Tapeflow: Streaming Gradient Tapes in Automatic Differentiation

by

Milad Hakimi

B.Sc., University of Tehran, 2020

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

Declaration of Committee

Name:	**Milad Hakimi**
Degree:	**Master of Science**
Thesis title:	**Tapeflow: Streaming Gradient Tapes in Automatic Differentiation**
Committee:	**Chair:** Saba Alimadadi Assistant Professor, Computing Science
	Arrvindh Shriraman Supervisor Associate Professor, Computing Science
	Alaa Alameldeen Committee Member Associate Professor, Computing Science
	Zhenman Fang Examiner Assistant Professor, Engineering Science

Abstract

Computing gradients is a fundamental task in a variety of fields, ranging from machine learning [3, 32] to physics simulations [12] and scientific computing [50]. Automatic differentiation (AD) [25, 24] is a widely used technique for computing gradients of arbitrary imperative code. In AD, the gradient tape is an auxiliary data structure that is used to transfer intermediary values required for gradient computation. However, managing the gradient tape in memory presents a significant challenge, as existing approaches suffer from limitations in spatial reuse, on-chip energy consumption, and cache size requirements. These limitations highlight the need for a new and efficient solution.

We present Tapeflow, a novel compiler framework that efficiently orchestrates and manages the gradient tape. Our approach offers three key contributions. First, we introduce the concept of *regions*, which transform the tape layout into an array-of-structs format, enhancing spatial reuse. Second, we schedule the execution into *layers* and explicitly orchestrate the tape operands using a scratchpad, effectively reducing required cache size and on-chip energy. Third, we stream the tape from the DRAM by organizing it into a FIFO of tiles, with tape operands arriving just-in-time for each layer. To evaluate the effectiveness of our approach, we conducted experiments on a wide range of algorithms across various domains. Our results show that Tapeflow outperforms Enzyme, the state-of-the-art compiler, by 1.3-2.5×, reduces on-chip SRAM usage by 5-40×, and saves 8× on-chip energy. By providing a general solution that does not rely on domain-specific knowledge, Tapeflow has the potential to significantly improve gradient tape management in various applications.

Keywords: Automatic Differentiation; Decoupled Access-Execute; LLVM; HPC; Dataflow Architecture; Machine Learning;

Dedication

To my best friend, Tarannom, who has made my life truly magical.
To my mother, who has been my inspiration to follow my dreams.

Acknowledgements

I am incredibly grateful to my supervisor, Arrvindh Shriraman, for his invaluable guidance and boundless patience throughout this endeavor.

I would also like to express my sincere appreciation to Ali Sedaghati, a remarkable friend and colleague, whose effective assistance has been indispensable in the completion of this project. Furthermore, I would like to extend my heartfelt thanks to my esteemed colleagues Minh, Reza, Amirali, Aditya, Anagha, Mohammadhossein, and Alireza for fostering such a professional research environment.

I am also grateful to my friends Mohammadreza, Mehdi, Mohammad, and Sina for their support throughout this journey.

I could not have undertaken this journey without my family, especially my spouse, Tarannom. Her patience and belief in me have been a constant source of motivation and encouragement during this process.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Automatic differentiation (AD) is a generalization of back-propagation in deep learning [45]. AD supports learning in arbitrary functions, instead of being restricted to blackbox DNN operators [50, 22, 31]. This enables control-flow-driven, dynamic, data-dependent learning models [39, 47]. AD reduces the amount of training data required for learning. Consequently, the community has been active in supporting gradient calculations in data structures [51, 8], C++ operators [4, 6, 9, 29], functional [35, 49], and imperative code [3, 24]. However, they do not characterize the performance challenges in AD.

1.1 Applications of Automatic Differentiation

Currently, AD targets problems involving optimization, parameter estimation, inverse problems in different fields such as scientific computing [5], robotics [12] and physics simulations [17, 53]. In these problems, AD provides a powerful tool for computing gradients of functions with respect to their inputs, which is essential for optimization and parameter estimation. For example, in scientific computing, AD can be used to compute gradients of complex simulations, such as molecular dynamics simulations, to optimize the parameters of the simulation and improve its accuracy. In robotics, AD can be used to compute gradients of the cost function that measures the performance of a robot, allowing for the optimization of the robot's behavior. In physics simulations, AD can be used to compute gradients of the equations of motion, enabling the optimization of the simulation parameters and the prediction of the system's behavior.

To better understand the concept of AD, let us consider a simple example illustrated in Figure 1.1. The example involves a cannon trying to hit a target, where the forward function calculates how the cannon fires the ball, i.e., $target(t) = f(angle = \theta, velocity = v)$. The goal is to find an unknown inverse function that derives the control parameters $\theta, v = f^{-1}(target = t)$ given a target (t). One approach to solving this problem is to approximate the unknown function f^{-1} using a neural network. However, training the neural network is challenging because there is no clear way to measure the loss for the output variables θ and v as we do not know the ground truth for those variables.

To address this challenge, we can use automatic differentiation to train the neural network. As shown in Figure 1.1, during training, the neural network's outputs, θ and v, are fed to the physics model to produce a target (t). The resulting target is then used to calculate the loss. The physics model is written in imperative code, and automatic differentiation deals with back-propagation through the model. The AD compiler breaks down the expression into intermediate terms (the SSA expressions), and the intermediate values (color-coded values in the figure) then flow through a reverse computation graph using the chain rule. This enables a single dataflow pass to compute the derivative with respect to all inputs.

In summary, the example illustrates how automatic differentiation can be used to train a neural network to approximate an unknown inverse function. By using AD to compute the gradients of the physics model, we can train the neural network to produce the desired

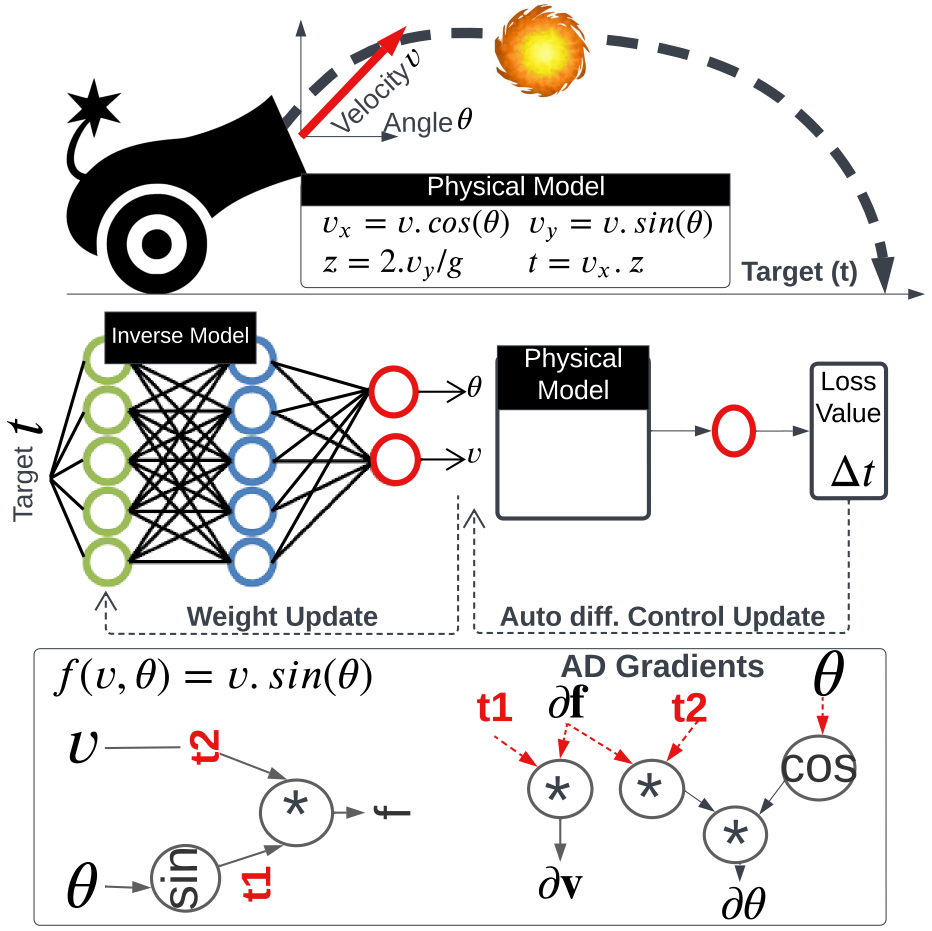


Figure 1.1: Illustration of Automatic Differentiation application. Top) The physical model that describes the world. Middle) The function for training the values, θ and v composed of a neural network and the physical model. Bottom) Dataflow of the physical model in both forward and reverse phase.

output variables θ and v that result in hitting the target. This approach demonstrates the power and versatility of automatic differentiation in solving complex problems in various domains.

1.2 Challenges of AD

The central challenge in making AD performant is the tape, an auxiliary data structure that holds the intermediary values required to calculate gradients. Figure 1.2 shows the gradient code generated by Enzyme [24], a state-of-the-art AD compiler. The gradient calculation involves two phases. i) Forward phase (FWD) that contains the code of the original function for which we wish to calculate gradients. All the instructions in the original function would appear in the FWD phase. ii) Reverse phase (REV) which computes the gradients (e.g., `d_x[i]`). During FWD, the compiler saves a shadow value on the tape for each Static Single Assignment (SSA) register, such as the output of `eval()`, which is then consumed during the REV. We refer to these intermediate values as tape values because they are stored in

```
def original(x[],y[]):
 for i:0 to N:
  sum += eval(y[i])*x[i]
 return sum
```

```
def gradient(x[],y[],d_sum):
 // Forward
 for i:0 to N:
  sum += eval(y[i])*x[i]
   TAPE[i] = eval(y[i])
 // Reverse
 for i=N to 0:
   d_x[i] += TAPE[i]*d_sum
 return d_x
```

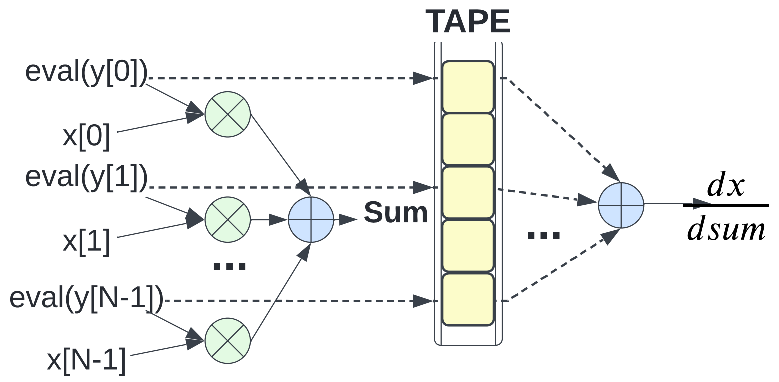


Figure 1.2: Illustration of gradient function. Top) Original function and its gradient function: The AD compiler creates the gradient function by placing the body of the original function in the forward phase and the gradient calculation of the forward operations in the reverse phase. The intermediate values between forward and reverse are passed through the `Tape[]`. Bottom) Dataflow of the gradient function. It shows the tape accesses during FWD and REV.

the tape. Currently, Enzyme focuses on the functionality of the tape but does not study the performance limitations introduced by tape layout and access pattern.

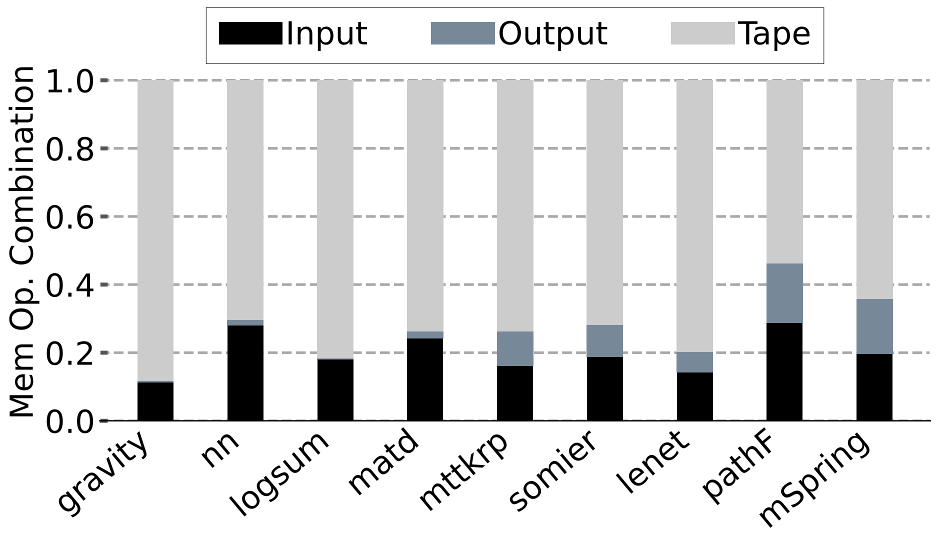


Figure 1.3: State Distribution in Automatic Differentiation.

The tape introduces several challenges such as:

- **Long lifetime of tape operands:** Tape values can only be used by the Reverse (REV) pass once the Forward (FWD) pass is finished, necessitating buffering of the tape throughout the FWD process.

- **Large size of the tape:** The tape is required to store the entire Static Single Assignment (SSA) state, which increases in proportion to the number of operations in the FWD pass.

- **Irregular tape reads during REV:** The tape has to be a general-purpose data structure that can accommodate arbitrary memory accesses and loop patterns. In contrast, DNNs work with blackbox operators and can organize gradients neatly into tensors.

- **Inconsistent reuse patterns of tape operands:** The gradient calculation reverses the dataflow in the REV pass, causing tape values generated earlier in the FWD to pass to be consumed later in the REV pass. This disrupts temporal locality. While the same pattern appears in the DNNs, the granularity of the accesses (coarse-grained tensors) and the separation of compute kernels through the abstractions (layer) make it easier to reason about the reuse distances. Chapter 2 includes a thorough comparison between DNNs and general purpose AD.

In Figure 1.3, we demonstrate the problem of the tape by analyzing a gradient function generated by Enzyme [24]. The tape increases the intermediate state by 2–4×. The state-of-

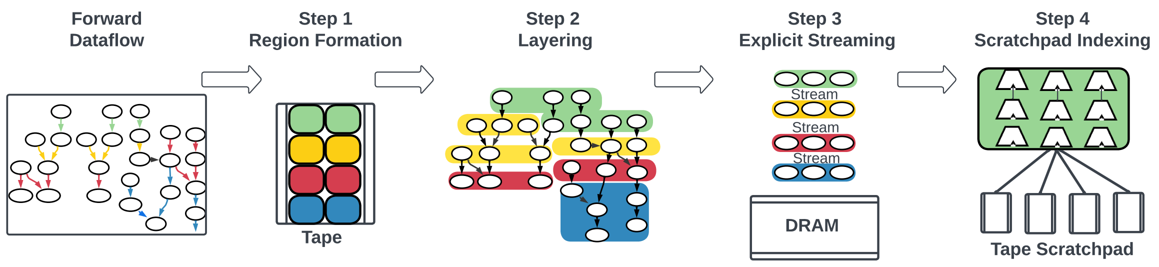


Figure 1.4: Overview of Tapeflow. Step 0: the input of the Tapeflow is the baseline dataflow. Step 1: Changing the tape layout. Step 2: The compiler splits the dataflow into layers. Step 3: Stream the tape from DRAM to layers. Step 4: statically schedule the execution of the FWD and REV layers.

the-art AD compilers [24, 18, 52] rely on the cache to implicitly optimize the tape accesses. However, caches and on-chip hierarchies can not optimize these accesses because they can not store the entire tape and capture the producer-consumer locality between the FWD and REV. The figure also classifies other accesses in the REV which increases overall working set (relative to the FWD): i) Input: immutable state that REV can reference directly, e.g., `x[]` in Figure 1.2. ii) Output: mutable, short-lived registers that form the output of the FWD and input to the REV. and iii) Tape: SSA values produced by FWD and consumed by the REV.

1.3 Overview of Tapeflow

Tapeflow introduces a method for creating high-performance tapes in AD, addressing the limitations of current AD compilers that do not optimize the tape. This method consists of two key components: tape management and tape orchestration. Tape management deals with the layout of operands within the tape. Tapeflow analyzes the FWD/REV dataflow to determine when to use the tape operands and creates a spatially optimized tape layout. Tape orchestration deals with how the tape is read from the DRAM and optimized for the temporal locality. We observed that imperative code and program graphs, despite being seemingly irregular and arbitrary, exhibit familiar layer-like behavior (like DNNs). We exploit this information to pipeline the tape segments and make the tape more amenable to streaming.

Tapeflow optimizes the tape through multiple passes, as shown in Figure 1.4. The process involves the following steps:

i) **Region Creation:** The tape is divided into coarse-grain regions based on producer-consumer lifetime. Each region is organized in an array-of-structs layout, which groups tape values with similar lifetime to improve spatial efficiency and utilization.

ii) **Layering:** FWD and REV are scheduled into sub-dataflows, and each layer is constrained to access tape operands in a tiled fashion. Unlike DNNs, Tapeflow's layers are compiler-generated and can be applied to arbitrary program graphs.
iii) **Explicit Streaming:** The tape is explicitly orchestrated, and stream operations are introduced at layer boundaries to move data from DRAM. This enables transfers with few registers (32-64).
iv) **Indexing:** The compiler generates the correct index for all tape accesses during compile time, redirecting the tape's DRAM loads and stores to the scratchpad.
Overall, Tapeflow's tape optimization process improves spatial efficiency and utilization, while also reducing the number of registers needed for data transfers.

Tapeflow is built as an extension to Enzyme [24], a state-of-the-art AD compiler that relies on hardware caches. In our evaluation, we compared Tapeflow's performance against Enzyme. We evaluate Tapeflow on a generic spatial architecture, generated using high-level-synthesis, and simulated in detail using gem5-salam [44].

1.4 Contributions

- **Tape Characterization:** As far as we are aware, we are the first paper to characterize the architectural implications and performance challenges of the AD tape.
- **Compiler for Tape Management:** This thesis presents Tapeflow, a compiler that optimizes the tape layout by using a novel array-of-structs layout and rearranging operands to improve spatial efficiency.
- **Explicit Tape Orchestration:** Tapeflow organizes arbitrary imperative code into pipelined layers, and converts the tape into an explicitly decoupled stream.
- **Improve hardware efficiency of tapes:** Compared to Enzyme, Tapeflow improves performance between 1.3 — 2.5$\times$, reduces on-chip SRAM between 5 — 40 $\times$, and saves 8 $\times$ on-chip energy.

1.5 Outcomes

During this research, I extended some existing tools such as Enzyme [24] and gem5-SALAM [44] and created new open-source tools that can be used in other research projects. Here is a brief introduction of these tools:

- **Modified Enzyme:** This enhanced version of Enzyme introduces the capability to arrange the tape values in the more efficient array-of-structs format. By adopting this approach, it significantly reduces required bandwidth, minimizes cache misses, and lowers the probability of page faults.

- Extending gem5-SALAM's supported instruction set. More information is given in the appendix.
- **Modified gem5-SALAM:** I enhanced gem5-SALAM to serve as a back-end for our compiler. Throughout the modification process, I identified, reported, and resolved various issues within this framework, including unhandled read-after-write hazards. Furthermore, I documented the steps involved in implementing my modifications to gem5-SALAM.
- **Register allocation tool:** I built multiple tools to statically or dynamically analyze the properties of different programs. One of these tools extracts the dynamic dataflow graph (DDG) and feeds it to a register allocator. This tool reports liveness analysis, minimum required registers, number of register file spills, etc.

