HARDWARE AND SOFTWARE OPTIMIZATIONS FOR DEEP LEARNING WORKLOADS ON GRAPHICS PROCESSING UNITS

Justin Matthew Garrigus

Thesis Prepared for the Degree of

MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

December 2024

APPROVED:

Hui Zhao, Major Professor
Song Fu, Committee Member
Kirill Morozov, Committee Member
Gergely Záruba, Chair of the Department of Computer Science and Engineering
Paul Krueger, Dean of the College of Engineering
Victor Prybutok, Dean of the Toulouse Graduate School

TABLE OF CONTENTS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

This thesis is an exploratory endeavor centered around optimizing the hardware and software of graphics processing units (GPUs) for deep learning training and inference. Usage of GPUs for deep learning was first popularized in the 2012 ImageNet competition by the AlexNet model [28] which exploited the massive parallelization capabilities of GPUs to train an image-classification convolutional neural network. Since then, the usage of GPUs for both training and inference of deep-learning tasks has skyrocketed [15, 56, 20, 57, 61, 55]. To meet these demands, researchers aim to reduce latency and increase throughput of training and inference tasks by modifying software to better utilize existing resources and adjusting hardware to make available new resources.

This thesis describes work on three areas of improvement for GPUs: chapter 1 gives an introduction to the relevant fields in deep learning and GPU acceleration; chapter 2 describes recent research in compilation techniques for deep-learning models and classifies different optimization approaches; chapter 3 introduces a new hardware modification to accelerate image-recognition workloads; chapter 4 characterizes the utilization of different hardware modules in distributed GPUs for training large language models (LLMs); and chapter 5 concludes the thesis by summarizing each development and offering future research directions.

1.1. Graphics Processing Unit

Graphics processing units (GPUs) are massively-parallel processors that can accelerate certain types of large-scale, homogeneous computations. GPUs work best when the computation consists of repetitive operations across arrays, typically found in 3D graphics applications [13], deep learning [28, 55], physics simulations [18, 17], and more [48], but techniques like CUDA Dynamic Parallelism can be used on recursive or tree-like data [66, 62] as well.

Historically, GPUs were first used to accelerate 3D graphics applications. These

applications consist of thousands of triangles in a 3D world-space which are translated to a 2D screen-space and drawn with textures, shadows, occlusion, and more [13]. The first 3D graphics accelerators were made in 1968 by the Evans and Sutherland Computer Corporation for flight simulators [5], and the next milestone occurred in the 1980s by Silicon Graphics [10]. Although, these types of processors still kept a significant portion of the graphics workload on the CPU. A shift occurred in 1999 with NVIDIA's GeForce 256 processor which was the first to name itself a "graphics processing unit" [10]. This was characterized by moving more components of the graphics pipeline offline, including the world-space-to-screen-space transformation and the drawing of lighting and shadows to textures. In the realm of computer graphics, NVIDIA, AMD, and Intel are each developing their own brands of graphics processing units [47, 14, 31].

Another shift happened in 2003 when GPUs were used for computing general linear-algebra expressions on vectors and matrices, which was shown to be beneficial in numerical computation and fluid dynamics simulations [30]. Before this, general-purpose (non-graphics) usage of GPUs required an intimate understanding of graphics and shading algorithms, and this new framework motivated the development in 2004 of the general-purpose GPU (GPGPU) programming language CUDA [10]. Interest in GPGPUs escalated after the image-recognition neural network AlexNet won the ImageNet competition in 2012 by using two NVIDIA GPUs trained over the course of two weeks [28]. Without the high throughput that GPUs offer, training a model with 60 million parameters would have been intractable on CPUs alone. Since then, libraries for accelerating general-purpose applications with GPUS have been developed like cuBLAS for linear algebra, cuDNN for deep learning, and cuFFT for signal processing [10].

On an architectural level, a GPU is a special kind of single-instruction multiple-data (SIMD) vector processor. Vector processors are efficient at executing singular instructions on successive items in a vector [16]: since the instructions are repetitive, the processor (1) only needs to fetch a single instruction to operate on lists of data, (2) it can batch together data loads and stores, and (3) it can pipeline instruction start-up times. Each vector instruction

consists of a pipeline of sub-instructions that take a constant amount of clock cycles to complete (in which the length of the pipeline determines the start-up time). Therefore, successive operations flow through the pipeline one cycle at a time, outputting one result each cycle after the start-up time completes. The instruction can be further parallelized by duplicating the pipeline several times, having each pipeline operate on a constant stride from the vector's base address, allowing the processor to output more than one result each cycle. A comparison of a standard CPU pipeline with each type of vector processor pipeline is shown in figure 1.1.

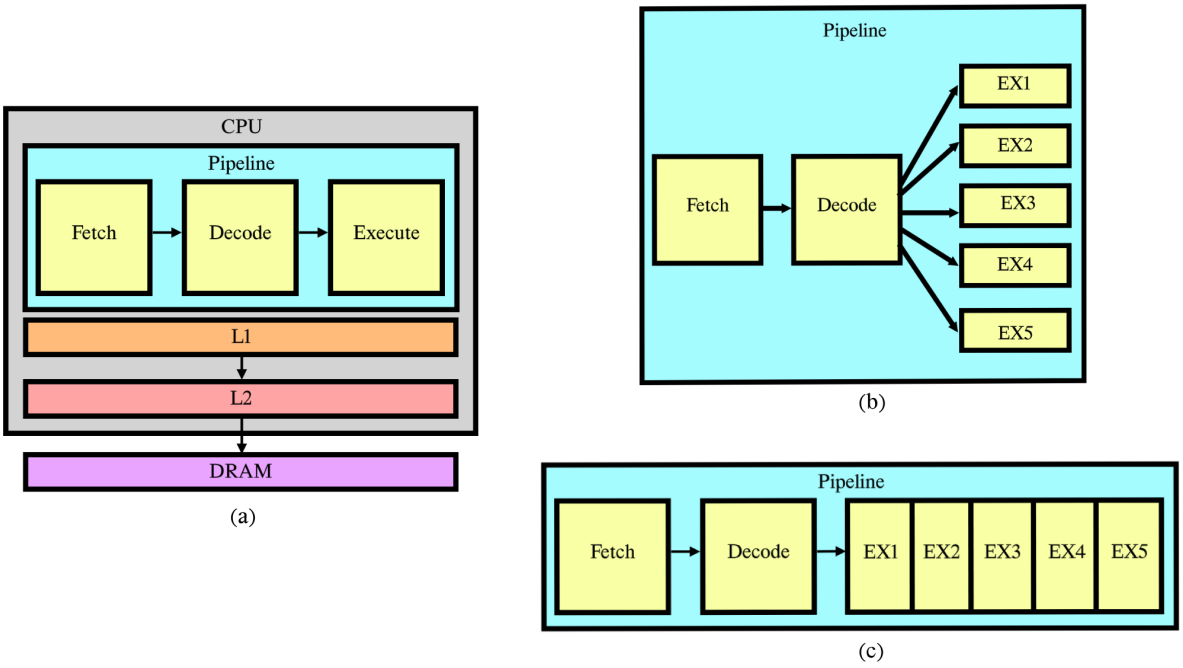


FIGURE 1.1. A comparison between (a) a standard CPU pipeline, (b) a multi-lane pipeline, and (c) a pipelined vector processor.

GPUs are a bit different than vector processors in that they do not explicitly have these vector instructions, but can still operate on vectors of data extremely efficiently. While a real vector processor might have a single instruction represent a pipelined operation against multiple elements of an array, NVIDIA GPUs have looped instructions that are distributed across a large collection of threads operating in lockstep. For figure 1.1, part (b) for GPU architectures and part (c) for vector architectures both have the same average latency. Vector

architectures will overcome their start-up times due to overlapped execution, while GPU architectures incur their start-up time each iteration while duplicating the number lanes to be equal to the pipeline length. Each can achieve a single-cycle latency on average if enough time passes or if the number of lanes is high enough.

If GPUs simply had a few hundred lanes each executing the same operation in lock-step, then it would lead to poor performance for two reasons: (1) it would require very large homogeneous vectors, which limits the use-case of GPUs significantly; and (2) it would require a very wide memory system with many banks to continually supply each lane with data. Instead, GPUs group each lane into *warps*, and they group each warp into *streaming multiprocessors* (SMs)[1]. Thus, warps can execute in lockstep while sharing L1 cache memory in a single SM, while different SMs can execute different code entirely at the same time. Figure 1.2 shows a simplified diagram of the GPU.

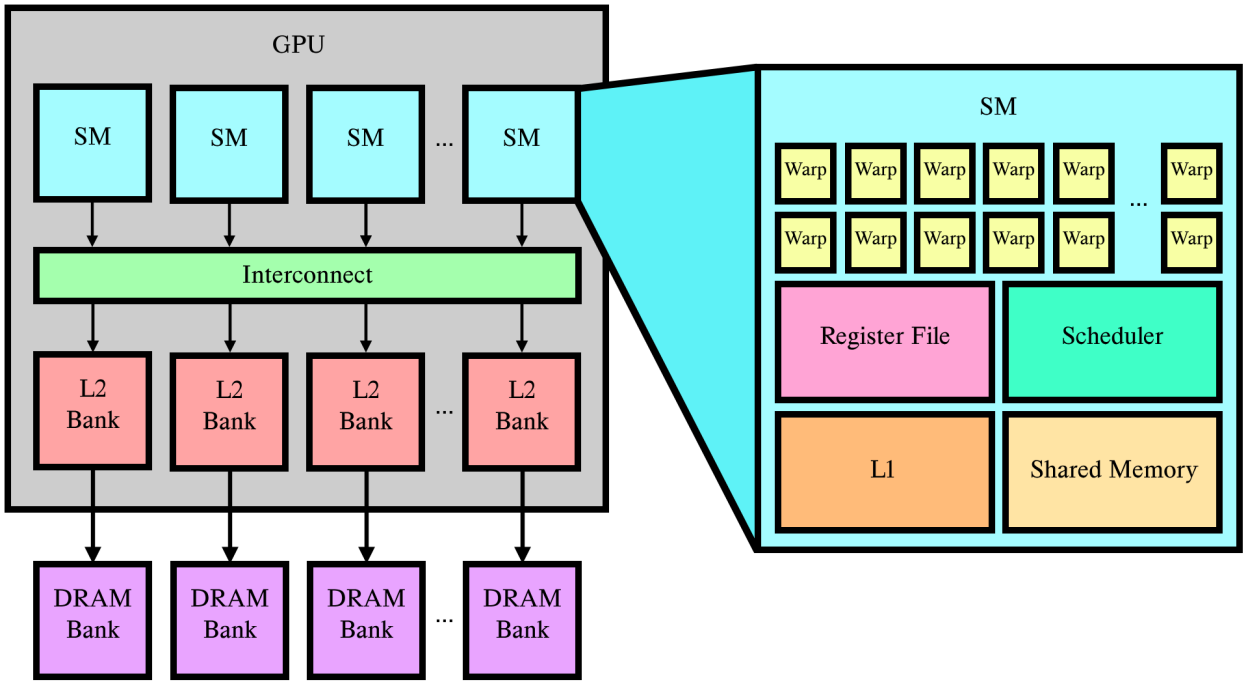


FIGURE 1.2. Schematic of a general-purpose graphics processing unit.

GPU programmers write "kernels", which are individual functions that are dispatched

[1]Note that the "warp" and "SM" terminology is specific to NVIDIA GPUs. AMD GPUs have their own terminology for nearly-identical structures which are not discussed in this work.

to the GPU at runtime. Each warp contains exactly 32 threads, meaning the best utilization is achieved when the length of input vectors are multiples of 32. It is possible for lanes within a warp to execute different instructions (which would occur when input vector lengths are not multiples of 32), but they are not able to do so at the same time; the hardware must switch its attention between the two instructions. Although, two *separate* warps within the same SM are able to execute different instructions within the same kernel at the same time, and two separate SMs can execute different instructions in different kernels at the same time. Using this, a single GPU can execute many different kinds of instructions at once, making it largely dissimilar to standard vector processors.

The efficiency of GPUs stems from the homogeneity of the workloads by virtue of them being vector-based, but all workloads come with some level of heterogeneity that GPUs adapt for as well. For example, a program might revisit some sections of memory within a single kernel while other sections of memory might be streaming (e.g., visited only once). This presents a difference in data load times due to the cache hierarchy. The GPU reconciles this difference through the use of schedulers and masking techniques: a *warp scheduler* switches out warps that are waiting for in-transit memory accesses with warps that have no stalls in progress [36], and the *SM scheduler* monitors the execution status of SMs before dispatching new kernel code to fill in gaps [73]. Other hardware structures can further aid in hiding irregular stalls in instruction types and memory accesses, like miss status holding registers (MSHRs) [29] which store the addresses of in-transit memory accesses, allowing redundant memory accesses to map to the same in-transit access and allowing new warps to execute while stalled warps are waiting for memory.

When kernels are dispatched to the GPU, the programmer specifies the number of threads (SIMD lanes) along with the number of *blocks*, in which each block is required to run on the same SM. Each block within an SM shares an L1 cache, shared memory (an explicit programmer-defined cache), tensor cores (a type of functional unit designed specifically for matrix-multiplications), and register file, allowing the programmer to group code together

into specific SMs to benefit from SM-wide synchronization alongside cache-friendly code[2]. The interconnect sits between the SMs and the lower memory system, where the L2 cache is shared between all SMs, and both the L2 cache and DRAM are banked allowing multiple accesses at the same time. Distributed GPU systems like those discussed in chapter 4 form an additional network connecting the memory of each GPU together through NVLink, a multi-GPU interconnect [37].

Finally, the CPU and GPU are connected through a PCI express bus. During runtime, the CPU sends an intermediate assembly code named PTX to each GPU, which is decoded just-in-time into the native hardware code named SASS. This just-in-time compilation allows NVIDIA to change hardware compilation techniques or add new instructions without breaking precompiled programs or modifying their user-visible PTX, allowing programmers to write code directly in PTX without worrying about portability.

1.2. Deep Learning

Historically, certain tasks like language understanding and image classification have been solved with heuristics that manually find abstractions in the input domain. These often require a human to utilize domain-specific knowledge to identify inferential rules that a branching-logic program might parse. In contrast, machine learning uses statistical methods to translate the input to the output using mathematical methods that draw inspiration from neurological processes. Instead of using a program to pick apart features directly from the input domain—like how the ELIZA chatbot searches for predefined keywords from input sentences to determine which outputs to construct based on predefined stencils [67]—machine learning utilizes some level of abstraction automatically learned through statistical methods to find parameters for a mathematical model that calculates the output from the input. Training data is fed into the model and the model is adjusted to find a set of parameters that minimizes a given cost function.

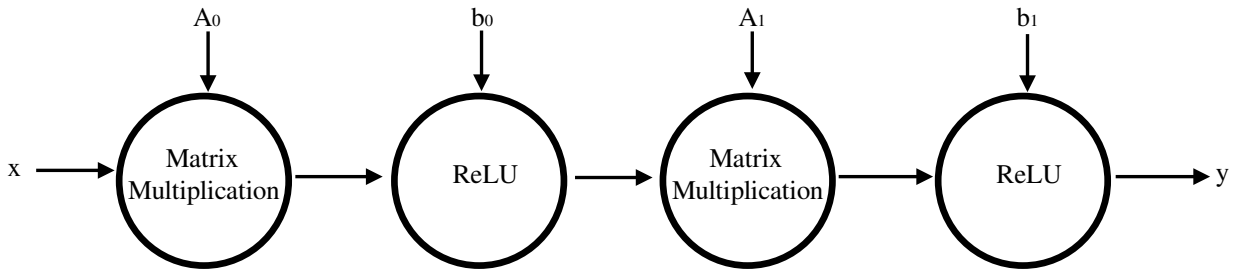
[2]It is not always the case that threads distributed across multiple SMs are faster than code that uses a single SM: if each thread loads from the same memory location, then redundant loads are significantly reduced with the MSHRs and L1 cache within an SM. This benefit is not seen across multiple SMs.

The separation between machine learning and deep learning is fuzzy, but deep learning is generally defined as an extension on machine learning with additional abstraction, especially in terms of "deeper" levels of functional composition. Deep learning models are algebraic equations consisting of the composition of sub-functions with non-linear "activation" functions separating them. These models are normally much larger and less understandable than machine learning models, as the features learned are entirely self-learned. Additionally, due to the models themselves being algebraic, they generally directly translate input examples into their respective outputs with minimal non-mathematical processes being applied in-between, besides pre-processing and post-processing. In other words, while a programmer might use machine learning as an intermediate step to cluster data together or find a regression equation to predict values, deep learning completely transforms the input into the output end-to-end. While machine learning is a superset of deep learning, this thesis focuses on deep learning specifically due to its large computational and memory requirements.

An example of a deep-learning model is shown in figure 1.3 alongside its mathematical representation. The main operators represent standard linear-algebra operations like matrix multiplication while the activation functions must be *non-linear*. The reason for this non-linearity can be expressed through single-layer perceptrons, the standard model for neural networks: if a model has no non-linearity (e.g., it contains no activation functions), then the composition of linear functions would therefore be linear (e.g., the output can be represented as a linear combination of the inputs). Furthermore, any linear model, regardless of its complexity or depth, would have an equivalent single-layer perceptron model. In other words, *any* composition of linear functions is equivalent to a matrix multiplication and addition. This has significant downsides in not only a lack of complexity, but also an outright inability to compute certain kinds of problems like XOR, motivating the inclusion of non-linear activation functions. Despite the requirement[3] that the model be optimized layer-by-

[3]The chain rule calculates derivatives one layer at a time, and an iterative method like gradient descent is required because the size of the dataset and number of trainable parameters makes a closed-form solution intractable to calculate. Regardless, iterative methods like gradient descent tend to produce better results than directly-computed global minimums since global minimums tend to overfit [9, 64].

layer, the improved ability to represent complex domains is a large improvement.[4]



$$y = \max_{ij}\{0, A_1 \times \max_{ij}\{0, A_0 \times x + b_0\} + b_1\}$$

FIGURE 1.3. Example deep-learning computational graph with an associated mathematical representation. The ReLU operation [43] is an element-wise maximum of zero and each value within the matrix.

