THE UNIVERSITY OF CHICAGO

AN END-TO-END PROGRAMMING MODEL FOR AI ENGINE ARCHITECTURES

A DISSERTATION SUBMITTED TO

THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES

IN CANDIDACY FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY

MAKSIM LEVENTAL

CHICAGO, ILLINOIS

JUNE 2024

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The proliferation of deep learning in various domains has led to remarkable advancements in artificial intelligence applications, such as large-language models for scientific use cases. However, the concomitant exponential growth in computational demands, driven by the development of ever-larger deep learning models, presents significant challenges in terms of resource consumption and sustainability. This dissertation addresses these escalating computational costs by investigating how the complexity of deep learning frameworks' and their abstractions can significantly impact resource usage. In order to mitigate these growing costs, this dissertation presents novel insights into memory planning, high-level synthesis, lightweight frontend development, and end-to-end programming models for accelerator architectures. These insights culminate in the design and implementation of an embedded domain-specific language (eDSL) tailored to deep learning development on a novel accelerator, specifically the AMD AI Engine architecture. By prioritizing access to low-level APIs, leveraging compiler technologies, and rigorous mathematical models, the eDSL demonstrates the feasibility of achieving performant deep learning implementations while maintaining productivity in the design and exploration of deep learning methods.

CHAPTER 1

INTRODUCTION

The field of deep learning has seen an unparalleled expansion over the last decade, driving significant advancements in artificial intelligence applications ranging from natural language processing (Otter et al., 2020) and image recognition (Li, 2022) to autonomous systems (Muhammad et al., 2020) and medical diagnostics (Bakator and Radosav, 2018). This rapid progress, however, has been accompanied by an exponential increase in the computational resources required by state-of-the-art deep learning models (Sevilla et al., 2022). The development of models with billions (and trillions) of parameters, capable of understanding and generating human-like text or accurately identifying objects in high-resolution images, has led to requirements for vast computational resources such as memory, power, and network bandwidth. Such requirements pose significant challenges, particularly in terms of accessibility and energy consumption associated with training and deploying deep learning models (Strubell et al., 2019). Thus, as the demand for more powerful deep learning methods, techniques, and models continues to grow, so grows the imperative need to find solutions that can mitigate the escalating computational costs.

This thesis addresses the aforementioned computational costs by proposing a domain-specific language that provides access to a wide range of abstractions and APIs and allows developers to tailor their approach to the specific needs of their deep learning model, striking a balance between ease of development and the efficiency of the final implementation. Specifically, we develop an "end-to-end" programming model for AMD's AI Engine architectures. AI Engine architectures are Coarse-Grained Reconfigurable Architectures (CGRAs) and offer unparalleled flexibility by allowing developers to selectively activate or deactivate components, resulting in superior performance per Watt across a broad range of applications compared to traditional processors like multicore CPUs and GPUs (Podobas et al., 2020). The programming models and languages for coarse-grained devices significantly reduce the

barrier to use. However, they do not eliminate the requirement that software developers be familiar with and manipulate various hardware layers. Thus, while many domain-specific languages exist for deep learning and CGRAs, we develop our language to span a wide range of abstraction levels (from objects to registers) in a single unified environment. Our domain-specific language, embedded in Python, is open source and closely integrated with the MLIR compiler framework (Lattner et al., 2020b) (a staged, modular compiler (Rompf and Odersky, 2010)), with portions of it having been upstreamed to the LLVM project (Lattner and Adve, 2004).

Our thesis is derived from experience over the course of several projects, each of which involved manipulating highly abstract APIs and struggling to achieve satisfactory runtime performance (see Section 1.2). Throughout these works we faced challenges primarily due to a need for access to lower-level APIs due to so-called abstractions. These failures of abstraction color our perspective on language design and implementation: the ideal programming language does not abstract anything by default. However, it does enable building *transparent abstractions* at every level (of abstraction). Transparent abstractions are accessible, interpretable, and, most importantly, *ductile*. A ductile abstraction can easily be adjusted and adapted to suit a fixed program with a narrow set of inputs rather than anticipating the requirements of some generic, nebulous space of programs and inputs. In trading generality for ductility, our operating hypothesis is that no lowest common denominator functionality (i.e., maximal abstraction) can be optimal across the range of possible use cases. Since most users solve a narrowly scoped set of problems, a language should empower them to optimize for their particular use cases. The costs incurred by the abstractions are naturally brought to the fore in such a language paradigm because no tradeoff (e.g., between space and time) made is outside user's purview (of the language).

1.1 Thesis Statement

The core thesis of this dissertation is that it is possible to design a language and programming model for expert ML developers to represent deep learning models simultaneously at multiple levels of abstraction, generate performant, target-specific, implementations and execute those models on novel accelerator architectures. We claim that such an approach to domain-specific language and programming model design, as it pertains to deep learning models, yields performant implementations without sacrificing the productivity of higher-level frameworks. Thus, if we show that it is possible, using our language and its programming model, to implement deep learning models and optimize those models for application-specific accelerators, we validate the stated thesis.

1.2 Contributions

The primary contribution of this thesis is the design and implementation of an embedded domain-specific language (eDSL) for representing, optimizing, and executing deep learning models on a novel accelerator architecture. That eDSL is available at `https://github.com/Xilinx/mlir-aie` and targets AMD AI Engine architectures. In addition, core, reusable, infrastructure components of the eDSL have been accepted as contributions to MLIR itself. The language design principles propounded by this thesis, and which culminate in the primary contribution, are the result of a sequence of projects, each of which constitutes a contribution to deep learning optimization research and yields deep insight into the necessary language design techniques:

- In *Memory Planning for Deep Neural Networks*, we studied memory allocation patterns in DNNs during inference in large-scale systems. In the context of multi-threading, we observed that such memory allocation patterns are subject to high latencies due to mutex contention in the system memory allocator. We presented an implementation

of a "memoizing" allocator, `MemoMalloc`, within the PyTorch deep learning framework and evaluated its memory consumption and execution performance on a wide range of DNN architectures.

  - **Key technical contribution of this work**: A novel allocation capture technique which uniquely associates all memory allocations with their high-level deep learning operation; a static analysis component which constructs an optimal allocation "plan" using ILP and CP-SAT formulations of the Dynamic Storage Allocation problem (Buchsbaum et al., 2003).
  - **Key language design insight**: High-level frameworks such as PyTorch, by virtue of their abstractions, cannot capture and expose the implementation-specific details necessary for specific crucial optimizations (even when those frameworks expose lower-level intermediate representations).

- In *`BraggHLS`: High-Level Synthesis for Low-latency Deep Neural Networks for Experimental Science*, we presented an open-source compiler framework, `BraggHLS`, based on high-level synthesis techniques for translating high-level representations of deep neural networks into lower-level representations, suitable for deployment to field-programmable gate arrays (FPGAs).

  - **Key technical contribution of this work**: A CP-SAT formulation of the Shared Operator Scheduling (Kruppe et al., 2021); a lifting method for raising MLIR to Python to enable domain-specific and user-extensible transformations on low-level representations of DNNs; finally, an implementation of the BraggNN (Liu et al., 2022b) network (for Bragg peak detection) which achieved a throughput 4.8μ s/sample, i.e., a 4$\times$ improvement over the previous state of the art.
  - **Key language design insight**: Expensive compiler optimization passes (such as loop-carried dependency analysis) must be exposed and manipulable by the user

so that they may avail themselves of domain-specific knowledge when optimizing their implementations.

- In ***`nelli`: *A Lightweight Frontend for MLIR***, we developed `nelli`, a lightweight, Python-embedded, domain-specific language for generating MLIR code. `nelli` enables generating a fully functional embedded (in Python) domain-specific language frontend for arbitrary dialects of MLIR.
  - **Key technical contribution of this work**: Automatic, extremely high-fidelity and generic Python bindings to MLIR dialects, including a novel approach to mapping arbitrarily nested conditionals and loop primitives to corresponding IR operations.
  - **Key language design insight**: Compilers should not assume the existence of high-level language frontends (such as PyTorch or TensorFlow) and instead expose APIs and IRs such that the operations and transformations may be employed in and of themselves.
- In *An End-to-End Programming Model for AI Engine Architectures*, we validate our core thesis by leveraging and extending MLIR to provide an embedded domain-specific language and code generation for a Coarse-Grained Reconfigurable Architecture, specifically, AMD AI Engines. We apply our end-to-end programming model to the challenge of designing a performant GEMM implementation.
  - **Key technical contribution of this work**: A fully end-to-end programming model for AMD AI Engines, including a language frontend, optimal stream routing (using ILP and CP-SAT formulations of congestion-aware traffic assignment (Temelcan et al., 2020)), runtime memory management, and packaging, distribution, deployment to device; a novel *stream broadcasting* primitive for reducing the semantic gap between array broadcasting and stream switch configuration; a novel

approach to metaprogramming MLIR in Python that enables using the same language for both metaprogramming and programming; finally, performant implementations of GEMM for the same architecture.

– **Key language design insight**: Integrated language and compiler design enables building a programming model that developers can use to access all features and device APIs necessary for achieving performant implementations of dataflow programs.

1.3 Thesis Organization

The remainder of this dissertation reviews the aforementioned works; Chapter 2 discusses *Memory Planning for Deep Neural Networks*, Chapter 3 discusses *`BraggHLS`: High-Level Synthesis for Low-latency Deep Neural Networks for Experimental Science*, Chapter 4 discusses *`nelli`: A Lightweight Frontend for MLIR*, Chapter 5 discusses *An End-to-End Programming Model for AI Engine Architectures*, and finally Chapter 6 concludes with a summary.

CHAPTER 2

MEMORY PLANNING FOR DEEP NEURAL NETWORKS

Deep neural networks (DNNs) are ubiquitous as components of research and production systems; they are employed to fulfill tasks across a broad range of domains, including image classification (Affonso et al., 2017), object detection (Zhao et al., 2019), speech recognition (Amodei et al., 2016), and content recommendation (Da'u and Salim, 2020). Traditionally, DNNs are deployed to multi-processor (or multi-core processor) server-class platforms, such as those found in commercial data centers and scientific high-performance clusters. This is because DNNs, generally, are resource-intensive, in terms of compute, memory, and network usage; see Table 2.1 for representative DNN workloads at Facebook, Inc., a large social media services company that employs DNNs in many of its products.

Indeed, as a result of latency constraints imposed by quality-of-service guarantees, data center deployments usually target CPU architectures (and corresponding memory hierarchies), as opposed to GPGPU architectures (Park et al., 2018). This is a consequence of the fact that CPUs are better suited for low latency applications, owing to their high clock speeds and synchronous execution model, as opposed to GPUs, which typically have lower clock speeds and an asynchronous execution model. Further, new DNN techniques, such as

Category	Model Type	Model Size (# params)	Typical Batch Size	Max # Live Activations	Latency (constraint)
Ranking	Linear	1 - 10M	1 - 100	>10K	$\sim$ 10 ms
	Embedding	>10 billion	1 - 100	>10K	$\sim$ 10 ms
Vision	ResNet50	25M	1 (image)	2M	N/A
	ResNeXt-101-32x4	43 - 829M	1 (image)	2.4 - 29M	N/A
	FasterRCNN	6M	1 (image)	13.2M	N/A
	ResNeXt3D-101	21M	1 (movie clip)	58M	N/A
Language	Seq2seq	100M - 1B	1 - 8 tokens	>100K	$\sim$ 10 ms

Table 2.1: Resource requirements of representative DNN inference workloads implemented on CPU. Reprinted with permission from (Park et al., 2018).

Transformers (Brown et al., 2020) and Mixture-of-Experts (Shazeer et al., 2017), lead to networks with billions, or even trillions (Fedus et al., 2021), of floating-point parameters (called *weights*), thus indicating (current) upper bounds on potential memory consumption; for instance, training BERT networks (a transformer) requires up to 16TB of memory (Shoeybi et al., 2020). Applying such complex DNNs effectively in high traffic services necessitates managing system resources carefully. To be specific, managing memory usage is important, both for preventing failures (such as out-of-memory conditions), and, as we discuss in the following, reducing latencies.

In this work, we focus on the implications of memory management for execution performance in server-class deployments of DNNs. It is well-known that in multi-threaded environments, with many non-uniform service requests, heap synchronization routines can lead to blocking that inhibits scaling performance gains (Boreham, 2000). Specifically, we refer to contention on locks (i.e., `mutex`es) held to enforce mutual exclusion on code that modifies the heap data structure (i.e., `malloc` and `free`). The standard mitigation of such issues is replacing system `malloc` with a caching allocator such as `jemalloc` (Evans, 2011), `tcmalloc` (Ghemawat and Menage, 2009), or SuperMalloc (Kuszmaul, 2015a). Caching allocators such as these alleviate lock contention by maintaining many independent heaps, each with its own `mutex`es, and distributing memory requests among them, thereby reducing pressure on any single lock. These allocators can be effective for many workloads and memory allocation patterns, but they are not a panacea. In the case of diverse DNN workloads on servers, where a process may exhibit 2×10^7 `malloc` requests per second, distributed across 2,000 concurrent threads (Hazelwood et al., 2018), it is still possible for a program to experience significantly reduced performance due to lock contention. For DNNs with many allocation requests, spanning a wide range of sizes, this can readily be observed (see Section 2.1.2).

It is important to note that DNNs allocate memory in addition to that needed for just

their weights; substantial temporary memory is associated with buffers (known as *tensors*) that correspond to intermediate results created during the evaluation of *layers* of the DNN. We observe that even with reasonable input sizes, the intermediate tensors of `resnext101_32x8d` (Xie et al., 2017) comprise 27% of the total 13GB run-time memory, 57% (of 760MB) for `squeezenet1_0` (Iandola et al., 2016a), and 66% (of 2473MB) for `mnasnet0_75` (Tan et al., 2019). Similar figures have been reported in prior work (Pisarchyk and Lee, 2020). These intermediates are often short-lived (serving only to propagate results between sequential *operations*) and overlap with only a small subset of the lifetimes of other intermediates. Thus, the effective memory needed to materialize the entire collection of intermediates is often much less than the sum total of the individual memories. Given foreknowledge of all lifetimes and sizes of intermediate tensors, and a *strategy* for computing corresponding offsets, memory can be allocated statically (or, at worst, just prior to inference). More importantly, as it pertains to performance, this single batch allocation effectively eliminates lock contention. Such an approach is called *static memory planning*, or *static allocation*. Unfortunately, due to pointer aliasing and control flow, comprehensive and robust lifetime and size data are difficult to derive statically (i.e., correctly, completely, and prior to any execution).

Hence, to reduce allocations while satisfying peak memory usage constraints, we propose a hybrid static-runtime memory management solution, called `MemoMalloc`, that makes use of both the statically known structure of the neural network and a single profiling pass. Specifically, our method uses a convenient representation of the neural network, along with lightweight stack tracing and pointer tagging, to reconstruct the lifetimes, sizes, and aliasing relationships of all intermediate tensors completely and accurately. Our system then constructs memory plans using one of several performant strategies. We present an implementation of the technique in the PyTorch (Paszke et al., 2019) deep learning framework and evaluate our implementation on a large and representative set of DNNs. In terms of ex-

ecution performance (as measured by latency) our solution outperforms PyTorch+`jemalloc` (i.e., PyTorch backed by the state-of-the-art caching allocator `jemalloc`). Specifically, across almost all input sizes and threading configurations (in terms of the number of threads) we observe, on average 20% lower inference latencies, and at best 40% lower latencies.

In summary, the principal contributions of this work are:

1. A study of the memory allocation patterns of a wide range of DNN architectures.
2. A study of several different exact and heuristic static allocation strategies, as they pertain to DNNs.
3. An implementation and evaluation of `MemoMalloc`, a system for managing memory for DNNs, which outperforms `jemalloc`.

The remainder of this chapter is organized as follows: Section 2.1 gives necessary background on representations of DNNs and memory allocators, along with a discussion of worst-case results concerning caching allocators and DNNs. Section 2.2 discusses our implementation, with a particular focus on how we resolve aliases exactly and performantly. Section 2.3 presents a thorough evaluation of our implementation, across various representative DNN architectures and workloads (in terms of input sizes and threading environment). Section 2.4 discusses the evaluation and the insights garnered thereof. Finally, Section 2.5 reviews prior work in this area and Section 2.6 concludes and discusses future work, including dynamics, training, GPUs, and applications to edge device deployments.

2.1 Background

We review the necessary background for our work. This includes a discussion of how DNNs are represented in deep learning frameworks (i.e., PyTorch) as it pertains to our manipulation of those representations. We then discuss the memory allocation issues addressed by caching allocators (including an empirical study of worst-case performance). Finally, we

define memory planning formally and introduce the memory planning strategies that inform the design of the static memory planning component of `MemoMalloc`.

2.1.1 Representations of DNNs

Deep neural networks are typically specified using high-level frameworks that can be compiled into low-level platform and hardware-specific code. For example, TVM (Chen et al., 2018) generates highly optimized, hardware-specific code for various hardware backends by efficiently exploring the space of possible DNN transformations (specifically, with respect to kernel fusion). Such transformations are carried out on a representation of the DNN (Relay (Roesch et al., 2018) of TVM, HLO of TensorFlow (Larsen and Shpeisman, 2019), TorchScript (tsi) of PyTorch) that captures the data and control flow dependencies between individual layers, as well as attributes of the data (i.e., tensors), such as type (e.g., `float32`, `int`, or `bfloat16`), memory layout (e.g., *contiguous*, *strided*, or *sparse*), and *shape*. Note that inputs to DNNs are characterized by their shape, i.e., the sizes of the dimensions of the input tensors, represented as arrays; a common shape corresponding to an image input for computer vision networks is (N, C, H, W), with corresponding size $N \times C \times H \times W \times size(\texttt{dtype})$, where $size(\texttt{dtype})$ is the width of the data type (e.g., 4 bytes for `float32`). This representation is called an *intermediate representation* (IR) since it functions as an intermediary between the high-level specification and the lower-level hardware characteristics.

TorchScript (TS) is a compiler infrastructure within the PyTorch deep learning framework that produces a type-annotated, static single assignment (SSA) IR (called TS IR). TorchScript is executed using an interpreter attached to a Just in Time (JIT) optimizer and compiler. There are two ways to generate TS IR from a PyTorch specified DNN:

- `torch.jit.trace`, which executes a forward pass iteration of a DNN and records the PyTorch *operators* (corresponding to the conceptual layers that comprise the DNN) that are invoked, thus "freezing" the code path of the DNN and hence eliminating

Listing 1 Example neural network

```
class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.linear = nn.Linear(4, 4)
        self.relu = nn.ReLU()

    def forward(self, x, h):
        y = self.linear(x) + h
        y = self.relu(y)
        return y
```

control flow;

- `torch.jit.script`, which analyzes the Python abstract syntax tree representation of the DNN and *lowers* it to TS IR.

In this work we exclusively make use of the `torch.jit.trace` path. Consider the example neural network, specified as a PyTorch model, presented in Listing 1. Given an input tensor with shape $(3, 4)$, it is "traced" to the TS IR presented in Listing 2.

Within TS IR, identifiers on the left-hand sides of assignments are called *values*, and identifiers on the right-hand sides are the operators invoked during execution. As prescribed by SSA semantics, each value is assigned only once, and thus the TS IR representation permits a one-to-one mapping with a directed, acyclic, control and *data flow* graph (hence, the pairing of operator and output are considered a *node* in this graph). Note, as well, that all values have type annotations of varying levels of specificity; for example (cf. Listing 2), the *concrete* annotation `Float(3, 4, strides=[4, 1])` uniquely determines the size of the intermediate tensor `%11` as $3 \times 4 \times size(\texttt{Float}) = 48$ bytes (`strides=[4, 1]` indicates the tensor is arranged contiguously in memory) while the *abstract* annotation `Tensor` indicates value `%12`'s type cannot be determined until runtime. The TS compiler has facilities for traversing and transforming these representations of DNNs. In particular one can implement

Listing 2 TS IR representation of neural network in Listing 1

```
graph(%x : Tensor, %h : Tensor):
  %6: int = prim::Constant[value=1]()
  %linear_weight: Float(4, 4, strides=[4, 1])
    = prim::Constant[value=<Tensor>]()
  %linear_bias: Float(4, strides=[1]) = prim::Constant[value=<Tensor>]()
  %11: Float(3, 4, strides=[4, 1]) = aten::linear(
    %x, %linear_weight, linear_bias
  )
  %12: Tensor = aten::add(%11, %h, %6)
  %13: Tensor = aten::relu(%12)
  return (%13)
```

graph rewrite passes that arbitrarily insert, remove, and rearrange nodes. We make use of these facilities in our implementation to augment the IR with memory allocation nodes that are then executed by the TS JIT and effectuate the memory plan (see Section 2.2).

2.1.2 Caching Allocators and Lock Contention

Caching allocators (Bonwick, 1994) address performance issues with memory allocation and de-allocation, at runtime. Specifically total memory usage (i.e., reduction of internal and external fragmentation of allocated memory), cache locality of sequences of allocations, and overall latency in allocating memory for complex objects. They accomplish their goals by caching recent allocations (typically for configurable lengths of time called *decay times*) in order to reduce the number of expensive system calls (`sbrk` and `mmap`). An implicit concern of allocators is the performance overhead of the use of the allocator itself. An allocator that allocates optimally (either in terms of cache locality or total usage) but does so at the cost of excessive blocking times per allocation is of questionable value for typical users.

In the context of multi-threaded applications running on multiprocessor systems, blocking occurs during synchronization to prevent race conditions on the cache data structures. Caching allocators balance these costs (against those associated with fragmentation) by de-

ploying multiple, independently managed caches (called *arenas*) and distributing allocation requests among them (thereby reducing request service and synchronization pressure on any one cache). In principle, this solution is in direct contradiction with the stated aim of reducing fragmentation: many caches managed by a single caching allocator degenerate to the same fragmentation pattern as many independent non-caching allocators managing their own subsets of system memory. Thus, care must be taken with respect to large allocations (typical of DNNs) to prevent severe fragmentation (i.e., mixing of small and large allocations in the same regions of memory).

"Per-thread" caching allocators, such as `jemalloc`, `tcmalloc`, and SuperMalloc, support thread-specific caching, in addition to maintaining multiple caches (called, appropriately, *thread caches*). That is to say, they maintain unique caches for each live thread executing on a system. This enables those allocations that can be serviced by the thread cache to happen without any synchronization and therefore very efficiently. This leads to very fast allocation in the common case, but also increases memory usage and fragmentation since a fixed number of objects can persist in each thread cache over the course of the entire execution of the program (Kuszmaul, 2015b). Effectively, this is the same failure mode (writ small) as that which betides conventional caching allocators operating many caches. To account for such fragmentation, thread caches are usually configured to be quite small; the default thread cache for `jemalloc` is 32KB in size. In addition, as in the case of DNN workloads, it is common to instantiate a manually managed arena for "oversized" allocations that has no thread cache at all; typical allocation size thresholds for this oversized arena are 1MB, 2MB, or 4MB.

To further illustrate the challenge posed by memory allocation patterns in the context of DNN workloads, with respect to latency, we perform a worst-case analysis; we exercise some common networks with `jemalloc` as the allocator with no thread cache and a single arena for all allocations. To be precise, we execute ten iterations of a forward pass on inputs sized

$(1, 3, 128, 128) \approx 192$KB and record (using `perf`) time spent in `malloc_mutex_lock_slow` (a `jemalloc` utility function related to locking). See Figure 2.1. The result is that even at moderate concurrency (16 threads on our 32-core test platform; see Section 2.3) most iterations spend considerable time contending with locks. We can further investigate lock contention by collecting statistics on blocking wait times for lock acquisition (as recorded by `mutexes.ctl.total_wait_time` and `mutexes.ctl.max_wait_time`[1]). The results, shown in Figure 2.2, can be understood given consideration of the sizes and frequencies of the intermediate allocations made by these DNNs. We observe that the DNNs most affected make many allocations, most below 1MB (see Figure 2.3), and incur high request rates on `jemalloc` and locks related to those allocation sizes, evident from statistics on individual arena bins (`jemalloc` partitions arenas into bins of size 2^k, and distributes allocations requests amongst those bins). We make use of this data to tune `jemalloc` during our evaluation (see Section 2.3).

2.1.3 Memory Planning

In general, memory planning can be framed as an instance of the offline dynamic storage allocation (DSA) problem. To be precise, given static knowledge of all intermediate tensor sizes and lifetimes, we seek to determine the initial allocation size and the set of suitable offsets such that all intermediate tensors fit within the allocation. Therefore, the offsets can be computed by solving the mixed-integer program (MIP) formulation of offline DSA (Sekiyama et al., 2018):

$$\begin{aligned} &\min \texttt{total\_mem} \\ &\text{s.t. } \texttt{offset}_i + \texttt{mem}_i \leq \texttt{total\_mem} \end{aligned} \tag{2.1}$$

1. http://jemalloc.net/jemalloc.3.html#tuning

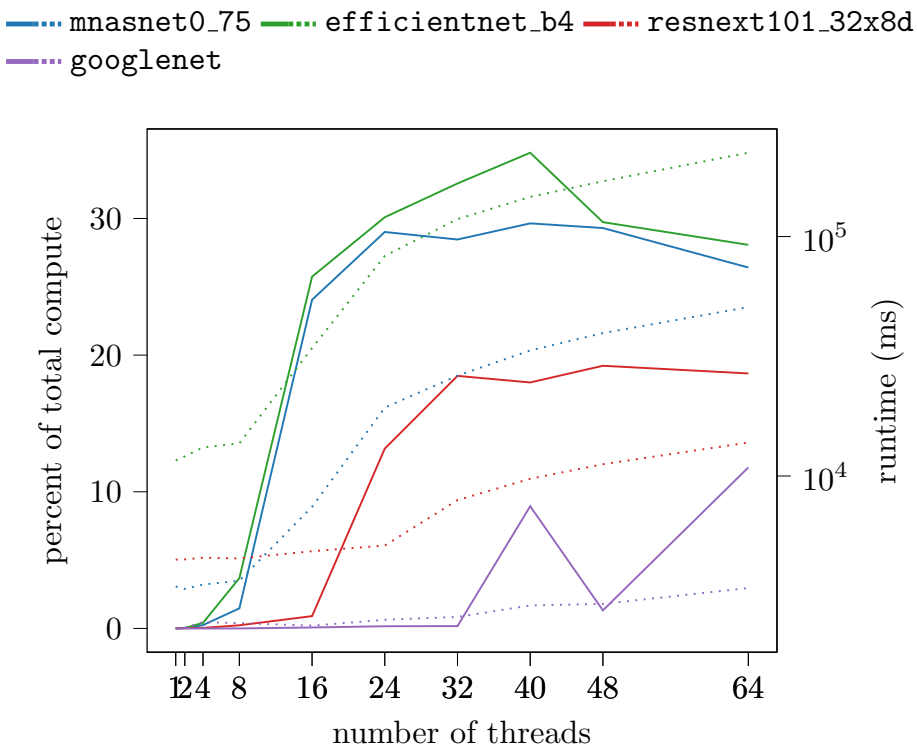


Figure 2.1: Total runtime (dashed line) and percent (solid line) of compute time spent in `malloc_mutex_lock_slow` as a function of the number of concurrent threads.

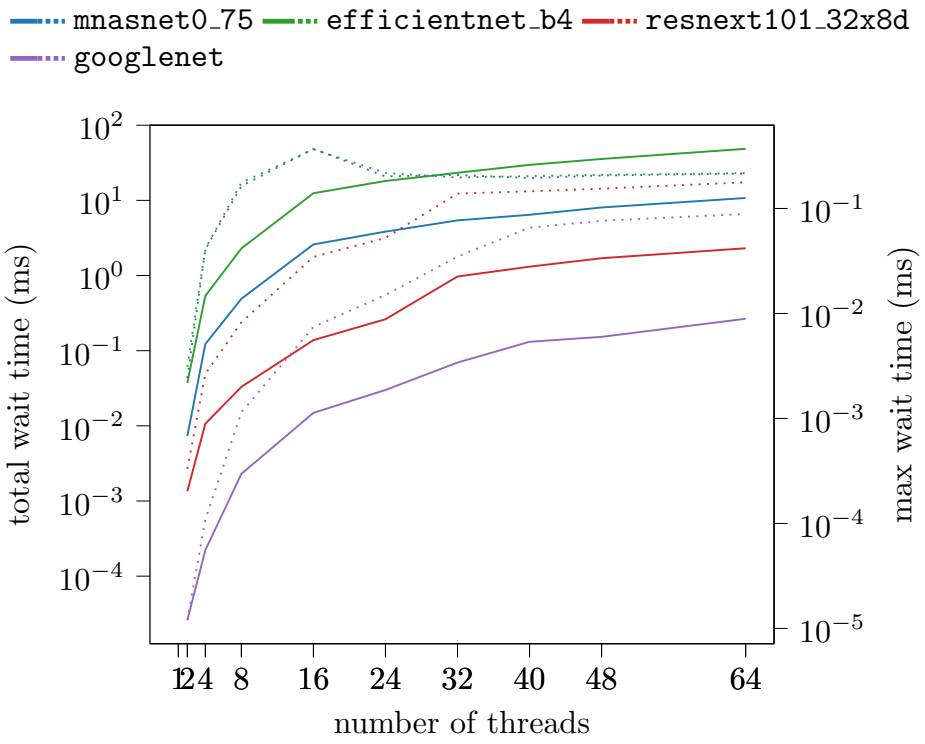


Figure 2.2: Maximum (dashed line) and total (solid line) lock wait times for the entire `jemalloc` arena.

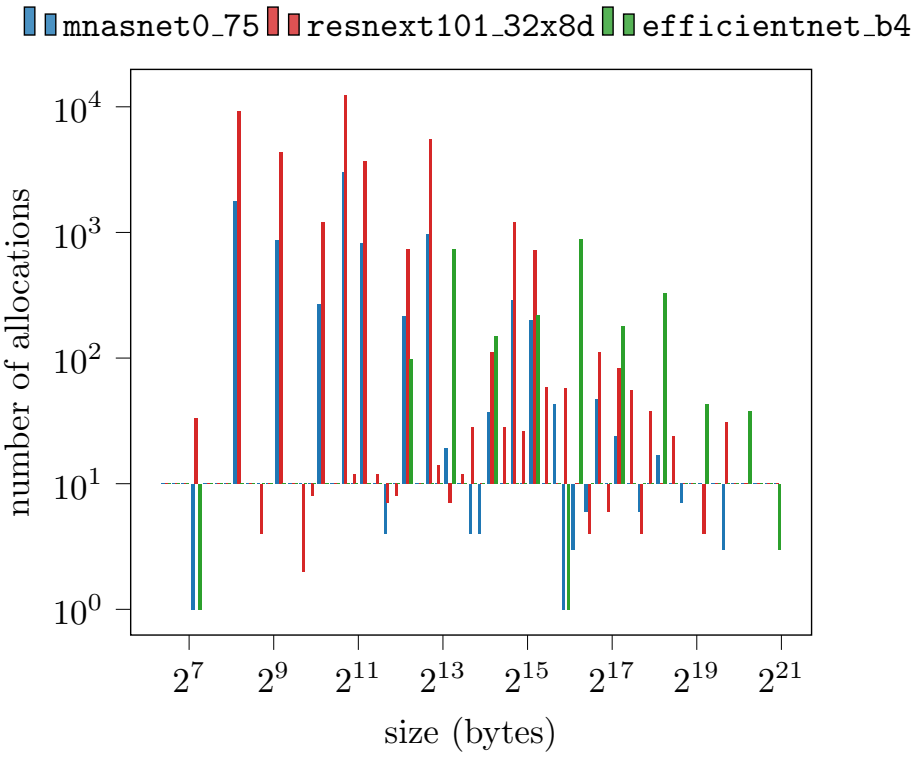


Figure 2.3: Distributions of intermediate allocations for various DNNs. Note that size is log scaled.

where tensors with overlapping lifetimes are constrained to be ordered in memory by

$$\texttt{offset}_i + \texttt{mem}_i \leq \texttt{offset}_j + z_{ij} * \texttt{total\_mem}$$

$$\texttt{offset}_j + \texttt{mem}_j \leq \texttt{offset}_i + \left(1 - z_{ij}\right) * \texttt{total\_mem}$$

Here z_{ij} are decision variables, defined as

$$z_{ij} \coloneqq \begin{cases} 0 & \text{if } \texttt{offset}_i + \texttt{mem}_i \leq \texttt{offset}_j \\ 1 & \text{if } \texttt{offset}_j + \texttt{mem}_j \leq \texttt{offset}_i \end{cases}$$

that determine ordering (in address space) of allocations that overlap in lifetime.

While the offsets that comprise the solution to the MIP formulation are provably correct and optimal, the MIP is, in general, computationally intractable (Li et al., 2004). The best-known polynomial-time approximation is $2 + \varepsilon$ by Buchsbaum (Buchsbaum et al., 2003), over the previously $3 + \varepsilon$ best by Gergov (Gergov, 1999). There also exist simpler heuristics that generally perform well in terms of peak memory usage, fragmentation, and planning time. In this work, we consider five distinct memory planning strategies:

- `bump_allocation`, the baseline allocation strategy that consists of iterating through all allocations and maintaining a maximum offset, which is incremented ("bumped") for each new allocation;
- `mip` (Sekiyama et al., 2018), i.e., offsets computed by solving the MIP optimization specified by Eqns. 2.1;
- `gergov` (Gergov, 1999), Gergov's $3 + \varepsilon$ approximation, based on constructing an infeasible solution and then transforming to a feasible solution using the Best Fit heuristic for interval graph coloring;
- `greedy_by_size` (Pisarchyk and Lee, 2020), that operates by sorting all intermediate

allocations by size and then proceeding to assign offsets for overlapping (in lifetime) tensors according to a best fit criterion;

- `mincost_flow` (Lee et al., 2019), which frames the allocation problem as a minimum cost flow problem (with edges in the flow network corresponding to memory reuse).

We evaluate these strategies for the purposes of designing the memory planning component of `MemoMalloc` (see Section 2.2.2).

2.2 Implementation

Our implementation consists of three components:

- A hybrid static analysis and profiling component that captures sizes and lifetimes of all memory allocations;
- A memory planner that constructs structured plans, consisting of an initial memory allocation and offsets for allocations associated with each operator of the DNN;
- A runtime component that effectuates the memory plan by computing runtime offsets and instantiating tensors, which are then consumed by operators.

We describe each component in turn.

2.2.1 Profiling

Recall the ultimate goal of our system: statically allocating all memory necessary for a forward pass iteration of a DNN. In order to accomplish this goal, it is necessary to describe accurately and uniquely all allocations made during a forward pass. Initial implementations involved recovering sizes of intermediate tensors wholly from the TS IR representation of a DNN. While practical and conceptually straightforward (involving propagating input shapes

on tensors and computing tensor sizes from outputs of operators) it suffers from a critical flaw: since TS IR is a higher-level representation of the DNN than the kernel-level implementations, it does not capture all allocations made during the execution of the DNN (see Table 2.2). Primarily, this a product of operators that delegate to generic implementations; for example, a `max_pool2d` operation could appear in the TS IR as

```
%input.177 : Float(1, 512, 15, 15, strides=[...])
                          = aten::max_pool2d(%input.151, %4, %3, %3, %3, %6)
```

and reflect only a single output tensor, but whose actual implementation (see Listing 3) delegates to one of various specializations, and then, potentially, immediately frees parts of the results. Such implementation-dependent allocations are not reflected at the IR level and are fairly common. While it might be argued that such issues should be handled in a principled manner (e.g., by refactoring `max_pool2d_with_indices`) such delegation is necessary given the breadth of operators that PyTorch supports.

Another complication involved in using TS IR to reconstruct all tensor lifetimes is the inherent aliasing of names; while TS is equipped with alias analysis infrastructure, it is, by necessity, conservative. For example, TS does not attempt to analyze aliasing of tensors that are inserted into containers (such as `Dict`, `List`, and `Tuple`). Nor is it able to precisely infer aliasing relationships between tensors that are never materialized but are actually *views* on tensors (e.g., *slices* of tensors). In fact, memory planning in the context of this type of aliasing leads to "over-planning", i.e., overestimation of memory needs due to planning for tensors that do not correspond to unique allocations.

Note that the diametrically opposed alternative, namely a purely memorization-based approach that depends solely on the order of allocations, would be brittle with respect to relationships between operators and allocations. This is because such relationships are critical for adjusting memory plans post any optimization passes (such as those performed by an optimizing JIT) that occur after constructing a memory plan. Consider a "ResBlock"

Listing 3 `max_pool2d` C++ implementation. Note, in the case of delegating to `at::max_pool2d_with_indices`, an immediate `free` occurs when `std::get<0>(output_and_indices)` is tail called.

```
Tensor max_pool2d(
    const Tensor& self,
    IntArrayRef kernel_size,
    IntArrayRef stride,
    IntArrayRef padding,
    IntArrayRef dilation,
    bool ceil_mode) {
  if (self.is_quantized()) {
    return at::quantized_max_pool2d(
      self, kernel_size, stride, padding, dilation, ceil_mode
    );
  }
  if (self.is_mkldnn()) {
    return at::mkldnn_max_pool2d(
      self, kernel_size, stride, padding, dilation, ceil_mode
    );
  }
  auto output_and_indices = at::max_pool2d_with_indices(
    self, kernel_size, stride, padding, dilation, ceil_mode
  );
  return std::get<0>(output_and_indices);
}
```

in a ResNet (see Figure 2.4) where control flow diverges after the `MaxPool` activation layer; since there is no total order of operations on distinct paths, a JIT compiler is free to reorder them. This has implications for the allocations performed by those operators. Consider the `Conv` + `BatchNormalization` pairs of operators, which make intermediate allocations of the same sizes but with differing lifetimes. If a given memory plan assigns memory addresses $[\texttt{offset}_1, \texttt{offset}_1 + \mathit{size})$ to the intermediate tensor in Group 1, computed under the assumption that its lifetime covers (see Figure 2.5a) the lifetime of the intermediate tensor in Group 2 (with assigned memory addresses $[\texttt{offset}_2, \texttt{offset}_2 + \mathit{size})$), then a reordering of those operations such that Group 1's `BatchNormalization` operator executes prior to Group 2's (see Figure 2.5b) would lead to an illegal address access by Group 2's

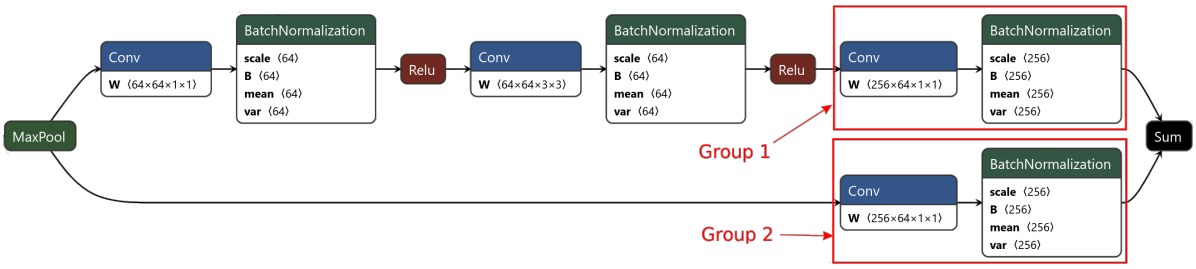
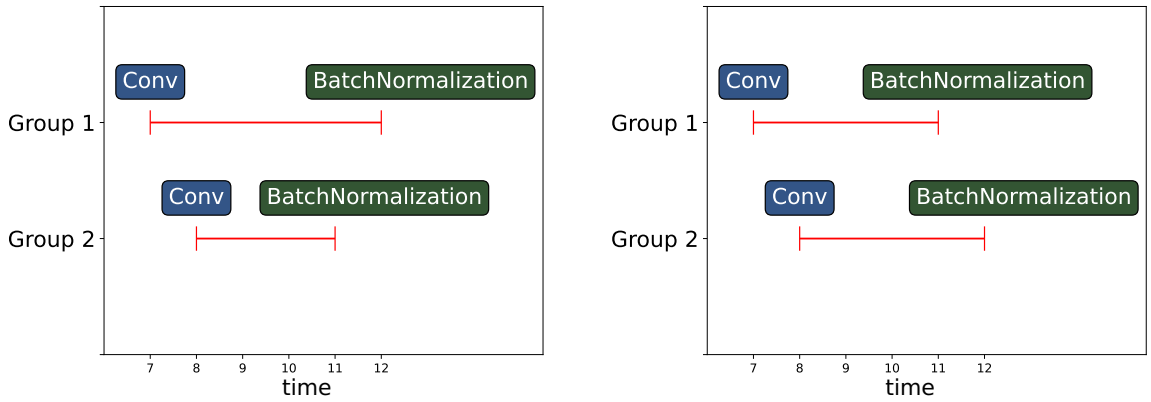


Figure 2.4: A "ResBlock" in a ResNet DNN, where the final `Conv` and `BatchNormalization` layers along both paths require allocations of the same size, but which can be made in arbitrary order (figure created using Netron (Roeder, 2022)).