1.6 Publications

This dissertation includes our recently submitted paper at the peer-reviewed conference ASPLOS, where I collaborated with my supervisor, Dr. Shriraman. Additionally, I had the privilege of collaborating with my esteemed colleagues, Ali Sedaghati and Reza Hojabr, resulting in the acceptance of another paper at the prestigious ISCA conference. In this project, I took on multiple responsibilities, including writing a compiler for *X-Cache*, implementing benchmarks, and extracting traces for evaluation. I also participated in the hardware implementation. Here is the list of my publications:

- ASPLOS 2024 (under review) — Tapeflow: Streaming Gradient Tapes in Automatic Differentiation. Milad Hakimi, Arrvindh Shriraman.
- ISCA 2022 — X-Cache: A Modular Architecture for Domain-Specific Caches[48]. Ali Sedaghati, Milad Hakimi, Reza Hojabr, Arrvindh Shriraman

1.7 Dissertation Outline

The dissertation is organized as follows: Chapter 2 describes the background and related works. Chapter 2.3 discuss the challenges of implicit orchestration and the motivations behind the work. Chapter 3 describes Tapeflow, a compiler developed based on our observations in the previous chapter. The evaluation of the work and describing the tool flow will be presented in Chapter 4. Finally, Chapter 5 will outline the conclusion, limitations, and future works.

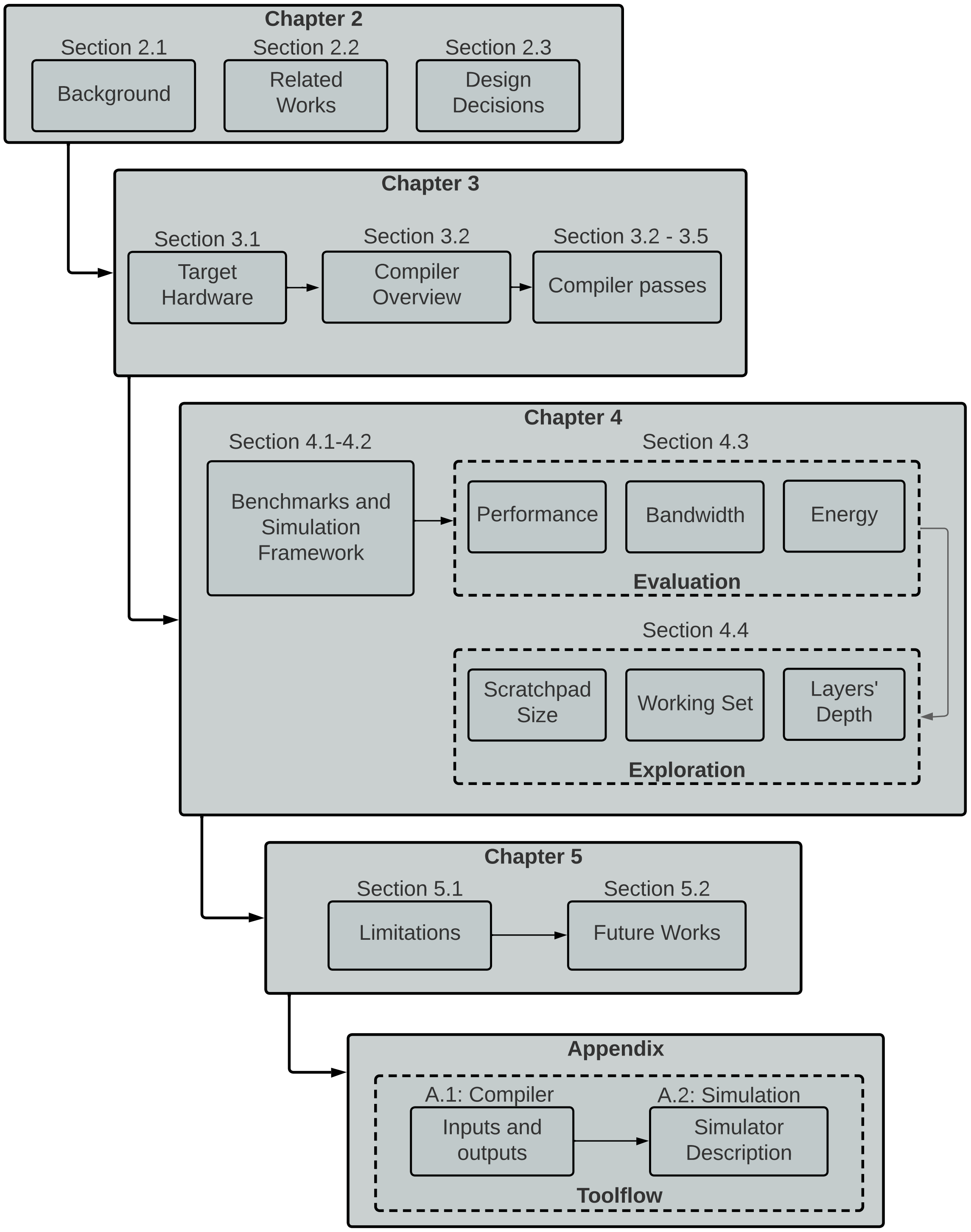


Figure 1.5: Dissertation Outline

Chapter 2

Background and Related Works

In this chapter, we first cover the basic principles of AD, different AD implementations, and their trade-offs. We then provide an application of AD as an example to show where AD is used. In the second part of this chapter, related works, we elaborate on the prior works that have been done in AD compilers, DNN accelerators, and decoupled access-execute paradigms and talk about the gaps in the field. Finally, we summarize our contributions compared to the state-of-the-arts AD frameworks in Table 2.1.

2.1 Background

Automatic Differentiation (AD)

Automatic Differentiation is a technique to evaluate the derivative of a function in a computer program. It exploits the fact that functions, at the lower levels, are composed of simple arithmetic operations with known derivatives. It uses the *chain rule* over these operations to compute the derivative of the function. AD takes the program `P` that implements a mathematical function and produces a new program that computes the gradient of `P`. AD tools do so by examining the instructions of the program (such as `add`, `mult`, `div`), and generating the partial derivative for each instruction. By applying the chain rule and accumulating the partial derivatives, they can produce the gradient of the original function. The chain rule can be applied in any order, from inputs to outputs or from outputs to inputs. The order affects the efficiency, ease of implementation, and memory usage. There are two popular strategies regarding how to apply the chain rule.

Forward (tangent) Mode

Applies the derivatives of instructions from inputs towards outputs. The derivatives are evaluated in the order of instructions' evaluation. Consider the function `z = f(x, y)`. The derivative of `z` with respect to the inputs would be as follows.
$\dot{z} = \frac{\partial f}{\partial x}\dot{x} + \frac{\partial f}{\partial y}\dot{y}$
To find the derivative for all the outputs of the program, z_1, z_2, ..., z_n, with respect to `x`,

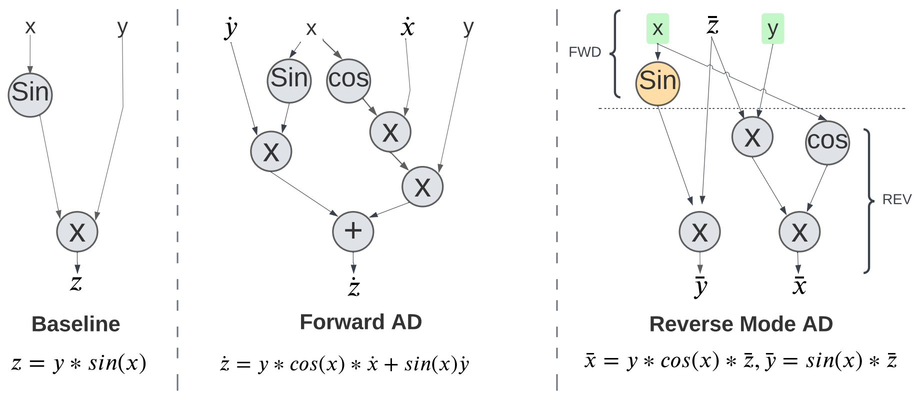


Figure 2.1: Different modes of Automatic Differentiation

we set $\dot{x}$ to 1 and $\dot{y}$ to 0. However, to find the derivative with respect to all the inputs (here `x` and `y`), we have to evaluate the function once for each input equals to 1 while others set to 0. Hence, we need a forward mode evaluation for every input, which is inefficient when the program has many inputs. An example of forward mode AD is shown in Figure 2.1. The derivatives and original computations takes place at the same time.

Reverse (adjoint) Mode

The reverse mode combines the derivative of instructions in a reverse pass that computes the derivative (adjoint) of the instructions in the reverse order of the original program's execution. It, then, propagates the derivatives from an instruction's outputs to its inputs. For the mentioned instruction `z = f(x, y)`, the derivative with respect to inputs can be computed as follows.

$\bar{x} = \frac{\partial f}{\partial x}\bar{z}$, $\bar{y} = \frac{\partial f}{\partial y}\bar{z}$

The derivative of each output with respect to the inputs can be evaluated by setting the $\bar{z_i} = 1$ before evaluating the partial derivatives and read the final value of $\bar{x}$ or $\bar{y}$. By this approach, we can evaluate the gradient of the output with respect to all the inputs in a single evaluation of the function. The reverse mode executes in two passes. First, the forward pass executes, evaluates the original function, and caches the intermediate variables. Then, the reverse pass executes, which uses the intermediate variables to calculate the partial derivatives and propagate them from the outputs towards the inputs. Evaluating the derivative with respect to multiple outputs requires a separate evaluation pass for each output. In practice, programs with many inputs, but a few outputs dominate the machine learning and scientific use cases, which makes them more compatible with reverse mode AD. Figure2.1 contains an example of reverse mode AD with annotated FWD and REV passes. First, `sin(x)` is evaluated, which belongs to the original function. The result is then used in the REV pass to calculate $\bar{y}$.

Implementation

Goal: Acknowledge the different differentiation methods at different abstraction levels and their trade-offs.
There are two primary approaches to implementing AD.

Operator Overloading computes the derivatives by providing a differentiable version of an existing language. For example, Adept [1], ADOL-C [52], and JAX [10] are C++ and python libraries that provide differentiable types. To take advantage of these libraries, we have to rewrite the programs such that they use a closed set of types or operators supported by these languages. This limits us to only use a subset of the language and prevents us from using different libraries. These approaches are dynamic, which means they record a history of the instructions and values during runtime and compute the gradient based on that history. Figure 2.2 shows an example of using the Adept library in C++. The stack records the instructions and intermediate values during runtime and is used to compute the gradients in the reverse pass.

Source Code Transformation analyzes the source code of the program and generates a new function that computes the derivative of the original function. Unlike the previous approach, this method does not need to track the operators executed by the program during runtime, because it generates a new function during compile time that executes to compute the gradients. In this thesis, we use Enzyme [24] that performs source-code transformation on the LLVM IR level. Enzyme is independent of the source language and can work with any language that can be lowered to the LLVM IR. Moreover, by using LLVM IR, it can run on different machines. SCT might suffer from long compilation time compared to operator overloading.

Both operator overloading and SCT approaches suffer from an inherent problem to reverse-mode AD, which is memory management. This issue arises because of the intermediate values that have to be passed from the forward pass (FWD) to the reverse pass (REV). In AD algorithm, they are stored in a stack-like data structure called tape. Operator overloading excessively relies on the stack to store values and instructions, and hence, it has a large memory footprint. SCT eliminates the need for the instruction tape by producing the source code, but it still needs the tape to store the values. Tape's size becomes problematic especially in Deep Neural Networks (DNNs) where the number of intermediate values becomes too large to fit on the GPU memory during training.

AD Compilers

Automatic differentiation (AD) can be implemented at various levels of abstraction, including the language level [33], library level [52], and intermediate representation (IR) level [18, 24]. Implementing AD at higher levels of abstraction provides benefits such as

```
// Rewrite to accept either
// double or adouble
template<typename T>
T relu3 (T val) {
  if (x > 0)
    return pow(x, 3)
  else
    return 0;
}

int main() {
  adept: :Stack stack;
  adept: :adouble inp = 3.14;
  // Store all instructions into stack
  adept:: adouble out(relu3(inp));
  out.set_gradient (1.00);

 // Interpret all stack instructions
 double res = inp.get_gradient (3.14);
}
```

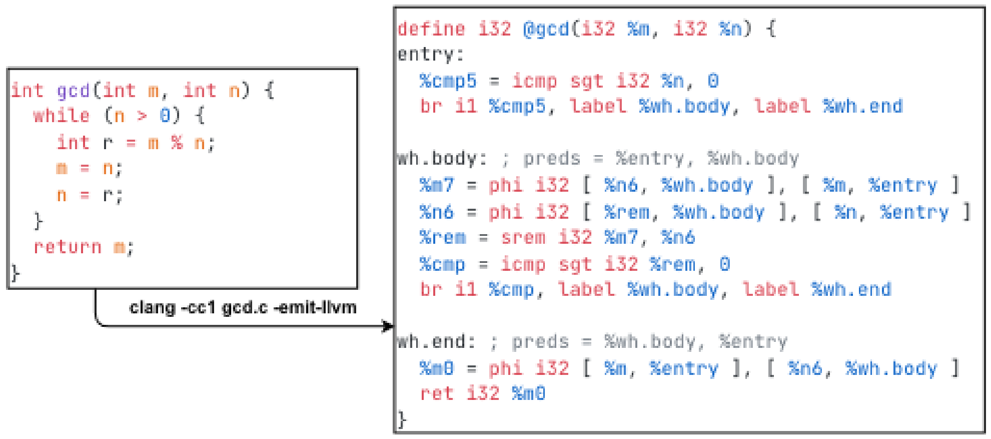
Figure 2.2: An example of operator overloading in Adpet.

ease of use and coarse-grain optimizations, such as mapping known kernels into proper hardware for execution in XLA [46]. Conversely, implementing AD at lower levels of abstraction can result in faster and more efficient functions due to compiler optimizations and lower-level knowledge such as dead code elimination, loop unrolling, and type resolution. Additionally, implementing AD at lower levels of abstraction increases the reusability of the framework because it can be adopted by different languages that can be lowered into that intermediate representation.

Enzyme [24] is an Automatic Differentiation (AD) framework that leverages LLVM for efficient gradient computation. LLVM is a widely adopted compiler infrastructure that offers a comprehensive suite of modular and reusable components, supporting various programming languages and target architectures. As part of the compilation process, LLVM-IR serves as an intermediate representation, providing a low-level and platform-independent representation of programs.

Figure 2.3 shows how a C program is represented in LLVM-IR. LLVM-IR adopts a typed, static single assignment (SSA) form, ensuring that each value in the program is assigned exactly once. This property simplifies program analysis and transformation. LLVM-IR expresses control flow through a control flow graph of basic blocks (nodes) and branches (edges). Each basic block, represents a dataflow graph between operations and values. The IR support for both high-level and low-level operations makes it suitable for a wide array of programming languages and optimization requirements.

One of the notable advantages of LLVM-IR is its language portability. Developers can write optimizations and analyses that can be applied to programs written in different lan-



```c
int gcd(int m, int n) {
  while (n > 0) {
    int r = m % n;
    m = n;
    n = r;
  }
  return m;
}
```

```llvm
define i32 @gcd(i32 %m, i32 %n) {
entry:
  %cmp5 = icmp sgt i32 %n, 0
  br i1 %cmp5, label %wh.body, label %wh.end

wh.body: ; preds = %entry, %wh.body
  %m7 = phi i32 [ %n6, %wh.body ], [ %m, %entry ]
  %n6 = phi i32 [ %rem, %wh.body ], [ %n, %entry ]
  %rem = srem i32 %m7, %n6
  %cmp = icmp sgt i32 %rem, 0
  br i1 %cmp, label %wh.body, label %wh.end

wh.end: ; preds = %wh.body, %entry
  %m0 = phi i32 [ %m, %entry ], [ %n6, %wh.body ]
  ret i32 %m0
}
```

Figure 2.3: `gcd` function lowered to LLVM-IR using clang.

guages, abstracting away the intricacies of each language's syntax and semantics. Enzyme, utilizing LLVM-IR, generates the gradient function after compiler optimizations, resulting in a more compact function with fewer memory accesses. These optimizations not only enhance Enzyme's ability to employ efficient heuristics for minimizing tape usage, but also reduce overall program memory consumption.

We specifically chose Enzyme due to its proficiency in targeting high-performance applications, its support for multiple languages that can be lowered to LLVM-IR, and its efficiency compared to other state-of-the-art general-purpose AD tools. By harnessing LLVM's capabilities and Enzyme's optimization strategies, we can achieve superior performance in gradient computation while effectively managing memory usage.

2.2 Related Works

Many researches have targeted optimizing memory accesses by reducing them or removing them from the critical path in DNN training, general-purpose AD, and other general purpose programs. We address three areas related to memory access optimization. The first section addresses research related to general-purpose AD compilers and their optimizations such as checkpointing, operation pruning, etc. The second section focuses on DNN training frameworks and accelerators and how they leverage domain knowledge to optimize the memory accesses. The third section discusses the decoupled access-execute paradigm and how it removes the memory accesses from the critical path of execution through prefetching. Finally, we compare Tapeflow and other AD tools such as DSLs, AD libraries, and DNN accelerators to mention the contributions of Tapeflow these frameworks.

2.2.1 AD Compilers

Source code transformation (SCT) and operator overloading are two approaches for implementing automatic differentiation (AD) in compilers. While operator overloading modifies the behavior of arithmetic operators such as addition and multiplication, SCT involves modifying the source code of the program directly. In terms of memory optimization, SCT has the advantage of enabling smaller tapes, leading to significant memory savings. This is because SCT does not need to store the sequence of operations during runtime like operator overloading does. Instead, the source code is transformed to incorporate the derivative computation directly into the program. AD compilers that use SCT, such as Enzyme [24], Zygote [18], and Halide [33], focus on further reducing the memory footprint caused by tape accesses. They achieve this through a technique called "checkpointing", which involves storing only a subset of the intermediate values during the computation, rather than storing all intermediate values.

Additionally, some frameworks [24, 18] rely on compiler optimizations such as dead code elimination, loop-invariant-code-motion, and others to aggressively prune the intermediate values. This results in fewer values being stored on the tape, and thus further reduces the memory required for tape storage.

Dr. Jit [20] is a specialized compiler for physically-based rendering that dynamically compiles differential simulations. During the kernel assembly step, Dr. Jit eliminates redundancies on a global scale based on the dynamic dataflow and control flow graph and generates LLVM-IR code. It also removes unused inputs and outputs, triggering simplifications throughout the program.

While state-of-the-art AD compilers have advantages in terms of memory optimization, they have limitations in their approach to tape orchestration. These compilers assume that the underlying hardware uses a cache, so they rely on implicit tape orchestration and do not use compile-time knowledge of the tape, such as alias information and reuse distance, to explicitly orchestrate the tape. This can lead to suboptimal memory usage and performance. As a result, there is a need for more advanced AD compiler techniques that can explicitly orchestrate the tape, taking into account the compile-time knowledge of the program's memory usage.

