Dynamic Automatic Differentiation of GPU Broadcast Kernels

Jarrett Revels[*]
Massachusetts Institute of Technology

Tim Besard
Ghent University

Valentin Churavy
Massachusetts Institute of Technology

Bjorn De Sutter
Ghent University

Juan Pablo Vielma
Massachusetts Institute of Technology

Abstract

We show how forward-mode automatic differentiation (AD) can be employed within larger reverse-mode computations to dynamically differentiate broadcast operations in a GPU-friendly manner. Our technique fully exploits the broadcast Jacobian's inherent sparsity structure, and unlike a pure reverse-mode approach, this "mixed-mode" approach does not require a backwards pass over the broadcasted operation's subgraph, obviating the need for several reverse-mode-specific programmability restrictions on user-authored broadcast operations. Most notably, this approach allows broadcast fusion in primal code despite the presence of data-dependent control flow. We discuss an experiment in which a Julia implementation of our technique outperformed pure reverse-mode TensorFlow and Julia implementations for differentiating through broadcast operations within an HM-LSTM cell update calculation. [2]

1 Introduction

In recent years, the prevalence of gradient-based optimization in machine learning (ML) has motivated an upsurge in the development of ML-specific modeling languages that incorporate automatic differentiation (AD) as a fundamental feature. However, contemporary ML research routinely seeks to utilize new modeling and optimization techniques that push these frameworks' AD capabilities to their limit. Both practical and exploratory implementations of such techniques demand advanced features such as nested differentiation, differentiation through data-dependent control flow, domain-specific hardware specialization, distributed parallelism, checkpointing, and more [20, 8, 18, 19, 4, 27, 3].

In the pursuit of solutions capable of incorporating such features, it has become clear that modeling languages' *expressiveness* must necessarily be constrained for the sake of *differentiability*. Recent endeavors [40, 7, 12, 11, 31, 22, 36, 32] that explore this tradeoff have been strongly guided by well-established methods from programming language research, provoking the evolution of a new research area known as *differentiable programming*. This is quite a natural development, as the narrative of traditional AD research has always been richly intertwined with PL and mathematical programming research.

[*]Corresponding author, jrevels@mit.edu

[2]The full version of this short paper contains additional examples, experiments, and analysis, and can be found at [1].

Preprint. Work in progress.

Particularly relevant in this regard is the work of Siskind and Pearlmutter, whose stated vision aligns surprisingly well with the goals of contemporary differentiable programming research: "Our vision: ...a unified intermediate language that supports both compiler optimizations and AD transformations for a variety of source and target languages." [33] When viewed through the lens of [23], it can be concluded that "optimal" differentiation of such a language cannot be achieved via pure forward- or reverse-mode approaches, but rather demands a *mixed-mode* approach. Achieving optimality, in this case, is often defined as minimizing the number of multiply-adds required to differentiate a given program via selecting the optimal mode for each subregion, and is known as the Optimal Jacobian Accumulation (OJA) problem. This problem has been shown to be NP-complete [24].

Despite this general theoretical intractability, mixed-mode AD offers a host of other advantages that can still be leveraged in practice by heuristically exploiting the local structure of the target language's primitive operations. This paper's primary concern is the application of this idea to a very common scientific computing primitive: the `broadcast` operation [37]. In §2, we present the `broadcast` operation and a reverse-mode-interleavable forward-mode method for its differentiation that exploits the special sparsity structure of its total Jacobian. In §3, we discuss an experiment that demonstrates our method's superiority over pure reverse-mode approaches for the differentiation of a data-dependent HM-LSTM cell update calculation on the GPU. **Via this paper, we wish to motivate the development of a new generation of differentiating compilers that do not operate purely in the forward or reverse modes, but rather choose the optimal mode for each target subprogram when such a choice is naively determinable from local structure.**

2 Methodology

2.1 Broadcast Operations

Throughout this paper, we will append a period to a function to denote *broadcasting* that function over its arguments. We define the broadcast of a function $b : \mathbb{R}^N \to \mathbb{R}^M$ as:

$$b.(\mathbf{X}_1 \dots \mathbf{X}_N) = \texttt{broadcast}(b, \mathbf{X}_1 \dots \mathbf{X}_N) = \texttt{map}(b, \boldsymbol{\mathcal{X}}_1 \dots \boldsymbol{\mathcal{X}}_N) = \mathbf{Y}_1 \dots \mathbf{Y}_M$$