`BatchNormalization` operator. This cannot be averted, since, at the time of allocation, a purely order-based solution could only distinguish allocations according to lifetime *start*s and tensor *size*s. In the structured approach (i.e., one that unambiguously associates allocations with operators), $\texttt{offset}_1$ and $\texttt{offset}_2$ would be effectively reordered along with their respective operators, thus avoiding any illegal memory accesses.



(a) Group 1's intermediate allocation covers Group 2's.

(b) Group 2's intermediate allocation outlives Group 1's.

Figure 2.5: Problematic orderings of operators. If a given memory plan assumes the ordering of operators in Figure 2.5a then a reordering such as that of Figure 2.5b leads to the `BatchNormalization` operator in Group 2 performing an illegal memory access (because its allocation should only "last" until time = 11).

As a result of all of these complexities, we refined our approach and designed a hybrid solution: we use profiling to capture all allocation sizes and lifetimes and avail ourselves of

Table 2.2: Statistics on captured intermediate allocations (total number and total memory), by TS IR versus allocations captured by our profiling approach.

Model	TS IR #	TS IR memory (MB)	Profiling #	Profiling memory
`mnasnet0_75`	98	11	12,931	44
`wide_resnet50_2`	121	41	662	71
`efficientnet_b4`	379	50	57,238	190
`resnext101_32x8d`	240	87	3370	194
`googlenet`	138	11	788	24

the TS IR representation of the DNN. We do so by instrumenting the allocator to record pointer values associated with sizes. We capture this information in tandem with lightweight stack tracing that establishes the provenance of an allocation (i.e., the operator and kernel within whose scope that allocation was made). The stack tracing is "lightweight" in the sense that it does not unwind the stack but maintains an auxiliary stack (which only records calls to functions in the `aten` namespace of the PyTorch library).

One challenging aspect of this approach is in the capture of lifetime endpoints; since calls to `free` only receive a `void*` pointer (and no other metadata about the use of the memory pointed to), there is, in principle, no way to bracket the lifetime of a tensor (i.e., associate `mallocs` with corresponding `frees`). A naive solution could rely on pointer values themselves (in combination with a lookup table that records the size corresponding to a pointer) to make this identification, but this approach fails when the system allocator (that has been instrumented) reuses an address (which one hopes it often does!).

Instead, we employ a tagged pointer (Nam et al., 2019) approach. Specifically, we make use of the fact that, on `x86_64` architectures, pointers only occupy the lower six bytes of an 8-byte word (on `AArch64`, this feature is called Top Byte Ignore (ARM)). Making full use of the upper two bytes, we store a unique identifier, corresponding to each allocation (up to 2^{16} unique allocations) made during the profiling pass. This identifier is then used to uniquely identify `frees` with their corresponding `mallocs`. Note, `x86_64` requires pointers to be in "canonical form" before they are de-referenced (otherwise a "stack fault" is generated). We

resolve this issue by encapsulating the tagged pointers in a smart pointer that canonicalizes (in a standards-compliant way) on dereference (see Listing 4). In addition to enabling us to determine tensor lifetimes, tagged pointers enable us to completely resolve aliases (by querying for this tag at operator and kernel boundaries). Using fully the resolved aliasing relationships, we can reconstruct relationships between operators and the kernels to which they delegate.

2.2.2 Memory Planner

After profiling to collect unambiguous tensor lifetimes and sizes, we statically plan memory allocation for subsequent forward pass iterations. In designing this aspect of the system, we considered the strategies discussed in Section 2.1. In order to evaluate the best planning strategy, we compared execution times and errors (relative to the optimum produced by the MIP). We observed that `greedy_by_size` generally achieves near-optimal results in terms of memory usage. We also evaluated the fragmentation incurred by various memory planning strategies and observed that `greedy_by_size` generally has acceptable fragmentation. In addition to being efficient with respect to peak memory usage, `greedy_by_size` is performant enough to be executed prior to every forward pass of a DNN (see Figure 2.6). Our memory planner executes the `greedy_by_size` strategy by default but can be configured to use any of the other aforementioned planning strategies.

Listing 4 Standards-compliant method of canonicalizing a tagged pointer. The first bitwise AND (`&`) clears the upper 16 bits of the pointer. Then, if bit 47 is 1, the bitwise OR (`|`) sets bits 47 through 63, but if bit 47 is 0, the bitwise OR is a no-op (since it is an OR with 0).

```
inline void* canonicalize(void* ptr) {
  uintptr_t p2 = (((uintptr_t)ptr & ((1ull << 48) - 1)) |
                  -(((uintptr_t)ptr & (1ull << 47)) - 1));
  return (void*)(p2);
}
```

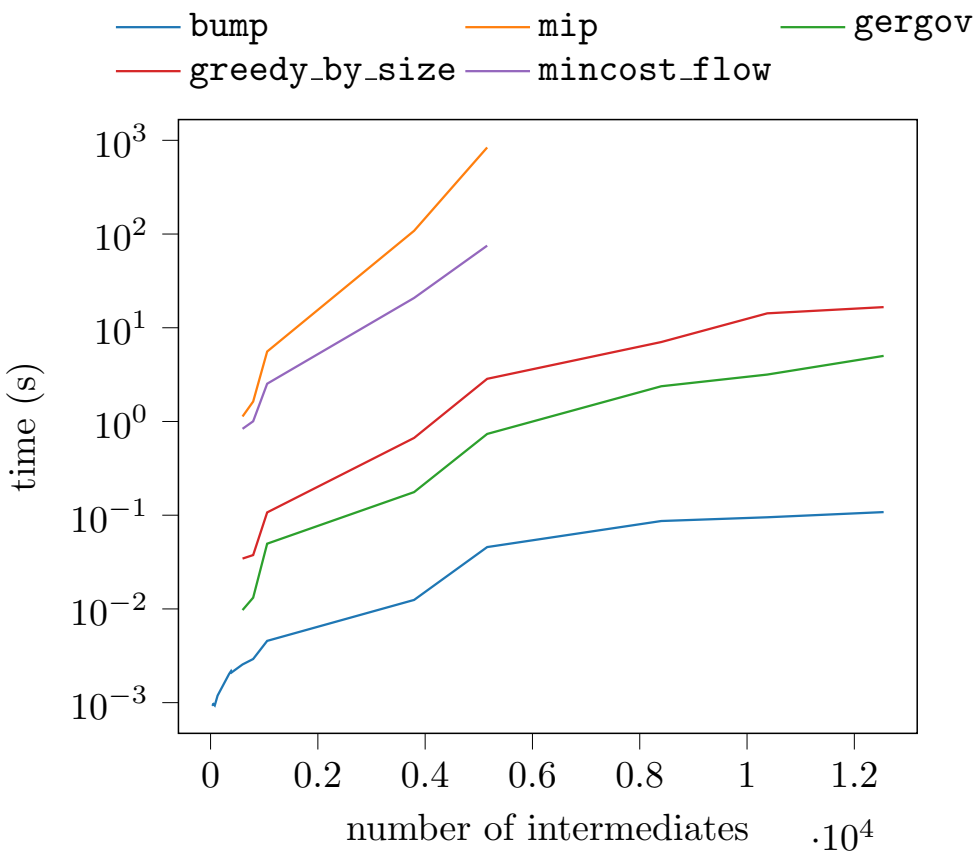


Figure 2.6: Runtimes for memory management strategies across various DNNs (i.e., various numbers of intermediate tensors). Note that `mip` and `mincost_flow` both time out for large numbers of intermediates.

2.2.3 Runtime

After performing memory planning, we use the TS IR to "scope" the allocations to each operator, in order to preserve the structure of the allocations (i.e., groupings of allocations made in the service of carrying out an operation). On subsequent inference passes, we leverage that structure to assign offsets to tensors requested by operators. As already discussed, the alternative, simply assigning offsets on subsequent execution passes in some fixed order, was deemed to be brittle because it prevents plans from being transformed by IR passes that optimize the DNN, i.e., passes that potentially reorder operators and their concomitant allocations (see the discussion in Section 2.2.1). Our extension of TS IR (and the corresponding TS runtime) includes two new primitive operators:

- `prim::AllocateSlab`, borrowing terminology common in the allocator literature, is an operator that allocates all the memory that will be necessary for the duration of the inference pass of the DNN. It takes, as an attribute, the `total_size` and returns a `Storage` value (called `%memory`) backed by this allocation.

- `prim::AllocateTensor`, which takes, as attributes, the `size` and `offset` for the planned allocation that will be requested by the immediately subsequent operator and takes as input the `%memory` value. Internally, it functions in one of two ways: it either constructs a `Tensor` with manually set address (using pointer arithmetic to calculate $\texttt{offset}' = \texttt{offset} + \mathit{start}(\texttt{\%memory})$) if the subsequent operator can directly consume the allocation (i.e., it is an *out variant* operator) or it queues allocations that will be made implicitly by the operator (using, counterintuitively, a stack structure owned by an instance of `MemoMalloc`).

See Listing 5 for a simple example. Note that tensors returned to the user (such as `%5` in Listing 5) are not managed since the solution aims to be orthogonal to other aspects of the PyTorch runtime (i.e., `MemoMalloc` should not own tensors that "escape" the DNN).

Listing 5 Simple memory planning example.

```
graph(%w : Tensor, %x: Tensor, %h: Tensor):
  %memory: Storage = prim::AllocateSlab[total_size=1344]()
  %1: Tensor = prim::AllocateTensor[size=448, offset=0](%memory)
  %2: Tensor = aten::mm(%w, %x, %1)
  %3: Tensor = prim::AllocateTensor[size=448, offset=488](%memory)
  %4: Tensor = aten::add(%2, %h, %3)
  %5: Tensor = aten::relu(%4)
  return (%5)
```

2.3 Evaluation

We evaluate our system (here denoted PyTorch+`MemoMalloc`) on several DNNs that are designed for various computer vision tasks; DCGAN (Radford et al., 2016) is used for representation learning; DeepLabv3 (Chen et al., 2017) and FCN (Shelhamer et al., 2016) are used for semantic segmentation; GoogLeNet (Szegedy et al., 2014), WideResNet (Zagoruyko and Komodakis, 2017), VGG16 (Simonyan and Zisserman, 2015), InceptionV3 (Szegedy et al., 2015), RegNet (Radosavovic et al., 2020), and SqueezeNet (Iandola et al., 2016b) are used for image classification. Due to shifting compute resources available, we made use of two test platforms over the course of our analysis (see Tables 2.3, 2.4), with slightly differing design matrices on each (see Tables 2.5, 2.6).

We evaluate our system against a baseline of PyTorch with memory managed by `jemalloc` (a common pairing in deployments of PyTorch). For PyTorch+`jemalloc`, we set the oversize arena (informed by our analysis in Section 2.1.2) threshold at 1MB, i.e., all allocations with sizes below 1MB are managed by `jemalloc` in the default way, making full use of the thread cache and $n \times 4$ arenas (where n is the number of processor cores, including hyper-threading, on each test platform). For allocations greater than 1MB, the PyTorch+`jemalloc` configuration uses one arena with no thread cache and default decay rates. These configuration parameters are comparable to those typical of PyTorch deployments on server-class platforms (Hazelwood et al., 2018). For PyTorch+`MemoMalloc`, neither a caching allocator nor

Table 2.3: Test platform 1 characteristics.

Component	Value
CPU	AMD(R) Threadripper(R) 3975WX 32-Cores (64 threads)
RAM	128GB DDR4
Hard drive	1.9T Samsung MZVLB2T0HALB-000L7

Table 2.4: Test platform 2 characteristics.

Component	Value
CPU	Intel(R) Xeon(R) Platinum 8339HC 24-Core (48 threads)
RAM	376GB DDR4

an oversize arena is used (i.e., only the single static allocation in combination with a memory plan).

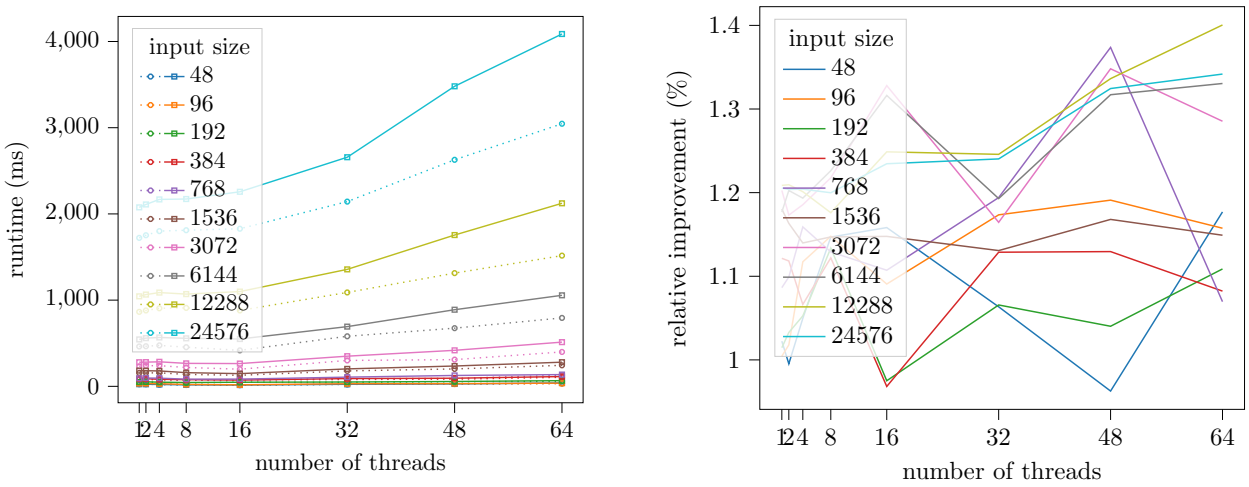
We run each design configuration in a multi-threaded fashion (with the number of threads being a design parameter). Each configuration performs `num_iterations` iterations of its forward pass on inputs with dimensions ranging in batch size and characteristic height/width (i.e., input images are square). Additionally, the configuration with `jemalloc` is run for a warmup period of 10 iterations. We repeat each configuration `num_repeats` times and collect the average execution time across all non-warmup iterations. We report the ratio of execution time between PyTorch+`jemalloc` and PyTorch+`MemoMalloc`. See tables 2.5 and 2.6 for our design matrices. Note that since `batch_size` and `height_width` completely determine input size we group results by input sizes, i.e., $\texttt{input\_size} = 4 \times 3 \times \texttt{batch\_size} \times \texttt{height\_width}^2$ (since all tensors are `float32` tensors, each element comprising 4 bytes, and all inputs have 3 channels). **We report input size in terms kilobytes** to reduce clutter on plots.

Table 2.5: Design matrix for evaluation on test platform 1.

Dimension	Values
`batch_size`	$[1, 4, 8]$
`height_width`	$[128, 256]$
`num_threads`	$[1, 32, 64, 128]$
`num_iterations`	10
`num_repeats`	10

Table 2.6: Design matrix for evaluation on test platform 2.

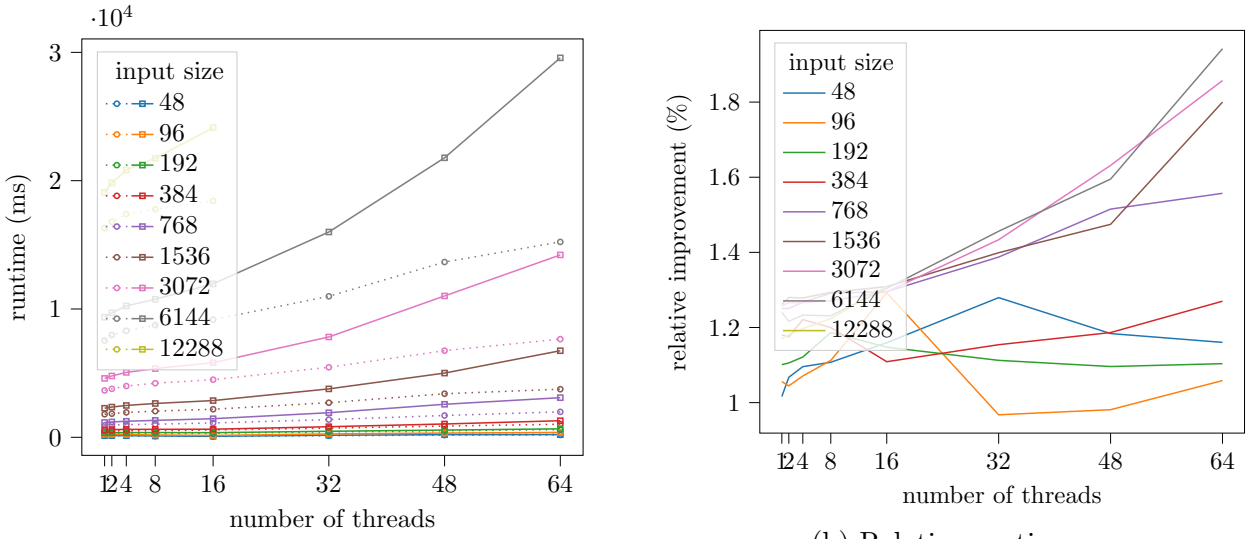
Dimension	Values
`batch_size`	$[1, 4, 8]$
`height_width`	$[64, 128, 256, 512]$
`num_threads`	$[1, 2, 4, 8, 16, 32, 48, 64]$
`num_iterations`	32
`num_repeats`	32



(a) Actual runtimes.

(b) Relative runtimes.

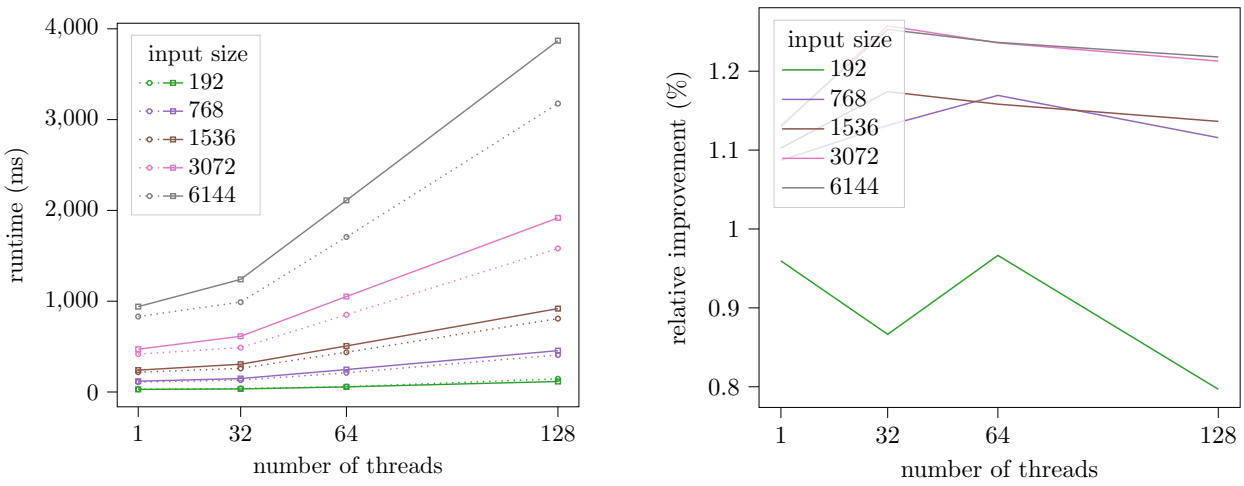
Figure 2.7: Evaluation on `alexnet` on platform 2.



(a) Actual runtimes.

(b) Relative runtimes.

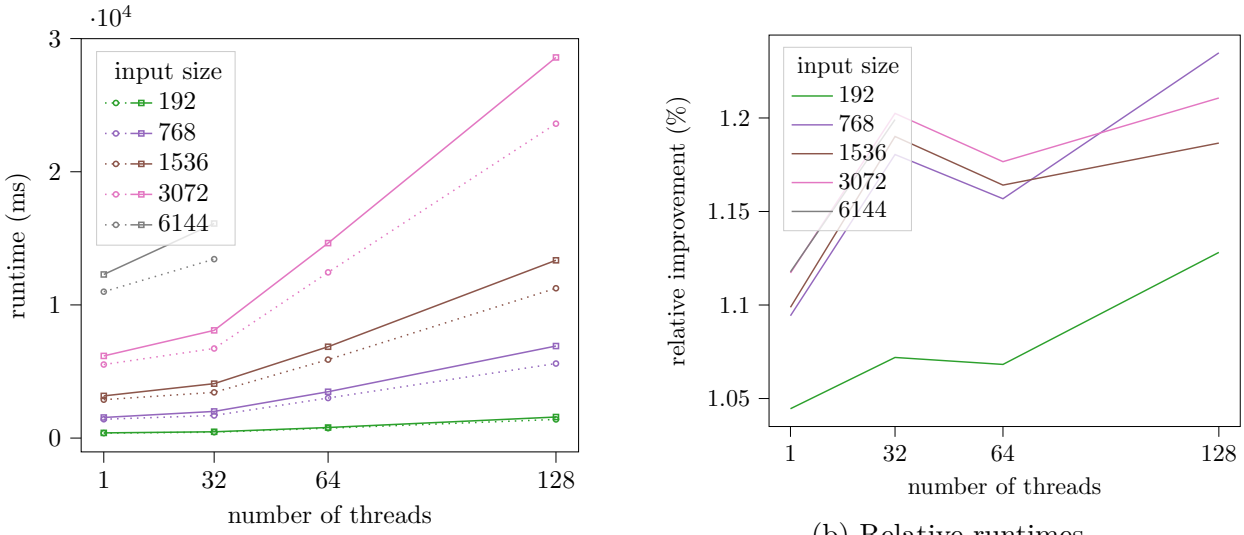
Figure 2.8: Evaluation on `densenet161` on platform 2.



(a) Actual runtimes.

(b) Relative runtimes.

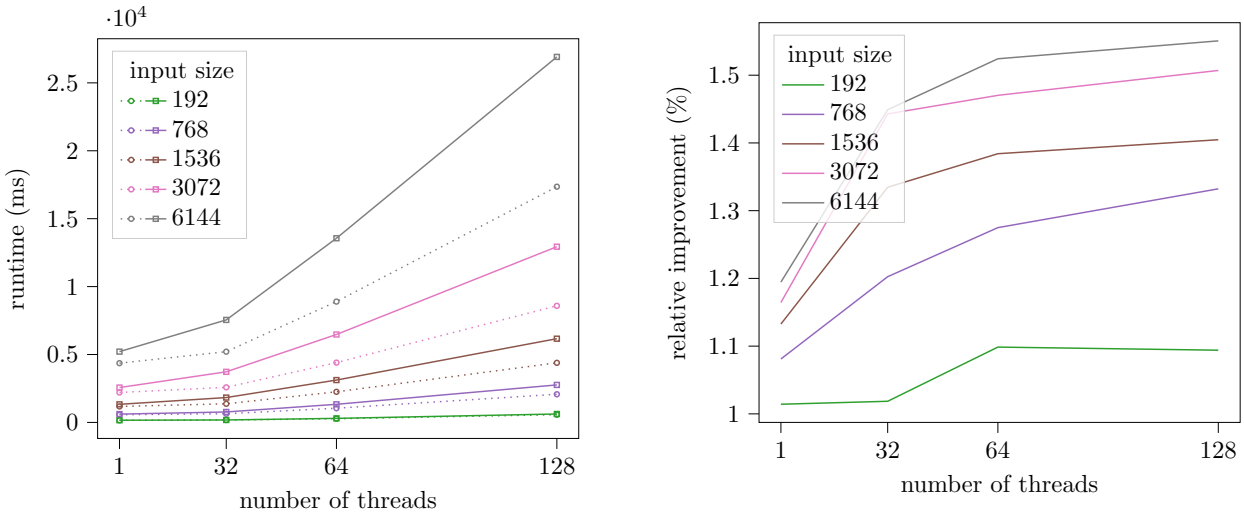
Figure 2.9: Evaluation on `dcgan` on platform 1.



(a) Actual runtimes.

(b) Relative runtimes.

Figure 2.10: Evaluation on `fcn_resnet50` on platform 1.



(a) Actual runtimes.

(b) Relative runtimes.

Figure 2.11: Evaluation on `regnet_x_8gf` on platform 1.

2.4 Discussion

See Figures 2.7, 2.8, 2.9, 2.10, and 2.11 for the resulting runtimes of our evaluation. We observe that PyTorch+`MemoMalloc` robustly performs better than PyTorch+`jemalloc`, in terms of latency, for almost all input sizes and thread counts. How large that performance

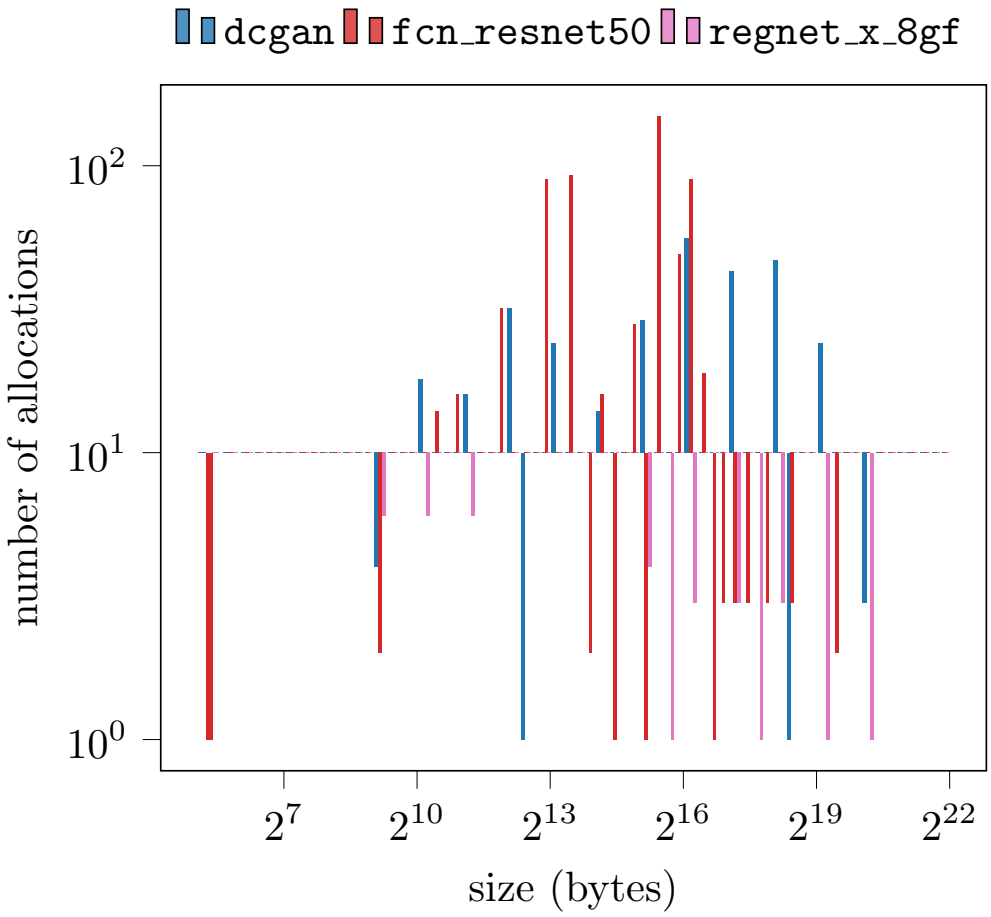


Figure 2.12: Distributions of intermediate allocations for DNNs for which PyTorch+`MemoMalloc` underperforms PyTorch+`jemalloc` at input size = 128.

advantage is, varies amongst the networks, most likely as a function of the arithmetic intensity of the kernels of those networks. In the instances that `MemoMalloc` performs worse, it is the case that most allocations made by those networks fall below the 1MB oversize threshold (see Figure 2.12) and thereby have allocations serviced primarily by `jemalloc`'s thread cache. That is to say, those allocations can be performed with low latency overheads by `jemalloc`'s thread cache, and thus `jemalloc` does not incur any overhead relative to `MemoMalloc`.

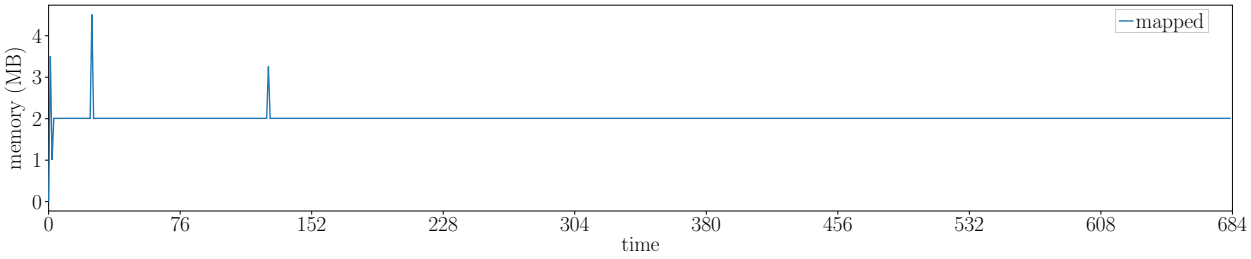
One notable feature of the performance trends is the reduction in relative performance with increasing thread count. That is to say, `MemoMalloc` performs well at 32 threads on

platform 1 and 24 threads on platform 2, but then that relative performance slowly decays. This is most likely because the processors on our test platforms in fact possess fewer physical cores than reported to the operating system (due to hyper-threading). The limited number of cores (relatively speaking) acts as a natural "speed bump" on the number of operations a given thread can perform over the course of executing the DNN (thus constraining the maximum amount of `mutex` contention in the PyTorch+`jemalloc` configuration). This is evident from the overall increase in runtime experienced for all input sizes as a function of thread count.

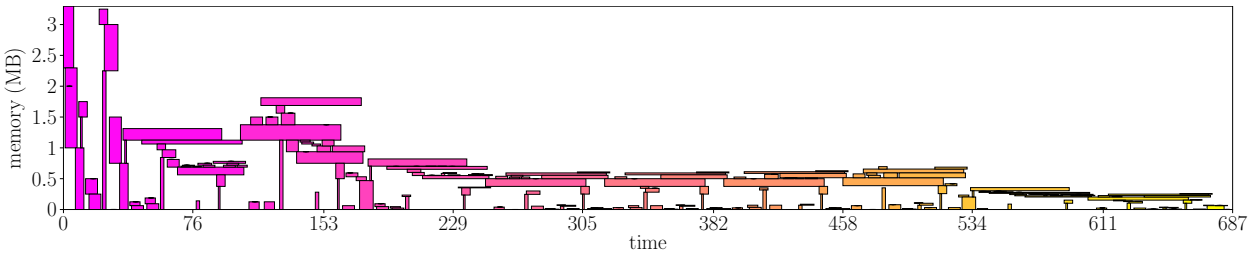
Finally, it is important to consider the tradeoffs made in deploying `MemoMalloc` over `jemalloc`. `MemoMalloc` trades latency for, potentially, higher average memory usage; while peak usage should be comparable (both allocators need to accommodate the maximum necessary memory at any given time), average usage should be higher with `MemoMalloc` because it does not perform any `free`s over the course of the forward pass. To investigate this tradeoff, we collect statistics on the total number of bytes in active extents actually mapped by `jemalloc` (gathered using `mallctl`). Note that `jemalloc` always allocates aligned memory, while `MemoMalloc` only sometimes allocates aligned memory (depending on adjacent allocations), and thus the comparison is only approximate. Consider `googlenet` for input size = 128 (see Figure 2.13). Indeed, we observe that peak usage by `MemoMalloc` is comparable to that of `jemalloc`, average usage is higher. This internal fragmentation is acceptable in environments that have ample memory, or in instances where DNN processes take priority, but could prevent the use of `MemoMalloc` in resource-constrained environments such as embedded devices (see Section 2.6).

2.5 Related work

There is ample related work in this area. Sekiyama et al. (Sekiyama et al., 2018) propose a profiling approach similar to ours. They formally define the offline DSA problem (we make



(a) Total number of bytes in active extents actually mapped by `jemalloc` for `googlenet` for input size = 128.



(b) Heap map for `MemoMalloc` with `greedy_by_size` strategy for `googlenet` for input size = 128.

Figure 2.13: Comparing memory usage for `googlenet` by `jemalloc` versus `MemoMalloc`. Note that the entire ∼3.5MB is kept allocated for the duration of the forward pass.

use of their formalization in Section 2.1.3) and then solve it using a "Best-Fit" heuristic (from (Burke et al., 2004)) for a related problem (the *orthogonal strip-packing problem*). They observe a moderate reduction in intermediate memory allocations across batch sizes and a commensurate reduction in inference latency due to how their framework of choice (Chainer (Tokui et al., 2019)) performs intermediate allocations. Their approach is distinct from ours in that it does not attempt to recover the structure of the DNN.

Lee et al. (Lee et al., 2019) study memory management for DNNs in the context of deployment to mobile devices. In this context, they aim to reduce peak memory usage such that networks may satisfy the memory constraints of on-device accelerators on various mobile phones. To this end, they describe two memory management algorithms: a greedy memory management algorithm that allocates a pool of shared objects on an operator-by-operator basis, and the `mincost_flow` strategy we described in Section 2.1.3. They report satisfactory performance improvements but primarily due to successfully migrating from CPU to the on-device accelerators. They do not attempt to capture allocations made by

kernel implementations of operators (which do occur in their framework of choice, TensorFlow Lite).

Pisarchyk et al. (Pisarchyk and Lee, 2020) also study memory management in the context of DNNs but with respect to peak usage rather than execution latency. They evaluate the same set of memory planning strategies as us, in addition to a strategy called *Greedy by Breadth*. Greedy by Breadth operates under the assumption that intermediate tensors of large sizes typically cluster, on an operator-by-operator basis (i.e., large inputs to operators produce large outputs). Thus, they sort (in decreasing order) operators by a measure they define as *breadth* (the sum of sizes of input and output tensors) and assign offsets in this order. Pisarchyk et al. evaluate their strategies on various DNNs tailored to deployment on edge devices. While they observe that Greedy by Size achieves near optimal results (in concordance with our evaluation) they do not make any use of the additional structure of the DNN, nor do they attempt to perform alias analysis of tensors.

Nimble (Shen et al., 2021) does make use of the intermediate representation of the DNN and similarly inserts primitive allocation operations into the IR, but, critically, Nimble does not introspect into implementations of operators and therefore elides any implicit allocations. Notably, TVM (closely related to Nimble) began discussions regarding static memory planning at approximately the same time as this project began.

One important body of work possessing high affinity with our own is the Multi-level Intermediate Representation (MLIR) project (Lattner et al., 2020a). In the MLIR framework, there exist many intermediate representations (called *dialects*), that enable the specification of DNNs at various levels of abstraction. In particular, in the `linalg` dialect, sequences of DNN operators are decomposed in terms of the corresponding linear algebra; consider the representation of `conv` in Listing 6. The important feature of this representation to note is that the allocation `%3 = memref.alloc()` for the output of the convolution is explicitly represented, along with its shape `memref<1x32x112x112xf32>` (along with the shapes of all

Listing 6 Representation of `conv` in the `linalg` dialect of MLIR.

```
func @conv(%input: tensor<1x3x225x225xf32>, %filter: tensor<32x3x3x3xf32>,
           %output: tensor<1x32x112x112xf32>)
  -> tensor<1x32x112x112xf32> {
    %0 = bufferization.to_memref %input : memref<1x3x225x225xf32>
    %1 = bufferization.to_memref %filter : memref<32x3x3x3xf32>
    %2 = bufferization.to_memref %output : memref<1x32x112x112xf32>
    %3 = memref.alloc() : memref<1x32x112x112xf32>
    linalg.copy(%2, %3) : memref<1x32x112x112xf32>,memref<1x32x112x112xf32>
    linalg.conv_2d_nchw_fchw
      {
        dilations = dense<1> : tensor<2xi64>,
        strides = dense<2> : tensor<2xi64>
      }
      ins(%0, %1: memref<1x3x225x225xf32>, memref<32x3x3x3xf32>)
      outs(%3: memref<1x32x112x112xf32>)
    %4 = bufferization.to_tensor %3 : memref<1x32x112x112xf32>
    return %4 : tensor<1x32x112x112xf32>
  }
```

other tensors). This straightforwardly enables the writing of a compiler pass that implements static memory planning; indeed in MLIR this is called a "comprehensive bufferization" and uses essentially the `mincost_flow` strategy.

