Blocked Algorithms for Neural Networks: Design and Implementation on GPUs

HARVARD UNIVERSITY

Graduate School of Arts and Sciences

DISSERTATION ACCEPTANCE CERTIFICATE

The undersigned, appointed by the

Harvard John A. Paulson School of Engineering and Applied Sciences
have examined a dissertation entitled:

"Blocked Algorithms for Neural Networks

Design and Implementation on GPUs"

presented by: Philippe Tillet

Signature H.T. Kung
Typed name: Professor H.T. Kung

Signature [illegible]
Typed name: Professor D. Cox

Signature [illegible]
Typed name: Professor D. Brooks

September 1, 2020

Blocked Algorithms for Neural Networks
Design and Implementation on GPUs

a dissertation presented
by
Philippe Tillet
to
School of Engineering and Applied Sciences

in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
in the subject of
Computer Science

Harvard University
Cambridge, Massachusetts
September 2020

Dissertation Advisor: Prof. H. T. Kung; Prof. David D. Cox Philippe Tillet

Blocked Algorithms for Neural Networks

Abstract

The recent emergence of Deep Neural Networks (DNNs) for machine learning has been largely enabled by the widespread availability of massively parallel computing devices. In particular, Graphics Processing Units (GPUs) have played a critical role, allowing the evaluation of increasingly large DNNs on increasingly large datasets. Unfortunately, the development of efficient programs for GPUs remains laborious, requiring advanced knowledge of specialized compute resources (e.g., tensor cores) and complex memory hierarchies (e.g., caches). This has made it challenging to write efficient and reusable libraries for novel research ideas (e.g., sparsity) in the field of Deep Learning.

In this thesis, we argue that programming paradigms based on blocked algorithms can facilitate the construction of high-performance compute kernels for neural networks. We specifically revisit traditional ”single program, multiple data” execution models for GPUs, and propose a variant in which programs – rather than threads – are blocked. We show that algorithms expressed using this paradigm define iteration spaces composed of a collection of blocks whose shape and schedule can be automatically optimized using *context-aware auto-tuning* and *block-level data-flow analysis*, respectively. We present the design and implementation of these novel techniques in the *Triton* language and compiler for blocked algorithms, and achieve significant speed-ups over state-of-the-art libraries (cuBLAS/cuDNN) for a wide range of matrix multiplication and convolution tasks commonly encountered in practice.

We finally show how this approach can facilitate the development of efficient compute kernels for some important emerging neural network architectures. We specifically focus on block-sparse self-attention mechanisms in transformers, and demonstrate significant performance gains for training tasks involving long sequence lengths.

Contents

Citations to Previously Published Work

Portions of Chapter 4 have appeared in the following:

P. Tillet, D. Cox, “Input-Aware Auto-Tuning of Compute-Bound HPC Kernels” in International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2017.

Portions of Chapter 5 have appeared in the following:

P. Tillet, H. T. Kung, D. Cox, “Triton: an intermediate language and compiler for tiled neural network computations” in Annual Machine Learning and Programming Languages Workshop (MAPL) @ PLDI, 2019.

Dedicated to my parents.

Acknowledgments

Thanks to my co-advisors, David Cox and H. T. Kung, for their precious guidance without which this dissertation would never have been possible. They have truly pushed me to surpass myself in so many different ways, and their passion for research has been an endless source of motivation throughout the years.

Thanks to my parents, who have always been here for me, and provided me with the best support system I could have ever hoped for.

Thanks to all my amazing friends, for making this journey fun and joyful. The kindness around me has been strong and unshakeable, but enumerating everyone responsible for it seems impossible. I am particularly grateful to Bradley, Xin, Cathy, Marcus, Donglai and Yu for always being responsive when I needed someone to talk to.

Thanks to everyone in H.T's and David's group during my time at Harvard, not only for their technical insights but also for their good mood and cheerful outlook on life.

Thanks to all the doctors and nurses I have seen throughout my numerous stays at Mount Auburn and Massachusetts General Hospital. I have truly avoided the worst, and will stay forever grateful to everybody who was here for me when I was at my worst.

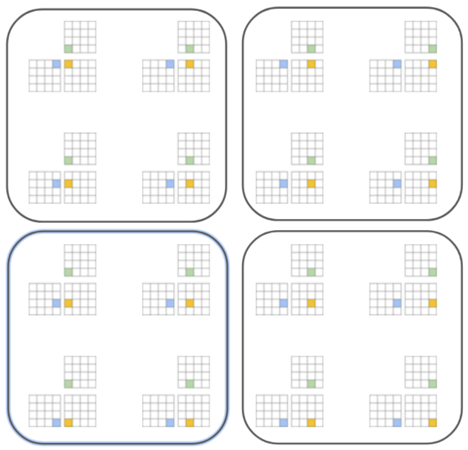
1 Introduction

Over the past decade, Deep Neural Networks (DNNs) have emerged as an important class of Machine Learning (ML) models, capable of achieving state-of-the-art performance across many domains ranging from natural language processing [1] to computer vision [2] to computational neuroscience [3]. The strength of these models lies in their hierarchical structure, composed of a sequence of parametric (e.g., convolutional) and non-parametric (e.g., rectified linearity [4]) *layers.* This pattern, though notoriously computationally expensive, also generates a large amount of highly parallelizable work particularly well suited for multi- and many- core processors.

As a consequence, Graphics Processing Units (GPUs) have become a cheap and accessible resource for exploring and/or deploying novel research ideas in the field. This trend has been accelerated by the release of several frameworks for General-Purpose GPU (GPGPU) computing, such as CUDA and OpenCL, which have made the development of high-performance programs easier. Yet, GPUs remain incredibly challenging to optimize for locality and parallelism, especially for computations that cannot be efficiently implemented using a combination of pre-existing optimized primitives. To make matters worse, GPU architectures are also rapidly evolving and specializing, as evidenced by the recent addition of tensor cores [5] to NVIDIA micro-architectures.

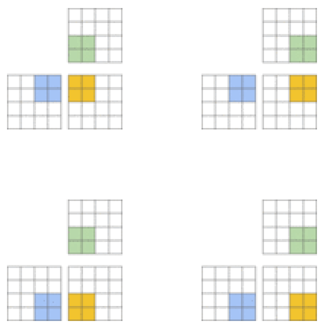
This tension between the computational opportunities offered by DNNs and the practical difficulty of GPU programming has created substantial academic and industrial interest for Domain-Specific Languages (DSLs) and compilers. Regrettably, these systems – whether they be based on polyhedral machinery (*e.g.*, Tiramisu [6], Tensor Comprehensions [7]) or scheduling languages (*e.g.*, Halide [8], TVM [9]) – remain less flexible and significantly slower than the best handwritten compute kernels available in libraries like cuBLAS [10], cuDNN [11] or TensorRT [12].

```
int tm = get_thread_id(0);
int tn = get_thread_id(1);
int bm = get_block_id(0);
int bn = get_block_id(1);
float acc = 0;
for(int k = 0; k < K; k += 1)
  acc += A[bm * MB + tm, k] *
         B[k, bn * NB + tn]
C[m, n] = acc;
```



(a) Matrix multiplication in the standard GPU programming model (scalar program, blocked threads)

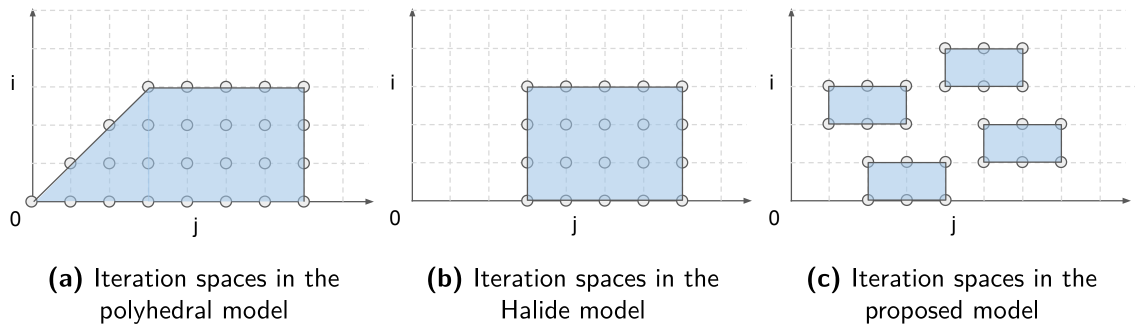
```
int m = get_program_id(0) * MB;
int n = get_program_id(1) * NB;
float acc[MB, NB] = 0;
for(int k = 0; k < K; k += U)
  acc += A[m : m + MB, k : k + U] *
         B[k : k + U,   n : n + NB];
C[m:m+MB, n:n+NB] = acc;
```



(b) Matrix multiplication in the proposed GPU programming model (blocked program, scalar threads)

Figure 1.1: Matrix multiplication in the (a) standard and (b) proposed GPU programming model

In this thesis, we argue that programming paradigms based on blocked algorithms [13] can facilitate the construction of high-performance compute kernels for neural networks. We specifically revisit traditional ”Single Program, Multiple Data” (SPMD [14]) execution models for GPUs (Figure 1.1a), and propose a variant in which programs – rather than threads – are blocked (Figure 1.1b). A key benefit of this approach is that it leads to block-structured iteration spaces (Figures 1.2) that offer programmers more flexibility than existing DSLs when implementing sparse operations, all while allowing compilers to aggressively optimize programs for data locality and parallelism.



(a) Iteration spaces in the polyhedral model **(b)** Iteration spaces in the Halide model **(c)** Iteration spaces in the proposed model

Figure 1.2: Existing (a) polyhedral and (b) schedule-based DSLs enforce restrictive iteration spaces incompatible with many sparse operations.

One of the main challenges posed by our proposed paradigm is that of work granularity, i.e., how much work each program instance should do. To address this issue, in Chapter 4, we present *context-aware auto-tuning*, a method for selecting the shape of each iteration block dynamically based on the value of important runtime parameters not known in advance (e.g., input tensor shapes). Our approach uses simple machine learning techniques (i.e., multi-layer perceptrons) to build a surrogate model for the performance of different block shapes, thereby alleviating the need for empirical performance measurements in auto-tuners [15]. To efficiently generate training data for this method, we propose a simple generative model of potentially efficient block shapes, and implement the relevant blocked algorithms in pseudo-assembly (i.e., PTX [16]) for faster compilation. Numerical experiments show performance often superior to handwritten vendor libraries (cuBLAS/cuDNN), suggesting that well-parameterized blocked algorithms can be competitive with state-of-the-art compute kernels when carefully implemented .

Another challenge posed by our proposed paradigm is that of work scheduling, i.e., how the work done by each program instance should be partitioned for efficient execution on modern GPUs. To address this issue, in Chapter 5, we present *block-level data-flow analysis*, a technique for scheduling iteration blocks statically based on the control- and data-flow structure of the target program. We present the design and implementation of this technique within Triton[1], a language and compiler for blocked algorithms that uses syntax similar to that shown in Figure 1.1b. The resulting system not only outperforms alternative DSLs when applicable, but also often matches the performance of the baseline parameterized pseudo-assembly implementations mentioned above.

Finally, in Chapter 6, we demonstrate how the above contributions can be used to productively build efficient structured-sparse primitives for emerging neural network architectures. We specifically present a collection of blocked algorithms for sparse self-attention mechanisms in transformers, which we enhance using *super-blocking* and *static load-balancing* optimization techniques. The resulting Triton implementation of sparse transformers achieves state-of-the-art throughput when long sequence lengths are used.

1.1 Thesis Roadmap

(**Chapter 2) Background** provides background on Deep Neural Networks, Graphics Processing Units and Compiler Construction. Readers familiar with these topics may proceed directly to Chapter 3.

(**Chapter 3) Related Work** reviews existing domain-specific languages and compilers for deep neural networks. It specifically focuses on polyhedral compilers and scheduling languages, highlighting the advantages and limitations of each solution so as to further motivate the methods introduced in the remainder of this dissertation.

[1] https://triton-lang.org

(**Chapter 4) Context-Aware Auto-Tuning** presents a machine learning-based framework for automatic block shape selection in the proposed block-based SPMD paradigm. It specifically discusses the issue of (1) code generation, (2) data synthesis, (3) regression analysis and (4) runtime inference for context-aware auto-tuners. Numerical experiments on common matrix multiplication and convolution tasks show 0.9-2x speed-ups over cuBLAS and cuDNN .

(**Chapter 5) Block-Level Data-Flow Analysis** presents a language and compiler for automatic program scheduling in the proposed block-based SPMD paradigm. It specifically discusses the issue of (1) specifying blocked algorithms using high-level directives (Triton-C), (2) representing the resulting programs in a format suitable for static analysis (Triton-IR), and (3) automatically optimizing the resulting representation for data locality and parallelism (Triton-JIT). A re-evaluation of the experiments conducted in Chapter 4 shows code quality on par with handwritten pseudo-assembly – and far superior to state-of-the-art DSLs for DNNs.

(**Chapter 6) Fast Sparse Transformers** validates our proposed block-based SPMD paradigm on emerging neural network architectures, and more specifically on block-sparse transformers. To this end, it presents (1) a set of naive blocked algorithms for sparse attention mechanisms in transformers, and (2) a set of optimization techniques that improve data-reuse and load-balancing for these workloads. An evaluation of their implementation in Triton shows state-of-the-art performance for end-to-end training in transformers using long attention windows.

2 Background

In this chapter, we introduce Deep Neural Networks (DNNs) and explain why they are a good fit for parallel processors. We then review the micro-architecture of modern Graphics Processing Units (GPUs) as well as the languages commonly used to program them. Finally, we recall the general architecture of a modern compiler, and discuss the basic principles behind program analysis.

2.1 Deep Neural Networks

2.1.1 General Overview

Deep Neural Networks are a class of hierarchical machine learning models composed of a succession of *layers*, each of which computes a parametric or non-parametric transformation of its input. The parameters (or *weights*) of each parametric layer are tuned to minimize a given *loss function* through a process known as *training*, generally done using the *backpropagation algorithm* [17] shown in Algorithm 1.

Algorithm 1: Backpropagation

```
   Input: Dataset $\mathcal{D}$;  Loss $\mathcal{L}$;  Optimizer $\mathcal{O}$;  Neural Network $\mathcal{N}$
1  repeat
       // Iterate over data-set, with labels if supervised
2      for $(x_n[, y_n])$ in $D$ do
           // Evaluate network output
3          $\hat{y}_n \leftarrow \mathcal{N}(x_n)$
           // Compute loss, using $y_n$ if supervised
4          $L \leftarrow \mathcal{L}(\hat{y}_n[, y_n])$
           // Gradient of the network weights w.r.t loss
5          $dW \leftarrow dL/d\mathcal{N}$
           // Update network weights
6          $\mathcal{N} \leftarrow \mathcal{O}(\mathcal{N}, dW)$
7  until convergence;
```

In this procedure, the weights of a given neural network $\mathcal{N}$ are repeatedly updated using an optimizer $\mathcal{O}$ so as to eventually minimize the value of a loss function $\mathcal{L}$ on a given dataset $\mathcal{D}$. Though different choices for $\mathcal{O}$, $\mathcal{L}$ and $\mathcal{D}$ can dramatically impact the performance of Algorithm 1, in this work we focus specifically on speeding up the evaluation of $\mathcal{N}$ (Line 3) and its gradients (Line 5).

At first sight, the layered structure of $\mathcal{N}$ may appear to create a large diversity of possible networks that could seem difficult to optimize. Fortunately, over the past few decades, a set of standard and regular network architectures have emerged, such as Multi-Layer Perceptrons (MLPs) [17], Convolutional Neural Networks (CNNs) [18] and Transformers [19].

2.1.2 Multi-Layer Perceptrons

Multi-Layer Perceptrons were originally invented in the 1980's as a generalization of Frank Rosenblatt's 1958 Perceptron [20] algorithm. As shown in Figure 2.1, they consist of a series of so-called *fully-connected layers* separated to one another by non-parametric nonlinearities (e.g., sigmoid function, rectified-linear unit).

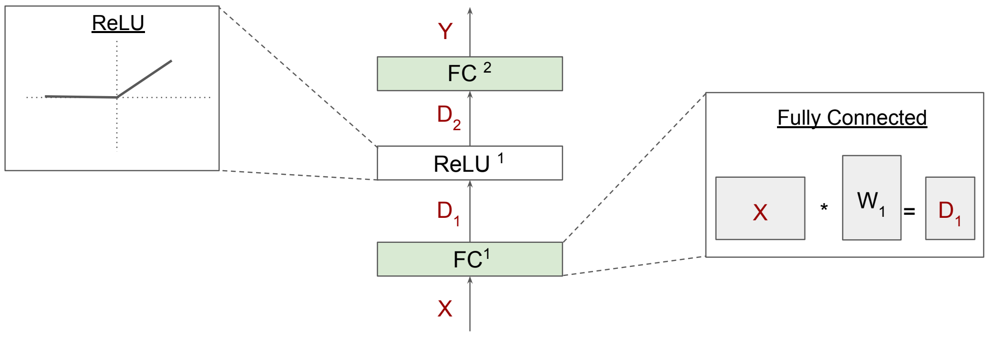


Figure 2.1: A multi-layer perceptron. Fully connected and non-parametric layers are shown in green and white respectively. For the sake of brevity, temporary results (red) and layer superscripts will be omitted in the remainder of this work.

Algebraically, fully-connected layers correspond to linear projections, and are therefore implemented as matrix multiplications with a weight matrix learned via backpropagation. In other words, given two weight matrices W_1 and W_2, the output of the neural network shown in Figure 2.1 can be computed as follows:

$$D_1 = X.W_1$$

$$D_2 = \max(D_1, 0)$$

$$Y = D_2.W_2$$

One of the main reasons behind the success of MLPs in the 1980s – and their resurgence in the 2010s – is their natural affinity with high-performance implementations of Basic Linear Algebra Subroutines (BLAS), such as GEneral Matrix Multiplication (GEMM). Nowadays, however, MLPs are generally only used when more novel architectures, such as those described below, are not applicable.

2.1.3 Convolutional Neural Networks

Convolutional Neural Networks were invented in 1989 [18] as a way to provide more parameter- and compute- efficient network architectures for computer vision tasks such as object classification, semantic segmentation and image upsampling. There, fully connected layers are replaced by *convolutional* layers that convolve their input with a set of learnable filters (or weights), as shown in Figure 2.2.

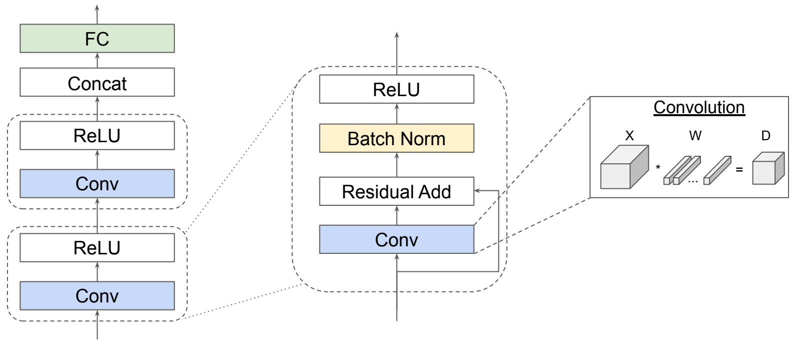


Figure 2.2: A standard convolutional neural network. Convolution and batch normalization layers are shown in blue and yellow respectively.

Specifically, convolutional layers are parameterized by a 4D tensor of weights containing $C \times K$ filters of $R \times S$ pixels each, where C and K respectively denote the number of channels in the input data and the number of filters in the layer. Given an input data tensor containing C images of $H \times W$ pixels, the output of the layer has K channels of $[P = (H - R + 1)] \times [Q = (W - S + 1)]$ pixels each, and is computed by convolving each image with each filter, and summing partial results across channels:

$$Y_k = \sum_{c=0}^{C} X_c \star W_{c,k}$$

In practice, this operation is usually repeated across N independent batches of input images:

$$\forall n \leq N \quad Y_{n,k} = \sum_{c=0}^{C} X_{n,c} \star W_{c,k}$$

The remarkable success of CNNs over the past several years was further reinforced by various enhancements made to their architecture, two very popular of which – residual connections and batch normalization – are described below.

Residual Connections

For decades, it was believed that very deep neural networks could not be trained to satisfying accuracy with backpropagation due to a phenomenon known as vanishing gradients [21]. In 2015, this issue was resolved through the addition of so-called *residual connections* after convolutional layers:

$$\text{ResidualAdd}(\text{Conv}(\text{X, W})) = \text{Conv}(X, W) + X$$

Though this breakthrough was discovered in the context of CNN classification on ImageNet, it has now found wider application across all other types of neural network architectures.

Batch Normalization

Deep Neural Networks trained using backpropagation are notoriously prone to *overfitting*, and may therefore poorly generalize when deployed in the real world. Throughout the years, various solutions to this problem have been proposed, the most popular of which is *batch-normalization.* There, the output of each residual connection $x = (x_1, \cdots, x_n)$ is normalized as follows:

$$\text{BatchNorm}(x) = \frac{x - E(x)}{\sqrt{\text{Var}(x)}}$$

The exact mechanism behind the regularizing properties of batch normalization remains largely unknown, but it has been proven to prevent gradient explosion in neural networks with residual connections initialized using standard methods such as Xavier [22] or Kaiming [23].

2.1.4 Transformers

The performance of DNNs in Natural Language Processing (NLP) tasks – for which convolutional layers have not been popular despite several interesting results [24, 25] – was largely improved by the emergence of Recurrent Neural Networks (RNNs) and, more recently, Transformer Networks. This dissertation will focus on the latter, which is inherently more parallelizable and therefore more suitable for modern many-core hardware.

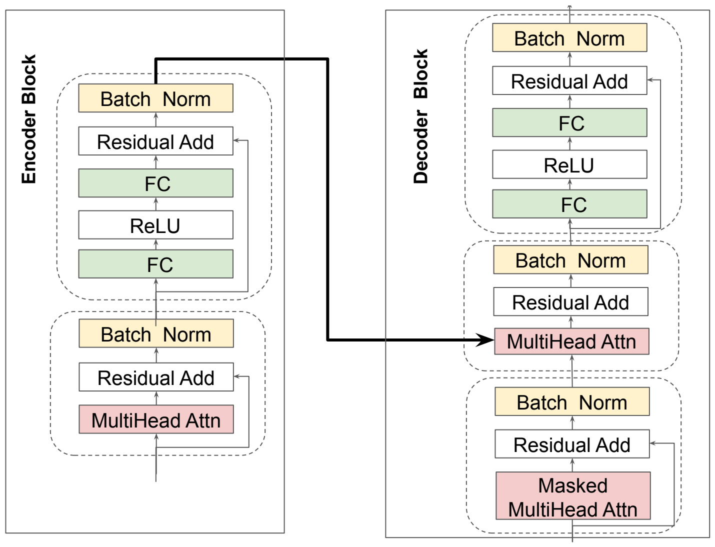


Figure 2.3: An encoder and decoder block in the transformer architecture. MultiHead Attention modules are shown in red.

Transformers are composed of a stack of *encoder blocks*, which encode a given sequence of *tokens* (e.g., words), and a stack of *decoder blocks*, which decode it. The structure of these basic building blocks, shown in Figure 2.3, relies on so-called *attention mechanisms* – i.e. functions that map a *query* and a *key-value* pair to an output representing how much each token should *attend* to other tokens in the same sequence. Typically, transformers use the *scaled dot-product* attention function:

$$\text{Attention}(Q, K, V) = \text{softmax}(QK^T/\sqrt{d_k})V$$

where $Q \in \mathbb{R}^{c_i \times d_k}$, $K \in \mathbb{R}^{c_o \times d_k}$ and $V \in \mathbb{R}^{c_o \times d_v}$ respectively denote a set of c_i d_k-

dimensional queries, c_o d_k-dimensional keys and c_o d_v-dimensional values.

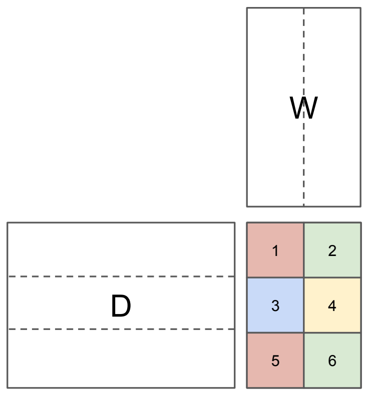
The above calculations are usually repeated for h different *heads* (i.e., embeddings) of the keys, values and queries – prior to projecting the concatenation of all these partial results onto yet another embedding for further processing by the neural network. In other words,

$$\text{MultiHead}(Q, K, V) = \text{Concat}(A_1, A_2, \cdots, A_h)W^O$$
$$\forall i \in [1, h], \quad A_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

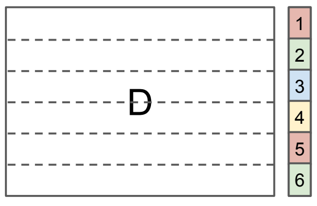
The resulting *attention layer* is followed by residual connections, batch normalization, fully-connected layers and rectified-linear units to form encoder and decoder blocks.

2.1.5 Opportunities for Parallelism

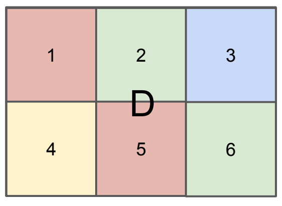
It should be clear from the above discussions that the majority of common neural network architectures can be efficiently implemented using just a small set of basic primitives for (1) matrix multiplications; (2) reductions; (3) elementwise operations; and (4) convolutions.



(a) matrix multiplication



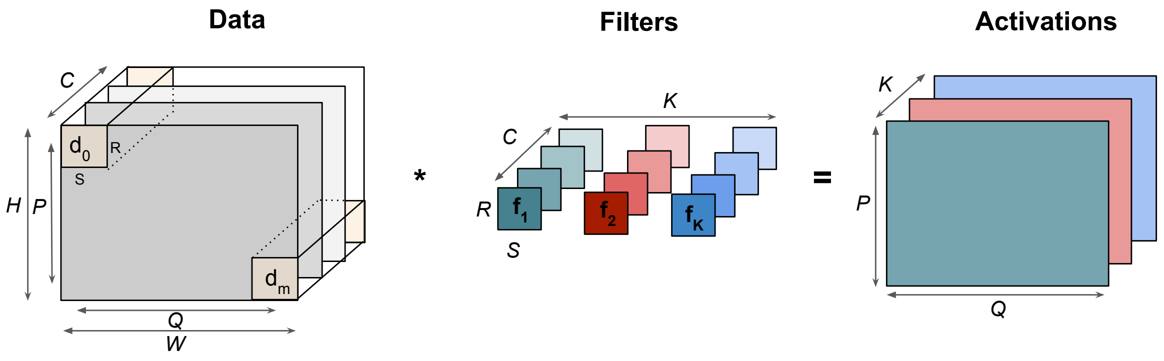
(b) reduction



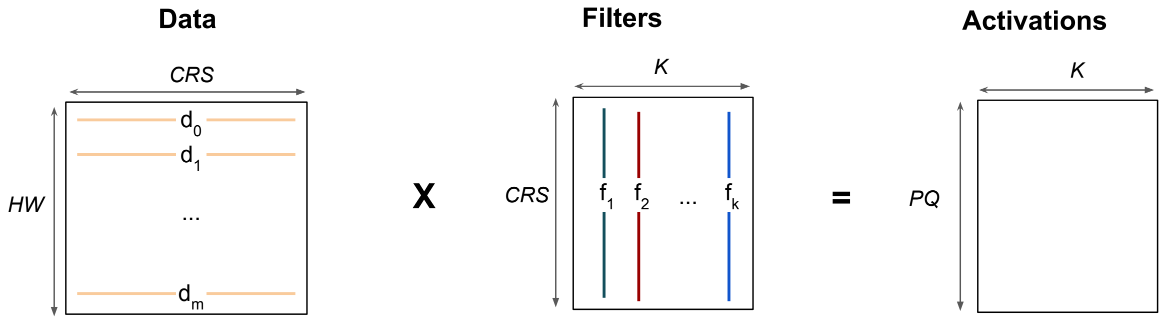
(c) element-wise operation

Figure 2.4: Parallelization strategies for (a) matrix multiplications, (b) reductions and (c) element-wise operations. Tasks which can be executed independently are numbered from 1-6, and different processors are represented by different colors

As shown in Figure 2.4, three of these primitives are embarassingly parallel: different blocks of their output result can be computed, independently from one another, on different processors. If the dimension of each block is carefully chosen (see Chapter 4) and the computation of each block properly scheduled (see Chapter 5), then this strategy can generally achieve close to peak performance on modern parallel hardware – even for structured-sparse computations (see Chapter 6). As it turns out, this straightforward parallelization strategy also works for convolutions, especially when they are represented as *implicit matrix multiplications*.



