DREXEL UNIVERSITY

Abstract

Faculty Name
College of Engineering

Masters

Thesis Title

by Tajung Jang

The landscape of deep learning compiler frameworks has evolved rapidly with the development of various tools, such as TVM, deeptools, TensorFlow, DLVM, nGraph, and Glow. These frameworks offer unique optimizations to address computation and data movement challenges in deep learning accelerators (DLAs). These approaches include graph or IR level optimizations related to intra node memory access optimizations, operator fusion, and various tiling techniques. Despite their unique approaches, these frameworks primarily concentrate on node level optimizations that focus on increasing the performance of executing a scheduled kernel operation in the graph and overlook the potential for inter-node data reuse optimizations within on-chip memory resources.

OnSRAM, a scratchpad management framework build to work with deep learning compilers, addresses this gap by focusing on internode scratchpad management in DLAs. OnSRAM exploits the static graph representations of deep learning models by identifying data structures that can be pinned to on-chip memory based on their reuse rate and cost of transfer from main host memory. OnSRAM has been implemented and evaluated on a single DLA that contains a monolithic scratchpad and is integrated as part of a custom deep learning compiler framework.

In this work, we extend the capabilities of OnSRAM by introducing an optimal dynamic scratchpad allocation for static graph execution models using any number of scratchpads using Integer Linear Programming (ILP) to optimize an accurate cost model of data transfers. This enhancement allows for more wholistic control over on-chip memory resources compared to the heuristic approach OnSRAM takes, providing increased flexibility and adaptability to better accommodate diverse deep learning accelerators and memory access patterns. By optimizing inter-node data movement and storage across multiple scratchpads, our approach further reduces energy consumption and latency associated with inter-node communication.

Contents

Abstract **i**

1 Background **1**
1.1 Accelerators and Scratchpad Memory . . . 1
1.1.1 Scratchpad Memory Management Schemes . . . 2
1.2 Deep Learning Frameworks . . . 3
1.2.1 Introduction . . . 3
1.2.2 Computational Graphs . . . 5
1.2.3 IR . . . 6
1.2.4 Front End . . . 7
1.2.5 Middle End . . . 7
1.2.6 Back End . . . 8

2 Related Work **11**
2.1 Existing Memory Optimizations in Deep Learning Frameworks . . . 11

3 Proposed approach **13**
3.1 OnSRAM . . . 13
3.1.1 Static Graph Execution . . . 13
3.2 Simulation and Architecture . . . 15
3.3 Extensions . . . 16
3.3.1 OnSRAM Limitations . . . 16
3.3.2 Motivating Example . . . 16
3.3.3 Proposed Approach . . . 17

4 Implementation **19**
4.1 Static Graph Analysis . . . 19
4.1.1 SMAUG . . . 19
4.1.2 Static Analysis . . . 20
4.1.3 SPM Mapping Representation . . . 20
4.2 Model Formulation . . . 21
4.2.1 Objective Function . . . 24
4.2.2 Constraints . . . 24
4.2.3 Solver . . . 25

5 Evaluation Method **27**
5.1 Data Collection . . . 27
5.2 Limiting Factors . . . 28

6 Experiments and Results **31**
6.1 Speedup . . . 31
6.2 Reduction of DMA Transfers . . . 32
6.3 Discussion of Results . . . 33

7 Future Work and Conclusion **37**
7.1 Conclusion 37
7.2 Future Work 37

Bibliography **39**

Chapter 1

Introduction

Chapter 2

Background

2.1 Accelerators and Scratchpad Memory

The growth in Deep Learning (DL) has grown substantially in the past few years enabled by the availability in hardware to train the Deep Neural Network (DNN) models. There exists many use cases of DL models such as natural language processing, image recognition, media generation, and media alteration. Each of these use cases demand different workloads as they use different DL model architectures and are deployed in a variety of environments. This creates the need of a diverse set of hardware called Deep Learning Accelerators (DLAs) that are specifically made to compute DL workloads efficiently for a specific purpose. This is due to the nature of DNNs. They are made up of millions of parameters, dozens of different operations that work on multi dimensional data structures made of thousands of floating point units. However, because of the finite set of primitive operations that are needed to support the computation of these models, the repetitive nature of these operations, and type of data they work with; DL models are highly parallelizable and specialized hardware can be built to exploit this with maximum efficiency. Some exist to be power efficient and work in embedded devices while others are made to have no power constraints so as to maximize compute speed in cloud clusters. In contrast, general purpose hardware are not able to meet power management needs of many workloads or cannot compute inference fast enough to meet timing requirements. Examples of DLAs used are TPUs from Google [1], NVDIA's NVDLA [2], and Intel NNP [3]. An example of an NVDLA based architecture is show in figure 1.1.

A key factor that determines the compute and power efficiency of a DLA is how well its Scratchpad Memory (SPM) is managed. Scratchpad memory is an on-chip SRAM memory that is managed by software. The transfer of data to and from main memory must be orchestrated explicitly by the programmer or compiler and the degree of utilization of the memory is dependant on how it is managed. This feature of SPMs is why they have the potential to be more compute and power efficient than caches due to the predictability of data transfer latencies and explicit management of data evictions and pinnings without tagging. Each data transfer incurs a latency between main and on-chip memory and also power costs to initiate DMA loads and stores. Caches have hardware that determines when and what data is transferred but can be unpredictable or unoptimal due to the fact that eviction and caching schemes are implemented in a heuristic manner [4]. As a result, SPMs are highly popular for predictable and repeatable workloads and are used in many embedded devices [5] and DLAs [6].

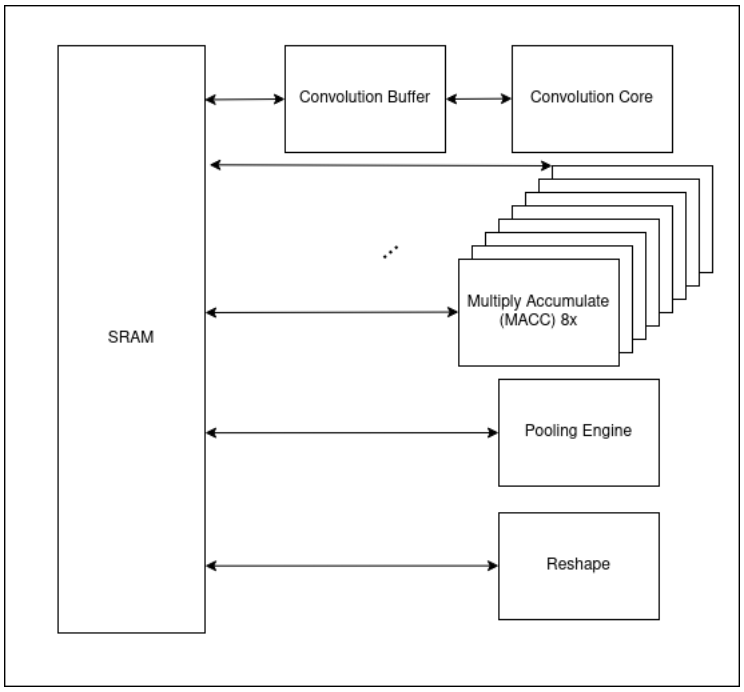

FIGURE 2.1: Architecture of SMAUG's NVDLA-inspired DLA

2.1.1 Scratchpad Memory Management Schemes

Because SPMs do not contain any hardware logic that deals with pinning and eviction, all memory transfers between main memory and on-chip-scratchpad-memory must be explicitly programmed. In workloads where the access patterns are very predictable or deterministic, this is ideal as the overhead of tag-decoding logic is avoided and superior performance gains can be had with carefully optimized memory transfers. Further, software-managed memory leads to more predictable timing and data transfer costs because of the lack of tag-decoding logic [5]. This makes the use of scratchpads highly desirable for deep learning workloads where the computations and dataflow are deterministic for any given architecture and therefore the memory management strategy can be tuned for. Consequently, the runtime performance of an application is highly dependant on the ability of a strategy to optimally manage memory and exploit data reuse.

Scratchpad memory management is usually manually programmed by developers that are specialized in a particular architecture. This is the case for DLA kernels as well, as most kernels are provided as vendor supplied libraries where the memory management is optimized [7].

While manual tuning is still used, there exist many limitations that make it unattractive. Manual tuning requires significant engineering effort, is vulnerable to bugs, and a strategy is not portable across different architectures. Due to these limitations, techniques for compiler based automated scratchpad memory management have been developed to ease the burden of programmers. The problem of portability and need for an automated alternative has been exacerbated for deep learning frameworks and engineers. The same model could be deployed on several different types of architectures, each with their own memory hierarchy. Creating memory optimized operator schedules, pinning directives, and kernels for each of these architectures is

time consuming and requires specialized knowledge of each architecture. The implementation of such strategies requires a low level language to describe as well, which forces a DL engineers who typically work in high level language frameworks to be an expert in many more fields at once. While most automated compiler implementations have been aimed at embedded devices programmed in C [5] [8], in recent years there have been compiler tools built for deep learning workloads as well [6] [9].

The goal of an SPM management strategy is to minimize incurring the cost of data transfers by maximizing the reuse of data in the SPM. To achieve this, three main techniques explored in the compiler space: graph coloring, Integer Linear Programming (ILP), and heuristic based approaches.

Graph coloring for SPMs comes from using graph coloring for the register allocation problem [10] [5]. A compiler can decide what variables should be kept in what registers during their lifetimes and how often they should be swapped between main memory. The objective of register allocations are the same as scratchpad management schemes: to minimize the amount of swaps. This is achieved by constructing interference graphs of variables whose lifetimes overlap and therefore require to be placed in separate registers. An interference graph can then be used to apply a greedy graph coloring algorithm where the number of registers maps to the number of colors to determine the optimal register allocation scheme. Scratchpad management can be done by extending this idea where the scratchpad is partitioned into psuedo-register files [5]. Applying this to deep learning workloads is non-trivial as the tensor sizes are variable and can be larger than the size of the scratchpad. [9] discusses graph coloring implantations for deep learning workloads in the context of maximizing SPM utilization within kernel code. Figure 1.2 shows an example of how TopLib [9] uses a graph coloring register allocation strategy to fit intermediate tensors onto a scratchpad in a kernel.

Similarly, by considering an SPM with fixed size partitions to emulate registers, an Integer Linear Program (ILP) made to solve the register allocation problem can be extended to find an optimal strategy for an SPM as discussed in [11].

2.2 Deep Learning Frameworks

2.2.1 Introduction

In order to create to run a DL model, a programmer must describe the architecture, its input and output layers, and the inputs that will be passed to the model. Beyond these properties, the implementation and challenges associated with deploying such a model on a DLA or distributed system is handled by deep learning frameworks. Deep learning frameworks abstract the implementation and execution details from the programmer such that the programmer may focus on the model creation only. The abstraction of such details forces the trade-off of performance and generality as the frameworks must optimize the same model for runtime and memory efficiency on a diverse set of available hardware that all have different memory hierarchies, number of PEs, and supported operations. Frameworks must be able to decipher model descriptions in high level languages such as Python 1.1, apply hardware agnostic optimizations, and map operations to a specific hardware to create an executable that will run the model to completion device.

Conventional compilers (eg. Clang) typically take a high level language down to an intermediate representation (IR) to then machine code that can run on a device. The front end is responsible for what parses and lexes the high level code into a control flow graph, the middle end is responsible for lowering it to IR and running

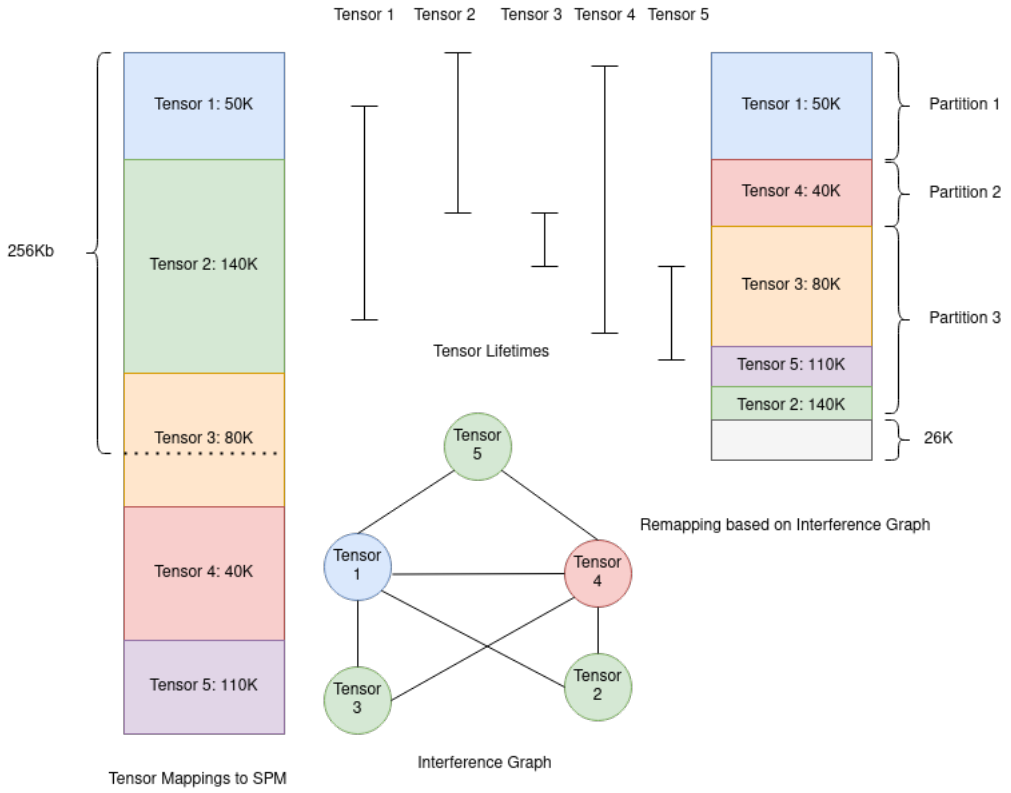

FIGURE 2.2: Example of graph coloring. Not all tensors can be mapped to the SPM at the same time so an interference graph is created based on their lifetimes and colored accordingly. The resulting mapping on the right contains partitions for tensors based on their lifetime and colors such that all tensors fit on the SPM without contention.

optimization passes on the IR, and the backend lowers the IR further into machine code. Similarly, DL frameworks can be broken down into a front-end, middle-end, and backend. However, there are major differences in the role of each section and how they are performed. Deep learning frameworks deal with a small set of operations that are highly parallelizable but with variable-sized data [12]. Some operations include: sigmoid, matrix multiply, vectorsum, L1 Norm, element wise multiply, and dropout among others [13]. Further, a programmer for a given deep learning framework will work with a domain specific language (DSL) such as Tensorflow [1]. Because DSLs only concern with the description of deep learning networks, the runtime execution, memory, and device management is managed by the framework. A framework that enables the deployment of a deep learning model onto hardware from only a high level language description is called an end to end framework.