The theory behind deep learning was prevalent throughout much of the 20th century, but it was not until 1990 with the development of LeNet that the field gained practical attraction [65]. LeNet was used by the United States Postal Service to recognize handwritten zip codes and demonstrated the utility of convolutional neural networks in image recognition [35]. Later, LSTMs were shown in 1997 to be a useful tool in natural-language understanding, especially in solving a common problem with vanishing gradients caused by recurrent networks [19]. Deep learning models using these and other techniques were successfully used in solving complex problems, but their popularity waned over time [54].

The 2012 ImageNet competition signaled the biggest shift in public opinion over deep neural networks: the winner, AlexNet, was the first in the competition's history to receive an error rate less than 25%, and the second place was a staggering 9.8% behind [53]. Most other submissions used traditional machine learning techniques like SVMs and random forests that have stronger theoretical support, but the following year saw almost exclusively deep learning

[4]While the limitation of linear functions has been shown, it does not answer why linear functions are used at all; it is possible to use purely nonlinear functions to compose a deep learning model by replacing matrix multiplications with polynomials [25], but linear functions have been empirically shown to perform well enough to not warrant the effort in switching to nonlinear.

submissions. While abstraction harms general understandability of how the model achieves high performance, it mirrors how actual neurological brains work in their ability to represent complex topics.

The popularity of deep learning continues to increase today. Beyond AlexNet, the utility of image-recognition improved even further with VGG's larger size [56], ResNet's deep residual connections [15], and GoogLeNet's multi-size convolutions [58], among many others. Each of these models consist of convolutional layers that recognize spatial patterns in the input space, pooling functions that reduce the dimensionality of the data, and fully-connected layers that translate the data into the required output format. Image classification has been expanded into other domains including image segmentation, object detection, and object localization [53].

The *transformer* model was a significant leap forward in natural language understanding due to the *attention* mechanism which selectively chooses to focus on different parts of the input stream [61]. Some kinds of natural language processing networks are separated into an encoding and decoding phase, in which the *encoder* converts the variable-width input into a fixed-width hidden representation, which the *decoder* then converts into a new variable-width output. The encoder will output only a single "hidden state" which represents the entire input stream in some abstracted form. This has a problem in representing very long input streams because it must find a way to relate the importance of far-apart sections of the input while constraining the hidden state to a fixed width. The attention mechanism fixes this by obtaining a hidden representation of *each* token of the input stream—instead of only the final step—and learning to "focus" on different sections of the input by itself. This allows much longer inputs to be represented than before. Several popular language-understanding models extend the transformer structure including the encoder-only BERT model [11] and the decoder-only GPT model [51].

In the present, deep learning models have continued to experience massive growth in model sizes. Many popular models use tens of billions of parameters, and some models have surpassed trillions of parameters [40]. These come with new problems in scalability, including

in processing time, power, and cost, which accelerator designs are likely to assist in solving. In 2021, the Megatron-LM model utilized 3072 GPUs to train over 1 trillion parameters [55, 44], showing that large-scale distributed systems of accelerators are necessary. Additionally, while GPUs continue to be beneficial in both training and inference, new kinds of accelerators are increasingly being developed [8]. The study of GPU acceleration is thus beneficial in both the short-term, with creating new kinds of GPUs and software that can readily improve the inference and training time on existing systems, and in the long-term, with motivating the development of different kinds of highly-parallel systems that share similarities with GPUs.

CHAPTER 2

DEEP LEARNING COMPILERS

Essentially, a deep neural network is a computation graph consisting of operations to perform on tensors. An important characteristic of deep neural networks is that operations can be reorganized, changed, or removed entirely without changing the output of the final network. As such, standard software compilation techniques can be employed to change the GPU kernels and other functions so that they perform better on hardware. Besides these standard methods, deep learning practitioners might employ further indirect optimization techniques that trade off compilation time and repeated execution of programs to find an even faster kernel. The possible optimizations can be collected in a pool, then a compiled program can be generated from a random selection from the pool, and a final selection of the optimizations that yielded the best performance can be returned from the compiler. As such, the *deep learning compiler* is a special compiler which combines both of these techniques: *direct optimizations* that change the source code in deterministic ways and *indirect optimizations* that require multiple executions of the program during the compilation phase to converge to a faster program.

2.1. Direct Optimizations

2.1.1. General Optimizations

Most compilers have an execution flow that consists of:

(1) Taking the source code in a human-readable format, evaluating pre-processing rules in the form of macros and other meta-expressions.

(2) Scanning the text for keywords and other tokens with a grammar consisting of regular expressions (lexical analysis).

(3) Grouping tokens together into syntactic units like expressions, statements, and functions by following a formal language, forming an abstract syntax tree (AST) in the process (syntax analysis).

(4) Modifying the AST to decrease memory usage, reduce total latency, and improve other metrics (optimization).

(5) Converting the code to a separate format for later execution (machine code generation) or running the code immediately (interpretation).

While most steps are straightforward and follow regular rules that run in linear time, the optimization stage can get significantly more complex. A few basic language-agnostic optimizations are:

- Value propagation: a variable might contain a value that is used at two separate points of the program. If the value is unchanged between the two points, then the value can be shared, letting it be calculated or obtained only once instead of twice. This can significantly reduce execution time in the case of memory operations, which take up some of the most time in the program to execute.
- Operator inlining: two operations are combined into one, reducing the set-up/tear-down overhead of the second. Invoking a function usually requires overhead in the form of allocating space on the program stack, saving registers to the stack, branching to new code, and re-loading registers that were overwritten; inlining this function removes these extra operations and duplicates the code from the inlined function to the calling code.
- Strength reduction: a high-cost operation is replaced with an equivalent low-cost operation. An example is vectorization, which replaces repetitions of the same instruction with a single pipelined instruction.
- Loop restructuring: a looped code is reformatted to both reduce the overhead of iteration and to keep the hardware pipeline occupied. Loop unrolling is a simple example which copies the loop body multiple times, while other techniques like the polyhedral method will completely restructure loops in extreme ways.

An important detail is that many of these optimizations are performed strictly without running the program to see if the performance actually improved or if it accidentally degraded. This can be the case of function inlining, as inlining too many functions can

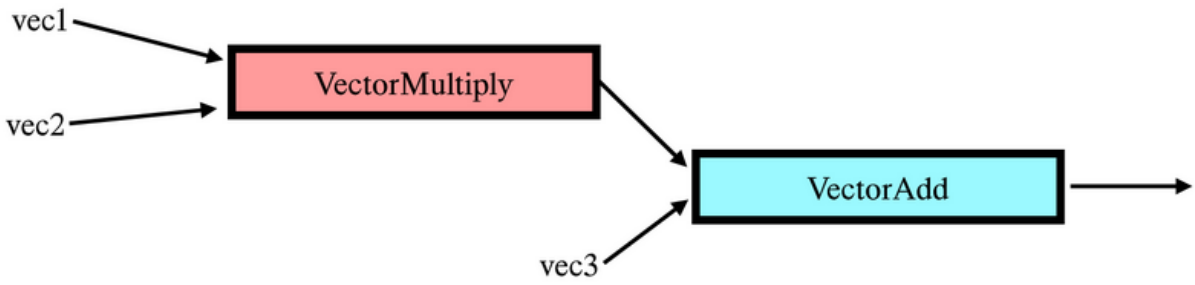
lead to a significant reduction in performance. Duplicate code would exist in the instruction cache, leading to a large amount of capacity misses and access behavior that looks more similar to streaming. In general, compilers will only make an optimization if it is clear it will lead to better performance, using heuristics and cost models to predict the impact of an optimization without actually running the code.

Several of the above optimizations are applied specifically on GPUs. For example, *tensorization* is a strength-reduction technique that replaces matrix-related code with instructions that utilize the GPU's tensor cores [12]. *Kernel fusion* is method of operator inlining that reduces the overhead involved in invoking a GPU kernel by combining two kernels together [38, 63], shown in figure 2.1. Similarly, kernel fusion also helps in value propagation; two adjacent kernels might use the same data buffer, with the first saving data to the buffer and the second loading the same data, so a combination of the kernels would lead to a reduction in the memory operations. Vertical fusion [63] is most commonly used, appending the second kernel to the first to better propagate values and eliminate startup overhead, while horizontal fusion [38]—putting two kernels in the same function, diverting some of the threads to one kernel and some to the other—is also employed in the case of independent kernels which are too small to completely use the resources on a single SM.

Deep learning generally uses the same optimization techniques as non-deep-learning applications do, but they have the added benefit of containing generalized abstractions of operators on a larger scale. In other words, deep learning compilers do not need to see the program strictly on a software level, and can instead look at the high-level operations being performed and then apply high-level optimizations as a result.

Other more specialized deep-learning optimizations are used on a fine-grained level. ALCOP is one framework which uses software pipelining in loops to overlap the computation of the current iteration with the asynchronous loading of the next iteration [21]. Other similar methods in software pipelining can be employed on a larger operator-level scale: operators which are memory-intensive are good contrasts to operators which are compute-intensive since they use different areas of the hardware, so a scheduler might try to overlap

execution such that these different types of operators run at the same time as opposed to two memory-intensive or two compute-intensive operators being overlapped. Another specialized optimization technique is VPPS, which caches the weights of dynamic recurrent networks in register files between recurrent operations [24]. Dynamic networks have a variable amount of computation depending on the input, leading to an unpredictable computation graph that makes caching difficult, and VPPS overcomes this through a just-in-time (JIT) compiler that generates code at runtime to keep model weights on-chip.



(A)

(B)

FIGURE 2.1. Demonstration of a deep learning optimization. If each cell shaded box represents a GPU kernel, (a) shows two discrete kernels and (b) shows a single GPU kernel containing both operations fused together. Fused implementations have a much better cache utilization because it prevents unnecessary data movement between kernel invocations.

2.1.2. Tensor Optimizations

Several tensor-specific optimizations can be performed by utilizing the unique mathematical properties of tensors. Generally, a tensor is a multidimensional data structure with each dimension having a consistent size, making them the abstraction of the one-dimensional vector and two-dimensional matrix but for any number of dimensions. The *order* of a tensor is the number of indices required to index a single value and a *mode* describes a single axis, meaning vectors contain one mode and matrices two modes. Tensor contraction is a popular operation that reduces the order of a tensor along a given collection of indices, and is naturally used in 1-dimensional convolution in the GoogLe Net vision network [58]. The Kronecker product between two matrices yields a three-dimensional tensor, with each element of the tensor being an element-wise product between a single element of the first matrix and the entirety of the second matrix, and is used in principle-component analysis for image and signal processing [49]. Other operations like standard convolution and multi-headed attention are naturally represented as operations on tensors instead of operations on lists of matrices.

Beyond these basic operations, other details can be noted about deep neural networks. Most networks are vastly over-parameterized, meaning redundant information is encoded within the network weights. This makes tensor decomposition techniques useful in extracting salient information from the network while reducing the noise and unnecessary information. Tucker decomposition is one example, which reduces a three-dimensional tensor into a smaller "core" tensor and three other matrices, used in mobile networks to compress convolutional layers [27]. Canonical polyadic decomposition is another example, which splits a tensor into a sum of vectors, used to separate variables in many-dimensioned tensors [34]. Although, finding these decomposed representations is itself a time-consuming operation, so it can be approximated further through deep-learning. To summarize, usage of mathematical methods on the tensor representation of weights and input data to deep neural networks can be used to reduce noise and dimensionality in over-fitted networks while helping in principle component analysis and clustering.

2.1.3. Polyhedral Model

The polyhedral model is another type of optimization organized around affine transformations on loops. Generally, iteration domains are extracted from loops, and dependency graphs are formed linking iterations together. If the iteration domain for a given loop is affine—such that each iteration variable for each nested loop has an initial value, bound, and increment all expressible by affine expressions (like $y = Ax + b$)—then the polyhedral model can restructure the loop to improve overall performance in such a way that the dependencies between variables are still met.

The polyhedral method is unique in its representation of loops as mathematical expressions; each iteration of the loop can be imagined as a point on a grid, which means integer linear programming (ILP) methods can be used to find minimums that directly optimize a function for different target properties. In this approach, a "scattering" function maps run-time instances of statements in the loop to different execution dates, and the goal is to discover new scattering functions that a different loop can implement. Then, each statement in the loop along with their bounds, iteration variables, and dependencies are represented as a set of linear equations, which can be minimized, converted to a new scattering function, and translated back into a new loop.

This field has many methods which use the polyhedral model to find scattering functions that optimize for properties like parallelism, memory usage, performance, and data locality. Physically, the resulting loop can be imagined as implementing different common loop-optimization techniques like loop tiling, fusion, splitting, and others, but it is important to remember that the method has no internal representation of these techniques. An example is shown in figure 2.2: once an iteration domain has been shifted to remove the dependencies between different iterations of the outer loop, then each iteration of the outer loop can be parallelized [3].

2.2. Indirect Optimizations

As mentioned before, a downside of the above direct optimizations is that a compiler has no way of knowing if an optimization will lead to better performance or not. The benefit

of function inlining or loop unrolling is evident up until a certain point, after which the repetition in code leads to congestion in the instruction cache and an increase in execution time. Due to the sheer size of the parameter space, with each instruction having their own associated set of possible optimizations that can be performed, it becomes important to have a different method of discovering the best software organization of a deep learning model besides guessed heuristics.

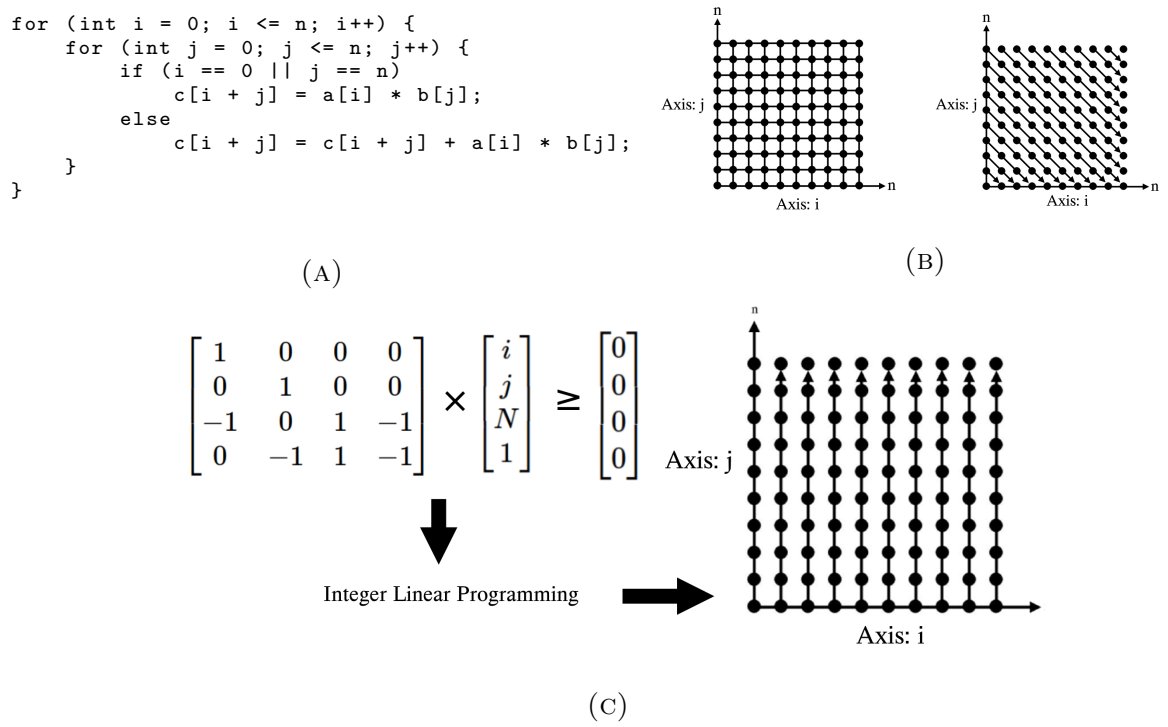


FIGURE 2.2. Visual overview of the polyhedral method. This example (a) takes loop code with bounds defined by affine expressions, (b) converts the loop into (left) an iteration domain and (right) a dependency graph as a function of the iteration variables, and (c) expresses these domains as math equations which can be solved to give a new parallelizable dependency graph.

The solution to this is to have the compiler attempt a certain set of optimizations and then measure the resulting program on the device. With enough attempts, the compiler can simply select the program which gave the best execution time. While this method is largely unacceptable for most programs due to the difficulty in setting up such a program,

most deep learning problems can be expressed in a way that leads to an iterative method like this easy. For example, a standard program might involve file operations; an iterative compiler would need to run a compiled program to obtain the final execution time without causing any side effects with file I/O or network operations that influence future iterations of the compiler. This is easy with deep learning because most neural networks (1) take tensors of a constant size; (2) pass those tensors through the model without causing any side effects through saved files, environment variables, or otherwise; and (3) yield tensors of constant size again. Since latency is the primary metric of importance for these compilers, the contents of the input tensor and output of the final tensor is unimportant, so random values can be passed in. Furthermore, in the cases that accuracy is actually important—which would occur with optimizations like quantization, which reduces the size of tensors to decrease execution time at the expense of accuracy [68]—the compiler can be adjusted to accept certain test-cases in the compilation process and only select those compiled programs which have an accuracy within a certain range. A schematic of this idea is shown in figure 2.3.

All iterative optimization problems can be expressed as either based on derivatives or based on a function value alone. Derivatives are a convenient method of expressing the rate of change for a cost function, and can directly be used to identify a path to changing the parameters that will lead to a lower cost. While deep learning *itself* is a derivative-based optimization method, as each operator in the network's operator graph is differentiable in most cases and leads to a parameter gradient being calculated through a method like gradient descent and backpropagation, deep learning *compilers* do not have this ability. This is because the impact of an optimization technique cannot be directly calculated without a device measurement: even straightforward optimization techniques like the unrolling of a loop by one iteration might lead to complex changes in data access patterns that cause the instruction cache to be too full or the pipeline to be underutilized.

The optimization problem can be visualized as any other standard optimization approach: a cost function is calculated after creating a set of parameters to characterize the

state of the system, then the optimizer repeats the process with more parameter sets and more measurements until some condition is met. For the deep learning compiler, the set of parameters is the set of program optimizations to perform and a measurement of the cost is the measurement of the execution time of the compiled program which was created from those optimizations. Geometrically, this is visualized in figure 2.4. The goal for the optimizer is both breadth of the optimization space (so that the global minimum has a fair chance of being discovered) as well as convergence time (since compilation and hardware measurement is a time-consuming procedure).

