Implementing a neural network for the GPU
(Bachelorproject)

Jelmer van der Linde, s1772791, j.van.der.linde.1@student.rug.nl,
Marco Wiering [*], Vali Codreanu [†]

July 4, 2012

Abstract

Automatic classification becomes more and more interesting as the amount of available data keeps growing. Also, modern computers are equipped with powerful hardware specifically designed for processing fast amounts of data, namely the GPU (graphical processing unit). We use OpenCL to implement a multilayer perceptron that runs on the GPU. Our implementation scales better than an implementation for the CPU, but this proves to be only an improvement for larger networks due to the overhead of calculating on a different device.

1 Introduction

Recent developments in GPU-land have resulted in the availability of tools to use the graphics processing unit (GPU) for other purposes. Artificial neural network (ANN) can take advantage of this. This fits perfectly with the demand to process increasingly large amounts of data.

Research Question: How fast is an implementation of a neural network for the GPU compared to an implementation for the CPU?

In this paper we explore how a multilayer perceptron can be implemented in OpenCL. We will compare the speed of this implementation with an implementation in C, and test whether the difference in speed and the scaling of speed is an advantage worth the effort. Larger networks might be faster to train on the GPU, but if the larger network does not yield better performance than a small network, there is no value in this increase in speed.

We expect it to be faster for larger networks, as in this configuration calculating the activation of nodes in parallel has a clear advantage. We also expect that for smaller networks it will be slower as there are costs to running the computation on a different device.

This paper is organized as follows: In Section 2 we discuss the background of implementing neural networks on GPUs: what has been done before. In Section 3 we describe the ANN in more detail. We then introduce OpenCL in Section 4 and how we use it to implement our neural network in Section 5. Finally, we will discuss the results and future work in Section 7.

2 Background

The development of GPUs has focused on optimizing throughput instead of latency. As a result the GPU can outperform the CPU in streaming tasks: tasks that require applying the same calculation on large amounts of data (Owens, Luebke, Govindaraju, Harris, Krger, Lefohn, and Purcell (2007)).

In the beginning, GPUs had been optimized for these tasks by designing the hardware itself around the algorithms. But in time, more and more parts of the GPU have become programmable, allowing the programmer to refine or redefine the applied computations (Owens et al. (2007)). Oh and Jung (2004), Luo, Liu, and Wu (2005) demonstrated that the GPU can be repurposed for non-graphical calculations and implemented a neural network by programming the pixel and vertex shaders to calcu-

[*]University of Groningen, Department of Artificial Intelligence
[†]University of Groningen, Department of Computer Science

late activation. This practice of using the GPU for non-graphical calculations also known as general-purpose computing on graphics processing units (GPGPU).

The hardware is still optimized for graphical tasks and speed, and as a consequence, GPUs were traditionally not equipped for precise floating-point mathematics. Also, one needed specific knowledge of the architecture and of programming with the graphics APIs such as DirectX or OpenGL. More modern hardware nowadays does support double precision floating-point mathematics, but it is still not commonplace.

In 2007 NVIDIA released CUDA, which allowed programmers to program the GPU in a variant of C, specifically designed for using the GPU hardware for tasks other than generating graphical images. Jang, Park, and Jung (2008) showed that CUDA could be used to implement a neural network without the need of significant background knowledge in graphics. The Deep neural network developed by Ciresan, Meier, and Schmidhuber (2012) currently yields the best performance on various datasets because it uses CUDA to train the network in a matter of days instead of months.

OpenCL is developed by the Khronos Group and is an open API for parallel programming. Unlike CUDA, it is not specifically designed for GPUs. Hardware vendors can implement it for both CPU and GPU as well as specialized hardware. OpenCL code can run on any device that implements the necessary APIs, making programs that depend on OpenCL more portable than programs depending on CUDA. The downside is that the OpenCL runtime may not be as tightly interknitted with the features of the hardware as CUDA. Karimi, Dickson, and Hamze (2010) and McConnell, Sturgeon, Henry, Mayne, and Hurley (2012) show that the performance of OpenCL is a bit less than the performance of CUDA.

3 Artificial Neural Network

The multilayer perceptron consists of an input layer, a number of hidden layers, and an output layer, as shown in Figure 1. The first hidden layer is fully connected to the input layer with weighted connections, the second hidden layer to the first hidden layer and so on. The output layer is fully connected to the last hidden layer. The weights of the connections between nodes are initialized with random values between -1 and 1, although often less extreme values closer to zero are chosen.