(a) Direct convolution



(b) Implicit matrix multiplication

Figure 2.5: Convolutional layers can be represented using (a) direct convolution or (b) implicit matrix multiplication.

Implicit matrix multiplication is a technique that emerged along with AlexNet [26] to speed up the evaluation of expensive convolutional layers using pre-optimized matrix multiplication subroutines (i.e., GEMM). There, the standard convolution operation is re-arranged by flattening each $C \times R \times S$ patch of input pixels into rows of an external data matrix – as shown in Figure 2.5. This operation, commonly known as *im2col*, is however itself expensive as it requires to replicate each pixel of the input image $R \times S$ times; it is therefore, nowadays, usually done in cache memory using complex blocked algorithms as described in Chapter 4.

2.2 Graphics Processing Units

2.2.1 General Microarchitecture

As mentioned above, GPUs have become one of the most popular hardware architectures available for prototyping and/or deploying novel research ideas in the field of DNNs. This is because their many-cores architecture is particularly suitable for the massively parallel computations outlined above.

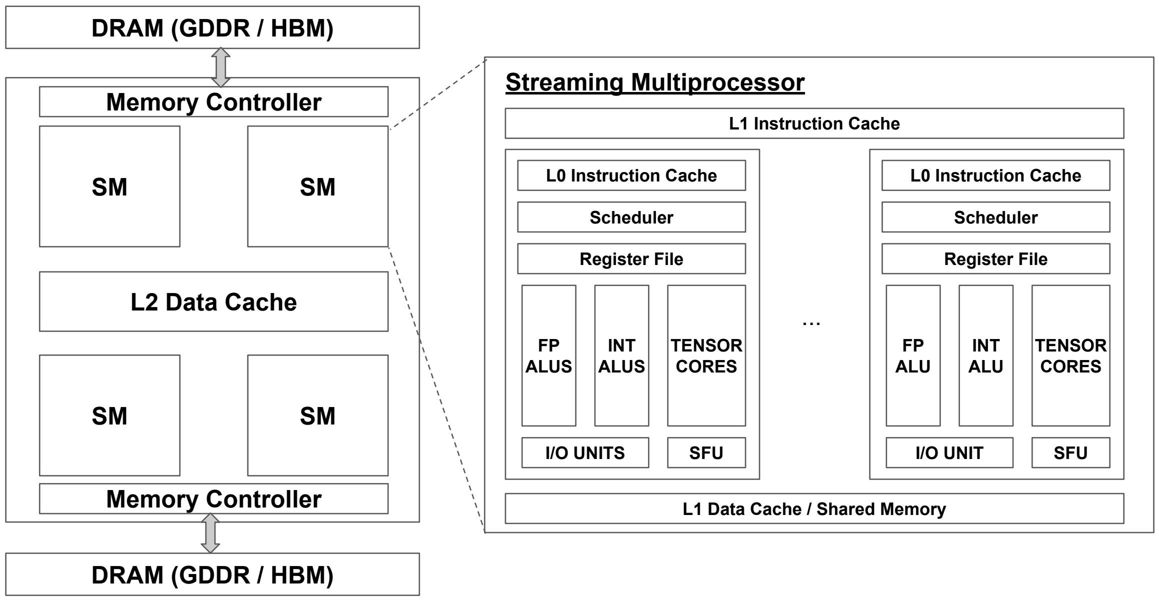


Figure 2.6: Typical architecture of a modern NVIDIA GPU.

As shown in Figure 2.6, modern GPUs are typically composed of a set of *Streaming Multiprocessor* (SM) – or cores – which share a common L2 Data Cache as well as multiple DRAM memory controllers. The number of SMs (resp. memory channels) available may vary from architecture to architecture, but it is generally on the order of 64 (resp.

16). Importantly, good choices of block dimensions for the parallelization strategies discussed above (Section 2.1.5) directly depend on these parameters, as bigger GPUs can only be fully utilized if more tasks are ran in parallel. Note that utilizing all of a GPU's SMs may not be necessary for I/O-bound workloads that already saturate all the memory channels.

Each SM includes a number (typically 1-4) of scheduling units capable of concurrently dispatching instructions onto independent groups of resources (FP ALUs, INT ALUs, Tensor Cores, SFU, I/O Unit), each with their own L0 instruction cache and register file. To reduce the latency of instruction fetching, each SM also includes an L1 instruction cache and, to reduce that of data fetching, an L1 data-cache that may be configured by users as addressable *local memory* (also known as *shared memory*).

In recent years, GPUs have rapidly specialized for DNNs, as evidenced by the recent incorporation of tensor cores – specialized arithmetic units for tensor computations – in streaming multi-processors. Tensor cores provide hardware-accelerated matrix multiplication accumulation (MMA) instructions capable of much higher throughput than standards fused-multiply-adds (FMA) (see Table 2.1), but are also much harder to program efficiently with existing tools and languages.

GPU Name	Architecture	FMA (TFLOPS)	MMA (TFLOPS)
Tesla A100	Ampere	19.5	312
Titan RTX	Turing	16.3	130
Tesla V100	Volta	15.7	120

Table 2.1: Theoretical peak performance of various modern NVIDIA GPUs

2.2.2 Programming Languages

Since GPUs were originally designed for computer graphics, they were for a long time only programmable using shading languages such as GLSL (from OpenGL) or HLSL (from DirectX). However, the release of CUDA in 2007 dramatically facilitated the development of General Purpose GPU (GPGPU) software, and nowadays DNN computations are almost always accelerated using this toolkit. Note that, however, shading languages still remain the most common way to achieve portable hardware-acceleration for DNNs on mobile GPUs [27].

Compute Unified Device Architecture (CUDA)

CUDA is a parallel computing platform that was released by NVIDIA in 2007, and which is now routinely used for DNN training and inference.

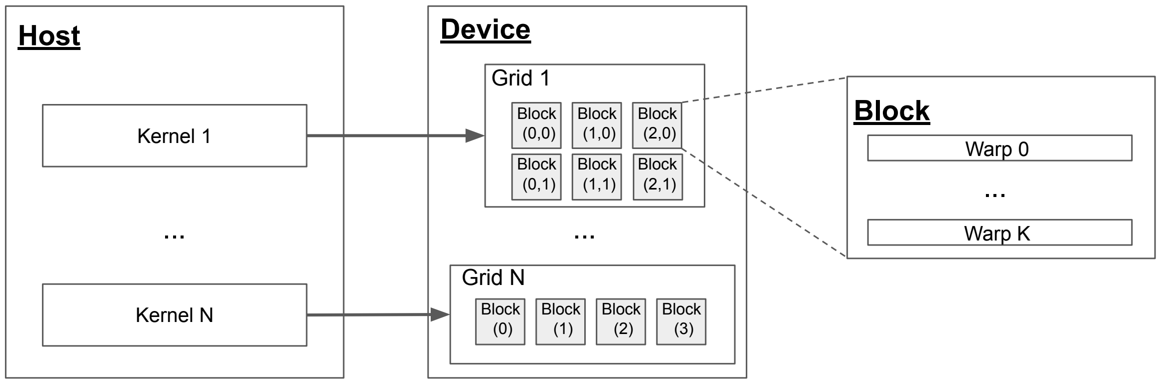


Figure 2.7: The CUDA execution model

The CUDA execution model, shown in Figure 2.7, is based on a standard Single Program, Multiple Data (SPMD) paradigm wherein the same *kernel* (i.e., program) is replicated multiple times on a one-to-three dimensional *grid* of instances (or *block*). Each instance has a unique identifier (*block id*) that encodes its position in the grid, and which can be used to access different portions of the input/output data as needed. All of these instances execute with a fixed amount of threads, arranged in groups of 32 called *warps*.

```
__global__ void relu(float *x, float *y, int N){
  int id = blockIdx.x * blockDim.x + threadIdx.x;
  if(id < N)
    y[id] = max(x[id], 0);
}
```

Listing 2.1: ReLU in CUDA-C

Each kernel is generally programmed using a C-based language known as CUDA-C (sometimes simply abbreviated as CUDA for simplicity). There, individual threads are handled separately and their ID, as well as that of their parent block, can be queried using threadIdx.{x,y,z} and blockIdx.{x,y,z} respectively. Listing 2.1 shows how easy it is to use this language to implement element-wise operations such as ReLU. Efficiently implementing other DNN primitives such as matrix-multiplication and convolutions in CUDA is however much more challenging. The main purpose of this dissertation is to show how most of these challenges (Section 2.2.3) disappear when kernels are blocked but single-threaded.

GPUs designed by Advanced Micro Devices (AMD), though not considered in this work, are similar. They are generally programmed using OpenCL instead of CUDA, and have the same high-level architecture except for two major differences: tensor cores are absent and threads are organized in groups of 64 called *wavefronts* rather than 32-wide warps.

2.2.3 Optimization Challenges

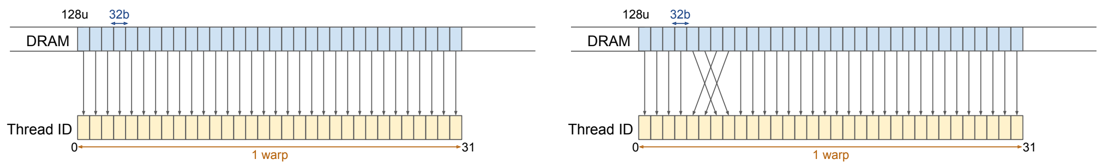
Occupancy

As mentioned above, GPUs are best utilized when the resource shown in Figure 2.6 (memory controllers, ALUs, tensor cores) are as busy as possible. This means that the amount of work done by each kernel must be adjusted adequately so that the number of instances launched is neither too high nor too low. This is a complicated problem, as the right amount of work to do depends not only on known architecture details (e.g., number of SMs) but also on information only available at runtime such as, for example, the size of the x and y arrays in Listing 2.1.

Some analytical models [28] have been developed to select good block sizes at runtime, but these rarely account perfectly for cache effects. Therefore, in Chapter 4, we will show how all this information can be learned automatically using *context-aware auto-tuning*.

Memory Efficiency

Memory operations on modern GPUs (i.e., DRAM load/store) are typically several orders of magnitude slower than FMAs and Tensor Cores. This *memory wall* is a well-known problem in high-performance computing, hence various techniques have been proposed over the years to mitigate its impact.



(a) coalesced memory accesses **(b)** uncoalesced memory accesses

Figure 2.8: Memory Coalescing

First, GPU memory controllers load data in *burst mode* to increase effective bandwidth. This can be leveraged in parallel languages by coalescing memory accesses – i.e., accessing consecutive memory locations from consecutive threads (see Figure 2.8). Second, modern GPU cores come with a large amount of L2 (∼10MB) and addressable L1 (∼128kB) caches which can be used to reduce the latency of memory accesses on data that has already been seen before. Last but not least, the latency of memory accesses can be *hidden* – i.e., overlapped with expensive computations – through the use of *instruction-level parallelism* and *data pre-fetching*.

Bank Conflicts

Managing the aforementioned addressable L1 cache is difficult. It is divided in many (generally around 32) independent banks that must all be used at the same time to minimize access latency. In other words, memory accesses from independent threads into the same bank are serialized, leading to significant performance loss. To alleviate this issue, developers are expected to *explicitly* arrange data in shared memory so as to minimize bank conflicts.

2.3 Compiler Construction

2.3.1 General Overview

As shown in Figure 2.9, modern compilers are typically composed of two sub-systems – a *frontend* and a *backend* – connected to one another by an *Intermediate Representation* (IR). Compiler frontends and backends play different roles, and they are therefore generally implemented in completely separate software packages, such as Clang and LLVM for example.

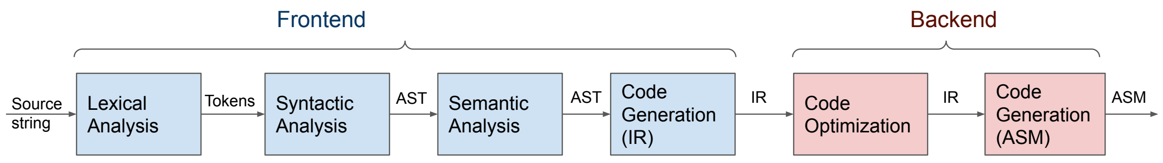


Figure 2.9: High-level architecture of a modern compiler.

The role of the compiler frontend is to transform a piece of source code written in a pre-defined input language (e.g., CUDA-C) into an IR (e.g., LLVM-IR) understandable by the target backend. This is usually done in four stages:

- **Lexical analysis** decomposes a given source string into a sequence of tokens – keywords, operators, identifiers, etc. – defined by the lexical grammar of the source language. This process is typically done using finite-state machines, and can be automated through the use of tools like *lex/flex*.
- **Syntactic analysis** analyzes this sequence of tokens to produce an *Abstract Syntax Tree* (AST) encoding the syntactical grammar of the source language. For the languages typically used in GPU programming, this can be done using recursive descent parsers, possibly via tools like *yacc/bison*.

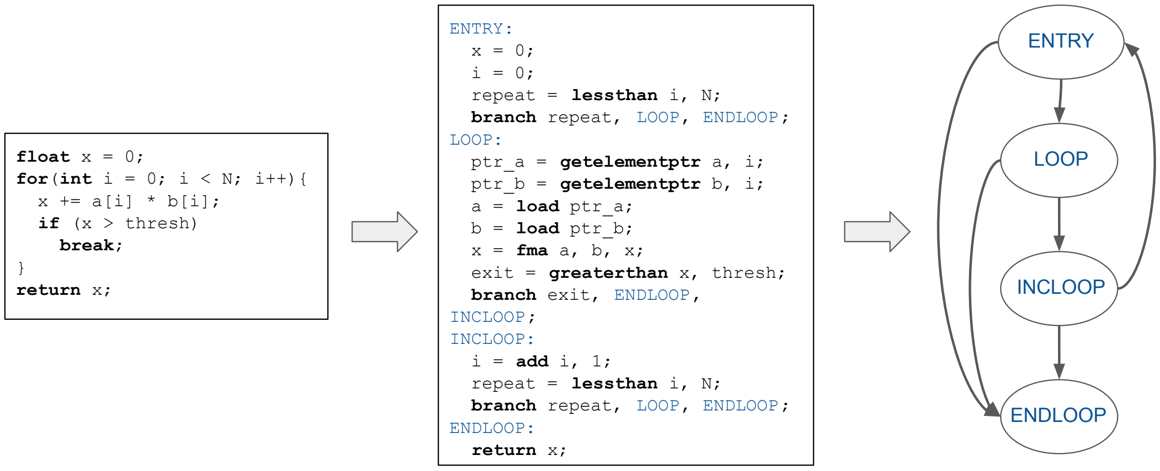
- **Semantic analysis** checks the above AST for semantic issues pertaining to type checking, type inference, variable declaration and – in the case of array operations – shape compatibility.
- **Intermediate code generation** finally transforms the semantically analyzed AST into an IR that can be optimized and translated into assembly by the target compiler backend.

The compiler backend then generates assembly code from the above IR in two major steps:

- **Code optimization** optimizes the IR produced by the compiler frontend. This phase is typically separated into *machine-independent optimizations* that reduce the arithmetic/memory complexity of the target program, and *machine-dependent optimizations* that aim to make best use of the target architecture.
- **Assembly code generation** transforms the optimized IR into assembly code that can be readily executed by the target architecture.

2.3.2 Data-Flow Analysis

The code optimization passes performed by compiler backends often require the knowledge of important information about the state of each variable at every possible program point. The process of computing this information, also known as *data-flow analysis*, can generally be done statically using the Control Flow Graph (CFG [29]) of the input program.



```
float x = 0;
for(int i = 0; i < N; i++){
  x += a[i] * b[i];
  if (x > thresh)
    break;
}
return x;
```

```
ENTRY:
  x = 0;
  i = 0;
  repeat = lessthan i, N;
  branch repeat, LOOP, ENDLOOP;
LOOP:
  ptr_a = getelementptr a, i;
  ptr_b = getelementptr b, i;
  a = load ptr_a;
  b = load ptr_b;
  x = fma a, b, x;
  exit = greaterthan x, thresh;
  branch exit, ENDLOOP,
INCLOOP;
INCLOOP:
  i = add i, 1;
  repeat = lessthan i, N;
  branch repeat, LOOP, ENDLOOP;
ENDLOOP:
  return x;
```

Figure 2.10: Source code, intermediate representation and control flow graph for the operation $x = \boldsymbol{a}^T\boldsymbol{b}$

In essence, control flow graphs are graphical representations of all the possible paths that might be taken by a given program throughout its execution. They take the form of a directed acyclic graph whose nodes are basic blocks, (i.e., straight-line code sequences that may only contain so-called *terminator* instructions at their end), and whose edges represent jumps in control flow. As shown in Figure 2.10, they can be usually computed rather easily by looking at the terminator of each basic block.

Algorithm 2: Worklist iterative data-flow analysis.

```
Input: control-flow graph $\mathcal{G} = (V, E)$
for $v \in V$ do
    $\mathrm{IN}(v) \leftarrow \emptyset$;
    $\mathrm{OUT}(v) \leftarrow \mathrm{GEN}(v)$;
do
    $v \leftarrow \mathrm{remove}(worklist)$;
    $last \leftarrow \mathrm{OUT}(v)$;
    $\mathrm{IN}(v) \leftarrow \bigcup_{p \in \mathrm{PRED}(v)} \mathrm{OUT}(p)$;
    $\mathrm{OUT}(v) \leftarrow \mathrm{GEN}(v) \cup (\mathrm{IN}(v) - \mathrm{KILL}(v))$;
    if $last \neq OUT(v)$ then
        $worklist \leftarrow worklist \cup \mathrm{SUCC}(v)$
while $worklist \neq \emptyset$;
```

Data-flow analysis algorithms assign, to each node in the CFG of the target program, a *transfer function* that determine how a given basic block may modify the state of its input variables. Each transfer function is then applied – in forward or backward order – repeatedly until the state of each variable at every program point reaches a fixpoint. It is very common for transfer functions to modify the state $\mathrm{IN}(v)$ of each basic block into a new state $\mathrm{OUT}(v) = \mathrm{GEN}(v) \cup (\mathrm{IN}(v) - \mathrm{KILL}(v))$, where GEN and KILL respectively denote policies for ”generating” new state attributes in a basic block or ”killing” existing ones. When this is the case, data-flow information can be computed using the worklist iterative algorithm shown in Algorithm 2. This procedure is guaranteed to terminate in polynomial time [30, 31].

3
Related Work

In this chapter, we briefly review the two most popular approaches for automatic neural network code generation on GPUs: polyhedral compilation and scheduling languages. The limitations of each method will be highlighted so as to motivate the need for the systems presented from Chapter 4 onward.

3.1 Polyhedral Compilation

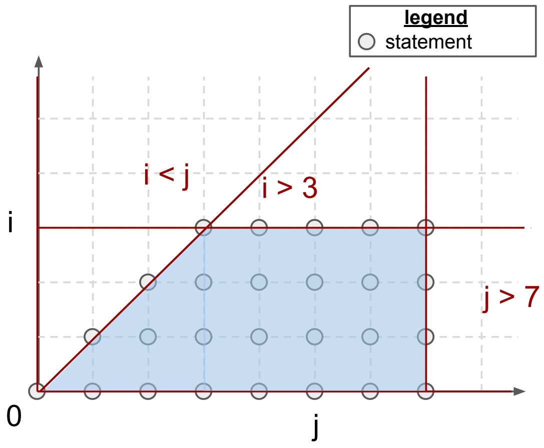
Traditional compilers typically rely on intermediate representations, such as LLVM-IR [32], that encode control flow information using (un)conditional branches. This relatively low-level format makes it difficult to statically analyze the runtime behavior (e.g., cache misses) of input programs, and to automatically optimize loops accordingly through the use of tiling [33], fusion [34] and interchange [35]. To solve this issue, polyhedral compilers [36] rely on program representations (Section 3.1.1) that have statically predictable control flow, thereby enabling aggressive compile-time program transformations (Section 3.1.2) for data locality and parallelism. Though this strategy has been adopted by many languages and compilers for DNNs such as Tiramisu [6], Tensor Comprehensions [7], Diesel [37] and the Affine dialect in MLIR [38], it also comes with a number of limitations that will be described in Section 3.1.3.

3.1.1 Program Representation

Polyhedral compilation is a vast area of research. In this section we only outline the most important aspects of this topic, but readers interested in the solid mathematical foundations underneath may refer to the work of Schrijver [39] for more information.

```
for(int i = 0; i < 3; i++)
  for(int j = i; j < 5; j++)
    A[i][j] = 0;
```

(a) source code



(b) iteration domain

Figure 3.1: Source code (a) and iteration domain (b) of an example program suitable for polyhedral optimization

Polyhedral compilers focus on a class of programs commonly known as *Static Control Parts* (SCoP), *i.e.,* maximal sets of consecutive statements in which conditionals and loop bounds are affine functions of surrounding loop indices and global invariant parameters. As shown in Figure 3.1, programs in this format (Figure 3.1(a)) always lead to iteration domains (Figure 3.1(b)) that are bounded by affine inequalities, i.e., polyhedral. These polyhedra can also be defined algebraically; for the above example:

$$\mathcal{P} = \{i, j \in \mathbb{Z}^2 \mid \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 2 \\ 0 \\ 4 \end{pmatrix} \geq 0\}$$

Each point (i, j) in $\mathcal{P}$ represents a *polyhedral statement*, that is a program statement which (1) does not induce control-flow side effects (e.g., `for`, `if`, `break`) and (2) contains only affine functions of loop indices and global parameters in array accesses. To facilitate alias analysis, array accesses are also mathematically abstracted, using so-called *access function*. In other words, `A[i][j]` is simply `A[f(i,j)]` where the access function f is defined by:

$$f(i, j) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} = (i, j)$$

Note that the iteration domains of am SCoP does not specify the order in which its statements shall execute. In fact, this iteration domain may be traversed in many different possible legal orders, i.e. *schedules*. Formally, a schedule is defined as a p-dimensional affine transformation Θ of loop indices $\boldsymbol{x}$ and global invariant parameters $\boldsymbol{g}$:

$$\Theta_S(\boldsymbol{x}) = T_S \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{g} \\ 1 \end{pmatrix} \qquad T_S \in \mathbb{Z}^{p \times (\dim(\boldsymbol{x}) + \dim(\boldsymbol{g}) + 1)}$$

Where $\Theta_S(\boldsymbol{x})$ is a p-dimensional vector representing the slowest to fastest growing indices (from left to right) when traversing the loop nest surrounding S. For the code shown above, the original schedule defined by the loop nest in C can be retrieved by using:

$$\Theta_S(\boldsymbol{x}) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i & j \end{pmatrix}^T = \begin{pmatrix} i & j \end{pmatrix}^T$$

where i and j are respectively the slowest and fastest growing loop indices in the nest. If T_S is a vector (resp. matrix), then Θ_S is a said to be one-dimensional (resp. multi-dimensional).

3.1.2 Program Transformations

Most of the program transformations done within the polyhedral framework actually boil down to the production of schedules and iteration domains that enable loop transformations promoting parallelism and spatial/temporal data locality (see Table 3.1).

name	description
fusion	combines multiple perfectly nested loops into a single one
fission	splits a single loop into multiple perfectly nested ones
interchange	permute two perfectly nested loops
tiling	reorganize a loop to process data in blocks for better locality
parallelization	execute iterations of a loop in parallel on multiple processors

Table 3.1: non-exhaustive list of common loop transformations

Polyhedral compilers go through complex verification processes to ensure that the semantics of their input program is preserved throughout this optimization phase. The iteration domain of each statement may however be altered into another polyhedron without causing any issue. Note that polyhedral optimizers are not incompatible with more standard optimization techniques. In fact, it is not uncommon for these systems to be implemented as a set of LLVM passes that can be run independently of more traditional compilation techniques [40].

Example

As an example, let us consider the matrix multiplication program shown in Listing 3.1. As we saw in Chapter 2, this operation is an important basic primitive for neural network computations; it is also highly regular, which makes it a good candidate for polyhedral optimization.

```
for(int i = 0; i < M; i++)
for(int j = 0; j < N; j++){
  C[i][j] = 0; // Statement R
   for(int k = 0; k < K; k++)
     C[i][j] += A[i][k] * B[k][j]; // Statement S
}
```

Listing 3.1: Matrix Multiplication

The original schedule of the statement S in this program can be written as the following (identity) affine transformation:

$$\Theta_S(\boldsymbol{x}) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} i & j & k \end{pmatrix}^T = \begin{pmatrix} i & j & k \end{pmatrix}^T$$

Loop interchange can then simply be implemented by swapping j and i in the original schedule, leading to the new schedule:

$$\Theta_S(\boldsymbol{x}) = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} i & j & k \end{pmatrix}^T = \begin{pmatrix} j & i & k \end{pmatrix}^T$$

Other loop transformations can be implemented similarly provided appropriate iteration domain transformations. A detailed review of the theory behind common loop transformations and their legality within the polyhedral framework can be found in the doctoral dissertation of U. Bondhugula [41].

3.1.3 Limitations

Polyhedral machinery is extremely powerful, when applicable. It has been shown to support most common loop transformations, and has indeed achieved performance comparable to state-of-the-art GPU libraries for dense matrix multiplication [37]. Additionally, it is also fully automatic and doesn't require any hint from programmers apart from source-code in a C-like format. Unfortunately, it also suffers from two major limitations that have prevented its adoption as a universal method for code generation in neural networks.

First, the set of possible program transformations $\Omega = \{\Theta_S \mid S \in \text{program}\}$ is large, and grows with the number of statements in the program as well as with the size of their iteration domain. Verifying the legality of each transformation can also require the resolution of complex integer linear programs, making polyhedral compilation very computationally expensive. To make matters worse, hardware properties (e.g., cache size, number of SMs) and contextual characteristics (e.g., input tensor shapes) also have to be taken into account by this framework, leading to expensive auto-tuning procedures [42].

Second, the polyhedral framework is not very generally applicable; SCoPs are relatively common [43] but require loop bounds and array subscripts to be affine functions of loop indices, which typically only occurs in regular, dense computations. For this reason, this framework still has to be successfully applied to sparse – or even structured-sparse – neural networks, whose importance has been rapidly rising over the past few years.

