Dynamic Automatic Differentiation of GPU Broadcast Kernels

Jarrett Revels (MIT) Tim Besard (Ghent) Valentin Churavy (MIT) Bjorn De Sutter (Ghent) Juan Pablo Vielma (MIT)

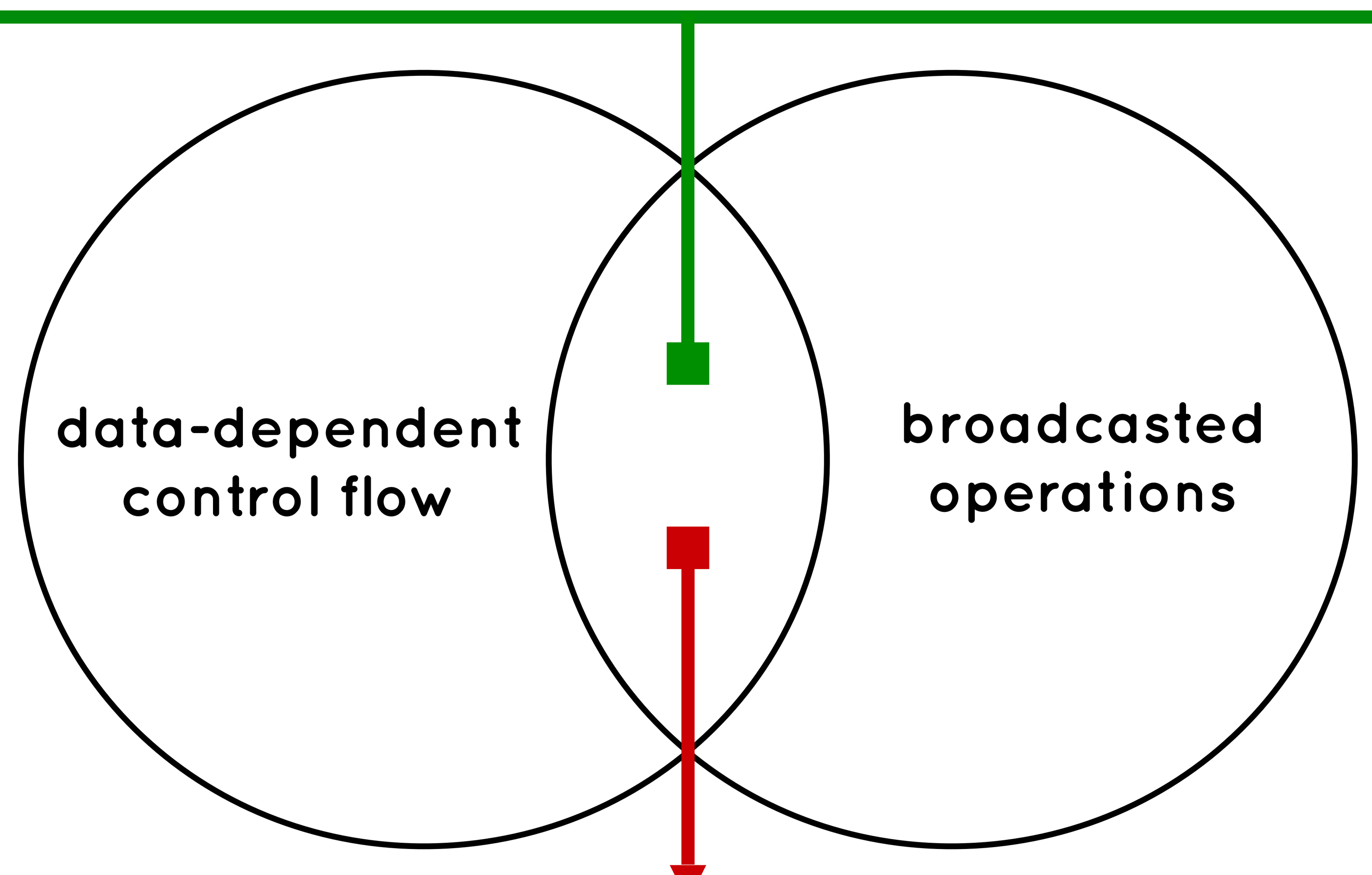
Abstract

We show how **forward-mode (FM) automatic differentiation** can be employed **within reverse-mode (RM) 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.

