the Haswell baseline. We increase the number of architectural registers to 96, unless otherwise specified. We never observed structural hazards due to unavailability of physical registers in our experiments.

We study the impact of local and inter-unit reuse by evaluating across three supported modes of VFMAs:

- NR (No Reuse): The VFMA unit reads all 3 register operands from PRF, requiring 3 PRF reads per cycle.
- LR (Local Reuse): The VFMA reads one operand from its Remote Register and other two from the PRF, taking advantage of local reuse, requiring 2 PRF reads per cycle.
- FR (Full Reuse): The VFMA reads one operand from its Remote Register, one from its InterVFMA link and one from the PRF, utilizing both local and inter-unit reuse, requiring 1 PRF read per cycle.

Table 5.2 lists the hardware design points that we use for our evaluation. We observe that for 2, 3 and 4 VFMAs, the PRF can have enough read ports to support NR mode. However, 5 and 6 VFMAs require 15 and 18 PRF read ports, at which point PRF cannot meet the timing constraints. Using VFMAs in LR and FR modes does not require 18 read ports. Therefore, we use a hybrid design for 5 and 6 VFMAs, where the number of PRF read ports are kept to 12 (2 per VFMA). Unless otherwise specified, we use these hardware design points for evaluation.

CNN Implementations. We evaluate the efficiency of our software-hardware mechanism on two CNN implementations: IM2COL+MatMult (shorthanded as IM2COLMM going forward) and Winograd. Our evaluation focuses mostly on IM2COLMM, while we focus on Winograd specifically in Section 5.4.6.

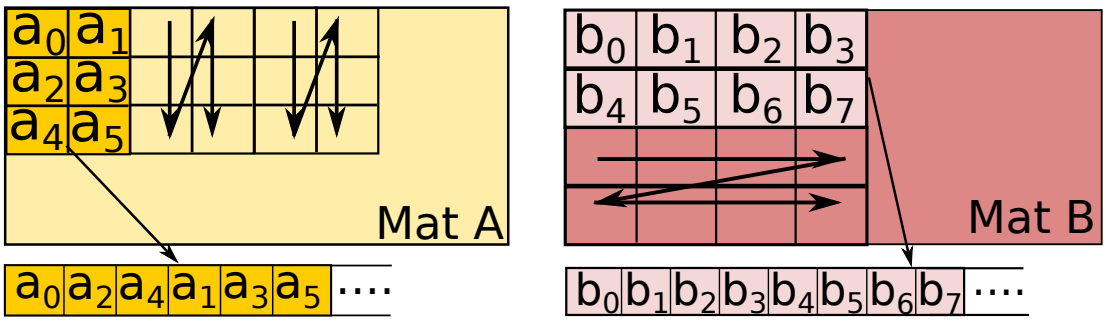


Figure 5.8: ACG's Prefetcher-friendly layout transformation

Processor	8-wide OoO core, 2.4 GHz 192-entry ROB, 72-entry load queue
Private L1 cache	32 KB, 8-way, 2-cycle, 64 B block
Private L2 cache	256 KB, 8-way, 5-cycle, 64 B block
Shared LLC	8 MB, 16-way, 12-cycle, 64 B block
Main memory	1 GB, 65 ns latency
L1, L2 and LLC prefetcher	Line prefetcher

Table 5.1: Baseline hardware configuration, modeled after an Intel Haswell server configuration

5.4.2 Performance and Energy Improvements

In this section, we examine the characteristics of LEDL to understand the tradeoffs the different hardware design points offer in terms of performance and energy usage.

Energy Delay Product. In the first experiment, we use ACG to generate software for each convolution layer in the CNNs of our application suite. We then measure the energy consumption of each layer for our hardware designs points and each VFMA mode. We accumulate the energy for each convolution layer per DNN and measure the Energy delay product (EDP). The findings of this experiment are presented in Figure 5.9. The figure shows EDP improvement for the best FMA mode for each hardware design point, over the Intel Haswell baseline.

We observe that increasing the number of VFMA units results in significant EDP improvements over the Haswell baseline. LEDL extensions substantially reduce the number of PRF reads, resulting in average EDP improvements of 2.0×, 2.5× and 2.7× with FR mode on 4, 5 and 6 VFMA units. For lower number of VFMA units (2 and 3), NR mode achieves better EDP due to better tile characteristics.

Design point name	VFMAs	PRF read and write ports
2-VFMA (Baseline)	2	6 read and 2 write
3-VFMA	3	9 read and 3 write
4-VFMA	4	12 read and 4 write
5-VFMA-Hybrid	5	12 read and 5 write (NR not supported)
6-VFMA-Hybrid	6	12 read and 6 write (NR not supported)

Table 5.2: Hardware design points

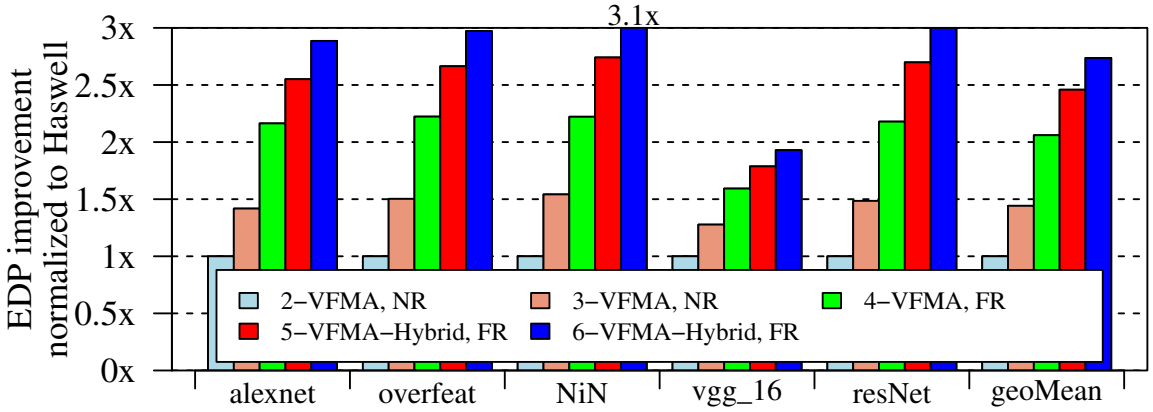


Figure 5.9: EDP improvement of increasing the VFMA units for end to end total convolution runtime.

Performance. Next, we perform the same experiment and measure the performance of each layer for our hardware designs points and each VFMA mode, giving us the total convolution runtime. The findings of this experiment are presented in Figure 5.10. The figure shows the speedup of the best reuse mode for each hardware design point against the Haswell baseline.

We observe that adding VFMA units results in geometric mean speedup of 1.4×, 1.7× for 3 and 4 VFMA units for NR mode. Further, PRF cannot meet latency constraints for supporting NR mode when the number of VFMA units are increased to 5 and 6. Here, LEDL's reuse capabilities reduce the PRF bandwidth requirements, resulting in hybrid designs that improve compute capacity, achieving a performance speedup of 2.0× and 2.1× for FR mode on 5 and 6 VFMA units respectively.

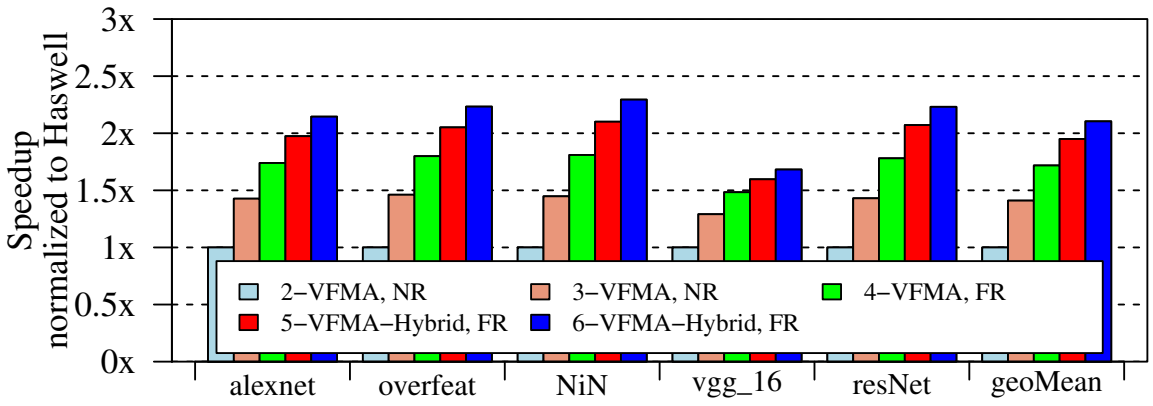


Figure 5.10: Performance improvements of increasing the VFMA units for end to end total convolution runtime

5.4.3 Impact of FMA modes

The LEDL-enabled FMA modes – LR and FR – reduce the number of PRF reads by taking advantage of local and inter-unit reuse, resulting in better energy consumption characteristics. In this section, we study the energy effect of these FMA modes by measuring the energy and execution time of each CNN layer in our application suite, giving us the total EDP of accumulated CNN layer execution. This experiment is performed for all hardware designs on the 3 VFMA modes. We present the findings of this experiment in Figures 5.11 and 5.12.

First, we show the impact of FMA modes on 2, 3 and 4 VFMA units in Figure 5.11. The figure shows EDP improvement of LEDL-enabled LR and FR modes normalized to the currently-supported NR mode for 2, 3 and 4 VFMA units. We observe that for 2 and 3 VFMA units, the LR and FR reuse modes result in minimal improvement. This is because the tile characteristics of code variant for LR and FR modes have higher energy consumption compared to NR mode. In this experiment, we also observe that PRF power is 10% of the total power at 2 VFMA units, but increases to 18% for 4 VFMA units. Due to this high increase in PRF power, we observe that at 4 VFMA units, LEDL starts achieving better EDP characteristics than NR mode. On average, LR and FR achieve EDP improvements of 8% and 10% for 4 VFMA

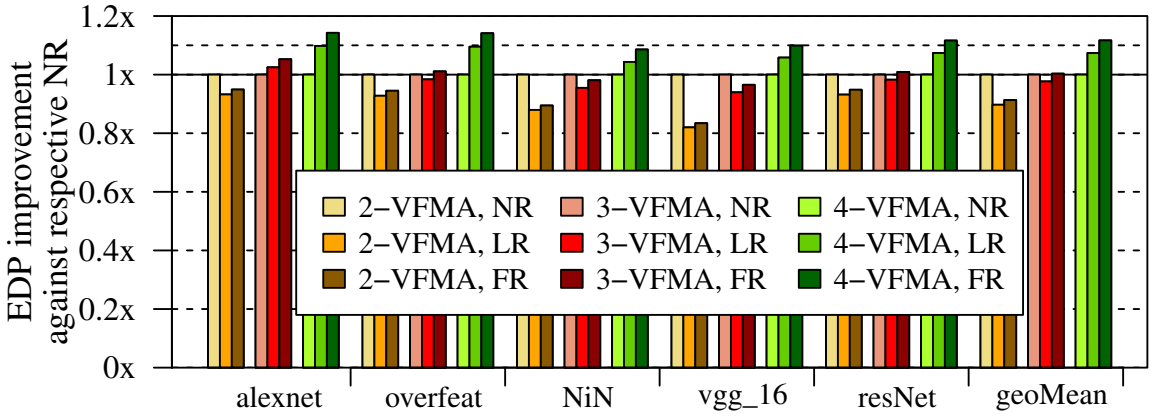


Figure 5.11: LEDL-enabled FMA modes comparison for 2, 3 and 4 VFMA units; LEDL-enabled modes achieve better EDP design point at 4 VFMA units

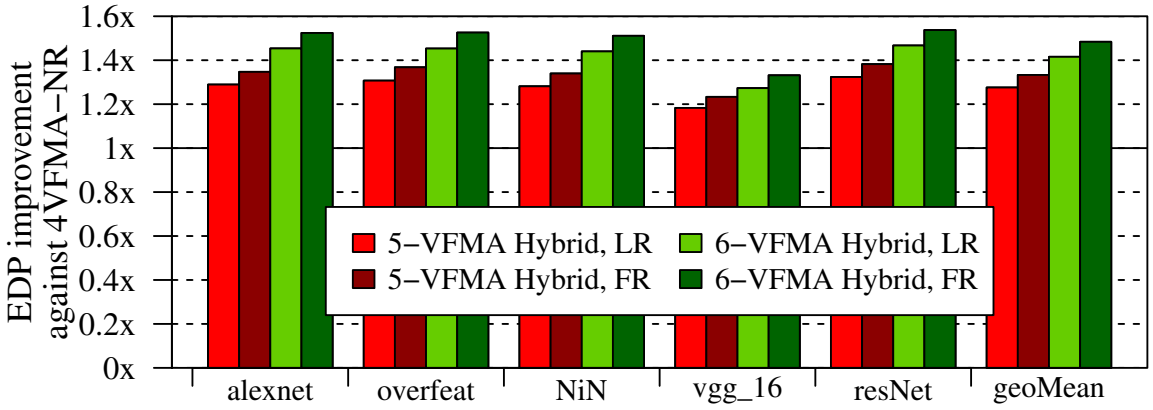


Figure 5.12: LEDL-enabled FMA modes comparison for 5 and 6 VFMA units. NR mode is not supported as PRF latency constraints cannot be met

units, respectively, compared to NR mode.

However, beyond 4 VFMA units, NR mode is not viable because the PRF latency constraints could no longer be met. LEDL, on the other hand, relaxes PRF bandwidth requirements, packing more VFMA units while keeping PRF latency in check. We therefore compare the EDP characteristics of LEDL-enabled modes on 5 and 6 VFMA units to 4 VFMA units with NR mode, the best hardware design point currently supported by NR mode. This comparison in shown in Figure 5.12. We observe that LEDL-enabled modes result in significant EDP improvements, achieving an EDP improvement of 1.35× and 1.47× for 6 VFMAs with FR mode.

5.4.4 Impact of Microarchitectural Parameters

In this section, we study the impact of microarchitectural parameters on the energy characteristics of different FMA modes on our hardware design points. We perform the analysis on the conv2 layer of Alexnet (Alexnet's most time-consuming layer).

In this experiment, we measure EDP for Alexnet conv2 layer for different number of architectural registers and different hardware design points. The experiment is conducted for all three VFMA modes. We show the result of this experiment for NR mode, modeling the baseline Haswell processor configuration, and LR and FR modes, LEDL enabled modes that reduce the number of PRF reads, in Figure 5.13(a), (b) and (c) respectively. The figures show EDP improvements for different hardware design points against a hardware design point having 2 VFMA units and 16 architectural registers.

VFMA units and VFMA modes. First, we observe that increasing VFMA units result in significant EDP improvements for 96 architectural registers. But more importantly, we observe that this increase is limited to 2.4× for 4 VFMA units in NR mode. Adding any more VFMA units requires extra PRF bandwidth, reaching a point where PRF latency constraints could no longer be met for NR mode (shown by the grey box in (a)). LR and FR modes reduce the number of PRF reads, substantially reducing the PRF bandwidth requirements. This lets us pack more compute units, extending the number of VFMA units to 5 and 6 for LR and FR modes, increasing the EDP improvements to 3.5× and 3.7× respectively, as shown in (b) and (c).

