Optimization-Based Mappers and Lower Bounds for Tensor Problems

Grace Dinh

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2023-293
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-293.html

December 22, 2023

Optimization-Based Mappers and Lower Bounds for Tensor Problems

By

Grace Dinh

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor James Demmel, Chair
Professor Kurt Keutzer
Professor Satish Rao
Professor Bernd Sturmfels

Fall 2023

Optimization-Based Mappers and Lower Bounds for Tensor Problems

Abstract

Optimization-Based Mappers and Lower Bounds for Tensor Problems

By

Grace Dinh

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Science

University of California, Berkeley

Professor James Demmel, Chair

Obtaining good performance for tensor problems requires computing an architecture-specific *mapping* from the algorithm to the target hardware. Choosing a mapping requires optimizing an objective function over a large, nonconvex space; this objective function represents a performance metric which may be modeled, simulated, or (if possible) measured. Each such objective function incurs different tradeoffs in terms of speed, accuracy, and the strength of results that can be formally proven, and as a result requires its own optimization methods. In this dissertation, we describe optimization approaches for several performance objectives.

- In a simple abstract memory hierarchy model, we derive an unconditional communication lower bound for "projective" tensor operations (which includes most dense linear algebra) and convolutions. We show that this can always be attained (up to a constant factor) by solving a mathematical optimization problem, and that these methods provide significant benefits in practice, halving the communication volume in one case.
- We extend these lower bound techniques - and corresponding algorithms - to handle combinations of communication and convolution for *randomized* matrix multiplication.
- We then describe how to incorporate support for additional mapping decisions and to account for more architectural information by extending our optimization-based approach to support analytical performance models. While these performance models are too complex to prove strong lower bounds for, empirical results show that optimization-based methods can find more performant mappings using significantly less runtime and sample complexity than pure search-based methods.

Contents

Acknowledgments

Many people have supported me throughout my time in graduate school, and I would not be the person I am today without them. This dissertation is dedicated to them.

I'd like to thank my advisor, James Demmel for close to a decade of mentorship, wisdom, support, and always fascinating discussions, both technical and nontechnical.

I've worked with many brilliant people during my time as a graduate student, especially Julian Bellavita, Gilbert Bernstein, Anthony Chen, Younghyun Cho, Armando Fox, Hasan Genc, Amir Gholami, Adrian Castello Gimeno, Mason Haberle, Olga Holtz, Charles Hong, Coleman Hooper, Jenny Huang, Yuka Ikarashi, Josh Kang, Iniyaal Kannan, Tarun Kathuria, Kurt Keutzer, Mahmoud Abo Khamis, Seehoon Kim, Nayiri Krzysztofowicz, Sherry Li, Yang Liu, Hengrui Luo, Hung Ngo, Jonathan Ragan-Kelley, Alex Reinking, Miles Rusch, Satish Rao, Alex Rusciano, Sophia Shao, Harsha Simhadri, Dan Suciu, Luca Trevisan, and Nate Young. Originally, I had thought to divide this list into "collaborator" and "mentor" categories, but in truth I've learned so much from everyone I've worked with that my attempts to draw such a distinction would be arbitrary, and for this they have my deepest gratitude.

I've also had the privilege of having mentors from industry: Randy Huang and Hongbin Zheng at Amazon; Bill Athas and Bryan Raines at Apple; and Andrew Krioukov and Andy Konwinski, who organized an incredibly interesting class at Berkeley on entrepreneurship.

My friends: Amit, Eve, Logan, Paras, SJ, Tarun, Zoe, plus a sizable subset of people above whose names have already been mentioned, and the entire Bay Area paragliding community (especially Glenn, who picked me up after I crashed and drove me to my hotel just in time prepare my talk about Chapter 4). Thank you for keeping me sane all these years.

Last, but not least, I would like to thank my family - my partner, Mae Milano, and my parents, Hoa Dinh and My Nguyen, for their love and support (and for putting up with me during dissertation-writing stress!).

Chapter 1

Introduction

Tensor operations play a central role in many applications, such as machine learning, signal processing, and dense linear algebra. These tensor operations are increasing significantly in size. The number of parameters used in popular large language models, for instance, has ballooned from 1.5 billion (GPT-2, 2019) to 175 billion (GPT-3, 2020) to 530 billion (Megatron-Turing, 2021)[1]. Furthermore, there is increasing demand to run tensor operations efficiently on systems with significant energy and latency constraints, such as edge and Internet of Things (IoT) devices.

As a result, many high-performance software libraries have been developed to run tensor operations performantly and efficiently on CPUs and GPUs, which are ubiquitous and capable of handling a large variety of workloads. The design of these software libraries significantly impacts the performance that can be achieved.

This is even more the case for software targeting *domain-specific accelerators*, which sacrifice features such as branch predictors and automatically managed caches for performance. Such accelerators offload many scheduling decisions, such as movement of data between parts of the memory hierarchy, to the programmer; as a result, their performance is even more affected by the exact manner in which tensor problems are targeted, or *mapped*, onto them.

As a result, it is increasingly important for implementors of performance software to determine how to *map* an algorithm onto a given CPU, GPU, or accelerator. However, the space of possible mappings (*mapspace*) is challenging to search, as the number of choices that comprise a mapping leads to a combinatorial explosion in the number of possible mappings. Furthermore, this space is highly nonconvex and changes significantly with the target architecture; as a result, we wish to develop *general* techniques that can be quickly *specialized* to different target architectures.

This dissertation addresses this problem by developing both *lower bounds* on the cost of different mappings (under some abstract computational models), and *optimization methods* that often attain these lower bounds.

Chapter 2 introduces the necessary background for this work: a characterization of the

[1] `https://huggingface.co/blog/large-language-models`

general types of hardware we consider, as well as a formal definition of the decisions a programmer must make when implementing a performant kernel targeting a particular piece of hardware, which we refer to as the *mapping.* Mappings may be evaluated using several different performance models, ranging from abstract models of computation to detailed architectural simulations. Motivated by previous studies that show that *loop tiling* is the most consequential decision that a programmer must make when designing a mapping, we focus on an abstract communication model, and describe techniques from prior work that describe how to derive theoretical lower bounds for these models.

Unfortunately, these techniques are practically limited by two main shortcomings: first, they are computationally challenging to solve in the general case, and second, they provide bounds that are not tight for problems that are not asymptotically large in all dimensions. We address these challenges by focusing on special cases of particular practical interest: *projective* nested loops (which encompass most dense linear algebra operations, tensor contractions, and n-body pairwise interactions), which we cover in Chapter 3, and convolutions, which we cover in Chapter 4. In both cases, we develop *efficiently computable* lower bounds that apply to problems of all sizes - not just asymptotically large ones - and show how to attain them through an appropriate loop tiling. For convolutions, we show that our approach provides significant practical speedups on a real machine learning accelerator. We extend the techniques used to prove these lower bounds and find algorithms to models incorporating both communication and computation in Chapter 5.

We then describe how these optimization approaches can be extended to performance models beyond the abstract communication model using Bayesian optimization in Chapter 6. While these models are sufficiently complex that proving lower bounds are not possible, we show that optimization techniques inspired by those we use to attain communication lower bounds provide good performance in practice on these more sophisticated lower bounds.

Chapter 2

Background

2.1 Architectures for Tensor Computations

We will begin with a brief overview of several architectures, focusing on their implications for the high-performance implementations of dense linear algebra and machine learning operations. For our purposes, the primary distinguishing features of an architecture will be the the *manner (and amount) of parallelism* it exposes, and the *design of its memory hierarchy.*

CPUs are equipped with several (usually three) levels of associative SRAM caches, usually set-associative, and expose parallelism to programmers in two forms: vector intrinsics and multithreading. Unfortunately, many CPU features add complexity and overhead unnecessary for dense tensor algebra; these include support for both branch prediction, multiple-instruction multiple-data (MIMD) parallelism, prefetching, and hardware-controlled caches. For example, a two-way set-associative cache uses 2.5× as much energy as a manually controlled scratchpad [33, §7.2]. Such overheads greatly limit the amount of parallelism that can be given for a given power and area budget. As a result, the use of CPUs for programs is generally limited to:

- applications (e.g. embedded systems) where cost or power constraints preclude the inclusion of additional processing units
- tasks where the cost of offloading data to an external chips may be expensive, or where the fixed costs outweigh the size-dependent cost. For example, linear algebra operations on small matrices may be individually cheap enough that the fixed overhead required to offload them onto an external chip would more than offset any potential performance gains. This may be alleviated by batching many small operations into a single, larger one, which can be efficiently offloaded onto a GPU; , but this is not always feasible (e.g. when the outputs of some operations are required as input for others).
- computations where data-dependent optimizations can be performed, such as taking advantage of sparsity, or performing approximations that "sparsify" inputs (e.g.

> through the use of locality-sensitive hashing [8]). These optimizations are largely beyond the scope of this work.

GPUs operate on similar principles to CPUs, trading off low latency for significantly higher parallelism (and therefore throughput) and memory bandwidth. Of note, GPUs are typically equipped with their own independent memory (VRAM), and copying data to and from VRAM - an expensive task - is left for the programmer to manage manually. Some modern GPUs have tensor cores, which can perform relatively small ($4 \times 4 \times 4$) tensor operations directly in hardware in the same manner as vector instructions do in CPUs. This, combined with built-in support for many lower precision types popular in machine learning, makes GPUs a common tool for machine learning and other tensor operations.

However, focusing specifically on simple tensor operations such as matrix multiplications and convolutions allows for even simpler architectures to be devised. Without the need for as much flexibility as a CPU or GPU, *tensor accelerators* eschew the complex mix of vectorization, multithreading, and out-of-order execution present in the aforementioned architectures for two-dimensional *systolic arrays* comprised of synchronously operating *processing elements* (PEs). Furthermore, dense tensor algebra's memory access and compute patterns are input-independent and can be determined statically from the problem size, allowing two key memory hierarchy optimizations. First, caches can be replaced with manually-controlled scratchpads, obviating the need for eviction and prefetching logic (and the corresponding area and power costs) and reducing the duplication inherent in an inclusive multi-level memory hierarchy. Furthermore, reduction operations, which are ubiquitous in tensor operations, can be sped up through the use of an *accumulator buffer* whose contents can be incremented and decremented by arbitrary values using one machine instruction. Together, these optimizations allow for significantly higher levels of throughput for a given power and area budget.

Producing optimized implementations of tensor kernels requires taking into account the target architecture, as well as hardware parameters such as the costs (in both latency and energy) for individual memory and compute operations, as well as memory bandwidth limits. Fortunately, as we will discuss in the following section, the process of *mapping* a tensor kernel onto a particular piece of hardware can be decomposed into several constituent *mapping decisions*, many of which can be analyzed - and therefore optimized over - analytically.

2.2 Mapping

As a simple motivating example, let us consider the problem of producing an optimized implementation for the classical three-nested-loop matrix multiplication on a simple simple vectorized CPU with a two-level memory hierarchy; we will generalize this to arbitrary nested loop programs and hardware targets in Section 2.2.1. We start with a simple three-nested loop implementation (Figure 1).

In order to run this matrix multiplication efficiently on a target architecture, we must emit a sequence of hardware instructions controlling not only arithmetic operations but

Algorithm 1: Classic three-nested loop matrix multiplication

```
for $m \in [0, M)$ :
    for $k \in [0, K)$ :
        for $n \in [0, N)$ :
            $C(m, n) += A(m, k) \times B(k, n)$
```

also data movement - either implicitly (for caches with automatic evictions) or explicitly (on accelerators with explicitly managed scratchpad memory). We can think of these instructions as being generated by a series of transformations, or *rewrites,* performed on the nested loop, each of which is associated with a set of choices, the final goal of which is to generate a sequence of machine instructions.

For instance, we may wish to first *tile* the loops in order to increase memory reuse of elements stored on fast memory, reducing the amount of communication required between fast and slow memory, generating Algorithm 2.stop algorithms end of chapter latex memoir If the tile sizes $T_{m,k,n}$ are not divisors of the problem sizes M, K, and N, we must manually handle the iterations at the end that do not perfectly fit into a tile, which we refer to as *tail iterations* or as a *tail case.* Algorithm 2 includes tail cases for n and k axes (we omit the tail case for the m axis for brevity). For the remainder of this section, we will assume for the sake of simplicity that $T_{m,n,k}$ perfectly divide the loop bounds M, N, and K, allowing us to omit handling for tail iterations. Furthermore, note that the instruction to load subsets of A, B, and C into main memory may be expressed either implicitly, if our fast memory is a cache, or explicitly (which would require a paired eviction/flush instruction), if our fast memory is an explicitly managed scratchpad.

In order to further improve reuse and reduce communication, we may elect to *reorder* the loops. For instance, consider the current ordering of the outer three loops of Algorithm 2: m_{out}, k_{out}, n_{out}. As the outermost loops are m_{out} and k_{out}, the subset of A (which is indexed by m and k) loaded into fast memory stays fixed in fast memory for each iteration of the k_{out} loop, while the subsets of B and C change; we call this a A-stationary dataflow. If the costs of accessing A, B, and C are different - for instance, if C is stored in a higher precision to improve floating point accuracy - it may be beneficial to rearrange the loops to m_{out}, n_{out}, k_{out} to create a C-stationary dataflow.

Notice that reordering loops within the inner block, i.e. swapping the positions of m_{in}, k_{in}, and n_{in} loops, will not change the amount of communication between fast and slow memory, as it only affects computations on tiles already resident in fast memory. However, changing the order of loops is essential for parallelizing the code.

Suppose, for instance, that our target machine supports a vector length 4, and we wish to vectorize our code using a fused-multiply add that performs $\vec{a} += s \cdot \vec{b}$, for length-4 vectors $\vec{a}, \vec{b}$ and a scalar s (on Intel AVX, this is equivalent to loading in s with `_mm_broadcast_ss`, loading $\vec{b}$ with `_mm_load_ss`, and performing the computation with `_mm_fmadd_ps`).

If B and C are both stored in row-major format, then the simplest solution is to vectorize

Algorithm 2: Three-nested loop matrix multiplication after tiling

```
for $m_{out} \in [0, \lfloor M/T_m \rfloor)$ :
    for $k_{out} \in [0, \lfloor K/T_k \rfloor)$ :
        for $n_{out} \in [0, \lfloor N/T_n \rfloor)$ :
            Load subsets of $A$, $B$, and $C$ associated with the indices $m, n, k$ spanned
              by the following inner loops into fast memory
            for $m_{in} \in [0, T_m)$ :
                for $k_{in} \in [0, T_k)$ :
                    for $n_{in} \in [0, T_n)$ :
                        $m = T_m m_{\text{out}} + m_{in}$
                        $k = T_k k_{\text{out}} + k_{in}$
                        $n = T_n n_{\text{out}} + n_{in}$
                        $C(m, n) += A(m, k) \times B(k, n)$
        Load subsets of $A$, $B$, and $C$ associated with the indices $m, n, k$ spanned by
          the following inner loops into fast memory
        for $m_{in} \in [0, T_m)$ : // Tail iterations for $n$
            for $k_{in} \in [0, T_k)$ :
                for $n \in [\lfloor N/T_n \rfloor T_n, N)$ :
                    $m = T_m m_{\text{out}} + m_{in}$
                    $k = T_k k_{\text{out}} + k_{in}$
                    $C(m, n) += A(m, k) \times B(k, n)$
    for $n_{out} \in [0, \lfloor N/T_n \rfloor)$ : // Tail iterations for $k$
        Load subsets of $A$, $B$, and $C$ associated with the indices $m, n, k$ spanned by
          the following inner loops into fast memory
        for $m_{in} \in [0, T_m)$ :
            for $k \in [\lfloor K/T_k \rfloor T_k, K)$ :
                for $n_{in} \in [0, T_n)$ :
                    $m = T_m m_{\text{out}} + m_{in}$
                    $n = T_n n_{\text{out}} + n_{in}$
                    $C(m, n) += A(m, k) \times B(k, n)$
    Load subsets of $A$, $B$, and $C$ associated with the indices $m, n, k$ spanned by the
      following inner loops into fast memory
    for $m_{in} \in [0, T_m)$ : // Tail iterations for $n$
        for $k \in [\lfloor K/T_k \rfloor T_k, K)$ :
            for $n \in [\lfloor N/T_n \rfloor T_n, N)$ :
                $m = T_m m_{\text{out}} + m_{in}$
                $C(m, n) += A(m, k) \times B(k, n)$
```

among the n axis. On the other hand, if A and C are stored in column-major format, we would wish to move m_{in} to be our innermost loop before performing vectorization along the

m axis.

In the following pseudocode (Alg. 3), we denote the loop performed by the vector intrinsic using a *spatial-for* loop, so named as each its iterations are *spatially* mapped to physically separate elements of the CPU. This code transformation also induces a tail case in the above code, as the tile size T_n may not be a multiple of 4. The tail may be executed either as scalar code or a masked vector operation; the choice of which one to use depends on both the number of loop iterations contained in the tail and the target hardware.

Algorithm 3: Three-nested loop matrix multiplication after tiling, vectorization

```
for $m_{out} \in [0, M/T_m)$ :
    for $k_{out} \in [0, K/T_k)$ :
        for $n_{out} \in [0, N/T_n)$ :
            Load subsets of $A$, $B$, and $C$ associated the indices $m, n, k$ spanned by
              the following inner loops into fast memory
            for $m_{in} \in [0, T_m)$ :
                for $k_{in} \in [0, T_k)$ :
                    $m = T_m m_{\text{out}} + m_{in}$
                    $k = T_k k_{\text{out}} + k_{in}$
                    for $n_{in} \in [0, \lfloor T_n/4 \rfloor)$ :
                        $n_{reg} = T_n n_{\text{out}} + 4n_{in}$
                        spatial for $n \in [n_{reg}, n_{reg} + 3]$ : // Run as vector op
                            $C(m, n) += A(m, k) \times B(k, n)$
                    for $n \in [4 \lfloor T_n/4 \rfloor, T_n]$ : // tail case
                        $C(m, n) += A(m, k) \times B(k, n)$
```

Each of the preceding steps is associated with different choices: the tile sizes $T_{m,k,n}$, the order of the loops, and the choice of which axes to parallelize. These choices must be made under constraints: for instance, the memory footprints of each inner tile may not exceed the size of fast memory, and we are only permitted to vectorize the innermost loop.

These choices can also significantly affect performance. In Fig. 2.2.1 we plot the energy-delay product (EDP) of 100K randomly chosen mappings for several matrix multiplications (top) and convolutions (bottom) the choices made in mappings can significantly affect performance. Only a small fraction of mappings achieves reasonable performance; the vast majority of mappings are over an order of magnitude more expensive to execute.

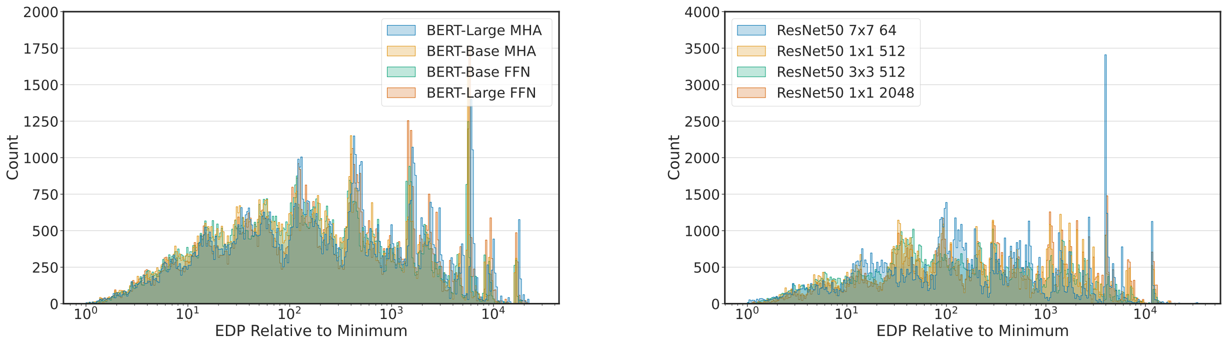


Figure 2.2.1: A histogram of energy-delay product (EDP - left is better) values for 100K randomly sampled valid mappings for BERT (matmul-dominated kernel, left) and Resnet50 (convolution, right), simulated on Timeloop [71]. Note that the x-axis is log-scale. Only a minuscule fraction of mappings, on the far left of each graph, provide acceptable performance for each processor.

The mapping problem applies to many tensor algorithms beyond simple matrix multiplies - many other tensor problems, such as convolutions, as seen in the figure, are similarly dependent on high-quality mappings for good performance. We can represent these programs as general *nested loop programs* operating on multidimensional arrays, as described in Algorithm 4. We will assume in this work that the operations may be performed in any order (i.e. there are no dependencies between different loops, and we are not concerned about floating-point artifacts induced by changing the order of operations).

Algorithm 4: General nested loops

```
data: number of nested loops $d \in \mathbb{Z}$
      loop bounds $L_1, \ldots, L_d \in \mathbb{Z}$
      multidimensional arrays $A_1, \ldots, A_n$, with $A_i$ being $d_i$-dimensional
      linear data access functions $\phi_i : \mathbb{Z}^d \to \mathbb{Z}^{d_i}$, for $i \in [n]$
for $x_1 \in [0, L_1), \ldots, x_d \in [0, L_d)$ :
    perform operations on $A_1 [\phi_1 (x_1, ..., x_d)], ..., A_n [\phi_n (x_1, ..., x_d)]$
```

At each iteration $x_1, \ldots, x_d$ of the the nested loops, the program accesses the arrays at addresses given by the *data access* functions $\phi_1, ..., \phi_n(x_1, \ldots, x_d)$, which fully specify the patterns the nested loop uses to access data. For instance, a classical matrix multiply (Algorithm 1) can be represented with the data access functions:

$$\begin{aligned}
\phi_1(x_1, x_2, x_3) &= (x_1, x_3) \\
\phi_2(x_1, x_2, x_3) &= (x_1, x_2) \\
\phi_3(x_1, x_2, x_3) &= (x_2, x_3)
\end{aligned}$$

Adjusting the data access functions ϕ allows us to represent most dense linear algebra and tensor kernels (including convolution), as well as many stencil kernels (especially for image processing). We will now generalize the preceding mapping process from matrix multiplication to general nested loops, by describing each step of the mapping process in detail.

2.2.1 Mapping Steps

This section describes the steps involved in mapping a general nested loop onto an accelerator. Each of these steps is associated with a set of choices, such as loop tile sizes and loop orders, which are *constrained* by the target hardware in a manner we will discuss here.

2.2.1.1 Tiling

Loop *tiling*, also referred to as *blocking* - that is, partitioning the loop nest into *tiles* which are executed in sequence - is a key optimization for increasing data reuse and reducing communication complexity of an operation. Note that we will use the term *tile* is to denote both the subsets of the loop nest, and subsets of the arrays used by each (loop nest) tile.

The most common form of tiling, which we refer to as *rectangular tiling*, breaks the iteration space into rectangular regions, as we did for matrix multiply earlier in Algorithm 2. More precisely, a rectangular tiling transforms a nested loop of the form given by Algorithm 4, given a set of *tile sizes* $T_1, \ldots, T_n \in \mathbb{Z}$ (such that $T_i \leq L_i$), into the loop nest shown in Algorithm 5.

Algorithm 5: Rectangular tiling of nested loop

```
1 for $x_{O,1} \in [0, L_1/T_1), \ldots, x_{O,d} \in [0, L_d/T_d)$ : // outer loop
2     Load subsets of $A_{1,\ldots,n}$ required for following nested loops into fast memory
3     for $x_{I,1} \in [0, T_1), \ldots, x_{I,d} \in [0, T_d)$ : // inner loop
4         $x_i = T_i x_{O,i} + x_{I,i} \quad \forall i \in [d]$
5         if $x_i > L_i$ for some $i$ : // tail case handling
6             break loop $i$
7         perform operations on $A_1 [\phi_1 (x_1, ..., x_d)], ..., A_n [\phi_n (x_1, ..., x_d)]$
8     Flush writes to slow memory
9     Evict subsets of $A_{1,\ldots,n}$ not required for next tile
```

If T_i does not perfectly divide L_i, a *tail case* is generated, corresponding to the final few iterations along the x_i axis. These are represented in Algorithm 5 with an *if* statement on Line 5 for compactness, but in practice (as branching is expensive, and often infeasible on many accelerator architectures) they are more commonly split off into separate loops as in Algorithm 2.

The choice of the tile sizes T_i heavily affects the amount of data movement (and therefore the arithmetic intensity) of the resulting mapping, which often dominates the total time and energy cost of many tensor workloads. In fact, both prior work [90, 44] and our own sensitivity analysis (discussed further in Section 6.2.5) has shown that loop tiling is by far the most important decision involved in constructing a mapping.

In order for a tiling to be executable on a system, the memory footprint of the inner loop - that is, the total size of the tensor tiles corresponding to the inner loop - must be at most the size of the fast memory. For instance, in Alg. 2, the memory footprint of a tile is $T_nT_m + T_nT_k + T_kT_m$, which must be bounded above by M. In the case of multiple buffers, with different tensors, each buffer must be able to accommodate the tiles for the tensors assigned to it. For instance, if we have a scratchpad of size M_S for the inputs and an accumulator of size M_A for the outputs, the constraints for the matrix multiply example become

$$\begin{aligned} T_nT_m &\leq M_A \\ T_nT_k + T_kT_m &\leq M_S \end{aligned}$$

For architectures and systems with multilevel memory hierarchies, the inner loop may be recursively tiled to generate a *multilevel tiling*, where each level's memory footprint fits inside a level of the memory hierarchy.

Other architectural features and optimizations can also affect the constraints for a tiling. For instance, *double-buffering* divides the scratchpad into two halves, with one half being used by the processor for computation while the other is loaded with data from memory, interleaving communication and computation in order to minimize latency. This effectively doubles the memory footprint of tiles corresponding to tensors that are not kept *stationary* in fast memory between successive tiles (see next section for more on stationarity).

Furthermore, certain axes may not be tilable. For instance, convolution accelerators are often built with the assumption that the entire filter (being sufficiently small) will be loaded onto purpose-built registers , and therefore will not support tiling along axes corresponding to filter dimensions. For each of these axes, we constrain the tile size T_i to be equal to the loop bound L_i.

Rectangular tilings are the main form of tiling used in practice, both because of ease of implementation and because they offer good performance for many common problems. In fact, we will show in Chapter 3 that it rectangular tiles are optimal for the case where all ϕ_i return subsets of their inputs (i.e. are *projections*). However, non-rectangular tiling schemes may be required to attain theoretical optima; we will describe one such case - strided convolutions - in Chapter 4. Loop nests with data dependencies between iterations may also benefit from *polyhedral transformations* [28, 5, 3], but are outside the scope of this work.

2.2.1.2 Loop Permutation

In our matrix multiply example earlier, we permuted the order of the loops in order to change which matrix would remain stationary in fast memory between successive tiles. In general, the tensors indexed by axes spanned by outermost loops surrounding a tile (e.g. $x_1, x_2, ...$ in Algorithm 5) will be held stationary between successive tiles. Depending on the target hardware, different loop orderings can result in different costs (especially if different tensors are stored in separate buffers with varying costs), and some orderings may be entirely unsupported.

Consider, for instance, an accelerator that performs accumulation in 32-bit precision but then writes the output to main memory in 16-bit precision. Consider the tiled matrix multiplication loop nest in Algorithm 2. Notice, that this loop ordering requires us to write partial sums of C (which is indexed by m and n) to main memory between successive tiles, as the innermost loop surrounding the tile is n_{out}, meaning that two consecutive tiles would correspond to different columns of C. However, writing out partial sums to main memory would require us to first round the 32-bit values present in fast memory to a 16-bit value, leading to loss of precision and possible numerical instability. As a result, this accelerator demands an *output-stationary* dataflow that keeps partial sums of C on-chip between tiles, requiring that m_{out} and n_{out} be the outermost loops outside the tile. We therefore can only choose between two loop orderings for the outer loops: $m_{\text{out}}, n_{\text{out}}, k_{\text{out}}$, or $n_{\text{out}}, m_{\text{out}}, k_{\text{out}}$.

2.2.1.3 Parallelization and Hardware Instruction Mapping

Our previous matrix multiply example obtained parallelism through one-dimensional vectorization: Algorithm 3 was obtained by first splitting the innermost loop into two loops, with the inner loop having four iterations (equal to the vector length on our target architecture), and replacing this inner loop with an equivalent vector instruction.

However, different architectures have dramatically diffrent forms of parallelism available to them, ranging from vectorization to systolic parallelism to CPU-style multithreading. A common abstraction that can be used to describe many of these forms is *spatio-temporal mapping*: each *for* loop will be mapped onto hardware *temporally,* with all its iterations being run in sequence (which we denote as a standard *for* loop), or *spatially*, with its iterations being mapped to separate physical resources on a chip (which we denote with the keyword *spatial-for*).

As a *spatial-for* loop maps each iteration to a separate processing element on the chip, the product of the sizes of loops mapped spatially must be bounded by the total amount of parallelism available on the chip. Beyond that, however, the interpretation of a *spatial-for* depends on both the axis and the target architecture. Along a non-reduction axis (e.g. m, n in our matrix multiply example), data required by separate iterations in a for loop may be either directly broadcasted to each processing element through dedicated wiring, as is done on the NVDLA [23] and ShiDianNao [21] accelerators, or be transferred systolically, as in TPU [42] and Gemmini [26]. Reduction axes, such as k, may be mapped to reduction trees, as in NVDLA and DianNao [10] or to systolic reductions as in TPU and Gemmini.

2.2.1.4 Graph-Level Optimizations

Most tensor workloads, especially in machine learning, consist of large *graphs* of operations, not just single operations, and are somewhat robust to perturbation. This opens up several avenues of optimization, which are beyond the scope of this work:

- L*ayer fusion* or *operator fusion* combines multiple layers (e.g., a matmul followed by a normalization or activation layer) into a single tensor operation to be scheduled and run at once, sometimes through the use of dedicated on-chip hardware to compute activations on the output. This reduces *interlayer* communication, as the results of one layer can remain on the chip as input without being written to and later read from main memory, at the cost of *intralayer* communication. The performance gains from such optimizations can vary widely depending on the kernel being run and the amount of hardware available.

- Taking advantage of *sparsity* in the inputs can provide significant speedups. Such optimizations are heavily data-dependent (even more so when sparsity is introduced during runtime - for instance, using locality-sensitive hashing to zero out dot products likely to be small [8]) and, as a result, cannot always be estimated a priori [44]. Sparsity-aware architectures and mappings are a topic of active research.

2.2.2 Representing, Evaluating, and Optimizing Mappings

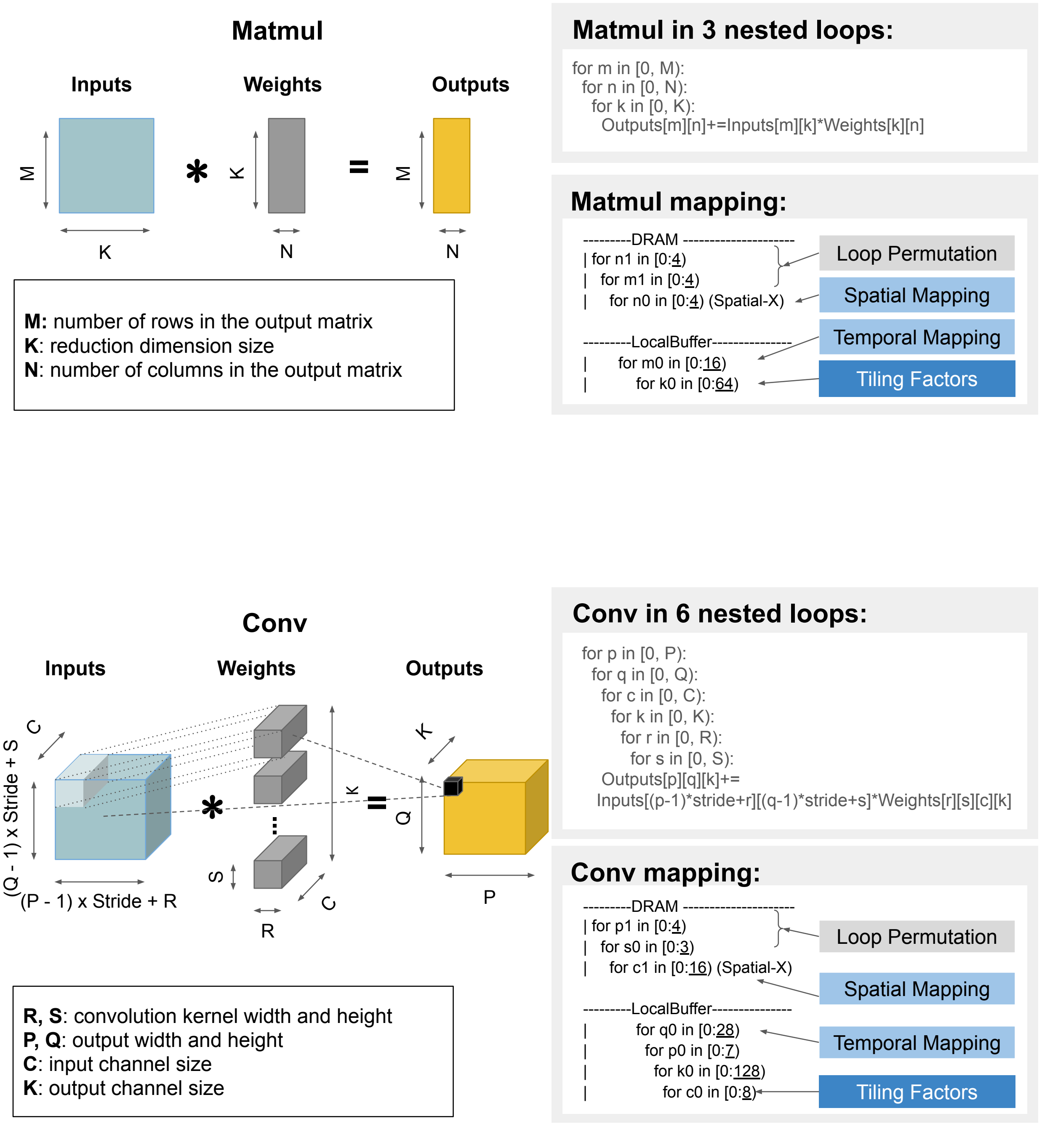
We refer to the set of choices - tile sizes, loop permutations, and tiling factors, e.g. as depicted in Figure 2.2.2 - introduced by the aforementioned rewrites as a *mapping*, and the set of all possible mappings as a *mapspace*. We will refer to each of the above choices as an *axis*.