Here, the $\mathbf{X}_j$ arguments are multidimensional arrays of arbitrary shape [3], subject to the constraint that each dimension of any argument must either have the same length as that dimension in other arguments, or must have length 1. Each $\boldsymbol{\mathcal{X}}_j$ is equivalent to the corresponding $\mathbf{X}_j$, but where length-1 dimensions are "copied" along that dimension to match its maximum length across all $\mathbf{X}_j$, such that all $\boldsymbol{\mathcal{X}}_j$ are of equal shape. The function b is then mapped elementwise across all $\boldsymbol{\mathcal{X}}_j$, resulting in the outputs $\mathbf{Y}_i$, each of which is the same shape as any $\boldsymbol{\mathcal{X}}_j$.[4] To denote broadcast of binary infix operators, we prepend (instead of append) the period, e.g. $f.(\mathbf{X}_1) \mathbin{.+} g.(\mathbf{X}_2) = +.(f.(\mathbf{X}_1), g.(\mathbf{X}_2))$.

Assuming that a pair of broadcasted operations have compatible shapes and are relatively side-effect free, the broadcast of their composition generally obeys the following relation:

$$g.(f.(\mathbf{X}_1 \dots \mathbf{X}_N)) = (g \circ f).(\mathbf{X}_1 \dots \mathbf{X}_N) \tag{1}$$

In programs containing broadcast operations, Eq. (1) can be exploited to perform *broadcast fusion*, a compiler-level optimization that transforms compositions of broadcast calls into a single broadcast call. This optimization imparts a couple of performance benefits. First, by obviating the need to compute and store intermediate results, broadcast fusion reduces memory usage, temporary allocations, and kernel invocations required to complete the computation. Second, broadcast fusion allows the fused broadcast operation to be parallelized without re-synchronization between intermediary broadcast operations [10, 17].

2.2 Automatic Differentiation of Broadcast Operations

2.2.1 Multidimensional Dual Numbers

A common way to formulate forward-mode AD is via the algebra of *dual numbers*. Dual numbers are similar to complex numbers, but instead of appending the imaginary unit i to $\mathbb{R}$, the dual number

[3] Note that scalars and single-element arrays are equivalent under this definition of `broadcast`.

[4] Shrewd broadcast implementations do not explicitly materialize the $\boldsymbol{\mathcal{X}}_j$ arguments, but rather index directly into the $\mathbf{X}_j$ arguments to invoke b elementwise (e.g. Eq. (**??**))

algebra appends the infinitesimal perturbation ϵ where $\epsilon^2 = 0, \epsilon \neq 0$ to $\mathbb{R}$. Via Taylor series expansion, unary function application on dual numbers is defined as $f(x+y\epsilon) = f(x)+f'(x)y\epsilon$ where $x, y \in \mathbb{R}$.

While this formulation is usually straightforward to implement, it is also quite limited - only a single scalar derivative can be calculated per call of the target function. To overcome this limitation, we can use an extended formulation of the dual numbers known as the *multidimensional dual numbers*, which are defined as $x + \sum_{i=1}^{k} y_i\epsilon_i$ where $\epsilon_i\epsilon_j = 0$. Multidimensional dual numbers allow for a "vector forward-mode" implementation of gradient calculation, where orthogonal ϵ components are appended to orthogonal input components to compute their individual directional derivatives [28, 30], i.e. $f(\mathbf{x}_\epsilon) = f(\mathbf{x}) + \sum_{i=1}^{k} \frac{\partial f(\mathbf{x})}{\partial x_i}\epsilon_i$ where $\mathbf{x} = (x_1 \dots x_k)$ and $\mathbf{x}_\epsilon = (x_1 + \epsilon_1 \dots x_k + \epsilon_k)$. To extract the ϵ components as a tuple from a multidimensional dual number, we utilize the tangent extraction function $\mathbf{tg}$, defined as $\mathbf{tg}_\alpha(x + \sum_{i=1}^{k} y_i\alpha[\epsilon_i]) = (y_1 \dots y_k)$. Note that $\mathbf{tg}$ utilizes the notion of *tagged perturbations*; $\mathbf{tg}_\alpha$ only extracts perturbations that are marked with the "tag" α, represented here via the bracket syntax $\alpha[\epsilon]$. This tagging machinery is necessary (but unfortunately not sufficient) to avoid a class of AD bugs known as *perturbation confusion* [35, 34, 21].

