# Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme

William S. Moses*, Valentin Churavy *, Ludger Paehler§, Jan Hückelheim †, Sri Hari Krishna Narayanan †, Michel Schanen †, Johannes Doerfert†
{wmoses,vchuravy}@mit.edu,ludger.paehler@tum.de,{jhuckelheim,snarayan,mschanen,jdoerfert}@anl.gov
MIT CSAIL *, Technical University of Munich §, Argonne National Laboratory†

## ABSTRACT

Computing derivatives is key to many algorithms in scientific computing and machine learning such as optimization, uncertainty quantification, and stability analysis. Enzyme is a LLVM compiler plugin that performs reverse-mode automatic differentiation (AD) and thus generates high performance gradients of programs in languages including C/C++, Fortran, Julia, and Rust. Prior to this work, Enzyme and other AD tools were not capable of generating gradients of GPU kernels. Our paper presents a combination of novel techniques that make Enzyme the first fully automatic reverse-mode AD tool to generate gradients of GPU kernels. Since unlike other tools Enzyme performs automatic differentiation within a general-purpose compiler, we are able to introduce several novel GPU and AD-specific optimizations. To show the generality and efficiency of our approach, we compute gradients of five GPU-based HPC applications, executed on NVIDIA and AMD GPUs. All benchmarks run within an order of magnitude of the original program's execution time. Without GPU and AD-specific optimizations, gradients of GPU kernels either fail to run from a lack of resources or have infeasible overhead. Finally, we demonstrate that increasing the problem size by either increasing the number of threads or increasing the work per thread, does not substantially impact the overhead from differentiation.

## CCS CONCEPTS

• **Mathematics of computing** → **Automatic differentiation**; • **Software and its engineering** → **Source code generation**; • **Theory of computation** → *Parallel computing models*; *Shared memory algorithms*; • **Computing methodologies** → *Machine learning*.

## KEYWORDS

Automatic Differentiation, AD, CUDA, ROCm, GPU, LLVM, HPC

**ACM Reference Format:**
William S. Moses*, Valentin Churavy *, Ludger Paehler§, Jan Hückelheim †,, Sri Hari Krishna Narayanan †, Michel Schanen †, Johannes Doerfert†. 2021. Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14-19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3458817.3476165

```
void init(double* ar, int N, double val) {
  parallel_for(int i=0; i<N ; i++)
    // Concurrent reads of val
    ar[i] = val;
}

double gradient_init(double* ar, double* d_ar,
                     int N, double val) {
  double d_val = 0.0;
  parallel_for(int i=0; i<N ; i++)
    ar[i] = val;
  parallel_for(int i=0; i<N ; i++) {
    // Concurrent writes to d_val
    d_val += d_ar[i];                    ⚡ race ⚡
    d_ar[i] = 0.0;
  }
  return d_val;
}
```

**Figure 1: A parallel initialize function (top) with a naive reverse mode AD gradient function (bottom) that does not take the parallelism into account. Consequently, the concurrent read of the variable `val` causes a race in the reverse-mode gradient computation.**

## 1 INTRODUCTION

Automatic differentiation (AD) provides an accurate way of computing derivatives of mathematical functions that are implemented in computer programs. *Gradients* (or *adjoints*), a special case of derivatives for functions with one output and many inputs, have applications in optimization [19], uncertainty quantification [21], inverse design, stability analysis,and machine learning [40]. Reverse-mode AD has been the tool of choice to compute these gradients for large applications with many input parameters.

As the research community has been continuously pushing the boundary of the size of problems they want to solve, large-scale applications have had to leverage the latest in high-performance computing including distributed computation, parallelism, and accelerators. For many machine learning and scientific computing applications, this means relying on kernels that are highly optimized for graphics processing units (GPUs).

While considerable effort has been expended to compute gradients of MPI and OpenMP programs (see Sec 2 for related work), no AD tool has been presented to date that can compute gradients of CUDA or ROCm (AMD) kernels. The cause rests with both the GPU's parallelism and its complex performance characteristics. The biggest issue for both performance and correctness is due to the implied computational flow reversal of reverse-mode AD; every read becomes a write in the adjoint computation and vice versa. Consider the program at the top of Figure 1. It contains a simple parallel for loop that reads from the same variable `val` in all threads and sets each index of the output array `ar` to that value. Since all threads read the same value, there is a *concurrent read access* on `val`, which does not impact the final result. Computing the gradient function of this program (i.e. the derivative of the input `val`), one must accumulate all of the partial derivatives of `val` generated by uses in the outputs. Such an action unfortunately leads to a *write race* on the gradient `d_val`, which may be updated by multiple threads at the same time. Special care must be taken to avoid undefined behavior and ensure the correctness of the gradient computation while preserving as much of its parallelism as possible.

GPUs often have relatively small amounts of memory per thread. Moreover, the memory of GPUs commonly has complex performance characteristics, with global memory being slow but large, shared memory being fast but small, and the use of certain types of memory preventing the simultaneous use of large thread counts. Potential remedies involve either sacrificing generality, by rewriting HPC applications in a differentiable domain-specific language (DSL), or resorting to approaches such as numerical differentiation.

To leverage the potential performance benefits of reverse-mode AD without sacrificing generality, one requires a tool that is capable of both handling the complex performance characteristics of GPU architectures and generating code that maintains the correctness of the gradient without sacrificing the inherent parallelism of the original program. Unlike many other tools, Enzyme[1] [41] performs AD alongside the traditional optimization pipeline by performing differentiation within the LLVM compiler [38]. Thus, we can leverage existing code transformation infrastructure to build the requisite analyses and transformations for maintaining the correctness and performance of the corresponding gradient kernel. Furthermore, LLVM provides frontends for most commonly used languages including C/C++, Fortran, Julia, Rust, Swift, and TensorFlow and backends for different hardware architectures including CPU, NVIDIA GPUs [31, 48, 61], and AMD GPUs, allowing us to build reverse-mode AD for multiple languages and architectures.

Overall, our paper makes the following contributions:

- An algorithm for correctly generating gradients of GPU kernels and a corresponding proof sketch of correctness
- An extension to the Enzyme AD engine for LLVM that can generate gradients for GPU kernels written in either CUDA (NVIDIA) or ROCm (AMD)
- A collection of optimization passes for Enzyme/LLVM that allow generated GPU gradients to run efficiently on modern hardware
- A study demonstrating, for the first time, the feasibility of reverse-mode automatic differentiation of GPU kernels through the use of GPU and AD-specific optimizations (caching and recomputation).

[1]https://enzyme.mit.edu

## 2 RELATED WORK

AD tools that differentiate programs at runtime are often relatively straightforward to develop using, for example, operator overloading in C++ [6, 26, 30, 39, 50, 59]. Unfortunately, in reverse mode, they generally produce a large *tape* to store operations and intermediate values for subsequent reverse differentiation, which causes challenges with their memory footprint in real-world applications.

A more efficient, but more challenging type of AD uses a compile-time transformation to translate the source code for a given function evaluation into the derivative function evaluation. Several such tools have been developed for Fortran and C including ADIFOR [11], Tapenade [29], TAF [22], OpenAD [58], ADIC [12], and ADIC2 [43]. Unlike these tools, Enzyme is based on the LLVM compiler instead of an AD-specific framework and emits gradient programs in LLVM IR instead of the original source language. This approach allows Enzyme to benefit from the language support, optimizations, and maturity of the LLVM platform.

For differentiating codes running in distributed-memory environments, libraries such as the *Adjoinable MPI* library have been developed that reverse the nonblocking communication patterns in the original code [17, 57]. Other studies have presented reverse-mode AD for OpenMP codes [10, 15, 16, 20, 34, 35] or hybrid OpenMP/MPI codes [23]. Some studies [10, 23] have identified that reverse-mode AD creates potential write races on multicore CPU programs and suggest atomic updates or privatization as solutions.

Derivatives can be computed on GPUs for programs written in certain domain-specific languages (DSLs) such as PyTorch [45], Halide [46], TensorFlow [1], or JAX [14]. The AD approach used in these languages uses the structure of, and high-level knowledge about, programs that can be written in those DSLs and does not easily generalize to arbitrary programs written in a general-purpose language such as C or CUDA. Previous works have discussed AD or symbolic differentiation for programs that call CUDA kernels [24, 25]. Such works, however, do not present differentiation of the kernels themselves or else use the forward mode of AD [13, 47].

## 3 AUTOMATIC DIFFERENTIATION

This section provides a brief summary of automatic differentiation concepts that are relevant to this work. For a more thorough introduction, we refer to [28, 41, 44].

AD takes as its input a computer program $P$ that implements a mathematical function and produces a new program that computes the derivative, or gradient, of that function. AD tools are able to produce such a derivative by examining the individual instructions of $P$ (such as add or mul) and generating the corresponding partial derivatives of the instructions. By applying the chain rule of calculus, they then compute the derivative of the entire program by accumulating the partial derivatives all instructions of $P$. Any order of accumulating these derivatives is correct, but the order affects the efficiency, ease of implementation, and memory usage. Two particular strategies have become popular.

***Forward or tangent mode*** combines the derivatives of instructions in the order in which the original instructions are evaluated, resulting in the propagation of derivatives from an instruction's input(s) to its output. Consider the instruction $v = f(w, u)$. The derivative of its output, $\dot{v}$, can be evaluated by computing

$$\dot{v} = \frac{\partial f}{\partial w}\ \dot{w}\ +\ \frac{\partial f}{\partial u}\ \dot{u}.$$

For the overall program, the derivative of all outputs $z_0, \ldots, z_m$ with respect to one of its inputs $x$ can thus be computed by setting $\dot{x} = 1$ at the start of the program, and reading the final value of the differential or ***shadow*** $\dot{z}_0 \ldots \dot{z}_m$ at the end of the program. Computing the derivative with respect to multiple inputs requires a forward mode evaluation for every input. This is also true for numeric differentiation or finite differences, where a separate evaluation with a small perturbation for each input variable is required. Numeric differentiation has the added disadvantage of being less accurate, and requiring the choice of a step size.

***Reverse or adjoint mode*** combines the derivative of instructions in a ***reverse pass***, which computes the derivative or ***adjoint*** of the instructions in the reverse order of the original program, and propagates them from an instruction's outputs to its inputs. Considering the same instruction $v = f(w, u)$, the derivative of inputs $\bar{w}, \bar{u}$ can be evaluated by computing[2]

$$\bar{w}{+}= \frac{\partial f}{\partial w}\ \bar{v}; \quad \bar{u}{+}= \frac{\partial f}{\partial u}\ \bar{v}; \quad \bar{v} = 0.$$

The derivative of output $z$ with respect to any input $x$ can then be computed by setting $\bar{z} = 1$ prior to evaluating all the partial derivatives, then reading the final value of the shadow input $\bar{x}$. This allows a single evaluation of reverse mode to compute the gradient (derivative of output with respect to all inputs) in a single evaluation. Evaluating the derivative with respect to multiple outputs, however, requires an evaluation per output. In practice, programs with a large number of inputs, but few outputs (e.g. a loss function) dominate both scientific and machine learning use cases. Since reverse mode can compute derivatives in this case asymptotically faster than other methods, our work focuses entirely on reverse-mode AD.

