Project: Porting Backprop to AMD

Adam Orucu

January 8, 2024

Url to code: `https://github.com/adamorucu/gpu-programming-course`
I had no group member, all work was done by me.

1 Introduction

Artificial neural networks, today, are one of the most popular machine learning algorithms, frequently used for making predictions given knowledge from existing data. A simple neural network can be viewed in Figure 1. In this example the neural network gets as input a data with 2 features and makes a prediction of size 1. The neural networks consists of layers, at each layer they multiply the values from a previous layer by some weights, sum them up and propagate them forward. A neural network can be taught by letting it make predictions and then "letting it know" how far from the correct answer it is so it can update its weights to lessen this error. More concretely, the gradient of the error given each unit is calculated starting at the output node and propagated backward using the chain rule until the start of the neural network. Given these gradients for each unit the weights of the units are updated to make the error smaller. This process is called backpropagation. The training of the neural network iterates over forward propagation and backward propagation, until the error converges.

This project uses a simple implementation of the forward and backward propagation processes if CUDA and ports them into HIP. The performances of both implementation - CUDA and HIP - are tested and compared on NVIDIA and AMD GPUs.

2 Methodology

I originally tried using more complex implementations of backpropagation which were basically small neural network libraries which make the implementation of neural networks more flexible and easier. However, I was not able to port them to work on AMD GPUs. For this reason I used a simpler implementation available in the Rodinia benchmark that was also used in the first assignment. This implementation is available at Rodinia download site. I used version 3.0 for this project.

For this project the relevant part of this benchmark is present at `cuda/backprop` within the folder structure. In this folder there are several files which when compiled provide a simple implementation of forward and backward propagation and a tiny nerual network. One important file in this folder is `backprop_cuda.cu`. This CUDA-compatiable code creates a neural network and does one forward and one backward propagation through it. The other file if `backprop_cuda_kernel.cu`. Here, one can find the implementations of forward propagation which as previously described makes many multiplication and addition operations to calculate the prediction as well as a function to update the weights of the neural network. Both of

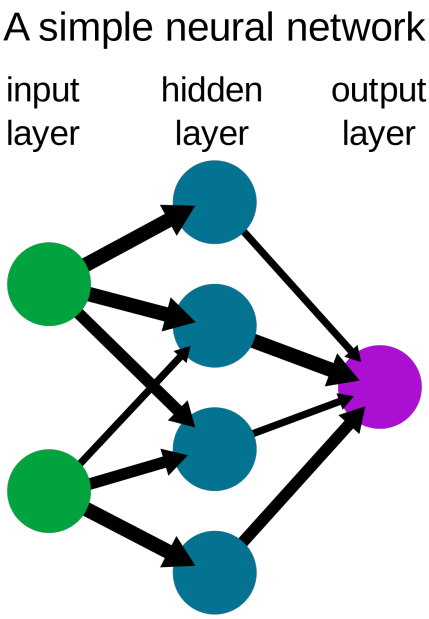

Figure 1: A simple neural network (source: Wikipedia)

these operations are made on the GPU. Rest of the files have useful functions for other calculations such as intialising the networks with random weights or making NN calculations on the CPU. These files will not be changed for the porting - only the CUDA files are adjusted to run using HIP.

The rest of this section describes the steps I went through to port the code to HIP.

Install ROCM

I installed ROCM with the following command `module load rocm/5.3.3`. This module contains hipify which can be used to more easily transform CUDA code to HIP code.

To disable time locale related warnings I also set the follwing:
`export LC_CTYPE=en_US.UTF-8; export LC_ALL=en_US.UTF-8`.

Hipify

For this purpose I used hipify-perl. From my understanding the is a simpler way of porting the code compared to hipify-clang, and basically does search-and-replace substituting CUDA operations with their HIP counterparts. This is done by running `hipify-perl CUDA_FILE > HIP_FILE`. To make the conversion more efficient and quicker I wrote a tiny shell script that does this for all cuda files in the folder (`amdize.sh`).

The conversion to HIP was almost perfect. I, just, had to remove a couple lines of the code. I also renamed some of the file names.

Makefile

The hardest part of porting the code was writing Makefile that would compile the relevant C++ and HIP files and link them togather to create one executable. While at the end the Makefile for the HIP version ended up being quite simple I had to go through many different ones to understand how to link different compiled files correctly.

Running the executable

The run an executable on the Dardel I first had to request a GPU as follows:

`salloc -A edu23.dd2360 -p gpu -N 1 -t 00:10:00`.

In fact this also had to be done in the previous steps to be able to port the code to HIP. After the gpu is allocated. The code is compiled using `make` which creates an executable named `backprop`.

This executable is run as follows: `srun -n 1 ./backprop`.