2.2.2 Sparse Forward-Mode Jacobians of Broadcasted Operations

Broadcasted operations generally take the form $b : \mathbb{R}^N \to \mathbb{R}^M$ where N is the input arity and M is the output arity. While b might be broadcasted over millions of input elements, the arities N and M are generally relatively small (often < 10). To automatically differentiate such a function in the forward-mode, we can define a Jacobian operator $\mathbf{D}$ using multidimensional dual numbers:

$$\mathbf{D}(b) = (x_1 \dots x_N) \mapsto \mathbf{tg}_\alpha.(b(x_1 + \alpha[\epsilon_1], x_2 + \alpha[\epsilon_2], \dots x_N + \alpha[\epsilon_N])) \tag{2}$$

Note that we are broadcasting $\mathbf{tg}_\alpha$ over the output tuple of b in order to extract all components of the Jacobian. For example, for $b : \mathbb{R}^3 \to \mathbb{R}^2$, the definition expands to the following:

$$\begin{aligned}
\mathbf{D}(b) &= (x_1, x_2, x_3) \mapsto \mathbf{tg}_\alpha.(b(x_1 + \alpha[\epsilon_1], x_2 + \alpha[\epsilon_2], x_3 + \alpha[\epsilon_3])) \\
&= (x_1, x_2, x_3) \mapsto \begin{pmatrix} \mathbf{tg}_\alpha(y_1 + \frac{\partial y_1}{\partial x_1}\alpha[\epsilon_1] + \frac{\partial y_1}{\partial x_2}\alpha[\epsilon_2] + \frac{\partial y_1}{\partial x_3}\alpha[\epsilon_3]) \\ \mathbf{tg}_\alpha(y_2 + \frac{\partial y_2}{\partial x_1}\alpha[\epsilon_1] + \frac{\partial y_2}{\partial x_2}\alpha[\epsilon_2] + \frac{\partial y_2}{\partial x_3}\alpha[\epsilon_3]) \end{pmatrix} \\
&= (x_1, x_2, x_3) \mapsto \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \frac{\partial y_1}{\partial x_3} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \frac{\partial y_2}{\partial x_3} \end{pmatrix}
\end{aligned}$$

The following observation is frequently utilized throughout the rest of the paper:

$$\begin{gathered}
\text{Given } b : \mathbb{R}^N \to \mathbb{R}^M \text{ s.t.} \\
b.(\mathbf{X}_1 \dots \mathbf{X}_N) = \texttt{map}(b, \boldsymbol{\mathcal{X}}_1 \dots \boldsymbol{\mathcal{X}}_N) = (\mathbf{Y}_1 \dots \mathbf{Y}_M), \text{then} \\
\mathbf{D}(b).(\mathbf{X}_1 \dots \mathbf{X}_N) \to \left\{ \texttt{diag}\left(\frac{\partial(\texttt{vec}(\mathbf{Y}_i))}{\partial(\texttt{vec}(\boldsymbol{\mathcal{X}}_j))} \right) \mid i \in 1 \dots M, j \in 1 \dots N \right\}
\end{gathered} \tag{3}$$

where `vec` is notation for vectorization (i.e. "flattening" the given tensor to a vector), and `diag` is notation for extracting the diagonal of a square matrix. In other words, Eq. (3) directly computes all elementwise partial derivatives of the total Jacobian of $b.(\mathbf{X}_1 \dots \mathbf{X}_N)$. This approach exploits the sparsity structure imposed on the Jacobian by the broadcast operation, avoiding calculation of the zero-valued cross-element partial derivatives (the off-diagonal elements of the $\frac{\partial(\texttt{vec}(\mathbf{Y}_i))}{\partial(\texttt{vec}(\boldsymbol{\mathcal{X}}_j))}$ matrices) by construction. Note, however, that if b entails side-effects that induce cross-element dependence, then the total Jacobian is not fully recovered by this method, since the uncalculated cross-element partial derivatives may be nonzero in this case.