LISTING 2.1: Python model description example

```
import tensorflow.compat.v2 as tf
from tensorflow import keras
import tensorflow_datasets as tfds
from preprocessDefinition import preprocess

# file below will be replaced by a testing set of identical format
data, info = tfds.load(name='oxford_flowers102', split=['train', 'validation'],as_supervised=Tru
train_ds = data[0]
valid_ds = data[1]
```

```
dataset=train_ds.map(preprocess)
dataset_valid=valid_ds.map(preprocess)

base_model=keras.applications.xception.Xception(weights='imagenet',include_top=False)
avg=keras.layers.GlobalAveragePooling2D()(base_model.output)
output=keras.layers.Dense(102,activation="softmax")(avg)
model=keras.models.Model(inputs=base_model.input,outputs=output)
model.compile(loss='sparse_categorical_crossentropy',optimizer=opt,
              metrics=['accuracy',top1err,top5err,top10err])

model.fit(dataset_tr,validation_data=dataset_valid_fit,epochs=25,
          callbacks=[checkpoint_cb,earlyStop_cb])

#returns the loss and metrics evaluated on the dataset
model.evaluate(dataset_ts)
model.save("flowersModel.h5")
```

2.2.2 Computational Graphs

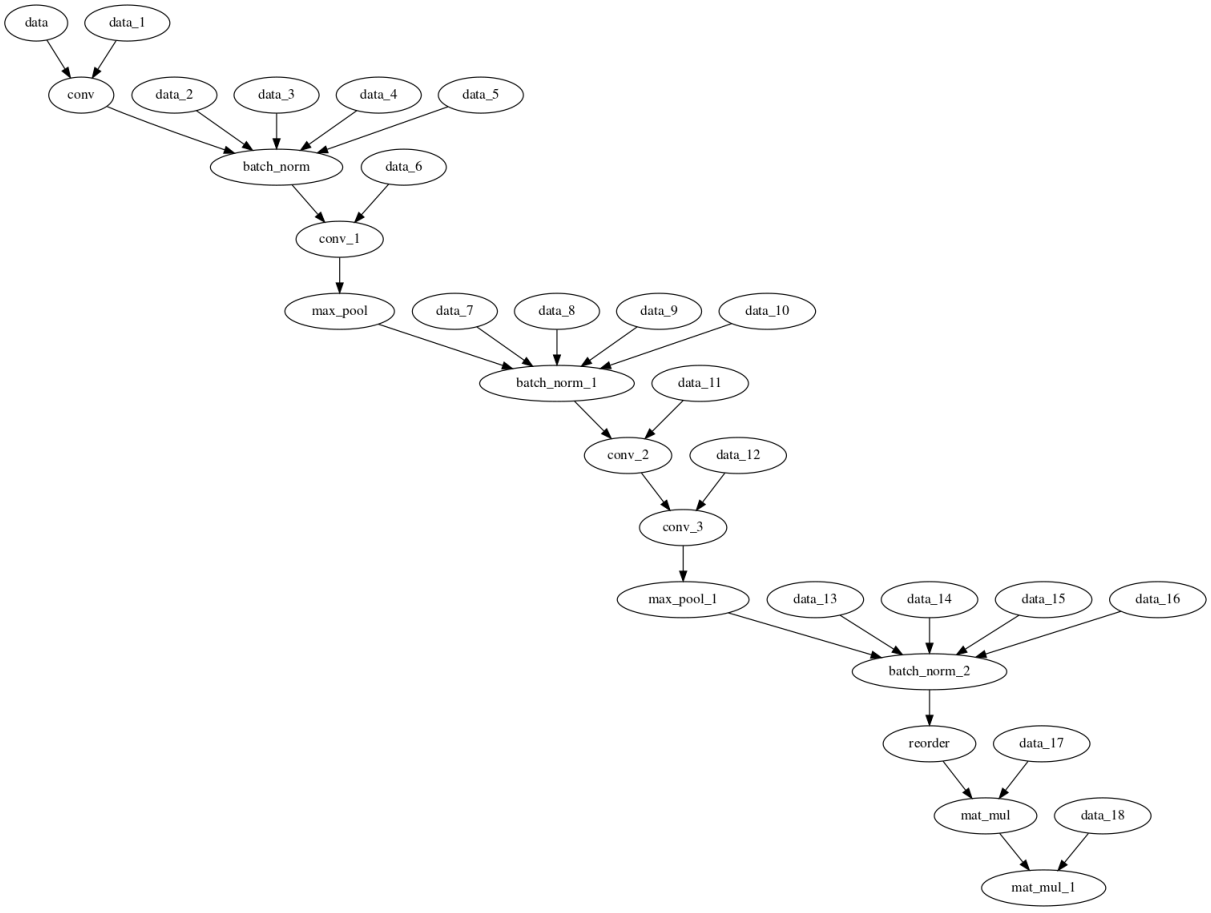

FIGURE 2.3: Computational graph of Cifar-CNN10

A common interface used by most DL frameworks is the computational graph [1] [13] that represents a network of operations needed to run a DL model. Each DL model can be broken down into a series of discrete computational operations that can be applied to an input to describe a given model. A graph made up of vertices and edges where the vertices represent an operation and the edges represent tensors that flow from one operation to the next can be used to fully describe a DL model. Computational graphs are directed acyclic graphs (DAG). Each vertex can

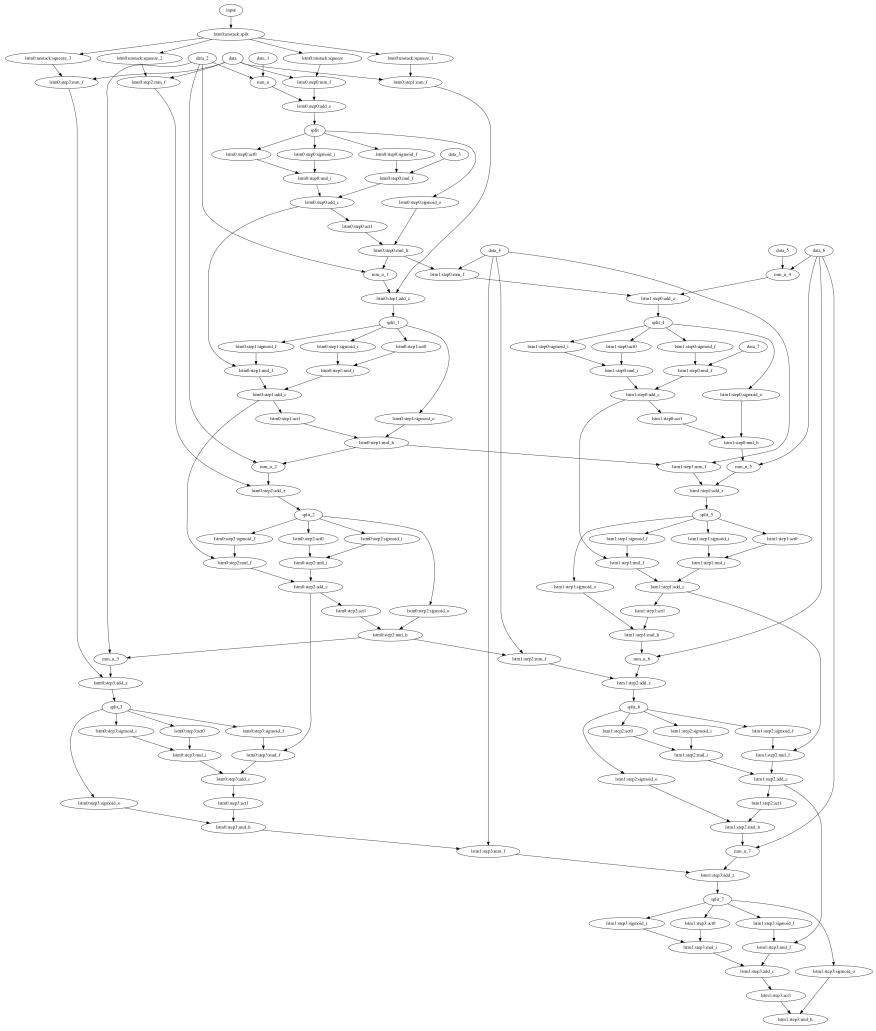

FIGURE 2.4: Computational graph of LSTM

have between zero and two inputs and zero to one outputs. So for each model, each node in each layer can be broken down into an operation that maps to a node in the graph. An operation is a particular computation that must be ran with a set of inputs and outputs. The implementation of the operation is defined by a kernel. A kernel is specific to a hardware architecture and are mapped to each operation by the backend. For example a matrix multiply operation is an essential operation that reoccurs frequently in computation graphs; the implementation of a matrix multiply will depend on how many processing elements (PE) the DLA has and its memory hierarchy. A tensor is a multi-dimensional array that can take on many types (eg., floating point, int8) and are the inputs and outputs of operations. Tensors are what are passed along edges of the graph. The dataflow represented by a graph can be used to infer dependencies and possible optimizations for the middle-end to apply. Examples of computational graphs are shown in figure 1.3 and figure 1.4.

2.2.3 IR

Intermediate representation (IR) is an alternative representation of DL models for some frameworks such as [14] [12] [15]. For each framework that uses IR, the framework typically uses its own IR, however there are tools such as ONNX that attempt

to standardize the IR used in frameworks.

IR in DL frameworks are a language that support linear algebra instructions and other general purpose instructions (eg., branch, conditional- branch). DL models are lowered into the framework's IR and a DAG of control flow [12] can be created where the vertices are operations and edges are data. This is similar to a computational graph, however, the expressiveness and modularity that IR provides may lead to more optimization opportunities [14]. In a computation graph, each kernel can be tuned to maximize its performance in isolation. Outside of manual kernel turning, performance gains come from the reordering of schedules and changing the access pattern of tensors. In IR based frameworks, the same types of optimizations can be done to each instruction that maps to a kernel. However, the kernels in IR are not picked from a set of pre-made kernels in a supported backend, but rather code generated into LLVM IR or another lower level IR based on the framework. This allows static analysis to better fuse redundant operations and kernels together in a more efficient way than sequentially ordering kernels as done in computational graph based frameworks. Further, different IRs may come with a wider range of operations that may be supported than the set of operations in a computational graph that allow the compiler and code generator to manipulate the DAG in more ways.

Once a DL model is lowered into IR and its optimization passes are done, the IR is passed into a code generator as backend code or a more supported IR such as LLVM IR. A transformer will transform the framework IR into code formatted for a particular backend. There must be a transformer made for every backend. Otherwise, in the case of LLVM based IR transformations, only backends that support LLVM IR are able to be used to generate a final binary.

2.2.4 Front End

The front end of frameworks are responsible for breaking down the high level descriptions of a DL model, typically in Python or C++ [1], and converting them into computation graphs or IR. These lower level abstractions allow the framework to more easily apply optimizations for memory and runtime efficiency. A high level overview of how the front end fits into the software stack of an end to end framework, as well as the middle and backend, are shown in figure **??**.

2.2.5 Middle End

Most optimizations that occur and the engineering focus of end to end frameworks lie in the middle end. Middle ends must have a diverse set of backends in mind and be hardware agnostic. In order to achieve this, an IR or computational graph representation of a DL model is received from the front end. Once a constructed dataflow graph in either computational graph or IR form is passed, several passes of static analysis and optimization occur [7] [14] [1]. Once a completely optimized dataflow graph is created, it must then be formatted to be compatible to a particular backend. This can either mean serializing a computational graph into a one that a backend can read or applying a transform to the optimized IR to generate code that is compatible with a backend. Backends are vendor specific compilers that deal with the device level details for kernel implementations and tuning. Back-ends will support a subset of formats (ie either serialized computational graphs or a specific IR) and to that end, there must be significant engineering effort to support a backend for a given middlend. Consequently, most frameworks only support a handful or backends such as Nvidia's CUDA backend or BLAS for CPUs. For supported backends, the

middlend must be sure to create a final computational graph or set of IR modules that do not contain operations or instructions that are not supported by the predefined kernels in the vendor provided library of a backend [7].