On the other hand, blocked program representations advocated by this dissertation are less restricted in scope (Chapter 6) and can achieve close to peak performance using standard dataflow analysis (Chapter 5).

3.2 Scheduling Languages

Separation of concerns [44] is a well-known design principle in computer science: programs should be decomposed into modular layers of abstraction that separate the semantics of their algorithms from the details of their implementation. Systems like Halide [8] and TVM [9] push this philosophy one step further, and enforce this separation at the grammatical level through the use of a *scheduling language*. The benefits of this methodology are particularly visible in the case of matrix multiplication, where, as one can see in Listing 3.2, the definition of the algorithm (Line 1-7) is completely disjoint from its implementation (Line 8-16), meaning that both can be maintained, optimized and distributed independently. The resulting code may however not be completely portable, as schedules can sometimes rely on execution models (e.g., SPMD) or hardware intrinsics (e.g., matrix-multiply-accumulate) that are not widely available. Note that polyhedral compilers (Section 3.1.2) also promote separation of concerns, though differently, by allowing programmers to achieve better performance with simpler source code.

```
1  // algorithm
2  Var x("x"), y("y");
3  Func matmul("matmul");
4  RDom k(0, matrix_size);
5  RVar ki;
6  matmul(x, y) = 0.0f;
7  matmul(x, y) += A(k, y) * B(x, k);
8  // schedule
9  Var xi("xi"), xo("xo"), yo("yo"), yi("yo"), yii("yii"), xii("xii");
10 matmul.vectorize(x, 8);
11 matmul.update(0)
12       .split(x, x, xi, block_size).split(xi, xi, xii, 8)
13       .split(y, y, yi, block_size).split(yi, yi, yii, 4)
14       .split(k, k, ki, block_size)
15       .reorder(xii, yii, xi, ki, yi, k, x, y)
16       .parallel(y).vectorize(xii).unroll(xi).unroll(yii);
```

Listing 3.2: Matrix multiplication in Halide.

Scheduling languages have gained in popularity over the past few years, and have even been used to accelerate mobile implementations of common convolutional neural networks [45]. In this section, we will describe their syntax and semantics (Section 3.2.1), their compilation mechanisms (Section 3.2.2) and their limitations (Section 3.2.3). Note that we will offer just a rudimentary overview of these aspects; for more details, readers may refer to the doctoral dissertation of Jonathan Ragan-Kelley [46].

3.2.1 Schedule Specification

Writing valid schedules for Halide/TVM is far from trivial. It requires the consideration of complex trade-offs between data locality, parallelism and redundant computations. To make the exploration of this complex design space easier and more tractable, scheduling languages generally provide a set of primitives that define optimization strategies on the loop nest induced by their algorithm's iteration domain.

name	description
parallel(x)	traverse the loop x in parallel
unroll(x)	unroll the loop x
vectorize(x)	vectorize the loop x
reorder(y, x)	flip the traversal order of x and y
split(x, o, i, α)	split loop x into an inner (i) and outer (o) loop s.t. $x = o * \alpha + i$
fuse(x, y)	fuse loops x and y

Table 3.2: (non-exhaustive) list of primitives in the Halide scheduling language

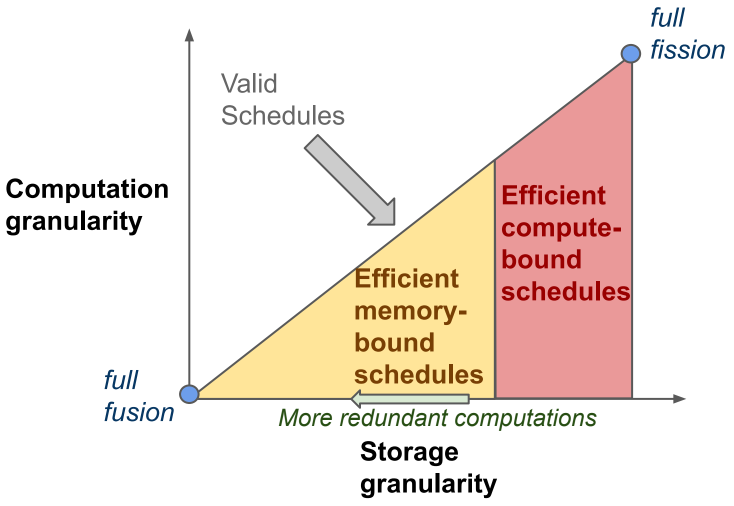


Figure 3.2: Two-dimensional representation of schedule spaces in Halide/TVM.

As a whole, these primitives control four different attributes of the algorithm's implementation: (1) The **iteration order** of its loop nests, which is similar to *polyhedral schedules* mentioned in Section 3.1.1; (2) The **storage order**, or internal layout, of its input, output and temporary buffers; (3) The **storage granularity** of intermediate temporary result; And (4) The **computation granularity** of blocks of function values.

Figure 3.2 shows a two-dimensional representation of possible schedules as a function of their storage granularity and compute granularity. By default, most scheduling languages forbid computations to occur at a larger granularity than storage, hence the space of valid schedules is delimited by a line. Different points in this space correspond to different trade-offs between data locality, parallelism and redundant computations. For example, using lower granularity for temporary storage (e.g., loop fusion) means that values will have to be recomputed whenever they are needed after having been evicted of the specified scratch space. On the other hand, performing coarse-grained computations into coarse-grained temporary buffers (e.g., loop fission) promotes parallelism at the expense of memory locality *between functions*. Note that compute-bound (resp. memory-bound) operations should avoid redundant computations (e.g., memory accesses) as much as possible, hence arithmetically intense computations such as dense matrix multiplications generally use larger storage granularity.

3.2.2 Compiling Scheduled Algorithms

Once all the functions in a given algorithm have been explicitly scheduled using the above primitives, the resulting pipeline is compiled into a sequence of loop nests – and possibly GPU kernels – that follows the optimization strategies specified by the user. Contrary to traditional compilers, this process does not require the use of any heuristic, as all the information needed for program transformations is already specified by the given schedule.

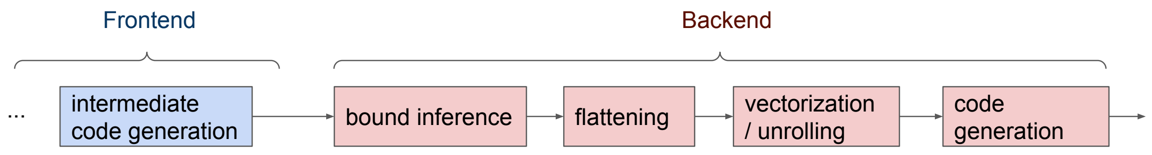


Figure 3.3: Typical compilation pipeline of a scheduling language.

A non-exhaustive list of optimizations done by Halide/TVM is shown in Figure 3.3.

Sliding window optimizations are voluntarily left out, since they are more applicable to standard image processing pipelines than neural network computations.

Intermediate Code Generation

The compilation process starts with an *intermediate code generation* phase, which transforms a given algorithm into a set of loop nests whose order, stride, minimum value and extent are specified by the provided schedule. These loops are represented using an intermediate representation that includes qualifiers for parallel, vectorized or unrolled loops, as well as basic allocation primitives for temporary storage and output buffers. Note that at this stage, many important quantities (loop bounds, allocation size) may still be known only symbolically and lack a proper explicit definition.

Bound Inference

The bound inference phase aims to assign specific values to all the symbolic expressions present in the above intermediate representation. This is done recursively back from the algorithm's output, by analyzing the intervals in which every input/temporary buffers are accessed. Note that the resulting loops can only describe iterations over axis-aligned boxes. We will see in Section 3.2.3 how this may be a problem for neural network computations.

Flattening

The intermediate representations used by scheduling languages generally allow for multi-dimensional array accesses and allocations. In order to ensure compatibility with low-level backends like C or LLVM-IR that rely on pointer dereferences, the flattening pass "flattens" these multi-dimensional operations into equivalent one-dimensional load/store/allocations, using memory strides specified in the provided schedule. Note that the memory stride of the innermost dimension must always be 1, so as to ensure memory alignment properties compatible with vectorized memory operations.

Vectorization/Unrolling

Once symbolic expressions have been resolved and memory accesses have been flattened, the body of each unrolled loop is replicated as many times as necessary, and every scalar instruction is vectorized as needed. Scheduling languages generally have no divergent control flow, so these transformations are always well-defined.

Automatic Parallelization

On CPUs, automatic parallelization is achieved by scanning the program for parallel loops, and compiling their body into separate functions that may take as argument any state variable necessary (e.g., buffer pointers, loop bounds, etc.). These functions are then enqueued into a thread pool managed by a task queue.

Schedules written for CPUs are not automatically portable to GPUs; parallel loops need to be annotated with the *block* and *thead* dimensions to which they correspond. Perfect nests of parallel loops are then mapped to GPU kernels, whose launch characteristics (e.g., number of threads, number of blocks) are usually determined through automatic performance tuning.

Backend Code Generation

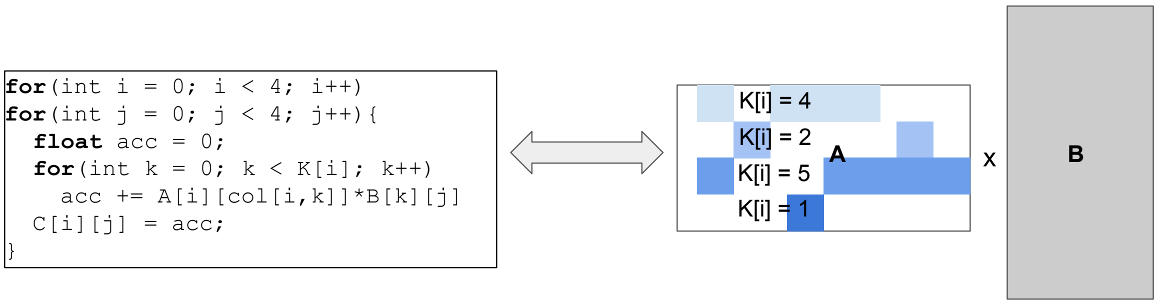
At this point, the backend code (e.g., C, CUDA, LLVM-IR) for the scheduled intermediate representation can finally be generated using simple pattern-matching instruction selection algorithms.

3.2.3 Limitations

Scheduling languages are, without a doubt, one of the most popular approaches for neural network code generation. Nonetheless, they suffer from a number of fundamental limitations that reduces their applicability to sparse computations, as well as their performance and portability to emerging hardware architecture.

Axis-Aligned Iteration Spaces

As mentioned above, existing scheduling languages generate loops whose bounds and increments cannot depend on surrounding loop indices. This is problematic for sparse computations, whose iteration spaces may be irregular.



```
for(int i = 0; i < 4; i++)
for(int j = 0; j < 4; j++){
  float acc = 0;
  for(int k = 0; k < K[i]; k++)
    acc += A[i][col[i,k]]*B[k][j]
  C[i][j] = acc;
}
```

Figure 3.4: Sparse matrix multiplications define irregular iteration spaces that cannot be naturally handled by existing scheduling languages.

As shown in figure 3.4, sparse matrix multiplication tasks tend to have inner loops whose bounds depend on outer loop indices, making them a poor fit for most scheduling languages. As it turns out, this operation has also been gaining in importance over the past few years, with the advent of neural network pruning [47] and sparse attention mechanisms [48, 49]. The block-based program representation that we advocate for in this dissertation does not have such limitations, and can be used to develop efficient sparse language models as shown in Chapter 6.

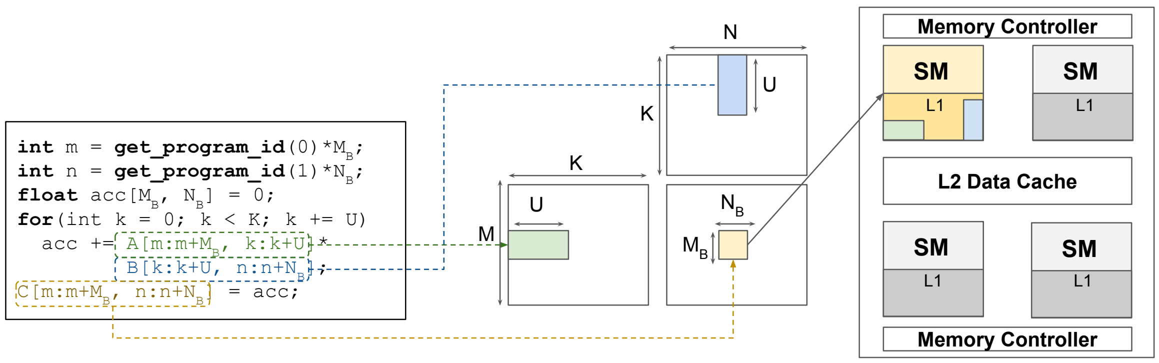
Tensor Intrinsics

Since the seminal release of Halide in 2013, graphics processor have been rapidly evolving and specializing for neural network computations. Scheduling languages have tackled this problem through the introduction of intrinsic instructions for tensor computations [50], which (as of August 2020) unfortunately lead to schedules that are difficult to write and also lack portability. In fact, even the best schedules available for mixed-precision convolutions remain much slower than good handwritten implementation at equal block sizes. On the other hand, the blocked program representation presented in this work enables block-aware instruction selection which guarantees automatic tensor core utilization – and achieves performance within 15% of the best handwritten code available (see Section 5.4.3).

4 Context-Aware Auto-Tuning

To overcome the limitations of existing DSLs for DNNs, this dissertation proposes a block-based programming paradigm in which programmers are expected to write code in terms of operations on block variables. This can be harder than it seems, as it requires the use of appropriate block shapes that efficiently balance the trade-off between three major external factors. First, **hardware characteristics** – such as data/instruction cache size, memory bandwidth, register file size and number of cores available – can make or break performance for certain block shapes. For example, blocked implementations of matrix multiplication only work well when sub-matrices used in arithmetically intense operations (e.g., in Figure 4.1, $A[m : m + M_B, k : k + U|$ and $B[k : k + U, n : n + N_B])$ can entirely fit in cache memory. Second, **compiler characteristics** – such as heuristic algorithms used for register allocation [51, 52] or instruction scheduling – can create artifacts that reduce the performance of otherwise efficient blocked algorithms. For example, setting

$M_B = N_B = 128$ in Figure 4.1 may cause register spilling when implemented in CUDA-C, leading to significantly worse performance than the equivalent handwritten assembly implementation found in cuBLAS. Finally, **contextual characteristics** – such as the shapes or sparsity pattern of input tensors – can limit opportunities for data parallelism when blocks are large. For example, blocked matrix multiplication tasks shown in Figure 4.1 typically require $\frac{M}{M_B} * \frac{N}{N_B} \geq N_{\text{SM}}$ to fully occupy a GPU with N_{SM} streaming multi-processors. This is, in fact, true of any runtime parameter (i.e., *execution context*) that may affect the amount of work done.



```
int m = get_program_id(0)*M_B;
int n = get_program_id(1)*N_B;
float acc[M_B, N_B] = 0;
for(int k = 0; k < K; k += U)
  acc += A[m:m+M_B, k:k+U]*
         B[k:k+U, n:n+N_B];
C[m:m+M_B, n:n+N_B] = acc;
```

Figure 4.1: Mapping of a matrix multiplication onto parallel hardware in our blocked SPMD paradigm. Large block shapes limit parallelism, while small block shapes limit data-reuse in L1 data caches.

In order to find acceptable trade-offs between all these factors, state-of-the-art implementations of blocked algorithms generally rely on *automatic performance tuning (auto-tuning)* techniques [15, 53]. There, the space of possible implementations is exhaustively (or stochastically) searched for programs achieving high empirical performance, the fastest of which is retained for subsequent use. Existing auto-tuning methods are typically *context-agnostic*, in that this search has to be repeated not only every time the underlying platform (hardware, compiler) changes but also whenever important runtime parameters (e.g., input shapes, sparsity pattern) do. This can lead to applications that spend more time auto-tuning algorithms than executing them productively.

By contrast, in this chapter, we propose a *context-aware* auto-tuning method capable of dynamically predicting block shapes that are preferable *at runtime*, given any contextual characteristics that may influence the amount of work done. Contrary to hand-written heuristics and approximate cost models [54], our method – which relies on machine learning – requires no expert knowledge about the algorithm of interest or the target hardware architecture. As shown in Figure 4.2, our system is composed of four major components. First, a code generator transforms a given set of block shapes $\boldsymbol{x}_B$ into a compute kernel $k(\boldsymbol{x}_B)$. The generated code is then repeatedly benchmarked in different contexts $\boldsymbol{x}_C$ so as to create a dataset of empirical performance measurements $\mathcal{D} = \{(\boldsymbol{x}_B, \boldsymbol{x}_C, y) \mid y = perf(k(\boldsymbol{x}_B), \boldsymbol{x}_C)\}$. Third, this dataset is used to build a surrogate regression model $\hat{y} = f(\boldsymbol{x_B}, \boldsymbol{x_C})$ for the performance of $k(\boldsymbol{x}_B)$ in any given $\boldsymbol{x}_C$. Finally, when a new execution context $\hat{\boldsymbol{x}}_C$ is seen at runtime, the block shapes that perform best on our performance model are returned: $\hat{\boldsymbol{x}}_B = \operatorname{argmax}_{\boldsymbol{x}_B} f(\boldsymbol{x_B}, \hat{\boldsymbol{x}}_C)$. Throughout this whole process, the overarching runtime environment (hardware, compiler) is assumed to be fixed. Each of these four components will be discussed sequentially from Section 4.1 to Section 4.4, and the performance of the resulting system will be evaluated on various computational tasks in Section 4.5.

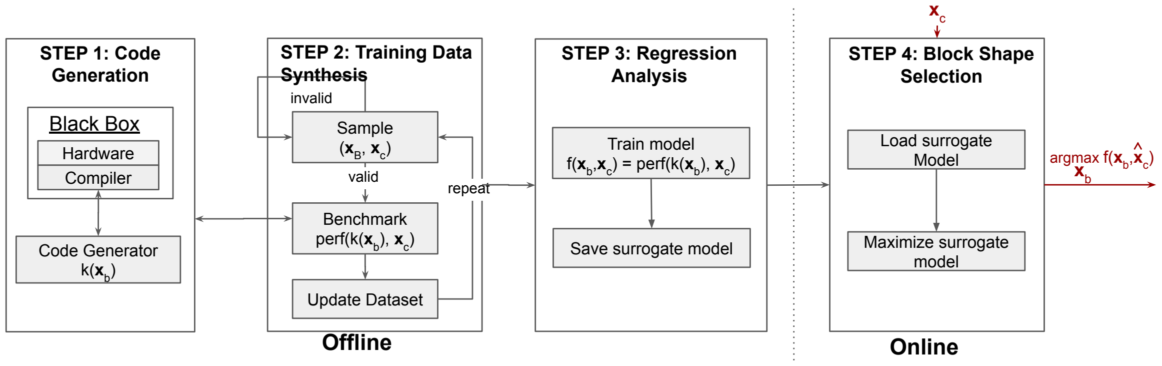


Figure 4.2: Context-aware auto-tuning.

4.1 Code Generation

The context-aware auto-tuning approach outlined above relies on the existence of a code generator $k(\boldsymbol{x}_B)$ capable of automatically generating high-performance implementations of pre-defined blocked algorithms, for any possible block shapes $\boldsymbol{x}_B$. Before we see, in Chapter 5, how to generalize the construction of such templates, this section describes the design of specialized code generators for matrix multiplications and convolutions.

4.1.1 Matrix Multiplication

Let us consider the matrix multiplication problem:

$$C = AB \qquad C \in \mathbb{R}^{M\times N},\ A \in \mathbb{R}^{M\times K},\ B \in \mathbb{R}^{K\times N}$$

Since the arithmetic intensity of this task $\alpha \sim 2M_L N_L/(M_L + N_L)$ can be high for certain values of (M_L, N_L), peak performance can only be achieved through sufficient data-reuse and latency hiding. For this reason, we develop a parameterized implementation of matrix multiplication which explores the trade-offs between (1) thread-level parallelism, (2) data pre-fetching and (3) data re-use.

Importantly, we do not consider instruction-level parallelism in our design. Indeed, modern GPUs typically outsource dependency analysis to their Instruction Set Architecture (ISA): assembly programs are required to specify *stall counts* (in addition to op-codes and operands) for every instruction. It is therefore not possible for programmers to control instruction-level parallelism unless writing Shader Assembly (SASS) directly, which we avoid for the sake of productivity. Instead, the code templates presented here use the Parallel Thread eXecution (PTX) pseudo-assembly language, where instruction-level parallelism is automatically optimized by the ptxas assembler during the instruction scheduling and selection process.

Thread Parallelism

Graphics processing units are warp-synchronous – they execute groups of 32 thread (i.e., warps) in lockstep – hence in the remainder of this dissertation we will refer to *warp-level* *parallelism* rather than thread-level parallelism. Warp-level parallelism is implemented in hardware through the use of one or multiple *warp scheduler*, which detect and pause warps that are stalled due to e.g., unfinished data transfers, freeing up compute resources that can then be used by any other warp available. For this reason, it is possible for programmers to modulate warp-level parallelism by controlling the number of kernel instances that are launched (*inter-block parallelism*) and/or the number of warps used in each instance (*intra-block parallelism*). As a result, our code generator exposes two parameters (M_L, N_L) to control the amount of work done by each kernel instance, and two other parameters (M_S, N_S) to control the amount of work done by each warp.

Data Pre-Fetching

Since the latency of DRAM memory accesses on modern GPUs is typically several orders of magnitude higher than the latency of FMAs [55, 56], it is generally recommended to overlap memory accesses with computations as much as possible. Listing 4.1-4.2 show how this can be done in the case of matrix multiplication, by *pre-fetching* blocks of data in the reduction loop before using them an iteration later. Note that this strategy is only helpful when $K \gg TK$, since the cost of the first – and not pre-fetched – memory loads needs to be amortized over many iterations.

```
int m = get_program_id(0) * ML;
int n = get_program_id(1) * NL;
float acc[ML, NL] = 0;
for(int k = 0; k < K; k += U) {
  // fetch
  float a[ML, U] = A[m:m+ML, k:k+U];
  float b[U, NL] = B[k:k+U, n:n+NL];
  // matrix-multiplication
  acc += dot(a, b);
}
C[m:m+ML, n:n+NL] = acc;
```

Listing 4.1: Blocked matrix multiplication without data pre-fetching.

```
int m = get_program_id(0) * ML;
int n = get_program_id(1) * NL;
float acc[8, 8] = 0;
// pre-fetch
float a[ML, U] = A[m:m+ML, 0:U];
float b[U, NL] = B[0:U, n:n+NL];
for(int k = 0; k < K; k += U) {
  acc += dot(a, b);
  // pre-fetch
  a = A[m:m+ML, k+U:k+2U];
  b = B[k+U:k+2U, n:n+NL];
}
C[m:m+ML, n:n+NL] = acc;
```

Listing 4.2: Blocked matrix multiplication with data pre-fetching.

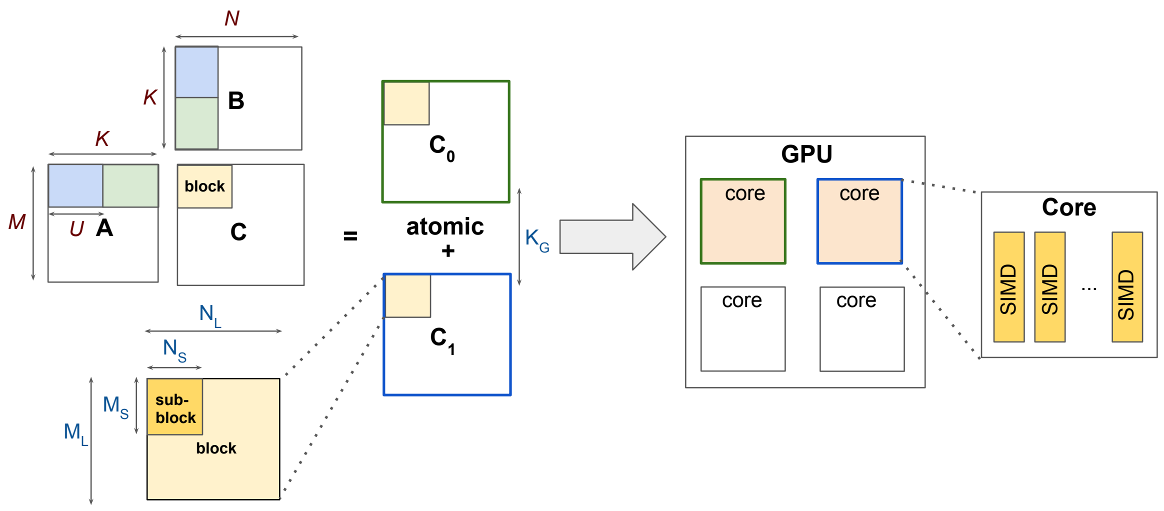


Figure 4.3: Proposed parameterization of matrix multiplication. Components of $\boldsymbol{x}_C$ and $\boldsymbol{x}_B$ are respectively shown in red and blue.

Data Reuse

As mentioned above, matrix multiplication is a potentially arithmetically intense operation which offers many opportunities for data-reuse. This property can be leveraged by making sure that the $M_L \times U$ blocks of A and the $U \times N_L$ blocks of B reside in shared memory before their elements are repeatedly accessed in the dot instruction. This reduces the latency of subsequent memory accesses and also makes latency hiding easier. The arithmetic intensity of the resulting algorithm is $\alpha = 2(M_L N_L)/(M_L + N_L)$, so M_L and N_L should be as large as possible, i.e., they should fit in shared memory and the resulting reduction loop should fit in the L1 instruction cache. Importantly, M_L and N_L should also be as close to one another as possible.

Trade-Offs

All the above optimization techniques exhibit trade-offs with one another. For example, large block shapes increase data-reuse but also require more hardware resources to function, potentially limiting the amount of warps that can run concurrently. Conversely, if blocks are too small, independent instructions will become rare and opportunities for parallelism will be reduced. Additionally, when the blocking factor along one direction (e.g., M_L) is constrained to be “small“ by e.g., the shape of input matrices, it becomes necessary to mindfully increase blocking along another dimension (e.g., N_L) to increase arithmetic intensity. Of course, what it means for a block to be “small“ or “large“ is a latent property of the underlying hardware and execution context, hence optimal block shapes depend not only on the target micro-architecture but also on characteristics of some important runtime parameters not necessarily known in advance. The system presented in this chapter automatically learns this relationship from empirical benchmarking data.

Putting It All Together

Everything considered, our parameterized implementation of matrix multiplication (Figure 4.3) is able to adjust the above factors over a wide range of values, covering many potential hardware architectures and input matrices. Each thread (resp. thread-block) computes a block of $M_S \times N_S$ (resp. $M_L \times N_L$) elements of C. In order maximize data-reuse, each program prefetches, into shared memory, $M_L \times U$ elements from A and $U \times N_L$ elements from B. These two blocks can be transposed in-place if necessary. The actual computations are then unrolled, producing a set of $M_S \times N_S \times U$ FMA instructions repeated K/U times. Because M_L and/or N_L may be constrained small by the size of A or B, it can become necessary to create additional independent work by splitting the computations along the reduction axis K, and accumulate the resulting partial results in a separate step, as shown in [57]. We therefore introduce a parameter K_G, which controls how many partial results should be computed in parallel. Accumulation may then be performed either in registers, shared memory or global memory via atomics. This technique, which we refer to as *reduction-splitting*, will be used repeatedly throughout this dissertation. To handle the cases where M (or N) is not a multiple of M_L (or N_L), we rely on predicated instructions in PTX rather than input padding.