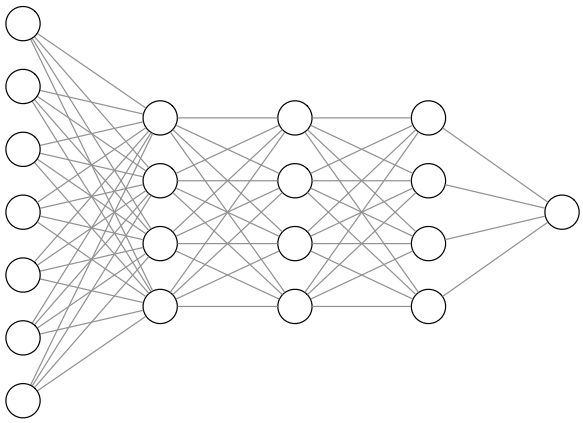

Figure 1: A fully connected neural network with three hidden layers and a single output node. The lines between the nodes each have their own weight.

The input layer has the same amount of nodes as the input data of the neural network. The output layer has as many nodes as there are classes in the dataset. The number of hidden layers, and the number of nodes in these layers are parameters which can be altered to increase the performance of the neural network.

The nodes in the first layers receive a value equal to the input data. The activation of each node a_j in the second layer (i.e. the first hidden layer) is calculated by taking the inner-product between the connecting weights w_{ij} and the values of each node a_i in the previous layer (i.e. the input layer). A bias b_j is added and the activation function f is applied.

$$a_j = f(\sum_{i=0}^{n} w_{ij} a_i + b_j) \tag{3.1}$$

The activation function f introduces nonlinearity into the network. We implemented these as follows:

$$f(x) = \frac{2}{1 + e^{-x}} - 1 \tag{3.2}$$

$$f'(x) = 1 - x^2 \tag{3.3}$$

3.1 Backprogagation

To train the NN the backpropagation algorithm is used. The weights between the layers are adjusted to better reproduce the target output in accordance to their impact and the activation of their connected nodes. We update the weights backwards: The weight connecting the last hidden layer and the output layer are adjusted first. First we calculate the error of the jth node in the output layer by subtracting the actual activation of the node a_j from the target value g_j:

$$e_j = g_j - a_j \tag{3.4}$$

The weight w_{ij} of the connection between the ith node in this layer and the jth node of the next layer is corrected by adding the adjustment d_j multiplied by the learning speed l and the activation of the node a_i:

$$w_{ij} = w_{ij} + a_i \cdot d_j \cdot l \tag{3.5}$$

The adjustment d_j equals the error multiplied by the derivative of activation function f applied to the activation of the jth node:

$$d_j = f'(a_j) \cdot e_j \tag{3.6}$$

For the hidden layers, the error of the ith node is equal to the sum of all the errors e_j in the next layer in proportion to their connecting weights w_{ij}:

$$e_i = \sum_{j=0}^{n} w_{ij} \cdot e_j \tag{3.7}$$

Finally, the bias for each node in this layer is updated by adding the error e_i:

$$b_i = b_i + e_i \cdot l \tag{3.8}$$

4 OpenCL

OpenCL allows us to define functions in a variant of C. Functions can be marked as kernels, which can be scheduled to be executed by calling the OpenCL API from our C++ program. Many instances of the kernel function are executed in parallel, and as such each instance of a kernel only needs to apply its calculations on a small subset of the data. Batches of kernel executions can be queued to be executed after each other. This in total results in the CPU queuing a series of commands once, and then waiting for the GPU to finish executing the queue.

Work units are instances of a kernel. Work units are grouped in work groups with whom they share memory. Each work unit is assigned a unique id which can be used inside the kernel to identify it. Only the execution of work units in the same work group can be synchronized with each other. This is important to note when removing inner loops that access the same memory, e.g. summations. We implement summation of Equation 3.7 across multiple work items through sum reduction as described by Hillis and Steele (1986).

As kernels are executed on a different device they cannot access the memory of the CPU, but only buffers allocated by OpenCL on the device. To minimize the copying of data between CPU and GPU as suggested by Luo et al. (2005), all data used by the network is kept in the memory on the GPU in multiple OpenCL buffers.

Only single precision floating-point arithmetics are required by OpenCL to be available. Newer hardware does support double precision floating-point data types, and when it does OpenCL can make use of this, but this is an optional feature. Recursive function calls are not supported according to the OpenCL-1.1 specification, even though newer hardware does support it (Munshi (2011)). CUDA 2.1 does allow recursion on hardware that supports it.

