Automatic Differentiation on CUDA

Jay Patel, Tanvi Karandikar

May 5, 2023

URL: 15618-cuda-autodiff.netlify.app

Abstract

We have implemented automatic differentiation on GPU (CUDA) and compared it with a CPU version and OpenMP version. We performed a detailed analysis of the systems' performance characteristics. Automatic differentiation will be supported for first-order partial derivatives of functions that are commonly used in Deep Learning i.e addition, multiplication, transpose, power, matrix multiplication, etc. We have compared performance of the library for 3 settings: **Single core CPU**, **OpenMP thread parallelism**, and **GPU (CUDA) SIMD parallelism**.

1 Background

Deep learning frameworks rely heavily on tensor operations, such as matrix multiplication and addition, for training neural networks through error backpropagation. Automatic differentiation is a key component of this process, as it allows for efficient computation of gradients for the training process. Automatic differentiation in itself does not benefit from parallelism, however tensor operations – which take up most of the compute time during training – can be parallelized. The key data structures in this project are tensors, which are multi-dimensional arrays of floating-point numbers. The part that is computationally expensive and could benefit from parallelization is the tensor operations, which take up most of the compute time during training. The workload can be broken down into individual tensor operations, which can be parallelized using SIMD instructions on CPUs or using GPU threads that execute the same instruction on different elements of the tensor in parallel.

1.1 Background: Automatic Differentiation

Automatic differentiation is a key component of many machine learning and optimization algorithms that require the computation of partial derivatives. It is a computational technique that allows for the efficient and accurate evaluation of gradients of functions, even when the functions are complex and nonlinear. Automatic differentiation involves breaking down a function into a sequence of elementary/primitive operations and applying the chain rule to compute the derivative of the function.

There are two modes of automatic differentiation: forward mode and reverse mode. Forward mode automatic differentiation computes the derivative of a function with respect to a single input variable, while holding all other input variables constant. It proceeds by evaluating the function and its derivatives at a single point, and then propagating these values through the sequence of elementary operations in the function.

Reverse mode automatic differentiation, on the other hand, computes the derivative of a function with respect to all of its input variables simultaneously. It proceeds by first computing the function and its derivatives at a single point, and then propagating these values backwards through the sequence of elementary operations in the function. Reverse mode automatic differentiation is particularly useful in the context of deep learning, where the number of input variables can be very large and the cost of computing the gradients can be prohibitive using traditional methods.

Concretely, given a function mapping an N-dimensional tensor to a scalar (1 dimension) value, $F : \mathbb{R}^D \to \mathbb{R}$ that is defined as the composition of four primitive functions (like matrix addition, multiplication, etc) , $F = D \circ C \circ B \circ A$. We decompose the function down so that we can utilize values generated in intermediate steps for the purpose of calculating gradients (or more formally Jabobians):

$$F(x) = y, (x \in \mathbb{R}^D, y \in \mathbb{R}) \tag{1}$$

where $y = C(b)$, $b = B(a)$, $a = A(x)$.

To calculate the partial derivatives of F with respect to a, b, c, d, and x we can use the chain rule and the formula given in equation (1):

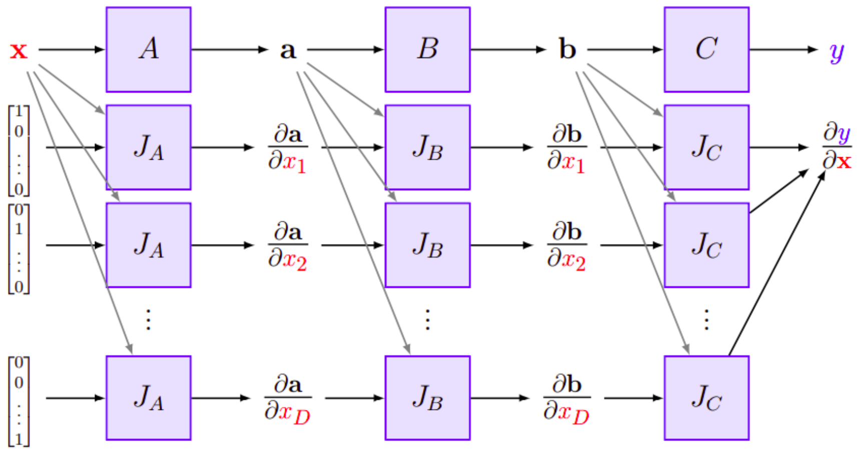
$$\frac{\partial F}{\partial c} = \frac{\partial y}{\partial c} \ = D'(c) \tag{2}$$