Optimizations in the middle end can be separated into IR specific and computational graph specific optimizations. For graph level optimizations, they are typically higher level than kernel tuning and involve graph restructuring and fusing of subgraphs. Specific optimizations include re-ordering or scheduling of operators to maximize data reuse to hide memory transfer latencies, mapping operators to available kernels for a target DLA, data layout transformations and operator fusion [7].

Operator fusion is the combining of multiple operators into a single operator. This is helpful because different operators that are distinct from one another that can otherwise be fused may be scheduled in an unoptimized way or incur unnecessary memory transfer costs. Every operation requires an input tensor to be loaded into the accelerator and an output tensor to be produced to be loaded back into main memory. Without explicit data pinning memory management or caching, or in cases where pinning may not be feasible due to memory size constraints of the DLA, intermediate values between two operators where the output of one is the input to the other, may be necessarily transferred back and forth between main and device memory. By fusing operators together, the cost of memory transfers due to separated operators are removed. Further, by fusing operators together, operator code generation can be optimized as certain intermediate values may not have to be stored or other computations may be able to be reduced into simpler forms [7] [15]. A disadvantage to this is, fused operations have to either be code generated where the performance of the kernel is dependant on the quality of the code generator without any manual tuning or vendor provided libraries must have pre-defined fused operators that are available to be mapped into the computational graph. As the number of operators on a network grows, the number of possible operators that can be fused also grows with it combinatorially [7]. This is exacerbated with the number of supported data layouts, data types, and DLA intrinsics. To this end, a large fused operator cannot be created such that memory transfer costs are fully hidden optimally.

Data layout transformations are the reformatting of how tensors are passed to a DLA based on the specific memory hierarchy that the DLA has. To exploit the shape of tensors the DLA best works with, tiling along certain dimensions or adding reshape operators to change the layout of the tensor in main memory before sending to the DLA is one way to minimize inference time and is done at the computational graph level.

Many of the optimizations that apply to computational graphs can also be done to IR based frameworks. But because they may be lowered into a lower level IR, more granular and kernel specific optimizations can also be applied before being passed to the backend. One optimization example is algebraic simplification and linear algebra fusion [14]. This involves reordering matrix multiplications and other linear operations into a single matrix operation. Another example is transforming subexpressions involving multiple types of operations into a single matrix operation such as a matrix multiply and an element-wise add [14].

2.2.6 Back End

A backend is typically not something that is component of a framework but a vendor provided compiler for a DLA that may be hooked into a compatible middle end. A backend may support a small set of input formats that describe a DL model in the form an optimized version of a computational graph or IR. At this point, all

optimizations and transforms that have occurred have only been applied to the DL model description provided by the programmer. The backend will be responsible in creating an execution engine, outputting a compiled executable, and managing a runtime environment.

A backend like TensorRT [16] consists of two phases: a build phase and runtime phase. During the build phase a kernel search is done to map available kernels to the operators in the dataflow graph that was passed from the middle end. A serialized version of the graph is then statically analyzed to be used as context to generate code for the execution context and runtime engine. An engine is a representation of the optimized model with a scheduler, memory manager, a network of operators, their inputs and outputs, and meta data of input and output tensors such as size and name of the tensors. During the runtime phase, an execution context and runtime engine will be instantiated. The execution context can then be saved to be loaded later or started to run inference tasks.

Kernels are highly specialized and must be tuned for a DLAs particular memory hierarchy, data layout, and intrinsics. Kernels are created either via code generation or supplied by a vendor library. Kernel tuning optimizations are handled by the backend. This includes loop unrolling, caching, loop reordering, and tiling. An example of what a kernel for a specific operator looks like is illustrated in figure 1.5

FIGURE 2.5: Estimated % of data transfers saved for varying scratch-pad sizes and models using the optimal pinning strategy

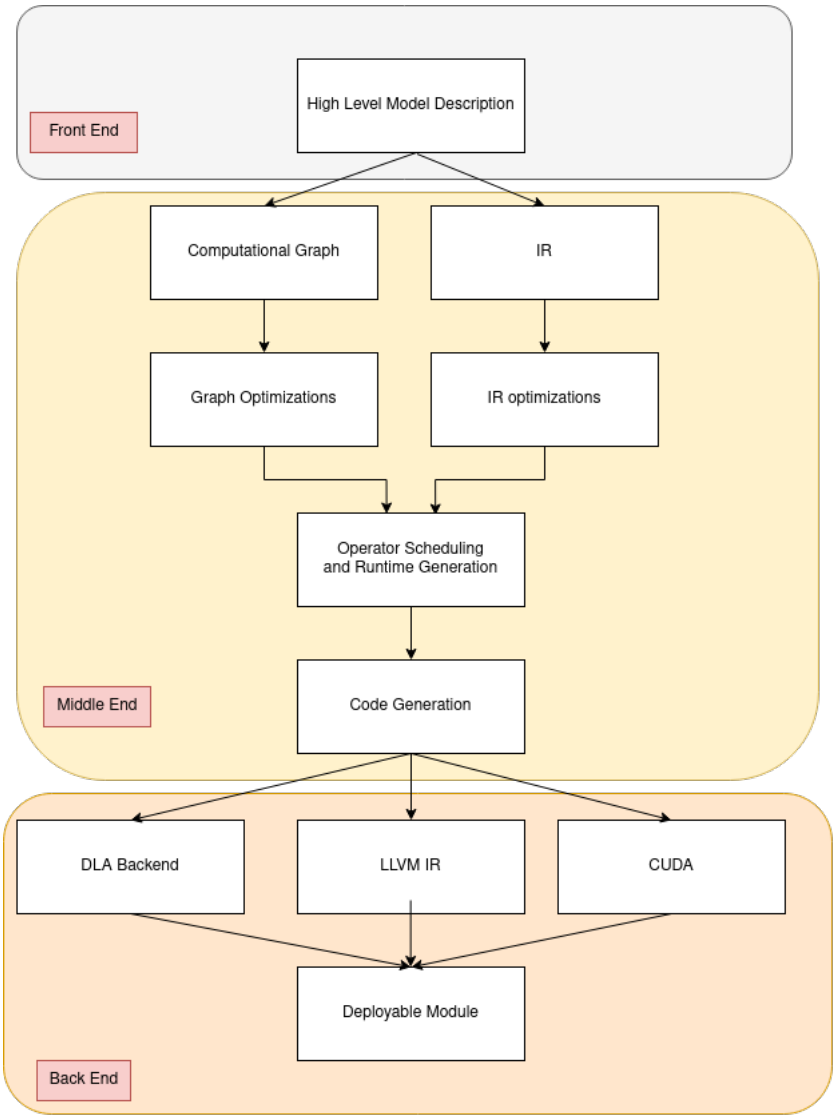

FIGURE 2.6: Overview of deep learning framework stacks

Chapter 3

Related Work

3.1 Existing Memory Optimizations in Deep Learning Frameworks

There exist many end to end deep learning frameworks that allow high level DSL programmers to create models that will be optimized to run on a given hardware. Because memory management is such a significant factor in runtime performance, all frameworks employ their own specialized optimization methods based on their unique model representation forms.

TVM by Chen, et al. 2018 is a graph based end to end framework. TVM aims to create a framework that targets multiple backends and accelerator architectures while still applying fine grained optimizations for both intra and inter operator computations. The main problem Chen et al. solved was the problem of existing deep learning compiler frameworks only applying high level graph optimizations that did not allow for tuning in operator specific implementations and left that to the backends. However, the existing frameworks only supported a small handful of GPU and CPU based backends. Creating support for new hardware backends requires significant manual effort and there was no automated solution that existed. TVM allows for multiple backends to be supported while also applying high graph level and operator implementation level optimizations within its framework. Optimizations include operator fusion, SPM management via operator rescheduling, pipelined memory access, and machine learning (ML) based cost modeling for code generation. Operator rescheduling is the rescheduling of operators based on data flow dependencies to maximize reuse by placing operators whose inputs are outputs of another next to each other. Pipelined memory accesses are when memory access instructions are executed without compute instructions having been completed yet so that the next compute instruction does not need to wait for the next memory instruction to complete. TVM also introduces an ML based modeling method that estimates the performance cost of different code generated and generates the lowest cost code to compile into the backed. This means that the search space of parameters such as loop tiling size and loop unrolling factors are automated and decided based on the ML model.

DLVM [14] (Wei et al. 2018) lowers a model into a domain specific IR such that modern compiler optimizations could be performed on the operations and simplified before LLVM IR is code generated for a backend to create individual kernels. Deep learning compilers [1] [17] prior to this work would take a computational graph of operations and use off the shelf vendor provided kernels to implement them. While the kernels themselves are highly tuned, further optimizations could be made such as algebra implication, kernel fusion, and other modern compiler techniques such as dead code elimination and sub expression elimination could be applied to further increase performance. By reducing the amount of operations needed to be applied by analyzing which linear algebra operations could be simplified and applying further

parallelization optimizations once it is lowered into LLVM IR, DLVM is able to minimize the amount of data transfer needed for kernels and how much data is needed to run the kernels themselves.

After DLVM, Cyphers, Bansal, et al. 2018 at Intel created nGraph [12], another IR based deep learning framework that takes an approach of using its own IR. Instead of lowering a domain specific IR into LLVM after optimization passes, nGraph takes the IR and passes it to a transformer, a code generator that produces optimized backend code or code to be linked to a specific backend's kernels. The goal was to create a more flexible IR that better supported tensor based instructions and could be compiled in a more targeted fashion compared to generic LLVM IR. Transformers also remove the dependency of backends having to support LLVM.

Li et al. 2020 introduced TOpLib [9]: a compiler-assisted operator template library for DNN accelerators that focused on automated kernel generation with optimized memory management using graph coloring. TOpLib builds on the graph coloring solution to the register allocation problem applied to SPM management for embedded devices but for DNNs. The key difference is that in non-DL workloads, allocations are targeted for single dimensional arrays whose sizes are known statically. Meanwhile, for DL workloads, tensors are high dimensional, variable sized, and can often be larger than the size of the SPM itself. This requires an approach to dynamically partition the SPM into variable sized psuedo-registers rather than equal sizing. The general concept that they apply is as follows. A static walk through of kernel code for a given operator is applied to analyze the liveness of each tensor used within it. The liveness of each tensor can then be used to create an interference graph of tensors where tensors on an interference graph have overlapping lifetimes. This graph can then be used to apply a graph coloring strategy to cluster tensors with non-overlapping lifetimes onto the same partition of the SPM, while overlapping lifetimes will be allocated separate partitions. The paper proved that such an automated solution reached close to or exceeded hand written assembly kernels and reach on average a 90% speed up over non-optimized kernel code. While this approach could have been generalized to the overall DNN case from the kernel case, this work applies a different approach similar to the work presented by Verma et al. 2004 using an ILP model. However, Li et al. demonstrate that a greedy graph coloring approach can reach close to optimal performance across a multitude of operators.

Verma et al. [11] introduce an ILP approach to the same register allocation problem for single dimensional arrays as the work Li et al. extend for DNNs. Verma et al. reformulate the graph colorign problem into an ILP form where constraints are added to ensure that tensors with overlapping lifetimes are not allocated the same partition. The objective function is created to maximize the number of data transfers saved. By using an ILP model over a greedy algorithm, an optimal solution can be found. However, again, this approach is not trivial to apply towards large variable tensors in DL workloads due to assumptions the model takes of fixed size psuedo-registers and single dimensional arrays.

The problem of using more than one SPM is an issue that has mostly been ignored until the work of Tao et al. 2021 where they solve for an optimal data size granularity to transfer between scratchpads and main memory to maximize performance using a linear programming model for multicore architectures. The paper achieves this via compiler-directed SPM data transfer model to formulate ann allocation scheme in a heterogeneous many-core architecture and then use the same allocations to determine the optimal data transfer granularity to maximize performance.

Chapter 4

Proposed approach

4.1 OnSRAM

IBM's OnSRAM (Subhankar, et al. 2022) introduces the notion of two types of runtime performance optimizations that can be done on computational graphs: intra-node and inter-node optimizations [6]. Intra node optimizations are those that are focused on optimizing the operation kernel that the node specifies. This means tiling in favorable sizes and across specific dimensions, loop ordering, and DMA pipelining between tiles [18]. Inter node optimizations are optimizations concerning the overall structure of the graph and the relationship between node connections. This includes operator fusion and node reordering[6].

Existing deep learning frameworks are not modular like modern programming language compilers [1] [7] [12]. Differences in supported operation and backends, optimization opportunities based on IR structuring, and lack of a standardized IR [12][14] all lead to why frameworks tend to be standalone projects that have little interoperability between layers of other compilers. Because of the fact that many of these frameworks are built from the ground up, they tend to re-implement many optimizations already explored in other frameworks for their own IR. In addition to the major engineering efforts required to do this, frameworks also implement their own IR for the sake of implementing their own unique kernel optimizations. Thus, almost all research efforts for memory efficiency within the deep learning framework space have been on intra-node optimizations facilitated through custom IR transformations and code generation. In contrast, OnSRAM exists as a interoperable compiler extension for computational graph based compilers to apply internode optimizations to maximize SPM utilization.