Despite its attractiveness for practical applications, reverse mode AD is not without challenges, two of which are particularly relevant for this work. First, for a nonlinear instruction (such as $x^2$), one requires the original input to compute the derivative (in this case $2x$). While this is true for both forward and reverse modes, it is a challenge during the reverse pass. To provide the necessary inputs, the AD tool must evaluate all original instructions in an ***augmented forward pass*** and cache the required intermediate values (potentially causing a large memory footprint), or store only selected intermediate variables from which others can be recomputed (trading some memory for additional computation). Our work addresses analysis strategies to reduce the amount of storage needed, but does not address recomputation strategies, which are an active research subject on their own [5, 27, 60] and are beyond the scope of this work. Second, since the derivative evaluation occurs in a different order than the original program, parallelization strategies that are correct for the original program may not be correct for the derivative program, and special care needs to be taken to avoid data races. This and other challenges are addressed in Section 4.

[2]In reverse mode, the derivative adds to the shadow value $\bar{w}$ rather than setting it directly. This ensure the derivatives from all uses of $w$ are taken into account. The total derivative of $w$ is finalized when all partial derivatives have been accumulated. This is guaranteed to occur before the reverse of the instruction that defines $w$ as all users of $w$ must occur after $w$ in the original program and thus all adjoints that update $w$ must occur prior the reverse of $w$'s defining instruction. Since we are adding to the shadow, we must also initialize the shadow to zero. This is primarily done in the forward pass when creating the primal variable. To accommodate variables which are redefined (e.g. when in a loop), the shadow is again zero initialized after its value is propagated to the shadow inputs.

| **Memory load** | **Memory store** |
|---|---|
| `%res = load %ptr` | `store %ptr = %val` |
| **Reverse memory load** | **Reverse memory store** |
| `%tmp = load %d_res` | `%tmp = load %d_ptr` |
| `store %d_res = 0` | `store %d_ptr = 0` |
| `atomic %d_ptr += %tmp` | `load/store %d_val += %tmp` |

**Figure 2: Rules for memory operations. Shadow registers `d_res` and `d_val` are thread-local since they shadow thread-local registers. There is no risk of racing on thread-local data and no special handling required. Both `ptr` and shadow `d_ptr` might be raced on and require atomics in the adjoint of the load. If `ptr` (and consequently `d_ptr`) is proven to be thread-local or have constant memory, the atomic update can be replaced with a serial update or reduction, respectively.**

The reverse mode of automatic differentiation is closely related to the backpropagation algorithm for neural networks, and both have been implemented in DSLs such as PyTorch [45], TensorFlow [2], and others [18, 32, 37, 53]. These DSLs do not differentiate compute kernels directly, but expose high-level operations such as matrix multiply, and provide existing superoptimized GPU kernels for both the original function and its derivative. This approach is very effective for programs that can be written within these DSLs. For existing HPC applications or those that do not easily map to a DSL, this is unfortunately not an option. For this reason, there continues to be a need for AD tools such as Enzyme that can differentiate programs written in general purpose languages.

## 4 REVERSE-MODE AD FOR GPU KERNELS

Enzyme performs reverse-mode automatic differentiation over the LLVM intermediate representation (LLVM-IR). Since Enzyme is tightly integrated within the LLVM pipeline, it can differentiate any programming language with an LLVM frontend and can target any architecture that has an LLVM backend. Most importantly, this alleviates the need for DSLs or language restrictions to apply AD to code. Prior to this work, GPU kernels could not be differentiated in reverse mode without being rewritten in an explicitly differentiable DSL (e.g., PyTorch). To differentiate GPU kernels, we extend Enzyme to handle shared-memory accesses, avoid data races in the presence of concurrent reads in the primal, differentiate parallel control flow (e.g., `sync_threads`), and differentiate GPU-specific intrinsic functions (e.g., the LLVM-IR representation of the CUDA thread identifier `threadIdx.x`).

Enzyme first performs an ***activity analysis*** [9], which deduces what instructions and values in the function could impact the resulting gradient computation. For every active value, Enzyme creates a corresponding ***shadow memory*** location, which is used to store intermediate derivative values. For active function arguments, Enzyme expects the callee of the gradient function to pass in the shadow of each argument (see Section 4.4). We refer to prior work [41] for a more detailed explanation of Enzyme on serial programs. Here, we will focus on our contribution of synthesizing gradient functions for GPU kernels and the necessary changes and improvements to Enzyme.

## 4.1 GPU Memory-Aware Gradient Synthesis

**Case 1**
```
A:
  store %ptr
  barrier
B:
  store %ptr
```

**Case 2**
```
A:
  store %ptr
  barrier
B:
  load %ptr
```

**Case 3**
```
A:
  load %ptr
  barrier
B:
  store %ptr
```

**Gradient Case 1**
```
∇B:
  load %d_ptr
  store %d_ptr = 0
  barrier
∇A:
  load %d_ptr
  store %d_ptr = 0
```

**Gradient Case 2**
```
∇B:
  atomicAdd d_ptr

  barrier
∇A:
  load %d_ptr
  store %d_ptr = 0
```

**Gradient Case 3**
```
∇B:
  load %d_ptr
  store %d_ptr = 0
  barrier
∇A:
  atomicAdd %d_ptr
```

**Figure 3: Illustrations for the case analysis of the `barrier` instruction adjoint definition.**

The most challenging aspect of generating fast and correct gradient code from parallel code is reasoning about memory operations, especially on the different memory types of the GPU. Both NVIDIA and AMD GPUs have thread-local, shared (block-local), and global memory, as well as constant memory that cannot change during the execution of a kernel. We define rules for synthesizing correct gradients according to which kind of memory is accessed. We define the shadow of constant memory to be global memory, to ensure that the reverse pass is able to write the corresponding gradient to the shadow. Our approach requires that the primal code is determinacy race-free. Thus, we assume the appropriate use of atomic accesses and barriers (see Section 4.2).

Memory that is known to be thread-local cannot be accessed concurrently by multiple threads and is therefore equivalent to memory in serial AD. The gradient computation can access and update non-atomically without introducing a race.

In contrast, global and shared memory can be accessed concurrently by multiple threads in the primal. In the gradient computation this can cause concurrent write accesses, and thus races, if the updates are performed non-atomically (see Figure 1). The generic solution is to perform all accesses and updates in the reverse pass atomically. Such an approach, however, has severe performance downsides. Instead, we translate loads and stores of global- and shared-memory locations according to the rules displayed in Figure 2. That is, locations that are accessible by other threads are accessed atomically, while thread-local locations such as the thread-local shadow locations are accessed non-atomically. Further, we identify the special case where all threads in a block load from the same memory location in shared memory. In this case we employ an efficient block-level reduction computation that uses synchronous warp shuffle operations instead of atomic accesses.

## 4.2 Adjoints of Barriers

In GPU programming, barriers (e.g. `sync_threads` in CUDA) can synchronize the execution of threads within a warp or block. This is especially important in the presence of shared memory because it allows threads to communicate efficiently without memory races. We define the adjoint of barrier calls to be another barrier at the corresponding location in the reverse pass and show that this is sufficient by case analysis.

Given two consecutive code blocks *A* and *B*, separated by a barrier, that write or read the same memory location, the barrier provides four distinct memory guarantees:

(1) All stores in A must complete prior to a store in B.
(2) All stores in A must complete prior to a load in B.
(3) All loads in A must complete prior to a store in B.
(4) All loads in A must complete prior to a load in B.

Figure 3 shows minimal examples for cases 1–3; all four cases are discussed in the following.

**Case 1: Store, Barrier, Store** In the primal, the store in `B` will clobber the store in `A`, causing subsequent loads to see the value stored in `B`. As a result, we must ensure that the gradient will increment only the derivative of the value stored in `B` and not the value stored in `A`. The barrier in the reverse pass ensures that only ∇`B` could read a nonzero adjoint from `d_ptr`, as desired.

**Case 2: Store, Barrier, Load** For the reverse code to be correct we require the load of `d_ptr`, which is the adjoint of the primal load, to happen after all `atomicAdd` operations, which are the results of the primal store. The barrier in the reverse pass is sufficient to guarantee that ordering.

**Case 3: Load, Barrier, Store** We require that all of the stores of `d_ptr`, which are caused by the primal load, complete prior to any `atomicAdd`, which is the adjoint of the primal store. The barrier in the reverse pass will ensure this. Note that there cannot be a race in ∇`B` because that would require a preexisting race in `B`, which is violating our precondition.

**Case 4: Load, Barrier, Load** In the case of a barrier between two loads, the barrier operation is superfluous and can be removed with no change in semantics. Therefore, no extra considerations are needed.

## 4.3 GPU Intrinsics and Shared-Memory Allocations

The gradient is independent of most GPU-specific built-ins and intrinsics (e.g., threadIdx.x) since they are known to LLVM to be pure, that is, independent of memory. Furthermore, most intrinisics are *inactive* and can consequently be recomputed without special handling by Enzyme. Exceptions include barriers and special memory accesses (e.g., tensor core or atomic memory operations). The former is described in Section 4.2, and the latter can be implemented in a manner similar to traditional memory operations.