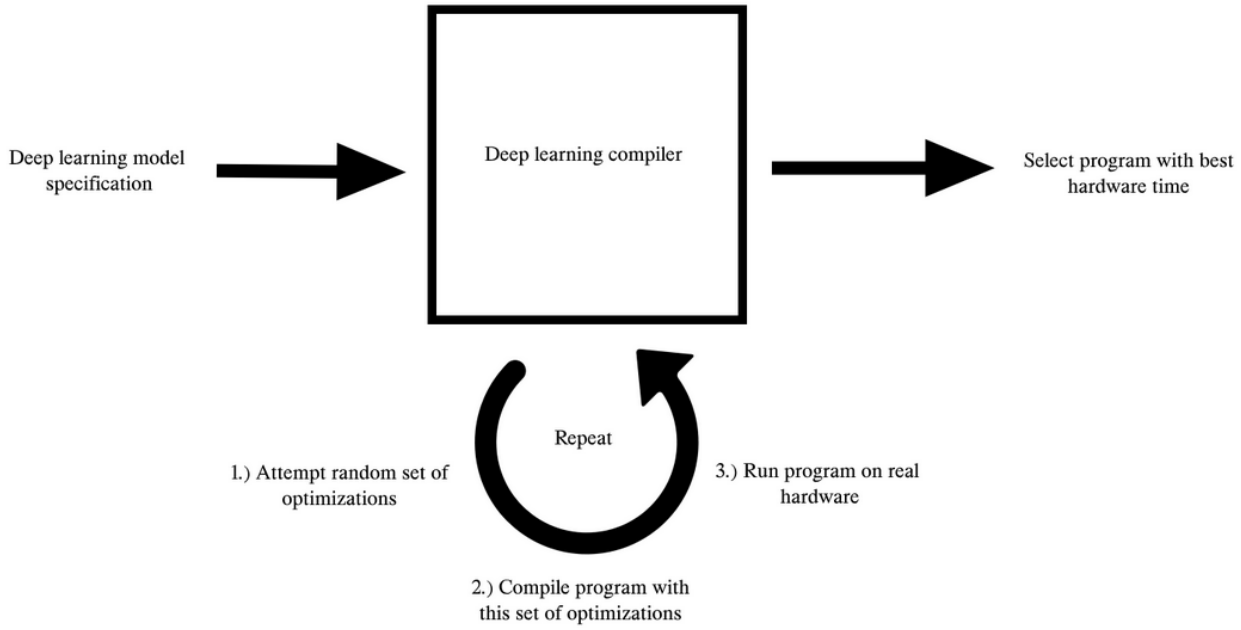


FIGURE 2.3. The deep learning compiler framework. Deep learning models are specified in some format, then programs are iteratively generated that implement the format and their execution time is measured on device. The fastest program is then selected after some number of iterations complete.

Due to the enormous volume of tunable optimization parameters in any given program, a good optimizer must be better than a grid search or random search. Two methods which are particularly noteworthy are *simulated annealing* and *genetic algorithms*, each of which use simple heuristics to identify candidate programs based on previous good-

performing programs.

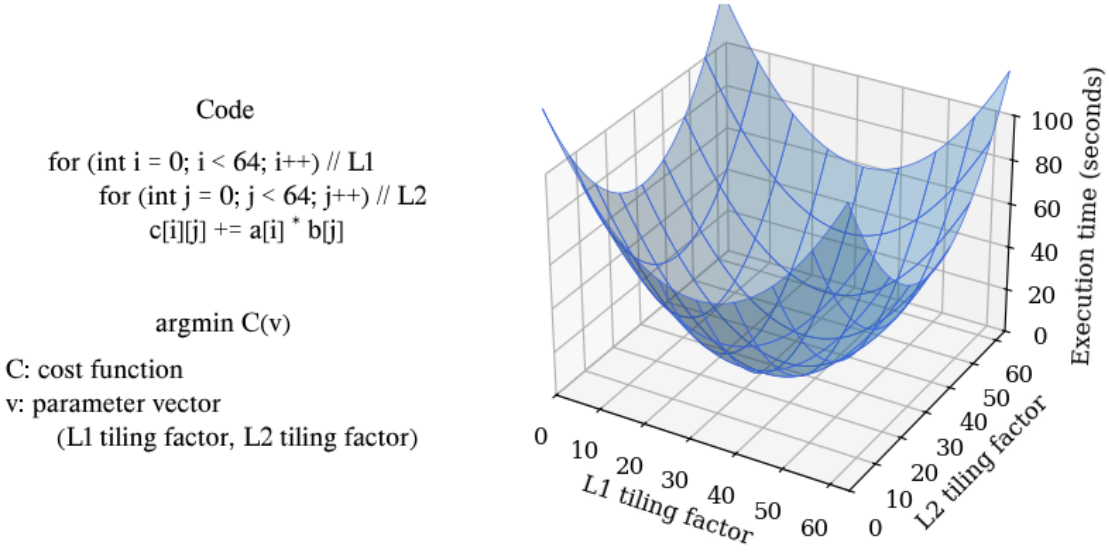


FIGURE 2.4. Optimization space for a program with two tunable parameters. An ideal loop tiling factor may be greater than one, but program performance decreases as tiling increases due to adverse effects on the cache.

Simulated annealing gets its name from metalworking, in which a hot metal is heated to a high temperature to allow it to be molded into a specific shape, before slowly cooling back to a stable state [4]. The start of the optimization process has the highest "temperature" and yields the largest variation in tested programs between each iteration, while over time the temperature decreases to create programs which are more similar to previous bests. This is similar to learning-rate modifiers in deep learning, which is typically high at first to allow the model to find a good starting-point, before reducing after some amount of time to give the model room to fine-tune to find a local minimum. Each iteration is based on the previous *best* iteration, so a program optimization which led to negative results will not overwrite another optimization which gave good results.

Figure 2.5 depicts an annealing algorithm. First, an initial measurement is made with random parameters. Then, a parameter state is chosen with a distance away from the previous best-performing program with a distance less than or equal to the temperature.

For the given one-parameter program, the loop tile size might range between 1 and 64, so the "distance" between two program optimizations would be the difference between their tile sizes. The program is then able to choose two program optimizations with a large difference in tile sizes at the start of the program, but as time increases, the decrease in temperature leads to a lower range in possible attempted values, causing the model to converge to a single program state.

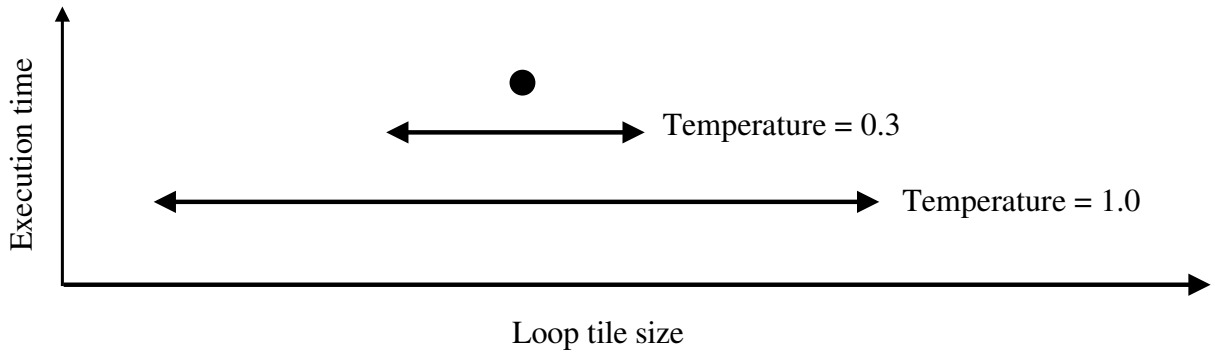


FIGURE 2.5. Simulated annealing on a single variable. Temperature decreases as the number of iterations increases, which leads to a convergence on an optimal program.

Genetic algorithms are a similar technique, taking inspiration from the genetic processes of living organisms [32]. Logically, an organism which is able to live long enough to reproduce must be considered "successful" at survival, so its genes are a good starting-point for offspring to inherit if their goals are also to survive and reproduce. Additionally, successful organisms either (1) produce a large volume of offspring, with the hopes that the small variations in each offspring will lead to a high performance in some of them, and/or (2) combine the traits of two successful organisms, yielding an organism which is more successful than either was individually. Both of these techniques are used in genetic optimization for programs: a program produces a list of "offspring" programs which are different in slight ways and programs can be combined by merging their optimizations, with the best-performing programs being selected to move on to the next iteration. Additionally, as with real biological systems, there always remains a slight chance at a drastic software optimization being

performed, which is improbable enough to not significantly delay convergence time but likely enough to potentially cause the model to explore towards an unknown optimization-space.

2.3. State of the Art and Challenges

This final subsection surveys the important papers in the field of deep learning compilers. To begin, Halide was the first project that demonstrated how an iterative approach to optimization can produce programs that are faster and more efficient than handwritten, heuristic-optimized programs [39, 52]. It was originally for the domain of image processing, which features heavy usage of stencil operations to process groups of operators on each cell of an image, shown in figure 2.6. Stencils feature a significant area for improvement for loop tiling: in the figure, a naive approach to calculating the stencil would complete the entire first operation (Add-3) before starting the second operation (Multiply-3), but this loads a significant amount of data redundantly due to the small size of the data caches. If the data caches were tiny, then the output of the result of the first operator (e.g., "3" in the middle layer) would get removed from the cache by the time the end of the first input was complete (e.g., "7" in the top layer). Then, the middle layer would get re-loaded, even though it was already loaded once earlier in the program. A more efficient solution would process multiple operators at once for certain parts of the stencil at a time, maximizing the usage of the cache and minimizing the amount of data needing to be re-loaded due to it being evicted from the cache before it was re-referenced.

An issue that Halide attempted to solve is the complexity of stencil programs. Certain image-processing workloads might contain dozens of operators, each connected in a nonlinear fashion, which makes a simple loop-tiling approach difficult. Halide's solution was to separate the high-level algorithm from the low-level implementation: the user could define operators in Halide's custom domain-specific language, and the Halide optimizer would identify an efficient implementation of the specification. During runtime, Halide would construct a search space of loops with different tiling factors and operator fusion amounts and would automatically search that space for the best implementation. This design proved effective in image-processing workloads in 2013, and it extended its usability in the domain of deep

learning as reflected in Apache TVM.

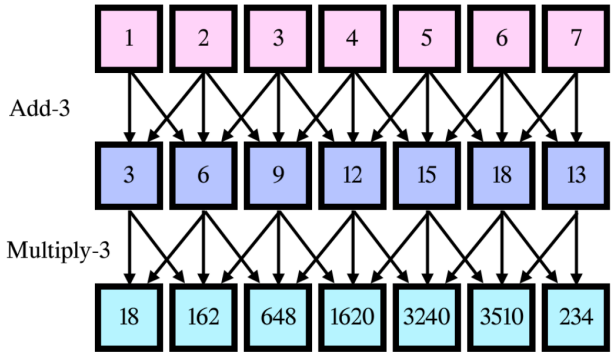


FIGURE 2.6. Stencil programs contain an input buffer (top row) which have some operation applied to groups of elements and placed into an output buffer (middle row). "Add-3" is an example operation which sets the output cell value equal to the sum of its top-left, top-middle, and top-right values. Multiple stencils can be put together to form computation graphs of operations similar to deep learning.

Similar to Halide, Apache TVM is split into its own high-level and low-level intermediate representations [6]. The high-level IR modifies the computation graph, allowing operator fusion and graph rewriting while looking only at general data-movement patterns, while the low-level IR performs local optimizations within a single operator. This separation between the IRs encourage flexibility in the kind of optimizations which can be performed. For example, the high-level IR is capable of defining operators as element-wise and easily-fusible, while the low-level IR can allow the user define endpoints for vectorization/tensorization and other loop-level optimizations. Both IRs use a search space, cost model, and simulated-annealing-based AutoTVM optimizer [7] to find fast program implementations of their deep-learning models.

Apache TVM spawned further research into deep learning compilers by encouraging the creation of new optimizers that fuse more operators, move more of the computation offline, and explore a wider parameter search space in less time. Ansor [72] is an example of one of these extensions, which uses rule-based kernel expansions to move more of the implemen-

tation details away from the programmer. With Ansor, only the mathematical definition of operators is supplied by the programmer: the compiler decomposes this operator to a high-level *sketch*, which contains basic for-loops without tiling dimensions or parallelization, and the optimizer uses a genetic algorithm to iteratively expand these loops with various inlining, tiling, fusion, and parallelization techniques.

Several other papers choose to look at more high-level details like the deep-learning model's computation graph. For example, some operators defined by the programmer might compute data consisting only of constant values which can be moved offline and computed ahead of time. Therefore, a compiler must be capable of classifying operators into different data-movement categories—like one-to-one, one-to-many, etc.—and then recognize mathematical relations like associativity, distributivity, and commutativity to enforce the reordering of operators within a graph. Several traditional compiler techniques can be applied in this case like constant folding and copy propagation. TASO [22] is one project which uses a theorem-prover to substitute subgraphs within the computation graph with new operators, and DNNFusion [45] is another project that explores fusion possibilities after classifying each operator's mathematical properties. Finally, the Apollo [71] project uses the polyhedral method along with a rule-based operator-classifier to form their iterative search space.

Lastly, while most projects look at implementation-specific details, hoping to form source code on the level of loops or intermediate representations like computation graphs, BOLT [69] is a unique project that tunes the parameters of template libraries like the CUDA linear algebra library CUTLASS. BOLT assumes that the most optimal implementation of certain operators is still defined by human programmers (in this case, CUTLASS was hand-written by programmers at NVIDIA), so instead of attempting to compile new low-level code, it tunes high-level template parameters like thread/block-counts, data alignment, synchronization, and more. CUTLASS functions contain operators like general matrix multiplication (GEMM) with the addition of prologues/epilogues to represent activation functions, so BOLT iteratively tries different orderings of these functions, indirectly changing the low-level code

through the high-level templates.

Each of these compilation techniques, from the optimizations directly on the source code to the indirect iterative optimizers that execute the target program many times during the compilation stage, can be used to create efficient software solutions for deep-learning models. Although, a ceiling exists in the amount of benefit software solutions are capable of achieving in training and inference. The next section discusses a modification to the GPU hardware itself which can give additional performance improvements and can be paired with software optimizations at the same time.

CHAPTER 3

L2 CACHE CHARACTERIZATION FOR CONVOLUTIONAL NEURAL NETWORKS

Convolutional neural networks are a popular method for recognizing the contents of images. Various domains of computer vision exists, including image classification, which assigns a label to the entirety of an image; object localization, which draws a bounding box containing each object in an image and classifies their contents; semantic segmentation, which associates each pixel in the image with a different classification label; and image captioning, which describes an image with plaintext instead of directly associating it with a predefined label. Each of these tasks utilizes two-dimensional convolution operations in some capacity due to their ability of spatially representing visual data in a way that fully-connected or attention operations are not able to do on their own.

This section describes a method of accelerating these two-dimensional convolutions with a popular technique called *lowering*, which reduces the sliding-kernel algorithm with a faster matrix multiplication. It begins with a description of three standard image-recognition neural networks and their convolution operations, the inherent duplication present within lowered matrices, and a new cache design which can reduce the duplication by mapping identical values to the same addresses.

3.1. GEMM-based Convolution for GPU Accelerators

The high-level schematics of three popular image-recognition networks—AlexNet [28], VGG16 [56], and ResNet18 [15]—is shown in figure 3.1. AlexNet revolutionized the deep learning field and used GPUs for the first time to accelerate deep learning operations at a large scale; VGG16 followed and used more convolutional layers with a much higher parameter count; and ResNet18 solved an issue with operators in deeper parts of the network losing information from shallower parts of the network, and used residual connections to greatly increase the depth of image networks. Each network has a similar structure with (1) a group of convolutional layers at the front, recognizing visual characteristics in the input images like lines, curves, and other shapes, while (2) fully-connected layers are at the back,

translating the high-level features into the target properties. Multiple convolutional layers are usually grouped together with activation functions separating them, and multiple groups are separated by a pooling function to reduce the dimensionality of the input data. This reduction in dimensionality allows a single cell in a deeper convolution layer to see a very wide patch of cells in a more shallow input layer; the change in receptive field size as the model gets deeper means individual cells can comment on visual details more abstract than just lines or other shapes by joining the details from a wide area. This has been shown to be more effective than simply using very wide kernels from the start.