4.1.2 Convolution

We now consider the convolution operator $C = A \star B$ such that

$$C_{c_o,:,:,z} = \sum_{c_i=0}^{C_i} A_{c_i,:,:,z} \star B_{c_i,:,:,c_o}$$

where

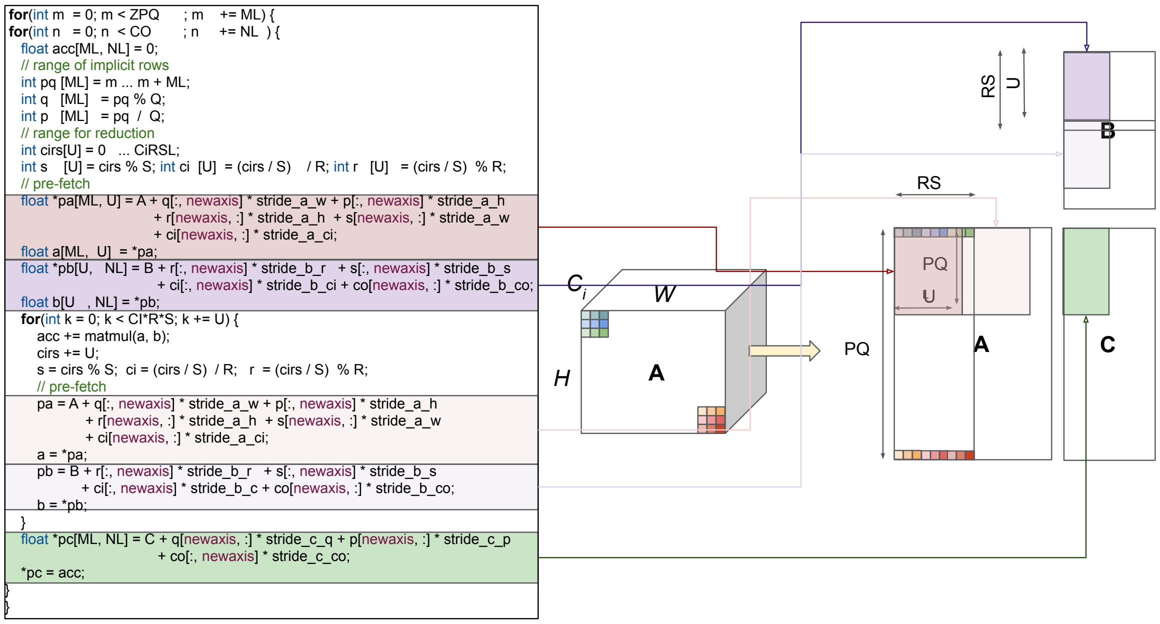
$$C \in R^{C_o \times P \times Q \times Z} \qquad A \in R^{C_i \times H \times W \times Z} \qquad B \in R^{C_i \times R \times S \times C_o}$$

This task, first described in Section 2.1, is a common generalization of the usual 2D convolution operator. Instead of convolving a single $H \times W$ image with a single $R \times S$ filter, a set of C_i different *channels* are convolved with C_i different filters, and the resulting images (of shape $P \times Q$) are summed together. This process is repeated for Z independent batches of images and C_o batches of filters, so as to eventually generate a $C_o \times P \times Q \times Z$ data tensor. This operator is one of the most important building blocks of convolutional neural networks, and it is therefore used in many different contexts for many different values of Z, C_o, C_i, H, W, R, S. As such, it constitutes a valuable benchmark for context-aware auto-tuning.

A blocked algorithm for convolution

In Section 2.1.5, we described how convolutions can be expressed as *implicit* matrix multiplications, and how doing so naively may cause significant performance issues. Following this observation, we present a blocked algorithm in which im2col is performed implicitly in cache memory, and show how it may be efficiently implemented using the exact same parameterization as that shown in Figure 4.3.

Our blocked algorithm is shown in Figure 4.4 and works as follows. The 4D output of the convolution $C \in R^{Z \times P \times Q \times C_o}$ is treated as matrix $C' \in R^{M \times N}$ where $M = ZPQ$



```
for(int m  = 0; m < ZPQ      ; m    += ML) {
for(int n   = 0; n  < CO       ; n     += NL  ) {
   float acc[ML, NL] = 0;
   // range of implicit rows
   int pq [ML] = m ... m + ML;
   int q   [ML]   = pq % Q;
   int p   [ML]   = pq  /  Q;
   // range for reduction
   int cirs[U] = 0   ... CiRSL;
   int s    [U] = cirs % S; int ci  [U]  = (cirs / S)    / R; int r   [U]   = (cirs / S)  % R;
   // pre-fetch
   float *pa[ML, U] = A + q[:, newaxis] * stride_a_w + p[:, newaxis] * stride_a_h
                                + r[newaxis, :] * stride_a_h  + s[newaxis, :] * stride_a_w
                                + ci[newaxis, :] * stride_a_ci;
   float a[ML,  U]  = *pa;
   float *pb[U,   NL] = B + r[:, newaxis] * stride_b_r   + s[:, newaxis] * stride_b_s
                                 + ci[:, newaxis] * stride_b_ci + co[newaxis, :] * stride_b_co;
   float b[U   , NL] = *pb;
   for(int k = 0; k < CI*R*S; k += U) {
      acc += matmul(a, b);
      cirs += U;
      s = cirs % S;  ci = (cirs / S)  / R;   r  = (cirs / S)  % R;
      // pre-fetch
      pa = A + q[:, newaxis] * stride_a_w + p[:, newaxis] * stride_a_h
              + r[newaxis, :] * stride_a_h  + s[newaxis, :] * stride_a_w
              + ci[newaxis, :] * stride_a_ci;
      a = *pa;
      pb = B + r[:, newaxis] * stride_b_r   + s[:, newaxis] * stride_b_s
             + ci[:, newaxis] * stride_b_c + co[newaxis, :] * stride_b_co;
      b = *pb;
   }
   float *pc[ML, NL] = C + q[newaxis, :] * stride_c_q + p[newaxis, :] * stride_c_p
                                 + co[:, newaxis] * stride_c_co;
   *pc = acc;
}
}
```

Figure 4.4: Blocked algorithm for dense convolutions. For simplicity, we assume $N = 1$.

and $N = C_o$, and computed using 2D blocks of shape $M_L \times N_L$, as before. These blocks can be computed concurrently by different thread-blocks in a 2D grid, using an *aggregate* ”row” dimension $m = (n, p, q)$ from which original dimensions can be retrieved by using the following identities.

$$\begin{cases} q &= m \ \% \ Q \\ p &= (m \ / \ Q) \ \% \ P \\ z &= (m \ / \ Q) \ / \ P \end{cases}$$

Similarly, we use an aggregate reduction dimension $k = (c_i, r, s)$ from which channel and filter indices can be retrieved using

$$\begin{cases} s &= k \ \% \ S \\ r &= (k \ / \ S) \ \% \ R \\ c_i &= (k \ / \ S) \ / \ R \end{cases}$$

Though A and B are both four-dimensional tensors, two-dimensional blocks can be loaded from memory by dereferencing, element-wise, blocks of pointers constructed using broadcasting semantics. The corresponding language constructs (e.g., `float *a[8, 8] = A[:,newaxis]`) should be intuitive to readers familiar with existing array languages such as Numpy, but a more precise definition of an appropriate C-based language will be given in Chapter 5. Once these blocks are loaded (into shared memory), they can be multiplied together to update an accumulator block that is then written back to C.

It follows that we can use the exact same parameterization method as that exposed in Figure 4.3, which is why these algorithms are often referred to as ”implicit matrix multiplication”. The only difference is that blocking is performed across aggregate dimensions ($M = PQZ, N = C_o, K = C_iRS$). Specifically, each thread (resp. thread-block) computes a block of $M_S \times N_S$ (resp. $M_L \times N_L$) elements of C. For the sake of data-reuse, each thread-block prefetches, into shared memory, $M_L \times U$ elements from A and $U \times N_L$ elements from B. The offsets for these load operations are obtained using the aforementioned indirection. The actual computations are then fully unrolled, and the reduction is split using a K_G programs per reduction.

4.2 Data Synthesis

Context-aware auto-tuning works by building a regression model $f(\boldsymbol{x}_B, \boldsymbol{x}_C)$ for the performance of the above kernel generator on any block shapes $\boldsymbol{x}_B$ and contextual characteristics $\boldsymbol{x}_C$. At runtime, when the context $\hat{\boldsymbol{x}_C}$ is fixed, this *surrogate* model is then optimized over $\boldsymbol{x}_B$ only to guide the selection of efficient block shapes $\hat{\boldsymbol{x}}_B = \text{argmax}_{\boldsymbol{x}_B} f(\boldsymbol{x}_{\boldsymbol{B}}, \hat{\boldsymbol{x}}_C)$. While $f(\boldsymbol{x}_B, \boldsymbol{x}_C)$ could technically be analytically approximated using expert knowledge, doing so would reduce its portability – and performance-portability – across future hardware architectures and applications. Instead, we learn $f(\boldsymbol{x}_B, \boldsymbol{x}_C)$ automatically from a large amount of empirical benchmarking data $\mathcal{D}$ gathered offline:

$$\mathcal{D} = \{(\boldsymbol{x}_B, \boldsymbol{x}_C, y) \mid y = perf(k(\boldsymbol{x}_B), \boldsymbol{x}_C)\}$$

Where, following the notations adopted in the previous subsection,

$$\begin{aligned}
\boldsymbol{x}_B &= (M_L, M_S, N_L, N_S, K_G) \\
\boldsymbol{x}_C^{\text{matmul}} &= (M, N, K) \\
\boldsymbol{x}_C^{\text{conv}} &= (M = ZPQ, N = C_o, K = C_iRS)
\end{aligned}$$

At first sight, the construction of $\mathcal{D}$ may seem straightforward: one can simply sample many different block shapes $\boldsymbol{x}_B$ and contextual characteristics $\boldsymbol{x}_C$, and measure the performance of every resulting *configuration*: $y = perf(k(\boldsymbol{x}_b), \boldsymbol{x}_c)$. While doing this is certainly possible, we observed that this tends to bias $\mathcal{D}$ towards inefficient samples, since most possible configurations are typically slow. To resolve this issue, Section 4.2.1 will present a rejection sampling method that conservatively discards samples unlikely to ever perform well. To improve the scalability of this method to higher-dimensional configuration spaces, Section 4.2.2 will introduce a Naive Bayes generative model that reduces the amount of samples rejected.

4.2.1 Rejection Sampling

In our parameterized implementation of matrix multiplication and convolution, the space of possible block shapes $\mathbb{X} \subset \mathbb{N}^5$ can be large. To make matters worse, many possible block shapes $\boldsymbol{x}_B$ can exhibit low device utilization due to poor data/instruction cache performance, register spilling or lack of parallelism. Therefore, sampling uniformly from $\mathbb{X}$ can skew the training dataset $\mathcal{D}$ towards inefficient samples and reduce the performance of our regression model where it matters. For this reason, we propose a rejection sampling method (see Figure 4.5) that only draws sample from the space of potentially efficient $(\boldsymbol{x}_B, \boldsymbol{x}_C)$ pairs, using pruning heuristics shown below.

First, we reject block shapes that lead to inner reduction loops unable to fit in the underlying hardware's instruction cache. In other words, assuming 16-bytes wide instruc-

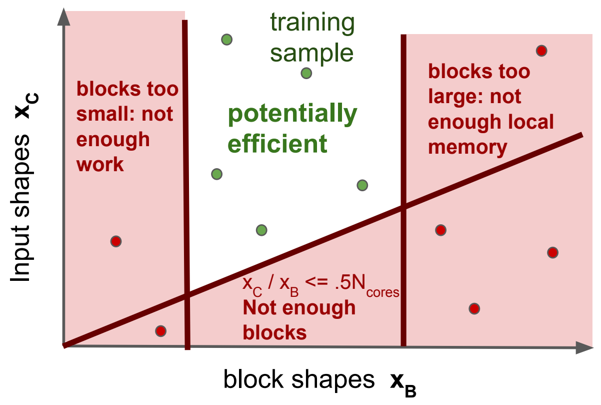


Figure 4.5: Potentially efficient samples (greens) are retained using rejection sampling.

tions (as is the case in Volta, Turing and Ampere) and an instruction cache of S_{IC} bytes per SM, we want $16N_{\text{loop}} \leq S_{IC}$, where the number of instruction N_{loop} can be approximated by $N^{A}_{\text{load}} + N^{B}_{\text{load}} + N_{\text{FMA}} = (MS \times U) + (NS \times U) + (MS \times NS \times U)$. In practice, $S_{IC} \sim 12KiB$, meaning that we may for example want $M_S = N_S = U = 8$.

Second, we reject block shapes that are not powers of two. This is not strictly necessary, but we found that this makes our search space more tractable without impacting performance. Conveniently, doing so also ensures that M_L is always divisible by M_S as long as $M_S < M_L$, and that the number of elements in a block $M_L.N_L$ is always divisible by the number of threads in a warp (32). Incidentally, this also means that blocks of size $M_L \times U$ or $N_L \times U$ can fit tightly in shared memory, which is itself usually allocated in powers of two by the driver.

Third, we reject block shapes that are too small. The rationale is that very small block sizes (e.g., 2×2, 4×4) have less elements than threads in a warp, leading inevitably to wasted computations. For this reason, we specifically discard blocks that have less than 32 elements.

Last, we reject pairs of $(\boldsymbol{x}_B, \boldsymbol{x}_C)$ that lead to low GPU occupancy. In other words, we want to avoid cases where there there are significantly less blocks than GPU cores available, which happens when $\|\boldsymbol{x}_C\|/\|\boldsymbol{x}_B\| \leq \rho N_{\text{cores}}$, where ρ is a tolerance parameter that we set to $\rho = 0.5$.

4.2.2 Generative Modeling

The pruning criteria outlined above lead to very high rejection rates – sometimes over 99.9% for matrix multiplication – that grow larger as the dimension of $\boldsymbol{x}_B$ and $\boldsymbol{x}_C$ increases. To overcome this curse of dimensionality, we propose to build a generative model $\mathcal{G}$ capable to sample more efficiently space of potentially efficient configurations (see Figure 4.6). It is easy to imagine scenarios where $\mathcal{G}$ would be defined by a complex graphical model, but this would require an analysis that is beyond the scope of this work. Instead, our framework uses a Naive Bayes method which is simpler yet significantly less wasteful than uniform sampling.

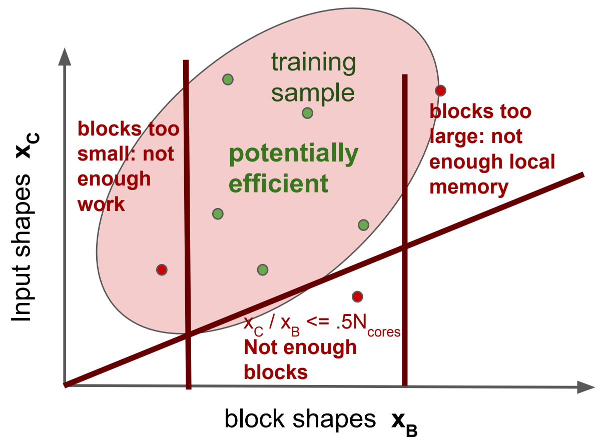

training sample
potentially efficient
blocks too small: not enough work
blocks too large: not enough local memory
$x_C / x_B <= .5N_{cores}$
Not enough blocks
Input shapes x_C
block shapes x_B

Figure 4.6: Rejection rate in high-dimensional spaces can be reduced using a Naive Bayes method.

Specifically, we treat $\boldsymbol{x} = (\boldsymbol{x}_B, \boldsymbol{x}_C)$ as a random vector whose components $\boldsymbol{x}_i$ are statistically independent assuming potential efficiency. In other words, we assume that the probability of a sample being potentially efficient is:

$$p(E = \boldsymbol{x} \in \hat{\mathbb{X}}) \; \alpha \; p(\boldsymbol{x}_0|E)p(\boldsymbol{x}_1|E) \cdots p(\boldsymbol{x}_N|E)$$

The probability distribution of each parameter $\boldsymbol{x}_i$ conditioned on E can be approximated empirically, as the proportion of accepted values after a short period of uniform sampling. For instance, assuming that $\boldsymbol{x}_1 = M_S$ may only take four values – say, $1, 2, 4, 8$ – which respectively appear 5, 20, 25 and 50 times out of 100 uniformly sampled valid configurations, our framework assigns:

$$p(\boldsymbol{x}_1 = 1) = .05 \qquad p(\boldsymbol{x}_1 = 2) = .2$$
$$p(\boldsymbol{x}_1 = 4) = .25 \qquad p(\boldsymbol{x}_1 = 8) = .5$$

To avoid setting any of these probabilities to zero, we initialize each count as a value $\alpha > 0$ (our implementation uses $\alpha = 100$).

4.3 Regression Analysis

Once a sufficient amount of training data $\mathcal{D}$ has been gathered using the above sampling method, our system builds a predictive model for the performance of any block shape $\boldsymbol{x}_B$ in any given context $\boldsymbol{x}_C$. This is known as *regression analysis*. We evaluated multiple machine learning models before opting for a multi-layer perceptron (MLP), as it (1) scales best with large datasets (given enough time and resources, our dataset can be made arbitrarily large) and (2) naturally handles common non-linearities found in performance modeling such as maximums and minimums.

4.3.1 Logarithmic Feature Transformation

Before going further, let us first quickly review the existing literature on GPU performance modeling, for which a comprehensive review was offered by Volkov in his doctoral dissertation [28]. A common strategy for estimating the average arithmetic and memory throughput (in instructions/cycles) of the target kernel K is:

$$t_{\text{arith}}(n) = \max\left(\frac{\text{alu\_latency}}{n}, \text{alu\_throughput}\right)$$
$$t_{\text{mem}}(n) = \max\left(\frac{\text{mem\_latency}}{n}, \text{mem\_throughput}\right)$$

Where n is the mean occupancy (in warps per multi-processor), and alu_throughput, mem_throughput are underlying hardware latency characteristics. The total execution time $t(n)$ of K is then:

$$t(n) = \max(t_{\text{arith}}(n) i_{\text{arith}}, \quad t_{\text{mem}}(n) i_{\text{mem}})$$

Where i_{arith} and i_{mem} are respectively the number of arithmetic and memory instructions in K. The entire premise of our approach is that all the quantities involved in these computations depend – more or less strongly – on the relationship between *hidden* hardware features (e.g., number of ALU, memory bandwidth, maximum throughput, banking structures) and *known* environmental or contextual parameters (e.g., input tensor shapes, block shapes). A successful MLP should (implicitly) learn not only these relationships but also the corresponding hidden variables.

As suggested by the above analytical model, it is expected that our target performance model include multiplications, divisions and maximums between components of $\boldsymbol{x}_B$ and components of $\boldsymbol{x}_C$. Because neural networks are not naturally designed to handle multiplications between different features, modifying the input features as $\boldsymbol{x}' = \log(\boldsymbol{x})$ can greatly improve the performance of our system. We also found the Rectified Linear Units (ReLU) to perform best than any other activation function, possibly due to their natural affinity with $\max$ computations commonly found in analytical models.

4.3.2 Accuracy

A common criticism of neural networks is that they are hard to engineer. In this subsection, we attempt to provide insights for designing good MLP architectures for context-aware auto-tuning, as well as intuition regarding the amount of training data necessary to achieve good performance. We used matrix multiplication for our analysis, but the same results were observed in convolutions, which is expected since the underlying parameterization is the same.

Table 4.1 shows the cross-validation MSE of several MLP architectures, as measured on a fixed set of $10,000$ cross-validation data-points separate from the $200,000$ samples used for training. Unsurprisingly, deeper networks seem to perform much better than shallower ones for a fixed parameter budget. In other words, the accuracy of the network can be improved by adding layers, at the cost of longer training and higher runtime latency. We emphasize the importance of the logarithmic feature transformation exposed in the previous subsection, without which our system would – at best – converge to much worse solutions.

Hidden layer sizes	#weights	MSE (no log)
64	1k	0.17 (1.2)
512	10k	0.13 (1.0)
32, 64, 32	5k	0.088 (0.80)
64, 128, 64	17k	0.08 (0.75)
32, 64, 128, 64, 32	21k	0.073 (-)
64, 128, 256, 128, 64	83k	0.067 (-)
64, 128, 192, 256, 192, 128, 64	163k	0.062 (-)

Table 4.1: Cross-validation MSE of various MLP architectures.

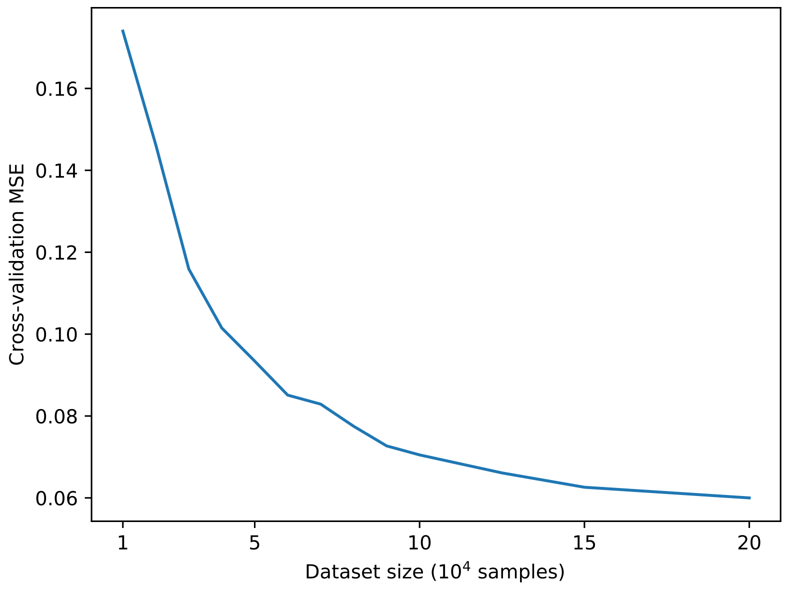


Figure 4.7: Cross-validation MSE of our most accurate MLP for various data-set sizes.

Figure 4.7 shows the evolution of our most accurate MLP's accuracy as the amount of training data available grows. Collecting more data does not seem to provide much benefits beyond $\sim 150,000$ samples, or 6 hours of data collection.

4.4 Runtime Inference

At this point, we have designed a trained regression model that can predict, for matrix multiplication and convolution, the performance of any given block shapes in any given context. This model can be evaluated very quickly, in parallel, and with roughly constant latency. This differs from *direct* auto-tuning methods for GPUs, which may be slow, lock the underlying device, or even time-out when very inefficient kernels meet large problems. At runtime, the context is known, and our regression model can be optimized over block shapes only. Any discrete optimization method (e.g., simulated annealing, genetic algorithm, exhaustive search) may be used for this purpose. In this work, we have opted for an exhaustive search, as it has several attractive properties when applicable:

- It is guaranteed to find the global optimum within the specified search range.
- The search is highly parallelizable. On our best-performing MLP, up to a million different configurations per second can be evaluated.
- It is straightforward to obtain the 10 (or more) fastest configurations instead of the single top prediction, and re-evaluate them to smooth out the imperfections of our predictive model.

The cost of this exhaustive search is relatively small – up to a few seconds – because our surrogate model is fast to evaluate. In comparison, standard context-agnostic auto-tuning procedure can be several orders of magnitude slower, taking up to 10 hours *per* *input matrix shape* for some large matrices. The resulting predictions may be used directly in applications where this latency would be negligible, or cached on the filesystem, or even used as heuristics for external libraries such as cuBLAS/cuDNN.

4.5 Numerical Evaluation

In this section, we evaluate the performance of our context-aware auto-tuning framework on various matrix-multiplication and convolution problem settings found in scientific computing, deep learning and signal processing.

4.5.1 Hardware architectures

While our work focuses primarily on context-awareness, it is equally important for our system to be performance-portable across existing and future hardware architectures, hence our numerical experiments will be repeated on two distinct GPUs whose main characteristics are summarized in Table 4.2.

	Pascal	**Volta**
GPU	GTX1070 Ti	Tesla V100
Micro-architecture	GP104	GV100
Die size	314 mm^2	815 mm^2
CUDA cores	2432	5148
Boost frequency	1683 MHz	1530 MHZ
Processing Power	8.1 TFLOPS	15.8 TFLOPS
Memory quantity	8 GB	32 GB
Memory Type	GDDR5	HBM2
Memory Bandwidth	256 GB/S	900 GB/s
TDP	180W	300W

Table 4.2: Hardware platforms considered in this section.

These two devices, though both designed by NVIDIA within the span of two years, differ in many ways. First, the Tesla V100 has a larger die size than the GTX1070 Ti, which leads to more CUDA cores. Second, the V100 offers twice the bandwidth of the GTX1070 TI due to its use of High Bandwidth Memory (HBM2) DRAM. It is worth pointing out that HBM2 (large bus width, low frequency) and GDDR5 (small bus width, high frequency) handle memory transfers in a different way, to the point where even IO-bound code designed for Pascal is not guaranteed to perform well on Volta.

Note that, for reasons that are beyond the scope of this work, NVIDIA's ptxas compiler tends to generate higher-quality code for Volta than Pascal.

4.5.2 Experimental setup

We compare our framework against cuBLAS 9.2 and cuDNN 7.0. Despite the development of new domain specific languages for array programming (Chapter 3), these two libraries have remained the gold standard for linear algebra and deep learning on GPUs. Both libraries rely on handcrafted heuristics for choosing among a set of statically optimized assembly implementation of matrix multiplication. The cuBLAS API, however, makes it possible to manually call individual kernels using the cublasGemmEx function, effectively allowing us to bypass any existing heuristics. We use this feature to select the best cuBLAS kernel available for each input shapes considered, ensuring that the performance gains we observe are not just due to poor block shape selection heuristics in cuBLAS. As for convolutions, we use the flag IMPLICIT_PRECOMP_GEMM to force cuDNN to use the algorithm presented in Section 4.1, with a scratch space of 64MB that remains on the device throughout the entire duration of our benchmarks.

4.5.3 GEMM Performance

General Matrix Multiplication (GEMM) sits at the heart of High-Performance Computing (HPC). It is a crucial workload for many applications, including scientific computing, machine learning and signal processing. In this section, we evaluate our proposed autotuning method on a set of input configurations that we believe are representative of its practical usage.

Square Matrices

Though square matrices are relatively uncommon in practice, they are often used as a benchmark for GPU implementations of matrix multiplication. For this reason, this subsection evaluates the performance of our system on square matrices of increasing size $128 \leq N \leq 3072$. As shown in Figure 4.8, context-aware auto-tuning (CAAT) almost exactly matches the performance of cuBLAS's handwritten assembly across the board on the GTX1070Ti, which indicates that handwritten SASS may not necessary to achieve peak FP32 performance on this device. This supports our decision to choose PTX as an intermediate language for code generation. Similar performance trends are observed on the Tesla V100.