2.6 Conclusion

We studied the memory allocation patterns of DNNs, with respect to latencies incurred by synchronization mechanisms in conventional caching allocators. We then proposed and implemented a memory planning system for reducing such latencies (during inference) for DNNs. We evaluated our system and observed that it performs better than `jemalloc` for typical DNN workloads. In the future, we intend to factor out `MemoMalloc` into an independent module with a uniform API such that it can be plugged into any of the popular deep learning frameworks.

Future work in this area includes several directions:

- **Dynamics**: All of our work here assumes that there is no control flow and that all intermediate tensor sizes are fixed. In practice, this is only the case in certain environments and it would be preferable to be able to perform memory planning in the context of both control flow and dynamic intermediate tensor sizes. Our preliminary work indicates that in fact, this is possible; for DNNs where intermediate tensor sizes can be algebraically inferred from input shapes, it is possible to construct memory plans ahead-of-time (and to cache them) for common input shapes. Such a regime is called *symbolic memory planning*, owing to the employment of *symbolic shape inference* in order to derive algebraic relationships between input shapes and intermediate tensor sizes. The simplest example of this is symbolic memory planning in the context of a dynamic batch size; in this context it can be analytically proven that the MIP solution scales linearly with batch size, thus enabling amortized MIP memory planning.

- **Training**: Our work here has targeted primarily DNN inference, on the assumption that latency matters most in this context. While it is the case that service-level agreements and quality-of-service guarantees impose hard constraints on inference latencies, it is also the case that during training of DNNs, lower latencies could proportionally reduce costs (associated with the research process). The added complexities of training are twofold: firstly, the graph corresponding to backpropagation of gradients must be obtained (i.e., the *backwards graph*), and secondly, intermediate tensors must be kept alive (or stored) in order to be available during gradient computation. Both of these aspects present new challenges for static memory planning. Obtaining the backwards graph in TS IR is currently not possible but alternative tracing mechanisms, such as LazyTensor (Suhan et al., 2021), could be used. Under current assumptions for heuristics memory planning strategies (such as `greedy_by_size`), intermediate tensors that need to be persisted or stored undoubtedly lead to highly fragmented memory plans. Thus, training necessitates a distinct set of heuristics for computing offsets.

- **GPUs**: Motivated by current deployment practices, we have only considered CPU deployment. But it is the case that GPUs are in fact, slowly being adopted as deployment targets for inference. GPUs introduce many novel complications, due to exotic scheduling environments and complicated memory hierarchies; for example, on NVIDIA devices, execution of a group of threads will block on data being absent from shared memory. Despite such complications, there is reason to believe that static memory planning could be feasible on GPUs as well; NVIDIA has recently released an extension to the CUDA API called CUDA Graphs[2] whose use entails "freezing" and reusing fixed sets of memory addresses for multiple iterations of arbitrary sequences of kernels. Preliminary exploration of this API has shown that it does in fact reduce many of the latencies associated with allocation.

- **Edge Devices**: Recently edge platforms (mobile phones, wearables, IoT sensors) have also become feasible deployment targets for DNNs, owing to advances in research on DNN architectures that maintain accuracy while reducing resource consumption (such as quantized (Wu et al., 2016) and sparse networks (Xu et al., 2018)). These advances notwithstanding, those platforms reproduce many of the phenomena of their larger scale analogues (Suo et al., 2021). Namely, memory consumption of DNNs on edge devices is of significant importance, due to proportionally scaled memories (i.e., relatively small), limited memory bandwidth capacities (Wu et al., 2019), and less powerful memory management units (Deligiannis and Kornaros, 2016). Simultaneously, limited threading capabilities impose constraints on the complexity (and therefore sophistication) of possible memory management schemes, such as dynamic allocators (Ramakrishna et al., 2008) and software virtual memory (Bai et al., 2009). We are investigating deploying `MemoMalloc` on such platforms.

2. https://developer.nvidia.com/blog/cuda-graphs/

CHAPTER 3

BRAGGHLS: HIGH-LEVEL SYNTHESIS FOR LOW-LATENCY DEEP NEURAL NETWORKS FOR EXPERIMENTAL SCIENCE

High data rates are observed and, consequently, large datasets are generated, across a broad range of science experiments in domains such as high-energy physics, materials science, and cosmology. For example, in high-energy physics, the LHCb detector at the Large Hadron Collider (LHC) is tasked with observing the trajectories of particles produced in proton-proton collisions at 40 MHz (Gligorov, 2015). With a packet size of approximately 50 kB (per collision), this implies a data rate of approximately 2 TB/s. Ultimately, in combination with other detectors, the LHC processes approximately 100 EB of data per year. In materials science, Bragg diffraction peak analysis, which provides non-destructive characterization of single-crystal and polycrystalline structure and its evolution in a broad class of materials, can have collection rates approaching 1 MHz (Hammer et al., 2021), with a corresponding packet size of 80 kB. In cosmology, the Square Kilometer Array, a radio telescope projected to be operational by 2027 (McMullin et al., 2022), will sustain data rates in excess of 10 TB/s (Grainge et al., 2017).

Storing and distributing such large quantities of data for further analysis is cost prohibitive. Thus, data must be compressed or (as we consider here) filtered to preserve only the most "interesting" elements at the time of collection, an approach that reduces storage needs but imposes stringent latency constraints on the filtering mechanisms. Typically, filtering mechanisms consist of either physics-based (Collaboration, 2020) or machine learning models (Gligorov and Williams, 2013); in either case, maximally efficient and effective use of the target hardware platform is important. Irrespective of the technique employed, almost universally, for ultra-low (e.g., sub-microsecond) latency use cases the implementation is de-

ployed to either field-programmable gate arrays (FPGAs) or application-specific integrated circuits (ASICs) (Duarte et al., 2018). Here we focus primarily on FPGAs.

Deep neural networks (DNNs), a particular type of machine learning model, have been shown to be effective in many scientific and commercial domains due to their representational capacity, i.e., their ability to represent (approximately) diverse sets of mappings (Alzubaidi et al., 2021). DNNs "learn" to represent a mapping over the course of "training," wherein they are iteratively evaluated on sample data while a "learning rule" periodically updates the *weights* that parameterize the DNN. In recent years, DNNs have been investigated for near real-time scientific use cases (Liu et al., 2019, Patton et al., 2018, Liu et al., 2022a) but their use for the lowest latency use cases has been limited (Duarte et al., 2018), for three reasons:

1. Graphics Processing Units (GPUs), the conventional hardware target for DNNs, are not sufficiently performant for these high data rate, low latency, use cases (due to their low clock speeds and low peripheral bandwidth, until recently (Aaij et al., 2020));

2. DNNs, by virtue of their depth, require substantial memory (for weights) and compute (floating-point arithmetic), thereby preventing their deployment to FPGAs, which, in particular, have limited static RAM;

3. DNNs are (typically) defined, trained, and distributed by using high-level frameworks (e.g., PyTorch (Paszke et al., 2017), TensorFlow (Abadi et al., 2016b), MXNet (Chen et al., 2015)), which abstract all implementation details, thereby making portability of model architectures to unsupported hardware platforms (e.g., FPGAs and ASICs) close to non-existent (barring almost wholesale re-implementations of the frameworks).

These three barriers demand a solution that can translate a high-level DNN representation to a low-level representation, suitable for FPGA deployment, while simultaneously optimizing resource usage and minimizing latency. In general, the task of *lowering* high-

level representations of programs to low-level representations is the domain of a compiler. Similarly, the task of *synthesizing* a *register-transfer level* (RTL) *design*, rendered in a *hardware description language* (HDL), from a program, is the domain of high-level synthesis (HLS) (Nane et al., 2016) tools. Existing HLS tools (Canis et al., 2013, Zhang et al., 2008, Ferrandi et al., 2021) struggle to perform needed optimizations in reasonable amounts of time (see Section 3.1.2) despite, often, bundling robust optimizing compilers.

Recently, deep learning compilers (e.g., TVM (Chen et al., 2018), MLIR (Lattner et al., 2020b), and Glow (Rotem et al., 2018)) have demonstrated the ability to reduce dramatically inference latencies (Liu et al., 2018), training times (Zheng et al., 2022), and memory usage (Chen et al., 2016). These compilers function by extracting intermediate-level representations (IRs) of the DNNs from the representations produced by the frameworks, and performing various optimizations (e.g., kernel fusion (Ashari et al., 2015), vectorization (Maleki et al., 2011), and memory planning (Chen et al., 2016)) on those IRs. The highly optimized IR is then used to generate code for various target hardware platforms. Given the successes of these compilers, it is natural to wonder whether they can be adapted to the task of sufficiently optimizing a DNN such that it might be synthesized to RTL, for deployment to FPGA.

In this work, we present `BraggHLS`, an open-source[1], lightweight compiler and HLS framework that can translate DNNs defined as PyTorch models to FPGA-compatible implementations. `BraggHLS` uses a combination of compiler and HLS techniques to compile the entire DNN into fully scheduled RTL, thereby eliminating all synchronization overheads and achieving low latency. `BraggHLS` is general and supports a wide range of DNN layer types, and thus a wide range of DNNs. To the best of our knowledge, `BraggHLS` is the first HLS framework that enables the use of DNNs, free of a dependence on expensive and opaque proprietary HLS tools, for science experiments that demand low-latency inference. In summary our specific

1. Available at `https://github.com/makslevental/BraggHLS`

contributions include:

1. We describe and implement a compiler framework, `BraggHLS`, that can efficiently transform, without use of proprietary HLS tools, unoptimized, hardware-agnostic PyTorch models into low-latency RTL suitable for deployment to FPGAs;

2. We show that `BraggHLS` generates lower latency designs than does a state-of-the-art commercial HLS tool (Xilinx's Vitis HLS) for many DNN layer types. In particular we show that `BraggHLS` can produce synthesizable designs that meet placement, routing, and timing constraints for `BraggNN`, a DNN designed for analyzing Bragg diffraction peaks;

3. We discuss challenges faced even after successful synthesis of RTL from a high-level representation of a DNN, namely during the place and route phases of implementation.

Note that while we focus here, for illustrative purposes, on optimizations relevant to a DNN used for identifying Bragg diffraction peaks in materials science, `BraggHLS` supports a wide range of DNNs, limited only by upstream support for DNN layers.

The rest of this chapter is as follows: Section 3.1 reviews key concepts from compilers, high-level synthesis, and RTL design for FPGA. Section 3.2 describes the `BraggHLS` compiler and HLS framework in detail. Section 3.3 evaluates `BraggHLS`'s performance, scalability, and competitiveness with designs generated by Vitis HLS, and describes a case study in which `BraggHLS` is applied to `BraggNN`, a Bragg peak detection DNN with a target latency of 1 µs/sample. Section 3.4 discusses related work in this area. Finally, Section 3.5 concludes and discusses future work.

3.1 Background

We briefly review relevant concepts from DNN frameworks and compilers, high-level synthesis, and FPGA design. Each subsection corresponds to a phase in the translation from

high-level DNN to feasible FPGA implementation.

3.1.1 Compilers: The path from high to low

The path from a high-level, abstract, DNN representation to a register-transfer level representation can be viewed as a sequence of progressive lowerings between adjacent levels of abstraction. Each level of abstraction is rendered as a programming language, IR, or HDL, and thus we describe each lowering in terms of the representations and tools used by `BraggHLS` to manipulate those representations:

1. An imperative, *define-by-run,* Python representation, in PyTorch;
2. High-level data-flow graph representation, in TorchScript;
3. Low-level data and control flow graph representation, in Multi-Level Intermediate Representation (MLIR).

PyTorch and TorchScript

Typically DNN models are represented in terms of high-level frameworks, themselves implemented within general purpose programming languages. Such frameworks are popular because of their ease of use and large library of example implementations of various DNN model architectures. `BraggHLS` targets the PyTorch framework. DNNs developed within PyTorch are *defined-by-run*: the author describes the DNN imperatively in terms of high-level operations, using Python, which, when executed, materializes the (partial) high-level data-flow graph (DFG) corresponding to the DNN (e.g., for the purposes of reverse-mode automatic differentiation). From the perspective of the user, define-by-run enables fast iteration at development time, possibly at the cost of some runtime performance.

Yet from the perspective of compilation, define-by-run precludes efficient extraction of the high-level DFG; since the DFG is materialized only at runtime, it cannot easily be statically

inferred from the textual representation (i.e., the Python source) of the DNN. Furthermore, a priori, the runtime-materialized DFG is only partially materialized (Paszke et al., 2017), and only as an in-memory data structure. Thus, framework support is necessary for efficiently extracting the full DFG. For this purpose, PyTorch supports a Single Static Assignment (SSA) IR, called TorchScript (TS) IR and accompanying tracing mechanism (the TS JIT), which generates TS IR from conventionally defined PyTorch models. Lowering from PyTorch to TS IR enables various useful analyses and transformations on a DNN at the level of the high-level DFG, but targeting FPGAs requires a broader collection of transformations. To this end, we turn to a recent addition to the compiler ecosystem, MLIR.

MLIR

MLIR (Lattner et al., 2020b) presents a new approach to building reusable and extensible compiler infrastructure. MLIR is composed of a set of *dialect* IRs, subsets of which are mutually compatible, either directly or by way of translation/legalization. The various dialects aim to capture and formalize the semantics of compute intensive programs at varying levels of abstraction, as well as namespace-related sets of IR transformations. The entrypoint into this compiler framework from PyTorch is the `torch` dialect (Silva and Elangovan, 2021), a high-fidelity mapping from TS IR to MLIR native IR, which, in addition to performing the translation to MLIR, fully refines all shapes of intermediate tensors in the DNN (i.e., computes concrete values for all dimensions of each tensor), a necessary step for downstream optimizations and eliminating inconsistencies in the DNN (Hattori et al., 2022).

While necessary for lowering to MLIR and shape refinement, the `torch` dialect represents a DNN at the same level of abstraction as TS IR: it does not capture the precise data and control flow needed for de novo implementations of DNN operations (e.g., for FPGA). Fortunately, MLIR supports lower-level dialects, such as `linalg`, `affine`, and `scf`. The `scf` (structured control flow) dialect describes standard control flow primitives, such as condi-

tionals and loops, and is mutually compatible with the `arith` (arithmetic operations) and `memref` (memory buffers) dialects. The `affine` dialect, on the other hand, provides a formalization of semantics that lend themselves to polyhedral compilation techniques (Bondhugula, 2020) that enable loop dependence analysis and loop transformations. Such loop transformations, particularly loop unrolling, are crucial for achieving lowest possible latencies (Ye et al., 2022) because loop nests directly inform the concurrency and parallelism of the final RTL design.

3.1.2 High-level synthesis

High-level synthesis tools produce RTL descriptions of designs from high-level representations, such as C or C++ (Canis et al., 2013, Ferrandi et al., 2021). In particular, Xilinx's Vitis HLS, based on the Autopilot project (Zhang et al., 2008), is a state-of-the-art HLS tool. Given a high-level, procedural, representation, HLS carries out three fundamental tasks, in order to produce a corresponding RTL design:

1. HLS schedules operations (such as `mulf`, `addf`, `load`, `store`) in order to determine which operations should occur during each clock cycle; such a schedule depends on three characteristics of the high-level representation: (a) the topological ordering of the DFG of the procedural representation (i.e., the dependencies of operations on results of other operations and resources); (b) the delay for each operation; and (c) the user's desired clock rate/frequency.

2. HLS associates (*binds*) floating point operations to RTL instantiations of intellectual property (IP) for those operations; for example whether to associate an addition operation followed by a multiply operation to IPs for each, or whether to associate them both with a single IP, designed to perform a fused multiply-accumulate (MAC). In the case of floating-point arithmetic operations, HLS also (with user guidance) determines the precision of the floating-point representation.

3. HLS builds a finite-state machine (FSM) that implements the schedule of operations as control logic, i.e., logic that initiates operations during the appropriate stages of the schedule.

In addition to fulfilling these three fundamental tasks, HLS aims to optimize the program. In particular, HLS attempts to maximize concurrency and parallelism (number of concurrent operations scheduled during a clock cycle) in order maximize the throughput and minimize the latency of the final implementation. Maximizing concurrency entails pipelining operations: operations are executed such that they overlap in time when possible, subject to available resources. Maximizing parallelism entails partitioning the DNN into subsets of operation that can be computed independently and simultaneously and whose results are aggregated upon completion.

While HLS aims to optimize various characteristics of a design automatically, there are challenges associated this automation. In particular, maximum concurrency and parallelism necessitates data-flow analysis in order to identify data dependencies among operations, both for scheduling and identifying potential data hazards. Such data-flow analysis is expensive and grows (in runtime) as better performance is pursued. This can be understood in terms of loop-nest representations of DNN operations.

For example, consider the convolution in Listing 7. A schedule that parallelizes (some of) the arithmetic operations for this loop nest can be computed by first unrolling the loops up to some "trip count" and then computing the topological sort of the operations. When using this *list scheduling* algorithm, the degree to which the loops are unrolled determines how many arithmetic operations can be scheduled in parallel. The issue is that the `store`s and `load`s on the `output` array prevent reconstruction of explicit relationships between the inputs and outputs of the arithmetic operations across loop iterations. The conventional resolution to this loss of information is to perform *store-load forwarding*: pairs of `store` and `load` operations on the same memory address are eliminated, with the operand of the

Listing 7 Python representation of a padding $\lfloor k/2 \rfloor$, stride 1, c_{out} filter convolution with $k \times k$ kernel applied to (b, c_{in}, h, w)-dimensional `input` tensor; b is batch size, c_{in} is number of channels, and (h, w) are height and width, respectively.

```
def conv2d(
  input: MemRef(b, c_in, h, w),
  output: MemRef(b, c_out, h, w),
  weight: MemRef(c_out, c_in, k, k)
):
  for i1 in range(0, b):
    for i2 in range(0, c_out):
      for i3 in range(0, h):
        for i4 in range(0, w):
          for i5 in range(0, c_in):
            for i6 in range(0, k):
              for i7 in range(0, k):
                _3 = i3 + i6
                _4 = i4 + i7
                _5 = input[i1, i5, _3, _4]
                _6 = weight[i2, i5, i6, i7]
                _7 = output[i1, i2, i3, i4]
                _8 = _5 * _6
                _9 = _7 + _8
                output[i1, i2, i3, i4] = _9
```

`store` forwarded to the users of the `load` (see Listing 8). Ensuring correctness of this transformation (i.e., that it preserves program semantics) requires verifying, for each pair of candidate `store` and `load` operations, that there is no intervening memory operation on the same memory address. These verifications are non-trivial since the iteration spaces of the loops need not be regular; in general it might involve solving a small constraint satisfaction program (Rajopadhye, 2002). Furthermore, the number of required verifications grows polynomially in the convolution parameters, since the loop nest unrolls into $b \times c_{out} \times h \times w \times c_{in} \times k^2$ `store-load` pairs on the `output` array.

Finally, note, although greedy solutions to the scheduling problem solved by HLS are possible, the scheduling problem, in principle, can be formulated as an integer linear program (ILP), for which the corresponding decision problem is complete for NP. In summary, HLS tools solve computationally intensive problems in order to produce an RTL description of a high-level representation of a DNN. These phases of the HLS process incur "development time" costs (i.e., runtime of the tools) and impose practical limitations on the amount of design space exploration (for the purpose of achieving latency goals) which can be performed. `BraggHLS` addresses these issues by enabling the user to employ heuristics during both the parallelization and scheduling phases which, while not guaranteed to be correct (but can be *behaviorally verified*) and have much lower runtimes (see Section 3.2.1).

3.1.3 FPGA design

Broadly, at the register-transfer level of abstraction, there remain two more steps prior to being able to deploy a design to an FPGA: a final lowering, so-called logic synthesis, and place and route (P&R). The entire process may be carried out by Xilinx's Vivado tool.

Logic synthesis is the process of mapping RTL to actual hardware primitives on the FPGA (so-called *technology mapping*), such as lookup tables (LUTs), block RAMs (BRAMs), flip-flops (FFs), and digital signal processors (DSPs). Logic synthesis produces a network list

Listing 8 Store-load forwarding across successive iterations (e.g., `i7` = 4, 5) of the inner loop in Listing 7, after unrolling. The forwarding opportunity is from the store on line 19 to the load on line 25; both can be eliminated and `_91` can replace uses of `_72`, such as in the computation of `_92` (and potentially many others).

```
1  def conv2d(
2    input: MemRef(b, c_in, h, w),
3    output: MemRef(b, c_out, h, w),
4    weight: MemRef(c_out, c_in, k, k)
5  ):
6    for i1 in range(0, b):
7      for i2 in range(0, c_out):
8        for i3 in range(0, h):
9          for i4 in range(0, w):
10           ...
11           # e.g., i5, i6, i7 = 2, 3, 4
12           _31 = i3 + i6
13           _41 = i4 + i7
14           _51 = input[i1, i5, _31, _41]
15           _61 = weight[i2, i5, i6, i7]
16           _71 = output[i1, i2, i3, i4]
17           _81 = _51 * _61
18           _91 = _71 + _81
19           output[i1, i2, i3, i4] = _91
20           # i5, i6, i7 = 2, 3, 5
21           _32 = i3 + i6
22           _42 = i4 + i7
23           _52 = input[i1, i5, _32, _42]
24           _62 = weight[i2, i5, i6, i7]
25           _72 = output[i1, i2, i3, i4]
26           _82 = _52 * _62
27           _92 = _72 + _82
28           output[i1, i2, i3, i4] = _92
29           ...
```

(*netlist*) describing the logical connectivity of various parts of the design. Logic synthesis, for example, determines the implementation of floating-point operations in terms of DSPs; depending on user parameters and other design features, DSP resource consumption for floating-point multiplication and addition can differ greatly. Logic synthesis also determines the number of LUTs and DSPs which a high-level representation of a DNN corresponds to, which is relevant to both the performance and feasibility of that DNN when deployed to FPGA.

After the netlist has been produced, the entire design undergoes P&R to determine which configurable logic block within an FPGA should implement each of the units of logic required by the digital design. P&R algorithms need to minimize distances between related units of functionality (in order to minimize wire delay), balance wire density across the entire fabric of the FPGA (in order to reduce route congestion), and maximize the clock speed of the design (a function of both wire delay, logic complexity, and route congestion). The final, routed design, can then be deployed to the FPGA by producing a proprietary *bitstream*, which configures the FPGA.

3.2 The Compiler and HLS framework

`BraggHLS` is an open source compiler and HLS framework that employs MLIR for extracting loop-nest representations of DNNs. Implemented in Python for ease of use and extensibility, it handles the DNN transformations as well as scheduling, binding, and FSM extraction. Importantly, there is no dependence on commercial HLS tools, a property that uniquely enables its use for applications that require the flexibility of open source tool (e.g., the ability to inspect and modify internals in order to adapt to special cases), such as low-latency physical science experiments. Figure 3.1 shows its overall architecture. `BraggHLS` first lowers DNNs from PyTorch to MLIR through TorchScript and the `torch` dialect (see Section 3.1.1) and then from the `torch` dialect to the `scf` dialect (through the `linalg` dialect). Such a

representation lends itself to a straightforward translation to Python (compare Listing 7 to Listing 9) and indeed `BraggHLS` performs this translation.

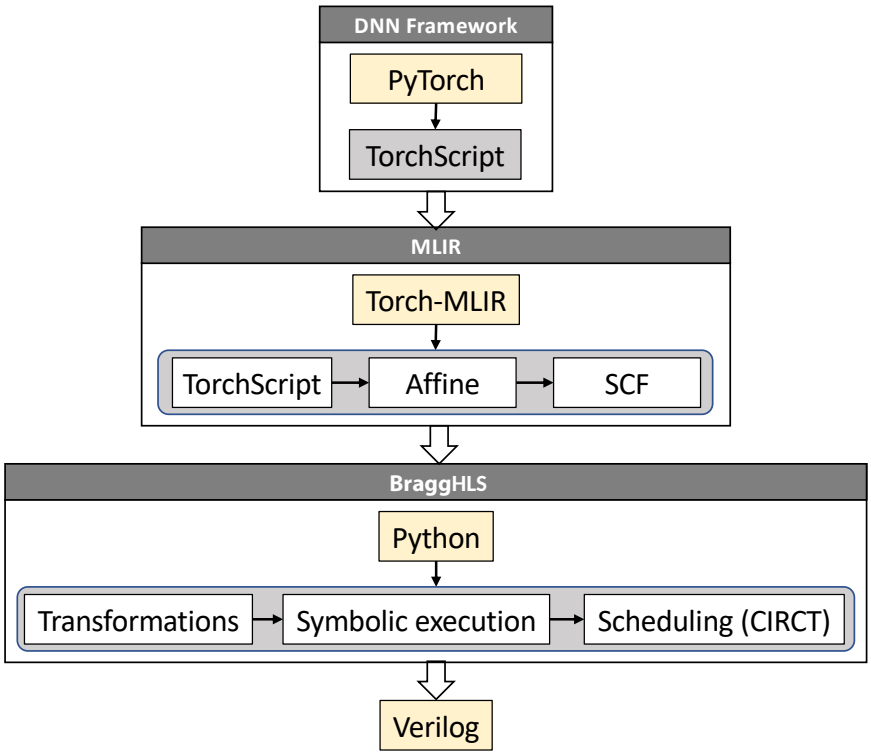


Figure 3.1: `BraggHLS` framework overview.

The benefits of translating `scf` dialect to Python are manifold: see Section 3.2.1. Ultimately, `BraggHLS` produces a representation of the DNN that is then fully scheduled by using the scheduling infrastructure in CIRCT (Oppermann et al., 2022) (an MLIR adjacent project). After scheduling, `BraggHLS` emits corresponding RTL (as Verilog).

`BraggHLS` delegates to the FloPoCo (de Dinechin, 2019) IP generator the task of generating pipelined implementations of the standard floating-point arithmetic operations (`mulf`, `divf`, `addf`, `subf`, `sqrtf`) at various precisions. In addition, we implement a few generic (parameterized by bit width) operators in order to support a broad range of DNN operations: two-operand maximum (`max`), unary negation (`neg`), and the rectified linear unit (`relu`). Transcendental functions, such as `exp`, are implemented by using a Taylor series expansion to k-th order (where k is determined on a case-by-case basis). Note that FloPoCo's

Listing 9 `scf` dialect loop representation of Listing 7.

```
@conv2d(
    %input: memref<b × c_in × h × w>,
    %weight: memref<b × c_out × h × w>,
    %output: memref<c_out × c_in × k × k>
) {
  scf.for %i1 = %c0 to b step %c1 {
    scf.for %i2 = %c0 to c_out step %c1 {
      scf.for %i3 = %c0 to h step %c1 {
        scf.for %i4 = %c0 to w step %c1 {
          scf.for %i5 = %c0 to c_in step %c1 {
            scf.for %i6 = %c0 to k step %c1 {
              scf.for %i7 = %c0 to k step %c1 {
                %3 = arith.addi %i3, %i6
                %4 = arith.addi %i4, %i7
                %5 = memref.load %input[
                  %i1, %i5, %i3, %3, %4]
                %6 = memref.load %weight[
                  %i2, %i5, %i6, %i7]
                %7 = memref.load %output[
                  %i1, %i2, %i3, %i4]
                %8 = arith.mulf %5, %6
                %9 = arith.addf %7, %8
                memref.store %9, %output[
                  %i1, %i2, %i3, %i4]
              }}}}}}}
  return %2
}
```

floating-point representation differs slightly from IEEE754, foregoing sub-normals and differently encoding zeroes, infinities and NaNs (for the benefit of reduced complexity) and our implementations `max`, `neg`, `relu` are adjusted appropriately.

We now discuss some aspects of `BraggHLS` in more detail.

3.2.1 Symbolic interpretation for fun and profit

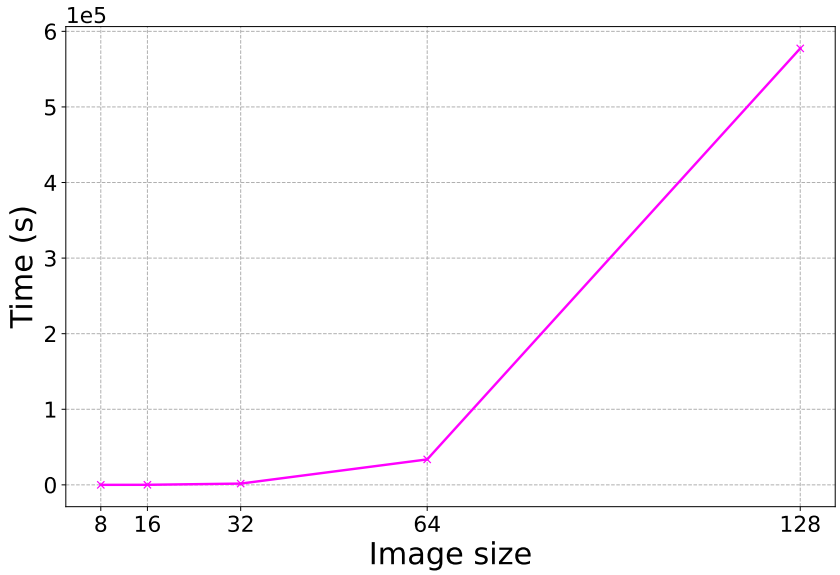


Figure 3.2: 3×3-kernel convolution (cf. Listing 9) full unrolling time vs. input (square) image size, with `store-load` forwarding using MLIR's `-affine-scalrep` pass. The longest time is 577,419 s (≈160 h) for a loop nest with a trip count of 128×128×3×3=147,456.

As noted in Section 3.1.2, maximizing concurrency and parallelism for a design entails unrolling loops and analyzing the data flow of their operations. As illustrated in Figure 3.2, the formally correct approach to unrolling loop nests can be prohibitively expensive in terms of runtime. In the case of `BraggNN` (see Listing 11), for example, the high cost of unrolling precluded effective search of the design space for a RTL representation achieving the target latency. Translating `scf` dialect to Python enables `BraggHLS` to overcome this barrier by enabling us to use the Python interpreter as a *symbolic interpreter*. Interpreting the resulting Python loop nests (i.e., running the Python program) while treating the arithmetic

and memory operations on SSA values as operations on symbols (i.e., Python classes with overloaded methods) enables us to:

1. Partially evaluate functions of iteration variables to determine array index operands of all stores and loads (for example, `memref.load %input[%i1,%i5,%i3,%3,%4]`) and thereupon perform memory dependence checks, thus transforming the problem of statically verifying memory dependence into one of checking assertions at runtime;

2. Unroll loops by recording each floating-point arithmetic operation executed while enforcing SSA; e.g., for a loop whose body has repeated assignments to the same SSA value (ostensibly violating SSA), we execute the loop and instantiate new, uniquely identified, symbols for the result of each operation;

3. Reconstruct all data flow through arithmetic operations and memory operations by interpreting `memref`s as *geometric symbol tables* (i.e., symbol tables indexed by array indices rather than identifiers/names) and `store`s and `load`s as reads and writes on those symbol tables;

4. Swap evaluation rules in order to support various functional modes, e.g., evaluating floating-point arithmetic operations by using (Python) bindings to FloPoCo's C++ functional models, thereby enabling behavioral verification of our designs.

See Table 3.3 for the translation rules from MLIR dialects to Python.

3.2.2 AST transformations and verification

Prior to interpretation, `BraggHLS` performs some simple AST transformations on the Python generated from `scf` dialect:

1. **Hoist globals**: Move fixed DNN tensors (i.e., weights) out of the body of the generated Python function (`BraggHLS` translates the MLIR `module` corresponding to the DNN

⟦MLIR⟧	Python
⟦%5⟧	`v5 = Val("%5")`
⟦memref<$b \times c_{in} \times h \times w$>⟧	`MemRef(`b`, `c_{in}`, `h`, `w`)`
⟦%5 = memref.load %input[%i1, %i5, %3, %4]⟧	⟦%5⟧ = ⟦%input⟧.__getitem__((⟦%i1⟧, ⟦%i5⟧, ⟦%3⟧, ⟦%4⟧))
⟦ memref.store %9, %output[%i1, %i5, %3, %4]⟧	⟦%output⟧.__getitem__((⟦%i1⟧, ⟦%i5⟧, ⟦%3⟧, ⟦%4⟧), ⟦%9⟧)
⟦ scf.for %i1 = %c0 to b step %c1⟧	for ⟦%i1⟧ in range(⟦%c0⟧, b, ⟦%c1⟧)
⟦%3 = arith.addi %i3, %i6⟧	⟦%3⟧ = ⟦%i3⟧ + ⟦%i6⟧
⟦%8 = arith.mulf %5, %6⟧	⟦%8⟧ = ⟦%5⟧.__mul__(⟦%6⟧)
⟦%9 = arith.addf %7, %8⟧	⟦%9⟧ = ⟦%7⟧.__add__(⟦%8⟧)
$\dfrac{\text{⟦\%63 = arith.cmpfugt \%10, \%c0⟧} \quad \wedge \quad \text{⟦\%64 = arith.select \%63, \%10, \%c0⟧}}{\text{⟦\%64⟧.relu(⟦\%10⟧)}}$	
$\dfrac{\text{⟦\%8 = arith.mulf \%5, \%6⟧} \quad \wedge \quad \text{⟦\%9 = arith.addf \%7, \%8⟧}}{\text{⟦\%9⟧ = fma(⟦\%5⟧, ⟦\%6⟧, ⟦\%7⟧)}}$	

Figure 3.3: Translation rules for mapping `scf`, `arith`, and `memref` dialects to Python.

into a single Python function in order to simplify analysis and interpretation) and into the parameter list, for the purpose of ultimately exposing them at the RTL module interface.

2. **Remove `if` expressions**: DNN `relu` operations are lowered to the `scf` dialect as a decomposition into `arith.cmpfugt` and `arith.select`; this transformation recomposes them into a `relu`.

3. **Remove MACs**: Schedule sequences of `load-multiply-add-store` (common in DNN implementations) jointly, coalescing them into a single `fmac` operation.

4. **Reduce `for`s**: Implement the reduction tree structure for non-parallelizable loop nests mentioned in Section 3.2.3.

These transformations on the Python AST are simple (implemented with procedural pattern matching), extensible, and efficient (marginal runtime cost) because no effort is made to verify their formal correctness. Thus, `BraggHLS` trades formal correctness for development

time performance. This tradeoff enables quick design space iteration, which for example, enabled us to achieve low latency implementations for `BraggNN` (see Section 3.3.2).

`BraggHLS` supports behavioral rather than formal verification. Specifically, `BraggHLS` can generate test-benches for all synthesized RTL. The test vectors for these test-benches are generated by evaluating the generated Python representation of the DNN on randomly generated inputs but with floating-point operations now evaluated using functional models of the corresponding FloPoCo operators. The test-benches can then be run using any IEEE 1364 compliant simulator. We run a battery of such test-benches (corresponding to various DNN operation types), using `cocotb` (Rosser, 2018) and `iverilog` (Williams), as a part of our continuous integration (CI) process.

3.2.3 Scheduling

Recall that HLS must schedule operations during each clock cycle in a way that preserves the DNN's data-flow graph. That schedule then informs the construction of a corresponding FSM. As already mentioned, scheduling an arbitrary DNN involves formulating and solving an ILP. In the resource-unconstrained case, due to the precedence relations induced by data flow, the constraint matrix of the associated ILP is a *totally unimodular matrix* and the feasible region of the problem is an integral polyhedron. In such cases, the scheduling problem can be solved optimally in polynomial time with a LP solver (Oppermann, 2019). In the resource-constrained case, resource constraints can also be transformed into precedence constraints by picking a particular (possibly heuristic) linear ordering on the resource-constrained operations. This transformation partitions resource-constrained operations into distinct clock cycles, thereby guaranteeing sufficient resources are available for all operations scheduled within the same clock cycle (Dai et al., 2018).

`BraggHLS` uses the explicit parallelism of the `scf.parallel` loop-nest representation to inform such a linear ordering on resource-constrained operations. By assumption, for loop

nests which can be represented as `scf.parallel` loop nests (see Listing 10), each instance of a floating-point arithmetic operation in the body corresponding to unique values of the iteration variables (e.g., `%i1`, `%i2`, `%i3`, `%i4` for Listing 10) is independent of all other such instances, although data flow within a loop body must still be respected. This exactly determines total resource usage per loop nest; for example, the convolution in Listing 10 would bind to $2K_i$ DSPs (assuming `mulf`, `addf` bind to one DSP each), where:

$$\begin{aligned} K_i \;\coloneqq\; & |\{\texttt{\%i1} = \texttt{\%c0} + \texttt{\%c1} \times \mathbb{N} \wedge \texttt{\%i1} < b\}| \;\times \\ & |\{\texttt{\%i2} = \texttt{\%c0} + \texttt{\%c1} \times \mathbb{N} \wedge \texttt{\%i2} < c_{out}\}| \;\times \\ & |\{\texttt{\%i3} = \texttt{\%c0} + \texttt{\%c1} \times \mathbb{N} \wedge \texttt{\%i3} < h\}| \;\times \\ & |\{\texttt{\%i4} = \texttt{\%c0} + \texttt{\%c1} \times \mathbb{N} \wedge \texttt{\%i4} < w\}| \end{aligned}$$

with $\texttt{\%c1} \times \mathbb{N}$ representing all multiples of `%c1`. That is to say, K_i is the cardinality of the cartesian product of the iteration spaces of the parallel iteration variables.

Defining $K \coloneqq \max_i K_i$ across all `scf.parallel` loop nests, we can infer peak usage of any resource. Then, after indexing available hardware resources $j = 1, \ldots, K$, we can bind the operations of any particular loop nest. This leads to a linear ordering on resource-constrained operations such that operations bound to the same hardware resource index j must be ordered according to their execution order during symbolic interpretation.[2] Note that this ordering coincides with the higher-level structure of the DNN, which determines the ordering of `scf.parallel` loop nests (and thus interpretation order during execution of the Python program).

For DNN operations that lower to sequential loop nests rather than `scf.parallel` loop nests (e.g., `sum`, `max`, or `prod`), we fully unroll the loops and transform the resulting, sequential, operations into a reduction tree; we use As-Late-As-Possible scheduling (Baruch, 1996) amongst the subtrees of such reduction trees.

