TapeFlow: Streaming Gradient Tapes in Automatic Differentiation

Milad Hakimi
School of Computing Sciences
Simon Fraser University
Burnaby, Canada
milad_hakimi@sfu.ca

Arrvindh Shriraman
School of Computing Sciences
Simon Fraser University
Burnaby, Canada
ashriram@sfu.ca

2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) | 979-8-3503-9509-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/CGO57630.2024.10444805

***Abstract*—Computing gradients is a crucial task in many domains, including machine learning, physics simulations, and scientific computing. Automatic differentiation (AD) computes gradients for arbitrary imperative code. In reverse mode AD, an auxiliary structure, the tape, is used to transfer intermediary values required for gradient computation. The challenge is how to organize the tape in the memory hierarchy since it has a high reuse distance, lacks temporal locality, and inflates working set by 2 — 4×.**

We introduce Tapeflow, a compiler framework to orchestrate and manage the gradient tape. We make three key contributions. i) We introduce the concept of *regions*, which transforms the tape layout into an array-of-structs format to improve spatial reuse. ii) We schedule the execution into *layers* and explicitly orchestrate the tape operands using a scratchpad. This reduces the required cache size and on-chip energy. iii) Finally, we stream the tape from the DRAM by organizing it into a FIFO of tiles. The tape operands arrive just-in-time for each layer. Tapeflow, running on the same hardware, 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. We demonstrate Tapeflow on a wide range of algorithms written in general-purpose language.

***Index Terms*—Automatic differentiation, Gradients, Streaming Algorithms, Back propagation**

Deep Learning has outlived its usefulness. Long live Differentiable Programming! – Yann LeCun [19]

I. Introduction

Automatic differentiation (AD) is a generalization of back-propagation in deep learning [38]. AD supports learning in arbitrary functions, instead of being restricted to blackbox DNN operators [19], [27], [41]. This enables control-flow-driven, dynamic, data-dependent learning models [3], [10], [28], [32], [39], [41].

AD reduces the amount of training data required for learning. Consequently, the community has been active in supporting gradient calculations in data structures [8], [42], C++ operators [4], [6], [9], [25], functional [30], [40], and imperative code [3], [21], [22], [43], [15]. However, they do not characterize the performance challenges in 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 illustrates the tape in a function generated by Enzyme [21], a state-of-the-art AD compiler. The gradient calculation involves two phases. A forward phase (FWD) is the original function for which

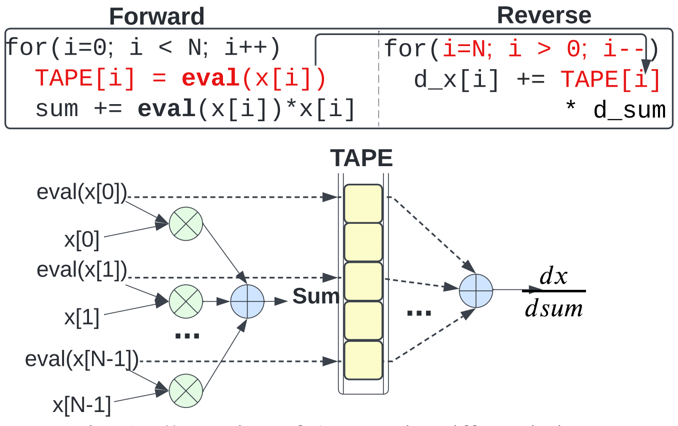


Fig. 1: Illustration of Automatic Differentiation

we wish to calculate gradients, and a reverse phase (REV) computes the gradients. During the 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. Currently, Enzyme focuses on the functionality of the tape but does not study the performance limitations introduced by tape layout and access pattern.

The tape introduces several challenges: i) **Long lifetimes:** The tape values cannot be consumed by the REV until the FWD is complete, requiring the tape to be buffered throughout the FWD. ii) **Large size of the tape:** The tape must store all the SSA state, which grows proportional to the number of operations in the FWD. iii) **Mixed reuse behavior of tape operands:** Since the gradient calculation inverts the REV's dataflow, tape values produced earlier in FWD are consumed later in REV. This disrupts the temporal locality.

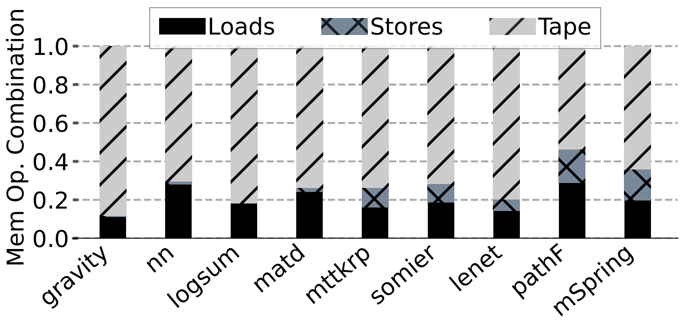


Fig. 2: State Distribution in Automatic Differentiation.

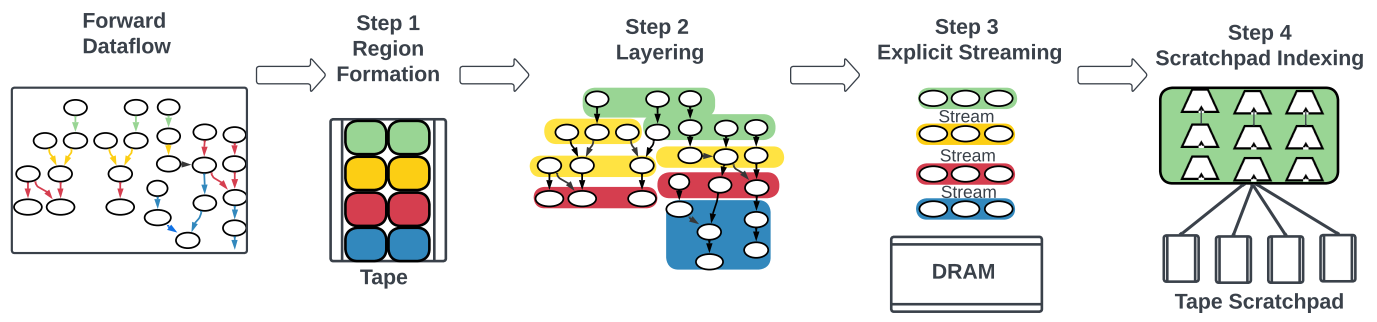


Fig. 3: Overview of Tapeflow. Step 0: the input of the Tapeflow is the baseline dataflow. Step 1: Changing the tape layout. Step 2: The compiler splits the dataflow into layers. Step 3: Stream the tape from DRAM to layers. Step 4: statically schedule the execution of the FWD and REV layers.

iv) **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 (see Section II-A).