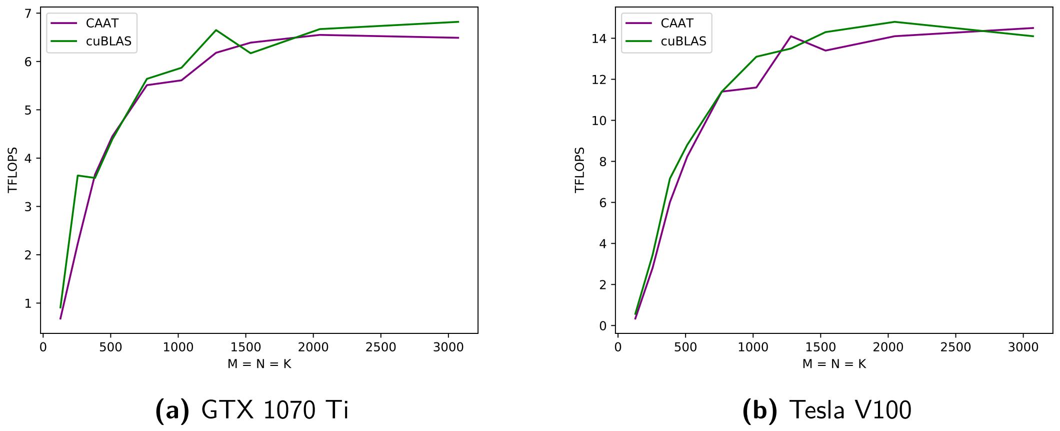
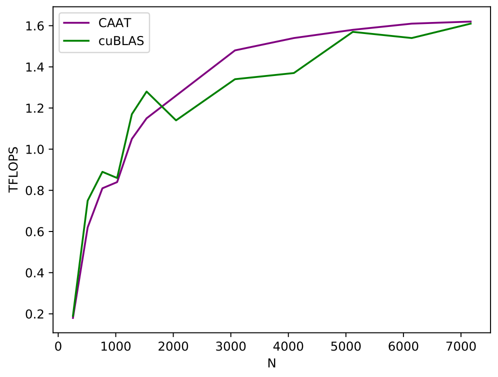


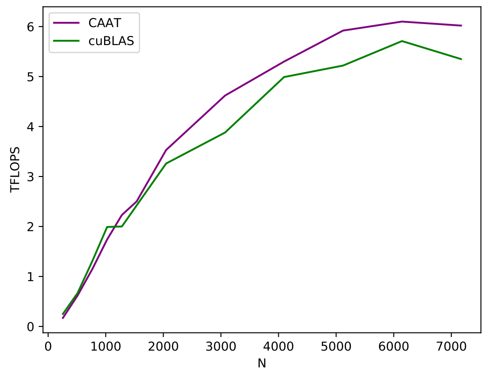
Figure 4.8: Performance of context-aware auto-tuning for square matrix multiplication.

Multi-Layer Perceptron

The benefits of CAAT are more apparent for tasks involving tall and skinny matrices. To evaluate this problem domain, we measure the performance of our framework on batched MLP inference tasks $C = A.B^T$, where $A \in \mathbb{R}^{N \times N}$ and $B \in \mathbb{R}^{16 \times N}$ for $256 \leq N \leq 7168$. This corresponds to forward propagation across linear layers of increasing size.



(a) GTX 1070 Ti



(b) Tesla V100

Figure 4.9: Performance of context-aware auto-tuning for batched MLP inference ($N = 16$).

Figure 4.9 shows the performance of CAAT and cuBLAS on both hardware considered. As we can see, CAAT and cuBLAS are largely on par when $N < 2048$. On the other hand, CAAT outperforms cuBLAS for larger matrices. This is because, as shown in Table 4.3, cuBLAS does not implement any kernel satisfying $N_L \leq 16$, leading to wasted computations that matter in compute-bound regimes since C only has 32 columns.

N	M_L	N_L	K_G	N_{programs}
512	32	16	1	16
1536	64	16	4	96
4096	64	16	4	256

(a) Context-Aware Auto-Tuning

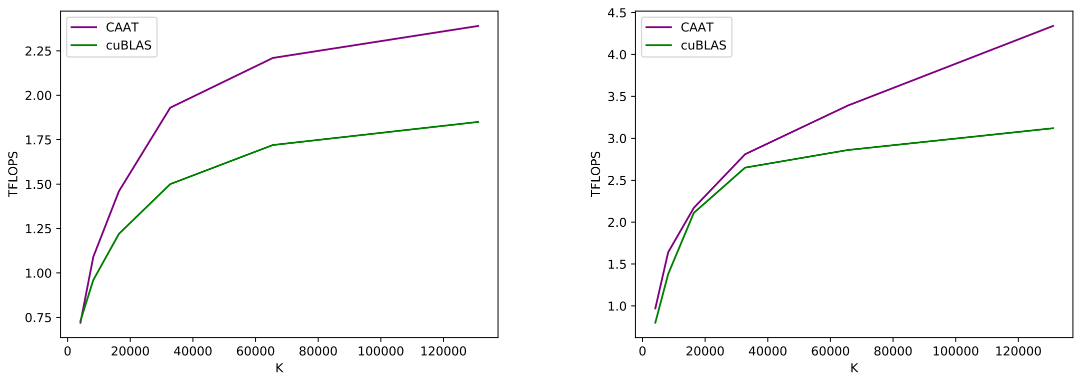
N	M_L	N_L	K_G	N_{programs}
512	32	32	4	64
1536	64	32	4	96
4096	64	32	6	384

(b) cuBLAS

Table 4.3: block shapes selection heuristics for different values of N on the Tesla V100.

Covariance

We now consider the matrix multiplication task $C = A.B^T$, where $A \in \mathbb{R}^{M\times K}$ and $B \in \mathbb{R}^{N\times K}$, for $M = N = 64$ and $4096 \leq K \leq 131072$. This task is common in statistical computing, when e.g., approximating the covariance matrix of $M = N$ random variables using K observations. Reduction-splitting ($K_G > 1$) is critical to achieve good performance in this scenario, since the result matrix $C \in \mathbb{R}^{64\times 64}$ is small and its computation cannot be massively parallelized without compromising data reuse.



(a) GTX 1070 Ti **(b)** Tesla V100

Figure 4.10: Performance of context-aware auto-tuning for covariance matrix computations.

Figure 4.10 shows the performance of CAAT and cuBLAS on this task. As one can see, our system achieves significant performance gains over cuBLAS for large value of K. To understand why this is the case, Table 4.4 shows the compute kernels selected by our auto-tuning framework for different values of K, along with the number of program instances required in each case. Since the GTX1070Ti is composed of 30 SMs containing 2 warp schedulers each, we approximate the maximum number of programs necessary to saturate this GPU as $2N_{SM} = 60$. Note that, while occupying a GPU completely is not necessary to achieve optimal performance in memory-bound workloads [28], occupying it at the expense of data-reuse can be harmful. The kernels selected by cuBLAS seem to fall prey to this trap, privileging reduction-splitting (i.e., parallelism) over larger block sizes (i.e., data re-use) for $K > 16384$. On the other hand, CAAT seems to automatically find better trade-offs, increasing N_L instead of K_G when the GPU is saturated ($K = 131072$), leading to the significant performance gains ($> 50\%$) as shown in Figure 4.10(a). Interestingly, we also note that CAAT automatically learns that larger block shapes are preferable.

Similar performance trends are observed on the Tesla V100, where speed-ups of $> 50\%$ are seen for very deep reductions (Figure 4.10b). Since this GPU has 80 SMs, up to 160

K	M_L	N_L	K_G	N_{programs}
4096	64	32	8	16
8192				
16384	64	32	16	32
32768				
65536				
131072	64	64	32	32

(a) Context-Aware Auto-Tuning

K	M_L	N_L	K_G	N_{programs}
4096	32	32	2	8
8192	32	32	4	16
16384	32	32	8	32
32768	32	32	16	64
65536	32	32	32	128
131072	32	32	64	256

(b) cuBLAS

Table 4.4: block shape selection heuristics for different values of K on an NVIDIA GTX 1070Ti. Configurations requiring more programs than the GPU can execute concurrently ($2N_{\mathsf{SM}} = 60$) are shown in red.

programs can execute concurrently without saturating the GPU (Table 4.5). Unsurprisingly, our system shows the largest performance gains under this regime ($K \geq 65536$), as no substantial difference with cuBLAS is noted for shallower reductions (i.e., $K \leq 32768$).

K	M_L	N_L	K_G	N_{programs}
4096	32	32	16	64
8192	32	32	16	64
16384	32	32	16	64
32768	32	32	16	64
65536	32	32	32	128
131072	32	32	32	128

(a) Context-Aware Auto-Tuning

K	M_L	N_L	K_G	N_{programs}
4096	32	32	16	64
8192	32	32	16	64
16384	32	32	32	128
32768	32	32	32	128
65536	32	32	64	256
131072	32	32	128	512

(b) cuBLAS

Table 4.5: block shape selection heuristics for different values of K on an NVIDIA Tesla V100. Configurations requiring more programs that the GPU can execute concurrently ($2N_{\mathsf{SM}} = 160$) are shown in red.

4.5.4 CONV Performance

In this subsection, we compare the performance of our work against that of state-of-the-art convolution routines from cuDNN. We specifically examine the use of 3×3 filters for inference ($Z = 1$), and evaluate two scenarios: one in which the resolution of the input image is fixed, and one in which its number of channels is fixed.

Fixed Input Resolution

We consider input images of 56 × 56 pixels – as in the middle layers of typical ResNets for ImageNet – and vary $C_i = C_o$ between 64 and 1024 so as to cover a wide range of layer widths commonly seen in the literature.

$C_i = C_o$	M_L	N_L	K_G	N_{programs}
64	64	64	1	49
128	64	128	1	49
256				98
384				147
512				196
768				294
1024				392

(a) Context-Aware Auto-Tuning

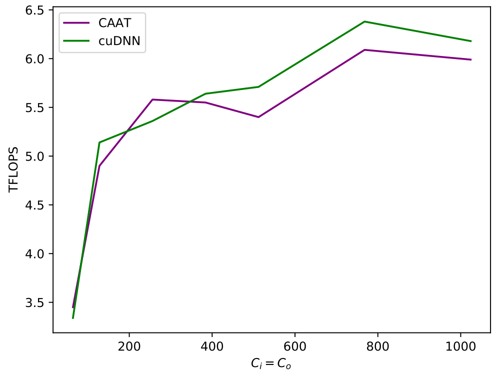
$C_i = C_o$	M_L	N_L	K_G	N_{programs}
64	128	64	1	25
128				50
256				100
384				150
512				200
768				300
1024				400

(b) cuDNN

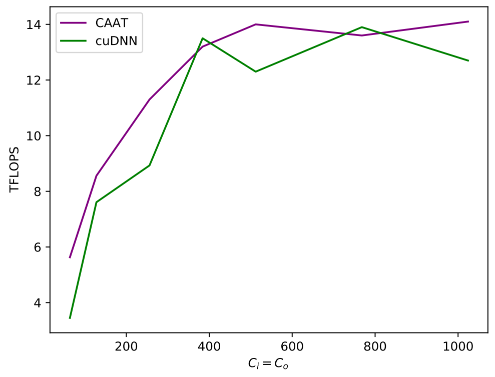
Table 4.6: block shape selection heuristics for different values of $C_i = C_o$ on an NVIDIA GTX 1070Ti.

As shown in Table 4.6, cuDNN always uses rather large tile sizes (i.e., $(M_L, N_L) = (128, 64)$), regardless of the number of input/output channels being processed. This is increases data reuse but also limits parallelism, which is not a major issue for small GPUs (GTX1070 Ti) that only have a few dozens SMs and can remain fully occupied by only 50 program instances. As a result, our system is generally on par with NVIDIA's handwritten assembly kernels on this hardware (see Figure 4.11(a)). Note however the slightly better ($\sim$ 10%) performance of cuDNN for compute-bounds routine ($C \geq 512$) where the slight advantages of handwritten assembly (ie, instruction scheduling, register allocation) are visible.

However, for larger GPUs (Table 4.7), the benefits of context-aware auto-tuning are more pronounced. Our framework privileges smaller block shapes that allow for finer-grained parallelism, while cuDNN uses the same 128 × 64 block shapes as before. This can lead to notable performance gains when the number of channels is low ($C \leq 256$) and smaller blocks are necessary to fully occupy our GPUs. As a result, for $C = 64$, our approach is 50% faster than cuDNN. For bigger layers ($C \geq 768$), CAAT uses 128 × 128



(a) GTX 1070 Ti



(b) Tesla V100

Figure 4.11: Performance of our context-aware auto-tuner on (a) GTX1070 Ti and (b) Tesla V100

block shapes, which provides more data reuse than in cuDNN.

$C_i = C_o$	M_L	N_L	K_G	N_{programs}
64	32	32	1	196
128				392
256	64	64	1	196
384				294
512				392
768	128	128	1	150
1024				200

(a) Context-Aware Auto-Tuning

$C_i = C_o$	M_L	N_L	K_G	N_{programs}
64	128	64	1	25
128				50
256				100
384				150
512				200
768				300
1024				400

(b) cuDNN

Table 4.7: block shape selection heuristics for different values of $C_i = C_o$ on an NVIDIA Tesla V100.

Fixed Channel Width

We now consider convolutional layers composed of $C_i = C_o = 256$ channels, and vary the resolution $H = W$ of their input image between 8 and 256, which allows us to consider a large variety of potential CNN workloads. Once again, we report the compute kernels selected by our method and cuDNN in Table 4.8, and observe the same trend as before: the heuristics used by cuDNN limit parallelism for small images. This reduces the performance of cuDNN, especially for large GPUs (e.g., Tesla V100). As shown in Figure 4.12, our system outperforms cuDNN by up to $2x$ in these scenarios (e.g., 0.4 vs 1 TFLOPs for $H = W = 8$).

$H = W$	M_L	N_L	K_G	N_{programs}
8	32	64	1	8
16				32
32	64	128	1	32
64				128
128				512
256				2048

(a) Context-Aware Auto-Tuning

$H = W$	M_L	N_L	K_G	N_{programs}
8	128	64	1	4
16				8
32				32
64				128
128				512
256				2048

(b) cuDNN

Table 4.8: block shape selection heuristics for different values of $H = W$ on a GTX 1070Ti.

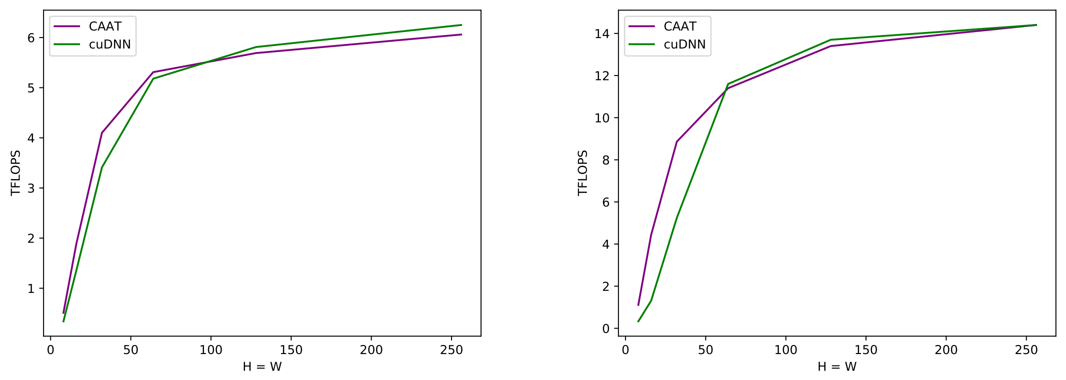
$H = W$	M_L	N_L	K_G	N_{programs}
8	32	32	1	16
16	32	32	1	64
32	64	64	1	64
64	64	128	1	128
128	64	128	1	256
256	128	128	1	512

(a) Context-Aware Auto-Tuning

$H = W$	M_L	N_L	K_G	N_{programs}
8	128	64	1	4
16				8
32				32
64				128
128				512
256				2048

(b) cuDNN

Table 4.9: block shape selection heuristics for different values of $H = W$ on an NVIDIA Tesla V100.



(a) GTX 1070 Ti

(b) Tesla V100

Figure 4.12: Performance of our context-aware auto-tuner on (a) GTX1070 Ti and (b) Tesla V100

4.6 Summary

In this chapter, we described the challenges of block shapes selection for blocked algorithms, and introduced a method for doing this task automatically using carefully sampled performance data. We first described, in Section 4.1, the design and implementation of efficient parameterized blocked algorithms for matrix multiplication and convolution. We then presented, in Section 4.2, a methodology for efficiently sampling the performance of these implementations in various execution contexts, and used these samples to train a predictive log-feature-transformed performance model in Section 4.3. In Section 4.4, we showed how this model may be used as a surrogate for empirical performance measurements to accelerate existing auto-tuning methods. The resulting system was benchmarked and analyzed in Section 4.5, where notable speed-ups over state-of-the-art vendor libraries were observed on both the GeForce GTX1070Ti and the Tesla V100 micro-architectures.

Although this approach seems to work well at first sight, it suffers from one major drawback: efficient parameterized code templates are hard to write, optimize and maintain. For reference, the blocked algorithms presented in this chapter took us 6 months to implement in PTX – making it impossible for us to consider emerging hardware features (i.e., tensor cores) while productively addressing the concerns of our system's users. Our PTX code templates are also unlikely to remain performance-portable as hardware specializes and new tensor intrinsics become available. What we need is a more systematic approach to code generation for blocked algorithms.

5

Block-Level Data-Flow Analysis

To support the claim that block-based programming paradigms facilitate the construction of efficient compute kernels for DNNs, we need to show the existence of systems capable of transforming high-level descriptions of blocked algorithms into high-performance GPU code. While this problem has been well-studied over the past few years (see Chapter 3), existing solutions do not support iteration spaces that may arise in emerging neural network architectures (e.g., sparse transformer). Therefore, this chapter presents the design and implementation of Triton [1], a language and compiler for the blocked ”single-program, multiple-data” paradigm discussed in Chapter 1. We specifically address, in order, the issue of (1) specifying, (2) representing and (3) compiling algorithms in this framework.

In Section 5.1, we introduce **Triton-C**, an imperative language for specifying blocked algorithm using relatively high-level primitives. Although the syntax of this language

[1]Triton is freely available under the MIT/X11 license at https://triton-lang.org

(Listing 5.1) may seem at first sight similar to that of CUDA-C, a deeper look reveals two major differences: (1) multi-dimensional blocks of data (e.g., `float* px[TM, TN]`) are first-class citizens and (2) compute kernels are *single-threaded*, though multiple instances of each kernel may execute in parallel. We believe that Triton-C could constitute a viable alternative to CUDA for developers unfamiliar with the details of modern GPU architectures.

```
void transpose(float* X __noalias __readonly __aligned(16),
               float* Y __noalias __writeonly __aligned(16),
               int M __multipleof(8),
               int N __multipleof(8)) {
  // program ids
  int pid0 = get_program_id(0);
  int pid1 = get_program_id(1);
  // range for rows
  int rm[TM] = pid0 * TM + 0 ... TM;
  // range for columns
  int rn[TN] = pid1 * TN + 0 ... TN;
  bool in_bounds[TM, TN] = rm[:, newaxis] < M &&
                           rn[newaxis, :] < N;
  // pointers to X
  float* px[TM, TN] = X + rm[:, newaxis] * 1
                        + rn[newaxis, :] * N;
  // pointers to Y
  float* py[TM, TN] = Y + rm[:, newaxis] * N
                        + rn[newaxis, :] * 1;
  // predicated write-back
  *?(in_bounds)py = *?(in_bounds)px;
}
```

Listing 5.1: $Y = X^T$ in Triton-C. Keywords specific to Triton are shown in purple.

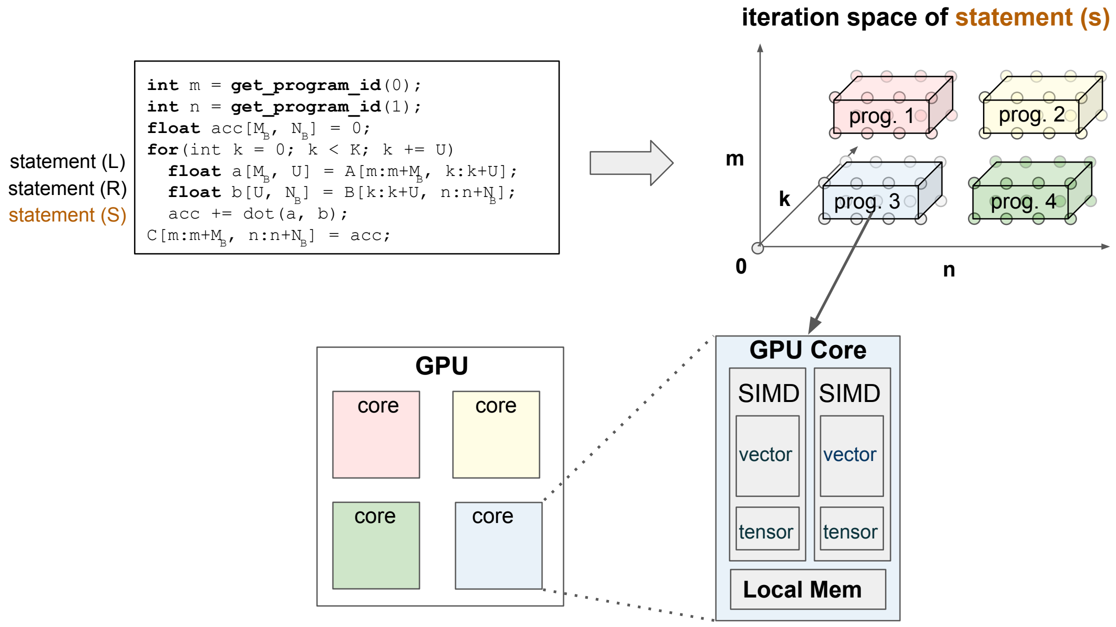
In Section 5.2, we then present **Triton-IR**, an LLVM-based Intermediate Representation (IR) for block-level program analysis, transformation and optimization. Listing 5.2 shows the Triton-IR code for the transposition kernel specified in Listing 5.1. In our system, Triton-IR programs are constructed directly from Triton-C after parsing, but automatic code generation from higher-level DNN compilers (e.g., TVM) could also be explored in the future.

Most importantly, in Section 5.3, we present **Triton-JIT**, a Just-In-Time (JIT) compiler and code generation backend which makes heavy use of the blocked structure of iteration spaces enforced by Triton-C and Triton-IR. We specifically discuss the issue of

```
def void transpose(f32* X .readonly .noalias .aligned(16) , f32* Y .writeonly .noalias
    .aligned(16) , i32 M .readonly , i32 N .readonly )
{
entry:
  pid0 = get_program_id(0) i32;
  pid1 = get_program_id(1) i32;
  ; range for rows
  %2 = make_range[0 : 32] i32<32>;
  %3 = mul i32 pid0, 32;
  %5 = splat i32<32> %3;
  rm = add i32<32> %5, %2;
  ; range for columns
  %10 = make_range[0 : 32] i32<32>;
  %11 = mul i32 pid1, 32;
  %13 = splat i32<32> %11;
  rn = add i32<32> %13, %10;
  ; bounds-checking
  %18 = splat i32<1, 32> N;
  %20 = reshape i32<1, 32> rn;
  %22 = icmp_slt i1<1, 32> %20, %18;
  %25 = broadcast i1<32, 32> %22;
  %27 = zext i32<32, 32> %25;
  %29 = splat i32<32, 1> M;
  %31 = reshape i32<32, 1> rm;
  %33 = icmp_slt i1<32, 1> %31, %29;
  %36 = broadcast i1<32, 32> %33;
  %38 = zext i32<32, 32> %36;
  %40 = and i32<32, 32> %38, %27;
  %43 = broadcast i32<32, 32> %40;
  in_bounds = trunc i1<32, 32> %43;
  ; pointers to X
  %47 = splat i32<1, 32> N;
  %49 = reshape i32<1, 32> rn;
  %51 = mul i32<1, 32> %49, %47;
  %54 = broadcast i32<32, 32> %51;
  %56 = splat i32<32, 1> 1;
  %57 = reshape i32<32, 1> rm;
  %59 = mul i32<32, 1> %57, %56;
  %62 = splat f32*<32, 1> X;
  %64 = getelementptr f32*<32, 1> %62, %59;
  %67 = broadcast f32*<32, 32> %64;
  px = getelementptr f32*<32, 32> %67, %54;
  ; pointers Y
  %72 = splat i32<1, 32> 1;
  %73 = reshape i32<1, 32> rn;
  %75 = mul i32<1, 32> %73, %72;
  %78 = broadcast i32<32, 32> %75;
  %80 = splat i32<32, 1> N;
  %82 = reshape i32<32, 1> rm;
  %84 = mul i32<32, 1> %82, %80;
  %87 = splat f32*<32, 1> Y;
  %89 = getelementptr f32*<32, 1> %87, %84;
  %92 = broadcast f32*<32, 32> %89;
  py = getelementptr f32*<32, 32> %92, %78;
  %97 = splat f32<32, 32> undef;
  %98 = masked_load f32<32, 32> px, in_bounds, %97;
  masked_store void py, %98, in_bounds;
}
```

Listing 5.2: $Y = X^T$ in Triton-IR. Keywords specific to Triton are shown in purple.

scheduling different blocks of this iteration space onto different GPU cores, and use block-level data-flow analysis to map work onto different SIMD units (see Figure 5.1). To this end, this section will present a collection of techniques to simplify computations, parallelize computations, parallelize memory accesses and maximize data-reuse at the level of individual GPU cores.



```
int m = get_program_id(0);
int n = get_program_id(1);
float acc[M_B, N_B] = 0;
for(int k = 0; k < K; k += U)
  float a[M_B, U] = A[m:m+M_B, k:k+U];
  float b[U, N_B] = B[k:k+U, n:n+N_B];
  acc += dot(a, b);
C[m:m+M_B, n:n+N_B] = acc;
```

Figure 5.1: High-level mapping of Block-Structured Iteration Spaces onto Generic GPU architectures. Individual iteration blocks are distributed onto the SIMD units of a GPU core.

Finally, in Section 5.4, we present a numerical evaluation of Triton that demonstrates performance within 10% of the state-of-the-art PTX code presented in the previous chapter, as well as significant speed-ups over TVM for matrix multiplication and convolutions – with or without tensor cores.

5.1 Triton-C

Throughout this dissertation, we have stressed the importance of blocked algorithms and described their core principles in pseudo-code. To facilitate their implementation on modern GPU hardware, we present Triton-C, a single-threaded imperative language in which block variables are first-class citizen. This language may be used either directly by developers familiar with low-level GPU programming, or as an intermediate language for existing (and future) transcompilers. In this section, we describe the CUDA-like syntax of Triton-C (Section 5.1.1), its Numpy[58]-like semantics (Section 5.1.2) and its "Single-Program, Multiple-Data" (SPMD) programming model (Section 5.1.3).

5.1.1 Syntax

The syntax of Triton-C is based on that of ANSI C (more specifically CUDA-C), but was modified and extended to accomodate the semantics and programming model described in the next two subsections. These changes fall into the following categories:

Variable declarations: We added special-purpose syntax for multi-dimensional array declarations (e.g., `int block[16, 16]`), which purposely differs from that of nested arrays found in ANSI C (e.g., `int block[16][16]`). Block dimensions must be constant but can also be made parametric with the use of pre-processor macros. One-dimensional blocks of integers may be initialized using ellipses (e.g., `int range[16] = 0 ... 16`).

Built-in function: The usual C operators were extended to support element-wise array operations (`+`, `-`, `&&`, `*`, etc.), and various built-in functions were added for concurrency (`get_program_id`) and common block-level linear algebra primitives (`dot`, `trans`).

Slicing and broadcasting: Multi-dimensional blocks can be broadcast along any particular dimension using numpy-like slicing syntax (e.g., `int array[8, 8] = range[:, newaxis]` for stacking columns). Note that, as of now, slicing blocks to retrieve sub-blocks (or scalars) is forbidden as it is incompatible with the automatic parallelization methods presented in Section 5.3.2.

