Practical GPGPU using OpenCL

Supplemental tutorial for INFOB3CC, INFOMOV & INFOMAGR

Jacco Bikker, 2017

Introduction

A typical consumer PC contains at least two processors. One is the CPU, which runs the operating system, communicates with peripherals such as keyboard, mouse and printers, and has access to mass storage. The second processor is the GPU. Traditionally, this is a special-purpose, fixed-function processor, responsible for generating images. Relatively recently GPUs became programmable. Initially this was limited to graphics-related tasks: pixel shaders, vertex shaders and somewhat later geometry shaders. A modern GPU however is a general purpose processor, capable of executing generic code, often much faster than the CPU. Leveraging this compute potential requires understanding of the characteristics of the GPU hardware, as well as programming models tailored for this hardware.

GPU Hardware

A typical CPU is equipped with multiple cores. A similar approach is used on GPUs: an NVidia GPU consists of a number of *Streaming Multiprocessors* (SM), which AMD simply refers to as *Compute Units* (CU). The numbers are higher though: NVidia's Titan XP uses 30 SMs; AMDs RX Vega 64 uses 64 CUs. Each SM executes *warps*, which are similar to threads on the CPU. But where a CPU is typically limited to two threads per core (through hyperthreading), NVidia's SMs process four warps in parallel, chosen from up to 64 warps that the GPU can switch to whenever a stall occurs. And finally the warps themselves: these are 32-wide SIMD lanes, while an AVX2-capable CPU is limited to 8 SIMD lanes.

In short, if we keep everything in CPU terminology, the Titan XP GPU is a 30-core processor, executing 120 threads in parallel, chosen from 1920 active threads. Every instruction executed is a 32-wide SIMD instruction. The SIMD behaviour is mostly hidden from the programmer; instead the device appears to execute 3,840 threads in parallel, chosen from 61,440 threads.

All these cores need to be fed data. The memory bandwidth of the Titan XP GPU is 550 GB/s, compared to ~38GB/s for a high-end Intel CPU.

Under ideal circumstances, the GPU is able to perform 12 *trillion* floating point operations per second (TFlops), while CPU performance is still measured in GFlops.

The above may sound like the description of the ideal processor. In practice, things are somewhat… *complicated*.

It starts with the 32-wide SIMD instructions. SIMD is an abbreviation of *Single Instruction Multiple Data*: on a 32-wide SIMD machine, each instruction in a program is executed for 32 threads. This is great when 32 pixels of a polygon need shading: each pixel will go through the same program, just with slightly different data. It is not so great if we have 32 different threads. In fact, if 32 threads run the same code, but do not agree on the outcome of an if-statement, our only option is to run the conditional code for some of the threads, while the others idle.

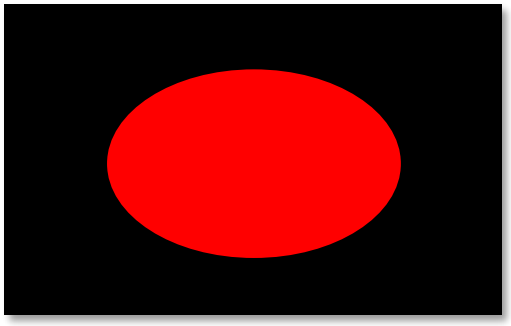
A second observation is that, compared to CPU cores, GPU cores are relatively simple. Where a CPU relies on long pipelines, branch prediction logic and large caches to deal with complex program flow, the GPU explicitly relies on massive parallelism. Whenever an instruction must wait for data from memory, the SM will switch to another warp. CPUs do this as well, but they can only switch between *two* threads. The GPU may choose from up to 64 threads.

This illustrates an important design principle of the GPU: it is a data parallel processor, and it expects a large number of uniform tasks. Given this kind of work, it will fly; any other kind of work is more efficiently executed by the CPU. On the other hand, if a program requires this kind of work, or if an algorithm can be executed in a data-parallel way, it is a tremendous waste *not* to use the GPU.