As the computations involved in a dense tensor computation do not change with input - there is no branching - finding a mapping can be done entirely at compile time. Furthermore, the simplicity and regularity of the program structure obviates the need for many forms of static analysis, such as dependency analysis. However, constructing mappings that result in *good performance* on accelerators is challenging. The significant number of choices involved in mappings leads to a combinatorial explosion in the number of possible mappings, giving rise to a mapspace with over 10^{20} points [72]. As Figure 2.2.1 illustrates, only a small fraction of mappings generates good results.

The goal of a *mapper* or *mapping algorithm* is to search this large, high-dimensional space to find a mapping that satisfies all hardware constraints and minimizes some objective - usually some combination of energy or latency on a target hardware architecture. Mappers can be evaluated not only on the performance of the mappings they return, but also on their speed of execution, sample complexity, and generalizability to different hardware and software targets.

2.2.2.1 Evaluating Mappings

In order to find a performant mapping, we must first define an objective function to optimize. If the target architecture exists and can be instrumented, actual measured performance - usually latency, energy consumption, or a combined figure such as energy-delay product



M: number of rows in the output matrix
K: reduction dimension size
N: number of columns in the output matrix

Matmul in 3 nested loops:

```
for m in [0, M):
  for n in [0, N):
    for k in [0, K):
      Outputs[m][n]+=Inputs[m][k]*Weights[k][n]
```

Matmul mapping:

```
---------DRAM ---------------------
| for n1 in [0:4)
|   for m1 in [0:4)
|     for n0 in [0:4) (Spatial-X)

---------LocalBuffer---------------
|       for m0 in [0:16)
|           for k0 in [0:64)
```

R, S: convolution kernel width and height
P, Q: output width and height
C: input channel size
K: output channel size

Conv in 6 nested loops:

```
for p in [0, P):
  for q in [0, Q):
    for c in [0, C):
      for k in [0, K):
        for r in [0, R):
          for s in [0, S):
    Outputs[p][q][k]+=
   Inputs[(p-1)*stride+r][(q-1)*stride+s]*Weights[r][s][c][k]
```

Conv mapping:

```
---------DRAM ---------------------
| for p1 in [0:4)
|   for s0 in [0:3)
|     for c1 in [0:16) (Spatial-X)

---------LocalBuffer---------------
|           for q0 in [0:28)
|             for p0 in [0:7)
|               for k0 in [0:128)
|                 for c0 in [0:8)
```

Figure 2.2.2: Mapping representations for matrix multiplication and convolutions. Originally published in [50]

(EDP) - can be used. However, this is not often not desirable or even possible for a variety of reasons:

- Hardware availability may change rapidly. For instance, for a multi-tenancy setup, the amount of available memory for a given task may be affected by the number and size of other tasks currently running on the same machine.

- The hardware itself may not exist. For instance, design space exploration during the process of developing an accelerator requires performance evaluation of workloads (and therefore mappings) on target hardware that may be be described only by a few parameters, with the full design unrealized.

- Simpler performance models (e.g. communication volume on an idealized memory model) allow us to develop *provable* bounds on performance that are infeasible to find for more complex models, and can be used to evaluate actual performance data (e.g. by comparing benchmark results to a roofline [87] model).

Different performance objectives offer different levels of fidelity, runtime cost, target workload scopes, compatibility for various mapping algorithms, and ability to prove theoretical results. We enumerate several here, in roughly increasing order of fidelity.

Abstract computational models, such as communication volume on a simple two-level memory hierarchy, are simple to define and can be easily adapted to a large variety of target hardware. They allow the development of theoretical *lower bounds* on performance, which we will cover in detail in the next section. However, their simplicity and generalizability comes at a cost - as they incorporate almost no assumptions on the target hardware, they can only serve as loose proxies for performance.

Adding architectural assumptions (e.g. on the structure of a memory hierarchy and parallelism available) and incorporating parameters such as memory bandwidth, systolic array size, and energy consumption per operation leads to more sophisticated **domain-specific analytical performance models** such as Timeloop and Maestro [71, 53, 57, 62, 38, 35]. These models leverage known iteration space bounds in tensor algebra workloads, as well as statically analyzable data access patterns, to estimate performance extremely quickly. Many of these models are expressed as closed-form mathematical expressions which can also be used directly as the objectives in optimization-based mappers, although their complexity generally precludes the derivation of theoretical lower bounds.

Another class of popular performance models are **data-driven ML models** [9, 32, 48]. Instead of building the performance model analytically to express known relations between mapping decisions and performance, these models use statistical techniques to iteratively fit a model to the mapping performance data collected over time, and often optimize over these models to produce performant mappings. They typically require large amounts of data and training time in order to learn and provide accurate predictions, but are fast to run once trained.

The major drawback of prior models is that the generated mappings might not perform optimally (or even well) on the actual accelerator since the models can fail to capture the implementation differences in the hardware accurately. **Cycle-exact software models** based on real hardware implementation can achieve higher fidelity [86, 77], as can **FPGA emulation** using platforms such as Firesim [46], which can be used to model hardware in development. However, these tools are several orders of magnitude more resource-intensive to run than analytical and trained ML models.

Turning Mappings into Code

Both cycle-accurate simulators and emulators and real hardware such platforms require more than a set of problem dimensions and a description of mappings - they require a stream of explicit instructions.

Generating this stream of instructions requires that one account for a large number of edge cases. For example, a simple tiling operation for a matmul - representable as a single line of tile sizes in a DNN accelerator modeling platform such as Timeloop [71] - requires both the insertion of instructions specifying memory movement between different levels of the memory hierarchy as well as the generation code for edge cases that appear when matrix dimensions are not evenly divisible by the tile size (as we did in Algorithm 2). Furthermore, many mapspace optimization methods rely on iterative search, which requires the evaluation of a large number of mappings; this in turn requires that mappings be translatable to code automatically.

As a result, *code generation* tools are used to actually implement mappings onto hardware (or simulators). Many of these tools integrate not only a specification of the target hardware but also mapping decision algorithms, often tuned for that hardware target. Because of this integration, evaluating the performance of mappings *not* generated by their internal framework and working on alternative hardware targets is challenging.

In order to address this problem, *user-schedulable languages* such as Halide [73], TVM [9], Rise/Elevate [30], and Exo [91] have been developed. These tools take as input a description of the computation to be performed and a point in the mapspace. They are generally defined by a set of *rewrite rules* on representing code transformations such as splitting and rearranging loops, replacing appropriate loops with ISA instructions, and fusing loops. These languages also allow the user to specify and customize the hardware instruction set and seamlessly convert mappings into executable code by representing them as a sequence of rewrite rules [64, 93].

2.2.2.2 Searching For Mappings

Given a performance objective, a variety of methods exist to optimize it over the mapspace. We can cluster them into three general categories:

- *Brute-force search* methods [15, 71, 90] randomly sample and enumerate large number of points in the mapspace. However, the size of the mapspace and the rarity of mappings that result in good performance require an extremely large number of points (typically thousands, if not tens of thousands) to be searched and evaluated to determine the mapping for a single kernel. As a result, these methods rely on fast analytical performance models and are generally unsuited for higher-fidelity simulated or emulated models.

- *Feedback-driven methods* address the sample inefficiency of brute-force search methods using use statistical or machine learning models. These approaches include black-box optimization techniques such as genetic algorithms [45, 43], reinforcement learning [89], and Bayesian optimization [76, 80], which aim to require fewer samples than brute-force methods. Alternatively *gradient-based methods* [32, 35] build differentiable surrogate functions that estimate performance based on input parameters. However, training gradient-based methods require millions of samples to build surrogate models, and the resulting surrogates cannot easily be used for hardware architectures not yet seen even after fine-tuning, and must be fully retrained from scratch at nontrivial expense [44].

- *Heuristics* perform one-shot analytic optimizations over a performance model (either defined explicitly to be used as an optimization target, or embedded implicitly in the heuristic). Such methods include polyhedral models [28, 51, 1]and constrained-optimization based approaches [38, 92]. Heuristics are efficient and generalize easily across hardware parameters and problem sizes, although often limited to optimizing certain axes of the mapspace (e.g. tilings, as we focus on in Chapters 2.3, 3, 4, or tilings and reorderings as in [68, 69], which build off our work). This makes them suited for guiding brute-force and feedback-driven methods, either by providing cheap, rapidly calculable features to them, or by eliminating several axes from the mapspace.

As a result, we will begin by examining heuristic tiling algorithms which can be used as building blocks to construct more general mappers, and corresponding lower bounds which can be proven in a simple abstract communication model.

2.3 Provably Optimal Tilings in the Two-Layer Memory Model

In order to develop tilings that are *provably* optimal, we must first specify a model of computation that allows us to derive lower bounds to prove optimality. As communication is the dominant cost in both time and energy for most accelerators, , and tiling largely affects communication, we use the *two-level memory hierarchy* model has a slow memory of infinite size connected to a processor with an onboard fast memory of size M; all inputs and outputs must be read from and written to fast memory. In this model, the cost is the amount of data transferred in both directions between slow and fast memory. We will largely measure this

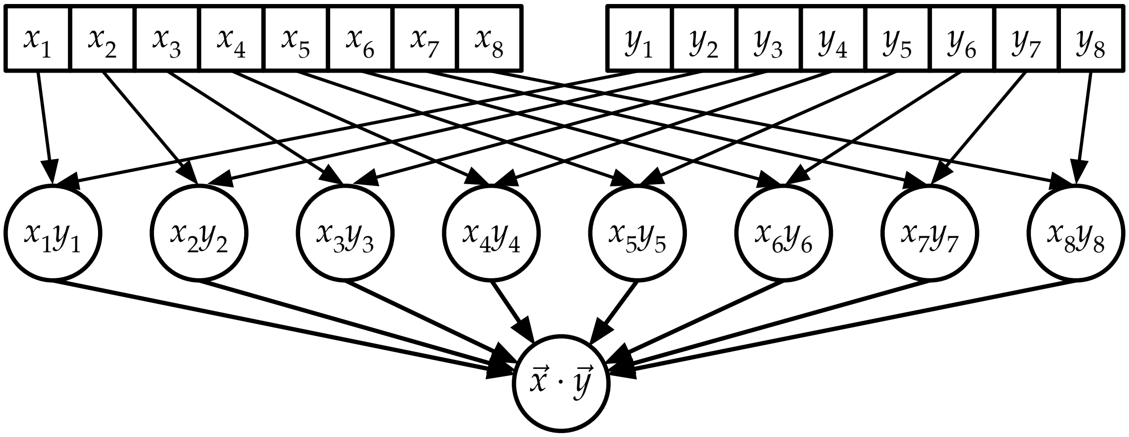


Figure 2.3.1: A CDAG for a dot product

in *words*, with one word corresponding to a single numeric value; however, problems with multiple datatypes (with different sizes), will require more detailed analysis.

Communication lower bounds derived for the two-level memory model can be used to describe a variety of computations, including multilevel memory hierarchies (by applying the model to bound communication on multiple levels) and distributed-memory systems.

Several methods have been proposed to bound communication in memory hierarchy models, for various models of computation. We establish the fundamentals of these methods in this chapter, and discuss how the derivation of communication lower bounds suggests ways to find tilings that attain them.

2.3.1 Computational DAGs and Pebble Games

A *straight-line computation* (i.e. one without branching) can be described as a computational directed acyclic graph (CDAG) (V, E) with vertices corresponding to computations and edges corresponding to data dependencies. Specifically, the computation represented by vertex v can only be executed after all computations represented by vertices u_i such that $(u_i, v) \in E$ have completed. We refer to a *schedule* of execution of the program represented by a CDAG D as an ordered traversal $(v_1, v_2, ..., v_n)$ of the vertices of the DAG that respects dependencies; specifically, all ancestors of v_i must occur before v_i in the schedule.

Pebble games were first proposed by Sethi [79] to bound the communication complexity of a CDAG in the context of register allocation. Given a CDAG $G = (V, E)$, a player may, at each step, perform one of several operations with pebbles. In the original variant, sometimes referred to as the *black pebble game* , each pebble corresponds to a register, and the following operations are permitted:

- place a pebble on a source vertex of G (initialize a register)

- place a pebble on a vertex v, provided that all vertices u such that $(u, v) \in E$ have pebbles on them (perform a computation whose inputs are all in registers, and write the output to another register)
- remove a pebble from any vertex in G (free a register)

The game completes when all sink vertices of G have pebbles on them. From this formulation, it is clear if a CDAG can be black-pebbled with r pebbles, it can be executed on a machine with r registers.

In order to model communication for a memory hierarchy model, this game must be augmented with an operation corresponding to *communicating* a value between fast memory and slow memory. The resulting *red-blue pebble game,* proposed by Hong and Kung [36], involves two pebble colors: red, corresponding to fast memory, and blue, corresponding to slow memory. At the beginning of the program, all source nodes in G have blue pebbles on them, as the inputs to the program are presumed to initially reside in slow memory. Each turn involves one of the following operations:

- (L) place a red pebble on any vertex with a blue pebble on it (load a value from slow memory to fast memory)
- (S) place a blue pebble on any vertex with a red pebble on it (store a value from fast memory to slow memory)
- (C) place a red pebble on a vertex v, provide that all vertices u such that $(u, v) \in E$ have *red* pebbles on them (perform a computation whose inputs are all in fast memory, and write the output to fast memory)
- (D) delete a pebble of any color from a vertex (remove a value from slow or fast memory)

The game concludes when all sink vertices of G have blue pebbles on them, indicating that the result has been computed and written to slow memory.

Given an instance of the red-blue pebble game, the maximum number of red pebbles on the graph at any time is the amount of fast memory required to execute the corresponding schedule, with the communication cost being the the number of (L) and (S) steps that occur. As a result, finding the communication cost of a CDAG in the two-level memory hierarchy model equivalent to determining the minimum number of (L) and (S) steps required to pebble the CDAG given M red pebbles.

Unfortunately, determining the *exact* minimum cost required to pebble an arbitrary CDAG is intractable. Determining the optimal pebbling scheme (and corresponding cost) for black-pebbling (register allocation) and “base” red-blue pebbling are both known to be PSPACE-complete [27, 17], and optimal pebbling for variants of red-blue pebble game that forbid or penalize recomputation have been shown to be NP-complete [70] in the general case.

However, pebbling bounds have been effectively derived, and tight pebbling based bounds have been efficiently derived for certain *classes* of computations.

The main approach is to *partition* the graph into subgraphs, each corresponding to a subset of computations.

Theorem 1 (Th. 3.1 from [36])**.** *Let an S-partition of a CDAG $G = (V, E)$ be defined as a partition of V into disjoint subsets $V_1, ..., V_h$ such that each subset V_i satisfies the following criteria*

- *V_i must have a* dominator set *(a set of vertices such that each path from a source vertex of G to a vertex in V_i must contain at least one vertex in the dominator set) of size at most S. Intuitively, V_i must depend on at most S values.*

- *The* minimum set *(set of vertices in V_i with no children in V_i) must be of size most S. Intuitively, the size of the output of V_i must be at most S*

- *Every path between two vertices in V_i consist entirely of vertices in V_i. Intuitively, this means the set of tasks represented by V_i can be executed as a single block, or equivalently, that there are no cyclic dependencies between different subsets.*

Suppose there exists a schedule for a CDAG G that, on a machine with M words of fast memory, with communication cost q. Then there exists a $2M$-partition of G into h subsets of vertices such that $M(h - 1) \leq q \leq Mh$.

Proof. (from [52]) divide the schedule into h contiguous, consecutive *segments* $S_1, ..., S_h$ each of which communicates exactly M words [1] (i.e. uses exactly M (L) and (S) steps). We will use V_i to denote the set of values computed during segment i. By construction, there can be no cyclic dependencies between segments.

We define the following sets of vertices:

- $D_{R,i}$ is set of vertices whose values are already in fast memory at the beginning of S_i, i.e. each of its vertices have red pebbles at the start of S_i . Since fast memory may have no more than M words in it at the beginning of each segment, and in fact at any time, $|D_{R,i}| \leq M$

- $D_{B,i}$ is the set of vertices whose values are loaded into fast memory during S_i, i.e. each of its vertices starts segment i with blue pebbles and gain red pebbles during S_i. By definition of segments, $|D_{B,i}| \leq M$.

- $W_{R,i}$ is the set of vertices whose values remain in fast memory at the end of S_i, i.e. each of its vertices have red pebbles at the end of S_i. By the constraint on the size of fast memory (as with $D_{R,i}$), $|W_{R,i}| \leq M$

[1] The last segment may communicate less than M words, since we are not guaranteed that the total communication is divisible by M. For simplicity, we will assume the last segment communicates exactly M words of.

- $W_{B,i}$ is the set of vertices whose values are written to slow memory during S_i, i.e. each of its vertices have blue pebbles placed on them during S_i. By definition of segments, $|W_{B,i}| \leq M$.

Notice that $D_{R,i} \cup D_{B,i}$ consist of all inputs to V_i and therefore forms a dominator set of V_i, and its size is at most $|D_{R,i}| + |D_{B,i}| \leq 2M$.

Furthermore, if a vertex $v \in V_i$ has no children in V_i, its value must either remain in fast memory and the end of S_i (to be used by a later segment), giving $v \in W_{R,i}$; or be written to slow memory during S_i (either to be read in and used by a later segment, or as part of the final output of the program), giving $v \in W_{B,i}$. Otherwise v would not be performing any useful work and could be deleted without changing the computation. As a result, $W_{R,i,} \cup W_{B,i}$ must contain the minimum set of V_i , giving an upper bound of $|W_{R,i}| + |W_{B,i}| \leq 2M$ on the size of the minimum set.

As a result, $V_1, ..., V_h$ is a $2M$-partition. As each segment communicates exactly M words, and there are h segments, the total communication q is Mh. □

We note that the key step of the proof is to divide the schedule into h segments, each of which communicates exactly M words. Determining the minimum number of segments required in such a construction will be key to our approach in the next section.

2.3.2 The Segment-Based Approach

Many straight-line computations of interest, including dense linear algebra, n-body problems, convolutions, can be expressed as a collection of *nested loops*, where each iteration accesses elements from several multidimensional arrays, indexed by some affine functions $\phi : \mathbb{Z}^d \to \mathbb{Z}^{d_n}$ (for some $d_n \leq d$) of the current loop iteration function:

$$\begin{array}{l} \text{for } x_1 \in [L_1], ..., \text{for } x_d \in [L_d] : \\ \quad \text{perform operations on } A_1[\phi_1(x_1, ..., x_d)], ..., A_n[\phi_n(x_1, ..., x_d)] \end{array} . \tag{2.3.1}$$

Let us assume that there are no dependencies between loop nests (i.e. any schedule is valid) and that we forbid recomputation. For operations such as matrix multiplication, tensor contraction, and convolution, these tend to be reasonably good assumptions - the accumulation is an associative and commutative operation, and there is no reason to perform recomputations since there is no reuse in the outputs.

For such problems, we apply the segmenting approach from the preceding section as follows; given an arbitrary schedule, divide the schedule into h segments $S_1, ..., S_h$, each corresponding to exactly[2] M words of communication. Our goal will be to establish a lower bound on h. We do so by establishing an *upper* bound U on the number of loop iterations

[2] Again, for simplicity, we assume the last segment uses exactly M words of communication, although it may be lower.

that can be performed in each segment. As the total number of operations is $\prod_{i\in[d]} L_i$, it follows that

$$h \geq \frac{\prod_{i\in[d]} L_i}{U} \ .$$

Notice that any computation performed in segment S_i may use (by reading from, or writing to) at most $2M$ words of data: the M words already present in fast memory at the beginning of the computation, and M words that may be read or written during the execution of the segment. As a result, the total communication cost must be at least

$$q \geq 2Mh = 2M\frac{\prod_{i\in[d]} L_i}{U} \ . \tag{2.3.2}$$

The crux of finding such lower bounds, as a result, is determining the upper bound U, the number of loop iterations that may be performed with access to at most $2M$ words of memory. Irony, Toledo, and Tiskin [39, 84] apply a geometric approach to this problem to develop such a bound for the classical three-nested loop matrix multiplication problem:

$$\begin{array}{c} \text{for } x_1 \in [L_1], x_2 \in [L_2], x_3 \in [L_3]: \\ A_3[x_1, x_2] += A_1[x_1, x_3] A_2[x_3, x_2] \end{array} \ . \tag{2.3.3}$$

In the above notation, the data access functions are:

$$\begin{aligned} \phi_1(x_1, x_2, x_3) &= (x_1, x_3) \\ \phi_2(x_1, x_2, x_3) &= (x_3, x_2) \\ \phi_3(x_1, x_2, x_3) &= (x_1, x_2) \ . \end{aligned}$$

Each iteration $\vec{x} = (x_1, x_2, x_3)$ may be represented as an point in a parallelepiped in $\mathbb{Z}^3$, and the array addresses required to execute that iteration are given by its projections ϕ_1, ϕ_2, and ϕ_3. It follows that the array addresses required for a *segment* $S \subseteq [L_1] \times [L_2] \times [L_3] \subseteq \mathbb{Z}^3$ - which can be thought of as a set of iteration points - are projections of S, specifically $\{[\phi_i(\vec{x})] : \vec{x} \in S\}$ for $i \in [3]$; we will denote these $\phi_i(S)$ for convenience. Therefore, the amount of data required to execute S_i is $|\phi_1(S_i)| + |\phi_2(S_i)| + |\phi_3(S_i)|$, which (by definition) must be bounded above by $2M$; this constraint can be relaxed to $|\phi_i(S)| \leq 2M$.

As a result, the problem may be reduced *geometrically* to determining an upper bound on the size of a set $S \subseteq \mathbb{Z}^3$ subject to constraints on the sizes of its projections ϕ_i. This is immediately given by a classic inequality of Loomis and Whitney:

Theorem 2 ((Discrete Loomis-Whitney inequality, Theorem 2 from [56]))**.** *Let S be a set of points in $\mathbb{Z}^d$, and $\phi_i(S)$ be the set of projections of S onto the $d-1$-dimensional hyperplane formed by fixing the ith coordinate. Then*

$$|S|^{d-1} \leq \prod_i |\phi_i(s)| \ .$$

It immediately follows from Theorem 2 that the size of the matrix multiply segment is bounded above by $|S|^2 \leq (2M)^3$; plugging into (2.3.2) gives a communication lower bound of

$$q_{\text{matmul}} = \frac{2ML_1L_2L_3}{(2M)^{3/2}} = \frac{L_1L_2L_3}{\sqrt{2M}} \tag{2.3.4}$$

It is also instructive to attempt to attain this lower bound. Let us assume as a ansatz that the rectangular tiling of Algorithm 2 is appropriate for this problem. Our goal will be to find the largest possible (in terms of number of operations) tiles whose memory footprint can fit inside fast memory. This results in the optimization program:

$$\begin{aligned} &\max T_1T_2T_3 \\ &\qquad \text{s.t. } T_1T_2 + T_2T_3 + T_1T_3 \leq M \end{aligned}$$

This optimization program may be directly solved using signomial optimization techniques [65]. However, simpler methods suffice to prove the (asymptotic) optimality of this lower bound. Let us relax the memory footprint constraints as we did above during the proof of the lower bound:

$$\begin{aligned} &\max T_1T_2T_3 \\ &\qquad \text{s.t. } T_1T_2 \leq M \\ &\qquad\qquad T_2T_3 \leq M \\ &\qquad\qquad T_1T_3 \leq M \end{aligned}$$

This relaxation changes the constraints by at most a factor of 3, which we will ignore as we are operating asymptotically (we will deal with constant factors more precisely in Chapter 4). Taking logs base M, and letting $\lambda_i = \log_M T_i$, we have:

$$\begin{aligned} &\max \lambda_1 + \lambda_2 + \lambda_3 \\ &\qquad \text{s.t. } \lambda_1 + \lambda_2 \leq 1 \\ &\qquad\qquad \lambda_2 + \lambda_3 \leq 1 \\ &\qquad\qquad \lambda_1 + \lambda_3 \leq 1 \end{aligned}$$

The solution to this linear program is $\lambda_1 = \lambda_2 = \lambda_3 = 1/2$, which suggests a tile size of $\sqrt{M} \times \sqrt{M} \times \sqrt{M}$, which implies that each tile consists of $M^{3/2}$ operations with a memory footprint of (ignoring constant factors) M. As a result, the number of tiles is $L_1L_2L_3/M^{3/2}$, leading to asymptotic communication cost of $L_1L_2L_3/\sqrt{M}$, which matches the lower bound (2.3.4).

We note that the preceding derivation gives no guarantee that a tile of size $\sqrt{M} \times \sqrt{M} \times \sqrt{M}$ will fit in the bounds of the problem. For example, if $L_3 = 1$ (i.e. a matrix-vector multiply), almost every such tiling would be larger than the problem itself. We address this issue in Chapter 3.

2.3.3 The Discrete Brascamp-Lieb Inequalities

In order to develop lower bounds and optimal tilings beyond matrix multiplication, we must generalize beyond the Loomis-Whitney inequality (Th. 2). Our main tool will be the *discrete Brascamp-Lieb inequalities* (often abbreviated as "BL" or "HBL", with the H being a reference to Hölder) first introduced in [14].

Definition 3. Given a finitely generated group G, let $\text{rank}(G)$ denote the size of the smallest generating set of G.

Theorem 4 (discrete Brascamp-Lieb inequality)**.** *Let G and G_j (for $j \in [n]$) be finitely generated Abelian group, and G be torsion-free. Let $\phi_j : G \to G_j$ be group homomorphisms. If the rank inequalities*

$$rank(H) \leq \sum_{j\in[n]} s_j rank(\phi_j(H)) \qquad \forall \textit{ subgroups } H \leq G \tag{2.3.5}$$

hold for some nonnegative numbers s_j, then so does the following functional inequality:

$$\sum_{x\in G} \prod_{j\in[n]} f_j(\phi_j(x)) \leq \prod_{j\in[n]} \|f_j\|_{1/s_j} \tag{2.3.6}$$

for all nonnegative functions f_j with a defined ℓ^{1/s_j} norm over G_j.

We can apply this theorem to generate a Loomis-Whitney style cardinality inequality as follows: let $S \subseteq \mathbb{Z}^d$ be a segment. Let $f_j : \mathbb{Z}^{d_j} \to \mathbb{Z}$ be defined to be the following indicator function:

$$f_j(x) = \begin{cases} 1 & x \in \phi_j(S) \\ 0 & \text{otherwise} \end{cases} .$$

It immediately follows from the definition of f_j that

$$\|f_j\|_{1/s_j} = |\phi_j(S)|^{s_j} . \tag{2.3.7}$$

Furthermore, notice that if $x \in S$, then $\phi_j(x) \in \phi_j(S)$ for all j, which implies $\prod_{j\in[m]} f_j(\phi_j(x)) = 1$. As a result, the cardinality of S is bounded by the left hand side of 2.3.6:

$$\begin{aligned} |S| &\leq \sum_{x\in\mathbb{Z}^d} \prod_{j\in[n]} f_j(\phi_j(x)) \\ &\leq \prod_{j\in[n]} \|f_j\|_{1/s_j} && \text{applying (2.3.6)} \\ &= \prod_{j\in[n]} |\phi_j(S)|^{s_j} && \text{applying (2.3.7)} \end{aligned}$$

which gives us the following generalization of the Loomis-Whitney inequality

Corollary 5. *Let $\phi_j : \mathbb{Z}^d \to \mathbb{Z}^{d_j}$ be linear functions, and S be a subset of $\mathbb{Z}^d$. Then for all $s_j \geq 0$ satisfying*

$$rank(H) \leq \sum_{j\in[n]} s_j rank(\phi_j(H)) \qquad \forall \text{ subgroups } H \leq \mathbb{Z}^d \tag{2.3.8}$$

it follows that

$$|S| \leq \prod_{j\in[n]} |\phi_j(S)|^{s_j} \; . \tag{2.3.9}$$

One can immediately apply the same argument from 2.3.2 to derive the following communication lower bound:

Theorem 6. *Let $\phi_j : \mathbb{Z}^d \to \mathbb{Z}^{d_j}$ be linear functions, and suppose we have the following loop nest:*

$$\begin{array}{l} \text{for } x_1 \in [L_1], ..., \text{for } x_d \in [L_d] : \\ \quad \text{perform operations on } A_1[\phi_1(x_1, ..., x_d)], ..., A_n[\phi_n(x_1, ..., x_d)] \end{array} . \tag{2.3.10}$$

The communication q required to execute this loop nest in the two-level memory hierarchy model with fast memory size M is bounded below by

$$q \geq \Omega\left(\frac{\prod_{i\in[d]} L_i}{M^{\sum_{j\in[n]} s_j - 1}}\right)$$

*for **any** s_j satisfying*

$$rank(H) \leq \sum_{j\in[n]} s_j rank(\phi_j(H)) \qquad \forall \text{ subgroups } H \leq \mathbb{Z}^d \; . \tag{2.3.11}$$

In order to find the strongest lower bound (which is also the only one possibly attainable by a tiling), we must minimize $\sum_{j\in[m]} s_j$ subject to the linear *rank constraints* (2.3.11) (which form a polytope that we will refer to as the *discrete Brascamp-Lieb polytope*); this is a simple linear programming problem, provided we can enumerate the rank constraints.

Simple enumeration of the subgroups H does not suffice to determine the rank constraints, as there are infinitely many possible subgroups H of $\mathbb{Z}^d$. Note that the number of possible rank constraints is bounded - in particular, there are only d possible values for rank(H), and only d_j possible values for each of the rank$(\phi_j(H))$, so any valid rank constraint would be of the form

$$a \leq \sum_{j\in[m]} s_j a_j$$

for $a \leq d$, $a_j \leq d_j$. Therefore, the most straightforward method to enumerate all the constraints of the Brascamp-Lieb polyhedron would be to enumerate all such possible a, a_j and pass them to the following decision problem:

Problem 7. Given positive integers $a \leq d$, $a_j \leq d_j$, does there exist some subgroup $H \leq \mathbb{Z}^d$ such that $\operatorname{rank}(H) = a$ and $\operatorname{rank}(\phi_j(H)) = d_j$ for all j?

Unfortunately, [14] showed that deciding Problem 7 would also decide Hilbert's tenth problem in the rationals: that is, deciding whether a finite set of multivariate polynomials with rational coefficients has a common rational root. The decidability of Hilbert's tenth problem in the rationals has been unknown for many decades, although the equivalent problem for the integers is well known to be undecidable [60].

Note that this does *not* preclude a complete description of the BL polytope. In particular, solving Problem 7 requires checking the validity of *every* possible rank inequality, including those implied - and therefore made redundant - by other valid rank inequalities. It is in fact possible to enumerate a set of constraints that completely defines the BL polytope - i.e. one that implies all the rank inequalities - through an algorithm given in [13]. Unfortunately, this algorithm has no bounds on runtime (other than a guarantee that it terminates), and as a result we are forced to rely on alternative means to do this.

The following theorem, a straightforward extension of the main result of [85] to to our discrete setting, can simplify the process of enumerating the constraints forming the Brascamp-Lieb polytope.