2.2.3 Employing Forward-Mode Within Reverse-Mode

Eq. (3) has significant performance and programmability implications when exploited within larger reverse-mode AD computations, since it enables the differentiation of fully-fused broadcast subgraphs without requiring the construction of a backwards pass. Specifically, Eq. (3) can be employed to easily calculate and cache the intermediate Jacobian of the broadcast subgraph during the forward pass of the overall reverse-mode computation. This Jacobian can then be backpropagated during the reverse pass instead of backpropagating through the broadcast subgraph directly. In this way, Eq. (3)

Table 1: Example Reverse-Mode Computation

Definition	Forward (Primal)	Reverse (Adjoint)
$h(x, \mathbf{y}) = g(\mathbf{f}(x, \mathbf{y}))$		$\overline{w}_2 = 1$ (seed)
$\mathbf{f}(x, \mathbf{y}) = b.(x, \mathbf{y})$	$\mathbf{w}_1 = \mathbf{f}(x, \mathbf{y})$	$\overline{\mathbf{w}}_1 = \overline{w}_2 \frac{\partial w_2}{\partial \mathbf{w}_1}$
$b : \mathbb{R}^2 \to \mathbb{R}$	$w_2 = g(\mathbf{w}_1)$	$\frac{\partial h}{\partial x} = \overline{\mathbf{w}}_1 \cdot \frac{\partial \mathbf{f}}{\partial x}$
$g : \mathbb{R}^N \to \mathbb{R}$		$\frac{\partial h}{\partial \mathbf{y}} = \overline{\mathbf{w}}_1 \cdot\times \frac{\partial f_i}{\partial y_i}$
$x \in \mathbb{R},\ \mathbf{y} \in \mathbb{R}^N$		

allows one to treat entire broadcast subgraphs as fused forward-mode primitives, obviating the need for reversible representations of these subgraphs.

To illustrate the use of Eq. (3) within a reverse-mode computation, consider the example defined in Table 1. The left column defines the target function h, the center column expresses the primal forward pass of the computation, and the right column expresses the adjoint pass used to compute $\frac{\partial h}{\partial x}$ and $\frac{\partial h}{\partial \mathbf{y}}$. In the adjoint pass, the actual calculation of $\overline{\mathbf{w}}_1$ can be accomplished via the usual reverse-mode approach of decomposing $g(\mathbf{w}_1)$ into a reversible subgraph built from known primitive operations. The calculation of $\frac{\partial \mathbf{f}}{\partial x}$ and $\frac{\partial f_i}{\partial y_i}$, however, can be accomplished via Eq. (3) without requiring the construction of reverse-mode computation subgraph at all: $(\frac{\partial \mathbf{f}}{\partial x}, \frac{\partial f_i}{\partial y_i}) = \mathbf{D}(b).(x, \mathbf{y})$.

Note that while it is mathematically useful to discuss $\mathbf{D}(b).(x, \mathbf{y})$ and $b.(x, \mathbf{y})$ as separate computations as we have done here, practical AD implementations leveraging this method can exploit the implicit computation of $b.(x, \mathbf{y})$ that occurs as part of computing $\mathbf{D}(b).(x, \mathbf{y})$. Instead of applying the $\mathbf{tg}_\alpha$ operator immediately as is done in Eq. (2), the primal and dual computation results can be extracted simultaneously. In other words, the $\mathbf{w}_1$ step in forward pass code can be replaced with a step that simultaneously calculates $\mathbf{w}_1$, $\frac{\partial \mathbf{f}}{\partial x}$, and $\frac{\partial f_i}{\partial y_i}$ by simply invoking $\mathbf{f}$ with dual number inputs.