GLSL

Consider the following GLSL program:

```
void mainImage( out vec4 O, in vec2 pos )
{
    vec2 res = iResolution.xy;
    vec2 centre = vec2( 0.5, 0.5 );
    vec2 d = pos / res - centre;
    if (length( d ) < .2) O.xyz = vec3( 1, 0, 0 );
    else O.xyz = vec3( 0, 0, 0 );
}
```



The output of this program is shown on the right. You can try it out on Shadertoy: just paste the above code and run. An interesting thing about this code is that *there is no loop*: the function is executed for every pixel of the image, yielding the red disc. This is a good example of a data parallel task:

- the input for every thread (in this case: a pixel) is the location of the pixel;
- the output is either red or blue, stored in O;
- threads can be executed in any order, and in parallel;
- threads do not exchange information.

Tasks like this are particularly suitable for GPU execution, and naturally benefit from a large number of threads. It may seem that this kind of task is rare. However, this is not necessarily the case: think of particle systems, neural networks, evaluating the AI for an army of virtual tanks, intersecting a million rays with complex geometry, or running a fluid simulation. In other cases an algorithm may be rephrased so that it becomes data parallel. Some sorting algorithms benefit from parallelism, while others do not, for example.

OpenCL

GLSL (OpenGL Shading Language), as the name suggests, is still firmly rooted in the world of graphics. For general purpose computations something more generic is needed. Several programming languages exist for this: well-known examples are CUDA and OpenCL. This document focuses on OpenCL, but most concepts translate well to CUDA.

OpenCL is roughly based on C, which means that for C/C++ and C# programmers the syntax is familiar. For C# programmers, the biggest hurdle is perhaps the use of pointers. Consider the following code snippet, adapted from Wikipedia, which multiplies a matrix and a vector:

```
// multiplies A*x, leaving the result in y.
// A is a row-major matrix, meaning the (i,j) element is at A[i*ncols+j].
__kernel void matvec( __global float* A, __global float* x,
                      uint ncols, __global float* y )
{
    int i = get_global_id( 0 );            // global id, used as the row index
    __global float* a = &A[I * ncols];     // pointer to the i'th row
    float sum = 0;                         // accumulator for dot product
    for( int j = 0; j < ncols; j++ ) sum += a[j] * x[j];
    y[i] = sum;
}
```

Here, matrix A is stored as a collection of floats. The corresponding C# code would simply mention 'float[] A', but in C (and OpenCL) we specify the memory address of the first element instead, by writing 'float* A' (pronounced as 'float pointer A'). The __global keyword specifies that A is in *global memory*: GPUs have other types of memory as well, so this needs to be explicitly specified.

The above function is a *kernel function*, which means that we can call it from the CPU (the *host*). A kernel function can call other functions, which we call *device functions*. For those, we simply omit the __kernel keyword.

Kernel functions are typically executed in massive parallel fashion, like the shader in GLSL. In this case, we start as many threads as there are rows in the matrix. OpenCL may execute these threads in any order, or in parallel (that's what we count on!), which means that we can make no assumptions about execution order. Each thread needs to know which row it is working on: we determine this by obtaining the index (or: *global id*) of the thread.

Executing a kernel function is a somewhat complex process. It requires communication between the host and the device: arguments need to be passed, and a 'go' signal must be issued. The CPU also needs to specify how many threads should execute the kernel code. We also must make sure that the data we operate on is already on the device: the GPU can't access host memory, so calling the above matrix multiplication kernel must be preceded by data transfer from system RAM to global device memory.

We will further explore these concepts by converting some existing CPU code to OpenCL. This is a typical use case: after identifying a bottleneck in CPU code and establishing it is indeed suitable for parallel execution on the GPU, we port the functionally complete code section.