Architectural Register Count. Next, we analyze the impact of architectural register count. There are three key observations. First, from (a), we observe that current Intel machines, which have 16 architectural registers, can improve their EDP by 35% just by increasing the architectural registers to 24. Second, number of architectural registers limit the EDP improvements when we increase the number of VFMA units for all VFMA modes. For example, in (a), 4 VFMA units achieve an EDP improve-

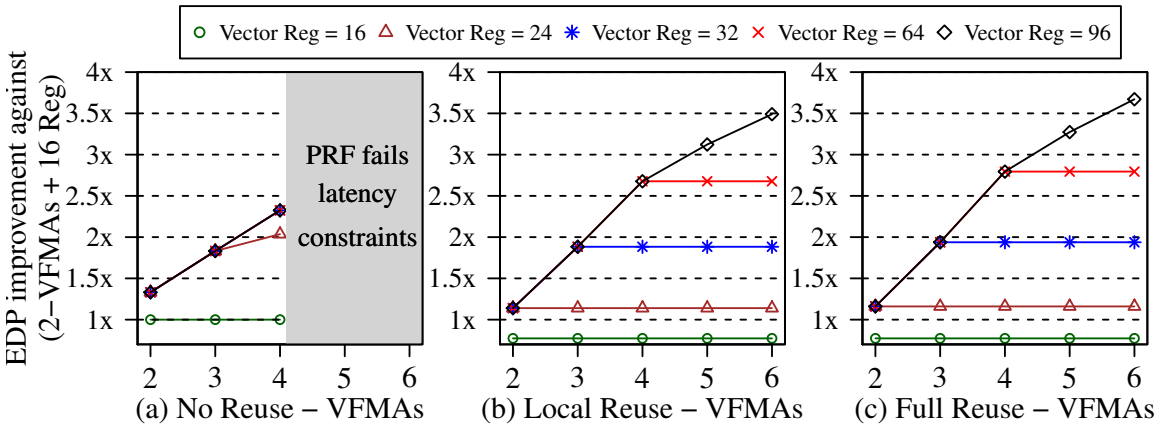


Figure 5.13: Impact of increasing architectural registers and VFMA units on Alexnet conv2 layer for different FMA modes

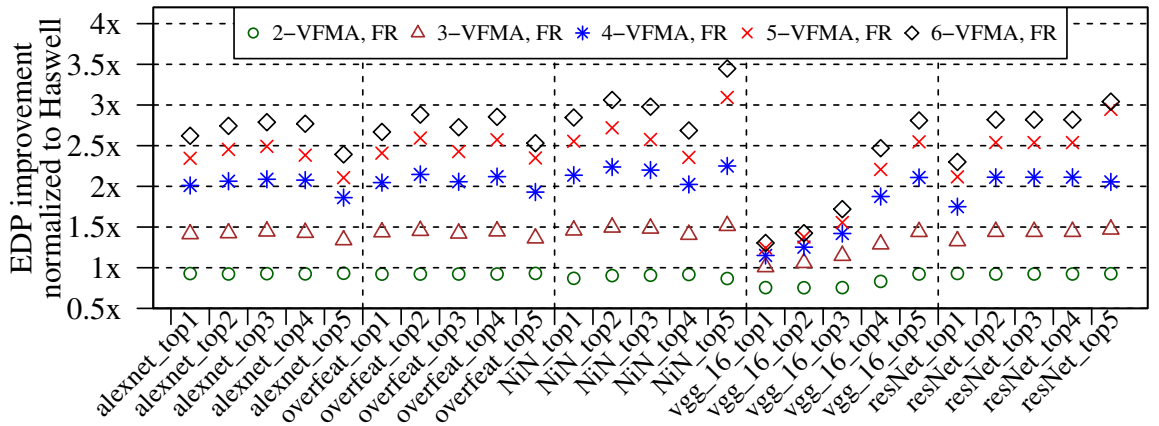


Figure 5.14: LEDL's EDP improvements on FR mode for the top 5 most time contributing layers of our application suite

ment of 2× at 24 registers, which can be increased to 2.4× at 32 registers. And last, we observe that FR and LR require larger number of architectural registers for same number of VFMA units as compared to NR (equations 5.1, 5.3). For example, for 3 VFMA units, NR requires 24 registers but FR requires 32.

5.4.5 Layer-by-Layer Analysis

Different convolution layers within the same network can have different performance and compute requirements. In this subsection, we present a layer-by-layer EDP analysis of our application suite. Due to space limitations we show only the top

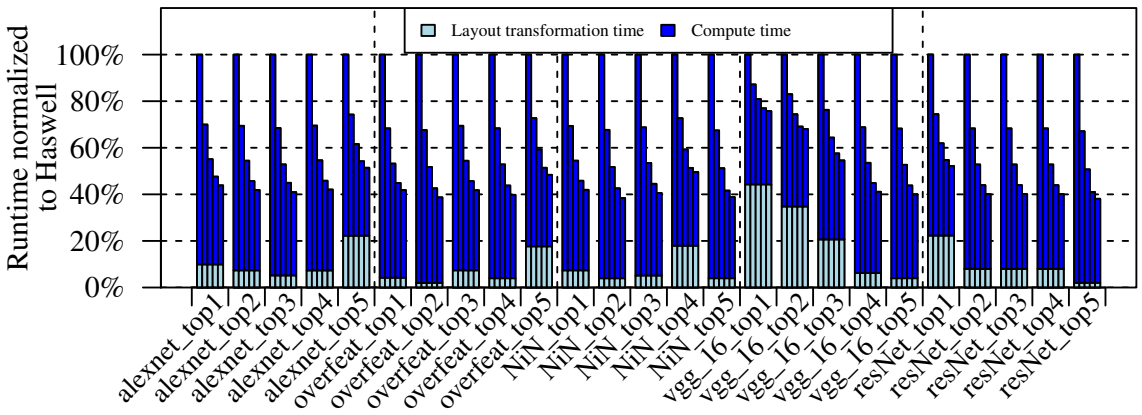


Figure 5.15: Breakdown of runtime in compute and layout transformation time, as the number of VFMA units are increased (2, 3, 4, 5 and 6 from left to right in each cluster)

5 layers in each network (ranked by their contribution to execution time).

EDP Analysis. In this experiment, we evaluate layer-by-layer EDP improvement when the number of VFMA units are increased. The number of architectural registers are kept fixed at 96. ACG takes the number of architectural registers and VFMA units as input and generates a code variant that uses FR mode for all the VFMAs. The findings of this experiment are presented in Figure 5.14, showing the EDP improvements over the Haswell baseline.

We observe that LEDL reuse features achieve significant EDP improvements when the number of VFMA units are increased. Alexnet, for example, achieve close to 2.5x EDP improvement for the top 4 most contributing layers. However we also observe, that some layers like top 3 layers of VGG do not achieve similar EDP improvements. This can be attributed to the high layout transformation time for these layers, which we evaluate next.

Layout Transformation Overhead. To understand the variation of EDP improvements across different convolution layers, we investigate the application execution time breakdown across compute and layout transformation steps. The findings of this experiment are presented in Figure 5.15, showing the breakdown of layer ex-

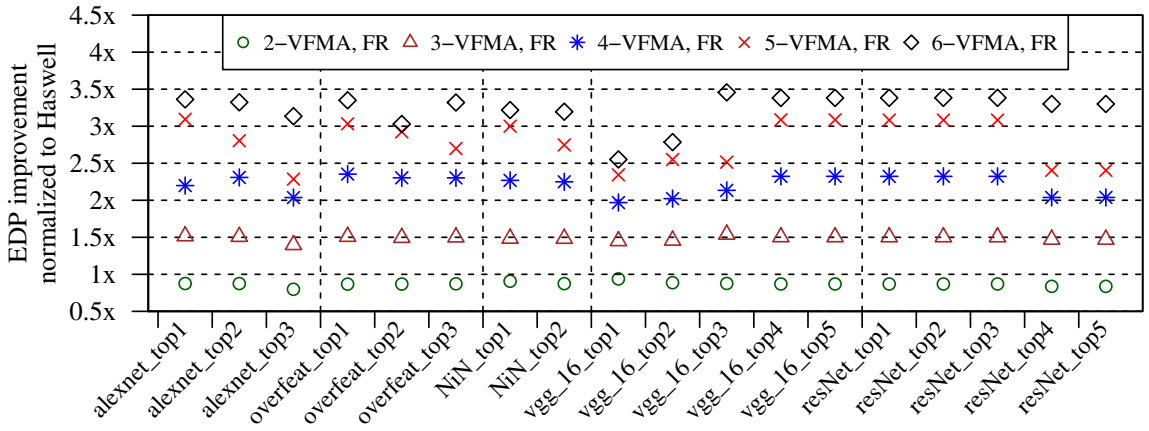


Figure 5.16: LEDL's EDP improvements on winograd algorithm

ecution time across compute and layout transformation portion when the number of VFMA units are increased. The runtime is normalized to Haswell baseline.

There are two keys observations from this experiment. First, the compute time scales down gracefully as more VFMA units are added into the CPU. Second, the data transformation time remains constant and is not affected by the VFMA units. These two factors explain the EDP improvement variations observed earlier in Figure 5.14. VGG_16 layers have large data transformation cost, resulting in smaller EDP improvements. Such layers are characterized by small value of M (rows in matrix A) and large values of K and N (rows and columns, respectively, in matrix B). Therefore, the compute time, in the order of $O(M * N * K)$, in these layers is comparable to data layout transformation time, which is in the order of $O(M * K + N * K)$. Alexnet and ResNet, on the other hand, have low layout transformation overhead, leading to higher EDP improvements.

5.4.6 Applicability to Other Algorithms

In this section, we present LEDL EDP analysis to other algorithms and application domains beyond CNN.

Winograd. First, we apply LEDL on Winograd convolution algorithm, which is

typically implemented using SGEMM kernel calls to efficiently utilize the hardware resources. We use NNPACK library to extract the SGEMM kernel call parameters [49]. Currently, NNPACK supports Winograd algorithm only for those convolution layers that have a 3×3 kernel. Therefore, this study covers only those layers that meet this kernel size constraint.

In this experiment, we evaluate layer-by-layer EDP improvement when the number of VFMA units are increased, while the VFMA units are configured in FR mode. We present the results normalized to Haswell baseline in Figure 5.16. We observe that reusing data and cutting down PRF reads results in significant EDP improvements. Overall, VGG_16, for example, achieves an EDP improvement of close to 2.2× for 4 VFMA units.

Fully Connected and Recurrent Neural Networks. We next measure LEDL impact on two other widely used DNN layers - Fully Connected (FC) and Long Short Term Memory (LSTM) layers (a type of recurrent layer) - that are also implemented atop SGEMM [84]. Recently, Google released an ASIC, having a fast matrix multiplication unit, to accelerate DNN inference – Tensor Processing Unit [84]. The research also showed that a subset of their FC and LSTM layers were compute bound on CPUs (refer to Figure 6 in [84]). Therefore, the extensions offered by LEDL have the potential to improve energy and performance in these cases as well.

To study the applicability of LEDL on these layers, we evaluate a variety of FC and LSTM layers from five application domains – FC layer for Parts of Speech [73], LSTM layer of 200 cells for Language Modelling [173], LSTM layer of 128 cells for Image Captioning [102], LSTM layer of 500 cells for Sentiment Analysis [161] and LSTM layer of 1024 cells for Sequence to Sequence encoder [152]. In this experiment, we measure the EDP improvement for these layers for all hardware design points and VFMA modes. The findings of this experiment are presented in Figure 5.17, showing EDP improvement for the best VFMA mode for all hardware design points

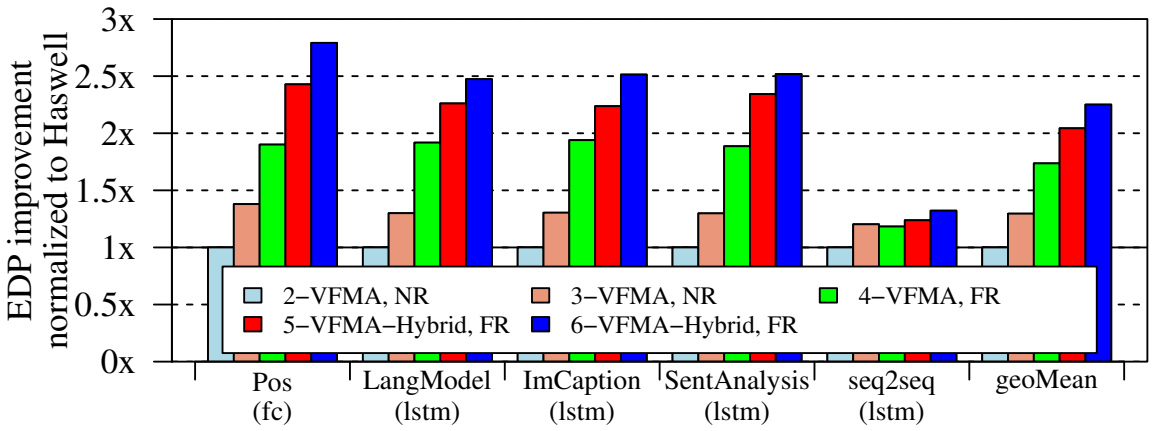


Figure 5.17: LEDL shows good EDP improvements for other widely used DNN layers – FC and LSTM

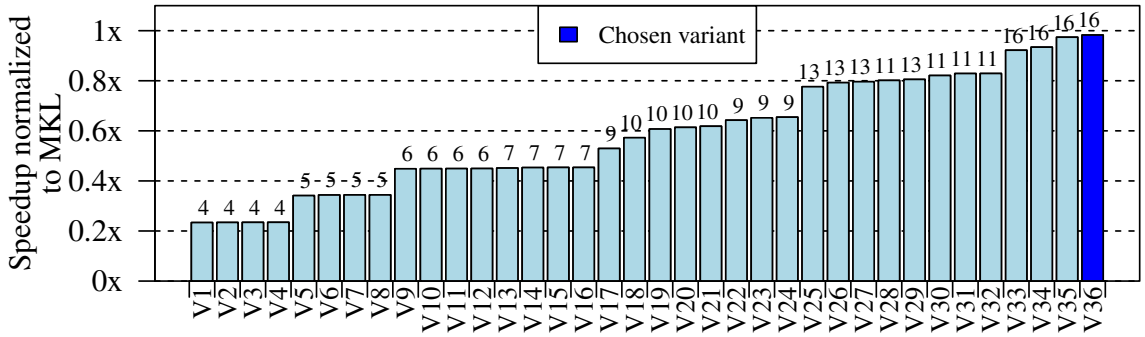


Figure 5.18: Performance of ACG variants against Intel MKL code. Variant register usage is shown at the top of each bar

normalized to a conventional Intel Haswell baseline.

We observe that increasing VFMA units achieve significant EDP improvement for 4 out of 5 layers. As we increase the number of VFMAs to 4, FR mode starts showing better EDP characteristics, resulting in average speedup of 1.7×, 2.0× and 2.3× for 4, 5 and 6 VFMA units. The last application, seq2seq LSTM layer, shows low EDP improvement because this layer is memory bandwidth bound, resulting in diminishing improvements for additional VFMA units.

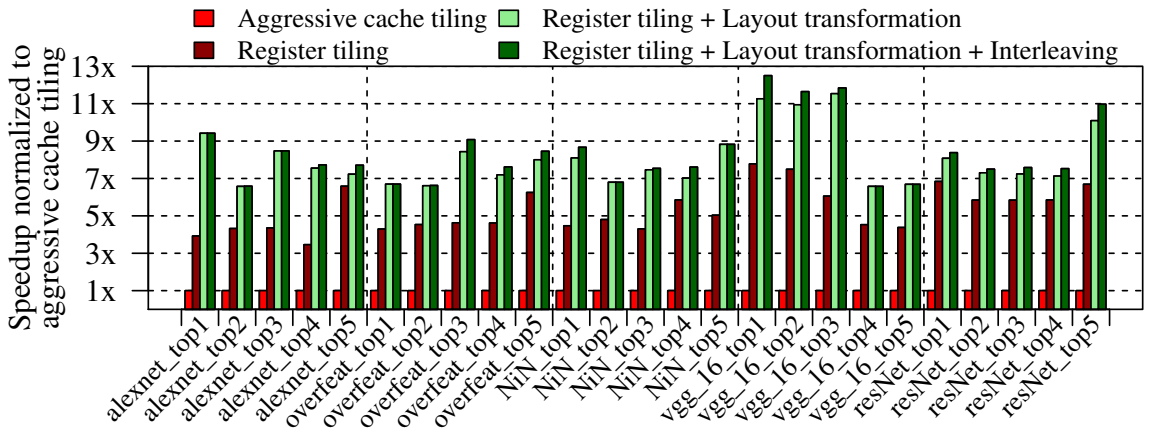


Figure 5.19: Speedup achieved by different ACG's optimizations

5.4.7 Area Overhead

Increasing raw computation capacity of a CPU requires adding more VFMA units as well as increasing the number of read ports in PRF. LEDL, in addition, introduces additional microarchitectural elements to reduce the PRF read bandwidth requirements. However, LEDL microarchitectural additions have minimal area overhead as VFMA remote register is local to its VFMA and InterVFMA links are also unidirectional with single link between two VFMAs. Therefore, the two major factors that govern the area overhead are VFMA units and PRF. We use McPAT to capture this area overhead for our hardware design points.