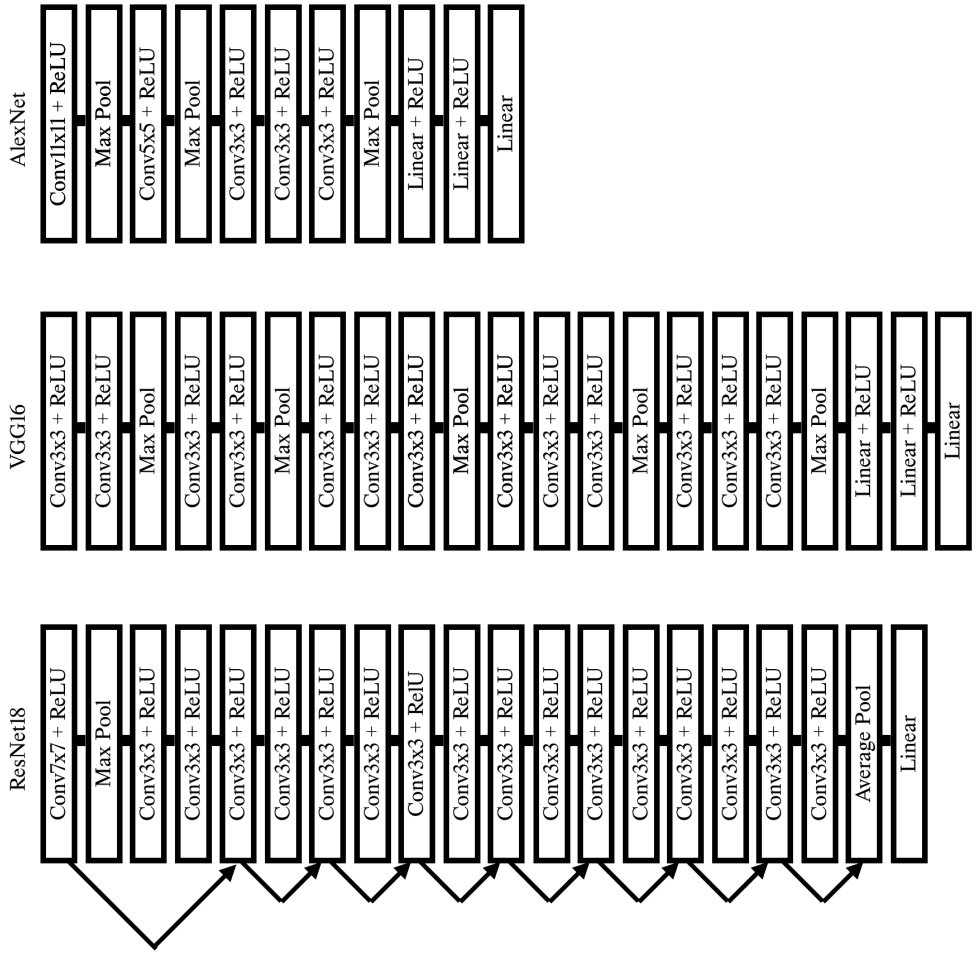
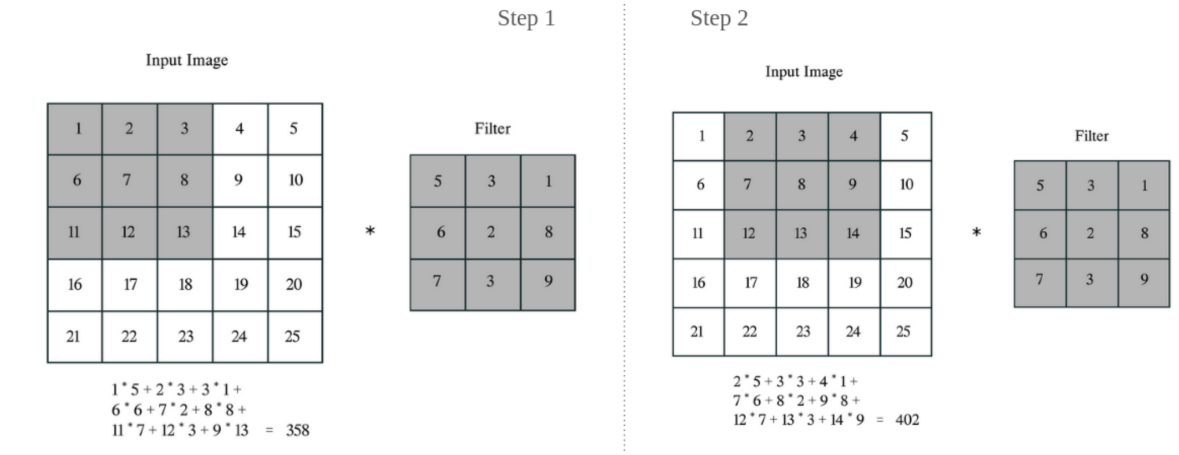


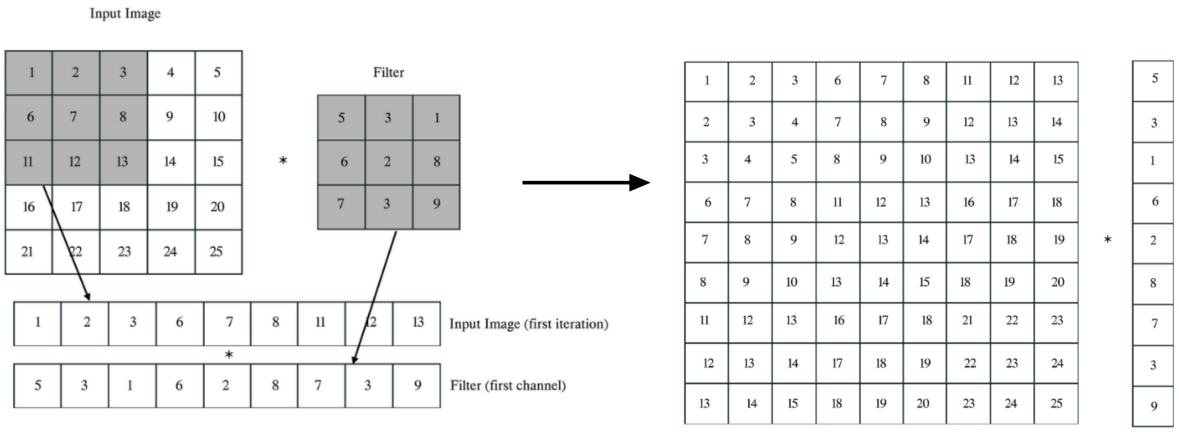
FIGURE 3.1. AlexNet, VGG16, and ResNet18 architectures.

The actual convolution operation is shown in part (a) of figure 3.2. A "filter" is a weight matrix filled with constant values, and it slides across the input image calculating dot

products between itself and a section of the image during each step. If the input has multiple channels—as is the case with color images having a red, green, and blue component—then multiple filters are aligned as a group, and a group of dot-products are summed together to get the final output value. If the output has multiple channels, then multiple *groups* of filters are used for each value of the input.



(A)



(B)

FIGURE 3.2. Two common approaches to implement a 2D convolution, either (a) direct or (b) GEMM-based (in fact, there are additional convolution methods that are not discussed in this work, such as FFT [1] and Winograd [33, 70].

Figure 3.3 gives code that demonstrates this convolution operation. Since the code contains many loops, the register file would have poor utilization. If tiling, unrolling, and vectorization were used to improve this, the code still runs into an issue with identical data being loaded repeatedly at different points in time. The weights would need to stay cached in fast-access memory since it is accessed the most frequently, while the multi-dimensional aspect of the input image would make the cache less useful than if the data were strictly one-dimensional. Furthermore, GPUs have a memory coalescing unit that encourages aligned loads to contiguous locations. Coalesced memory accesses are combined together into a single load request, which makes vector operations significantly more coalesce-able and thus places less strain on the memory system when compared to direct convolution. To summarize, the multi-loop implementation of this operation is not the most hardware-friendly method.

```
for (int n = 0; n < N; n++)
  for (int c_o = 0; c_o < C_o; c_o++)
    for (int c_i = 0; c_i < C_i; c_i++)
      for (int h_o = 0; h_o < H_o; h_o++)
        for (int w_o = 0; w_o < W_o; w_o++)
          for (int h_f = 0; h_f < H_f; h_f++)
            for (int w_f = 0; w_f < W_f; w_f++)
              O[n][c_o][h_o][w_o] +=
                I[n][c_i][h_o + h_f][w_o + w_f] *
                W[c_o][c_i][h_f][w_f]
```

FIGURE 3.3. Code that demonstrates the convolution operation. From top to bottom, the loop bounds are the number of batches (N), the number of output channels (C_o), the number of input channels (C_i), the height of the input (H_o), the width of the input (W_o), the height of the filter (H_f), and the width of the filter (W_f).

.

This motivates the usage of lowered convolution. *Lowering* describes any method which translates data from one representation to another equivalent representation in a

more basic form. The specific lowering method used for convolution is named im2col. It exploits the fact that the convolution code described above can be expressed purely in terms of dot products: the sliding kernel can be flattened to a vector, and the receptive field of the input can be flattened to another vector. If the input consists of multiple channels, then each channel and each filter within a group can be concatenated to form two long vectors which can each have a dot-product calculated again. The benefit of formulating the problem this way is that groups of dot products can be imagined as a matrix-multiplication, which brings about many benefits as opposed to the loop-based method. Part (b) of figure 3.2 gives a visual representation of this matrix multiplication.

As opposed to the loop-based method, matrix multiplications are very cache-friendly depending on the implementation. The data representation is not as dense as the loop-based method, but the matrix-multiplication problem has existed for so long in so many domains that modern hardware has fast implementations built-in already specifically to solve it. An example of this is shown in the tensor cores on GPUs: these are implemented as a systolic array of processing elements that compute a single 4x4 matrix-multiply and accumulate (MMA) instruction per cycle [41]. The NVIDIA Tesla V100 GPUs have been shown to reach a peak performance of 125 tera-flops per second by further exploiting a mixed-precision representation of the input data, representing the input in 16-bit floating-point and the output in 32-bit among other data formats. Furthermore, high-performance matrix-multiplication libraries like NVIDIA's CUTLASS and cuBLAS offer pre-optimized code that can approach this throughput easily, making matrix multiplication a more attractive choice than the loop-based convolution.

The convolution operation takes a significant amount of time on image networks, and in some cases is the performance bottleneck. For example, one work showed that the VGG16 network spends 93.1% of the total execution time on convolutional layers [23]. Lowering the operation significantly reduces this performance bottleneck at the expense of memory usage, shown in another work demonstrating that the performance of ResNet18 is improved by 14.5x at the expense of 8.4x more memory transactions [26].

3.2. Inefficient Resource Utilization in GEMM Computation Due to Data Duplication

Lowering the convolution with GEMM has clearly been shown to be an efficient operation since it reduces the execution time so dramatically, but the problem it presents with its memory accesses is a large area for improvement. A system which might reduce the memory accesses to the kernel while keeping the matrix-multiplication representation of the data would be ideal, and this is possible with the help of modifications to the hardware.

Lowering a convolution operation to matrices leads to predictable patterns of duplication in the output. This is demonstrated in the right of part (b) in figure 3.2; compared to the initial matrix, which contained no duplicate values, the final matrix has many repetitions of the same values at different positions. This is a result of the receptive field of the filters having overlapping elements in each slide. This is most apparent in cases like VGG16's first layer which has a filter size of 3 and a stride of 1, shown in figure 3.4: within a row, a filter will slide across the same value three times; within a column, a filter will slide across the same value three times; so in total, in most cases the same value will appear in nine different dot products.

This duplication is better visualized in figure 3.5. In it, each of the four images displayed contain 1024x1024 cells, with each cell representing a contiguous 2-byte section of memory starting at the base address of the lowered data matrix (specifically, the lowered *input data* matrix, which is the only one which experiences duplication; the weight matrix does not have any duplication in its lowered representation the because each filter naturally represents a single dot product to perform). The 2-byte size was chosen because the tensor core operation is mixed-precision, and takes two operand matrices containing half-precision (2-byte) data and outputs a resulting matrix with single-precision (4-byte) data. The figure visually shows duplication in terms of the brightness of each cell: a black cell contains completely unique data that appears only once, while a red cell contains data that appears in at least one other location in the matrix, and a yellow cell contains data that appears eight or more times in other locations. The four layers are the first layer of AlexNet, the fifth (last) layer of AlexNet, the first layer of VGG16, and the first layer of ResNet18.

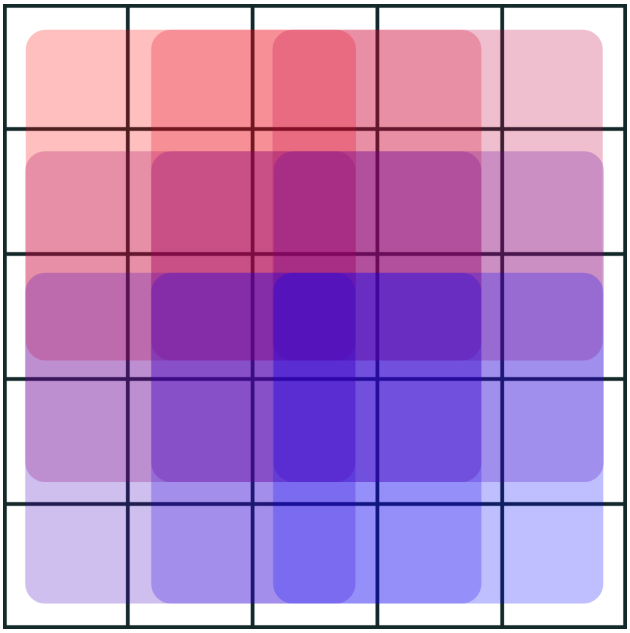


FIGURE 3.4. Figure showing the repetition present in VGG's first layer. Each colored square represents one dot-product between the filter and image. The center cell is present in nine different dot products.

This duplication has its cause in three sources: in tensor core padding, in rows, and in columns. First, tensor cores require that all operand matrices have a size that is a multiple of 16. Matrices that do not have this multiple are padded with zeros, leading to a high volume of duplicated values in some cases. Second, while a filter slides across a row, it encounters the same value multiple times. Third, when a filter slides to the next column, it then repeats the same number of row-accesses it encountered the first time. The number of times a cell can be repeated is then at most equal to

$$\lceil \frac{\text{filter}_{\text{width}}}{\text{stride}_{\text{horizontal}}} \rceil \times \lceil \frac{\text{filter}_{\text{height}}}{\text{stride}_{\text{vertical}}} \rceil$$

where the actual amount of repetition a data value experiences depends on its alignment with the filter. The complete method of finding repetitions programmatically is shown next.

An example of a duplication-aware architecture is the Duplo GPU architecture, which identifies the inherent duplication present in lowered matrices [26]. Duplo works by mitigating loads to data that was already loaded previously by utilizing a register-renaming unit

attached to the tensor cores. Specifically, some tensor core instructions load data from an address operand, and Duplo passes those addresses through a duplication-detection unit with a load-history buffer and dynamically renames registers with values that exist elsewhere in the register file. The duplication-detection unit is accessed in parallel with the L1 cache, and if the unit does identify pre-existing values elsewhere in the register file, then the L1 access is canceled.

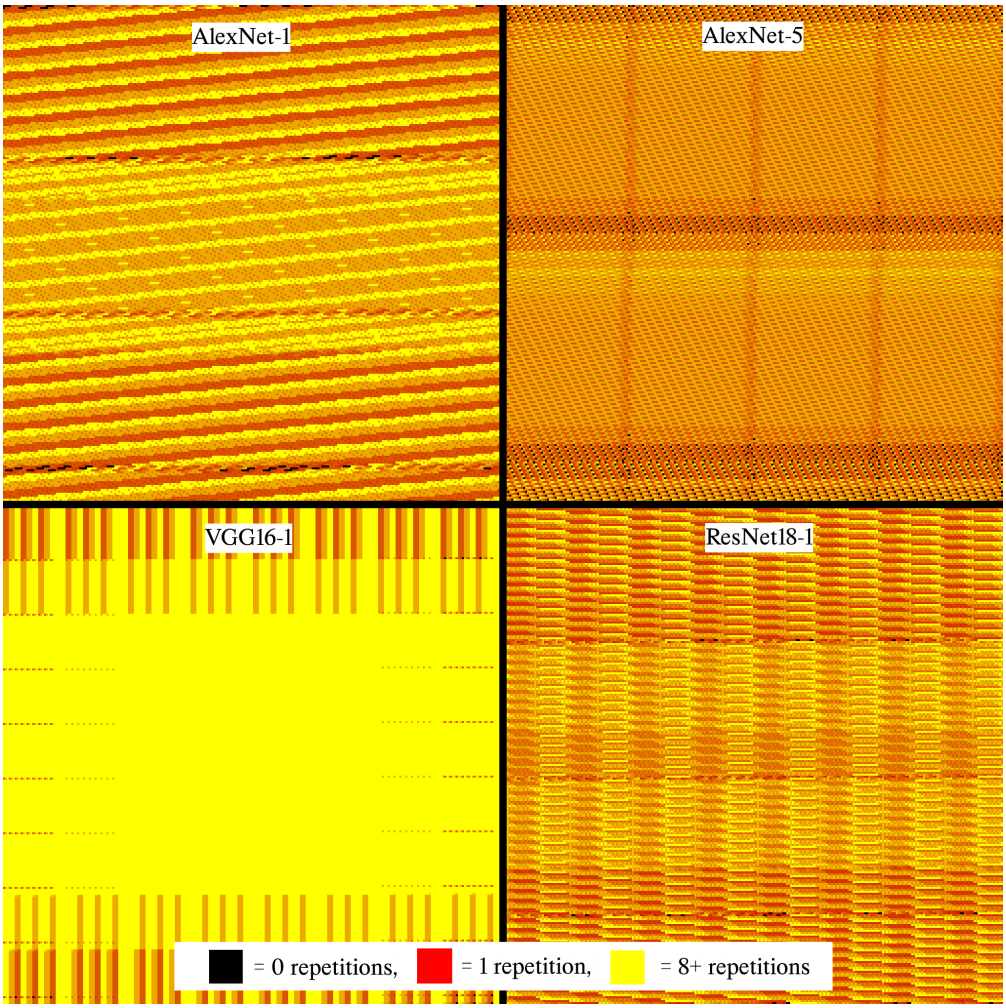


FIGURE 3.5. Repetition within the B matrix of the GEMM operation ($D = A \times B + C$). Each cell in the image is a two-byte memory address. Cells that are black contain data that only appears once in the memory system and are entirely unique, while cells that are yellow contain eight or more duplicates in other places.