Cloo

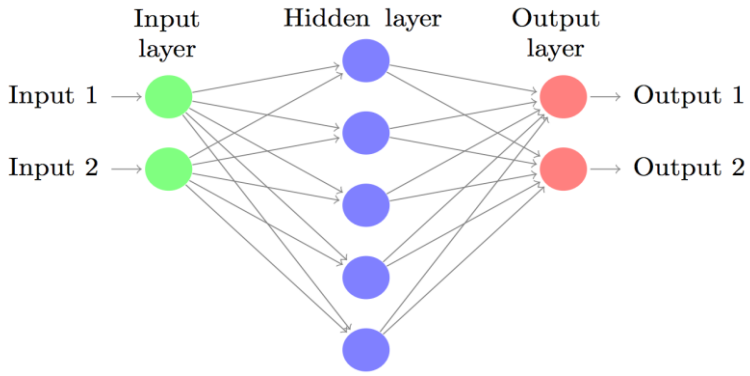
To communicate with OpenCL from C# we will use the Cloo C# wrapper for OpenCL. In the complementary example application Cloo is already included, along with some helper classes that facilitate OpenCL initialization and host-device data transfer.

Neural Network

The example C# application implements a basic backpropagation neural network, and is based on C++ code by Bobby Anguelov. In the example, the net is trained to recognize handwritten characters, using the MNIST training data, which consists of 60,000 categorized images of 28x28 pixels. You can find the implementation of the neural net training code in source file `neuralnet.cs`.

This is obviously not a tutorial on neural nets, but to better understand the remainder, we'll briefly investigate how things work.

A neural net is a collection of virtual 'brain cells', which take some input and may produce output:



In this image, as in the application, three layers are used. The *input layer* is fed input data. In our case this is a 28x28 pixel image; we therefore use 784 input cells. The input cells are connected to the *hidden layer*. Each connection has a *weight*, which scales the value provided by the input layer. The 150 cells in the hidden layer each sum the incoming weighted values, and if the result exceeds a certain threshold, a value is sent to the *output layer*, again using weighted connections. In the application the output layer consists of 10 cells: one for each number between zero and nine. We read the result by finding the output with the highest value: this is the 'best guess' of the net. This functionality is implemented in the `Evaluate` function in `neuralnet.cs`.

Training the network is a matter of adjusting the weights. The functionality for this can be found in the `Backpropagate` function, which uses the difference between the result produced by Evaluate and the desired result to improve the weights.

We train the network using a large number of images, for which we know the correct classification. Training happens in waves, called *epochs*. For each epoch, we feed the network all images of the training set, and validate the result using a second (smaller) set of different images. This functionality can be found in function `RunEpoch`.

Running the original code makes clear that training is time consuming: a single epoch, using only 4,000 of the MNIST images, takes roughly four seconds. The network requires about 200 epochs to reach a reasonable level of accuracy, and reaches optimal (not perfect) accuracy after about 600 epochs, which takes about 40 minutes.

At the same time, there appears to be plenty of room for parallel execution: there are no loop dependencies in the for loops in Evaluate and Backpropagate, which suggests we can easily update many cells at the same time.

Data Layout

The data for the neural net is specified in class NeuralNet. We have simple arrays of floats for the cells themselves: `inputNeurons`, `hiddenNeurons` and `outputNeurons`. Additional arrays store the weights for the connections between the input layer and the hidden layer (`weightsInputHidden`) and between the hidden layer and the output layer (`weightsHiddenOutput`).

The training set is stored in a separate class `TrainingSet`, which contains an array of `TrainingEntry`'s. Each TrainingEntry stores greyscale values for the pixels (scaled to 0..1 and stored as floats), and a set of expected values: 0 for each wrong number, 1 for the correct one.

Finally, we have a collection of data needed for backpropagation: `deltaInputHidden`, `deltaHiddenOutput`, `errorGradientsHidden` and `errorGradientsOutput`.