The area measurement is performed assuming a traditional CPU server, having 8 CPU cores, each having private L1 and L2 caches and sharing a LLC, whose parameters are listed in Table 5.1. We observe that the additional area for 3-VFMA, 4-VFMA, 5-VFMA and 6-VFMA-Hybrid design is 4%, 8%, 11% and 15% respectively. Most of this increase is because of the additional VFMA units. For example, 14% area (compared to total of 15%) for 6 VFMA-hybrid design point is just because of additional VFMA units. Pollack's Rule states that performance increase due to microarchitectural advances are roughly proportional to the square root of increase in complexity, where complexity refers to the area [23]. We observe that LEDL leads

to significant performance and energy improvements, that greatly outstrip the typical Pollack's Rule tradeoff.

5.4.8 Code Generator Efficacy

ACG is designed to generate codes that can take advantage of additional VFMA units, while also maximizing the local and inter-unit reuse. In this section, we evaluate the efficacy of the ACG, both on real hardware and simulation.

ACG Software Variants. In this experiment, we show the inner workings of ACG for Alexnet Conv2 layer on real Intel Haswell machines. Instead of choosing a particular set of tile parameters, we use ACG to sweep the tiling parameters over a small range to generate many software variants and measure their performance on real hardware. The results of this experiment are presented in Figure 5.18. The figure shows variants' performance against Intel MKL, an aggressively tuned code for Intel Haswell machines. The register usage of each software variant is presented at the top of its bar.

There are two key observations from the figure. First, to achieve the high performance, the SW variant has to efficiently utilize the register storage. Current Intel Haswell processor has 16 architectural registers. The figure shows that the highest performing variant utilizes all of these registers. Second, ACG achieves close to Intel MKL performance, which is an aggressively hand-tuned library.

ACG Optimization Breakdown. ACG uses Register tiling, Prefetcher-friendly layout transformation and Interleaving to achieve high performance on CPUs. In this experiment, we analyze the importance of each of these optimizations on Intel Haswell processor across the top 5 most contributing layers for each network in our application suite. The performance speedup of the optimizations is presented in Figure 5.19.

We start with an aggressively Cache-tiled code, that performs cache tiling across L1, L2 and L3 caches. We observe this code performs poorly, leading to heavy un-

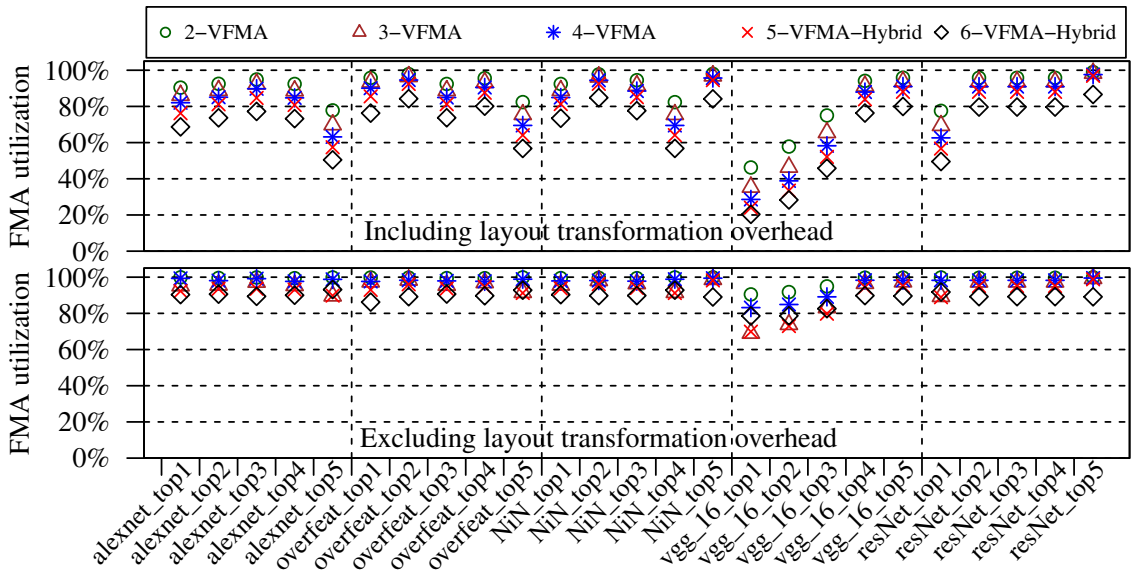


Figure 5.20: VFMA utilization achieved by ACG including and excluding the layout transformation overhead. ACG generated code achieves high VFMA utilization

der utilization of CPU resources. We then apply Register tiling to our software, leading to huge performance improvement for several convolution layers. Next, we apply the prefetcher-friendly layout transformation. This optimization makes accesses prefetcher friendly, again leading to substantial performance improvements. Finally, we apply Interleaving between compute and transformation, reducing transformation cost by overlapping it with some compute portion, leading to small additional performance improvements.

FMA Utilization. Finally, we study ACG performance when the number of FMA units are increased. In this experiment, we analyze the VFMA utilization for the top 5 most contributing layers for each of our network. We increase the number of VFMA units and instruct ACG to generate software using the FR mode of the VFMAs. We present VFMA utilizations of this experiment with and without the transformation overhead in Figure 5.20 (a) and (b) respectively. We observe that ACG efficiently utilizes the compute for majority of Alexnet, Overfeat and ResNet CNN layers. However, top three layers of VGG_16 have low utilization. To investigate this low VFMA utilization, we exclude the transformation overhead and measure the

VFMA utilizations. The findings, presented in part (b), show that ACG achieves high VFMA utilization in SGEMM compute portion.

5.5 Summary

In this work, we focus on identifying and alleviating the microarchitectural bottlenecks that prevent us from improving CPU performance on CNN computations. Our study shows that designing a PRF capable of feeding computational units is the primary barrier on achieving higher CPU FLOPS. We present Locality Extensions for Deep Learning (LEDL) , a novel, minimally intrusive set of microarchitectural and ISA extensions that address this problem, along with an automatic code generator needed to take advantage of our design. Our detailed evaluation shows that applying these extensions allows packing more compute in the CPUs, and can achieve a 2× performance improvement and a 2.7× energy-delay product improvement compared to Haswell processors.

CHAPTER VI

Gist: Efficient Data Encoding for Deep Neural Network Training

The availability of large datasets and powerful computing resources has enabled a new breed of deep neural networks (DNNs) to solve hitherto hard problems such as image classification, translation, and speech processing [75, 77, 88, 90, 160]. These DNNs are *trained* by repeatedly iterating over datasets. The DNN training process has large compute and memory requirements and primarily relies on modern GPUs as the compute platform. Unfortunately, as DNN models are getting larger and deeper, the size of available GPU main memory quickly becomes the primary bottleneck[1], thus limiting the size of the DNNs that a GPU can support [128, 129]. Modern DNNs are already facing this issue, prompting researchers to develop memory-efficient implementations of the networks [122].

Many researchers have recognized this shortcoming and proposed approaches to reduce the memory footprint of DNN training. However, prior approaches are not able to simultaneously achieve all of the following three desirable properties: (i) *provide high memory footprint reduction, (ii) low performance overhead, and (iii) minimal effect on training accuracy.* Most prior works propose efficient techniques

[1]For GPU main memory (GDDR5/GDDR5X) the first order concern is bandwidth as many GPU applications are bandwidth-bound. It is hard to get both high bandwidth and high density DRAM-based memory at low cost [100].

to reduce the memory footprint in DNN *inference* with an emphasis on reducing model size (also referred to as weights) [67, 71, 98, 72, 69, 66]. However for DNN training, weights are only a small fraction of total memory footprint. In training, intermediate computed values (usually called *feature maps*) need to be stored/stashed in the forward pass so that they can be used later in the backward pass. These feature maps are the primary contributor to the significant increase in memory footprint in DNN training compared to inference. This important factor renders prior efforts, that target weights for memory footprint reduction, ineffective for training. State-of-the-art memory footprint reduction approaches for training transfer data structures back and forth between CPU and GPU memory but pay a performance cost in doing so [128]. Finally, approaches that explore lower precision computations for DNN training, primarily in the context of ASICs and FPGAs, either do not target feature maps (and thus unable to achieve high memory footprint reduction) or, when used aggressively, result in reduced training accuracy [44, 63, 39].

The key insight of this work is in acknowledging that a feature map typically has two uses in the computation timeline and that these uses are spread far apart temporally. Its first use is in the forward pass and second is much later in the backward pass. Despite these uses being spread far apart, the feature map is still stashed in single precision format (32-bits) when they are unused between these accesses. We find that we can store the feature map data with efficient encodings that result in a much smaller footprint between the two temporal uses. Furthermore, we propose that if we take layer types and interactions into account, we can enable highly efficient *layer-specific encodings* – these opportunities are missed if we limit ourselves to a layer-agnostic view. Using these key insights, we design two layer-specific lossless encodings and one lossy encoding that are *fast, efficient in reducing memory footprint, and have minimal effect on training accuracy.*

Our first lossless encoding, Binarize, specifically targets ReLU layers followed by

a pooling layer. Upon careful examination of ReLU's backward pass calculation, we observe that the ReLU output, that has to be stashed for the backward pass, can be encoded using just 1-bit values, leading to 32× compression for the ReLU outputs. Our second lossless encoding, Sparse Storage and Dense Compute (SSDC), that specifically targets ReLU followed by convolution layer, is based on the observation that ReLU outputs have high sparsity. SSDC facilitates storage in memory-space efficient sparse format but performs computation in dense format, retaining the performance benefits of highly optimized cuDNN dense computation, while exploiting sparsity to achieve high reduction in memory footprint. Finally, in the lossy domain, our key insight of representing the stashed feature maps in smaller format *only* between the two temporal uses enables us to be very aggressive with precision reduction without any loss in accuracy. This lossy encoding, Delayed Precision Reduction (DPR), delays precision reduction to the point where values are no longer needed in the forward pass and achieves significant bit savings (as small as 8 bits).

Utilizing all these encodings, we present ACME that specifically targets feature maps to reduce the training memory footprint. It performs a static analysis on the DNN execution graph, identifies the applicable encodings, and creates a new execution graph with relevant encode and decode functions inserted. ACME also performs a static liveness analysis on the affected feature maps and newly generated encoded representations to assist the DNN framework's memory allocator, CNTK [145] in our case, to achieve an efficient memory allocation strategy.

This paper makes the following contributions:

- **Systematic Memory Breakdown Analysis.** We perform a systematic memory footprint analysis, revealing that *feature maps* are the major memory consumers in the DNN training process. We also make a new observation that the feature maps have high data redundancy and can be stored in much more efficient formats between their forward and backward use.

- **Layer-specific Lossless Encodings.** We present two layer-specific encodings – (1) Binarize that achieves 32× compression for ReLU outputs for layer combination of ReLU followed by Pool, and (2) Sparse Storage and Dense Compute that exploits high sparsity exhibited in ReLU outputs for ReLU followed by convolution layers.

- **Aggressive Lossy Encoding.** We present DPR, that applies precision reduction only for the backward use of the feature maps – values in the forward pass are kept in full precision – leading to aggressive bit savings without affecting accuracy.

- **Footprint Reduction on a Real System.** We observe that ACME reduces the memory footprint by 2× across 5 state-of-the-art image classification DNNs, with an average of 1.8× with only 4% performance overhead. By reducing memory footprint, ACME can fit larger minibatches in the GPU memory, improving GPU utilization and speeding up the training for very deep networks, e.g., a speedup of 22% for Resnet-1202. We also show that further optimizations to existing DNN libraries and memory allocation can result in even larger memory footprint reductions (upto 4.1×).

6.1 Motivation

DNNs typically consist of an input and output layer with multiple hidden layers in between. Recently, convolution neural networks (CNNs), a class of DNNs, have been shown to achieve significantly better accuracy compared to previous state-of-the-art algorithms for image classification [92, 149, 154, 75, 106]. CNNs have been growing deeper with every iteration, consisting of few convolution layers at the start (AlexNet) to having hundreds of convolution layers in recent ones (Inception).

6.1.1 Training vs. Inference

DNNs have two distinct modes of operation: (i) *training*, when a model is trained based on a set of inputs (training set) and a corresponding set of expected outputs, and (ii) *inference*, when an already trained network is used to generate predictions for new inputs.

For this paper, we focus on two main differences between training and inference. First, training consists of two phases: *forward* and *backward* passes [132, 42]; inference only involves a forward pass. The goal of the backward pass in DNN training is to backpropagate error and find weight error gradients that can be applied to the weights to steer the parameters in the *right* direction. Training is performed in batches of input images, commonly known as a minibatch [25]. Training on minibatches as opposed to training on an image-by-image basis has been shown to achieve better accuracy and better hardware utilization [57, 73, 41].

Second, in inference the major part of storage overhead comes from *weights*. These weights are fixed after training, and hence many different optimizations can be applied to reduce their storage requirements [66, 13, 67, 31]. In contrast, training has many distinct data structures, e.g., weights that change over the course of training, weight gradients, feature maps (intermediate layer outputs) that need to be stashed in the forward pass for use in the backward pass, and backward gradient maps.

6.1.1.1 Why Memory Can Be a Problem in Training?

To understand the memory requirements of GPU-based training, we study the breakdown of memory footprint on five state-of-the-art CNNs in CNTK. While using FPGAs [174] and ASICs [84, 80] is also possible, most of these designs are either proprietary or in a relatively early development stages. Hence, we conduct our study on a modern GPU (Maxwell GTX Titan X in our case). However, our approaches are applicable to optimized hardware as well (Section 6.4.8).

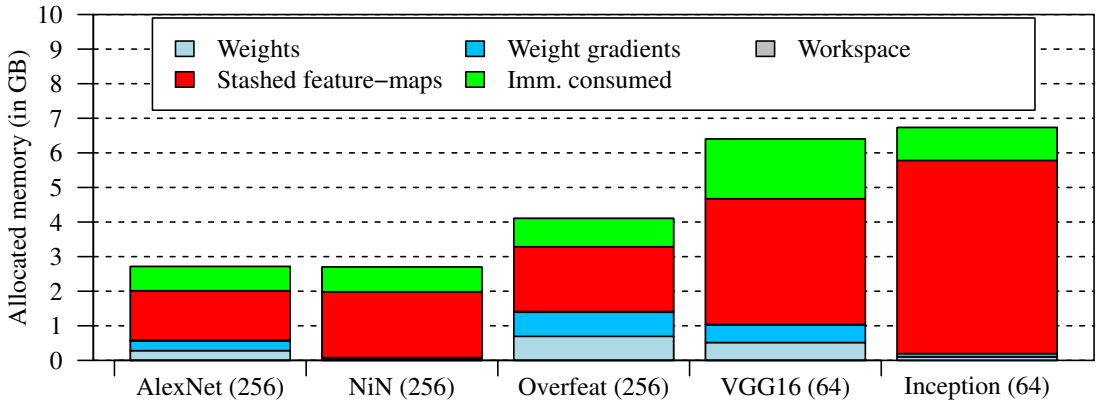


Figure 6.1: Breakdown of memory footprint in DNN training amongst different data structures

Figure 6.1 shows the breakdown of total memory footprint across different data structures. Feature maps are the intermediate layer outputs that are passed on as an input to the following layer. Gradient maps are the intermediate gradients generated in the backward pass and passed as an input to the previous layer. In CNNs, not every feature map has to be saved for the backward pass. We thus distinguish *stashed* feature maps (generated in forward and used in both forward and backward passes) from *immediately consumed* feature maps (generated in forward pass and consumed immediately in forward) and gradient maps (generated in backward pass and consumed immediately). Stashed feature maps are required in the backward pass and thus stored for a long time in a minibatch processing. In contrast, immediately consumed feature maps and gradient maps can be discarded as soon as they are used. Finally, *workspace* is cuDNN's intra-layer storage to support layer computations [35]. cuDNN provides a choice between memory-optimal and performance-optimal implementations, translating to tradeoff between algorithm performance and workspace storage requirements. In this work, we choose its memory-optimal implementation as an optimized baseline.

