Automatic Differentiation Using CUDA

Jay Patel (japatel) & Tanvi Karandikar (tkarandi)

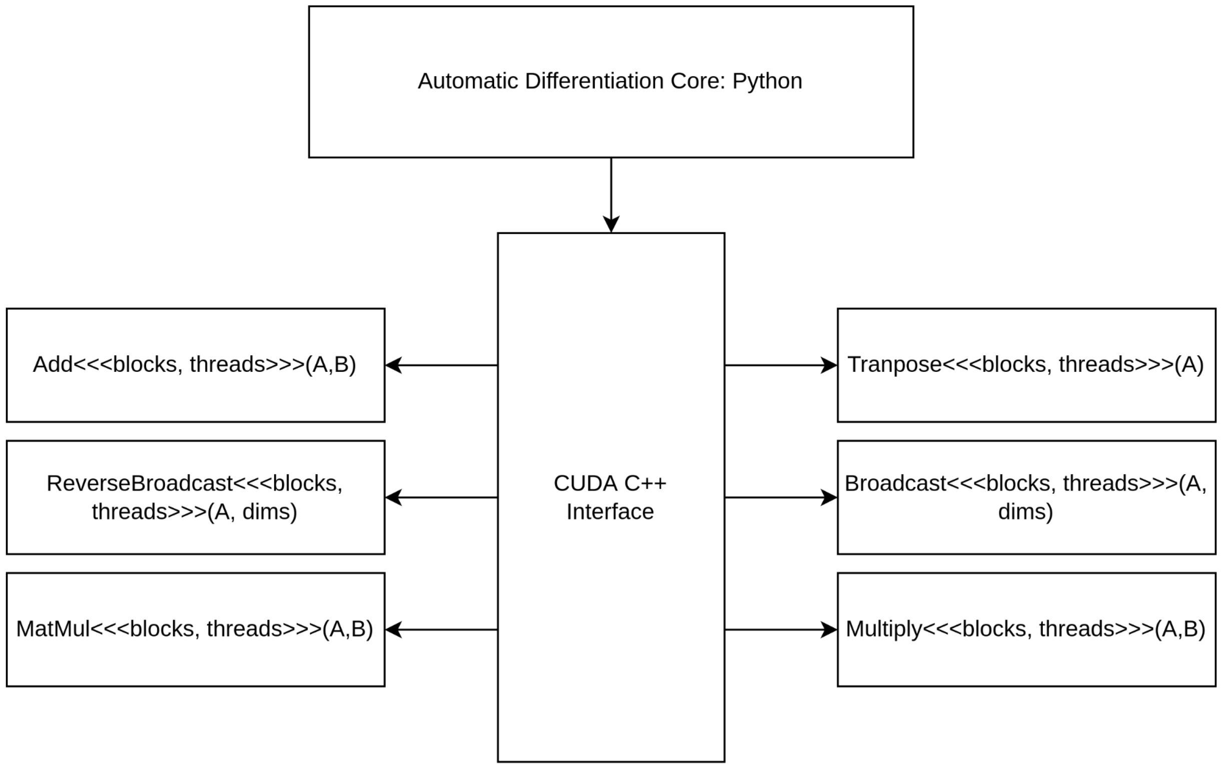
URL: 15618-cuda-autodiff.netlify.app/

SUMMARY:

We are going to implement automatic differentiation on GPU (CUDA) and compare it with a CPU version and perform a detailed analysis of both 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 (a stretch goal is to also support 2D convolution).

BACKGROUND:

Deep learning involves a lot of computation on tensors, multi-dimensional arrays of floats. Modern deep learning frameworks use automatic differentiation for training neural networks (error backpropagation). Training neural networks is compute-intensive due to a lot of tensor operations (matrix multiplication, addition, etc.). 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. This parallelism can be achieved by using vectorization, which involves performing the same operation on multiple elements of a tensor simultaneously. This can be achieved using SIMD instructions on CPUs, or using GPU threads that execute the same instruction on different elements of the tensor in parallel.


Automatic Differentiation Core: Python
CUDA C++ Interface
Add<<<blocks, threads>>>(A,B)
ReverseBroadcast<<<blocks, threads>>>(A, dims)
MatMul<<<blocks, threads>>>(A,B)
Tranpose<<<blocks, threads>>>(A)
Broadcast<<<blocks, threads>>>(A, dims)
Multiply<<<blocks, threads>>>(A,B)

The automatic differentiation library will be written in python – so that we can quickly implement the core logic. However, tensor operations will be carried out on GPUs using CUDA. We will either use pycuda or ctypes as an interface with CUDA C++ code.

THE CHALLENGE:

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 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.

RESOURCES:

- We will be starting from scratch.
- For the automatic differentiation implementation, Dougal Maclaurin thesis's (Chapter 4 describes Autograd) as a reference.

GOALS AND DELIVERABLES:

PLAN TO ACHIEVE:

- CPU version and CUDA version implementations of the following operations:
    - Addition
    - Multiplication
    - Transpose
    - Power
    - Matrix Multiplication
- Detailed analysis of the performance characteristics of both the implementations across different input matrix sizes.
- Implementation of a library in Python that uses the CPU and GPU versions of operations defined in the previous point to perform automatic differentiation. (This is a low-priority part, and in case the work goes more slowly we will not do this)

HOPE TO ACHIEVE:

- CPU version and CUDA version implementations to support 2D convolutions
- Detailed analysis of the performance characteristics of both the implementations across different architectures (different GPUs, memory per CPU etc).

FINAL DELIVERABLE AT POSTER SESSION:

We plan to have an interactive demo of our Python library that demonstrates how our code runs for some small demo matrix operations. We will also present detailed visualizations of our benchmarking experiments and hope to show a much better improvement in runtime in CUDA over the CPU-based implementation. We hope to see up to 20x speedup over the CPU-based implementation.

PLATFORM CHOICE:

The platform we hope to use for deployment and benchmarking is the GHC machines. We plan to use the GHC machines for development, testing and benchmarking as they provide a convenient development environment and have reasonably powerful GPUs.

We choose C++ as our language for tensor operations so that we can leverage CUDA C++. C++ allows for a lower level of memory management and thus will allow for application of more sophisticated parallelization paradigms. We choose Python to implement our frontend and library as it will be quick to implement and we can focus more effort on the development of parallel code.

SCHEDULE:

Deadline	**Task**
April 9	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.
April 16	Writing the CPU implementation of Multiplication
	Writing the CUDA implementation of Multiplication
	Writing the CPU implementation of Transpose
	Writing the CUDA implementation of Transpose
April 23	Writing a basic Python interface as a library that uses the CPU and CUDA implementations on the kernels
	Writing the CPU implementation of Power
	Writing the CUDA implementation of Power
April 30	Writing the CPU implementation of Matrix Multiplication
	Writing the CUDA implementation of Matrix Multiplication
	Visualizing the performance of all implemented kernels across different matrix sizes.
May 4	Preparing final webpage and poster for presentation.