2.2.2 DNN Training Frameworks and Accelerators

Several closely related works to Tapeflow are pipelined DNNs [37, 13] and distributed DNNs [27]. While these works operate with DNNs that utilize a user/model-provided layer graph containing blackbox DNN operators like convolution and LSTM, Tapeflow distinguishes itself by defining layers based on static hardware constraints, specifically the size of on-chip scratchpads, resulting in the formation of arbitrary sub-dataflows. Unlike previous approaches that primarily focused on blackbox operators with predefined data characteristics, such as GEMM, and treated training gradients as separate tensors, Tapeflow introduces the capability for gradients to reference specific memory locations.

Research in the field of deep neural networks (DNNs) has extensively investigated various training optimizations [55, 54, 15], which include specialized implementations for specific models or compositions of DNN layers. Output re-computation techniques [14, 15] represent specialized versions of the widely used "checkpointing" technique employed by compilers. These techniques discard certain outputs when the available memory is insufficient, and then re-compute them as needed. However, it is important to note that this approach typically results in performance degradation compared to alternative approaches that avoid such recomputation [43].

Another line of related work focuses on partitioning tensors in DNNs across heterogeneous locations [37, 42, 43], such as between DRAM and FPGA [56] or across multiple GPUs [40]. These works effectively save energy. In the context of Tapeflow, we work with a low-level compiler IR because generalizing differentiation requires us to operate at the low-level operand level. Consequently, data is expressed in scalars and pointers rather than tensors, making applying prior techniques that rely on data shape impractical.

DNN accelerators such as Pipedream [28], Gist [19], and vDNN [43] optimize the tape by explicitly orchestrating the movement of feature maps through the memory hierarchy to enhance performance and enable training of larger models. These accelerators employ on-chip memory management techniques and bandwidth optimization methods like compression, sparsity, and quantization. By capitalizing on the spatial and temporal locality of DNN computations, these accelerators efficiently utilize on-chip memory while minimizing data movement between on-chip and off-chip memories. Overall, the optimizations employed in DNN frameworks and accelerators aim to reduce the memory footprint of the tape and enable training of larger models with limited memory resources.

However, it's important to note that these techniques are specific to DNNs and rely on knowledge of coarse-grained operations, known program graphs, and specific types. They

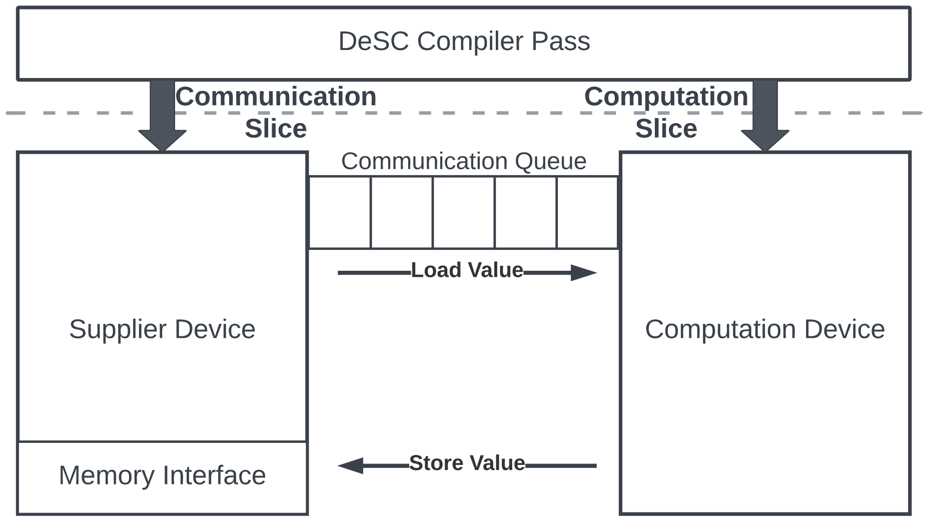


Figure 2.4: DeSC overview

may not be easily applicable to other AD compilers that operate on more arbitrary programs and lack the same knowledge of program structure.

2.2.3 Decoupled Accesses-Execute

Many researchers have explored the decoupled access-execute paradigm as a solution to address the issue of memory access latency and its impact on program performance [16, 36, 11, 30, 38]. In traditional computer architectures, instructions are executed sequentially, causing each instruction to wait for the completion of the preceding instruction before it can be executed. Consequently, if an instruction requires data that is not present in the processor's cache or register file, it must wait for the data to be fetched from memory, resulting in significant delays of hundreds of cycles or more.

To mitigate this problem, the decoupled access-execute paradigm separates the memory access phase from the execution phase of the program. Initially, instructions are loaded into a buffer, and then data is fetched from memory as a separate step. This approach enables the processor to begin executing instructions without waiting for memory access to complete, leading to substantial performance improvements.

Figure 2.4 shows the overview of a decoupled access-execute architecture proposed by DeSC [16] reduces the memory bottlenecks automatically by leveraging the compiler support. The approach is to provide performance and energy efficiency of scratchpad memory while offering programmability similar to caches. DeSC's compiler splits the program into a communication slice and a computation slice. The communication slice performs the memory loads, stores, and address generations. This part fetches all the data required by the compute slice and stores them in the communication queue. This queue will be stored in content-accessible memory in the computation device for fast memory access. The computation slice executes the operations and sends the results back to the communication slice to store them in the memory. The communication slice is responsible for address generation for both loads and stores.

Stream-dataflow [30] aims to accelerate computation by exploiting spatial parallelism and data locality. It follows a decoupled access-execute paradigm, where data movement and computation are separated and executed simultaneously. The stream component provides support for an ordered set of stream commands, which specify long and concurrent patterns of memory access. The dataflow component exposes instructions and their dependencies through a dataflow graph (DFG), which is a spatial specification of computation. The hardware implementation consists of a CGRA, a programmable scratchpad, stream engines, and a control core. Execution occurs in phases specified by the programmer, starting with a stream command and ending with a barrier command that synchronizes the control core.

The existing methods in this paradigm have certain limitations in terms of automatic optimization and ease of use. Automatic methods like DeSC [16] may be conservative and fail to fully exploit the performance potential due to their lack of knowledge about program execution and control flow. Moreover, DeSC introduces an additional pipeline for data fetching, which incurs additional energy overhead. The compiler implements a FIFO structure between decoupled memory and computation, limiting the access order. Conversely, low-level interfaces such as those in Stream-Dataflow [30] and Buffets [36], which require users to program memory accesses, can be tedious and error-prone. Additionally, they can not be used by the users of AD compilers because the tape accesses are introduced after the user writes the FWD function. Thus, there is a need for high-performance automatic memory orchestration methods that can bridge this gap.

2.2.4 Comparison

Generalized compiler-driven Automatic Differentiation (AD) has gained recent attention, particularly with the introduction of Enzyme. However, there is a rich history of prior work in gradient-based computation. Some of the most notable contributions include Deep Neural Network (DNN) training techniques, such as vDNN[43] and Gist[19]; Domain-Specific Languages (DSLs) designed for machine learning applications, like TensorFlow[3] and DiffTaichi[17]; and differentiable libraries, for example, C++ Adept[1]. Table 2.1 provides a comparison of these approaches, highlighting their respective limitations and features. The table is divided into two sections: the target section, which outlines the properties crucial for enabling gradient calculation in imperative code, and the design section, which enumerates desirable characteristics. This section aims to offer an overview of the existing work in this domain and to elucidate the distinctions between these methods and Tapeflow.

		DNN Training		Other DSLs	Diff. Lib.	AD Compilers	
		vDNN [43], Gist [19]	TensorFlow [3], PyTorch [34]	DiffTaichi [17], Halide [33]	Adept [1]	Enzyme[24]	Tapeflow
Target	Domain	DNNs and Machine learning		Physics Sim., Img	Dataflow	General purpose	
	Operators	Dense, Conv	Dense, Conv	Arbitrary	Lib. Specific	Arbitrary	
	Access Flexibility	Low(Fixed kernels)	Low(Fixed kernels)	High	Only-FIFO	High	
	Tape Alloc.	Compiler	Varies	User	Compiler	Compiler	
	Access Granularity	Tensor	Tensor	Element-wise	Element-wise	Element-wise	
Design	Alloc. Granularity	Tensor	Tensor	Array	Element	Array	Regions
	Tape Orch.	Varies	Implicit	Implicit	Implicit	Implicit	Explicit
	Tape Layout	Tensors(SoA). High Bandwidth			FIFO only	Arrays(SoA)	Struct(AoS)
	Mem. Hierarchy	Flexible		Cache. conflict/capacity misses. Energy			Scratchpad
	Preloading	Yes	No double buffering or prefetching.				Yes

Table 2.1: Tapeflow vs SOTA frameworks. Red indicates undesirable features for AD (may be acceptable for specific domains)

DNN Training Frameworks and Accelerators

The DNN training frameworks have specialized the reverse mode Automatic Differentiation as the back-propagation algorithm. The tape problems in AD such as being large, long-lived, and having mixed reuse also appear in back-propagation. Using operations with higher levels of abstraction over both data types (e.g., tensors) and operations (e.g., convolution) makes the tape accesses more regular and easier to optimize in DNN frameworks. This is because the tape accesses within the layers are completely separated from the tape accesses of the other layers. Hence, it is easier to detect and manage the values with similar lifetimes. The DNN training frameworks such as TensorFlow [3] or Pytorch [34] exploit this fact to optimize the memory accesses within the kernels and across the kernels.

DNN training accelerators are specialized hardware designed to accelerate the computation involved in training large neural networks. They are optimized for specific types of computation, such as matrix multiplication, convolution, etc. which are the building blocks of DNNs. DNN frameworks typically have one tape per layer output, and the tape structure mirrors the underlying model graph. In addition, DNN training accelerators are often designed to work with fixed-size data structures, such as tensors of fixed shape, which reduces the complexity of tape allocation for the compiler but is not suitable for arbitrary functions with varying input sizes.

These approaches can not be easily adopted by general-purpose AD compilers because they often require the ability to handle variable-sized inputs and use arbitrary kernels and data types, which may require different memory allocation and computational strategies than those used in DNN training frameworks and accelerators. In other words, not all applications can be composed of the building blocks available in the DNN frameworks. Especially, high-performance applications require fine-tuning the kernels and the data structures for better performance and energy, which is not feasible in these frameworks.

Domain Specific Languages (DSLs)

DSLs extend support to more generalized operators and iterative computations, but they lack support for fine-grained memory accesses, data-dependent control flow, and imperfect loops due to their level of abstraction. Further, the user has to manually allocate and optimize the tape for each operator, which can affect the correctness and the performance. The users must be careful not to overwrite buffers required for REV, as doing so can easily lead to corrupted results. On one side, the responsibility of the tape management is on the user, and on the other side, the user can not optimize the tape because it requires knowledge of the gradient function which will be produced later by the compiler. By doing so, the compiler would not perform some common tape optimizations done in AD such as checkpointing or recomputation because the tape was already allocated before the differentiation take place. Users may choose to allocate tape even in cases where recomputation is faster, which can negatively affect performance. Moreover, DSLs do not support differentiating external libraries or user-defined types. To differentiate external functions, the function has to be explicitly rewritten to a format supported by these frameworks. Furthermore, the DSLs do not explicitly manage the tape and rely on the underlying hardware (cache).

Libraries

Library-based approaches rely on types and operations introduced by the library to manage the tape, but the programmer must tediously rewrite the model using the library's API, and the required operators need to be built into the library. Hence, the user might not be able to differentiate external functions and has to rewrite using the library's supported operations and types. Differential libraries manage the tape as a stack of operands, limiting parallelism and bandwidth utilization. Moreover, the differentiable approaches that rely on the execution history and use a stack to keep it have a high runtime overhead for storing all the operations and their relations during the execution.

2.3 Observations and Design Decisions

In this section, we explore the importance of tape management and orchestration at lower levels of abstraction. We provide a quantitative analysis to demonstrate the inefficiencies of relying on the implicit orchestration of tape in general-purpose AD compilers. Based on our analysis, we highlight the need for automatic explicit tape orchestration in compilers, supported by both quantitative and qualitative justifications. This section answers the following questions:

- What percentage of the memory accesses are to the tape, and what are the implications?
- What is the difference between the lifetime of tape and non-tape values?
- In what order the tape values are consumed, and what are the implications for that?
- How does the tape impact the bandwidth?
- What are the characteristics of AD that make it possible to explicitly orchestrate the tape?

The plots of this section are generated in a multistep process. Initially, we instrumented the code for each benchmark using the LLVM compiler and marked various memory accesses to distinguish tape and non-tape accesses. Next, we simulated the execution of the dataflow, represented by the LLVM-IR, using gem5-salam [44], which is a dataflow accelerator simulator. Throughout the execution, we monitored the tagged instructions to track their distribution and production/consumption cycle. Moreover, gem5-salam allowed us to control crucial system parameters, such as the number of functional units, cache size, and registers and see their effect on the runtime and bandwidth. The specific setting for our system is explained in Chapter 4 in more details.

2.3.1 Tape management in AD compilers

Summary: we need low-level (IR level) tape management, and the AD compiler is responsible for managing and orchestrating the tape.

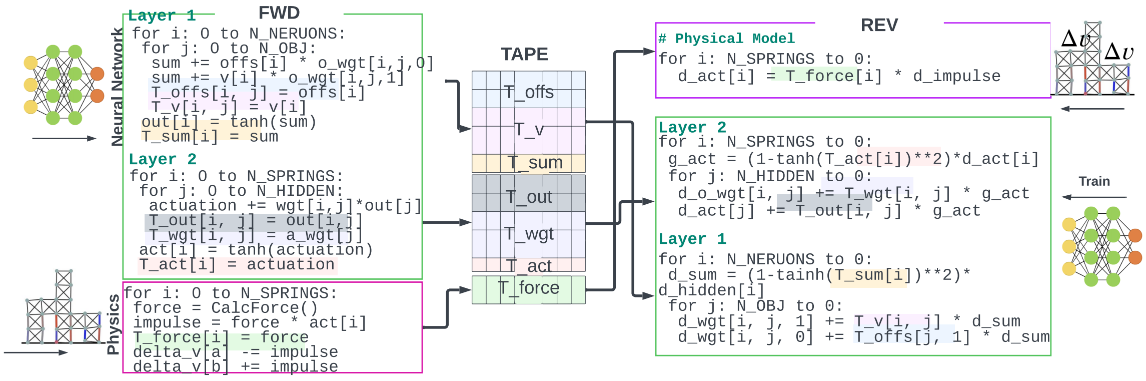
One of the challenges in implementing AD is managing the tape, a data structure that records the sequence of operations performed during the forward pass (FWD) of the computation. The tape is used to compute the gradients during the reverse pass (REV), and it can become a bottleneck in terms of memory usage and computational efficiency. The size of the tape can grow rapidly for complex computations, and managing the tape can become a challenging problem in AD.

Observation 1: *The inverse solvers may employ imperfect loops, if-else conditions, and indirect array indexing. Hence, we need low level tape management needs to support general* *program graphs and not be limited to blackboxes.*

One important observation in AD is that tape management needs to support general program graphs and not be limited to blackboxes. While high-level AD frameworks such as TensorFlow and library-level AD tools provide convenient interfaces for implementing AD, they may not be able to handle the full range of program graphs encountered in practice. In contrast, IR-level AD provides a more flexible and powerful approach to tape management, enabling the efficient computation of gradients for a wide range of programs, including those with imperfect loops, if-else statements, and array indexing. By prioritizing IR-level AD, we can ensure that the tape management techniques are capable of handling the full range of program graphs encountered in practice, enabling the development of more efficient and scalable AD techniques. This will enable the application of AD to a wider range of problems, including those in scientific computing, robotics, and physics simulations, and enable the development of more complex and expressive models.

Observation 2: *The compiler is responsible for tape allocation, management, and orchestration, which are essential for efficient and scalable AD. Poor decisions in these areas can significantly impact the performance of AD, leading to inefficient memory usage, high bandwidth, and cache conflicts.*

Figure 2.5 shows Enzyme's [24], the state-of-the-art AD compiler, gradient function for a simple particle system [17]. The system has three arrays for each particle: position (`x`), velocity (`v`), and `offset`. We color-coded the tape accesses in FWD and REV, and their layout in DRAM. Enzyme allocates a separate tape for each of the arrays which creates a struct-of-arrays layout in DRAM which results in inefficient memory access and high bandwidth. During the gradient calculation in the REV, elements from multiple tape arrays are accessed simultaneously. As each tape entry is from a different cache block, multiple DRAM accesses are necessary, resulting in inefficient bandwidth utilization. Enzyme does not make orchestration decisions regarding tape movement onto the chip or reuse from



```
Layer 1
for i: 0 to N_NERUONS:
  for j: 0 to N_OBJ:
    sum += offs[i] * o_wgt[i,j,0]
    sum += v[i] * o_wgt[i,j,1]
    T_offs[i, j] = offs[i]
    T_v[i, j] = v[i]
  out[i] = tanh(sum)
  T_sum[i] = sum
Layer 2
for i: 0 to N_SPRINGS:
  for j: 0 to N_HIDDEN:
    actuation += wgt[i,j]*out[j]
    T_out[i, j] = out[i,j]
    T_wgt[i, j] = a_wgt[j]
  act[i] = tanh(actuation)
  T_act[i] = actuation
```

```
for i: 0 to N_SPRINGS:
  force = CalcForce()
  impulse = force * act[i]
  T_force[i] = force
  delta_v[a] -= impulse
  delta_v[b] += impulse
```

```
# Physical Model
for i: N_SPRINGS to 0:
  d_act[i] = T_force[i] * d_impulse
```

```
Layer 2
for i: N_SPRINGS to 0:
 g_act = (1-tanh(T_act[i])**2)*d_act[i]
 for j: N_HIDDEN to 0:
  d_o_wgt[i, j] += T_wgt[i, j] * g_act
  d_act[j] += T_out[i, j] * g_act

Layer 1
for i: N_NERUONS to 0:
 d_sum = (1-tainh(T_sum[i])**2)*
d_hidden[i]
 for j: N_OBJ to 0:
  d_wgt[i, j, 1] += T_v[i, j] * d_sum
  d_wgt[i, j, 0] += T_offs[j, 1] * d_sum
```

Figure 2.5: Illustration of Mass Spring example. The tape state is color-coded to indicate producer-consumer dependency.

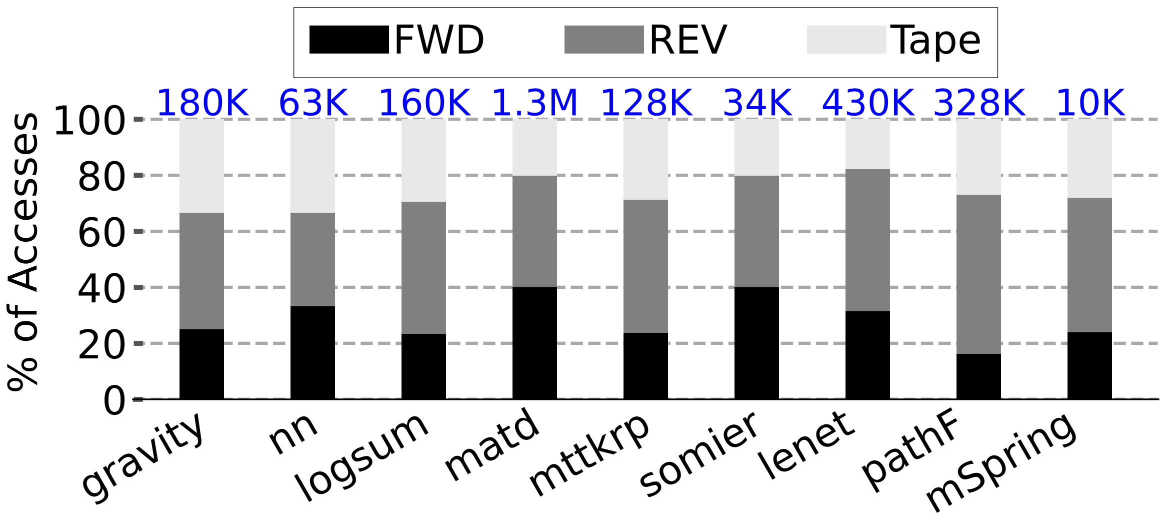


Figure 2.6: The distribution of FWD, REV, and TAPE edges in the dataflow. Numbers on top indicate working set.