Definition 8. Let $\{H_i\} := \{H_1, \ldots, H_k\}$ be groups. The *lattice* of $\{H_i\}$, denoted $\mathcal{L}_{\{H_i\}}$, is defined as the smallest set of subspaces containing $\{H_i\}$ closed under intersection and direct sum: that is, if $V_1, V_2 \in \mathcal{L}_{\{H_i\}}$, then so are $V_1 \cap V_2$ and $V_1 + V_2 = \{v_1 + v_2 : v_1 \in V_1, v_2 \in V_2\}$.

In this lattice, the intersection serves as the meet operation, while the sum serves as the join.

Theorem 9. *Theorems 4 and 6 and Corollary 5 still hold if they are rewritten so the rank conditions (2.3.5), (2.3.8), and (2.3.11) are required only to apply for* $H \in \mathcal{L}_{\{\ker(\phi_i):i\in[n]\}}$.

Proof. Let $\Phi_j : \mathbb{Q}^d \to \mathbb{Q}^{d_j}$ extend ϕ_j to a $\mathbb{Q}$-linear map. In Section 2.2 of [13], it is proved that the polytope of (s_j) satisfying the rank conditions (2.3.5), (2.3.8), and (2.3.11) is exactly equal to the polytope of (s_j) satisfying, for each subspace $V \leq \mathbb{Q}^d$,

$$\dim V \leq \sum_{j=1}^{m} s_j \dim \phi_j(V).$$

In [85], Theorem 8 states that it suffices to check only these inequalities from subspaces in $\mathcal{L}_{\{\ker \Phi_j\}}$. To check these, it suffices to check the original rank conditions (2.3.5), (2.3.8), and (2.3.11) on the subgroups in $\mathcal{L}_{\{\ker \phi_j\}}$ and then to take $\mathbb{Q}$-linear spans of these subgroups. This completes the proof. □

We will use Theorem 9 to find lower bounds for convolutions in Chapter (4).

Furthermore, we can also examine subsets of the discrete Brascamp-Lieb polytope, which lead to (potentially weaker, but guaranteed correct) lower bounds. One such subset is given by the *real* counterpart to Theorem 4 .

Theorem 10 (Real Brascamp-Lieb inequality, from [4]). *Let $B_j : \mathbb{R}^d \to \mathbb{R}^{d_j}$ be linear functions for $j \in [n]$, and s_j be nonnegative reals*[3]. *If the* ***rank constraints***

$$\dim(V) \leq \sum_j s_j \dim(B_j V) \qquad \forall \textit{subspaces } V \leq \mathbb{R}^d \tag{2.3.12}$$

and the ***scaling constraint***

$$d = \sum_j s_j d_j \tag{2.3.13}$$

hold as well, then there exists a nonnegative, finite constant C

$$\int_{x \in \mathbb{R}^d} \prod_{j \in [n]} (f_j(B_j x))^{s_j}\, dx \leq C \prod_{j \in [n]} \left(\int_{x_j \in \mathbb{R}^{d_j}} f_j(x_j) dx_j \right)^{s_j}$$

for all nonnegative functions f.

This inequality is very similar to the discrete Brascamp-Lieb inequality, with the exception of the value C, which is referred to as the *capacity* of the inequality. In the discrete formulation, C is always 1 when the inequality is valid; the same does not hold in the real case.

We will refer to the polytope described by (2.3.12) and (2.3.13) as the *real Brascamp-Lieb* polytope. Notice that the rank constraints are very similar to the ones for the discrete case, except that they span over subspaces of $\mathbb{R}^d$ rather than a finitely generated group. In fact, as these subspaces include those with integer bases, the rank constraints (2.3.12) are a superset of the discrete rank constraints (2.3.8). As a result, the real BL polytope is a subset of the discrete BL polytope, implying that every (B, s) that satisfies the *real* rank and scaling constraints generates a valid *discrete* BL inequality and therefore communication lower bound. Although there is no guarantee that a stronger bound - one whose exponents satisfy the discrete scaling rank constraints but not the real ones - does not exist, optimizing over the real BL polytope produces tight inequalities in many practical cases, such as the Loomis-Whitney inequality (Theorem 2).

The advantage of considering real BL polytope is that a *bounded time* separation oracle [25] exists for it, which allows for the optimization of linear objectives (such as $\sum_j s_j$) over the polytope. Furthermore, as the number of possible inequalities is finite, this leads to a bounded time algorithm for enumerating the constraints of the real BL polytope. However, as the cost of optimizing a linear function over the real BL polytope is exponential (as the separation oracle runs in time polynomial to the greatest common divisor of the s_js, which increases as we approach an optimum), such results are of less practical interest than methods that allow us to analyze *special cases* of the Brascamp-Lieb polytope that typically arise from real code.

[3] Most papers on real BL inequalities, e.g. [4, 85, 25] use p instead of s, but we use s here for consistency with the discrete case.

Chapter 3

Tilings and Lower Bounds for Projective Nested Loops

For many loop nests of interest - matrix multiplication, tensor contraction, n-body pairwise interaction, among others - the data access functions ϕ_i are *projections* returning subsets of their inputs - that is, for all j, $\phi_j(x_1, ..., x_n) = (x_{j_1}, ..., x_{j_k})$ for some integers $j_1, ..., j_k \in [d]$. We refer to both the set $\{x_{j_1}, ..., x_{j_k}\}$ and the set $\{j_1, ..., j_k\}$ (depending on context) as the *support* of ϕ_j, which we denote $\text{supp}(\phi_j)$. The following theorem from [14] provides a simple description the Brascamp-Lieb polytope for in this case.

Theorem 11 (Th. 6.6 from [14]). *Let e_i be the subgroup of $\mathbb{Z}^d$ generated by the vector $[0, \ldots, 0, 1, 0, \ldots 0]^T$ with zero entries at all indices except for i. If ϕ_j are projections for all j, then the rank conditions*

$$rank(H) \leq \sum_{j \in [n]} s_j rank(\phi_j(H))$$

are satisfied for all subgroups $H \leq \mathbb{Z}^d$ if and only if they are satisfied for $e_1, \ldots, e_d$ - that is, if

$$1 \quad \leq \quad \sum_{j \ s.t. \ supp(\phi_j) \ni i} s_j \tag{3.0.1}$$

for all $i \in [d]$.

Proof. The "only if" direction is obvious, since $e_1, \ldots, e_d$ are subgroups of $\mathbb{Z}^d$.

Conversely, suppose (3.0.1) holds. Let H be some subgroup of $\mathbb{Z}^d$ with rank h. Then H can be represented as the integer span of columns of some $d \times h$ matrix $\mathcal{H}$. Since $\mathcal{H}$ has rank h, we can extract a full-rank $h \times h$ submatrix $\mathcal{H}_\mathcal{R}$ by extracting a subset of its rows; denote this subset $\mathcal{R}$.

Notice that all ϕ_j does is extract a subset some of $\mathcal{H}$'s rows, including $\sum_{i \in \mathcal{R}} \mathbf{1}_{i \in \text{supp}(\phi_j)}$ rows from $\mathcal{H}_\mathcal{R}$, where we use the notation $\mathbf{1}_x$ to denote an indicator function that has value

1 when x is true and 0 otherwise.. As the rows extracted from $\mathcal{H}_{\mathcal{R}}$ are guaranteed by construction to be linearly independent, we get the following lower bound on the rank of $\phi_j(H)$:

$$\operatorname{rank}\left(\phi_j(H)\right) \geq \sum_{i\in\mathcal{R}} \mathbf{1}_{i\in \operatorname{supp}(\phi_j)} \tag{3.0.2}$$

which means that

$$\begin{aligned}
\sum_{j\in[n]} s_j \operatorname{rank}(\phi_j(H)) &\geq \sum_{j\in[n]} s_j \sum_{i\in\mathcal{R}} \mathbf{1}_{i\in\operatorname{supp}(\phi_j)} && \text{substituting (3.0.2)}\\
&= \sum_{j\in[n]} \sum_{i\in\mathcal{R}} s_j \mathbf{1}_{i\in\operatorname{supp}(\phi_j)}\\
&= \sum_{i\in\mathcal{R}} \sum_{j\in[n]} s_j \mathbf{1}_{i\in\operatorname{supp}(\phi_j)}\\
&= \sum_{i\in\mathcal{R}} \sum_{j \text{ s.t. } \operatorname{supp}(\phi_j)\ni i} s_j\\
&\geq \sum_{i\in\mathcal{R}} 1 && \text{substituting (3.0.1)}\\
&= h
\end{aligned}$$

as desired. □

As a result, (3.0.1) provides a description of the Brascamp-Lieb polytope in the projective case. In this case, Theorem 5 is equivalent to the *AGM bound* [2]which provides the following intuition for the constraints 3.0.1: define a hypergraph $\mathscr{H}$ whose vertices are the set of loop indices $i \in [d]$, and whose hyperedges are the supports of the ϕ_j. If assigning weights s_j to each hyperedge corresponding to ϕ_j forms a fractional edge cover for $\mathscr{H}$ (that is, the sum of the weights of the hyperedges touching every vertex is at least 1), the rank constraints are satisfied, and the inequality holds.

Combining Theorems and 6 and 11 gives us the following communication lower bound

$$\Omega\left(\frac{\prod_{i\in[d]} L_i}{M^{\sum_{j\in[n]} s_j - 1}}\right)$$

where $s_j \geq 0$ are the solution to the linear program:

$$\min \sum s_j \qquad \text{subject to} \qquad 1 \leq \sum_{j \text{ s.t. } \operatorname{supp}(\phi_j)\ni i} s_j \qquad \forall i \in [1..d] \tag{3.0.3}$$

Thinking of the ϕ_i as 0-1 vectors with 1s in the indices contained in its support, and letting $\vec{s}$ denote the vector $[s_1, ..., s_n]^T$, we can rewrite the linear program (omitting nonnegativity constraints) as follows: minimize $\vec{1}^T \vec{s}$ subject to:

$$\begin{bmatrix} | & & | \\ \phi_1 & \cdots & \phi_n \\ | & & | \end{bmatrix} \vec{s} \geq \vec{1}\ . \tag{3.0.4}$$

Now that we have a lower bound, we would like to find an actual tiling that attains it in order to show that it is tight. Given that this problem is a generalization of matrix multiplication, we will assume as an ansatz that a rectangular tiling is optimal: that is, the that the optimal tile is a hyperrectangle of dimensions $b_1 \times ... \times b_d$, where the b_i are constants which we wish to determine.

We wish to select a tile whose volume (that is, $\prod_{i\in\{1..d\}} b_i$) is as large as possible, but we are subject to memory limitations: the subsets of each array that are used must fit in cache. Since the subsets of array A_i required to complete the operations in this hyperrectangle are of size $\prod_{j\in\text{supp}(\phi_i)} b_j$, we obtain the constraint (again, ignoring constant factors) $\prod_{j\in\text{supp}(\phi_i)} b_j \leq M$. Taking logs base M and letting λ_i denote $\log_M b_i$, we obtain the following linear program: maximize $\vec{1}^T [\lambda_1, ..., \lambda_d]$ subject to:

$$\begin{bmatrix} - & \phi_1 & - \\ & \vdots & \\ - & \phi_n & - \end{bmatrix} \begin{bmatrix} \lambda_1 \\ ... \\ \lambda_d \end{bmatrix} \leq \vec{1}\ . \tag{3.0.5}$$

Taking the dual gives us (3.0.4), which implies that this tiling obtains the lower bound.

It is easy to imagine this is completes the task: we both have a lower bound, and a tiling that obtains the lower bound. Unfortunately, the lower bound is not always attainable.

Consider, for instance, the matrix multiplication example. Solving the LP (3.0.3) gives the Loomis-Whitney bound (2.3.4): multiplying an $m \times k$ and $k \times n$ matrix requires at least $\Omega(mnk/\sqrt{M})$ communication on a two-level memory hierarchy model with fast memory size M. In the case of matrix-vector multiply, however, $n = 1$, so the lower bound becomes $\Omega(mk/\sqrt{M})$, which is clearly not attainable since it is smaller than the size of one of the input matrices.

In order to account for this, we can encode the constraint $b_i \leq L_i$ in the LP (3.0.5), but it is not clear whether or not this is optimal. This chapter will show that it is. Specifically, our contributions in this chapter are:

- a family of lower bounds for projective loop nests *allowing for small loop indices,* and
- a proof that that the tiling generated by the LP (3.0.5), augmented with constraints $b_i \leq L_i$, always attains a communication lower bound for projective ϕ_j.

The results in this chapter were previously published as [19].

3.1 The Lower Bound

3.1.1 One Small Index

We will start our approach to small loop bounds by considering the case when all loop indices but one are assumed to be bounded by arbitrarily large values. Our approach will be to (a) find an upper bound for a tile restricted to single "slice" of the iteration space formed by fixing the loop index with a small bound, (b) calculate an upper bound for the entire tile by summing individual slice bounds together over all possible values of the same index, and (c) divide the total number of operations by the aforementioned quantity to achieve a communication lower bound.

Let us first consider the case where a single loop bound - say, L_1, the upper bound on x_1 - is small, and the others are large. We may assume without loss of generality that $L_1 \leq M$; if the opposite is true, then L_1 would be large enough for the analysis from the introduction to this chapter to apply, as any tile whose memory footprint is at most M would fit in the L_1 dimension. Furthermore, suppose without loss of generality that $\phi_1, ..., \phi_p$ (for some integer p) all contain x_1 and $\phi_{p+1}, ..., \phi_n$ do not. We will now find a communication lower bound for the subset of instructions whose x_1 index is fixed (since the loop bounds are constant and therefore independent of x_1, the result is the same for all possible values of x_1).

Let $\phi'_1, ..., \phi'_p$ be the functions with x_1 removed. For instance, if $\phi_1 = (x_1, x_2, x_3)$, then $\phi'_1 = (x_2, x_3)$. A communication lower bound for a single "slice" of operations with x_1 fixed can be found by using LP 3.0.3, with the ϕ replaced with ϕ', to compute an upper bound for the max tile size:

$$\min \sum \hat{s}_j \qquad \text{subject to} \qquad 1 \leq \sum_{j \text{ s.t. } \operatorname{supp}(\phi'_j)\ni i} \hat{s}_j \qquad \forall i \in [1..d]$$

This amounts to removing the first row in the constraint matrix of the LP (3.0.4).

To find a upper bound for the size of a tile, we sum over the upper bounds for the size each of its slices, each of which corresponds to a single value of x_1. Let $\phi_1|_{x_1=k}, ..., \phi_n|_{x_1=k}$ be the functions with x_1 fixed to k. Then, the maximum tile size is found by maximizing the following quantity (with V representing the tile):

$$\sum_{i\in[L_1]} \prod_{j\in[n]} |\phi_j|_{x_1=i}(V)|^{\hat{s}_j} \quad = \quad M^{\sum_{i\in[p+1,n]} \hat{s}_i} \sum_{i\in[L_1]} \prod_{j\in[p]} |\phi_j|_{x_1=i}(V)|^{\hat{s}_j} \tag{3.1.1}$$

subject to:

$$\sum_{i\in[L_1]} |\phi_j|_{x_1=i}(V)| \leq M \qquad \forall j \in [p] \ . \tag{3.1.2}$$

We maximize (3.1.1) subject to the constraints (3.1.2), and compute the maximum tile size, as follows:

Lemma 12. *The maximum tile size for a tile V, subject to the constraints that (a) that $\phi_i(V) \leq M$ for all i and (b) the set of all distinct x_1-coordinates of elements of V is at most L_1 in cardinality (i.e. the tile fits inside the loop bounds), is bounded above by M^κ, where*

$$\kappa = \max\left\{\sum_{i=1}^{n} \hat{s}_i + \beta_1\left(1 - \sum_{i=1}^{p} \hat{s}_i\right), \sum_{i=1}^{n} \hat{s}_i\right\} .$$

Proof. There are three cases: □

1. If $\sum_{i\in[p]} \hat{s}_i < 1$, the maximum of the quantity (3.1.1) is achieved when we distribute the weight across terms in the sum, i.e. for all $j \in [1..p]$, let $|\phi_j|_{x_1=i}(V)| = M/L_1$ for all $i \in [1..L_1]$, which leads to a tile size of M^κ where

$$\kappa := \sum_{i=1}^{n} \hat{s}_i + \beta_1\left(1 - \sum_{i=1}^{p} \hat{s}_i\right) \tag{3.1.3}$$

and $\beta_1 = \log_M L_1$.

a) If $\sum_{i\in[p]} \hat{s}_i > 1$,the maximum is achieved when we concentrate the entire weight into one term of the sum (i.e. for all $j \in [1..p]$, let $|\phi_j|_{x_1=i'}(V)| = M$ for some i' and let $|\phi_j|_{x_1=i}(V)| = 0$ for $i \neq i'$), which leads to a tile size of M^κ where

$$\kappa := \sum_{i=1}^{n} \hat{s}_i . \tag{3.1.4}$$

b) If $\sum_{i\in[p]} \hat{s}_i = 1$, then both (3.1.3) and (3.1.4) are equal. Furthermore, since the only difference between $\hat{s}$ and s is that the latter must satisfy the additional constraint $\sum_{i\in\{1..p\}} s_i \geq 1$ in the constraint (which is satisfied in this case by $\hat{s}$ as well), we get an upper bound of $M^{\sum_{i=1}^{n}\hat{s}_i} = M^{\sum_{i=1}^{n} s_i}$ immediately from (3.0.3).

Proof. For convenience, denote $|\phi_i|_{x_1=x_i'}(V)|$, the slice of V corresponding to x_1', as $y_{i,x_1'}$. We want to maximize

$$\sum_{x_1=1}^{L_1} y_{1,x_1}^{\hat{s}_1} \cdots y_{p,x_1}^{\hat{s}_p}$$

subject to

$$\sum_{x_1=1}^{L_1} y_{i,x_1} - M \leq 0 \qquad \forall i \in [p] .$$

Without loss of generality, assume all the $\hat{s}_i$ are positive; if $\hat{s}_i = 0$, then we can remove y_{i,x_1} from both the statement of the maximization problem (e.g. by setting it to 1 for all x_i) and from the quantities (3.1.3) and (3.1.4) without affecting the rest of the proof.

Since any slack in any one of the above inequalities can be removed by increasing one of the y_{i,x_i}, and doing so will only increase the quantity we're trying to maximize, we can take these inequalities to be equalities. The Lagrange multipliers for this problem are:

$$\begin{aligned}\mathcal{L} &= \sum_{x_1=1}^{L_1} y_{1,x_1}^{\hat{s}_1} \dots y_{p,x_1}^{\hat{s}_p} \\ &\quad -\lambda_1 \left(\sum_{x_1=1}^{L_1} y_{1,x_1} - M \right) \\ &\quad \vdots \\ &\quad -\lambda_p \left(\sum_{x_1=1}^{L_1} y_{p,x_1} - M \right) \ .\end{aligned}$$

Setting the gradient (with respect to both $y_{i,j}$ and λ_i) to 0, and looking at the derivative with respect to $y_{i,j}$, we get:

$$\hat{s}_i y_{1,j}^{\hat{s}_1} ... y_{i-1,j}^{\hat{s}_{i-1}} y_{i,j}^{\hat{s}_i - 1} y_{i+1,j}^{\hat{s}_{i+1}} ... y_{p,j}^{\hat{s}_p} = \lambda_i \ . \tag{3.1.5}$$

These equations are invariant in j: that is, no matter which value j we fix x_1 to, the set of equations that $y_{i,j}$ must satisfy are identical (this intuitively follows from symmetry across the x_i).

As a result, we may assume $\lambda_i \neq 0$; if it is in fact zero, then the quantity we're trying to maximize would be zero, which clearly cannot be the case since we can construct a tile containing only one element (i.e. with our objective being 1) that satisfies all the constraints of the maximization problem.

In particular, $\lambda_i y_{i,j}/\hat{s}_i = y_{1,j}^{\hat{s}_1} ... y_{p,j}^{\hat{s}_p}$ must remain invariant as i varies (with a fixed j), which implies that for any i_1, i_2, j,

$$\frac{\lambda_{i_1} y_{i_1,j}}{\hat{s}_{i_1}} = \frac{\lambda_{i_2} y_{i_2,j}}{\hat{s}_{i_2}}$$

implying that the ratio between $y_{i,j}$ for two different values of i is independent of the j (i.e. slice) we choose, remaining fixed at

$$\frac{y_{i_1,j}}{y_{i_2,j}} = \frac{\lambda_{i_2} \hat{s}_{i_1}}{\lambda_{i_1} \hat{s}_{i_2}}$$

Therefore, the point we're trying to solve for satisfies this relationship:

$$y_{i,j} = \frac{\lambda_1 \hat{s}_i}{\lambda_i \hat{s}_1} y_{1,j} \tag{3.1.6}$$

For any given j, one of two cases must hold: either $y_{i,j} = 0$ for all i (in which case the tile does not intersect at all with the slice $x_1 = j$) or all $y_{i,j}$ are nonzero, and we can substitute (3.1.6) into (3.1.5) to get:

$$\begin{aligned}
\frac{\lambda_i}{\hat{s}_i} &= y_{1,j}^{\hat{s}_1} \dots y_{i-1,j}^{\hat{s}_{i-1}} y_{i,j}^{\hat{s}_i - 1} y_{i+1,j}^{\hat{s}_{i+1}} \dots y_{p,j}^{\hat{s}_p} \\
&= \frac{\prod_{k=1}^{p} y_{k,j}^{\hat{s}_k}}{y_{i,j}} \\
&= \frac{\prod_{k=1}^{p} \left(\frac{\lambda_1 \hat{s}_k}{\lambda_k \hat{s}_1} y_{1,j} \right)^{\hat{s}_k}}{\frac{\lambda_1 \hat{s}_i}{\lambda_i \hat{s}_1} y_{1,j}} \\
&= y_{1,j}^{-1+\sum_k \hat{s}_k} \frac{\lambda_i \hat{s}_1}{\lambda_1 \hat{s}_i} \prod_{k=1}^{p} \left(\frac{\lambda_1 \hat{s}_k}{\lambda_k \hat{s}_1} \right)^{\hat{s}_k} \\
&= y_{1,j}^{-1+\sum_k \hat{s}_k} \frac{\lambda_i \hat{s}_1}{\lambda_1 \hat{s}_i} \left(\frac{\lambda_1}{\hat{s}_1} \right)^{\sum_k \hat{s}_k} \prod_{k=1}^{p} \left(\frac{\hat{s}_k}{\lambda_k} \right)^{\hat{s}_k}
\end{aligned}$$

Canceling $\frac{\lambda_i}{\hat{s}_i}$ from both sides, and moving the first term in the last expression over to the left, we get

$$y_{1,j}^{1-\sum_k \hat{s}_k} = \left(\frac{\lambda_1}{\hat{s}_1} \right)^{\sum_k \hat{s}_k - 1} \prod_{k=1}^{p} \left(\frac{\hat{s}_k}{\lambda_k} \right)^{\hat{s}_k} .$$

We may assume that $1 - \sum_{k=1}^{p} \hat{s}_k$ is nonzero, as the case when it is zero is covered by case (3) above. Therefore, since the right hand side is independent of j, it follows that all nonzero values of $y_{1,j}$ are equal. Since $y_{1,j}$ determines the value of $y_{i,j}$ for all i via (3.1.6), it follows that each $y_{i,j}$ must either be (a) equal to some nonzero constant independent of j or (b) be equal to zero, if and only if all $y_{i',j}$ for the same j must also be zero.

Let the number of j such that $y_{1,j} \neq 0$ be ϑ, which must fall between 1 and L_1 inclusive (since the number of slices is at most equal to the loop bound corresponding to the dimension we're summing over). Therefore, in order to satisfy (a), the remaining $y_{i,j}$ must be equal to

$$y_{i,j} = \frac{M}{\vartheta} .$$

Substituting this into (3.1.1), we get that the max tile size is:

$$M^{\sum_{i \in [p+1,n]} \hat{s}_i} \vartheta \prod_{i=1}^{p} \left(\frac{M}{\vartheta} \right)^{\hat{s}_i} = M^{\sum_{i=1}^{n} \hat{s}_i} \vartheta^{1-\sum_{i=1}^{p} \hat{s}_i}$$

so the log (base M) of tile size is:

$$\sum_{i=1}^{n} \hat{s}_i + (\log_M \vartheta) \left(1 - \sum_{i=1}^{p} \hat{s}_i \right) .$$

Therefore, either $1 - \sum_{i=1}^{p} \hat{s}_i$ is positive, in which case the maximum occurs when we set ϑ to L_1, giving (recall that $\beta_1 = \log_M L_1$):

$$\sum_{i=1}^{n} \hat{s}_i + \beta_1 \left(1 - \sum_{i=1}^{p} \hat{s}_i\right) \ ,$$

or $1 - \sum_{i=1}^{p} \hat{s}_i$ is negative, in which case the maximum occurs at $\vartheta = 1$, in which case we get

$$\sum_{i=1}^{n} \hat{s}_i \ .$$

as desired. □

3.1.2 Multiple Small Bounds

We now generalize the results from Section 3.1.1 to the case where multiple loop bounds are taken to be small. Suppose that the loops indexed by x_i have bounds L_i. Let $R_j \subseteq \{1..n\}$ denote the set of indices i such that $\text{supp}(\phi_i)$ contains x_j.

As before, our approach considers the communication lower bound for a "slice" - that is, a subset of the iteration polytope formed by restricting certain loop indices to fixed values - and summing these slice lower bounds over all possible values of the fixed indices. This time, however, each slice will be formed by simultaneously fixing multiple indices, which we assume without loss of generality are x_1 through x_q (the following argument holds for any q, and is independent of the actual value of q). As was the case in the single-variable case, an upper bound on max tile size for a single slice is given by $M^{\sum_{j\in\{1..n\}} \hat{s}_j}$,where $\hat{s}_j$ are any nonnegative numbers that satisfy $1 \leq \sum_{j \text{ s.t. } \text{supp}(\phi'_j)\ni x_i} \hat{s}_j$, where ϕ'_j now corresponds to removing $x_1, ..., x_q$ from ϕ_j (or, alternatively, chopping off the first q rows of the HBL LP constraint matrix (3.0.4)).We now develop an analog to Lemma 12 in order to maximize the sum of the slices over $\{x_1, ..., x_q\} \in \{1..L_1\} \times ... \times \{1..L_q\}$. Our main result is as follows:

Theorem 13. *Let $q \in [1..d]$, and $\hat{s}_i$ be any nonnegative numbers such that $1 \leq \sum_{j \text{ s.t. } supp(\phi'_j)\ni x_i} \hat{s}_j$, where ϕ'_j is obtained by removing $x_1, ..., x_q$ from ϕ_j. Then M^k, where*

$$k = \sum_{i=1}^{n} \hat{s}_i + \sum_{j\in[q] \text{ s.t. } \sum_{i\in R_j} \hat{s}_i \leq 1} \left[\beta_j \left(1 - \sum_{i\in R_j} \hat{s}_i\right)\right]$$

represents an upper bound on the tile size.

Notice that this theorem holds for all possible q, as well as reorderings of the variables. As a result, this lemma in fact generates 2^d separate upper bounds for tile size (one for each subset $\mathscr{Q}$ of indices that we hold to be small). Therefore, the smallest upper bound on tile size (which corresponds to the largest lower bound on communication) we can achieve in this manner is $M^{\hat{k}}$ for

$$\hat{k} = \min_{\mathscr{Q} \subseteq [d]} \sum_{i=1}^{n} \hat{s}_{\mathscr{Q},i} + \sum_{j \in \mathscr{Q} \text{ s.t. } \sum_{i \in R_j} \hat{s}_i \leq 1} \left[\beta_j \left(1 - \sum_{i \in R_j} \hat{s}_{\mathscr{Q},i} \right) \right]$$

where $\hat{s}_{\mathscr{Q},i}$ is the solution to the HBL LP (3.0.4) with the rows indexed by elements of $\mathscr{Q}$ removed.

Notice that this theorem holds for all possible q, as well as reorderings of the variables. As a result, this lemma in fact generates 2^d separate upper bounds for tile size (one for each subset $\mathscr{Q}$ of indices that we hold to be small). Therefore, the smallest upper bound on tile size (which corresponds to the largest lower bound on communication) we can achieve in this manner is $M^{\hat{k}}$ for

$$\hat{k} = \min_{\mathscr{Q} \subseteq [d]} \sum_{i=1}^{n} \hat{s}_{\mathscr{Q},i} + \sum_{j \in \mathscr{Q} \text{ s.t. } \sum_{i \in R_j} \hat{s}_i \leq 1} \left[\beta_j \left(1 - \sum_{i \in R_j} \hat{s}_{\mathscr{Q},i} \right) \right]$$

where $\hat{s}_{\mathscr{Q},i}$ is the solution to the HBL LP (3.0.4) with the rows indexed by elements of $\mathscr{Q}$ removed.

Proof. By induction on q. The base case, for $q = 1$, is simply Lemma 12.

Let $\hat{s}'_i$ be defined as $\hat{s}_{[q-1],i}$. Suppose for induction that M^k, for

$$k = \sum_{i=1}^{n} \hat{s}'_i + \sum_{j \in [q-1] \text{ s.t. } \sum_{i \in R_j} \hat{s}_i \leq 1} \left[\beta_j \left(1 - \sum_{i \in R_j} \hat{s}'_i \right) \right]$$

represents an upper bound on the tile size.

We start by finding an upper bound on the tile size, as before, by summing over several "slices", each being defined as the subset of the elements where x_1 through x_q are set to fixed values.

We begin by generalizing the notion of slices to the case where multiple indices may be small. As before, let $\phi_i|_{\{\hat{x}_1,\ldots,\hat{x}_q\}}$ denote ϕ_i with x_j fixed to $\hat{x}_j$ for all $j \in [q]$. By definition, as ϕ_i only depends on indices in its support, $\phi_i|_{\{x_1,\ldots,x_q\}}$ must be identical to $\phi_i|_{\{x_1,\ldots,x_q\} \cap \mathrm{supp}(\phi_i)}$.

We wish to maximize the size of the entire tile - that is, the sum of all the sizes of the slices:

$$\sum_{x_1 \in [1..L_1], \ldots, x_q \in [1..L_q]} |\phi_1|_{\{x_1,\ldots,x_q\}}(V)|^{\hat{s}_1} \ldots |\phi_n|_{\{x_1,\ldots,x_q\}}(V)|^{\hat{s}_n}$$

subject to the memory constraints

$$\sum_{x_k \in [1..L_k] \text{ for } k \in [1..q] \cap \mathrm{supp}(\phi_i)} |\phi_i|_{\{x_1,\ldots,x_q\}}(V)| \leq M \qquad \forall i \in \bigcup_{j \in [1..q]} R_j \ .$$

As before, we will simplify our notation by defining $y_{j,\{x_1,...,x_q\}} := |\phi_j\restriction_{\{x_1,...,x_q\}}(V)|$. Our optimization problem therefore can be rewritten as maximizing:

$$\sum_{x_1\in[1..L_1],...,x_q\in[1..L_q]} y^{\hat{s}_1}_{1,\{x_1,...,x_q\}} \cdots y^{\hat{s}_n}_{n,\{x_1,...,x_q\}} \tag{3.1.7}$$

subject to the constraints:

$$1 \leq \sum_{x_k\in[1..L_k] \text{ for } k\in[q]\cap\text{supp}(\phi_i)} y_{i,\{x_1,...,x_q\}} \leq M \qquad \forall i \in \bigcup_{j\in[q]} R_j \ . \tag{3.1.8}$$

The definition of $\phi_i\restriction_{\{\hat{x}_1,...,\hat{x}_q\}}$ (and therefore of $y_{j,\{x_1,...,x_q\}}$) requires us to further impose an additional constraint on the solution: for all i, the value of $y_{i,\{x_1,...,x_q\}}$ must remain independent of indices not in the support of ϕ_i. Formally, if $x_k \notin \text{supp}(\phi_i)$, then

$$y_{i,\{x_1,...x_{k-1},a,x_{k+1},...,x_q\}} = y_{i,\{x_1,...x_{k-1},,b,x_{k+1},....,x_q\}} \tag{3.1.9}$$

for any a, b. Our approach will be to find a candidate solution which ignores this constraint, and then to show that this candidate solution actually does satisfy (3.1.9) (i.e. that this constraint is redundant).

Furthermore, in order to make it easier to reason about the constraints (3.1.8), we will multiply them all by the appropriate values in order to ensure that the sum is over the same set of variables: x_1 through x_q:

$$\prod_{j\in[q]\setminus\text{supp}(\phi_i)} L_j \leq \sum_{x_1\in[1..L_1],...,x_q\in[1..L_q]} y_{i,\{x_1,...,x_q\}} \leq M \prod_{j\in[q]\setminus\text{supp}(\phi_i)} L_j \qquad \forall i \in \bigcup_{j\in[q]} R_j \ . \tag{3.1.10}$$

Since our goal is to find an upper bound on the tile size, which is the result of this constrained maximization problem, we can remove the lower bound constraints on

$$\sum_{x_1\in[1..L_1],...,x_q\in[1..L_q]} y_{i,\{x_1,...,x_q\}}$$

(i.e. the leftmost inequality in (3.1.10)) without affecting correctness.