In Figure 2, we demonstrate the problem of the tape by analyzing a gradient function generated by Enzyme [21]. The tape increases the intermediate state by 2–4×. This makes it challenging for caches and on-chip hierarchies to 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. 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.

Current AD compilers do not optimize the tape. Tapeflow addresses the challenge of how to create high-performance tapes in AD. Our approach includes two parts: 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 (not unlike DNNs). We exploit this information to pipeline the tape segments and make it amenable to streaming.

Tapeflow incorporates multiple passes for optimizing the tape into an explicit stream (Figure 3). i) **Regions:** First, Tapeflow splits the tape into coarse-grain regions based on def-use lifetime. Each region is organized in an array-of-structs layout and tape values with similar life are collocated. This improves spatial efficiency and utilization. ii) **Layers:** Tapeflow schedules the FWD and REV into sub-dataflows, and constrains each layer to access tape operands in a tiled fashion. Unlike DNNs, Tapeflow's layers are compiler generated and is generalizable to arbitrary program graphs. iii) Finally, Tapeflow explicitly orchestrates the tape, and the compiler introduces **stream operations** at layer boundaries to stream data from the DRAM with few registers (32—64).

Tapeflow is built as an extension to Enzyme [21], which is 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 [37]. Our 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 Managment** We present 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×, reduces on-chip SRAM between 5 — 40 ×, and saves 8 × on-chip energy.

II. Motivation and Scope

A. *Overview of Automatic Differentiation*

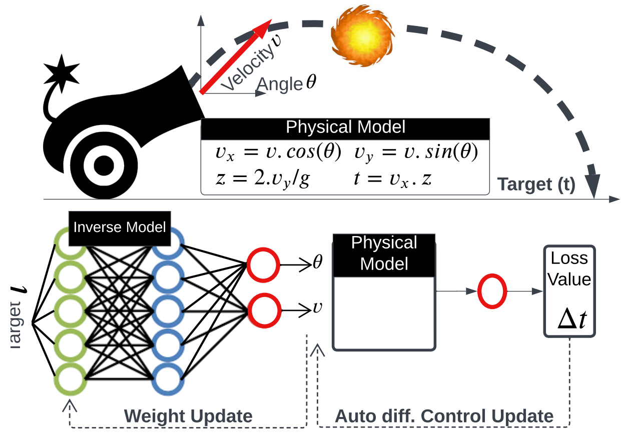
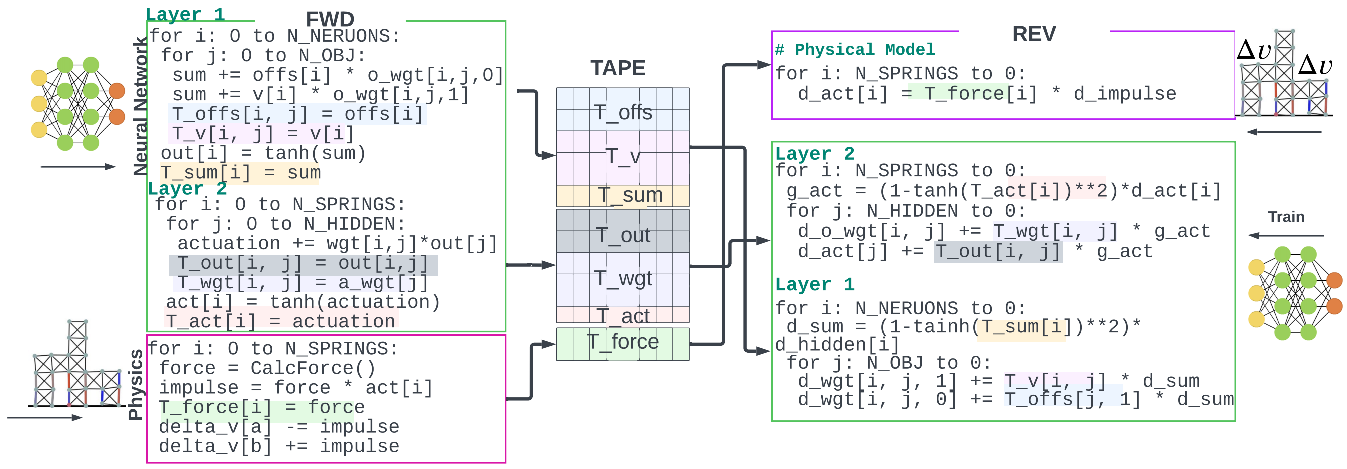


Fig. 4: Illustration of Automatic Differentiation.

Currently, AD targets inverse problems such as protein folding [5], robotics [10] and physics simulations [16], [44].



```
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
```

Fig. 5: Illustration of Mass Spring example. The tape state is color-coded to indicate producer-consumer dependency.

We illustrate this concept with a simple example in Figure 4, which depicts a cannon trying to hit a target. 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). This unknown function f^{-1} can be approximated using a neural network, but it is challenging to train because there is no clear way to measure the loss for the output variables θ and v. To solve this, we feed the neural net's outputs, θ and v , to the physics model to produce a target (t). During training, automatic differentiation deals with back-propagation through the physics model written in imperative code.

B. *How do current AD compilers organize the tape?*

Observation 1: *Tape management needs to support general program graphs that may employ imperfect loops, if-else, and array indexing.*

Figure 5 shows Enzyme's [21], the state-of-the-art AD compiler, gradient function for a simple particle system [16]. he system has three arrays for each particle: 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 i.e., a struct-of-arrays layout. However, this 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 decisions regarding tape movement onto the chip or reuse from DRAM or schedule operand execution. Consequently, Enzyme relies on the cache to handle tape orchestration. However, long-term reuse tape accesses conflict with short-term reuse non-tape accesses, resulting in cache conflicts and overflows.