DRAM or schedule operand execution, and relies on the cache to handle tape orchestration. This increases the cache conflicts and overflows because the long-term reuse tape accesses conflict with short-term reuse non-tape accesses.

Observation 3: *The biggest challenge in generating fast REV code is understanding memory operations within the dataflow.*

Enzyme mixes in the tape-accesses into the overall dataflow and makes it challenging for the compiler to analyze and optimize. The tape is accessed using conventional load/store operations (like non-tape accesses) which confounds the alias analysis. Enzyme assumes the tape is in the DRAM, and the underlying hardware includes a cache hierarchy. We show, in the next section, why caches cannot handle tape accesses. Furthermore, the operators in the REV's dataflow have intermingled dependencies with mixed temporal reuse, one comes from FWD (tape access with non-temporal reuse) and the other from the REV (non-tape access with temporal reuse). For example, in Figure 2.5:REV, the last statement in REV uses `T_offs[i][0]` from FWD and `d_sum` from REV. This means that the tape access has to be buffered until the REV operand is created.

2.3.2 Tape Characterization

The goal of this section is to quantitatively and analytically describe the tape and discuss why is it hard to put it inside the cache and why should we consider a compiler-managed scratchpad approach.

Limitations of implicit orchestration

Observation 1.1: *Tape accesses are 20 — 40 % of the total memory accesses. Along with the REV, it adds 4–5× memory accesses to the baseline FWD. See Figure 2.6.*
Implication: *The tape is large and will overflow both explicit (scratchpad) and implicit (cache) memory hierarchies.*

As the tape is large and the number of tape values is proportional to the operations in the program, it is highly probable that the tape will overflow the cache. This can lead to cache thrashing, where the cache is constantly evicting and reloading data from the tape, resulting in slower performance. Moreover, the overflow of the cache by the tape can cause conflicts with non-tape accesses, as the cache may need to evict non-tape data to make room for tape data. This can result in poor performance and can significantly impact the overall efficiency of the system.
Observation 1.2: *Tape values live up to 100 × longer than other registers. See Figure 2.7.*
Implication: *The tape does not manifest temporal locality, which makes the cache an inefficient hardware for exploring tape's reuse.*

As depicted in Figure 2.7, the average lifetime of the tape could be up to 100 × more than non-tape values. The lifetime and reuse distance of tape values are high due to two reasons. Firstly, tape values need to wait for the FWD to finish before they get consumed by the REV operators. This is illustrated in Figure 2.5, which shows the consumption of the color-coded tape values using arrows. All tape values are consumed in the REV after the FWD finishes. Secondly, the consumption order of the tape values is the reverse of their production order. For instance, in Figure 2.5, the tape entry `T_offs[0][0]` is produced at the beginning (iteration 0 of phase 1) but consumed by the last statement of the last iteration of the REV. These characteristics of tape values make them unsuitable for storage in the cache, which is designed to store data with high temporal locality.
Observation 1.3: *Tape values have mixed reuse pattern by having both short and long reuse distances. See Figure 2.8.*
Implication: *Changing the cache's replacement policy does not make it suitable to store the tape.*

Based on the mixed reuse pattern of tape values seen in Figure 2.8, it is evident that the cache is not suitable for storing the tape. Changing the cache replacement policy in favor of long-lived values would negatively impact the performance of non-tape accesses, which have much shorter producer-consumer distance (Figure 2.7). Moreover, Figure 2.8 shows that lifetime distribution of the tape values varies significantly across different applications. In some applications like `nn`, the values with higher lifetime are dominant, while in some others, like `logsum`, the short-lived values are dominant. Hence, devising a replacement policy in favor of the tape is non-trivial, as it depends on the application.

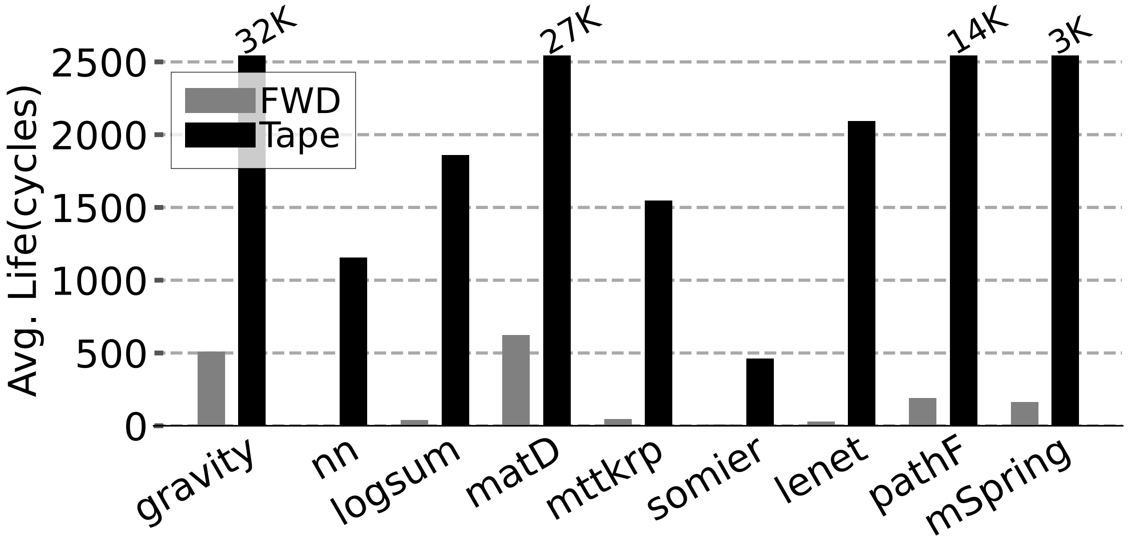


Figure 2.7: Average lifetime. Tape edges vs. FWD.

Observation 1.4: *The use of tape in a program can significantly increase the demand for resources such as bandwidth and cache blocks.*
Implication: *To minimize the impact on system performance, the tape's layout in memory should be optimized to reduce bandwidth and cache usage.*

As shown in Figure 2.6, the memory accesses of the gradient function are dominated by the REV and tape accesses. The REV memory accesses involve loads and stores to calculate the partial derivatives, and the REV dataflow and control flow are dictated by how the FWD is written and do not change. Therefore, it is essential to arrange the tape's layout in memory to match its access pattern in FWD and REV to increase the bandwidth and cache utilization.

Motivations for explicit orchestration

Observation 2.1: *The compiler has perfect alias information about the tape and knows the producer-consumer operations between the FWD and REV.*
Implication: It is the AD framework's responsibility to orchestrate and manage the Tape.

Given the original function, the AD framework generates the gradient function by placing the original function in the FWD, creating the corresponding REV operations for each FWD operation, allocating the tape, and connecting the producers in FWD to the consumers in REV through the tape. Consequently, the compiler has the perfect alias information about the tape and knows the mapping of operations between FWD and REV. If the AD tool does not exploit this information to optimize the tape, the high-level knowledge will be lost when the code is lowered through the compilation, making it infeasible for the lower-level compiler to optimize the tape.

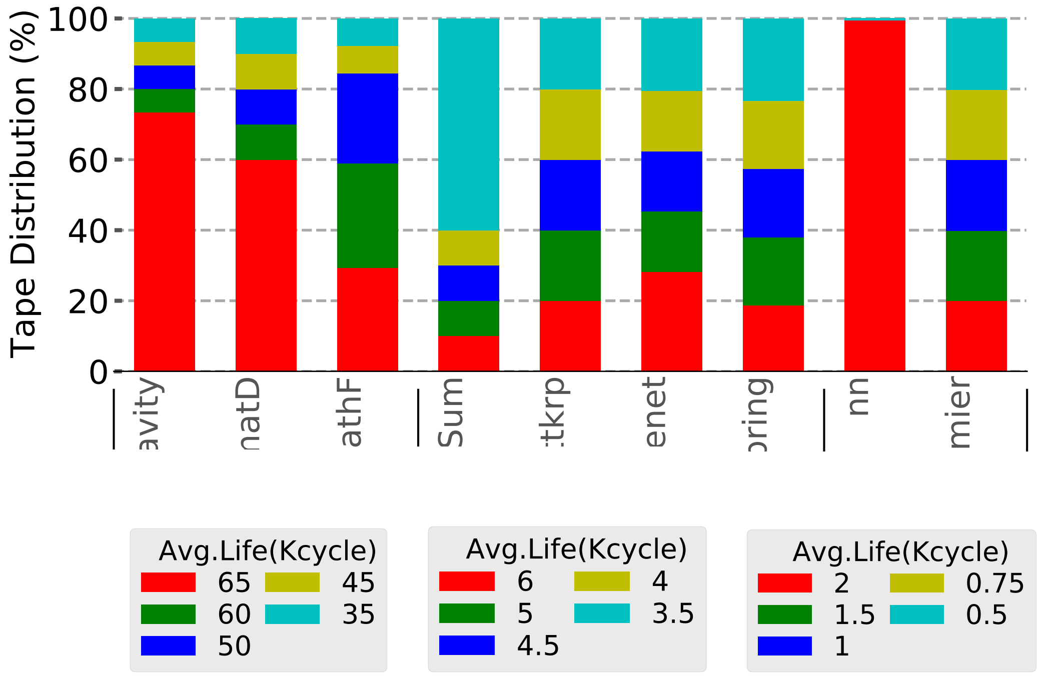


Figure 2.8: Tape lifetime distribution. Benchmark cluster: Similar lifetime (in Kcycles). Y-axis: Distribution of edges.

Observation 2.2: *Tape entries can be partitioned into groups with similar lifetime (Figure 2.8).*

Implication: Tape can be implemented using streams.

Observation 2.3: *Tape values produced together in the FWD are consumed together in the REV.*

Implication: Tape should be organized as array-of-structs. Elements consumed together should be packed together.

We divided the tape values into 5-quantiles based on their lifetimes and represented them in Figure 2.8. The height of each stack represents the number of tape entries within a particular quantile, while their color denotes their lifetime (light blue: shortest, red: longest). The tape entries exhibiting similar lifetimes are produced and consumed by adjacent operations in FWD and REV, allowing us to organize the tape for streaming, and preload it from the DRAM. This observation is further supported by looking at the AD algorithm employed by Enzyme or similar AD tools when constructing the gradient function. In this algorithm, the basic blocks within the REV contain the gradients of the operations present in the FWD basic blocks. Consequently, the values generated by the FWD basic block are utilized by the corresponding REV block, resulting in identical lifetimes for all values. In Figure 2.5, layer-1's first iterations produce values with the longest lifetime, mapped to the red bar in Figure 2.8. The tape entries, produced by the physical model (`force`), are mapped to the light blue bar (low lifetime). FWD:NN layer-1 (Figure 2.5) writes the two color-coded values to the tape in each iteration. These values are read together in the REV, allowing for static scheduling of the accesses. As they are consumed together, it is best to store them together in the tape, which makes it possible to implement the tape as an array-of-structs (AoS) to optimize stream efficiency.

Summary

It is challenging to map the tape to the cache because: i) the tape is large; The size of the tape is proportional to the number of operators in the FWD and the critical path. ii) The tape contains operands that exhibit both short and long reuse. iii) Both FWD and REV have non-tape memory accesses. Using the cache for both tape and non-tape values can result in cache conflicts. Consequently, the AD compiler must take responsibility for the tape orchestration because it has sufficient knowledge of the tape to explicitly orchestrate it. In Chapter 3, we describe the compiler passes inspired by the tape's characteristics that enable us to explicitly orchestrate the tape.

Chapter 3

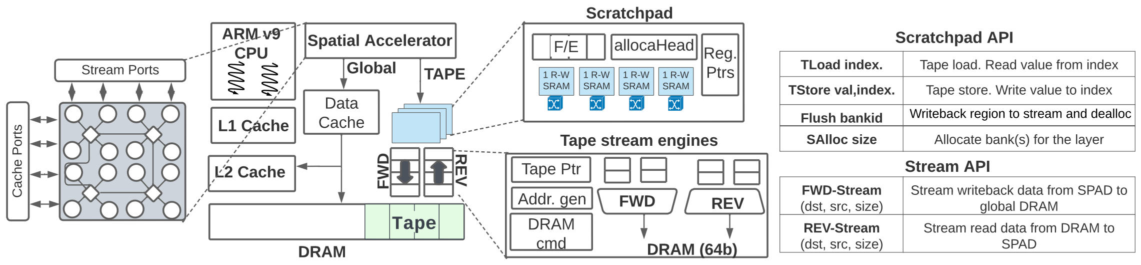
Tapeflow: Compiler Support for Tape

In this chapter, we address the question of how to explicitly orchestrate the tape using the compiler. We present a detailed explanation of the underlying hardware and the compiler passes in Tapeflow and their capabilities to transform the tape layout and efficiently manage DRAM transfers. These passes draw inspiration from the tape's characterization analysis discussed in Chapter 2.3. By leveraging these insights, we aim to optimize the tape management process and enhance the overall performance of the system.

3.1 Target Hardware for Tapeflow Compiler

Tapeflow is specifically designed to optimize the memory hierarchy of a generic spatial architecture [16, 30]. The compiler operates on a post-optimized LLVM-IR that includes loop unrolling, enabling a high level of operation-level parallelism on floating-point data types. Spatial accelerators offer excellent support for parallelism and allow explicit scheduling of both compute operations and memory accesses. By leveraging a spatial accelerator, we can evaluate the capabilities of our compiler passes without being constrained by the specific architecture.

To begin, let's examine the compute tile. It comprises a grid of processing elements (PEs) onto which the forward (FWD) or reverse (REV) dataflow is mapped. The spatial accelerator takes the form of a loosely coupled Coarse-Grained Reconfigurable Array (CGRA) consisting of 4×4 double-precision functional units, similar in design to Dyser [7]. The decision to use double-precision is based on the baseline Enzyme compiler's utilization of doubles, and the exploration of low-precision floats in the context of Automatic Differentiation (AD) remains an ongoing research topic. To schedule the LLVM IR operations onto the functional unit grid, we employ a CGRA mapping pass. Each functional unit in the grid corresponds to a single instruction from the dataflow graph of the FWD and REV operations. Data dependencies between these operations are explicitly routed over a static mesh operand network. The



Scratchpad API

TLoad index.	Tape load. Read value from index
TStore val,index.	Tape store. Write value to index
Flush bankid	Writeback region to stream and dealloc
SAlloc size	Allocate bank(s) for the layer

Stream API

FWD-Stream (dst, src, size)	Stream writeback data from SPAD to global DRAM
REV-Stream (dst, src, size)	Stream read data from DRAM to SPAD

Figure 3.1: Overview of Tapeflow Architecture. The layers are mapped to the spatial accelerator, the non-tape memory accesses go through the cache while tape accesses go through the scratchpad and streamed to the DRAM by the stream engine.

spatial mapper builds upon prior work [7] and incorporates relevant techniques and insights from that domain.

By leveraging the capabilities of the spatial architecture and employing techniques like loop unrolling, operation-level parallelism, and explicit scheduling, Tapeflow aims to optimize the memory hierarchy of the target architecture and maximize performance for the given AD tasks.

Tapeflow adopts a partitioned memory model facilitated by statically analyzable tape accesses. Tape accesses are orchestrated by Tapeflow the compiler into the scratchpad, while non-tape accesses are handled by the cache. Enzyme adopts a unified memory model in which both tape and non-tape accesses are orchestrated by the hardware cache. Here, we describe the features of AD that make Tapeflow's optimization feasible.

Choice 1: Incoherent tape regions. A typical challenge with partitioned address spaces is coherency. AD's tape accesses sidestep these challenges. The tape is auxiliary to the program state and completely invisible to the user. This makes it feasible to redirect tape accesses.

Choice 2: Shared scratchpad. The tape read and writes are partitioned in time by a barrier, which allows for sharing of the scratchpad by FWD and REV. Additionally, compiler-generated REV makes tape accesses statically analyzable.

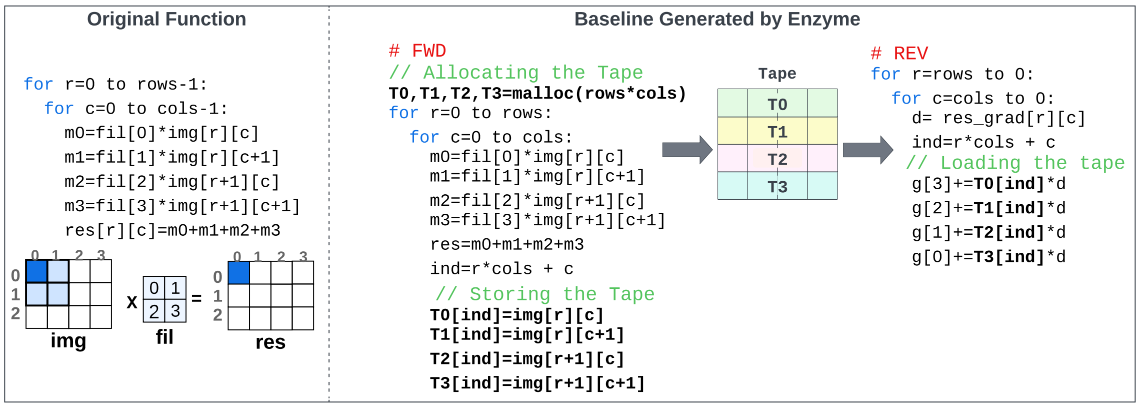
Choice 3: No dirty bits; lazy consistency. In FWD, all values written to the scratchpads will be written to the tape (only dirty entries). In REV, the tape entries are only read (no dirty entries). We overload the full/empty bits to serve as synchronization and write barriers.
Tapeflow partitions the tape accesses away from the cache hierarchy and employs software-managed stream engines to move the tape between the scratchpad and the DRAM. Here, we describe the scratchpad and stream engine.

Scratchpad: The scratchpad is a set of banked stream registers that can be indexed individually from the datapath. Indexing enables unordered accesses from the datapath and the reuse of tape entries. The entry indices are generated by the compiler (like registers) and will not exceed the size of the scratchpad, since the compiler would break the regions in case of an overflow. Tape stores and loads are similar to register operations and are alias free. During FWD, the datapath only issues stores to the scratchpad and streams out to the DRAM. During the REV phase, the datapath only performs loads, with no demand misses, and streams transfer data tiles to and from the scratchpad. The F/E bits act as memory barriers, triggering a stream out to DRAM when set to full (1) during FWD and indicating that a tape region's values are available on-chip during REV. The overall throughput of this system will be determined by the available memory bandwidth for streaming from the DRAM. Finally, each bank is an SRAM bank that includes 1 R/W port; at any given instant, either read or write phase is active per bank.