The resulting problem is almost identical to that of Lemma 12, except with different limits (one may think of this 'flattening' the q-dimensional tensor $x_1, ..., x_q$ into a single vector in order to get a single sum as we did in the previous section). Recall that none of the steps we used to compute the maximum in our proof of Lemma 12 actually used the value of the right sides of the constraints, since all those constants were all differentiated away as a constant factor when taking gradients; as a result, the same result applies here. Specifically, the maximum is obtained at a point specified as follows: select some subset $\mathscr{S} \subseteq \{1..L_1\} \times ... \times \{1..L_q\}$ of integer tuples, which represent x_i-indices for which $y_{i,\{x_1,..,x_q\}}$ will be nonzero. For each $\{x_1, .., x_q\}$ in $\mathscr{S}$, $y_{i,\{x_1,..,x_q\}}$ must be equal to a constant value independent of $\{x_1, ..., x_q\}$. In order to maximize (3.1.7), we set constraint (3.1.8) to obtain:

$$y_{i,\{x_1,\ldots,x_q\}} = \frac{M}{|\mathscr{S}_i|}\ \forall i \tag{3.1.11}$$

where $\mathscr{S}_i$ is ϕ_i (restricted to $x_1 \ldots x_q$) applied to $\mathscr{S}$.

For indices not in $\mathscr{S}$, set $y_{i,\{x_1,\ldots,x_q\}}$ to zero for all i. The resulting upper bound for tile size is therefore:

$$\begin{aligned}\sum_{x_1,\ldots,x_q \in \mathscr{S}} \prod_i \left(\frac{M}{|\mathscr{S}_i|}\right)^{\hat{s}_i} &= |\mathscr{S}| \prod_i \left(\frac{M}{|\mathscr{S}_i|}\right)^{\hat{s}_i} \\ &= \frac{|\mathscr{S}|}{\prod_i |\mathscr{S}_i|^{\hat{s}_i}} M^{\sum_i \hat{s}_i} \end{aligned} \tag{3.1.12}$$

where the first equality is a result of the independence of the summand with x, with the number of nonzero terms in the sum being $|\mathscr{S}|$.

Claim: without loss of generality, we can assume that $\mathscr{S}$ is a rectangle; that is, it can be written as set $C_1 \times \cdots \times C_q$ for some sets $C_i \subseteq [L_i]$

Proof of claim: Suppose not. Then there exist points $x', x'' \in \mathscr{S}$ such there exists some point $x^* \notin \mathscr{S}$, where each x^*_j is equal to x'_j for all j except a single value j^*, at which it takes on the value of x''_j. To see why this is true, take any two distinct $x', x'' \in \mathscr{S}$, and repeatedly change one component of x' to match the corresponding component of x'', stopping when either $x' = x''$, or $x' \notin \mathscr{S}$. In the latter case, set $x^* = x'$, and let x' denote its immediate predecessor in this process. If we never end up with an x^* for any distinct pairs of x' and x'' in $\mathscr{S}$, then $\mathscr{S}$ must be a rectangle.

Our goal will be to show that this configuration is suboptimal. Consider the set of functions ϕ_i for $i \in R_{j^*}$, that is, the set of functions containing x_{j^*}.

Let us consider the following categories, distinguished by how ϕ_i maps x', x'', and x^*. □

1. $\phi_i(x') = \phi_i(x^*) \neq \phi_i(x'')$. Notice that replacing x'' with x^* in $\mathscr{S}$ either reduces $|\mathscr{S}_i|$ by one (if there is no other $x^\dagger$ such that $\phi_i(x^\dagger) = \phi_i(x''))$ or keeps it the same (if such an $x^\dagger$ exists); notice that in the latter case, adding x^* to $\mathscr{S}$ keeps $|\mathscr{S}_i|$ constant. We denote these cases (1a) and (1b) respectively.

   a) $\phi_i(x'') = \phi_i(x^*) \neq \phi_i(x')$. Analogously to case (1), replacing x' with x^* either reduces $\mathscr{S}_i$ (case (2a)) or keeps it the same (case (2b)).

   b) ϕ_i maps x', x'', and x^* to three distinct points.

   c) $\phi_i(x') = \phi_i(x'') = \phi_i(x^*)$.

   d) $\phi_i(x') = \phi_i(x'') \neq \phi_i(x^*)$. Notice that this category must be empty: If $x''_j = x'_j$, then by definition this quantity is also x^*; therefore, going to $*$ from $''$ can only make the number of agreements better under any projection.

Proof. In the above categories, i satisfying (1) and (4) implies that $i \notin R_{j^*}$, while i satisfying (2) and (3) implies that $i \in R_{j^*}$. We will show that $\mathscr{S}$ is suboptimal by providing strict improvements on it. □

1. If there are any i in category (1a), we replace x'' with x^* in $\mathscr{S}$, reducing $|\mathscr{S}_i|$. In order to see how this change affects the values of $\mathscr{S}_i$ for other i, we first note that for other $i \notin R_{j^*}$, $\phi_i(x') = \phi_i(x^*)$, so this change can only keep constant or decrease $|\mathscr{S}_i|$ for such i. For all i in any of the other categories - (1b), (2ab), (3), or (4) - $|\mathscr{S}_i|$ remains the same. Therefore, as the value of $|\mathscr{S}_i|$ either remains the same or decreases (with at least one strict decrease), and $|\mathscr{S}|$ remains constant, we obtain a strict increase in the value of (3.1.12).

   a) If some i falling into category (3): Denote the set of i such that ϕ_i maps x', x'', and x^* onto different values as Q. We will split into two cases, based on the values of $\sum_{i \in R_{j^*}} \hat{s}_i$:

      i. Suppose $\sum_{i \in R_{j^*}} \hat{s}_i \geq 1$. Consider the assignment to the $y_{i,x}$ given by $\mathscr{S}$; its objective (3.1.12) is:

$$\sum_{x_1 \in [L_1], \ldots, x_{j^*-1} \in [L_{j^*-1}], x_{j^*+1} \in [L_{j^*+1}], \ldots, x_q \in [L_q]} \left(\sum_{x_{j^*} \in [L_{j^*}]} y_{1,\{x_1,\ldots,x_q\}}^{\hat{s}_1} \cdots y_{n,\{x_1,\ldots,x_q\}}^{\hat{s}_n} \right)$$

Factoring the innermost term into terms that are constant w.r.t. x_{j^*} and those that are not, we can rewrite this as:

$$\sum_{x_1 \in [L_1], \ldots, x_{j^*-1} \in [L_{j^*-1}], x_{j^*+1} \in [L_{j^*+1}], \ldots, x_q \in [L_q]} \left(\prod_{i \in [n] \setminus R_j^*} y_{i,\{x_1,\ldots,x_q\}}^{\hat{s}_i} \sum_{x_{j^*} \in [L_{j^*}]} \prod_{i \in R_j^*} y_{i,\{x_1,\ldots,x_q\}}^{\hat{s}_i} \right).$$

Let us restrict our attention a single "slice": that is, an instance of the term

$$\sum_{x_{j^*} \in [L_{j^*}]} \prod_{i \in R_{j^*}} y_{i,\{x_1,\ldots,x_q\}}^{\hat{s}_i} \tag{3.1.13}$$

with fixed values for x_1 through x_q, excluding x_{j^*}. By equality constraints, we get that all the nonzero values of $y_{i,\{x_1,\ldots,x_q\}}^{\hat{s}_i}$ must be equal to a constant independent of $x_1, \ldots, x_q$ (but dependent on i). Let $m_i = \sum_{x_i^* \in [L_{j^*}]} y_{i,\{x_1,\ldots,x_q\}}$, and σ denote the number of x_{j^*} such that $(x_1, \ldots, x_q)$, with all coordinates except x_{j^*} set to our fixed values, are in $\mathscr{S}$ (and therefore, nonzero terms in the above sum (3.1.13)); this restricts the nonzero values of $y_{i,\{x_1,\ldots,x_q\}}$ to m_i/σ. Therefore, we may rewrite the above sum as:

$$\begin{aligned} \sum_{x_{j^*} \in [L_{j^*}]} \prod_{i \in R_{j^*}} y_{i,\{x_1,\ldots,x_q\}}^{\hat{s}_i} &= \sigma \prod_{i \in R_{j^*}} \left(\frac{m_i}{\sigma} \right)^{\hat{s}_i} \\ &= \sigma^{1 - \sum_{i \in R_{j^*}} \hat{s}_i} \prod_{i \in R_{j^*}} m_i^{\hat{s}_i} \end{aligned}$$

As $\sum_{i\in R_{j^*}} \hat{s}_i \geq 1$, the exponent of σ in the above expression is nonpositive; therefore, this term is bounded above by

$$\prod_{i\in R_{j^*}} m_i^{\hat{s}_i}$$

which we get when we set σ to 1. As this upper bound holds individually for each "slice", the value of the objective (3.1.12) is upper bounded by setting σ to 1 for every slice, i.e. adding an additional constraint forcing L_{j^*} to 1, which is equivalent to removing x_{j*} from the problem entirely. Applying our inductive hypothesis, we get that an upper bound is $M^{k'}$ where

$$k' = \sum_{i=1}^{n} \hat{s}'_i + \sum_{j\in[1..q]\setminus\{j^*\} \text{ s.t. } \sum_{i\in R_j} \hat{s}'_i \leq 1} \left[\beta_j \left(1 - \sum_{i\in R_j} \hat{s}'_i\right)\right] .$$

Since $\sum_{i\in R_{j^*}} \hat{s}_i \geq 1$, there is no difference between $\hat{s}'_i$ and $\hat{s}_i$ for all i, as the only constraint that the former must satisfy that the latter is not required to is $\sum_{i\in R_{j^*}} \hat{s}_i \geq 1$, which holds regardless in this case. Therefore, we can replace $\hat{s}'_i$ with $\hat{s}_i$ in order to completing the induction for the entire proof of Lemma 13 in this particular case.

ii. Now suppose $\sum_{i\in R_{j^*}} \hat{s}_i < 1$. As $Q \subseteq R_{j^*}$, it immediately follows that $\sum_{i\in Q} \hat{s}_i \leq \sum_{i\in R_{j^*}} \hat{s}_i < 1$. Consider the values of $y_{i,x'}$, $y_{i,x''}$, and y_{i,x^*}; we will show that a reassignment of these three values strictly increases the objective.
Without loss of generality, we will assume $\prod_{i\notin Q} y_{i,x^*}^{\hat{s}_i}$ may be taken as nonzero. Why? Given some i' is not in Q, then by definition $\phi_{i'}$ must map x', x^* to the same point, or x'', x^* to the same point. In the former case, we can set $y_{i',x^*} = y_{i',x'}$ without violating any constraint, as we can substitute the two terms freely in any constraint-sum involving them; in the latter case, the same applies if we set $y_{i',x^*} = y_{i',x''}$. As $x', x'' \in \mathscr{S}$, both $y_{i',x'}$ and $y_{i',x''}$ must be nonzero, so y_{i',x^*} must be nonzero as well. As nonzero values of $y_{i,x}$ are independent of x for all i, we must have

$$y_{i',x^*} = y_{i',x''} = y_{i',x'} \tag{3.1.14}$$

For all i, let the value of $y_{i,x'} + y_{i,x''} + y_{i,x^*}$ be denoted μ_i, and define k', k'', k^* such that $y_{i,x'} = k'\mu_i$ (and likewise for k'', k^*); our starting configuration, with $\mathscr{S}$ containing x', x'' but not x^*, is $k' = k'' = 1/2$, $k^* = 0$. So as not to break any constraints, we will require that the value of $y_{i,x'} + y_{i,x''} + y_{i,x^*}$ stay constant, so we will enforce $k' + k'' + k^* = 1$. The contribution of these three tiles to the objective (3.1.7) is:

$$\begin{aligned}
&\prod_{i\in Q} y_{i,x'}^{\hat{s}_i} \prod_{i\notin Q} y_{i,x'}^{\hat{s}_i} + \prod_{i\in Q} y_{i,x''}^{\hat{s}_i} \prod_{i\notin Q} y_{i,x''}^{\hat{s}_i} + \prod_{i\in Q} y_{i,x^*}^{\hat{s}_i} \prod_{i\notin Q} y_{i,x^*}^{\hat{s}_i} \\
&= \left(\prod_{i\in Q} y_{i,x'}^{\hat{s}_i} + \prod_{i\in Q} y_{i,x''}^{\hat{s}_i} + \prod_{i\in Q} y_{i,x^*}^{\hat{s}_i} \right) \prod_{i\notin Q} y_{i,x'}^{\hat{s}_i}
\end{aligned}$$

with equality following from (3.1.14). We substitute $y_{i,x'} = k'\mu_i$ and the corresponding definitions for $y_{i,x''}, y_{i,x^*}$ to rewrite the above expression as:

$$\begin{aligned}
&\left(\prod_{i\in Q} (k'\mu_i)^{\hat{s}_i} + \prod_{i\in Q} (k''\mu_i)^{\hat{s}_i} + \prod_{i\in Q} (k^*\mu_i)^{\hat{s}_i} \right) \prod_{i\notin Q} y_{i,x'}^{\hat{s}_i} \\
&= \left(k'^{\sum_{i\in Q} \hat{s}_i} \prod_{i\in Q} \mu_i^{\hat{s}_i} + k''^{\sum_{i\in Q} \hat{s}_i} \prod_{i\in Q} \mu_i^{\hat{s}_i} + k^{*\sum_{i\in Q} \hat{s}_i} \prod_{i\in Q} \mu_i^{\hat{s}_i} \right) \prod_{i\notin Q} y_{i,x'}^{\hat{s}_i} \\
&= \left(k'^{\sum_{i\in Q} \hat{s}_i} + k''^{\sum_{i\in Q} \hat{s}_i} + k^{*\sum_{i\in Q} \hat{s}_i} \right) \prod_{i\notin Q} y_{i,x'}^{\hat{s}_i} \prod_{i\in Q} \mu_i^{\hat{s}_i}
\end{aligned}$$

We will leave $y_{i,x'}$ constant for all $i \notin Q$, and we will not vary μ_i, the sum of $y_{i,x'}$, $y_{i,x''}$, and y_{i,x^*}, so it suffices to maximize

$$k'^{\sum_{i\in Q} \hat{s}_i} + k''^{\sum_{i\in Q} \hat{s}_i} + k^{*\sum_{i\in Q} \hat{s}_i}$$

subject to

$$k' + k'' + k^* = 1.$$

As $\sum_{i\in Q} \hat{s}_i < 1$, the solution to this maximization problem is obtained by setting $k' = k'' = k^* = 1/3$; all other assignments (including the current one) are suboptimal. As we do not vary any $y_{i,x}$ for $i \notin Q$ and any x, this change does not affect the constraints corresponding to any other ϕ_i than those in Q, which all must be still satisfied as we do not vary μ_i; therefore, both these assignments satisfy the constraints (3.1.8). Therefore the current assignment under $\mathscr{S}$, with k^* set to 0 and k', k'' set to $1/2$, must be suboptimal, providing us with our contradiction in this case.

b) If there exists i in category (2a), but none in (1a) and (3), we will replace x' with x^* in $\mathscr{S}$, decreasing $|\mathscr{S}_i|$ by one. The values of $|\mathscr{S}_i|$ for other i, in this case, either also decrease (for other is falling in case (2a)), remain the same (for is falling in cases (1b), (2b), (4)), therefore corresponding to a strict improvement in the value of (3.1.12).

c) If we have i in categories (1b), (2b), or (4) (but none in categories (1a), (2a), or (3), all of which were dealt with in earlier cases) add x^* to $\mathscr{S}$; this does not change any value of $\mathscr{S}_i$, but increases $\mathscr{S}$ by 1, leading to a strictly improved solution.

Proof. Each of these cases (except case (3a), which uses the inductive hypothesis), presents a strict improvement to the value of (3.1.12). Therefore, $\mathscr{S}$ must not be optimum, providing a contradiction. We can therefore conclude that optimum value of $\mathscr{S}$ must have no triple $x', x'' \in \mathscr{S}$, $x^* \notin \mathscr{S}$ such that x^* agrees with x' everywhere except one coordinate where it agrees with x'', and therefore $\mathscr{S}$ must be a rectangle, as desired. ■

Now that we've shown that $\mathscr{S}$ is a rectangle, let us assume that its dimensions are $\rho_1, ..., \rho_q$. Then $\mathscr{S}$ has cardinality $\prod_{i\in[q]} \rho_i$, and $\mathscr{S}_i$ has cardinality $\prod_{j\in[q]\cap\text{supp}(\phi_i)} \rho_j$. Substituting into (3.1.12), we get:

$$\begin{aligned}\frac{|\mathscr{S}|}{\prod_i |\mathscr{S}_i|^{\hat{s}_i}} M^{\sum_i \hat{s}_i} &= \frac{\prod_{i\in[q]} \rho_i}{\prod_{i\in[d]} \left(\prod_{j\in[q]\cap\text{supp}(\phi_i)} \rho_j\right)^{\hat{s}_i}} M^{\sum_i \hat{s}_i} \\ &= \frac{\prod_{i\in[q]} \rho_i}{\prod_{j\in[q]} \left(\prod_{i\in R_j} \rho_j^{\hat{s}_i}\right)} M^{\sum_i \hat{s}_i} \\ &= \prod_{j\in[q]} \rho_j^{1-\sum_{i\in R_j} \hat{s}_i} M^{\sum_i \hat{s}_i}\end{aligned}$$

Since we have full control over the value of ρ_i, we can maximize the value of this expression by setting the ρ_i to their maximum possible value, L_i if $1 - \sum_{i\in R_j} \hat{s}_i \geq 0$, and to their minimum possible value, 1, if $1 - \sum_{i\in R_j} \hat{s}_i \leq 0$.

Therefore, the maximum value of our objective (3.1.7) is obtained at:

$$M^{\sum_i \hat{s}_i} \prod_{j\in[q] \text{ s.t. } \sum_{i\in R_j} \hat{s}_i \leq 1} L_j^{1-\sum_{i\in R_j} \hat{s}_i}$$

or equivalently, M^k where

$$k = \sum_{i=1}^{n} \hat{s}_i + \sum_{j\in[q] \text{ s.t. } \sum_{i\in R_j} \hat{s}_i \leq 1} \left[\beta_j \left(1 - \sum_{i\in R_j} \hat{s}_i\right)\right] . \tag{3.1.15}$$

as desired.

Finally, we need to modify our solution to satisfy (3.1.9) with no change to the objective value.

Let $y'_{i,\{x_1,...,x_q\}}$ be $\max_{x_j \text{ s.t. } j\in\text{supp}(\phi_i)} y_{i,\{x_1,...,x_q\}}$, which takes on the value $\frac{M}{|\mathscr{S}_i|}$ if there is some nonzero element of $\mathscr{S}$ that matches $(x_1, ..., x_q)$ at the indices in the support of ϕ_i, and is zero otherwise. In order to show that this modification does not change the value of objective (3.1.7), it suffices to show that

$$y_{1,\{x_1,...,x_q\}}^{\hat{s}_1} \cdots y_{n,\{x_1,...,x_q\}}^{\hat{s}_n} = \left(y'_{1,\{x_1,...,x_q\}}\right)^{\hat{s}_1} \cdots \left(y'_{n,\{x_1,...,x_q\}}\right)^{\hat{s}_n} \tag{3.1.16}$$

Suppose $(x_1, ..., x_q) \in \mathscr{S}$. Both sides are nonzero, and by equality constraint it is obvious that they must be the same.

Suppose $(x_1, ..., x_q) \notin \mathscr{S}$. Clearly the left is zero. Recall that $\mathscr{S}$ is a rectangle; that is, it can be written as set $\{(x_1, ..., x_n) : x_i \in C_i \forall i\}$ for some sets $C_i \subseteq [L_i]$. By definition, there must exist some k such that $x_k \notin C_k$. There must be some some j' such that $\phi_{j'}$ contains x_k; by definition, $y'_{j',\{x_1,...,x_k,...,x_j\}}$ - and therefore, the entire right-hand-side of (3.1.16) - must be zero as well.

Furthermore, in order to show that this solution does not violate any of the constraints, consider

$$\sum_{x_k \in [1..L_k] \text{ for } k \in [q] \cap \text{supp}(\phi_i)} y'_{i,\{x_1,...,x_q\}}$$

By definition, at most $\mathscr{S}_i$ of these terms may be nonzero, and each since must have value $M/|\mathscr{S}_i|$, this term must be at most M, as desired.

Notice that this proof works if we fix any subset of $1..q$ rather than the entire set. In other words, we can freely replace the sum from 1 to q with a sum over any subset of 1 to q and still get a valid upper bound (by changing the sum from $j \in [q]$ to summing over a subset of $[q]$ in equation (3.1.15)). □

3.2 Tiling construction

In this section, we describe an explicit construction of a tiling that achieves the upper bound on tile size (and therefore achieves the lower bound on computation) from section 3.1.

Consider the LP that gives us the tiling in this case. We start with the linear program (3.0.5) and add constraints requiring that the blocks be no larger than the loop bounds (in log-space, $\lambda_i \leq \beta_i$):

$$\max \sum_{i \in d} \lambda_i \quad \text{s.t.} \sum_{i \text{ s.t. } x_i \in \text{supp}(\phi_j)} \lambda_i \leq 1 \ \forall j \in [n], \quad \lambda_i \leq \beta_i \ \forall i \in [q], \quad \lambda_i \geq 0 \ \forall i \in [d]. \tag{3.2.1}$$

Theorem 14. *The rectangular tile with dimensions given by the solution to (3.2.1) has cardinality equal to one of the upper bounds for tile size from Section 3.1 for a loop program defined by the ϕ_j; in other words, the solution to (3.2.1) equals*

$$\sum_{i=1}^{n} \hat{s}_{\mathscr{Q},i} + \sum_{j \in \mathscr{Q} \text{ s.t. } \sum_{i \in R_j} \hat{s}_i \leq 1} \left[\beta_j \left(1 - \sum_{i \in R_j} \hat{s}_{\mathscr{Q},i} \right) \right] \tag{3.2.2}$$

for some $\mathscr{Q} \subseteq [q]$, where $\hat{s}_{\mathscr{Q},i}$ satisfies the constraints of the HBL LP (3.0.4) with the rows indexed by elements of $\mathscr{Q}$ removed:

$$\begin{bmatrix} \multicolumn{3}{l}{\textit{remove rows not in } \mathscr{Q}} \\ | & & | \\ \phi_1 & \cdots & \phi_n \\ | & & | \end{bmatrix} \begin{bmatrix} \hat{s}_1 \\ \vdots \\ \hat{s}_n \end{bmatrix} \geq \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \tag{3.2.3}$$

Let us write the constraints of (3.2.1) in the following fashion:

$$\begin{array}{r} \\ \\ \\ \\ q\left\{\begin{array}{c} \\ \\ \\ \\ \end{array}\right. \end{array} \left[\begin{array}{ccccccc} - & & & \phi_1 & & & - \\ & & & \vdots & & & \\ - & & & \phi_n & & & - \\ 1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \multicolumn{4}{c}{\underbrace{0 \quad 0 \quad \cdots \quad 1}_{q}} & 0 & \cdots & 0 \end{array}\right] \begin{bmatrix} \lambda_1 \\ \vdots \\ \lambda_d \end{bmatrix} \leq \begin{bmatrix} 1 \\ \vdots \\ 1 \\ \beta_1 \\ \vdots \\ \beta_q \end{bmatrix} \tag{3.2.4}$$

The dual of this linear program, with variables $\zeta_1, ..., \zeta_q, s_1, ..., s_n$ is to minimize

$$\sum_{i\in[q]} \beta_i \zeta_i + \sum_{j=1}^{n} s_j \tag{3.2.5}$$

subject to

$$\begin{array}{r} q\left\{\begin{array}{c} \\ \\ \\ \end{array}\right. \\ \\ \\ \end{array} \left[\begin{array}{cccccc} 1 & \cdots & 0 & & & \\ \vdots & \ddots & \vdots & | & & | \\ 0 & \cdots & 1 & \phi_1 & \cdots & \phi_n \\ \vdots & & \vdots & | & & | \\ 0 & \cdots & 0 & & & \end{array}\right] \begin{bmatrix} \zeta_1 \\ \vdots \\ \zeta_q \\ s_1 \\ \vdots \\ s_n \end{bmatrix} \geq \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \tag{3.2.6}$$

(as well as nonnegativity constraints $\zeta_i \geq 0$ for all $i \in [q]$, $s_i \geq 0$ for all $i \in [n]$, which we omit from the matrix for brevity)

We now show that the optimal value of (3.2.5) is equivalent to (3.2.2) for some $\hat{s}_i$ satisfying (3.2.3).

Proof. By induction on q.

For the base case, suppose $q = 0$. This is just the case in introduction to this chapter.

Suppose for induction that the solution to

$$\begin{array}{rcll} \max \displaystyle\sum_i \lambda_i \text{ s.t.} & & & \\ \displaystyle\sum_{i \text{ s.t. } x_i \in \operatorname{supp}(\phi_j)} \lambda_i & \leq & 1 & \forall j \in [n] \\ \lambda_i & \leq & \beta_i & \forall i \in [q-1] \end{array} \tag{3.2.7}$$

takes the form

$$\sum_{i=1}^{n} \hat{s}_{\mathscr{Q},i} + \sum_{j\in\mathscr{Q} \text{ s.t. } \sum_{i\in R_j} \hat{s}_i \leq 1} \left[\beta_j \left(1 - \sum_{i\in R_j} \hat{s}_{\mathscr{Q},i}\right)\right]$$

for some $\mathscr{Q} \subseteq [q-1]$ and $\hat{s}_i$ satisfying (3.2.3).

Consider the LP: minimize (3.2.5) subject to (3.2.6). Denote its solution by ζ'_i, s'_i; we wish to discover the minimum value of the objective (3.2.5).

We will rewrite the LP (3.2.6) in such a way that preserves the optimal value of the objective. First, we remove one variable - say, ζ_q - from it. Since there is no benefit to setting ζ_q any larger than necessary (it increases the objective (3.2.5), and does not come into play in any other constraints) we can fix its value as necessary to ensure that either the qth constraint or the nonnegativity constraint $\zeta_q \geq 0$ is tight. We have two cases:

Case 1: $\sum_{i\in R_q} s'_i \geq 1$. In this case, the qth constraint is satisfied at the optimal point regardless of the value of ζ'_q, so we may set ζ_q to 0. Now, the objective (3.2.5) becomes:

$$\sum_{i=1}^{q-1} \beta_i \zeta_i + \sum_{j=1}^{n} s_j$$

Since the qth constraint is the only one containing ζ_q, we can delete the qth column on the left block of the constraint matrix (3.2.6) and remove ζ_q from the LP entirely. Therefore, the resulting LP is therefore exactly the dual of (3.2.7), which, by inductive hypothesis, has optimal objective value of the form:

$$\sum_{i=1}^{n} \hat{s}_{\mathscr{Q},i} + \sum_{j\in\mathscr{Q} \text{ s.t. } \sum_{i\in R_j} \hat{s}_i \leq 1} \left[\beta_j\left(1 - \sum_{i\in R_j} \hat{s}_{\mathscr{Q},i}\right)\right]$$

for $\mathscr{Q} \subseteq [q-1] \subset [q]$, and $\hat{s}_{\mathscr{Q},i}$ satisfying (3.2.3) as desired.

Case 2: $\sum_{i\in R_q} s'_i < 1$. Without loss of generality, assume this holds for R_1 through R_{q-1} as well (if not, find j such that $\sum_{x\in R_j} s'_i \geq 1$, permute the LP to swap the positions of ζ_j and ζ_q, and proceed to case 1).

Therefore, we may modify the LP by setting ζ_1 to $1 - \sum_{i\in R_1} s_i$ to keep it tight, and do the same with ζ_2 through ζ_q; this does not change the optimal objective value. Removing those constraints (since they've all been encoded into the objective), we get a new objective to replace (3.2.5) in our linear program:

$$\min \sum_{i=1}^{n} s_i + \sum_{j=1}^{q} \left[\beta_j\left(1 - \sum_{i\in R_j} s_i\right)\right]$$

Furthermore, since $\sum_{i\in R_j} s'_i < 1$ for all $j \in [q]$ this objective at its optimizer, $s'_1, ..., s'_q$, is precisely equal to

$$\sum_{i=1}^{n} s'_i + \sum_{j\in[q] \text{ s.t. } \sum_{i\in R_j} \hat{s}_i \leq 1} \left[\beta_j\left(1 - \sum_{i\in R_j} s'_i\right)\right]$$

which is of the same form as (3.2.2).

Furthermore, we may remove the first q constraints from (3.2.6), since our choices for values of $\zeta_1, ..., \zeta_q$ guarantee that they will be satisfied. The resulting constraint matrix is identical to (3.2.3).

Therefore, the tile whose dimensions are given by 3.2.1 attains the lower bound given by Lemma 13 with $\mathscr{Q} = [q]$, as desired. □

3.3 Examples and Applications

We demonstrate several applications of our theory below.

3.3.1 Matrix-Matrix and Matrix-Vector Multiplication

We start by re-deriving the classical lower bound (2.3.4) for the triply-nested-loop matrix multiplication

$$\begin{aligned}&\text{for}\,\{x_1, x_2, x_3\} \in [L_1] \times [L_2] \times [L_d]\\ &\qquad A_1(x_1, x_3) += A_2(x_1, x_2) \times A_3(x_2, x_3)\end{aligned}$$

Our memory accesses are given by the functions:

$$\begin{aligned}\phi_1(x_1, x_2, x_3) &= (x_1, x_3)\\ \phi_2(x_1, x_2, x_3) &= (x_1, x_2)\\ \phi_3(x_1, x_2, x_3) &= (x_2, x_3)\end{aligned}$$

Therefore, the HBL LP is to minimize $s_1 + s_2 + s_3$ subject to

$$\begin{bmatrix}1 & 1 & 0\\ 0 & 1 & 1\\ 1 & 0 & 1\end{bmatrix}\begin{bmatrix}s_1\\ s_2\\ s_2\end{bmatrix} \geq \begin{bmatrix}1\\ 1\\ 1\end{bmatrix}. \tag{3.3.1}$$

The optimal value of this LP is obtained when all the s_i are $1/2$, giving a tile size upper bound of $M^{1/2+1/2+1/2} = M^{3/2}$, which provides the standard $L_1 L_2 L_3 / M^{1l2}$ lower bound.

Now let us consider the case where L_3 may be small, which corresponds to problem sizes approaching matrix-vector multiplications (which occurs $L_3 = 1$). In this case, our tile, which has length $M^{1/2}$ in the L_3 dimension, cannot fit in our iteration space.s

We first find a lower bound. Removing the row corresponding to x_3 from (3.3.1), we get that given any $\hat{s}_i$ satisfying

$$\begin{bmatrix}1 & 1 & 0\\ 0 & 1 & 1\end{bmatrix}\begin{bmatrix}\hat{s}_1\\ \hat{s}_2\\ \hat{s}_3\end{bmatrix} \geq \begin{bmatrix}1\\ 1\\ 1\end{bmatrix} \tag{3.3.2}$$

raising M to the power

$$\max\{\hat{s}_1 + \hat{s}_2 + \hat{s}_3, \hat{s}_1 + \hat{s}_2 + \hat{s}_3 + (\log_M L_3)(1 - \hat{s}_1 - \hat{s}_3)\}$$

represents a valid upper bound on the tile size.

Since (3.3.2) is satisfied when $\hat{s}_2 = 1$ and $\hat{s}_1, \hat{s}_3 = 0$, this term becomes

$$\max\{1, 1 + \log_M L_3\}$$

giving an upper bound of $\max\{M, ML_3\} = ML_3$ (as L_3 is always positive); therefore the communication lower bound is given by

$$\frac{L_1 L_2 L_3}{ML_3} M = L_1 L_2 \ .$$

This is as expected, since we need to read at least $L_1 L_2$, the size of A_2, into fast memory to perform the operation.

Now let us consider the question of finding the tile. Instantiating LP (3.2.1) with the relevant values of $\phi_{1,2,3}$, we get:

$$\begin{aligned} &\max \lambda_1 + \lambda_2 + \lambda_3 \ \text{ s.t.} \\ &\lambda_1 + \lambda_3 \leq 1 \\ &\lambda_1 + \lambda_2 \leq 1 \\ &\lambda_2 + \lambda_3 \leq 1 \\ &\lambda_3 \leq \beta_3 = \log_M L_3 \end{aligned} \tag{3.3.3}$$

There are two cases here: if $\beta_3 \geq 1$, then the last constraint is of no relevance, so the solution becomes $3/2$, as in the case above .