$$\frac{\partial F}{\partial b} = \frac{\partial y}{\partial c}\frac{\partial c}{\partial b} \ = D'(c)C'(b) \tag{3}$$

$$\frac{\partial F}{\partial a} = \frac{\partial y}{\partial c}\frac{\partial c}{\partial b}\frac{\partial b}{\partial a} \ = D'(c)C'(b)B'(a) \tag{4}$$

$$\frac{\partial F}{\partial x} = \frac{\partial y}{\partial c}\frac{\partial c}{\partial b}\frac{\partial b}{\partial a}\frac{\partial a}{\partial x} \ = D'(c)C'(b)B'(a)A'(x) \tag{5}$$

The functions A', B', C', and D' are used to compute the Jacobians of A, B, C, and D, respectively. These Jacobians represent the partial derivatives of the output of each primitive function with respect to its input. By chaining these Jacobians together using the chain rule, we can calculate the partial derivative of F with respect to each intermediate variable a, b, c, and d. This allows us to efficiently compute gradients of complex functions.



$\mathbf{x}$
A
$\mathbf{a}$
B
$\mathbf{b}$
C
y
J_A
J_B
J_C
$\frac{\partial \mathbf{a}}{\partial x_1}$
$\frac{\partial \mathbf{b}}{\partial x_1}$
$\frac{\partial y}{\partial \mathbf{x}}$
$\frac{\partial \mathbf{a}}{\partial x_2}$
$\frac{\partial \mathbf{b}}{\partial x_2}$
$\frac{\partial \mathbf{a}}{\partial x_D}$
$\frac{\partial \mathbf{b}}{\partial x_D}$

Figure 1: Forward mode

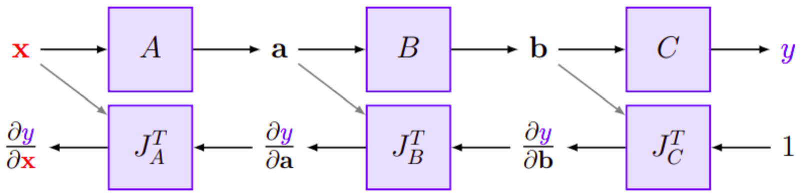
In forward mode, the Jacobians are calculated with respect to the input variable, for example, $\frac{\partial b}{\partial x}$. Since $x \in \mathbb{R}^D$ is a vector, this Jacobian contains D times as many entries as the corresponding value of b. In reverse mode, we calculate the Jacobians with respect to the output variable, for example, $\frac{\partial y}{\partial b}$. Since $y \in \mathbb{R}$ is a scalar, this Jacobian contains only as many values as b. Therefore, reverse mode is more efficient for evaluating the gradient of a vector-to-scalar function, once the primitive Jacobians $A'(x)$, $B'(a)$, $C'(b)$, and $D'(c)$ have been evaluated.

In forward-mode, we compute Jacobians like $\frac{\partial a}{\partial x}$, $\frac{\partial b}{\partial x}$, which correspond to the derivatives of each of the intermediate variables with respect to the input x. We accumulate these values by left-multiplying the previous step's Jacobian by the current primitive Jacobian. In contrast, reverse-mode differentiation works by accumulating Jacobians in the other direction, from output to input. It computes values such as $\frac{\partial y}{\partial a}$, $\frac{\partial y}{\partial b}$, which correspond to the derivatives of the output y with respect to each of the intermediate variables. These values are accumulated by right-multiplying the previous step's Jacobian by the current primitive Jacobian.

If the input x is a vector in $\mathbb{R}^D$ and the output y is a scalar, reverse-mode differentiation is more efficient than forward-mode because it accumulates values that are a factor of D smaller. This means that the computation is less complex and requires fewer calculations. For our implementation, we have used reverse mode automatic differentiation. For most operations, each element is independent of other elements in the same matrix – This is a good criteria for parallelism. Using GPU SIMD, we aim to parallelize as many computations as possible while efficiently managing memory usage.