Fusing the primal and derivative calculations in this manner avoids redundant computation, but requires that the resulting partial derivatives be cached until they are backpropagated in the reverse pass. This fusion, then, may not be desirable if there is not sufficient memory available to sustain such a cache. Conversely, computing $\mathbf{D}(b).(x, \mathbf{y})$ during the reverse pass redundantly computes $b.(x, \mathbf{y})$, but the resulting partial derivatives can be backpropagated immediately, and thus their storage can be freed (or reused) immediately. Ultimately, the choice of whether $\mathbf{D}$ should be applied in the forward pass or in the reverse pass depends on the memory/compute bandwidth of the overall computation.

2.2.4 Forward-Mode vs. Reverse-Mode For $\mathbf{D}(b)$

The previous sections implemented $\mathbf{D}$ as a forward-mode differentiation operator, but $\mathbf{D}$ could have also been implemented via reverse-mode AD without invalidating Eq. (3). Why, then, is forward-mode the better choice for this use case? The answer to this question can be summarized in three points:

1: If $N > M$, then reverse-mode is algorithmically superior to forward-mode. However, b is generally low-arity, and in practice, forward-mode often outperforms reverse-mode for low-arity functions regardless of the N/M ratio. There are two reasons for this. First, reverse-mode implementations often incur relatively high constant costs that are not amortized in the low-arity regime. Second, forward-mode's additional chain rule applications can be offset for low-arity functions by leveraging stack allocation schemes that make better use of cache bandwidth and allow for the exploitation of instruction-level parallelism [30].

2: If the target function contains data-dependent control flow, reverse-mode implementations must dynamically allocate the data-dependent regions of the computation graph[5]. For low-arity functions,

[5]This requirement is not implementation-specific, but rather a hard theoretical limit; capturing intermediate values which depend on run time data will always require run time allocation in the general case, though certain optimizations may alleviate this burden in special cases. This requirement applies even to reverse-mode tools that claim to be "tapeless" by statically generating backwards pass code [38, 22], or performing equivalent

the overhead of dynamic trace allocation can easily dwarf the cost of the target function's primal evaluation. For broadcasted operations, this high overhead would be incurred for every elementwise invocation, rendering the reverse-mode approach in this case wholly unsuitable for the GPU where excessive dynamic allocation is infeasible.

3: Following from the previous point, using forward-mode for broadcast differentiation allows data-dependent control flow to occur within broadcasted scalar operations, thus avoiding several disadvantages inherent to vectorized control flow primitives currently employed by reverse-mode frameworks (e.g. TensorFlow's `where` [3]). The first disadvantage is programmability; vectorized control flow primitives are often more cumbersome to use than their naive scalar counterparts. The second disadvantage is that many vectorized control flow primitives require computing untaken branches. While these primitives do have the benefit of clearly avoiding warp divergence on the GPU, the experiment described in §3 demonstrates that this benefit does not necessarily offset the cost of computing untaken branches on newer GPU architectures - especially if the difference in cost between branches is substantial - since newer architectures support executing different instructions across a warp without forcing serialized execution [25].

3 Performance Experiments

In this section [6], we describe an experiment performed to compare this paper's forward-mode broadcast differentiation technique with existing reverse-mode approaches. Our test case for this experiment was a cell update calculation that occurs during the execution of a hierarchical multiscale LSTM (HM-LSTM) [9], a real-world example of a broadcast operation that is amenable to differentiation via Eq. (3). We chose this operation as our experimental test case because it is self-contained, hinges on data-dependent control flow, has a substantial computational cost difference between branches, and is relevant to a machine learning audience.

For a given time step t and layer ℓ, the update calculation for the cell $\mathbf{c}_t^\ell$ is:

$$\mathbf{c}_t^\ell = \begin{cases} \sigma.(\mathbf{f}_t^\ell) \mathbin{.\times} \mathbf{c}_{t-1}^\ell \mathbin{.+} \sigma.(\mathbf{i}_t^\ell) \mathbin{.\times} \tanh.(\mathbf{g}_t^\ell) & \text{if } z_{t-1}^\ell = 0, z_t^{\ell-1} = 1 \text{ (UPDATE)} \\ \mathbf{c}_{t-1}^\ell & \text{if } z_{t-1}^\ell = 0, z_t^{\ell-1} = 0 \text{ (COPY)} \\ \sigma.(\mathbf{i}_t^\ell) \mathbin{.\times} \tanh.(\mathbf{g}_t^\ell) & \text{if } z_t^{\ell-1} = 1 \text{ (FLUSH)} \end{cases} \tag{4}$$