2. `BraggHLS` only needs to construct a partial precedence ordering $\texttt{op}_a < \texttt{op}_b$ for operations $\texttt{op}_a, \texttt{op}_b$ which CIRCT then combines with the delays of the operations to construct constraints such as $\texttt{start\_op}_a +$

Listing 10 Parallel loop representation of Listing 7, exhibiting explicitly the resource partitioning and ordering strategy we employ to construct a feasible schedule of operations.

```
@conv2d(
    %input: memref<b × c_in × h × w>,
    %weight: memref<b × c_out × h × w>,
    %output: memref<c_out × c_in × k × k>
) {
  scf.parallel (%i1, %i2, %i3, %i4) =
               (%c0, %c0, %c0, %c0) to
               (b, c_out, h, w) step
               (%c1, %c1, %c1, %c1) {
    scf.for %i5 = %c0 to c_in step %c1 {
      scf.for %i6 = %c0 to k step %c1 {
        scf.for %i7 = %c0 to k step %c1 {
          %3 = arith.addi %i3, %i6
          %4 = arith.addi %i4, %i7
          %5 = memref.load %input[%i1, %i5, %i3, %3, %4]
          %6 = memref.load %weight[%i2, %i5, %i6, %i7]
          %7 = memref.load %output[%i1, %i2, %i3, %i4]
          %8 = arith.mulf %5, %6
          %9 = arith.addf %7, %8
          memref.store %9, %output[%i1, %i2, %i3, %i4]
        }}}}
  return %2
}
```

3.3 Evaluation

We evaluate `BraggHLS` both on individual DNN layers, and end-to-end, on our use-case `BraggNN`. We compare `BraggHLS` to Xilinx's Vitis HLS by comparing the latencies and resource usages of the final designs generated by each. We also compare the runtimes of the tools themselves. Both `BraggHLS` and Vitis HLS produce Verilog RTL, on which we run a synthesis pass by using Xilinx's Vivado. The particular FPGA target is Xilinx Alveo U280. We measure LUT, DSP, BRAM, and FF usage. For the DNN layer evaluations, we use FloPoCo (5,11)-floating point representations (5-bit exponent, 11-bit mantissa), corresponding to Vitis HLS's IEEE half-precision IPs. We synthesize all designs for a 10 ns target clock period and report end-to-end latency as a product of the total schedule interval count of the design and achieved clock period (*10-WNS*, where *WNS* is the worst negative slack reported). In the case of Vitis HLS, which potentially explicitly pipelines the design and therefore implements with an initiation interval strictly less than the total schedule interval count, we report in terms of the best possible interval count (`LatencyBest` from the Vitis HLS reports). All other measurements are collected from Vivado synthesis reports. As Vitis HLS operates on C++ representations, we generate such a representation for our test cases by first lowering each DNN layer to the `affine` dialect and then applying the `scalehls-translate` tool of the ScaleHLS project (Ye et al., 2022) to emit C++. Importantly, we do not make any use of `scalehls-opt` optimization tool (of the same project).

Since our ultimate goal is low latency inference, and since the strategy that `BraggHLS` employs in the pursuit of this goal is loop unrolling, in order to produce a like for like comparison, we similarly unroll the representation that is passed to Vitis HLS. Thus, all Vitis HLS measurements are reported in terms of *unroll factor*: an unroll factor of k corresponds to a k-fold increase in the number of statements in the body of a loop and commensurate k-fold decrease in the trip count of the loop. For loop nests, we unroll inside out: if k is

$\texttt{delay}_a \leq \texttt{start\_op}_b$.

greater than the trip count t of the innermost loop, we unroll the innermost loop completely and then unroll the enclosing loop by a factor of $k - t$. We do not perform any store-load forwarding during this preprocessing but we annotate all arrays with the directive `array_partition complete dim=1` in order that Vitis HLS can effectively pipeline. All representations generated by `BraggHLS` correspond to full unrolling of the loop nests.

3.3.1 DNN layers

We evaluate `BraggHLS` vs. Xilinx's Vitis HLS by comparing the latency of the final design on five DNN layer types, chosen to cover a range of arithmetic operations (`mulf`, `divf`, `addf`, `subf`, `sqrtf`) and data access patterns (iteration, accumulation, reduction):

- `addmm(a, b, c)` : Matrix multiply: `a` $\times$ `b` + `c`;
- `batch_norm_2d(num_features)` : Batch normalization over a 4D input (Ioffe and Szegedy, 2015);
- `conv_2d(`c_{in}`,` c_{out}`,` k`)` : 2D convolution with bias, with $k \times k$ kernel, over a $b \times c_{in} \times h \times w$ input, producing $b \times c_{out} \times h' \times w'$ output;
- `max_pool_2d(`k`, stride)` : 2D max pooling, with $k \times k$ kernel, and striding;
- `soft_max` : $\text{softmax}(\boldsymbol{x}) \coloneqq \left[\frac{\exp(x_i)}{\sum_j \exp\left(x_j\right)}\right]$

The parameter values and input dimensions used during evaluation are summarized in Table 3.1.

Figure 3.4 shows Vitis HLS vs. `BraggHLS` resource usage and latency vs. unroll factor and Figure 3.5 shows the runtimes of Vitis HLS as function of increasing unroll factor. We observe that while Vitis HLS end-to-end latencies decrease with increased unroll factor, they never match that achieved by `BraggHLS`. Even at an unroll factor of 1024 (which corresponds to fully unrolled for all loop nests comprising these layer types), Vitis HLS is only within

Table 3.1: DNN layers used for evaluation of `BraggHLS`.

Layer	Parameter values	Input dimensions
`addmm`	N/A	`a`, `b`, `c` : $(16, 16)$
`batch_norm_2d`	`num_features` $= 2$	`input` : $(10, 2, 3, 3)$
`conv_2d`	$c_{in} = 1, c_{out} = k = 3$	`input` : $(1, 1, 16, 16)$
`max_pool_2d`	$k = 3$, `stride` $= 2$	`input` : $(1, 3, 16, 16)$
`soft_max`	N/A	`input` : $(1, 3, 16, 16)$

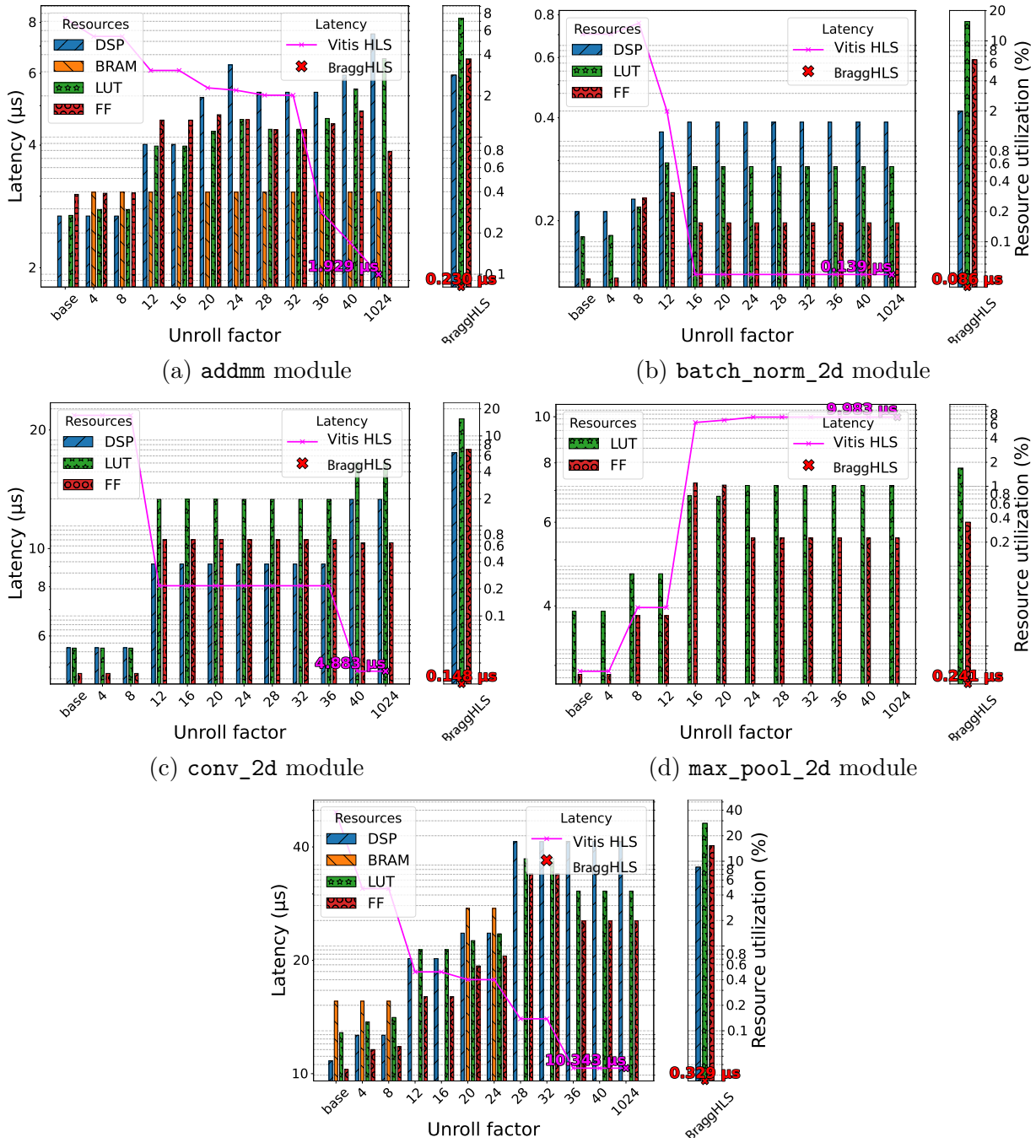
10× of `BraggHLS`. We attribute this to Vitis HLS's inability to pipeline effectively, due to its inability to eliminate memory dependencies, either through `store-load` forwarding or further array partitioning. Conversely, `BraggHLS`'s ability to effectively perform `store-load` forwarding is evident in the complete lack of BRAM usage: all weights are kept on FFs or LUTs. While infeasible for larger designs (which would be constrained by the number of available FFs), this unconstrained usage of FFs is acceptable for our use case. The increasing latency (as a function of unroll factor) in the `max_pool_2d` case is due to Vitis HLS's failure to meet timing, i.e., while the interval count decreases as a function of unroll factor, the clock period increases.

3.3.2 `BraggNN` case study

High-energy diffraction microscopy enables non-destructive characterization for a broad class of single-crystal and polycrystalline materials. A critical step in a typical HEDM experiment is an analysis to determine precise Bragg diffraction peak characteristics. Peak characteristics are typically computed by fitting the peaks to a probability distribution, e.g., Gaussian, Lorentzian, Voigt, or Pseudo-Voigt. As noted, HEDM experiments can collect data at more than 80 GB/s. These data rates, though more modest than at the LHC, merit exploring low latency approaches in order to enable experiment modalities that depend on measurement-based feedback (i.e., experiment steering).

Listing 11 `BraggNN` model architecture for scaling factors *s=1,2*.

```
BraggNN(s)(
  (cnn_layers_1): Conv2d(s × 16, kernel=3, stride=1)
  (nlb): NLB(
    (theta_layer): Conv2d(s × 16, s × 8, kernel=1, stride=1)
    (phi_layer): Conv2d(s × 16, s × 8, kernel=1, stride=1)
    (g_layer): Conv2d(s × 16, s × 8, kernel=1, stride=1)
    (out_cnn): Conv2d(s × 8, s × 16, kernel=1, stride=1)
    (soft): Softmax()
  )
  (cnn_layers_2): Sequential(
    (0): ReLU()
    (1): Conv2d(s × 16, s × 8, kernel=3, stride=1)
    (2): ReLU()
    (3): Conv2d(s × 8, s × 2, kernel=3, stride=1)
    (4): ReLU()
  )
  (dense_layers): Sequential(
    (0): Linear(in_features=s × 50, out_features=s × 16)
    (1): ReLU()
    (2): Linear(in_features=s × 16, out_features=s × 8)
    (3): ReLU()
    (4): Linear(in_features=s × 8, out_features=s × 4)
    (5): ReLU()
    (6): Linear(in_features=s × 4, out_features=2)
    (7): ReLU()
  )
)
```



(a) `addmm` module

(b) `batch_norm_2d` module

(c) `conv_2d` module

(d) `max_pool_2d` module

(e) `soft_max` module

Figure 3.4: Vitis HLS vs. `BraggHLS` resource usage and latency vs. unroll factor for five DNN modules, exhibiting the large runtime cost incurred in using Vitis HLS to search the design space (of possible low-latency designs for each layer). The lines give latencies (left axes); the bars give the % of the resource used (right axes). All y-scales are log.

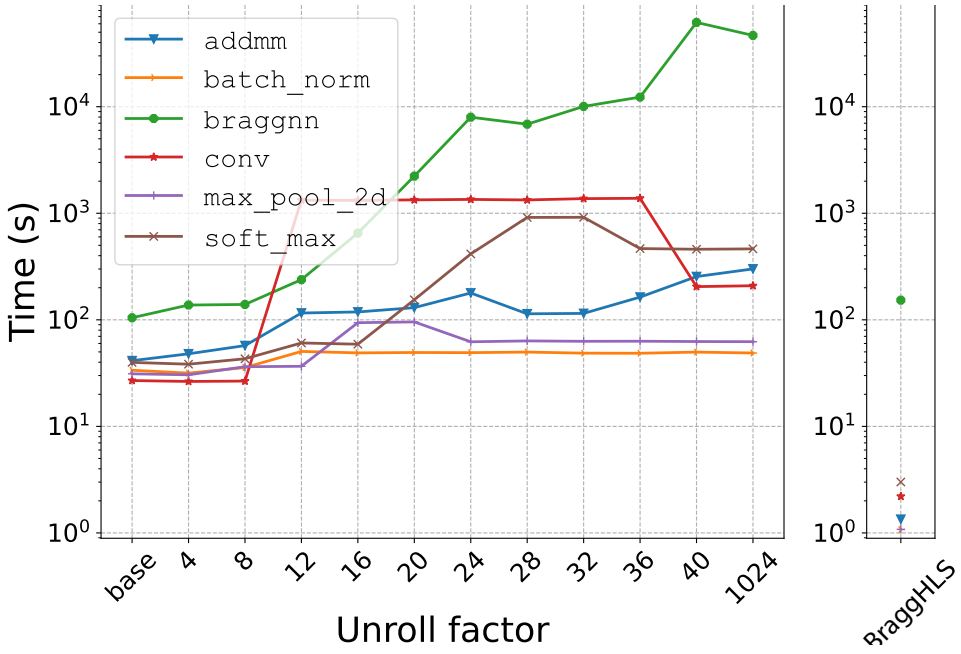


Figure 3.5: Vitis HLS vs. `BraggHLS` runtime vs. unroll factor, illustrating the large runtime cost incurred in using Vitis HLS to search over possible low-latency `BraggNN` designs.

`BraggNN` (Liu et al., 2022b), a DNN aimed at efficiently characterizing Bragg diffraction peaks, achieves a throughput (via batch inference) of approximately 22 µs/sample on a state-of-the-art GPU: a large speedup over classical pseudo-Voigt peak fitting methods, but still far short of the 1 µs/sample needed to handle 1 MHz sampling rates. In addition, the data-center class GPU such as a NVIDIA V100 (or even a workstation class GPU such as a NVIDIA RTX 2080Ti) required to run the current `BraggNN` implementation cannot be deployed at the edge, i.e., adjacent or proximal to the high energy microscopy equipment. With the goal of reducing both per-sample time and deployment footprint, we applied `BraggHLS` to the PyTorch representation of `BraggNN(`s`=1)` (see Listing 11) and achieved a RTL implementation which synthesizes to a 1238 interval count design that places, routes, and meets timing closure for a clock period of 10 ns (for a Xilinx Alveo U280). The design consists of a three stage pipeline with the longest stage measuring 480 intervals, for a throughput of 4.8 µs/sample. See Figure 3.6 for a comparison with designs generated by Vitis HLS (using the same flow as in 3.3).

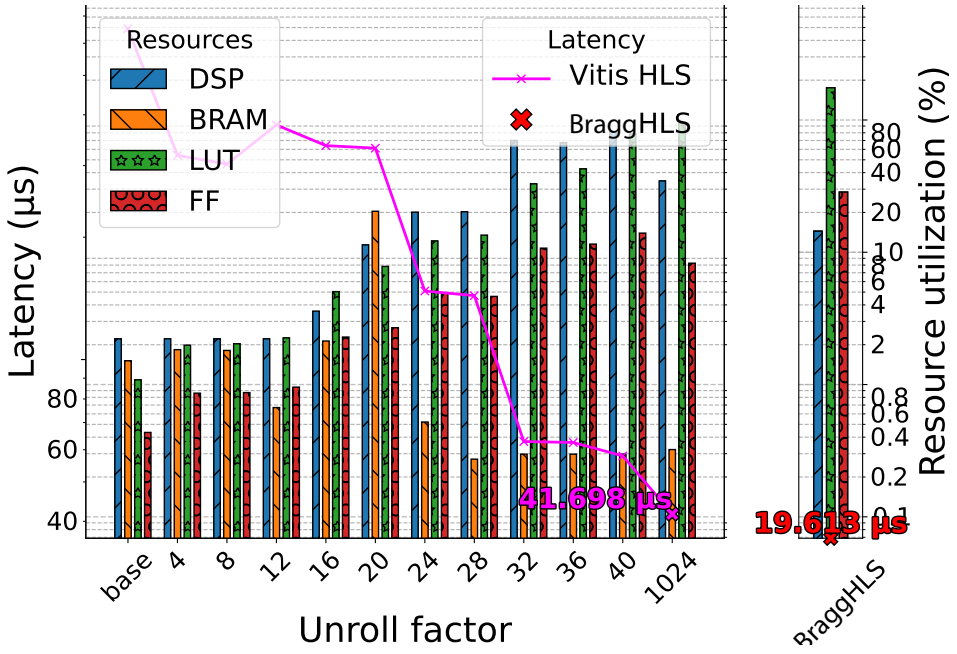


Figure 3.6: `BraggNN` Vitis HLS vs. `BraggHLS` resource usage and latency vs. unroll factor (with both at half-precision) throughout the design space of possible low-latency designs.

The most challenging aspect of implementing `BraggNN` was minimizing latency while satisfying compute resource constraints (LUTs, DSPs, BRAMs) and achieving routing closure, i.e., not exceeding available routing resources and avoiding congestion. We made two design choices to reduce resource consumption. The first was to reduce the precision used for the floating-point operations, from half precision to FloPoCo (5,4)-precision (5-bit exponent, 4-bit mantissa), a choice justified by examination of the distribution of the weights of the fully trained `BraggNN` (see Figure 3.7).

Reducing the precision enabled the second design choice, to eliminate BRAMs from the design, since, at the lower precision, all weights can be represented as registered constants. The reduced precision also drove the Vivado synthesizer to infer implementations of the floating-point operations that make no use of DSPs, likely because the DSP48 hardware block includes a 18-bit by 25-bit signed multiplier and a 48-bit adder (gui, 2021), neither of which neatly divides the bit width of FloPoCo (5,4)-precision cores. (The actual width for FloPoCo (5,4)-precision is 12 bits: 1 extra bit is needed for the sign and 2 for handling of

exceptional conditions.)

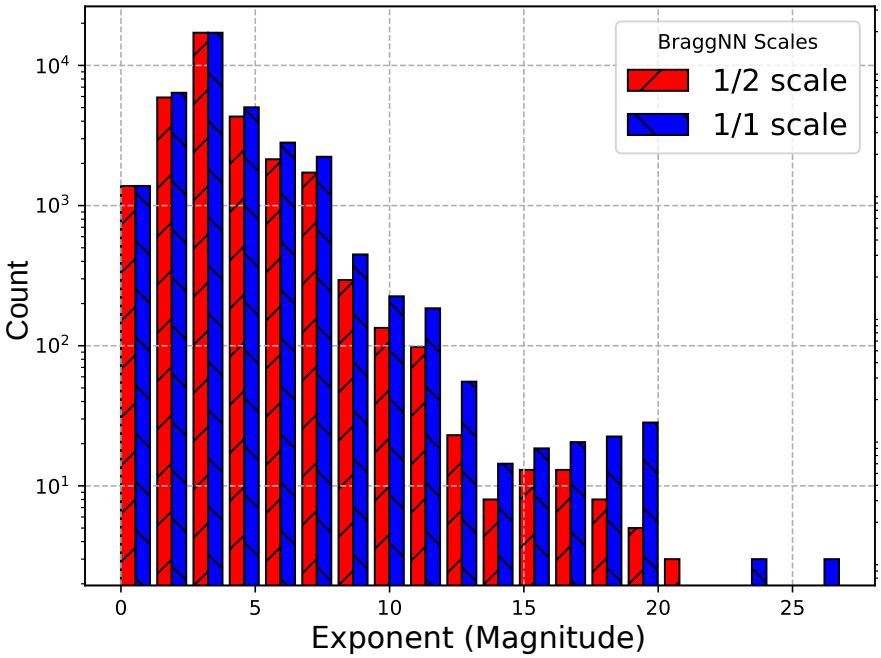
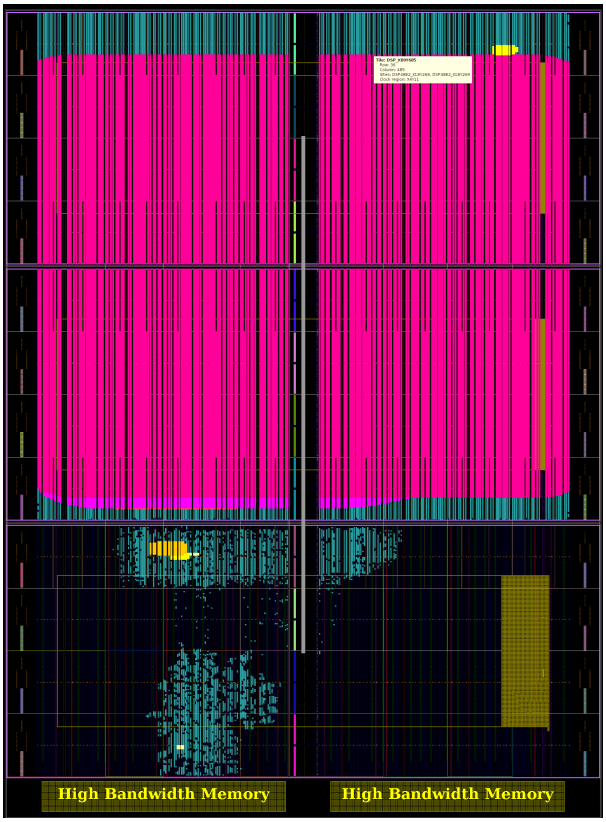


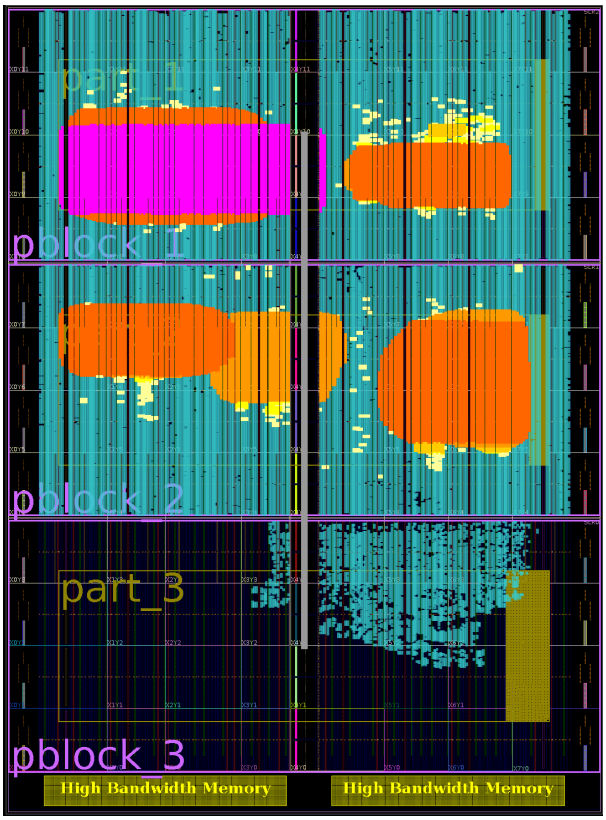
Figure 3.7: `BraggHLS` weights exponent distribution, illustrating the narrow distribution of observed weight exponents thereby justifying reduced precision.

Achieving routing closure was difficult due to the nature of the Xilinx's UltraScale architecture, of which the Alveo U280 is an instance. The UltraScale architecture achieves its scale through Stacked Silicon Interconnect (SSI) technology (Leibson et al., 2013), which implies multiple distinct FPGA dies, called Super Logic Regions (SLRs), on the same chip, connected by interposers. Adjacent SLRs communicate with each other over a limited set of Super Long Lines (SLLs), which determine the maximum bus width that spans two SLRs. On the Alveo U280 there are exactly 23,040 SLLs available between adjacent SLRs and at (5,4)-precision `BraggNN`(s=1) needs 23,328 SLLs between SLR2 and SLR1. [We route from SLR2 to SLR1 the outputs of `cnn_layers_1` (1×16×9×9×12 wires) and `soft(theta_layer`×`phi_layer)`×`g_layer` (1×8×9×9×12 wires).] Thus, we further reduced the precision to (5,3). Finally, since multiple dies constitute independent clock domains, the SLLs that cross SLRs are sensitive to hold time violations due to the higher multi-die variability (rap). This multi-die variability leads to high congestion if not addressed. Thus, routing across SLRs

needs to be handled manually, using placement and routing constraints for logic in each SLR and the addition of so-called "launch" and "latch" registers in each SLR. Figure 3.8 illustrates the effect of using launch and latch registers as well as placement and routing constraints.



(a) `BraggNN` fails to achieve routing closure without placement and routing constraints and launch and latch registers.



(b) `BraggNN` achieves routing closure with use of per SLR placement and routing constraints (`pblock_1`, `pblock_2`, `pblock_3`) and launch and latch registers (not highlighted).

Figure 3.8: Congestion maps for `BraggNN` on a Xilinx Alveo U280. Magenta indicates areas of high congestion.

Thus, these design choices (in combination with compiler level optimizations performed by `BraggHLS`) plus careful management of routing constraints enable us to lower, compile, synthesize, place, and route `BraggNN`(s=1) to Xilinx's Alveo U280 at a throughput of 4.8 µs/sample: ~5× higher latency than the target 1 µs/sample, but a ~4× improvement over the PyTorch GPU implementation.

3.4 Related work

Several projects aim to support translation from high-level representations of DNNs to feasible FPGA designs. Typically, they rely on commercial HLS tools for the scheduling, binding, and RTL emission phases of the translation, such as in the cases of DaCeML (Rausch et al., 2022), hls4ml (Duarte et al., 2018), and ScaleHLS (Ye et al., 2022), which all rely on Xilinx's Vitis HLS. Thus, they fail to efficiently (i.e., without incurring the aforementioned runtime costs) produce feasible and low-latency designs. One notable recent work is the SODA Synthesizer (Bohm Agostini et al., 2022), which does not rely on a commercial tool but instead relies on the open-source PandA-Bambu HLS tool (Ferrandi et al., 2021); though open-source and mature, we found in our own tests that PandA-Bambu also could not handle fully unrolled designs efficiently.

Alternatively, some projects do not rely on HLS for scheduling, binding, and RTL emission, and also attempt to translate from high-level representations of DNNs to feasible FPGA designs, such as DNN Weaver (Sharma et al., 2016) and NNGen (Takamaeda-Yamazaki, 2015). Both of the cited projects function as parameterized/templatized RTL generators and thus lack sufficient generality for our needs; primarily they seek to produce implementations of kernels that emulate GPU architectures (i.e., optimizing for throughput rather than latency). In our experiments they were unable to generate low-latency implementations, either by achieving unacceptable latencies or by simply failing outright. (NNGen, due to the nature of templates, supports only limited composition, and produced "recursion" errors.)

3.5 Conclusion

We have presented `BraggHLS`, an open-source MLIR-based HLS compilation framework that supports translating DNN models to RTL without the use of commercial HLS tools. The `BraggHLS` end-to-end compilation pipeline provides a PyTorch front-end and Verilog emission

back-end. An extensible Python intermediate layer supports use-case-specific optimizations (e.g., `store-load` forwarding) that are not possible otherwise. Experimental results demonstrate that `BraggHLS` outperforms, in terms of end-to-end latency, Vitis HLS on a range of DNN layer types and on a real-world Bragg peak detection DNN.

Future work in this area includes several directions:

- **Framework Integration**: Better integration between the Python layer and MLIR: it is preferable that the transformations on the Python representation could make use of various MLIR facilities, such as affine analysis, for the purposes of exploring loop transformations that improve latency;

- **Scheduling**: Expanding the set of scheduling algorithms available: for example, resource aware scheduling (Dai et al., 2018); Integration of scheduling-aware placement and vice-versa (placement-aware scheduling): currently `BraggHLS` can be used to inform placement but does not explicitly emit placement constraints (see Section 3.3.2); a more precise approach, such as in (Guo et al., 2021), would potentially enable better pipelining and thus higher throughput.

CHAPTER 4

`NELLI`: A LIGHTWEIGHT FRONTEND FOR MLIR

MLIR is a modular and extensible compiler infrastructure (Lattner et al., 2020b) for progressively transforming (*lowering*) programs from high-level (in terms of abstraction), architecture-independent representations to low-level, architecture-specific representations. Such Intermediate Representations (IRs) are termed *dialects* in the MLIR context in order to emphasize their mutual compatibility and the unified interface that MLIR provides for transforming between them, a process referred to as *running passes*.

MLIR has been applied to various problem domains and in its default distribution (other, so-called "out-of-tree," implementations exist) supports representing programs ranging in abstraction level from dataflow compute graphs, such as can be used to represent Deep Neural Networks (DNNs), to architecture-specific vector instructions. Other less quotidian applications of MLIR include modeling gate-level primitives (Eldridge et al., 2021), quantum assembly languages (McCaskey and Nguyen, 2021), and database query languages (Blockhaus and Broneske, 2022, Jungmair et al., 2022). By virtue of its close connection to LLVM (Lattner and Adve, 2004), MLIR supports code generation for CPUs, GPUs, and other compute platforms, including abstract runtimes (as opposed to concrete hardware architectures) such as OpenMP.

While the primary value of MLIR is its support for efficient (quick) construction of IRs modeling novel domains, an undeniable secondary value is the ability to use existing dialects, corresponding to established programming models, in combination with novel transformations tailored to problem-specific hardware configurations. For example, while there has been much research on the use of MLIR to lower DNNs to high-performance CPU and GPU platforms (Vasilache et al., 2022), such as data-center class devices and high-powered mobile devices (e.g., expensive mobile phones), there is a dearth of work on efficiently targeting low-power edge devices, such as micro-controllers and single-board computers. Yet those

latter edge devices, while relatively underpowered, can be an attractive DNN deployment target in instances where power is a scarce commodity, such as IoT, AgTech, and urban infrastructure monitoring. Indeed, it is conceivable that, given sufficient directed design space exploration (such as can be realized by using MLIR), these low-power edge devices could effectively support edge inference of DNNs.

However, while MLIR provides the edge device software architect with a rich existing repository of useful dialects and transformations, the effective use of those capabilities for edge device programming is hindered by the lack of a point of ingress to MLIR capabilities that is not encumbered by assumptions about the roles of the existing dialects and their mutual relationships. Specifically, almost all extant ingress points take the form of high-level DNN frameworks, such as PyTorch (Paszke et al., 2017), TensorFlow (Abadi et al., 2016b), or ONNX (Jin et al., 2020)—but most optimization actually occurs on lower-level dialects, such as the affine, structured control-flow, and vector dialects. Thus, in order to productively investigate possible optimization opportunities one must distinguish artifacts of the lowering process from the kernel representations themselves. For example, consider investigating the optimization of a (32×32) linear layer (i.e., `torch.nn.Linear(32, 32)`). This ubiquitous DNN operation lowers to the loop nests in Listing 16; note that the third loop nest is readily identified as corresponding directly to a matrix-multiplication kernel, but the other three are somewhat mysterious[1]. Thus, in longer programs (complete DNNs) it becomes difficult to identify, isolate, and manipulate (e.g., to optimize) IR corresponding most closely to the compute kernel itself, amongst IR that reflects certain assumptions/contracts. Conversely, there currently exists no simple and efficient way to emit any of the lower-level dialects in MLIR (such as `scf`, `affine`, `memref`, or `vector`) short of writing the IR "by hand."

In order to address the problem of MLIR's lower-level dialects being inaccessible, we

1. Case in point: the seemingly redundant initialization and subsequent copy into an intermediate buffer in the lowering of `torch.nn.Linear(32, 32)` is the result of `torch-mlir` enforcing value semantics (Smith, 2002) on `torch.tensor`s, which, while important, obscures the actual compute kernel.

Listing 12 `nelli` mapping between Python's `if` and MLIR's `scf` dialect.

```
@mlir_func
def ifs(M: F64, N: F64):
    one = 1.0
    if M < N:
        two = constant(2.0)
        mem = MemRef.alloca([3, 3], F64)
    else:
        six = constant(6.0)
        mem = MemRef.alloca([7, 7], F64)
  return
```

```
func.func @ifs(%M: f64, %N: f64) {
  %one = arith.constant 1.000000e+00 : f64
  %cond = arith.cmpf olt, %arg0, %arg1 : f64
  scf.if %cond {
    %two = arith.constant 2.000000e+00 : f64
    %mem = memref.alloca() : memref<3x3xf64>
  } else {
    %six = arith.constant 6.000000e+00 : f64
    %mem = memref.alloca() : memref<7x7xf64>
  }
  return
}
```

Listing 13 `nelli` mapping between Python's `for` and MLIR's `affine` dialect.

```
M, N, K = 4, 16, 8

@mlir_func
def matmul(
    A: MemRef[(M, N), F32],
    B: MemRef[(N, K), F32],
    C: MemRef[(M, K), F32]
):
    for i in range(M):
        for j in range(N):
            for k in range(K):
                a = A[i, j]
                b = B[j, k]
                c = C[i, k]
                d = a * b
                e = c + d
                C[i, k] = e
```

```
func.func @matmul(
    %A: memref<4x16xf32>,
    %B: memref<16x8xf32>,
    %C: memref<4x8xf32>
) {
  affine.for %i = 0 to 4 {
    affine.for %j = 0 to 16 {
      affine.for %k = 0 to 8 {
        %a = memref.load %A[%i, %j]
        %b = memref.load %B[%j, %k]
        %c = memref.load %C[%i, %k]
        %d = arith.mulf %a, %b : f32
        %e = arith.addf %c, %d : f32
        memref.store %e, %C[%i, %k]
      }
    }
  }
  return
}
```

present `nelli`[2], a lightweight frontend for MLIR. This Python embedded domain-specific language (eDSL) builds on top of existing MLIR Python bindings to map Python primitives (such as `if`s, `for`s, and `class`es) to various MLIR dialects. Our foremost goal in designing `nelli` was to make MLIR more ergonomic and thereby more accessible. To this end, we make `nelli` "Pythonic" while preserving MLIR semantics vis-a-vis the in-tree Python bindings. See Listings 12 and 13 for some examples of `nelli` syntax. Notably, `nelli` captures program control flow and produces fully typed IR with little static analysis on the Python source (hence, *lightweight*). Additionally, since `nelli` is a Python eDSL, it fully interoperates with existing Python tooling (IDEs, debuggers,etc.) and other elements of the Python ecosystem.

In the following, we discuss in greater detail `nelli` design goals, the eDSL implementation approaches that we investigated, and the implementation details of our chosen approach. We also present three use cases: 1) a kernel tuner that uses a black-box, gradient-free, optimizer (Rapin and Teytaud, 2018), demonstrating the power and convenience of Python interoperability; 2) a pipeline for lowering kernels to target GPUs and then evaluating performance on a Raspberry Pi edge device, demonstrating ease of integration with LLVM, downstream of MLIR; 3) and a pipeline for translating parallelizable kernels to OpenMP programs. In summary, this work makes the following contributions:

1. A thorough discussion of several alternative eDSL implementation approaches (in Python) and their relative merits and deficiencies.

2. A discussion of the design and implementation of an embedded domain-specific language (`nelli`) with minimal static (ahead-of-time) analysis and complexity;

3. Implements several lowerings that demonstrate capabilities of `nelli`, with a focus on deploying compute intensive kernels to diverse hardware platforms.

The remainder of the chapter is structured as follows: Section 4.1 reviews the relevant back-

2. `https://github.com/makslevental/nelli`

ground on eDSLs and MLIR; Section 4.2 discusses the implementation of `nelli`; Section 4.3 demonstrates the capabilities of `nelli`; and, finally, Section 4.4 compares `nelli` to similar tools.

4.1 Background

We quickly review the necessary background on MLIR, in particular with respect to DNN deployment to edge devices, and eDSL construction in general.

4.1.1 MLIR

MLIR is an approach to building reusable and extensible compiler infrastructure. Practically this means that MLIR constitutes a collection of utilities for

1. Defining mutually compatible IRs, known as *dialects*, that model programs in particular domains, supporting operations (including attributes) and types (including traits);
    - Using the Operation Definition Specification (ODS) language implemented against LLVM's TableGen[3] utility;
2. Defining intra-dialect transformations, such as canonicalization, inlining, and dead-code elimination;
    - Using a subgraph matching[4] and rewriting concept known as a `RewritePattern`;
3. Defining inter-dialect transformations, known as *conversions;*
    - Using `ConversionPattern`s and `TypeConverter`s.

In addition to a thriving ecosystem of dialects, tools, and down-stream projects, MLIR has

3. MLIR is an "in-tree" LLVM project and thus reuses and extends many of LLVM's existing facilities.

4. Directed, acyclic, graph matching is strictly more powerful than tree matching (Ebner et al., 2008).

many "in-tree" dialects that model programs across the abstraction-level spectrum. It also supports target-specific code generation and runtime execution through the various backends provided by LLVM; this includes both `x86_64` and `aarch64`/`arm64` CPU instruction set architectures (ISAs), NVPTX[5] and SPIR-V[6] GPU pseudo-ISAs, as well as minimal runtimes for each. See Figure 4.1 for the "ladder of abstraction" (Hayakawa, 1948) in terms of MLIR dialects. We briefly describe a few of the dialects (in-tree and out-of-tree) relevant for DNN deployment to edge devices.

High-level dialects

At the highest level of abstraction, MLIR supports representing DNNs specified using high-level frameworks such as TensorFlow and PyTorch[7]. The role of these dialects (`tf`, `tfl`, `torch`) is to faithfully represent the source model as specified in the chosen framework and thus they function as points of ingress into MLIR. As mentioned in the introduction, this effectively makes TensorFlow and PyTorch the only mature points of ingress into MLIR (see Section 4.4). In addition, as evinced by Listing 14, the lowering/translation process incurs a high cost with respect to legibility; naturally, lowering the level of abstraction necessitates the inclusion of explicit specification of operations that are implicit (or, at least, taken for granted) in the high-level representation. With Listings 14 in mind, note that in specifying the operation `torch.nn.Linear(32, 32)`, one implicitly specifies:

1. The bias tensor `%0 = torch.vtensor.literal` needs to be initialized;

5. NVIDIA Parallel Thread Execution is a virtual machine instruction set architecture used by NVIDIA's GPUs as an interface layer between CUDA and SASS; SASS is the low-level assembly language that compiles to binary microcode, which executes natively on NVIDIA GPU hardware (nvi, 2015).