5 Method

The calculations of the activation for each node in a layer are only dependent on the calculation of the activation of all the nodes in the previous layer and the weights connecting them to the node, but not on the other nodes in the same layer. Therefore these activations can easily be calculated in parallel. This also holds true for the calculation of the error and the adjustment of the weights in the backpropagation step.

We first rewrite the implementation of the neural network to do all the calculations in C functions. This way we can implement the neural network, complying with the constraints of OpenCL one by one, while having all the tools for the gcc programming environment available. At the time of writ-

ing, no such tools are available for OpenCL for the platform we use for development. For easy comparison later on, we maintain the same C++ interface for all three implementations: the original C++ implementation, the C++/C implementation (we will refer to this as the C implementation) and the OpenCL implementation.

We allocate a number of buffers in the GPU's global memory where we keep the activation of all the nodes and the weights of all the connections in our network. Initially, the random values for the weights are initialized by the CPU and then copied to the GPU as OpenCL does not offer the functionality to generate random values. Thereafter, only the inputs of the first layer are copied to the GPU and only the outputs of the final layer are retrieved from the GPU. We then queue the copying of the data of the first layer to the buffer. Second, the execution of many instances of the kernel implementing Equation 3.1 is scheduled to be applied on the buffers for the first layer. The same is scheduled for succeeding layers, to be executed when the batch for the previous layer has been completed. Finally, we queue the copying of the data from the output buffer back to the CPU memory. The CPU then blocks and waits till the GPU has emptied the queue.

To create the OpenCL kernels we try to replace most of the loops in our code with scheduling the execution of a kernel function for each would-be iteration of the loop, using the work unit id as an index to indicate which iteration this would have been. These kernel functions are equal to the old body of the loop applied to a single argument.

We schedule multiple kernels per node in each layer. This means the summarizing is done together by multiple work units, and as a result these work units have to be synchronized. This limits the amount of kernels we can assign per node to the maximum number of work units inside a work group.

6 Results

6.1 Setup

We test the performance of our implementations on an iMac running Mac OS X 10.7 Lion using the OpenCL implementation Apple provides. It houses an ATI Radeon HD 5670 with 512 MB of GDDR5 memory and an Intel Core i3 3.2 Ghz with 12 GB of 1333 Mhz DDR3 RAM memory. To compare the performance of OpenCL on different hardware we also compare the results of the iMac with results generated on a Mac Pro with an ATI Radeon HD 5770 with 1 GB of GDDR5 memory. The code of all implementations is compiled with compiler optimization flags enabled.

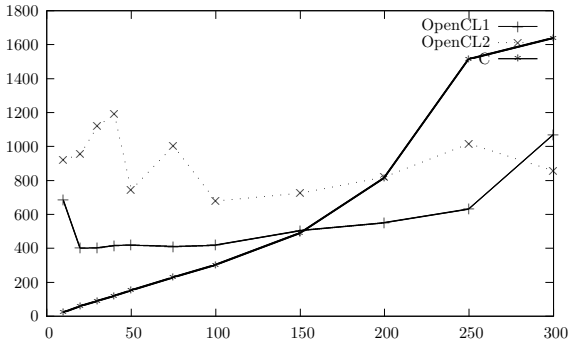

Figure 2: Comparing speed across different implementations using data from Table 1.

To compare and test our neural network we use a dataset of handwritten digits in Bangla. These digits have been preprocessed and normalized into 5381 values between 0 and 1. This will be the input data for the network. It contains 2000 samples, of which 200 will be randomly picked to test the performance. The network has to classify them into one of 10 categories.

As a consequence, our network has 5381 nodes in the input layer and 10 nodes in the output layer. When the network is trained to learn the symbol for *six*, the input layer in presented with the data, and the target is set to -1 for all nodes, except for the sixth node, which is set to 1.

6.2 Comparison OpenCL and C

As can be seen in Table 1 and accompanying Figure 2, our OpenCL implementation of the neural network is faster than the single core CPU implementation only when the network is configured with a large number of hidden units per layer. The speed of the OpenCL[1] configuration seems to be consistent across most values for the number of nodes per hidden layer, but skyrockets at the end. For

hidden units	OpenCL^1	OpenCL^2	C
10	684	923	23
20	401	955	60
30	403	1119	90
40	417	1191	120
50	420	744	154
75	410	1003	229
100	420	678	304
150	504	724	490
200	551	819	816
250	632	1016	1516
300	1071	858	1640

Table 1: Measuring speed in seconds to train 100 epochs on the Bangla dataset, varying number of hidden units. 3 hidden layers. OpenCL^1: 4 work units per node. OpenCL^2: 16 work units per node. Average of 3 runs per configuration.