where $\mathbf{f}$ and $\mathbf{i}$ are memory gates, $\mathbf{g}$ is a cell proposal vector, and z is a boundary state. Our benchmark is the computation of $\frac{\partial \mathbf{c}_t^\ell}{\partial \mathbf{c}_{t-1}^\ell}$, $\frac{\partial \mathbf{c}_t^\ell}{\partial \mathbf{f}_t^\ell}$, $\frac{\partial \mathbf{c}_t^\ell}{\partial \mathbf{i}_t^\ell}$, and $\frac{\partial \mathbf{c}_t^\ell}{\partial \mathbf{g}_t^\ell}$.

The first implementation tested in our experiment was a TensorFlow-based implementation derived from [26]. This implementation makes use of TensorFlow's vectorized control flow primitive `where`, which eagerly computes both branches of the conditional statement before returning the branch specified by the given predicate. A visualization of the implementation's post-optimization intermediate representation (IR) can be found in [2], depicted as a computation graph in the High Level Optimizer (HLO) format. From this graph, it can be seen that TensorFlow's XLA compiler broke up the entire computation into six separate kernels, each representing a partially fused region of the forward and reverse passes, including broadcasted `select` operations that were generated from the initial code's `where` invocations.

The second implementation tested in our experiment was a reverse-mode implementation in the Julia language [6]. This implementation was directly derived from the HLO graph of the TensorFlow implementation described in the previous section. The intent was to exactly mirror TensorFlow's operations at the abstraction level of its HLO representation in order to better bridge comparisons between the reverse-mode TensorFlow and forward-mode Julia implementations. To accomplish this, the HLO graph operations were manually transcribed as native Julia code, additionally using the CUDAnative package to enable execution on the GPU [5].

---

transformation via language-level constructs such as delimited continuations or closures [39]. As Pearlmutter and Siskind remark, it is "impossible" to "eliminate the tape from reverse-mode AD" because "the tape stores intermediate values computed during the forward phase that are needed during the reverse phase." [29]

[6] Note that all implementation, benchmark, and test code employed for this section is publicly available in its entirety at [2]. For a detailed description of our experimental setup, including additional measurements and analysis of warp divergence and hardware utilization scaling w.r.t. the forward-mode technique, please see the full version of this paper at [1].

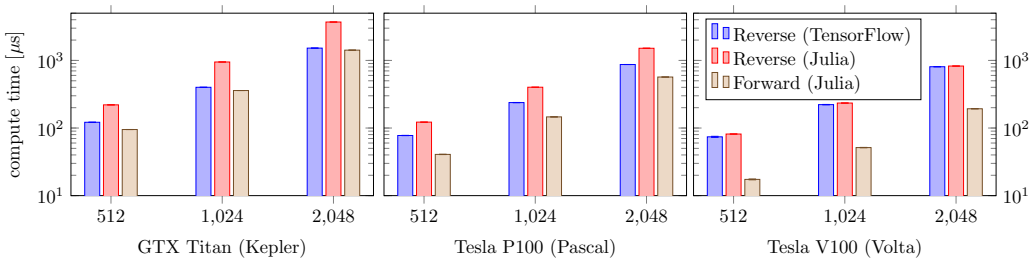

Figure 1: Total kernel compute times across different AD implementations

The third implementation tested in our experiment was a native Julia implementation of forward-mode broadcast differentiation as described by Eq. (3), using the ForwardDiff package's multidimensional dual number implementation [30] and the CUDAnative package for execution on the GPU. While the reverse-mode implementations expressed control flow via vectorized primitives, the forward-mode approach allows the full broadcast fusion (including control flow) of Eq. (4) via Eq. (1) without incurring the reverse-mode-specific performance penalties discussed in §2.2.4.