On the other hand, if $\beta_3 \leq 1$, then adding the second and fourth inequalities gives

$$\lambda_1 + \lambda_2 + \lambda_3 \leq 1 + \lambda_3 \leq 1 + \beta_3 \ . \tag{3.3.4}$$

We again split based on whether or not $\beta_3 \geq 1/2$; intuitively, we may consider this a question of whether the L_3 is sufficiently large (at least $\sqrt{M}$) to fit the $\sqrt{M} \times \sqrt{M} \times \sqrt{M}$ tile derived above, or whether we must modify the tile's shape to get it to fit in the L_3 dimension.

If $\beta_3 \geq 1/2$, then the optimum for the LP without the fourth constraint, $\lambda_1 = \lambda_2 = \lambda_3 = 1/2$, satisfies the fourth constraint and is therefore optimal, leading to the same $\sqrt{M} \times \sqrt{M} \times \sqrt{M}$ as in the "large loop bound" cases discussed above.

If $\beta_3 \leq 1/2$, then we can set $\lambda_3 = \beta_3$ to make the fourth inequality tight, and then set $\lambda_1 = 1 - \beta_3$ and $\lambda_2 = \beta_3$ to tighten 3.3.4 in addition to the first inequality in the LP; as three irredundant inequalities are tight and we only have three variables, this solution must be optimal as well. This obtains a tile size of $M/L_3 \times L_3 \times L_3 = ML_3$ (with a communication cost of $L_1 L_2$, a quantity that is equal to the size of A_2 and therefore must be optimal) as expected.

Alternatively, we could achieve the same tile size with a tile of size $\sqrt{M} \times \sqrt{M} \times L_3$ (corresponding to $\lambda = \lambda_2 = 1/2$, $\lambda_3 = \beta_3$). In fact, the LP is optimized by any point between the two solutions we found previously; specifically, for any $\alpha \leq 1$,

$$\begin{aligned} \lambda_1 &= \alpha/2 + (1-\alpha)(1-\beta_3) \\ \lambda_2 &= \alpha/2 + (1-\alpha)\beta_3 \\ \lambda_3 &= \beta_3 \end{aligned}$$

optimizes LP (3.3.3); this corresponds to a tile size of:

$$\frac{M^{1-\alpha/2}}{L_3^{1-\alpha}} \times M^{\alpha/2} L_3^{1-\alpha} \times L_3 \ .$$

When attempting to optimize this matrix multiplication on a real-world system, we may select any tiling from the above α-parameterized family of optimal tilings in order to find one that runs well in practice (e.g. inner loops being multiples of cache line lengths or vector units).

As the communication cost's derivation is symmetrical (i.e. it continues to be valid when we swap the subscripts) and the tile for the small-L_3 case above remains be a legal tiling if L_3 is the smallest loop index, we obtain the following *tight* lower bound for matrix multiplication's communication cost:

$$\max(L_1 L_2 L_3/\sqrt{M}, L_1 L_2, L_2 L_3, L_1 L_3)$$

3.3.2 Tensor Contraction

Let $1 \leq j < k-1 < d$. Let us consider a tensor contraction of the form

$$\begin{aligned} &\text{for}\, \{x_1, ..., x_d\} \in [L_1] \times ... \times [L_d] \\ &\qquad A_1(x_1, ..., x_j, x_k, ..., x_d) += A_2(i_1, ..., i_{k-1}) \times A_3(x_{j+1}, x_d) \end{aligned}$$

This nested-loop model encapsulates several machine learning applications. For instance, *pointwise convolutions* - convolutions with 1×1 filters, often used along depth-separable convolutions [37] to mimic the effect of standard machine learning convolutions with less memory usage, may be represented as tensor contractions:

$$\begin{aligned} &\text{for}\, \{b, c, k, w, h\} = 0 : \{B, C, K, W, H\} - 1 \\ &\qquad Out(k, h, w, b) += Image(w, h, c, b) \times Filter(k, c) \end{aligned} \tag{3.3.5}$$

The same holds for fully connected convolutional layers.

Optimizing over the BL polytope provides a communication lower bound for the large-loop bound case of $L_1 ... L_d/\sqrt{M}$ [14].

We instantiate the LP 3.2.1 to get:

$$\max \lambda_1 + ... + \lambda_d$$

subject to

$$\begin{aligned}
\lambda_1 + ... + \lambda_j + \lambda_k + ... + \lambda_d &\leq 1 \\
\lambda_1 + ... + \lambda_{k-1} &\leq 1 \\
\lambda_{j+1} + ... + \lambda_d &\leq 1 \\
\lambda_1 &\leq \beta_1 = \log_M L_1 \\
&\vdots \\
\lambda_d &\leq \beta_d = \log_M L_d
\end{aligned}$$

The structure of this linear program is much like that of matrix multiplication, and it can be transformed into one identical to that for matrix multiplication. Let $\gamma_1 = \sum_{i\in[j]} \lambda_i$, $\gamma_2 = \sum_{i\in[j+1,k-1]} \lambda_i$, and $\gamma_3 = \sum_{i\in[k,d]} \lambda_i$. Then we can rewrite the linear program as maximizing $\gamma_1 + \gamma_2 + \gamma_3$ subject to:

$$\begin{aligned}
\gamma_1 + \gamma_3 &\leq 1 \\
\gamma_1 + \gamma_2 &\leq 1 \\
\gamma_2 + \gamma_3 &\leq 1 \\
\gamma_1 &\leq \sum_{i\in[j]} \beta_i \\
\gamma_2 &\leq \sum_{i\in[j+1,k-1]} \beta_i \\
\gamma_3 &\leq \sum_{i\in[k,d]} \beta_i
\end{aligned}$$

As this linear program is identical to that for matrix multiplication, it immediately follows that its optimum is either $3/2$ or $1 + \min\left\{\sum_{i\in[j]} \beta_i, \sum_{i\in[j+1,k-1]} \beta_i, \sum_{i\in[k,d]} \beta_i\right\}$, whichever is smaller for the given program.

3.3.3 n-body Pairwise Interactions

Suppose we have a list of n objects, and each object interacts with every other object. This comes up frequently in many scientific computing applications (e.g. particle simulations), as well as database joins.

The nested loops for this problem are (for some arbitrary function f):

```
for $\{x_1, x_2\} \in [L_1] \times [L_2]$
    $A_1[x_1] = f(A_2[x_1], A_3[x_3])$
```

Instantiating 3.2.1, we get:

$$\begin{aligned} \max \lambda_1 + \lambda_2 \ \text{ s.t.} \\ \lambda_1 &\leq 1 \\ \lambda_2 &\leq 1 \\ \lambda_1 &\leq \beta_1 = \log_M L_1 \\ \lambda_2 &\leq \beta_2 = \log_M L_2 \end{aligned}$$

which gives us a maximum tile size of $\min\{M^2, L_1M, L_2M, L_1L_2\}$ and a maximum communication cost of $\min\{L_1L_2/M, L_2, L_1, M\}$. The last term, M, is a result of the assumption in our model that each tile carries M words of memory into cache. Therefore, it is important to note that *if total amount of memory required to execute the program without going back to main memory is less than M, the output of the program will still be M, when in the actual cost is in fact the sum of the sizes of the matrices.*

Chapter 4

Tilings and Communication Lower Bounds for Convolutions

In many cases, such as the projective case explored in Chapter 3, rectangular tilings are provably optimal and can be easily calculated. Unfortunately, not all tensor algorithms of interest have projective data access functions, and for some of these algorithms, a non-rectangular tiling is required to obtain a lower bound. This chapter explores one such common case, that of *convolutions*. A common 2D convolution is given in Algorithm 6. Dealing with the non-projective structure of data accesses to *Input* requires techniques we will develop in this chapter. Our main results are as follows:

Algorithm 6: 7-nested loop 2D convolution

```
input : stride sizes $\sigma_w, \sigma_h$
        $L_b$ batches of input images of size $\sigma_w L_w + L_r \times \sigma_h L_h + L_s$, each with $C$ channels
        $L_r \times L_s$ filters, mapping each of $L_c$ input channels to $L_k$ output channels
        datatype sizes $p_O$, $p_I$, and $p_F$, specifying number of words required to store elements of the output, input, and filter respectively
output: $L_b$ batches of output images of size $L_w \times L_h$, each with $L_k$ channels
for $x_b \in [0, L_b), x_c \in [0, L_c), x_k \in [0, L_k), x_w \in [0, L_w), x_h \in [0, L_h), x_r \in [0, L_r), x_s \in [0, L_s)$ :
    $\text{Out}(x_k, x_h, x_w, x_b) += \text{Input}(x_r + \sigma_w x_w, x_s + \sigma_h x_h, x_c, x_b) \times \text{Filter}(x_k, x_r, x_s, x_c)$
```

- A communication lower bound for convolutions. While we focus on 2D convolutions, which are the most common in machine learning applications, our techniques are straightforwardly generalize to higher dimensions.

- An optimal, *non-rectangular* tiling for attaining these lower bounds.

- Performance measurements of applying our tiling on the GEMMINI [26] accelerator.

The work in this chapter has been previously published in [18, 7].

4.1 The Lower Bound

Our main theorem in this section will be the following lower bound on the communication complexity of Algorithm 6 on a two-level memory hierarchy model. Notice that we allow the three tensors to be datatypes of different sizes: the Input tensor has entries which are p_I words in length, the Filter tensor's elements are of size p_F, and the Output tensor's elements are of size p_O.

Theorem 15. *If q_{conv} is the number of words communicated by a seven-nested loop 2D convolution within a single-processor memory model with M words of fast memory, where* Input*,* Filter*, and* Output *are non-overlapping tensors and $G := L_bL_cL_kL_wL_hL_rL_s$ is the total number of multiply-add operations performed during the computation, we have*

$$q_{conv} \geq \max \left\{ \begin{matrix} p_I|Input| + p_F|Filter| + p_O|Output|, \\ \dfrac{C_pG}{M} - M, \\ \dfrac{2(p_Ip_Fp_O)^{1/2}(\sigma_w\sigma_h)^{1/2}G}{(L_rL_sM)^{1/2}} - 2M \end{matrix} \right\} \tag{4.1.1}$$

where the value of $C_p = C_p(p_I, p_F, p_O)$ depends on the precisions satisfying a triangle condition:

$$C_p = \begin{cases} \frac{1}{4}(p_I + p_F + p_O)^2 & p_j \leq p_k + p_\ell \quad \textit{for all distinct } j,k,\ell \\ p_j(p_k + p_\ell) & p_j > p_k + p_\ell \quad \textit{for some distinct } j,k,\ell \end{cases}$$

In the "standard" case when all three tensors contain elements of the same datatype (which we can assume without loss of generality to have size 1), $C_p = 9/4$. The first bound corresponds to accessing each entry of the input, output, and filter at least once. The second bound dominates when individual $w_F \times h_F$ filters are large relative to the memory size M, and the third bound dominates when filters are small relative to M. In all practical cases, the precisions satisfy the triangle condition, so the first expression for C_p is more relevant.

For example, in the standard precision case $p_I = p_F = p_O = 1$, the bound becomes

$$q_{conv} \geq \max \left\{ \begin{matrix} |\text{Input}| + |\text{Filter}| + |\text{Output}|, \\ \dfrac{9G}{4M} - M, \\ \dfrac{2G(\sigma_w\sigma_h)^{1/2}}{(L_rL_sM)^{1/2}} - 2M \end{matrix} \right\}$$

The first bound is simply the size of the input, filter, and output tensors, and is independent of the memory size M. The second bound exhibits $\Omega(1/M)$ decay, while the third bound exhibits $\Omega(1/M^{1/2})$ decay. However, it is important to note that the third bound only exceeds the second bound when $w_F h_F < \frac{64M\sigma_w\sigma_h}{81}$, i.e. when the filters are small relative to the memory size.

4.1.1 Derivation of the Trivial Bound

The first part of the lower bound (4.1.1) is trivial.

Lemma 16. *With the setup in Theorem 15, the number of words communicated X satisfies*

$$X \geq p_I|I| + p_F|F| + p_O|O|$$

Proof. Every entry of Input and Filter must be accessed at least once, and every entry in Output must be filled by the computation. All three arrays reside in slow memory, so at minimum p_j words must be communicated for every entry in the jth array, for $j \in \{I, F, O\}$. So the number of words communicated X satisfies

$$X \geq p_I\,|I| + p_F\,|F| + p_O\,|O|$$

□

4.1.2 Derivation of the "Large Loop Bound" Lower Bound

In order to prove the second lower bound in (4.1.1), we will first enumerate the constraints of the associated Brascamp-Lieb polytope. By Theorem 9, it suffices generate one rank constraint for each subgroup in the lattice of subgroups generated by the kernels of ϕ_O, ϕ_I and ϕ_F, which are:

$$\begin{aligned}
\phi_I(x_b, x_c, x_k, x_w, x_h, x_r, x_s) &= (x_b, x_c, x_r + \sigma_w x_w, x_s + \sigma_h x_h) && \text{(input)} \\
\phi_F(x_b, x_c, x_k, x_w, x_h, x_r, x_s) &= (x_c, x_k, x_r, x_s) && \text{(filter)} \\
\phi_O(x_b, x_c, x_k, x_w, x_h, x_r, x_s) &= (x_b, x_k, x_w, x_h) && \text{(output)}
\end{aligned} \tag{4.1.2}$$

Their kernels are (letting x_i represent free variables):

$$\begin{aligned}
\ker(\phi_I) &= (0, 0, x_k, x_w, x_h, -\sigma_w x_w, -\sigma_h x_h) \\
\ker(\phi_F) &= (x_b, 0, 0, x_w, x_h, 0, 0) \\
\ker(\phi_O) &= (0, x_c, 0, 0, 0, x_r, x_s)
\end{aligned} \tag{4.1.3}$$

In order to enumerate the elements generated by the lattice of subgroups of these kernels, we must first develop some machinery. For convenience, we will always include $\{0\}$ in our lattices, since this does not change other members of the lattice, and simplifies some expressions below.

Suppose $A = \{A_1, ..., A_n\}$ and $B = \{B_1, ..., B_m\}$ are finite sets of subgroups of an abelian group. We will call them *independent* if

$$\sum_i A_i \cap \sum_j B_j = \{0\}$$

Let $\mathcal{L}_A$ denote the lattice generated by the subgroups in A, and similarly for $\mathcal{L}_B$; we will add $\{0\}$ to these lattices if they do not already contain it. Then from the definition of a lattice it is easy to see that $\mathcal{L}_A$ and $\mathcal{L}_B$ are independent if A and B are independent.

Lemma 17. *Suppose A and B are independent. Then*

$$\mathcal{L}_{A\cup B} = \mathcal{L}_A + \mathcal{L}_B \equiv \{C + D : C \in \mathcal{L}_A, D \in \mathcal{L}_B\}$$

Proof. Since both lattices include $\{0\}$,

$$A \cup B \subset \mathcal{L}_A + \mathcal{L}_B.$$

It suffices to show that if $A_1 + B_1$ and $A_2 + B_2$ are both in $\mathcal{L}_A + \mathcal{L}_B \subseteq \mathcal{L}_{A\cup B}$, then so are their sum and intersection. The sum

$$(A_1 + B_1) + (A_2 + B_2) = (A_1 + A_2) + (B_1 + B_2) \in \mathcal{L}_A + \mathcal{L}_B$$

follows from being abelian, and a lattice being closed under addition.

The intersection

$$(A_1 + B_1) \cap (A_2 + B_2) = (A_1 \cap A_2) + (B_1 \cap B_2) \in \mathcal{L}_A + \mathcal{L}_B$$

follows from independence, and a lattice being closed under intersection. □

As a result, then it is easy to describe $\mathcal{L}_{A\cup B}$ if we have simple, finite descriptions of $\mathcal{L}_A$ and $\mathcal{L}_B$, for independent A and B.

Furthermore, note that independence also implies that

$$\operatorname{rank}(A_i + B_j) = \operatorname{rank}(A_i) + \operatorname{rank}(B_j) \qquad \forall i, j \tag{4.1.4}$$

and

$$\operatorname{rank}(\phi_j(A_i + B_k)) = \operatorname{rank}(\phi_j(A_i)) + \operatorname{rank}(\phi_j(B_k)) \ . \tag{4.1.5}$$

As a result, our strategy will be to decompose the kernels (4.1.3) into independent subgroups.

We can identify the following independent families of indices: $\{x_b\}$, $\{x_c\}$, $\{x_k\}$, $\{x_w, x_r\}$, and $\{x_h, x_s\}$. We call these independent because they give rise to the following pairwise independent collections of subgroups which generate the kernels - and hence the lattice - that we want:

$$\begin{aligned}
C_1 &= \{(x_b, 0, 0, 0, 0, 0, 0)\} = \{C_{1,1}\} \\
C_2 &= \{(0, x_c, 0, 0, 0, 0, 0)\} = \{C_{2,1}\} \\
C_3 &= \{(0, 0, x_k, 0, 0, 0, 0)\} = \{C_{3,1}\} \\
C_4 &= \{(0, 0, 0, x_w, 0, 0, 0), (0, 0, 0, 0, 0, x_r, 0), (0, 0, 0, x_w, 0, -\sigma_w x_w, 0)\} \\
&= \{C_{4,1}, C_{4,2}, C_{4,3}\} \\
C_5 &= \{(0, 0, 0, 0, x_h, 0, 0), (0, 0, 0, 0, 0, 0, x_s), (0, 0, 0, 0, x_h, 0, -\sigma_h x_h)\} \\
&= \{C_{5,1}, C_{5,2}, C_{5,3}\}
\end{aligned}$$

These subgroups give the following breakdown of the kernels:

$$\begin{aligned}
\ker(\phi_I) &= C_{3,1} + C_{4,3} + C_{5,3} \\
\ker(\phi_F) &= C_{1,1} + C_{4,1} + C_{5,1} \\
\ker(\phi_O) &= C_{2,1} + C_{4,2} + C_{5,2}
\end{aligned}$$

By (4.1.4) and (4.1.5) it suffices to check the rank constraints only on subgroups in the five lattices, $\mathcal{L}_{(C_j)}$. Fortunately, $\mathcal{L}_{(C_j)} = C_j$ for $j = 1, 2, 3$. For C_4 and C_5 we have:

$$\begin{aligned}
\mathcal{L}_{(C_4)} &= C_4 \cup \{(0, 0, 0, i_4, 0, i_6, 0)\} = C_4 \cup \{C_{4,4}\} \\
\mathcal{L}_{(C_5)} &= C_5 \cup \{(0, 0, 0, 0, i_5, 0, i_7)\} = C_5 \cup \{C_{5,4}\}
\end{aligned}$$

Now suppose $s_I, s_F, s_O \in [0, 1]$. Applying Theorem 9, the Brascamp-Lieb polytope is defined by the following inequality for each H in some $\mathcal{L}_{(C_j)}$:

$$\operatorname{rank}(H) \leq s_I \operatorname{rank}(\phi_I(H)) + s_F \operatorname{rank}(\phi_F(H)) + s_O \operatorname{rank}(\phi_O(H)).$$

We enumerate these inequalities in the table below:

H	rank(H)	rank$(\phi_I(H))$	rank$(\phi_F(H))$	rank$(\phi_O(H))$	Constraint
$C_{1,1}$	1	1	0	1	$1 \leq s_I + s_O$
$C_{2,1}$	1	1	1	0	$1 \leq s_I + s_F$
$C_{3,1}$	1	0	1	1	$1 \leq s_F + s_O$
$C_{4,1}$	1	1	0	1	$1 \leq s_I + s_O$
$C_{4,2}$	1	1	1	0	$1 \leq s_I + s_F$
$C_{4,3}$	1	0	1	1	$1 \leq s_F + s_O$
$C_{4,4}$	2	1	1	1	$2 \leq s_I + s_F + s_O$
$C_{5,1}$	1	1	0	1	$1 \leq s_I + s_O$
$C_{5,2}$	1	1	1	0	$1 \leq s_I + s_F$
$C_{5,3}$	1	0	1	1	$1 \leq s_F + s_O$
$C_{5,4}$	2	1	1	1	$2 \leq s_I + s_F + s_O$

Removing repeated inequalities the Brascamp-Lieb polytope generated by ϕ_I, ϕ_F, ϕ_O is given by:

$$\begin{aligned} 1 &\leq s_I + s_F \\ 1 &\leq s_I + s_O \\ 1 &\leq s_F + s_O \\ 2 &\leq s_I + s_F + s_O \end{aligned} \tag{4.1.6}$$

Maximizing $s_I + s_F + s_O$ over these constraints gives us 2, which, by Theorem 6, provides a communication lower bound of $\Omega(G/M)$. However, we can achieve a more precise lower bound that takes constant factors into account with a more careful analysis, which follows.

First, we handle the case when the triangle condition is met.

Lemma 18. *With the setup in Theorem 15, as long as $p_I \leq p_F + p_O$, $p_F \leq p_I + p_O$, and $p_O \leq p_I + p_F$, the number of words communicated X satisfies*

$$X \geq \frac{(p_I + p_F + p_O)^2 G}{4M} - M$$

Proof. We split the execution of the convolution into L segments of contiguous updates. In each segment, we allow exactly T words to be loaded/stored, except for possibly the last segment which may have $\leq T$ words; T is a parameter we will optimize.

We fix our attention to a single segment. Let V be the set of indices of updates computed during the current segment. V contains tuples $(x_j) \in \mathbb{Z}^7$. Then $\phi_I(V)$ is the set of indices of Input whose data must be accessed during the segment, $\phi_F(V)$ the set of indices of Filter, and $\phi_O(V)$ the set of indices of Output. We have at most M words of data in fast memory before the segment begins, and may load at most T more during the segment. The number of words we access from the jth array during the segment is $p_j|\phi_j(V)|$. Then the number of words we access during this segment is

$$p_I|\phi_I(V)| + p_F|\phi_F(V)| + p_O|\phi_O(V)| \leq M + T.$$

Let s_I, s_F, and s_O satisfy the Brascamp-Lieb rank constraints 4.1.6; we know the strongest asymptotic lower bound is obtained at $s_I + s_F + s_O = 2$. Theorem 4 states that for every set $V \subset \mathbb{Z}^7$,

$$|V| \leq |\phi_I(V)|^{s_I} |\phi_F(V)|^{s_F} |\phi_O(V)|^{s_O}.$$

Let $v_j := 2p_j|\phi_j(V)|/(M + T)$. The inequality becomes

$$|V| \leq \frac{(M+T)^2}{4} (v_I/p_I)^{s_I} (v_F/p_F)^{s_F} (v_O/p_O)^{s_O}.$$

Denote $f(v_I, v_F, v_O) = v_I^{s_I} v_F^{s_F} v_O^{s_O}$. Then the maximum number of updates $|V|$ possible during this segment is bounded by

$$\frac{(M+T)^2}{4} \max f(v_I, v_F, v_O)$$

subject to the constraint $v_1 + v_2 + v_3 \leq 2$.

We assume $v_I + v_F + v_O = 2$ and apply the technique of Lagrange multipliers:

$$\begin{aligned} v_I + v_F + v_O &= 2 \\ s_I v_I^{s_I - 1} v_F^{s_F} v_O^{s_O} &= s_F v_F^{s_F - 1} v_I^{s_I} v_O^{s_O} = s_O v_O^{s_O - 1} v_I^{s_I} v_F^{s_F} \\ &= \lambda \end{aligned}$$

Taking an inner product with (v_I, v_F, v_O) we find:

$$(s_I + s_F + s_O) v_I^{s_I} v_F^{s_F} v_O^{s_O} = \lambda (v_I + v_F + v_O) \quad \Longrightarrow \quad v_I^{s_I} v_F^{s_F} v_O^{s_O} = \lambda$$

since $s_I + s_F + s_O = 2 = v_I + v_F + v_O$. Substituting into each equation and dividing, we find $s_I = v_I$, $s_F = v_F$, $s_O = v_O$. Then we have shown that the maximum number of updates during the current segment is

$$|V| \leq \frac{1}{4}(M+T)^2 (s_I/p_I)^{s_I} (s_F/p_F)^{s_F} (s_O/p_O)^{s_O}$$

This holds for all triples (s_j) with $s_I + s_F + s_O = 2$ and $s_I, s_F, s_O \leq 1$. In particular, it holds for the triple (s_j) which minimize the right hand side. We apply Lagrange multipliers again ignoring the last three constraints on the s_j:

$$\begin{aligned} s_I + s_F + s_O &= 2 \\ (1 + \log(s_I/p_I))(s_I/p_I)^{s_I} (s_F/p_F)^{s_F} (s_O/p_O)^{s_O} &= \lambda \\ (1 + \log(s_F/p_F))(s_I/p_I)^{s_I} (s_F/p_F)^{s_F} (s_O/p_O)^{s_O} &= \lambda \\ (1 + \log(s_O/p_O))(s_I/p_I)^{s_I} (s_F/p_F)^{s_F} (s_O/p_O)^{s_O} &= \lambda \end{aligned}$$

Equating and dividing by $(s_I/p_I)^{s_I} (s_F/p_F)^{s_F} (s_O/p_O)^{s_O}$ we find $s_I/p_I = s_F/p_F = s_O/p_O$. This leads to $s_j = 2p_j/(p_I + p_F + p_O)$. Note that these minimizers always satisfy $s_j \leq 1$ for all j. Indeed, the triangle condition guarantees $2p_j \leq p_I + p_F + p_O$ for all j. Then we have shown that the maximum number of computations during any segment is

$$\begin{aligned} |V| &\leq \frac{1}{4}(M+T)^2 (s_I/p_I)^{s_I} (s_F/p_F)^{s_F} (s_O/p_O)^{s_O} \\ &= \frac{1}{4}(M+T)^2 \left(\frac{2}{p_I + p_F + p_O} \right)^2 \\ &= \frac{(M+T)^2}{(p_I + p_F + p_O)^2} \end{aligned}$$

Since we must do G updates in total, the total number of segments is bounded below:

$$L \geq \left\lfloor \frac{G}{|V|} \right\rfloor \geq \frac{(p_I + p_F + p_O)^2 G}{(M+T)^2} - 1$$

Each segment besides the last has T loads/stores, so the total number of words moved is:

$$X \geq T\left(\frac{(p_I + p_F + p_O)^2 G}{(M+T)^2} - 1\right) = \frac{(p_I + p_F + p_O)^2 TG}{(M+T)^2} - T$$

To choose optimal segment length, we note that $T/(M+T)^2$ is maximized when $T = M$ and we find the following lower bound on the communication cost:

$$X \geq \frac{(p_I + p_F + p_O)^2 G}{4M} - M$$

□

Should the triangle condition fail, we slightly modify the last proof by finding a valid set of minimizers. Note that only one of the three constraints may fail at once: if $p_j > p_k + p_\ell$, then $p_k + p_j > p_\ell$.

Lemma 19. *With the setup in Theorem 15, if $p_j > p_k + p_\ell$ for some distinct $j, k, \ell \in \{I, F, O\}$, the number of words communicated X satisfies*

$$X \geq \frac{p_j(p_k + p_\ell)G}{M} - M$$

Proof. The proof is the same as the proof of Lemma 18, except now we prescribe $s_j = 1$ and $s_k + s_\ell = 1$. This guarantees that all conditions for HBL are met. We maximize $(s_k/p_k)^{s_k}(s_\ell/p_\ell)^{s_\ell}$ with respect to s_k and s_ℓ as before, and find $s_k/p_k = s_\ell/p_\ell$. This leads to $s_k = p_k/(p_k + p_\ell)$ and $s_\ell = p_\ell/(p_k + p_\ell)$. All constraints are satisfied. Then we have shown that the maximum number of computations during any segment is

$$\begin{aligned}
|V| &\leq \frac{1}{4}(M+T)^2 (s_j/p_j)^{s_j} (s_k/p_k)^{s_k} (s_\ell/p_\ell)^{s_\ell} \\
&= \frac{1}{4}(M+T)^2 \frac{1}{p_j}\left(\frac{1}{p_k + p_\ell}\right)^{s_j + s_k} \\
&= \frac{(M+T)^2}{4p_j(p_k + p_\ell)}
\end{aligned}$$

Since we must do G updates in total, the total number of segments is bounded below:

$$L \geq \left\lfloor \frac{G}{|V|} \right\rfloor \geq \frac{4p_j(p_k + p_\ell)G}{(M+T)^2} - 1$$

Each segment besides the last has T loads/stores, and we choose $T = M$, so the total number of words moved is:

$$X \geq \frac{p_j(p_k + p_\ell)G}{M} - M$$

□

4.1.3 Derivation of the "Small Filter" Lower Bound

When $M > (C_p G)^{1/2}$, the previous bounds become trivial. When the filter size $L_r L_s$ is small relative to M, we are able to reduce the decay in our bounds from $1/M$ to $1/M^{1/2}$ and extend them to larger memory sizes. To show this third "small filter" bound, we rewrite the problem and exploit new array access homomorphisms. In particular, we rewrite the loops over x_r and x_s as loops over x'_r, x''_r, x'_s, x''_s. We have $x_r = \sigma_w x'_r + x''_r$ for $x''_r \in [0, \sigma_w - 1]$ and $x'_r \in [0, L_r/\sigma_w - 1]$, and we similarly divide x_s by σ_h for x'_s and x''_s. This has the effect of lifting *Input* and *Filter* to higher dimensional arrays, with 6 indices instead of 4. Under the lift, our new data access functions are:

$$\begin{aligned}
&\text{Input}(x_b, x_c, x_w + x'_r, x''_r, x_h + x'_s, x''_s)\\
&\text{Filter}(x_c, x_k, x'_r, x''_r, x'_s, x''_s)\\
&\text{Output}(x_b, x_k, x_w, x_h)
\end{aligned}$$

In our proof, we will find it valuable to fix the indices x'_r and x'_s, so we opt to introduce a new collection of array access homomorphisms which ignore these. With an implicit translation by $\vec{q} = (x'_r, x'_s)$, we define the homomorphisms $\phi'_I, \phi'_O : \mathbb{Z}^7 \to \mathbb{Z}^4$, $\phi'_F : \mathbb{Z}^7 \to \mathbb{Z}^6$:

$$\begin{aligned}
\phi'_I(x_b, x_c, x_k, x_w, x_h, x''_r, x''_s) &= (x_b, x_c, x_w, x''_r, x_h, x''_s)\\
\phi'_F(x_b, x_c, x_k, x_w, x_h, x''_r, x''_s) &= (x_c, x_k, x''_r, x''_s)\\
\phi'_O(x_b, x_c, x_k, x_w, x_h, x''_r, x''_s) &= (x_b, x_k, x_w, x_h)
\end{aligned}$$

These data access functions are projective, and in fact are an instance of the tensor contraction case discussed in Section 3.3.2. Following the discussion there, we set $s_I = s_F = S_O = 1/2$, which provides the following an BL inequality for finite subsets V of $\mathbb{Z}^7$:

$$|V| \leq |\phi'_I(V)|^{1/2} |\phi'_F(V)|^{1/2} |\phi'_O(V)|^{1/2}.$$

We can now begin the proof of the third bound.