Fusion for these cases: **easy to perform** + **greatly beneficial**!



fusion → massive amount of data-dependent scalar operations

reverse-mode + dynamic scalar ops → fine-grained dynamic allocations

massive amount of fine-grained allocations → **BAD FOR GPU!**

Broadcasting $p : \mathbb{R}^3 \to \mathbb{R}^2$ over a matrix $\mathbf{A}$, scalar α, and vector $\mathbf{a}$:

$$p.(\mathbf{A}, \alpha, \mathbf{a}) = \left(\begin{bmatrix} p(A_{11}, \alpha, a_1)_1 & \dots & p(A_{1m}, \alpha, a_1)_1 \\ \vdots & \ddots & \vdots \\ p(A_{n1}, \alpha, a_n)_1 & \dots & p(A_{nm}, \alpha, a_n)_1 \end{bmatrix}, \begin{bmatrix} p(A_{11}, \alpha, a_1)_2 & \dots & p(A_{1m}, \alpha, a_1)_2 \\ \vdots & \ddots & \vdots \\ p(A_{n1}, \alpha, a_n)_2 & \dots & p(A_{nm}, \alpha, a_n)_2 \end{bmatrix} \right)$$

Reverse-mode AD for an operation containing a broadcast:

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

Our technique: replace b broadcast in forward pass with FM AD'd version to get elementwise derivatives without needing to RM b

$$\boxed{\left(\frac{\partial \mathbf{f}}{\partial x}, \frac{\partial f_i}{\partial y_i} \right) = \mathbf{D}(b).(x, \mathbf{y})}$$

Solution: <u>REVERSE-MODE AD</u> the overall computation, but <u>FORWARD-MODE AD</u> the broadcasted operation

- (FM mul-add count)/(RM mul-add count) ~ N/M. **For low-arity scalar ops** (e.g. broadcasted ops), though, **FM performance often matches RM performance even when N > M** by making good use of cache bandwidth and exploiting instruction-level parallelism.
- Unlike RM, **FM does not require a reversible representation of the operation**. Regardless of RM implementation (tape, graph, closure, continuation, etc.), handling **data-dependent control flow in target code generally requires dynamic allocation for RM.**
- Thus, **FM** enables data-dependent control within broadcasted operations. This capability **renders more operations fusable** and **allows users to express** kinds of **scalar control-flow** previously disallowed in target code, **increasing model programmability**.

An example where this technique pays off: **HM-LSTM cell update**!

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

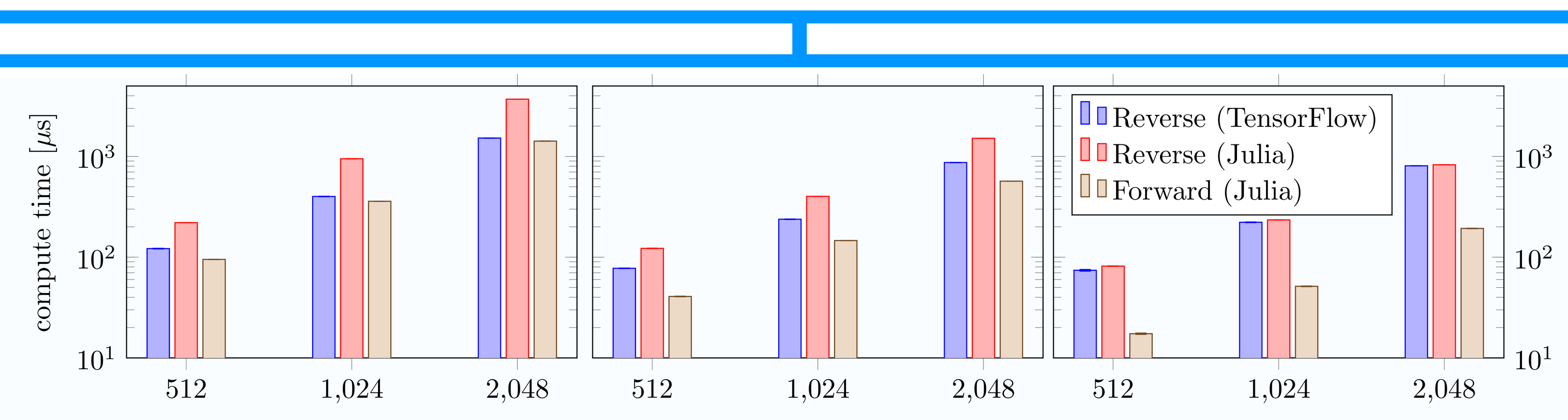
←**expensive branch** (UPDATE)
←**cheap branch** (COPY)