Stream Engines: There are logically two stream engines: one **from** DRAM (REV-S) and the other **to** the DRAM (FWD-S). They provide the compiler with maximum flexibility to partition layers in the dataflow graph and maximize instruction parallelism and locality for a given scratchpad size. The compiler introduces operations in the dataflow to initiate the stream accesses. `REV-stream` loads data from memory, taking as arguments a destination scratchpad id and the size of data that has to be read from the tape. `FWD-stream` takes data from the designated scratchpad and writes it to the end of the tape on the DRAM. Each stream moves a variable length worth of data (depending on a tape tile) to/from the DRAM.

3.2 Compiler Overview

In this section, we provide an overview of the compiler passes used in Tapeflow by using an example in Figure 3.2. Our tool, built on top of Enzyme, supports arbitrary code encompassing loops with unknown bounds, indirect array accesses, etc. The example showcases a convolution operation where a filter, denoted as `fil`, is applied to an image, with the

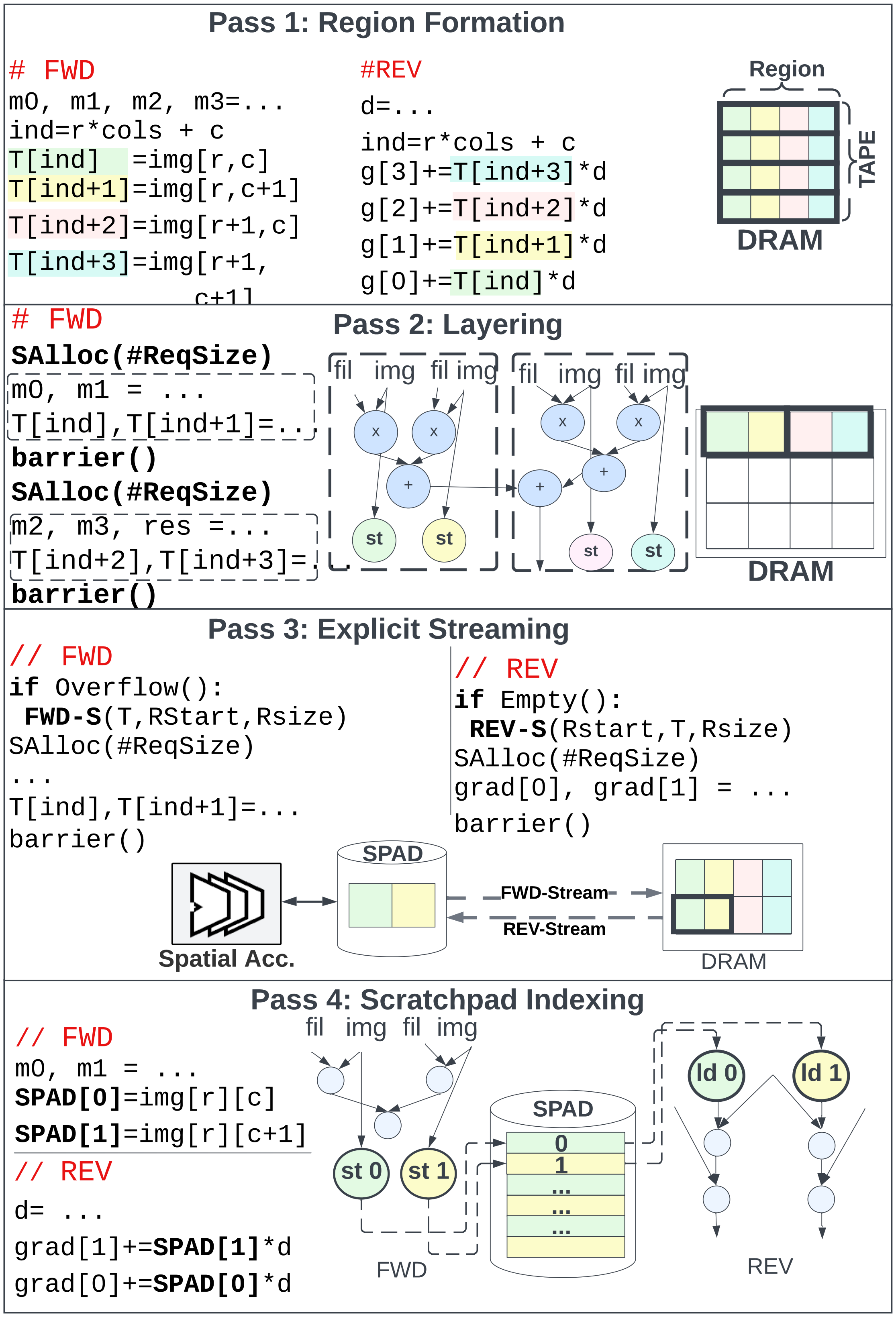


```
for r=0 to rows-1:
  for c=0 to cols-1:
    m0=fil[0]*img[r][c]
    m1=fil[1]*img[r][c+1]
    m2=fil[2]*img[r+1][c]
    m3=fil[3]*img[r+1][c+1]
    res[r][c]=m0+m1+m2+m3
```

```
# FWD
// Allocating the Tape
T0,T1,T2,T3=malloc(rows*cols)
for r=0 to rows:
  for c=0 to cols:
    m0=fil[0]*img[r][c]
    m1=fil[1]*img[r][c+1]
    m2=fil[2]*img[r+1][c]
    m3=fil[3]*img[r+1][c+1]
    res=m0+m1+m2+m3
    ind=r*cols + c
     // Storing the Tape
    T0[ind]=img[r][c]
    T1[ind]=img[r][c+1]
    T2[ind]=img[r+1][c]
    T3[ind]=img[r+1][c+1]
```

```
# REV
for r=rows to 0:
  for c=cols to 0:
    d= res_grad[r][c]
    ind=r*cols + c
   // Loading the tape
    g[3]+=T0[ind]*d
    g[2]+=T1[ind]*d
    g[1]+=T2[ind]*d
    g[0]+=T3[ind]*d
```

Figure 3.2: Example code and the baseline gradient function generated by Enzyme.



```
# FWD
m0, m1, m2, m3=...
ind=r*cols + c
T[ind]  =img[r,c]
T[ind+1]=img[r,c+1]
T[ind+2]=img[r+1,c]
T[ind+3]=img[r+1,
            c+1]
```

```
#REV
d=...
ind=r*cols + c
g[3]+=T[ind+3]*d
g[2]+=T[ind+2]*d
g[1]+=T[ind+1]*d
g[0]+=T[ind]*d
```

```
# FWD
SAlloc(#ReqSize)
m0, m1 = ...
T[ind],T[ind+1]=...
barrier()
SAlloc(#ReqSize)
m2, m3, res =...
T[ind+2],T[ind+3]=.
barrier()
```

```
// FWD
if Overflow():
 FWD-S(T,RStart,Rsize)
SAlloc(#ReqSize)
...
T[ind],T[ind+1]=...
barrier()
```

```
// REV
if Empty():
 REV-S(Rstart,T,Rsize)
SAlloc(#ReqSize)
grad[0], grad[1] = ...
barrier()
```

```
// FWD
m0, m1 = ...
SPAD[0]=img[r][c]
SPAD[1]=img[r][c+1]
// REV
d= ...
grad[1]+=SPAD[1]*d
grad[0]+=SPAD[0]*d
```

Figure 3.3: Overview of Tapeflow Compiler Passes. Pass1: Region Formation and Array-of-Structs Layout. Pass2: Layering and Tiling. Pass3: Explicit Streaming. Pass4: Scratchpad Indexing.

resulting output stored in the array `res`. The objective is to compute the gradient of `res` with respect to `fil`.

Initially, the original function is fed into the Enzyme to generate the gradient function. Enzyme handles the creation of the reverse pass (REV) and allocates four distinct tape arrays for each SSA value generated during the unrolling process, `T0` to `T3`. It stores the tape values in these arrays during FWD and loads from them to calculate the gradients during REV. To enable explicit management of the memory hierarchy, Tapeflow introduces four additional passes that extend Enzyme's capabilities. We provide a brief description of each pass below:

Pass 1 (Region Formation): The first pass focuses on converting the layout to an array-of-structs (AoS) representation and updating the tape loads and stores accordingly. This transformation enhances memory access patterns and optimizes the tape structure.

Pass 2 (Layering and Tiling): The second pass divides the computation into layers and tiles the tape into chunks that fit within the scratchpad. This process facilitates efficient utilization of the scratchpad memory and improves performance by reducing data movement.

Pass 3 (Explicit Streaming): The third pass decouples the tape from the computation and inserts preloads of the tape at layer boundaries. By separating the tape management from the computational operations, this pass enables efficient streaming of data, further enhancing performance.

Pass 4 (Scratchpad Indexing): The final pass rewrites the tape accesses by replacing tape loads and stores in each layer with `Tload`s and `Tstore`s, respectively. This rewriting process generates the appropriate scratchpad addresses, effectively connecting the LLVM-IR to the scratchpad memory.

In the rest of this chapter, we delve into each pass in detail, illustrating their transformations and the corresponding impact on the example provided in Figure 3.2. It is worth mentioning that all the FWD-REV dependencies are known at compile time because the REV is compiler generated. In the algorithms described in each pass, the `FtoR` map is a map from FWD operations to their corresponding REV operation.

3.3 Pass1: Region Formation

Goal: *Change the tape layout from Struct-of-Arrays to Array-of-Structs. This reduces the number of cache blocks touched and bookkeeping required for the tape.*

We rewrite the tape into an array-of-structs layout shown in Figure 3.3: Pass1. For each tape operand, we have to assign a slot in the tape. The question is which slot to choose? Tapeflow chooses the slot based on spatial use within the REV's dataflow. Tapeflow merges multiple tape arrays, and assigns adjacent slots to the tape values that are used together in the dataflow in an array-of-structs fashion. This increases the spatial locality of the tape and improves the DRAM efficiency. A region is a single struct that mixes tape operands from different arrays. For example, in Figure 3.3:Pass1, we merge the tape arrays `t0`, `t1`, `t2`, and `t3`. The color-coded tape stores are all mapped to adjacent slots in the same region in the tape.

We transform the tape by merging the tape arrays into a single array-of-structs. Algorithm 1 demonstrates the process for each tape access. The slot's address is generated in two steps. First, we generate the base address of the struct in the array-of-structs, which is common across all the operands in the region (lines 3 and 6). Second, we calculate the offset within the struct based on the temporal order (lines 8 and 11). This is feasible because the tape is a compiler introduced data-structure and all FWD-REV dependencies are known.
Input: Unorganized Tape **Output:** Array-of-Structs Tape.

Algorithm 1: Create AoS layout

```
    Input: FWDLoops: Set // Loops in FWD
    Input: FtoR: Map // Dictionary of gradient ops in REV
 1  foreach $loop \in FWDLoops$ do
 2    RSize: // Size of loop's region.
 3    FWDBase←loop.Tape + loop.Index × RSize
 4    // Region's base address.
 5    REVLoop ← FtoR[loop]
 6    REVBase←loop.Tape+ REVLoop.Index×RSize
 7    foreach $TStore \in loop.Ops$ do
 8      TStore.dst←FWDSBase + offset
 9      // Find tape load in REV.
10      TLoad←FtoR[TStore]
11      TLoad.src←REVSBase + offset
12      offset←offset + TStore.size
```

3.4 Pass2: Layering

Goal: *Schedule execution into layers, such that each layer's tape accesses are restricted to a single region.*
Input: Program graph **Output:** Partitioned layer graph

The pass splits the computation into layers. A layer is a partition of dataflow that is constrained to source operands from a single tape region. This means a layer can only touch a specific range of addresses in the tape. Further, a constraint is set on the size of each region to a fixed limit, which is the size of the on-chip scratchpad. Each layer starts with `SAlloc`, which allocates space for the tape region, and terminates with a barrier (Figure 3.3:Pass2). The barrier forces the layer to access tape only within the region. Algorithm 2 illustrates how Tapeflow populates `Layers`, a map that contains the set of layers for each loop. During the FWD, the layers execute in the FIFO order, and their order is inverted in the REV.

Algorithm 2: Layer Generation

```
Input: FWDLoops: Set // Loops in FWD
Input: FtoR: Map // Dictionary of gradient ops
1  foreach $Loop \in FWDLoops$ do
2    REVLoop←FtoR[Loop]
3    // Init layer map in FWD and REV.
4    Layers[Loop], Layers[REVLoop]←[ ]
5    // Init FWD and REV list of ops.
6    FWDOps, REVOps←[ ], Count←0
7    foreach $Op \in Loop.Ops$ do
8      if $Op$ is $TStore$ then
9        if $Count > SPAD.size$ then
10         // Terminate layer
11         Layers[Loop]←{FWDOps}
12         Layers[REVLoop]←{REVOps}
13         FWDOps,REVOps←[], Count←0
14       Count←Count + 1
15     FWDOps←{Op}
16     REVOps←{FtoR[Op]}
```

To generate the layers, we iterate over the operation within a loop and track the ops of the current layer in `FWDOps` and `REVOps`. When the layer exceeds the size of a region or scratchpad (line 10), we terminate it and start a new one. We use an auxiliary data structures, `FtoR`, that maps FWD operations to gradient operation in the REV.

3.5 Pass3: Explicit Streaming

Goal: *Insert decoupled streams at layer boundaries to runahead and preload a tape region into the scratchpad.*

Input: Layer graph **Output:** Streaming layer graph

This pass seeks to make efficient use of DRAM bandwidth. Tapeflow iterates over the dataflow and calculates the tape usage across all the operations. This allows us to dynamically build tiles of regions that can be streamed from the DRAM. At each layer boundary, if we run out of space in the scratchpad, we spill the entire scratchpad to the DRAM using a `FWD-stream` and concatenates it to the tape in the DRAM (Figure 3.3:Pass3).

`FWD-Stream` receives three input parameters: i) destination address in DRAM, ii) source address in scratchpad, and iii) transfer size. The destination address is computed based on the tape address allocated for this loop by the malloc call in pass 1. The source address shows the start of this layer in the SPAD. The transfer size is the size of the current region, `RSize`. The tape pointer in DRAM (destination) and the scratchpad source address (source) are maintained by Tapeflow during execution as the program states. The tape pointer is updated every time a stream command (`FWD-Stream` or `REV-Stream`) is issued. The scratchpad source address is updated upon visiting a `SAlloc`.

A challenge is how to coordinate streaming with the REV's execution since only the FWD is aware when the tape is spilled to the DRAM and its size. To solve this, we record the parameters of the `FWD-stream` in a stack and insert `REV-Stream`s at the head of each layer in the REV. The `REV-Stream`s read from the top of the stack to know how many tape values they need to bring from DRAM.

During the FWD execution, streaming facilitates the transfer of layer data from the scratchpad to the DRAM. To elaborate, before executing layer 2, the tape data from layer 1 is moved from the scratchpad to the DRAM. The layers are processed sequentially during the FWD. To keep track of the streams being transferred to the DRAM, Tapeflow employs a stack. This stack maintains the DRAM addresses for each stream, with the most recent stream always occupying the top of the stack. During the REV execution, the layers are also processed sequentially but in reverse order compared to the FWD layers. This implies that the first REV layer calculates the gradient of the last FWD layer, while the last REV layer calculates the gradient of the first FWD layer. The execution order of the layers aligns with the order of the stream stack. Consequently, to determine the DRAM address for the subsequent streams in the REV phase, Tapeflowreads the top of the stack and removes it (pops it).

Other than the order of the tape across layers, streaming preserves the order of the layer's tape in the scratchpad. Since the tape store and its corresponding tape load use the same index in the scratchpad, the order of the tape values must be preserved when the scratchpad is evicted during FWD and reloaded in the REV. By streaming the values, Tapeflowguarantees the order remains intact.

3.6 Pass4: Scratchpad Indexing

Goal: *Replace tape loads and stores with scratchpad loads and stores to reduce DRAM bandwidth.*

Input: Streaming layer graph **Output:** Layer graph with scratchpad accesses.

The goal of this pass is to rewrite tape accesses to use the scratchpad, which is an alternative local address space. The scratchpad is accessed using a base and offset pair. Tapeflow injects a `SAlloc` instruction at the start of each layer to obtain the base address within the scratchpad. The next step finds an offset or slot within the scratchpad for each tape operand. Finally, the compiler lowers or binds the tape loads and stores to scratchpad loads and stores. We assign offsets and addresses in memory order of the tape stores within the FWD's layer; the offset counter is initialized to 0 and incremented on every tape store. The same offset is used in the corresponding tape load in the reverse. This offset is com-

Algorithm 3: Scratchpad Indexing

```
Input: FWDLoops: Set // Loops in FWD
Input: FtoR: Map // Dictionary of gradient ops
Input: Layers:Map // Pass 2 output. Layer set
1  foreach Loop ∈ FWDLoops do
2    foreach Layer ∈ Layers[Loop] do
3      offset←0
4      foreach TStore ∈ Layer.Ops do
5        // Set the dest of TStore in SPAD.
6        TStore.dst← Layer.SAlloc + offset
7        // Update the source of the corresponding tape load.
8        TLoad ← FtoR[TStore]
9        RevLayer ←FtoR[layer]
10       TLoad.src←RevLayer.SAlloc + offset
11       offset←offset + TStore.size
```

bined with the scratchpad base address to obtain the effective address for the tape operand (Figure 3.3:Pass 4, `T[ind]` is replaced by `SPAD[0]`).

3.7 Layer Creation Trade-offs

The size of the scratchpad is the primary constraint in layer creation. A smaller scratchpad allows for fewer tape values to be produced or consumed, resulting in smaller layers. Conversely, a larger scratchpad enables the exploitation of parallelism through the creation of larger layers. The effects of scratchpad size on layers and performance are as follows:

- Parallelism: In dataflows with numerous parallel operations, a larger scratchpad enables the formation of larger layers, facilitating the utilization of parallelism. Conversely, a small scratchpad necessitates the division of the dataflow into smaller layers, and since the layers are executed sequentially, both performance and hardware utilization suffer.

- Redundant tape stores: In certain cases, tape stores may have multiple consumers in the REV. When layers are small, these consumers may fall into separate layers. To ensure that the tape values in the FWD layers of all consumers remain isolated in the scratchpad, Tapeflowintroduces redundant tape stores to the FWD layers to duplicate the tape values.

- Barriers and idle time:With a small scratchpad, Tapeflowcontinually breaks the dataflow into smaller layers that fit within the scratchpad. As a result, having more layers leads to more synchronization points and increased idle time.

It is important to note that the scratchpad size is not the sole factor restricting performance, and other architectural limitations can act as bottlenecks. For example, the number of floating-point units (FPUs) determines the level of parallelism achievable in floating-point operations. Similarly, the number of cache ports is another limiting factor. When the number of memory accesses is high and there are insufficient cache ports, accesses will be serialized, thereby limiting performance.