OnSRAM achieves its goal of maximizing SPM utilization by analyzing inter-node data-dependencies in computational graphs and using identified data reuse opportunities to create an optimal data allocation scheme for on-chip memory of the target DLA. OnSRAM shows that there exists a potential of up to 5.2x speedup opportunity for DL inference after integrating their SPM management in the runtime of Tensorflow [6].

4.1.1 Static Graph Execution

As previously described, computational graphs consist of nodes that represent operations and the edges represent data-dependencies between operations. During execution, each operation is loaded from main memory by the host device into the accelerator's SPM along with the operation's required weights and input tensors. Each transfer of data between main and on-chip memory incurs DMA transfer costs exacerbated by the bandwidth of the bus. Similarly, when a DLA executes an operation, the resulting output tensor is placed onto on-chip memory and again transferred

back to main memory. When two connected nodes, ie. an output of one operation is the input of the next, are scheduled sequentially, a reuse opportunity exists: the output tensor of the first operation can be pinned on the SPM without being transferred back and used immediately by the next operation, thereby eliminating unnecessary transfer costs 3.1 . OnSRAM creates an SPM management framework to minimize inference runs by creating an optimal pinning tensors onto the SPM.

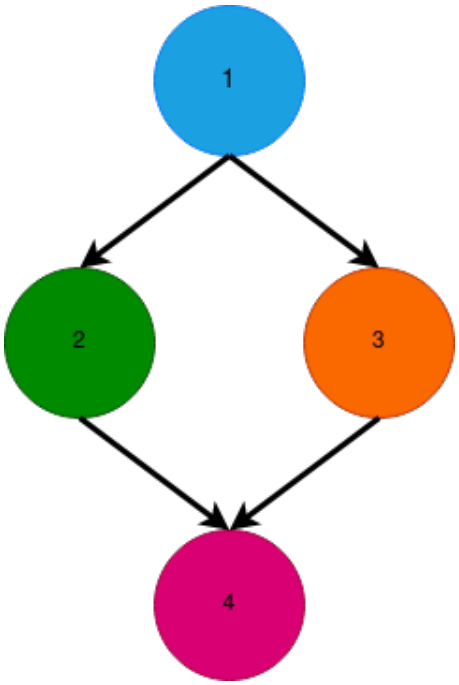

FIGURE 4.1: Example of a graph with data reuse

OnSRAM-Static, the static DNN graph based SPM manager is described as follows. A graph of operations that represents a DNN is passed to OnSRAM-Static as an input. All input and output data tensors of each node is analyzed for the start time, end time, number of reuses, distance between reuses, and size of the tensor. A weighted sum of these properties is calculated for each tensor to determine a metric for the cost incurred should a tensor be saved back to main memory and reloaded onto the SPM again. Based on the graph execution schedule, the tensors are psuedo-sorted in cost order. They are psuedo-sorted since tensors that do not overlap in their lifetimes do not need to be sorted relative to each other. These sorted tensors are then considered in a greedy fashion where tensors of the highest cost are considered for pinning first. Tensors are only pinned if they can be pinned for the entire duration of their lifetime without obstructing tensors needed for other operations before their next reuse. Figure 3.2 shows an overview of how OnSRAM fits as a compiler extension into deep learning frameworks.

OnSRAM-Static only applies to input and output tensors, not weights. This is due to the fact that weights are single use objects that are only needed for the layers they are used in. There exists no reuse opportunity unlike output tensors since all layers contain independently different weights. Thus, weights are not considered for pinning for both OnSRAM-Static and our extension work as well.

Another aspect that OnSRAM does not consider is applicability towards training. The assumptions of the problem change when considering training and significantly

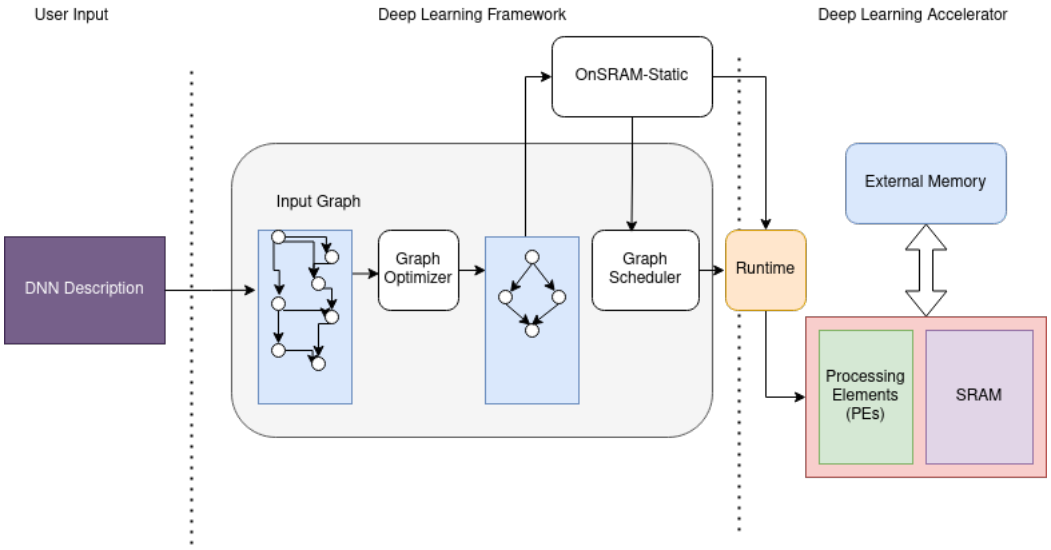

FIGURE 4.2: Overview of OnSRAM

increase the difficulty and search space for an optimal solution. This requires separate compiler extension implementations to be designed into separate portions of the framework. Further, batched training, partitioned accelerators, distributed systems, and large reuse distances due to backward passes and batched data [6] bring significant design challenges that are not present in inference. Thus we have also chosen not to extend this work in this regard as well.

4.2 Simulation and Architecture

Differences in hardware and simulators don't affect the relative performance of a non-pinning compiler compared to a pinning compiler if the model, inputs, framework, and hardware stay constant. However there are some important environment variables that will consequently affect repeatability and benchmark performance. Most notably, the simulator and hardware architecture in use.

OnSRAM uses a cycle accurate simulator to model and assess the performance of their algorithm [6]. We have opted to use the SMAUG[19] framework that depends on the gem5 Aladdin[18] system to model our accelerator and implement our experiments. While gem5 is not cycle accurate, it still advertises a close to cycle accurate performance metrics and creates a relative base of comparison between base models and our optimized model.

Intra-node optimizations that both simulators do but may not do the same way that may affect performance comparisons: loop tiling, loop ordering, unrolling, node re-ordering, and pipelined DMA operations for maximizing intra-node reuse.

We use the default SMAUG provided accelerator inspired by NVDLA[19]. The accelerator contains eight PEs, multiply accumalate (MACC) arrays, and contains 3 scratchpad memories[19]. Two scratchpads are configured to be used as inputs and one for output. Each scratchpad is sized at 32Kb as a default configuration.

4.3 Extensions

4.3.1 OnSRAM Limitations

While OnSRAM has contributed a novel SPM management scheme that no framework has explored to date [6], there are limitations that do not allow it to generalize in a hardware agnostic scenario. This is due to its assumption that the accelerator contains only one shared scratchpad for input and output tensors. A consequence of this is that the greedy algorithm proposed to map possible pin candidates is non-trivial to apply to an accelerator with more than one scratchpad. Further, because of the heuristic approach, OnSRAM is not able to achieve the optimal memory mappings with respect to minimizing memory transfer.

4.3.2 Motivating Example

Consider the computational graph of in figure **??**. The graph clearly shows that node 1's output will be reused as inputs to node 2 and 3. In a single SPM case, we can see how the tensors would be mapped for each timestep. Clearly, the leftover space of the SPM is what we can use as leftover pinning space between operations. However, in the case of a multi-SPM architecture, this is not the case as tensors cannot be split over different SPMs. To effectively navigate the optimal mapping of pinned tensors, what SPM a tensor is pinned on greatly affects the accumulated number of data transfers because of its effect on the mapping of future tensors to an SPM. Once a tensor is pinned to a particular SPM, mappings of inputs and outputs associated to other operations must be remapped in accordance to the remaining capacity of all SPMs. This issue can be compared to the bin-packing problem where the SPMs are bins and the tensors are the items. Hence, for every possible mapping of a particular tensor to a SPM, there exists new mapping for all other tensors that will exist for that particular operation and future operations relative to the position of the pinned tensor. The combinatorial explosion of possible mappings of tensors, how long each one is mapped for, and where they are mapped is a search space that grows with the depth of the deep learning model. An example of an un-optimal pinning strategy placing inputs and outputs without full consideration of all future tensor placement combinations is shown in figure **??**. In the figure, we can see that even though some tensors are able to be pinned, there is unnecessary wasted space in some of the operations as well as other tensors being written back and reloaded. Figure **??** shows how a different combination of tensor placements yields more reuse and less write backs to main memory.

Notably, the single SPM case does not involve the concern of the exact location for pinning a tensor. Rather, the decision making process in such a case is guided solely by a tesnor's relative cost, its size, and the degree of overlap it has with other tensors in terms of liveness. These factors shape how a pinned tensor could influence future pinning opportunities for other tensors. Due to these new considerations, simply porting the OnSRAM greedy algorithm leads to inefficient results since only a single scratchpad in the DLA can be used for pinning.

Rather than porting the OnSRAM approach; a naively greedy approach in the context of multiple SPMs has also been considered, i.e., initially pinning an item to whatever SPM has the most space and remapping the inputs and outputs accordingly. This process involves pinning outputs of every operation and evicting the tensors when the pinned tensor cannot accommodate all other necessary tensors for an operation, or when the output of the present operation is of higher importance. The initial decision of selecting an SPM to pin an item, which depends on the primary remapping of inputs

and outputs around a pinned tensor, can considerably influence the pinnability of future items. This might, in certain instances, lead to the unpinned state of items immediately prior to their reuse.

In such an approach, the system may not adequately identify opportunities for reuse, leading to unpredictable decision-making and near-suboptimal results. This limitation persists, albeit in a reduced form, even when the size of the SPMs are increased. Therefore, to devise an algorithm that meticulously analyzes possible decisions and maps tensors in a manner that effectively encourages reuse, one could implement techniques such as graph coloring or ILP, which are similar to the methods used in solving the pseudo-register SPM register allocation problem.

Choosing an optimal pinning strategy to minimize the amount of data transfers within this search space is the goal of this work.

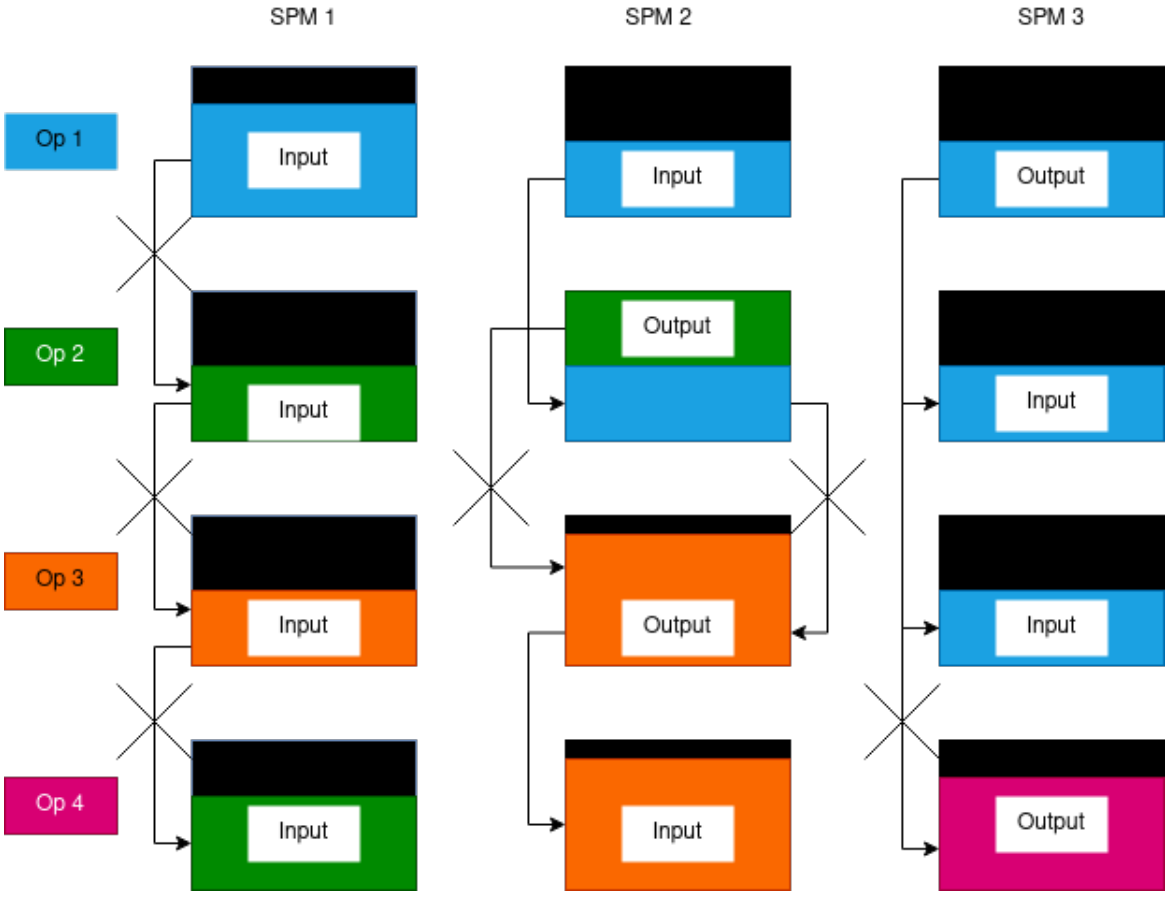

FIGURE 4.3: Example of a un-optimal pinning strategy. Arrows denote being pinned to the SPM and the crosses denote an SPM being evicted due to capacity constraints. Black denotes unused memory.