Masked pointer dereferencement: Block-level operations in Triton-C are atomic: they execute either completely or not at all. Basic control-flow for block-level operations can nonetheless be achieved using ternary operators and the *masked pointer dereferencement* operator exemplified in Listing 5.3.

```
// create mask
bool mask[16, 16] = ...;
// conditional addition
float x[16, 16] = mask ? a + b : 0;
// conditional load
float y[16] 16] = mask ? *ptr : 0;
// conditional store
*?(mask)ptr = y;
```

Listing 5.3: Block-Level control flow in Triton-C.

5.1.2 Semantics

Block-Level Semantics

The existence of built-in *block-level* types, variable and operations in Triton-C offers two main benefits. First, it simplifies the structure of blocked programs by hiding important details pertaining to concurrent programming such as memory coalescing [59], cache management [60] and specialized tensor instrinsics [61]. Second, it opens the door for compilers to perform these optimizations automatically, as discussed in Section 5.3.

Broadcasting Semantics

Block variables in Triton are strongly typed, meaning that certain instructions statically require their operands to satisfy strict shape constraints. For example, a scalar may not be added to an array unless it is first appropriately broadcast. *Broadcasting semantics* [58] provides two rules for performing these conversions automatically in the case of binary operators (Listing 5.4): (1) the shape of the lowest-dimension operand is left-padded with ones until both operands have the same dimensionality; and (2) the content of both operands is replicated as many times as needed until their shape is identical. An error is emitted if this cannot be done.

```
int a[16], b[32, 16], c[16, 1];
// a is first reshaped to [1, 16]
// and then broadcast to [32, 16]
int x_1[32, 16] = a[newaxis, :] + b;
// Same as above but implicitly
int x_2[32, 16] = a + b;
// a is first reshaped to [1, 16]
// a is broadcast to [16, 16]
// c is broadcast to [16, 16]
int y[16, 16] = a + c;
```

Listing 5.4: Broadcasting semantics in practice.

5.1.3 Programming Model

As discussed in Section 2.2, The execution of CUDA [62] code on GPUs is supported by an SPMD [14] programming model in which each kernel instance is associated with an identifiable *thread-block*, itself decomposed into *warps* of 32 *threads*. The Triton programming model is similar, but each kernel is *single-threaded* – though automatically parallelized – and associated with a global program id which varies from instance to instance. This approach leads to simpler kernels in which CUDA-like concurrency primitives (shared memory synchronization, inter-thread communication, etc.) do not exist. The global program ids associated with each kernel instance can be queried using the get_program_id(axis) built-in function in order to create e.g., blocks of pointers as shown at the beginning of this chapter (Listing 5.1).

5.2 Triton-IR

Triton-IR is an LLVM-based Intermediate Representation (IR) whose purpose is to provide an environment suitable for block-level program analysis, transformation and optimization. In this work, Triton-IR programs are constructed directly from Triton-C after parsing, but they could also be formed directly by higher-level DSLs in the future. Triton-IR and LLVM-IR programs share the same high-level structure (recalled in Section 5.2.1), but the former also includes a number of extensions necessary for block-level data-flow

(Section 5.2.2) and control-flow (Section 5.2.3) analysis. These extensions are crucial for carrying out the optimizations outlined in Section 5.3.

5.2.1 Structure

Modules

At the highest level, Triton-IR programs consist of one or multiple basic units of compilation known as *modules*. These modules are compiled independently from one another, and eventually aggregated by a linker whose role is to resolve forward declarations and adequately merge global definitions. Ech module itself is composed of functions, global variables, constants and other miscellaneous symbols such as metadata and attributes.

Functions

Triton-IR function definitions consist of a return type, a name and a potentially empty arguments list. Additional visibility, alignment and linkage specifiers can be added if desired. Function attributes (such as inlining hints) and parameter attributes (such as ”readonly”, aliasing hints) can also be specified, allowing compiler backends to perform more aggressive optimizations by, for instance, making better use of read-only memory caches found on NVIDIA GPUs. This header is followed by a body composed of a list of basic blocks whose interdependencies form the *Control Flow* Graph (CFG) of the function.

Basic Blocks

Basic blocks are straight-line code sequences that may only contain so-called *terminator* instructions (i.e., branching, return) at their end. To simplify program analysis, Triton-IR uses the Static Single Assignment (SSA) form, meaning that each variable in each basic block must be (1) assigned to only once and (2) defined before being used. In so doing, each basic block implicitly defines a *Data-Flow* Graph (DFG) whose different paths correspond to *use-def* chains in the program's SSA representation. The SSA form can be created directly from C-like Abstract Syntax Trees (ASTs) as shown in [63].

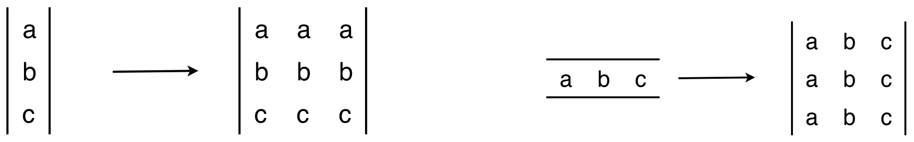
5.2.2 Block-Level Data-Flow

Types

Multi-dimensional blocks are at the center of data-flow analysis in Triton-JIT. They can be declared using syntax similar to vector declarations in LLVM-IR. For example, i32<8, 8> is the type corresponding to 8×8 blocks of 32-bit integers. Note that there is no preprocessor in Triton-IR, hence parametric shape values must be resolved before programs are generated. In our case, this is done by Triton-JIT's auto-tuner.

Instructions

Triton-IR introduces a set of *reblocking* instructions whose purpose is to support broadcasting semantics as described in Section 5.1.2. The reshape instruction creates a block of the specified shape using the data from its input argument. This is particularly useful to re-interpret variables as higher-dimensional arrays by padding their input shapes with ones in preparation for broadcasting. The broadcast instruction creates a block of the specified shapes by replicating its input argument as many times as necessary along dimensions of size 1 – as shown in Figure 5.2.



(a) $[3 \times 1]$ input **(b)** $[1 \times 3]$ input

Figure 5.2: The broadcast <3,3> instruction

Usual scalar instructions (cmp, getelementptr, add, load...) were preserved and extended to signify element-wise operations when applicable. Finally, Triton-IR also exposes specialized arithmetic instructions for transpositions (trans) and matrix multiplications (dot).

5.2.3 Block-Level Control-Flow

In Triton-IR, operations on block variables are atomic: they execute either in full or not at all. As a result, traditional control flow structures (e.g., conditional, loops) are not applicable to individual block elements. This is problematic, since a program may need to e.g., partially guard blocked loads against memory access violations.

This issue could be resolved through the use of the Predicated SSA (PSSA) form [64] and ψ-functions [65]. This would require the addition of two instruction classes (see Listing 5.5) to Triton-IR: cmpp and phi. The former is similar to the usual comparison (cmp) instruction, but returns two opposite predicates instead of one. Conversely, the psi instruction merges instructions from different streams of predicated instructions created using cmpp.

```
;pt[i,j], pf[i,j] = (true, false) if x[i,j] < 5
;pt[i,j], pf[i,j] = (false, true) if x[i,j] >= 5
%pt, %pf = icmpp slt %x, 5
@%pt %x1 = add %y, 1
@%pf %x2 = sub %y, 1
; merge values from different predicates
%x = psi i32<8,8> [%pt, %x1], [%pf, %x2]
%z = mul i32<8,8> %x, 2
```

Listing 5.5: The Predicated SSA (PSSA) form.

This machinery creates a lot of unnecessary complexity for GPUs, where the benefits of PSSA are close to none as divergent program paths within warps are serialized anyway. Therefore, recent versions of Triton handle intra-block control flow in a much simpler way, using conditional instructions such as select, masked_load and masked_store (see Listing 5.6).

```
; For all indices [idx], return cond[idx] ? true_value[idx] : false_value[idx];
select       TYPE<TS1, ... , TSN> cond, true_value, false_value;
; For all indices [idx], return cond[idx] ? *true_addr[idx] : false_value[idx];
masked_load  TYPE<TS1, ... , TSN> cond, true_addr, false_value;
; For all indices [idx], execute *true_addr[idx] = true_value[idx] if cond[idx]
masked_store TYPE<TS1, ... , TSN> cond, true_addr, true_value;
```

Listing 5.6: Intra-Block Control Flow in Triton.

5.3 Triton-JIT

The existence of block-level data-flow information in Triton-IR enables new program optimizations that are out of the reach of traditional compiler backends. In this section, we discuss how the block-based structure of iteration spaces induced by Triton-IR programs can be exploited to develop novel techniques for simplifying computations (Section 5.3.1), parallelizing computations (Section 5.3.2), parallelizing memory accesses (Section 5.3.3) and maximizing data-reuse (Section 5.3.4).

5.3.1 Simplifying Computations

Peephole optimization [66] is a well-known instruction selection technique that replaces small sequences of instructions (”peepholes”) with functionally equivalent sequences that require less clock cycles to execute. Table 5.1 shows a few examples of peephole optimizations commonly done in scalar programs.

Before	After
x = add x, x;	x = ashl x, 1;

y = exp x;
z = log y;
	z = x;

x = x + y;
x = x - y;
	nop;

Table 5.1: Traditional Peephole Optimizations

The existence of block-level data-flow information in Triton-IR exposes new algebraic identities that can be leveraged to improve existing peephole optimizers. Table 5.2 shows two such examples. First, chains of transpositions can be simplified using the identity

$X = (X^T)^T$. Second, chains of reductions can be collapsed into a single reduction on a ”flattened” array, leading to more regular (i.e., one-dimensional) shared memory access patterns that require less inter-thread synchronization.

Before	After

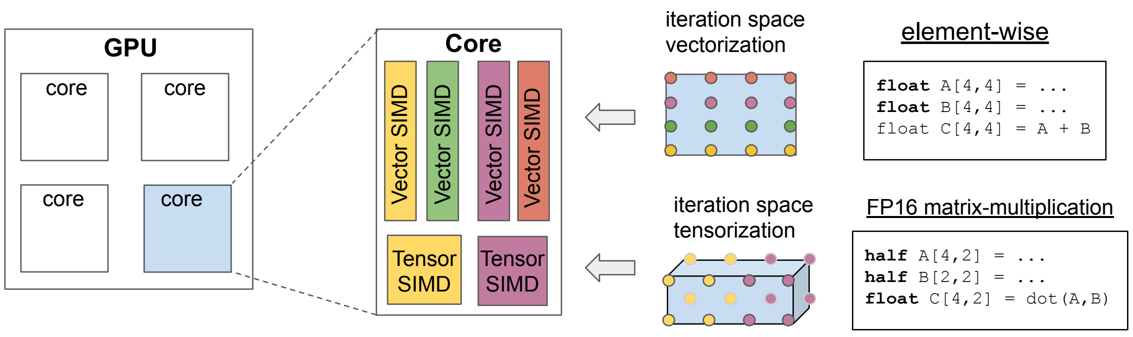
y = trans x;
z = trans y;
	z = x;

y = sum x, 0;
z = sum y, 0;
	y = reshape x, -1;
	z = sum y, 0;

Table 5.2: Block-Based Peephole Optimizations

5.3.2 Parallelizing Computations

Triton programs are single-threaded. This makes them easier to write, maintain, optimize and debug, but also begs for the existence of automatic parallelization mechanisms capable of generating efficient multi-threaded GPU code from high-level specifications of blocked algorithms.



```
float A[4,4] = ...
float B[4,4] = ...
float C[4,4] = A + B
```

```
half A[4,2] = ...
half B[2,2] = ...
float C[4,2] = dot(A,B)
```

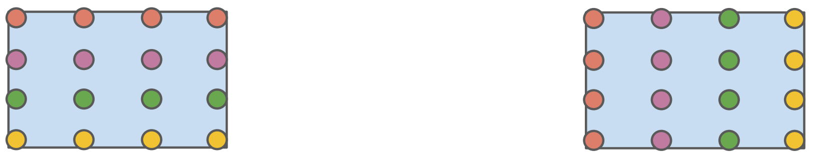
Figure 5.3: Sub-blocking in the Triton-IR machine model.

In Triton, each operation implicitly defines a block of the algorithm’s iteration space (i.e., an *iteration block*) that we aim to schedule efficiently on the SIMD units of a given GPU core. To this end, we propose a *sub-blocking* technique that divides each iteration block into a collection of fragments, each of is stored in registers/SRAM and computed using different SIMD units. Importantly, the decomposition of an iteration block into fragments may vary from operator to operator. For example, we vectorize iteration spaces corresponding to element-wise operators, and tensorize those corresponding to FP16 block-level matrix-multiplications. This strategy is shown in Figure 5.3, where we use a sub-block shape of 1×4 for element-wise operation and $2 \times 2 \times 2$ for FP16 matrix multiplications.

Note that this is a generalization of the sub-blocking mechanism used for context-aware auto-tuning (Section 4.1), hence the number of fragments per iteration block can be automatically optimized so as to increase resource utilization. As we will see below, their shape can also be optimized so as to increase memory efficiency.

5.3.3 Parallelizing Memory Accesses

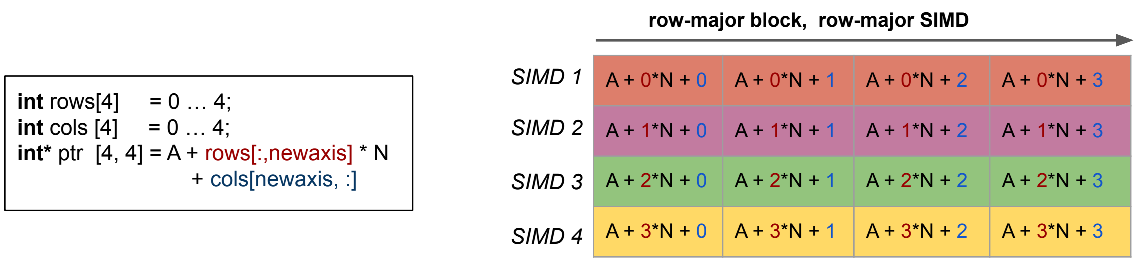
The above sub-blocking mechanism does not define the order in which different sub-blocks should be scheduled (i.e., row-major vs. column-major, see Figure 5.4). This can have profound consequences as modern DRAM controllers are designed to be accessed in burst mode, reading from and writing to global memory in large batches. For this reason, it is important for memory accesses to be *coalesced*, meaning that adjacent threads in the same SIMD unit should access adjacent memory locations.



(a) row-major **(b)** column-major

Figure 5.4: Iteration sub-blocks can be scheduled in different orders.

Fortunately, as shown in Figure 5.5, the order in which memory accesses should be partitioned between SIMD units can sometimes be determined statically. This is done using a *contiguity* analysis pass that determine where memory accesses are contiguous, and partition work accordingly.



```
int rows[4]    = 0 … 4;
int cols [4]    = 0 … 4;
int* ptr  [4, 4] = A + rows[:,newaxis] * N
                         + cols[newaxis, :]
```

SIMD 1	A + 0*N + 0	A + 0*N + 1	A + 0*N + 2	A + 0*N + 3
SIMD 2	A + 1*N + 0	A + 1*N + 1	A + 1*N + 2	A + 1*N + 3
SIMD 3	A + 2*N + 0	A + 2*N + 1	A + 2*N + 2	A + 2*N + 3
SIMD 4	A + 3*N + 0	A + 3*N + 1	A + 3*N + 2	A + 3*N + 3

Figure 5.5: Automatic memory coalescing on GPUs. It is possible to determine statically that memory accesses are row-major, and order warps (shown in different colors) accordingly.

5.3.4 Maximizing Data Reuse

Shared Memory Allocation

DRAM memory accesses are expensive. It is therefore necessary to amortize this cost by re-using data as much as possible after each load. Contrary to CUDA and OpenCL where this kind of memory management is left to the discretion of programmers, Triton-JIT can detect potential for data reuse in the input program, and store data in fast shared memory when this is deemed beneficial. In practice, this means that a block of data is stored to shared memory whenever it is used as an operand in an arithmetically intense operation (e.g., matrix multiplication). More formally, the arithmetic intensity of each block-level operation v can be approximated (ignoring L2 cache effects) as

$$\alpha(v) = \text{comp}(v) / \sum_{p \in \text{pred}(v)} \text{mem}(p)$$

where comp(v) and mem(p) respectively denote the computation requirements of the blocked operation v and the amount of memory transferred from DRAM by its predecessors p.

It is also possible to determine *where*, in shared memory, each data block should reside (when applicable). This can be done, as illustrated in Figure 5.6, by calculating the *live range* of each variable that ought to reside there, as per the above three rules. A linear-time static storage allocation algorithm [67] can then be used to assign a portion of the

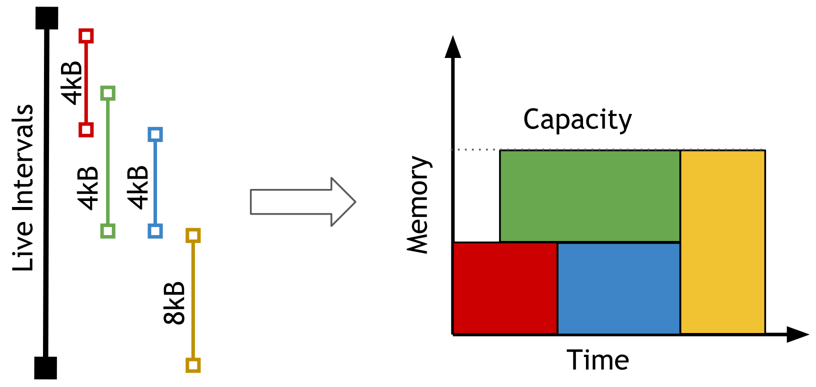


Figure 5.6: Shared memory allocation.

shared memory to each live range. Though this heuristic algorithm may not find the best solution in every case, we found it to be sufficient for the neural network workloads considered in this dissertation.

We recall that liveness analysis can be done in polynomial time by using iterative data-flow analysis (see Algorithm 2 in Chapter 2) where the GEN(v) and KILL(v) set denote respectively the set of variables that are used in v before any assignment, and the set of variables that are assigned a new value in v.

Shared Memory Synchronization

Reads from and writes to shared memory are asynchronous on GPUs. This helps with latency hiding, but also means that programs which do not manage shared memory properly are prone to undefined behavior. To preserve functional correctness in our program, we need our compiler to automatically inserts *barriers* at the appropriate place in the generated GPU source code. Of course, having too many barriers may cause unnecessary synchronization overhead, which is bound to hurt performance.

This can be done using data-flow analysis, by maintaining a set of unsynchronized memory buffers. Every time unsafe behavior is detected, a barrier is inserted and this set is emptied. Read-after-writes (RAW) and write-after-read (WAR) hazards can be detected using forward data-flow analysis with the following data-flow equations:

$$\text{IN}^{(RAW)}(s) = \bigcup_{p \in \text{pred}(s)} \text{OUT}^{(RAW)}(p)$$

$$\text{IN}^{(WAR)}(s) = \bigcup_{p \in \text{pred}(s)}$$

$$\text{OUT}^{(WAR)}(p)$$

$$\text{OUT}^{(RAW)}(s) = \begin{cases} \emptyset & \textbf{if } \text{IN}^{(RAW)}(s) \cap \text{read}(s) \neq \emptyset \text{ (barrier)} \\ \text{IN}^{(RAW)}(s) \cup \text{write}(s) & \textbf{otherwise} \end{cases}$$

$$\text{OUT}^{(WAR)}(s) = \begin{cases} \emptyset & \textbf{if } \text{IN}^{(WAR)}(s) \cap \text{write}(s) \neq \emptyset \text{ (barrier)} \\ \text{IN}^{(WAR)}(s) \cup \text{read}(s) & \textbf{otherwise} \end{cases}$$

where read(s) and write(s) respectively denotes the intervals at which shared memory is read from / written to in the statement s.

5.4 Numerical Experiments

In Chapter 4, we saw that context-aware auto-tuning could match (and sometimes exceed) the performance of state-of-the-art CUDA libraries – provided good enough code templates. In this section, we wish to see whether or not this observation still holds for templates generated using Triton. To ensure fairness and consistency with the results presented then, we consider the same workloads, use the same environment (Pascal/Volta + cuBLAS 9.2 / cuDNN 7.0) and follow the same experimental protocol. When applicable, we also report the performance of Auto-TVM v0.7, auto-tuning official schedule templates for each problem size considered.

5.4.1 Matrix Multiplication Performance

In this section, we benchmark the performance of the Triton matrix multiplication kernel shown in Listing 5.7. The values of TM, TN and TK are determined by the context-aware auto-tuning method presented in the previous chapter. In-place transpositions are handled by setting the value of STRIDE_AK appropriately. For now, we use TYPE = float; readers interested in the half precision of Triton (TYPE = half) may refer to Section 5.4.3 for an evaluation of Tensor Cores.

```
__global__ void matmul(TYPE * A, TYPE * B, TYPE * C, float alpha,
                       int M, int N, int K, int lda, int ldb, int ldc) {
    // prologue
    int ridx = get_program_id(0);
    int ridy = get_program_id(1);
    int ridz = get_program_id(2);
    K = K / TZ;
    int rm[TM] = ridx * TM + 0 ... TM;
    int rn[TN] = ridy * TN + 0 ... TN;
    int rk[TK] = ridz * TZ + 0 ... TK;
    // pointers to operands
    int offa[TM, TK] = rk[newaxis, :] * STRIDE_AK + rm[:, newaxis] * STRIDE_AM;
    int offb[TK, TN] = rk[:. newaxis] * STRIDE_BK + rn[newaxis, :] * STRIDE_BN;
    TYPE* pa[TM, TK] = A + offa;
    TYPE* pb[TK, TN] = B + offb;
    // prefetches operands
    bool checka[TM, TK] = rk[newaxis, :] < K;
    bool checkb[TK, TN] = rk[:, newaxis] < K;
    TYPE a[TM, TK] = checka ? *pa : 0;
    TYPE b[TK, TN] = checkb ? *pb : 0;
    // reduction loop
    float acc[TM, TN] = 0;
    for(int k = K; k > 0; k -= TK){
      acc += A @ B;
      bool checka[TM, TK] = k > TK;
      bool checkb[TK, TN] = k > TK;
      pa += TK * STRIDE_AK;
      pb += TK * STRIDE_BK;
      a = *?(checka)pa;
      b = *?(checkb)pb;
    }
    acc = acc * alpha;
    TYPE c[TM, TN] = acc;
    // epilogue
    int rxm[TM] = get_program_id(0) * TM + 0 ... TM;
    int rxn[TN] = get_program_id(1) * TN + 0 ... TN;
    int offc[TM, TN] = rxm[:, newaxis] * ldc + rxn[newaxis, :];
    TYPE* pc[TM, TN] = C + offc;
    bool checkc[TM, TN] = (rxm[:, newaxis] < M) && (rxn[newaxis, :] < N);

#if (TZ==1)
    *?(checkc) pc = c;
#else
    // accumulate partial result using spin-locks
```

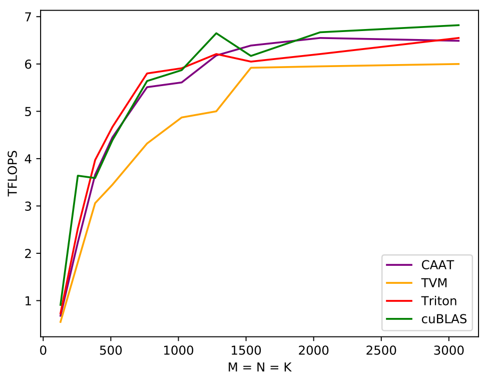
```
      int *plock  = locks + rid;
      int *pcount = plock + get_num_programs(0) * get_num_programs(1);
      for(int repeat = 1; repeat == 1; repeat = atomic_cas(plock, 0, 1));
      int count = *pcount;
      if(count == 0)
        *?(checkc) pc = c;
      else
        *?(checkc) pc = c + *?(checkc)pc;
      atomic_xchg(pcount, (count + 1) % TZ);
      atomic_xchg(plock, 0);
#endif
}
```

Listing 5.7: Matrix Multiplication in Triton-C.

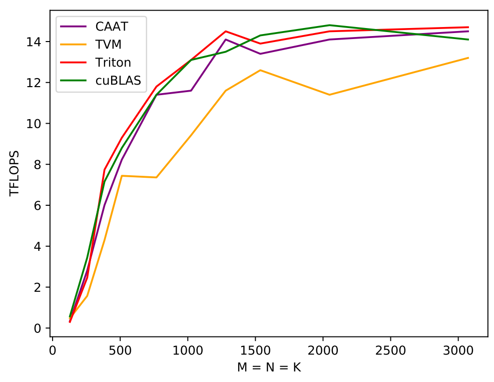
As in Chapter 4, we measure the performance of this code template on (1) square matrices, (2) covariance computation, and (3) batched MLP inference.

Square Matrices

In Figure 5.7, we show the performance of Triton on square matrix multiplication tasks $C = A.B^T$ where $A, B \in \mathbb{R}^{N \times N}$ for $128 \leq N \leq 3072$. As one can see, Triton is on par with cuBLAS accross the board for both GPUs considered. The modest performance loss observed on the GTX 1070 Ti is attributed to compilation artifact (e.g., worse instruction scheduling), since close inspection of the generated assembly revealed abnormally high register pressure.



(a) GTX 1070 Ti



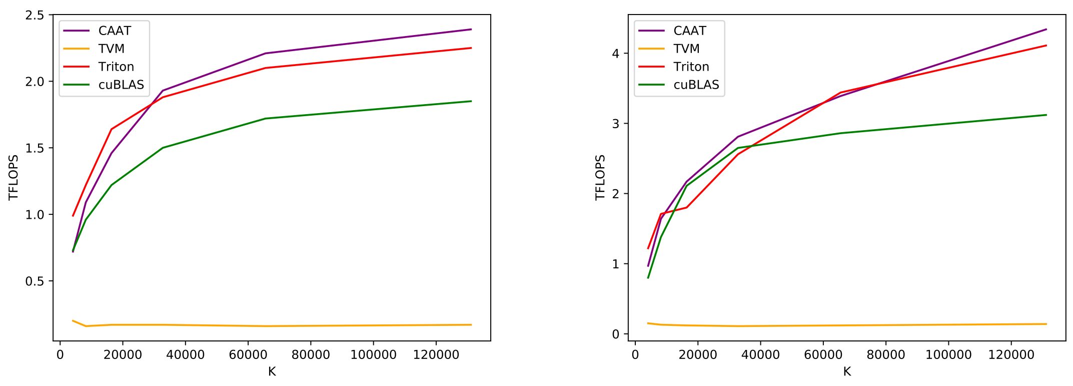
(b) Tesla V100

Figure 5.7: Performance of Triton for square matrix multiplications.

Importantly, Triton outperforms TVM's auto-tuned official matrix-multiplication schedule by a non-negligible margin (up to $> 30\%$). This suggests that, even after auto-tuning, scheduling languages currently fail to achieve peak GPU performance – especially for smaller matrices.

Covariance

Figure 5.8 shows the performance of Triton on covariance matrix computations of the form $C = A.B^T$ where $A \in \mathbb{R}^{M\times K}$ and $B \in \mathbb{R}^{N\times K}$ for $M = N = 64$ and $4096 \leq K \leq 131072$. The absence of reduction-splitting mechanisms compatible with matrix-multiplication in TVM leads to performance an order of magnitude lower than any other system considered. Triton is on par with the PTX code template presented in Chapter 4, suggesting code quality similar to that of pseudo-assembly handwritten by experts.