We draw two major conclusions from this figure. First, larger (deeper) DNNs consume large amount of memory even with relatively small minibatch sizes (64).

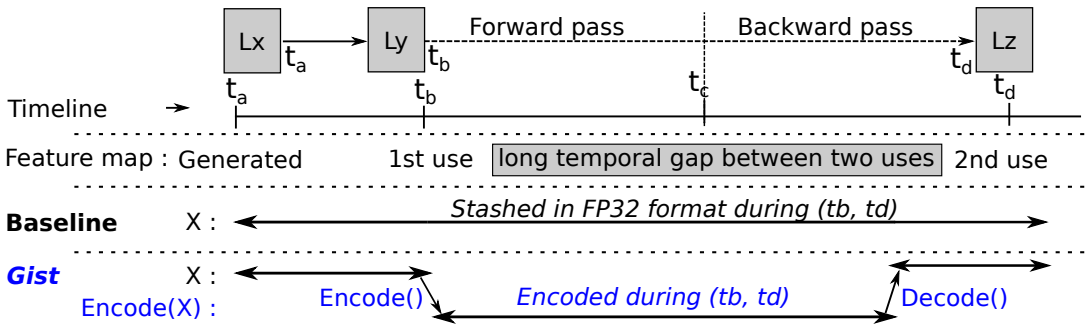


Figure 6.2: The two temporal uses of feature map are far apart

VGG16 and Inception can only fit in our GPU memory if the minibatch size is 64 and start exceeding the 12 GB GPU memory limit at higher minibatch size. Higher minibatch size is desirable as it leads to better GPU utilization [57]. Second, training memory footprint tends to be dominated primarily by stashed feature maps, followed by immediately consumed data structures. For example, in VGG16, 83% of memory is consumed by stashed feature maps and immediately consumed data; this number grows to 97% for Inception. This result stands in stark contrast to inference where feature maps don't need to be stashed and memory consumption is dominated by weights. We conclude that the stashed feature maps and immediately consumed data structures (in that order of importance) are key for optimizing GPU memory consumption in CNN training.

6.1.2 Limitations of Prior Work

In this paper, we develop techniques to reduce the DNN training memory footprint. Here, we briefly describe the limitations of existing approaches.

Prefetch and Swap-out. One potential approach is to move parts of the working set between CPU and GPU memory using PCIe links and smart prefetching analysis [128]. However, this approach still suffers from significant overheads of data transfer with respect to power/energy and their inability to completely mask the performance cost of swapping data in and out of GPU memory (upto 27% for Inception).

In addition, it uses a shared resource, PCIe links, that is of critical importance in distributed DNN training [41].

Reducing Minibatch Size. While reducing minibatch size is effective at reducing the memory footprint during training, it adversely affects the runtime of the training process because smaller minibatches lead to GPU underutilization [57]. This performance hit can be recovered by using more GPUs, where each GPU works on a smaller minibatch. But this is also an inefficient solution as GPU machines are both costly and power hungry, and might also result in sub-linear scaling due to stragglers and transfer across workers [41].

Recompute. Instead of a saving the output of a large layer, prior work has considered recomputing the output of a layer's forward pass again in the backward pass [32]. Unfortunately, we observe that the largest layers are usually the ones that also take the longest to recompute, that can cause significant performance overhead. Yet, this technique is still applicable for some specific layers (like batch normalization) and can be used in conjunction with our work.

6.2 Gist: Key Ideas

In this work, we design techniques to reduce DNN training memory footprint by focusing on the primary contributors – feature maps. We find that a feature map typically has two uses in the computation timeline and these uses are spread far apart temporally, as shown in Figure 6.2. Its first use is in the forward pass and the second use is much later in the backward pass. In the baseline, the data is stashed in single precision (FP32) even though these uses are far apart. In our approach, we represent the data in much smaller encoded format in the temporal gap and decode it just before it is needed again in the backward pass; the forward use still gets the data in FP32 format, but the memory space is relinquished as soon as the forward

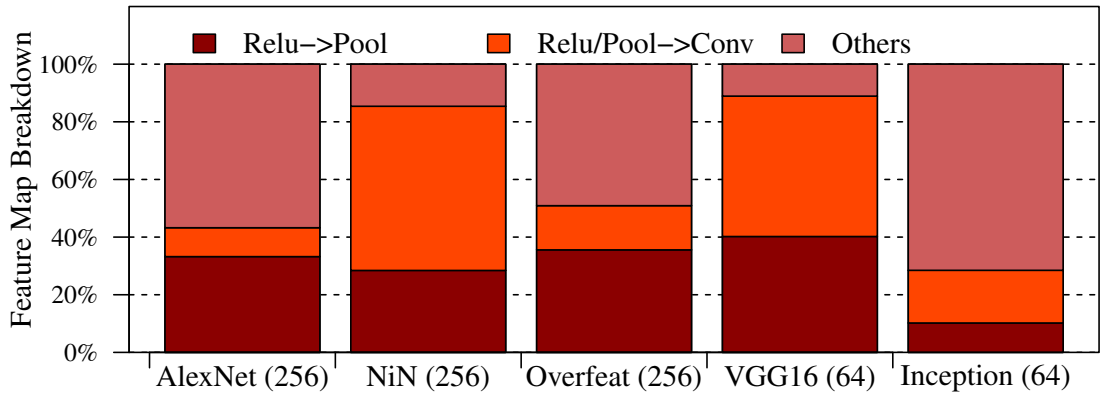


Figure 6.3: Breakdown of memory within stashed feature maps. ReLU layer consumes a major portion of the footprint.

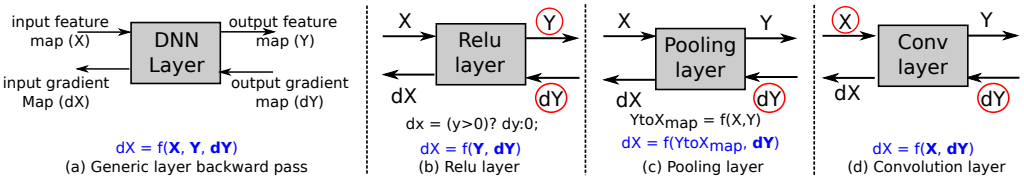


Figure 6.4: Backward pass computation. Only circled data structures are needed in the backward pass

use is complete. This results in an efficient memory sharing strategy (Section 6.3.3), reducing total memory footprint.

Target data structures	**Footprint Reduction Technique**	**Type**
ReLU-Pool feature map	*Binarize*	Lossless
ReLU-Conv feature map	*Sparse Storage and Dense Compute*	Lossless
Other feature map	*Delayed Precision Reduction*	Lossy
Immediately consumed	*Inplace computation*	Lossless

Table 6.1: Summary of ACME techniques

Our second key insight is that taking layer types and interactions into account opens up opportunities of designing highly aggressive encoding schemes, which were earlier hidden due to the layer agnostic nature of prior work. In this section, we first identify such opportunities that let us design two lossless and one lossy encodings specifically targeted to reduce memory footprint of stashed feature maps. Next, we observe that inplace computations can reduce the size of immediately consumed data structures. Table 6.1 shows a brief outline of our techniques and their target data

structures.

6.2.1 Opportunities For Lossless Encodings

In lossless encoding domain, we target ReLU activation functions which are heavily used in all major CNNs targeted for vision-related tasks [75, 160, 60, 90]. In these CNNs, the convolution layers are typically followed by ReLU activations. This group of a Conv-ReLU pair is either followed by the same group or by a pool layer, resulting in many ReLU-Conv and ReLU-Pool layer pairs. Consequently, we observe that ReLU feature maps form a major fraction of total memory footprint.

This is shown in Figure 6.3 which zooms in on the stashed feature maps (in Figure 6.1) and analyze their breakdown across different CNN layers, with an emphasis on three different categories: (i) ReLU outputs followed by a Pool layer (*ReLU-Pool*), (ii) ReLU/Pool outputs followed by a conv layer (*ReLU-Conv*), and (iii) remaining stashed feature maps (*Others*). We observe that significant portion of memory footprint is attributed to *ReLU outputs* (Pool outputs have very low contribution). For example, VGG16 has 40% and 49% of the stashed feature maps for ReLU-Pool and ReLU-Conv respectively (89% total used for ReLU outputs).

We make two key observations for these ReLU outputs that let us store the stashed feature map with much fewer bits. First, when carefully examining the backward pass computation of ReLU layer, we observe that ReLU outputs for ReLU-Pool combination can be stored in just 1-bit. Second, we observe that ReLU outputs typically have high sparsity that can be exploited to encode the feature map in much smaller sparse format. Next, we expand on these opportunities.

ReLU-Pool. Typically, in a backward pass calculation, a layer uses its stashed input feature map (X), stashed output feature map (Y) and output gradient map (dY) to calculate input gradient map (dX), as shown in Figure 6.4(a). However, every layer does not require all this data for the backward pass. Upon further investigation,

we discover that although feature maps are stashed with the same precision across layer types, it is mostly for convenience, not out of computation necessity. We show backward pass calculation for *ReLU* in Figure 6.4(b). It only requires Y and dY to calculate dX. Moreover, an element of dY is passed to dX, only if corresponding element in Y is positive, else dX is set to 0. With this observation in mind, it is natural to consider replacing Y with 1-bit values. Unfortunately, it is not always possible, because the next layer might require its stashed input feature map X (ReLU output in this case) for the backward pass calculation. However, upon further examination, we observe that in the case of ReLU-Pool, the pool backward pass does not require the actual values of ReLU output as shown in Figure 6.4(c) (described in more details in Section 6.3.1), resulting in significant encoding opportunities. This observation becomes the basis of our first lossless encoding, called Binarize.

ReLU-Conv. Binarize is not applicable to ReLU-Conv pair, because convolution requires its stashed input feature map for the backward pass calculation (as shown in Figure 6.4(d)). However, upon careful data analysis, we observe that ReLU outputs have high sparsity (large number of zeroes) induced by the ReLU calculations in the forward pass. For example, for VGG16, we observe high sparsity, going even over 80%, for all the ReLU outputs, motivating us to apply sparse compression and computation for these feature maps. However, switching both compute and memory to sparse domain results in significant performance degradation [171, 76]. Building on this observation, we present Sparse Storage and Dense Compute (SSDC) encoding that stores the data in Compressed Sparse Row (CSR) encoding format while keeping the computation on dense format.

6.2.2 Opportunities For Lossy Encodings

For lossy encoding, we investigate precision reduction as it is amenable to GPU architecture compared to prior offline approaches like quantization and huffman en-

coding [67]. We make two observations that let us achieve aggressive bit savings without any loss in accuracy.

Non-Uniform Precision Reduction. We observe that *uniform* precision reduction to even 16 bits, where all the data structures are represented in 16 bits, leads to severe accuracy losses. We observe that if, instead, we restrict the precision reduction to only gradient maps, then the training accuracy is not affected. Note that, although this type of selective precision reduction has been studied in a recent work [44], it does not study the effect of reducing precision in stashed feature maps.

Delayed Precision Reduction. But more importantly, we find that the way the precision reduction is applied in training should be significantly changed. Currently, the conventional wisdom [44, 63, 39, 65] is to apply precision reduction right after the value is generated by a particular layer. This design choice leads to the situation when the error generated by the precision reduction is injected directly into the next layer and is then propagated (potentially increasing in magnitude) over all future layers in the forward pass. In our work, we observe that it is better to separate the two uses of every output layer in the forward and backward pass (as previously shown in Figure 6.2). The first immediate use (by the next layer) significantly benefits from the more precise (usually FP32) representation, keeping forward pass error-free. While the second use, much later in the backward pass, can tolerate lower precision. This separation, implemented in our Delayed Precision Reduction encoding, push the bit lengths to very small value like 8 bits for multiple DNNs (unseen in prior work).

6.2.3 Opportunities For Inplace Computation

Next, we shift our focus from stashed feature maps to immediately consumed data structures. We observe that a good portion of immediately consumed data can be removed by performing inplace computation. As discussed in a previous work [32], this optimization is applicable for the layers (specifically ReLU) that have a *read-once*

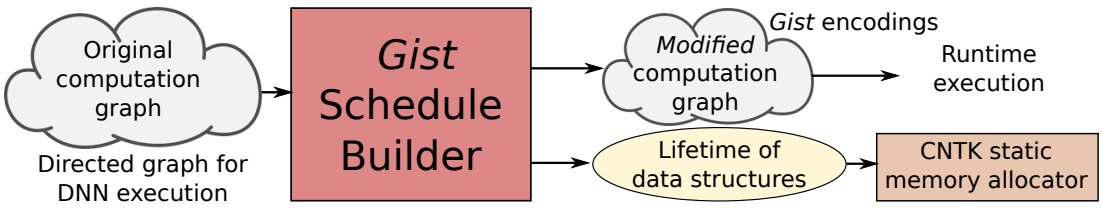


Figure 6.5: ACME System Architecture - Schedule Builder finds applicable ACME encodings and performs liveness analysis.

and write-once property between each element of input and output. In the absence of inplace optimization, convolution output shows up in immediately consumed category. With inplace computation, the memory space for convolution is reused by ReLU, reducing immediately consumed memory footprint.

6.3 Design and Implementation

Based on the observations in the previous section, we present the design of our system ACME. Figure 6.5 shows our system architecture. Typically, DNN frameworks like CNTK [145] and TensorFlow [12] represent DNNs as a series of computational steps via a directed execution graph. ACME's *Schedule Builder* takes the original execution graph, identifies the edges where new encodings/decodings are needed, and creates a new execution graph with encode/decode functions inserted.

In isolation, encoded representations only add to the memory footprint. To solve this problem, we utilize the CNTK memory allocator that uses the lifetimes of various data structures to find an efficient static memory *sharing* strategy. Schedule Builder performs liveness analysis, infers the lifetime of affected stashed feature maps and encoded representations, and presents them to the CNTK memory allocator for optimization. ACME encodings reduce the lifetime of FP32 stashed feature maps, opening up more opportunities for memory sharing, and thus reducing the total memory footprint. In this section, we present the details of ACME encodings and design of ACME's Schedule Builder and its interaction with CNTK memory allocator.

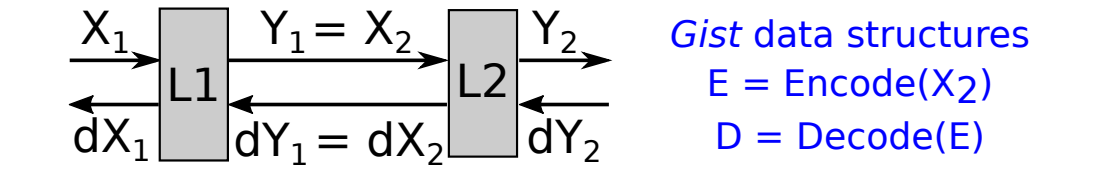
6.3.1 Encodings

We present a generic view of encodings in Figure 6.6, illustrating the difference between baseline and ACME encodings in terms of new data structures and changes in backward pass calculations (shown in blue color). For baseline, backward pass calculation is a function of intermediate feature map (Y_1 in the figure) along with other data structures. ACME introduces two new data structures – E, the Encoded intermediate feature map in the forward pass, and D, the decoded intermediate feature map in the backward pass. These new data structures change the backward pass calculation, which are now dependent on D instead of Y_1. We now present the details of encodings.

Lossless Encoding - Binarize. As discussed in Section 6.2, we observe that for ReLU-Pool layer combination, ReLU output can be encoded much more efficiently, because (i) the backward pass of ReLU only needs to know if the stashed feature map is positive (1 bit), and (ii) the pool layer backward pass can be optimized so that it does not need ReLU outputs.