Observation 2: *The biggest challenge in generating fast REV code is decomposing memory operations.*

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 and the other from the REV. For example, the last statement in REV uses offset[i][0] from FWD and d_sum from REV (REV:Figure 5).

C. *Characterizing Tape Accesses*

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 original FWD. See Figure 6.*

Observation 1.2: *Tape lifetime is ≃100 × longer..*

Architecture Implication: *The tape will overflow both explicit (scratchpad) and implicit (cache) hierarchies.*

The number of tape values is proportional to the operations in the program. The lifetime and reuse distance are high for two reasons: i) Tape values need to wait for the FWD to finish before they get fully consumed. Figure 5 shows the consumption of the color-coded tape values using arrows. They all are consumed in the REV after the FWD finishes. ii) The consumption order of the tape values is the reverse of their production order. In Figure 5, the tape entry offset[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.

Observation 2.1: *Tape entries can be partitioned into groups with similar lifetime (Figure 7).* **Architecture Implication:** Tape can be implemented using streams.

Observation 2.2: *Tape values produced together in the FWD are consumed together in the REV.* **Architecture Implication:** Tape should be organized as array-of-structs. Elements consumed together should be packed together.

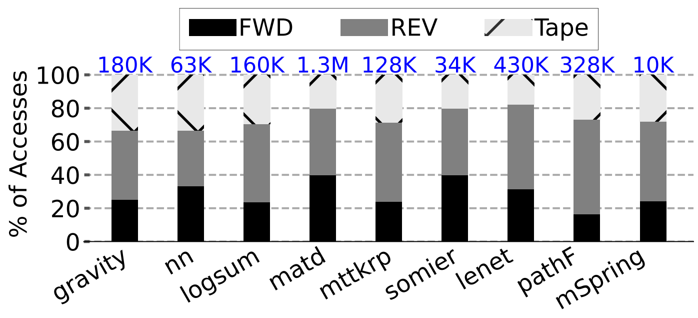


Fig. 6: The distribution of FWD, REV, and TAPE edges in the dataflow. Numbers indicate working set.

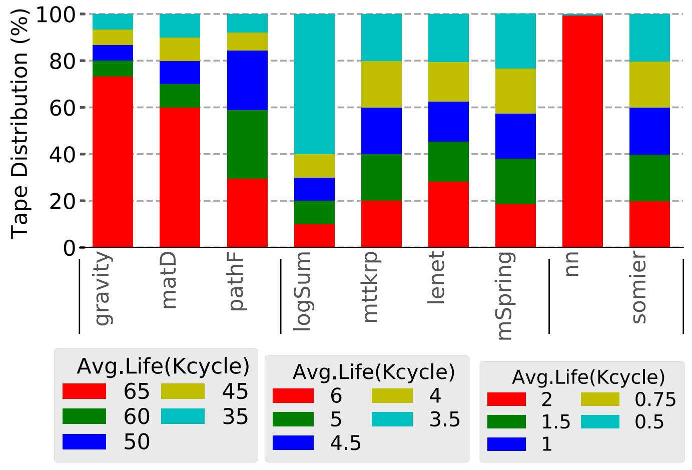


Fig. 7: Tape lifetime distribution. Benchmark cluster: Similar lifetime (in Kcycles). Y-axis: Distribution of edges.

We divided the tape values into 5-quantiles based on their lifetimes and represented them in Figure 7. The height of each stack represents the tape entries within a particular quantile, while their color denotes their lifetime (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. In Figure 5, layer-1's first iterations produce values with the longest lifetime mapped to the red bar in Figure 7. The tape entries produced by the physical model (`force`), are mapped to the blue bar (low lifetime). FWD:NN layer-1 (Figure 5) writes 2 values to the tape in each iteration, highlighted with red. 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.

D. Tapeflow vs SOTA Gradient Frameworks.

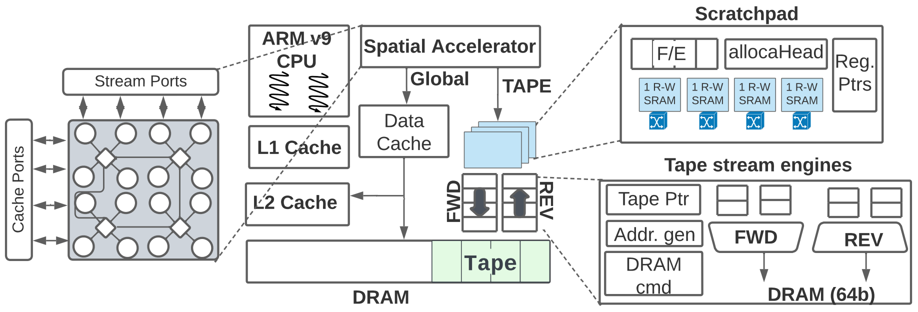
While generalized compiler-driven AD has been recently popularized by Enzyme, there has been extensive prior work on gradient-based computation. The most notable are DNN training (vDNN [36], Gist [17]), DSLs targeting machine learning (e.g., Tensorflow [3], DiffTaichi [16]), and differentiable libraries (e.g., C++ Adept [1]). Table I highlights the limitations and features of each approach. The target section lists properties essential for supporting gradient calculation in imperative code, and the design section lists desirable features. DNN frameworks typically have one tape per layer output, and the tape structure mirrors the underlying model graph. 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. Further, they manually optimize the tape for each operator and only support blackbox operators with regular affine loop structures (e.g., convolutions). Library-based approaches rely on types to manage the tape. Still, the programmer must tediously rewrite the model using the library's API, and the required operators need to be built into the library. Most recent updates to libraries such as Clad [43] invoke enzyme in the back-end[1]. As we operate with Intermediate Representation (IR) post-enzyme and already facilitate this workflow, the fusion with Tapeflow stands to be seamless. For some specific features, where Enzyme support is not available, Clad uses clang source-to-source translation (e.g., C++ classes) and it can invoke the Tapeflow's methods as a library. Differential libraries and DSLs manage the tape as a stack of operands, limiting parallelism and bandwidth utilization. DNN training only supports a limited set of model graphs. Enzyme does not orchestrate the data from the DRAM and relies on caches.