2 Approach

In the context of automatic differentiation, the overall speedup of the system is of interest rather than the speedup achieved for individual operations. While speeding up a single



$\mathbf{x}$
A
$\mathbf{a}$
B
$\mathbf{b}$
C
y
$\frac{\partial y}{\partial \mathbf{x}}$
J_A^T
$\frac{\partial y}{\partial \mathbf{a}}$
J_B^T
$\frac{\partial y}{\partial \mathbf{b}}$
J_C^T
1

Figure 2: Reverse mode

operation can be relatively straightforward – implementing a kernel, managing memory becomes complicated when dealing with multiple operations that produce tensors required for backpropagation. To reduce memory copy overhead, tensors must be kept in local memory until backpropagation is complete. Additionally, since tensor sizes are not fixed, a mechanism must be developed to automatically determine block size and thread count for each kernel operation along with data locality – especially for the matrix multiplication operations. Therefore, to fully optimize the system, we need to consider the interdependence of operations, memory management, and runtime optimization of the kernel operations.

Figure 3, shows a high-level design for an automatic differentiation library. The library is divided into two major components: *autodiff.core* and *autodiff.tensorlib*. *tensorlib* is reponsible for the low-level operations on the actual data, while *core* manages computational graphs for the computed functions.

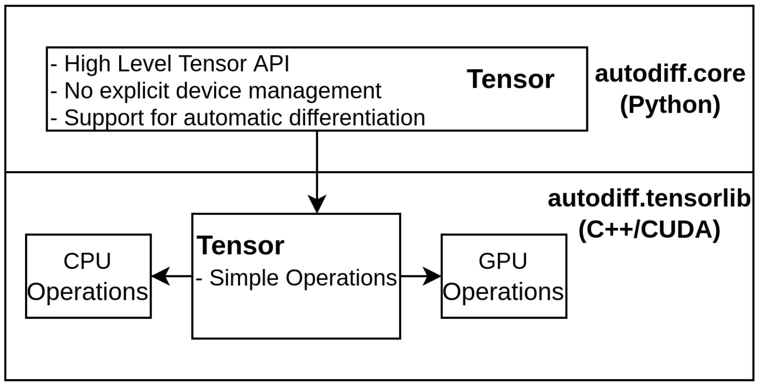


Figure 3: High-level design

The autodiff.tensorlib component (Figure 4) is a C++/CUDA module that holds 2D tensor objects and exposes an operations API to the Python module for carrying out tensor operations such as addition, multiplication, etc. It is designed to support operations on CPUs as well as Nvidia GPUs, which makes it a highly versatile and scalable component. The ability to use GPUs for tensor operations can significantly accelerate the computation of gradients in large models, making this component a valuable asset for machine learning applications.

The autodiff.core component (Figure 4) is a Python module that performs reverse-mode automatic differentiation. It does so by keeping track of operations and maintaining a topologically sorted computational graph. This graph contains details about how partial derivatives of a tensor can be calculated, making it possible to compute the gradients of any function that can be expressed as a composition of tensor operations. Similar to PyTorch's autograd, calling .backward() on a tensor will trigger gradient backpropagation in the computational graph, allowing users to easily compute gradients and optimize parameters in their models.