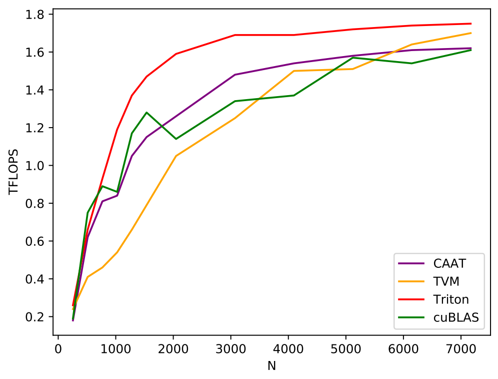
(a) GTX 1070 Ti **(b)** Tesla V100

Figure 5.8: Performance of Triton for covariance matrix computation.

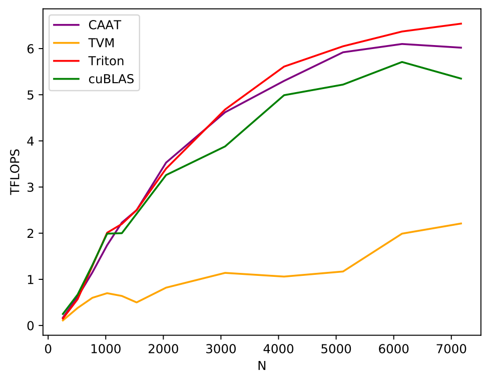
Multi-Layer Perceptron

Figure 5.9 shows the performance of Triton on batched MLP inference tasks of the form $C = A \times B^T$, where $A \in \mathbb{R}^{N\times N}$ and $B \in \mathbb{R}^{16\times N}$, which are omnipresent in batched MLP inference. We note that Triton is about 20-30% faster than CAAT on the GTX 1070 Ti, but on par with it on the Tesla V100. We think that this means that the code generated by Triton for matrix multiplication has higher quality, since the block shapes chosen by CAAT and Triton are the same.

The poor performance of TVM on Volta is surprising, but makes sense considering that the large number of SMs on this GPU cannot be fully occupied for matrix of these shapes without the use of any reduction-splitting mechanism.



(a) GTX 1070 Ti



(b) Tesla V100

Figure 5.9: Performance of Triton for batched MLP inference (N = 16).

5.4.2 Convolutions

To assess the performance of Triton on common convolution tasks, we reconsider the workloads presented in Section 4.5, using the Triton-C implementation of *implicit* matrix multiplication (Figure 4.4) shown in Listing 5.8.

The values of TM, TN and TK are once again determined using context-aware auto-tuning. By contrast, TVM relies on a context-aware auto-tuning approach that searches over *billions* different implementations everytime new input tensor shapes are considered; this necessitates approximate methods that are not guaranteed to find the optimal compute kernels even after running for hours.

```
__global__ void conv(TYPE *A, TYPE *B, TYPE *C, int *ADELTA,
                         float alpha, int M, int N, int K,
                         int pad_h, int pad_w, int stride_h, int stride_w,
                         int lda_z, int lda_ci, int lda_h, int lda_w,
                         int ldb_ci, int ldb_r, int ldb_s, int ldb_co,
                         int ldc_z, int ldc_co, int ldc_p, int ldc_q) {
    // prologue
    int ridx = get_program_id(0);
    int ridy = get_program_id(1);
    int rm[TM] = ridx * TM + 0 ... TM;
    int rn[TN] = ridy * TN + 0 ... TN;
    int rk[TK]  = 0 ... TK;
    // unpack aggregate rows m = (z, p, q)
    int rq[TM]   = rm  % Q;
    int rzp[TM]  = rm  / Q;
    int rp[TM]   = rzp % PP;
    int rz[TM]   = rzp / PP;
    // unpack aggregate reduction k = (ci, r, s)
    int rs   [TK] = rk % S;
    int rcir [TK] = rk / S;
    int rr   [TK] = rcir % R;
    int rci  [TK] = rcir / R;
    // padding / striding
    int rh_0[TM] = rp * stride_h - pad_h;
    int rw_0[TM] = rq * stride_w - pad_w;
    int rh[TM, TK] = rh_0[:, newaxis] + rr[newaxis, :];
    int rw[TM, TK] = rw_0[:, newaxis] + rs[newaxis, :];
    // pointers to lhs
    int offa[TM, TK] = rz  [:, newaxis]   * lda_z  + rci[newaxis, :]   * lda_ci +
                       rh                  * lda_h  + rw                 * 1;
    TYPE* pa[TM, TK] = A + offa;
    int* padelta[TK] = ADELTA + rk;
    // pointers to rhs
    int offb[TK, TN] = rci[:, newaxis] * ldb_ci + rr  [:, newaxis] * ldb_r  +
                       rs  [:, newaxis] * ldb_s  + rn  [newaxis, :] * 1;
    TYPE* pb[TK, TN] = B + offb;
    // prefetches operands
    bool checka[TM, TK] = rm[:, newaxis] < M && rh >= 0 && rh < H && rw >= 0 && rw < W;
    bool checkb[TK, TN] = rn[newaxis, :] < N && rk[:, newaxis] < K;
    TYPE a[TM, TK] = checka ? *pa : 0;
    TYPE b[TK, TN] = checkb ? *pb : 0;
    int total = 0;
    // reduction loop
    float acc[TM, TN] = 0;
    for(int k = K; k > 0; k -= TK){
      acc += a @ b;
      // increment A
      int adelta[TK] = *padelta;
      padelta += TK;
      pa += adelta[newaxis, :];
      // bounds-checking A
      rk += TK;
      rs = rk % S;
      rcir = rk / S;
      rr = rcir % R;
      rh = rh_0[:, newaxis] + rr[newaxis, :];
      rw = rw_0[:, newaxis] + rs[newaxis, :];
      bool checka[TM, TK] = rm[:, newaxis] < M && rh >= 0 && rh < H && rw >= 0 && rw < W;
      // increment B
      pb += TK * ldb_s;
      // bounds-checking B
      bool checkb[TK, TN] = rn[newaxis, :] < N && k > TK;
      a = checka ? *pa : 0;
64       b = *?(checkb)pb;
65     }
66     acc = acc * alpha;
67     TYPE c[TM, TN] = acc;
68     // epilogue
69     rm  = ridx * TM + 0 ... TM;
70     rn  = ridy * TN + 0 ... TN;
71     rq  = rm  % Q;
72     rzp = rm  / Q;
73     rp  = rzp % PP;
74     rz  = rzp / PP;
75     int offc[TM, TN] = rz [:, newaxis] * ldc_z + rn [newaxis, :] * ldc_co +
76                        rp [:, newaxis] * ldc_p + rq [:, newaxis] * 1;
77     TYPE* pc[TM, TN] = C + offc;
78     bool checkc[TM, TN] = rm[:, newaxis] < M && rn[newaxis, :] < N;
79     *?(checkc) pc = c;
80 }
```

Listing 5.8: Implicit matrix multiplication in Triton-C

To reduce the cost of 64-bit pointer arithmetics in the inner reduction loop, we precompute pointer increments for A in ADELTA $\in \mathbb{Z}^{C_i RS}$ as shown in Listing 5.9. Since ADELTA is usually small enough to fit in the GPU's L1 cache, the cost of dereferencing it (line 48-49) is usually smaller than that of the equivalent integer arithmetics.

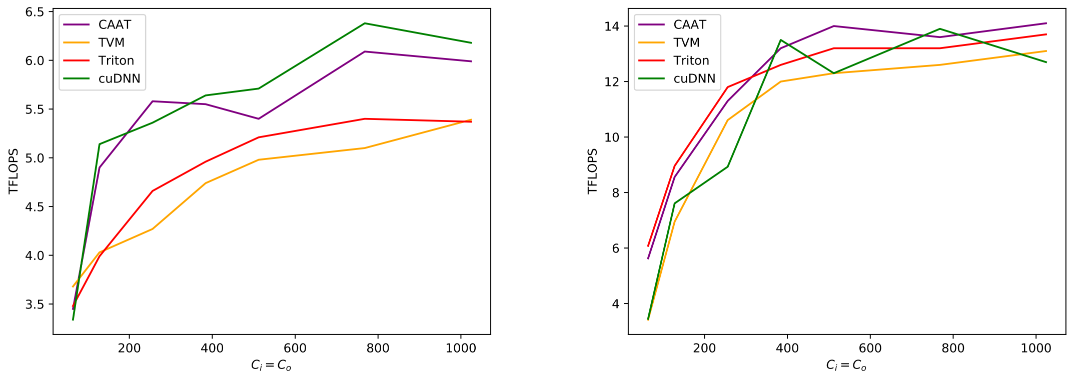
```
for(int i = 0; i < CI*R*S; i++){
  int s = i % S;
  int cr = i / S;
  int r = cr % R;
  int c = cr / R;
  int nexti = i + TK;
  int nexts = nexti % S;
  int nextcr = nexti / S;
  int nextr = nextcr % R;
  int nextc = nextcr / R;
  ADELTA[i] = (nextc - c)*W*H + (nextr - r)*W + (nexts - s);
}
```

Listing 5.9: Pointer increment precomputation.

Fixed Input Resolution

Like in Chapter 4, we consider input images of 56×56 pixels and vary $C_i = C_o$ between 64 and 1024 for inference ($Z = 1$). The resulting performance trends, shown in Figure 5.10, are largely consistent with the matrix multiplication benchmarks shown above. The difference between Triton and our handwritten PTX kernel is small (5% on the Tesla V100,

$10 - 15\%$ on the GTX1070Ti), and both systems outperform cuDNN on the Volta architecture.



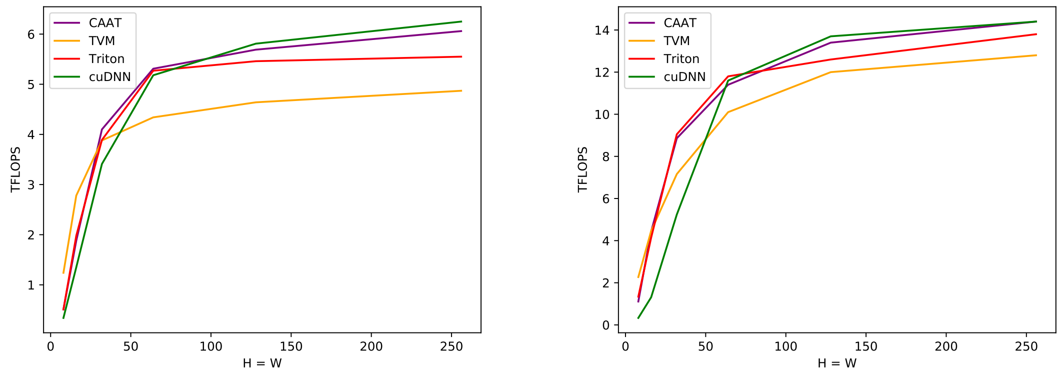
(a) GTX 1070 Ti **(b)** Tesla V100

Figure 5.10: Performance of Triton on fixed-channel-width convolutions.

TVM is consistently slower than Triton – even after hours of auto-tuning – despite relying on schedule templates handwritten by experts. Though this gap is likely to close as new scheduling primitives become available and the TVM compiler improves, it highlights the potential of the new programming paradigm presented in this chapter.

Fixed Channel Width

We now consider the convolution tasks characterized by $Z = 1$ and $C_i = C_o = 256$ as $H = W$ varies between 8 and 256. As shown in Figure 5.11, Triton is once again only $< 10\%$ slower than state-of-the-art handwritten PTX and SASS, cementing the efficacy of the compilation techniques presented in Section 5.3. Once again, TVM is almost always slower than Triton, despite it being highly optimized for the inference workloads considered here [68, 69].

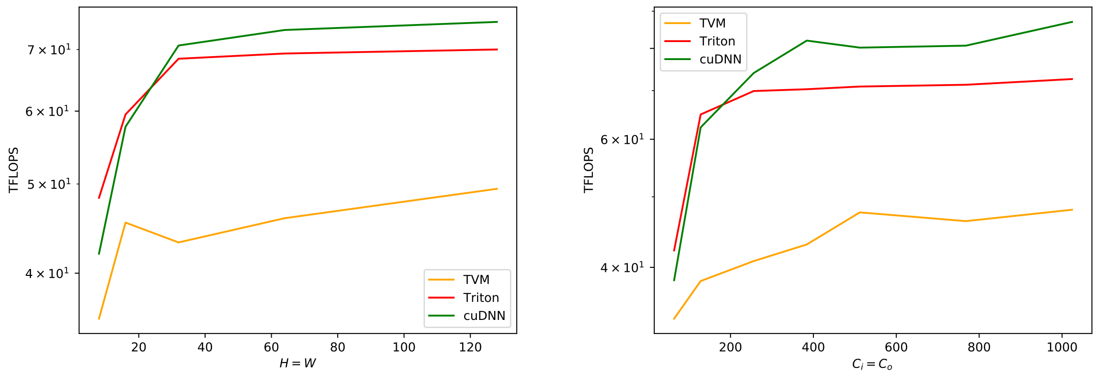


(a) GTX 1070 Ti **(b)** Tesla V100

Figure 5.11: Performance of Triton on fixed-channel convolutions.

5.4.3 Tensor Cores

Specialized hardware intrinsics for Deep Neural Networks have become increasingly popular over the past few years, culminating with the introduction of tensor cores in 2017. As a result, GPUs have become not only more powerful, but also harder to program. Scheduling languages have dealt with this issue by introducing new, *non-portable* primitives to deal with this added complexity, making the task of writing efficient schedules even harder than before.



(a) Fixed channel width **(b)** Fixed input resolution

Figure 5.12: Performance of Triton on convolutions with tensor cores.

By contrast, the usage of tensor cores in Triton can be transparently activated by merely using `#define TYPE half` in Listing 5.7 and Listing 5.8. As shown in Figure 5.12, the resulting GPU kernel is only $10 - 15\%$ slower than cuDNN and $\sim 50\%$ faster than TVM's (nonportable) schedule. For small images $H = W \leq 16$ and narrow layers $C_i = C_o \leq 128$, our mixed-precision implementation of implicit matrix multiplication is even faster than cuDNN's highly optimized assembly – due to the use of smaller block shapes.

5.5 Summary

Motivated by the limitations of existing DSLs for DNNs, this chapter introduced Triton, a language and compiler for blocked algorithms in the ”single-program, multiple-data” execution model. We first presented Triton-C, an alternative to CUDA in which compute kernels are single-threaded and multi-dimensional blocks of data are first class citizen. To ensure compatibility with different frontends and provide a stable environment for block-level program analysis, we then presented Triton-IR, an LLVM-based intermediate representation in which block-level data- and control-flow information is made explicit through the use of block-level instructions. We finally presented Triton-JIT, a just-in-time compiler which leverages the blocked structure of iteration spaces defined by Triton programs to efficiently schedule work on individual GPU cores. A validation of our system on dense neural network workloads showed performance on par with the handwritten PTX code template presented in Chapter 4 – and clearly superior to state-of-the-art DSLs for DNNS.

The main advantage of our approach, however, remains its potentially higher applicability to emerging neural network architectures. More experiments are therefore needed to establish whether or not our proposed block-based paradigm has any benefits beyond enabling moderately faster DSLs for dense linear algebra.

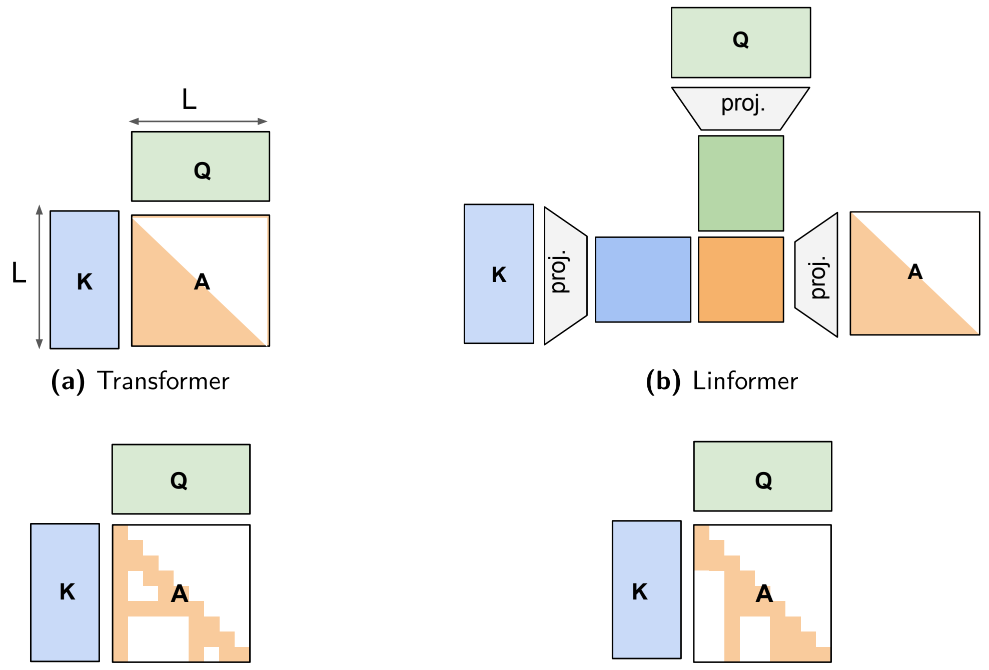
6 Fast Sparse Transformers

Over the past two chapters, we have shown how to write, tune and compile blocked algorithms for dense neural networks on graphics processor. This is encouraging, but does not say much about the general applicability of our proposed block-based programming paradigm to deep learning research: is it useful for anything beyond dense linear algebra? can it be used to explore novel research ideas in the field?

To help answer these questions, this final chapter shows how blocked algorithms may facilitate the development of efficient compute kernels for structured-sparse transformers. After exposing, in Section 6.1, the opportunities and challenges presented by sparse self-attention mechanisms, we describe naive implementations of important basic primitives for block-sparse computations in Section 6.2, which we optimize using *super-blocking* and *static load-balancing* in Section 6.3. We then evaluate, in Section 6.4, the resulting system on the GPT-2 [70] architecture, where we show not only considerable speed-ups over a dense PyTorch baseline, but also close to optimal performance on our hardware.

6.1 Motivations

Transformers are an important class of neural network architectures, applicable to many different problems ranging from natural language processing [71, 72] to computer vision [73] to biology [74]. As we saw in Chapter 2, they are composed of a succession of *attention mechanisms* aimed at transforming a given sequence of tokens (e.g., words) into a higher-level embedding. Longer sequences are typically preferred, as they expose long-term dependencies that increases predictive performance.



(a) Transformer **(b)** Linformer **(c)** Longformer **(d)** Sparse Transformer

Figure 6.1: Faster attention mechanisms in Transformers.

Unfortunately, the cost of forward- and backward- propagation in these models grows quadratically with the length of input sequence (i.e., *attention window*) [19] – both in terms of memory and compute. This limitation has created substantial research interest for novel attention mechanisms exhibiting better scaling properties. Figure 6.1 compares the structure of three such alternatives against that of the original transformer architecture. The **Linformer** [75] leverages the low-rank structure of attention matrices to project keys and values onto a lower-dimensional space where an embedding of the attention scores can be computed more efficiently. The **Longformer** [76] computes the attention scores directly, but only for certain combinations of queries and keys. These combinations are computed both locally using a sliding window (i.e., band-diagonal), and globally for fixed long-term dependencies (i.e., dense rows/columns). The **Sparse Transformer** [48] works similarly, but uses slightly different sparsity patterns for attention scores. Note that, to ensure compatibility with tensor cores, sparse attention mechanisms are generally *block-sparse*, meaning that non-zeros values in the attention matrix are clustered in blocks of 8×8 or 16×16 elements.

Although Linformers can be implemented efficiently using dense BLAS primitives, Longformers and Sparse Transformers require the construction of custom compute kernels for sparse matrix computations. Unfortunately, state-of-the-art implementations of these primitives are generally written in CUDA-C, thereby requiring programmers to carefully schedule computations not only *within* attention blocks but also *across them*. This results in large codebases that are challenging to build, maintain, distribute and upgrade. In the remainder of this chapter, we propose to implement these primitives using Triton-C instead, which allows attention blocks to be scheduled automatically using block-level data-flow analysis (see Chapter 5). To address the issue of efficiently scheduling computations *across* different attention blocks, we will present *super-blocking* and *static load-balancing* optimization techniques aimed at improving data-reuse and device utilization in the aforementioned Triton-C programs.

6.2 Sparse Self-Attention

Sparse self-attention mechanism are composed of three major components shown in Figure 6.2. First, sparse attention scores are computed by multiplying a set of key vectors with a set of query vectors. In the litterature, this operation is generally known as a sampled dense-dense matrix multiplication (SDMM [77]). Second, sparse attention scores are post-processed and normalized using a sparse softmax operation. Third, the normalized sparse attention scores are utilized, i.e., multiplied with a value matrix to form a new embedding of the input sequence. In this section, we will present a naive blocked algorithm for each of these operations.

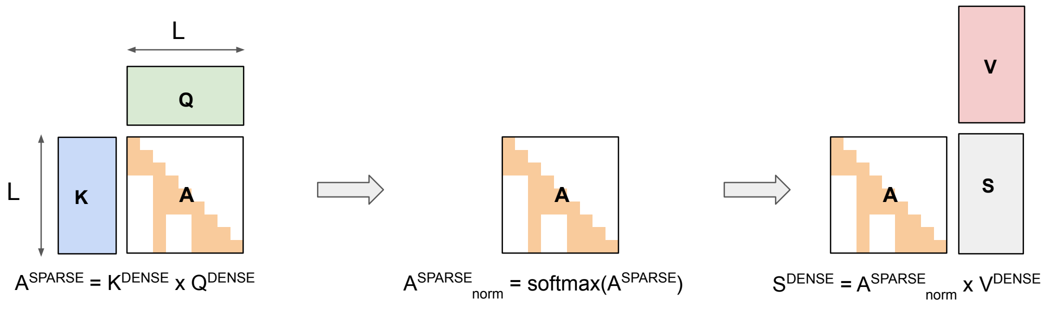

L
Q
L
K
A
A
V
A
S
A^SPARSE = K^DENSE x Q^DENSE
A^SPARSE_norm = softmax(A^SPARSE)
S^DENSE = A^SPARSE_norm x V^DENSE

Figure 6.2: Flowchart of a sparse attention mechanism.

We will specifically consider attention tensors of the form

$$A \in \mathbb{R}^{Z \times H \times L \times L}$$

where Z, H and L respectively denote the number of batches being processed, the number of attention heads in the model and the length of the input sequence. The sparsity layout S of this attention tensor is composed of non-zero blocks of $b \times b$ elements, whose positions are shared between all batches, but may vary from head to head, i.e., $S \in \mathbb{R}^{H \times W/b \times W/b}$. In this section we assume a Coordinate (COO) storage format for the attention tensors (see Figure 6.3).

6.2.1 Computing Attention Scores

Attention scores in sparse transformers can be computed using a subroutine known as Sampled Dense-Dense Matrix Multiplication (SDDMM). There, blocks of the attention

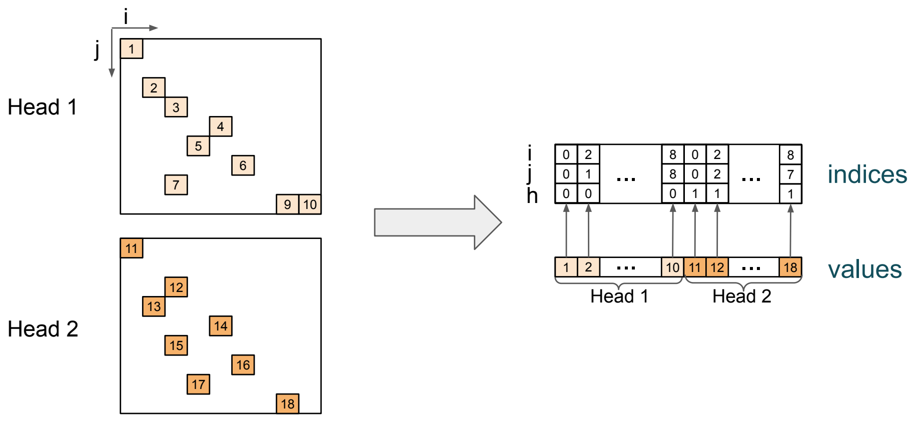


Figure 6.3: The sparse attention tensor is stored using a Coordinate (COO) storage format.

tensors are computed in parallel by different program instances, resulting in as many program instances as there are non-zero blocks in the sparse attention layout. Each such program can be written in Triton-C, using a variant of the matrix-multiplication implementation shown in the previous chapter (Listing 5.7). In this variant, the ranges of rows/-columns in A and B (i.e., rm and rn) are determined by the layout of the output tensor rather than coordinate of the program in the launch grid. These changes can be implemented by modifying the prologue (Line 3-10) of Listing 5.7 as follows.

```
// non-zero block index
int ridx = get_program_id(0);
// batch index
int ridy = get_program_id(1);
// reduction splitting index
int ridz = get_program_id(2);
// coordinates of the block
int* coo = layout + ridx*3;
int  m   = *(coo + 0);
int  n   = *(coo + 1);
int  h   = *(coo + 2);
int rm[TM] = m * TM + 0 ... TM;
int rn[TN] = n * TN + 0 ... TN;
// reduction splitting as before
K = K / TZ;
int rk[TK] = ridz * TZ + 0 ... TK;
```

Listing 6.1: Triton-C prologue for an SDDM with block size BLOCK.

where TM = TN = BLOCK. Note that our implementation of this algorithm also uses reduction-splitting for increased performance in deep reduction loops.

As shown in Figure 6.4, this naive implementation is likely to limit data-reuse (for small block sizes) because it replicates rows and columns of K and Q must be across different cores. We will see in Section 6.3 how this issue may be resolved through the use of *super-blocking*.