Shared-memory allocations require explicit handling to provide adjoint locations, also allocated in shared memory, that act as shadows. In LLVM-IR, a shared-memory allocation is represented as a global value with an explicit address space that is effectively uninitialized at kernel launch time. Therefore, in addition to the shadow allocation, we generate initialization code that is executed at the very beginning of differentiated kernels.

```
__device__ void inner(float* a, float* x, float* y) {
  y[threadIdx.x] = a[0] * x[threadIdx.x];
}
__device__ void __enzyme_autodiff(void*, ...);

__global__ void gradient_kernel(float* a, float* da,
                                float* x, float* dx,
                                float* y, float* dy) {
  __enzyme_autodiff((void*)inner, a, da, x, dx, y, dy);
}

// Synthesized by Enzyme on the LLVM-IR level from the
// definition of the inner function.
__device__ void gradient_inner(float* a, float* da,
                               float* x, float* dx,
                               float* y, float* dy) {
  y[threadIdx.x] = a[0] * x[threadIdx.x];

  float dy_tmp = dy[threadIdx.x];
  dy[threadIdx.x] = 0.0f;

  float dx_tmp = a[0] * dy_tmp;
  atomic { dx[threadIdx.x] += dx_tmp; }

  float da_tmp = x[threadIdx.x] * dy_tmp;
  atomic { da[0] += da_tmp; }
}
```

**Figure 4: A simple GPU function, `inner`, that is differentiated by Enzyme within the CUDA kernel `gradient_kernel` (top). A high-level representation of the synthesized gradient Enzyme would generate is shown as `gradient_inner` (bottom). The call to `__enzyme_autodiff` is replaced by a call to the newly generated derivative function.**

## 4.4 Usage

Enzyme is available as a plugin for the LLVM "core" compiler component. When Enzyme is loaded into compilers such as Clang, an optimization pass is enabled that acts on calls to the `__enzyme_autodiff` function.[3] The first argument to this function is the primal that is differentiated, followed by the primal arguments interleaved with shadow locations for pointers. For usage within CUDA, one calls `__enzyme_autodiff` from inside a device kernel that is launched through the normal CUDA API. Figure 4 shows how the GPU function `inner` is differentiated and how the synthesized gradient, `gradient_inner`, looks conceptually[4].

## 5 OPTIMIZATIONS

GPU architectures feature multiple kinds of memory that differ in their access latency, visibility, and size. While registers and shared memory are much faster than global memory, they are limited resources on GPUs and are allocated for a kernel at launch time. If a kernel requests a large number of registers or a large shared-memory allocation, the effective available parallelism (occupancy) of the kernel is lowered to fulfill the request. This can become a bottleneck for applications since a major benefit of using GPUs is their high throughput offered by plentiful parallelism. To achieve good performance, Enzyme must consequently consider trade-offs between using slower global memory or increasing the use of registers and shared memory, which may result in fewer kernel instances being run simultaneously.

Like all reverse-mode AD tools, Enzyme may need to preserve values generated in the forward pass for use in the reverse. If a value is available in the reverse pass, for example, if the memory that holds it was not overwritten, Enzyme will simply use it. When a memory location holding a value required for the reverse pass is modified, however, Enzyme must ensure that the value is preserved, or cached, an action that inevitably requires additional storage.

While it is generally beneficial to reduce the amount of memory used to cache values, doing so is especially important for GPU execution. In general, the number of memory locations that need to be cached is not known at compile time. Consequently, Enzyme has to cache values in thread-local storage, allocated through the dynamic allocation function malloc. In CUDA, malloc is backed by global memory and cached in the L1 cache. Global memory is substantially slower to access than registers or shared memory, which is why cache use can dramatically increase the kernel runtime. Moreover, excessive caching can require more than the available GPU heap memory and prevent the program from being run at all. Since memory size and bandwidth are the primary bottlenecks, most of our optimizations aim to minimize global memory accesses. Our experimental results in Section 6 demonstrate that significant GPU-specific and AD-specific optimizations are necessary to run the reverse pass in a reasonable time. Below, we briefly explain the most important optimizations that we use for this work.

***Alias Analysis.*** Alias analysis [4, Ch. 12] is fundamental to Enzyme's ability to determine whether an instruction can be recomputed or must be cached. Instructions that do not access memory are trivially recomputable. For instructions that read memory, Enzyme uses LLVM's alias analysis pipeline to determine whether the value is overwritten before it is required in the reverse pass. Depending on the quality of available alias information, for example, from types and restrict qualifiers, this can reduce the number of cached values significantly. However, if there are potentially aliasing pointers (e.g. two plain pointer arguments), Enzyme is required to assume that writes to one might modify any element read through the other. In the worst case, this uncertainty can force Enzyme to cache all read accesses of a constant input array.

In our analysis, we found that common math functions, such as cos, are seen as being able to write to *any* global memory and thus potentially overwrite most memory locations. LLVM models `libm` implementations of these functions as `writeonly` because they can set the global `errno` variable, assuming the user does not explicitly disable this potential side effect. The situation is different for CUDA code since there is no `libm` available. Instead, Clang will effectively map all available math functions onto respective CUDA builtin functions, for example, `__nv_cos`. Since the LLVM analyses and optimizations are not aware of these CUDA-specific functions, they are conservatively assumed to read and write arbitrary memory. For the sake of Enzyme's cache, we allow alias analysis to assume that common math functions do not act as barriers to recomputation.

[3]As Julia is JIT compiled, Enzyme.jl can explicitly call Enzyme's ABI for creating derivatives, rather than loading Enzyme into an existing optimization pipeline.
[4]Note that while we show CUDA code for readability, Enzyme acts on the lower level LLVM-IR that can be targeted by various languages and parallel programming models.

**(a)**

```
for (int i=0; i<N; i++) {
    for (int j=0; j<M; j++) {
        use(array[j]);
    }
}
overwrite(array);
```

**(b)**

```
double *cache = new double[N*M];
for (int i=0; i<N; i++) {
    for (int j=0; j<M; j++) {
        cache[i*M+j] = array[j];
        use(array[j]);
    }
}
overwrite(array);
diffe_overwrite(array);
for (int i=N-1; i>=0; i--) {
    for (int j=M-1; j>=0; j--) {
        diffe_use(cache[i*M+j]);
    }
}
delete[] cache;
```

**(c)**

```
double *cache = new double[M];
memcpy(cache, array, M*sizeof(double));
for (int i=0; i<N; i++) {
    for (int j=0; j<M; j++) {
        use(array[j]);
    }
}
overwrite(array);
diffe_overwrite(array);
for (int i=N-1; i>=0; i--) {
    for (int j=M-1; j>=0; j--) {
        diffe_use(cache[j]);
    }
}
delete[] cache;
```

**Figure 5: In (a), there is a sample program that uses values of an array in a loop nest. The loads of the array cannot be hoisted by LICM. The array is overwritten outside of the loop nest. Enzyme would require caching a value for every execution of the load instruction, as shown in (b) and using $\Theta(NM)$ memory. Using the cache LICM optimization, the cache could be hoisted outside the loop as shown in (c), requiring only $\Theta(M)$ memory.**

Another significant barrier to performance is the aliasing behavior of `sync_threads`. In order to ensure correctness for multi-threaded GPU programs, LLVM's aliasing properties of architecture-specific `barrier` intrinsics assume that `barrier` can read and write to most memory locations. For the same reasons as above, this assumption forces Enzyme to unnecessarily cache values. We extend Enzyme to define a `barrier` instruction `S` as having the aliasing behavior of all instructions that precede `S` until it reaches another `barrier` or the start of the kernel being differentiated.

***Loop-Invariant Cache.*** Enzyme caches the results of individual instructions rather than memory ranges. This approach can be more efficient for general programs, especially if memory access patterns are sparse. This can be problematic, however, in cases where many instructions load from the same piece of memory that must be cached. Enzyme relies on LLVM-based optimizations such as common sub-expression elimination (CSE) and loop-invariant-code-motion (LICM) [42, Sec. 13.2] to remove such equivalent accesses in the original program and subsequently prevent unnecessary caching. In several cases, however, the LLVM optimizations may not be legal, or even beneficial for the original code, but would otherwise result in a large amount of unnecessary caching.

For example, consider the program shown in Figure 5(a). The load cannot be optimized by LICM since it depends on the innermost iteration variable `j`. If the load is required for a reverse-pass computation, Enzyme must cache every result of the load as shown in Figure 5(b), resulting in an $\Theta(NM)$ cache. However, we notice that the array is only potentially overwritten outside of the loop nest, and we could have instead chosen to simply cache the total size of the memory used ($\Theta(M)$) as in Figure 5(c). This cache optimization detects scenarios where it is legal and profitable to cache loads from a parent loop nest, thereby reducing the total cache.

***Equivalent Load Cache.*** Similar to how the loop-invariant cache optimization remedies issues where LICM may not optimize the initial code to reduce the cache, we also present a cache-variant of common sub-expression elimination. Consider two loops that both load from an array. Because the loops are not fused, these loads cannot be deduplicated by common sub-expression elimination. Consequently, Enzyme would have to create two separate caches. However, since both of these load from the same memory without a potential write in between, we can instead cache the array once and use it during the reverse pass in both places.

***Cache Forwarding.*** GPU programs commonly use shared memory as a cache for global memory when it may be used by many threads. This is highly beneficial because accesses from shared memory are much faster than loads from global memory. If that shared memory is overwritten, however, it may need to be cached for the reverse pass. The original global memory it is derived from, however, may not have been overwritten. In this case, instead of allocating a cache to preserve the overwritten values in shared memory, we can simply reload the underlying memory the shared memory is acting as a cache for, preventing an unnecessary allocation of global memory for the cache. An additional though yet unimplemented extension to this optimization is to reuse the faster shared memory as a cache for the reverse pass rather than having to load from the slower global memory.

***PHI Unwrapping.*** In addition to load and call instructions that may not be recomputable, Enzyme may also have to cache PHI instructions. PHI instructions occur when the current basic block has multiple potential predecessors. The PHI instruction forwards a value from the actual predecessor that just branched to the current block, preventing recomputation and requiring caching.

This optimization aims to compute an equivalent value to the PHI by determining a condition `C` that determines the actual predecessor of the basic block. The PHI node can then be recomputed by recomputing the condition `C` and selecting the corresponding value the PHI node would have when coming from the predecessor corresponding to `C`. Computing `C` can be done by traversing the function's control-flow graph and attempting to identify a chain of conditions to branch instructions that lead to the PHI node from a given predecessor. This cannot always be done at compile-time but

**(a)**
```
use(x[0] + y[0]);
overwrite(x, y);
```

**(b)**
```
double x_cache = x[0];
double y_cache = y[0];
use(x[0] + y[0]);
overwrite(x, y);
diffe_overwrite(x, y);
diffe_use(x_cache[i] + y_cache[i]);
```

**(c)**
```
double sum_cache = x[0] + y[0];
use(x[0] + y[0]);
overwrite(x, y);
diffe_overwrite(x, y);
diffe_use(sum_cache);
```

**Figure 6: (a) A sample program that loads two variables x and y and then perform some computation with the result. These variables are subsequently overwritten and thus would require caching to be available in the reverse pass. A naive cache algorithm would produce the code in (b) in which both overwritten memory locations x and y are cached. As shown in (c), one could instead cache the sum since neither x nor y is individually necessary to compute the gradient.**

nevertheless allows Enzyme to avoid caching many PHI instructions in unnecessary allocations.