4.3.3 Proposed Approach

In order to analyze the search space and create a pin mapping for all tensors such that the number of data transfers between SPMs and main memory on an inter-node graph level are minimized, we propose an ILP model to create an optimal strategy for a multi-scratchpad architecture.

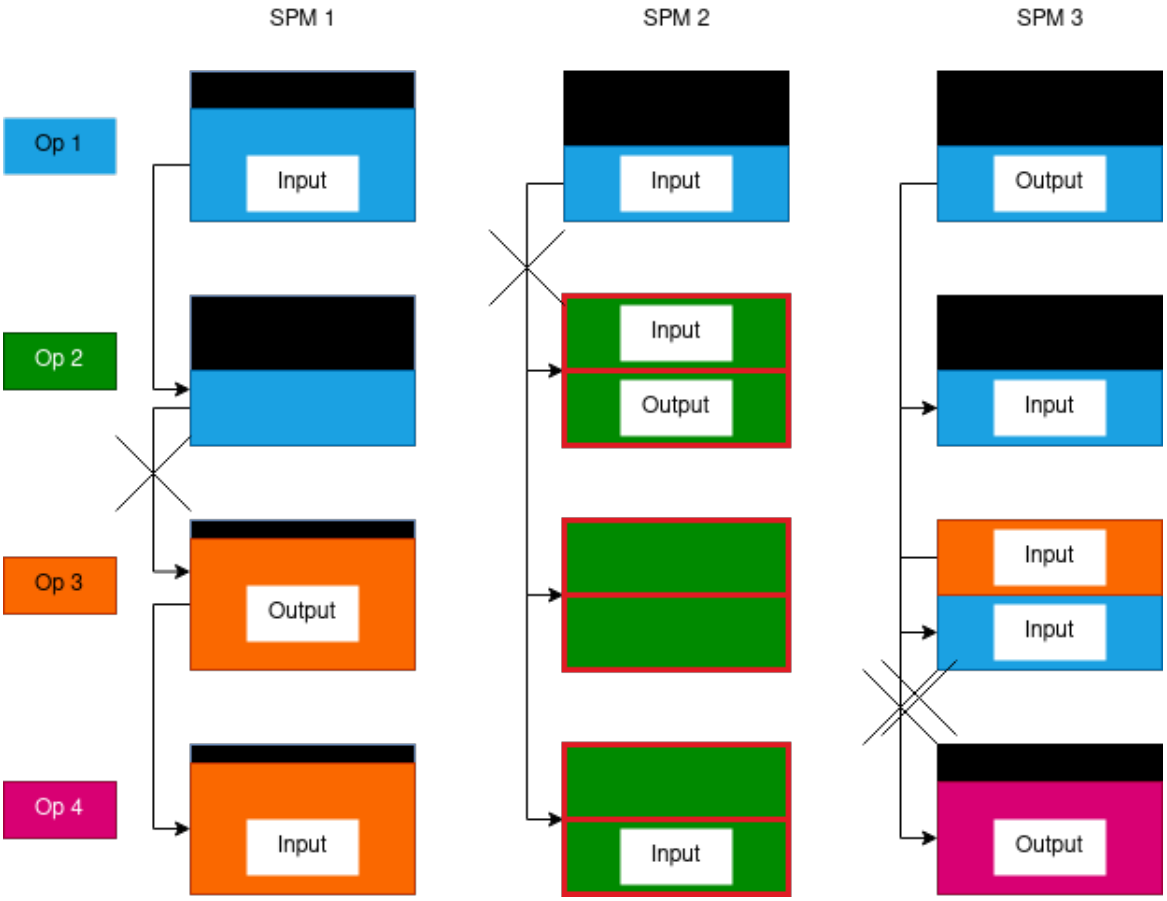

FIGURE 4.4: Example of an optimal pinning strategy. Arrows denote being pinned to the SPM and the crosses denote an SPM being evicted due to capacity constraints. Red borders signify distinction between different tensors created from the same operation. Black denotes unused memory.

To do this we first take an optimized computational graph and create a final schedule of operators that will be ran. Using this schedule of operators, we then aggregate all tensors that are used in the DNN. Each input and output tensor is annotated with the operator from which it was created and all operators that require it as a dependency. All tensor sizes, number of SPMs, and the size of the SPMs are gathered as well. Using this information an initial naive mapping scheme can be created where all inputs and outputs are mapped to a designated input SPM and output SPM. All outputs are assumed to be saved back to memory and reloaded when needed. We then use the initial mapping as an input into the ILP solver to realize the optimal strategy.

Chapter 5

Implementation

5.1 Static Graph Analysis

5.1.1 SMAUG

We first describe the simulation and compiler framework we work with in order to show how tensors and mappings are obtained. SMAUG [19] is an end to end full stack simulation framework for deep learning work loads. As an end-to-end framework SMAUG allows for a high level DSL to be used in order to describe a model so that many deep learning architectures can be evaluated very quickly. Further, SMAUG leverages the gem5-Aladdin [18] simulation framework to describe custom hardware architectures, easily change their configurations, all without the need for a dedicated backend compiler for each custom architecture. This allows researchers to iterate and evaluate on many deep learning workloads, and tune every layer of the stack without the need for RTL design and laborious backed integrations. To the best of our knowledge, no other simulation framework allows for the accurate performance analysis and provides end-to-end configuration support.

From SMAUG we can understand the amount of time and cycles being spent on compute time over DMA transfers. Different deep learning models call for different operators that result in diverse levels of memory and compute boundedness. Being able to analyze operator level differences in DMA and compute times allow us to weight tensors more heavily towards memory bound operations in the final ILP model.

SMAUG is divided into a Python front end used to describe DL topologies, a runtime middle end that handles execution, and a backend of kernels. An overview of these parts are shown in figure 4.1. Models are created using Python APIs that describe what operations, hardware configuration, weights and input data, and which set of acerbated kernels a user wants to use. The front end takes this model description and serialized it into a form the runtime can read and create an execution context around. The runtime takes the serialized model description and creates a computation graph. Example code of a Minerva model description for SMAUG is show in listing 4.1. The graph is used to perform tiling optimizations and an operator schedule. Once an operator schedule is created, the runtime executes each operation sequentially by dispatching each operation as previous operator outputs are received back into main memory. The backend contains all the accelerator kernels and are invoked every time the runtime dispatches an operator. Kernels can be either written in native gem5 or using Aladdin.

Listing 5.1: Model description example

```
"""Create the Minerva network."""

import numpy as np
import smaug as sg
```

```
def generate_random_data(shape):
  r = np.random.RandomState(1234)
  return (r.rand(*shape) * 0.005).astype(np.float16)

def create_minerva_model():
  with sg.Graph(name="minerva_smv", backend="SMV") as graph:
    # Tensors and kernels are initialized as NCHW layout.
    input_tensor = sg.Tensor(
        data_layout=sg.NHWC, tensor_data=generate_random_data((1, 28, 28, 1)))
    fc0_tensor = sg.Tensor(
        data_layout=sg.NC, tensor_data=generate_random_data((256, 784)))
    fc1_tensor = sg.Tensor(
        data_layout=sg.NC, tensor_data=generate_random_data((256, 256)))
    fc2_tensor = sg.Tensor(
        data_layout=sg.NC, tensor_data=generate_random_data((256, 256)))
    fc3_tensor = sg.Tensor(
        data_layout=sg.NC, tensor_data=generate_random_data((10, 256)))

    act = sg.input_data(input_tensor)
    act = sg.nn.mat_mul(act, fc0_tensor, activation="relu")
    act = sg.nn.mat_mul(act, fc1_tensor, activation="relu")
    act = sg.nn.mat_mul(act, fc2_tensor, activation="relu")
    act = sg.nn.mat_mul(act, fc3_tensor)
    return graph
```

5.1.2 Static Analysis

Like OnSRAM, we use the computational graph provided by the SMAUG runtime implemented in C++ to run our static analysis on tensors to devise a SPM management framework. We allow SMAUG to apply its optimizations and graph preprocessing to create a final operator schedule. This schedule is what we use to analyze operator data dependencies and tensor meta information. Figure 4.2 shows how a DNN graph is converted into a schedule where each node has a sequence number in the schedule. The schedule can be visualized a new graph where nodes flow down sequentially in schedule order and vertices show data dependencies. In SMAUG, each operation is represented as a class has a list of associated input tensors, a list of output tensors, and what computation the operation requires. Each tensor is also represented as a class that holds relevant information such as tensor size, dimensions, and dimension layout.

We iterate through the schedule of operators and create a map where an operation name is mapped to its sequence number in the operator schedule For each operator we encounter for each iteration of the schedule, we take note of the operators input and output tensors. These inputs and outputs are used to create an array of unique tensors. This array of tensors is then used to create a mapping of each tensor to a tensor number where every tensor number is the index of the tensor in the array. A map of tensor numbers to the corresponding tensor size is also created. We then build a map of tensor numbers to an array of operation sequence numbers they are required in where the first element of the array represents the operation that the tensor is created and the last element represents the final operation it is used in.

5.1.3 SPM Mapping Representation

The way an SPM mapping is represented is a three dimensional array in which one dimension represents time i.e the operation sequence number, another represents the unique tensors, and the final dimension represents what scratchpad they belong to. Using the dimensions we can show the relationship between what tensor is mapped to what SPM during what operation. We build such a matrix after the static analysis

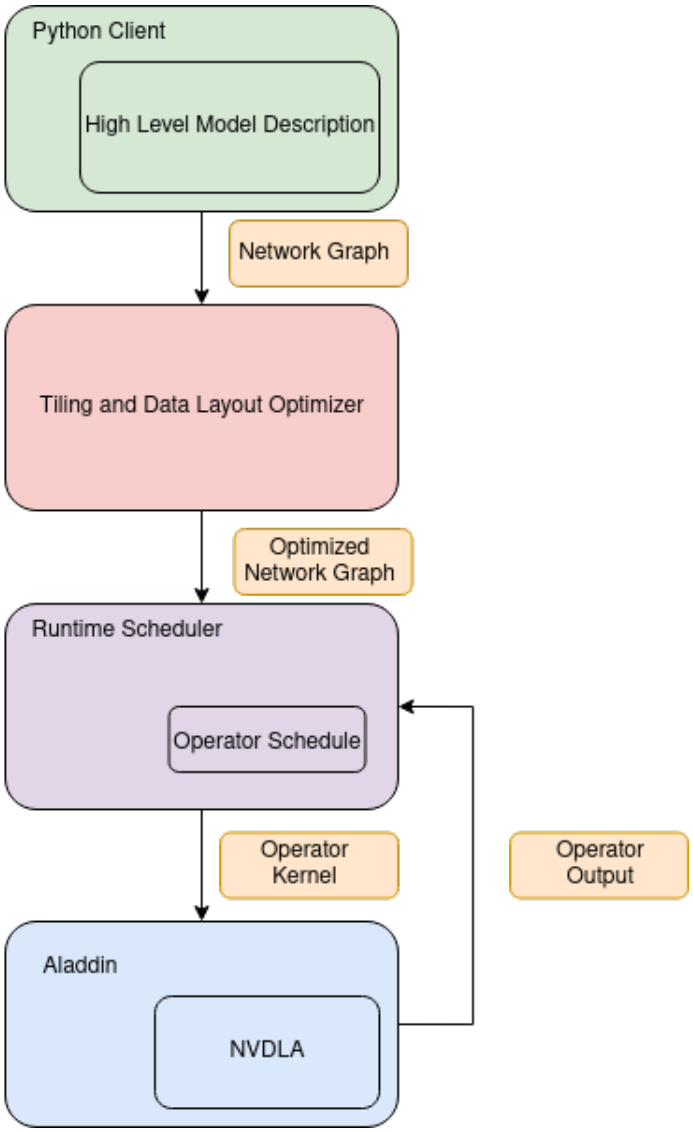

FIGURE 5.1: Overview of SMAUG framework stack

phase to formulate an initial naive SPM mapping representation that would be executed by default on SMAUG. Figure 4.4 illustrates in two dimensions the resulting naive SPM mapping matrix.