The most closely related works to Tapeflow are pipelined or distributed DNNs [11], [24], [31]. They only support feed-

[1]https://clad.readthedocs.io/en/latest/user/UsingEnzymeWithinClad.html

TABLE I: TAPEFLOW VS SOTA FRAMEWORKS

		DNN Training		Other DSLs	Diff. Library	AD Compilers	
		vDNN [36],Gist [17]	TensorFlow [3],PyTorch [29]	DiffTaichi [16], Halide [20]	Adept [1]	Enzyme [21]	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



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

Fig. 8: 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.

forward static graphs with DNN operators (e.g., convolution, LSTM). Tapeflow targets arbitrary imperative functions in automatic differentiation. Further, Tapeflow needs to calculate gradients for arbitrary memory references in the function (not just tensors.). Research in the DNN space has sought to optimize training [13], [45], [46]. They either develop model-specific implementations or allow only a composition of DNN layers. Another line of related work partitions the tensors in DNNs across heterogeneous locations [31], [35], [36], between DRAM and FPGA [47], and across multiple GPUs [33]. In AD, we work with a low-level compiler IR since generalizing the differentiation requires us to work with low-level operands. Thus, data is not expressed in terms of tensors, but scalars and pointers. It is not feasible to apply prior techniques that rely on data shape. Output re-computation [12], [13] discards some outputs when the memory is insufficient, and re-computes them when required. This approach incurs performance degradation, relative to approaches that do not perform recomputation [36]. Tapeflow builds on Enzyme-optimized IR; it already applies recomputation where feasible.

III. Target Hardware for Tapeflow Compiler

In this paper, we target a generic spatial architecture [14], [26]. The compiler works on post-optimized IR with loop unrolling, resulting in a high degree of operation-level parallelism on floating-point data types. Spatial accelerators support a high degree of parallelism and explicit scheduling of the compute operations and memory accesses. The spatial accelerator allows us to evaluate the limits of our compiler passes without being limited by the architecture. We, first, review the compute tile. The datapath consists of a grid of PEs onto which the FWD or REV's dataflow is mapped. The accelerator is a loosely coupled CGRA with 4×4 double functional units similar in design to Dyser [7]. We use doubles since the baseline Enzyme compiler uses doubles, and low-precision floats in AD are an open-research topic. We use a CGRA mapping pass that schedules the LLVM IR operations onto the grid of function units. Each functional unit in the grid maps a single instruction from the dataflow graph of the FWD and REV. The data dependencies between the operations are explicitly routed over a static mesh operand network. The mapper builds on prior work [7].

Tapeflow adopts a partitioned memory model facilitated by statically analyzable tape accesses. Tape accesses are orchestrated by Tapeflow 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 reads and writes are partitioned in time by a barrier, which allows for the 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.

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.

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.

IV. Tapeflow: Compiler Support for Tape

In this section, we describe Tapeflow passes and their ability to transform the tape layout and orchestrate the DRAM transfers. Figure 9 illustrates using a convolution example; we would like to emphasize that Tapeflow can support arbitrary code. The inputs are `img`, and `fil`, and we calculate the derivative of `res` with respect to `fil`.

A. 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.*
Input: Unorganized Tape **Output:** Array-of-Structs Tape.

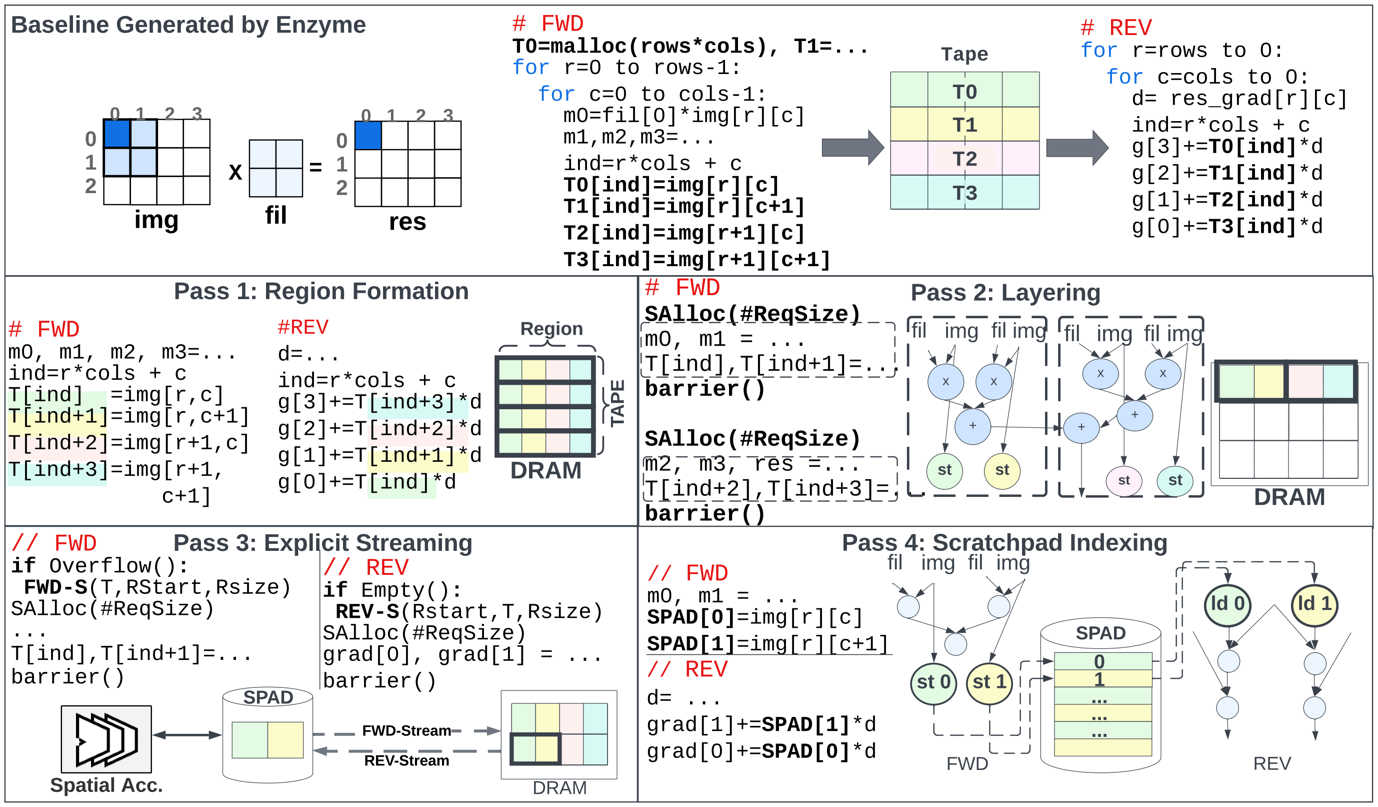
We rewrite the tape into an array-of-structs layout shown in Figure 9: Pass1. For each tape operand, we have to assign a slot in the tape. The question is which slot to choose? Tapeflow chooses the slot based on spatial use within the REV's dataflow. Tapeflow merges multiple tape arrays, and assigns 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 9 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 4 and 7). Second, we calculate the offset within the struct based on the temporal order (lines 10 and 13). This is feasible because the tape is a compiler introduced data structure and all FWD-REV dependencies are known.