Chapter 4

Evaluation

4.1 Benchmarks

Table 4.1 lists the benchmarks we use for evaluating Tapeflow. These benchmarks were derived from physics models [24], scientific computing, and tensor compilers [21]. The benchmarks are composed of loops reading data from memory and doing computation on it. We categorized them into two groups, regular and irregular, based on their cache pressure. A program with only one loop working with sequential elements of an array (logsum) would not impose much pressure on the cache. The cache is more likely to experience conflicts when there are multiple data dependent nested loops, each accessing various blocks of memory (mass spring). Using the cache for the tape accesses increases these conflicts even further. Step size also contributes to the cache pressure. When the step size is one, it usually means the loop accesses consecutive elements, exploits the spatial locality, and have a high hit rate. We draw the line between regular and irregular by considering the associativity of our

	Regular					
Name	tensor/ loop	Work. Set(K)	Inp. (K)	Suite	input params	layer count
Gravity	2	180	8	Diff [17]	Array: 512	15616
NN	2.6	63	80	Enz. [24]	Img: 28 x 28	490
Logsum	2	160	160	Enz. [24]	Input: 10K	1251
Matd.	3.5	1280	2400	Enz.[24]	M,N: 400	10400
	Irregular					
MTTK.	7.	104	128	Taco [21]	8x8x8	512
Somier	8.	34	48	RV[41]	8x8x8	216
Lenet-5	4.	430	10	Lenet[2]	-	11895
Pathf.	4.	328	164	RV [41]	R:128, C:256	5842
Mass Spring	6.	10	28	Diff [17]	Obj:128, hidden:32	46000

Table 4.1: Benchmarks description

baseline cache (4 way). We mark the benchmarks as regular when they access three or less different arrays (including tape) in a single loop body with stride one.

For each benchmark, we generate the compiler IR using the state-of-the-art Enzyme compiler [24]. For each target function, `f()`, the Enzyme generates a gradient function `grad_f`. We use `-O3` and `-mem2reg` optimizations to aggressively optimize the memory footprint of the benchmark. We define this as our baseline, `Enzyme`. Tapeflow adds additional passes to Enzyme to segment `grad_f` into layers and insert instructions for interacting with the tape.

4.2 Scratchpad Setup

The scratchpad serves as a dedicated on-chip memory resource that holds the tape to be locally accessed by the accelerator. Our scratchpad is organized as follows: The total size of the scratchpad is 1KB, divided into 16 banks. Each bank consists of 8 × 8B entries, providing a total of 128 entries across the scratchpad. This organization allows for efficient data storage and retrieval, enabling accelerated access to the tape values.

The scratchpad configuration remains the same for all the benchmarks across all the evaluations. we investigate the impact of the scratchpad size on performance metrics such as instruction-level parallelism (ILP) and overall execution time in the design exploration section 4.5. By systematically varying the scratchpad size, we analyze the trade-offs between the available on-chip memory capacity and the performance benefits obtained from reduced memory latency and improved data access patterns.

4.3 Simulation Framework

We simulated the complete accelerator (including OOO core), in detail, using gem5-salam [44]. The FWD and REV are mapped to the spatial accelerator that includes its own data cache, which is coherent at the LLC. We model a CGRA, a spatial heterogeneous fabric accelerator. Gem5-salam takes as input the LLVM representation from Tapeflow and schedules it on the underlying spatial accelerator. It includes support for partitioning the memory space and composing multiple memory models. This allows us an apples-to-apples comparison, where we can fix the accelerator datapath while varying the memory model. Table 4.2

gem5-salam config									
CPU				Memory			Scratchpad		
OOO ARM v9 2GHz, fetch width:8, issue width: 8				DDR4 4 GB 19.2 GB/s			1 KB, 16 banks of 64B		
Storages Energy Consumption									
Storage	scratchpads	$Cache_{1k}$	$Cache_{2k}$	$Cache_{4k}$	$Cache_{8k}$	$Cache_{16k}$	$Cache_{32k}$	$Cache_{64k}$	$Cache_{128k}$
Energy (pj)	100	120	440	450	460	470	2990	10800	11350
Associativity	-	2-way	2-way	2-way	2-way	2-way	4-way	fully	8 way

Table 4.2: System configuration

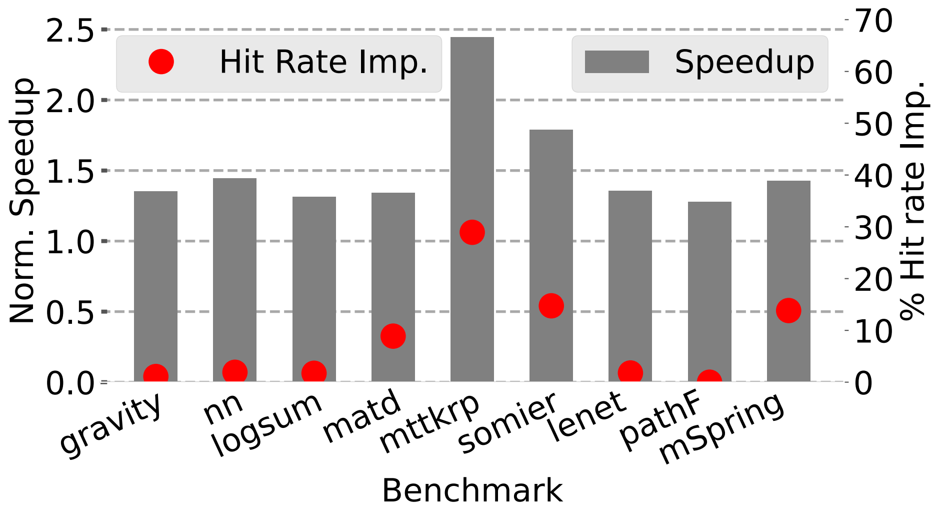
shows the configurations we used for our simulations with gem5-salam. The configuration for each cache size is also specified in table 4.2. AD requires floating point support and all our designs use a generic 16 PE system, in which each PE includes support for double FPUs. Thus, Tapeflow or cache needs to supply up to 32 doubles from the tape at a time. We simulate the following configurations:

- $Enzyme_{32k--128k}$: 32k: 4-way. 128k: 8-way. The Enzyme compiler maintains the tape, and on-chip movement is handled by the cache. We ran the benchmarks with different cache sizes between the range of 1k to 128k. We use 32k as baseline since it achieved 90% of 128k's performance. We use 128k, 8-way as upper bound.
- $Tflow_{32k}$ (*ISO-Energy* setup): In Tapeflow, tape accesses always use the scratchpad (8 entry, 16 banks in baseline plots). In addition, Tapeflow also includes need a cache for non-tape accesses.
- $Tflow_{2k}$ (*ISO-Perform* setup): 2k, 2-way. Since Tapeflow eliminates the tape accesses from the cache, a smaller cache suffices to achieve comparable performance. We picked the lowest energy cache (2k) to demonstrate what performance it can achieve compared to $Enzyme_{32k}$.
- $Tflow_{1k}$ **and** $Enzyme_{1k}$: 1k, 2-way. We also choose a small cache that can not fit the working set and tape. We demonstrate how Tapeflow is able to improve the behavior for such atypical small caches.

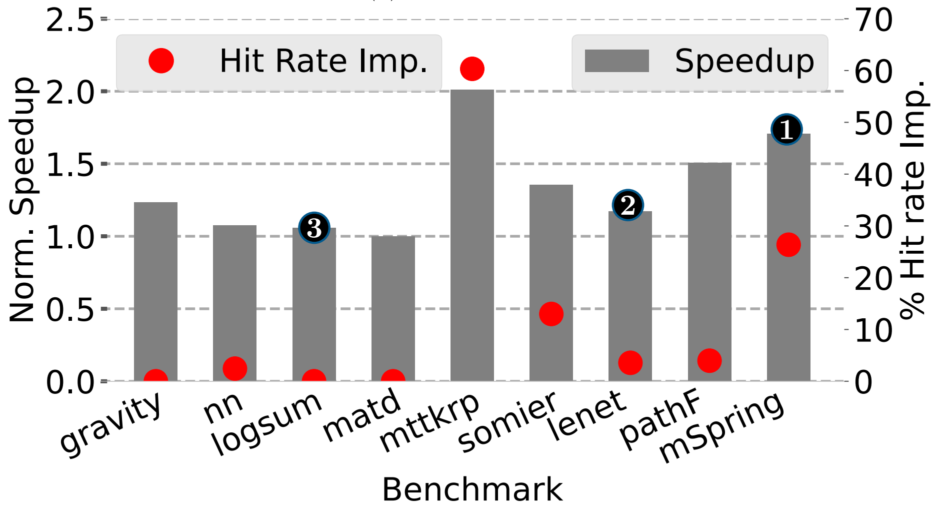
4.4 Results

We answer the following questions:

- How much can Tapeflow improve performance relative to Enzyme [24]? **Answer:** 2.4×. § 4.4.1
- How much DRAM bandwidth can Tapeflow save by streaming ? **Answer:** 14×. § 4.4.2
- What is the impact of Tapeflows array-of-struct layout, in isolation, on DRAM bandwidth? **Answer:** up to 30% reduction. § 4.4.2
- How much on-chip energy and cache size can Tapeflow save ? **Answer:** 8×. § 4.4.3
- How much can Tapeflow shrink the cache size while maintaining the same performance? **Answer:** 4-8×. § 4.4.4



(a) REV Speedup



(b) FWD Speedup

Figure 4.1: Left Y-axis: speed-up. Right Y-axis: Hit rate improvement (higher is better)

4.4.1 Performance Evaluation

Result: *Tapeflow achieves up to* 2.4× *improvement over Enzyme for three reasons: i) Tapeflow removes the tape loads and stores from the critical path. ii) Tapeflow reduces the REV's compulsory misses by steaming in the tape, and iii) Tapeflow reduces the cache pressure by eliding the tape stores in FWD, and tape loads in the REV.*

To elaborate on the Tapeflow's speed-up over the Enzyme, we plot the FWD and REV speed-ups separately. The performance improvement is due to the interplay of three effects:

i) *Streaming accesses:* Tapeflow streams in the tape data proactively in the REV and eliminates the reactive cache fills. Tape accesses would not stall the REV anymore because they are already on chip when the REV layer starts its execution.

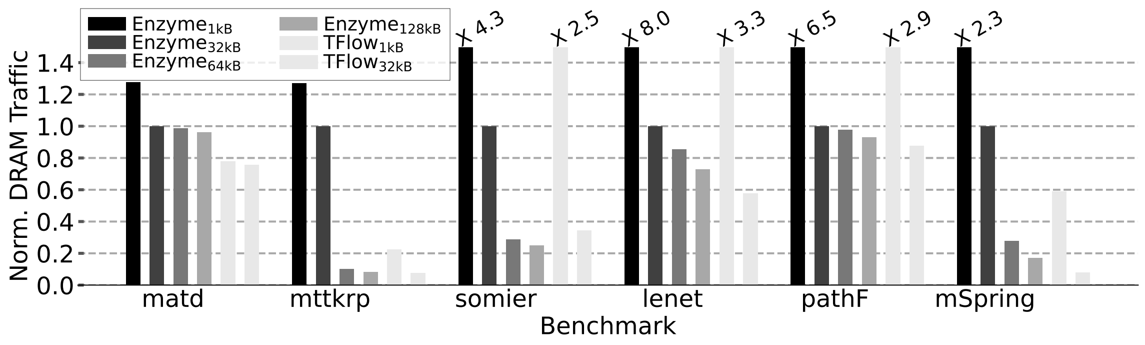


Figure 4.2: Normalized DRAM access (lower is better). 1 = $Enzyme_{32k}$

ii) *Eliminating conflict misses:* Tapeflow reduces the pressure from the cache, as the tape accesses do not interfere with other accesses anymore. For instance, ❶ *mass spring* accesses 3 arrays (`offs`, `v`, and `o_wgt`) in the first innermost loop of the FWD. It also writes 4 values to a tape organized as a struct-of-arrays in Enzyme. This means every red value in Figure 2.5 is written to a separate array. As a result, each iteration accesses 7 different arrays (3 input-output and 4 tape) resulting in lots of cache conflicts. Tapeflowredirects the tape accesses from the cache to the scratchpads and eliminates conflicts caused by the tape. In cases the contention is not so high, *Lenet* for example ❷, and the arrays are not touched all at once in a single layer, the access pattern is considered to be more regular with less room for conflict resolution.

iii) *Improving data retention:* The irregular workloads are more likely to suffer from early evictions. The REV regions have multiple tape loads from different blocks. Figure 4.1 shows that Tapeflow has increased the hit rate in the REV, especially in Irregular benchmarks. In workloads that already exhibit streaming behavior ❸ (Logsum) or when the tape fits in the cache (*pathF*), the access pattern is already optimized and Tapeflow cannot improve it any further.

4.4.2 DRAM Bandwidth

Result: *Tapeflow reduces the DRAM traffic against the same size cache by up to* 14× *($Tflow_{32k}$ vs. $Enzyme_{32k}$).*
Result: *Tapeflow is less prone to varying the cache size.*
Result: *$Tflow_{32k}$'s DRAM traffic is comparable to a much larger cache $Enzyme_{128k}$. It consumes up to* 20% *less traffic despite being* 4× *smaller, having lower associativity, and consuming* 3.8× *less energy.*

Figure 4.2 shows the normalized DRAM access across benchmarks for all our configurations. All plots are normalized to the $Enzyme_{32k}$ cache. We plot the DRAM traffic for different Enzyme cache sizes ($Cache_{1k}$ to $Cache_{128k}$) (Table 4.2). We varied the Enzyme

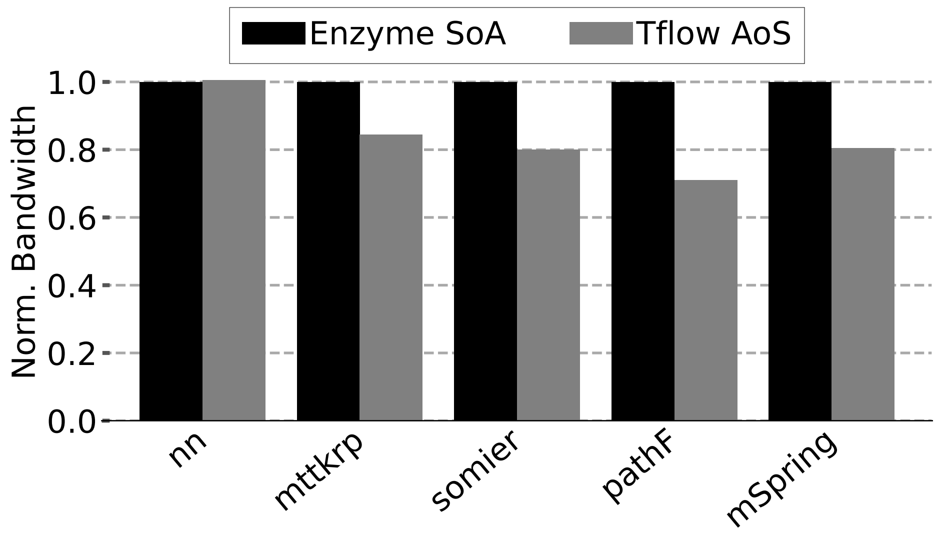


Figure 4.3: Enzyme's vs. Tapeflow's Layout (lower is better).

cache setups to find configurations that exhibit DRAM traffic similar to $Tflow_{32k}$. We also compare the effect of cache size reduction on traffic by plotting the $Tflow_{1k}$.

We increase the size and associativity of the Enzyme cache until it reaches the DRAM traffic of the $Tflow_{32k}$. Increasing the size and associativity of the $Enzyme_{32k}$ reduces the conflict misses caused by tape accesses and consequently, the DRAM traffic reduces. However, this is achieved at the cost of having a larger cache with higher associativity than the $Tflow_{32k}$. Tapeflow reduces the conflict misses by separating the tape accesses from non-tape accesses and managing them using small and power-efficient stream buffers.

The benchmarks with regular behavior are not impacted by the cache size in both Enzyme and Tapeflow (e.g., *Matd*). Tapeflow requires less DRAM traffic than Enzyme ($Enzyme_{32k}$ and $Tflow_{32k}$ bars). Since the tape accesses are partitioned away from the caches, the conflict misses are eliminated. ii) Irregular benchmarks observe significant reduction because their tape accesses interfere with non-tape accesses extensively (14× improvement in *MTTKRP*). In the 32k setting, *Lenet* and *Pathf* exhibit only marginal improvement since their working sets fit in the cache due to their high associativity. In the 1k settings for these two workloads ($Enzyme_{1k}$ and $Tflow_{1k}$), we observe 2.3× improvement. The regular workloads do not suffer from tape traffic and hence see minimal improvement (1.3× for *Matdescent*). *Matdescent* is matrix-vector multiplication and has a streaming access pattern to both input arrays and the tape and does not have many conflict misses.

Struct-of-Arrays vs. Array-of-Structs:

Figure 4.3 shows the impact of pass1 that changes the layout to array-of-structs. We compare the bandwidth of the code generated after pass 1 against the baseline enzyme. In this comparison, both enzyme and Tapeflow rely on the cache to perform the orchestration.

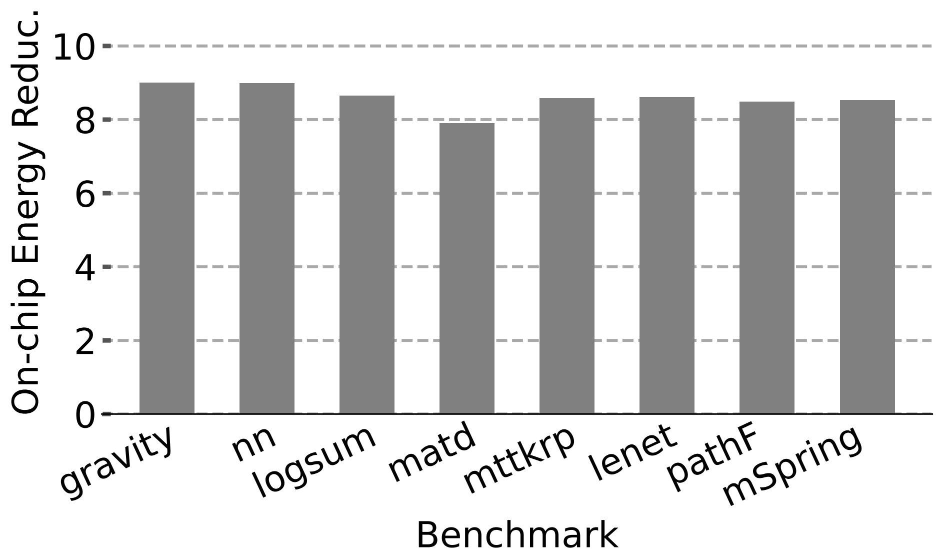


Figure 4.4: On-chip Energy Reduction. (Higher is better)

There are two reasons contributing to this improvement: i) In the AoS layout, we read tape values stored in a single block instead of scattered blocks across different struct-of-arrays. ii) The struct-of-arrays increases the chance of cache conflicts and overflows, leading to additional accesses for evicted blocks.