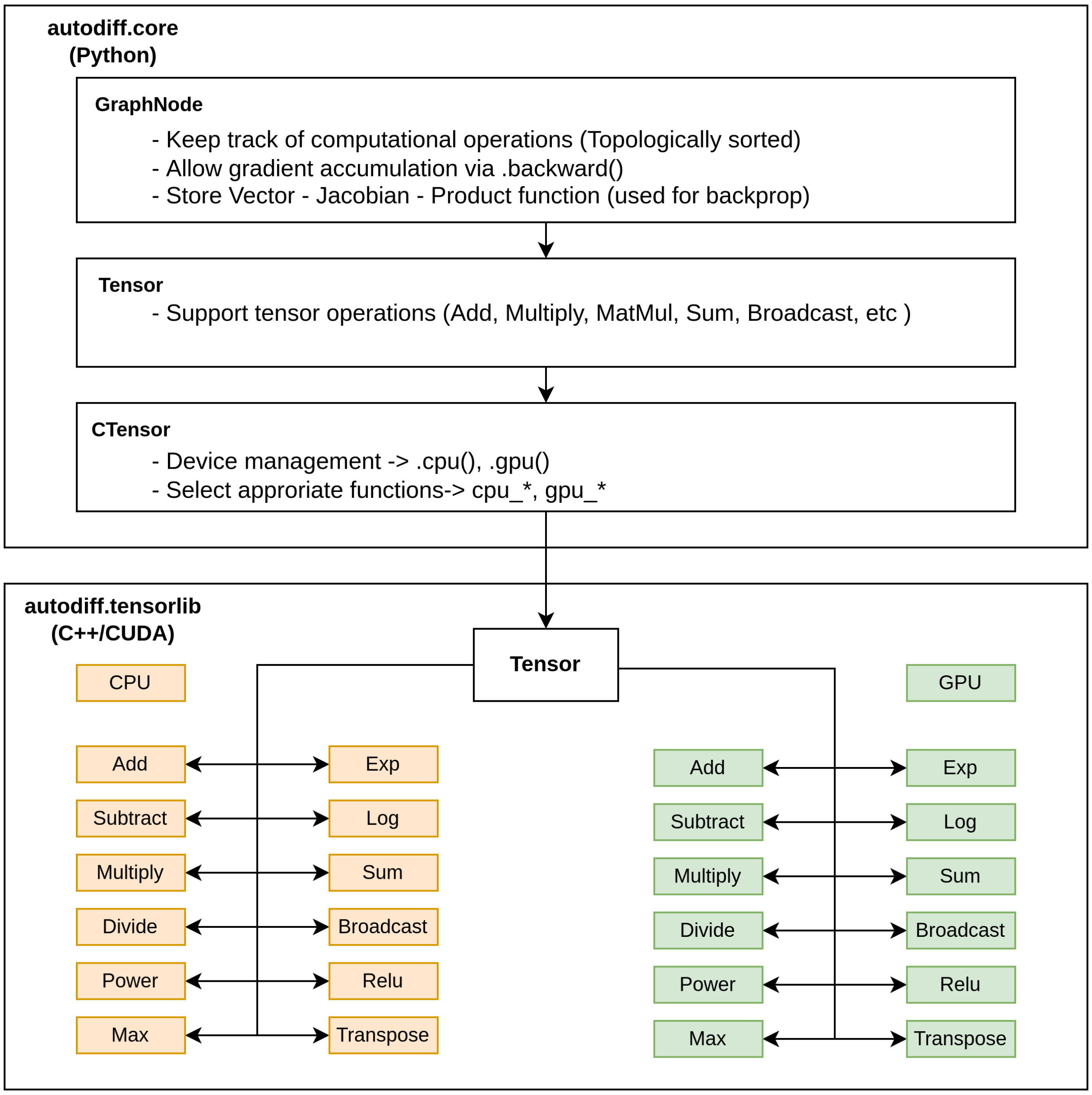

autodiff.core
(Python)
GraphNode
- Keep track of computational operations (Topologically sorted)
- Allow gradient accumulation via .backward()
- Store Vector - Jacobian - Product function (used for backprop)
Tensor
- Support tensor operations (Add, Multiply, MatMul, Sum, Broadcast, etc)
CTensor
- Device management -> .cpu(), .gpu()
- Select approriate functions-> cpu_*, gpu_*
autodiff.tensorlib
(C++/CUDA)
Tensor
CPU
Add
Subtract
Multiply
Divide
Power
Max
Exp
Log
Sum
Broadcast
Relu
Transpose
GPU
Add
Subtract
Multiply
Divide
Power
Max
Exp
Log
Sum
Broadcast
Relu
Transpose

Figure 4: Implementation

We implemented the following tensor operations, for both – single core CPU and GPU via cuda kernels:

1. **Add**: This operation adds two tensors element-wise. The input tensors must have the same shape.
2. **Broadcast**: This operation broadcasts the smaller tensor along the specified axis to match the shape of the larger tensor, and then performs element-wise addition. The broadcast axis must have the same size.
3. **Copy**: This operation creates a new tensor with the same contents as the input tensor.
4. **Divide**: This operation divides two tensors element-wise. The input tensors must have the same shape.
5. **Exponential**: This operation applies the exponential function element-wise to the input tensor.
6. **Logarithm**: This operation applies the natural logarithm function element-wise to the input tensor.
7. **Matrix multiplication**: This operation performs matrix multiplication between two tensors. The first tensor must have shape (M, N) and the second tensor must have shape (N, K), resulting in an output tensor with shape (M, K).
8. **Max**: This operation computes the maximum element along the specified axis of the input tensor. The output tensor has one less dimension than the input tensor.
9. **Multiply**: This operation multiplies two tensors element-wise. The input tensors must have the same shape.
10. **Negate**: This operation negates all elements of the input tensor.
11. **Power**: This operation raises each element of the first tensor to the corresponding element of the second tensor.
12. **Rectified Linear Unit (ReLU)**: This operation applies the ReLU function element-wise to the input tensor. The output tensor has the same shape as the input tensor.
13. **Subtract**: This operation subtracts two tensors element-wise. The input tensors must have the same shape.
14. **Sum**: This operation computes the sum of elements along the specified axis of the input tensor. The output tensor has one less dimension than the input tensor.
15. **Transpose**: This operation transposes the input tensor by swapping its rows and columns. The output tensor has the same shape as the input tensor, but with the dimensions rearranged.

For each of the operations, we use a similar strategy to parallelize calculations as described here. Each thread computes a single element in the output tensor. Blocks were used to group threads and each block processed a chunk of the input tensors. The number of blocks and threads per block were determined based on the size of the input tensors and the maximum number of threads supported by the GPU. We tried varying number of threads for each function – more on this in the result section.

To enable better mapping to the parallel machine, the original serial algorithm was modified to use parallelism in the form of GPU kernels. Each operation was implemented as a separate kernel.

3 Results

3.1 Understanding behaviour of different operators

The core parallelizable parts of our algorithm are the 15 operators demonstrated in Figure 2. We perform extensive benchmarking of these and based on our findings, we group the operators into subgroups. Each subgroup of operators exhibits similar behavior – we demonstrate the behavior by presenting results from on of the operators in the subgroup.

Our benchmarking experiments consisted of testing the change in behavior along two kind of plots. First, we analysed how the speedups for the CUDA implementation changed as we changed the number of threads per block. Second, we analysed how the speedups for OpenMP and CUDA changed when we kept the number of threads per block constant (chosen to be 1024) and varied the size of the matrices (we used square matrices for our experiments here).

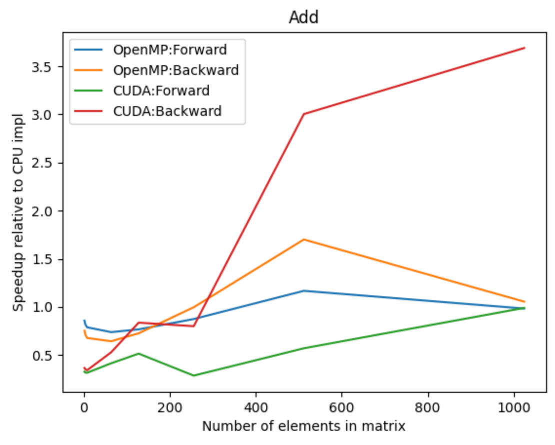
3.1.1 Add, Divide, Multiply, Negate, Subtract, Transpose, Sum, Broadcast

All operators in this group showed similar behavior in their plots. This is expected as the general behavior of all the operators in this group is the same: to apply element-wise transformations on one/two matrices. We arbitrarily choose to present the results for the Add operator in Figure 5.

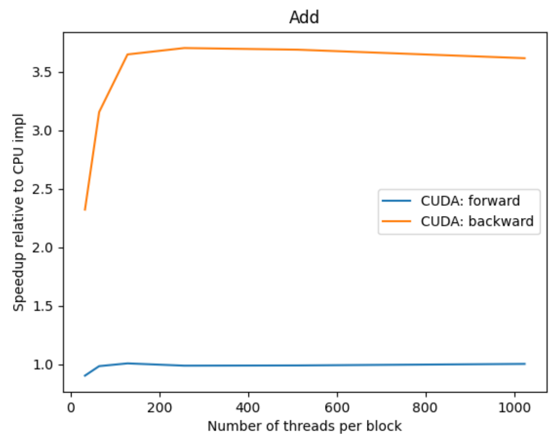
For really small matrices, we observe that GPU has an overall slow down simply because time is wasted on launching threads. However, as we increase the size of matrices, we observe a speedup for the backward operation as the calculation of the Jacobian (multiplication in this case) is parallelized as well. As we increase the number of threads (GPU), beyond 200 threads per block, we observe no speedup because the number of the total number of threads that can be executed in parallel exceeds the number of available processing ALUs.