CNNs typically use MaxPool layer to subsample the input matrix. MaxPool's forward pass slides a window of a specific size over the input matrix X, finds the maximum value in this window, and passes it on to the output Y. For the backward pass, it passes on dY to that location of dX in the window, from where the maximum value of X was chosen in the forward pass. In baseline CNTK implementation, the MaxPool layer stashes both input and output feature maps for the backward pass to find the location of maximum values. We instead, create a mapping from Y to X in the forward pass that keeps track of these locations ($YToX_{map}$ in Figure 6.4 (b)). We use this mapping in the MaxPool backward pass calculation, removing the dependence on its input and output stashed feature maps.

With this optimization, Binarize encoding lets us achieve significant reduction in memory footprint for ReLU-Pool layer combination. Binarize adds two encoded



Baseline: $dX_1 = f(\mathbf{Y_1}, X_1, dY_1)$, $dX_2 = f(\mathbf{X_2}, Y_2, dY_2)$

Gist: $dX_1 = f(\mathbf{D}, X_1, dY_1)$, $dX_2 = f(\mathbf{D}, Y_2, dY_2)$

Layer pairs	*Encode function*	*Decode function*
ReLU - Pool	FP32 to 1-bit	Work on encoded data
ReLU/Pool - Conv	FP32 to CSR-like	CSR-like to FP32
Others	FP32 to FP8/10/16	FP8/10/16 to FP32

Figure 6.6: ACME encodings

feature maps. First, a 1-bit data structure that replaces the ReLU feature map, storing information whether the stashed feature map value was positive. Second, for Pool layer, Binarize stores a Y to X mapping of location of maximum values. This data structure has as many elements as the Pool output (typically, one-fourth or one-ninth of preceding ReLU output), where each element is stored in 4 bits (the largest sliding window in our application suite is 3×3). Therefore, these encoded data structures result in a compression of close to 16× (32× for ReLU output and 8× for MaxPool output) for ReLU-Pool stashed feature maps.

Tying back to Figure 6.6, both encode and decode functions are implemented within ReLU and pool layers using CUDA. Pool and ReLU backward pass have been updated to perform computation on the encoded data structures itself. We observe small performance improvements with Binarize encoding, because Binarize encoding significantly increases effective memory bandwidth for ReLU layers, improving memory-bandwidth bound ReLU backward pass computation.

Lossless Encoding - Sparse Storage and Dense Compute. As discussed in Section 6.1, other set of ReLU layers – ReLU-Conv, exhibit high amount of sparsity induced by ReLU calculations, making it suitable to store these feature maps in sparse format. We also observe high sparsity for a few Pool-Conv layer combinations if the

preceding ReLU layer has high sparsity. SSDC encoding is applicable to both layer combinations.

However, switching to sparse computation on GPUs shows performance improvements only when the sparsity is very high ($> 97.5\%$), which is typically not the case in CNNs [171, 76]. To tackle this problem, we present Sparse Storage and Dense Compute (SSDC) encoding, that isolates computation and storage, facilitating storage of data in sparse format and computation in dense (FP32) format. SSDC stores the data in sparse format for the majority of its life time, and converts the data back into dense format only before it is actually required for computation. This achieves significant memory footprint reduction, while retaining the performance benefits of highly optimized cuDNN library.

For choosing a suitable sparse format, we compare 3 commonly used formats - ELL, Hybrid and Compressed Sparse Row (CSR). We observe that CSR achieves lowest format-conversion latency among these options, achieving the best compression-performance overhead tradeoff. This format stores all the non-zero values, along with a meta array that holds the column indices of the non-zero values in each row. (There is an extra meta array which is very small in size, and, thus omitted for the rest of the discussion). Most DNN frameworks store data structures in an n-dimensional matrix, which can always be collapsed into two dimensions. We take these 2D matrices and convert them into a CSR format.

We use Nvidia cuSPARSE library to perform the encodings/decodings, listed in Figure 6.6. However, the original implementation stores each index as a 4-byte value, resulting in no improvement with compression if the sparsity is below 50%. This is due to cuSPARSE conservative assumption that the number of elements in a row of the matrix can be high, allotting 4 bytes for every column index. We perform *Narrow Value Optimization*, where we reshape the 2D matrix and restrict the number of columns to 256, requiring only 1 byte per column index. This reduces the minimal

sparsity requirement for compression to be effective from 50% to 20%, resulting in both wider applicability and higher compression ratios.

Lossy Encoding - Delayed Precision Reduction. ACME's third encoding uses precision reduction to exploit CNN error tolerance to reduce memory footprint of remaining stashed feature maps. Similar to SSDC encoding, we isolate the storage from computation. The computation still happens in FP32 format while the data is stored with lower precision for most of its lifetime. Though backward pass implementation can be modified to work directly on precision-reduced values, we convert back to FP32 because cuDNN is closed-sourced library. Nevertheless, we discuss the impact of such optimization on compression ratio in Section 6.4.8.

Our usage of precision reduction differs significantly from the previous research that applies it immediately after computation finishes. We delay the precision reduction until the feature map has been consumed in the forward pass, thus naming it Delayed Precision Reduction (DPR). This lets us achieve more aggressive precision reduction on GPUs. DPR is applicable to any layer combination. We also apply it over SSDC encoding, compressing the non-zero values array in the CSR format. We do not touch the meta array in CSR format and Binarize encoded data structures as these affect control, and thus are not suitable for lossy encoding.

Figure 6.6 lists the encode and decode functions for DPR encoding. We use three smaller representations of 16, 10 and 8 bits, packing 2, 3 and 4 values, respectively, into 4 bytes. For packing 3 values into 4 bytes, 10 bits is the largest length possible (9 bits leave 5 bits unused, 11 bits requires one extra bit). For 16 bits, we use IEEE half precision floating point format (1 sign, 5 exponent and 10 mantissa bits), referred to as FP16. For 8-bits (FP8), we choose 1 bit for sign, 4 for exponent and 3 for mantissa, and for 10-bits (FP10), we use 1 sign, 5 exponent and 4 mantissa bits. In FP10, three 10-bit values are stored in a 4-byte space, rendering 2-bits useless. We ignore denormalized numbers as they have negligible effect on CNNs accuracy. We

use *round-to-nearest* rounding strategy for these conversions. The value is clamped at maximum/minimum value if the FP32 value is larger/smaller than the range of the smaller format. We write CUDA implementations to perform these conversions. Since conversions can happen in parallel, DPR results in minimal performance overhead.

6.3.2 Schedule Builder

In ACME, the values in the forward pass are in FP32 format (for both lossless and lossy encodings) while only the data that is required for the backward pass is stored in an encoded format. However, since the feature maps are still generated in original FP32 format in the forward pass before they are encoded, without any further optimization, the encodings will result in increased memory footprint. This gives rise to the question that how does ACME leads to memory footprint reduction.

This task is handled by ACME's Schedule Builder that has two responsibilities. First, identifying the applicable layer encodings from the CNTK execution graph, performing a static analysis to distinguish between forward and backward use of a feature map, and inserting the encode and decode functions in the execution graph, thus creating a new execution graph that is used at runtime. And, second, performing a static liveness analysis for the affected stashed feature maps and newly generated encoded/decoded representations, and pass it on to the CNTK static memory allocator that finds an efficient memory allocation strategy (Section 6.3.3).

Figure 6.2 illustrates the liveness analysis performed by the Schedule Builder. The figure shows the two uses of a feature map that are temporally far apart in the computation timeline – one in the forward pass and one much later in the backward pass. In the baseline, the lifetime of this feature map is very long and it is stored in FP32 format for this whole duration. ACME breaks this lifetime into three regions – FP32 format that is live only for the immediate forward use, encoded (much smaller) format that is live for the long temporal gap between the two uses, and FP32 format

for the decoded value that is live only for the immediate backward use. The figure also shows the points at which Schedule Builder inserts encode and decode functions.

6.3.3 CNTK Memory Allocator

Schedule Builder passes this liveness analysis to the CNTK static memory allocator that finds an efficient strategy to allocate the memory. CNTK, similar to other deep learning frameworks, performs static memory allocation. The other alternative, dynamic allocation, results in a lower footprint but at the expense of many expensive cudaMalloc calls for each minibatch, resulting in performance overhead. Nevertheless, we discuss the impact of dynamic memory allocation on compression ratio in Section 6.4.8.

The key idea employed in the CNTK memory allocator is *memory sharing*. It takes lifetimes of different data structures and their sizes as input, and finds an efficient memory sharing strategy. The memory allocator create groups of data structures whose lifetimes do not overlap and thus can share the same memory space. Therefore, the size of this group is the largest size of the member within the group, as opposed to the sum of size of the members in the group. To come up with an efficient strategy, it first sorts the data structures on the basis of size, and then forms these groups, so that large data structures can share the same memory space. At the end of this process, memory allocator have multiple groups which are either dominated by feature maps that are stored for the backward pass or by immediately consumed feature maps or gradient maps. ACME encodings, by reducing the lifetime of FP32 stashed feature map, create higher opportunities of memory sharing, resulting in lower memory footprint.

Example - Putting it all together. We present an example in Figure 6.7 to illustrate the interactions between static memory manager and ACME encodings. The example shows life-times of five variables (X, A, B, C and D). In part (a), we

show the output of CNTK memory allocator for the baseline. Memory allocator forms 2 groups, resulting in total size of 18 MB, 10 for stashed feature map (X) and 8 for immediately consumed. In part (b), we apply SSDC encoding, which breaks the lifetime of original X into three separate timelines. In this case, CNTK memory allocator again forms two groups, however the total size is reduced to 12, 2 for (encoded) stashed feature map and 10 for immediately consumed. As shown in the figure, *ACME encodings convert the original FP32 stashed feature maps into immediately consumed data, and the encoded (and much smaller) data structure is now stashed for the backward pass.* This might increase the total immediately consumed data (8 to 10 MB here), but reduces the stashed feature map significantly (10 to 2 MB), resulting in overall reduction in memory footprint (18 to 12 MB).

6.4 Evaluation

6.4.1 Methodology

Infrastructure. We evaluate ACME memory reduction capabilities on Microsoft CNTK deep learning framework. We implement ACME encodings, inplace optimization, Schedule Builder, and make necessary changes in CNTK static memory allocator. The evaluation is performed on an Nvidia Maxwell GTX Titan X [81] card with 12 GB of GDDR5 memory using cuDNN v6.0.

Applications. We evaluate ACME on 6 state-of-the-art image classification CNNs: AlexNet [92], NiN [106], Overfeat [146], VGG16 [149], Inception [154] and Resnet [75], using ImageNet training dataset [133]. These CNNs present a wide range of layer shapes and sizes, while also capturing the evolution of CNNs in past few years.

Baselines. Our first baseline, referred to as *CNTK baseline*, is CNTK original static memory allocation strategy, without any of our optimizations. In Section 6.1, we show that stashed feature maps and immediately consumed data structures are

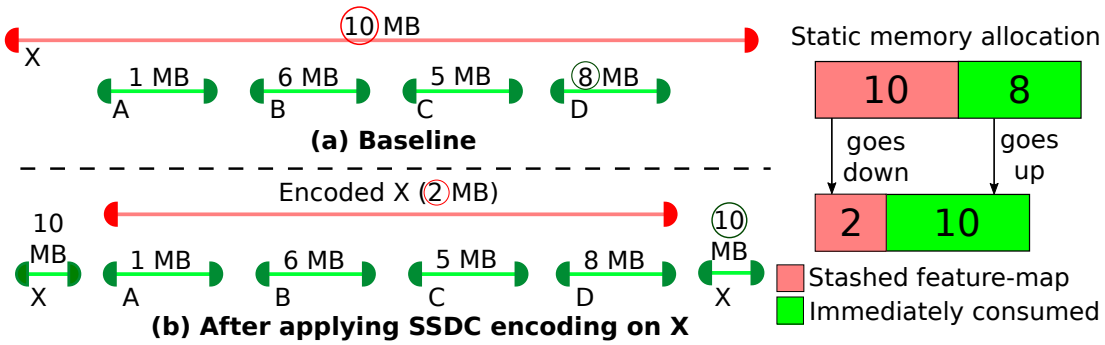


Figure 6.7: Example illustrating the interaction between ACME encodings and CNTK static memory allocator

the major contributors to the total memory footprint, hence CNTK baseline consists of only these two data structures. It does not include weights, weight gradients, and workspace (also in line with previous work on DNN training [128, 129]).

We also use a second baseline to study the effect of different encodings in isolation. This baseline, referred to as *investigation baseline*, is modified CNTK baseline where memory sharing is not allowed for stashed feature maps. Other data structures are shared exactly the same way as in the CNTK baseline. This baseline allows us to study the impact of our encodings on different data structures in isolation. For end-to-end memory reduction numbers, we still use CNTK baseline.

For performance overhead evaluation, we use memory-optimized cuDNN configuration as the focus of our work is memory footprint reduction. Memory-optimized cuDNN presents an optimized baseline for comparison. Note that CNTK baseline and investigation baseline have the same performance as they do not affect computation.

Comparison Metric. We use *Memory Footprint Ratio (MFR)* to evaluate the efficacy of ACME on reducing the memory footprint. MFR is described as follows.

$$MFR = \frac{Memory\ Footprint\ of\ Baseline}{Memory\ Footprint\ after\ encoding} \tag{6.1}$$

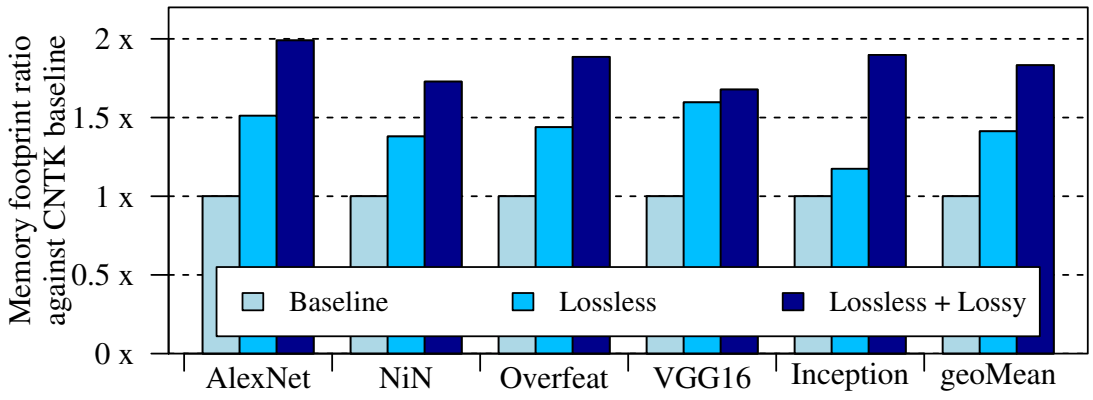


Figure 6.8: Evaluation of memory footprint reduction - ACME cuts down total memory footprint significantly

6.4.2 Gist's Memory Footprint Reduction

ACME is designed to tackle the increasing memory footprint in DNN training. In this section, we evaluate ACME's efficacy from this aspect. In our experiment we, first, apply all lossless optimizations – Binarize, SSDC and inplace – and measure the total memory footprint (stashed feature maps and immediately consumed data structures). Then, we apply ACME's lossy encoding – DPR – on top of lossless optimizations and measure additional reduction in memory footprint. For lossy encoding, we choose the smallest floating point representation that does not affect the training accuracy (detailed in Section 6.4.4.1). The findings of this experiment are presented in Figure 6.8.

Figure 6.8 shows the Memory Footprint Ratio (MFR) achieved by *Lossless* and *Lossy* optimizations when compared to CNTK *Baseline*. We observe that the lossless optimizations result in a MFR of more than 1.5× for AlexNet and VGG16 (1.4× on average). DPR, on top of lossless, further reduces the total memory footprint, achieving MFR of upto 2× for AlexNet, with an average of 1.8×. This experiment shows that ACME optimizations result in significant memory footprint reductions, making it possible to fit a network that can be twice as large (deep) compared to the current state-of-the-art.