5.2 Model Formulation

We now describe our ILP model to solve the SPM mapping optimization problem. The goal is to minimize the amount of cost weighted data transfers as much as possible by pinning as many re-usable tensors as possible. As described in the previous section, we can breakdown a DNN graph into the following variables.

There exists a set of tensors required as inputs and outputs per operation and a unique list of tensors can be constructed.

Let $N \coloneqq \{n \mid n$ represents a tensor ID used in the DNN graph$\}$

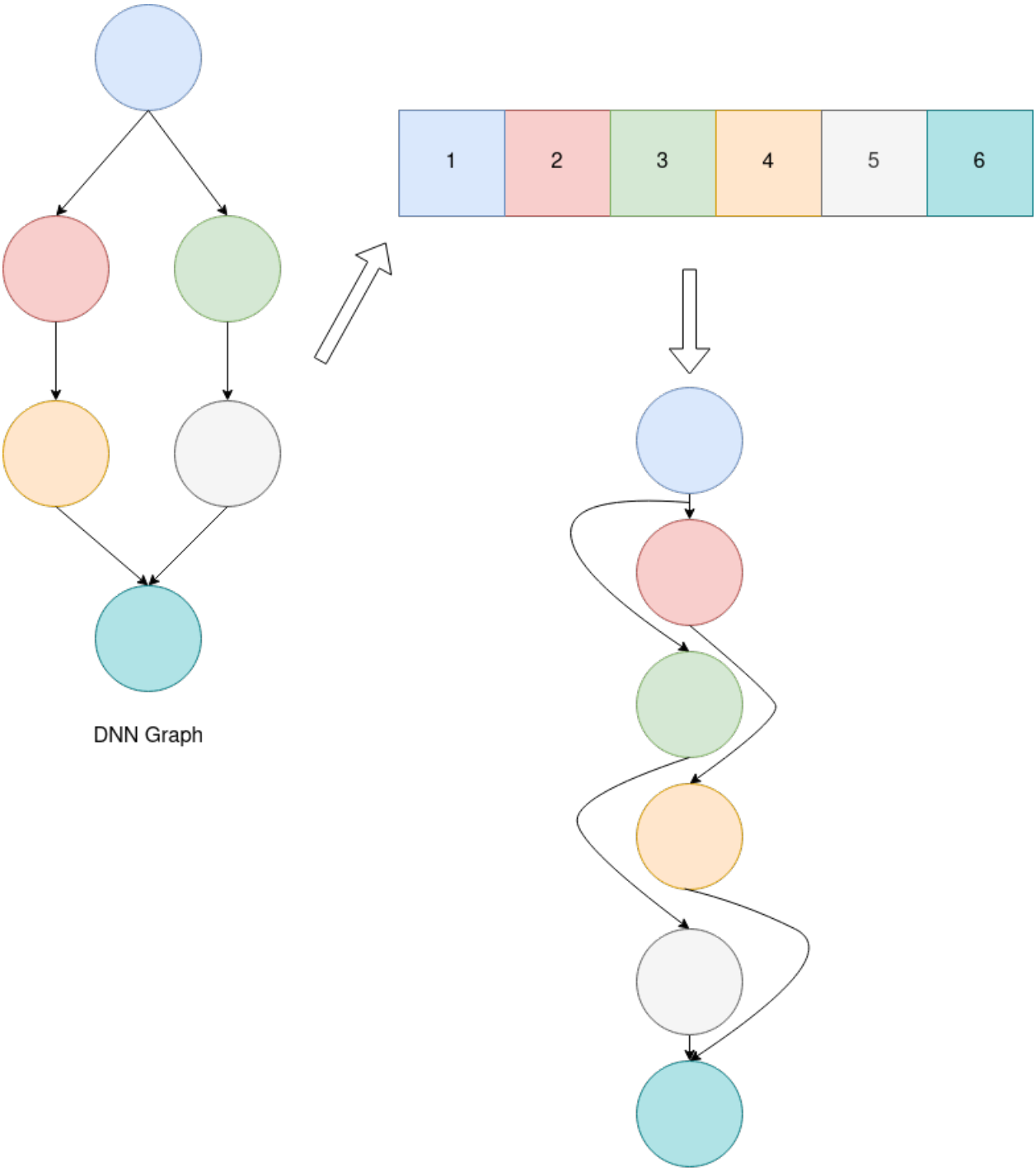

FIGURE 5.2: Illustration of a DNN graph being converted to a schedule

For each DLA architecture, there exists at least 1 scratchpad with a corresponding size. To refer to each scratchpad and its size on the DLA:

Let $K \coloneqq \{k \mid k \text{ represents a scratchpad ID}\}$
Let $Q \coloneqq \{q \mid \text{where } Q_k \text{ represents the size of Scratchpad}_k \in K\}$

Once a graph has been scheduled, each operation can be referenced by its sequence

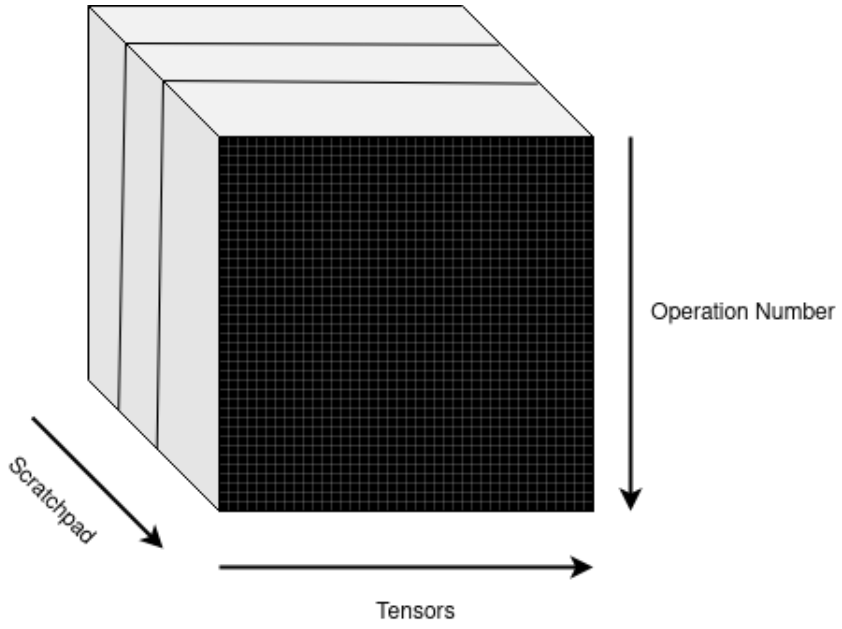

FIGURE 5.3: Illustration of mapping matrix

```
1 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0    1 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0    0 1 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0    0 0 1 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0    0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0    0 1 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0 0 0    0 0 1 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1 0 0    0 0 0 1 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 1 0    0 0 0 0 1 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 1 0
                                                 0 0 0 0 0 0 0 0 0 0 1
```

FIGURE 5.4: Naive SPM mapping matrix representation as an input to the ILP solver

number.

Let $M \coloneqq \{m \mid m \text{ represents an operation in the DNN graph}\}$

Due to limited bus bandwidth, tensors with different sizes will require different amounts of DMA transfers proportional to their size. In order to keep track of the sizes so that we can weight each tensor with a certain cost:

Let $S \coloneqq \{s \mid \text{where } S_n \text{ represents the size of tensor}_n \in N\}$

So the mapping matrix can be described as a three dimensional matrix where every position is a binary variable such that

$x[n][k][m] \in \{0,1\} \forall n,k,m$
$x[n][k][m] = 1$ represents a tensor n that occupies scratchpad k at operation m
$x[n][k][m] = 0$ represents a tensor n does not exist on scratchpad k at operation m

If $x[n][k][m] = 0 \forall k$ then it means that tensor n is saved to main memory at operation m.

5.2.1 Objective Function

The objective function we would like to solve for such that we minimize the amount of cost weighted data transfers is the following:

$$min(\sum_{n}\sum_{k}\sum_{m} S_n * (|x[n][k][m+1] - x[n][k][m]|))$$

A difference between $|x[n][k][m+1] - x[n][k][m]|$ represents either a data transfer of tensor n from SPM k to main memory if $x[n][k][m+1] - x[n][k][m] = -1$ or main memory to SPM if $x[n][k][m+1] - x[n][k][m] = 1$. $|x[n][k][m+1] - x[n][k][m]|$ shows that either a tensor n remains on SPM k between operations m and $m+1$ or stays in main memory.

By minimizing the sum of total data transfers multiplied by the tensor size, we minimize on a cost basis the amount of DMA loads and stores and maximize tensors being reused on the SPM. However, the absolute value in the objective function would make it non-linear. Thus, we add auxiliary variables to remove the absolute values and reformulate the objective function as:

$$min(\sum_{n}\sum_{k}\sum_{m} y[n][k][m])$$

5.2.2 Constraints

We now describe the following set of constraints on the model:

- All necessary input and output tensors for a given operation will be present on the SPMs

  Let $A \coloneqq \{a \mid$ where $A[n] = 1$ represents a tensor$_n$ is required as an input or output for operation

  $$A[n][m] \in \{0,1\} \forall n,m$$

  $$A[n][m] = 1 \implies \sum_{i \in K} x[n][i][m] = 1$$

- All tensors mapped on an SPM$_k$ must fit on within the given scratchpad space

  $\sum_{i \in N} x[i][k][m] * s[i] \leq Q[k] \forall m,k$

- Tensors are not mapped before the operation in which they're lifetime begins

Let $B \coloneqq \{b_n \mid b_n \text{represents the start time for tensor } n\}$

$x[n][m][k] = 0 \forall m < B_m$

- Tensors are not mapped after the operation in which they're lifetime ends

  Let $E \coloneqq \{e_n \mid e_n \text{represents time for tensor } n\}$

  $x[n][m][k] = 0 \forall m > E_m$

- Absolute Value Constraints
  Let $Y = \{y[0][0][0], y[0][0][1], ...y[n][k][m]\prime\}$
  $(x[n][k][m+1] - x[n][k][m]) * S_n <= y[n][k][m]$
  $(x[n][k][m] - x[n][k][m+1]) * S_n <= y[n][k][m]$

5.2.3 Solver

This model description can now be used to find the exact solution to the SPM mapping problem. We use the Gurobi ILP solver and the Python API to create our optimized mapping matrix. The listed constraints can be used as is and the original X variable is initialized with the naive mapping matrix. An illustration of the optimized matrix is show in figure 4.5. This new matrix can be used to integrate a pinning strategy in the framework.

```
0 0 0 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0 0 0 0      0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0 0 0 0      0 1 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0 0 0 0      0 1 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0      1 0 0 0 0 0 0 0 0 0 0      0 1 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0      1 0 0 0 1 0 0 0 0 0 0      0 1 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0      1 0 0 0 1 1 0 0 0 0 0      0 1 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0      1 0 0 0 1 1 1 0 0 0 0      0 1 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0      1 0 0 0 1 1 1 1 0 0 0      0 0 1 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0      1 0 0 0 1 1 1 1 1 0 0      0 0 1 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0      1 0 0 0 1 1 1 1 1 1 1      0 0 1 0 0 0 0 0 0 0 0
```

FIGURE 5.5: Optimal SPM mapping matrix representation as an input to the ILP solver

Chapter 6

Evaluation Method

6.1 Data Collection

To do evaluation, we first gather baseline performance data from SMAUG. The DLA architecture we choose to simulate is the DLA provided by SMAUG: a NVDLA-inspired convolution engine with 32-way multiply-accumulate (MACC) arrays and three SPMs. We configure and test our DLA with a constant three SPM configuration with varying sizes. We test on 32Kb, 64Kb, 128Kb, 256Kb, 512Kb, 1024Kb, 2048Kb size variations. For each configuration, the following models were evaluated on: VGG, Lenet5, Minerva, Resnet, Cifar-CNN, Elu, Large-Elu, and LSTM. The baseline strategy is ran in conjunction with the intra-node optimizations SMAUG applies already such as tile reordering, tile chunk size tuning, loop unrolling, and pipelined DMA transfers. Thus, the metrics gathered using our pinning strategy are in effect on top of any gains made by intra-node optimizations.

The metrics we consider most important to measure are the speedup and total data transfer reduction. We show total speedup to see how much of a practical performance gain we can obtain using an ideal pining strategy for all workloads. We are interested in the total data transfer reduction since the speedup is dependant on how memory bound a particular DNN model's workload is based on the type of operations it uses and its graph connections. Thus, based on the memory boundness of a model, we can estimate how close we reach the theoretical upper bound, i.e no DMA transfers at all, of speedup we achieve with our model.

The baseline performance for a non-optimized SPM managmenet strategy is collected by running the simulation in SMAUG and reading the total accelerator and DMA cycles used to run inference. The DMA cycles are included in the total accelerator cycles, so we can calculate the fraction of inference time spent on DMA transfers, hence get a metric of how memory bound a particular DNN model is.

$$Memory\ Boundness = (total\ accelerator\ cycles - DMA\ cycles)/total\ accelerator\ cycles$$

The upper bound of potential speedup from memory transfer savings is given by

$$Potential\ Speedup = 1/(1 - Memory\ Boundness)$$

Such a speed up is only possible when there are no memory transfers at all and all inputs and weights are preloaded from the beginning.