6. Khronos Group's binary intermediate language SPIR-V for representing graphics shaders and compute kernels (Kessenich et al., 2018); both the Vulkan graphics API and the OpenCL compute API support SPIR-V.

7. The `tf` (TensorFlow), `tfl` (TensorFlowLite), `torch` (PyTorch), and `mhlo` dialects are all "out-of-tree" dialects.

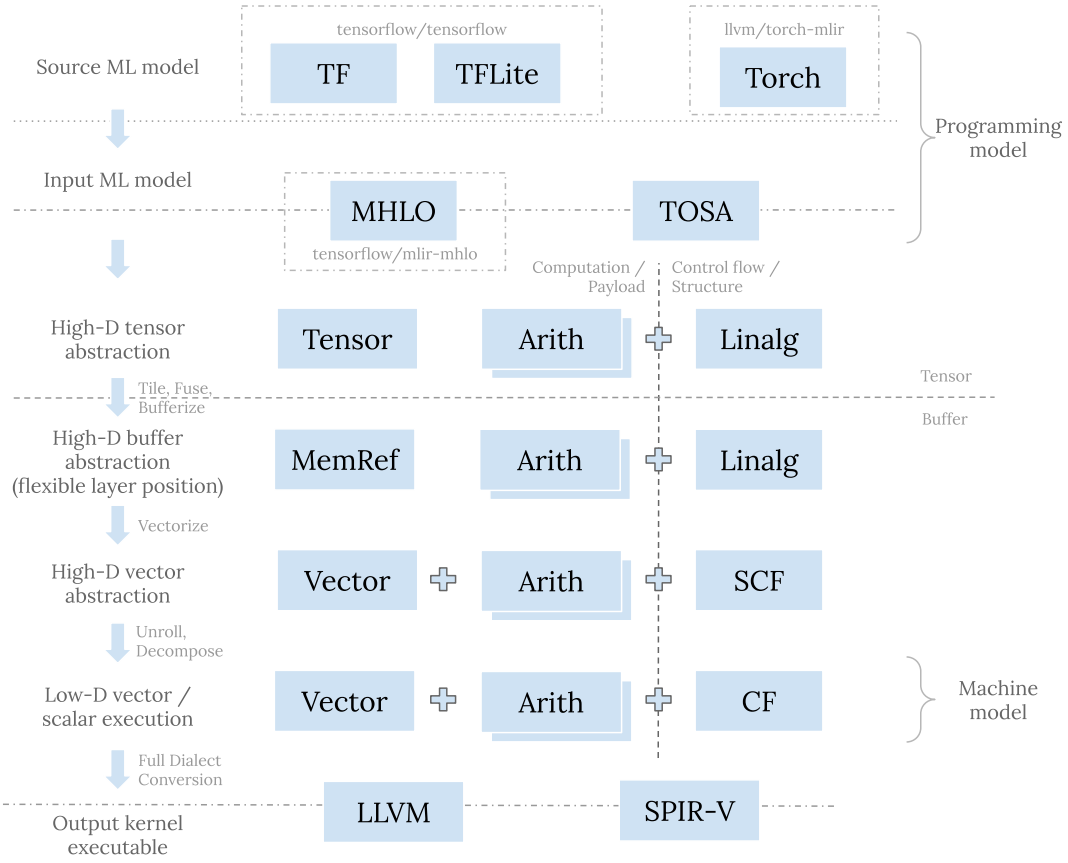


Figure 4.1: Ladder of dialect abstraction in terms of dialect types and dialect operations (reprinted with permission from (Lei Zhang, 2022)); with respect to types, a progressive lowering needs representations for tensors, buffers, vectors, and scalars, while with respect to operations, it needs to support computation/payload (i.e., arithmetic) and control flow.

Listing 14 Simple neural network with a `torch.nn.Linear(32, 32)` layer rendered in the `torch` dialect.

```
class MyMatmul(nn.Module):
  def __init__(self):
    super().__init__()
      self.matmul = nn.Linear(
        32, 32
      )

  def forward(self, x):
    return self.matmul(x)
```

```
module attributes {
func.func @forward(
    %arg0: !torch.vtensor<[32,32],f32>
    ) -> !torch.vtensor<[32,32],f32> {
  %int0 = torch.constant.int 0
  %int1 = torch.constant.int 1
  %float1.0e00 = torch.constant.float 1.0
  %0 = torch.vtensor.literal(dense<1.0>)
  %1 = torch.vtensor.literal(dense<1.0>)
  %2 = torch.aten.transpose.int %1, %int0, %int1
  %3 = torch.aten.mm %arg0, %2
  %4 = torch.aten.add.Tensor %3, %0, %float1.0e00
  return %4 : !torch.vtensor<[32,32],f32>
}
}
```

2. The weight tensor `%1 = torch.vtensor.literal` needs to be initialized;

3. A transpose `%2 = torch.aten.transpose.int %1, %int0, %int1` on the weight tensor needs to be performed (since, in PyTorch, weights are stored in column-order);

4. The bias needs to be added `%4 = torch.aten.add.Tensor %3, %0, %float1.0e00` to the result of the matrix multiplication (`%3 = torch.aten.mm %arg0, %2`).

The ultimate effect of this translation process is that targeting the operation of interest (e.g., `torch.nn.Linear(32, 32)`) for investigation and transformation is made more difficult. It's important to not underestimate the significance of the last point: subgraph matching, as implemented in MLIR by the `RewritePattern`, "anchors" on a target operation. Thus, if an optimizing transformation (such as loop-unrolling, loop-fusion, loop-tiling) is implemented targeting the loop nests generated from this high-level representation (see Listing 16), then running that pass will incur high(er) runtime cost[8] and much higher devel-

8. Imagine targeting the third loop-nest in Listing 16; you might develop a `RewritePattern` that matches on `scf.for` but then there are $2 + 2 + 3 + 2 = 9$ possible such matches. Thus, one needs to further

Listing 15 `linalg.generic` representation for `torch.nn.Linear(32, 32)`.

```
#map3 = affine_map<(d0, d1, d2) -> (d0, d2)>
#map4 = affine_map<(d0, d1, d2) -> (d2, d1)>
#map5 = affine_map<(d0, d1, d2) -> (d0, d1)>
%3 = linalg.generic {
  indexing_maps = [#map3, #map4, #map5],
  iterator_types = ["parallel", "parallel", "reduction"]
} ins(%arg0, %1 : tensor<32x32xf32>, tensor<32x32xf32>)
  outs(%2 : tensor<32x32xf32>) {
^bb0(%in: f32, %in_2: f32, %out: f32):
  %5 = arith.mulf %in, %in_2 : f32
  %6 = arith.addf %out, %5 : f32
  linalg.yield %6 : f32
} -> tensor<32x32xf32>
```

opment time. In MLIR, the partial resolution to this problem is called *structured code generation* (Vasilache et al., 2022), i.e., high-level operations such as `torch.nn.Linear(32, 32)` are first lowered to a structured representation, such as `linalg.generic` (see Listing 15), which is itself transformed and lowered to optimized loop-nests (see Listing 16). But, as can be observed in Listing 15, these structured transformations are (currently) limited to kernels implemented in terms of `parallel` and `reduction` iterators.

Intermediate-level dialects

An intermediate-level dialect is one that can be used to represent a kernel explicitly but is abstract with respect to hardware implementation. Thus, the structured control flow dialect (`scf`), which models loops (`scf.for`, `scf.while`, `scf.parallel`), and the `memref` dialect, which is intended to model creation and manipulation of objects with memory reference semantics (i.e., buffers). See Listing 16 for the representation of `torch.nn.Linear(32, 32)` purely in terms of these dialects.

filter the possible matches (e.g., by filtering on whether the body contains a sequence of `arith.mulf` and `arith.addf`).

Listing 16 Loop-level representation for `torch.nn.Linear(32, 32)` through `torch-mlir`, `linalg`, and `scf`. The blue shading highlights the matrix-multiplication loop nest (lines 14-26) amidst artifacts of the lowering process.

```
1  %alloc = memref.alloc() {alignment = 64 : i64} : memref<32x32xf32>
2  scf.for %arg1 = %c0 to %c32 step %c1 {
3    scf.for %arg2 = %c0 to %c32 step %c1 {
4      memref.store %cst, %alloc[%arg1, %arg2] : memref<32x32xf32>
5    }
6  }
7  %alloc_0 = memref.alloc() {alignment = 64 : i64} : memref<32x32xf32>
8  scf.for %arg1 = %c0 to %c32 step %c1 {
9    scf.for %arg2 = %c0 to %c32 step %c1 {
10     %2 = memref.load %alloc[%arg1, %arg2] : memref<32x32xf32>
11     memref.store %2, %alloc_0[%arg1, %arg2] : memref<32x32xf32>
12   }
13 }
14 memref.dealloc %alloc : memref<32x32xf32>
15 scf.for %arg1 = %c0 to %c32 step %c1 {
16   scf.for %arg2 = %c0 to %c32 step %c1 {
17     scf.for %arg3 = %c0 to %c32 step %c1 {
18       %2 = memref.load %cast[%arg1, %arg3] : memref<32x32xf32>
19       %3 = memref.load %0[%arg3, %arg2] : memref<32x32xf32>
20       %4 = memref.load %alloc_0[%arg1, %arg2] : memref<32x32xf32>
21       %5 = arith.mulf %2, %3 : f32
22       %6 = arith.addf %4, %5 : f32
23       memref.store %6, %alloc_0[%arg1, %arg2] : memref<32x32xf32>
24     }
25   }
26 }
27 %alloc_1 = memref.alloc() {alignment = 64 : i64} : memref<32x32xf32>
28 scf.for %arg1 = %c0 to %c32 step %c1 {
29   scf.for %arg2 = %c0 to %c32 step %c1 {
30     %2 = memref.load %alloc_0[%arg1, %arg2] : memref<32x32xf32>
31     %3 = memref.load %1[%arg2] : memref<32xf32>
32     %4 = arith.addf %2, %3 : f32
33     memref.store %4, %alloc_1[%arg1, %arg2] : memref<32x32xf32>
34   }
35 }
```

The least abstract dialects at this level of abstraction are the `arith` dialect, which models basic integer and floating point mathematical operations, and the `vector` dialect, a generic, re-targetable, higher-order (i.e., multi-dimensional) vector that carries semantically useful information for transformations that enable targeting vector ISAs on concrete targets (e.g., AVX-512, ARM SVE, etc.).

The dialects at this level of abstraction, especially `scf` and `vector`, are where the real optimization work occurs; transformations such as loop-unrolling, loop-fusion, loop-tiling can have enormous impact on the runtime performance of any code (Zhao et al., 2018), but are especially important for numerics intensive code, such as can be found to constitute the majority of kernels in a DNN. Furthermore, explicit vectorization (rather than auto-vectorization) is critical to achieving good performance of various compute-intensive kernels, deployed to both CPUs and GPUs (Dickson et al., 2011). Hence, it's important to be able to efficiently and effectively manipulate representations of DNNs at this level of abstraction, even more-so than what MLIR currently enables.

Low-level dialects (target-specific code generation)

At the lowest level of abstraction, MLIR contains implementations of dialects that can interface with hardware specific runtimes and ISAs, such as `nvvm`, which models NVPTX instructions, `spirv`, and `llvm`, which faithfully models LLVM IR and therefore enables targeting all backends supported by LLVM (including `x86_64` and `aarch64`/`arm64` CPU ISAs). The latter dialect includes support for managed runtimes on top of ISAs (such as OpenMP, in combination with the `omp` dialect) and coroutines (in combination with the `async` dialect). These target-specific, code-generation focused, dialects enable end-to-end compilation of MLIR programs (e.g., DNNs) to a variety of execution environments, including single-core CPU, multi-core (threaded) CPU, and GPU, including SoTA NVIDIA platforms but also lesser known vendors that implement the SPIR-V standard (see Section 4.3 for

demonstrations of `nelli`'s end-to-end compilation features).

4.1.2 eDSL construction in Python

Given a host language, there are (invariably) several ways to implement an embedded domain-specific language; the set of avenues available is only circumscribed by the facilities of the host language and the goals of the DSL designer. `nelli` is embedded in Python and so we discuss two eDSL implementation approaches (including merits and deficiencies) with Python as the host language. Indeed, each of these two approaches was validated (i.e., implemented) over the course of developing `nelli` and discarded in favor of the chosen approach (see Section 4.2).

Compiling

The most straightforward approach to implementing an eDSL in any host language (conceptually) is to build a compiler using that language for (a subset of) that language. This involves static (ahead-of-time) source analysis, including lexing, abstract syntax tree (AST) construction, control-flow analysis, type inference, and code generation (for the target language, MLIR or otherwise). Suffice it to say, this is a monumental undertaking. Nonetheless, the undertaking has been undertaken, in the context of Python and, specifically, numerics intensive programs, many times to varying degrees of success (Behnel et al., 2010, Kay Hayen, 2023, Shajii et al., 2023).

The scope of such an undertaking is slightly improved by the fact that Python provides, in its standard library, source lexing (for Python source code), AST construction, and AST traversal utilities (in the `ast` package). But, comparatively speaking, these aspects of the undertaking are the least challenging[9]; the principal challenges are control-flow analysis and

9. Indeed, there exist many lexers and parsers for Python (Zimmerman, 2022, Parr and Quong, 1995) implemented in other, more performant, languages, i.e., preferable alternatives to the `ast` package, if one's goal were to build a Python compiler.

type inference. With respect to the latter, Python's highly permissive runtime and "duck typing"[10] paradigm requires a compiler to reckon with all mutations of an instantiated object; any object instance can be made to quack like a duck at any point in the execution of a Python program. More seriously (supposing property mutations were prevented), Python does not have nested lexical scopes below the level of a function body: for example, in the following

```
def leaky(a):
    if a % 2 == 0:
        b = 5
        c = 3 * b
    elif a == 5:
        b = "5"
        c = "3" + b
    else:
        pass
    return c
```

the conditional actually "yields" two values (`b` in addition to `c`) and the same is true for all such regions (i.e., `for` s and `with` s), i.e., they "leak" definitions and (possibly) grow the use-def chains of identifiers in subsequent regions. In addition, irrelevant of lexical scoping, the conditional actually yields union types (`b, c: int | str | None`) and hence the target language needs to support such union types. Currently, MLIR does not support such union

10. Python is believed to be "dynamically typed": this is a widely held misconception. In fact, every value manipulated by the Python runtime is a subclass of `<class object>` : `(1).__class__.__bases__ == (<class object>)` . Thus, method resolution (**which can be patched at runtime**) determines the effective type of a value: `(1).__class__.__mro__ == (<class int>, <class object>)` .

types[11].

Tracing

An alternative to ahead-of-time (AOT) compilation of a program is just-in-time (JIT) compilation, and in particular, compilation of only the ordered sequence of operations executed during some execution the program; such a compiler is called a *tracing* JIT, alluding to the "tracing" of the execution path of the program. Several such tracing JITs have been built for general purpose Python (Bolz et al., 2009, Lam et al., 2015b, pys, 2023, Anthony Shaw, 2023). A tracing JIT approach obviates the need to perform control-flow analysis and type-inference, because both are fully reified at runtime. However, a tracing JIT does not eliminate the need to parse a source representation of the program, e.g., as in the case of Python, the bytecode representation. Indeed, Numba (Lam et al., 2015b), Pyston (pys, 2023), and Pyjion (Anthony Shaw, 2023) compile CPython virtual machine bytecode instructions (as opposed to textual source) directly to (target) assembly language[12]. It's important to emphasize that while, in principle, each of Numba, Pyston, and Pyjion can be used to compile entire Python programs, they are frequently used as eDSLs for accelerated implementations of the numerics intensive portions of Python code, through their partial-compilation APIs (`@njit` and `pyjion.enable()`, for Numba and Pyjion respectively).

An alternative to JIT compiling Python at the bytecode level (i.e., handling all opcodes), especially relevant for eDSL construction, is instrumenting ("hooking") only a subset of operations in the host language. For example, function calls and arithmetic operations. The various Python DNN frameworks (PyTorch (Paszke et al., 2017), TensorFlow (Abadi et al., 2016b), JAX (Frostig et al., 2018)) take this approach; by restricting user programs

11. Certainly MLIR supports modeling union types but recall that the broader goal is to translate Python to existing MLIR dialects, rather than mapping Python to a novel dialect.

12. All three projects employ a more generic JIT (LLVM for the former two and the CoreCLR (Troelsen et al., 2017) for the latter) for the "last mile" of code generation.

Listing 17 Sketch of operator overloading on a proxy `Tensor` object for purposes of performing translation to the MLIR `tensor` dialect.

```
class Tensor:
    def __add__(self, other: Tensor):
        emit(f"tensor.add %{self}, %{other}")
        ...
    def __mul__(self, other: Tensor):
        emit(f"tensor.mult %{self} %{other}")
        ...
    def __getitem__(self, item: tuple[int]):
        # indexed load
        emit(f"tensor.extract ${self}[{*item}]")
        ...
    def __setitem__(self, key: tuple[int], value):
        # indexed store
        emit(f"tensor.insert ${value} into ${self}[{*item}]")
        ...
```

to make calls to functions in their own namespaces and by overloading various operators on proxy objects (see Listing 17), the eDSL can wholly own the means of production[13], and thereby perform source-to-source translation. While simple and effective, hooking function calls and operator overloading suffers from an aesthetically displeasing deficiency: in a host language (such as Python) where control-flow primitives such as `if`s and `for`s cannot be instrumented, they must be replaced (within the context of the eDSL) with explicit proxies (e.g., `tf.while_loop` and `jax.lax.cond`). More critically, existing such eDSLs suffer from a fundamental limitation of the tracing approach: if host-language conditionals are allowed in any capacity, then the path less traveled by the program will be not captured by the eDSL. For example, in the following

13. Recall, a production is a rewrite rule specifying a symbol substitution that can be recursively performed to generate new symbol sequences. A finite set of productions P is the main component in the specification of a formal grammar (such as that of a programming language).

```
def single_path(x: Tensor, a: int):
    if a % 2 == 0:
        y = 2 * x
    else:
        y = 3 * x
    return y
```

Despite being able to effectively capture all arithmetic operations on a `Tensor`, no tracing eDSL can capture both arms of the conditional. `nelli` addresses this limitation.

4.2 Design and implementation of `nelli`

The primary design goal of `nelli` is to be *easy to use* and *simple to understand*, while remaining faithful to the semantics of MLIR. By semantics of MLIR, we mean that dialects as rendered in `nelli` (i.e., names and uses of operations) should reflect as closely as possible their rendering in MLIR IR. Note, we draw a subtle distinction between easy and simple: easy to use implies that it should work (generate MLIR IR) with very little fanfare while simple to understand means studying the implementation should reward a modicum of effort (without requiring an inordinate investment). Addressing the former, much effort on our part has been invested in packaging `nelli` for distribution (it can be directly `pip install`ed without compiling LLVM/MLIR). Further, in order to reduce the barrier to reuse of existing code, `nelli` is also extensible (in and of itself) and exposes MLIR in an extensible way.

Addressing the latter precludes various metaprogramming techniques, such as wholesale source rewriting and Python `metaclass` programming. Additionally, it precludes the use of dynamic scoping (using `contextvars`) to implement patterns such as stacks of monadic interpreters (Kiselyov, 2012, Amin and Rompf, 2017). `nelli` uses three techniques to accomplish the stated design goals: operator overloading, trivial source rewriting, and bytecode

Listing 18 Instantiating `func.func` with a `scf.for` using the upstream MLIR Python bindings compared with specifying the same program using `nelli`.

```
with Context() as ctx:
  with Location.unknown(context=ctx) as loc:
    index_type = IndexType.get()
    f = func.FuncOp("simple_for", ([], []))
    with InsertionPoint(f.add_entry_block()):
      lb = arith.ConstantOp.create_index(0)
      ub = arith.ConstantOp.create_index(42)
      step = arith.ConstantOp.create_index(2)
      three = arith.ConstantOp.create_index(3)
      loop = scf.ForOp(lb, ub, step, iter_args)
      with InsertionPoint(loop.body):
        three_i = arith.MulIOp(
          three,
          loop.induction_variable
        )
        scf.YieldOp([])
      func.ReturnOp([])
```

```
@mlir_func
def simple_for():
  for i in range(0, 42, 2):
    two_i = 3 * i
```

rewriting. We discuss each in turn (effectively, in order of increasing complexity). We also discuss how `nelli` addresses extensibility.

4.2.1 Upstream manicuring and operator overloading

MLIR, irrelevant of `nelli`, procedurally generates Python bindings for functionality related to emitting MLIR IR. This procedural generation is made possible by virtue of the fact that almost all operations, in all MLIR dialects, are defined using ODS (see Section 4.1.1). Nonetheless, convenient (and robust) as these existing bindings might be, they are quite verbose, requiring specifying most attributes of operations explicitly; see Listing 18 for an example. Thus, some of the work of `nelli` involves normalizing the upstream APIs; in particular we implement operator overloading for various arithmetic operations on values that are results of `arith` operations (see Listing 20), as well indexing and slicing on results of `memref` and `tensor` operations (see Listing 13). Additionally we overload Python param-

eter annotations to implement a minimal form of Hindley-Milner[14], as well as instantiating `func`s with typed parameters. Finally, we use Python `class` namespaces as models for `module`s, including nested `gpu.module`s (see Listing 19).

4.2.2 Trivially rewriting the AST

It's important to understand how the upstream MLIR Python bindings function (as a reflection of how MLIR functions). Consider the instantiation of `scf.for` in Listing 18; operations to be inserted into the body of the `scf.for` must have their `InsertionPoint`s set to (somewhere in) the body of the `scf.for`. Thus, the Python bindings corresponding to those operations (i.e., `arith.MulIOp`) must be executed within the context of `InsertionPoint`. Eliminating the indentation due to the `with` (which indicates a nested scope where none exists) is worthwhile. One trivial way to accomplish this is to explicitly `__enter__` and `__exit__` the `InsertionPoint(loop.body)` context manager; see Listing 21. But requiring the user to explicitly indicate the end of the `for` loop transforms Pythonic `for` loops to Pascal-style `for` loops. Thus, `nelli` rewrites user functions (at the AST level) and automatically inserts such opening and closing calls for all `for`s and `if`s (see Listing 22) . Note, since we rewrite the AST (not the source itself), we are able to patch line numbers for all nodes to reflect original source locations and thus all Python IDE, error-reporting, and debugging infrastructure is undeterred i.e., users are able to set breakpoints in functions and inspect objects just the same as for any Python code[15].

14. In reality, MLIR performs the type inference, `nelli` simply requires fully type-annotated function parameters.

15. This is emphatically not the case for eDSLs like Numba and Pyjion which compile and execute Python using, effectively, their own bytecode interpreters.

Listing 19 Overloading `class`es to support nested `gpu.module`s.

```
class MyClass1(GPUModule):
  def kernel(
    self,
    A: MemRef[(M, N), F32],
    B: MemRef[(N, K), F32],
    C: MemRef[(M, K), F32],
  ):
    x = block_id_x()
    y = block_id_y()
    a = A[x, y]
    b = B[x, y]
    C[x, y] = a * b
    return

m = MyClass1(
  func_attributes={
    "spirv.entry_point_abi":
      spirv.entry_point_abi(
        workgroup_size=[1, 1, 1]
      ),
 }
)

@mlir_func
def main(
  A: MemRef[(M, N), F32],
  B: MemRef[(N, K), F32],
  C: MemRef[(M, K), F32],
):
  m.kernel(A, B, C,
    grid_size=[4, 4, 1],
    block_size=[1, 1, 1]
  )
```

```
module attributes {gpu.container_module}
{
  gpu.module @MyClass1 {
    gpu.func @kernel(
      %A: memref<4x16xf32>,
      %B: memref<16x8xf32>,
      %C: memref<4x8xf32>)
    kernel attributes {
      spirv.entry_point_abi =
        #spirv.entry_point_abi<
          workgroup_size = [1, 1, 1]
        >
    } {
      %x = gpu.block_id  x
      %y = gpu.block_id  y
      %a = memref.load %A[%x, %y] : ...
      %b = memref.load %B[%X, %Y] : ...
      %c = arith.mulf %a, %b : f32
      memref.store %C, %C[%0, %1] : ...
      gpu.return
    }
  }
  func.func @main(
    %A: memref<4x16xf32>,
    %B: memref<16x8xf32>,
    %C: memref<4x8xf32>) {
    %c4 = arith.constant 4 : index
    %c1 = arith.constant 1 : index
    gpu.launch_func async
      @MyClass1::@kernel
      blocks in (%c4, %c4, %c1)
      threads in (%c1, %c1, %c1)
      args(
        %A : memref<4x16xf32>,
        %B : memref<16x8xf32>,
        %C : memref<4x8xf32>
      )
    return
  }
}
```

Listing 20 Operator overloading of results of `arith` operations.

```
one = arith.constant(1.0)
two = arith.constant(2.0)
three = one + two
```

⇒

```
%one = arith.constant 1.0e+00 : f32
%two = arith.constant 2.0e+00 : f32
%three = arith.addf %one, %two : f32
```

Listing 21 Trivially rewriting user functions in order to explicitly manage context managers for MLIR operations with regions; the `scf_range` (in addition to instantiating the `scf.for`) triggers `__enter__` (on a thread-local handle to a context manager) and the `scf_range` triggers `__exit__`.

```
@mlir_func(rewrite_ast_=True)
def simple_for():
    for i in range(0, 42, 2):
        two_i = 3 * i
```

⇒

```
@mlir_func
def simple_for():
    for i in scf_range(0, 42, 2):
        two_i = 3 * i
        scf_endfor()
```

Listing 22 AST rewriting of conditionals for manual (but implicit) management of context managers for lowering to `scf.if`.

```
@mlir_func(rewrite_ast_=False)
def ifs(M: F64, N: F64):
    one = constant(1.0)
    if scf_if(M < N):
        one = constant(1.0)
        scf_endif_branch()
    else:
        scf_else()
        two = constant(2.0)
        scf_endif_branch()
        scf_endif()
```

⇒

```
@mlir_func(rewrite_ast_=True)
def ifs(M: F64, N: F64):
    one = constant(1.0)
    if M < N:
        one = constant(1.0)
    else:
        two = constant(2.0)
```

4.2.3 Trivially rewriting bytecode

Rewriting the source AST enables mapping Python control-flow primitives to various MLIR control-flow operations except for one caveat: as mentioned in section 4.1.2, relying on runtime execution of Python code (and hooks) to capture programs precludes faithful capture of conditionals. For example, irrespective of arbitrary AST transformations, only one arm of the conditional in Listing 22 can be traced. Although it's debatable whether multi-arm conditionals are crucial (`scf.if` does support an `scf.else` branch), it would be a strange language that supported only single-arm conditionals.

The resolution to the conundrum of the multi-arm conditional lies in rewriting the program on a deeper level than the AST; recall Python programs are compiled (by the CPython implementation) to bytecode instructions. The CPython implementation of Python is a stack-based virtual machine (Ike-Nwosu, 2015) that implements conditionals like many other virtual machines: using jump instructions (see Listing 23). Thus, the solution is to simply rewrite the bytecode of the user's function and remove those jumps[16], thereby forcing the CPython interpreter to execute all instructions in all arms of the conditional. It's important to emphasize that this transformation is reasonable given the stated goals of `nelli`: the eDSL program is not computing on data and has no intended side-effects other than to emit MLIR IR. Thus program capture, rather than evaluation, permits (and encourages) us to fundamentally alter the semantics of conditionals in this way; certainly, under different circumstances, such a transformation would be wholly nonsensical.

16. In fact, the jump instruction is replaced by a `NOP` (no-op) instruction in order to prevent invalidating stack size calculations.

Listing 23 CPython bytecode instructions corresponding to multi-arm `if`; note the `POP_JUMP_IF_FALSE` that executes a jump to the else branch (lines 14-16) if the condition (`COMPARE_OP`) evaluates to `False` (whereas, otherwise the true branch, lines 10-11, is executed).

```
@mlir_func(rewrite_ast_=True)
def ifs(M: F64, N: F64):
    one = constant(1.0)
    if M < N:
        one = constant(1.0)
    else:
        two = constant(2.0)
```

```
 8 LOAD_GLOBAL              (scf_if)
12 LOAD_FAST                (M)
18 LOAD_FAST                (N)
20 CALL_FUNCTION
22 COMPARE_OP               (<)
24 CALL_FUNCTION
26 POP_JUMP_IF_FALSE        (to 46)
...

30 LOAD_CONST               (2.0)
34 STORE_FAST               (two)
...

46 LOAD_GLOBAL              (scf_else)
54 LOAD_CONST               (6.0)
58 STORE_FAST               (six)
...
66 LOAD_GLOBAL              (scf_endif)
...
72 LOAD_CONST               (None)
74 RETURN_VALUE
```

4.2.4 Extensibility

Currently MLIR is extensible to a limited extent[17] `nelli` addresses extensibility in four ways: the first and second being exercises of existing (but infrequently employed) MLIR APIs, and with the remaining two being novel (relative to MLIR):

1. `nelli` uses the `_site_initialize` to register (in-tree) dialects at load-time;
2. `nelli` subclasses existing `ir.OpView` classes thereby extending their functionality despite being out-of-tree;
3. `nelli` is built with exported symbols, thus enabling users to extend the various C++ utility classes that comprise the upstream bindings (without recompiling MLIR);
4. `nelli` implements AST walking functionality (akin to Python's `ast.NodeTransformer`) which, along with upstream improvements contributed by the authors, enables writing simple IR rewrites wholly in Python.

Extension points (1) and (2) exercise existing MLIR Python bindings APIs in unconventional ways to demonstrate (1) extending the set of registered (available at runtime) dialects and (2) wrapping/polishing existing `ir.OpView` APIs (constructors, getters, setters) without repackaging the bindings. Extension points (3) and (4) are more nuanced.

By default, MLIR Python bindings are built with "hidden" symbols[18], making those symbols unavailable for symbol resolution of subsequently loaded libraries. This has the effect that none of the bindings' utility classes can be extended without recompiling. `nelli` exports these symbols, thus downstream projects can `pip install nelli` immediately extend bindings for existing MLIR dialects, operations, types, and attributes. With respect to the final extension point (writing simple IR rewrites wholly in Python), recall that the

17. The state of affairs is steadily improving; between starting and finishing this manuscript, the authors contributed three substantive improvements.

18. This is actually a design choice made by `pybind11` (Jakob et al., 2016) rather than MLIR.

conventional method for transforming/rewriting IR is by building a `RewritePattern` using the C++ API. While the MLIR C++ API is robust and well-designed, like all such APIs, it is rigid and demanding; that is to say, it is not suited for experimentation and quick iteration. By contrast the Python AST utilities (under the standard library package `ast`) present a lightweight API that enables building small (but effective) rewrites of the Python AST. Taking this user experience as our inspiration, we implement similar functionality by generating AST visitors procedurally from the upstream bindings; one `DialectVisitor` for each registered MLIR dialect, including out-of-tree dialects. Using these `DialectVisitor`s, along with the recently contributed `replace_all_uses_with` upstream API, we are able to build small but non-trivial IR transformations on MLIR `ir.Module`s just as one does using `ast.NodeTransformer`.

4.3 Demonstration and evaluation

We demonstrate the capabilities of `nelli` with three small exercises:

1. an end-to-end (including execution) GPU example of a batched, multi-channel, 2D convolution that lowers to both NVIDIA devices and Vulkan supporting devices, such as the VideoCore VI 3D found in the Raspberry Pi 4 Model B and the Apple Neural Engine (found in the Apple M1 series of laptops);

2. an end-to-end that lowers the same kernel to the managed OpenMP runtime;

3. the previous two examples integrated with a black-box, gradient-free, optimizer (Rapin and Teytaud, 2018) for searching the space of possible optimizing transformations.

4.3.1 End-to-end GPU

We implement a standard 2D-NCHW convolution and parallelize across output elements; see Listing 24. Note, we set the default range to be mapped to `scf.for` but explicitly

Listing 24 Standard representation of a 2D-NCHW convolution parallelized across output elements and corresponding pass pipeline. Note, we freely interleave `scf.parallel` and `scf.for` (i.e., `range`). Note also that only lines 13, 14 of the pass pipeline are specific to NVIDIA devices.

```
@mlir_func(range_ctor=scf_for)
def conv2d_nchw_fchw(
  input: MemRef[(N, CI, HI, WI), F32],
  kernel: MemRef[(CO, CI, K, K), F32],
  output: MemRef[(N, CO, HO, WO), F32],
):
  for n, co, ho, wo in parallel(
    (0, 0, 0, 0), (N, CO, HO, WO)
  ):
    for ci in range(0, CI):
      for ki in range(0, K):
        for kj in range(0, K):
          ii = ho + ki
          jj = wo + kj
          inp = input[n, ci, ii, jj]
          ker = kernel[co, ci, ki, kj]
          output[n, co, ho, wo] +=
                              inp * ker
```

```
pipeline=Pipeline()
  .FUNC()
    .gpu_map_parallel_loops()
  .CNUF()
  .convert_parallel_loops_to_gpu()
  .FUNC()
    .lower_affine()
    .convert_scf_to_cf()
    .gpu_kernel_outlining()
  .CNUF()
  .GPU()
    .strip_debuginfo()
    .convert_gpu_to_nvvm()
    .gpu_to_cubin(chip="sm_75")
  .UPG()
  .gpu_to_llvm()
```

map the outermost four loops to a `scf.parallel` thereby specifying that each output element (`n, co, ho, wo`) should be computed in parallel. The pass pipeline (cf. Listing 24) that accompanies the kernel is specialized for NVIDIA devices but only in the final (hardware-specific) passes. The higher-level passes effectively perform two functions: `gpu_map_parallel_loops` assigns nested `scf.parallel` loops to the corresponding level of the GPU workgroup hierarchy (grid, block, and thread) and `gpu_kernel_outlining` outlines GPU kernel code so that it can be separately compiled and serialized. Note that while this basic example only has one `scf.parallel` and is thus mapped only to blocks, further transformations (such as tiling) can introduce nested `scf.parallel`s, thereby inducing a distribution across both blocks and threads.

The presented code lowers fully to both NVIDIA and Vulkan targets almost unaltered[19] and therefore this kernel successfully executes on all three of the aforementioned hardware platforms (NVIDIA 3080Ti, Apple Neural Engine, and VideoCore VI). In fact, since MLIR provides utilities for mapping NumPy arrays to the various GPU runtime native buffers (using a buffer descriptor called `StridedMemRef`), and utilities for interfacing with the hardware runtimes (CUDA runtime and Vulkan runtime), all end-to-end experiments can be executed without ever leaving the comfort of `nelli`.

4.3.2 End-to-end OpenMP

Since the aforementioned 2D-NCHW convolution kernel is implemented in terms of the `scf` dialect, which, as the name suggests, models programs in terms of abstract but structured control-flow operations, the same 2D-NCHW convolution kernel can be lowered to the OpenMP runtime with just the flip of a switch; using `nelli`'s `Pipeline`, it's just a matter of substituting `convert_scf_to_openmp` for `convert_parallel_loops_to_gpu`. This wraps the loop nest in a `omp.parallel` context and implements the `scf.parallel` loop

19. Vulkan does not support 4D buffers so in practice inputs have to be packed along the batch and channel

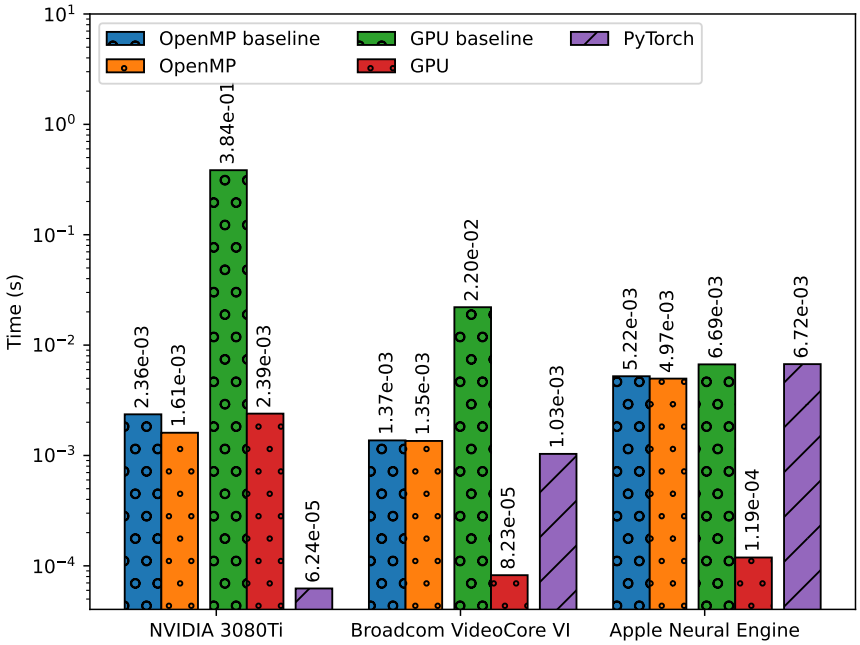


Figure 4.2: Nevergrad optimization for tiling and inner-loop unrolling of a 2D-NCHW convolution kernel.

as a `omp.wsloop`, i.e., worksharing loop. Just as with the end-to-end GPU implementation, thanks to LLVM's support for OpenMP, all end-to-end experiments can be executed without ever leaving the comfort of `nelli`.

4.3.3 Derivative-free optimization

Algorithmic correctness is a necessary but not sufficient condition for achieving high performance implementations of numerically intensive kernels; due to the variety of hardware platforms, fine-tuning of the implementation for each platform is critical (Li et al., 2009). In many cases, where a formal cost model of the hardware platform is not available, a directed search of the program transformation space cannot be realized. In such instances, black-box (gradient-free) optimization techniques can be employed. To demonstrate the value of having seamless interoperability with the Python ecosystem, we connect `nelli` to `Nevergrad` (Rapin and Teytaud, 2018), a Python package for gradient-free optimization.

We experiment with applying two loop nest transformations: tiling and unrolling (Cardoso et al., 2017), using the MLIR pass `scf_parallel_loop_tiling` and a loop-unrolling operation (`loop_ext.LoopUnrollOp`). Both transformations are parameterized (by tile sizes and unroll factor, respectively) and it is this parameter space that can be searched over to find the optimal kernel for each hardware platform.