OpenCL Data Layout

In OpenCL we have plenty of options to replicate the above layout. We can make structs, use type_defs, and so on, e.g.:

```
typedef struct myRecord
{
    int        id;
    char       chars[2];
    int        numerics;
    float      decimals;
    float3     vector;
} record;
```

However, there is a problem. Often, an OpenCL struct does *not* have the same layout as the same struct in C# or C/C++. Sometimes, types do not have the same size: e.g., a `float3` is 16 bytes in OpenCL, not 12, as you would expect. In other cases, *data alignment* is a problem. In the above structure, `id` is four bytes, and is located right at the start of each instance of `record`. Array `chars` is located at offset 4, but `int numerics` is located at offset 8, not 6 as you might expect. The rules for data alignment differ between C/C++, C# and OpenCL, which may lead to quite unpredictable and hard to debug situations.

We can significantly reduce data layout problems if we handle the layout ourselves. The neural net itself for example consists of 784 floats for the input layer, 150 for the hidden layer and 10 for the output layer. Added to that we store 784 * 150 weights for the connections between the input layer and hidden layer, and 150 * 10 weights to connect the hidden layer and output layer. The network can thus be stored in a simple float array large enough to contain all these values:

```
float[] ND = new float[128 * 1024]; // 128KB of neural net data
```

Accessing the input layer is now done using a set of simple helper functions that return the location of data in the float array:

```
int inputNeuron( i ) { return i; }
int hiddenNeuron( int i ) { return i + 800; }
int outputNeuron( int i ) { return i + 960; }
int weightInputHidden( int i, int h ) { return 1024 + h * 800 + i; }
int weightHiddenOutput( int h, int o ) { return 124 * 1024 + h * 10 + o; }
```

To access input neuron x, we now write:

```
ND[inputNeuron( x )] = 0.0f;
float neuronValue = ND[inputNeuron( x )];
```

To access the weight of the connection between input neuron A and hidden neuron B we write:

```
float weight = ND[weightInputHidden( A, B )];
```

There are two benefits to this approach:

1. It is completely clear where each float is, and therefore we can use the same data layout in C# and OpenCL;
2. The entire neural net is in a single array, which can be conveniently transferred to the GPU in a single copy.

The second advantage is important: copies to and from the GPU take time. This time is dominated by the *overhead* of each transfer, not the size. In practice, it is faster to send a single 10MB buffer to the GPU, than two 1KB buffers.

Doing the same to the training data, we allocate 4,800,000 floats to store 6,000 sets of 800 floats. Each set of 800 floats contains 784 inputs (a 28x28 image to train the net with) and 10 floats for the expected output. The original data used integers, but this is not necessary for correct operation.

```
float[] TD = new float[4800000];
// training data access
int input( int i, int n ) { return i * 800 + n; }
int expected( int i, int n ) { return i * 800 + 788 + n; }
```

Finally, we do the same for the delta and gradient data used for backpropagation.

```
float[] DG = new float[128 * 1024];
// delta / gradient data access
int deltaInputHidden( int i, int h ) { return h * 800 + i; }
int deltaHiddenOutput( int h, int o ) { return 118 * 1024 + h * 10 + o; }
int errorGradientHidden( int i ) { return 127 * 1024 + i; }
int errorGradientOutput( int i ) { return 127 * 1024 + 800 + i; }
```

Once we have the new data layout in place, we can modify the C# code to use it. This is quite a bit of work, but it allows us to verify the layout, and once we have the OpenCL code in place, it allows us to switch between OpenCL and C# execution, even for parts of the code.

Here is the converted ‘update hidden neurons’ loop of the Evaluate function:

```
for( int i = 0; i < NUMHIDDEN; i++ )
{
    ND[hiddenNeuron( i )] = 0;
    // get weighted sum of pattern and bias neuron
    for( int j = 0; j <= INPUTSIZE; j++ )
    {
        ND[hiddenNeuron( i )] += ND[inputNeuron( j )] * ND[weightInputHidden( j, i )];
    }
    // apply activation function
    ND[hiddenNeuron( i )] = SigmoidActivationFunction( ND[hiddenNeuron( i )] );
}
```

The fully converted code can be found in the second version of the neural network application, NN_prep. Executing it reveals that is works, and still runs at the same speed.