***Allocation Optimizations.*** Enzyme performs most cache allocations on the heap, backed by global memory. By running the heap-to-stack optimization pass, we can lower a heap allocation into a stack allocation and subsequently open the possibility of promoting the stack allocation to individual registers. Additionally, Enzyme may make several separate allocations for different instruction caches. A function call (such as a call to `malloc` or `free`) is expensive on the GPU. We provide a further optimization that coalesces several individual allocations into a larger allocation, thereby reducing the overhead of allocating cache memory.

***Recompute versus Cache Heuristics.*** When Enzyme deduces that a value `V` is required in the reverse pass, Enzyme explicitly caches all loads, calls, and PHI instructions necessary to compute `V`. We extend Enzyme with a heuristic to instead directly cache the value being recomputed, rather than the loads necessary to recompute it, if we predict that this will result in a smaller amount of cached memory as shown in Figure 6. We also extend this heuristic to find the minimal set of values to cache by determining a minimum branch cut between values that must be cached and instructions that require values from the forward pass. In general, solving for the *optimal* cache size is difficult to do at compile time because many relevant parameters such as loop bounds may not be known.

***Loop Bound Calculation.*** Enzyme frequently computes the bounds of loops, for example, to determine the size of cache space allocations or to index into the cache. Enzyme piggybacks on top of LLVM's existing scalar evolution analysis to attempt to statically deduce the size of loops. This allows Enzyme to allocate the required cache memory in advance. However, not all loops have statically known bounds. For these dynamically sized loops, Enzyme must continuously reallocate the cache inside the loop to ensure sufficient memory exists to contain the values from all iterations. When the total number of iterations is not statically analyzable, Enzyme adds a variable to cache the count for use in index computations.

Consequently, it is desirable for Enzyme to statically deduce the bounds of loops. However, LLVM's analysis passes must be conservative and account for behavior like potential integer wraparound, causing hard-to-analyze bounds on seemingly simple loops. Enzyme extends LLVM's scalar evolution to take advantage of a key fact: if one is indeed evaluating code in the reverse pass, none of the forward-pass loops could have been infinite loops. When computing bounds for cache sizes, we can consequently add the extra fact that the loop is not infinite, allowing Enzyme to statically compute bounds of additional loops.

***Register Locality.*** In contrast to virtual instruction sets like LLVM, physical architectures have a fixed set of registers available for computation. To map a computation onto a physical instruction set such as that used by a GPU, one must perform register allocation to map the virtual registers used by LLVM to a fixed set of physical registers. When there are insufficient registers available to represent all virtual registers, the compiler must spill the instruction into a stack allocation (which on NVIDIA GPUs spills to the L1 cache and subsequently global memory). Therefore, it is crucial for Enzyme to maximize the locality of virtual register uses to avoid spilling. By default, Enzyme reuses a value from the forward pass if it dominates its potential use in the reverse pass, because it will always be available without an explicit allocation. This scheme is problematic for the GPU, however, because it may increase the lifetime of registers, leading to spilling and increased global memory use.

To remedy this situation, Enzyme will choose to recompute loads from shared memory if there is register pressure. While a load from shared memory is certainly slower than reusing a register, it is still faster than a load from global memory in a potential spill.

***Inlining.*** Choosing to inline or call a function can have substantial performance implications. Inlining a function may be beneficial because it may allow Enzyme to combine loads or otherwise reduce redundant cache allocations through the loop-invariant cache or equivalent load cache optimizations. On the other hand, by calling a function rather than inlining it, Enzyme will explicitly recompute data structures generated by the function being called in the reverse pass. This action can increase register locality and may require fewer instructions to recompute PHI nodes since there are fewer potential predecessors.

## 6 EVALUATION

We evaluate our approach on five established GPU-based HPC proxy applications:

- CUDA-based RSBench [55] and XSBench [56], two implementations of Monte Carlo neutron transport algorithms
- An extended version of the CUDA lattice-Boltzmann method (LBM) solver from the Parboil benchmark suite [54], with applications in computational fluid dynamics

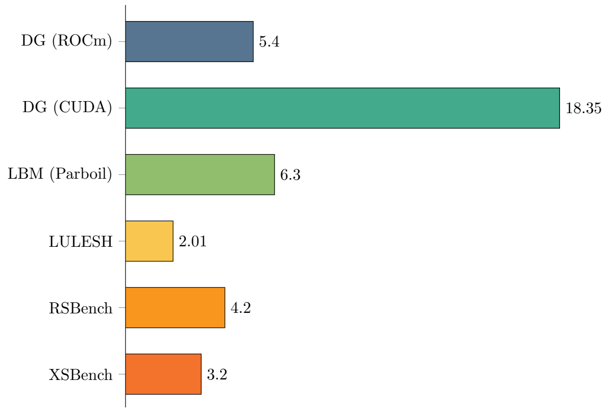

**Figure 7: AD overhead of the benchmark applications, as compared with a single evaluation of the forward pass. An overhead of $N$ can be read as saying that collecting the gradients of all inputs (as well as running the original code) is equivalent to running the original code $N$ times.**

- CUDA-based Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) code [36], a proxy application for computational fluid dynamics solvers
- A discontinuous-Galerkin (DG) volume integral[5] [3] kernel as used in the pure Julia [8] climate code ClimateMachine.jl[6] [52] and implemented for both CUDA and AMD GPUs

## 6.1 Setup

For each application, we time just the evaluation of the code being differentiated, excluding time taken for device memory initialization and transfer or other calling code. For CUDA kernels, we explicitly increase the size of the device heap to 1 GB. RSBench, XSBench, and the CUDA.jl version of DG were evaluated on an NVIDIA 2080 Super. LBM was evaluated on an NVIDIA V100. LULESH was evaluated on an NVIDIA RTX A6000. The AMDGPU.jl version of DG was evaluated on an AMD Vega 64. Benchmarks were tested with LLVM main at commit `8dab25954b0acb53731c4aa73e9a7f4f98263030`, Julia 1.6, and Enzyme at commit `ec75831a8cb0`. The benchmark suite is available at https://github.com/wsmoses/Enzyme-GPU-Tests.

All benchmarks were evaluated a minimum of five times, taking the geometric mean as the final result. For each benchmark we evaluated the original kernel and the combined forward/reverse pass generated by Enzyme (Figure 7); the combined forward and reverse pass with various optimizations described in Section 5 disabled (Figure 10); the compile times of the benchmarks (Figure 14); and the scalability of the gradients compared to the original code (Figures 11 and 12).

With the exception of the LBM benchmark (see below), modifying a benchmark to enable differentiation simply required allocating and initializing shadow arrays (to store the output gradients), and creating a kernel which calls `__enzyme_autodiff` on the kernel to be differentiated, as demonstrated in Figure 4.

[5]https://github.com/lcw/Heptapus.jl
[6]https://github.com/CliMA/ClimateMachine.jl/

```
void kern(float* src, float* dst) {
    streamCollide<<<...>>>(src, dst);
}

void lbm(int nTimeSteps, float* src, float* dst) {
    for (unsigned int i=0; i<nTimeSteps/2; i++) {
        kern(src, dst);
        kern(dst, src);
    }
}
```

**Figure 8: Simplified version of the computation within LBM. The `kern` function calls a GPU kernel that iterates the simulation one timestep forward in time, storing the result in `dst`. The `lbm` CPU function calls the GPU kernel until all iterations have completed. The iteration must happen outside the kernel to ensure that all threads from one timestep have completed prior to performing another timestep.**

The correctness of the generated gradients was verified by comparing with numeric differentiation. Since our benchmarks have too many parameters to use numeric differentiation effectively, only a few inputs per benchmarks were tested.

## 6.2 Benchmark Descriptions

***RSBench and XSBench.*** RSBench and XSBench are U.S. Department of Energy proxy applications that represent the core computation of Monte Carlo simulations within particle transport algorithms such as in OpenMC [49]. The majority of the runtime of XSBench is spent in memory operations with a semi-random access pattern. By calculating neutron cross-sections with the multipole method, RSBench trades off several magnitudes of memory in exchange for a significant amount of computation to unpack the data. Together, RSBench and XSBench allow us to differentiate both compute-bound and memory-bound applications, respectively.

***Lattice Boltzmann Method (LBM).*** LBM is a particle-based fluid dynamics simulation method. It works by modeling fluid density on a lattice (grid) and in each time step performing a streaming step (allowing fluid to flow into adjacent grid cells) and a collision step (which models the interaction of fluids flowing into a particular cell from neighboring cells). This so-called stream-collide sequence is responsible for the majority of the computational cost of typical LBM solvers and is implemented in the CUDA version of Parboil LBM in a method called `performStreamCollide_kernel`. CPU driver code calls this kernel in a loop to advance the simulation by several timesteps, as shown in Figure 8.

Unlike the other benchmarks tested, where the entire function being differentiated was on the GPU, differentiating LBM requires the differentiation of heterogeneous programs. Since LLVM does not yet support modules which contain both CPU and GPU code, we perform differentiation in two steps. First, we use Enzyme to generate an augmented forward and reverse pass for the GPU kernel. The forward pass is equivalent to the original function, saving any data that is required for the reverse pass and may be overwritten. The forward and reverse pass of the GPU kernels can then be imported into the CPU code by using Enzyme's support for

(a)

```
// CPU Code
void aug_kern(float* src, float *dsrc,
              float* dst, float* ddst) {
  void* tape = Allocator.allocate(...);
  aug_streamCollide<<<...>>>(src, dsrc, dst, ddst, tape);
}
void grad_kern(float* src, float *dsrc,
               float* dst, float* ddst, void* tape) {
  grad_streamCollide<<<...>>>(src, dsrc, dst, ddst, tape);
  Allocator.free(tape);
}
__attribute__((enzyme(aug_kern, grad_kern)))
void kern(float* src, float* dst);

void grad_lbm(int nTimeSteps, float* src, float* dsrc,
                              float* dst, float* ddst) {
  __enzyme_autodiff(lbm, nTimeSteps, src, dsrc, dst, ddst);
}
```

(b)

```
// GPU Code
__global__
void aug_streamCollide(float* src, float* dsrc,
                       float* dst, float* ddst, void** tape) {
  size_t idx = threadIdx.x + ...;
  tape[idx] = __enzyme_augmentfwd(streamCollide, src, dsrc,
                                  dst, ddst);
}

__global__
void grad_streamCollide(float* src, float* dsrc,
                        float* dst, float* ddst, void** tape) {
  size_t idx = threadIdx.x + ...;
  __enzyme_reverse(streamCollide, src, dsrc,
                   dst, ddst, tape[idx]);
}
```

**Figure 9: Differentiation of the combined CPU+GPU computation in LBM. The code in (a) represents host code, which differentiates the overall function `lbm`, defined in Figure 8. The `kern` function is annotated with custom forward and reverse passes `aug_kern` and `grad_kern`. These functions allocate a tape and call the `aug_streamCollide` and `grad_streamCollide` kernels, which are generated by Enzyme in (b).**

custom derivatives. The heterogeneous AD setup is demonstrated in Figure 9. Note that while we demonstrate this shim layer for clarity, in practice this can be simplified for end users through the use of advanced compiler transformations or macros.