Fig. 1 shows the execution times to compute the aforementioned derivatives for each implementation described across three generations of NVIDIA GPUs, with $\mathbf{c}_{t-1}^{\ell}$, $\mathbf{f}_t^{\ell}$, $\mathbf{i}_t^{\ell}$, and $\mathbf{g}_t^{\ell}$ taking $n \times n$ random 32-bit floating point matrix values and z_{t-1}^{ℓ} and $z_t^{\ell-1}$ taking n-element random 32-bit floating point vector values where $n \in \{512, 1024, 2048\}$. As can be seen in Fig. 1, the forward-mode Julia implementation features a speedup of 4.28x, 2.66x, and 2.60x over the reverse-mode Julia implementation on the Volta, Pascal, and Kepler architectures, respectively. Compared to the reverse-mode TensorFlow implementation, these speedups are 4.18x, 1.53x and 1.07x, respectively.

4 Conclusion

In this paper, we presented a reverse-mode-interleavable forward-mode method for the differentiation of `broadcast` that outperforms pure reverse-mode methods on the GPU and simultaneously obviates the need for reverse-mode-specific programmability restrictions on user-authored broadcasted operations. This mixed-mode technique is, in fact, already well-utilized in the Julia ecosystem. It was first introduced in 2016 by the ReverseDiff package (developed by this paper's first author) [16], whose original implementation of the method has since propagated to the Flux and Zygote packages [13, 22].

In the future, higher-order mixed-mode AD is likely to present interesting new challenges in the vein of perturbation/sensitivity confusion. For example, consider the forward-mode differentiation of the broadcast of a function that closes over variables naively tracked by a surrounding reverse-mode implementation. More research is needed to identify these potentially problematic scenarios and explore their ramifications.

Additional work has been planned to implement first-class mixed-mode AD for Julia within the upcoming Capstan package [14], which will build on recently developed tools enabling third-party packages to extend Julia's compiler with new, context-specific behaviors by dynamically injecting code transformation passes into Julia's just-in-time (JIT) compilation cycle [15].

5 Acknowledgments

Jarrett Revels was supported by MIT's NEC Corporation Fund for Research in Computers and Communications, MIT's Skoltech Next Generation Program and a Lawrence Livermore National Lab Subcontract. Tim Besard was supported by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), and by the Research Foundation – Flanders (FWO). Valentin Churavy was supported by NSF DMS-1312831, Darpa XDATA, and an ARAMCO MITEI grant. The authors would also like to thank James Bradbury, Peter Ahrens, Mike Innes, Deniz Yuret, and Simon Danisch for multiple helpful conversations and contributions to Julia's AD and GPU ecosystems.

References

[1] https://arxiv.org/abs/1810.08297.

[2] https://github.com/jrevels/MixedModeBroadcastAD.jl.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, volume 16, pages 265–283, 2016.

[4] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18(153):1–153, 2017.

[5] Tim Besard, Christophe Foket, and Bjorn De Sutter. Effective extensible programming: Unleashing Julia on GPUs. *CoRR*, abs/1712.03112, 2017.

[6] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, 59(1):65–98, 2017.

[7] Olivier Breuleux and Bart van Merriënboer. Automatic Differentiation in Myia. 2017.

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: end-to-end compilation stack for deep learning. In *SysML Conference*, 2018.

[9] Junyoung Chung, Sungjin Ahn, and Yoshua Bengio. Hierarchical multiscale recurrent neural networks. *arXiv preprint arXiv:1609.01704*, 2016.

[10] Alain Darte. On the complexity of loop fusion. *Parallel Computing*, 26(9):1175–1193, jul 2000.

[11] Conal Elliott. The Simple Essence of Automatic Differentiation. *Proc. ACM Program. Lang.*, 2(ICFP):70:1–70:29, July 2018.

[12] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. 2018.

[13] Mike Innes. Flux: Elegant machine learning with Julia. *Journal of Open Source Software*, 3(25):602, may 2018.

[14] Jarrett Revels. Capstan: Cassette-Based Automatic Differentiation for Julia. https://github.com/JuliaDiff/Capstan.jl, 2018.

[15] Jarrett Revels. Cassette: A Tool For Dynamically Extending the Julia Compiler. https://github.com/jrevels/Cassette.jl, 2018.