Duplo's duplication-detection unit uses an ID generator to assign a unique identifier to each unique data element. This ID can be interpreted as a position indexing an element within the original convolutional input tensor, since this original tensor—which originally contained only unique data—was lowered and duplicated. These identifiers note that the positions of the duplication are regularly spaced in the memory system and can be predicted as long as the original convolution parameters are known, so a simple set of equations are used to generate them. An adjusted version of Duplo's equations is as follows. First, the size of a single dot product is found by the concatenation of each filter together, where the number of channels determines the total number of filters.

$$\text{dprod}_{\text{size}} = \text{filter}_{\text{rows}} \times \text{filter}_{\text{cols}} \times \text{input}_{\text{channels}}$$

Next, the number of dot products in a row of the image, column of the image, and entire output is found by sliding the filter across the rows and columns once per output filter.

$$\text{dprod}_{\text{row}} = 1 + \frac{\text{input}_{\text{rows}} - \text{filter}_{\text{rows}}}{\text{stride}_{\text{rows}}}$$

$$\text{dprod}_{\text{col}} = 1 + \frac{\text{input}_{\text{cols}} - \text{filter}_{\text{cols}}}{\text{stride}_{\text{cols}}}$$

$$\text{dprod}_{\text{total}} = \text{dprod}_{\text{row}} \times \text{dprod}_{\text{col}} \times \text{output}_{\text{channels}}$$

A physical address is provided to the ID function. To make this independent of the data type size, the address is converted to an index with respect to the data type size and the starting address of the lowered matrix. This identifies both a dot product within the lowered matrix and a column within the dot product. Duplo uses the terminology "workspace" to refer to this position within the lowered matrix and dot-product, which we will continue.

$$\text{workspace}_{\text{index}} = \frac{\text{address} - \text{input}_{\text{start\_address}}}{\text{size}_{\text{data\_type}}}$$

$$\text{workspace}_{\text{row}} = \text{workspace}_{\text{index}} / \text{dprod}_{\text{size}}$$

$$\text{workspace}_{\text{col}} = \text{workspace}_{\text{index}} \bmod \text{dprod}_{\text{size}}$$

Another way of visualizing $\text{workspace}_{\text{row}}$ and $\text{workspace}_{\text{col}}$ is that, if all of the dot products for the matrix are organized in a list, then $\text{workspace}_{\text{row}}$ is the index of the dot product being performed and $\text{workspace}_{\text{col}}$ is the index of the element within the dot product. Next, each output channel is seen as a collection of filters, and the entirety of one filter's output is calculated before moving on to the next filter. In the lowered matrix, this means one output channel's dot products appears before the next output channel's. The index of the output channel is then found with a division. We can draw a distinction between the global workspace position and the local position within a specific filter (output channel) to simplify the expression.

$$\text{channel}_{\text{output}} = \text{workspace}_{\text{row}} / (\text{dprod}_{\text{row}} \times \text{dprod}_{\text{col}})$$

$$\text{local}_{\text{row}} = \text{workspace}_{\text{row}} \bmod (\text{dprod}_{\text{row}} \times \text{dprod}_{\text{col}})$$

$$\text{local}_{\text{col}} = \text{workspace}_{\text{col}}$$

Then, we want to convert this workspace position into a position within the original image: if the filter only slides across the image in discrete locations, then the filter that contains the input address is identified with a row and column.

$$\text{filter}_{\text{row}} = (\text{workspace}_{\text{row}} / \text{dprod}_{\text{col}}) \times \text{stride}_{\text{rows}}$$

$$\text{filter}_{\text{col}} = (\text{workspace}_{\text{row}} \bmod \text{dprod}_{\text{col}}) \times \text{stride}_{\text{cols}}$$

Similarly, the relative position of the element *within* the filter (its row and column) along with the index of the input channel can be found with the index within the dot product.

$$\text{relative}_{\text{row}} = (\text{workspace}_{\text{col}} / \text{filter}_{\text{cols}}) \bmod \text{filter}_{\text{rows}}$$

$$\text{relative}_{\text{col}} = \text{workspace}_{\text{col}} \bmod \text{filter}_{\text{cols}}$$

$$\text{channel}_{\text{input}} = \text{workspace}_{\text{col}} / (\text{filter}_{\text{rows}} \times \text{filter}_{\text{cols}})$$

Finally, the absolute positions of the row and column can be obtained by combining the position of the filter with the relative position of the element within the filter.

$$\text{row} = \text{filter}_{\text{row}} + \text{relative}_{\text{row}}$$

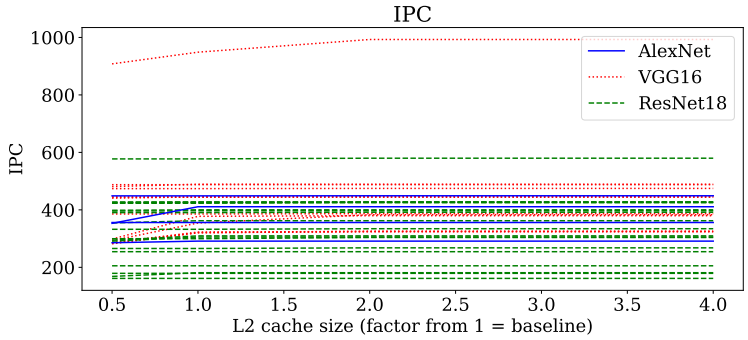
$$\text{col} = \text{filter}_{\text{col}} + \text{relative}_{\text{col}}$$

Thus, we have obtained an index into the original convolutional input tensor. If the weight tensor was a four-dimensional array, then we could obtain the same unique element with `weights[channel_output][channel_input][row][col]`. For the benefit of the hardware, this can optionally be packaged into a single value instead of three separate values.

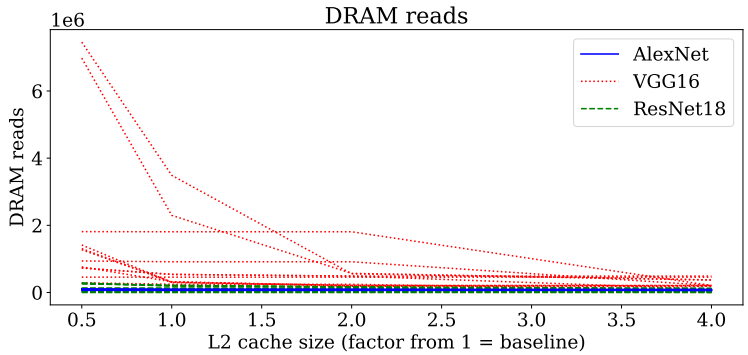
$$\begin{aligned}\text{id} = &\ \text{channel}_{\text{output}} \times (\text{input}_{\text{channels}} \times \text{input}_{\text{rows}} \times \text{input}_{\text{cols}}) \\ &+ \text{channel}_{\text{input}} \times (\text{input}_{\text{rows}} \times \text{input}_{\text{cols}}) \\ &+ \text{row} \times \text{input}_{\text{cols}} \\ &+ \text{col}\end{aligned}$$

Duplo received positive results, but a critical downside of its design is its requirement that duplication is prevented only at the *register* level. In other words, duplication still exists within the L2 cache. This means a significant portion of the cache is used representing redundant data, which lowers its effective capacity; a more intelligent cache design would be capable of indexing and storing only the unique, non-duplicated data within the cache, thereby allowing the storage of more data and its effective increase in size.

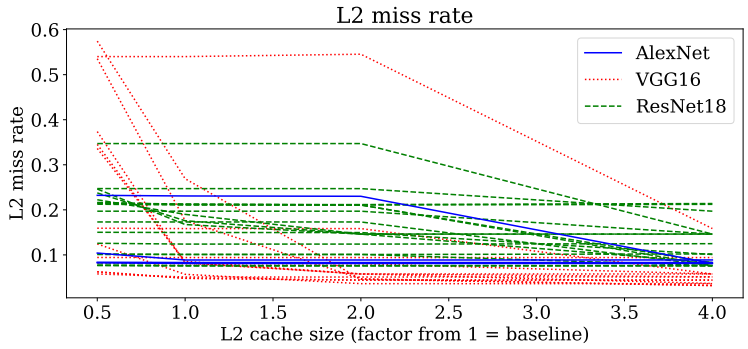
Figure 3.6 depicts the effect of the L2 cache size on the system IPC, DRAM, and L2 itself. IPC is typically a good measure of parallelism since a system which is able to complete more instructions per cycle is better utilizing the hardware and is not running into significant stalls or bottlenecks. Across each cache size, IPC is largely unaffected except for VGG16, the largest model out of the three. These figures were collected with a batch size of one image, which implies more images would put a larger strain on the hardware and exemplify VGG16's effects more due to requiring a larger matrix-multiplication size. Besides this, DRAM reads and L2 miss rate are corollaries since a lower L2 miss rate implies the L2



(A)



(B)



(C)

FIGURE 3.6. After changing the size of the L2 cache from [0.5x, 1.0x, 2.0x, 4.0x] its baseline size, the (a) instructions per cycle, (b) DRAM reads, and (c) L2 miss rate of each deep neural network.

is able to service more of the read/write requests, which means the DRAM would need to involve itself less frequently.

An important detail about this figure is that general performance in terms of IPC

is mostly unaffected with respect to increasing the L2 cache size. While this might imply that the L2 is not a bottleneck, it does not imply the inverse: that a *reduction* in the L2 size would not become a *new* bottleneck, as can be seen in each figure with a half-sized L2 having worse results in many cases. The latency-hiding ability of GPUs with schedulers and out-of-order execution does not always make it so a larger cache means better performance, especially since GEMM workloads are largely tuned to be effective for many baseline cache sizes, but a larger effective L2 cache would allow us to reduce the L2 size without suffering performance decreases. This becomes our new goal: instead of increasing the effective cache size, we can instead make the L2 cache more effective, which lets us reduce the physical size of the L2 on the GPU to save power and area for other critical hardware modules, which we can use to add more SMs to increase IPC directly.

3.3. Improving Cache Utilization for GEMMs

An important part of Duplo's work was finding the element ID equation shown in the previous section. In it, the authors developed a function id $= f(\text{address})$ which was capable of converting an address pointing within the lowered matrix into a identifier, in which multiple distinct addresses map to the same identifier. Duplo's method then performed register-renaming such that load requests were avoided if another existing register contained data with the same identifier as the requested data. This works well for the small tensor core warp register file, but it suffers significant drawbacks with the L2 cache: the Tesla V100 contains an 6144 KB L2 cache [46], so caching the element ID of each possible element within this L2 cache would require an additional cache containing $6144\text{KB}/2\text{bytes} \times 4\text{bytes} = 12288\text{KB}$, in which each data element within the lowered matrix is a two-byte float and each element ID is a four-byte integer. The size of the auxiliary cache would be greater than the size of the actual data cache.

Existing de-duplication cache approaches [60, 42, 59] typically work by taking a cache line and representing all or parts of it with a compressed format, then de-compressing it when the line is requested. Multiple compressed lines can then be represented within a single actual cache line, improving the effective capacity. For example, a typical method is to use zero-run

de-duplication [2] that replaces runs of zeros with a number representing how many zeros were replaced. The base-delta-immediate cache [50] is another method that recognizes the low dynamic range of data values within a cache line and converts lines into a base value and a list of offsets (deltas) from that base value.

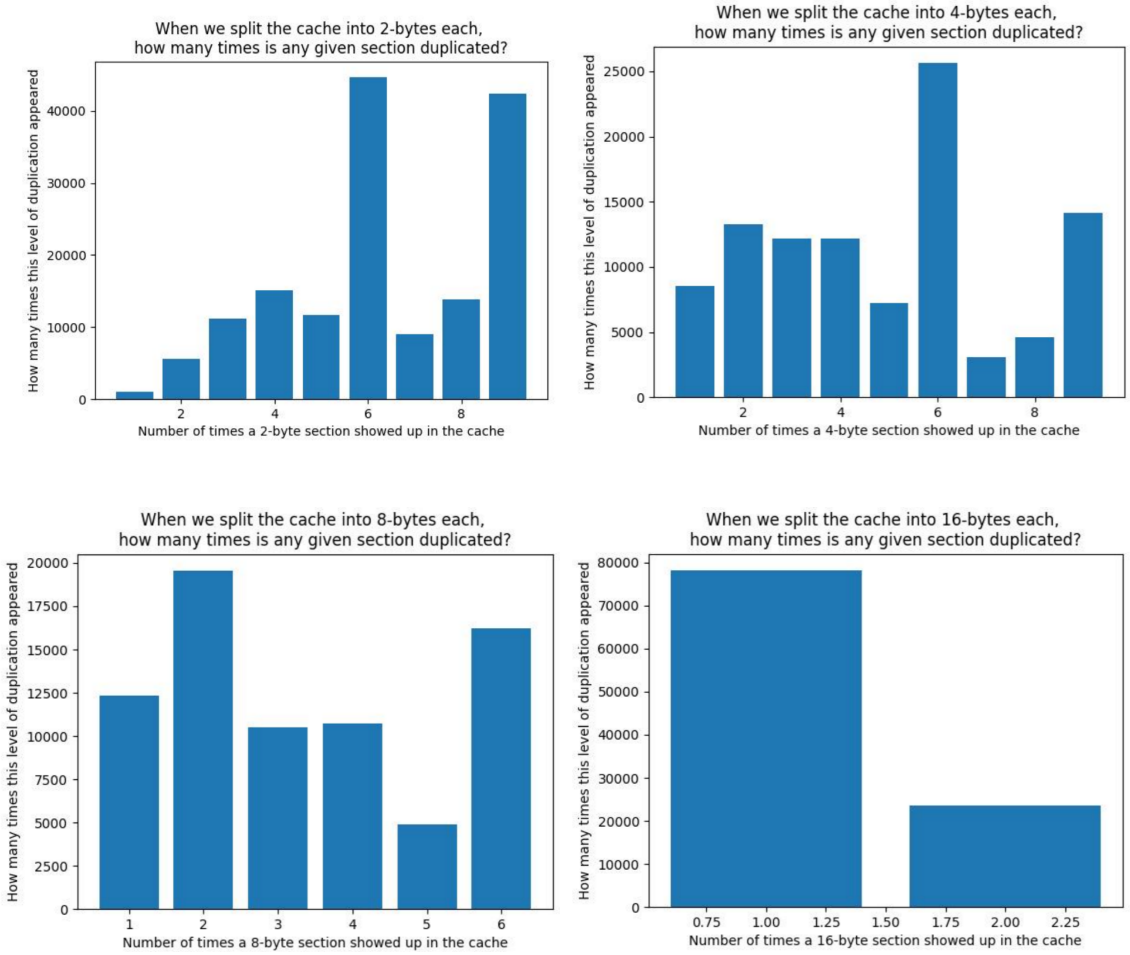


FIGURE 3.7. Non-locality in the lowered matrix. The graphs show that when the matrix is split into [2, 4, 8, 16]-byte aligned groups, the number of times a single group is completely duplicated in another part of the cache decreases with group size.

Although, neither of these general approaches are ideal for the pattern of duplication seen in lowered matrices. To begin with, the duplication seen within the L2 cache is not necessarily local within a single cache line, shown in figure 3.7. A typical method of de-

duplication is to show how to different lines of the cache contain the same contiguous group of bytes, to which the data can be stored in a special auxiliary cache and the line data can be replaced with an address to this auxiliary line. Although, this figure shows there is not enough long-lasting contiguous data within the cache to justify this method.

Additionally, an existing value-based de-duplication method does not fully utilize the inherent order present within the data: while other cache methods must recognize duplication at runtime and mitigate them without knowing the problem domain beforehand, this lowered-convolution field has the special benefit of being describable with a math equation. Furthermore, other de-duplication approaches must work with writes as well as reads, in which a write might disrupt the compressed cache state and require on-the-fly decompression, adjustment, and re-compression. Since the buffers used in our deep-learning inference software can be non-overlapping, we can then designate the lowered matrix as read-only during GEMM inference, mitigating the issue with writes possibly modifying compressed lines. To combine both of these benefits, we can use a custom mapping algorithm to re-map duplicated lines and prevent loads to DRAM in the case of pre-existing data.

A requirement to the proposed technique is an inverse to the equation proposed by Duplo: instead of converting a physical address to an element ID, we must convert an element ID into a list of physical addresses which map to the same ID. Then, we can check each address within the list and see if they hit or miss within the L2. The inverted equation uses the same $\text{dprod}_{\text{size}}$, $\text{dprod}_{\text{row}}$, $\text{dprod}_{\text{col}}$, and $\text{dprod}_{\text{total}}$ as Duplo's equation. First, the cumulative number of elements within a dimension (dimsize) is provided to simplify the expression.

$$\text{dimsize}_{\text{col}} = \text{input}_{\text{cols}}$$

$$\text{dimsize}_{\text{row}} = \text{dimsize}_{\text{col}} \times \text{input}_{\text{rows}}$$

$$\text{dimsize}_{\text{channel\_input}} = \text{dimsize}_{\text{row}} \times \text{input}_{\text{channels}}$$

$$\text{dimsize}_{\text{channel\_output}} = \text{dimsize}_{\text{channel\_input}} \times \text{output}_{\text{channels}}$$

Thus, $\text{dimsize}_{\text{channel\_output}}$ contains the number of weights within the entire convolution op-

eration. Then, the 4D index into the weight tensor can be re-obtained from the single ID.

$$\text{channel}_{\text{output}} = \text{id}/\text{dimsize}_{\text{channel\_input}}$$

$$\text{channel}_{\text{input}} = (\text{id mod dimsize}_{\text{channel\_input}})/\text{dimsize}_{\text{row}}$$

$$\text{row} = (\text{id mod dimsize}_{\text{row}})/\text{dimsize}_{\text{col}}$$

$$\text{col} = \text{id mod dimsize}_{\text{col}}$$

There are many different dot products which might contain the element at the given address, which is the source of the duplication. We want to find the position of the first dot product within the lowered matrix that contains this index. Each counter below represents how many dot products would need to occur across different columns, rows, and in total until the first occurrence of the element ID is seen.

$$\text{counter}_{\text{col}} = \begin{cases} 1 + \frac{\text{col} - \text{filter}_{\text{cols}}}{\text{stride}_{\text{cols}}} & \text{if col} > \text{filter}_{\text{cols}} \\ 0 & \text{otherwise} \end{cases}$$

$$\text{counter}_{\text{row}} = \begin{cases} 1 + \frac{\text{row} - \text{filter}_{\text{rows}}}{\text{stride}_{\text{rows}}} & \text{if row} > \text{filter}_{\text{rows}} \\ 0 & \text{otherwise} \end{cases}$$

$$\text{counter}_{\text{dprod}} = \text{counter}_{\text{row}} \times \text{dprod}_{\text{row}} + \text{counter}_{\text{col}}$$

This can be used with the relative position of the element within the filter, similar to Duplo's case.

$$\text{relative}_{\text{col}} = \text{col} - \text{counter}_{\text{col}} \times \text{stride}_{\text{cols}}$$

$$\text{relative}_{\text{row}} = \text{row} - \text{counter}_{\text{row}} \times \text{stride}_{\text{rows}}$$

Both the above can then construct the workspace row and column of the first appearance of the element ID.

$$\text{start}_{\text{row}} = \text{counter}_{\text{dprod}} + \text{channel}_{\text{output}} \times (\text{dprod}_{\text{row}} \times \text{dprod}_{\text{col}})$$

$$\text{start}_{\text{col}} = \text{relative}_{\text{row}} \times \text{filter}_{\text{cols}} + \text{relative}_{\text{cols}}$$