The number of data transfers from the each SPM management strategy is obtained by summing over the number of transitions in the mapping matrix. We compare the two sums between the non-optimized and optimized strategies to estimate a speed up and DMA transfers saved factor. We call the fraction of data transfers saved over the total data transfers of the unoptimized strategy the reduction factor.

$$Total\ Data\ Transfers = \sum_{n}\sum_{k}\sum_{m} |x[n][k][m+1] - x[n][k][m]|$$

$$Data\ Transfers\ Saved = Total\ Data\ Transfers\ UnOptimized - Total\ Data\ Transfers\ Optimzed$$

$$Reduction\ Factor = Data\ Transfers\ Saved / Total\ Data\ Transfers\ UnOptimized$$

With the number of total data transfers saved, we can estimate a potential speedup based on the memory boundless of a particular model. This is done by estimating what the new total cycles would be by subtracting the saved DMA cycles based on the reduction factor. The formula to obtain the speedup is as follows:

$$Speedup = (total\ accelerator\ cycles / (total\ accelerator\ cycles - DMA\ Cycles * Reduction\ Factor))$$

6.2 Limiting Factors

A key limiting factor to our model and how we evaluate our cost savings is how we model tiling. Tiling is invisible from an inter-graph level. As a consequence, for inputs and outputs that are required to be tiled due to the tensor size being larger than any scratchpad on the DLA, it would break the ILP constraint that tensors must be at most the size of the scratchpad. To properly take tiling into account, a graph can be reconfigured such that a single operation broken up into many operations sequentially scheduled next to each other as shown in figure 5.1. This will show the effects of the extra data transfers necessary such that the ILP model can account for them. We leave the extension to reconfigure graphs to incorporate tiling for future work. Instead, for our purposes to get estimate upper bound achievable performance metrics, we resize all tensors that must be tiled to the scratchpad size. This way we force tensors that are not immediately reused to be evicted from the scratchpad. Further, because the baseline model also does not take into account tiling, the speedup and transfer costs may be consistent on a relative basis for a graph reconstructed to account for tiling.

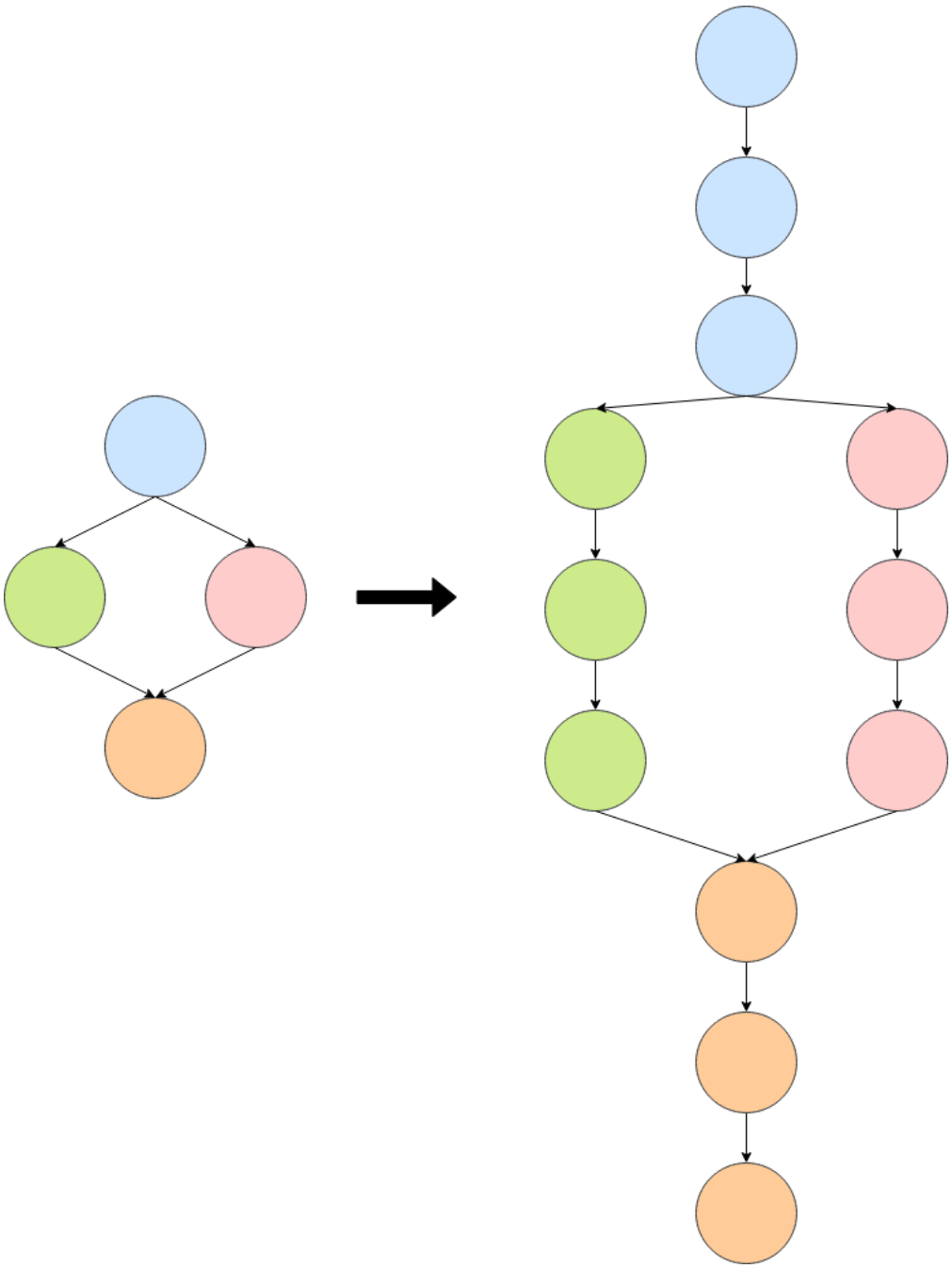

FIGURE 6.1: A graph reconverted to account for tiling

Chapter 7

Experiments and Results

7.1 Speedup

We show the estimated performance benefits of our optimal strategy in figure 6.1. The figure shows the speedup as a percentage of the baseline performance. We show that we can more than a 1.5x improvement in performance using our spm management strategy for even the smallest spm size we test on. It also shows that we show that we reach close to our roofline performance of a infinite scratchpad even for our base architecture configuration of a 32kb size per scratchpad for some models.

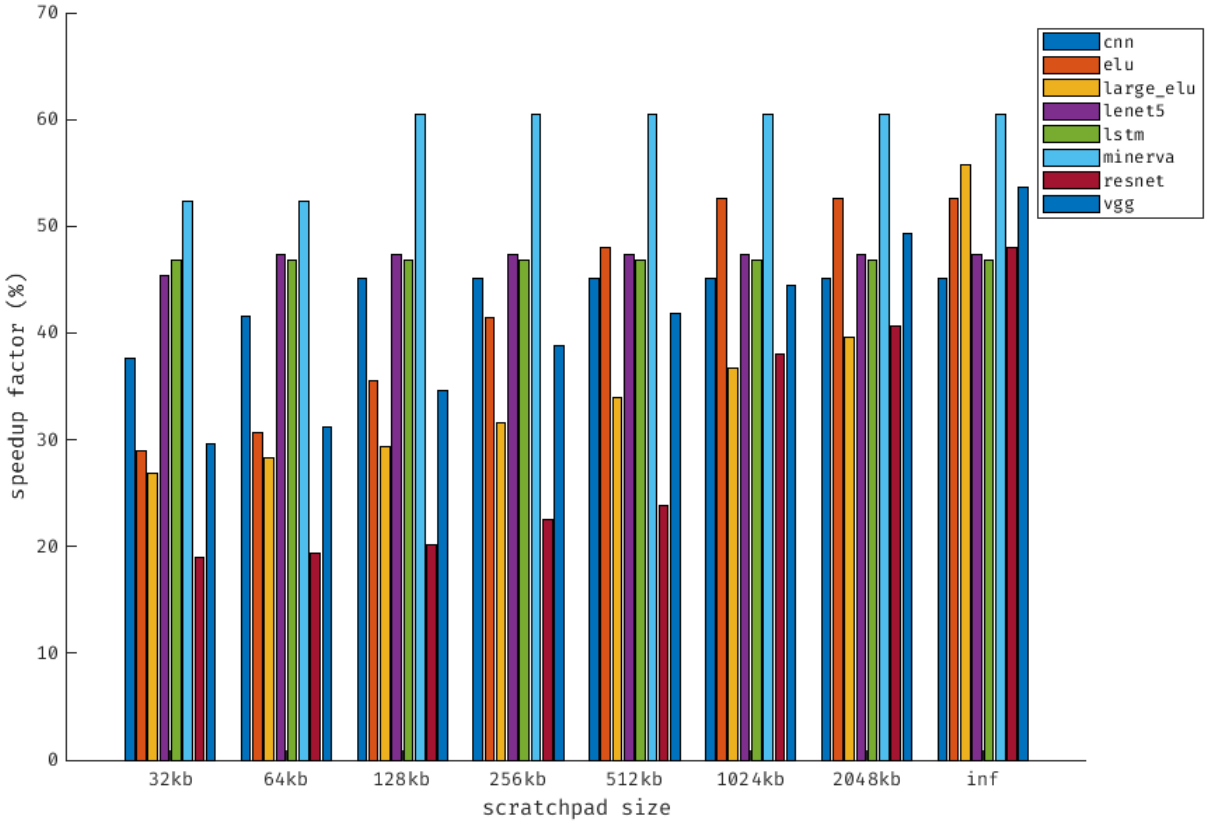

FIGURE 7.1: Estimated speedup for varying scratchpad sizes and models using the optimal pinning strategy

The table 6.1 shows how memory bound each model architecture is, i.e the fraction of total cycles spent on DMA transfers, and the possible speedup if one were to

Model	Memory Boundness	Theoretical Speedup	Inf SPM Speedup	Average Speedup
CNN	39.83%	66.20%	45.16%	41.81%
Lenet5	40.79%	68.89%	47.30%	42.91%
Lstm	42.27%	73.21%	47.76%	42.93%
Minerva	47.65%	91.04%	60.46%	43.10%
Elu	43.81%	77.96%	52.66%	42.08%
Large-Elu	45.21%	82.51%	55.81%	42.55%
Resnet	41.50%	70.95%	48.01%	43.10%
VGG	44.12%	78.95%	53.65%	43.71%

TABLE 7.1: Memory Boundness, upper bound of speedup if no DMA transfers were needed at all, the speedup possible from an SPM with infinite capacity, the average speedup for all capacities tested on for each model

eliminate DMA transfers in totality. Next to those metrics, the possible speedup gained if we applied our strategy to an infinite capacity SPM is shown next to the average gained speed up for all SPM sizes tested on. On average we achieve at least a 1.41x speedup across all models. At a minimum 1.19x speedup can be obtained for the worst case and 1.60x speedup for the best case.

Figure 6.2 shows for each SPM capacity tested, how much of the maximum achieved speedup given an infinite sized SPM our pinning strategy obtained. It can be seen that DNN workloads with sequential operator scheduling such as Lenet5, Minerva, and Elu quickly converge to the maximum possible speedup given an infinite SPM since every output simply gets pinned for immediate reuse until there there is a capacity limit. At the worst case for Resent on the 32Kb SPM size, we obtain a 39.5% of the max achievable speedup and 100% for LSTM, Minerva, Lenet5, and CNN for SPM sizes of at least 128Kb.

7.2 Reduction of DMA Transfers

Figure 6.3 shows a plot of how much of the total necessary transfers were avoided through our pinning strategy. It is noted that the infinite SPM still does not meet 100% data transfers saved since inputs and weights must still be initially loaded into the SPM.

Similar to the speedup increase as scratchpad size increases, we see a linear correlation of increased scratchpad size and the total data transfers saved. For the same models where the maximum efficiency was obtained, we see that the savings in DMA costs are also close to the maximum achievable savings. Figure 6.4 illustrates this by showing the percentage of total DMA transfers incurred versus the minimum achievable DMA transfers incurred by an infinite capacity SPM for each SPM size tested.

It appears that regardless of how sequential operators may seem to be connected in a graph, the scheduled ordering of the operators matters much much more in terms of how reusable the outputs become. Clearly, even though LSTM seems much more complex of an architecture with many diverging paths, our pinning strategy still achieves its maximum possible speedup before sequential models such as Minerva and obtains a higher overall speedup to VGG.