[16] Jarrett Revels. ReverseDiff: Reverse-Mode AD in Julia. https://github.com/JuliaDiff/ReverseDiff.jl, 2018.

[17] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 301–320, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[18] Griffin Lacey, Graham W. Taylor, and Shawki Areibi. Deep Learning on FPGAs: Past, Present, and Future. *CoRR*, abs/1602.04283, 2016.

[19] 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 (TOG)*, 37(4):139, 2018.

[20] Dougal Maclaurin, David Duvenaud, and Ryan Adams. Gradient-based hyperparameter optimization through reversible learning. In *International Conference on Machine Learning*, pages 2113–2122, 2015.

[21] Oleksandr Manzyuk, Barak A Pearlmutter, Alexey Andreyevich Radul, David R Rush, and Jeffrey Mark Siskind. Confusion of tagged perturbations in forward automatic differentiation of higher-order functions. *arXiv preprint arXiv:1211.4892*, 2012.

[22] Mike Innes. Zygote: 21st Century AD in Julia. `https://github.com/FluxML/Zygote.jl`, 2018.

[23] U Naumann. Optimized Jacobian Accumulation Techniques. 2000.

[24] Uwe Naumann. Optimal Jacobian accumulation is NP-complete. *Mathematical Programming*, 112(2):427–441, 2008.

[25] NVIDIA Corporation. NVIDIA Tesla V100 GPU Architecture. `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`, 2017.

[26] Paige Finkelstein. A Tensorflow Implementation of a Hierarchical and Multiscale RNN. `https://github.com/bolducp/hierarchical-rnn`, 2017.

[27] 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. In *NIPS-W*, 2017.

[28] Barak A Pearlmutter and Jeffrey Mark Siskind. Lazy multivariate higher-order forward-mode AD. In *ACM SIGPLAN Notices*. ACM, 2007.

[29] Barak A. Pearlmutter and Jeffrey Mark Siskind. Reverse-mode ad in a functional framework: Lambda the ultimate backpropagator. *ACM Trans. Program. Lang. Syst.*, 30(2):7:1–7:36, March 2008.

[30] J. Revels, M. Lubin, and T. Papamarkou. Forward-Mode Automatic Differentiation in Julia. *arXiv:1607.07892 [cs.MS]*, 2016.

[31] Richard Wei. First-Class Automatic Differentiation in Swift: A Manifesto. `https://gist.github.com/rxwei/30ba75ce092ab3b0dce4bde1fc2c9f1d`, 2018.

[32] Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, Simon L. Peyton Jones, and Christoph Koch. Efficient Differentiable Programming in a Functional Array-Processing Language. *CoRR*, abs/1806.02136, 2018.

[33] Jeffrey M Siskind and Barak A Pearlmutter. Using polyvariant union-free flow analysis to compile a higher-order functional-programming language with a first-class derivative operator to efficient fortran-like code. 2008.

[34] Jeffrey Mark Siskind and Barak A Pearlmutter. Perturbation confusion and referential transparency: Correct functional implementation of forward-mode ad. 2005.

[35] Jeffrey Mark Siskind and Barak A Pearlmutter. Nesting forward-mode ad in a functional framework. *Higher-Order and Symbolic Computation*, 21(4):361–376, 2008.

[36] The PyTorch Team. Torch script documentation. `https://pytorch.org/docs/master/jit.html`, 2018.

[37] Stéfan van der Walt, S Chris Colbert, and Gaël Varoquaux. The NumPy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, mar 2011.

[38] Bart van Merriënboer, Alexander B. Wiltschko, and Dan Moldovan. Tangent: Automatic Differentiation Using Source Code Transformation in Python. 2017.

[39] Fei Wang, Xilun Wu, Gregory Essertel, James Decker, and Tiark Rompf. Demystifying Differentiable Programming: Shift/Reset the Penultimate Backpropagator. *arXiv preprint arXiv:1803.10228*, 2018.

[40] Richard Wei, Lane Schwartz, and Vikram Adve. Dlvm: A modern compiler framework for neural network dsls. *Urbana*, 51:61801, 2017.