Next, the number of duplicated elements—or the number of addresses that map to the same element ID given in the main function—need to be found. An element will never be duplicated within a single dot product, so this can be rephrased as the number of dot products (or the number of sliding filters) across the entire convolution operation that contain this element. Since we know the location of the first dot product that contains the element, the number of times a filter will repeatedly contain it while sliding across the column and across the row can be found as a function of the position, stride, and alignment within the lowered matrix.[1]

$$\begin{aligned}
\text{fix}_{\text{col}} &= (\text{input}_{\text{cols}} - \text{filter}_{\text{cols}}) \bmod \text{stride}_{\text{cols}} \\
\text{fix}_{\text{row}} &= (\text{input}_{\text{rows}} - \text{filter}_{\text{rows}}) \bmod \text{stride}_{\text{rows}} \\
\text{repeat}_{\text{col}} &= 1 + \frac{\text{relative}_{\text{cols}}}{\text{stride}_{\text{cols}}} - \max\{0, \text{col} - \frac{\text{input}_{\text{cols}} - \text{filter}_{\text{cols}} - \text{fix}_{\text{col}}}{\text{stride}_{\text{cols}}}\} \\
\text{repeat}_{\text{row}} &= 1 + \frac{\text{relative}_{\text{rows}}}{\text{stride}_{\text{rows}}} - \max\{0, \text{row} - \frac{\text{input}_{\text{rows}} - \text{filter}_{\text{rows}} - \text{fix}_{\text{row}}}{\text{stride}_{\text{rows}}}\} \\
\text{count} &= \text{repeat}_{\text{row}} \times \text{repeat}_{\text{col}}
\end{aligned}$$

Here, "count" represents, for a given element ID, the total number of values within the lowered matrix that contain the same element ID. These can be used to re-create the workspace index from Duplo.

$$\begin{aligned}
\text{element}_{\text{row}_i} &= \text{start}_{\text{row}} + r_i \times (\text{dprod}_{\text{row}} - \text{repeat}_{\text{col}}) + c_j \\
\text{element}_{\text{col}_j} &= \text{start}_{\text{col}} - (1 - r_i) \times \text{stride}_{\text{rows}} \times \text{filter}_{\text{cols}} - c_j \times \text{stride}_{\text{cols}} \\
\text{workspace}_{\text{index}_{ij}} &= \text{channel}_{\text{input}} \times (\text{filter}_{\text{rows}} \times \text{filter}_{\text{cols}}) \\
&\quad + \text{element}_{\text{row}_i} \times \text{dprod}_{\text{size}} \\
&\quad + \text{element}_{\text{col}_j}
\end{aligned}$$

where the indices (i, j) are equivalent to a two-dimensional loop with elements

[1]Note that fix_{col} and fix_{row} in the following equation will only be non-zero in the case of convolutional kernels that have filters within a row or column that do not exactly meet the edge of the input image, which rarely appear in the real world (e.g., in AlexNet's first layer with an input size of 228x228, filter size of 11x11, and stride of 4x4). Usually, convolution will be aligned so that no extra weight or input values are included in the model which are unused due to unaligned filters, so fix_{col} and fix_{row} can be discarded in most models.

$$i = \{0, 1, ..., \text{repeat}_{\text{row}} - 1\}$$

$$j = \{0, 1, ..., \text{repeat}_{\text{col}} - 1\}$$

Finally, the workspace index must be translated back into a byte address by using the starting address of the lowered matrix and the size of each data type.

$$\text{address}_{ij} = \text{input}_{\text{start\_address}} + \text{workspace}_{\text{index}_{ij}} \times \text{size}_{\text{data\_type}}$$

The result of the above equations is a function which can turn an element ID into a list of physical addresses within the lowered matrix that map to the same location as that element ID and is an integral part of the proposed method. While Duplo must use an auxiliary data structure to map element IDs to registers, we can work backwards from the addresses and identify the duplication within the L2 cache without requiring an auxiliary structure at all.

The proposed method is depicted generally in figure 3.8. First, a cache read is sent to the L2. Both the L1 and L2 are sectored, splitting each cache line of 128 bytes into four groups of 32 bytes each, so our technique need only fill 32 bytes at a time. There are three situations we consider:

(1) The access already exists in the cache. This is rendered a hit, and the requested 32-byte sector is returned to the L1 cache.

(2) The access does not *directly* exist in the cache, but it can *indirectly* be created from a collection of other accesses. This is shown visually in the figure itself: each of the 16 element IDs within a 32-byte sector request is calculated, and the cache is searched for those element IDs. Since a single element ID can have different physical addresses that map to it, each of the physical addresses are checked in the cache to see if they exist using the inverted equation shown previously. If all 16 element IDs are hits, then a new line is constructed and returned to the L1 cache which is a combination of the other lines without issuing a read to DRAM. In the figure,

sector 0 consists of four values (eight bytes) which all hit in various locations within the cache, and is thus returned to the L1.

(3) The access cannot be indirectly created from other existing cache lines. Then, the line accesses is forwarded to DRAM and inserted into the L2 on DRAM response.

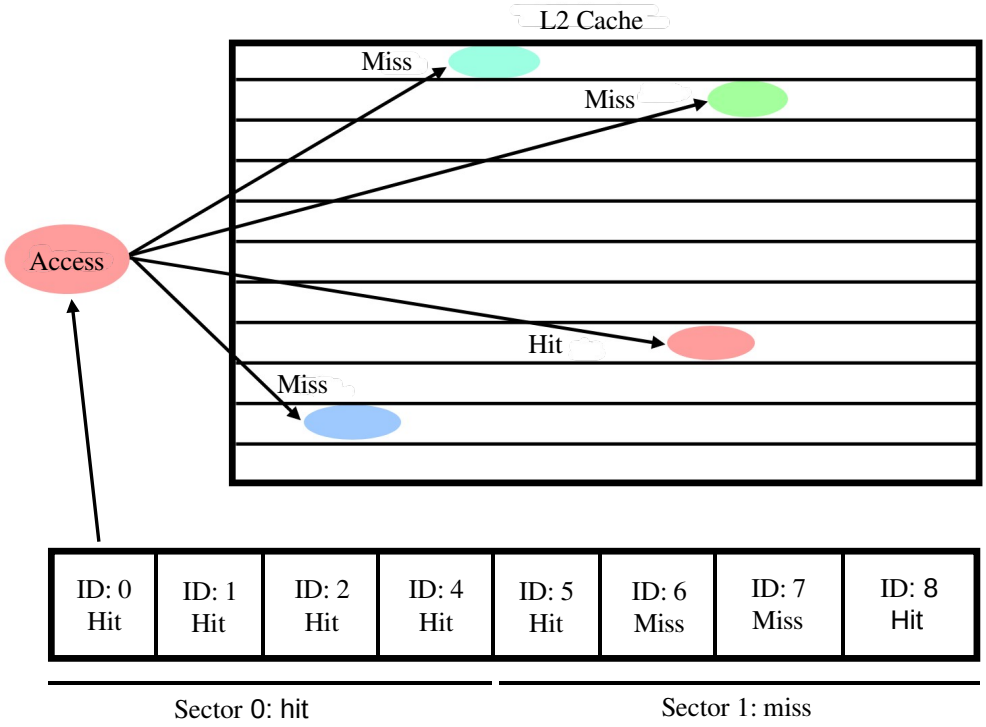


FIGURE 3.8. Potential cache design which implements the de-duplication technique. A 16-byte, or 8-element, cache access is sent to the L2, and each element ID is calculated with a technique similar to Duplo's [26]. Then, each element ID is translated to a list of addresses, which are each checked to see if at least one exists elsewhere within the cache for each element. If all elements within a sector have at least one address mapping to the same ID, then an entire sector is formed and sent to the L1.

This design is then exclusive in that the L1 might contain a sector or line which does not exist in the same format in the L2 cache. The design would benefit from smaller sectors, exemplified in figure 3.7 which shows how a sector size of 2 bytes would perform much better than a more realistic sector size of 32 bytes.

The proposed design improves several of Duplo's faults. First, it negates the requirement of a secondary data structure for containing element IDs. Since the size of the L2 cache is much greater than the register file, an auxiliary data structure containing all element IDs within the L2 cache would be impossible to maintain. The inverted equation shown previously is beneficial because it allows the calculation of addresses mapping to the same information without directly representing it. Second, Duplo's method was unable to prevent the duplication within the lower-level cache structures and thus could not afford to decrease the L2 size any further from the baseline (see figure 3.6), while our design can do this due to a potential reduction in duplicated data and improvement in effective L2 capacity.

A potential drawback of our method is the computational requirements of the inverted equation. While there are only integer expressions involved, it lies on the critical path of the L2 accesses. Although, the high latency-hiding ability of GPUs might make this setback unimportant, as a savings in DRAM might be more beneficial than a small increase in L2 access time. Another potential drawback of our method is the requirement of indexing multiple cache lines for each sector accessed. While L2 lines are highly banked and accesses are already parallelized, this still leads to a high probability of bank conflicts. An ideal solution to this problem is thus likely a dual solution combining an auxiliary data structure to prevent frequent L2 accesses and our inverted equation to reduce DRAM reads.

CHAPTER 4

CHARACTERIZING LLM ACCELERATION USING GPUS

Large language models (LLMs) are becoming increasingly popular as conversational chatbots are engaging on a wide variety of tasks with fluent English proficiency, spanning natural language understanding to question answering and semantic analysis. These models are distinct from other types of models in their sheer scale: many of them contain billions to trillions of parameters and require weeks to fully train and fine-tune. Megatron-LM is one example of a project which utilized more than 6,000 GPUs to distributively train an LLM [44], which is only possible due to the wide array of parallelization utilities NVIDIA and PyTorch offer.

The large size and high complexity of LLMs make optimization a challenging task. This chapter analyzes several of the hardware characteristics of two LLMs, BERT [11] and GPT-2 [51], describing bottlenecks, important software/hardware details, and more. It begins with a schematic and high-level description of LLMs and how they differ from other recurrent networks, and it transitions to an exploration of the execution profile of our two networks. It ends with future research directions on optimizing these models.

4.1. Large Language Models and Distributed Parallelism

To begin, recurrent neural networks have been studied as long as non-recurrent deep-neural networks have. Generally, a recurrent operator can be imagined as an operator which takes its own output in a previous iteration as an input for the current iteration. This runs into an important issue when it comes to very long sequences, in which small deviations in weights cause the gradients to tend to zero (called the vanishing gradients problem [19]), which was addressed in the seminal Long Short-Term Memory (LSTM) paper with the addition of additional gates that selectively remember and forget information.

A typical implementation of a recurrent neural network in the domain of language translation is with the encoder-decoder architecture [57], depicted in figure 4.1. In it, an input sequence is represented as a variable-width collection of tokens (e.g., words in French). The

encoder is the first half of the model, which updates once per token in the input sequence. Each update applies some function to the token and to its hidden state to create a new fixed-width hidden state, which is used as input to the next iteration of the encoder for the next token in the sequence. After all tokens of the input is processed, the encoder passes its fixed-width hidden state to the *decoder* in the second half of the model. The decoder then updates itself with the encoder's hidden state as an input, outputting a new hidden state which is used as the decoder's input in the next iteration. The output of the decoder is this hidden state as well as a new sequence of tokens containing a translated sentence.

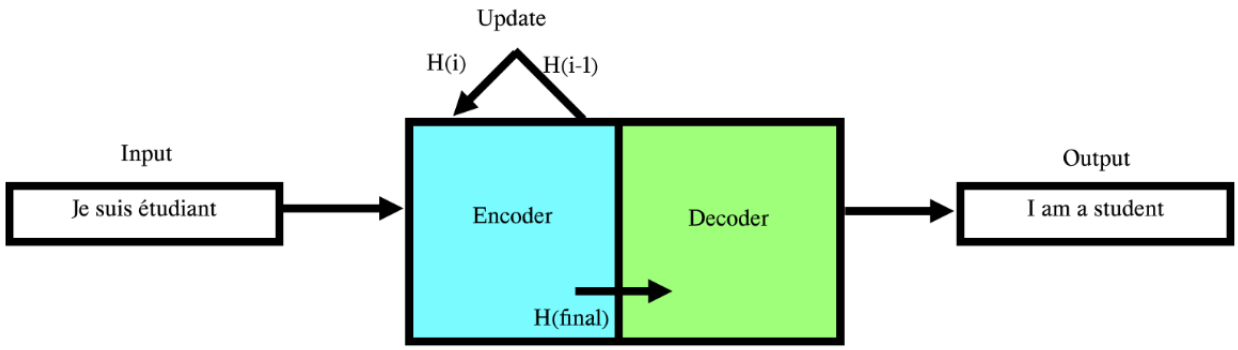


FIGURE 4.1. The encoder-decoder architecture takes an input sequence and updates a fixed-width hidden parameter once per token in the input. Then, once the entire input is processed, the fixed-width hidden state is passed to a decoder module, which updates a new hidden state a fixed number of times to generate a new output sequence.

A critical downside of this encoder-decoder model is the restriction that the output of the encoder is a fixed-width vector. These vectors can encode short-term relationships in input sequences and can represent non-linear translations in which a sentence does not align 1-1 with its translation, but it is overall poor in encoding long-term relationships in strings. If a sentence has two words that contradict each other, separated by a long sequence of words between them, then the fixed-width representation needs to encode the information of the first word within itself at least until the contradicting word is found. It is apparent that any fixed-width representation will eventually encounter issues as the size of the input increases;

at the least, a section of the fixed-width state must be reserved to represent certain words, which is impossible to do for all words in a very long sequence.

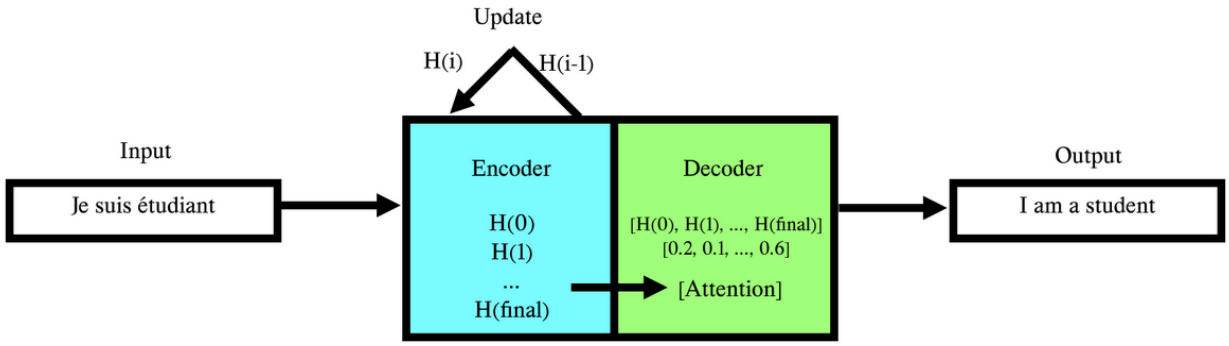


FIGURE 4.2. The attention mechanism allows the decoder to accept *every* hidden state from the encoder instead of just the encoder's final state. The attention mechanism can then selectively pay attention to certain hidden states at a time.

The *attention* mechanism was devised to resolve this issue, with a corresponding transformer architecture which implements the attention mechanism on an encoder-decoder model [61], shown in figure 4.2. Instead of only passing the final hidden state of the encoder, the transformer passes *all* of the encoder's hidden states. While this is a huge increase in dimensionality and would degrade performance in most cases, the attention mechanism informs the decoder on which hidden states to pay "attention" to. This is formulated as a softmax of coefficients on each hidden state, which are summed together to form a new hidden state at each iteration of the decoder. This hidden state is then processed in the same way as the previous encoder-decoder representation. The result of this method is the ability to better represent long-term dependencies in input sequences: since the hidden state of the start and end of the contradicting tokens are both given to the decoder, the attention mechanism can choose to give attention to these far-apart tokens and use them to form the translation of the next output token. Transformers significantly improved performance in the language-understanding domain, and have been successful in the image-processing domain especially for multi-modal data.

The ability of transformer models to represent longer sequences has come with it a significant increase in parameter counts. Transformers can vary in the number of attention heads, with each head containing a method of processing the input with different matrix-multiplications and normalizations, along with the number of layers within each encoder/decoder and the number of encoder/decoders themselves. In fact, these parameter counts have become so large that they are unable to fit within a single GPU, which warrants multi-GPU inference and training solutions to prevent the model from loading and storing a portion of the model for each forwards/backwards pass. Megatron-LM is the most striking example of distributed GPUs being used, with their largest 462-billion parameter model being trained on 6144 GPUs.

Distributed GPUs communicate with NVIDIA's NCCL communication collectives library. Four of the most important functions are:

- ReduceScatter: each GPU combines their tensors through some reduction operation (e.g., element-wise addition, element-wise multiplication, etc.), then distributes a portion of the resulting tensor to each GPU.
- AllReduce: a reduction operation combines each input tensor together.
- Broadcast: a tensor from one GPU is sent to every other GPU.
- AllGather: each GPU distributes a small section of a tensor to every other GPU, forming a large tensor in the process.

NVIDIA GPUs use NVLink to directly connect GPUs to each other with a communications link, instead of requiring GPUs transfer data through the CPU as an intermediary. NVLink enables this high-bandwidth multi-GPU communication with configurable topologies and dynamic transfer rates, making large networks of GPUs like Megatron-LM's flexible and elastic.

The method of distributing tensors across each GPU is another configurable design parameter. This distribution method can be classified as either:

- Data parallel: an input sequence consists of multiple distinct examples organized in a batch, which is then split and evenly distributed to each GPU. This requires

each GPU to contain a full copy of the model weights, which is not ideal or even possible in many LLMs, making this method better suited for small models that can completely fit within a single GPU.

- Tensor parallel: a deep-learning operation like matrix multiplication has its weights and inputs split into different sub-operations for each device, and their results are collectively reduced and communicated after the operation completes. In the case of matrix multiplication, the operation can be visualized instead as a sequence of independent dot-products; thus, the dot-products can be split, with each part being given to a different GPU to complete. This requires much more communication, but it results in less duplication among model weights overall.
- Pipeline parallel: each operation of the model is given to a different GPU. This results in stalls on some GPUs for the start of the model, but it can lead to better utilization and less intra-operator communication. Since communication and computation occupy different parts of the hardware, the system can distribute the output for an operation while calculating the operation on the next input. Although, this method is inefficient if the models are very deep or if the model does not have enough inputs to fill the pipeline, which is similar drawbacks to a pipeline within a processor.

Each of these design parameters describe the execution state of the training and inference on each LLM. The final detail to note is that most of the compute-heavy operations within the LLM can be reduced to GEMMs. As mentioned in chapter 3, GEMMs are very efficient on the hardware, so it is good that the multi-headed attention mechanism within transformers can be expressed in a way to utilize tensor cores. The next section uses a GPU simulator along with NVIDIA's profiler to describe different hardware and software characteristics of each LLM compared with the image-recognition models from chapter 3.