3.1.2 MatMul

MatMul exhibited unique behavior, so we discuss its behavior separately. See Figure 6. We observed that the speedup of a parallel algorithm for matrix multiplication on a GPU is influenced by several factors, such as the size of the matrix being multiplied. At smaller matrix sizes, the speedup tends to be low due to the overheads associated with launching parallel threads on the GPU. However, as the matrix size increases, we infer that the parallel



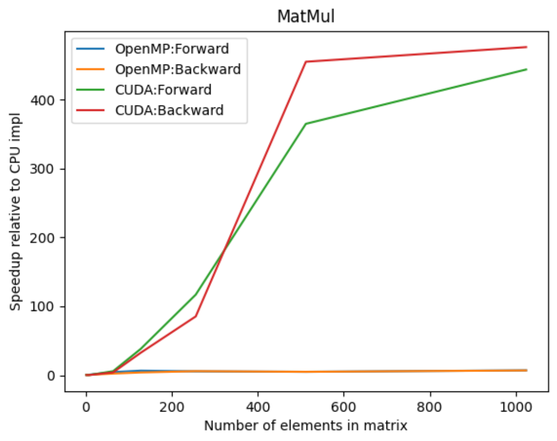
(a)



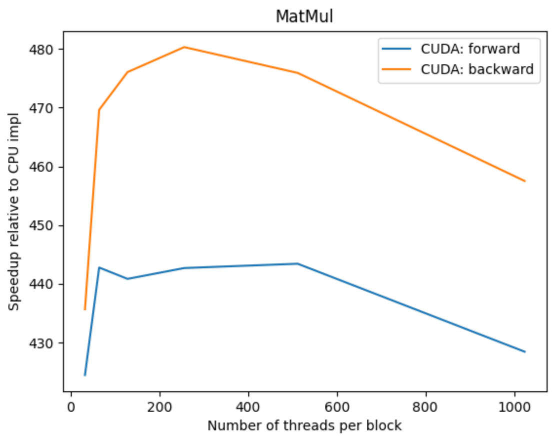
(b)

Figure 5: Observing speedups for the Add operator

algorithm can take advantage of the massive parallelism offered by the GPU, leading to an increase in speedup. This results in the initial rise of the speedup curve. We further speculate that as the matrix size continues to increase, the parallel algorithm may start to experience resource contention and memory bandwidth limitations, which can limit the speedup. At this point, we infer that the speedup curve tends to plateau, indicating that the performance improvement from parallelism has reached its limit.



(a)

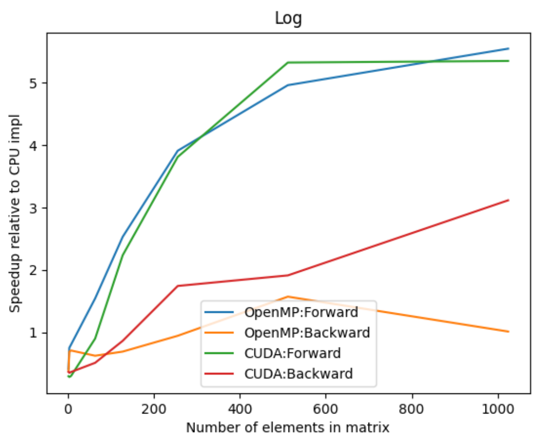


(b)

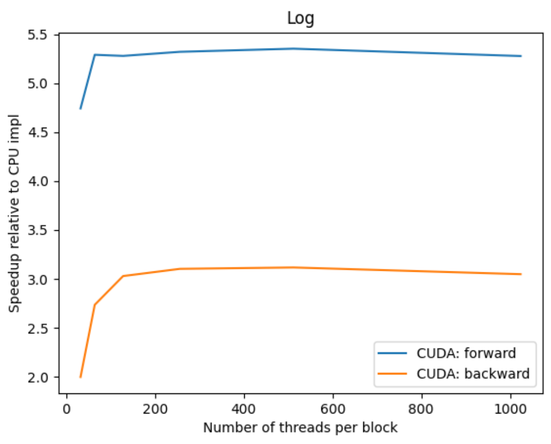
Figure 6: Observing speedups for the MatMul operator

3.1.3 Log, Exp, Power, Max, ReLu

Exp and Log are similar to operators in the previous group, in that they apply element-wise transformations to a single matrix. They however did exhibit a particular unique behavior, see Figure 7. As we increase the number of (GPU) threads, beyond 100 threads per block, we observe no speedup because the number of the total number of threads that can be executed in parallel exceeds the number of available processing ALUs.