Thus, we set `Nevergrad` loose on our kernel on three different platforms: a workstation with a NVIDIA 3080Ti GPU, an Apple M1 MacBook Pro with a Neural Engine GPU, and a Raspberry Pi 4 Model B with Broadcom VideoCore VI GPU. On each platform we apply the 2D-NCHW convolution on an input with shape $(N, C, H, W) \coloneqq (1, 1, 1280, 1280)$, using a $(C_i, C_o, K) \coloneqq (1, 3, 3)$ filter (where C_i, C_o are channels-in, channels-out, respectively), and search the space of possible tile sizes and loop-unroll factors. In addition, just for fun, we apply loop-unrolling to the OpenMP implementation. As well, we compare it to the PyTorch implementation of the same kernel. Figure 4.2 shows the results of the experiment. A few noteworthy observations:

1. for this small kernel, OpenMP is fairly performant but loop-unrolling has almost no effect on the implementation because the worksharing loop distribution already distributes work maximally across all available cores;

2. tiling and loop-unrolling is hugely important for achieving good performance on GPUs;

3. PyTorch has highly optimized implementations of kernels for common platforms (such as NVIDIA) but otherwise can be outperformed by fine-tuning for a user's specific hardware configuration.

Indeed, each of these conclusions is well-known and understood and thus we observe that enabling users to easily, and transparently, perform this kind of fine-tuning, through a representation like MLIR and an interface like `nelli`, is valuable.

---

dimensions.

4.4 Related Work

There are several projects in the MLIR ecosystem that aim to support a Python frontend for some MLIR dialect (or collection of dialects) but as far as we are aware (i.e., at the time of writing) none that aim to expose *all* builtin dialects through a Python frontend. Most famous amongst these are JAX (Bradbury et al., 2018), which provides a NumPy-conformant interface to the `hlo` (sta, 2023) family of dialects, and TensorFlow. Each of these effectively provides a very high-level, Python-embedded, DSL for machine learning (specifically neural network) operations implemented against their respective dialects. Note, while superficially the Torch-MLIR project (tor, 2023) resembles JAX and TensorFlow, in that it transforms Python to an MLIR dialect (`torch-mlir`), the Python frontend is in fact provided by PyTorch itself[20], we do not count it amongst this group of projects. In this category there is also the interesting `numba-mlir` project, a MLIR-based Numba backend, where, Numba (Lam et al., 2015a) translates from NumPy-specific Python code to lower-level representations (Numba IR and then LLVM IR) by analyzing the CPython bytecode instructions[21] the Python compiles to. `numba-mlir` lowers to several high-level (higher than LLVM IR) domain-specific dialects (`ntensor`, `plier`) by recovering Regionalized Value State Dependence Graphs (Reissmann et al., 2020) from the Numba IR. Finally, there is the Pylir project, implemented on top of MLIR, which aims to be an optimizing, ahead-of-time, compiler for all of Python. Thus, Pylir's goal is not to be a frontend for MLIR in and of itself but to compile Python code to native executables; it accomplishes this impressive feat by parsing Python to its own dialects.

Alternatively, there exist projects that approach the problem orthogonally - they aim to provide a Python interface to a MLIR-like framework but not MLIR itself. The projects in

20. Torch-MLIR operates an the intermediate representations exported by PyTorch rather than directly on Python source.

21. CPython is a stack-based virtual machine with its own assembly/bytecode instructions (see `https://docs.python.org/3/library/dis.html`).

this category generally aim to be completely independent of upstream MLIR and thus parse MLIR-native IR into proprietary ASTs and manipulate those ASTs in various ways (transforming, serializing, etc.). For example, pyMLIR (Tal Ben-Nun, Kaushik Kulkarni, Mehdi Amini, Berke Ates, 2023) implements a LALR(1) grammar (extracted from the upstream MLIR documentation) and parser. pyMLIR's parser generates an AST representation that further implements Python's `ast.NodeTransformer` interface, thus enabling various AST transformations. The resulting ASTs can be once again serialized to conformant MLIR. Note, `nelli`'s AST transformation functionality is inspired by pyMLIR's, but instead of operating on a proprietary AST representation, `nelli`'s operates on the canonical MLIR AST. On the other hand, xDSL (Brown et al.) is a Python-native compiler framework influenced by MLIR but not coupled directly to MLIR; xDSL emitted IR is validated against MLIR but only as part of its continuous integration process. These projects are all very interesting and impressive accomplishments but they are orthogonal to our goals, i.e., providing a Python frontend to MLIR itself, rather than an arbitrary compiler framework (irrespective of how feature-ful or MLIR-like that framework might be).

4.5 Conclusion

We described `nelli`, a lightweight, open source, Python frontend for the Multi-Level Intermediate Representation compiler infrastructure. `nelli` aims to make MLIR more accessible by providing a Pythonic syntax for the various MLIR dialects while remaining faithful to MLIR's semantics. `nelli` uses operator overloading, AST rewriting, and bytecode rewriting to map Python control flow primitives, like conditionals and loops, to MLIR control flow operations, amongst other design choices. `nelli` is designed to be simple to use and understand. It performs minimal static analysis and thus incurs minimal complexity in perform the translation to MLIR, compared to existing frontends. As a Python eDSL, it interoperates with existing Python tooling is fully extensible, in terms of dialects, operations, types,

and attributes supported. Further, we demonstrated the utility of `nelli` by showing end-to-end compilation of an example kernel for different hardware platforms, including integration with a derivative-free optimization library to automatically optimize for those platforms. In summary, `nelli` provides an easy way to interface with MLIR and manipulate intermediate representations directly, avoiding the complexities and artifacts of lowering from high-level frameworks. This enables more flexible program analysis and transformation compared to those existing MLIR frontends.

Future work in this area will strongly focus on extending these techniques to domains where there is currently a lack of readily available language frontends. For example, the MLIR-AIE[22] project currently implements a fully functional MLIR dialect but does not effectively support any direct frontend language. It supports various upstream deep learning frameworks (such as PyTorch and TensorFlow) but only through the IREE[23] project. This path is robust but imprecise, relying on various heuristic compiler passes to translate the extremely high-level framework operations into extremely low-level AIE dialect operations. In the future, we plan to design a frontend language that will enable developers to optimize their AIE programs precisely and effectively.

22. `https://github.com/Xilinx/mlir-aie`

23. `https://iree.dev`

CHAPTER 5

AN E2E PROGRAMMING MODEL FOR AI ENGINE ARCHITECTURES

Coarse-Grained Reconfigurable Architectures (CGRAs) are architectures that can selectively use or disuse various components, subsets of the architecture to gain much higher performance per Watt over a more diverse set of applications than conventional processors such as multi-core CPUs and GPUs (Choi and Kee, 2015). In particular, owing to the reconfigurability of the device topology, CGRAs are well-suited for dataflow programs, i.e., specifications for the connectivity of functional units that explicitly represent and correspond to the flow of data in a program (Charitopoulos and Pnevmatikatos, 2020). Archetypical in this class of programs are multi-layered Deep Neural Networks (DNNs), wherein individual layers potentially map to discrete subsets of the CGRA and with data flowing between them in the form of activations (Choi and Kee, 2015). Finally, recently, as the necessary fabrication techniques have evolved to enable it, CGRAs have evolved to include processing elements (PEs) with functionality as rich as that of more conventional, standalone processors; today, the PEs found in CGRAs devices potentially have access to large local memories (data and program), scalar and vector ALUs, independent DMA controllers, and high-bandwidth streaming connections (both to other PEs and the host) (WP506, 2022). Note the distinction between a CGRA and a GPU: while both provide access to an array of powerful PEs, only CGRAs (as of this writing) allow explicit specification and manipulation of connectivity between those PEs.

Conceptually, CGRAs have been around since the 1980s, but have failed to see wide deployment (C. Penha et al., 2019). Primarily, there are two reasons for this: firstly, prior to the "deep learning renaissance", there was not such a surfeit of programs that could, naturally, be represented as dataflow graphs (and, thus, the platform lacked a strong, compelling use-case); secondly, prior to the availability of prefabricated CGRAs, deployment

required designing one "whole cloth", either on FPGA or in ASIC. Besides being the veritable antithesis of "coarse-grained", digital design at the RTL level is typically far outside the comfort zone of most software developers. This lack of robust and familiar programming models prevented software developers from productively utilizing CGRAs and potentially still impedes their broader adoption. Recently, deep learning has taken over the world and prefabricated CGRAs such as AMD's AI Engine (AIE), Cerebras CS-1, SambaNova's SN40L have become commercially available[1]. The software "stacks" entailed by these coarse-grained devices significantly reduce the barrier to use by raising the abstraction level of the programming model from RTL. That notwithstanding, they do not eliminate the requirement that software developers be familiar with and manipulate various hardware layers.

In this work, we develop an "end-to-end" programming model for AMD's AI Engine such that a developer can design a dataflow, program the individual PEs, configure the device, launch the program, and manage host-device communications (vis-a-vis memory buffers) all in one Python script (or Jupyter notebook). Our flow is open source and even available as a `pip install`-able package. This work extends previous work, which introduced the frontend language design techniques in a more target-agnostic context (Levental et al., 2023). The work's contributions can be summarized as follows:

1. A programming language frontend (Python embedded domain-specific language) which can be used to represent/specify CGRA specific concepts such as data movement, streaming connections, DMA access patterns, as well as PE-specific concepts (scalar and vector arithmetic operations); in addition, our frontend supports metaprogramming designed to enable easy extension/reuse by users;

2. An MLIR-based compiler with support for two stream router implementations (optimal and approximate), buffer placement/allocation, and auto-vectorization;

1. Cause and effect?

3. Integrations with target codegen compilers, and various host-side runtime bindings that enable seamless host-device-host data passing using familiar (NumPy) APIs;

4. An evaluation of our end-to-end programming model for GEMMs on the Ryzen AI platform (an edge-device deployment of the AI Engine architecture).

To our knowledge, our flow is the first implementation of such an end-to-end programming model for AIE devices.

The remainder of the chapter is organized as follows: Section 5.1 reviews the necessary background on dataflow-style programming and AIE devices, Section 5.2 discusses our "bottom-up" approach to compiler and language frontend design, Section 5.3 evaluates our flow by presenting an implementation of a dataflow architecture for General Matrix Multiply (GEMM), and finally Section 5.5 concludes by discussing related work.

5.1 Background

5.1.1 Dataflow programs

A dataflow-style program is a computing paradigm wherein computation is orchestrated based on the availability of streams of data (rather than on streams of instructions), with tasks being executed (possibly in parallel) as soon as their inputs become available. This paradigm enables highly parallelized execution, facilitating efficient utilization of computational resources and often leading to substantial performance improvements, especially in scenarios where tasks can be decomposed into independent units of work with explicit data dependencies. A dataflow program is (abstractly) represented by a directed graph, called a *dataflow graph* (DFG), wherein vertices represent tasks, and edges denote the flow of data between these tasks. Unlike traditional control flow architectures, which rely on explicit sequencing of instructions, dataflow programs are driven by the propagation of data through the DFG, triggering the execution of tasks as data becomes available.

Mapping dataflows to CGRAs can present several challenges due to the inherent differences between the architectures and characteristics of dataflow graphs; we discuss some of these challenges here to address them in Section 5.2. Firstly, dataflow graphs imply a schedule for tasks but only implicitly, i.e., in terms of dependencies between the tasks. Translating such schedules to communication patterns and actual dataflow routes that achieve low latency and high throughput requires careful analysis. CGRAs have finite quantities of PEs, memory, and interconnect bandwidth. Mapping complex dataflow graphs onto these architectures while meeting resource constraints can be a non-trivial optimization problem. Specifically, because CGRAs are likely coarser-grained than typical dataflow graphs, mapping fine-grained tasks or operations from a dataflow graph onto these architectures carelessly might lead to inefficient resource utilization. Some CGRAs support dynamic reconfiguration, allowing the hardware configuration to be modified at runtime. While this flexibility can be advantageous for adapting to changing computational requirements throughout a single task, managing the reconfiguration overhead and ensuring seamless transitions between different configurations adds complexity to the mapping process. Finally, mapping dataflow graphs onto CGRAs often relies on specialized software tools and methodologies. However, the availability and maturity of these tools can vary, and designing efficient mappings may require significant expertise in the tool and manual intervention.

The space of all dataflow graphs that might be mapped to CGRAs is large, but in this work, we focus on GEMM kernels as a specific but important subset. The reason to focus on GEMM kernels is because they often dominate the computation for DNN models, during training and inference (Jia, 2014, Georganas et al., 2020, Liu and Vinter, 2014). Thus, Formally, given input matrices $A_{M\times K}$, $B_{K\times N}$ with independent dimensions M, N and common dimension K, matrix multiplication computes output matrix $C_{M\times N} = A \times B$. The possible dataflows for matrix multiplication broadly fall into three categories: *inner-product*, *outer-product*, and *row-wise* (Li et al., 2023):

- **Inner-product** computes element $C[m, n]$ of the result as $A[m, :] \cdot B[:, n]$, i.e., as the inner-product of the corresponding row of A and column of B;
- **Outer-product** computes K outer products $A[:, k] \otimes B[k, :]$ and accumulates them[2];
- **Row-wise** computes $C[m, :] = \sum_k A[m, k] \cdot B[k, :]$, i.e., K partial sums of scalar-vector products.

In light of the conventional triple-nested loop formulation of matrix multiplication, these three dataflow schemes can be recognized as a permutation of the loop orders, specifically the reduction loop, i.e., the k dimension (see Figure 5.1).

When on-chip memory capacities are not big enough for all of A, B, thus requiring partitioning, each scheme also has different implications for data reuse; without loss of generality, assume A dictates execution order and is partitioned either along the rows in the inner-product scheme and row-wise scheme or along the columns in the outer-product scheme. Given this assumption, in the inner-product scheme, C achieves full reuse since partial sums for each result element are accumulated in place. However, each B partition (each column) must be fetched multiple times, once for each row of A (resulting in poor reuse of B's columns). By contrast, in outer-product, each B row is fetched only once per column of A, but partial sums of C, if too big for local memory, must be written out and then read again for accumulation, incurring lots of traffic. Finally, as a compromise between extremes, row-wise requires storing relatively small partial sums and achieves good result (C) reuse. Note that in each of the three schemes, assuming very large operands A, B, further partitioning is possible along the K dimension into K/K' sub-partial sums. In Section 5.2 we will append to this discussion by including broadcasting as a possible approach for distributing A, B.

2. Using the matrix identity $C = A \times B = \sum_{k=1}^{p} A[:, k] \otimes B[k, :]$.

Inner product

```
for m in range(M):
  for n in range(N):
    for k in range(K):
      C[m,n]+=A[m,k]*B[k,n]
```

Outer product

```
for k in range(K):
  for m in range(M):
    for n in range(N):
      C[m,n]+=A[m,k]*B[k,n]
```

Row-wise product

```
for m in range(M):
  for k in range(K):
    for n in range(N):
      C[m,n]+=A[m,k]*B[k,n]
```

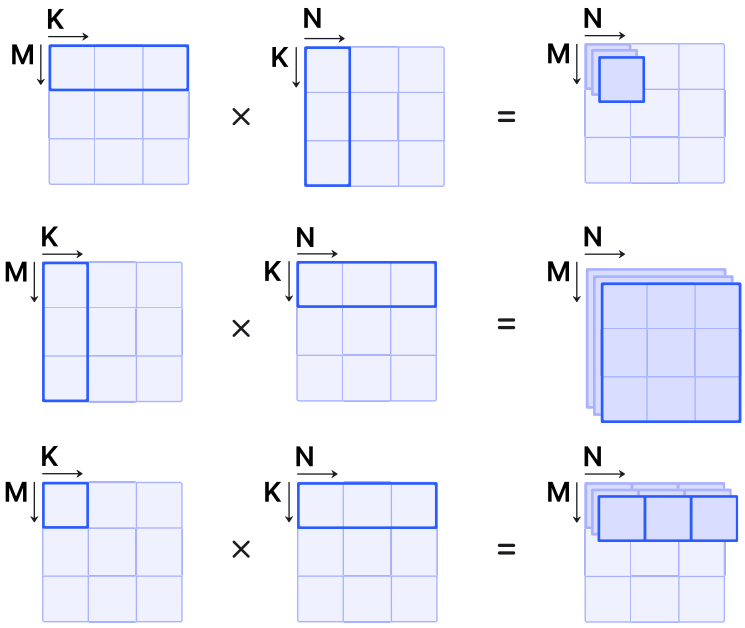


Figure 5.1: Matrix multiplication dataflow strategies; note the relationship between the accumulation of partial sums and the loop ordering (the loop on k is typically called a *reduction*).

5.1.2 AI Engines

AI Engine architectures comprise a two-dimensional array of processing elements, called *tiles*, connected by an AXI4-Stream interface, routed through configurable stream switches (see Figure 5.2). The tiles come in three "flavors", each with differing functionality and purpose:

- **Core tiles**, which contain relatively small data memories (64KiB), program memories (16KiB) and can perform integer and floating point arithmetic (see Figure 5.3);
- **Shim tiles**, located at the edge of the array, and which neither contain memory nor compute resources but provide a host-device interface for the remainder of the array, both for configuration and high-bandwidth dataflow;
- **Memory tiles** (or **memtiles**), located at the boundary between shim tiles and core tiles. They (naturally) contain only large memories (512KiB) but support more streaming connections (than core and shim tiles), thus functioning as an effective L2 cache

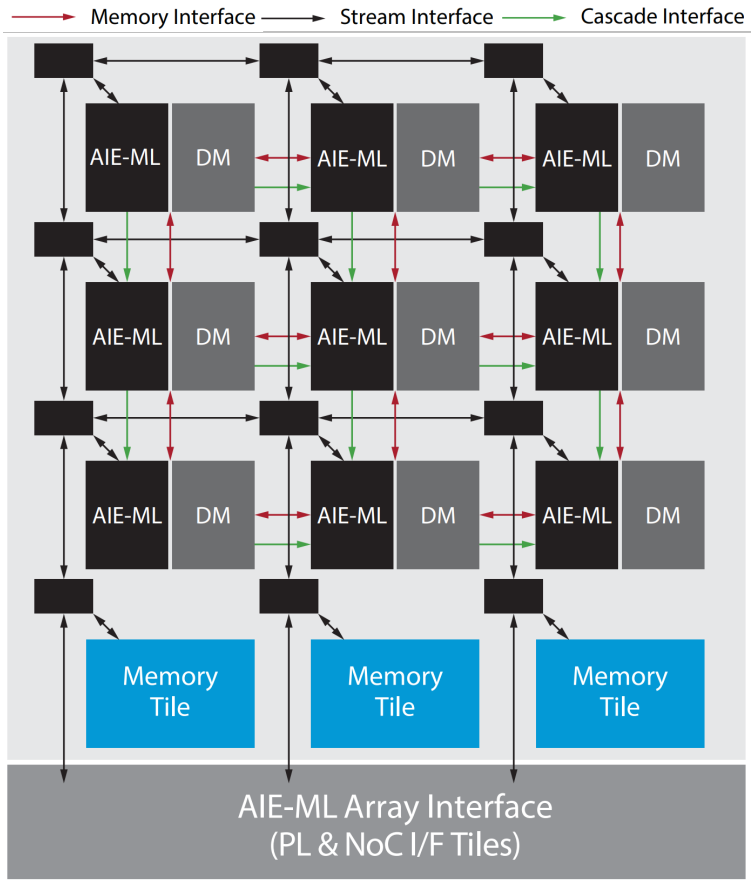


Figure 5.2: AI Engine Array showing the three "flavors" of tiles: core tiles (along with Data Memory blocks), memory tiles, and shim tiles (here referred to as "NoC" tiles). The various arrows indicate the connectivity possible between the tiles.

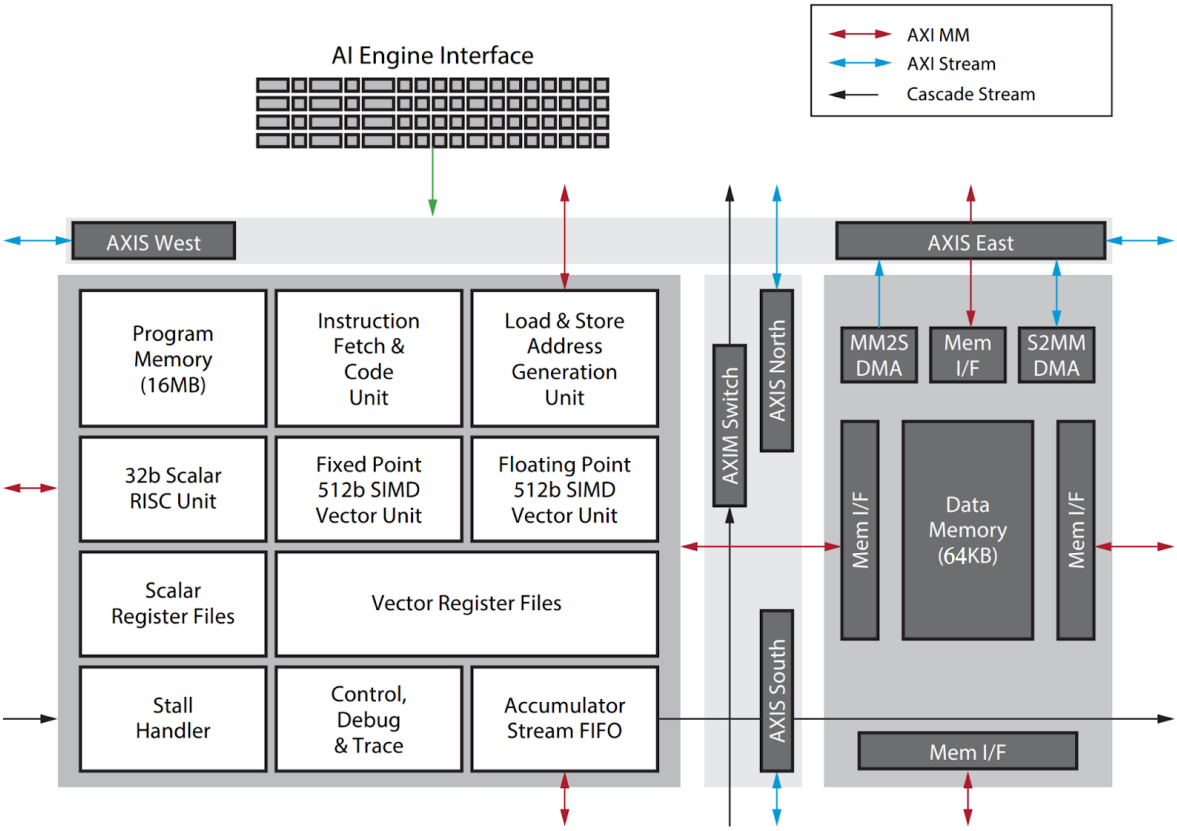


Figure 5.3: AI Engine core tile, with all interfaces and functional blocks.

layer for the entire array.

Each flavor of tile contains independent DMA engines which support multi-channel streaming-to-memory and memory-to-streaming connections (2 for core and shim tiles, 6 for memtiles), as well as counting semaphores to synchronize those DMAs with core accesses. Each DMA engine supports *N-dimensional tensor address generation* (3D for core and shim, 4D for mem) used for delinearizing array indexes to construct non-trivial traversal orders. Each category of tile can also share data and semaphores with adjacent tiles directly (via memory-mapped address ranges). Core tiles, the figurative stars of the show, include, in addition to memory and connectivity functionality, a VLIW vector processor which supports 6-way instruction issue, contains two 512b SIMD vector units (one fixed-point and one floating-point) and a scalar ALU. These processors are programmed using a mixture of conventional programming language primitives (loops, conditionals, etc.) and vector intrinsics.

The important thing to understand about AIE architectures is that the developer can orchestrate almost all aspects of the data movement and compute. This includes configuring all streaming interfaces, all DMAs, and all semaphores (and, of course, all core programs). The benefit of this freedom is that AIE architectures are more flexible than comparable spatial architectures (such as GPUs) with respect to what classes of workloads they can effectively support. The cost of this freedom is that for an arbitrary workload, i.e., program, several scheduling and resource allocation problems need to be solved by the programmer (in concert with the compiler). In particular, we identify two such problems:

1. Programs that necessitate streaming data need to establish connections between tiles; this requires solving an instance of the *congestion-aware traffic assignment problem* (Boyles et al., 2023), with stream switches as endpoints/vertices and connections between them as edges;

2. Orchestrating DMAs such that data is coherent across producers and consumers requires scheduling transfers between DMAs on participating tiles, each synchronized

with its respective core.

In addition, we identify two problems of language ergonomics (desiderata) :

1. Users should be able to avail themselves of broadcast semantics both in specifying the movement of data and in specifying operations on that data;

2. Users should be able to "metaprogram", i.e., design and apply transformations to their programs.

The former is critical for supporting well-known kernel techniques that have proven performant in various other architectures (Tan et al., 2013), while the latter is critical for enabling efficient design space exploration.

5.2 Bottom-up Toolchain Design

We call our approach to compiler and language frontend design "bottom-up" because parts of the compiler preceded the language frontend, whereas typically, compiler implementations follow/succeed some pre-existing language. This inversion of precedence enabled us to design the language specifically to suit the compiler's needs (and thus, the architecture's) and accommodate minor compromises (between the compiler and the language). Our flow, including the Python-embedded domain-specific language (eDSL), is a component of and greatly extends MLIR-AIE, a toolchain based on the MLIR compiler framework (Lattner et al., 2021). The toolchain's explicit design goals address the problems/challenges we identified in Section 5.1.2.

Note that when we say compiler here, we specifically mean the compiler that optimizes representations of CGRA-specific concepts such as data movement, streaming connections, and DMA access patterns; the single core compilers (SCCs), which compile conventional kernel code to the VLIW vector ISA, though well integrated in our toolchain, are beyond scope for this work. In addition, though this work focuses on the end-to-end programming

model, we note that MLIR-AIE encompasses and supports other frontends, such as PyTorch and TensorFlow (through IREE). see Figure 5.4 for a diagram of the programming model (including alternative frontends). The remainder of this section discusses in the programming model greater detail, starting with a review of the AIE dialect, some extensions contributed over the course of this work (including optimal routing). Then, it proceeds to the language frontend.

5.2.1 AIE dialect

The core abstraction layer of our compiler is the AIE dialect[3] which models data movement, streaming connections, DMA access patterns. For example, consider the program in Listing 25; all AIE programs begin by specifying the *tiles* to be orchestrated (using column, row indices). Most AIE programs include streaming connections between the tiles[4]; for this the compiler allows specifying only the endpoints of the stream (i.e., stream switch ports and channels) using *flows*, and performs routing automatically (see Section 5.2.1). Note that when specifying a flow at the IR (dialect) level, both port type and channel index are required – `flow(%tile_0_1, DMA : 0, %tile_0_2, DMA : 0)` specifies a flow between channel 0 of the DMA port on the stream switch associated with `%tile_0_1` and channel 0 of the DMA port on the stream switch associated with `%tile_0_2` – while the language frontend supports automatic assignment. *Buffers* are associated with tiles and are defined by standard MLIR types (`memref` on scalar types such as `i32`). Semaphores are associated are also associated with tiles (using the semantically overloaded `lock` operation) and can be initialized; such initializations are primarily useful for designing multi-producer/multi-consumer communication protocols (between cores and their DMAs or adjacent tiles).

The AIE dialect is designed to enable users to represent complex (but statically sched-

3. See (Lattner et al., 2021) or (Levental et al., 2023) for a review of dialects and other MLIR-specific concepts.

4. Host-device communications are always through a streaming interface.

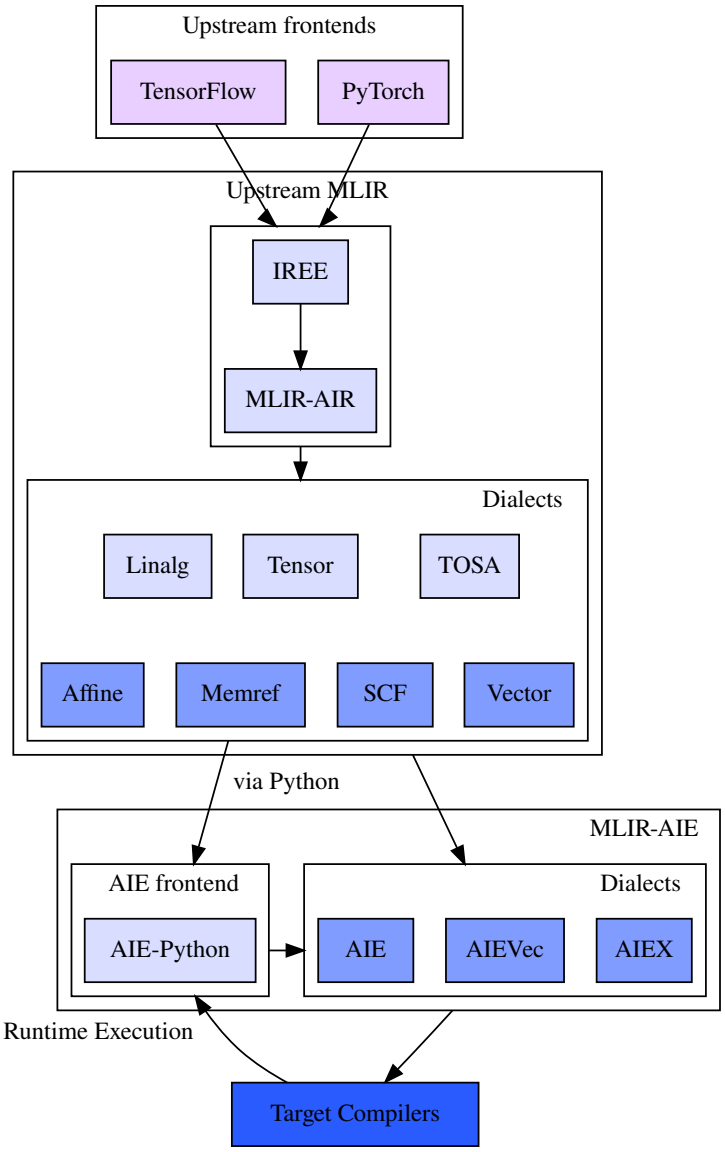


Figure 5.4: MLIR-AIE end-to-end programming model; lightly-shaded components indicate higher level abstractions supported by and interoperating with MLIR-AIE (and vice-versa for darkly-shaded).

Listing 25 The AIE dialect with some of the core operations highlighted; lines 2-4 specify the tiles that make up the design, lines 5-6 specify the stream flows, lines 8-9 specify the buffers and semaphores used by the tiles, lines 11-30 specify the memtile DMA configuration, lines 32-36 specify the core tile DMA configuration, and lines 37-42 specify the core program. Note the `linalg.add` operation operates on `memref<4x4xi32>` buffers (instead of `memref<256xi32>`) thanks to the tiling implemented by the `memtile_dma(tile_0_1)`.

```
1  device(ipu) {
2    %tile_0_0 = tile(0, 0)
3    %tile_0_1 = tile(0, 1)
4    %tile_0_2 = tile(0, 2)
5    flow(%tile_0_1, DMA : 0, %tile_0_2, DMA : 0)
6    flow(%tile_0_1, DMA : 1, %tile_0_0, DMA : 0)
7    flow(%tile_0_2, DMA : 0, %tile_0_1, DMA : 1)
8    %memtile_buffer = buffer(%tile_0_1) : memref<256xi32>
9    %memtile_sem = lock(%tile_0_1, init = 0)
10   %memtile_dma_0_1 = memtile_dma(%tile_0_1) {
11     dma_start(S2MM, 0, ^read_in, ^start_write_out)
12   ^read_in:
13     use_lock(%memtile_sem, AcquireEqual, 0)
14     dma_bd(%memtile_buffer : memref<256xi32>)
15     use_lock(%memtile_sem, Release, 1)
16     next_bd(^read_in)
17   ^start_write_out:
18     dma_start(MM2S, 0, ^write_out, ^end)
19   ^write_out:
20     use_lock(%memtile_sem, AcquireEqual, 1)
21     dma_bd(%memtile_buffer : memref<256xi32>, dims = [
22       <size=4, stride=64>, <size=4, stride=4>,
23       <size=4, stride=16>, <size=4, stride=1>
24     ], len = 16, iteration_step = 16)
25     use_lock(%memtile_sem, Release, -1)
26     next_bd(^write_out)
27   ^end:
28     aie_end()
29   }
30   %core_buffer = buffer(%tile_0_2) : memref<256xi32>
31   %core_sem = lock(%tile_0_2, init = 0)
32   %mem_0_2 = mem(%tile_0_2) {
33     dma_start(S2MM, 0, ^bb1, ^bb3)
34     ...
35     dma_start(MM2S, 0, ^bb4, ^bb6)
36   }
37   %core_0_2 = core(%tile_0_2) {
38     use_lock(%core_sem, AcquireGreaterEqual)
39     linalg.add ins(%core_buffer, %core_buffer)
40                outs(%core_buffer : memref<4x4xi32>)
41     use_lock(%core_sem, Release)
42   }
43 }
```

uled) DMA state machines, i.e., sequences of tasks performed by distinct *buffer descriptor*s, represented by `dma_bd` operations. One way the AIE dialect represents this functionality is unstructured control flow (basic blocks and jumps). Consider the sequence of DMA operations performed by the `%memtile_dma_0_1` in Listing 25 (lines 11-30). The DMA state machine represented is a synchronized transmission of the `memref<256xi32>` buffer; upon acquiring the `%memtile_sem` semaphore, the first buffer descriptor task reads data from the stream (through the S2MM[5] interface) into the `%memtile_buffer`, then increments the `%memtile_sem` by 1, and loops around (jumps to the top of the `^read_in` basic block) and waits to reacquire `%memtile_sem`. Concurrently, the MM2S interface, waits to acquire `%memtile_sem` until its state equals 1. Once acquired, the buffer descriptor task reads from `%memtile_buffer` using N-d tensor addressing. The sequence of `<size=..., stride=...>` attributes represents a linearization[6] of the implicit array indices and effectively produces a `memref<4x4x4x4xi32>`, which corresponds to a 4×4 row-major tiling of the buffer (see Figure 5.5). After writing out the slice of the buffer to the stream, the second buffer descriptor decrements the `%memtile_sem` by -1, loops around and waits to reacquire `%memtile_sem`. Decrementing `%memtile_sem` by -1 releases the first buffer descriptor task to read from the stream again, and so on.

Stream switch, DMA, and semaphore configuration code is separated from single core code[7] by the `core` operation. By being based on MLIR, the AIE dialect interoperates with all the standard MLIR dialects, such as `tosa`, `linalg`, `tensor`, `affine`, and `vector`. This enables users to represent core code using conventional primitives and to avail themselves of all upstream passes for tiling, vectorization, and bufferization (including `transform` dialect).

5. Stream to Mapped Memory.

6. The easiest way to understand the sequence is as a set of nested `for` loops:

```
for i in range(sizes[0]):
  for j in range(sizes[1]):
    for k in range(size[2]):
      buffer[i * strides[0] + j * strides[1] + k * strides[2]]
```

7. Compute kernel code that runs on the VLIW vector processor.

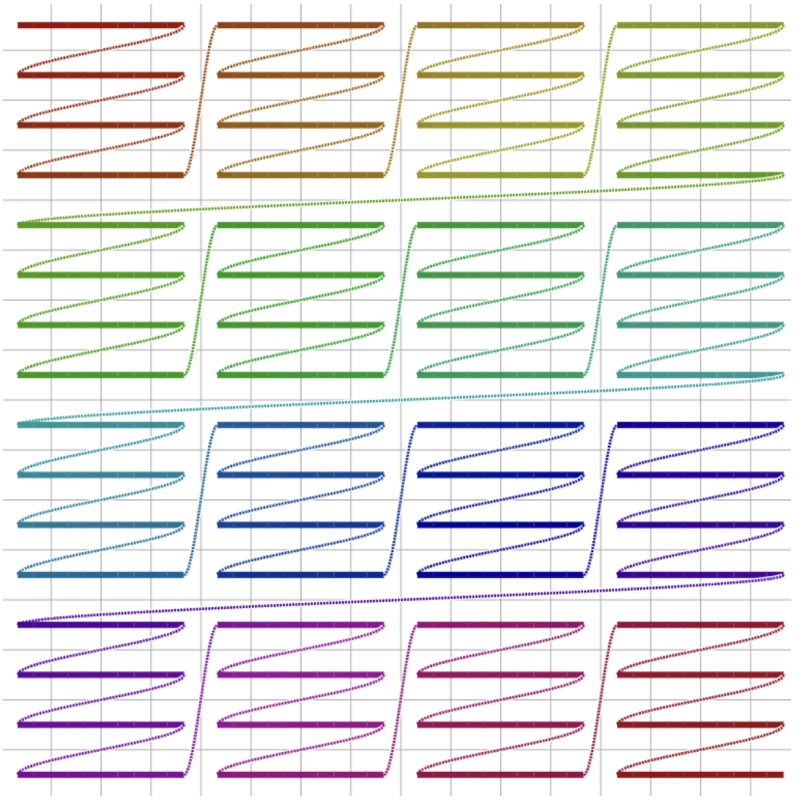


Figure 5.5: 4×4 row-major tiling of the buffer in Listing 25 on page 114.

The AIE dialect also interoperates with the AIEVec dialect, which models and targets the AI Engine vector units. Core code is lowered to LLVM IR for export to SCCs, for target codegen. In Section 5.2.2 we discuss how configuration and core code are bundled and loaded onto the device at runtime.

It is important to note that unstructured control flow is merely the lowest-level abstraction available in the AIE dialect for scheduling DMA operations; it is generally used to represent state machines more complex than simple loops. For loop-like state machines, several structured control flow representations exist, such as the `dma` operation (see Listing 27) and the `objectfifo` operation. The `objectfifo` operation is a FIFO-style abstraction which completely abstracts both the synchronization and task specification; see Listing 27 which demonstrates using a pair `objectfifo`s to schedule and synchronize DMA reads and writes, including coordination with the core code.