4.2. Hardware and Software Characterization

First, an immediate distinction between the image-recognition networks shown previously and the large-language models given here is their relative complexity: figure 4.3

shows a schematic of the transformer architecture, which is significantly more complex than the image recognition networks given in 3.1 due to it having more branching collections of operators and significantly more operators in general. Additionally, the GPU kernels themselves divide operators into smaller groups so that a single operator might correspond to many different GPU kernels being executed. We ran a custom implementation of AlexNet in PyTorch against NVIDIA's version of BERT and compared statistics between the two in table 4.1.

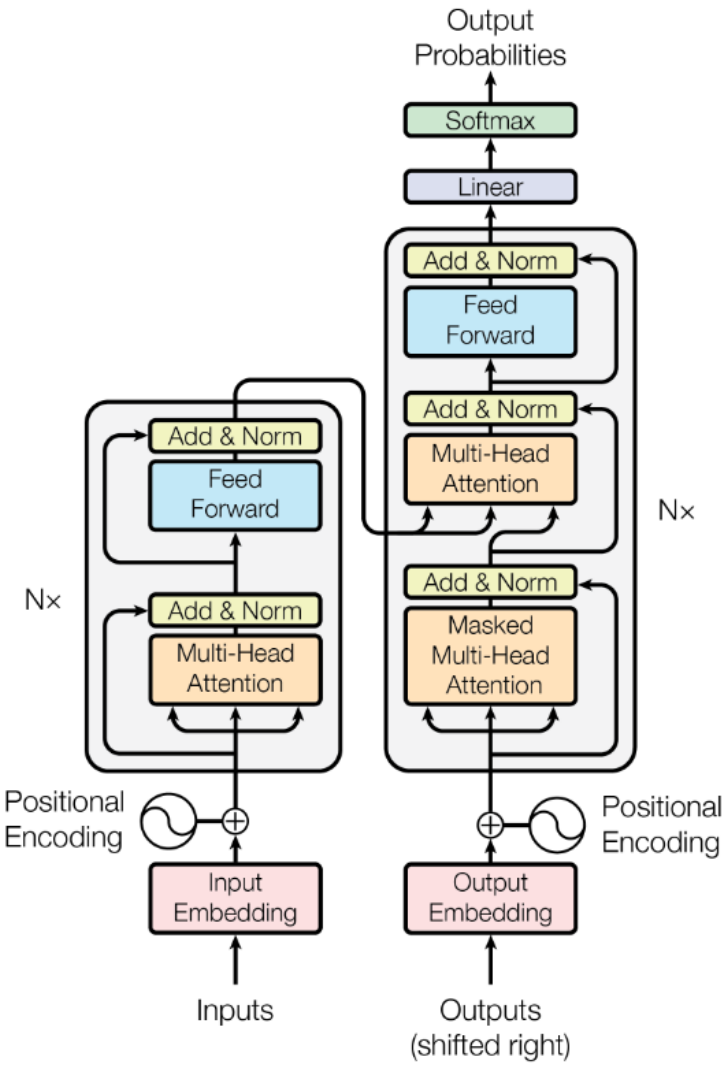


FIGURE 4.3. The transformer LLM architecture. Reproduced from the original paper [61] with permission.

Name	AlexNet	BERT
Parameters	62 million	110 million
Details	11 layers	12 encoders, 12 attention heads
Input	1 image, inference	64-example training, 8-batch, 2 epochs
GEMM Kernels	8	274,825
Non-GEMM Kernels	0	638,408

TABLE 4.1. Comparison between the model parameters and execution environment for AlexNet and BERT.

While our execution environment was different between the two networks, with BERT performing training and AlexNet performing inference, the increase in GEMM and non-GEMM kernels is dramatic. The workload contains both GEMM kernels—which represent most of the compute-heavy operations like multi-headed attention and fully-connected layers—as well as non-GEMM layers, like element-wise operations, softmaxes, normalizations, and data movement operations. In BERT's case, we further characterized the execution time of each GPU kernel type (GEMM vs. non-GEMM) on a single GPU and found that GEMM is the bottleneck at 41 seconds versus non-GEMM at 30 seconds.

Looking more closely at the individual kernels which get executed, we can find patterns in the kernel types over time. Specifically, we found groups of 28 kernels that repeat itself frequently across the entire program runtime, up to 12 times in a row and again at separate points in the program, shown in figure 4.4.

This most likely corresponds to a single transformer block, either in the encoder or the decoder depending on the model used (for example, BERT lacks a decoder and GPT-2 lacks an encoder, so any specific repetition found in either most likely corresponds to the module the network does have). It is also possible that operators are split further so that a single operator in the high-level operator graph of the architecture is divided into multiple sub-operations using a parallelism technique like data-parallelism or tensor-parallelism. Splitting operators into multiple groups can be a smart way of increasing the total utilization of the

hardware, as multiple concurrently-executing sub-operators might increase the amount of computation happening on the GPU at any moment.

(1) GEMM
(2) Elementwise
(3) Dropout
(4) Elementwise
(5) Layer Normalization
(6) Layer Normalization
(7) GEMM
(8) Elementwise
(9) Elementwise
(10) GEMM
(11) Elementwise
(12) Dropout
(13) Elementwise
(14) Layer Normalization
(15) Layer Normalization
(16) GEMM
(17) Elementwise
(18) GEMM
(19) Elementwise
(20) GEMM
(21) Elementwise
(22) GEMM
(23) Elementwise
(24) Elementwise
(25) Softmax
(26) Dropout
(27) GEMM
(28) Elementwise

FIGURE 4.4. A repeated block of 28 kernels within the single-GPU BERT.

Since these groups of kernels appear so frequently within BERT, they were analyzed further with our GPU simulator. Several general statistics can be shown in figure 4.5, in which "L2 cache size = 1.0" can be interpreted as the baseline statistics for each type of operator. From the figure, it can be seen that GEMM remains the bottleneck on an operator-level, as it has the highest cycle count and the lowest IPC. The other operators are either streaming (e.g., each data element within the tensors are loaded exactly once) or are semi-streaming, which explains their high IPC due to a low requirement for on-chip computational resources, at least compared to GEMM which has a much higher demand for shared memory and tensor cores.

The IPC does not seem to be affected for GEMM except in some cases, which is nearly the same as the image-networks shown in chapter 3. The streaming operators should have no effect with increasing the L2 size in theory, but in practice, qualities like data access patterns, interconnect congestion, and schedulers might slightly change the ordering of instructions

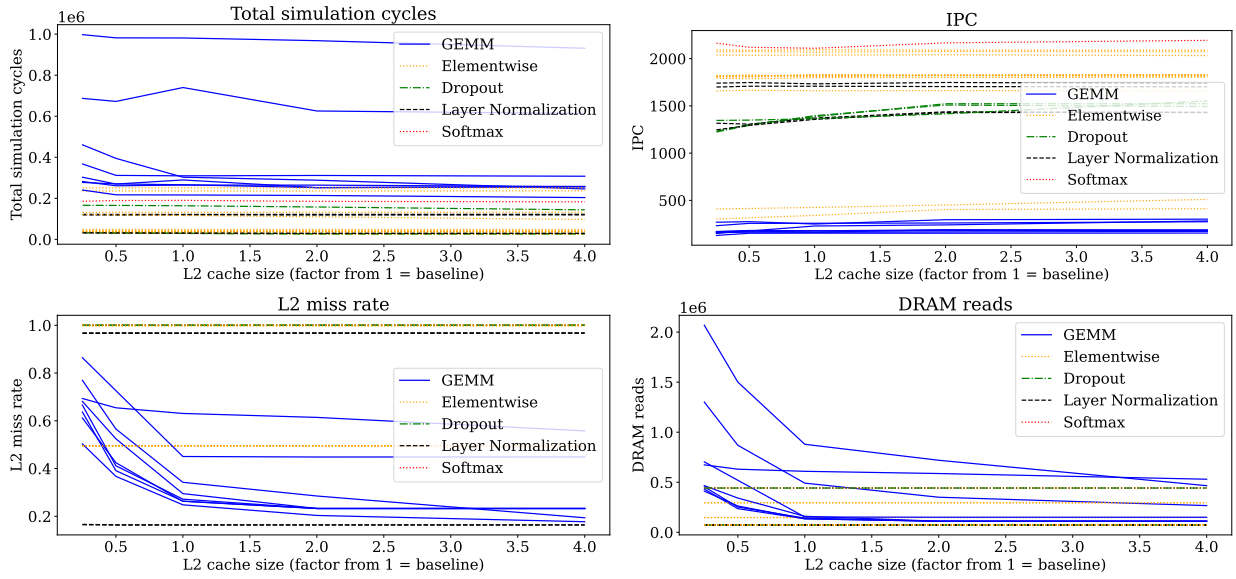


FIGURE 4.5. Comparison between different layers in a repeated group within BERT.

and cause bank conflicts where there would not be any in other cases, which occurs in the IPC slightly reducing with a changed cache size in a softmax layer. Although, the benefits of the L2 cache is more clearly seen with the L2 miss rate and DRAM reads, as changing the size at all has a clear and sometimes drastic effect on both metrics, justifying the size of the current cache overall.

Cache statistics are given in figure 4.6 for the kernel group. The simulator was adjusted to log each L2 cache access to a file, which was then parsed to measure the amount of overlap between two subsequent kernels. From the results, we can see that one kernel, Elementwise/Softmax, experiences a perfect 100% overlap, while most other kernels are around or below 40%. A 100% overlap implies all the data operated on by one kernel is used by the next kernel, which is the case for subsequent one-to-one operators that map to the same input and output buffers. Although, operators which use any kind of weights will naturally have less than 100% overlap because the weights accessed by one kernel will most likely not be accessed by the next kernel.

More cache statistics are shown in figure 4.7, depicting the total access, unique accesses, and revisit frequency (total / unique) for each type of GPU kernel within the repeated

BERT block. GEMM has a cache revisit frequency of about 6.3, meaning each address within the L2 cache is revisited on average 6.3 times, while the streaming dropout and softmax kernels have close to 1.0 revisit frequency. It is important to note that, for each data element within the memory system, the revisit frequency might be significantly higher if shared memory and the L1 cache was incorporated in the statistic: since GEMM kernels utilize shared memory to a high degree, it would make sense for large amounts of the GEMM data to be cached in shared memory, while the L2 accesses are reserved for data that was not able to fit within the shared memory.

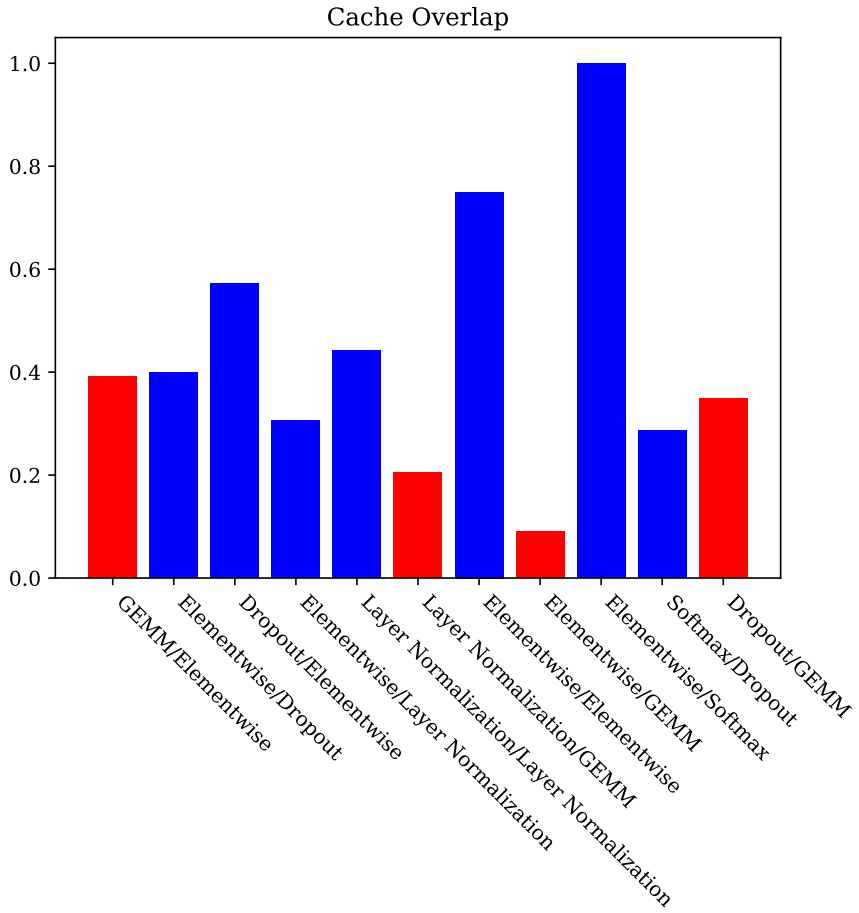
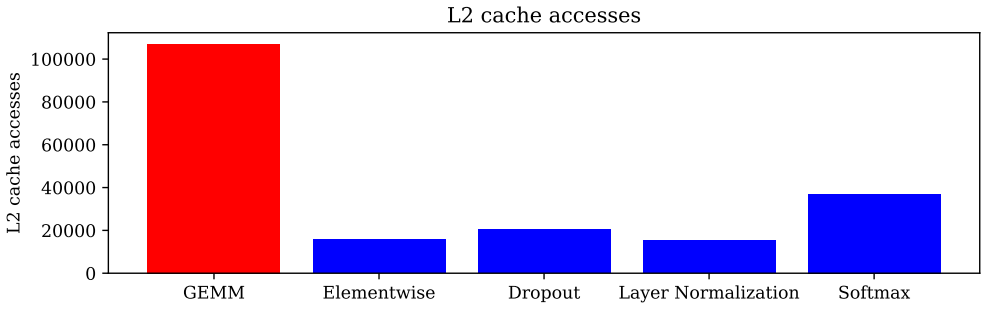
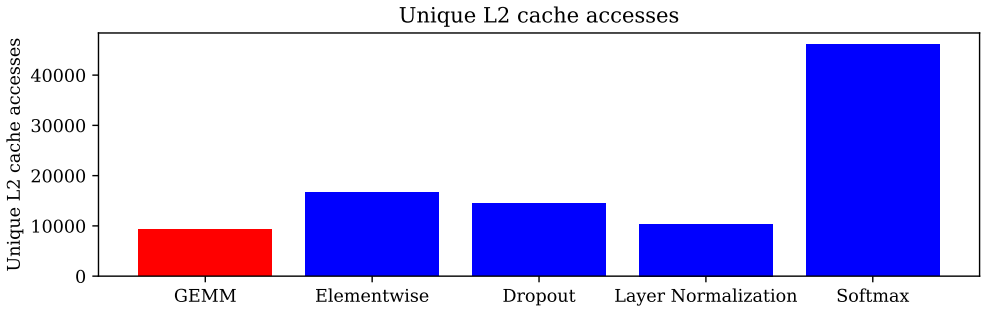


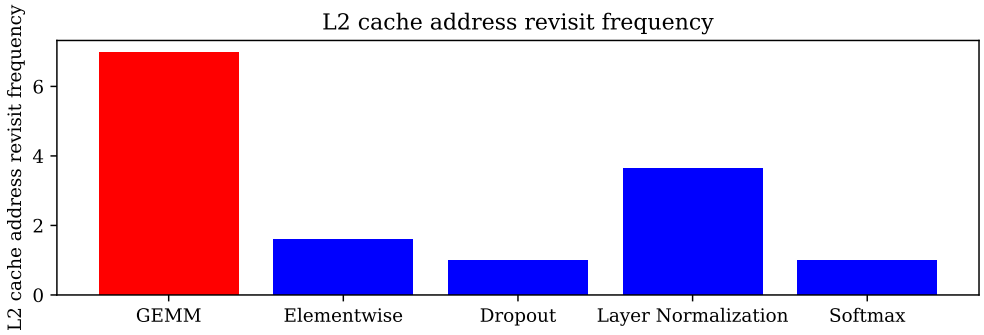
FIGURE 4.6. Of all the L2 cache accesses two successive layers make within the BERT repeated block (figure 4.4), this is the average amount of overlap between them as a percentage of the total accesses. Operators involving GEMM are highlighted in red.



(A)



(B)



(C)

FIGURE 4.7. Comparison in cache access stats between each operator in a repeated BERT block (figure 4.4), with (a) the total access counts to the L2 cache, (b) the total *unique* access counts, and (c) the "revisit rate", or the total number of accesses divided by the number of unique accesses.

Finally, we analyze the operator execution times in the case of multiple GPUs connected to each other for different data distribution techniques, shown in table 4.2. The most obvious detail is that the amount of communication between operators increases when using

tensor parallelism compared to data parallelism, due to the fact data parallelism requires only inter-operator synchronization and tensor parallelism requires inter-operator *and* intra-operator synchronization. Therefore, a model which is severely limited in communication capacity might benefit from distributing the data with data-parallelism to reduce the amount of communication being performed; then, the bottleneck is moved mainly to GEMM operations, which is also visible in the case of single GPUs as seen in image-recognition deep neural networks (see chapter 3) and in BERT (see figure 4.7). Besides this, the latency for non-GEMM computation kernels is overall insignificant for tensor parallel models, but relatively more significant for data parallel models.

This section described an analysis of LLMs on GPUs from both a hardware and software perspective. A large focus was placed on the L2 cache system due to its representation of other adversarial statistics; looking at the L2 revisit frequencies informs if a kernel is one-to-one (memory-bound, streaming, element-wise) or many-to-many (computation-bound, as is the case with GEMM), and viewing the miss statistics informs if the program's working set is unable to fit within the scope of the cache system. This data can help inform new optimizations to pursue in terms of both hardware and software.

Name	2-A100-GPT2-XL	4-3090-GPT2-XL	4-3090-GPT2
Parameters	1.5 billion	1.5 billion	124 million
Distribution	Tensor parallel	Tensor parallel	Data parallel
Num GPUs	2	4	4
GPU type	A100	3090	3090
Communication	**57.10%**	**63.80%**	**22.10%**
AllGather	34.30%	30.70%	9.80%
ReduceScatter	21.90%	32.80%	10.50%
AllReduce	0.90%	0.30%	1.80%
GEMM	**30.20%**	**23.10%**	**44.30%**
Other kernels	**9.00%**	**7.40%**	**21.60%**
Element-wise	3.50%	2.40%	9.90%
Softmax	2.70%	2.30%	5.30%
Dropout	2.80%	2.70%	6.40%

TABLE 4.2. Relative operator latencies for GPT-2 on different model sizes, GPUs, and data distribution methods. Operators can either be communication, GEMM, or other; specific kernel percentages are given below their categories.