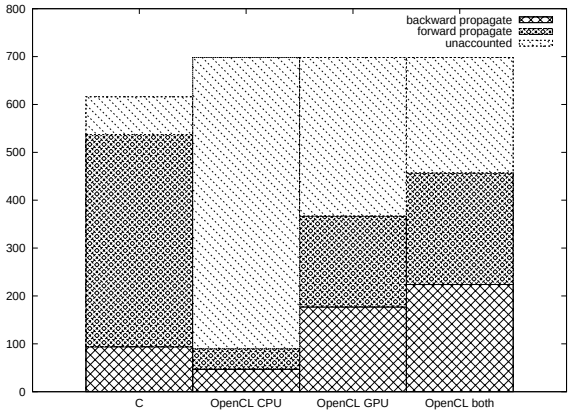

Figure 3: Profiling implementations using data from Table 3.

the same configuration with more work units per node, OpenCL^2, the implementation is slower. Notice how the time taken by OpenCL^1 at 300 nodes ($4*300$) is almost equal to the time of OpenCL^2 at 75 nodes ($75*16$). This might explain the bump at the end of OpenCL^1, but does not explain why the line for OpenCL^2 does not show a clear trend.

To confirm that our implementation performs equally well as the reference C implementation, we measured the performance of both implementations on an unseen test set. Results are shown in Table 2. It could have been possible that the lack of floating point precision degraded performance, but this seems not to be the case. This table also shows that, with our current unoptimized configuration, using more than 100 nodes in each of the hidden layers does not yield a noticeable benefit for the performance.

For a more in-depth comparison of both implementations, we measure the CPU time taken by the forward- and backpropagation steps for the C implementation, and the total time taken by running the tasks on the GPU. The times for the OpenCL implementation are obtained through the OpenCL runtime. The results are shown in Table 3 and Figure 3. Part of the time unaccounted for is because the test phase and the loading of the data are not measured in the CPU and GPU time, but are in the total time.

As can be seen, forward propagation takes most of the time in the C implementation, mainly due to the exponent in the calculation of Equation 3.2.

The profiling of the OpenCL implementation is split into the CPU and GPU part. The CPU only issues commands to the GPU, and then waits. This explains the large amount of time unaccounted for. The GPU part shows that both forward and backward propagation happen equally fast. This may indicate that the kernel function is not the bottleneck, as the C implementation shows that forward propagation is computationally more intensive.

6.3 Comparison between hardware

We run the OpenCL network on both the iMac and Mac Pro to see whether the ideal number of work units per node differs across different hardware. As can be seen in Table 4 the speed of the network differs significantly between both GPUs. The fastest run on both GPUs is with 4 work units per node, but the hardware does not differ enough to conclude this is the right value for all configurations.

7 Discussion

Noticeable is the jumpy performance of the OpenCL implementation across different numbers of units in the hidden layers. It might be that the OpenCL runtime is unable to divide the number of requested work units into optimal batches, executing more batches for e.g. 75 nodes than for 100 nodes per hidden layer. We did find this pattern in

	OpenCL			C		
hidden units	min	average	max	min	average	max
10	0.12	0.31	0.75	0.13	0.31	0.74
20	0.12	0.51	0.83	0.13	0.49	0.84
30	0.13	0.55	0.85	0.12	0.57	0.84
40	0.12	0.59	0.86	0.13	0.60	0.85
50	0.13	0.62	0.86	0.12	0.61	0.85
75	0.12	0.63	0.87	0.12	0.64	0.86
100	0.13	0.65	0.88	0.13	0.64	0.87
150	0.13	0.66	0.87	0.12	0.65	0.87
200	0.12	0.64	0.88	0.11	0.64	0.87
250	0.10	0.61	0.87	0.11	0.62	0.87
300	0.09	0.56	0.86	0.10	0.61	0.86

Table 2: Comparing performance OpenCL and C implementations. 100 learning iterations. Shown is the accuracy on the test set. Average of 3 runs per configuration.