TensorFlow

```python
def cell_update(z, zb, c, f, i, g):
    i = tf.sigmoid(i)
    g = tf.tanh(g)
    f = tf.sigmoid(f)
    flush_control = tf.equal(z, tf.constant(1., dtype=tf.float32))
    flush = tf.multiply(i, g)
    copy_control = tf.equal(zb, tf.constant(0., dtype=tf.float32))
    copy = tf.identity(c)
    update = tf.add(tf.multiply(f, c), tf.multiply(i, g))
    return tf.where(flush_control, flush, tf.where(copy_control, copy, update))
```

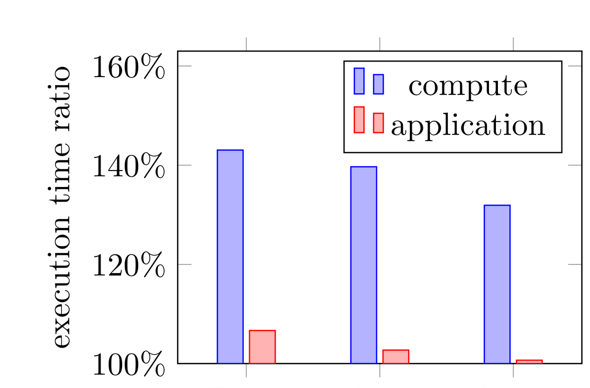
Native Julia

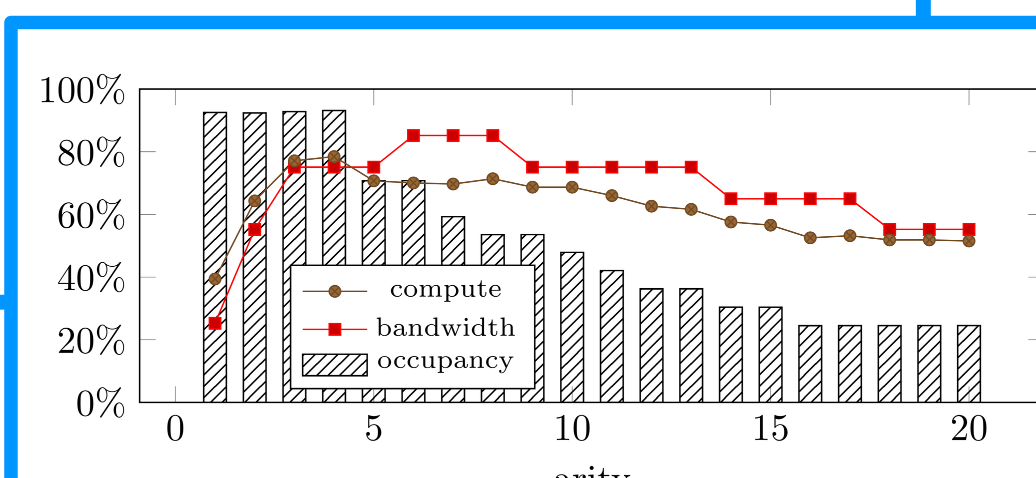
```julia
# can be broadcasted at callsite, e.g. cell_update.(z, zb, c, f, i, g)
function cell_update(z, zb, c, f, i, g)
   if z == 1.0f0
       return sigm(i) * tanh(g)
    elseif zb == 0.0f0
       return c
    else
       return sigm(f) * c + sigm(i) * tanh(g)
    end
end
```



- **FM beats out RM 2x-4x**; moreso when controlling for RM implementation
- **Newer GPU architectures** mitigate negative effects of warp divergence, and thus **benefit greatly from FM-enabled fusion**

- performance ratio for executing with totally warp-uniform control inputs vs. totally warp-divergent control inputs
- **Application overhead due to warp divergence on V100: < 1%**





- Hardware utilization drops as register pressure rises
- **Need heuristics to tune "chunk size" w.r.t. arity**; start by capping at 10.

This work is already used by **Flux/Zygote** on CPUs, GPUs, and **TPUs**.
We're planning a new tool called **Capstan** for mixed-mode AD.
Want more code, math, and the bibliography? **See the paper**!