Lemma 20. *The number of words communicated X satisfies*

$$X \geq \frac{2(p_I p_F p_O)^{1/2} (\sigma_w \sigma_h)^{1/2} G}{(L_r L_s M)^{1/2}} - 2M$$

Proof. We split the our into L segments with T loads/stores as before. Let V be the set of updates computed during a given segment. For fixed indices $\vec{q} = (x'_r, x'_s)$, let $V(\vec{q})$ be the slice of V with those two coordinates held constant: $V(\vec{q}) = \pi^{-1}(\vec{q})$ where π is the projection of $\mathbb{Z}^9$ onto the $\vec{q}$ coordinates. Then to compute every update in $V(\vec{q})$ we must access the entries of Input corresponding to indices $\phi'_I(V(\vec{q}))$, and similarly for Filter and Output. This is an abuse of notation: technically speaking, $V(\vec{q})$ is not a subset of the domain of ϕ'_I, but it is in bijection with one if we ignore the constant $\vec{q}$ coordinates. We apply our HBL inequality to the set $V(\vec{q})$,

$$|V(\vec{q})| \leq |\phi'_I(V(\vec{q}))|^{1/2} |\phi'_F(V(\vec{q}))|^{1/2} |\phi'_O(V(\vec{q}))|^{1/2}$$

Note that V is the disjoint union of the $V(\vec{q})$'s. Also, the set of indices of *Filter* accessed is the disjoint union of the $\phi'_F(V(\vec{q}))$'s. Let u be the number of indices of *Input* accessed during the segment, v the number of indices of *Output* accessed during the segment, and $w(\vec{q}) = |\phi'_F(V(\vec{q}))|$ the number of indices of Filter accessed by each slice. We have:

$$|\phi'_I(V(\vec{q}))| \leq u, \quad |\phi'_O(V(\vec{q}))| \leq v \qquad \forall \vec{q}$$
$$\left|\bigcup_{\vec{q}} V(\vec{q})\}\right| = \sum_{\vec{q}} w(\vec{q})$$

We have at most M words in memory before the segment begins, and may load at most T more:

$$p_I u + p_O v + p_F \sum_{\vec{q}} w(\vec{q}) \leq M + T$$

Using our HBL inequality,

$$|V| = \sum_{\vec{q}} |V(\vec{q})| \leq u^{1/2} v^{1/2} \sum_{\vec{q}} w(\vec{q})^{1/2} \ .$$

Denote

$$f(u, v, w(\vec{q})) = u^{1/2} v^{1/2} \sum_{\vec{q}} w(\vec{q})^{1/2} \ .$$

The maximum number of updates during this segment is bounded by $\max f(u, v, w(\vec{q}))$, subject to

$$p_I u + p_O v + p_F \sum_{\vec{q}} w(\vec{q}) \leq M + T \ .$$

We assume equality and apply Lagrange multipliers:

$$p_I u + p_O v + p_F \sum_{\vec{q}} w(\vec{q}) = M + T \tag{4.1.7}$$
$$\frac{1}{2} \frac{v^{1/2}}{u^{1/2}} \sum_{\vec{q}} w(\vec{q})^{1/2} = p_I \lambda \tag{4.1.8}$$
$$\frac{1}{2} \frac{u^{1/2}}{v^{1/2}} \sum_{\vec{q}} w(\vec{q})^{1/2} = p_O \lambda \tag{4.1.9}$$
$$\frac{1}{2} \frac{u^{1/2} v^{1/2}}{w(\vec{q})^{1/2}} = p_F \lambda \qquad \forall \vec{q} \tag{4.1.10}$$

Then by dividing (4.1.8) and (4.1.9), $p_I u = p_O v$, and by equating instances of (4.1.10), all the $w(\vec{q}) =: w$ are equal. Using the fact that there are $\frac{L_r L_s}{\sigma_w \sigma_h}$ pairs of $(\vec{q})$ and equating (4.1.8) and (4.1.10),

$$\begin{aligned}\frac{1}{p_I}\sum_{\vec{q}} w(\vec{q})^{1/2} &= \frac{1}{p_I}\frac{L_r L_s}{\sigma_w \sigma_h} w^{1/2} \\ &= \frac{u}{p_F w^{1/2}}\end{aligned}$$

so that $p_F w = \frac{\sigma_w \sigma_h}{L_r L_s} p_I u$. Therefore, the maximizing values are

$$\begin{aligned}u &= \frac{M+T}{3p_I} \\ v &= \frac{M+T}{3p_O} \\ w(\vec{q}) &= \frac{\sigma_w \sigma_h}{L_r L_s}\frac{M+T}{3p_F}\end{aligned}$$

for all $\vec{q}$. Using these values, the maximum number of updates during the current segment is

$$\begin{aligned}|V| &\leq f(u, v, w(\vec{q})) \\ &\leq \frac{M+T}{3p_I^{1/2} p_O^{1/2}}\frac{L_r L_s}{\sigma_w \sigma_h}\left(\frac{\sigma_w \sigma_h}{L_r L_s}\frac{M+T}{3p_F}\right)^{1/2} \\ &= \frac{(M+T)^{3/2}}{3^{3/2}(p_I p_F p_O)^{1/2}}\frac{(L_r L_s)^{1/2}}{(\sigma_w \sigma_h)^{1/2}}\end{aligned}$$

and the number of segments L is bounded below by

$$L \geq \left\lfloor \frac{G}{|V|} \right\rfloor \geq \frac{3^{3/2}(p_I p_F p_O)^{1/2}(\sigma_w \sigma_h)^{1/2} G}{(L_r L_s)^{1/2}(M+T)^{3/2}} - 1$$

Each segment besides the last has at most T loads/stores, so the communication cost is

$$X \geq \frac{3^{3/2}(p_I p_F p_O)^{1/2}(\sigma_w \sigma_h)^{1/2} T G}{(L_r L_s)^{1/2}(M+T)^{3/2}} - T$$

To choose optimal segment length, we note that $T/(M+T)^{3/2}$ is maximized when $T = 2M$ and we find the communication cost

$$X \geq \frac{2(p_I p_F p_O)^{1/2}(\sigma_w \sigma_h)^{1/2} G}{(L_r L_s M)^{1/2}} - 2M$$

□

Taken together, Lemmas 16, 18, 19, and 20 complete the proof of Theorem 15.

4.2 A Tiling that Obtains the Lower Bound

This chapter describes a tiling that attains (up to a constant factor) the lower bound found in Section 4.1. For simplicity, we focus on asymptotics and ignore boundary conditions and other constant factors.

As in Section 4.1.3, we will first lift the tensors into higher dimensions, rewriting x_r into $\sigma_w x'_r + x''_r$ for $x''_r \in [0, \sigma_w - 1]$, $x'_r \in [0, L_r/\sigma_w - 1]$ and x_s similarly in order to allows us to align them with the $\sigma_w x_w$ and $\sigma_h x_h$ terms. We will then apply a rectangular tiling to the lifted array structure, generating Algorithm 7, where we use for $x = \alpha : \beta : \gamma$ to denote iterating from α to γ with a step size of β. To minimize the communication cost, it suffices to maximize the size of each block (that is, $b_b b_c b_k b_w b_h b_{r'} b_{r''} b_{s'} b_{s''}$) subject to the following constraints:

Algorithm 7: Tiled 7-nested loop 2D convolution

```
input  : stride sizes $\sigma_w, \sigma_h$
         $L_b$ batches of input images of size $\sigma_w L_w + L_r \times \sigma_h L_h + L_s$, each with $C$
         channels
         $L_r \times L_s$ filters, mapping each of $L_c$ input channels to $L_k$ output channels
         datatype sizes $p_O$, $p_I$, and $p_F$, specifying number of words required to
         store elements of the output, input, and filter respectively
output:  $B$ batches of output images of size $L_w \times L_h$, each with $L_k$ channels
data   : tile sizes $b_b, b_c, b_k, b_w, b_h, b_{r'}, b_{r''}, b_{s'}, b_{s''}$

for $y_{\{b,c,k,w,h\}} = 0 : b_{\{b,c,k,w,h\}} : L_{\{b,c,k,w,h\}} - b_{\{b,c,k,w,h\}}$ :
    for $y_{r'} = 0 : b_{r'} : L_r/\sigma_w - b_{r'}$ :
        for $y_{r''} = 0 : b_{r''} : \sigma_w - b_{r''}$ :
            for $y_{s'} = 0 : b_{s'} : L_s/\sigma_h - b_{s'}$ :
                for $y_{s''} = 0 : b_{s''} : \sigma_h - b_{s''}$ :
                    // inner loop
                    for $z_{\{b,c,k,w,h,r',r'',s',s''\}} \in [0, b_{\{b,c,k,w,h,r',r'',s',s''\}})$ :
                        $x_{\{b,c,k,w,h,r',r'',s',s''\}} = y_{\{b,c,k,w,h,r',r'',s',s''\}} + z_{\{b,c,k,w,h,r',r'',s',s''\}}$
                        $\mathrm{Out}(x_k, x_h, x_w, x_b) += \mathrm{Input}(x_{r''} + \sigma_w(x_{r'} + x_w), x_{s''} + \sigma_h(x_{s'} +$
                          $x_h), x_c, x_b) \times \mathrm{Filter}(x_k, \sigma_w x_{r'} + x_{r''}, \sigma_h x_{s'} + x_{s'}, x_c)$
```

1. Each block size must be positive:

$$b_{\{b,c,k,w,h,r',r'',s',s''\}} \geq 1$$

2. The block size in each dimension is smaller than the loop bound for that dimension. For the first five indices, we have:

$$b_{\{b,c,k,w,h\}} \leq L_{\{b,c,k,w,h\}}$$

The loop bounds on r'' and s'' (given by their definitions) give the following two constraints:

$$\begin{aligned} b_{r''} &\leq \sigma_w \\ b_{s''} &\leq \sigma_h \end{aligned}$$

To ensure that the blocks for r' and s' are of appropriate size, recall that $r = \sigma_w r' + r''$ and $s = \sigma_h s' + s''$, which gives

$$\begin{aligned} \sigma_w b_{r'} + b_{r''} &\leq R \\ \sigma_h b_{s'} + b_{s''} &\leq S \end{aligned}$$

Since $b_{r''} \leq \sigma_w$ and $b_{s''} \leq \sigma_h$, we can safely omit those from the inequality (since their effect on left-hand side is at most equivalent to adding 1 to $b_{r'}$ and $b_{s'}$, and we are only interested in asymptotics) to get

$$\begin{aligned} \sigma_w b_{r'} &\leq R \\ \sigma_h b_{s'} &\leq S \end{aligned}$$

3. The size of each block does not exceed the size of fast memory M. This is straightforward for Out:

$$p_O b_b b_k b_w b_h \leq M$$

as well as $Filter$:

$$p_F b_c b_k b_{r'} b_{r''} b_{s'} b_{s''} \leq M \ .$$

For $Input$, notice that if a block for $x_{r'}$ is $[r'_{start}, r'_{end}]$, a block of $x_{r''}$ is $[r''_{start}, r''_{end}]$ and a block for x_w is $[w_{start}, w_{end}]$, then the indices of $Input$ in the w-dimension accessed will be of the form $i + \sigma_w j$, where $i \in [r''_{start}, r''_{end}]$ and $j \in [w_{start} + r'_{start}, w_{end} + r'_{end}]$. As a result, the number of indices in the w-dimension accessed is $(b_w + b_{r'})b_{r''}$; similarly for the x_h-dimension. Therefore, the total number of elements accessed from $Input$ must be:

$$p_I b_b b_c (b_w + b_{r'})(b_h + b_{s'}) b_{r''} b_{s''} \leq M \ .$$

As we will see shortly, it is convenient to recast our maximization problem as a linear program by taking logs; for this to happen, we only want products in the inequality. Multiplying out the left-hand side of the above gives a sum of four terms; bounding each of them by M is sufficient for an asymptotic analysis. Therefore, we get:

$$\begin{aligned} p_I b_b b_c b_w b_h b_{r''} b_{s''} &\leq M \\ p_I b_b b_c b_w b_{s'} b_{r''} b_{s''} &\leq M \\ p_I b_b b_c b_{r'} b_h b_{r''} b_{s''} &\leq M \\ p_I b_b b_c b_{r'} b_{s'} b_{r''} b_{s''} &\leq M \end{aligned}$$

Taking the log base M of the objective and all the constraints, we get the following linear program, defining $\lambda_{\{b,\ldots,s''\}} = \log_M b_{\{b,\ldots,s''\}}$:

$$\begin{aligned}
\max \lambda_b + \lambda_c + \lambda_k + \lambda_w + \lambda_h + \lambda_{r'} + \lambda_{r''} + \lambda_{s'} + \lambda_{s''} \ s.t.& \\
\lambda_{\{b,c,k,w,h,r',r'',s',s''\}} &\geq 0 \\
\lambda_{\{b,c,k,w,h\}} &\leq \log_M L_{\{b,c,k,w,h\}} \\
\lambda_{r''} &\leq \log_M \sigma_w \\
\lambda_{s''} &\leq \log_M \sigma_h \\
\log_M \sigma_w + \lambda_{r'} &\leq \log_M L_r \\
\log_M \sigma_h + \lambda_{s'} &\leq \log_M L_s \\
\lambda_b + \lambda_k + \lambda_w + \lambda_h &\leq 1 - \log_M p_O \\
\lambda_c + \lambda_k + \lambda_{r'} + \lambda_{r''} + \lambda_{s'} + \lambda_{s''} &\leq 1 - \log_M p_F \\
\lambda_b + \lambda_c + \lambda_w + \lambda_h + \lambda_{r''} + \lambda_{s''} &\leq 1 - \log_M p_I \\
\lambda_b + \lambda_c + \lambda_w + \lambda_{s'} + \lambda_{r''} + \lambda_{s''} &\leq 1 - \log_M p_I \\
\lambda_b + \lambda_c + \lambda_{r'} + \lambda_h + \lambda_{r''} + \lambda_{s''} &\leq 1 - \log_M p_I \\
\lambda_b + \lambda_c + \lambda_{r'} + \lambda_{s'} + \lambda_{r''} + \lambda_{s''} &\leq 1 - \log_M p_I
\end{aligned} \tag{4.2.1}$$

In order to find the optimal tile size for a single problem, we simply solve the above linear program after substituting in the appropriate values for the loop bounds. However, we wish to find a closed-form solution in order to both allow the tiling parameters to be calculated using a basic lookup table of inequalities and arithmetic operations rather than a more complex linear program and to give us the ability to formally prove optimality by showing that *every tiling* that this algorithm generates matches one of the lower bounds 4.1.1.

Algorithms for determining a closed-form solution to the parameterized linear programs in the form of a piecewise linear function have been studied extensively in the context of control theory [24, 6, 83, 41]. We adapt the geometric algorithm from [6] into Algorithm 8.

The intuition for the algorithm is as follows: start with a (possibly open) region in parameter space; during the first iteration of the algorithm, this is the set of all possible valid loop bounds and strides. Pick a random point inside that region, and, setting the parameters to the coordinates of that point, numerically solve the resulting LP. Note which constraints are made tight; if the number is not equal the number of variables, resample the parameters again. The optimizer in the region containing the point is equal to the symbolic solution to the system of linear equations defined by the tight constraints. The region itself is defined as the polytope within which the slack constraints at the point we sampled remain slack at the optimizer. We remove this polytope from our original region and recursively repeat on the remainder until the entire region is partitioned.

Running the algorithm on the above linear program with precisions equal to 1 (as the precisions are very small compared to M in all cases of interest) gives us a partitioning

Algorithm 8: Multiparametric linear program

```
Data: An initial region $R = A\theta \leq b$ to explore
Result: A set $\{(R_i, \hat{\lambda}^i(\theta))\}$, where $R_i$ form a partition of $R$ and $\hat{\lambda}^i(\theta)$ are logs
        (base $M$) of the optimal tile sizes for $\theta \in R_i$
if $R$ is lower dimension or empty :
    return $\emptyset$
Randomly sample element $\theta_0 \in R$
$\lambda_0^* \leftarrow$ optimizer for $\min c^T \lambda$ s.t. $G\lambda \leq w + F\theta_0$ (solve using simplex)
$A(\theta_0) \leftarrow$ indices of zeros of $G\lambda_0^* - F\theta$
$(G_t, w_t, F_t) \leftarrow$ rows $A(\theta_0)$ of $(G, w, F)$
$(G_s, w_s, F_s) \leftarrow$ rows $\{1, ..., |w|\} \backslash A(\theta_0)$ of $(G, w, F)$
if $|A| =$ dimension of $\lambda$ :
    $\hat{\lambda}^1(\theta) \leftarrow G_t^{-1} F_t \theta + G_t^{-1} w_t$
else:
    Row-reduce the linear system $\left[\begin{array}{c|c} G_t & -F_t \end{array}\right] \left[\begin{array}{c} \lambda^*(\theta) \\ \hline \theta \end{array}\right] = w_t$ to
    $\left[\begin{array}{c|c} U & P \\ \hline 0 & D \end{array}\right] \left[\begin{array}{c} \lambda^*(\theta) \\ \hline \theta \end{array}\right] = \left[\begin{array}{c} q \\ \hline r \end{array}\right]$
    if $D, r \neq 0$ :
        // This occurs w.p. 0
        Resample $\theta_0$ and restart.
    $\hat{\lambda}^1(\theta) \leftarrow -U^{-1} p + U^{-1} q$
$R_1 \leftarrow \{\theta : G_s \lambda^*(\theta) \leq w_s + F_s \theta\}$
$S_i \leftarrow$ polytope consisting of points that violates the $i$th of $R_1$ and satisfies
  constraints 1 through $i - 1$
Recursively partition each $S_i$ to get set of regions, optimizers $T_i$.
return $\{(R_1, \hat{\lambda}^1(\theta))\} \cup T_1 \cup T_2 ...$
```

of the parameter space defined by the loop bounds L list of two hundred disjoint polytopes. Programmatically examining this tiling scheme shows that the communication cost takes on one of five values, depending on the input parameters: $L_b L_c L_k L_w L_h L_r L_s / M$, $L_b L_c L_k L_w L_h (L_r L_s \sigma_W \sigma_H / M)^{1/2}$, or the size of *Input, Output,* or *Filter*; these are exactly (up to a constant factor) the communication lower bounds (4.1.1).

4.3 Performance Analysis

4.3.1 Comparison of Convolution Algorithms

Now that we know that the lower bound (4.1.1) is attainable, we wish to compare it to other approaches to performing convolutions. For instance, (2.3.4) states that it is impossible for any matmul-based convolution algorithm, such as im2col, to communicate less than $(\#\text{flops})/M^{1/2}$ words. As a result, asymptotically speaking, direct convolution using our tiling uses $\min(M^{1/2}, (L_r L_s/\sigma_h \sigma_w)^{1/2})$ times less communication than any possible matrix-multiply based convolution approach.

In practice, however, convolution sizes do not grow uniformly in every dimension; as a result, we are interested in non-asymptotic behavior. We will compare the communication cost of direct convolution using our tiling to four algorithms - im2col [78], Winograd [63], FFT [94], and a naive, untiled implementation - by symbolically calculating the amount of communication each one requires. We use the FFT communication bound provided in [22] and the matrix multiplication communication bound provided in [52] to compute the relevant communication volumes for the first two layers of Resnet50 [31], comparing them with our communication lower bound (4.1.1) The results are presented in Figure 4.3.1.

We observe several trends. Blocking and im2col scale better than FFT and Winograd in the memory size, and the relative performance of blocking and im2col is dependent on the ratio $\frac{\sigma_w \sigma_h}{L_r L_s}$. This is expected because of how the small filter blocking is used. We note that in all cases, the communication bound is not attained precisely. Work remains to either strengthen the communication bound or devise better algorithms to meet the bound.

4.3.2 Benchmarks on accelerators

To show real-world applicability of this tiling, we benchmark our results on a GEMMINI [26] machine learning accelerator running on Firesim [46], a cycle-accurate hardware simulator.

GEMMINI's memory architecture consists of two separate memory buffers: a *scratchpad*, which holds the input and the filter, and an *accumulator* buffer, which holds the output at a higher precision (to prevent floating-point rounding issues) and performs additions to it. At each tile, the input and the filter are reloaded from off-chip memory, but the partially summed output is held in the accumulator until it is fully summed (the loop ordering is fixed to ensure that the innermost loop axes in the outer loops correspond to reduction axes), at which point it is rounded and written off-chip in low precision.

We use the default GEMMINI chip configuration, in which, the scratchpad is 256KiB, holding 8-bit words, while the accumulator is 64KiB and holds 32-bit words. However, memory accesses are interleaved with computation using *double-buffering*, in which only half of the scratchpad and the accumulator are accessible to the processor at any one time (with the other half pulling in data from main memory). As a result, for tiling calculations, our memory sizes are halved: the scratchpad can hold 128K words, while the accumulator can hold 8K words.

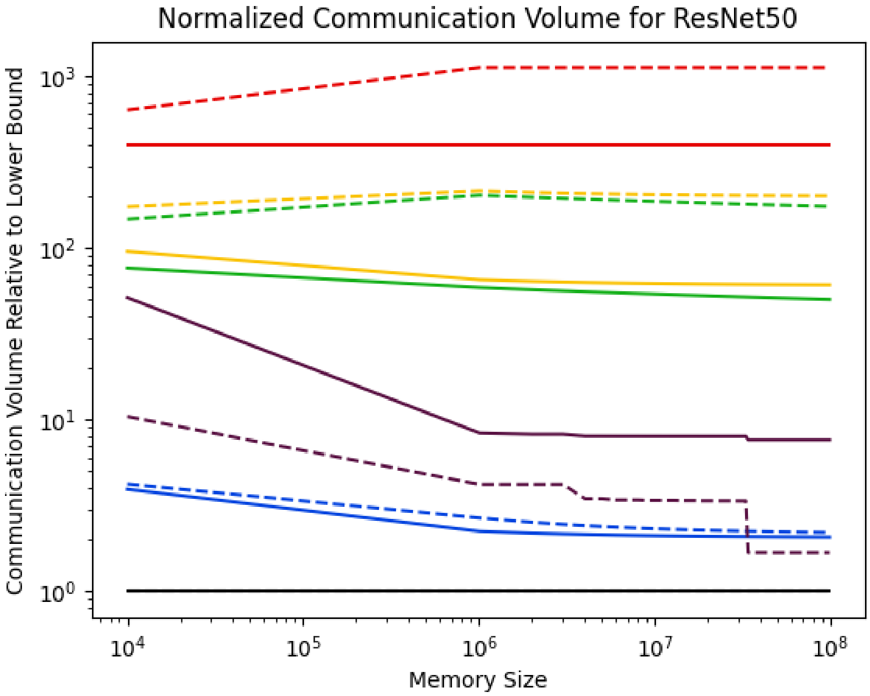


Figure 4.3.1: Theoretically computed communication volumes for mixed precision ResNet50 layers 1 and 2, relative to the communication bound. We take $\sigma_I = \sigma_F = 1$ and $\sigma_O = 2$. The communication bound is in black, im2col is in blue, blocking is in purple, green is FFT, yellow is Winograd, and red is naive. Solid lines are for layer conv1 and dashed lines are for the convolution of layer conv2_x as specified in [31]. We use a batch size of 1000. We see that in general, communication volumes are a constant multiple of the communication bound. However, we do see scaling in our tiled direct convolution, and for conv2_x, the strides of 1 are more favorable to our tiling, and it beats im2col for sufficiently large memory sizes. ResNet50 also has convolutional layers conv3_x, conv4_x, and conv5_x (not depicted), resemble conv2_x. Our tiling also has a sharp drop in performance towards the right of the graph for conv2, when the memory size grows sufficiently large to fit in memory; that drop occurs further to the right for other algorithms.

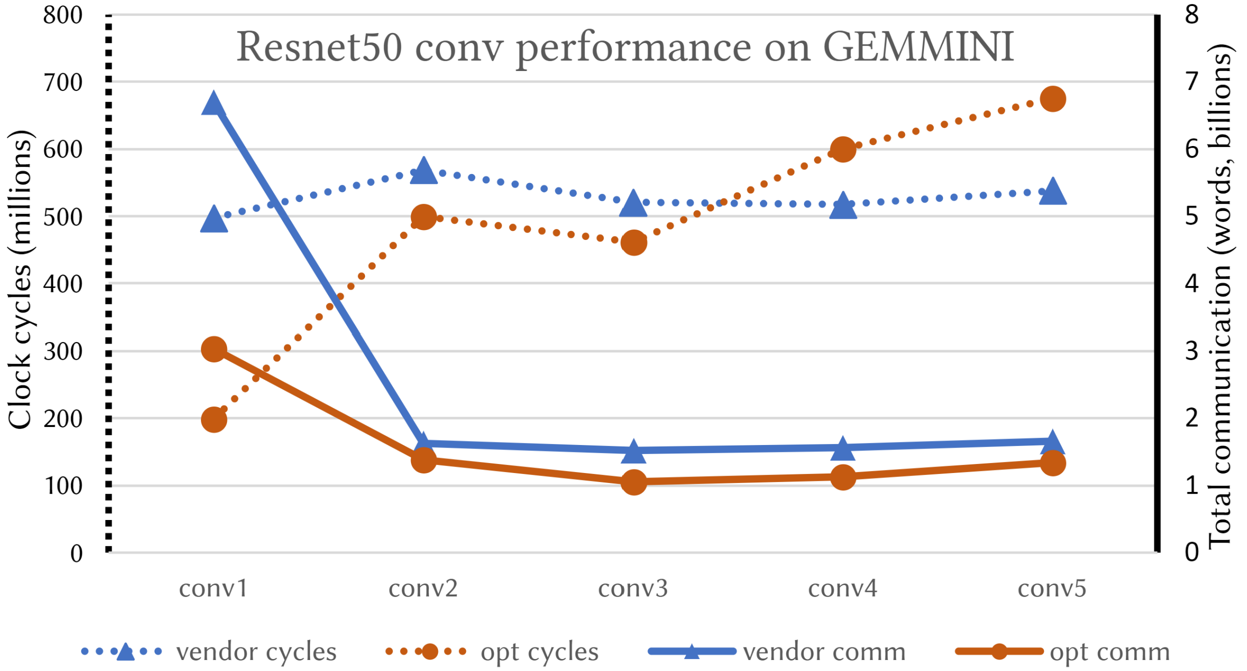


Figure 4.3.2: Experimentally measured Resnet50 layer performance (both total clock cycles and communication) on GEMMINI accelerator. Our optimization-generated tiling consistently uses less communication than the GEMMINI's default tiling ("vendor"), which leads to performance increases for layers where the default tiling has poor scratchpad utilization (convs 1 through 3).

As a result, we modify the optimization problem from 4.2 to account for buffer sharing between the input and the filter and to enforce integral tile sizes. Although this introduces nonlinearity (and an integrality constraint), the built-in numerical optimization routine `NMaximize` on Mathematica is still able to find a tile in around 400 iterations, or about five seconds on our test laptop.

We compare the performance of the five standard ResNet convolution sizes [31] evaluated on GEMMINI using both our tiling and the default tiler included with GEMMINI, which greedily expands the tile size, one dimension at a time, until the memory limit is exceeded. We measure both the estimated communication complexity (the number of scratchpad and accumulator rows allocated by chip's memory controller per tile, multiplied by the total number of operations divided by the size of a tile) and the counted number of clock cycles taken by each computation.

As shown in Figure 4.3.2, our system consistently uses between 45% and 85% as much estimated communication (a good proxy for energy[90]) compared to the default tiling on all ResNet layers. Furthermore, for convs 1, 2, and 3, where the default tiling was unable to take full advantage of the buffer (indicated by low scratchpad utilization per-tile), our tiling reduces clock cycle count (i.e. runtime) by 2.5× for conv1 and 13% for conv2 and conv3. However, for layers 4 and 5, where the default tiling already achieves scratchpad utilizations of 99% and 93% respectively, there is little room for improvement; in these cases, our tiling, which does not take into account non-memory related, hardware-specific factors such as optimal microkernel size and memory coalescing, performs worse with respect to clock cycles. In such cases, additional constraints may be added to encode these factors, as we discuss in the following chapter. For instance, for conv5, simply adding a single constraint forbidding the 7×7 image from being tiled (as an entire row will fit in a line of scratchpad) reduces cycles count from 124% to 104% of the default figure.

Chapter 5

Communication-Computation Combinations for Randomized Matrix Multiply

Our first step in examining models beyond pure communication will be to examine communication-computation tradeoffs in abstract models where

This chapter considers a simple example that exhibits a communication-computation tradeoff, from *randomized* linear algebra [66]. Specifically, we will consider the problem of multiplying a matrix A by a random matrix S. This kernel is often used for dimension reduction [40, 54]; A can be thought of as a collection of random vectors which we wish to map onto a lower-dimensional space using random projections S.

To be precise, our goal is to compute $\hat{A} = SA$, where:

- $\hat{A} \in \mathbb{R}^{d \times n}$
- $S \in \mathbb{R}^{d \times m}$ is a random matrix, generated at runtime using a pseudorandom number generator (PRNG).
- $A \in \mathbb{R}^{m \times n}$ is present in memory at the beginning of the program.

One straightforward approach to this problem is to compute S in its entirety, then perform a tiled matrix multiply (Algorithm 2). However, as S is the output of a PRNG, we can generate elements of S on the fly from a single stored seed (which is of negligible size), allowing us to use the entirety of fast memory to store A and $\hat{A}$. This exposes a tradeoff between *communication* cost, which is dependent on the size of fast memory available to store A, and the *computational* cost of possibly recomputing elements of S, which we will refer to as *rematerialization.*

To model this problem, we extend of the two-level memory hierarchy model in 2.3 to encompass computation as well. Each word communicated between fast and slow memory will incur a cost of 1, as before, but we now have the ability to generate an element of S

(and place it in fast memory) for a cost h. We will not count the number of operations incurred in actually performing the computations, since this is always $2dmn$. We will divide this cost into a *load cost*, which encompasses the number of reads from memory and h times the number of rematerializations, and a *store cost*, which describes the number of stores onto main memory.

In this chapter, we will derive a lower bound in this computational model for randomized matrix multiplication, and a tiling that attains it.

5.1 The Lower Bound

To find a lower bound for the randomized matrix multiplication problem, we will divide our program into segments similar to that from previous sections. However, instead of defining segments as being bounded by *communication*, we will define segments as being bounded by *total cost* $M+x$, where cost is defined as the sum of communication and h times the number of elements of S rematerialized. We will then optimize over these parameter x as we did in the proof of Lemma 18.

Lemma 21. *The load cost of computing $\hat{A} = SA$ in the two-level memory model where each element of S can be rematerialized for cost h is*

$$2dmn\sqrt{\frac{\min(h,1)}{M}}$$

Proof. Define the following notation: let $\hat{a}$, a, and s be the number of elements of $\hat{A}$, A, and S, respectively, available to compute during a segment. Let $\hat{a} = \hat{a}_b + \hat{a}_m$, $a = a_b + a_m$, and $s = s_b + s_m + s_r$, where

- $\hat{a}_b$, a_b, and s_b are the number of elements preexisting in fast memory at the beginning of a segment
- $\hat{a}_m$, a_m, s_m are the number of elements read from and written to slow memory during the execution of a segment
- s_r is the number of elements of S rematerialized during the execution of a segment.

By the Loomis-Whitney inequality (Theorem 2), the number of multiply-add operations per segment is bounded above by

$$\sqrt{\hat{a}as} \tag{5.1.1}$$

which we wish to maximize subject to constraints on the number of words available at the beginning of a segment:

$$\hat{a}_b + a_b + s_b \leq M \tag{5.1.2}$$

and the number of words that can be either read from main memory or rematerialized during a segment:

$$\hat{a}_m + a_m + s_m + hs_r \leq M + x \ . \tag{5.1.3}$$

Let us first consider the case of maximizing (5.1.1) for a fixed value of s. We have:

$$\begin{aligned} \hat{a} + a &= \hat{a}_b + \hat{a}_m + a_b + a_m && \text{by definition} \\ &\leq M + x - s_b + M - s_m - hs_r && \text{substituting from (5.1.2), (5.1.3)} \\ &= 2M + x - s_b - s_m - hs_r && (5.1.4) \end{aligned}$$

To maximize $\hat{a}a$ (and therefore $\sqrt{\hat{a}as}$) under (5.1.4), we set

$$\hat{a} = a = \frac{2M + x - s_b - s_m - hs_r}{2} \ .$$

As a result, for any $s = s_b + s_m + s_r$, the maximum of $\sqrt{\hat{a}as}$ is:

$$\begin{aligned} \frac{2M - s_b - s_m - hs_r}{2}\sqrt{s} &= \frac{2M + x - (s - s_r) - hs_r}{2}\sqrt{s} \\ &= \frac{2M + x + (1-h)s_r - s}{2}\sqrt{s} && (5.1.5) \end{aligned}$$

We now determine the optimal value of s_r. In the case where $h \leq 1$, the coefficient of s_r in (5.1.5) is positive, so we set s_r to its maximal legal value, $s_r = s$, giving us an objective of

$$\frac{2M + x - hs}{2}\sqrt{s} \ . \tag{5.1.6}$$

On the other hand, if $h > 1$, the coefficient of s_r in (5.1.5) is negative, so we set $s_r = 0$, giving us an objective of

$$\frac{2M + x - s}{2}\sqrt{s}$$

which is identical to (5.1.6) with $h = 1$; as a result, both objectives are equal to (5.1.6) with h replaced with $h' = \min(h, 1)$.

We now maximize this quantity with respect to s by differentiating with respect to s and setting to zero, which returns the optimal value of s as:

$$s = \frac{2M + x}{3h'}$$

which (substituting into 5.1.6) gives the following upper bound on the number of multiply-add operations per segment

$$\begin{aligned} \frac{2M + x - h's}{2}\sqrt{s} &= \frac{2M + x - h'\frac{2M+x}{3h'}}{2}\sqrt{\frac{2M + x}{3h'}} \\ &= \sqrt{\frac{(2M + x)^3}{27h}} \ . \end{aligned}$$

We now find a cost lower bound. The total number of arithmetic instructions is dmn. Therefore, the minimum number of segments required in the operation is

$$dmn\sqrt{\frac{27h}{(2M+x)^3}}$$

and as each segment incurs cost $M+x$, the minimum cost is

$$dmn(M+x)\sqrt{\frac{27h}{(2M+x)^3}}\ . \tag{5.1.7}$$

As this lower bound holds for any x, we wish to maximize (5.1.7) with respect to x to find the strongest lower bound. Differentiating with respect to x tells us that (5.1.7) is maximized at $x = M$, which gives a lower bound of

$$2dmn\sqrt{\frac{h'}{M}}$$

as desired. □

This lower bound is not always tight; for instance, if $h = 0$ (i.e. we can rematerialize elements of S for free), it provides a lower bound of 0 on the total load cost, which is clearly impossible. This suggests an approach for another lower bound: ignoring the costs (either memory or rematerialization) associated with S entirely - in other words, treat multiplication by the the appropriate value of S as as an arithmetic operation without any associated data or rematerialization cost, as in Algorithm 9. We now can perform a standard communication analysis for the projective data access functions

Algorithm 9: Randomized matrix multiply, ignoring S

```
for $x_d \in [0, d)$ :
    for $x_m \in [0, m)$ :
        for $x_n \in [0, n)$ :
            $\hat{A}(x_d, x_n) += f_{d,m}(A(x_m, x_n))$
```

$$\phi_{\hat{A}}(x_d, x_m, x_n) = (x_d, x_n)$$
$$\phi_A(x_d, x_m, x_n) = (x_m, x_n)$$

using Theorem 11, resulting in a communication (and therefore cost) lower bound of $\rho dmn/M$.