4.4.3 On-Chip Energy

Result: *Tapeflow reduces the on-chip energy by up to* 8.2× *by streaming the tape accesses (the dominant portion of the working set). Each tape access can be satisfied in less than 10 pJ.* **Result:** *Tapeflow enables the REV and FWD working sets to use a smaller cache (80pJ) while achieving performance comparable to the baseline* $Enzyme_{32k}$ *(300pJ).*

Figure 4.4 shows the energy improvement of Tapeflow over Enzyme. The *ISO-Perform* setup picks the smallest cache that matches the performance of the $Enzyme_{32k}$. We use Cacti [26] for per-access cost. For the larger caches, (32k and 64k) we use a dual port, 4-way, 2 banks. For the rest of the cache setups, we use a 2-way set-associative cache with 1 read/write port. Tapeflow exhibits better energy efficiency. i) Tapeflow partitions out the tape accesses from the cache and reduces cache pressure. Thus, we can use a smaller cache for non-tape accesses, while maintaining performance similar to the baseline. We find a 2k cache is sufficient. This 2k setup requires 6.8× less energy per access. Streaming the tape allows the *ISO-Perform* to remain competitive with large caches and operate with the same speed as $Enzyme_{32k}$ despite having a much smaller cache. ii) Tapeflow directs tape accesses to the scratchpad. Scratchpad is accessed using indexes (like registers) with no misses. They are highly banked (16 banks, 8 entries per bank) and updated with streams from the DRAM. Per tape access, they exhibit 30× less energy than the cache in Enzyme (10pJ vs 300pJ). Enzyme [24] reduces the number of taped values by recomputing them in

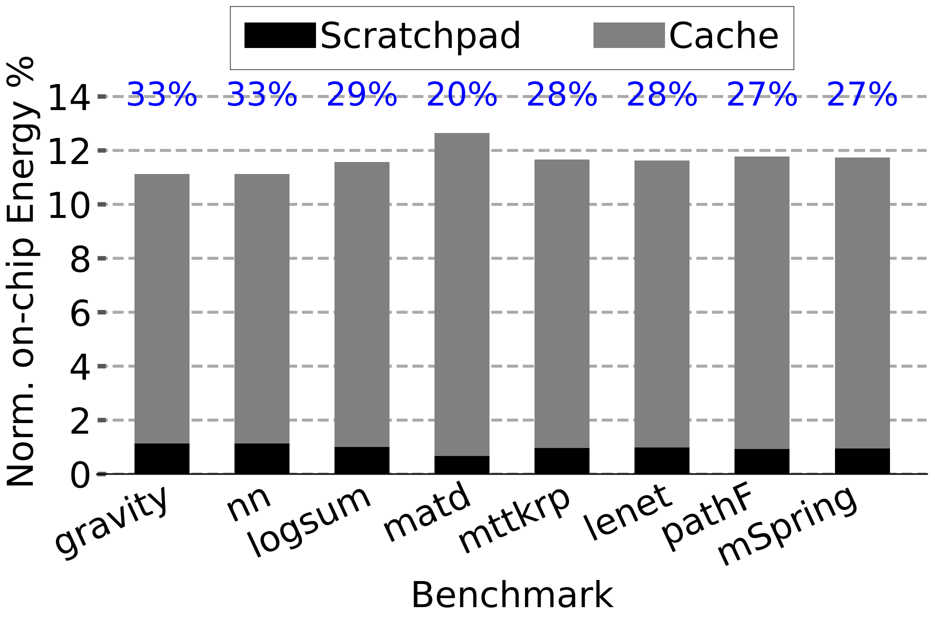


Figure 4.5: Normalized on-chip energy(Lower is better). Numbers on the top show cache access reduction.

the REV. Thus, the smaller cache has a more prominent effect on energy reduction. Our Tapeflow demonstrates that the actual gains can be compounded by partitioning out the tape accesses from the cache.

Figure 4.4 illustrates the combined energy consumption of the cache and scratchpad in the Tapeflow system. The energy values are normalized to Enzyme, and the blue numbers represent the reduction in cache accesses compared to Enzyme. For example, in the `nn` benchmark, 33% of the cache accesses in Enzyme were tape accesses, which are now offloaded to the scratchpad. The scratchpad consumes only 1% of the energy consumed by the 32 KB cache in Enzyme. Remarkably, Tapeflow achieves the same performance while consuming only 12% of the energy consumed by Enzyme. Furthermore, comparing the `matd` and `nn` benchmarks reveals that the more tape accesses a benchmark has, the greater energy savings it can achieve.

4.4.4 Energy-Performance Trade-off

Result: *Tapeflow is more cost-effective than the Enzyme, as it delivers the same performance, consuming much less energy.*

Figure 4.6 plots the performance vs. energy trade-off for each benchmark. The X-axis represents the energy (towards origin is better) and Y-axis the performance (away from origin is better). All configurations are normalized to $Enzyme_{1k}$. The optimal quadrant that we would like to achieve is the top-left (the green line is a visual indicator). Tapeflow is less sensitive to the cache size. Tapeflow redirects all the tape's accesses away to stream

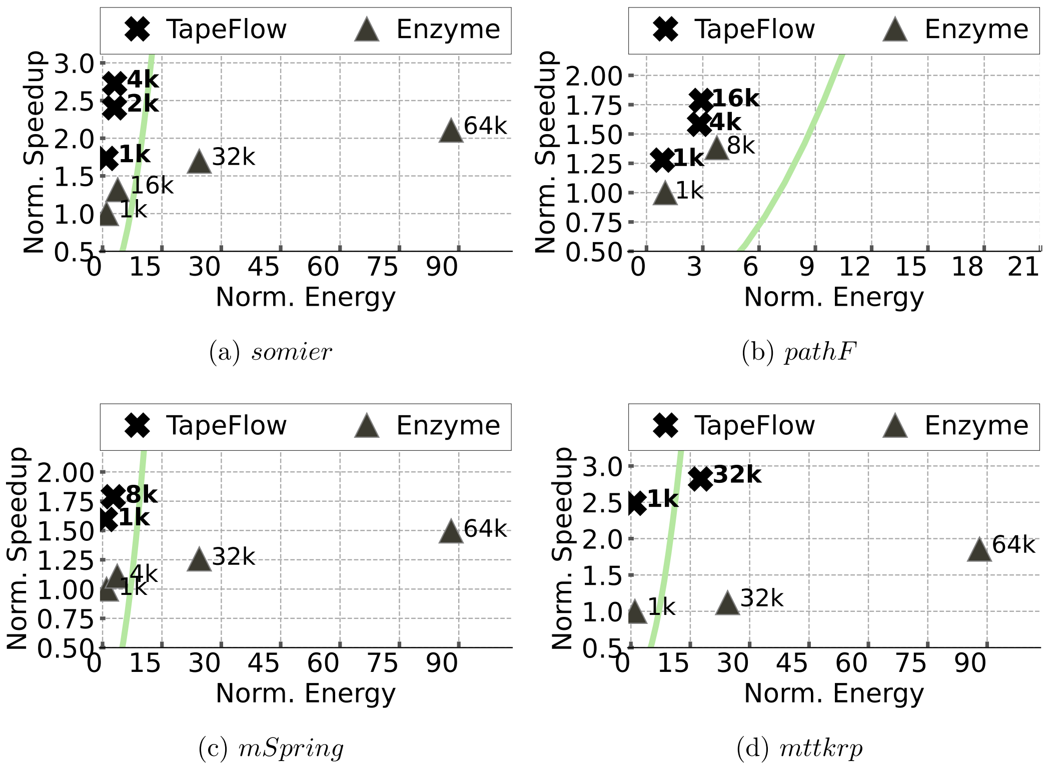


Figure 4.6: Performance-Energy Sweep. Green: Markers to the left have a better trade-off. X-axis: Normalized Energy (Towards origin is better). Y-axis: Normalized performance (Away from origin is better).

buffers, while in Enzyme we need a large cache (and high associativity) to reduce the mingling between the streaming tape and REV accesses. The tape accesses have minimal reuse and lead to many wasted evictions of REV's global data. The performance benefit is more pronounced on smaller caches. $Tflow_{1k}$ and $Tflow_{2k}$ achieve competitive performance against $Enzyme_{32k}$. We achieve high energy benefit even for iso-capacity setups where we expend a similar amount of SRAM (e.g., $Enzyme_{32k}$ vs $Tflow_{32k}$). For instance, in *PathF* benchmark, 30 % of the memory accesses are tape accesses. In Tapeflow, we redirect tape accesses to stream buffers that use less energy per access than the caches.

4.5 Design Exploration

4.5.1 Scratchpad Size

Result: *Increasing the scratchpad size from 64 B to 1 KB, increases the performance of the Tapeflow between 25-50%. A larger scratchpad permits layer size to increase, and consequently uncover more parallel operations.*

Figure 4.7 demonstrates the impact of scratchpad size. The speedup is normalized to $Enzyme_{32k}$. In all benchmarks, the inner loop is fully unrolled, and we sized the spatial ac-

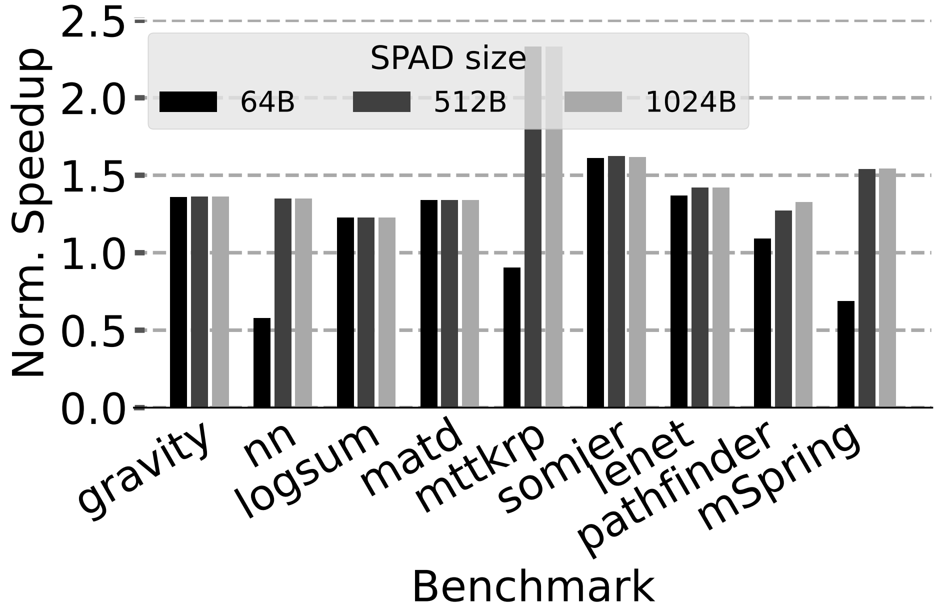


Figure 4.7: Scratchpad size vs. Norm. Performance. Y-axis: Higher is better. X-axis: Scratchpad size

celerators to exploit all the available parallelism, i.e., no compute bottlenecks. The speedup shown in Figure 4.7 is different from Figure 4.1 as it includes the combination of FWD and REV speedup. As we increase the scratchpad size, two things happen: i) *More parallel ops:* Tapeflow can increase the size of each layer; more opportunity to uncover data parallel operations. For the large scratchpads (1 KB) the eventual constraint is parallel ops within layer. ii) Fewer `Sync` operations: Layers are separated by sync barriers. We do use double-buffering to run ahead. However, when the layers are too small (limited by a 64byte scratchpad), the overhead of Sync limits parallelism and performance. This is visible in $somier_{64B}$; performance is the same as Enzyme. Increasing the scratchpad size increases the performance until non-tape accesses limit performance.

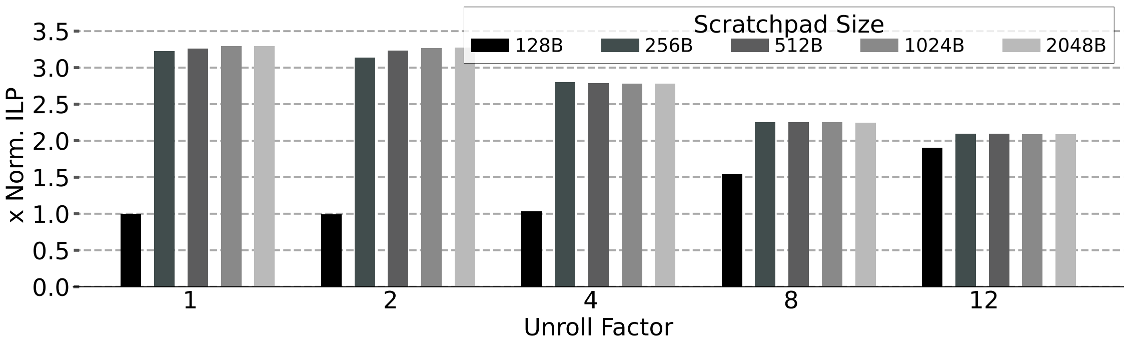


Figure 4.8: Normalized ILP (Higher is better). X-axis: Scratchpad size

Figure 4.8 depicts the influence of scratchpad size on instruction-level parallelism (ILP) for the `somier` benchmark at various unroll factors. The ILP values are normalized to the configuration with $unrollfactor = 1$ (no unrolling) and $scratchpadsize = 128$.

The figure demonstrates that the small scratchpad (128B) consistently acts as a limiting factor, constraining the ILP. However, as the scratchpad size increases, the ILP also increases until another limiting factor arises. Notably, expanding the scratchpad from 1 KB to 2 KB has no impact on the ILP. This is because accesses to non-tape values (e.g., input arrays and partial derivatives) stored in DRAM are serialized due to the limited number of available cache ports.

When the scratchpad size is sufficiently large (2 KB), increasing the unroll factor leads to a reduction in ILP. This reduction primarily stems from the change in how partial derivatives are stored in DRAM when the loop is unrolled. Each SSA in the FWD loop has its partial derivatives stored in distinct DRAM arrays. With a higher unroll factor, more SSA values are generated for the loop, resulting in an increased number of arrays for partial derivatives. Consequently, more cache blocks are required to accommodate these arrays, leading to a higher likelihood of conflicts.

4.5.2 Working set impact

Result: *Tapeflow reduces traffic for large working sets that trash the cache in Enzyme.*

Figure 4.9 demonstrates the effect of working set size on Tapeflow's DRAM accesses. We tune the parameters of the benchmarks (such as input size, number of objects, etc.) such that their tape occupies $\frac{1}{2}$, 1, and 4 $\times$ of the cache capacity. The DRAM access is normalized over total loads + stores of each benchmark. i) On medium or large inputs, as we increase input size, the tape gets larger. Enzyme starts to have more DRAM accesses than Tapeflow because it experiences more conflicts between tape and non-tape accesses. ii) On small inputs, the cache fully captures the working set between the FWD and REV. The scratchpads are stream registers, and not meant for capturing working sets. Thus, Tapeflow by default streams out the FWD and re-streams to the REV leading to higher DRAM traffic in some cases.

4.5.3 Deep vs. Shallow Layer Graphs

Result: *Tapeflow improves performance by 2$\times$ for shallow graphs with wider layers, i.e., more parallelism per layer*

In AD, layer graphs are compiler defined. We pick an irregular workload (pathfinder) that pipelines multiple loop nests. We fix the problem size and control the unrolling factor to create shallow graphs with wide layers (high unroll factor), or deep graphs with narrow layers (low unroll factor). Figure 4.10 depicts how the change in the unroll factor affects the speed-up of $Tapeflow_{32K}$ (all normalized to $Enzyme_{32K}$). On the right Y-axis, we include per-layer parallelism to explain the gains (the red dots in the plot). As layer parallelism

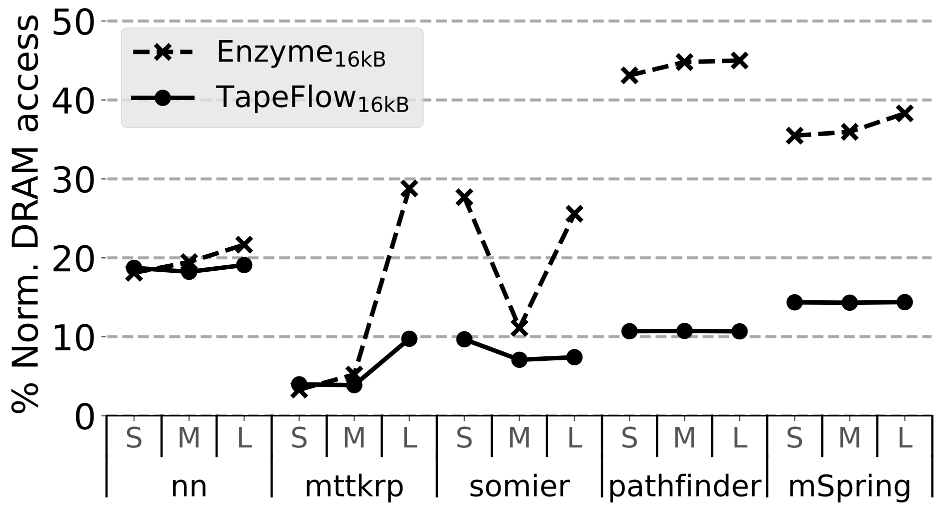


Figure 4.9: Working set size vs. DRAM Traffic. Lower is better.

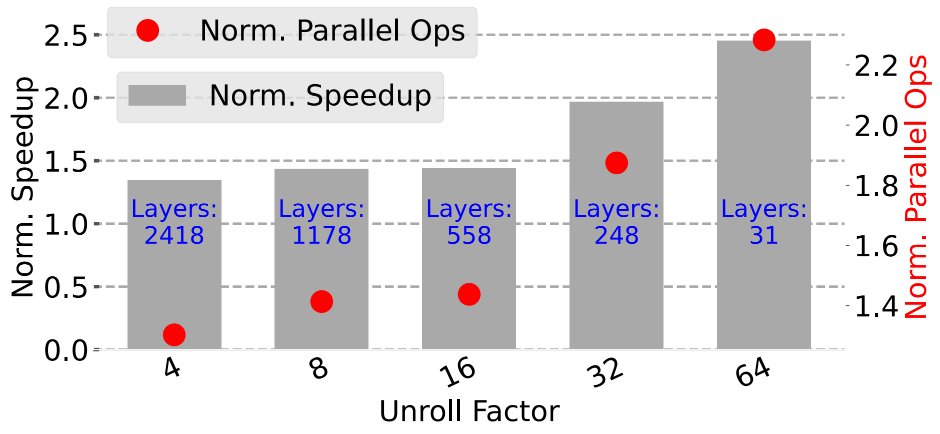


Figure 4.10: Shallow vs. Deep Layer graphs. Benchmark: Pathfinder. Left Y axis: Norm. Speedup. Right Y axis: Norm. parallelism/layer.

increases with wider layers (in shallow graphs), the performance gains of Tapeflow increase. Tapeflow improves gains by: i) removing tape accesses from the cache, ii) supplying parallel compute ops from the scratchpads, and iii) streaming data from the DRAM and eliminating cache pressure.

Chapter 5

Discussion

One of the challenges in automatic differentiation (AD) lies in the memory access, which directly impacts the performance of AD applications. These applications involve a substantial long-lived state that must be maintained in memory, hidden from the user. Consequently, this places additional pressure on the memory hierarchy, leading to increased cache conflicts and a higher demand for DRAM bandwidth.

To address these issues, various studies have been conducted in AD compilers, DNN training frameworks, and DNN accelerators, aiming to optimize memory accesses in AD. AD compiler techniques primarily focus on reducing the tape's size through checkpointing and recomputation heuristics. On the other hand, DNN frameworks and accelerators leverage their compile-time knowledge of the black-box operations to optimize the tape further or explicitly manage its movement throughout the memory hierarchy. Additionally, the decoupled accesses-execute paradigm has exhibited promising results in optimizing memory accesses for arbitrary programs by eliminating memory access latency from the critical path.