3 Experimental setup

For the tests both NVIDIA and AMD GPUs were used. The test on the NVIDIA GPU where carried out in Google Colab. Here, I used a Tesla T4 on the Ubuntu operating system. The version of CUDA and gcc where 12.2 and 11.4.0, respectivelly.

For the AMD tests I used the Dardel supercomputer provided for this course. The computer used SUSE Linux Enterprise Server 15 SP3. The HIP version is 5.0.13601 and gcc version is 7.5.0.

The code I used can be found in my GitHub page at the beggining of the report. The specific codes for CUDA and HIP implementations are located in `project/rodinia/cuda` and `project/rodinia/hip`.

4 Results

I have started by evaluating both the CUDA and HIP programs on their respective GPUs. I did this by testing them on neural networks with different input layer sizes. Results can be viewed in Figure 2. The results show both the forward and backward propagation times for both of the GPUs. For the largest two tests the NVIDIA GPU was not able to make the calculations and failed with an error. It can be seen, from the results, that the AMD GPU is much faster even at a simple experiment such as this. This is, ofcourse, expected since the MI250X is much more powerful than the Tesla T4.

The second experiment was to see what is the time spent on for the operations in the GPU. From the results in Figure 3 it is clear that almost all the time is spent copying the data to and from the GPU. This suggest that with a more efficient placing of operations of the algortihm an improvement in computation time should be possible. Other possible improvements are discussed in the next section.

5 Discussion and conclusion

There are several things that would be intersting to test or add for this project. Firstly, making the neural network deeper, i.e. adding more layers. This not only helps make better predictions but more importantly for this project could make the Memcopy part of the program take less time, percentage-wise. Since in such a case more calculations will be necessary the values will be stored closer to the processor until the calculations are over and then transfer back to the CPU. Since, there will be more of these calculations it will be a more effective way to make use of the GPU.

Similarly, using more epochs (more iteration of learning) be interesting to see how the performance changes if the total training time is longer.

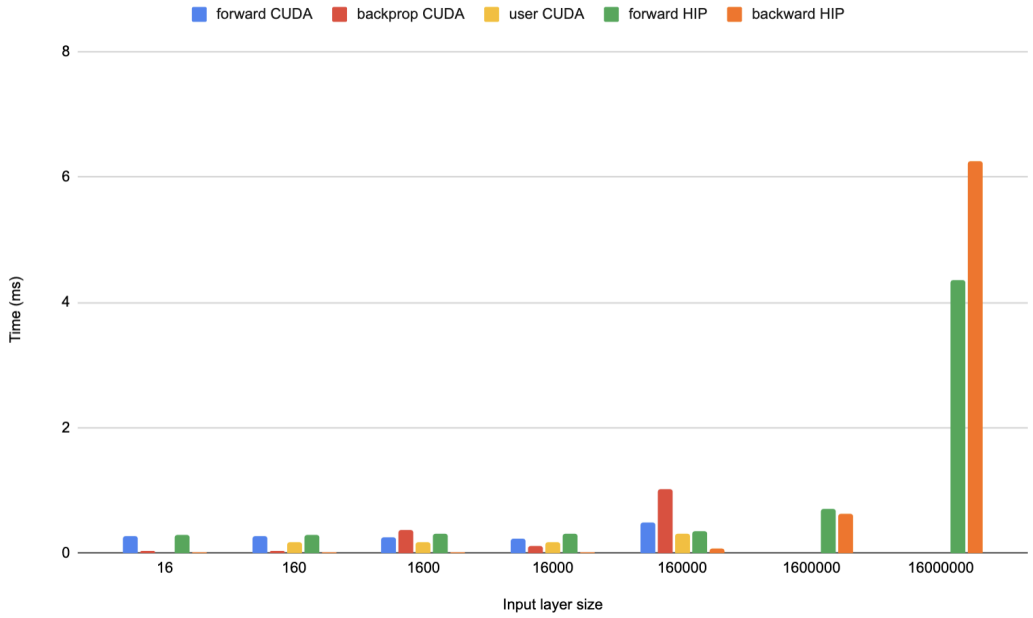

Figure 2: Evaulating CUDA vs HIP

Figure 3: Tracing results for AMD

A technique that is frequently used in deep learning is the batch the data into groups instead of passing it one sample at a time, this would be interesting to study since it adds another dimension to the data and the GPU code would need to be adapted to suit this.

Lastly, in the implementation on Rodinia the calculations of gardients seem to be done on the CPU and not on the GPU, and the required updates are sent to GPU to be applied. It might be more efficient to make these calculations on the GPU as well.

6 References

- Lecture slides
- Rodinia for the initial CUDA implimentation.
- HIP user guide on docs.amd.com to read more about HIP.
- Guide on how to port to HIP to understand what flags to use in the compilation.