***LULESH.*** LULESH [36] is an unstructured explicit shock hydrodynamics solver, which was initially introduced as a proxy application for computational fluid dynamics on high-performance computing systems and has since been employed as a proxy application for complex fluid dynamics codes. LULESH emulates complex hydrodynamic solvers by splitting the computational domain into volumetric elements on an unstructured mesh. This allows LULESH to mimic the complex data movement characteristics of unstructured data structures. All measurements were analyzed with NVIDIA NSight Compute to discern the individual measurements of the gradient *ApplyMaterialPropertiesAndUpdateVolume* kernel from the general application runtime.

***Discontinuous Galerkin (DG).*** The discontinuous-Galerkin volume integral[3] kernel is part of a fluid dynamics simulation model. It is written in Julia, and we use `CUDA.jl` [7] and `AMDGPU.jl` [51] in combination with Enzyme.jl [41] to synthesize and execute the kernel and its derivative. The code features GPU-specific features, such as shared memory, and is memory bound. We modified the original code to use noncoherent memory loads in the case of `CUDA.jl` and constant memory loads in the case of `AMDGPU.jl`.

## 6.3 Results

The original Enzyme paper [41] demonstrated that by embedding AD within the compiler, one can perform AD after optimization which is on average 4.2× faster than AD before optimization. Since prior tools perform AD at a source level, they must perform AD prior to any compiler optimizations. Although there exist no tools that we can compare against that perform reverse-mode AD on GPU kernels, we attempted to perform a similar ablation analysis here to see what a tool not implemented within a compiler might be able to achieve, if one were to be written. Without applying standard LLVM optimizations prior to AD, RSBench and XSBench take an indefinite amount of time to run. LULESH has an overhead of 2979.1× without preprocessing optimizations. LBM is able to be differentiated without preprocessing optimizations for two iterations, but exhausts GPU memory on anything larger (scaling tests use 50-600 iterations). In order to legalize Julia code for the GPU (such as the ROCm and CUDA DG codes), it is necessary to run the LLVM optimization pipeline, along with Julia's custom optimization passes. We therefore conclude that the ability to run optimizations alongside AD is in fact a precondition of successful reverse-mode AD of general GPU programs.

Overall, the combined forward and gradient generated by Enzyme have a reasonable overhead when compared with that of the forward pass (Figure 7). RSBench and XSBench have a $3-4\times$ overhead due to the need to cache intermediate computations from the forward pass. Similarly, LBM must cache the current state variables every iteration leading to an overhead of 6.3×. The kernel evaluated in LULESH does not need to cache additional values, and as a result the 2.01× overhead is spent performing the corresponding gradient computations. The DG benchmark has a 5.4× overhead when run on AMD, primarily from the additional computation, whereas it has a 18× overhead on CUDA as it quickly exhausts the amount of available registers and the CUDA assembler decides to spill a large number of registers into global memory.

***AD and GPU-Specific Optimizations.*** To evaluate the effectiveness of the optimizations described in Section 5, we evaluated all benchmarks with several AD and GPU-specific optimizations being successively disabled. Not all benchmarks benefit from the same optimizations, and the order in which compiler optimizations are applied can dramatically impact performance [33]. For each

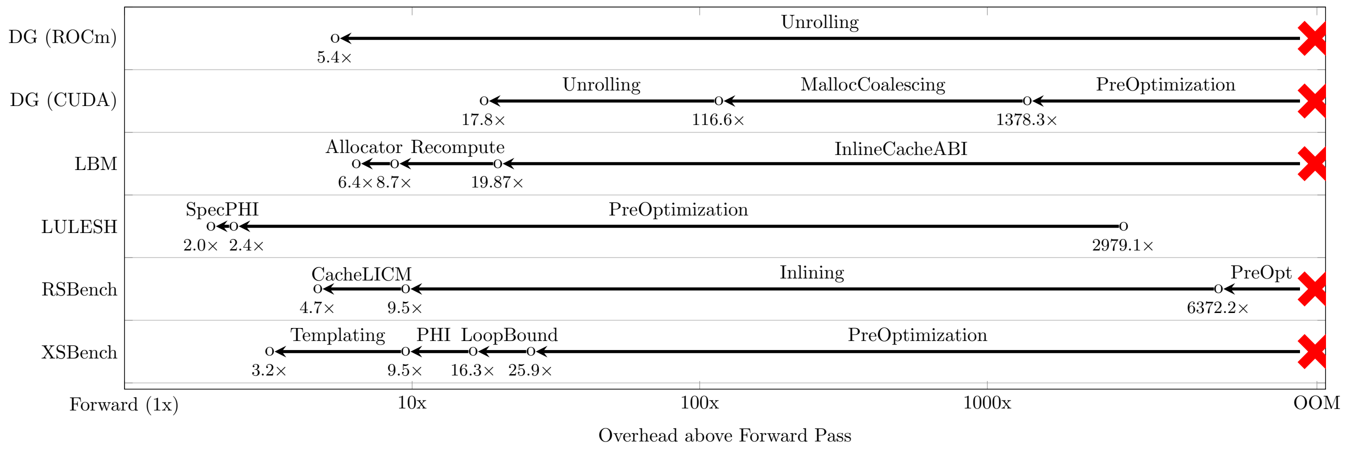

**Figure 10: Overhead of selectively disabling AD and GPU-specific optimizations described in Section 5. OOM indicates running out of memory or an indefinite runtime. Each dot represents the overhead of AD compared to the forward pass alone.**

benchmark, we visualize a path through the exponentially large optimization space that attempts to enable each optimization when it will have the largest impact on performance. The results of this analysis are shown in Figure 10. An end user trying to maximize their performance wouldn't explore all optimization combinations/paths, instead simply enabling all optimizations. As disabling optimizations quickly blows up the runtime of the program, the ablation analysis of benchmarks was run at a smaller test size to ensure the computation completed in a reasonable time where necessary.

For the ROCm DG kernel, an unrolling optimization was necessary to allow Enzyme to create the gradient without caching any additional values. Without unrolling, the GPU was unable to allocate sufficient device memory to succeed.

For the CUDA DG kernel, simply applying the standard Julia+LLVM optimization pipeline enabled the gradient to run, though at a 1378.3× overhead. Running an optimization that coalesced multiple allocations into a single `malloc` call reduced this runtime to 116.6×. Like in the ROCm case, applying unrolling eliminates any need to cache values, reducing the overhead to 17.8×.

For ablation analysis, we ran the LBM kernel for 150 iterations. The use of an efficient CPU to GPU calling convention for caching values was necessary for the gradient to run on a problem of this size. Applying the improved recompute vs cache heuristic allowed Enzyme to detect that it could cache a double which representing a sum, rather than the individual of overwritten values. This analysis reduced the size of the cache from 80 bytes per thread to 20 bytes per thread. As a result, the AD overhead was reduced from 19.87× to 8.7×. Finally, using a LIFO allocator rather than `cudaMalloc` to allocate cache memory brought the AD overhead down to 6.4×.

For ablation analysis, LULESH was run on a computational domain size of $90^3$. Applying just LLVM optimizations prior to AD brought the LULESH gradient overhead down to 2.4× from 2979.1×. As the LULESH kernel was particularly branch heavy, enabling speculative execution of $\phi$ predecessors when recomputing values in the reverse pass reduced the AD overhead down to 2.01×.

Running RSBench on a problem size of 10,200, LLVM optimizations alone resulted in an overhead of 6374×. By applying additional inlining, this overhead was reduced to 9.5× as LLVM could optimize between functions, enabling Enzyme to eliminate redundant values being cached, as well as use a more efficient intraprocedural caching infrastructure. Enabling the loop invariant cache and equivalent load cache optimizations reduced the overhead down to 4.7×.

Running XSBench on a problem size of 17,000,000 with LLVM optimizations, the overhead was 25.9×. Allowing Enzyme to avoid caching loop bounds when it can prove that all its instructions are inactive, drops the overhead to 16.3×. Performing PHI restructuring reduces the overhead to 9.5×. Passing the mode of simulation through a C++ template eliminates code generation of helper routines from different simulation modes and reduces the overhead to 3.2×. This leads to fewer branches in the forward pass and allows Enzyme to avoid analogous branches in the reverse pass.

***Scalability.*** We compare the scalability of our approach in two ways. First, we consider applications where increasing the problem size increases the number of threads, while maintaining constant work per thread. We plot the overhead as a function of problem size for DG and LULESH, XSBench, and RSBench in Figure 11. DG on CUDA, LULESH, XSBench, and RSBench maintain a constant overhead as the problem size increases. DG on AMD's overhead increases at the start but quickly asymptotes. When the problem size is increased in the LBM benchmark, the amount of work and number of kernel calls increase without increasing the number of threads. As demonstrated by Figure 12, the overhead quickly asymptotes as the additional setup required by Enzyme gets amortized across a larger number of iterations.

***LULESH Case Study.*** Automatic differentiation of LULESH's compute kernels is a prime example of the importance of running optimizations prior to reverse-mode automatic differentiation on GPUs. While the generated gradient has a 2979.1× overhead without any LLVM optimizations prior to AD, this is reduced to 2.4× by simply running LLVM's standard optimization pipeline. This does not require deep changes to LULESH or manual tuning. Using all the optimizations described in Section 5 resulted in a reduction of the AD overhead to ∼ 2.01×. Because of the effectiveness of

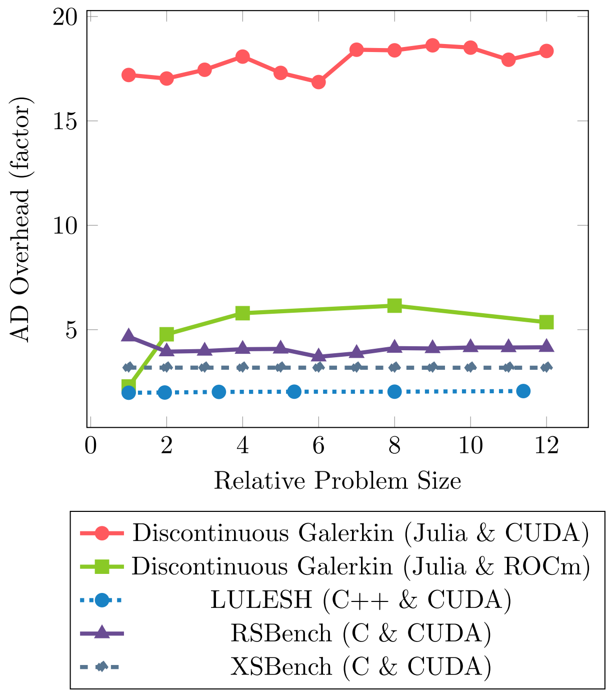

**Figure 11: Overhead of Enzyme compared with the forward pass as the problem size and number of threads increase, with constant work per thread.**

the optimizations and low overhead, we looked at the memory access patterns in depth to understand the impact Enzyme and its optimizations had on the memory system of the GPU.