Our research aimed to conduct a comprehensive quantitative analysis of the tape and delve into its memory access behavior, specifically focusing on aspects such as temporal and spatial reuse. By gaining a deep understanding of these characteristics, we were able to propose a novel compiler, Tapeflow, that effectively utilizes the insights gleaned from our analysis. Through intelligent management and orchestration, our compiler optimizes the movement of the tape within the memory hierarchy, further enhancing performance and efficiency.

Our analysis revealed that while tape accesses may initially appear arbitrary and exhibit a mixed reuse pattern, the compiler can effectively manage them through static orchestration. Moreover, unlike DNN frameworks which only target a small set of operations, we can support arbitrary operations and data types including memory references. This is made possible by leveraging the compile-time information of the AD program. Based on the insights gained from our analysis, we devised a technique to reorder the tape in the DRAM, resulting in spatial proximity of tape accesses.

Our evaluation results show that Tapeflow outperforms Enzyme, a compiler that relies on the cache, by 1.3-2.5×, reduces on-chip SRAM usage by 5-40×, and saves 8× on-chip energy.

5.1 Limitations

Although Tapeflow significantly improves the performance and energy usage, there were several limitations to this study. We mention some limitations in this section:

5.1.1 Benchmarks

The first limitation was related to the benchmarks. There are no benchmark suits for measuring the performance of AD applications that have numerous memory accesses. Existing AD benchmarks challenge the generation of the gradient function rather than the runtime. Consequently, we implemented our benchmarks from the open source AD programs written in DSL languages such as DiffTaichi [17] and Halide [23].

5.1.2 Serial Execution of Layers

Currently, Tapeflow does not support executing multiple layers at the same time. This design decision was made to ease the streaming logic, and make sure the layers are consumed in REV the same order they were produced by FWD. However, the effect of other implementations that enable more degrees of parallelism should be explored.

5.2 Future Works

The main focus of Tapeflow was to explore the behavior of tape in general-purpose AD applications and enable the compiler to explicitly orchestrate the tape. By demonstrating the feasibility of a compiler-managed tape and its advantages over caches, there are promising directions for future work, particularly in the context of low-power devices and GPUs. These areas offer opportunities for further advancements in memory access optimization:

5.2.1 GPUs

Another promising area for future research is the application of our memory access optimization techniques to GPUs. GPUs are widely used in high-performance computing, machine learning, and scientific simulations. Exploring how our techniques can be extended or tailored to optimize memory accesses on GPUs could significantly enhance the performance and efficiency of GPU-accelerated applications. This may involve leveraging GPU-specific features such as shared memory, thread block organization, and memory hierarchy optimizations. Evaluating the impact of our approaches on GPU-based systems and quantifying the resulting performance gains would provide valuable insights.

5.2.2 Training on Edge

One avenue for future research is to explore the applicability of our memory access optimization techniques in low-power devices, such as mobile platforms, Internet of Things (IoT) devices, and embedded systems. These devices often have limited resources and stringent power constraints which require them to use smaller caches (or even no caches), making efficient memory access crucial for achieving optimal performance. Investigating how our approaches can be tailored and optimized specifically for training on edge devices. This could involve adapting the techniques to account for resource limitations, exploring power-aware tape reordering strategies, and evaluating the energy efficiency gains achieved through memory access optimization.

Bibliography

[1] Adept software library. `http://www.met.reading.ac.uk/clouds/adept`.

[2] Lenet-5 | Lenet-5 Architecture | Introduction to Lenet-5, March 2021.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, USA, November 2016. USENIX Association.

[4] Karsten Ahnert and Mario Mulansky. Solving ordinary differential equations in C++. *AIP Conference Proceedings*, 1389(1):1586–1589, September 2011.

[5] Mohammed AlQuraishi. End-to-End Differentiable Learning of Protein Structure. page 265231. bioRxiv, February 2018.

[6] Bradley M Bell. CppAD: A package for C++ algorithmic differentiation. *Computational Infrastructure for Operations Research*, 57(10), 2012.

[7] J Benson, R Cofell, C Frericks, Chen-Han Ho, V Govindaraju, T Nowatzki, and K Sankaralingam. Design, integration and implementation of the DySER hardware accelerator into OpenSPARC. *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12, 2012.

[8] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. ADIFOR-Generating Derivative Codes from Fortran Programs. *Scientific Programming*, 1(1):11–29, January 1992.

[9] Christian H Bischof, Lucas Roh, and Andrew J Mauer-Oats. ADIC: An extensible automatic differentiation tool for ANSI-C. *Software: Practice and Experience*, 27(12):1427–1456, 1997.

[10] 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.

[11] Tao Chen and G. Edward Suh. Efficient data supply for hardware accelerators with prefetching and access/execute decoupling. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[12] Jonas Degrave, Michiel Hermans, Joni Dambre, and Francis wyffels. A Differentiable Physics Engine for Deep Learning in Robotics. *Frontiers in Neurorobotics*, 13, 2019.

[13] Nikoli Dryden, Naoya Maruyama, Tom Benson, Tim Moon, Marc Snir, and Brian Van Essen. Improving strong-scaling of cnn training by exploiting finer-grained parallelism. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 210–220. IEEE, 2019.

[14] Jianwei Feng and Dong Huang. Optimal gradient checkpoint search for arbitrary computation graphs. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11433–11442, 2021.

[15] Audrūnas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. Memory-efficient backpropagation through time.

[16] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. Desc: Decoupled supply-compute communication management for heterogeneous architectures. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 191–203, 2015.

[17] Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. DiffTaichi: Differentiable programming for physical simulation. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*, 2020.

[18] Mike Innes, Alan Edelman, Keno Fischer, Chris Rackauckas, Elliot Saba, Viral B. Shah, and Will Tebbutt. A Differentiable Programming System to Bridge Machine Learning and Scientific Computing, July 2019. arXiv:1907.07587 [cs].

[19] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient Data Encoding for Deep Neural Network Training. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[20] Wenzel Jakob, Sébastien Speierer, Nicolas Roussel, and Delio Vicini. Dr.jit: A just-in-time compiler for differentiable rendering. *CoRR*, abs/2202.01284, 2022.

[21] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, October 2017.

[22] Yann LeCun. Deep learning est mort. vive differentiable programming!, 2018.

[23] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. Differentiable programming for image processing and deep learning in halide. 37(4):1–13.

[24] William S. Moses and Valentin Churavy. Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[25] William S. Moses, Valentin Churavy, Ludger Paehler, Jan Hückelheim, Sri Hari Krishna Narayanan, Michel Schanen, and Johannes Doerfert. Reverse-mode automatic differentiation and optimization of GPU kernels via enzyme. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, November 2021.

[26] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *PROC of the 40th MICRO*, 2007.

[27] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, October 2019.

[28] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, Huntsville Ontario Canada, October 2019. ACM.

[29] Uwe Naumann. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation.* SIAM, 2011.

[30] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. Stream-dataflow acceleration. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 416–429, 2017.

[31] Christopher Olah. Neural networks, types, and functional programming, 2015. `http://colah.github.io/posts/2015-09-NN-Types-FP/`.

[32] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. page 4.

[33] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. *CoRR*, abs/1912.01703, 2019.

[34] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. *CoRR*, abs/1912.01703, 2019.

[35] Barak A. Pearlmutter and Jeffrey Mark Siskind. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems*, (2), March.

[36] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W. Keckler, Christopher W. Fletcher, and Joel Emer. Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 137–151, Providence RI USA, April 2019. ACM.

[37] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based GPU Memory Management for Deep Learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[38] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A reconfigurable architecture for parallel patterns. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 389–402, June 2017.

[39] Christopher Rackauckas, Alan Edelman, Keno Fischer, Mike Innes, Elliot Saba, Viral B. Shah, and Will Tebbutt. Generalized physics-informed learning through language-wide differentiable programming. In Jonghyun Lee, Eric F. Darve, Peter K. Kitanidis, Matthew W. Farthing, and Tyler J. Hesser, editors, *Proceedings of the AAAI 2020 Spring Symposium on Combining Artificial Intelligence and Machine Learning with Physical Sciences, Stanford, CA, USA, March 23rd - to - 25th, 2020*, volume 2587 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2020.

[40] 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.

[41] Cristóbal Ramírez, César Alejandro Hernández, Oscar Palomar, Osman Unsal, Marco Antonio Ramírez, and Adrián Cristal. A risc-v simulator and benchmark suite for designing and evaluating vector architectures. *ACM Trans. Archit. Code Optim.*, 17(4), nov 2020.

[42] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li. Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 598–611. IEEE, 2021.

[43] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design, July 2016.

[44] Samuel Rogers, Joshua Slycord, Mohammadreza Baharani, and Hamed Tabkhi. gem5-salam: A system architecture for llvm-based accelerator modeling. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[45] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. In *Nature*, 1988.

[46] Amit Sabne. Xla : Compiling machine learning for peak performance, 2020.

[47] Frank Schäfer, Mohamed Tarek, Lyndon White, and Chris Rackauckas. Abstract-Differentiation.jl: Backend-Agnostic Differentiable Programming in Julia. Number arXiv:2109.12449. arXiv, February 2022.

[48] Ali Sedaghati, Milad Hakimi, Reza Hojabr, and Arrvindh Shriraman. X-cache: A modular architecture for domain-specific caches. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 396–409, New York, NY, USA, 2022. Association for Computing Machinery.

[49] Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, and Simon Peyton Jones. Efficient differentiable programming in a functional array-processing language. *Proceedings of the ACM on Programming Languages*, 3, July 2019.

[50] Rick Stevens, Valerie Taylor, Jeff Nichols, Arthur Barney Maccabe, Katherine Yelick, and David Brown. AI for Science Report | Argonne National Laboratory, 2019.

[51] Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch. OpenAD/F: A Modular Open-Source Tool for Automatic Differentiation of Fortran Codes. *ACM Transactions on Mathematical Software*, 34(4):1–36, July 2008.

[52] Andrea Walther, Andreas Kowarz, and Andreas Griewank. A Package for the Automatic Differentiation of Algorithms Written in C/C++.

[53] Nils Wandel, Michael Weinmann, and Reinhard Klein. Learning Incompressible Fluid Dynamics from Scratch - Towards Fast, Differentiable Fluid Models that Generalize. In *International Conference on Learning Representations*, March 2021.

[54] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic gpu memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 41–53, 2018.

[55] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, 2019.

[56] Xuechao Wei, Yun Liang, and Jason Cong. Overcoming data transfer bottlenecks in fpga-based dnn accelerators via layer conscious memory management. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2019.

Appendix A

Toolflow

We made Tapeflow's source code available to allow reproducibility. The code is available through the link below:

https://github.com/sfu-arch/TapeFlow.git

In the GitHub documentation, we explained how to build the tool and compile a program. In this chapter, we will show the input and outputs of our compiler. Moreover, we briefly show some abstractions added to the gem5-SALAM's `llvm-interface` so that it can understand the new instrumentation and convert them to the appropriate system instructions.

```
double function(double *x, double *y) {
    double u = 1;
    for (int i=0; i < N; i++) {
        double r = x[i] * y[i]*y[i];
        u *= -1.0 / (r + 0.001);
    }
    return u;
}

int main(int argc, char** argv) {
    // Allocate variables
    double *x, *y, *x_grad = ...;
    // Initialize the inputs
    // ...
    // Calculate the gradient.
    __enzyme_autodiff<double>(function, x, x_grad, enzyme_const, y) ;
}
```

An example of Tapeflow's input program.

A.1 Input and Output:

A.1.1 Input

The input to Tapeflow is a C/C++ program that contains a function to be differentiated. This function could be in the form of Figure ??. In line `16`, we pass the function to the Enzyme in order to generate the gradient function. `x_grad` will store the gradient of the function with respect to `x`. `enzyme_const` tells the compiler not to calculate the gradient for the variable `y`.

A.1.2 Compilation

To compile the function, we use the following commands:

```
clang++ -O3 input.cpp -emit-llvm -c -S -o input.ll
opt input.ll -load=Tapeflow.so -enzyme -enable-tf -spad-size=1024 -S -o output.ll
```

The first line lowers the input program to LLVM-IR. The second line invokes Enzyme to generate the gradient function and enables Tapeflow for optimizing the tape. Here is a short description of the flags given to the `opt` command.

- **`-load:`** The address of the Tapeflow's dynamic library.
- **`-enzyme:`** Invokes Enzyme to generate the gradient function.
- **`-enable-tf:`** Enables Tapeflow's compiler passes, described in Chapter 3.2.
- **`-spad-size:`** Determines the size of the scratchpad in bytes. It is employed to create layers.

A.1.3 Output

Figure ?? illustrates the output of Tapeflow. Note that the code is trimmed for readability purposes. Lines `4` and `13` contain `!SpadAlloc` indicating the start of a layer in FWD and REV, respectively. Lines `8` and `18` indicate the end of a layer using `!Barrier`. The tape value, `u`, is stored to the tape during FWD in line `6` and loaded from the tape during REV in line `15`. We mark the FWD store with a `!write` metadata and the load with a `!read` metadata. Moreover, both `write` and `read` are assigned the same relative index, `0`. in gem5-SALAM, we use this index to generate the scratchpad address.

A.2 Simulator Description

We used gem5-SALAM to simulate the underlying hardware to run Tapeflow. Here is the list of the parts modeled and simulated by gem5-SALAM:

```
define void @diffe_function(double* %x, double* %"x'", double* %y, double ↩
    %differeturn) {
for.body:
  %iv = phi i64 [ %iv.next, %for.body ], [ 0, %entry ]
  %0 = alloca i32, align 4, !SpadAlloc 1, !size 1
  %1 = getelementptr inbounds double, double* %u.017_malloccache, i64 %iv
  store double %u.017, double* %1, align 8, !write 0
  ...
  %4 = alloca i32, align 4, !size 1, !Barrier 1
  br i1 %exitcond, label %for.cond label %for.body

invertfor.body:
  iv' = phi i64 [ 4, %inv_bb1 ], [ %26, %inv_bb2 ]
  %6 = alloca i32, align 4, !SpadAlloc 1, !size 1
  %7 = getelementptr inbounds double, double* %u.017_malloccache, i64 iv'
  %8 = load double, double* %7, align 8, !read 0

  ...
  %25 = alloca i32, align 4, !size 1, !Barrier 1
  ...
}
```

Tapeflow's output program.

- **CGRA:** The CGRA runs the instructions. It is modeled using the accelerator cluster in gem5-SALAM. We can change the number of functional units in the CGRA and their latencies.
- **Dependencies:** The dependencies between instructions are modeled by creating a dataflow graph.
- **Scheduling:** gem5-SALAM contains a scheduling phase in which it launches all the instructions that are ready to execute. The resources that describe the hardware, such as the number of ALUs, ports, etc., determine the number of instructions that can run at each cycle. We do not assign latency or penalty to the scheduling phase.
- **Memory hierarchy and interconnects:** gem5-SALAM models different components in the memory hierarchy, such as cache and scratchpad, and how they are connected to the other components of the systems. For instance, we can dedicate more ports to the scratchpad so that we have fewer serializations over the ports.
- **DMA and Stream Engines:** gem5-SALAM models DMA devices such as *Stream DMA* and *Non-coherent DMA* that move values between the DRAM and scratchpad.

To create the dataflow of the program, gem5-SALAM uses an LLVM interface that maps LLVM instructions to SALAM instructions and connects them together. We exploited this feature to add extra instructions to SALAM and create a back-end for our compiler. It caputres Tapeflow's new instructions, such as `Tload` and `SpadAlloc`, and execute them based on our policies. Moreover, it allows us to add static and dynamic dependencies between instructions. This way, we can make sure the precedence between instructions is correct. For example, a `TStore` must be dependent to all its previous `SpadAlloc` instructions to make sure the scratchpad is ready and contains the required values.

```cpp
SALAM::Instruction *createInstruction(llvm::Instruction *inst) {
  if (inst->hasMetadata("Barrier")) {
    return SALAM::createBarrierInst(id, this, debug(), OpCode,
                                    hw->cycle_counts->load_inst,
                                    functional_unit);
  } else if (inst->hasMetadata("SpadAlloc")) {
    return SALAM::createSpadAllocInst(id, this, debug(), OpCode,
                                      hw->cycle_counts->load_inst,
                                      functional_unit);
  }
}

void SpadAllocInst::initialize(Value *irval, irvmap *irmap, valueListTy *↩
    valueList) {
    llvm::Instruction *inst = llvm::dyn_cast<llvm::Instruction>(irval);
    auto *N = inst->getMetadata("size");
    auto *S = llvm::dyn_cast<llvm::MDString>(N->getOperand(0));
    alloc_size_ = std::stoi(S->getString().str());
    SALAM::Instruction::initialize(irval, irmap, valueList);
}

void handleSpadAllocInst(std::shared_ptr<SALAM::SpadAllocInst> inst) {
  if (MemoryRequest *mem_req = SpadAlloc(inst->getAllocSize())) {
    stream(SPM, mem_req->getAddress(), mem_req->getLength());
  }
}

MemoryRequest SpadAlloc(size_t request_size) {
  size_t region_size = end_ - start_;
  MemoryRequest *req = nullptr;
  switch (phase_) {
    case FWD:
      if (region_size + request_size >= spad_limit_) {
        req = new MemoryRequest(tape_ptr_, buffer(), region_size);
        tape_ptr_ += region_size;
        end_ = start_;
      }
      head_ = end_;
      end_ += request_size;
      break;
    case REV:
      ...
      break;
  }
  return req;
}
```

SpadAlloc Instruction in gem5-SALAM.

A.2.1 Modifications to gem5-SALAM

gem5-SALAM takes an LLVM-IR as an input, parses it, creates corresponding SALAM instructions for each LLVM instruction, builds the dataflow, schedules the instructions, and executes the graph. We list our modifications to each step:

- Creating SALAM instruction: In this step, we modified this function to detect and create the Tapeflow's new instructions.
- Building dataflow: This step requires specifying the dependencies between the instructions. In this step, we add the dependencies introduced by Tapeflow to guarantee the correct order of execution.
- Execution: For each Tapeflow instruction, we added a handler that is called during the execution phase. The handler makes sure the instruction is executed based on our policies. For example, the `TLoad`s only read values from the local address space that belongs to the scratchpad.

Figure **??** shows one of our new SALAM instructions, `SpadAlloc`, and the way we handle it during execution.

In the `initialize` function, we extract the size for the allocation. In the `SpadAlloc`, we check if we need to move the scratchpad values to the DRAM to make room for the next values. In case we require moving them, we issue the `stream` command that invokes the stream engine to move the values.

Here is the list of all instructions added to gem5-SALAM:

- **`TStore %address, %data:`** Stores the data to the scratchpad. The difference between a `TStore` and a normal store is that when executed, it sends the value to the address space reserved for the scratchpad, while a normal store uses the global address space to store the data.
- **`TLoad %address:`** Same as `TStore` but used for reading values from scratchpad.
- **`SpadAlloc %size:`** Allocates a buffer in the scratchpad to keep the upcoming layer. The buffer is created by updating the scratchpad's state variables, `stard` and `end`. If the scratchpad is full, we trigger a stream to the DRAM before executing the layer. The logic is shown in Figure **??**.
- **`Barrier:`** Synchronizes the layers. It makes sure the layers are finished before the next layers start.