CHAPTER 5

CONCLUSION AND FUTURE WORK

This thesis discussed software and hardware methods for reducing the execution time of deep-learning workloads. It is important to look at the software from this dual perspective to unearth possible adjustments that might dramatically improve performance; software methods are convenient because they can be done readily on existing devices, but hardware methods are useful for attaining more performance than what is possible through software alone.

Chapter 2 described the deep-learning compiler and surveyed existing methods in the field. By exploiting the unique domain characteristics of constant-runtime[1] transformational functions with no software side-effects, iterative methods can be used to repeatedly compile, execute, and adjust program optimizations to discover fast-running programs. Simple optimization approaches based on simulated annealing and genetic algorithms can apply rules and heuristics to create a search space of loop transformations that improve cache utilization and reduce redundant memory accesses of deep-learning software better than any deterministic compiler could, and with the benefit of working on all devices with no hardware adjustments required.

Chapter 3 gave an overview of image-processing deep-neural networks like AlexNet, VGG16, and ResNet18, along with a specific modification to the GPU L2 cache which can reduce the inherent duplication present in 2d-convolution-lowered matrix-multiplication kernels. The duplication is due to the convolution filter sliding across the same elements within the input image, so that when the filter dot-products are lowered to a matrix multiplication, duplicated elements appear in different spaces in the outputted matrix. Previous work showed how to predict the duplication through a function which maps physical addresses to unique identifiers such that two addresses that map to the same identifier must contain the

[1]While not discussed in this thesis, some deep learning models are *not* constant in their runtime, including tree-based, variable-sized, and recurrent networks. Deep-learning compilers have been created for these unique models by using additional techniques to account for potential differences in execution time between different inputs.

same data, and our work has inverted this function to instead map identifiers to a list of physical addresses. We proposed a cache design which can exploit this mapping to mitigate DRAM loads if a cache sector could be allocated with the same data found elsewhere in the cache, ultimately allowing us to reduce the physical size of the L2 cache while not experiencing a reduction in the effective cache capacity, giving us the opportunity to use the saved space and power for more other important hardware modules like SMs.

Chapter 4 detailed how large-language models work, especially in regards to parallelizing them across multiple GPUs with different communication collectives and operation-splitting techniques, and characterized two large language models run on one GPU and distributed across several GPUs. Through these results, we found that communication takes a significant portion of the execution time as GPUs increase, and that matrix-multiplications are the single most time-consuming operator within each model type. Furthermore, we showed how the memory access statistics appear in different LLMs, specifically with the amount of overlap in accessed memory between subsequent operators in a repeated block of GPU kernels and with the amount of revisited memory accesses within each single operator.

Future work will investigate on the latter two chapters further. While a method was proposed for the lowered cache which might improve the L2 hit rate and effective capacity, it was not yet implemented on a GPU simulator besides preliminary tests. Additionally, it would be beneficial to test it on batch sizes with more than one image in it. An important detail to note about the lowering method is that *overall* duplication reduces with a larger batch size: since duplication cannot exist *between* two images within a convolution, the performance of the lowering method can only decrease as images are added. Investigating the extent to which this method can be applied as the batch size increases is important. Furthermore, obtaining power and area statistics would be beneficial to accurately characterize the effect of the design. While the L2 access latency would no doubt be increased with the proposed design, a characterization of the relationship between L2 latency and total IPC, similar to OLE's figure seven [23] for GPU workloads would help as well. GPUs are in a more special circumstance compared with OLE's CPU implementation because GPUs are

much more effective at hiding high memory latency, so it could be that our design is afforded much more leeway in access latency, allowing our high-latency task to still succeed.

For the LLM project, one idea which we are currently investigating is with prefetching: if we assume a *streaming* operator is executed directly before a *GEMM*, then we can forward all memory accesses from the streaming operator directly to DRAM, bypassing the cache. Then, we can load GEMM weights into the caches in the background while the streaming operator is executing. Because streaming operators do not benefit from a cache since each data element is accessed exactly once, we can fill the L2 with prefetched data during that time. The GEMM operations have a high data reuse rate, so prefetching their values has a good chance of improving overall utilization. The only requirement is that the duration of the streaming operators must be long enough to prefetch a significant portion of the GEMM tensors, and the GEMM-prefetching cannot significantly increase the latency of the streaming operator due to bank conflicts or else the overall system performance would decrease.

Besides these, future work should focus on jointly optimizing hardware and software for deep learning applications. By exploiting the unique characteristics of these applications, significant gains can be obtained in performance when the software is tuned to be aware of the low-level hardware details. Deep learning compilers are one way of doing this, and it can be coupled with custom accelerator designs to automatically discover the best software organization for that accelerator. Future researchers can use the ideas presented within this thesis to develop new hardware and software designs for the GPU that maximize the utilization and reduce unnecessary hardware components so that more important components may be added without penalty.

REFERENCES

[1] T. Abtahi, A. Kulkarni, H. Homayoun, and T. Mohsenin, *Accelerating convolutional neural network with fft on embedded hardware*, IEEE Transactions on Very Large Scale Integration Systems (2018).

[2] A.R. Alameldeen and D.A. Wood, *Frequent pattern compression: A significance-based compression scheme for l2 caches*, Department of Computer Science at the University of Wisconsin-Madison, Technical Report (2004).

[3] M.W. Benabderrahmane, L.N. Pouchet, A. Cohen, and C. Bastoul, *The polyhedral model is more widely applicable than you think*, Proceedings of the International Conference on Compiler Construction (2010).

[4] D. Bertsimas and J. Tsitsiklis, *Simulated annealing*, Statistical Science 8 (1993), 10–15.

[5] W.E. Carlson, *Computer graphics and computer animation: A restrospective overview*, pp. 379–384, The Ohio State University, 2017.

[6] T. Chen, T. Moreau, Z. Jiang, L. Zheng, and E. Yan, *Tvm: An automated end-to-end optimizing compiler for deep learning*, Proceedings of the 13th USENIX Symposium on Operating System Design and Implementation (2018).

[7] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, *Learning to optimize tensor programs*, Advances in Neural Information Processing Systems (2018).

[8] Y. Chen, Y. Xie, L. Song, F. Chen, and T. Tang, *A survey of accelerator architectures for deep neural networks*, Engineering (2020), 264–274.

[9] A. Choromanska, M. Henaff, M. Mathieu, G.B. Arous, and Y. LeCun, *The loss surfaces of multilayer networks*, Artificial Intelligence and Statistics 38 (2015), 192–204.

[10] W.J. Dally, S.W. Keckler, and D.B. Kirk, *Evolution of the graphics processing unit (gpu)*, IEEE MICRO 41 (2021), 42–51.

[11] J. Devlin, M.W. Chang, K. Lee, and K. Toutanova, *Bert: Pre-training of deep bidirectional transformers for language understanding*, arXiv (2018), arXiv:1810.04805.

[12] S. Feng, B. Hou, H. Jin, W. Lin, J. Shao, R. Lai, Z. Ye, L. Zheng, C.H. Yu, Y. Yu, and T. Chen, *Tensorir: An abstraction for automatic tensorized program optimizations*, ACM International Conference on Architectural Support for Programming Languages and Operating Systems (2023), 804–817.

[13] R. Fernando, *Gpu gems: Programming techniques, tips and tricks for real-time graphics*, Addison-Wesley Professional, 2004.

[14] P. Gera, H. Kim, H. Kim, S. Hong, V. George, and C. Luk, *Performance characterization and simulation of intel's integrated gpu architecture*, IEEE International Symposium on Performance Analysis of Systems and Software (2018).

[15] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, IEEE Conference on Computer Vision and Pattern Recognition (2016).

[16] J.L. Hennessy and D.A. Patterson, *Computer architecture: A quantitative approach*, 5 ed., pp. 261–333, Morgan Kaufmann, 2012.

[17] O. Hennigh, S. Narasimhan, M.A. Nabian, A. Subramaniam, K. Tangsali, M. Rietmann, J.A. Ferrandis, W. Byeon, Z. Fang, and S. Choudhry, *Nvidia simnet: An ai-accelerated multi-physics simulation framework*, International Conference on Computational Science (2021), 447–461.

[18] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard, *Multi-gpu and multi-cpu parallelization for interactive physics simulation*, Euro-Par Parallel Processing (2010), 235–246.

[19] S. Hochreiter and J. Schmidhuber, *Long short-term memory*, Neural Computation 9 (1997), 1735–1780.

[20] A.G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, *Mobilenets: Efficient convolutional neural networks for mobile vision applications*, arXiv (2017), arXiv:1704.04861.

[21] G. Huang, Y. Bai, L. Liu, Y. Wang, B. Yu, Y. Ding, and Y. Xie, *Alcop: Automatic load-compute pipelining in deep learning compiler for ai-gpus*, Conference on Machine Learning and Systems (2023).

[22] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, *Taso: Optimizing deep learning computation with automatic generation of graph substitutions*, Proceedings of the 27th ACM Symposium on Operating System Principles (2019), 47–62.

[23] M. Kang, S. Hyun, T.H. Han, J. Kim, and S. Hong, *On-the-fly lowering engine: offloading data layout conversion for convolutional neural networks*, IEEE Access (2022).

[24] F. Khorasani, H.A. Esfeden, N. Abu-Ghazaleh, and V. Sarkar, *In-register parameter caching for dynamic neural nets with virtual persistent processor specialization*, IEEE/ACM International Symposium on Microarchitecture (MICRO) (2018).

[25] J. Kileel, M. Trager, and J. Bruna, *On the expressive power of deep polynomial neural networks*, Conference on Neural Information Processing Systems (2019).

[26] H. Kim, S. Ahn, Y. Oh, B. Kim, W.W. Ro, and W.J. Song, *Duplo: Lifting redundant memory accesses of deep neural networks for gpu tensor cores*, IEEE/ACM International Symposium on Microarchitecture (2020).

[27] Y.D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, *Compression of deep convolutional neural networks for fast and low power mobile applications*, International Conference on Learning Representations (2016).

[28] A. Krizhevsky, I. Sutskever, and G.E. Hinton, *Imagenet classification with deep convolutional neural networks*, Advances in Neural Information Processing Systems 25 (2012), 1097–1105.

[29] D. Kroft, *Lockup-free instruction fetch/prefetch cache organization*, International Symposium on Computer Architecture (1981), 81–87.

[30] J. Krüger and R. Westermann, *Linear algebra operators for gpu implementation of numerical algorithms*, ACM Transactions on Graphics 22 (2003), 908–916.

[31] J. Kwack, J. Tramm, C. Bertoni, Y. Ghadar, B. Homerding, and E. Rangel, *Evaluation of performance portability of applications and mini-apps across amd, intel, and nvidia gpus*, International Workshop on Performane, Portability, and Productivity in HPC (2021).

[32] A. Lambora, K. Gupta, and K. Chopra, *Genetic algorithm: A literature review*, International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (2019).

[33] A. Lavin and S. Gray, *Fast algorithms for convolutional neural networks*, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (2016).

[34] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, *Speeding-up convolutional neural networks using fine-tuned cp-decomposition*, International Conference on Learning Representations (2015).

[35] Y. LeCun, B. Boser, J.S. Denker, D. Henderson, R.E. Howard, W. Hubbard, and L.D. Jackel, *Backpropagation applied to handwritten zip code recognition*, Neural Computation 1 (1989), 541–551.

[36] S. Lee, A. Arunkumar, and C. Wu, *Cawa: Coordinated warp scheduling and cache prioritization for critical warp acceleration of gpgpu workloads*, ACM SIGARCH Computer Architecture News 43 (2015), 515–527.

[37] A. Li, S.L. Song, J. Chen, J. Li, X. Liu, N. Tallent, and K. Barker, *Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect*, IEEE Transactions on Parallel and Distributed Systems (2019).

[38] A. Li, B. Zheng, G. Pekhimenko, and F. Long, *Automatic horizontal fusion for gpu kernels*, IEEE/ACM International Symposium on Code Generation and Optimization (2022), 14–27.

[39] M. Li, Y. Liu, Q. Sun, X. You, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian, *The deep learning compiler: A comprehensive survey*, IEEE Transactions on Parallel and Distributed Systems 32 (2020), 708–727.

[40] X. Lian, B. Yuan, X. Zhu, Y. Wang, Y. He, H. Wu, L. Sun, H. Lyu, C. Liu, X. Dong, Y. Liao, M. Luo, C. Zhang, J. Xie, H. Li, L. Chen, R. Huang, J. Lin, C. Shu, X. Qiu, Z. Liu, D. Kong, L. Yuan, H. Yu, S. Yang, C. Zhang, , and J .Liu., *Persia: An open, hybrid system scaling deep learning-based recommenders up to 100 trillion parameters*,

ACM SIGKDD Conference on Knowledge Discovery and Data Mining (2022), 3288–3298.

[41] S. Markidis, S.W.D. Chien, E. Laure, I.B. Peng, and J.S. Vetter, *Nvidia tensor core programmability, performance and precision*, IEEE International Parallel and Distributed Processing Symposium Workshops (2018), 522–531.

[42] C. Molina, C. Aliagas, M. Garcia, A. Gonzalez, and J. Tubella, *Non redundant data cache*, Proceedings of the 2003 International Symposium on Low Power Electronics and Design (2003), 274–277.

[43] V. Nair and G.E. Hinton, *Rectified linear units improve restricted boltzmann machines*, International Conference on Machine Learning (2010).

[44] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashihkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. Zaharia, *Efficient large-scale language model training on gpu clusters using megatron-lm*, Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (2021), 1–15.

[45] W. Niu, J. Guan, Y. Wang, G. Agrawal, and B. Ren, *Dnnfusion: Accelerating deep neural networks execution with advanced operator fusion*, Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (2021).

[46] *Nvidia tesla v100 gpu architecture*, Tech. report, NVIDIA, 2017.

[47] N. Otterness and J.H. Anderson, *Amd gpus as an alternative to nvidia for supporting real-time workloads*, Euromicro Conference on Real-Time Systems (2020), 1–23.

[48] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips, *Gpu computing*, Proceedings of the IEEE 96 (2008), 879–899.

[49] Y. Panagakis, J. Kossaifi, G.G. Chrysos, J. Oldfield, M.A. Nicolaou, A. Anandkumar, and S. Zafeiriou, *Tensor methods in computer vision and deep learning*, Proceedings of the IEEE (2021).

[50] 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, Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (2012), 377–388.

[51] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, *Language models are unsupervised multitask learners*, 2019.

[52] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, and F. Durand S. Amarasinghe, *Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines*, ACM SIGPLAN Conference on Programming Language Design and Implementation (2013), 519–530.

[53] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Santheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A.C. Berg, and L. Fei-Fei, *Imagenet large scale visual recognition challenge*, International Journal of Computer Vision 115 (2014), 211–252.

[54] I.H. Sarker, *Deep learning: A comprehensive overview on techniques, taxonomy, applications and research directions*, SN Computer Science 2 (2021), 420.

[55] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, *Megatron-lm: Training multi-billion parameter language models using model parallelism*, arXiv (2019), arXiv:1909.08053.

[56] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, International Conference on Learning Representations (2015).

[57] I. Sutskever, O. Vinyals, and Q.V. Le, *Sequence to sequence learning with neural networks*, Conference on Neural Information Processing Systems 2 (2014), 3014–3112.

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

[59] Y. Tan, C. Xu, J. Xie, Z. Yan, H. Jiang, and W. Srisa-an, *Improving the performance of deduplication-based storage cache via content-driven cache management methods*, IEEE Transactions on Parallel and Distributed systems (2020), 214–228.

[60] Y. Tian, S.M. Khan, D.A. Jimenez, and G.H. Loh, *Last-level cache deduplication*, Proceedings of the 28th ACM International Conference on Supercomputing (2014), 53–62.

[61] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, L. Kaiser, and I. Polosukhin, *Attention is all you need*, Neural Information Processing Systems (2017).

[62] F. Wang, J. Dong, and B. Yuan, *Graph-based substructure pattern mining using cuda dynamic parallelism*, Intelligent Data Engineering and Automated Learning (2013), 342–349.

[63] G. Wang, Y. Lin, and W. Yi, *Kernel fusion: An effective method for better power efficiency on multithreaded gpu*, IEEE/ACM International Conference on Green Computing and Communications and International Conference on Cyber, Physical, and Social Computing (2010).

[64] H. Wang, N.S. Keskar, C. Xiong, and R. Socher, *Identifying generalization properties in neural networks*, arXiv (2018), arXiv:1809.07402.

[65] H. Wang and B. Raj, *On the origin of deep learning*, arXiv (2017), arXiv:1702.07800.

[66] J. Wang and S. Yalamanchili, *Characterization and analysis of dynamic parallelism in unstructured gpu applications*, IEEE International Symposium on Workload Characterization (2014).

[67] J. Weizenbaum, *Eliza—a computer program for the study of natural language communication between man and machine*, Communications of the ACM 9 (1966), 36–45.

[68] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, *Integer quantization for deep learning inference: Principles and empirical evaluation*, arXiv (2020), arXiv:2004.09602.

[69] J. Xing, L. Wang, S. Zhang, J. Chen, A. Chen, and Y. Zhu, *Bolt: Bridging the gap between auto-tuners and hardware-native performance*, Proceedings of Machine Learning and Systems (2022), 204–216.

[70] D. Yan, W. Wang, and X. Chu, *Optimizing batched winograd convolution on gpus*, Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (2020), 32–44.

[71] J. Zhao, X. Gao, R. Xia, Z. Zhang, D. Chen, L. Chen, R. Zhang, Z. Geng, B. Cheng,

and X. Jin, *Apollo: Automatic partition-based operator fusion through layer by layer optimization*, Proceedings of the 5th Conference on Machine Learning and Systems (2022).

[72] L. Zheng, C. Jia, M. Sun, Z. Wu, C.H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J.E. Gonzalez, and I. Stoica, *Ansor: Generating high-performance tensor programs for deep learning*, Proceedings of the USENIX Conference on Operating Systems Design and Implementation (2020).

[73] J. Zhong and B. He, *Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling*, IEEE Transactions on Parallel and Distributed Systems 25 (2014).