In Figure 13, we analyze the memory characteristics of the *ApplyMaterialPropertiesAndUpdateVolume* kernel, using NVIDIA’s NSight-Compute analyzer. We use the memory workload analysis as a guide to evaluate the performance of the synthesized gradient kernel and judge whether there are potentially missed optimizations, or common access patterns within the gradient kernel. For this kernel, the profile shows that there is an $\sim 50\%$ increase in memory traffic when performing gradient calculations. If excessive caching or register spilling occurred, we would have seen an increase in *Local* memory traffic. This performance report is typical of an efficient gradient kernel, which is reflected in the low AD overhead of 2.01×.

***Discontinuous Galerkin (DG) Case Study.*** We evaluated the DG kernel on both AMD and NVIDIA GPUs. The NVIDIA variant shows an overhead of 18× versus an overhead of 5.4× for the AMD variant. Performance analysis of the NVIDIA implementation unveiled two bottlenecks in the gradient kernel. The first bottleneck was caused by a large number of values reused from the forward pass. This created excessive register spilling and correspondingly increased global memory traffic. Second, some atomic increment operations on shared memory were heavily contended. Surprisingly, the AMD implementation performs much better. We hypothesize that AMD is faster because the AMDGPU LLVM backend directly optimizes for the target architecture and can perform optimizations such as target register allocation. In contrast, the NVIDIA

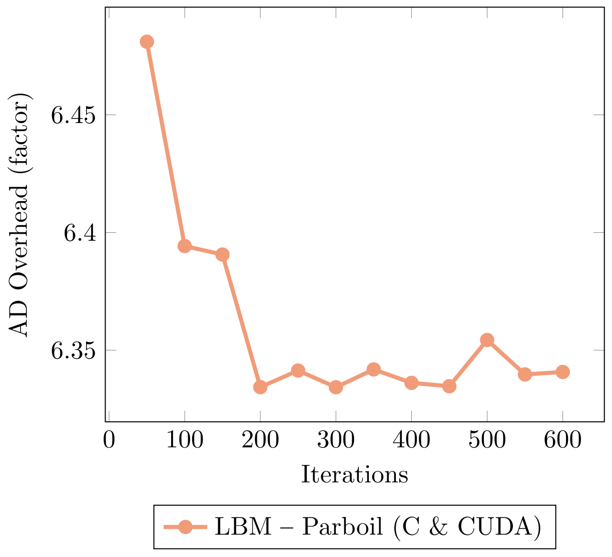

**Figure 12: Overhead of Enzyme compared with the forward pass where work is increased while maintaining a constant number of threads.**

LLVM backend targets NVIDIA’s virtual instruction set architecture NVPTX and leaves register allocation to `ptxas`.

Enzyme allows the user to specify whether the gradient should be calculated with respect to an argument. We used the DG kernel to verify that applying Enzyme with all arguments set to be constant (not differentiated), would not incur any overhead.

***Compile Time.*** We compare the time spent to compile kernels with and without the gradient generated by Enzyme. In practice, when running large simulations the compile time is negligible compared with the runtime. Nevertheless, it is useful to verify that also compiling the derivatives does not substantially change the program’s overall compile time. For the four C/C++ benchmarks (LBM, LULESH, RSBench, and XSBench), we measured just the compile time of the file that contained the kernel being differentiated. This is then compared with compiling the same source file, but also generating all the requisite derivative information. This involves creating additional functions, running a second round of optimizations, and running the backend code generator for the additional kernel(s). For codes that just compile the combined forward and reverse passes (LULESH, RSBench, and XSBench), we would expect a $\sim 3\times$ overhead as in addition to the original kernel, there is now a second kernel which is twice the size (containing the forward and reverse pass). For codes in which a forward and reverse pass are requested separately, we would expect a $\sim 4\times$ overhead to account for the additional augmented forward pass, and the split reverse pass (which contains its own forward and reverse pass). These compile times are all within expectation.

The two Julia codes must be analyzed separately. As Julia is a JIT, Enzyme.jl works by running its own additional compilation within Julia’s runtime and performing foreign function calls into Enzyme loaded as a dynamic library As a result, a direct comparison is not meaningful. Nevertheless we demonstrate that the “forward” time,

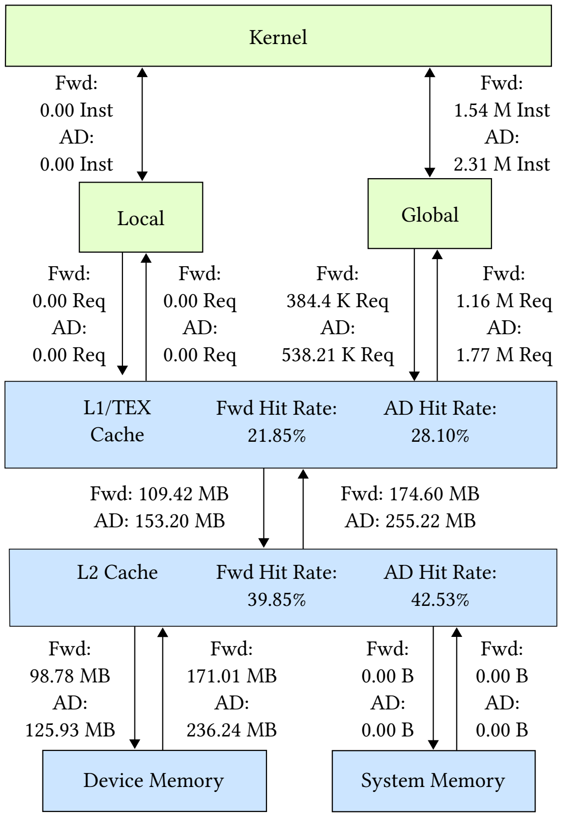
Kernel
Fwd: 0.00 Inst AD: 0.00 Inst
Fwd: 1.54 M Inst AD: 2.31 M Inst
Local
Global
Fwd: 0.00 Req AD: 0.00 Req
Fwd: 0.00 Req AD: 0.00 Req
Fwd: 384.4 K Req AD: 538.21 K Req
Fwd: 1.16 M Req AD: 1.77 M Req
L1/TEX Cache
Fwd Hit Rate: 21.85%
AD Hit Rate: 28.10%
Fwd: 109.42 MB AD: 153.20 MB
Fwd: 174.60 MB AD: 255.22 MB
L2 Cache
Fwd Hit Rate: 39.85%
AD Hit Rate: 42.53%
Fwd: 98.78 MB AD: 125.93 MB
Fwd: 171.01 MB AD: 236.24 MB
Fwd: 0.00 B AD: 0.00 B
Fwd: 0.00 B AD: 0.00 B
Device Memory
System Memory

**Figure 13: Memory workload analysis for LULESH at size $135^3$ comparing the original code (Fwd) to the gradient (AD).**

taken to compile the original kernel, is comparable with the "AD" time to perform a foreign function call to `libEnzyme.so`, which generates the derivative runtime function.

## 7 CONCLUSION

By extending Enzyme, an AD tool for LLVM, we have created the first AD tool capable of generating gradients of GPU kernels without rewriting entire applications with a differentiable DSL. Reverse-mode differentiation of GPU kernels adds several challenges including potential data races caused by the GPU's parallelism and the GPU's complex performance characteristics. We demonstrate an algorithm for differentiating GPU-based parallel control flow and other intrinsics that ensures the correctness of the resultant gradients. To maximize performance of the generated gradients, we introduce several novel AD and GPU-specific optimizations. Through various ablation analyses, we show how without these optimizations reverse-mode GPU AD is intractable in practice. We demonstrate reasonable performance and scalability on several applications relevant to the HPC community.

There exist several avenues for future work. Many of the optimizations described in Section 5, especially those involving caching, could make better use of shared memory, when available. For example, with rare exception, Enzyme currently maintains the GPU schedule described in the forward pass for use in the reverse. One could imagine allowing Enzyme to reschedule a kernel in such a way that minimizes potential races and therefore allows better performance. Moreover, Enzyme currently identifies constant shared-memory indices as the only scenarios where it can perform a reduction rather than falling back to an atomic increment. Extending Enzyme to more aggressively identify locations where it can perform a reduction rather than atomics can result in additional performance boosts, especially in kernels that, like the DG kernel, make heavy use of shared memory (see Section 6.3). Extending Enzyme to support Forward and Mixed-Mode [7] AD may provide potential performance boosts by allowing Enzyme to choose the differentiation algorithm expected to perform fastest for a particular workload. Moreover, support for parallelism demonstrated here in the context of GPUs can be extended to support both CPU parallelism and distributed frameworks such as MPI to allow Enzyme to efficiently differentiate a wider variety of HPC applications.

| Test | Forward | AD | Overhead |
|---|---|---|---|
| LBM | 1.54 | 5.65 | 4.32× |
| LULESH | 8.34 | 23.82 | 2.86× |
| RSBench | 14.99 | 33.29 | 2.22× |
| XSBench | 15.9 | 23.5 | 1.48× |
| Julia DG CUDA | 0.50 | 3.41 | 6.82× |
| Julia DG AMD | 1.12 | 2.56 | 2.29× |

**Figure 14: Compile time in seconds of the source file with and without derivatives.**

## ACKNOWLEDGMENTS

Many thanks to Paul Hovland (ANL) for plenty of thoughtful suggestions on automatic differentiation. The authors would like to thank Lilly Chin (MIT) for their help in performing numerous hardware upgrades, without which this work would not have been possible.

William S. Moses was supported in part by a DOE Computational Sciences Graduate Fellowship DE-SC0019323. Valentin Churavy was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR0011-20-9-0016, and in part by NSF Grant OAC-1835443. Ludger Paehler was supported in part by the German Research Council (DFG) under grant agreement No. 326472365. This research was supported in part by Los Alamos National Laboratories grant 531711; in part by the Applied Mathematics activity within the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program, under contract number DE-AC02-06CH11357; in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software,applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative; and in part by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein. We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.

## REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283.

[3] Daniel S Abdi, Lucas C Wilcox, Timothy C Warburton, and Francis X Giraldo. 2019. A GPU-accelerated continuous and discontinuous Galerkin non-hydrostatic atmospheric model. *The International Journal of High Performance Computing Applications* 1 (2019), 81–109. https://doi.org/10.1177/1094342017694427 arXiv:https://doi.org/10.1177/1094342017694427

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (second ed.). Addison-Wesley.

[5] Guillaume Aupy, Julien Herrmann, Paul Hovland, and Yves Robert. 2016. Optimal multistage algorithm for adjoint computation. *SIAM Journal on Scientific Computing* 38, 3 (2016), C232–C255.

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

[7] Tim Besard, Christophe Foket, and Bjorn De Sutter. 2018. Effective Extensible Programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems* (2018). https://doi.org/10.1109/TPDS.2018.2872064 arXiv:1712.03112 [cs.PL]

[8] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. 2017. Julia: A fresh approach to numerical computing. *SIAM review* 59, 1 (2017), 65–98.

[9] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. 1992. ADIFOR–generating derivative codes from Fortran programs. *Scientific Programming* 1, 1 (1992), 11–29.

[10] Christian Bischof, Niels Guertler, Andreas Kowarz, and Andrea Walther. 2008. Parallel Reverse Mode Automatic Differentiation for OpenMP Programs with ADOL-C. In *Advances in Automatic Differentiation*, Christian H. Bischof, H. Martin Bücker, Paul D. Hovland, Uwe Naumann, and J. Utke (Eds.). Springer, 163–173. https://doi.org/10.1007/978-3-540-68942-3_15

[11] Christian Bischof, Peyvand Khademi, Andrew Mauer, and Alan Carle. 1996. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science and Engineering* 3, 3 (1996), 18–32.

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

[13] Johannes Blühdorn, Nicolas R. Gauger, and Matthias Kabel. 2020. AutoMat – Automatic Differentiation for Generalized Standard Materials on GPUs. arXiv:2006.04391 [cs.CE]

[14] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. 2020. JAX: composable transformations of Python+NumPy programs, 2018. *URL http://github. com/-google/jax* 4 (2020), 16.

[15] H. Martin Bücker, Bruno Lang, Dieter an Mey, and Christian H. Bischof. 2001. Bringing Together Automatic Differentiation and OpenMP. In *Proceedings of the 15th ACM International Conference on Supercomputing, Sorrento, Italy, June 17–21, 2001*. ACM Press, New York, 246–251. https://doi.org/10.1145/377792.377842

[16] H. M. Bücker, B. Lang, A. Rasch, C. H. Bischof, and D. an Mey. 2002. Explicit Loop Scheduling in OpenMP for Parallel Automatic Differentiation. In *Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications, Moncton, NB, Canada, June 16–19, 2002*, J. N. Almhana and V. C. Bhavsar (Eds.). IEEE Computer Society Press, Los Alamitos, CA, 121–126. https://doi.org/10.1109/HPCSA.2002.1019144

[17] J.I. Cardesa, L. Hascoët, and C. Airiau. 2020. Adjoint computations by algorithmic differentiation of a parallel solver for time-dependent PDEs. *Journal of Computational Science* 45 (2020), 101155. https://doi.org/10.1016/j.jocs.2020.101155

[18] Filipe de Avila Belbute-Peres, Kevin Smith, Kelsey Allen, Josh Tenenbaum, and J Zico Kolter. 2018. End-to-end differentiable physics for learning and control. *Advances in neural information processing systems* 31 (2018), 7178–7189.

[19] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research* 12, 7 (2011).

[20] Michael Förster. 2014. *Algorithmic Differentiation of Pragma-Defined Parallel Regions: Differentiating Computer Programs Containing OpenMP*. Ph.D. Dissertation. RWTH Aachen.

[21] Roger Ghanem, David Higdon, and Houman Owhadi. 2017. *Handbook of uncertainty quantification*. Vol. 6. Springer.

[22] R. Giering, T. Kaminski, and T. Slawig. 2005. Generating efficient derivative code with TAF: Adjoint and tangent linear Euler flow around an airfoil. *Future Generation Computer Systems* 21, 8 (2005), 1345–1355. https://doi.org/10.1016/j.future.2004.11.003

[23] Ralf Giering, Thomas Kaminski, Ricardo Todling, Ronald Errico, Ronald Gelaro, and Nathan Winslow. 2005. Tangent Linear and Adjoint Versions of NASA/GMAO's Fortran 90 Global Weather Forecast Model. In *Automatic Differentiation: Applications, Theory, and Implementations*, H. Martin Bücker, George F. Corliss, Paul D. Hovland, Uwe Naumann, and Boyana Norris (Eds.). Springer, 275–284.

[24] Markus Grabner, Thomas Pock, Tobias Gross, and Bernhard Kainz. 2008. Automatic Differentiation for GPU-Accelerated 2D/3D Registration. In *Advances in Automatic Differentiation*, Christian H. Bischof, H. Martin Bücker, Paul D. Hovland, Uwe Naumann, and J. Utke (Eds.). Springer, 259–269. https://doi.org/10.1007/978-3-540-68942-3_23

[25] Felix Gremse, Andreas Höfter, Lukas Razik, Fabian Kiessling, and Uwe Naumann. 2016. GPU-accelerated adjoint algorithmic differentiation. *Computer physics communications* 200 (2016), 300–311.

[26] Andreas Griewank, David Juedes, and Jean Utke. 1996. Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software (TOMS)* 22, 2 (1996), 131–167.

[27] Andreas Griewank and Andrea Walther. 2000. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software (TOMS)* 26, 1 (2000), 19–45.

[28] Andreas Griewank and Andrea Walther. 2008. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM.

[29] L. Hascoët and V. Pascual. 2013. The Tapenade Automatic Differentiation tool: Principles, Model, and Specification. *ACM Transactions On Mathematical Software* 39, 3 (2013). http://dx.doi.org/10.1145/2450153.2450158

[30] Robin J Hogan. 2014. Fast reverse-mode automatic differentiation using expression templates in C++. *ACM Transactions on Mathematical Software (TOMS)* 40, 4 (2014), 1–16.

[31] Justin Holewinski. 2011. PTX back-end: GPU programming with LLVM. In *The Ohio State University. LLVM Developer's Meeting. November*, Vol. 18.

[32] Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. 2020. DiffTaichi: Differentiable Programming for Physical Simulation. In *International Conference on Learning Representations*.

[33] Qijing Huang, Ameer Haj-Ali, William Moses, John Xiang, Ion Stoica, Krste Asanovic, and John Wawrzynek. 2019. Autophase: Compiler phase-ordering for hls with deep reinforcement learning. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 308–308.

[34] J.C. Hückelheim, P.D. Hovland, M.M. Strout, and J.-D. Müller. 2018. Parallelizable adjoint stencil computations using transposed forward-mode algorithmic differentiation. *Optimization Methods and Software* 33, 4-6 (2018), 672–693. https://doi.org/10.1080/10556788.2018.1435654 arXiv:https://doi.org/10.1080/10556788.2018.1435654

[35] Jan Hückelheim, Paul Hovland, Michelle Mills Strout, and Jens-Dominik Müller. 2019. Reverse-mode algorithmic differentiation of an OpenMP-parallel compressible flow solver. *The International Journal of High Performance Computing Applications* 33, 1 (2019), 140–154. https://doi.org/10.1177/1094342017712060 arXiv:https://doi.org/10.1177/1094342017712060

[36] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes,number = LLNL-TR-641973*. Technical Report. 1–9 pages.

[37] Dmitrii Kochkov, Jamie A. Smith, Ayya Alieva, Qing Wang, Michael P. Brenner, and Stephan Hoyer. 2021. Machine learning accelerated computational fluid dynamics. arXiv:2102.01010 [physics.flu-dyn]

[38] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE, 75–86.

[39] Johannes Lotz, Klaus Leppkes, and Uwe Naumann. 2011. dco/c++-derivative code by overloading in C++. *Aachener Informatik Berichte (AIB-2011-06)* (2011).

[40] Dougal Maclaurin, David Duvenaud, and Ryan Adams. 2015. Gradient-based hyperparameter optimization through reversible learning. In *International conference on machine learning*. PMLR, 2113–2122.

[41] William S. Moses and Valentin Churavy. 2020. Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients. In *Advances in Neural Information Processing Systems*. Vol. 33. 12472–12485.

[42] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

[43] Sri Hari Krishna Narayanan, Boyana Norris, and Beata Winnicka. 2010. ADIC2: Development of a component source transformation system for differentiating C and C++. *Procedia Computer Science* 1, 1 (2010), 1845–1853. https://doi.org/10.1016/j.procs.2010.04.206 ICCS 2010.

[44] Uwe Naumann. 2011. *The art of differentiating computer programs: an introduction to algorithmic differentiation*. SIAM.