Combining this lower bound with with Lemma 21 gives:

Theorem 22. *The load cost of multiplying $\hat{A} = S \cdot A$, where $\hat{A} \in \mathbb{R}^{d\times n}$, $A \in \mathbb{R}^{m\times n}$, and $S \in \mathbb{R}^{d\times m}$, where S is a random matrix whose elements can be rematerialized at cost h, is at least*

$$\Omega\left(dmn \cdot \max(\sqrt{\frac{\min(h,1)}{M}}, \frac{1}{M})\right)\ .$$

5.2 Algorithms for Attaining the Lower Bound

In this section, we will develop algorithms that attain the lower bound in Theorem 22.

We will assume in this section that $h \leq 1$. To see why this assumption incurs no loss of generality, suppose we are given an algorithm $\mathscr{A}$ that computes $S \times A$ which is optimal for $h \leq 1$. Then, in the setting where $h > 1$, we simply perform $\mathscr{A}$, storing each $S(i,j)$ into slow memory the first time it is used loading them from slow memory upon subsequent uses. If some of these subsequent uses involve rematerializing $S(i,j)$, modify $\mathscr{A}$ so that they load $S(i,j)$ from slow memory instead, which reduces the total cost. The resulting algorithm costs at most dm more than the original algorithm (the maximum number of stores required), which is a lower order term since dm is the size of its input S and therefore a lower bound on the cost of $\mathscr{A}$.

As we have done in the rest of this dissertation, will denote the indices of the *elements* of $\hat{A}$, A, and S with indices x_d,x_n, x_m in parentheses; i.e. elements of $\hat{A}$ will be denoted $\hat{A}(x_d, x_n)$, elements of A will be denoted $A(x_m, x_n)$, and elements of S will be denoted $S(x_d, x_m)$. We will denote the indices of the *tiles* with indices t_d, t_n, t_m in square brackets; i.e. tiles of $\hat{A}$ will be denoted $\hat{A}[t_d, t_n]$, tiles of A will be denoted $A[t_m, t_n]$, and tiles of S will be denoted $S[t_d, t_m]$. We will denote indices of elements of *tiles* using square brackets followed by parentheses, e.g. $S[t_d, x_m](i_d)$.

We first attempt a simple tiling in Algorithm 10. As an initial attempt, let us tile $\hat{A}$ into tiles of size $\left(\sqrt{M/h} - 1\right) \times \sqrt{Mh}$, and have each inner loop consist of a rank-1 update to this $\hat{A}$-tile using a $\left(\sqrt{M/h} - 1\right) \times 1$ tile taken from S and a $1 \times \sqrt{Mh}$ taken from A. Note that as the tile size corresponding to the dimension m is 1, t_m and x_m are identical in Algorithm 10; we will use x_m. First, we check that this algorithm is executable. The total memory footprint of the $\hat{A}$ tile is $\left(\sqrt{M/h} - 1\right)\sqrt{Mh} = M - \sqrt{Mh}$, and the total memory footprint of the A tile is $\sqrt{Mh}$, leading to a total memory footprint of M (as S is never stored on fast memory, it does not count towards memory footprint), which is legal. Our only constraint, therefore, is that $\sqrt{Mh} \geq 1$; that is, $h \geq 1/M$. We first examine the case when it holds.

Let us analyze the cost of this algorithm. The number of loads from A is simply the size of a tile of A ($\sqrt{Mh}$) multiplied by product of the bounds of the loops surrounding Line 4 in Algorithm 10:

$$\frac{d}{\sqrt{M/h} - 1} \frac{n}{\sqrt{Mh}} m \sqrt{Mh} = \frac{dmn}{\sqrt{M/h} - 1}$$
$$= \frac{dmn\sqrt{h}}{\sqrt{M} - \sqrt{h}}$$

As we assume that $h \leq 1$, this is roughly equal to

$$dmn\sqrt{\frac{h}{M}}\ .$$

Algorithm 10: Tiled random matrix multiplication, with rank-1 updates

```
input  : input matrix $A \in \mathbb{R}^{m \times n}$
         random matrix $S \in \mathbb{R}^{d \times m}$, each of whose elements can be (re)materialized
         with cost $h$
output: output matrix $\hat{A} = S \times A \in \mathbb{R}^{d \times n}$
1 for $t_d \in [1, \frac{d}{\sqrt{M/h}-1})$ :
2     for $t_n \in [1, \frac{n}{\sqrt{Mh}})$ :
3         Initialize $\hat{A}[t_d, t_n] = 0$ in fast memory for $x_m \in [1, m)$ :
              // perform outer product $S[t_d, x_m] \times A[x_m, t_n]$ and add it to
                 $\hat{A}[t_d, t_n]$ (rank-1 update)
4             Load $A[x_m, t_n]$ into fast memory for $i_d \in [1, \sqrt{M/h} - 1)$ :
5                 Materialize $s = S[t_d, x_m](i_d)$ Increment $i_d$th row of $\hat{A}[t_d, t_n]$ by
                   $s \times A[x_m, t_n]$
6         Store $\hat{A}[t_d, t_n]$ to main memory.
```

Similarly, we can count the number of materializations of elements in S: a single element of S is rematerialized in Line 5 of Algorithm 10, so we just multiply the product of the bounds of the loops surrounding it to get

$$\frac{d}{\sqrt{M/h} - 1} \frac{n}{\sqrt{Mh}} m \left(\sqrt{\frac{M}{h}} - 1 \right) = \frac{dmn}{\sqrt{Mh}} \ .$$

Therefore, the total load cost of Algorithm 10 is roughly

$$dmn\sqrt{\frac{h}{M}} + h\frac{dmn}{\sqrt{Mh}} = 2dmn\sqrt{\frac{h}{M}}$$

which matches Theorem 22.

Furthermore, since each element of $\hat{A}$ is stored exactly once in Algorithm 10, it attains the trivial store cost lower bound of dn.

In the case where $h < 1/M$, we will (much like we did in Section 3.3.1) attempt to round the (illegal) tile dimension $\sqrt{Mh}$ to the nearest legal value, which is 1. This leads to a tile size of $(M - 1) \times 1$ for $\hat{A}$ and S and a 1×1 tile size for A. Following the same approach as above, the number of loads from A is

$$\frac{d}{M-1} \frac{n}{1} m \cdot 1 = \frac{dmn}{M-1} \approx \frac{dmn}{M}$$

and the number of materializations of elements in S is

$$\frac{d}{M-1} \frac{n}{1} m \, (M - 1) = dnm$$

for a total cost of approximately

$$\frac{dmn}{M} + hdmn = \left(\frac{1}{M} + h\right) dmn \leq \frac{2dmn}{M}$$

which again matches Theorem 22.

Chapter 6

Optimizing More Realistic Performance Models and Simulated Performance

In the previous chapters, we have worked with abstract computational models in which we can prove strong lower bounds on cost, as well as describing mappings to attain these lower bounds. However, these models make many simplifying assumptions; for instance, we have ignored the cost of tail cases in previous chapters as lower-order terms. However, in practice, these simplifications, combined with the lack of accounting for architecture-specific factors such as memory alignment and interconnect topology, can cause abstract model performance to be an imperfect proxy for actual performance, as we saw in Section 4.3.2.

Analytical performance models for accelerators, such as Timeloop [71] and Maestro [53] incorporate significantly more information about target hardware than the simple models we have used so far, giving performance figures that can be far closer to actual performance than more abstract models such as communication volume. This hardware-specific also allows them to directly handle more axes in the mapspace than our previous analyses; for instance, Timeloop supports multilevel tiling, spatio-temporal mapping, and loop ordering.

Unfortunately, analytical performance models are sufficiently complex that proving nontrivial lower bounds under them are infeasible. However, optimization methods for these analytical models can provide excellent performance in practice on real hardware, even if it is infeasible to prove optimality over them. As most analytical performance models lack closed-form descriptions, we cannot directly solve for an optimal mapping as we have done in previous chapters; we will discuss methods to work around this limitation in this chapter.

However, analytical performance models are not perfect. Despite their significant advantages with respect to accuracy over abstract computational models, analytic models can still diverge from actual performance significantly [88]. To see the significance of this difference, we generated 2000 randomly chosen mappings for a variety of convolution and matrix multiplication problems. We then evaluated the cost of these mappings, first by executing the code corresponding to these mappings on the GEMMINI [26] DNN accelerator running on the cycle-accurate Firesim [46] RTL emulation platform, then by using an inexpensive analytic performance model, Timeloop. Hardware parameters (memory bandwidth/sizes, systolic array dimensions, etc.) were identical for both targets. Figure 6.0.1 shows a log-log scatter plot of the ratios of the cycle counts generated by Timeloop and Firesim, which can differ by up to two orders of magnitude.

However, using measured performance data comes with its own caveats, especially during the hardware design process, when the target hardware has not been taped out yet. In this setting, cycle-accurate simulation or emulation can be used, but at a cost several orders of magnitude more expensive than running an analytical model. As a result, optimization methods targeting measured performance data must be *sample-efficient* to be feasible.

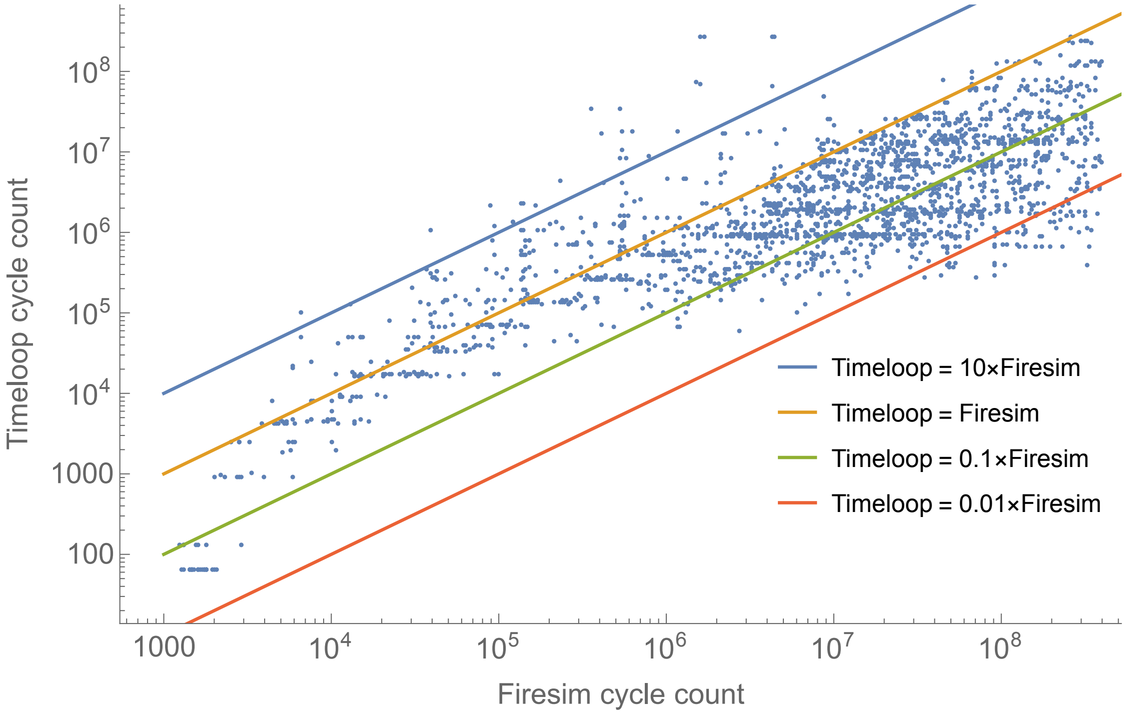


Figure 6.0.1: Comparison of EDP of various mappings from the Firesim emulator and the Timeloop analytical performance model.

This chapter describes two optimization techniques suited to these performance objectives:

- First, in Section 6.1, we will describe how a *manually written* closed-form surrogate model can be used to obtain good performance on an analytical performance model. As the surrogate model is of closed form, it can be directly optimized using commercial optimization tools.

- We will then describe in Section 6.2 how Bayesian optimization, with a domain-specific parameter encoding, can be used to iteratively optimize mappings in a sample-efficient manner. Furthermore, Bayesian optimization produces a surrogate closed-form performance model which can be used for *transfer learning* to increase the efficiency of mapspace search for unseen hardware.

Work in this chapter was previously published as [38, 20].

6.1 Directly Optimizing Proxy Models

As both analytical performance models and measured performance models do not have closed-form mathematical expressions, our first approach will be to manually construct a surrogate model for performance which we can mathematically optimize. Our approach in this section will be to augment the optimization programs we used to construct optimal tilings in Chapters 3 and 4 to incorporate spatio-temporal mapping and loop permutation.

In this and the next section, we will focus on convolutions and matrix multiplications, and only consider rectangular tilings. While non-rectangular tilings of the filter were shown to be necessary for optimality for some convolutions in Chapter 4, most filter dimensions run in practice (e.g. Resnet's [31]) are very small, ranging from 3×3 to 7×7, and as a result most hardware and software platforms do not support tiling along the filter dimension and are therefore limited to rectangular tilings.

6.1.1 Representing mappings

We must first describe a unified mathematical representation of a mapping - tile sizes, loop orderings, and spatio-temporal mappings - graphically described in Figure 2.2.2. We will do so by representing mapping as an *allocation problem.* Each loop will be split into *subloops*, one for each prime factor of the original loop bound.

Each of these subloops will then be assigned to a level of the memory hierarchy, providing a tiling. Within each level of the memory hierarchy, each subloop will be assigned a rank, specifying loop permutation, and an indication of whether the subloop is to be mapped spatially or temporally. Formally, we represent a mapping as a four-dimensional tensor X, whose values are either 0 (represented by a blank space) or 1 (represented by a checkmark). X is indexed by four indices:

- j, representing the loop axis,
- n, representing subloops (each corresponding to a prime factor of the the loop bound corresponding to loop axis j)
- k, representing whether a subloop is mapped spatially (index 0) or temporally (index 1)
- i, which represents the level of the memory hierarchy and loop permutation ordering. Specifically, each level of the memory hierarchy has Z 'slots'. For each level of the memory hierarchy, subloops assigned to that level will be permuted in the order of

Index			Permutation	Schedule								
j	Loop axis			R = 3		...	K = 4				N = 3	
n	Prime Factors			3		...	2		2		3	
k	Spatio-temporal Mapping			s	t		s	t	s	t	s	t
i	Memory Levels	Register	...									
		...										
		Input Buffer	...				✓					
		Global Buffer	O_0									
			O_1									✓
			O_2						✓			
			...									
			O_Z	✓								

Figure 6.1.1: Allocation-based mapping encoding

rank, with the lowest-ranked factor being innermost, and the highest-ranked factor outermost. The number of permutation ranks Z is the number of total subloops (i.e. the total number of prime factors of all the loop bounds).

$X_{(j,n),i,k}$ being 1 denotes that subloop n of loop j is assigned to the memory level/order i, and the subloop is mapped either spatially or temporally depending on the value of k. All other values of X are zero.

A concrete example of such a tensor is given in Figure 6.1.1, denotes a mapping for a nested loop with loop bounds $R = 3$, $K = 4$, and $N = 3$ onto an accelerator with three levels of the memory hierarchy: a global buffer, an input buffer, and a register. Here, dimension K is split into two tiles, where the inner tile of size 2 is allocated to the input buffer, and the outer tile of size 2 is allocated in the global buffer. Each level of the memory hierarchy in X is further divided into *permutation ranks* $O_0, O_1, ..., O_Z$ where

Whether a subloop is executed spatially or temporally is indicated by whether the factor is mapped to a spatial column s or a temporal column t in the table. In this example, both prime factors for K are spatially mapped.

We note that this factorization-based representation restricts our tilings to those that perfectly divide the loop bounds; i.e. no tiling with a tail case can be generated. However, in practice, this is not a significant concern: problem sizes for most machine learning workloads tend to have small prime factors (many are powers of 2, in fact), and the few that are not can be zero-padded to a size that factors into small subloops.

6.1.2 Mapping Constraints

We now consider what constraints a mapping must satisfy in order to be valid. First, we have memory (buffer capacity) constraints. As in the analysis of Algorithm 10, the memory footprint of a tile is simply the product of the loop bounds of all subloops within it. Let

$A_{j,v} = 1$ if tensor v is indexed by index j and 0 otherwise; this generalizes the constraint matrix of (3.0.4).

We will also account (as we did in Section 4.3.2) for the fact that some tensors have dedicated buffers, e.g. the accumulator buffer for the output tensors. To formalize this, let $B_{I,v} = 1$ if tensor v may be stored in buffer I and zero otherwise.

Then the memory footprint of tensor v at level I is

$$\prod_{i\in[0,I-1]} \prod_{j,n,k} \begin{cases} prime\_factor_{j,n}, & X_{(j,n),i,k}A_{j,v}B_{I,v} = 1 \\ 1, & \text{otherwise} \end{cases} \tag{6.1.1}$$

which we restrict to a maximum value of $M_{I,v}$, the space allocated to tensor v at memory buffer I. Taking logs

$$\sum_{i\in[0,I-1]} \sum_{j,n,k} \log(prime\_factor_{j,n})X_{(j,n),i,k}A_{j,v}B_{I,v} \leq \log(M_{I,v}) \qquad \forall I\ . \tag{6.1.2}$$

We have a similar constraint for parallelism: the amount of parallelism exposed by spatially-mapped X must not exceed the amount of parallelism available. Let buffer I expose S_i parallelism (for instance, if I corresponds to shared memory in a multicore system, S_i would be the number of cores connected to shared memory). Then the amount of parallelism exposed at level I is product of the subloop bounds corresponding to subloops mapped spatially is

$$\prod_{j,n} \begin{cases} prime\_factor_{j,n}, & X_{(j,n),I,0} = 1 \\ 1, & \text{otherwise} \end{cases}$$

which leads to the following constraint:

$$\sum_{j,n} \log(prime\_factor_{j,n})X_{(j,n),I,0} \leq \log(S_I) \qquad \forall I\ . \tag{6.1.3}$$

As no loop may be simultaneously mapped both spatially and temporally, we must also have

$$\sum_{k\in\{0,1\}} X_{(j,n),i,k} = 1 \qquad \forall j,n,i\ .$$

6.1.3 Optimization

With the constraints defined, we can an objective function - that is, a surrogate model that can be numerically optimized over the constraints described above. We describe several possible objective functions in this section, which can be optimized for individually or combined into a weighted sum.

The (normalized) logs of the memory footprint (6.1.2) and compute utilization (6.1.3) are both possible objectives, as maximizing utilization of hardware resources is generally correlated with higher performance. Furthermore, as linear functions in the mapping variables X, they are easy for optimization libraries to tackle.

We also wish to determine the communication cost of a mapping for each tensor v. We will decompose this as the product of three terms, or equivalently, we will decompose the log of the communication cost as the sum of three terms:

- The **data size per tile** is simply the total memory footprint (across all buffers), i.e. (6.1.1) without the $B_{I,v}$ factors. Its log is therefore

$$D_v := \sum_{i \in [0, I-1]} \sum_{j,n,k} \log(prime\_factor_{j,n}) A_{j,v} X_{(j,n),i,k}$$

- The **amount of spatial communication** incurred on a spatial mapping. When a subloop is spatially mapped, the tensors are either *unicasted* or *multicasted* (or accumulated) to all processing elements. To determine which is the case, notice that tensor v is unicasted or accumulated if the index j being spatially mapped does not index the tensor (i.e. $A_{j,v} = 0$), as in this case, all processing elements access the same elements of the tensor. Otherwise, different parts of tensor v must be distributed to each processing element; this multicast incurs additional memory traffic by a factor of the number of parallel targets. Taking logs:

$$L_v := \sum_{j=0,n=0}^{6,N_j} \log(prime\_factor_{j,n}) X_{(j,n),I,0} A_{j,v}$$

- A **temporal iteration** multiplier. If a subloop n of loop j is temporally mapped, the tensors indexed by loop indices nested inside this subloop are accessed $prime\_factor_{j,n}$ times. To determine if a tensor v has an indexed in loop j using arithmetic functions (to avoid branching), we introduce an additional zero-one indicator variable $Y_{v,z}$, constrained as follows

$$\begin{aligned} Y_{v,z} &\geq \sum_{j,n} X_{(j,n),z,1} A_{j,v} B_{I,v} \qquad \forall z, v \\ Y_{v,z} &\geq Y_{v,z-1} \qquad \forall z > 0, \forall v \end{aligned} \tag{6.1.4}$$

  The log of the temporal iteration multiplier can therefore be expressed as

$$T_v = \sum_{z=0}^{Z-1} \sum_{j=0,n=0}^{6,N_j} \log(prime\_factor_{j,n}) Y_{v,z} X_{(j,n),z,1} \tag{6.1.5}$$

  which is quadratic in variables Y and X.

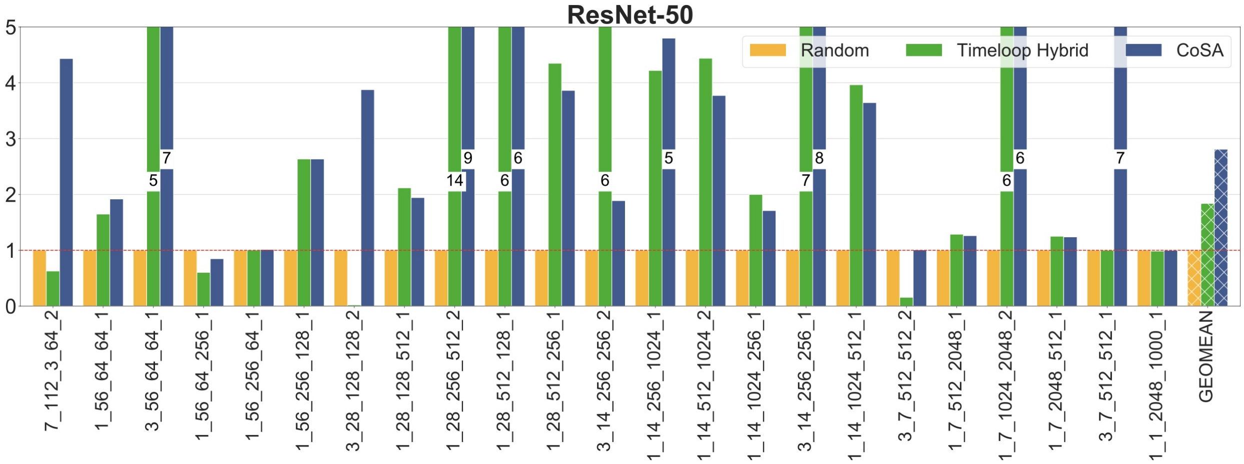


Figure 6.1.2: Runtime of optimization-generated schedule (CoSA) relative to schedules found by random search and Timeloop's hybrid search. All runtimes are normalized to that off random search.

The log of the total traffic for tensor v is simply $D_v + L_v + T_v$.

In practice, it is often useful to minimize a weighted sum of these different objectives, with the weights determined by microbenchmarks describing the costs of communication and computation on a specific target architecture. This quadratic optimization problem can be directly solved by commercial mixed integer linear program solvers such as Gurobi [29], and as seen in Figure 6.1.2, provides mappings for Resnet50 layers that on average outperform both those given by brute-force search and Timeloop's *hybrid search* [67] mapper, which prunes then linearly explores mappings around randomly chosen loop factorizations. As solving this optimization program does not rely on sampling points, it determines a mapping roughly 90 times faster than Timeloop's hybrid search.

6.2 Bayesian Optimization

As a result, we wish to develop feedback-driven methods for finding performant methods with *high sample efficiency*, allowing their use on expensive but accurate performance models. Furthermore, we would like our approach to *generalize* inexpensively to new hardware configurations. Our approach is to to use Bayesian optimization, which has been shown to be effective for optimizing complex functions with a limited number of evaluations due to its faster convergence and ability to handle multiple parameters. One of its key strengths is its ability to automatically construct closed-form *surrogate models* using Gaussian processes, which are powerful tools for modeling complex interactions and potentially noisy functions; these can be directly optimized over or incorporated into other performance models.

Bayesian approaches have been used to perform black-box optimization in domains where sample efficiency is paramount, including algorithm optimizations on supercomputers [58, 11] and optimizing hardware parameters for accelerators [74, 61, 89]. We apply similar techniques to optimize over *software* mapspaces for accelerators, improving on previous attempts to do so [80] by using an efficient encoding scheme to embed mapping parameters into a mathematical space that can be more easily searched.

This section considers mapspaces consisting of tile sizes and loop orderings (dataflows) for multilevel memory hierarchies. Our techniques directly generalize to mapspaces including spatio-temporal mappings as well; we leave benchmarking those to future work.

6.2.1 Mapspace Encoders

Bayesian optimization assumes an objective function (in this case, a performance metric such as latency, cycle count, or energy) that takes as input a set of numerical, usually continuous, variables. However, decisions that comprise a point in the mapspace, such as loop orderings, are discrete. These discrete variables fall into one out of two categories.

Discrete numerical variables, such as tile sizes, are mostly integral. However, depending on the hardware target, their discreteness may take the form of a requirement to be a multiple (e.g. of the size of a vector unit) or factor (e.g. of the problem size, to allow for perfectly nested loops without tail cases) of some integer. We represent such variables as continuous variables in the optimization program and round them in order to find the actual mapping parameters; such rounding approaches have experimentally been shown to match or exceed discrete surrogate based approaches [47].

Categorical variables, such as loop orderings, are members of a finite, unordered set. These variables can be dealt with in one of two ways:

- By *directly applying surrogate-based optimization approaches* to them. Handling categorical variables in optimization, especially in Bayesian optimization, poses unique challenges due to their discrete and unordered nature, especially in domains comprised of both continuous and categorical variables. Several approaches for integrating the continuous and categorical optimization methods have been studied, including one-hot encoding [81], bandit models [75], and hybrid Monte Carlo tree search [58]. However, the combinatorial complexity of the mapping problem complicates such approaches; for instance, a batched convolution with seven nested loops has $7! = 5040$ possible loop orderings per memory level, and general categorical optimization methods are unable to take problem-specific information that could guide search over this space into account (for instance, that performance is likely to be changed less by swapping the order of two loops than reversing the entire loop nest).

- By creating a *mapspace-specific encoding* from continuous variables. For example, for loop orderings, we optimize over *scores* for each axis and order the axes from lowest to highest score. A similar approach, as in [55], can be used for spatio-temporal mappings.

Dealing with hardware constraints. The set of valid mappings is bound by a set of constraints, which we will address in this section by developing a mapping from an *uncon-*

strained feature space $\mathbf{f} = (0,1]^d$ (for some integer d), which can be easily optimized over, to the set of valid mappings. We will denote axes of this feature spaces as f_i.

Many constraints are simple constant bounds on the numeric variables (for instance, ensuring that tile sizes must be smaller than the sizes of the data tensors) and can be dealt with by scaling the variables appropriately. For instance, instead of optimizing a loop tile size t under the constraint that it is bounded above by the size s of the input problem, we can instead optimize a value $f \in (0,1]$ and set $t = sf$.

However, other constraints may result in more complicated inequalities. For example, consider a 2D convolution with b batches, c input channels, k output channels, and windows of size $r \times s$, outputs of size $w \times h$. If we wish to tile these axes in such a way that the tiled inputs and weights can fit into a scratchpad of size M, the tile sizes $t_b, t_c, ...$ must satisfy the following constraint:

$$t_c t_k t_r t_s + t_b t_c (t_w + t_r)(t_h + t_s) \leq M \tag{6.2.1}$$

Rejection sampling is often used to handle such constraints, but has two drawbacks. First, setting an objective value to assign to invalid mappings is a nontrivial hyperparameter optimization problem; an overly high value can cause unwanted behavior in a learned surrogate function (leading to unpredictable behavior when, for instance, doing transfer learning), while an overly low value may not be enough to discourage the optimizer from considering invalid maps. Furthermore, the rejection probability can be high - increasingly so as the dimensionality increases - significantly driving up the number of iterations required. In fact, prior work [80] requires sampling $22K$ points in order to produce 150 valid mappings, which drastically increases the cost of this approach.

As a result, our goal is to develop a mapping from every point of $\mathbf{f}$ to a *valid* point in the mapspace. Since all nontrivial constraints in the mapspace take the form of capacity constraints similar to that of (6.2.1) over the tile sizes [38], we instead optimize the *aspect ratio* of the tiles, and then scale all the tile sizes by the same factor to maximize memory utilization. We believe this approach also improves the ability of the learned model to generalize across problem sizes, as communication-optimal tiles for many problems such as matrix multiplication retain the same aspect ratio (square tiles) as long as problem sizes are sufficiently large.

More concretely, consider the memory constraint given in (6.2.1). Instead of directly optimizing over the tile sizes $t_{b,c,...}$, we optimize the variables $f_{b,c,...} \in (0,1]$, which we scale by a *common* multiplier α to obtain

$$t_{b,c,...} \approx \alpha f_{b,c,...} \tag{6.2.2}$$

In order to determine the value of α, notice that substituting (6.2.2) into (6.2.1) gives the following inequality:

$$\alpha^4 \left[f_c f_k f_r f_s + f_b f_c (f_w + f_r)(f_h + f_s) \right] \leq M \tag{6.2.3}$$

As there is no reason not to maximize memory utilization, we replace the inequality with equality, which therefore provides the value of α:

$$\alpha = \left(\frac{M}{f_c f_k f_r f_s + f_b f_c (f_w + f_r)(f_h + f_s)} \right)^{1/4}$$

We can then round the resulting values of $\alpha f_{b,c,...}$ down to the nearest valid value (to satisfy discreteness and maximum tile size constraints) of $t_{b,c,...}$, ensuring that each point $\vec{f} \in (0,1]^d$ corresponds to a valid mapping.

6.2.2 Evaluation

For our experiments, we optimize for energy cost on a hardware model based on GEMMINI[26] with a four-level memory hierarchy: a register, an accumulator for the outputs, a scratchpad for the inputs and weights, and DRAM. We test our mappings on Timeloop[71], which takes as input an algorithm and a hardware configuration and provides (1) an analytic performance *model* for energy and latency and (2) a pruned random search based *mapper*. While our approach is designed to target hardware models with far higher per-sample cost than than Timeloop's model, we use Timeloop in order to allow for the use of its random-search mappers (which would be infeasibly expensive if run with a cycle-accurate simulator) as a comparison target. We leave benchmarking on an (expensive) cycle-accurate simulator and comparing performance to model-based (brute-force and heuristic) mappers to future work.

To perform Bayesian optimization, we use GPTune [12], an autotuning suite designed for optimizing applications by utilizing Bayesian approaches. GPTune incorporates multi-task learning and transfer learning algorithms to share knowledge of obtained performance samples among multiple tasks, improving tuning results. It enables quick prediction of optimal tuning parameters for new tasks using data from existing tasks. Additionally, GPTune supports multi-objective tuning, hybrid models [58] for mixed categorical and continuous variables, and non-smooth objective tuning [59].

In order to reduce statistical variance, all experiments were averaged over three independent runs.

6.2.3 Convergence

Figure 6.2.1 shows the energy consumption of the best mapping found so far at each iteration, comparing Timeloop and GPTune (we run 100 iterations for GPTune).