B. 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. Layers are a partition of dataflow that is constrained to source operands from a single tape region. Further, a constraint is set on the size of each region to a fixed limit. Each layer starts with `SAlloc`, which allocates space for the tape region, and terminates with a barrier (Figure 9 Pass2). The barrier forces the layer to access tape only within the region. Algorithm 2 illustrates



```
# FWD
T0=malloc(rows*cols), T1=...
for r=0 to rows-1:
  for c=0 to cols-1:
    m0=fil[0]*img[r][c]
    m1,m2,m3=...
    ind=r*cols + c
    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
    g[3]+=T0[ind]*d
    g[2]+=T1[ind]*d
    g[1]+=T2[ind]*d
    g[0]+=T3[ind]*d
```

```
# 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
```

Fig. 9: Overview of Tapeflow Compiler Passes. Pass1: Region Formation and Array-of-Structs Layout. Pass2: Layering and Tiling. Pass3: Explicit Streaming. Pass4: Scratchpad Indexing.

how Tapeflow populates `Layers`, a map that contains the set of layers for each loop. In the FWD the layers execute in the FIFO order, and their order is inverted in the REV. 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 9), we terminate it and start a new one. We use an auxiliary data structure, `FtoR`, that maps FWD operations to gradient operations in the REV.

```
Algorithm 1: Create AoS layout
Input: FWDLoops: Set // Loops in FWD
Input: FtoR: Map // Dictionary of gradient ops in REV
foreach loop ∈ FWDLoops do
  RSize: // Size of loop's region.
  FWDBase←loop.Tape + loop.Index × RSize
  // Region's base address.
  REVLoop ← FtoR[loop]
  REVBase←loop.Tape+ REVLoop.Index×RSize
  foreach TStore ∈ loop.Ops do
    TStore.dst←FWDSBase + offset
    // Find tape load in REV.
    TLoad←FtoR[TStore]
    TLoad.src←REVSBase + offset
    offset←offset + TStore.size
```

```
Algorithm 2: Layer Generation
Input: FWDLoops: Set // Loops in FWD
Input: FtoR: Map // Dictionary of gradient ops
1  foreach Loop ∈ 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    foreach Op ∈ Loop.Ops do
7      if op is TStore :
8        if Count > SPAD.size :
9          // Terminate layer
10         Layers[loop]←{FWDOps}
11         Layers[REVLoop]←{REVOps}
12         FWDOps,REVOps←[], Count←0
13       Count←Count + 1
14     FWDOps←{Op}
15     REVOps←{FtoR[Op]}
```

C. 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 concatenate it to the tape in the DRAM (Figure 9:Pass3). A challenge that arises is how to coordinate streaming with the REV's execution since only the FWD is aware when tape is spilled to the DRAM. To solve this, we record the parameters of the `FWD-stream` and insert streams at the head of each layer in the REV.

D. 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 the 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 combined with the scratchpad base address to obtain the effective address for the tape operand (Figure 9:Pass 4, `T[ind]` is replaced by `SPAD[0]`). Algorithm 3 lists the pseudo code.

```
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
foreach Loop ∈ FWDLoops do
  foreach Layer ∈ Layers[Loop] do
    offset←0
    foreach TStore ∈ Layer.Ops do
      // Set the dest of TStore in SPAD.
      TStore.dst← Layer.SAlloc + offset
      // Update the source of the
      corresponding tape load.
      TLoad ← FtoR[TStore]
      RevLayer ←FtoR[layer]
      TLoad.src←RevLayer.SAlloc + offset
      offset←offset + TStore.size
```

V. Methodology and Setup

In order to evaluate Tapeflow, we use a set of benchmarks that were derived from physics models [21], scientific computing, and tensor compilers [18]. Table II lists these benchmarks and categorizes them into two groups, regular and irregular, based on the size of SSA state and tape. The regular workloads (gravity, nn, logsum, matdescent) have high arithmetic intensity and simple strided access, and typically have only one or two input arrays. In MTTKRP, Somier, Lenet, and PathFinder, the tape imposes pressure on the caches leading to increased conflicts between the tape and REV. For each benchmark, we generate the compiler IR using the Enzyme compiler [21]. For each target function `f()`, Enzyme generates a gradient function `grad_f`. We use the $-O3$ and $-mem2reg$ optimizations to aggressively optimize the

TABLE II: BENCHMARKS

Name	*Regular*					
	tensor/ loop	Work. Set(K)	Inp. (K)	Suite	input params	layer count
Gravity	2	180	8	Diff [16]	Array: 512	15616
NN	2.6	63	80	Enz. [21]	Img: 28 x 28	490
Logsum	2	160	160	Enz. [21]	Input: 10K	1251
Matd.	3.5	1280	2400	Enz. [21]	M,N: 400	10400
	Irregular					
MTTK.	7.	104	128	Taco [18]	8x8x8	512
Somier	8.	34	48	RV [34]	8x8x8	216
Lenet-5	4.	430	10	Lenet [2]	-	11895
Pathf.	4.	328	164	RV [34]	R:128, C:256	5842
Mass Spring	6.	10	28	Diff [16]	Obj:128, hidden:32	46000

memory footprint of the benchmark. This is our baseline, `Enzyme`. Tapeflow works with the optimized IR generated by enzyme. Enzyme optimizes tape accesses in favor of recomputation and checkpointing where feasible.