Listing 26 More concise representation of the same `memtile` DMA schedule as in Listing 25 using the `dma(...)` operation (lines 4,9).

```
1  %memtile_dma_0_1 = memtile_dma(%tile_0_1) {
2    %memtile_buffer = buffer(%tile_0_1) : memref<128xi32>
3    %memtile_sem = lock(%tile_0_1, init = 0)
4    %0 = dma(S2MM, 0) [{
5      use_lock(%memtile_sem, AcquireGreaterEqual)
6      dma_bd(%memtile_buffer : memref<128xi32>)
7      use_lock(%memtile_sem, Release)
8    }]
9    %1 = dma(MM2S, 0) [{
10     use_lock(%memtile_sem, AcquireGreaterEqual)
11     dma_bd(%memtile_buffer : memref<128xi32>, dims = [...])
12     use_lock(%memtile_sem, Release)
13   }]
14   end
15 }
```

Listing 27 `objectfifo` operations for abstractly managing DMA schedules and synchronization. Here the `objectfifo`s connect `tile_0_0` and `tile_0_2` (just as in Listing 25) but include the definitions of the necessary buffers and semaphores (through an analysis pass called `--aie-objectFifo-stateful-transform`).

```
1  %tile_0_0 = tile(0, 0)
2  %tile_0_2 = tile(0, 2)
3  objectfifo @in(%tile_0_0, {%tile_0_2}, 2)
4    : !objectfifo<memref<128xi32>>
5  objectfifo @out(%tile_0_2, {%tile_0_0}, 2)
6    : !objectfifo<memref<128xi32>>
7  %core_0_2 = core(%tile_0_2) {
8    scf.for %arg0 = %c0 to %c4294967295 step %c1 {
9      scf.for %arg1 = %c0_0 to %c4 step %c1_1 {
10       %0 = objectfifo.acquire @out(Produce, 1)
11       %1 = objectfifo.subview.access %0[0]
12       %2 = objectfifo.acquire @in(Consume, 1)
13       %3 = objectfifo.subview.access %2[0]
14       func.call @scale_int32(%3, %1)
15       objectfifo.release @in(Consume, 1)
16       objectfifo.release @out(Produce, 1)
17     }
18   }
19   aie.end
20 }
```

Listing 28 Stream switch manual configuration corresponding to the automatically routed flow in 25.

```
%tile_0_0 = tile(0, 0)
%tile_0_1 = tile(0, 1)
%tile_0_2 = tile(0, 2)
%switchbox_0_0 = switchbox(%tile_0_0) {
  connect<South : 3, North : 0>
  connect<South : 7, North : 1>
  connect<North : 0, South : 2>
}
%shim_mux_0_0 = shim_mux(%tile_0_0) {
  connect<DMA : 0, North : 3>
  connect<DMA : 1, North : 7>
  connect<North : 2, DMA : 0>
}
%switchbox_0_1 = switchbox(%tile_0_1) {
  connect<South : 0, DMA : 0>
  connect<DMA : 0, North : 0>
  connect<South : 1, DMA : 1>
  connect<DMA : 1, North : 1>
  connect<North : 0, DMA : 2>
  connect<DMA : 2, South : 0>
}
%switchbox_0_2 = switchbox(%tile_0_2) {
  connect<South : 0, DMA : 0>
  connect<South : 1, DMA : 1>
  connect<DMA : 0, South : 0>
}
```

Stream Routing

While configuring stream switches explicitly is supported (see Listing 28), most programs will specify flows between endpoints (using `flow` operations) and rely on automatic routing to configure intermediate switches. The MLIR-AIE compiler currently supports two routers: a fast but approximate label-setting router (Dreyfus, 1969) (based on Dijkstra's shortest path algorithm) and an optimal (but more costly) router that solves the congestion-aware traffic assignment problem. Briefly[8], we define

- $(i, j) \in A$: connections between switches;
- h^{π}: all possible stream flows, indexed by possible paths $\pi \in \Pi(r, s)$ from path-endpoint

8. See (Boyles et al., 2023) for a much more in-depth review.

(source) r to path-endpoint (target) s;

- d^{rs}: number of flows required to route the required streams from r to s;
- x_{ij}: magnitude of flow across a connection (i, j);
- $t_{ij}\left(x_{ij}\right)$: the link performance across connection (i, j);
- c_{ij}: capacity for connection (i, j).

where our link performance function is

$$t_{ij}\left(x_{ij}\right) := 1 + \alpha\left(\frac{x_{ij}}{c_{ij}}\right)^{\beta}$$

similar to the Bureau of Public Roads (BPR) link performance function (Gore et al., 2023). Then, the congestion-aware traffic assignment optimization problem is

$$\min_{h^{\pi},\, \pi \in \Pi(r,s)} \sum_{x_{ij} \in h^{\pi}} t_{ij}\left(x_{ij}\right) \cdot x_{ij} \tag{5.1}$$

$$\text{s.t. } x_{ij} = \sum_{\pi \in \Pi} \delta_{ij}^{\pi} h^{\pi} \tag{5.2}$$

$$d^{rs} = \sum_{\pi \in \Pi(r,s)} h^{\pi} \tag{5.3}$$

$$x_{ij} \in \left\{0, \ldots, c_{ij}\right\} \tag{5.4}$$

where (5.1), the objective, models Total System Travel Time, i.e., distance times time aggregated across all flows, (5.3) represents flow conservation constraints, (5.3) enforces the requirement that all flows should be routed successfully, and (5.4) enforces capacity constraints on each flow (as determined by the number of available channels in each stream switch). The compiler currently includes two solver implementations, an ILP problem using

Gurobi (Gurobi Optimization, LLC, 2023) and as a CP-SAT problem using OR-tools (Perron et al., 2023), and supports extensions ("bring-your-own-solver").

5.2.2 Language frontend

The language frontend for our toolchain is a Python eDSL, implemented on top of the Python bindings infrastructure in MLIR (Levental et al., 2023). Using this MLIR functionality, we are able to generate Python bindings to the various operations and attributes in the AIE dialect "for free", as well as bindings to upstream dialects that interoperate with the AIE dialect. This enables users to fully specify AIE programs (including stream switch, DMA, semaphore configuration code **and** single core code) in Python. In addition, we extend the automatically generated bindings in various ways to provide the functionality discussed in Section 5.1.2, namely broadcasting semantics for routing, DMA configuration.

Consider the program in Listing 29. This simple example demonstrates both the stream broadcasting and metaprogramming features available in our language frontend. The program represents a 4×4 design (4 columns of 4 rows of compute tiles) wherein the memtile in each column broadcasts successive 4×4 tiles of a 4×16 "fat row" to the entire column (i.e., the successive rows of 5.5 on page 116). to recognize the broadcast semantics, It is important to understand that `tiles`, an instance of `TileArray`, obeys conventional slicing semantics. So `tiles[:, 1]` is an "array" with shape $(4, 1)$, while `tiles[:, 2:]` is an "array" with shape $(4, 4)$, and `>>` is a binary operator that supports broadcasting its operands. Thus, `tiles[:, 1] >> tiles[:, 2:]` instantiates

```
flow(%tile_k_1, DMA : 0, %tile_k_j, DMA : 0)
```

operations for $j \in \{2, 3, 4, 5\}$ for each $k \in \{0, 1, 2, 3\}$, i.e., from each memtile in each column (which occupy row 1) to all of the core tiles in the same column. Note that broadcast implies one-to-many stream connections (e.g., one memtile to many core tiles) but not the

Listing 29 Python language frontend with broadcast semantics for stream routing and metaprogramming features; lines 9-12 demonstrate broadcast routing semantics; lines 13, 29, and 35 demonstrate our mapping from Python function scopes to MLIR regions; lines 38, 39, and 41 demonstrate our metaprogramming facilities.

```
1  n_tiles, m, n = 4, 4, 4
2  K = n_tiles * m * n
3  channel = 0
4  @device(ipu)
5  def ipu():
6    tiles = TileArray(
7      cols=[0, 1, 2, 3],
8      rows=[0, 1, 2, 3, 4, 5]
9    )
10   tiles[:, 0] >> tiles[:, 1]
11   tiles[:, 1] >> tiles[:, 2:]
12   tiles[:, 1] << tiles[:, 2:]
13   for t in tiles[:, 1]:
14     @memtile_dma(t.tile)
15     def dma():
16       mem_buffer = buffer(t.tile, (K,), T.i32())
17       mem_sem = lock(t.tile, init=0)
18       receive_bd(channel, mem_sem, mem_buffer,
19         acq_action=Acquire, acq_val=0,
20         dims=[(4, 4), (4, 16), (4, 1)],
21       )
22       send_bd(channel, mem_sem, mem_buffer,
23         acq_action=Acquire, acq_val=1, rel_val=0,
24         repeat_count=n_tiles - 1, len=m * n,
25         iter=bd_dim_layout(size=n_tiles, stride=m * n),
26       )
27   for t in tiles[:, 2:]:
28     core_sem = lock(t.tile, init=0)
29     core_buffer = buffer(t.tile, (m, n), T.i32())
30     @mem(t.tile)
31     def dma():
32       receive_bd(channel, core_sem, core_buffer,
33         acq_action=Acquire, acq_val=0,
34         dims=[(4, 4), (4, 1)],
35       )
36     @core(t.tile)
37     def core():
38       x = memref.alloc(m, n, T.i32())
39       with hold_lock(core_sem):
40         for i in range(m):
41           for j in range_(i, n):
42             linalg.add(core_buffer, x, core_buffer)
```

Listing 30 Broadcast `flow`s corresponding to `tiles[0, 1] >> tiles[0, 2:]` and `tiles[0, 1] << tiles[0, 2:]`. Note that in the former, all flows originate from channel 0 (and connect to channel 0), while in the latter all flows originate from channel 0 **but connect to unique channels.**

```
flow(%tile_0_1, DMA : 0, %tile_0_2, DMA : 1)
flow(%tile_0_1, DMA : 0, %tile_0_3, DMA : 0)
flow(%tile_0_1, DMA : 0, %tile_0_4, DMA : 3)
flow(%tile_0_1, DMA : 0, %tile_0_5, DMA : 0)
```

`tiles[0, 1] >> tiles[0, 2:]`

```
flow(%tile_0_2, DMA : 0, %tile_0_1, DMA : 1)
flow(%tile_0_3, DMA : 2, %tile_0_1, DMA : 2)
flow(%tile_0_4, DMA : 0, %tile_0_1, DMA : 3)
flow(%tile_0_5, DMA : 1, %tile_0_1, DMA : 4)
```

`tiles[0, 1] << tiles[0, 2:]`

inverse (many-to-one). Therefore, the language frontend automatically assigns DMA channels correctly to implement true broadcast: left operand connections all originate from the same source DMA channel. The same semantics apply to the `<<` operator (operands are broadcast), however in `tiles[:, 1] << tiles[:, 2:]` **the left-hand operand corresponds to the destination of the flow, and connections are made to unique DMA channels** instead of the same channel. See Listing 30 for the actual `flow` operations that are instantiated.

Listing 29 also demonstrates two more important language features. Firstly, notice the decorators `@memtile_dma`, `@mem`, and `@core` take as inputs `tile` objects, i.e., handles to `%tile_i_j` SSA values, and immediately instantiate[9] the corresponding AIE dialect operations. Such decorators enable specifying all MLIR "region-bearing" operations[10] and are used extensively in our language frontend for the many such operations in the AIE dialect

9. These decorators, which use an upstream feature called `region_op`, immediately execute the function they decorate. Thus, effectively, what you see is what you get, with respect to the MLIR that is generated.

10. Yes, even multi-region operations.

and other, upstream, dialects. Secondly, notice on lines 40 and 42 there are two subtly different `for` loops. The `for i in range(m)` on line 40 is a conventional Python for loop; it will execute its body `m` times, i.e., emit `linalg.fill ins(%i) outs(%alloc : memref<4x4xi32>)` and execute the inner `for` loop. In contrast, the `for i in range_(n)` on line 42 **is not a Python for loop** and will only execute its body once; it is in fact a constructor for the `scf.for` operation (from the upstream `scf` dialect). Thus, line 42 will have the effect of emitting

```
scf.for %j = %c0 to %c4 step %c1 {
  linalg.add ins(%core_buffer, %x)
             outs(%core_buffer : memref<4x4xi32>)
}
```

In combination, m `linalg.fill`s and `scf.for`s are emitted, i.e., a further partitioning of the work in each core. It is important to note that the Python `for` interoperates with the MLIR operation vis-a-vis its loop index (`%i`). See Listing 31 for the fully "unrolled" representation. We believe this ability to mix metaprogramming and actual DSL primitives in one language seamlessly enables complex programs to be written concisely while maintaining complete transparency for the sophisticated user; all such "macros" are implemented purely in Python and are thus easily investigated and extended.

It is important to note that much of the upstream functionality necessary for building our downstream language frontend, such as automatically typed types[11] and attributes[12], `region_op`[13], enum generation[14], value builders[15] and casters[16], has been de-

11. https://reviews.llvm.org/D150927
12. https://reviews.llvm.org/D151840
13. https://github.com/llvm/llvm-project/pull/75673
14. https://reviews.llvm.org/D157934
15. https://github.com/llvm/llvm-project/pull/68308
16. https://github.com/llvm/llvm-project/pull/69644

Listing 31 Fully "unrolled" double loop corresponding to the core code in Listing 29.

```
%core_3_4 = core(%tile_3_4) {
  %alloc = memref.alloc() : memref<4x4xi32>
  use_lock(%core_sem, AcquireGreaterEqual)
  scf.for %arg0 = %c0 to %c4 step %c1 {
    linalg.add ins(%core_buffer, %alloc)
               outs(%core_buffer : memref<4x4xi32>)
  }
  scf.for %arg0 = %c1 to %c4 step %c1 {
    linalg.add ins(%core_buffer, %alloc)
               outs(%core_buffer : memref<4x4xi32>)
  }
  scf.for %arg0 = %c2 to %c4 step %c1 {
    linalg.add ins(%core_buffer, %alloc)
               outs(%core_buffer : memref<4x4xi32>)
  }
  scf.for %arg0 = %c3 to %c4 step %c1 {
    linalg.add ins(%core_buffer, %alloc)
               outs(%core_buffer : memref<4x4xi32>)
  }
  use_lock(%core_sem, Release)
}
```

veloped and upstreamed by us over the course of this work. Thus, we have made available the same techniques for other language frontends and compiler projects.

Runtime and Distribution

The runtime facilities of our language frontend enable specifying, compiling (using SCCs), loading, and launching AIE programs from Python, including host-device data buffer communication. We use the Python Buffer Protocol[17], in concert with NumPy APIs, to provide NumPy compatible views off host-side data. This enables easily generating input data for tests and quick experimentation. In addition, through NumPy's compatibility with PyTorch and TensorFlow tensors (and vice-versa), we are able to quickly and efficiently load various weights tensors from fully trained DNNs. See Listing 32 for an example runtime script. Finally, the compiler and language toolchain are readily packaged as `pip install`-able wheels.

17. `https://peps.python.org/pep-0688/`

Listing 32 Example runtime script for compiling, loading, and launching an AIE program (represented by `module`). Line 7 maps device accessible host memory to buffers that support the Python memory buffer protocol, line 11 then "casts" those buffers to `np.ndarray`s, lines 15-16 initializes the memory mapped buffers with data, lines 18-22 initiates host-device communication. It then launches the kernel on the device (thereby executing the AIE program).

```
1  # module defined above...
2  compile(module)
3  buffer_args = [f"arg_{c}" for c in col]
4  xclbin_path = make_xclbin(module)
5  with FileLock("/tmp/ipu.lock"):
6    xclbin = XCLBin(xclbin_path)
7    buffers = xclbin.mmap_buffers(
8      [(K,)] * len(buffer_args), np.int32
9    )
10
11   arrays = list(map(np.asarray, buffers))
12   for col in cols:
13     A = np.random.randint(0, 10, (K,), dtype=np.int32)
14     C = np.zeros((K,), dtype=np.int32)
15     np.copyto(arrays[2 * col], A)
16     np.copyto(arrays[2 * col + 1], C)
17
18   xclbin.sync_buffers_to_device()
19   xclbin.run()
20   print("Running kernel")
21   xclbin.wait(30)
22   xclbin.sync_buffers_from_device()
23
24   for i, w in enumerate(arrays):
25     print(buffer_args[i], w)
```

We note this makes them easily deployable and compatible with various use-cases, including production systems and student and researcher environments i.e., the toolchain is usable within a Jupyter notebook environment.

5.3 Evaluation

We evaluate our toolchain by implementing GEMM programs according to the three approaches in Section 5.1.1, namely inner-product, outer-product, and row-wise. We then benchmark our programs on a Ryzen AI device and compare the performance, in terms of throughput, against existing approaches to programming AI Engines. Specifically we

compare against CHARM (Zhuang et al., 2023), an open source framework for composing GEMM programs on AI Engine architectures and the publically available RyzenAI SDK[18]. Note, though the RyzenAI SDK is the only AMD supported programming model for Ryzen AI edge devices, it does not enable full user programmability. That is to say, RyzenAI SDK does not enable users to reprogram switches, DMAs, cores, or semaphores and only enables users to adjust kernel dimensions for a fixed set of distributed (binary) kernel implementations.

Therefore, it's important to note the caveats of our evaluation approach:

- CHARM supports only AIE1, the first generation of the AIE architecture, while we develop for AIE2, the second generation of the AIE architecture;
- the kernels distributed with the RyzenAI SDK are primarily suited to attention kernels, i.e., matrix-vector multiplication.

With respect to the limitation in comparing against CHARM, there are some notable differences in the resources available on each architecture generation. More importantly, CHARM only deploys to Versal devices, which are data-center-class devices with much larger arrays of tiles than the Ryzen AI edge device (~400 versus 20). In addition, AIE1 supports fp32 operations natively while AIE2 supports fp32 via emulation through the bfloat16 path; this results in 3-9 bfloat16 MAC operations per fp32[19].

For our single-core microkernel we implement a $16 \times 32 \times 16$ vectorized matrix multiplication using the AIEVec dialect, i.e., a direct mapping to hardware intrinsics. Typically, the AIEVec dialect is not written explicitly by the user but is emitted using a combination of auto-vectorization (from the `affine` dialect) and conversion from `vector` dialect. Despite this, in keeping with our goal of enabling programming at all levels of abstraction, our

18. `https://github.com/amd/RyzenAI-SW`

19. Depending on the choice of `AIE2_FP32_EMULATION_ACCURACY` (AI Engine-ML Intrinsics User Guide, 2023); herein we use the default `AIE2_FP32_EMULATION_ACCURACY_SAFE` which require nine MAC opera-

Listing 33 Core kernel for a $16 \times 32 \times 16$ vectorized matrix multiplication.

```
1  @func.func
2  def matmul(A, B, C):
3    for j in range_(0, 16):
4      c_vec = upd(vec16float, C, [j, c0])
5      accum = ups(vec16accfloat, c_vec)
6      for k in range_(0, 32, 16):
7        a_vec = upd(vec16float, A, [j, k])
8        for i in range(0, 16):
9          broad_a = broadcast(vec16float, a_vec, idx=i)
10         b_vec = upd(vec16float, B, [k + i, c0])
11         accum = mac_elem(vec16accfloat, broad_a,
12                          b_vec, accum)
13       shift_round_sat = srs(vec16float, accum, 0)
14       vector.transfer_write(shift_round_sat, C,
15         [j, c0], permutation_map=perm_map,
16       )
```

language frontend supports writing AIEVec for users for whom it is important to extract maximum possible performance. See Listing 33. We point out again that our approach to metaprogramming allows users to perform transformations such as loop unrolling explicitly; the loops on lines 3 and 6 are again `scf.for` constructors while the loop on line 8 is a conventional Python loop that unrolls itself when the IR is emitted (thereby producing a loop nest that can be effectively optimized by the single core compiler).

Figure 5.6 shows the results of our comparison against CHARM. We see that CHARM outperforms by over 10x at almost all matrix dimensions but recall CHARM programs use 384 AIE1 core tiles while our programs use only 16 AIE2 core tiles. In addition, the AIE1 architecture supports 8 fp32 MACs per clock cycle while AIE2 supports $128/9 = 14$ MACs per clock cycle i.e., only a factor 1.75 more MACs. In addition, Versal devices with AIE1 architectures support higher bandwidth to core tiles than Ryzen AI devices. Given these differences, the performance gap between the two frameworks can reasonably be said to be the result of asymmetrical hardware resources.

On the contrary, our matrix multiplication implementations, which exploit our program-

tions to perform $a * b$ due to three bfloat16 splits each operand.

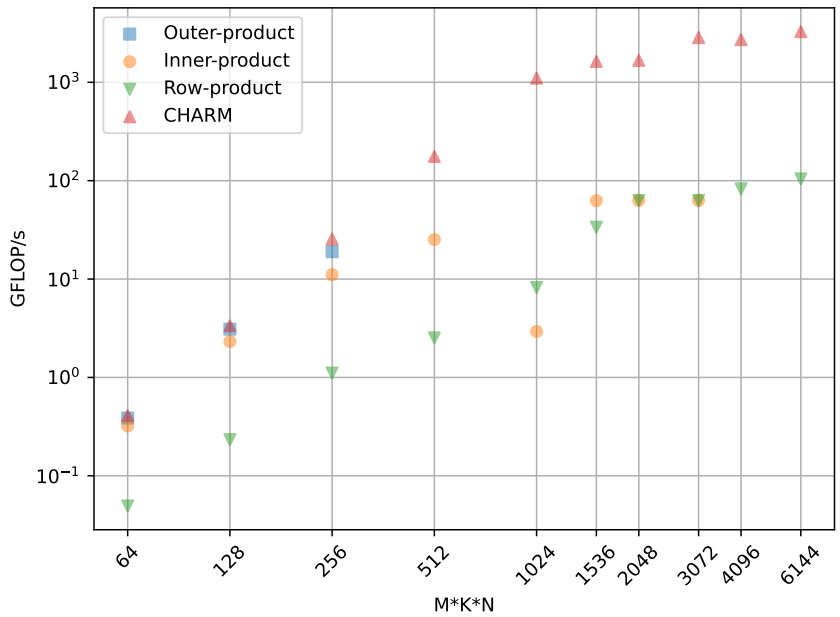


Figure 5.6: Performance as a function of GEMM matrix characteristic dimension, i.e., $A_{M\times K}$, $B_{K\times N}$ with $M = K = N$. CHARM indicates (Zhuang et al., 2023) while Outer-product, Inner-product, and Row-product indicate which of the three dataflow strategies discussed in Section 5.1.1 is being compared. Note, omitted observations are due to core tiles limited local memory failing to accommodate storage requirements for the respective dataflow approaches.

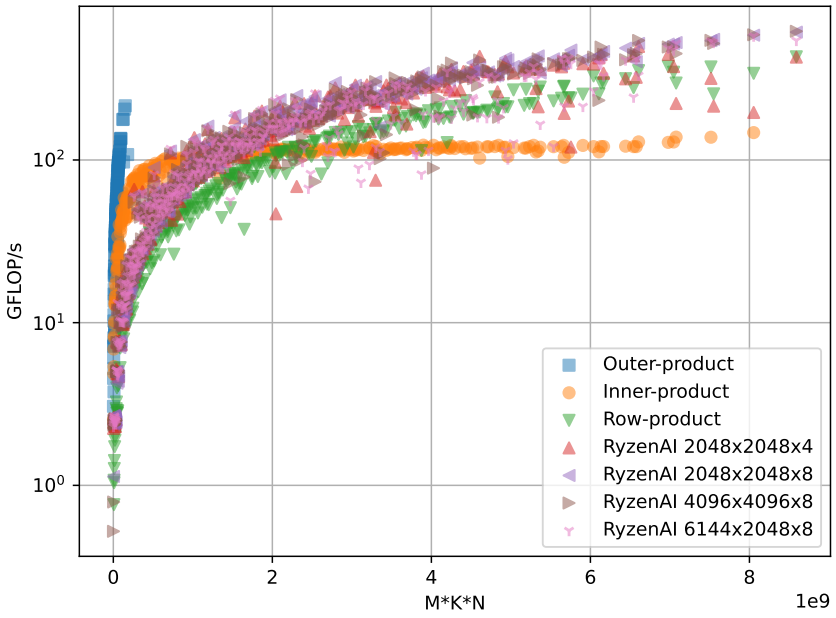


Figure 5.7: Performance as a function of GEMM matrix characteristic dimension compared with RyzenAI SDK distributed kernels (NB: $M \times K \times n$ indicates the kernel is best suited to perform a $M \times K$-matrix-n-row-vector multiplication).

ming model but are otherwise unoptimized, nearly achieve parity with RyzenAI SDK (see Figure 5.7). It's important to underline the significant differences between our programming model and the RyzenAI SDK: the RyzenAI SDK lacks the capability for end-user reconfigurability of any aspect of the kernels, a feature that our model offers. The kernels distributed by the SDK have fixed configurations for stream switches, DMAs, and singular (but parameterized) core programs. The core programs are also highly optimized microkernels but only for specific input dimensions. It stands to reason that if we further optimized our single-core microkernel (see Listing 33) by refining our vectorization strategy, we would meet and exceed their performance metrics. Indeed, the converse is also true: if developers of the SDK were to rearchitect their kernels for the input dimensions we tested, they might further extend their lead. The key distinction is that our programming model empowers the user to optimize and refine their kernels, while the SDK programming model relies entirely on the expertise of the internal teams at AMD.

We make some observations about the relative performance of the three dataflow strategies. Firstly, note that the outer-product approach is the most performant but is unable to support large input matrix sizes; recall in Section 5.1.1 we noted that the outer-product approach achieves excellent data reuse but requires each compute element having enough space for the (sub)result matrix; the formula for outer-product matrix multiplication, $C_k = A[:, k] \otimes B[k, :]$, allows for local accumulation of partial sums but each partial sum $A[:, k] \otimes B[k, :]$ has dimensions equal to the (sub)result matrix. Since our implementation of outer-product partitions the result matrix into 4×4 blocks (each block assigned to one core tile), at 4 bytes per matrix element, each core tile can hold at most a $A_{64\times64} \times B_{64\times64}$ matrix multiplication: $3 \times (64 \times 64) \times 4\text{B} = 48\text{KiB} < 64\text{KiB}$. Secondly, the inner-product approach, having the poorest operand reuse of the three approaches, performs most poorly; recall that the inner-product requires each compute element having access to $O(mk + kn)$ operands (over the course of evaluating the complete matrix multiplication). However, core tiles have limited local memory, operands must be reloaded, at worst, $O(k)$ times. Finally, row-wise product performs relatively well at small and medium matrix sizes due to its (relatively) good reuse characteristics but fails to remain performant for larger sizes. We hypothesize this is due to bank conflicts in the memtiles: each memtile has 512KiB of memory consisting of 16 banks of 32KiB with each bank being single ported, i.e., allowing only 1 read or 1 write every cycle. Thus, depending on operand sizes and initial data layout (when loaded from the host) it is possible to incur significant concurrent accesses to the same bank by multiple DMA channels.

5.4 Related Work

Though there is a dearth of related work targeting AIE architectures specifically, there has been ample work on programming models for CGRAs in general. The majority of this work either addresses CGRAs deployed as softcores to FPGAs (Verdoscia et al., 2016, Heid

et al., 2016) or includes the co-design of the architecture as well (Hsu et al., 2022, Rucker et al., 2024, 2021). Such approaches allow for more unification between the programming model because the design of the CGRA itself can be adjusted. Thus, our work distinguishes itself by developing for an existing, fixed, CGRA architecture and still achieving a unified programming model. Note, in addition to CHARM, we mention the recent work GEMM for AIE1: MaxEVA (Taka et al., 2023). We did not compare to MaxEVA because the goals of that work were obtaining high-performance GEMMs through optimal *placement* of kernels on the array; since Ryzen AI devices have relatively fewer core tiles, they necessarily require using all, thus preventing us from exploring placement strategies.

In addition, ample research and development has generally focused on building DSLs (in various languages) that enable developers to target hardware primitives more effectively. For example, AMD's Composable Kernel (CK) library[20] provides a framework for writing machine learning workloads kernels that work across multiple architectures. The framework takes a hierarchical C++ template approach, where templates are subclassed and partially instantiated as a composition mechanism (see Listing 34). The primary disadvantage of this approach is that while the C++ templating metalanguage is Turing Complete (Veldhuizen, 2003), it is a wholly distinct language (from C++) and unfamiliar to most users (even the developers of CKL). This approach leads to unbounded template parameter lists because of a lack of almost any inference in the signatures (see Listing 35). Additionally, errors in using the templates become extremely difficult to debug (C++ compilers are well-known to produce poor error messages for templating failures). A similar approach is taken by Google's XNNPACK library[21] but substitutes the standard C++ templating language with a bespoke, proprietary literal template language. That is to say, XNNPACK kernels are designed by interspersing external language primitives and standard C/C++ (see Listing 36),

20. `https://github.com/ROCm/composable_kernel`

21. `https://github.com/google/XNNPACK`

Listing 34 GEMM kernel template from AMD's Composable Kernel Library. Note `DeviceGemmDl` actually inherits from `DeviceGemm`.

```
template <
  typename ADataType,
  typename BDataType,
  typename CDataType,
  typename AccDataType,
  typename ALayout,
  typename BLayout,
  typename CLayout,
  typename AElementwiseOperation,
  typename BElementwiseOperation,
  typename CElementwiseOperation,
  GemmSpecialization GemmSpec,
  index_t BlockSize,
  index_t MPerBlock,
  index_t NPerBlock,
  index_t K0PerBlock,
  index_t K1,
  index_t M1PerThread,
  index_t N1PerThread,
  index_t KPerThread,
  typename M1N1ThreadClusterM1Xs,
  typename M1N1ThreadClusterN1Xs,
  typename ABlockTransferThreadSliceLengths_K0_M0_M1_K1,
  typename ABlockTransferThreadClusterLengths_K0_M0_M1_K1,
  typename ABlockTransferThreadClusterArrangeOrder,
  typename ABlockTransferSrcAccessOrder,
  typename ABlockTransferSrcVectorTensorLengths_K0_M0_M1_K1,
  typename ABlockTransferSrcVectorTensorContiguousDimOrder,
  typename ABlockTransferDstVectorTensorLengths_K0_M0_M1_K1,
  typename BBlockTransferThreadSliceLengths_K0_N0_N1_K1,
  typename BBlockTransferThreadClusterLengths_K0_N0_N1_K1,
  typename BBlockTransferThreadClusterArrangeOrder,
  typename BBlockTransferSrcAccessOrder,
  typename BBlockTransferSrcVectorTensorLengths_K0_N0_N1_K1,
  typename BBlockTransferSrcVectorTensorContiguousDimOrder,
  typename BBlockTransferDstVectorTensorLengths_K0_N0_N1_K1,
  typename CThreadTransferSrcDstAccessOrder,
  index_t CThreadTransferSrcDstVectorDim,
  index_t CThreadTransferDstScalarPerVector,
  enable_if_t<
    is_same_v<AElementwiseOperation, ck::tensor_operation::element_wise::PassThrough> &&
    is_same_v<BElementwiseOperation, ck::tensor_operation::element_wise::PassThrough> &&
    is_same_v<CElementwiseOperation, ck::tensor_operation::element_wise::PassThrough>,
    bool> = false>
struct DeviceGemmDl : public DeviceGemm<ALayout, BLayout, CLayout, ADataType, BDataType,
                                        CDataType, AElementwiseOperation,
                                        BElementwiseOperation, CElementwiseOperation>
```

Listing 35 Instantiated CKL GEMM kernel using the `DeviceGemmDl` in Listing 34.

```
using DeviceGemmInstance = ck::tensor_operation::device::DeviceGemmDl<
    ADataType, BDataType, CDataType, AccDataType, ALayout, BLayout, CLayout,
    AElementOp, BElementOp, CElementOp, GemmDefault, 256, 128, 128, 16, 2, 4, 4,
    1, S<8, 2>, S<8, 2>, S<2, 1, 4, 2>, S<8, 1, 32, 1>, S<0, 3, 1, 2>,
    S<0, 3, 1, 2>, S<1, 1, 4, 1>, S<0, 3, 1, 2>, S<1, 1, 4, 2>, S<2, 1, 4, 2>,
    S<8, 1, 32, 1>, S<0, 3, 1, 2>, S<0, 3, 1, 2>, S<1, 1, 4, 1>, S<0, 3, 1, 2>,
    S<1, 1, 4, 2>, S<0, 1, 2, 3, 4, 5>, 5, 4>;
```

Listing 36 GEMM kernel template from Google's XNNPack library. Note all `$` escaped syntax indicates a proprietary template language.

```
void xnn_bf16_gemm_minmax_ukernel_${MR} x${NR} c8__neonfma_zip(
    size_t mr, size_t nc, size_t kc, const void *restrict a, size_t a_stride,
    const void *restrict w_ptr, void *restrict c, size_t cm_stride,
    size_t cn_stride,
    const union xnn_bf16_minmax_params params[restrict XNN_MIN_ELEMENTS(1)]) {
  assert(mr != 0);
  assert(mr <= ${MR});
  assert(nc != 0);
  assert(kc != 0);
  assert(kc % sizeof(uint16_t) == 0);
  assert(a != NULL);
  assert(w_ptr != NULL);
  assert(c != NULL);

  const uint16_t* a0 = (const uint16_t*) a;
  uint16_t* c0 = (uint16_t*) c;
  $for M in range(1, MR):
    const uint16_t* a${M} = (const uint16_t*) ((uintptr_t) a${M-1} + a_stride);
    uint16_t* c${M} = (uint16_t*) ((uintptr_t) c${M-1} + cm_stride);
    $if M % 2 == 0:
      if XNN_UNPREDICTABLE(mr <= ${M}) {
        a${M} = a${M-1};
        c${M} = c${M-1};
      }
    $elif M + 1 == MR:
      if XNN_UNPREDICTABLE(mr != ${M+1}) {
        a${M} = a${M-1};
        c${M} = c${M-1};
      }
    $else:
      if XNN_UNPREDICTABLE(mr < ${M+1}) {
        a${M} = a${M-1};
        c${M} = c${M-1};
      }
  ...
}
```

Listing 37 GEMM kernel from Intel's oneDNN library. Note this kernel operates on registers (`const Ymm &reg00, const Ymm &reg01, const Ymm &reg02, ...`) and emits literal assembly at runtime (e.g., `vaddps(reg00, reg00, reg12)`).

```
void kernel(int unroll_m, int unroll_n, bool isLoad1Unmasked,
            bool isLoad2Unmasked, bool isDirect, bool isCopy, bool useFma,
            const Ymm &reg00, const Ymm &reg01, const Ymm &reg02,
            const Ymm &reg03, const Ymm &reg04, const Ymm &reg05,
            const Ymm &reg06, const Ymm &reg07, const Ymm &reg08,
            const Ymm &reg09, const Ymm &reg10, const Ymm &reg11,
            const Ymm &reg12, const Ymm &reg13, const Ymm &reg14,
            const Ymm &reg15, const Ymm &reg16, const Ymm &reg17,
            const Ymm &reg18, const Ymm &reg19, const Ymm &reg20,
            const Ymm &reg21, const Ymm &reg22, const Ymm &reg23) {
  ...
  align(16);

  L(labels[5]);
  if (unroll_m == 16) {
    if (unroll_n <= 3) {
      vaddps(reg00, reg00, reg12);
      vaddps(reg01, reg01, reg13);
      vaddps(reg02, reg02, reg14);
      vaddps(reg06, reg06, reg18);
      vaddps(reg07, reg07, reg19);
      vaddps(reg08, reg08, reg20);
    }
  }

  ...

  test(K, 1);
  jle(labels[6], T_NEAR);
  innerkernel1(unroll_m, unroll_n, isLoad1Unmasked, isLoad2Unmasked, isDirect,
               isCopy, useFma, reg00, reg01, reg02, reg03, reg04, reg05, reg06,
               reg07, reg08, reg09, reg10, reg11);
  align(16);

  ...
}
```

and then the final C/C++ source is generated by passing the templatized source through a preprocessor. The disadvantages of this approach are apparent: the non-standard template language is even less understood than standard templates, and the preprocessor provides limited feedback about errors during the generation step (relying on the downstream compiler to supply authoritative error messages). Finally, Intel's oneDNN library[22], which aims to provide optimized kernels for various CPU architectures, takes a relatively novel approach: core kernels in oneDNN are implemented by Just-in-Time (JIT) compiling an API-centric DSL to vector instructions (not intrinsics) using the Xbyak JIT assembler[23]. This approach is very similar to ours in spirit (our frontend, similarly JIT translates Python to MLIR) and appealing from the perspective of a competent developer because it enables using the host language (C++) while retaining access to the lowest level APIs of the hardware (assembly). While MLIR currently supports directly emitting instructions for some architectures (e.g., ARM-Neon[24], ARM-SVE[25] and ARM-SME[26]), MLIR-AIE does not currently support directly emitting instructions for the vector VLIW processors.

5.5 Conclusion

We presented an "end-to-end" programming model for AMD's AI Engine such that a developer can design a dataflow, program the individual PEs, configure the device, launch the program, and manage host-device communications (vis-à-vis memory buffers) all in one Python script (or Jupyter notebook). We then evaluated our programming model against the current state-of-the-art by implementing three different dataflow approaches to matrix multiplication and measuring their performance, in terms of GFLOP/s, on a Ryzen AI edge

22. `https://github.com/oneapi-src/oneDNN`
23. `https://github.com/herumi/xbyak`
24. `https://mlir.llvm.org/docs/Dialects/ArmNeon`
25. `https://mlir.llvm.org/docs/Dialects/ArmSVE`
26. `https://mlir.llvm.org/docs/Dialects/ArmSME`

device. After accounting for resource differences, we observe that our programming model enables users to implement dataflow programs with performance comparable to the state of the art while benefiting from all the features of a high-level language frontend and a unified end-to-end workflow.

Future work in this area includes several directions:

- **Single-core Compiler Integration**: Currently, our end-to-end approach delegates the code generation for the vector VLIW processor to external single-core compilers. The reasons for this are not technical, and in the future, when some of these external compilers become open source, we hope to integrate with them more closely, even possibly enabling emitting the vector VLIW instructions directly (as mentioned in Section 5.4).

- **Framework Integration**: We currently support upstream deep learning frameworks (such as PyTorch and TensorFlow) but only through the IREE runtime. While this approach is fully functional and robust, it incurs a great deal of incidental complexity in the building and managing both the frameworks and IREE[27] itself. In principle, it should be straightforward to integrate directly with both PyTorch and TensorFlow as they each now have MLIR dialects as egress IRs. We plan to explore these integration paths in the near future.

27. `https://iree.dev`

CHAPTER 6

CONCLUSION

This dissertation has made significant contributions to deep learning optimization, compiler design, and embedded domain-specific languages. The primary contribution lies in creating and implementing an eDSL for representing, optimizing, and executing deep learning models on a novel accelerator architecture, specifically AMD AI Engine architectures. This eDSL, complemented by core, reusable infrastructure components that have been integrated into MLIR, validates the core thesis of the dissertation: that it is possible to design a language and programming model that enables representing deep learning models simultaneously at multiple levels of abstraction, generating performant, target-specific, implementations, and executing those models on novel accelerator architectures. The development towards this primary contribution has proceeded through four distinct investigations into deep learning language and compiler design, each offering valuable insights and technical innovations.