Implementation	backward	forward	total time	unaccounted for
C	94	443	616	79
OpenCL CPU	47	43	698	608
OpenCL GPU	177	190	698	331
OpenCL both	224	233	698	241

Table 3: Profiling C and OpenCL implementations. Time measured in seconds. 100 learning iterations, 3 hidden layers, 150 nodes per layer, 4 work units per group.

all our experiments but have not been able to describe the exact conditions under which it occurs.

The hardware we used does not offer support for double precision floating-point calculations and as a result our implementation uses single precision calculations. As can be seen in Table 2 there is no noticeable difference.

The OpenCL implementation is only faster than the C implementation when there are more than 150 nodes per hidden layer when using three hidden layers. This might indicate that using OpenCL adds a large overhead. Table 2 shows that using more than 75 units does not yield an improvement in the results, although this might be different with other values for learning speed and random weight initialization. Larger networks with more input nodes and larger layers might profit from using the OpenCL implementation as it does seem to scale better. For smaller networks our implementation is not faster than the reference C implementation.

The OpenCL implementation adds the number of work units per node as another parameter to optimize. In addition, the speed of the network greatly depends on the number of nodes in the hidden layers.

7.1 Future work

We have tried to find good values for the number of work units we should use to calculate a layer with the comparison in Table 4, but Table 1 showed that this comparison was too limited. Future work might start with trying to formulate the optimal number of work units depending on the amount of nodes that need to be calculated and the number of processors the particular GPU has available.

It is worth noting that the GPUs used in our experiments are basic models. Results will differ significantly when these experiments are run on top of the market models offered today. Testing the OpenCL implementation on newer hardware and platforms other than Mac OS X might yield far better results.

Our implementation of the network in OpenCL does not make use of the shared memory of the graphics card, except for the sum reduction step.

work units per node	ATI Radeon HD 5670	ATI Radeon HD 5770
1	387	815
4	282	474
8	316	597
16	398	823
32	566	1281
*	1575	3947

Table 4: Comparing speed in seconds of OpenCL implementation for multiple configurations of work units per node on different hardware. 3 hidden layers, 128 nodes per layer. Star denotes the number of nodes in the previous layer.

Shared memory is faster than the global memory in which we store the network. Shared memory is faster but smaller, and only shared across every work unit in a work group. In the understanding that one serial copy operation would be faster than numerous small non-serial reads from the global memory, we tried to copy all values needed for calculating a layer into the shared memory. But this did not fit. Smarter prefetching of the values needed in the calculations might yield an increase in performance as it takes some load off the global memory. Another method to speed up the reading of data might be to reorder the way weights and node values are stored in memory or to make accessing them more predictable for the GPU.

The reduction of branching as suggested by Owens et al. (2007) may yield an increase in performance as the GPU is better able to predict what data will be needed. We did some small experiments with it, separating the kernel for backpropagation into two separate kernels, removing an if-statement in the kernel function. We did not notice a change in performance, but we have not made a significant effort.

References

Dan Ciresan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. *CoRR*, abs/1202.2745, 2012.

W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, December 1986. ISSN 0001-0782. doi: 10.1145/7902.7903.

Honghoon Jang, Anjin Park, and Keechul Jung. Neural network implementation using cuda and openmp. In *Proceedings of the 2008 Digital Image Computing: Techniques and Applications*, DICTA '08, pages 155–161, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3456-5. doi: 10.1109/DICTA.2008.82.

Kamran Karimi, Neil G. Dickson, and Firas Hamze. A performance comparison of cuda and opencl. *CoRR*, abs/1005.2581, 2010.

Zhongwen Luo, Hongzhi Liu, and Xincai Wu. Artificial neural network computation on graphic process unit. In *Neural Networks, 2005. IJCNN '05. Proceedings. 2005 IEEE International Joint Conference on Neural Networks*, volume 1, pages 622 – 626 Vol. 1, july 2005. doi: 10.1109/IJCNN.2005.1555903.

Sabine McConnell, Robert Sturgeon, Gregory Henry, Andrew Mayne, and Richard Hurley. Scalability of self-organizing maps on a gpu cluster using opencl and cuda. *Journal of Physics: Conference Series*, 341(1):012018, 2012.

Aaftab Munshi, editor. *The OpenCL Specification, version 1.1*. Khronos OpenCL Working Group, 2011.

Kyoung-Su Oh and Keechul Jung. Gpu implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, 2004. ISSN 0031-3203. doi: 10.1016/j.patcog.2004.01.013.

John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.