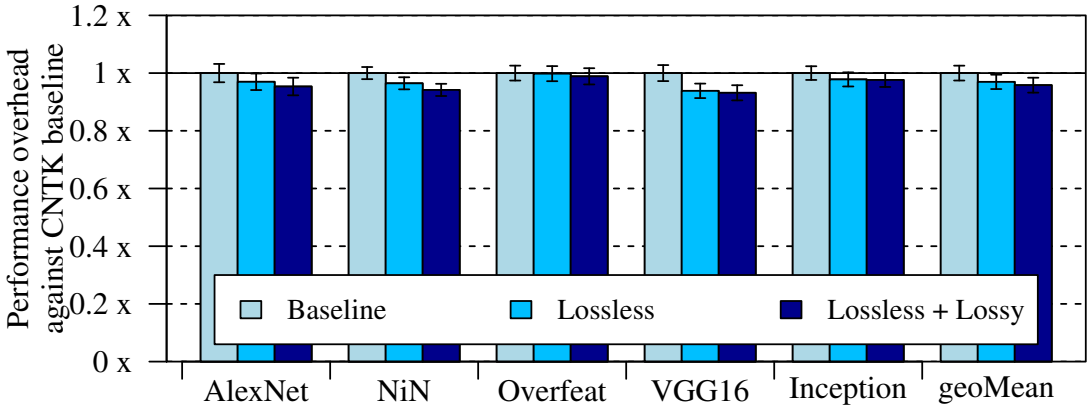


Figure 6.9: Performance overhead of ACME encodings. ACME results in minimal performance overhead

Performance Overhead. Next, we measure the performance overhead introduced by ACME encodings. We run the same experiment again, measuring the execution time of processing a minibatch, averaged over 5 minutes of training time (hundreds of minibatches). The findings of this experiment are presented in Figure 6.9. The figure shows the performance degradation for *Lossless* and *Lossy* encodings, with error bars capturing the performance variation. We observe minimal performance degradation across CNNs, resulting in an average 3% degradation for lossless and 4% for lossy and lossless optimizations combined, with a maximum overhead of 7% for VGG16 when both lossy and lossless optimizations are applied. This shows that ACME achieves significant MFR with minimal performance overhead.

6.4.3 Lossless Encodings

In this section, we evaluate the impact of lossless techniques on memory footprint and performance.

6.4.3.1 Impact on Memory Footprint

In this experiment, we apply ACME lossless encodings – Binarize and SSDC – in isolation and evaluate how they affect the memory consumed by stashed feature maps

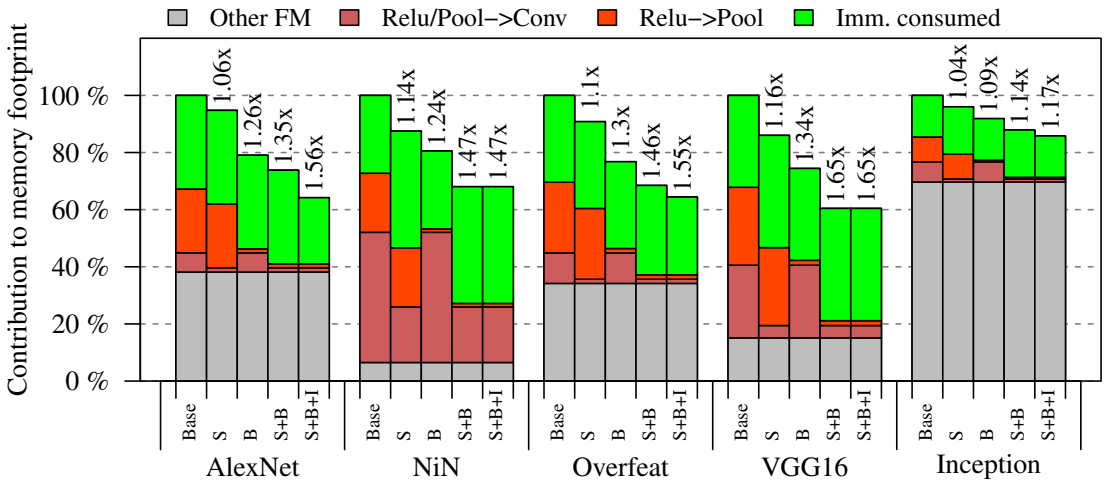


Figure 6.10: Impact of ACME lossless techniques (S - SSDC, B - Binarize, I - Inplace) on memory footprint of different data structures. Total MFR for each configuration is present at the top of each bar

and immediately consumed data structures. Then, we study the same effect when both encodings are applied together. Finally, we evaluate inplace optimization, that targets the immediately consumed data structures. The findings of this experiment are presented in Figure 6.10.

We perform this study on the *investigation baseline*, where stashed feature maps are not allowed in memory sharing, allowing us to study the impact of encodings in isolation. When an encoding is applied, the stashed feature map is converted to an immediately consumed data structure (possibly increasing its footprint), and this much smaller encoded data structure is now stashed for the backward pass, as discussed in Section 6.3.2. The figure shows this effect by breaking down the total memory footprint into 4 regions: ReLU/Pool-Conv (suitable for SSDC), ReLU-Pool (Binarize), other feature maps (untouched in this experiment as they are suitable for DPR), and immediately consumed.

The first bar shows the breakdown across these categories for the baseline. Then, we apply SSDC encoding, that reduces ReLU/Pool-Conv footprint significantly and slightly increases the immediately consumed memory footprint. For example, for

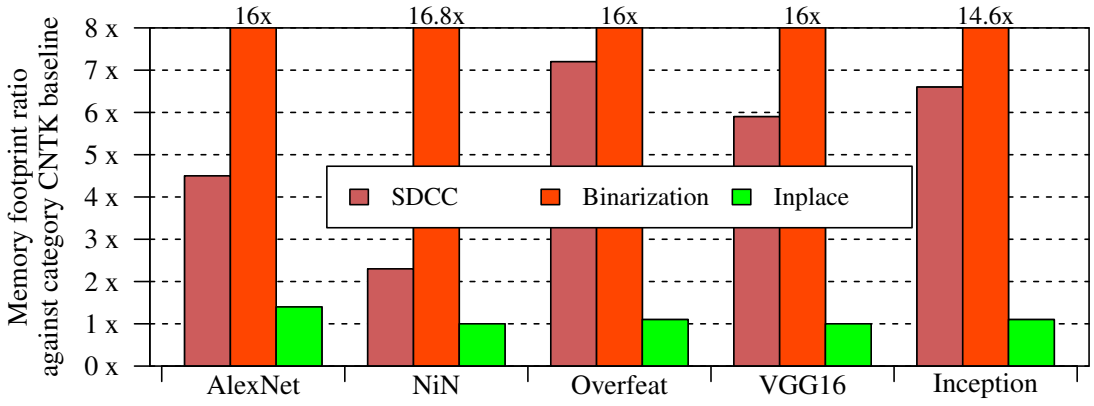


Figure 6.11: ACME lossless encodings MFR on target categories

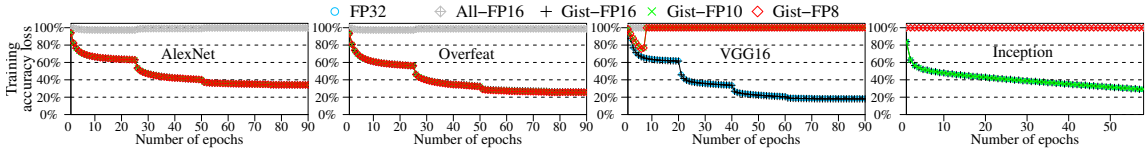


Figure 6.12: Impact of DPR Encoding on network accuracy. Smallest representation, with no accuracy loss, for AlexNet and Overfeat is FP8, for Inception is FP10 and for VGG16 is FP16; DPR achieves aggressive bit savings.

AlexNet, it results in the total MFR of 1.06×. Similarly, the third bar is for Binarize encoding, targeting ReLU-Pool category, resulting in MFR of 1.26×. Then, we apply both these encodings together, shown in the fourth bar, resulting in total of 1.35× MFR. Finally, we apply inplace optimization that targets immediately consumed data structure, further increasing the MFR to 1.56×. These techniques result in different footprint reduction for CNNs, as the proportion changes across different categories.

Note that Figure 6.10 only shows the total MFR, however ACME encodings reduce memory consumption of different categories to a different extent. We show this effect in Figure 6.11, presenting MFR for different optimizations on their target data structures. We observe that, as expected, Binarize results in significant memory savings, reaching close to 16× MFR (32× for ReLU output and at least 8× for pool output). Reduction for SSDC varies significantly across CNNs, providing upto 7× MFR for Overfeat. Finally, inplace optimization results in upto 1.4× MFR for AlexNet. Inplace optimization does not always reduce the total memory footprint, because the

affected immediately consumed data structure might not be a heavy hitter in the memory groups formed by CNTK memory allocator (Section 6.3.3).

6.4.3.2 Impact on Performance

To evaluate the performance overhead of lossless techniques, we run the previous experiment and measure the performance overhead. We observe that Binarize, instead of showing performance degradation, results in small performance improvement. This is because ReLU backward pass calculations are memory bandwidth bound and Binarize encoding increases effective memory bandwidth by representing the data in 1-bit format. We observe that SSDC encoding results in small performance overhead – upto 4% on average. The combination of two lossless encodings result in slightly better performance than just SSDC encoding alone, because of the better performing Binarize encoding. And, finally, inplace optimization has no effect on performance as it does not incur any encoding or decoding overhead.

6.4.4 Lossy Encodings

In this section, we study the impact of lossy encodings on accuracy, memory footprint, and performance.

6.4.4.1 Impact on Accuracy

First, we study the effect of applying precision reduction on the training accuracy. In this experiment, we, first, train the network in FP32 precision (shown as *Baseline-FP32*). Second, in line with previous research [63, 39, 65], we represent all the data structures throughout the network in FP16 format and then train the network (shown as *All-FP16*). Finally, we train the network using FP16, FP10 and FP8 DPR encodings (shown as *ACME-FP**). The computation is still performed with FP32 precision. The values are converted back to FP32 format just before the computation.

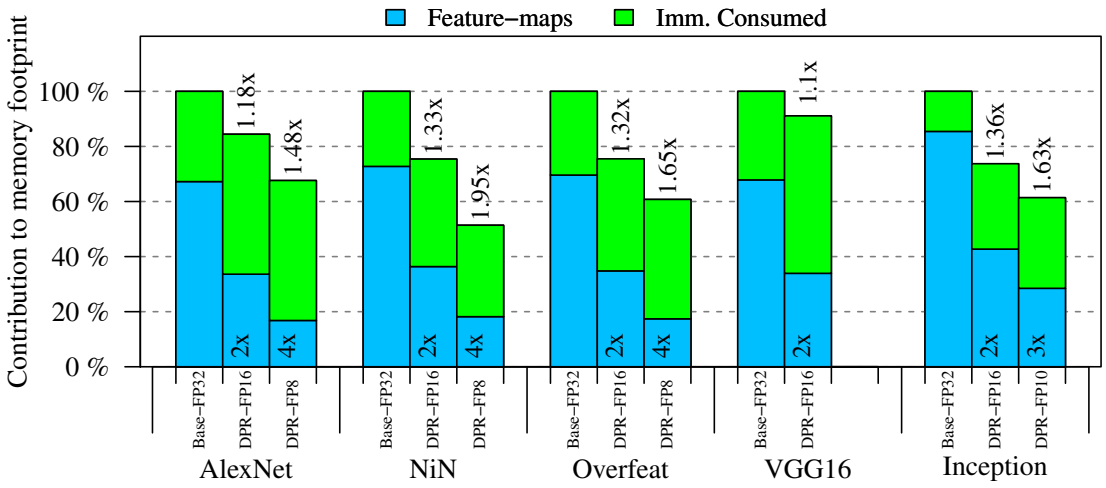


Figure 6.13: Impact of DPR encodings. Total MFR is present at the top and MFR achieved on the stashed feature maps is present at the bottom of each bar. Lowest precision for VGG16 with no loss in accuracy is FP16.

The findings of this experiment are shown in Figure 6.12.

There are two key observations from this figure. First, representing all data structures in 16 bits leads to severe accuracy losses. This is because precision reduction is applied immediately after each layer output is computed, propagating the error in the forward pass and resulting in severe accuracy losses. Second, applying precision reduction only for the backward use of stashed feature maps, as done in DPR, can result in aggressive bit savings (as low as 8 bits for AlexNet and Overfeat). For Inception, we observe that when FP8 is applied, the network stops training, but FP10 has enough precision to result in no accuracy losses. VGG16 needs highest precision, and does not train with representation smaller than FP16, showing that the minimum acceptable precision is network dependent.

6.4.4.2 Impact on Footprint Reduction

In this section, we evaluate how much MFR this efficient representations achieve. The experiment involves running DPR FP16 and the next smallest representation (FP10/FP8) that has no accuracy losses, and measuring the total memory footprint.

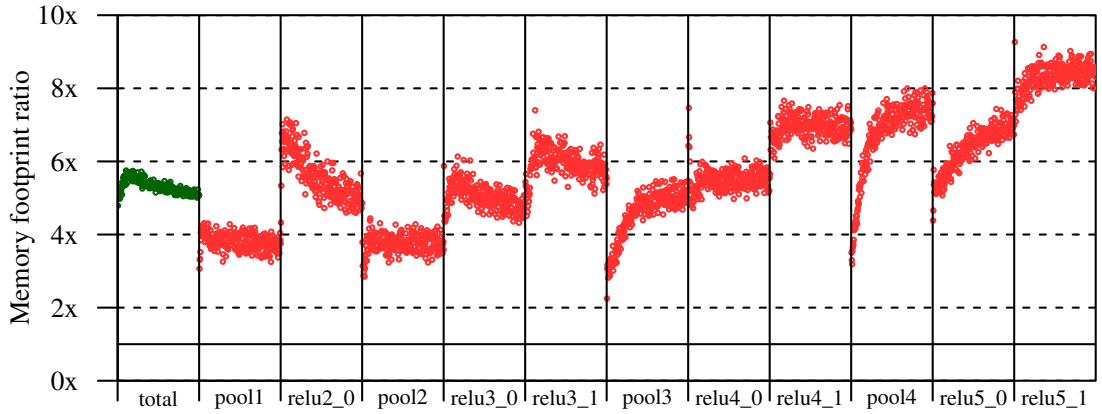


Figure 6.14: SSDC sensitivity to Sparsity (15 epochs, VGG16)

The findings of this experiment are presented in Figure 6.13, showing the MFR against investigation baseline.

DPR encoding converts the stashed feature maps into immediately consumed data and stashes the encoded feature map for the backward pass. To see this effect, we break the total memory consumption into stashed feature maps and immediately consumed. When DPR encoding uses FP16, the stashed feature maps are compressed 2×, with some increase in immediately consumed footprint, resulting in the total MFR of 1.18× for AlexNet as an example. FP8 further cuts down the memory footprint, resulting in MFR of 4× for stashed feature maps and a total of 1.48× MFR for AlexNet. As shown previously, FP8 does not result in similar accuracy as FP32 for VGG16, and thus we omit results for FP8 for VGG16.

6.4.4.3 Performance Overhead

Next, we evaluate the performance overhead of DPR encoding. For this study, we run the last experiment again and measure the execution time of a minibatch processing, averaged over 5 minutes of training. We observe that DPR encoding, being very parallel, has minimal performance overhead, with an average of 1%.