In *Memory Planning for Deep Neural Networks*, the investigation into memory planning for deep neural networks shed light on the inefficiencies inherent in existing frameworks and memory allocators, leading to the development of a novel allocation capture technique, ILP and CP-SAT models of memory allocation, and a working implementation of an allocator that significantly reduces thread contention and improves DNN performance. This work emphasized the importance of exposing implementation-specific details for achieving crucial optimizations, which traditional frameworks often obscure.

In *`BraggHLS`: High-Level Synthesis for Low-latency Deep Neural Networks for Experimental Science*, the development of `BraggHLS` showcased the potential of high-level synthesis techniques for translating deep neural network representations into FPGA-deployable code. Notably, this project highlighted the necessity of empowering users with control over compiler optimization passes, enabling them to leverage domain-specific knowledge effectively. This work produced an implementation of a Bragg peak detection DNN which achieved

a throughput improvement of 4× over the previous state of the art. This work also produced ILP and CP-SAT models for Shared Operator Scheduling (Kruppe et al., 2021) and a novel lifting approach for enabling domain scientists to perform transformations on low-level representations of DNNs.

In *`nelli`: A Lightweight Frontend for MLIR*, the creation of `nelli`, a lightweight frontend for MLIR, introduced an innovative approach to generating Python-embedded domain-specific languages directly from MLIR dialects. This initiative underscored the importance of providing APIs and intermediate representations that enable direct access to compiler operations and transformations when developing DNNs, thus eliminating the need for high-level frameworks such as PyTorch and TensorFlow during inference. One specifically valuable technical contribution of this work is a novel approach to mapping arbitrarily nested conditionals and loop primitives to various MLIR operations.

Finally, in *An End-to-End Programming Model for AI Engine Architectures*, the validation of the core thesis was demonstrated through the development of an end-to-end programming model for AMD AI Engines, leveraging and extending MLIR to provide a comprehensive language frontend, single-core code generation, runtime management, and optimization capabilities. This endeavor emphasized the integration of language and compiler design to furnish developers with the necessary tools for achieving performant implementations of dataflow programs such as GEMM. This work also yielded ILP and CP-SAT formulations of congestion-aware traffic assignment (Temelcan et al., 2020) for stream routing, a novel stream broadcasting primitive for combining array broadcasting and stream switch configuration and a novel approach to metaprogramming MLIR in Python.

We envision that our work will lead to engagement from expert developers, research scientists and domain experts and produce highly optimized implementations of DNNs for many unique use cases.

REFERENCES

Create placed and routed dcp to cross slr. https://www.rapidwright.io/docs/SLR_Crosser_DCP_Creator_Tutorial.html. Accessed: 2022-10-15.

Torchscript language reference. https://pytorch.org/docs/stable/jit_language_reference_v2.html. Accessed: 2021-09-30.

Ptx and sass assembly debugging. https://docs.nvidia.com/gameworks/content/developertools/desktop/ptx_sass_assembly_debugging.htm, 2015. [Online; accessed 26-March-2023].

Ultrascale architecture dsp slice. Technical report, XiLinx, 2021. URL https://docs.xilinx.com/v/u/en-US/ug579-ultrascale-dsp.

Pyston. https://github.com/pyston/pyston, 2023. [Online; accessed 26-March-2023].

StableHLO: Backward compatible ml compute opset inspired by hlo/mhlo, 2023. URL https://github.com/openxla/stablehlo.

Torch-MLIR: The torch-mlir project aims to provide first class support from the pytorch ecosystem to the mlir ecosystem, 2023. URL https://github.com/llvm/torch-mlir.

Roel Aaij et al. Allen: A high-level trigger on gpus for lhcb. *Computing and Software for Big Science*, 4(1):1–11, 2020.

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016a.

Martín Abadi et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2016b. URL https://arxiv.org/abs/1603.04467.

H. Abelson, G.J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. Electrical engineering and computer science series. MIT Press, 1996. ISBN 9780262011532.

Carlos Affonso, André Luis Debiaso Rossi, Fábio Henrique Antunes Vieira, and André Carlos Ponce de Leon Ferreira de Carvalho. Deep learning for biological image classification. *Expert Systems with Applications*, 85:114–122, 2017. ISSN 0957-4174. doi:https://doi.org/10.1016/j.eswa.2017.05.039. URL https://www.sciencedirect.com/science/article/pii/S0957417417303627.

AI Engine-ML Intrinsics User Guide. Considerations when using emulated FP32 Intrinsics, 2023. URL https://www.xilinx.com/htmldocs/xilinx2023_2/aiengine_ml_intrinsics/intrinsics/group__intr__gpvectorop__mul.html.

Laith Alzubaidi et al. Review of deep learning: Concepts, cnn architectures, challenges, applications, future directions. *Journal of Big Data*, 8(1):1–74, 2021.

Nada Amin and Tiark Rompf. Collapsing towers of interpreters. 2(POPL), dec 2017. doi:10.1145/3158140. URL https://doi.org/10.1145/3158140.

Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International conference on machine learning*, pages 173–182. PMLR, 2016.

Anthony Shaw. Nuitka the python compiler. https://pyjion.readthedocs.io/en/latest/, 2023. [Online; accessed 26-March-2023].

ARM. Arm cortex-a series programmer's guide for armv8-a. https://developer.arm.com/documentation/den0024/a/ch12s05s01. Accessed: 2023-01-30.

Arash Ashari et al. On optimizing machine learning workloads via kernel fusion. 50(8): 173–182, 2015. ISSN 0362-1340. doi:10.1145/2858788.2688521. URL https://doi.org/10.1145/2858788.2688521.

Lan Bai, Lei Yang, and Robert Dick. Memmu: Memory expansion for mmu-less embedded systems. *ACM Trans. Embedded Comput. Syst.*, 8, 04 2009. doi:10.1145/1509288.1509295.

Mihalj Bakator and Dragica Radosav. Deep learning and medical diagnosis: A review of literature. *Multimodal Technologies and Interaction*, 2(3):47, 2018.

Zoltan Baruch. Scheduling algorithms for high-level synthesis. *ACAM Scientific Journal*, 5 (1-2):48–57, 1996.

Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13 (2):31–39, 2010.

Marianne Bellotti. Programming in Z3 by learning to think like a compiler. https://bellmar.medium.com/programming-in-z3-by-learning-to-think-like-a-compiler-401fd46828d5, 2021.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer, 2020.

Paul Blockhaus and Ing David Broneske. *A Framework for Adaptive Reprogramming Using a JIT-Compiled Domain Specific Language for Query Execution.* PhD thesis, Master's thesis. Ottovon-Guericke University Magdeburg, 2022.

Nicolas Bohm Agostini et al. Bridging python to silicon: The soda toolchain. *IEEE Micro*, 2022. doi:10.1109/MM.2022.3178580.

Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605585413. doi:10.1145/1565824.1565827. URL https://doi.org/10.1145/1565824.1565827.

Uday Bondhugula. Polyhedral compilation opportunities in mlir. https://acohen.gitlabpages.inria.fr/impact/impact2020/slides/IMPACT_2020_keynote.pdf, 2020.

Jeff Bonwick. The slab allocator: An object-caching kernel. In *USENIX Technical Conference*, Boston, MA, June 1994. USENIX Association. URL https://www.usenix.org/conference/usenix-summer-1994-technical-conference/slab-allocator-object-caching-kernel.

David Boreham. Malloc() performance in a multithreaded Linux environment. In *USENIX Annual Technical Conference*, San Diego, CA, June 2000. USENIX Association. URL https://www.usenix.org/conference/2000-usenix-annual-technical-conference/malloc-performance-multithreaded-linux.

Stephen D. Boyles, Nicholas E. Lownes, and Avinash Unnikrishnan. *Transportation Network Analysis*, volume 1. 2023. edition 0.91.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Nick Brown, Tobias Grosser, Mathieu Fehr, Michel Steuwer, and Paul Kelly. xdsl: A common compiler ecosystem for domain specific languages.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. ArXiv 2005.14165.

Adam L. Buchsbaum, Howard Karloff, Claire Kenyon, Nick Reingold, and Mikkel Thorup. Opt versus load in dynamic storage allocation. In *35th Annual ACM Symposium on Theory of Computing*, STOC '03, pages 556–564, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581136749. doi:10.1145/780542.780624. URL https://doi.org/10.1145/780542.780624.

Edmund K Burke, Graham Kendall, and Glenn Whitwell. A new placement heuristic for the orthogonal stock-cutting problem. *Operations Research*, 52(4):655–671, 2004.

Jeronimo C. Penha, Lucas B. Silva, Jansen M. Silva, Kristtopher K Coelho, Hector P. Baranda, José Augusto M. Nacif, and Ricardo S. Ferreira. Add: Accelerator design and deploy-a tool for fpga high-performance dataflow computing. *Concurrency and Computation: Practice and Experience*, 31(18):e5096, 2019.

Andrew Canis et al. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Trans. Embed. Comput. Syst.*, 13(2), 2013. ISSN 1539-9087. doi:10.1145/2514740. URL `https://doi.org/10.1145/2514740`.

João M.P. Cardoso, José Gabriel F. Coutinho, and Pedro C. Diniz. Chapter 5 - source code transformations and optimizations. In João M.P. Cardoso, José Gabriel F. Coutinho, and Pedro C. Diniz, editors, *Embedded Computing for High Performance*, pages 137–183. Morgan Kaufmann, Boston, 2017. ISBN 978-0-12-804189-5. doi:https://doi.org/10.1016/B978-0-12-804189-5.00005-3. URL `https://www.sciencedirect.com/science/article/pii/B9780128041895000053`.

George Charitopoulos and Dionisios N. Pnevmatikatos. A cgra definition framework for dataflow applications. In Fernando Rincón, Jesús Barba, Hayden K. H. So, Pedro Diniz, and Julián Caba, editors, *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, pages 271–287, Cham, 2020. Springer International Publishing. ISBN 978-3-030-44534-8.

Junjie Chen, Yihua Liang, Qingchao Shen, Jiajun Jiang, and Shuochuan Li. Toward understanding deep learning framework bugs. *ACM Transactions on Software Engineering and Methodology*, 32(6):1–31, 2023.

Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation, 2017.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost, 2016. URL `https://arxiv.org/abs/1604.06174`.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation*, pages 578–594, 2018.

Tianqi Chen et al. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems, 2015. URL `https://arxiv.org/abs/1512.01274`.

Yishen Chen, Charith Mendis, Michael Carbin, and Saman Amarasinghe. Vegen: a vectorizer generator for simd and beyond. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 902–914, 2021.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers, 2019.

I. Stephen Choi and Yang-Suk Kee. Energy efficient scale-in clusters with in-storage processing for big-data analytics. In *Proceedings of the 2015 International Symposium on Memory Systems*, MEMSYS '15, pages 265–273, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336048. doi:10.1145/2818950.2818983. URL https://doi.org/10.1145/2818950.2818983.

Francois Chollet. *Deep learning with Python*. Simon and Schuster, 2021.

Timothy Colburn and Gary Shute. Abstraction in computer science. *Minds and Machines*, 17(2):169–184, 2007. doi:10.1007/s11023-007-9061-7. URL https://doi.org/10.1007/s11023-007-9061-7.

LHCb Collaboration. Comparison of particle selection algorithms for the LHCb upgrade. Technical report, 2020. URL https://cds.cern.ch/record/2746789.

Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick. Operator strength reduction. *ACM Trans. Program. Lang. Syst.*, 23(5):603–625, sep 2001. ISSN 0164-0925. doi:10.1145/504709.504710. URL https://doi.org/10.1145/504709.504710.

Steve Dai, Gai Liu, and Zhiru Zhang. A scalable approach to exact resource-constrained scheduling based on a joint sdc and sat formulation. In *ACM/SIGDA Intl Symposium on Field-Programmable Gate Arrays*, pages 137–146, 2018. ISBN 9781450356145.

Aminu Da'u and Naomie Salim. Recommendation system based on deep learning methods: a systematic review and new directions. *Artificial Intelligence Review*, 53(4):2709–2748, 2020.

Florent de Dinechin. Reflections on 10 years of flopoco. In *IEEE 26th Symposium on Computer Arithmetic*, pages 187–189, 2019.

Ioannis Deligiannis and George Kornaros. Adaptive memory management scheme for mmu-less embedded systems. pages 1–8, 05 2016. doi:10.1109/SIES.2016.7509439.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

Neil G Dickson, Kamran Karimi, and Firas Hamze. Importance of explicit vectorization for cpu and gpu software performance. *Journal of Computational Physics*, 230(13):5383–5398, 2011.

Stuart E. Dreyfus. An appraisal of some shortest-path algorithms. *Oper. Res.*, 17(3):395–412, jun 1969. ISSN 0030-364X. doi:10.1287/opre.17.3.395. URL https://doi.org/10.1287/opre.17.3.395.

J. Duarte et al. Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation*, 13(07):P07027–P07027, 2018.

Dietmar Ebner, Florian Brandner, Bernhard Scholz, Andreas Krall, Peter Wiedermann, and Albrecht Kadlec. Generalized instruction selection using ssa-graphs. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 31–40, 2008.

Schuyler Eldridge, Prithayan Barua, Aliaksei Chapyzhenka, Adam Izraelevitz, Jack Koenig, Chris Lattner, Andrew Lenharth, George Leontiev, Fabian Schuiki, Ram Sunder, et al. Mlir as hardware compiler infrastructure. In *Workshop on Open-Source EDA Technology (WOSET)*, 2021.

Jason Evans. Scalable memory allocation using jemalloc. *Notes Facebook Eng*, 2011.

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity, 2021. ArXiv 2101.03961.

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

Fabrizio Ferrandi et al. Bambu: an open-source research framework for the high-level synthesis of complex applications. In *58th ACM/IEEE Design Automation Conference*, pages 1327–1330. IEEE, 2021.

Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 4(9), 2018.

Evangelos Georganas, Kunal Banerjee, Dhiraj Kalamkar, Sasikanth Avancha, Anand Venkat, Michael Anderson, Greg Henry, Hans Pabst, and Alexander Heinecke. Harnessing deep learning via a single building block. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 222–233. IEEE, 2020.

Stefanos Georgiou, Maria Kechagia, Tushar Sharma, Federica Sarro, and Ying Zou. Green ai: Do deep learning frameworks have different costs? In *Proceedings of the 44th International Conference on Software Engineering*, pages 1082–1094, 2022.

Jordan Gergov. Algorithms for compile-time memory optimization. In *10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 907–908, 1999.

Sanjay Ghemawat and Paul Menage. TCMalloc: Thread-caching malloc, 2009. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

V V Gligorov and M Williams. Efficient, reliable and fast high-level triggering using a bonsai boosted decision tree. *J. Instrumentation*, 8(02), 2013.

Vladimir Gligorov. Real-time data analysis at the LHC: present and future. In *NIPS Workshop on High-energy Physics and Machine Learning*, volume 42, pages 1–18, 2015.

Gert Goossens, Dirk Lanneer, Werner Geurts, and Johan Van Praet. Design of asips in multi-processor socs using the chess/checkers retargetable tool suite. In *2006 International Symposium on System-on-Chip*, pages 1–4, 2006. doi:10.1109/ISSOC.2006.321968.

Ninad Gore, Shriniwas Arkatkar, Gaurang Joshi, and Constantinos Antoniou. Modified bureau of public roads link function. *Transportation Research Record*, 2677(5):966–990, 2023. doi:10.1177/03611981221138511.

Keith Grainge et al. Square kilometre array: The radio telescope of the xxi century. *Astronomy reports*, 61(4):288–296, 2017.

Licheng Guo et al. Autobridge: Coupling coarse-grained floorplanning and pipelining for high-frequency hls design on multi-die fpgas. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, page 81–92, 2021. ISBN 9781450382182.

Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL `https://www.gurobi.com`.

M. Hammer, K. Yoshii, and A. Miceli. Strategies for on-chip digital data compression for x-ray pixel detectors. *Journal of Instrumentation*, 16(01):P01025–P01025, 2021. doi:10.1088/1748-0221/16/01/p01025. URL `https://doi.org/10.1088%2F1748-0221%2F16%2F01%2Fp01025`.

Momoko Hattori, Naoki Kobayashi, and Ryosuke Sato. Gradual tensor shape checking, 2022. URL `https://arxiv.org/abs/2203.08402`.

SI Hayakawa. 'the art of plain talk'. *American Speech*, 23(2):138–141, 1948.

Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *IEEE International Symposium on High Performance Computer Architecture*, pages 620–629, 2018. doi:10.1109/HPCA.2018.00059.

Kris Heid, Jan Weber, and Christian Hochberger. μstreams: a tool for automated streaming pipeline generation on soft-core processors. In *2016 International Conference on FPGA Reconfiguration for General-Purpose Computing (FPGA4GPC)*, pages 25–30. IEEE, 2016.

Olivia Hsu, Alexander Rucker, Tian Zhao, Kunle Olukotun, and Fredrik Kjolstad. Stardust: Compiling sparse tensor algebra to a reconfigurable dataflow architecture, 2022.

Niall Hurley and Scott Rickard. Comparing measures of sparsity. In *2008 IEEE Workshop on Machine Learning for Signal Processing*, pages 55–60, 2008. doi:10.1109/MLSP.2008.4685455.

Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally,

and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size, 2016a. ArXiv 1602.07360.

Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size, 2016b.

Obi Ike-Nwosu. Inside the python virtual machine, 2015.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015. URL `https://arxiv.org/abs/1502.03167`.

Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. pybind11 - seamless operability between c++11 and python, 2016. https://github.com/pybind/pybind11.

Yangqing Jia. *Learning semantic image representations at a large scale*. University of California, Berkeley, 2014.

Qingye Jiang, Young Choon Lee, and Albert Y Zomaya. The limit of horizontal scaling in public clouds. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 5(1):1–22, 2020.

Tian Jin, Gheorghe-Teodor Bercea, Tung D. Le, Tong Chen, Gong Su, Haruki Imai, Yasushi Negishi, Anh Leu, Kevin O'Brien, Kiyokuni Kawachiya, and Alexandre E. Eichenberger. Compiling onnx neural network models using mlir, 2020. URL `https://arxiv.org/abs/2008.08272`.

Michael Jungmair, André Kohn, and Jana Giceva. Designing an open framework for query optimization and compilation. *Proc. VLDB Endow.*, 15(11):2389–2401, jul 2022. ISSN 2150-8097. doi:10.14778/3551793.3551801. URL `https://doi.org/10.14778/3551793.3551801`.

Kay Hayen. Nuitka the python compiler. `https://nuitka.net/`, 2023. [Online; accessed 26-March-2023].

John Kessenich, Boaz Ouriel, and Raun Krisch. Spir-v specification. *Khronos Group*, 3:17, 2018.

Oleg Kiselyov. Typed tagless final interpreters. *Generic and indexed programming: International spring school, sSGIP 2010, oxford, uK, march 22-26, 2010, revised lectures*, pages 130–174, 2012.

Donald E Knuth. Structured programming with go to statements. *ACM Computing Surveys (CSUR)*, 6(4):261–301, 1974.

Hanna Kruppe, Lukas Sommer, Lukas Weber, Julian Oppermann, Cristian Axenie, and Andreas Koch. Efficient operator sharing modulo scheduling for sum-product network inference on fpgas. In *International Conference on Embedded Computer Systems*, pages 242–258. Springer, 2021.

Bradley C. Kuszmaul. SuperMalloc: A super fast multithreaded malloc for 64-bit machines. In *International Symposium on Memory Management*, pages 41–55, New York, NY, USA, 2015a. Association for Computing Machinery. ISBN 9781450335898. doi:10.1145/2754169.2754178. URL https://doi.org/10.1145/2754169.2754178.

Bradley C. Kuszmaul. Supermalloc: A super fast multithreaded malloc for 64-bit machines. *SIGPLAN Not.*, 50(11):41–55, jun 2015b. ISSN 0362-1340. doi:10.1145/2887746.2754178. URL https://doi.org/10.1145/2887746.2754178.

Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, pages 754–768, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369381. doi:10.1145/3352460.3358252. URL https://doi.org/10.1145/3352460.3358252.

Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA, 2015a. Association for Computing Machinery. ISBN 9781450340052. doi:10.1145/2833157.2833162. URL https://doi.org/10.1145/2833157.2833162.

Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–6, 2015b.

Rasmus Munk Larsen and Tatiana Shpeisman. Tensorflow graph optimizations, 2019.

Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore's law, 2020a.

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

Chris Lattner et al. Mlir: A compiler infrastructure for the end of moore's law, 2020b. URL https://arxiv.org/abs/2002.11054.

Marc Le Fur. Scanning parameterized polyhedron using fourier-motzkin elimination. *Concurrency: Practice and Experience*, 8(6):445–460, 1996.

doi:https://doi.org/10.1002/(SICI)1096-9128(199607)8:6<445::AID-CPE253>3.0.CO;2-G. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291096-9128%28199607%298%3A6%3C445%3A%3AAID-CPE253%3E3.0.CO%3B2-G.

Juhyun Lee, Nikolay Chirkov, Ekaterina Ignasheva, Yury Pisarchyk, Mogan Shieh, Fabio Riccardi, Raman Sarokin, Andrei Kulik, and Matthias Grundmann. On-device neural net inference with mobile GPUs, 2019. ArXiv 1907.01989.

Lei Zhang. Mlir codegen dialects for machine learning compilers. https://www.lei.chat/posts/mlir-codegen-dialects-for-machine-learning-compilers/, 2022. [Online; accessed 26-March-2023].

Steve Leibson et al. Xilinx ultrascale: The next-generation architecture for your next-generation architecture. *Xilinx White Paper WP435*, 143, 2013.

Maksim Levental, Alok Kamatar, Ryan Chard, Kyle Chard, and Ian Foster. nelli: a lightweight frontend for mlir, 2023.

Shuai Cheng Li, Hon Wai Leong, and Steven K Quek. New approximation algorithms for some dynamic storage allocation problems. In *International Computing and Combinatorics Conference*, pages 339–348. Springer, 2004.

Yinan Li, Jack Dongarra, and Stanimire Tomov. A note on auto-tuning gemm for gpus. In *Computational Science–ICCS 2009: 9th International Conference Baton Rouge, LA, USA, May 25-27, 2009 Proceedings, Part I 9*, pages 884–892. Springer, 2009.

Yinglong Li. Research and application of deep learning in image recognition. In *2022 IEEE 2nd international conference on power, electronics and computer applications (ICPECA)*, pages 994–999. IEEE, 2022.

Zhiyao Li, Jiaxiang Li, Taijie Chen, Dimin Niu, Hongzhong Zheng, Yuan Xie, and Mingyu Gao. Spada: Accelerating sparse matrix multiplication with adaptive dataflow. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, pages 747–761, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399166. doi:10.1145/3575693.3575706. URL https://doi.org/10.1145/3575693.3575706.

Xiangru Lian, Binhang Yuan, Xuefeng Zhu, Yulong Wang, Yongjun He, Honghuan Wu, Lei Sun, Haodong Lyu, Chengjun Liu, Xing Dong, et al. Persia: An open, hybrid system scaling deep learning-based recommenders up to 100 trillion parameters. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 3288–3298, 2022.

Luís Gabriel Lima, Francisco Soares-Neto, Paulo Lieuthier, Fernando Castor, Gilberto Melfe, and João Paulo Fernandes. Haskell in green land: Analyzing the energy behavior of a purely functional language. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 517–528, 2016. doi:10.1109/SANER.2016.85.

Weifeng Liu and Brian Vinter. An efficient gpu general sparse matrix-matrix multiplication for irregular data. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 370–381. IEEE, 2014.

Yizhi Liu et al. Optimizing cnn model inference on cpus. 2018. doi:10.48550/ARXIV.1809.02697. URL `https://arxiv.org/abs/1809.02697`.

Yongtao Liu et al. Exploring physics of ferroelectric domain walls in real time: Deep learning enabled scanning probe microscopy. *Advanced Science*, 2022a.

Zhengchun Liu et al. Deep learning accelerated light source experiments. In *IEEE/ACM 3rd Workshop on Deep Learning on Supercomputers*, pages 20–28. IEEE, 2019.

Zhengchun Liu et al. *BraggNN*: fast X-ray Bragg peak analysis using deep learning. *IUCrJ*, 9(1):104–113, 2022b.

Saeed Maleki et al. An evaluation of vectorizing compilers. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 372–382. IEEE, 2011.

Karl Marx and Friedrich Engels. The communist manifesto. 1848. *Trans. Samuel Moore. London: Penguin*, 15(10.1215):9780822392583–049, 1967.

Alexander McCaskey and Thien Nguyen. A mlir dialect for quantum assembly languages. In *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 255–264, 2021. doi:10.1109/QCE52317.2021.00043.

Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst.*, 18(4):424–453, jul 1996. ISSN 0164-0925. doi:10.1145/233561.233564. URL `https://doi.org/10.1145/233561.233564`.

J McMullin et al. The square kilometre array project update. In *Ground-based and Airborne Telescopes IX*, volume 12182, pages 263–271. SPIE, 2022.

Mohammad Motamedi, Philipp Gysel, Venkatesh Akella, and Soheil Ghiasi. Design space exploration of fpga-based deep convolutional neural networks. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 575–580. IEEE, 2016.

Khan Muhammad, Amin Ullah, Jaime Lloret, Javier Del Ser, and Victor Hugo C de Albuquerque. Deep learning for safe autonomous driving: Current challenges and future directions. *IEEE Transactions on Intelligent Transportation Systems*, 22(7):4316–4336, 2020.

Myoung Jin Nam, Periklis Akritidis, and David J Greaves. Framer: A tagged-pointer capability system with memory safety applications. In *35th Annual Computer Security Applications Conference*, ACSAC '19, pages 612–626, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450376280. doi:10.1145/3359789.3359799. URL `https://doi.org/10.1145/3359789.3359799`.

Razvan Nane et al. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016. doi:10.1109/TCAD.2015.2513673.

Julian Oppermann. *Advances in ILP-based Modulo Scheduling for High-Level Synthesis*. PhD thesis, Technische Universität, Darmstadt, 2019. URL `http://tuprints.ulb.tu-darmstadt.de/9272/`.

Julian Oppermann et al. How to make hardware with maths: An introduction to CIRCT's scheduling infrastructure. In *European LLVM Developers' Meeting*, 2022.

Daniel W Otter, Julian R Medina, and Jugal K Kalita. A survey of the usages of deep learning for natural language processing. *IEEE transactions on neural networks and learning systems*, 32(2):604–624, 2020.

Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, Juan Pino, Martin Schatz, Alexander Sidorov, Viswanath Sivakumar, Andrew Tulloch, Xiaodong Wang, Yiming Wu, Hector Yuen, Utku Diril, Dmytro Dzhulgakov, Kim Hazelwood, Bill Jia, Yangqing Jia, Lin Qiao, Vijay Rao, Nadav Rotem, Sungjoo Yoo, and Mikhail Smelyanskiy. Deep learning inference in Facebook data centers: Characterization, performance optimizations and hardware implications, 2018. ArXiv 1811.09886.

Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32:8026–8037, 2019.

Adam Paszke et al. Automatic differentiation in PyTorch. In *31st Conference on Neural Information Processing Systems*, 2017.

Robert M Patton et al. 167-Pflops deep learning for electron microscopy: From learning physics to atomic manipulation. In *SC'18*, pages 638–648. IEEE, 2018.

Laurent Perron, Frédéric Didier, and Steven Gay. The cp-sat-lp solver. In Roland H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:2, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-300-3. doi:10.4230/LIPIcs.CP.2023.3. URL `https://drops.dagstuhl.de/opus/volltexte/2023/19040`.

Yury Pisarchyk and Juhyun Lee. Efficient memory management for deep neural net inference, 2020.

Artur Podobas, Kentaro Sano, and Satoshi Matsuoka. A survey on coarse-grained reconfigurable architectures from a performance perspective. *IEEE Access*, 8:146719–146743, 2020.

Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, sep 1999. ISSN 0164-0925. doi:10.1145/330249.330250. URL `https://doi.org/10.1145/330249.330250`.

Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks, 2016.

Ilija Radosavovic, Raj Prateek Kosaraju, Ross Girshick, Kaiming He, and Piotr Dollár. Designing network design spaces, 2020.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

Sanjay V Rajopadhye. Dependence analysis and parallelizing transformations. In *The Compiler Design Handbook*. 2002.

M. Ramakrishna, Jisung Kim, Woohyong Lee, and Youngki Chung. Smart dynamic memory allocator for embedded systems. In *23rd International Symposium on Computer and Information Sciences*, pages 1–6, 2008. doi:10.1109/ISCIS.2008.4717922.

J. Rapin and O. Teytaud. Nevergrad - A gradient-free optimization platform. `https://GitHub.com/FacebookResearch/Nevergrad`, 2018.

Oliver Rausch et al. DaCeML: A data-centric optimization framework for machine learning. In *36th ACM International Conference on Supercomputing*, 2022.

Nico Reissmann, Jan Christian Meyer, Helge Bahmann, and Magnus Själander. Rvsdg: An intermediate representation for optimizing compilers. *ACM Trans. Embed. Comput. Syst.*, 19(6), dec 2020. ISSN 1539-9087. doi:10.1145/3391902. URL `https://doi.org/10.1145/3391902`.

Lutz Roeder. Netron, visualizer for neural network, deep learning, and machine learning models, 01 2022. URL `https://github.com/lutzroeder/netron`.

Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: a new ir for machine learning frameworks. *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, Jun 2018. doi:10.1145/3211346.3211348. URL `http://dx.doi.org/10.1145/3211346.3211348`.

Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach

to runtime code generation and compiled dsls. In *Proceedings of the ninth international conference on Generative programming and component engineering*, pages 127–136, 2010.

Benjamin John Rosser. Cocotb: a python-based digital logic verification framework, 2018. `https://docs.cocotb.org`.

Nadav Rotem et al. Glow: Graph lowering compiler techniques for neural networks, 2018. URL `https://arxiv.org/abs/1805.00907`.

Alexander Rucker, Matthew Vilim, Tian Zhao, Yaqi Zhang, Raghu Prabhakar, and Kunle Olukotun. Capstan: A vector rda for sparsity. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, pages 1022–1035, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385572. doi:10.1145/3466752.3480047. URL `https://doi.org/10.1145/3466752.3480047`.

Alexander Rucker, Shiv Sundram, Coleman Smith, Matthew Vilim, Raghu Prabhakar, Fredrik Kjolstad, and Kunle Olukotun. Revet: A language and compiler for dataflow threads, 2024.

Bertrand Russell. *Principles of Mathematics*. Routledge, 1937.

Jean-Paul Sartre. *Existentialism is a Humanism*. Yale University Press, 2007.

Taro Sekiyama, Takashi Imamichi, Haruki Imai, and Rudy Raymond. Profile-guided memory optimization for deep neural networks, 2018. ArXiv 1804.10001.

Jaime Sevilla, Lennart Heim, Anson Ho, Tamay Besiroglu, Marius Hobbhahn, and Pablo Villalobos. Compute trends across three eras of machine learning. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2022. doi:10.1109/IJCNN55064.2022.9891914.

Ariya Shajii, Gabriel Ramirez, Haris Smajlović, Jessica Ray, Bonnie Berger, Saman Amarasinghe, and Ibrahim Numanagić. Codon: A compiler for high-performance pythonic applications and dsls. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, CC 2023, pages 191–202, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700880. doi:10.1145/3578360.3580275. URL `https://doi.org/10.1145/3578360.3580275`.

Hardik Sharma et al. From high-level deep neural models to fpgas. In *49th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–12, 2016.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer, 2017. ArXiv 1701.06538.

Evan Shelhamer, Jonathan Long, and Trevor Darrell. Fully convolutional networks for semantic segmentation, 2016.

Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. Nimble: Efficiently compiling dynamic neural networks for model inference, 2021.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

Sean Silva and Anush Elangovan. Torch-MLIR. `https://mlir.llvm.org/OpenMeetings/2021-10-07-The-Torch-MLIR-project.pdf`, 2021.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, Boston, MA, third edition, 2013. ISBN 113318779X.

Steven S. Skiena. *The Algorithm Design Manual*. Springer Publishing Company, Incorporated, 2nd edition, 2008. ISBN 1848000693.

Graeme Smith. Introducing reference semantics via refinement. In *Formal Methods and Software Engineering: 4th International Conference on Formal Engineering Methods, ICFEM 2002 Shanghai, China, October 21–25, 2002 Proceedings 4*, pages 588–599. Springer, 2002.

Volker Strassen et al. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4): 354–356, 1969.

Bjarne Stroustrup. Foundations of c++. In *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP'12, pages 1–25, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 9783642288685. doi:10.1007/978-3-642-28869-2_1. URL `https://doi.org/10.1007/978-3-642-28869-2_1`.

Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp. *arXiv preprint arXiv:1906.02243*, 2019.

Alex Suhan, Davide Libenzi, Ailing Zhang, Parker Schuh, Brennan Saeta, Jie Young Sohn, and Denys Shabalin. Lazytensor: combining eager execution with domain-specific compilers, 2021.

Kun Suo, Yong Shi, Chih-Cheng Hung, and Patrick Bobbie. Quantifying context switch overhead of artificial intelligence workloads on the cloud and edges. In *36th Annual ACM Symposium on Applied Computing*, SAC '21, pages 1182–1189, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450381048. doi:10.1145/3412841.3441993. URL `https://doi.org/10.1145/3412841.3441993`.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions, 2014.

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision, 2015.

Hamid Tabani, Ajay Balasubramaniam, Elahe Arani, and Bahram Zonooz. Challenges and obstacles towards deploying deep learning models on mobile devices, 2021.

Endri Taka, Aman Arora, Kai-Chiang Wu, and Diana Marculescu. Maxeva: Maximizing the efficiency of matrix multiplication on versal ai engine. In *2023 International Conference on Field Programmable Technology (ICFPT)*, pages 96–105, 2023. doi:10.1109/ICFPT59805.2023.00016.

Shinya Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *International Symposium on Applied Reconfigurable Computing*, pages 451–460. Springer, 2015.

Tal Ben-Nun, Kaushik Kulkarni, Mehdi Amini, Berke Ates. pyMLIR: Python interface for the multi-level intermediate representation, 2023. URL `https://github.com/spcl/pymlir`.

Li Tan, Longxiang Chen, Zizhong Chen, Ziliang Zong, Dong Li, and Rong Ge. Improving performance and energy efficiency of matrix multiplication via pipeline broadcast. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–5, 2013. doi:10.1109/CLUSTER.2013.6702672.

Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. MnasNet: Platform-aware neural architecture search for mobile, 2019. Arxiv 1807.11626.

Gizem Temelcan, Hale Gonce Kocken, and Inci Albayrak. Solving the system optimum static traffic assignment problem with single origin destination pair in fuzzy environment. In *International Online Conference on Intelligent Decision Science*, pages 521–530. Springer, 2020.

Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19, 2019.

Seiya Tokui, Ryosuke Okuta, Takuya Akiba, Yusuke Niitani, Toru Ogawa, Shunta Saito, Shuji Suzuki, Kota Uenishi, Brian Vogel, and Hiroyuki Yamazaki Vincent. Chainer: A deep learning framework for accelerating the research cycle. In *25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2002–2011, 2019.

Andrew Troelsen, Philip Japikse, Andrew Troelsen, and Philip Japikse. The philosophy of. net core. *Pro C# 7: With. NET and. NET Core*, pages 1245–1253, 2017.

Nicolas Vasilache, Oleksandr Zinenko, Aart J. C. Bik, Mahesh Ravishankar, Thomas Raoux,

Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, Stella Laurenzo, and Albert Cohen. Composable and modular code generation in mlir: A structured and retargetable approach to tensor compiler construction, 2022. URL `https://arxiv.org/abs/2202.03293`.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

Todd L Veldhuizen. C++ templates are turing complete. *Available at citeseer. ist. psu. edu/581150. html*, 2003.

Lorenzo Verdoscia, Roberto Giorgi, et al. A data-flow soft-core processor for accelerating scientific calculation on fpgas. *Mathematical Problems in Engineering*, 2016, 2016.

Stephen Williams. Icarus verilog, 1998–2020. `http://iverilog.icarus.com`.

Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 887–898, 2012.

D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997. doi:10.1109/4235.585893.

WP506. Ai engines and their applications. Technical report, Advanced Micro Devices, Inc., 2022.

Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. Machine learning at Facebook: Understanding inference at the edge. In *IEEE International Symposium on High Performance Computer Architecture*, pages 331–344, 2019. doi:10.1109/HPCA.2019.00048.

Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized convolutional neural networks for mobile devices, 2016.

Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks, 2017. ArXiv 1611.05431.

Xiaofan Xu, Mi Sun Park, and Cormac Brick. Hybrid pruning: Thinner sparse networks for fast inference on edge devices, 2018.

Hanchen Ye et al. Scalehls: A new scalable high-level synthesis framework on multi-level intermediate representation. In *IEEE International Symposium on High-Performance Computer Architecture*, 2022.

Raphael Yuster and Uri Zwick. Fast sparse matrix multiplication. *ACM Transactions On Algorithms (TALG)*, 1(1):2–13, 2005.

Sergey Zagoruyko and Nikos Komodakis. Wide residual networks, 2017.

Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. Gamma: leveraging gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 687–701, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi:10.1145/3445814.3446702. URL `https://doi.org/10.1145/3445814.3446702`.

Zhiru Zhang et al. Autopilot: A platform-based esl synthesis system. In *High-Level Synthesis*, pages 99–112. Springer Netherlands, Dordrecht, 2008. ISBN 978-1-4020-8588-8.

Jie Zhao, Michael Kruse, and Albert Cohen. A polyhedral compilation framework for loops with dynamic data-dependent bounds. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 14–24, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356442. doi:10.1145/3178372.3179509. URL `https://doi.org/10.1145/3178372.3179509`.

Zhong-Qiu Zhao, Peng Zheng, Shou-Tao Xu, and Xindong Wu. Object detection with deep learning: A review. *IEEE Transactions on Neural Networks and Learning Systems*, 30 (11):3212–3232, 2019. doi:10.1109/TNNLS.2018.2876865.

S. Zheng et al. Neoflow: A flexible framework for enabling efficient compilation for high performance dnn training. *IEEE Transactions on Parallel and Distributed Systems*, 33 (11):3220–3232, 2022. ISSN 1558-2183.

Jinming Zhuang, Jason Lau, Hanchen Ye, Zhuoping Yang, Yubo Du, Jack Lo, Kristof Denolf, Stephen Neuendorffer, Alex Jones, Jingtong Hu, Deming Chen, Jason Cong, and Peipei Zhou. Charm: Composing heterogeneous accelerators for matrix multiply on versal acap architecture. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '23, pages 153–164, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394178. doi:10.1145/3543622.3573210. URL `https://doi.org/10.1145/3543622.3573210`.

Joe Zimmerman. langcc: A next-generation compiler compiler, 2022.