[45] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS 2017 Workshop Autodiff*.
[46] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *ACM SIGPLAN Notices* 48, 519–530. https://doi.org/10.1145/2499370.2462176
[47] Jarrett Revels, Tim Besard, Valentin Churavy, Bjorn De Sutter, and Juan Pablo Vielma. 2018. Dynamic Automatic Differentiation of GPU Broadcast Kernels. *CoRR* abs/1810.08297 (2018). arXiv:1810.08297 http://arxiv.org/abs/1810.08297
[48] Helge Rhodin. 2010. A PTX code generator for LLVM. *Oct* 29 (2010), 1–63.
[49] Paul K Romano and Benoit Forget. 2013. The OpenMC Monte Carlo particle transport code. *Annals of Nuclear Energy* 51 (2013), 274–281.
[50] Max Sagebaum, Tim Albring, and Nicolas R Gauger. 2019. High-performance derivative computations using codipack. *ACM Transactions on Mathematical Software (TOMS)* 45, 4 (2019), 1–26.
[51] Julian Samaroo, Valentin Churavy, Wiktor Phillips, Jason Barmparesos, Julia TagBot, Takafumi Arakaki, Stephan Antholzer, Alessandro, chriselrod, and Tim Besard. 2021. JuliaGPU/AMDGPU.jl: v0.2.6 for Zenodo. https://doi.org/10.5281/zenodo.4677701
[52] Tapio Schneider, Shiwei Lan, Andrew Stuart, and João Teixeira. 2017. Earth System Modeling 2.0: A Blueprint for Models That Learn From Observations and Targeted High-Resolution Simulations. *Geophysical Research Letters* 44, 24 (2017), 12,396–12,417. https://doi.org/10.1002/2017GL076101 arXiv:https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1002/2017GL076101
[53] Samuel Schoenholz and Ekin Dogus Cubuk. 2020. JAX md: a framework for differentiable physics. *Advances in Neural Information Processing Systems* 33 (2020).
[54] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012).
[55] John R. Tramm, Andrew R. Siegel, Benoit Forget, and Colin Josey. 2014. Performance Analysis of a Reduced Data Movement Algorithm for Neutron Cross Section Data in Monte Carlo Simulations. In *EASC 2014 - Solving Software Challenges for Exascale*. Stockholm. https://doi.org/10.1007/978-3-319-15976-8_3
[56] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XS-Bench – The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*. Kyoto. https://www.mcs.anl.gov/papers/P5064-0114.pdf
[57] J. Utke, L. Hascoët, P. Heimbach, C. Hill, P. Hovland, and U. Naumann. 2009. Toward adjoinable MPI. In *2009 IEEE International Symposium on Parallel Distributed Processing*. 1–8. https://doi.org/10.1109/IPDPS.2009.5161165
[58] Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch. 2008. OpenAD/F: A Modular, Open-Source Tool for Automatic Differentiation of Fortran Codes. *ACM Trans. Math. Software* 34, 4 (2008), 18:1–18:36. https://doi.org/10.1145/1377596.1377598
[59] A. Walther and A. Griewank. 2012. Getting started with ADOL-C. In *Combinatorial Scientific Computing*, U. Naumann and O. Schenk (Eds.). Chapman-Hall CRC Computational Science, Chapter 7, 181–202.
[60] Qiqi Wang, Parviz Moin, and Gianluca Iaccarino. 2009. Minimal repetition dynamic checkpointing algorithm for unsteady adjoint calculation. *SIAM Journal on Scientific Computing* 31, 4 (2009), 2549–2567.
[61] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. 2016. Gpucc: An Open-Source GPGPU Compiler. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (Barcelona, Spain) *(CGO '16)*. Association for Computing Machinery, New York, NY, USA, 105–116. https://doi.org/10.1145/2854038.2854041

## A ARTIFACT DESCRIPTION/ARTIFACT EVALUATION

Our paper explored how reverse-mode automatic differentiation could be performed on existing GPU kernels by extending the Enzyme automatic differentiation engine for LLVM. The paper demonstrates how various novel optimization passes and differentiation rules for GPU instructions allow for correct and efficient evaluations of the gradient.

The paper evaluated 6 benchmarks: XSBench (CUDA), RSBench (CUDA), Parboil LBM (CUDA), LULESH (CUDA), DG (CUDA), DG (AMD). Of these benchmarks, the AMD and CUDA DG codes are Julia-based where as the remainder are C++/CUDA. All benchmarks are available on Github https://github.com/wsmoses/Enzyme-GPU-Tests at commit `d4daa0cd5931494bc4dfbfffc7874c3b00b1e3a4` with a DOI of `10.5281/zenodo.5147573`. Evaluation of these benchmarks allowed the paper to demonstrate the efficiency of the generated gradients in comparison to the original code, the impact of the novel optimizations, the scalability of the generated gradients, and the correctness of the tool.

To run the benchmarks used in the paper, we first need to build the LLVM compiler toolchain before we can subsequently link the compiler plugin of Enzyme against our built LLVM version. For our compiler toolchain we used LLVM 13 (main) at commit `8dab25954b0acb53731c4aa73e9a7f4f98263030`. To install LLVM, please follow the following steps:

```
$ cd ~
$ git clone https://github.com/llvm/llvm-project
$ cd llvm-project
$ git checkout 8dab25954b0acb53731c4aa73e9a7f4f98263030
$ mkdir build && cd build
$ cmake ../llvm -DLLVM_ENABLE_PROJECTS="clang" \
-DLLVM_TARGETS_TO_BUILD="X86;NVPTX" \
-DCMAKE_BUILD_TYPE=Release -G Ninja
$ ninja # This may take a while
# clang is now be available
# in ~/llvm-project/build/bin/clang
```

We now must build an Enzyme based off of our chosen LLVM version. We use Enzyme at commit `ec75831a8cb0170090c36`.

```
$ cd ~
$ git clone https://github.com/wsmoses/Enzyme
$ cd Enzyme/enzyme
$ git checkout ec75831a8cb0170090c366f8da6e3b2b8
$ mkdir build && cd build
$ cmake ../enzyme -DLLVM_DIR=/path/to/llvm/build \
-DCMAKE_BUILD_TYPE=Release -G Ninja
$ ninja
# ClangEnzyme-13.so will now be available in
~/Enzyme/enzyme/build/Enzyme/ClangEnzyme-13.so
```

Some of the C++ benchmarks require a custom CUDA libdevice (the implementation of various CUDA intrinsics). This is to remedy an issue within LLVM that prevents common math functions from being identified as LLVM intrinsics (this is being worked on upstream). For the default CUDA installation, libdevice can be found at `/usr/local/cuda/nvvm/libdevice.10.bc`. The following code snippet describes how to replace a libdevice file, assuming you are using CUDA 11.2. The instructions are similar for a different CUDA installation, with the path changed accordingly. Note that you may need to be root to perform the change, and that you should always make a backup of your previous libdevice.

```
# Save a copy of your current libdevice
$ sudo cp /usr/local/cuda-11.2/nvvm/libdevice/libdevice.10.bc\
 /usr/local/cuda-11.2/nvvm/libdevice/libdevice.10.bc.old
$ sudo cp /path/to/new/libdevice.10.bc\
 /usr/local/cuda-11.2/nvvm/libdevice/libdevice.10.bc
```

We have created Python3 scripts to ease setting up and running our experiments. They will attempt to deduce appropriate paths given the following environmental variables. In the event that there is an issue, you likely may need to change `bench.py` or the `Makefile` for the test as appropriate. All of the `bench.py` benchmarking scripts follow the same structure.

```
# The index of the CUDA GPU desired
$ export DEVICE=1
# The path to the CUDA installation
$ export CUDA_PATH=/usr/local/cuda-11.2
# The path to the Clang++ binary we built above
$ export CLANG_PATH=/path/to/llvm/build/bin/clang++
# The path to the Enzyme plugin we built above
$ export ENZYME_PATH=\
/path/to/Enzyme/enzyme/build/Enzyme/ClangEnzyme-13.so
```

Let's now clone the benchmark suite.

```
$ git clone https://github.com/wsmoses/Enzyme-GPU-Tests
```

The benchmark suite folder breaks down into the following structure:

- DG, a Discontinuous Galerkin benchmark (CUDA & ROCm in Julia)
- LBM, a Lattice Boltzmann benchmark (CUDA in C++)
- LULESH, a Lagrangian Hydrodynamics benchmark (CUDA in C++)
- RSBench, a Monte Carlo Particle Transport benchmark (CUDA in C++)
- XSBench, a Monte Carlo Particle Transport benchmark (CUDA in C++)

We can now enter one of the 4 C++ test directories (XSBench, RSBench, LBM, LULESH) and run the corresponding benchmark.

```
$ cd Enzyme-GPU-Tests/LBM
$ python3 bench.py
# output of benchmark times printed out here
```

The `bench.py` script will run first an ablation analysis that enables or disables differentiation, along with several optimizations. The result of these tests will be the execution time of the gradient and/or original kernel. The script will then run by scaling tests for both the gradient and original kernel by evaluating on increasing problem sizes. Some benchmarks (XSBench, LBM, LULESH, DG (CUDA)) will end by printing out the derivative as computed by both numeric differentiation and Enzyme. These will include `"VERIFY=yes"` as part of the run line. All other run lines will contain the execution time of that benchmark. Be aware that LULESH's ablation analysis includes benchmarks configurations, which do not perform compiler optimizations and are hence significantly slower than the other benchmarks.

XSBench and RSBench require the libdevice found in Enzyme-GPU-Tests/libdevice1, and were run using CUDA 11.2 on an NVIDIA 2080 Super.

LBM uses the packaged libdevice from NVIDIA and was run using CUDA 11.3 on an NVIDIA V100.

LULESH uses the libdevice found in Enzyme-GPU-Tests/libdevice2 and was run using CUDA 11.2 on an NVIDIA A6000. The LULESH benchmark suite furthermore relies on NVIDIA's N-Sight compute utility (NCU). NCU has known issues with access to the GPU Performance counters, which you will need to benchmark the gradient kernel of LULESH. If you should run into this issue please have a look at the following documentation to remedy it https://developer.nvidia.com/nvidia-development-tools-solutions-err-nvgpuctrperm-nvprof.

Odd performance results or a compiler error is a potential indicator of using an incorrect libdevice.

For example, when compiling one of the ablation tests of RS-Bench without the correct libdevice, one may see the following error when running `bench.py`:

```
cannot handle (augmented) unknown intrinsic
  %5 = tail call i32 @llvm.nvvm.d2i.hi(double %0) #21
error in backend: (augmented) unknown intrinsic
clang-13: error
```

The two DG tests were run using Julia 1.6. Julia at this version must be found in your path before being able to run the Julia tests. To obtain a working Julia installation see https://julialang.org/downloads/ and follow the provided installation instructions.

DG (CUDA) was run with the libdevice found in Enzyme-GPU-Tests/libdevice1 on CUDA 11.2 by an NVIDIA 2080 Super.

DG (AMD) was run on an AMD Vega 64.

We have provided a similar `bench.py` script for DG. While printed in a different format (CSV-style), it contains the same information about runtimes for both ablation and scaling as the C++ CUDA tests (DG AMD does not have an ablation analysis as it does not run without all optimizations applied).

```
$ cd Enzyme-GPU-Tests/DG/cuda
$ python3 bench.py
```

Note that the numeric verification may come earlier in the script's output, and should look something like this:

```
# Enzyme derivative as the first element of tuple
# followed by numeric approximation on the right
(dQ.dval[1], (o2 - o1) / 0.0001) =\
(-1.105959f0, [-1.10626220703125])
```

The forward pass alone in the CSV-style output are denoted as the "primal" rows, whereas the derivative runtimes are marked as "all_dub".

The DG tests may require additional setup. For example, you see an output like below (note that this may also occur if you try to run DG (AMD) on a system without the relevant AMD libraries available).

```
Warning: HSA runtime has not been built,\
runtime functionality will be unavailable.
Please run Pkg.build("AMDGPU") and reload AMDGPU.
```

You may then need to explicitly run various setup routines within Julia's package manager. To fix the Julia setup for the test, perform the following to enter an interactive shell.

```
$ cd Enzyme-GPU-Tests/DG/rocm
$ julia --project=.
julia> using Pkg; Pkg.build("AMDGPU")
```