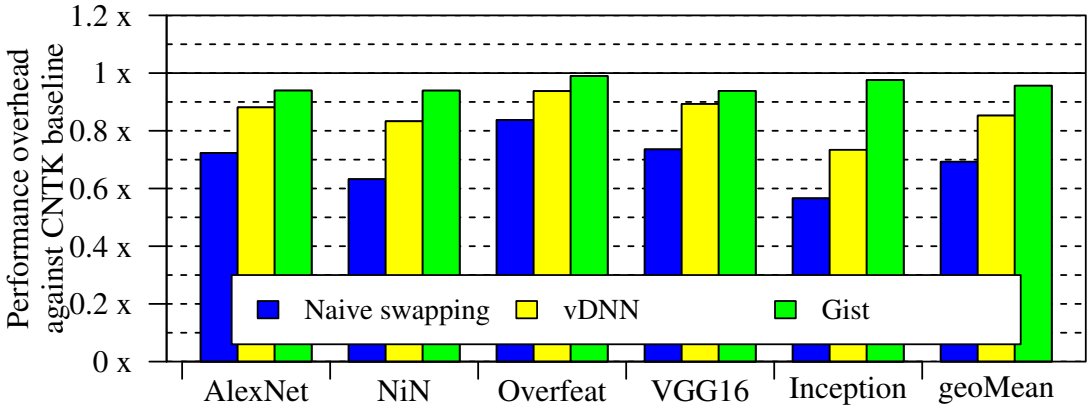


Figure 6.15: Performance comparison of ACME against naive swapping and vDNN

6.4.5 Sensitivity Study

Most of the ACME encodings result in fixed MFR, as they are agnostic to data values. However, SSDC encoding depends on the sparsity of data structure that changes during the training. In this section, we perform a sensitivity study for SSDC encoding on VGG16. The experiment involves applying the SSDC encoding, training the network for 15 epochs, while recording the achieved compression ratio for the applicable feature maps after every 1000th minibatch. The findings of this experiment are presented in Figure 6.14.

This figure shows the MFR achieved for each applicable layer (each block is a single layer) over time. We observe significant MFR across all layers, varying across the layers, and also across time within a single layer. The MFR is typically much larger than 1, except only for a small duration of first few minibatches (close to 200) of the first epoch (one epoch for VGG16 has 20K minibatches). This happens because at the start of the training, the network weights are initialized randomly. It takes few minibatches for weights to change and for sparsity to come into effect.

6.4.6 Comparison with Prior Work

Another way to reduce the memory footprint is to *swap* the parts of working set between the CPU and GPU memory using PCIe links. In this section, we compare Gist's performance with such approaches. vDNN, the most relevant prior work to Gist, is built upon this approach and uses a prefetching analysis to find suitable overlap between the data transfer and computation time [128]. We implement vDNN in CNTK and present the comparison in Figure 6.15, showing the performance overhead of naive swapping (no prefetching), vDNN and ACME against CNTK baseline.

We observe that naive swapping results in heavy performance loss, averaging 30%, because there is no overlap between kernel execution and data transfer. vDNN improves this performance by performing prefetching analysis. However, vDNN still has high overhead as the data transfer time cannot be completely hidden, resulting in an average slowdown of 15%, with a maximum of 27% for Inception. By keeping the data within GPU, ACME is not limited by the PCIe bandwidth and observes an average slowdown of only 4% (upto 7%).

6.4.7 Impact on Machine Learning Trend

A look at past ImageNet challenge winners show that networks are getting deeper over time [133]. Thus, training them incurs a higher memory footprint potentially exceeding the limited size of GPU DRAM (16 GB for the most expensive card). This means that to train deeper networks one has to use smaller minibatch sizes that fit in GPU memory, resulting in underutilized hardware and high training time. Next, in the context of training deep networks with small minibatch sizes, we show how ACME allows for faster training by enabling the use of larger minibatch sizes.

We use Resnet [75], the winner of 2015 ImageNet challenge, for this study. Resnet presents a highly composable structure, enabling us to vary the depth of the network and project this trend of deeper networks. The original Resnet paper evaluates the

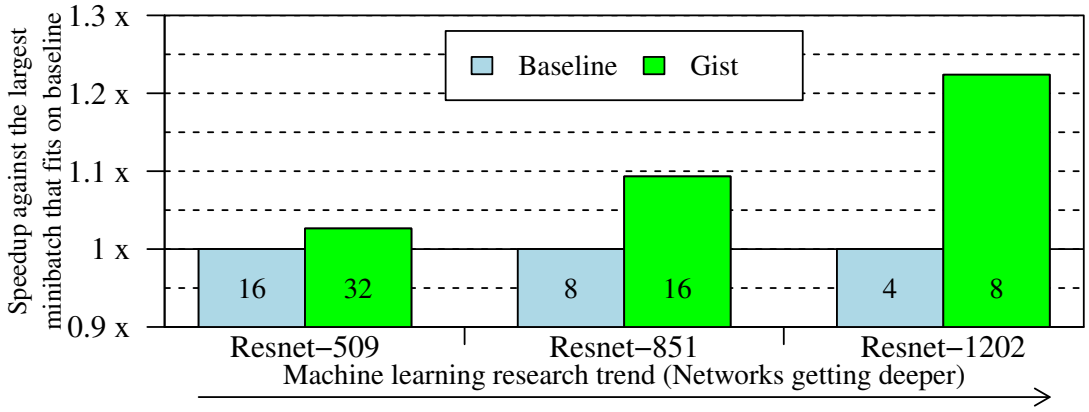


Figure 6.16: ACME enables training deeper networks with larger minibatch, achieving better performance. The largest minibatch that fits in the GPU memory is present at the bottom of the bar

accuracy to the maximum depth of 1202 layers [75]. In line with the paper, we vary the number of layers to 509, 851 and 1202. We present the findings of this experiment in Figure 6.16, showing the speedup achieved by the largest minibatch that fits with ACME compared to the largest minibatch that fits with CNTK Baseline. We observe that by enabling larger minibatches, ACME increases GPU utilization and improves training time, for e.g., a speedup of 22% for Resnet-1202. In general, due to better utilization of GPU resources with larger minibatch sizes, Gist's performance improvements positively correlate with the existing machine learning trend of deeper modern networks.

6.4.8 Discussion – Memory Allocation

Deep learning frameworks typically perform static memory allocation on GPUs to avoid expensive *cudaMalloc* calls while processing a minibatch. However, there are ongoing efforts to accelerate training on FPGAs [174] and ASICs [84, 80]. In such scenarios, dynamic memory allocation can be a preferable choice if the memory allocation is itself implemented in hardware and results in minimal performance overhead. The question then arises, in the presence of such *optimized hardware*, how much footprint reduction does dynamic memory allocation achieve, and what is the

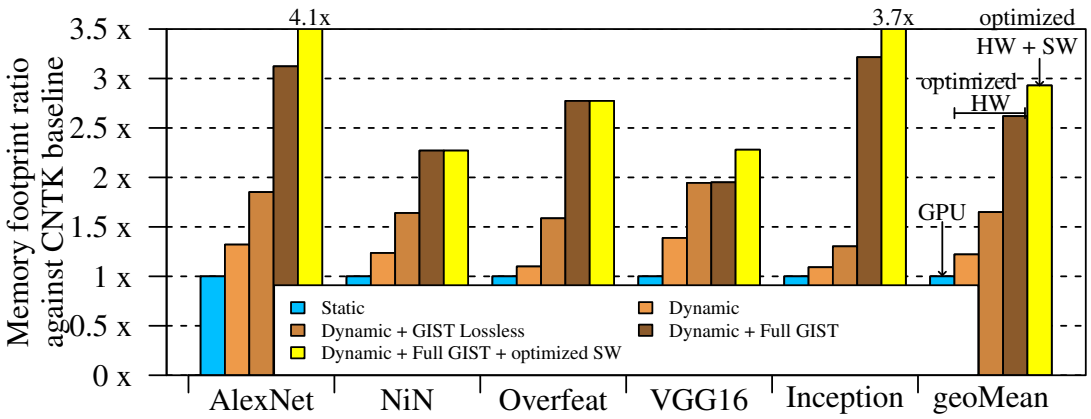


Figure 6.17: Impact of optimized hardware (dynamic allocation), ACME and optimized software (cuDNN)

impact of the ACME encodings when memory is allocated dynamically.

Second, for DPR encodings, we decode the data back into FP32 format, because cuDNN library requires data to be in the FP32 format. This requires an extra memory block (though only for immediate usage) for the decoded FP32 representation. We believe that most of this decoding can be pulled inside cuDNN implementation, which can either execute directly on the encoded data, or decode only the data that is required in near future, for example, the data required for the near-future tile in a tiled matrix multiplication. Such *optimized software* can remove the need of decoded data structure, potentially resulting in higher MFR.

In this section, we discuss the impact of such optimized hardware and software on MFR. For dynamic memory allocation, we modify the liveness analysis module from CNTK and simulate dynamic memory management, allocating a region only when it is required and relinquishing it as soon as it is dead. We find the peak memory consumed in this scheme and compare it against static memory allocation. Similarly, for *optimized software*, we modify the liveness analysis module to remove the decoded FP32 values and reallocate the memory. We present the findings in Figure 6.17.

The figure shows the achieved MFR for dynamic memory allocation, ACME encodings in presence of dynamic allocation, and optimized software with ACME en-

codings and dynamic allocation, against CNTK baseline. There are three key observations from this graph. First, dynamic memory allocation results in good MFR, going over 1.5× for Overfeat, with an average of 1.2× across all CNNs. Second, ACME encodings are still applicable in the presence of dynamic memory allocation. We observe that ACME lossless and lossy encodings achieve MFR of 1.7× and 2.6× respectively. Finally, optimized software can further cut down the memory footprint, resulting in a MFR of upto 4.1× for AlexNet against CNTK baseline, with an average of 2.9× across all CNNs.

6.5 Summary

In this paper, we investigate approaches to reduce the memory footprint of DNN training, enabling training of deeper DNNs on GPUs. We present, ACME, that employs two layer-specific lossless and one aggressive lossy encoding schemes, targeting the primary contributor to total memory footprint (feature maps). A common approach in our encodings is to store an encoded representation of feature maps and decode this data in the backward pass; the full-fidelity feature maps are used in the forward pass and relinquished immediately. ACME reduces the memory footprint by 2× across 5 state-of-the-art image classification DNNs, with an average of 1.8× with only 4% performance overhead and no effect on training accuracy.

CHAPTER VII

Conclusion

Emerging applications are data intensive, operating on a large amount of data, putting high pressure on the memory subsystem. In this thesis, we find that the pressure is not restricted to one memory structure. The pressure can be present at any level in the memory hierarchy - off-chip memory, on-chip memory or physical register file. This dissertation addresses four such memory bottlenecks spread across the memory hierarchy, reducing memory pressure for the emerging applications.

First, we find that the physical register file bandwidth is the primary bottleneck in improving the performance of CPUs on DNNs. To solve this problem, we present LEDL, locality extensions for deep learning on CPUs, that entails a rearchitected FMA and PRF design tailored for the heavy data reuse inherent in DNN inference. Second, we observe that many floating-point applications have a large number of marginal bits that do not contribute to the application accuracy, wasting unnecessary space and transfer costs. To remove these marginal bits from the memory, we present ACME, an asymmetric compute-memory paradigm, that stores data in concise format in the memory while keeping the computation in full precision. Third, we find that static compiler optimizations like cache tiling, that are intimately linked to the resource availability, need rethinking in post multi-core era, where resource availability can change at runtime. To adapt the cache tiling to the runtime, we

present ShapeShifter that continuously monitors the runtime environment, detects changes in the cache availability and dynamically retiles the application on the fly to efficiently utilize the cache capacity, with minimal performance overhead. Lastly, we find that the GPU DRAM size is the primary bottleneck that limits DNN training as the DNNs get deeper and larger. To address this problem, we perform a detailed breakdown of the DNN training memory footprint and present *Gist*, a runtime system, that uses three efficient data encoding techniques to reduce the footprint of the heaviest contributors in DNN training.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Intel Math Kernel Library. In *http://software.intel.com/en-us/articles/intel-mkl/*.

[2] NervanaGPU library. In *https://github.com/NervanaSystems/nervanagpu*.

[3] An introduction to the intel quickpath interconnect. In *http://www.intel.com/content/www/us/en/io/quickpath-technology/quickpath-technology-general.html*, 2009.

[4] Virtualization is coming to a platform near you. In *https://www.arm.com/files/pdf/System-MMU-Whitepaper-v8.0.pdf*, 2011.

[5] Intel advanced encryption standard (aes) new instructions set. 2012.

[6] AMD64 architecture programmer's manual. 2013.

[7] Amazon EC2 Spot Instances. `http://aws.amazon.com/ec2/purchasing-options/`, 2016. Online; accessed 5-Aug-2016.

[8] Nvidia nvlink high-speed interconnect. In *http://www.nvidia.com/object/nvlink.html*, 2016.

[9] Nvidia nvlink high-speed interconnect. In *http://www.nvidia.com/object/nvlink.html*, 2016.

[10] ARM architecture reference manual. 2017.

[11] Intel 64 and ia-32 architectures software developer's manual. In *Volume 3*, 2017.

[12] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, GA, 2016. USENIX Association.

[13] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[14] A. Arelakis and P. Stenstrom. Sc2: A statistical compression cache scheme. In *International Symposium on Computer Architecuture (MICRO)*, 2014.

[15] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis. Parallel frame rendering: Trading responsiveness for energy on a mobile gpu. In *Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[16] W. Baek and T. M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Programming Language Design and Implementation (PLDI)*, 2010.

[17] S. S. Baghsorkhi, N. Vasudevan, and Y. Wu. Flexvec: Auto-vectorization for irregular loops. In *Programming Language Design and Implementation (PLDI)*, 2016.

[18] B. Bao and C. Ding. Defensive loop tiling for shared cache. In *Code Generation and Optimization (CGO)*, 2013.

[19] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, 2017.

[20] L. A. Barroso, J. Clidaras, and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 2013.

[21] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The Gem5 simulator. In *SIGARCH Computer Architecture News*, 2011.

[22] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Programming Language Design and Implementation (PLDI)*, 2008.

[23] S. Borkar and A. A. Chien. The future of microprocessors. In *Communications of the ACM*, 2011.

[24] J. Bornholt, T. Mytkowicz, and K. S. McKinley. Uncertain<T>: A first-order type for uncertain data. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[25] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[26] M. Carbin, S. Misailovic, and M. C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2013.

[27] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[28] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1994.

[29] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *International Symposium on Workload Characterization (ISWC)*, 2009.

[30] C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *Code Generation and Optimization (CGO)*, 2005.

[31] 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 *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[32] T. Chen, B. Xu, C. Zhang, and C. Guestrin. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174, 2016.

[33] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. Dadiannao: A machine-learning supercomputer. In *International Symposium on Microarchitecture (MICRO)*, 2014.

[34] Y.-H. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[35] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cuDNN: Efficient primitives for deep learning. In *arXiV:1410.0759*, 2014.

[36] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *Operating Systems Design and Implementation (OSDI)*, 2014.

[37] V. Chippa, S. Chakradhar, K. Roy, and A. Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In *Design Automation Conference (DAC)*, 2013.

[38] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Programming Language Design and Implementation (PLDI)*, 1995.

[39] M. Courbariaux, Y. Bengio, and J. David. Low precision arithmetic for deep learning. *CoRR*, abs/1412.7024, 2014.

[40] M. Courbariaux, Y. Bengio, and J.-P. David. Low precision arithmetic for deep learning. In *arXiv:1412.7024*, 2014.

[41] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 4. ACM, 2016.

[42] Y. L. Cun, B. Boser, J. S. Denker, R. E. Howard, W. Habbard, L. D. Jackel, and D. Henderson. Advances in neural information processing systems 2. chapter Handwritten Digit Recognition with a Back-propagation Network, pages 396–404. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[43] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. RAPL: memory power estimation and capping. In *International Symposium on Low-Power Electronics and Design (ISLPED)*, 2010.

[44] C. De Sa, M. Feldman, C. Ré, and K. Olukotun. Understanding and optimizing asynchronous low-precision stochastic gradient descent. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, New York, NY, USA, 2017.

[45] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[46] A. Delmas, P. Judd, S. Sharify, and A. Moshovos. Dynamic stripes: Exploiting the dynamic precision requirements of activation values in neural networks. *CoRR*, abs/1706.00504, 2017.

[47] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. Whaley, and K. Yelick. Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 2005.

[48] L. Deng and D. Yu. Deep learning: Methods and applications. Technical report, 2014.

[49] M. Dukhan. NNPACK: Acceleration package for neural networks on multi-core cpus. In *https://github.com/Maratyszcza/NNPACK*, 2016.