(a)



(b)

Figure 7: Observing speedups for the Log operator

We observed that the performance of ”log” and ”exp” functions on a matrix increases as we increase the size of the matrix. This increase in performance is due to the inherent parallelism that these functions offer, which can be exploited better as the size of the matrix increases. However, we also observed that for the backward pass, which involves the calculation of the Jacobian, we do not see as much speedup as we increase the size of the matrix. This is because the overhead associated with the calculation of the Jacobian limits the performance improvement that can be achieved through parallelism. As a result, the performance gains from increasing matrix size are limited, and the speedup is not as significant as for ”log” and ”exp” functions.

3.2 Overall speedup achieved by the algorithm: Test on a 3 layer neural network

We trained multiple three-layered fully connected neural networks of varying sizes to measure and understand the overall speedup of automatic differentiation. We observed the OpenMP version did not scale as the CPU utilization during training was 100% alluding the fact that all cores had been used up, while the GPU version kept on speeding up as we increased the number of parameters of the neural network.

4 Credits

The split of the work was 50-50 between the two teammates. The work done by each person is as follows.

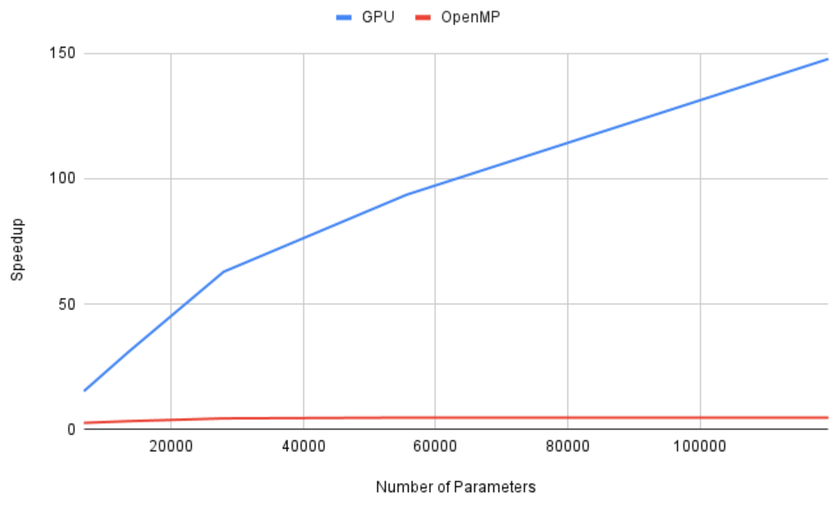


Figure 8: Implementation

Jay:

- Create a skeleton Tensor interface (C++/CUDA) that supports tensor operations
- Writing the CPU implementation of Sum (Reduce dimensions)
- Writing the CUDA implementation of Sum (Reduce dimensions)
- Writing the CPU implementation of Multiplication
- Writing the CUDA implementation of Broadcast (Increase dimensions)
- Implement Add, Multiply, Broadcast, Sum operations in python
- Add support for automatic differentiation for Add, Multiply, Sum, Broadcast
- Writing the CPU implementation of Power
- Writing the CUDA implementation of Power
- Writing the CPU implementation of Matrix Multiplication

Tanvi:

- Writing the CPU implementation of Addition
- Writing the CUDA implementation of Addition
- Putting in place a testing framework that allows to compare the results and performance of both operations.
- Writing the CUDA implementation of Multiplication

- Writing the CPU implementation of Broadcast (Increase dimensions)
- Create a skeleton Tensor interface (Python) that supports tensor operation tracking
- Update Add and Multiply to allow auto-broadcasting
- Writing the CPU implementation of Transpose
- Writing the CUDA implementation of Transpose
- Writing the CUDA implementation of Matrix Multiplication
- Add support for automatic differentiation for Power, Transpose, Matrix Multiplication
- Adding the OpenMP version of the code

Both:

- Benchmark CPU vs GPU performance for all operations with various tensor sizes with detailed analysis
- Preparing final webpage and poster for presentation.