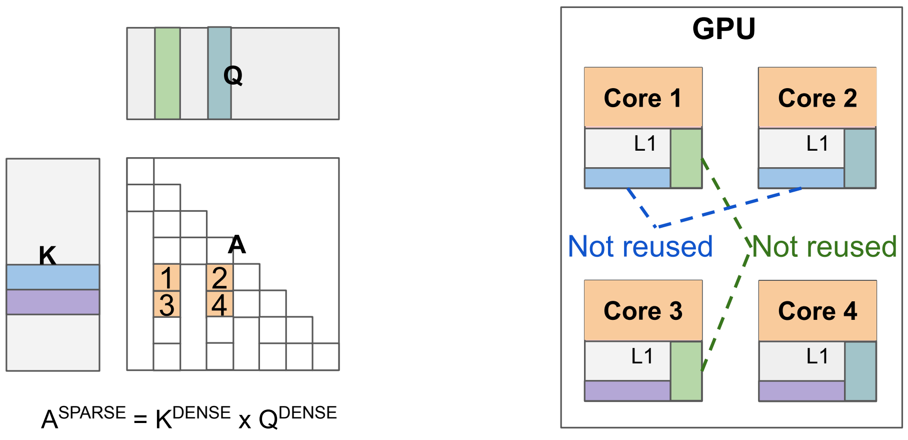


Figure 6.4: Naive implementation of SDDMM limits data-reuse

6.2.2 Normalizing Attention Scores

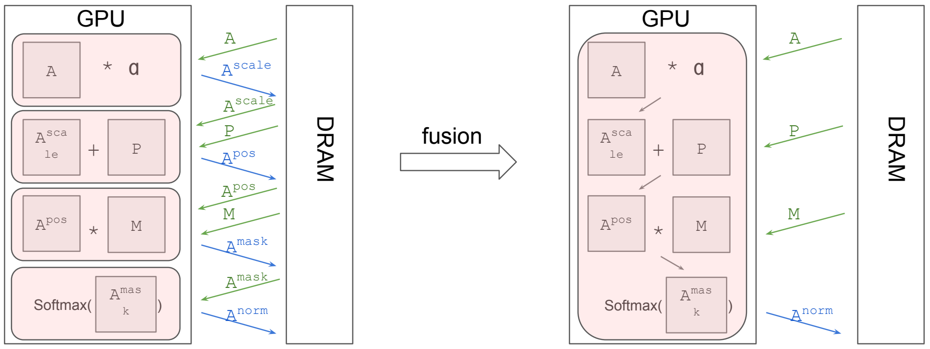
Once the attention scores $A \in \mathbb{R}^{Z\times H\times L\times L}$ have been computed using the above SDDMM subroutine, they are generally re-scaled ($\alpha \in \mathbb{R}$), masked ($M \in \mathbb{R}^{L\times L}$) and added to relative position embeddings [78] ($P \in \mathbb{R}^{Z\times H\times L\times L}$) to form a new tensor $\tilde{A}$ such that

$$\tilde{A}_{z,h,l_i,l_j} = (\alpha A_{z,h,l_i,l_j} + P_{z,h,l_i,l_j})M_{l_i,l_j} \quad \forall z \leq Z, \quad \forall (h, l_i, l_j) \in \mathrm{nnz}(A)$$

This tensor is then normalized using the following softmax function to yield the final, normalized attention scores $\hat{A}$:

$$\hat{A} = \mathrm{softmax}(\tilde{A}) = \frac{\exp(\tilde{A}_{z,h,l_i,l_j})}{\sum_{\hat{l_j}} \exp(\tilde{A}_{z,h,l_i,\hat{l_j}})} \quad \forall z \leq Z, \quad \forall (h, l_i, l_j) \in \mathrm{nnz}(A)$$

Implementing all these operations naively, using a separate compute kernel for each binary operator, results in many unecessary memory transfers to/from temporary buffers in DRAM, as shown in Figure 6.5 for an attention density factor ρ.



(a) Each operation is done in a separate kernel, requiring $8\rho ZHL^2 + ZHL^2 + L^2$ memory transfers from DRAM

(b) All operations are done in the same kernel, requiring $2\rho ZHL^2 + ZHL^2 + L^2$ transfers from DRAM.

Figure 6.5: Memory cost of (a) naive and (b) fused attention normalization. Shaded red regions correspond to compute kernels.

By contrast, our ”fused” implementation implements all the operations in the same compute kernel, which makes it possible to store temporary tensors in registers rather than DRAM. The implementation of this algorithm in Triton-C is shown below. Note that the softmax computation is always done in float regardless of the data-type of the attention matrix. To further improve numerical stability, we also substract to each row its maximum before exponentiating and normalizing it (line 45)

```
1  __global__ void softmax(TYPE *X, TYPE *P, TYPE *M,  int *LUT,
2                         float scale, int num_blocks, long stride_zx, int stride_zm,
3                         long stride_zp, int stride_hp,  int stride_sp, int stride_zp){
4    int pidhm = get_program_id(0);
5    int pidz  = get_program_id(1);
6    // create index ranges
7    int rxm      = pidhm % BLOCK;
8    int rbm      = pidhm / BLOCK;
9    int rxn[TN]  = (0 ... TN) % BLOCK;
10   int rbn[TN]  = (0 ... TN) / BLOCK;
11   // extract information from look-up table
12   int* header = LUT + rbm * 2;
13   int size    = *(header + 0);
14   int offset  = *(header + 1);
15   // bounds checking
16   bool check[TN] = rbn < size;
17   int   rbmn[TN] = check ? rbn : size - 1;
18   // block id and column id
19   long blockid  [TN]  = *(LUT + offset + rbmn*4 + 0);
20   long columnid[TN]   = *(LUT + offset + rbmn*4 + 1);
21   long rowid    [TN]  = *(LUT + offset + rbmn*4 + 2);
```

```
  long headid   [TN]   = *(LUT + offset + rbmn*4 + 3);
  // load input
  TYPE* px[TN]   = X + pidz * stride_zx
                     + blockid * BLOCK * BLOCK
                     + rxm * BLOCK
                     + rxn;
  TYPE x[TN] =  check ? *px : -INFINITY;
  // load relative position embedding
  TYPE* pp[TN] = p + pidz * stride_zp
                   + headid * stride_hp
                   + (rxm + rowid * BLOCK) * stride_sp
                   +  rxn + columnid * BLOCK;
  TYPE p[TN] = check ? *pp : 0;
  // load attention mask
  TYPE* pm[TN] = M + columnid * BLOCK
                   + (rxm + rowid * BLOCK) * stride_zm
                   + rxn;
  TYPE m[TN] = check ? *pm : -INFINITY;
  m = (m == 0) ? -INFINITY : 0;
  // apply scale, relative position embeddings and attention mask
  x = x * scale + p + m;
  // compute softmax in float32
  float Fx[TN] = x;
  float Fy[TN] = exp(Fx - Fx[max]);
  float Fysum = (check ? Fy : 0)[+];
  // write-back
  *?(check)px = Fy / Fysum;
}
```

Listing 6.2: Triton-C implementation of scaled, masked softmax with relative position embeddings.

6.2.3 Utilizing Attention Scores

These normalized attention scores $\hat{A} \in \mathbb{R}^{Z \times H \times L \times L}$ are then multiplied with a E-dimensional embedding of the mechanism's *values* $V \in \mathbb{R}^{Z \times H \times L \times E}$ to form a *dense* output sequence embedding $S \in \mathbb{R}^{Z \times H \times L \times E}$.

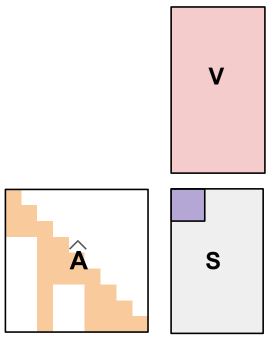


Figure 6.6: Naive implementation of $S = \hat{A} \times V$.

The corresponding batched sparse-dense matrix multiplication $S = \hat{A} \times V$ can be implemented efficiently in Triton-C by computing different blocks of the output results in different programs (see Figure 6.6). Note that normalized attention scores are not only sparse but also lower diagonal in the case of auto-regressive (e.g., GPT) models. This means that different rows of the output matrix are computed using sometimes vastly different reduction sizes, which can lead to load-balancing issues. This will be addressed in the next section of this chapter.

6.3 Optimizations

6.3.1 Super-Blocking

The naive implementation of SDDMM presented above works by computing different blocks of attention scores in parallel. This limits data-reuse and can therefore hurt performance, especially for small block sizes (e.g., 16×16).

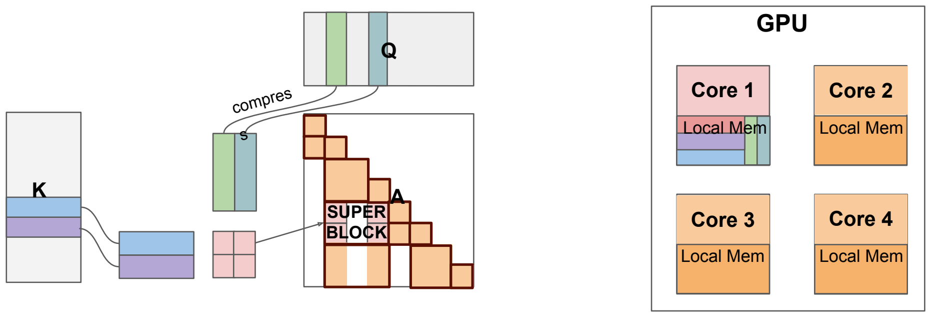


Figure 6.7: Different blocks of attention scores can be computed in parallel to increase data-reuse (i.e., super-blocking).

To mitigate this issue, we present *super-blocking*. There, blocks of attention scores are arranged into super-blocks that can be computed using larger matrix multiplications which promote data-reuse on both operands, as shown in Figure 6.7. Note that the maximum size of super-blocks is limited by hardware resource constraints: large super-blocks may not fit in shared memory and/or registers, hence we limit their size to 128×128 elements (or 8×8 blocks of 16×16 elements each).

Algorithm 3: Super-Blocking

```
Input: Attention Scores $A$;   Partial Results $Q$;   Superblock size $b$
Output: List of superblocks $L$
// Iterate over data-set, with labels if supervised
for $i$ in $[0, W]$ do
    for $j$ in $[0, W]$ do
        // Row index of first nonzero in $A$ above $A[i, j]$
        $i_0 \leftarrow$ first_nnz_above$(A, i, j)$;
        // Column index of first nonzero in $A$ left of $A[i, j]$
        $jj \leftarrow$ first_nnz_left_of$(A, i, j)$;
        // Width is 1 if blocks cannot be merged
        // (i.e., there's a 1 in the "middle"
        $can\_merge \leftarrow$ True;
        for $ii$ in $[i_0, i]$ do
            if first_nnz_left_of$(A, ii, j) > j_0$ then
                $can\_merge \leftarrow False$
        for $jj$ in $[j_0, j]$ do
            if first_nnz_above$(A, i, jj) > i_0$ then
                $can\_merge \leftarrow False$
        // Compute $Q[i, j]$
        if $can\_merge$ then
            $Q[i, j] = \min(Q[i_0, j_0], Q[i_0, j], Q[i, j_0]) + 1$
        else
            $Q[i, j] = 1$
        // Greedy approach: super-blocks that are sufficiently big are retained
        // as soon as they are encountered
        if $Q[i,j] = b$ then
            // Create super-block
            $superblock \leftarrow []$;
            for $ii$ in $[i_0, i]$ do
                for $jj$ in $[j_0, j]$ do
                    if $A[ii, jj] = 1$ then
                        append$(superblock, (ii, jj))$;
            // Append super-block
            append$(L, superblock)$
```

To determine which blocks can be grouped together into super-blocks while limiting the amount of wasted computations, we use dynamic programming. Let us assume, without loss of generality, that $Z = H = 1$ and define a function $q(i, j)$ as:

$$\forall i, j \in [0, W]^2 \qquad q(i, j) = \text{size of largest super-block at position (i,j)}$$

The value of $q(i, j)$ can be computed from the value of its closest nonzero neighbours $q(i_0, j_0)$, $q(i_0, j)$ and $q(i, j_0)$ as shown in Algorithm 3. To determine *which* super-blocks should be retained and in which order they should, our super-blocking algorithm uses a greedy approach which selects super-blocks as soon as they are encountered. This procedure is ran multiple times on superblock of decreasing sizes (e.g., 128, 64, 32, 16) so as to promote larger super-blocks.

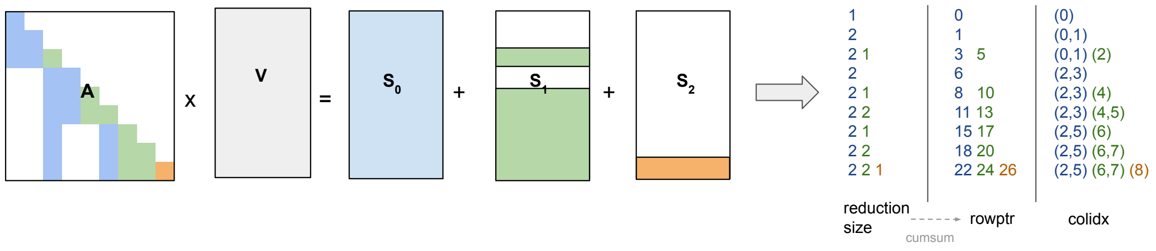
Sequence Length	Naive	Super-Blocking
512	**8.10**	3.80
1024	12.5	**13.4**
2048	15.0	**30.5**
4096	15.7	**36.7**

Table 6.1: Performance (in TFLOPS) of blocked/superblocked SDDMM.

Table 6.1 shows the performance of our Triton-C implementation of super-blocking on (87.5% sparse) attention matrices for sparse transformers, using an embedding dimension of 768 (as in GPT-2) and a Tesla V100 GPU with tensor cores. As one can see, this optimization can provide up to 2.3x speed-up over a naive blocked implementation of SDDMM on large sequence lengths. Note that, when the sequence length is small, there may not enough super-blocks to fully occupy the GPU, causing super-blocking to *reduce* the performance of SDDMM.

6.3.2 Static Load-Balancing

Naive implementations of sparse-dense matrix-multiplications are prone to load balancing issues, as different GPU cores may not do the same amount of work when computing a block of the output result S. As shown in Figure 6.8, this can be mitigated by decomposing the computation of S into a sum of partial tensors requiring the reduction of at most k blocks (in Figure 6.8, k = 2). All these partial results can be computed in parallel, and summed together using atomics and semaphores similarly to the reduction-splitting mechanism introduced in Chapter 4.



reduction size	rowptr	colidx
1	0	(0)
2	1	(0,1)
2 1	3 5	(0,1) (2)
2	6	(2,3)
2 1	8 10	(2,3) (4)
2 2	11 13	(2,3) (4,5)
2 1	15 17	(2,5) (6)
2 2	18 20	(2,5) (6,7)
2 2 1	22 24 26	(2,5) (6,7) (8)

Figure 6.8: Static load-balancing in sparse-dense matrix-matrix multiplication.

6.4 Numerical Evaluation

In this section, we measure the end-to-end performance (in ms/batch, including SGD) of our implementation of the sparse transformer against that of a dense baseline, as a function of the sequence length. Importantly, we use gradient checkpointing ([79]) to reduce the memory construction of the dense transformer we considered, which causes extra computations during back-propagation. For a given embedding size E, the number of operations performed by dense attention mechanisms is:

$$
\begin{aligned}
N_{\text{op}}^{\text{dense}} &= 3E^2W \quad \text{query/key/value embedding (forward)} \\
&+ 3E^2W \quad \text{query/key/value embedding (backward)} \\
&+ 2EW^2 \quad \text{attention + multiplication w/ values (forward)} \\
&+ 2EW^2 \quad \text{attention + multiplication w/ values (backward)} \\
&+ 2EW^2 \quad \text{attention + multiplication w/ values (checkpoint)} \\
&= 6E^2W + 6EW^2
\end{aligned}
$$

Similarly, the number of operations performed by our sparse transformer – which does not require check-pointing – is $N_{\text{op}}^{\text{sparse}} = 6E^2W + \rho 4EW^2$, where $\rho = 0.125$ denotes the density factor of the attention mechanism. The theoretical minimum runtime reported in Figure 6.9 is then $t_{\text{min}} = (N_{\text{op}}^{\text{sparse}}/N_{\text{op}}^{\text{dense}})t_{dense}$, where t_{dense} is measured using the cuBLAS-accelerated PyTorch implementation of the dense Transformer.

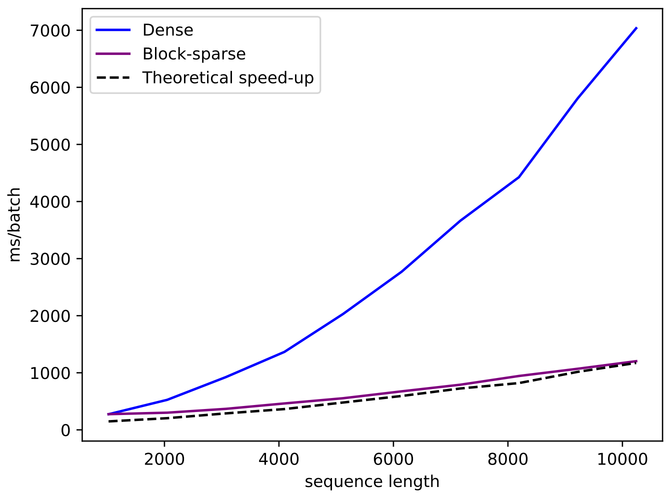


Figure 6.9: Performance of our block-sparse Transformer

As shown in Figure 6.9 for $E = 768$ (i.e., GPT-2), the performance of our Triton-C implementation of the sparse transformers achieves close to peak performance, achieving up to $> 5x$ speed-ups over the aforementioned dense baseline.

6.5 Summary

Transformers are an important but computationally expensive class of neural network architectures. And while block-sparse self-attention mechanisms have recently emerged as a way to reduce the cost of forward- and backward- propagation in these models, their implementation within conventional GPU programming paradigms (e.g., CUDA) remains challenging. For this reason, in this chapter, we have presented a set of blocked algorithms for sparse transformers, along with their implementation in Triton. A major benefit of this strategy is that makes it possible for compilers to automatically schedule computation within attention blocks, using the techniques presented in Chapter 4 and Chapter 5; this leads to acceptable performance even when using naive implementations.

To address the issue of efficiently scheduling computations *across* different attention blocks, we presented *super-blocking* and *static load-balancing*, two optimizations that increase data-reuse and GPU utilization for the sparsity structures typically found in sparse transformers. Finally, we presented a numerical evaluation of our work on large transformers (i.e., GPT-2), and demonstrated close to optimal performance gains over a state-of-the-art dense baseline.

7

Conclusion

Traditionally, GPUs have been programmed using a ”single program, multiple data” execution model in which different *blocks of threads* execute different instances of a given *scalar* program. This approach is great for general-purpose GPU programming – different threads within the same block can do radically different work – but unnecessarily complex for the kind of regular computations commonly encountered in DNNs. To simplify the development of high-performance compute kernels for DNNs, we have proposed a new paradigm in which programs are blocked instead of threads. We have shown how this approach could overcome the limitations of existing DSLs for DNNs through the enforcement of block-structured iteration spaces that are more suitable for sparse computations than the polyhedra or boxes used in prior work.

We showed, in Chapter 4, how the shape of each block in this paradigm may be automatically determined using learning-based methods. We specifically introduced context-aware auto-tuning, a framework for dynamic block size selection using machine learning-based techniques on empirical performance data sampled from simple generative models.

We showed, in Chapter 5, how the schedule of each block in this paradigm may be automatically determined using novel compilation techniques, which we implemented in the Triton language and compiler for blocked algorithm. We most importantly addressed the issue of automatically scheduling block-structured iteration spaces for efficient execution on GPUs using block-level data-flow analysis.

Finally, to validate the applicability of the proposed paradigm to emerging neural network architectures, we described a set of blocked algorithms for sparse attention mechanisms in transformers, along with their implementation in Triton. To increase data-reuse and hardware utilization in the resulting tasks, we presented *super-blocking* and *static load-balancing* techniques, and achieved up to $> 5x$ end-to-end speed-ups over GPT-2 for long sequence lengths.

While these results are encouraging, we believe that they are not sufficient to create a paradigm shift for something as established as GPU programming. We hope that our work will inspire other researchers to further explore the relevance of blocked algorithms for deep learning research, as well as their use in compilers for special-purpose hardware architectures.

References

[1] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in Neural Information Processing Systems 27*, 2014.

[2] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *2016 IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[3] K. Lee, J. Zung, P. Li, V. Jain, and H. S. Seung, "Superhuman accuracy on the SNEMI3D connectomics challenge," *CoRR*, 2017.

[4] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013.

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

[6] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, 2019.

[7] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," *CoRR*, 2018.

[8] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

[9] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, (USA), 2018.

[10] NVIDIA, "The cublas library." https://developer.nvidia.com/cublas.

[11] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *CoRR*, 2014.

[12] NVIDIA, "The tensorrt library." https://developer.nvidia.com/tensorrt.

[13] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[14] M. Auguin and F. Larbey, "Opsila: an advanced simd for numerical analysis and signal processing," in *Ninth EUROMICRO Symposium on Microprocessing and Microprogramming*, 1983.

[15] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, 1998.

[16] D. Lustig, S. Sahasrabuddhe, and O. Giroux, "A formal analysis of the nvidia ptx memory consistency model," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[17] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning Internal Representations by Error Propagation*, p. 318–362. MIT Press, 1986.

[18] 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*, 1989.

[19] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30*, 2017.

[20] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, 1958.

[21] S. Hochreiter, "The vanishing gradient problem during learning recurrent neural nets and problem solutions," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 1998.

[22] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the International Conference on Artificial Intelligence and Statistics*, 2010.

[23] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, 2015.

[24] Y. Kim, "Convolutional neural networks for sentence classification," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

[25] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. Dauphin, "Convolutional sequence to sequence learning," in *Proceedings of the 36th International Conference on Machine Learning*, 2017.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, 2012.

[27] J. Lee, N. Chirkov, E. Ignasheva, Y. Pisarchyk, M. Shieh, F. Riccardi, R. Sarokin, A. Kulik, and M. Grundmann, "On-device neural net inference with mobile gpus," *CoRR*, 2019.

[28] V. Volkov, *Understanding Latency Hiding on GPUs.* PhD thesis, University of California, Berkeley, 2016.

[29] F. E. Allen, "Control flow analysis," *SIGPLAN Notices*, 1970.

[30] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," POPL '95, 1995.

[31] J. Knoop, B. Steffen, and J. Vollmer, "Parallelism for free: Efficient and optimal bitvector analyses for parallel programs," 1996.

[32] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis transformation," in *International Symposium on Code Generation and Optimization*, 2004.

[33] M. Wolfe, "More iteration space tiling," in *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, 1989.

[34] A. Darte, "On the complexity of loop fusion," in *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, 1999.

[35] J. R. Allen and K. Kennedy, "Automatic loop interchange," *ACM SIGPLAN Notices*, 1984.

[36] C. Ancourt and F. Irigoin, "Scanning polyhedra with do loops," in *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1991.

[37] V. Elango, N. Rubin, M. Ravishankar, H. Sandanagobalane, and V. Grover, "Diesel: Dsl for linear algebra and neural net computations on gpus," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL'2018)*, 2018.

[38] C. Lattner and J. Pienaar, "Mlir primer: A compiler infrastructure for the end of moore's law," *CoRR*, 2019.

[39] A. Schrijver, *Theory of Linear and Integer Programming.* John Wiley & Sons, 1986.

[40] T. Grosser, A. Größlinger, and C. Lengauer, "Polly - performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, 2012.

[41] U. K. R. Bondhugula, *Effective Automatic Parallelization and Locality Optimization Using the Polyhedral Model.* PhD thesis, Ohio State University, 2008.

[42] Y. Sato, T. Yuki, and T. Endo, "An autotuning framework for scalable execution of tiled code via iterative polyhedral compilation," *ACM Transactions on Architecture and Code Optimization*, 2019.

[43] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam, "Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies," *International Journal of Parallel Programming*, 2006.

[44] E. W. Dijkstra, "On the role of scientific thought," *Selected Writings on Computing: A personal Perspective*, pp. 60–66, 1982.

[45] L. Zheng and T. Chen, "Optimizing deep learning workloads on arm gpu with tvm," in *Proceedings of the 1st on Reproducible Quality-Efficient Systems Tournament on Co-Designing Pareto-Efficient Deep Learning*, 2018.

[46] J. Ragan-Kelley, *Decoupling Algorithms from the Organization of Computation for High Performance Image Processing.* Ph.d. thesis, Massachusetts Institute of Technology, 2014.

[47] D. Blalock, J. Ortiz, J. Frankle, and J. Guttag, "What is the state of neural network pruning?," in *Proceedings of Machine Learning and Systems*, 2020.

[48] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating long sequences with sparse transformers," *CoRR*, 2019.

[49] C. Malaviya, P. Ferreira, and A. F. T. Martins, "Sparse and constrained attention for neural machine translation," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, 2018.

[50] "Tvm tutorial on tensor core optimization." https://tvm.apache.org/docs/tutorials/optimize/opt_conv_tensorcore.html.

[51] G. J. Chaitin, "Register allocation & spilling via graph coloring," *ACM SIGPLAN Notices*, 1982.

[52] M. Poletto and V. Sarkar, "Linear scan register allocation," *ACM Transactions on Programming Languages and Systems*, 1999.

[53] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimization of software and the atlas project," *Parallel Computing*, 2000.

[54] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on gpus," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010.

[55] Z. Jia, M. Maggioni, B. Staiger, and D. Scarpazza, “Dissecting the nvidia volta gpu architecture via microbenchmarking,” *CoRR*, 2018.

[56] Z. Jia, M. Maggioni, J. Smith, and D. Scarpazza, “Dissecting the nvidia turing t4 gpu via microbenchmarking,” *CoRR*, 2019.

[57] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, “A three-dimensional approach to parallel matrix multiplication,” *IBM Journal of Research and Development*, 1995.

[58] T. Oliphant, “NumPy: A guide to NumPy.” http://www.numpy.org/, 2006–.

[59] J. W. Davidson and S. Jinturkar, “Memory access coalescing: A technique for eliminating redundant memory accesses,” in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, 1994.

[60] R. Nath, S. Tomov, and J. Dongarra, “Accelerating gpu kernels for dense linear algebra,” in *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*, 2011.

[61] M. Martineau, P. Atkinson, and S. McIntosh-Smith, “Benchmarking the nvidia v100 gpu and tensor cores,” in *Proceedings of the 24th International Conference on Parallel and Distributed Computing*, 2019.

[62] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel programming with cuda,” *Queue*, vol. 6, pp. 40–53, Mar. 2008.

[63] M. Braun, S. Buchwald, S. Hack, R. Leissa, C. Mallon, and A. Zwinkau, “Simple and efficient construction of static single assignment form,” in *Proceedings of the 22Nd International Conference on Compiler Construction*, 2013.

[64] L. Carter, B. Simon, B. Calder, L. Carter, and Ferrante, “Predicated static single assignment,” in *Proceedings of the PACT 1999 Conference on Parallel Architectures and Compilation Techniques*, 1999.

[65] A. Stoutchinin and F. de Ferriere, “Efficient static single assignment form for predication,” in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, 2001.

[66] W. M. McKeeman, “Peephole optimization,” *Communications of the ACM*, 1965.

[67] J. Gergov, “Algorithms for compile-time memory optimization,” in *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1999.

[68] Y. Liu, Y. Wang, R. Yu, M. Li, V. Sharma, and Y. Wang, “Optimizing cnn model inference on cpus,” in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, 2019.

[69] L. Wang, Z. Chen, Y. Liu, Y. Wang, L. Zheng, M. Li, and Y. Wang, “A unified optimization approach for CNN model inference on integrated gpus,” *CoRR*, 2019.

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

[71] A. Radford, "Improving language understanding by generative pre-training," in *Proceedings of the 37th International Conference on Machine Learning*, 2018.

[72] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, and D. Amodei, "Language models are few-shot learners," *CoRR*, 2020.

[73] M. Chen, A. Radford, R. Child, J. Wu, H. Jun, P. Dhariwal, D. Luan, and I. Sutskever, "Generative pretraining from pixels," *CoRR*, 2020.

[74] J. Vig, A. Madani, L. R. Varshney, C. Xiong, R. Socher, and N. F. Rajani, "Bertology meets biology: Interpreting attention in protein language models," *bioRxiv*, 2020.

[75] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, "Linformer: Self-attention with linear complexity," *CoRR*, 2020.

[76] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," *CoRR*, 2020.

[77] I. Nisa, A. Sukumaran-Rajam, S. E. Kurt, C. Hong, and P. Sadayappan, "Sampled dense matrix multiplication for high-performance machine learning," in *2018 IEEE 25th International Conference on High Performance Computing*, 2018.

[78] P. Shaw, J. Uszkoreit, and A. Vaswani, "Self-attention with relative position representations," *CoRR*, 2018.

[79] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," *CoRR*, 2016.