For GPTune, we show results for both approaches to optimizing over categorical variables (in this case, loop orderings) described in Subsection 6.2.1. We note that directly embedding loop orderings into the problem produces superior results to the score-based approach for matrix multiplication but inferior results for convolutions, likely because of the higher dimensionality of convolutions compared to matmuls: a 3-nested loop matmul leads to roughly

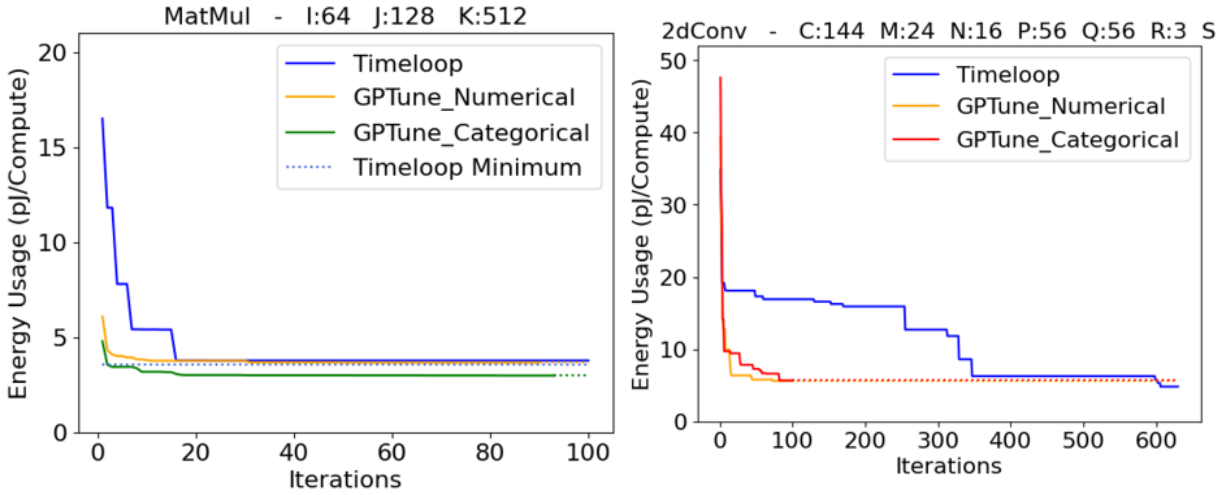


Figure 6.2.1: Energy (lower is better) attained by Timeloop's brute-force mapper and GPTune for matrix multiplication (left) and 2D convolution (right). GPTune was run for 100 iterations; the best value found is indicated with dotted line extending to the right.

$(3!)^4 = 1296$ choices for loop orderings over the four levels of the memory hierarchy, while the 7-nested loop convolution results in roughly $(7!)^4 \approx 6e14$ choices. This suggests using categorical encodings works well for relatively low-dimensional problems, whereas score-based encodings are better for higher-dimensional problems.

For matrix multiplication, GPTune converges in roughly 20 runs to 2.98 pJ/compute, a value 16% better than the 3.56 pJ/compute that Timeloop achieves after 4000 runs (note that Timeloop plateaus after an average of 420 iterations).

For 2D convolutions, GPTune converges in (on average) 50 iterations to a minimum of 5.26 pJ/compute, a value that it took an average of 627 iterations for Timeloop to beat. Furthermore, after 4000 iterations, Timeloop's best value was 4.32 pJ/J, roughly 17% better than GPTune's.

6.2.4 Transfer Learning

In many settings, such as hardware DSE, the ability to leverage data collected on one or more hardware configurations to guide search on a hitherto unseen hardware configuration can prove useful. However, support for transfer learning across hardware configurations has proven limited so far. Random search and many black-box optimization algorithms, such as genetic algorithms (e.g. GAMMA[43]) do not support transfer learning and must be run from scratch for every hardware target and algorithm. Attempts to apply the differentiable surrogate models found by Mind Mappings[32] to hardware architectures not in the training set resulted in performance one to two orders of magnitude worse than running Timeloop's random mapper and GAMMA from scratch. Previous Bayesian optimization based approaches to mapspace search [80] have not considered transfer learning.

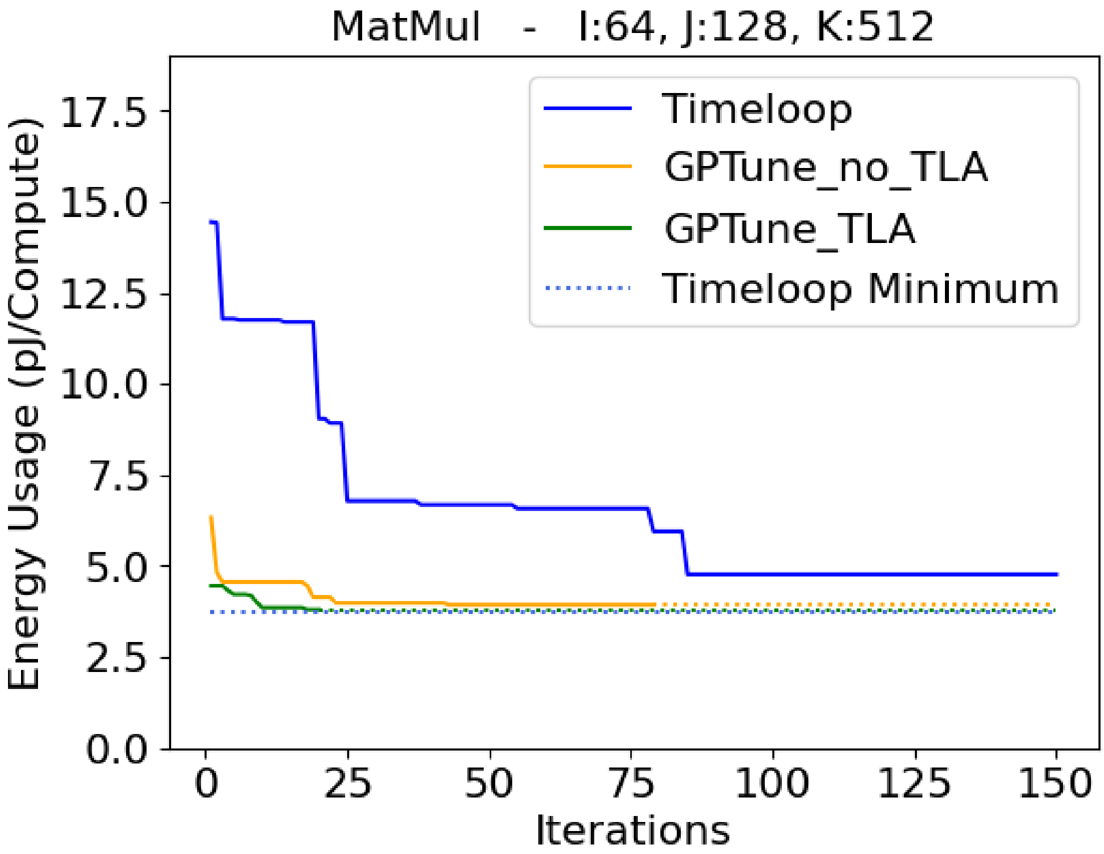


Figure 6.2.2: Transfer learning to a new (not in training set) hardware configuration for matrix multiplication, compared to GPTune with no prior knowledge and Timeloop.

The Gaussian process surrogate models produced by GPTune possess the capability to facilitate transfer learning. We first train a surrogate model taking into consideration both task parameters (i.e., tensor dimensions) and hardware parameters (i.e., memory hierarchy specifications), utilizing GPTune's multitask learning algorithm for four distinct memory hierarchy configurations. Subsequently, we refine this model for 20 iterations, employing the target hardware configuration that was absent from the initial training set.

Figure 6.2.2 shows the transfer learning, which converges to a mapping providing 3.81 pJ/compute (on par with an uninitialized GPTune) in 10 iterations (roughly half that of an uninitialized GPTune). This figure requires Timeloop an average of 1600 iterations to beat.

6.2.5 Sensitivity Analysis

The surrogate models can be used for sensitivity analysis as well, by applying Sobol analysis[82] to attribute the part of the variance of the output can be attributed to each of the inputs. We leverage GPTune's sensitivity analysis interface, which internally invokes

SALib[34] for computing Sobol indices from the trained surrogate model. For matrix multiplication, the most important axes were the tilings of the $64 \times 512 \times 128$ matrix multiplication example shown in Figure 6.2.1, the most important axes were the tilings of k at the register and accumulator levels, and the tiling of j at the register level; the surrogate model was several orders of magnitude more sensitive to the tiling parameters than the loop ordering ones, which lines up with previous work[90, 44] showing that tilings are the most important mapping parameter.

For high-dimensional problems such as convolutions, we believe this surrogate model may be used to perform automated dimension reduction - perhaps even during the optimization process itself; we leave this to future work.

Chapter 7

Conclusion and Future Work

In this dissertation, we have attacked the problem of finding performant mappings for various tensor problems by expressing them as *optimization problems* targeting various performance objectives. For certain performance models, we have derived *lower bounds* on the cost of computation.

Perhaps the most straightforward application of our work is to the design of high-performance tensor libraries and compilers, especially those targeting domain-specific accelerators. We leave implementing our work as compiler optimizations and integrating them into existing compilers (either as fully automated rewrite passes, or as guides to programmers writing optimizations in user-schedulable languages such as Halide, Exo, and TVM) as future work, pointing to [69, 68] as an initial step towards this goal.

Furthermore, the work in this dissertation can also be applied to hardware *design-space exploration* (DSE), in which different hardware configurations and parameters are explored to optimize the performance of a set of target workloads over power, performance, and area (PPA) budgets. Accurately evaluating performance of a workload on a candidate hardware design requires the computation of a performant mapping from the workload. As the hardware target is often defined by only a few parameters, and changes often during the search process, it is most straightforward to target an abstract computational model or analytical performance model. In [35], we combine a differentiable analytical performance model similar to that described in Section 6.1 with measured performance figures in order to provide an objective used by a gradient-descent optimizer to perform DSE, targeting a multilayer neural network. This is also an example of *multi-fidelity optimization* - combining multiple multiple performance models with different execution times and fidelities; exploring it more generally is something we leave to future work.

Sparse tensor algorithms (and accelerators for them) have also seen significant recent interest. Extending our lower bound techniques to support sparse computations is a natural progression from our work. One promising direction is to adapt tools from database theory. As we briefly discussed in Chapter 3, the discrete Brascamp-Lieb inequality in the projective case (Theorem 11) is in fact equivalent to the *AGM bound* of Asterias et al. [2], a powerful tool for bounding the cardinality of a join query based on the cardinalities of its individual relations. However, there are cardinality bounds based on *database statistics* beyond simple cardinality, such as degree sequences [16, 49] which may provide a way to find new data-dependent lower bounds and algorithms for families of sparse matrices.

Bibliography

[1] Aravind Acharya, Uday Bondhugula, and Albert Cohen. 2018. Polyhedral auto-transformation with no integer linear programming. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, Philadelphia PA USA, (June 2018), 529–542. ISBN: 978-1-4503-5698-5. DOI: 10.1145/3192366.3192401.

[2] Albert Atserias, Martin Grohe, and Dániel Marx. 2013. Size Bounds and Query Plans for Relational Joins. *SIAM Journal on Computing*, 42, 4, (Jan. 2013), 1737–1767. DOI: 10.1137/110859440.

[3] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: a polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (CGO 2019). IEEE Press, Washington, DC, USA, (Feb. 2019), 193–205. ISBN: 978-1-72811-436-1. Retrieved Aug. 4, 2023 from.

[4] Jonathan Bennett, Anthony Carbery, Michael Christ, and Terence Tao. 2008. The Brascamp–Lieb Inequalities: Finiteness, Structure and Extremals. *Geometric and Functional Analysis*, 17, 5, (Jan. 2008), 1343–1415. DOI: 10.1007/s00039-007-0619-6.

[5] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. 2015. PLuTo: A practical and fully automatic polyhedral program optimization system. In.

[6] F. Borrelli, A. Bemporad, and M. Morari. 2003. Geometric algorithm for multiparametric linear programming. *Journal of Optimization Theory and Applications*, 118, 3, (Sept. 2003), 515–540. DOI: 10.1023/B:JOTA.0000004869.66331.5c.

[7] Anthony Chen, James Demmel, Grace Dinh, Mason Haberle, and Olga Holtz. 2022. Communication bounds for convolutional neural networks. In *Proceedings of the Platform for Advanced Scientific Computing Conference* (PASC '22). Association for Computing Machinery, New York, NY, USA. ISBN: 978-1-4503-9410-9. DOI: 10.1145/3539781.3539784.

[8] Beidi Chen, Tharun Medini, James Farwell, Sameh Gobriel, Charlie Tai, and Anshumali Shrivastava. 2020. SLIDE : In Defense of Smart Algorithms over Hardware Acceleration for Large-Scale Deep Learning Systems. *Proceedings of Machine Learning and Systems*, 2, (Mar. 2020), 291–306. Retrieved Nov. 20, 2023 from https://proceedings.mlsys.org/paper_files/paper/2020/hash/ca3480d82599b9b9b7040655483825c1-Abstract.html.

[9] Tianqi Chen et al. 2018. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 578–594.

[10] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '14). ACM, New York, NY, USA, 269–284.

[11] Younghyun Cho, James W Demmel, Jacob King, Xiaoye S Li, Yang Liu, and Hengrui Luo. 2023. Harnessing the crowd for autotuning high-performance computing applications. In *The 37th IEEE International Parallel and Distributed Processing Symposium (IPDPS23)*. IEEE, 1–12.

[12] Younghyun Cho, James W. Demmel, Grace Dinh, Xiaoye S. Li, Yang Liu, Hengrui Luo, Osni Marques, and Wissam M. Sid-Lakhdar. 2022. GPTune user guide. (2022). `https://github.com/gptune/GPTune/tree/master/Doc`.

[13] Michael Christ, James Demmel, Nicholas Knight, Thomas Scanlon, and Katherine Yelick. 2015. On Holder-Brascamp-Lieb inequalities for torsion-free discrete Abelian groups. (Oct. 2015). arXiv: `1510.04190 [math]`. DOI: `10.48550/arXiv.1510.04190`.

[14] Michael Christ, James Demmel, Nicholas Knight, Thomas Scanlon, and Katherine A. Yelick. 2013. Communication Lower Bounds and Optimal Algorithms for Programs That Reference Arrays - Part 1: tech. rep. Defense Technical Information Center, Fort Belvoir, VA, (May 2013). DOI: `10.21236/ADA584726`.

[15] Shail Dave, Youngbin Kim, Sasikanth Avancha, Kyoungwoo Lee, and Aviral Shrivastava. 2019. DMazeRunner: Executing perfectly nested loops on dataflow accelerators. *ACM Transactions on Embedded Computing Systems*, 18, 5s, (Oct. 2019), 1–27. DOI: `10.1145/3358198`.

[16] Kyle Deeds, Dan Suciu, Magda Balazinska, and Walter Cai. 2022. Degree Sequence Bound For Join Cardinality Estimation. (Mar. 2022). arXiv: `2201.04166 [cs]`. DOI: `10.48550/arXiv.2201.04166`.

[17] Erik D. Demaine and Quanquan C. Liu. 2018. Red-Blue Pebble Game: Complexity of Computing the Trade-Off between Cache Size and Memory Transfers. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures* (SPAA '18). Association for Computing Machinery, New York, NY, USA, (July 2018), 195–204. ISBN: 978-1-4503-5799-9. DOI: `10.1145/3210377.3210387`.

[18] James Demmel and Grace Dinh. 2018. Communication-optimal convolutional neural nets. *CoRR*, abs/1802.06905. `http://arxiv.org/abs/1802.06905` arXiv: `1802.06905`.

[19] Grace Dinh and James Demmel. 2020-02, 2020. Communication-optimal tilings for projective nested loops with arbitrary bounds. (2020-02, 2020). arXiv: `2003.00119 [cs.DS]`. DOI: `10.48550/arXiv.2003.00119`.

[20] Grace Dinh, Iniyaal Kannan, Hengrui Luo, Charles Hong, Younghyun Cho, James Demmel, Sherry Li, and Yang Liu. 2023. Sample-Efficient Mapspace Optimization for DNN Accelerators with Bayesian Learning. In *Architecture and System Support for Transformer Models (ASSYST @ISCA 2023)*. (June 2023). Retrieved Dec. 15, 2023 from `https://openreview.net/forum?id=el1iPvMqqTY`.

[21] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 92–104. DOI: `10.1145/2749469.2750389`.

[22] Venmugil Elango. 2016. *Techniques for Characterizing the Data Movement Complexity of Computations*. PhD thesis. Ohio State University.

[23] Frans Sijstermans. 2018. The NVIDIA Deep Learning Accelerator. Stanford, CA, (Aug. 2018). Retrieved Dec. 6, 2023 from `https://old.hotchips.org/hc30/2conf/2.08_NVidia_DLA_Nvidia_DLA_HotChips_10Aug18.pdf`.

[24] Tomas Gal and Josef Nedoma. 1972. Multiparametric linear programming. *Management Science*, 18, 7, 406–422. eprint: `https://doi.org/10.1287/mnsc.18.7.406`. DOI: `10.1287/mnsc.18.7.406`.

[25] Ankit Garg, Leonid Gurvits, Rafael Oliveira, and Avi Wigderson. 2018. Algorithmic and optimization aspects of Brascamp-Lieb inequalities, via Operator Scaling. *Geometric and Functional Analysis*, 28, 1, (Feb. 2018), 100–145. DOI: `10.1007/s00039-018-0434-2`.

[26] Hasan Genc et al. 2021. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *Proceedings of the 58th Annual Design Automation Conference (DAC)*.

[27] John R. Gilbert, Thomas Lengauer, and Robert Endre Tarjan. 1979. The pebbling problem is complete in polynomial space. In *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing* (STOC '79). Association for Computing Machinery, New York, NY, USA, (Apr. 1979), 237–248. ISBN: 978-1-4503-7438-5. DOI: `10.1145/800135.804418`.

[28] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly—Performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22, (Dec. 2012). DOI: `10.1142/S0129626412500107`.

[29] Gurobi Optimization, LLC. 2020. Gurobi optimizer reference manual. (2020). `http://www.gurobi.com`.

[30] Bastian Hagedorn, Johannes Lenfers, Thomas Kœhler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020. Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies. *Proceedings of the ACM on Programming Languages*, 4, ICFP, (Aug. 2020), 92:1–92:29. DOI: `10.1145/3408974`.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. (June 2016), 770–778. DOI: `10.1109/CVPR.2016.90`.

[32] Kartik Hegde, Po-An Tsai, Sitao Huang, Vikas Chandra, Angshuman Parashar, and Christopher W. Fletcher. 2021. Mind mappings: enabling efficient algorithm-accelerator mapping space search. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, (Apr. 2021). DOI: `10.1145/3445814.3446762`.

[33] John L. Hennessy and David A. Patterson. 2019. *Computer Architecture: A Quantitative Approach*. (Sixth edition ed.). Morgan Kaufmann Publishers, an imprint of Elsevier, Cambridge, Mass. ISBN: 978-0-12-811905-1.

[34] Jon Herman and Will Usher. 2017. SALib: An open-source python library for sensitivity analysis. *Journal of Open Source Software*, 2, 9, 97. DOI: `10.21105/joss.00097`.

[35] Charles Hong, Qijing Huang, Grace Dinh, Mahesh Subedar, and Yakun Sophia Shao. 2023. DOSA: Differentiable Model-Based One-Loop Search for DNN Accelerators.

[36] Jia-Wei Hong and H. T. Kung. 1981. I/O complexity: The red-blue pebble game. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing* (STOC '81). Association for Computing Machinery, New York, NY, USA, (May 1981), 326–333. ISBN: 978-1-4503-7392-0. DOI: `10.1145/800076.802486`.

[37] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*. arXiv: `1704.04861`.

[38] Qijing Huang, Minwoo Kang, Grace Dinh, Thomas Norell, Aravind Kalaiah, James Demmel, John Wawrzynek, and Yakun Sophia Shao. 2021. CoSA: Scheduling by constrained optimization for spatial accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 554–566.

[39] Dror Irony, Sivan Toledo, and Alexander Tiskin. 2004. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64, 9, 1017–1026. DOI: `10.1016/j.jpdc.2004.03.021`.

[40] William B. Johnson and Joram Lindenstrauss. 1984. Extensions of Lipschitz mappings into a Hilbert space. In *Contemporary Mathematics*. Vol. 26. Richard Beals, Anatole Beck, Alexandra Bellow, and Arshag Hajian, editors. American Mathematical Society, Providence, Rhode Island, 189–206. ISBN: 978-0-8218-5030-5 978-0-8218-7611-4. DOI: 10.1090/conm/026/737400.

[41] C.N. Jones, M. Baric, and M. Morari. 2007. Multiparametric linear programming with applications to control. *European Journal of Control*, 13, 2, 152–170. DOI: 10.3166/ejc.13.152-170.

[42] Norman P. Jouppi et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (ISCA '17). Association for Computing Machinery, New York, NY, USA, (June 2017), 1–12. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080246.

[43] Sheng-Chun Kao and Tushar Krishna. 2020. GAMMA: automating the HW mapping of DNN models on accelerators via genetic algorithm. In *Proceedings of the 39th International Conference on Computer-Aided Design* (ICCAD '20). Association for Computing Machinery, New York, NY, USA, (Dec. 2020), 1–9. ISBN: 978-1-4503-8026-3. DOI: 10.1145/3400302.3415639.

[44] Sheng-Chun Kao, Angshuman Parashar, Po-An Tsai, and Tushar Krishna. 2022. Demystifying map space exploration for NPUs. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, (Nov. 2022). DOI: 10.1109/iiswc55918.2022.00031.

[45] Sheng-Chun Kao, Michael Pellauer, Angshuman Parashar, and Tushar Krishna. 2022. DiGamma: Domain-aware genetic algorithm for HW-Mapping co-optimization for DNN accelerators. In *Proceedings of the 2022 Conference and Exhibition on Design, Automation and Test in Europe* (DATE '22). European Design and Automation Association, Leuven, BEL, 232–237. ISBN: 978-3-9819263-6-1.

[46] S. Karandikar et al. 2018. FireSim: FPGA-Accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 29–42. DOI: 10.1109/ISCA.2018.00014.

[47] Rickard Karlsson, Laurens Bliek, Sicco Verwer, and Mathijs de Weerdt. 2021. Continuous surrogate-based optimization algorithms are well-suited for expensive discrete problems. In *Communications in Computer and Information Science*. Springer International Publishing, 48–63. DOI: 10.1007/978-3-030-76640-5_4.

[48] Sam Kaufman, Phitchaya Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. 2021. A Learned Performance Model for Tensor Processing Units. *Proceedings of Machine Learning and Systems*, 3, (Mar. 2021), 387–400. Retrieved Dec. 6, 2023 from https://proceedings.mlsys.org/paper_files/paper/2021/hash/6bcfac823d40046dca25ef6d6d59cc3f-Abstract.html.

[49] Mahmoud Abo Khamis, Vasileios Nakos, Dan Olteanu, and Dan Suciu. 2023. Join Size Bounds using Lp-Norms on Degree Sequences. (June 2023). arXiv: 2306.14075 [cs, math]. DOI: 10.48550/arXiv.2306.14075.

[50] Sehoon Kim et al. 2023. Full stack optimization of transformer inference: a survey, (Feb. 2023). https://arxiv.org/pdf/2302.14017.pdf eprint: 2302.14017.

[51] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. 2013. When polyhedral transformations meet SIMD code generation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '13). Association for Computing Machinery, New York, NY, USA, (June 2013), 127–138. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2462187.

[52] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. 2019. Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, Denver Colorado, (Nov. 2019), 1–22. ISBN: 978-1-4503-6229-0. DOI: 10.1145/3295500.3356181.

[53] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. 2020. MAESTRO: A Data-Centric Approach to Understand Reuse, Performance, and Hardware Cost of DNN Mappings. *IEEE Micro*, 40, 3, (May 2020), 20–29. DOI: 10.1109/MM.2020.2985963.

[54] Kasper Green Larsen and Jelani Nelson. 2017. Optimality of the Johnson-Lindenstrauss Lemma. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*. (Oct. 2017), 633–638. DOI: 10.1109/FOCS.2017.64.

[55] Yujun Lin, Mengtian Yang, and Song Han. 2021. NAAS: Neural accelerator architecture search. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE Press, San Francisco, CA, USA, 1051–1056. DOI: 10.1109/DAC18074.2021.9586250.

[56] L. H. Loomis and H. Whitney. 1949. An inequality related to the isoperimetric inequality. *Bulletin of the American Mathematical Society*, 55, 10, 961–962. DOI: 10.1090/S0002-9904-1949-09320-5.

[57] Liqiang Lu, Naiqing Guan, Yuyue Wang, Liancheng Jia, Zizhang Luo, Jieming Yin, Jason Cong, and Yun Liang. 2021. TENET: a framework for modeling tensor dataflow based on relation-centric notation. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (ISCA '21). IEEE Press, Virtual Event, Spain, (Nov. 2021), 720–733. ISBN: 978-1-4503-9086-6. DOI: 10.1109/ISCA52012.2021.00062.

[58] Hengrui Luo, Younghyun Cho, James W. Demmel, Xiaoye S. Li, and Yang Liu. 2022. Hybrid models for mixed variables in bayesian optimization. *arXiv:2206.01409*, 1–56. arXiv: 2206.01409.

[59] Hengrui Luo, James W. Demmel, Younghyun Cho, Xiaoye S. Li, and Yang Liu. 2021. Non-Smooth Bayesian Optimization in Tuning Problems. Tech. rep. arXiv, (Sept. 2021). DOI: 10.48550/arXiv.2109.07563.

[60] Jurij V. Matijasevič. 1996. *Hilbert's Tenth Problem.* (3. print ed.). *Foundations of Computing.* MIT Press, Cambridge, Mass. ISBN: 978-0-262-13295-4.

[61] Atefeh Mehrabi, Aninda Manocha, Benjamin C. Lee, and Daniel J. Sorin. 2020. Bayesian optimization for efficient accelerator synthesis. *ACM Transactions on Architecture and Code Optimization*, 18, 1, (Dec. 2020), 1–25. DOI: 10.1145/3427377.

[62] Linyan Mei, Pouya Houshmand, Vikram Jain, Sebastian Giraldo, and Marian Verhelst. 2021. ZigZag: Enlarging Joint Architecture-Mapping Design Space Exploration for DNN Accelerators. *IEEE Transactions on Computers*, 70, 8, (Aug. 2021), 1160–1174. DOI: 10.1109/TC.2021.3059962.

[63] Lingchuan Meng and John Brothers. 2019. Efficient winograd convolution via integer arithmetic. (2019). arXiv: 1901.01965 [cs.NE].

[64] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.*, 35, 4, (July 2016). DOI: 10.1145/2897824.2925952.

[65] Riley Murray, Venkat Chandrasekaran, and Adam Wierman. 2021. Signomial and polynomial optimization via relative entropy and partial dualization. *Mathematical Programming Computation*, 13, 2, (June 2021), 257–295. DOI: 10.1007/s12532-020-00193-4.

[66] Riley Murray et al. 2023. Randomized Numerical Linear Algebra : A Perspective on the Field With an Eye to Software. (Apr. 2023). Retrieved Oct. 12, 2023 from `http://arxiv.org/abs/2302.11474` arXiv: `2302.11474 [cs, math]`.

[67] Tony Nowatzki, Newsha Ardalani, Karthikeyan Sankaralingam, and Jian Weng. 2018. Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques* (PACT '18). Association for Computing Machinery, New York, NY, USA, (Nov. 2018), 1–15. ISBN: 978-1-4503-5986-3. DOI: `10.1145/3243176.3243212`.

[68] Auguste Olivry, Guillaume Iooss, Nicolas Tollenaere, Atanas Rountev, P. Sadayappan, and Fabrice Rastello. 2021. IOOpt: automatic derivation of I/O complexity bounds for affine programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, (June 2021). DOI: `10.1145/3453483.3454103`.

[69] Auguste Olivry, Julien Langou, Louis-Noël Pouchet, P. Sadayappan, and Fabrice Rastello. 2020. Automated derivation of parametric data movement lower bounds for affine programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '20). Association for Computing Machinery, New York, NY, USA, 808–822. ISBN: 978-1-4503-7613-6. DOI: `10.1145/3385412.3385989`.

[70] Pál András Papp and Roger Wattenhofer. 2020. On the Hardness of Red-Blue Pebble Games. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, Virtual Event USA, (July 2020), 419–429. ISBN: 978-1-4503-6935-0. DOI: `10.1145/3350755.3400278`.

[71] Angshuman Parashar et al. 2019. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 304–315.

[72] Suchita Pati, Shaizeen Aga, Nuwan Jayasena, and Matthew D. Sinclair. 2022. Demystifying BERT: System Design Implications. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. (Nov. 2022), 296–309. DOI: `10.1109/IISWC55918.2022.00033`.

[73] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48, 6, (June 2013), 519–530. DOI: `10.1145/2499370.2462176`.

[74] Brandon Reagen, Jose Miguel Hernandez-Lobato, Robert Adolf, Michael Gelbart, Paul Whatmough, Gu-Yeon Wei, and David Brooks. 2017. A case for efficient accelerator design space exploration via Bayesian optimization. In *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 1–6. DOI: `10.1109/ISLPED.2017.8009208`.

[75] Binxin Ru, Ahsan Alvi, Vu Nguyen, Michael A Osborne, and Stephen Roberts. 2020. Bayesian optimisation over multiple continuous and categorical inputs. In *International Conference on Machine Learning*. PMLR, 8276–8285.

[76] Chirag Sakhuja, Zhan Shi, and Calvin Lin. 2023. Leveraging domain information for the efficient automated design of deep learning accelerators. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 287–301. DOI: `10.1109/HPCA56546.2023.10071095`.

[77] Ananda Samajdar, Jan Moritz Joseph, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. 2020. A Systematic Methodology for Characterizing Scalability of DNN Accelerators using SCALE-Sim. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. (Aug. 2020), 58–68. DOI: `10.1109/ISPASS48437.2020.00016`.

[78] Pablo San Juan, Adrián Castelló, Manuel F. Dolz, Pedro Alonso-Jordá, and Enrique S. Quintana-Ortí. 2020. High performance and portable convolution operators for multicore processors. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 91–98.

[79] Ravi Sethi. 1975. Complete Register Allocation Problems. *SIAM Journal on Computing*, 4, 3, (Sept. 1975), 226–248. DOI: 10.1137/0204020.

[80] Zhan Shi, Chirag Sakhuja, Milad Hashemi, Kevin Swersky, and Calvin Lin. 2020. Using bayesian optimization for hardware/software co-design of neural accelerators. In *Workshop on ML for Systems at the Conference on Neural Information Processing Systems (NeurIPS)*.

[81] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. 2012. Practical Bayesian optimization of machine learning algorithms. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2* (NIPS'12). Curran Associates Inc., Red Hook, NY, USA, (Dec. 2012), 2951–2959. Retrieved Dec. 5, 2023 from.

[82] Ilya M Sobol. 2001. Global sensitivity indices for nonlinear mathematical models and their Monte Carlo estimates. *Mathematics and computers in simulation*, 271–280.

[83] Jørgen Spjøtvold, Petter Tøndel, and Tor A. Johansen. 2005. A method for obtaining continuous solutions to multiparametric linear programs. *IFAC Proceedings Volumes*, 38, 1, 253–258. DOI: 10.3182/20050703-6-CZ-1902.00903.

[84] Alexandre Tiskin. 1996. The bulk-synchronous parallel random access machine. In *Euro-Par'96 Parallel Processing* (Lecture Notes in Computer Science). Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors. Springer, Berlin, Heidelberg, 327–338. ISBN: 978-3-540-70636-6. DOI: 10.1007/BFb0024720.

[85] S. Valdimarsson. 2010. The Brascamp-Lieb polyhedron. *Canadian Journal of Mathematics*, 62, 4, 870–888.

[86] Rangharajan Venkatesan et al. 2019. MAGNet: A Modular Accelerator Generator for Neural Networks. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, Westminster, CO, USA, (Nov. 2019), 1–8. ISBN: 978-1-72812-350-9. DOI: 10.1109/ICCAD45719.2019.8942127.

[87] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52, 4, (Apr. 2009), 65–76. DOI: 10.1145/1498765.1498785.

[88] Sam Likun Xi, Hans Jacobson, Pradip Bose, Gu-Yeon Wei, and David Brooks. 2015. Quantifying sources of error in McPAT and potential impacts on architectural studies. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 577–589. DOI: 10.1109/HPCA.2015.7056064.

[89] Qingcheng Xiao, Size Zheng, Bingzhe Wu, Pengcheng Xu, Xuehai Qian, and Yun Liang. 2021. HASCO: Towards agile HArdware and software CO-design for tensor computation. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, (June 2021). DOI: 10.1109/isca52012.2021.00086.

[90] Xuan Yang et al. 2020. Interstellar: Using halide's scheduling language to analyze dnn accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 369–383.

[91] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Jonathan Ragan-Kelley, and Hasan Genc. 2022. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '22). New York, NY, USA. DOI: 10.1145/3519939.3523446.

[92] Yaqi Zhang, Nathan Zhang, Tian Zhao, Matt Vilim, Muhammad Shahbaz, and Kunle Olukotun. 2021. SARA: Scaling a reconfigurable dataflow accelerator. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (ISCA '21). IEEE Press, Virtual Event, Spain, 1041–1054. ISBN: 978-1-4503-9086-6. DOI: 10.1109/ISCA52012.2021.00085.

[93] Lianmin Zheng et al. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. Tech. rep. arXiv, (Nov. 2020). DOI: 10.48550/arXiv.2006.06762.

[94] Aleksandar Zlateski, Zhen Jia, Kai Li, and Fredo Durand. 2019. The anatomy of efficient FFT and winograd convolutions on modern CPUs. In (ICS '19). Association for Computing Machinery, New York, NY, USA, 414–424. ISBN: 978-1-4503-6079-1.