In our experiments, both Enzyme and TapeFlow were evaluated on the same hardware platform to ensure a fair and direct comparison. The hardware setup used for the experiments provided an equivalent environment for both Enzyme and TapeFlow, thus eliminating any potential bias due to differences in hardware configurations. We simulated the complete accelerator (including OOO core), in detail, using gem5-salam [37]. Table III lists the hardware configurations. We evaluate the following systems.

- $Enzyme_{32k--128k}$: 32k: 4-way. 128k:8way. The tape 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.
- $Tflow_{32k}$ (*ISO-Energy* setup): In Tapeflow, tape accesses always use the scratchpad (64 entry, 8 bank in baseline plots). The cache in Tapeflow is used by non-tape accesses.
- $Tflow_{2k}$ (*ISO-Perform* setup): 2k, 2-way. Since Tapeflow eliminates the tape accesses from the cache, we study how small a cache can suffice for high performance.
- $Tflow_{1k}$ **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.

VI. EVALUATION

- How much can Tapeflow improve performance relative to Enzyme [21]? **Answer:** 2.4×. § VI-A
- How much DRAM bandwidth can Tapeflow save by streaming ? **Answer:** 14×. § VI-B
- What is the impact of Tapeflows array-of-struct layout, in isolation, on DRAM bandwidth? **Answer:** up to 30% reduction. § VI-B0a
- How much on-chip energy and cache size can Tapeflow save ? **Answer:** 8×. § VI-C
- How much can Tapeflow shrink the cache size while maintaining the same performance. **Answer:** 4-8×. § VI-D?

A. Performance Evaluation

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. 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 inner most 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 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 10 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.

B. DRAM Bandwidth

Figure 12 shows the normalized DRAM access. All bars are normalized to the $Enzyme_{32k}$ cache. We plot the DRAM traffic for different cache sizes ($Cache_{1k}$ to $Cache_{128k}$) (Table

TABLE III: SYSTEM CONFIGURATION

gem5-salam config									
CPU					Memory		Scratchpad		
OOO ARM v9 2GHz, fetch width:8, issue width: 8					DDR4 4GB 19.2 GB/s		512B, 8 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

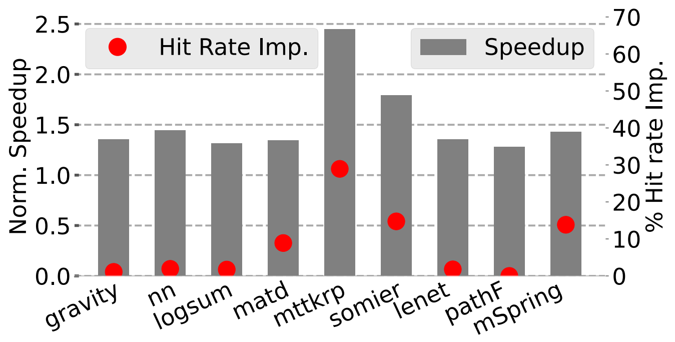


Fig. 10: **REV speedup.** Right Y-axis: Hit rate improvement (higher is better)

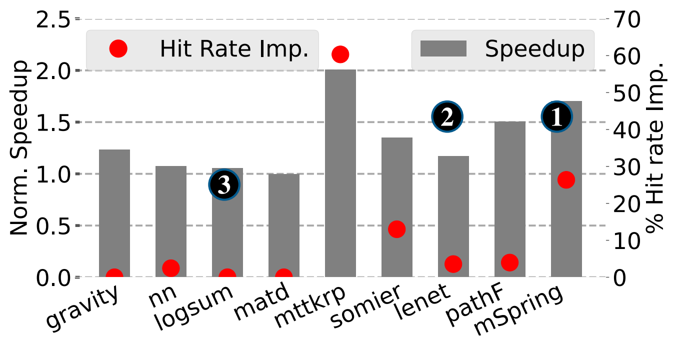


Fig. 11: **FWD speedup.** Right Y-axis: Hit rate improvement (higher is better)