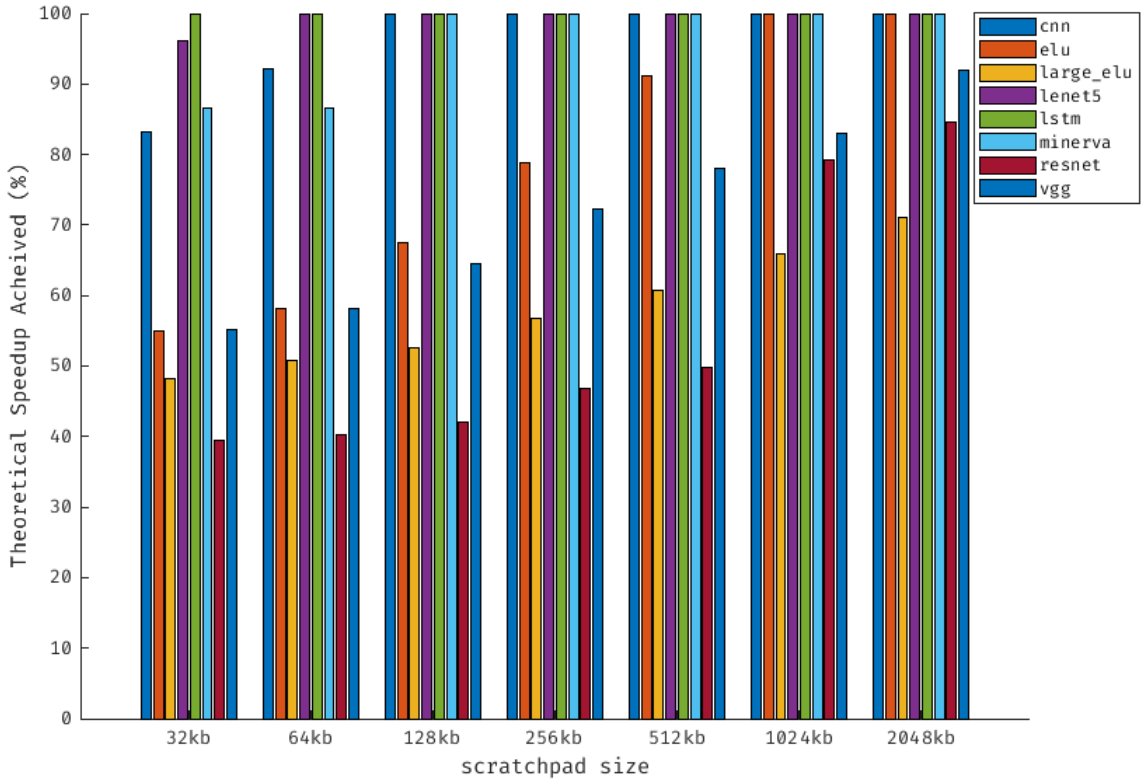

FIGURE 7.2: How much of the theoretical speedup from an infinite sized SPM was achieved in (%) for each SPM capacity

7.3 Discussion of Results

Given our results, we conclude that a pinning strategy on a graph with no tiling and capped tensor sized can achieve at least 40% of the theoretical maximum speedup and DMA cost savings and for some cases the maximum. We note that, although the maximum capped tensor size was scaled to be the size of the SPM for each experiment, we still see a linear increase in speedup and DMA cost savings as the SPM capacity increases. In some cases, the size of the tensor does not matter due to the workload characteristics and operator schedule.

The results are difficult to compare against different hardware arhictures, simulators, models, and number of scratchpads. Still, we compare OnSRAM results to ours. OnSRAM tests on a 2MB SPM DLA so we will be comparing our 1024KB scratchpad results as the closest comparison in terms of total SPM capacity. OnSRAM-static obtains an average of 50% increase in performance relative to their baseline SPM management strategy and achieves 90% of the ideal infinite SPM [6]. However, out of the 13 DNN workloads evaluated on, only 5 of those models reach over a 50% increase while the rest remain under a 20-30% speedup. In comparison we achieve a minimum of a 40% speedup across all workloads from our baseline 32Kb configuration and a minimum of 65% of ideal speedup obtained for the 1024Kb configuration. The models that both our work and OnSRAM has evaluated include: VGG, Lenet, and Resnet. For all three cases, both OnSRAM-Static and our strategy gains most of the of ideal performance possible. Overall, it seems as though the greedy strategy achieves close

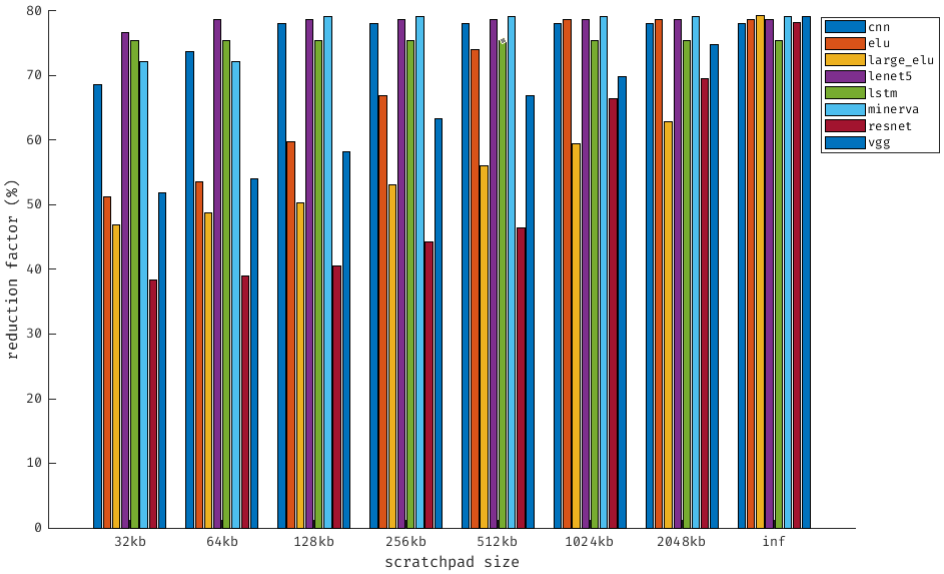

FIGURE 7.3: Estimated % of data transfers saved for varying scratchpad sizes and models using the optimal pinning strategy

to the optimal, and in some cases the optimal, level of performance possible. This may mean for the single scratchpad case, the overhead required for an ILP solver may not be necessary.

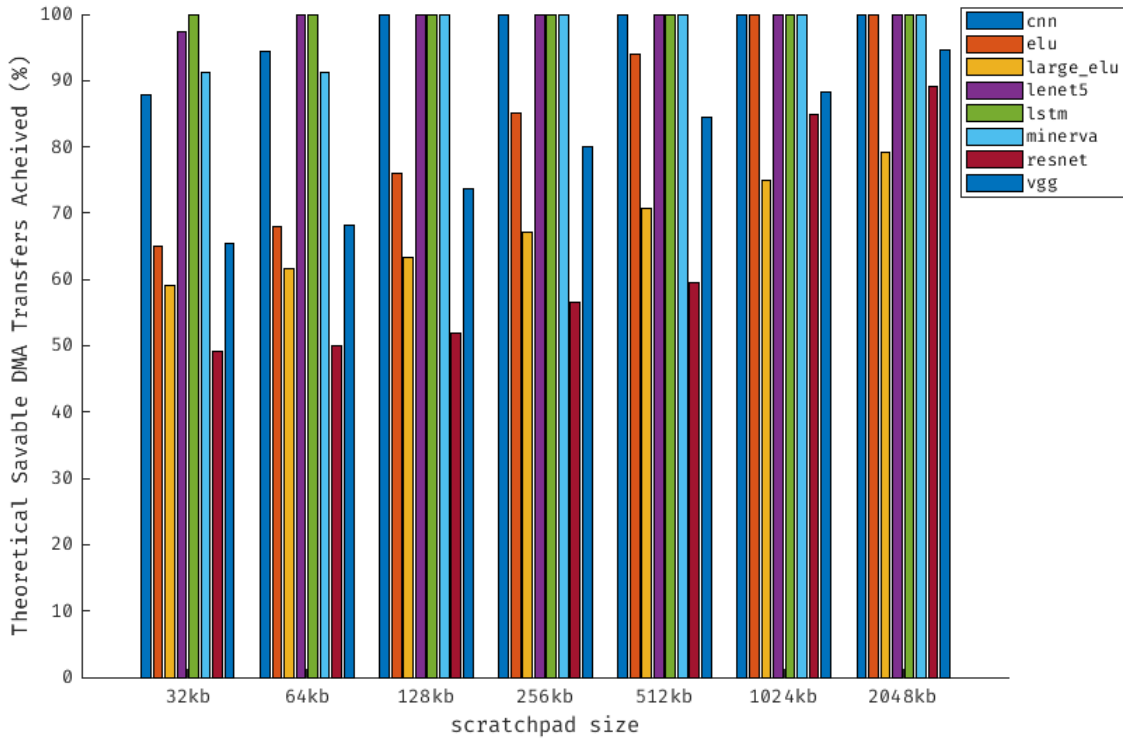

FIGURE 7.4: Percentage of total DMA transfers incurred versus the minimum achievable DMA transfers incurred by an infinite capacity SPM for each SPM size tested

Chapter 8

Future Work and Conclusion

8.1 Conclusion

As DNN model sizes continue to grow and DLAs become more prevalent, it becomes necessary to remove the software bottlenecks through good programming. We provide a solution via a memory management technique for scratchpads on an inter-node level for DLAs.

8.2 Future Work

1. A graph reconstructor to account for tiling so tensors that are larger than the SPM will be modelled more accurately than being size capped to the SPM size.

2. Integration and testing into SMAUG or a full compiler extension that is inter-operable with other computational graph frameworks

3. Comparing the ILP solution to graph coloring and heursitic approaches for the many scratchpad case.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. 2016. arXiv: 1603.04467 [cs.DC].

[2] *NVDLA Home Page*. https://nvdla.org/. Accessed: 2022-11-05.

[3] Ofri Wechsler, Michael Behar, and Bharat Daga. "Spring Hill (NNP-I 1000) Intel's Data Center Inference Chip". In: *2019 IEEE Hot Chips 31 Symposium (HCS)* (2019), pp. 1–12.

[4] Xiaohan Tao, Jianmin Pang, Jinlong Xu, et al. "Compiler-directed scratchpad memory data transfer optimization for multithreaded applications on a heterogeneous many-core architecture". In: *The Journal of Supercomputing* 77 (Dec. 2021). DOI: 10.1007/s11227-021-03853-x.

[5] Lian Li, Hui Feng, and Jingling Xue. "Compiler-Directed Scratchpad Memory Management via Graph Coloring". In: *ACM Trans. Archit. Code Optim.* 6.3 (2009). ISSN: 1544-3566. DOI: 10.1145/1582710.1582711. URL: https://doi.org/10.1145/1582710.1582711.

[6] Subhankar Pal, Swagath Venkataramani, Viji Srinivasan, et al. "OnSRAM: Efficient Inter-Node On-Chip Scratchpad Management in Deep Learning Accelerators". In: *ACM Trans. Embed. Comput. Syst.* 21.6 (2022). ISSN: 1539-9087. DOI: 10.1145/3530909. URL: https://doi.org/10.1145/3530909.

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, et al. "TVM: End-to-End Optimization Stack for Deep Learning". In: *CoRR* abs/1802.04799 (2018). arXiv: 1802.04799. URL: http://arxiv.org/abs/1802.04799.

[8] Lian Li, Lin Gao, and Jingling Xue. "Memory coloring: a compiler approach for scratchpad memory management". In: *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. 2005, pp. 329–338. DOI: 10.1109/PACT.2005.27.

[9] Jiansong Li, Wei Cao, Xiao Dong, et al. "Compiler-Assisted Operator Template Library for DNN Accelerators". In: *Int. J. Parallel Program.* 49.5 (2021), 628–645. ISSN: 0885-7458. DOI: 10.1007/s10766-021-00701-6. URL: https://doi.org/10.1007/s10766-021-00701-6.

[10] Preston Briggs, Keith D. Cooper, and Linda Torczon. "Improvements to Graph Coloring Register Allocation". In: *ACM Trans. Program. Lang. Syst.* 16.3 (1994), 428–455. ISSN: 0164-0925. DOI: 10.1145/177492.177575. URL: https://doi.org/10.1145/177492.177575.

[11] M. Verma, L. Wehmeyer, and P. Marweclel. "Dynamic overlay of scratchpad memory for energy minimization". In: *International Conference on Hardware/-Software Codesign and System Synthesis, 2004. CODES + ISSS 2004.* 2004, pp. 104–109.

[12] Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, et al. *Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning*. 2018. arXiv: 1801.08058 [cs.DC].

[13] Dong Yu, Adam Eversole, Mike Seltzer, et al. *An Introduction to Computational Networks and the Computational Network Toolkit*. Tech. rep. MSR-TR-2014-112. 2014. URL: https://www.microsoft.com/en-us/research/publication/an-introduction-to-computational-networks-and-the-computational-network-toolkit/.

[14] Richard Wei, Lane Schwartz, and Vikram Adve. *DLVM: A modern compiler infrastructure for deep learning systems*. 2018. arXiv: 1711.03016 [cs.PL].

[15] Tian Jin, Gheorghe-Teodor Bercea, Tung D. Le, et al. *Compiling ONNX Neural Network Models Using MLIR*. 2020. arXiv: 2008.08272 [cs.PL].

[16] *NVIDIA TensorRT Docs Hub*. https://docs.nvidia.com/tensorrt/index.html. Accessed: 2022-11-05.

[17] Adam Paszke, Sam Gross, Francisco Massa, et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: 1912.01703 [cs.LG].

[18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, et al. "TVM: End-to-End Optimization Stack for Deep Learning". In: *CoRR* abs/1802.04799 (2018). arXiv: 1802.04799. URL: http://arxiv.org/abs/1802.04799.

[19] Sam Likun Xi, Yuan Yao, Kshitij Bhardwaj, et al. *SMAUG: End-to-End Full-Stack Simulation Infrastructure for Deep Learning Workloads*. 2019. arXiv: 1912.04481 [cs.LG].