[50] M. Ektefa, S. Memar, F. Sidi, and L. S. Affendey. Intrusion detection using data mining techniques. In *2010 International Conference on Information Retrieval Knowledge Management (CAMP)*, pages 200–203, March 2010.

[51] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[52] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *International Symposium on Microarchitecture (MICRO)*, 2012.

[53] F. Franchetti, S. Kral, J. Lorenz, and C. W. Ueberhuber. Efficient utilization of simd extensions. In *Proceedings of the IEEE*, 2005.

[54] C. Gao, A. Gutierrez, M. Rajan, R. G. Dreslinski, T. Mudge, and C.-J. Wu. A study of mobile device utilization. In *International Symposium on the Performance Analysis of Systems and Software (ISPASS)*, 2015.

[55] K. Gillespie, H. R. Fair, C. Henrion, R. Jotwani, S. Kosonocky, R. S. Orefice, D. A. Priore, J. White, and K. Wilcox. Steamroller: An x86-64 core implemented in 28nm bulk cmos. In *Solid-State Circuits Conference (ISSCC),*, 2014.

[56] V. Govindaraju, C. H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. 2012.

[57] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[58] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Autotuning a high-level language targeted to gpu codes. In *Innovative Parallel Computing (InPar)*, 2012.

[59] E. Grefenstette, P. Blunsom, N. de Freitas, and K. M. Hermann. A deep architecture for semantic parsing. In *arXiV:1404.7296*, 2014.

[60] E. Grefenstette, P. Blunsom, N. de Freitas, and K. M. Hermann. A deep architecture for semantic parsing. *CoRR*, abs/1404.7296, 2014.

[61] T. Grosser, A. Groesslinger, and C. Lengauer. Polly – performing polyhedral optimizations on a low-level intermediate representation. In *Parallel Processing Letters*, 2012.

[62] A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves. Memory-efficient backpropagation through time. *CoRR*, abs/1606.03401, 2016.

[63] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, 2015.

[64] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. In *arXiv:1502.02551*, 2015.

[65] P. Gysel, M. Motamedi, and S. Ghiasi. Hardware-oriented approximation of convolutional neural networks. *CoRR*, abs/1604.03168, 2016.

[66] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. EIE: Efficient inference engine on compressed deep neural network. In *International Symposium on Computer Architecture (ISCA)*, ISCA '16, 2016.

[67] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR*, abs/1510.00149, 2015.

[68] S. Han, J. Pool, J. Tran, and W. J. Dally. Learning both weights and connections for efficient neural networks. In *Neural Information Processing Systems (NIPS)*, 2015.

[69] S. Han, J. Pool, J. Tran, and W. J. Dally. Learning both weights and connections for efficient neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems*, NIPS'15, pages 1135–1143, Cambridge, MA, USA, 2015. MIT Press.

[70] S. Han, J. Pool, J. Tran, and W. J. Dally. Learning both weights and connections for efficient neural networks. In *arXiv:1506.02626*, 2015.

[71] S. J. Hanson and L. Pratt. Advances in neural information processing systems 1. chapter Comparing Biases for Minimal Network Construction with Back-propagation, pages 177–185. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1989.

[72] B. Hassibi and D. G. Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in Neural Information Processing Systems 5, [NIPS Conference]*, pages 164–171, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[73] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. In *International Symposium on Computer Architecture (ISCA)*, 2015.

[74] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang, and J. Mars. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[75] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. 2015.

[76] P. Hill, A. Jain, M. Hill, B. Zamirai, M. Laurenzano, C.-H. Hsu, S. Mahlke, L. Tang, and J. Mars. Addressing compute and memory bottlenecks for dnn execution on gpus. In *International Symposium on Microarchitecture (MICRO)*, 2017.

[77] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, Nov 2012.

[78] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[79] J. Holewinski, R. Ramamurthi, M. Ravishankar, N. Fauzia, L.-N. Pouchet, A. Rountev, and P. Sadayappan. Dynamic trace-based analysis of vectorization potential of applications. In *Programming Language Design and Implementation (PLDI)*, 2012.

[80] https://www.blog.google/topics/google-cloud/google-cloud-offer-tpus-machine learning/. Build and train machine learning models on our new google cloud tpus, 2017.

[81] https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan x/specifications. Nvidia GeForce GTX Titan X, 2017.

[82] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. Automatic cpu-gpu communication management and optimization. In *Programming Language Design and Implementation (PLDI)*, 2011.

[83] A. Jain, P. Hill, S. C. Lin, M. Khan, M. E. Haque, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars. Concise loads and stores: The case for an asymmetric compute-memory architecture for approximation. In *International Symposium on Microarchitecture (MICRO)*, 2016.

[84] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *International Symposium on Computer Architecture (ISCA)*, ISCA '17, 2017.

[85] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, N. E. Jerger, and A. Moshovos. Proteus: Exploiting numerical precision variability in deep neural networks. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 23:1–23:12, New York, NY, USA, 2016. ACM.

[86] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos. Stripes: Bit-serial deep neural network computing. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Oct 2016.

[87] N. Kalchbrenner, E. Grefenstette, and P. Blunsom. A convolutional neural network for modelling sentences. In *arXiV:1404.2188*, 2014.

[88] N. Kalchbrenner, E. Grefenstette, and P. Blunsom. A convolutional neural network for modelling sentences. *CoRR*, abs/1404.2188, 2014.

[89] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei. Large-scale video classification with convolutional neural networks. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[90] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei. Large-scale video classification with convolutional neural networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1725–1732, June 2014.

[91] Y. Kim. Convolutional neural networks for sentence classification. In *arXiV:1408.5882*, 2014.

[92] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Neural Information Processing Systems (NIPS)*, 2012.

[93] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1991.

[94] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization (CGO)*, 2004.

[95] M. A. Laurenzano, P. Hill, M. Samadi, S. Mahlke, J. Mars, and L. Tang. Input responsiveness: using canary inputs to dynamically steer approximation. In *Programming Language Design and Implementation (PLDI)*, 2016.

[96] M. A. Laurenzano, Y. Zhang, L. Tang, and J. Mars. Protean code: Achieving near-free online code transformations for warehouse scale computers. In *International Symposium on Microarchitecture (MICRO)*, 2014.

[97] A. Lavin. Fast algorithms for convolutional neural networks. 2015.

[98] Y. LeCun, J. S. Denker, and S. A. Solla. Optimal brain damage. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 598–605. Morgan-Kaufmann, 1990.

[99] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *International Symposium on Computer Architecture (ISCA)*, 2009.

[100] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu. Adaptive-latency DRAM: Optimizing DRAM timing for the common-case. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 489–501, 2015.

[101] D. Li, M. Rhu, D. R. Johnson, M. O'Connor, M. Erez, D. Burger, D. S. Fussell, and S. W. Redder. Priority-based cache allocation in throughput processors. In *High Performance Computer Architecture (HPCA)*, 2015.

[102] J. Li, D. Jurafsky, and E. H. Hovy. When are tree structures necessary for deep learning of representations? In *arXiV:1503.00185*, 2015.

[103] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *International Symposium on Microarchitecture (MICRO)*, 2009.

[104] J. Lifflander, S. Krishnamoorthy, and L. V. Kale. Optimizing data locality for fork/join programs using constrained work stealing. In *High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[105] R. LiKamWa, Y. Hou, J. Gao, M. Polansky, and L. Zhong. Redeye: Analog convnet image sensor architecture for continuous mobile vision. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[106] M. Lin, Q. Chen, and S. Yan. Network in network. In *arXiV:1312.4400*, 2013.

[107] M. D. Linderman, M. Ho, D. L. Dill, T. H. Meng, and G. P. Nolan. Towards program optimization through automated analysis of numerical precision. In *Code Generation and Optimization (CGO)*, 2010.

[108] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: Saving dram refresh-power through critical data partitioning. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[109] C. Lomont. Introduction to intel advanced vector extensions. In *Intel White Paper*, 2011.

[110] N. M. Ravindra. International technology roadmap for semiconductors (ITRS) symposium. In *Journal of electronic materials*, 2001.

[111] J. Mars and L. Tang. Whare-map: heterogeneity in homogeneous warehouse-scale computers. In *International Symposium on Computer Architecture (ISCA)*, 2013.

[112] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *International Symposium on Microarchitecture (MICRO)*, 2011.

[113] S. Mehta, G. Beeraka, and P.-C. Yew. Tile size selection revisited. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2013.

[114] J. Meng, S. Chakradhar, and A. Raghunathan. Best-effort parallel execution framework for recognition and mining applications. In *International Symposium on Parallel and Distributed Processing (IPDPS)*, 2009.

[115] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. Cacti 6.0: A tool to model large caches. *HP Laboratories*, 2009.

[116] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for simd. In *Programming Language Design and Implementation (PLDI)*, 2006.

[117] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. Chung. Accelerating deep convolutional neural networks using specialized hardware. In *HotChips*, 2015.

[118] 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 44th Annual International Symposium on Computer Architecture*, ISCA '17, New York, NY, USA, 2017.

[119] Y. Park, J. J. K. Park, H. Park, and S. Mahlke. Libra: Tailoring simd execution using heterogeneous hardware and dynamic configurability. In *International Symposium on Microarchitecture (MICRO)*, 2012.

[120] D. A. Patterson and J. L. Hennessy. *Computer organization and design: the hardware/software interface*. 2013.

[121] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Base-delta-immediate compression: Practical data compression for on-chip caches. In *Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[122] G. Pleiss, D. Chen, G. Huang, T. Li, L. van der Maaten, and K. Q. Weinberger. Memory-efficient implementation of densenets. *CoRR*, abs/1707.06990, 2017.

[123] L.-N. Pouchet. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.

[124] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *International Symposium on Microarchitecture (ISCA)*, 2006.

[125] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *International Symposium on Computer Architecture (ISCA)*, 2009.

[126] A. Raha, H. Jayakumar, S. Sutar, and V. Raghunathan. Quality-aware data allocation in approximate dram. In *Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2015.

[127] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[128] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[129] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, and S. W. Keckler. Compressing DMA engine: Leveraging activation sparsity for training deep neural networks. *CoRR*, abs/1705.01626, 2017.

[130] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Programming language design and implementation (PLDI)*, 1998.

[131] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning assistant for floating-point precision. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[132] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, Cambridge, MA, USA, 1988.

[133] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, and F. Li. Imagenet large scale visual recognition challenge. *CoRR*, abs/1409.0575, 2014.

[134] M. A. Russell. Mining the social web: Data mining facebook, twitter, linkedin, google+, github, and more. " O'Reilly Media, Inc.", 2013.

[135] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. In *Programming Language Design and Implementation (PLDI)*, 2011.

[136] A. Sampson, J. Nelson, K. Strauss, and L. Ceze. Approximate storage in solid-state memories. In *International Symposium on Microarchitecture (MICRO)*, 2013.

[137] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze. Expressing and verifying probabilistic assertions. In *Programming Language Design and Implementation (PLDI)*, 2014.

[138] J. San Miguel, J. Albericio, A. Moshovos, and N. Enright Jerger. Doppelganger: A cache for approximate computing. In *International Symposium on Microarchitecture (MICRO)*, 2015.

[139] J. San Miguel, M. Badr, and N. Jerger. Load value approximation. In *International Symposium on Microarchitecture (MICRO)*, 2014.

[140] D. Sanchez and C. Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In *International Symposium on Computer Architecture (ISCA)*, 2011.

[141] S. Sardashti, A. Seznec, and D. A. Wood. Skewed compressed caches. In *International Symposium on Microarchitecture (MICRO)*, 2014.

[142] S. Sardashti and D. A. Wood. Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching. In *International Symposium on Microarchitecture (MICRO)*, 2013.

[143] V. Sarkar and N. Megiddo. An analytical model for loop tiling and its solution. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2000.

[144] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Can traditional programming bridge the ninja performance gap for parallel computing applications? In *ACM SIGARCH Computer Architecture News*, volume 40, pages 440–451. IEEE Computer Society, 2012.

[145] F. Seide and A. Agarwal. CNTK: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 2135–2135, New York, NY, USA, 2016. ACM.

[146] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. In *arXiv:1312.6229*, 2014.

[147] S. Sharify, A. D. Lascorz, P. Judd, and A. Moshovos. Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks. *CoRR*, abs/1706.07853, 2017.

[148] K. Simonyan and A. Zisserman. Two-stream convolutional networks for action recognition in videos. In *arXiV:1406.2199*, 2014.

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

[150] S. Srinath, B. Ilbeyi, M. Tan, G. Liu, Z. Zhang, and C. Batten. Architectural specialization for inter-iteration loop dependence patterns. In *International Symposium on Microarchitecture (MICRO)*, 2014.

[151] J. Srinivas, W. Ding, and M. Kandemir. Reactive tiling. In *Code Generation and Optimization (CGO)*, 2015.

[152] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *arXiV:1409.3215*, 2014.

[153] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer. Efficient processing of deep neural networks: A tutorial and survey. In *arXiV:1703.09039*, 2017.

[154] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[155] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. 2014.

[156] S. Tavarageri, L. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. Dynamic selection of tile sizes. In *High Performance Computing (HiPC)*, 2011.

[157] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun. Fast convolutional nets with fbfft: A GPU performance evaluation. In *arXiV:1412.7580*, 2014.

[158] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan. Scaledeep: A scalable compute architecture for learning and evaluating deep networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 13–26, New York, NY, USA, 2017. ACM.

[159] R. K. Venkatesan, S. Herr, and E. Rotenberg. Retention-aware placement in dram (rapid): software methods for quasi-non-volatile dram. In *High-Performance Computer Architecture (HPCA)*, 2006.

[160] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. *CoRR*, abs/1411.4555, 2014.

[161] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. In *arXiV:1411.4555*, 2014.

[162] I. Wallach, M. Dzamba, and A. Heifets. Atomnet: A deep convolutional neural network for bioactivity prediction in structure-based drug discovery. 2015.

[163] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 2001.

[164] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Programming Language Design and Implementation (PLDI)*, 1991.

[165] M. Wolfe. More iteration space tiling. In *Conference on Supercomputing (SC)*, 1989.

[166] S. L. Xi, H. Jacobson, P. Bose, G. Y. Wei, and D. Brooks. Quantifying sources of error in mcpat and potential impacts on architectural studies. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[167] J. Xiong, J. Johnson, R. Johnson, and D. Padua. SPL: A language and compiler for dsp algorithms. In *Programming Language Design and Implementation (PLDI)*, 2001.

[168] A. Yazdanbakhsh, D. Mahajan, B. Thwaites, J. Park, A. Nagendrakumar, S. Sethuraman, K. Ramkrishnan, N. Ravindran, R. Jariwala, A. Rahimi, H. Esmaeilzadeh, and K. Bazargan. Axilog: Language support for approximate hardware design. In *Design, Automation and Test in Europe (DATE)*, 2015.

[169] A. Yazdanbakhsh, J. Park, H. Sharma, P. Lotfi-Kamran, and H. Esmaeilzadeh. Neural acceleration for gpu throughput processors. In *International Symposium on Microarchitecture (MICRO)*, 2015.

[170] J. Yosinski, J. Clune, A. M. Nguyen, T. J. Fuchs, and H. Lipson. Understanding neural networks through deep visualization. In *arXiV:1506.06579*, 2015.

[171] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 548–560, New York, NY, USA, 2017. ACM.

[172] T. Yuki, L. Renganarayanan, S. Rajopadhye, C. Anderson, A. E. Eichenberger, and K. O'Brien. Automatic creation of tile size selection models. In *Code Generation and Optimization (CGO)*, 2010.

[173] W. Zaremba, I. Sutskever, and O. Vinyals. Recurrent neural network regularization. In *arXiV:1409.2329*, 2014.

[174] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. In *Proceedings of the 35th International Conference on Computer-Aided Design*, ICCAD '16, pages 12:1–12:8, New York, NY, USA, 2016. ACM.

[175] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *International Symposium on Field-Programmable Gate Arrays*, 2015.

[176] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang. Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers. In *International Symposium on Microarchitecture (ISCA)*, 2014.

[177] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *International Symposium on Computer Architecture (ISCA)*, 2009.