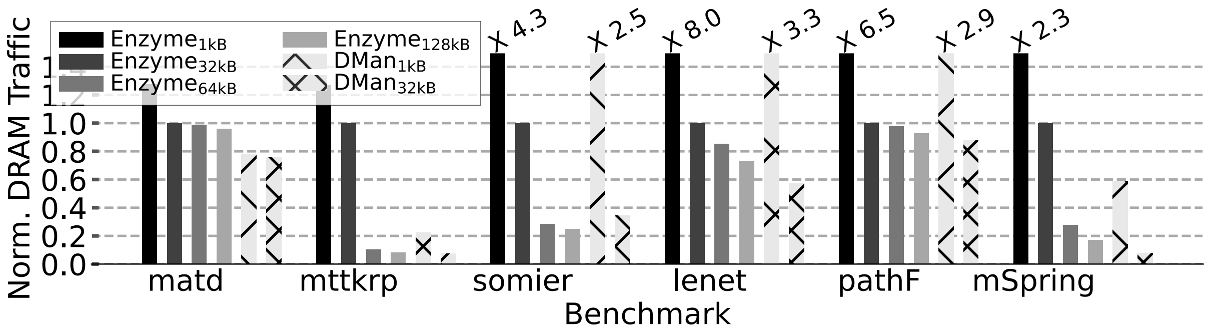


Fig. 12: Normalized DRAM access (lower is better). 1 = $Enzyme_{32k}$

III). We also compare the effect of cache size reduction by plotting the $Tflow_{1k}$. Tapeflow reduces the conflict misses by separating the tape accesses from non-tape accesses and managing them using small and power-efficient stream buffers. Regular benchmarks are not affected by cache size in both enzyme and Tapeflow (e.g., *Matd*). Tapeflow generates less DRAM traffic than Enzyme, thanks to partitioning tape accesses away from caches, which eliminates conflict misses. Irregular benchmarks experience significant reduction due to extensive interference between tape accesses and non-tape accesses (e.g., 14$\times$ improvement in *MTTKRP*). *Lenet* and *Pathf*'s primary working sets fit in 32k cache, resulting in marginal improvement, but a 2.3$\times$ improvement is observed in the 1k settings ($Enzyme_{1k}$ and $Tflow_{1k}$). Regular workloads with minimal tape traffic experience minimal improvement, such as 1.3$\times$ in *Matdescent*, which has a streaming access pattern to both input arrays and the tape and does not have many conflict misses.

a) Impact of Tapeflow's Struct-of-Arrays layout: Figure 13 shows the impact of pass1 that changes the layout to array-of-structs. We use the cache for both enzyme and Tapeflow to isolate the impact of layout alone. We still see improvement for two reasons: 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.

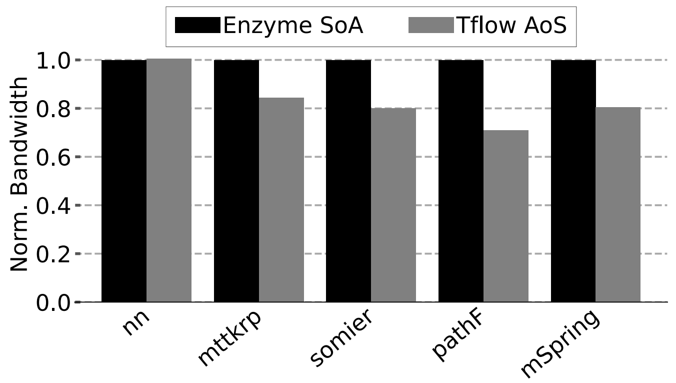


Fig. 13: Enzyme's vs. Tapeflow's Layout (lower is better).

C. On-Chip Energy

Result: *Tapeflow reduces the on-chip energy by up to* 8.2$\times$ *by streaming the tape accesses. Each tape access consumes less than 10 pJ.*

Figure 14 shows the energy improvement of Tapeflow over Enzyme. The *ISO-Perform* illustrates the smallest cache that matches the performance of the $Enzyme_{32k}$. We use Cacti [23] for modeling cache energy. For 32k and 64k caches, we model a dual port, 4-way. For smaller caches, we model a 2-way cache. Tapeflow exhibits better energy efficiency: i) Tapeflow partitions out the tape accesses from the cache and reduces cache pressure. Thus, we can employ a smaller cache; a 2k cache is sufficient for non-tape accesses. The 2k setup requires 6.8$\times$ less energy per access. Streaming the tape

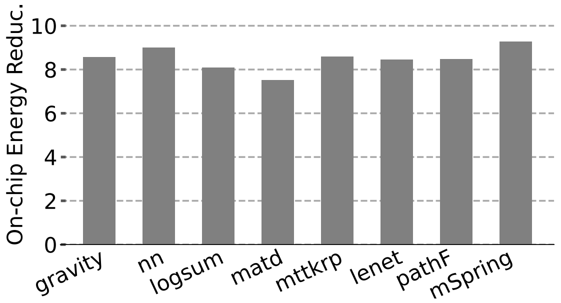


Fig. 14: On-chip Energy Reduction. (Higher is better)

allows the *ISO-Perform* to remain performance competitive with large caches, $Enzyme_{32k}$. ii) Tapeflow directs tape accesses to the scratchpad. The scratchpad is accessed using indexes (like registers) with no misses. Per tape access, they exhibit 30× less energy than the cache (10pJ vs 300pJ).

D. Energy-Performance Trade-off

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

Figure 15 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 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. Tapeflow redirects tape accesses to stream buffers that have low per-access energy.

VII. Design Exploration

A. Scratchpad Size

Result: *Increasing the scratchpad size from 64 kB to 1 MB, increases the performance of the Tapeflow between 25-50%.*

Figure 16 demonstrates the impact of scratchpad size. The speedup is normalized to $Enzyme_{32k}$. 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 really large scratchpads (1MB) the eventual constraint is parallel ops within layer. ii) Fewer `Sync` operations: When the layers are too small (limited by 64byte scratchpad), the overhead of Sync, limits parallelism and performance. This is visible in nn_{64B}; performance is same as Enzyme.

B. Deep vs. Shallow Layer Graphs

Result: *Tapeflow improves performance by 2× 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) which 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 17 depicts how the change in the unroll factor effects 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 plot). As layer parallelism increases with wider layers (in shallow graphs), the performance gains of Tapeflow increases. Tapeflow improves gains by: i) removing tape accesses from cache, ii) supplying parallel compute ops from the region pad, and iii) streaming data from the DRAM and eliminating cache pressure.

C. Fully Unrolling The Loops

Result: *Tapeflow improves performance by up to 14 × i.e., more parallelism.*

In Figure 18, we compare the performance of two tools: Enzyme and Tapeflow, under the condition where the benchmarks are fully unrolled. This full unrolling process leads to an increase in the number of DRAM accesses at each instance,

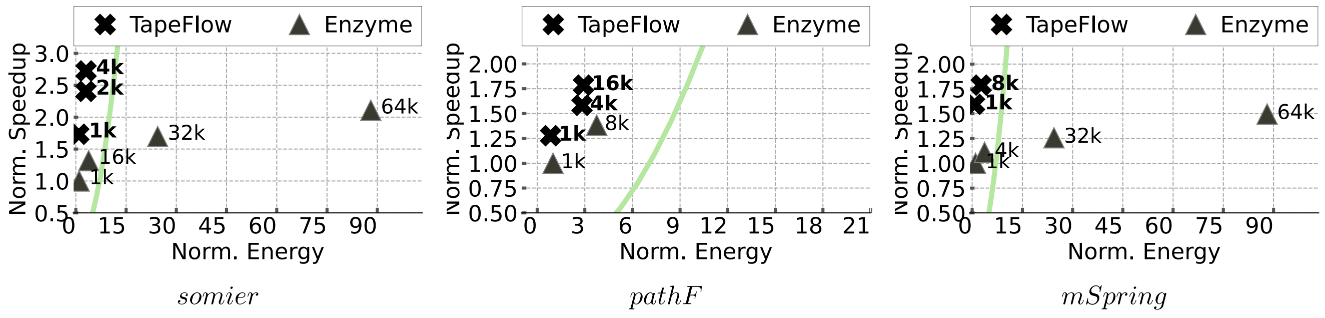


Fig. 15: Performance-Energy Sweep. Green: Markers to the left have a better trade-off. X-axis: Normalized Energy (Towards origin is better). Y-axis: Normalized performance (Away from origin is better).

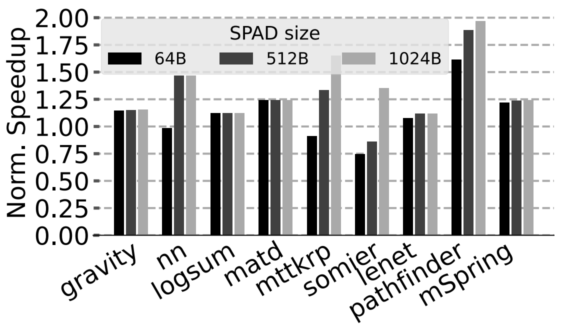


Fig. 16: Scratchpad size vs. Norm. Performance. Y-axis: Higher is better. X-axis: scratchpad size

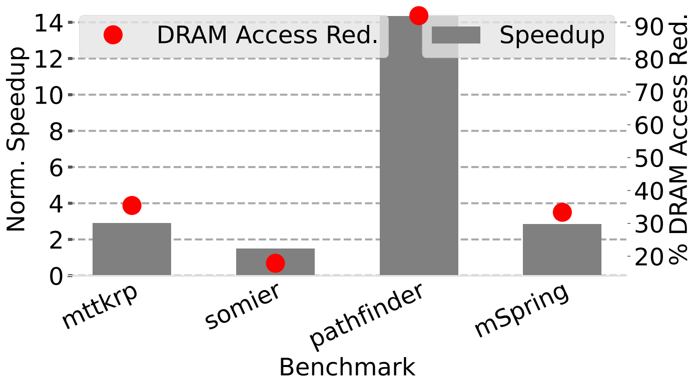


Fig. 18: Performance comparison between Tapeflow and Enzyme for fully unrolled benchmarks.

subsequently elevating the demand for bandwidth and causing cache conflicts.

Tapeflow consistently outperforms Enzyme in this scenario because it experiences fewer conflicts and demands less DRAM bandwidth, primarily due to its separate storage of tape values. Notably, in the case of the "pathF" benchmark, we achieved a remarkable 14× performance improvement. This substantial gain can be attributed to the fact that approximately 90% of its memory accesses involve tape accesses, which are rerouted to the scratchpad instead of DRAM. This reduction in DRAM access not only minimizes serialization over the limited cache and memory ports but also enhances Instruction-Level Parallelism (ILP).

While other benchmarks also benefit from the removal of tape accesses from DRAM, their performance improvements are not as significant as in the "pathF" benchmark. This is because even after excluding tape accesses, these benchmarks still have a considerable number of non-tape DRAM accesses, accounting for approximately $70-90\%$ of their original memory accesses, which limits their ILP gains.

VIII. Conclusion

We will be releasing the compiler and gem5-based models to enable future work. We propose Tapeflow, a compiler framework for managing tapes in reverse-mode automatic differentiation (AD). This is the first paper to evaluate compiler support for optimizing gradient tapes in AD. Tapeflow improves the performance of AD by leveraging compiler analysis to organize the tape for explicit streaming.

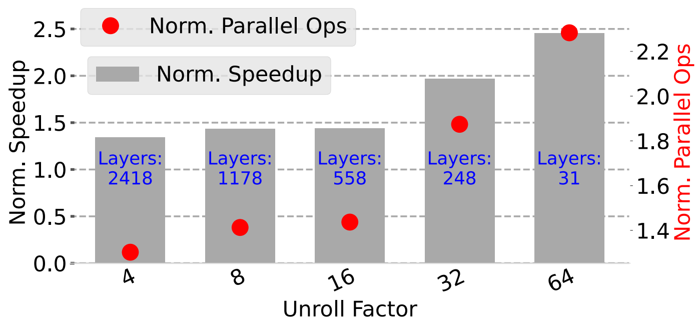


Fig. 17: Shallow vs Deep Layers. Benchmark: Pathfinder. Left axis: Norm. Speedup. Right axis: Norm. parallelism/layer.

References

[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] Jonas Degrave, Michiel Hermans, Joni Dambre, and Francis wyffels. A Differentiable Physics Engine for Deep Learning in Robotics. *Frontiers in Neurorobotics*, 13, 2019.
[11] 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.
[12] 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.
[13] Audrūnas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. Memory-efficient backpropagation through time.
[14] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. DeSC: decoupled supply-compute communication management for heterogeneous architectures. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 191–203, Waikiki Hawaii, December 2015. ACM.

[15] Laurent Hascoet and Valérie Pascual. The Tapenade automatic differentiation tool: Principles, model, and specification. *ACM Transactions on Mathematical Software*, 39(3):1–43, April 2013.

[16] 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.

[17] 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.

[18] 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.

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

[20] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. Differentiable programming for image processing and deep learning in halide. *ACM Transactions on Graphics*, 37(4):1–13, August 2018.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. Stream-Dataflow Acceleration. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 416–429, Toronto ON Canada, June 2017. ACM.

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

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

[40] 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.

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

[42] 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.

[43] V. Vassilev, M. Vassilev, A. Penev, L. Moneta, and V. Ilieva. Clad – Automatic Differentiation Using Clang and LLVM. volume 608, page 012055. IOP Publishing, may 2015.

[44] 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.

[45] 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.

[46] 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.

[47] 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.