With the modified data in place, we are well prepared to start the conversion to OpenCL.

PART 2 – ACTUAL CONVERSION

We will start the actual conversion with a cleaned-up version of the NN_prep project. In the NN_clean project you will find the same code, but without all the original lines in comments. Additionally, some basic C# multithreading has been used to verify the lack of loop dependencies. See functions `Evaluate` and `Backpropagate` for examples. These changes also bring a bit of extra speed: the network now trains in less than half the original time, on my poor-man's Intel i3 system.

OpenCL Buffers

Before we do any OpenCL work we must initialize the API. The easiest way to do this is using the supplied `OpenCLProgram` helper class, which initializes an OpenCL context, finds a suitable device (most systems have more than one OpenCL capable device), loads the specified OpenCL source file, and compiles it.

```
static OpenCLProgram ocl = new OpenCLProgram( "../../program.cl" );
```

Once we have an OpenCL context, we can create OpenCL buffers, using the `OpenCLBuffer` helper class. It's constructor conveniently takes regular C# arrays:

```
OpenCLBuffer<float> gpuND = new OpenCLBuffer<float>( ocl, ND );
OpenCLBuffer<float> gpuTD = new OpenCLBuffer<float>( ocl, TD );
OpenCLBuffer<float> gpuDG = new OpenCLBuffer<float>( ocl, DG );
```

An OpenCLBuffer object actually encapsulates two buffers: one on the host, and one on the device. This is convenient: we typically want to prepare data on the CPU, send it to the GPU, operate on it using a kernel, and get it back to the CPU. `OpenCLBuffer` methods `CopyToDevice` and `CopyFromDevice` facilitate this.

My First Kernel

Now that we have initialized OpenCL and created some buffers we can test the logistics. In program.cl we put the following kernel code:

```
__kernel void device_function( __global float* a )
{
    int id = get_global_id( 0 );
    a[id] = id;
}
```

This kernel takes a single float buffer, and fills it with thread ids. We access it in the compiled `OpenCLProgram` by instantiating an `OpenCLKernel`:

```
OpenCLKernel testKernel = new OpenCLKernel( ocl, "device_function" );
```

The kernel takes a single parameter. We can pass it our neural net data to test it:

```
testKernel.SetArgument( 0, gpuND );
```

After making sure that gpuND is actually on the device, we execute the kernel, and fetch the data back to the CPU:

```
gpuND.CopyToDevice();
testKernel.Execute( 10000 );
gpuND.CopyFromDevice();
```

If we set a breakpoint after the last line, we can verify the contents of the float buffer: the first 1,000 elements now contain the values 0..1000, which means that we made the GPU do some work.

Porting Backpropagate

The most time consuming part of the training process is the code in Backpropagate. There are two loops in this function. The first one is:

```
// modify deltas between hidden and output layers
for( int i = 0; i < NUMOUTPUT; i++ )
{
    // get error gradient for every output node
    DG[errorGradientOutput( i )] = GetOutputErrorGradient(
       TD[expected( entryIdx, i )], ND[outputNeuron( i )] );
}
```

In the NN_clean project, the second loop is the Parallel.For. Not that there is a dependency between the two loops: the second loop uses data set by the first one, but in a different order. To ensure that the first loop completes before we start the second one, we will put each loop in its own kernel.

The kernel that implements the first loop looks like this:

```
__kernel void Backpropagate1( int entryIdx, __global float* ND,
                              __global float* TD, __global float* DG )
{
    int id = get_global_id( 0 ); // get thread id, will be 0..NUMOUTPUT
    // get error gradient for every output node
    DG[errorGradientOutput( id )] = GetOutputErrorGradient(
       TD[expected( entryIdx, id )], ND[outputNeuron( id )] );
}
```

This kernel receives as arguments pointers to the float arrays that hold the data for the neural network, and the index of the current training set entry in entryIdx. The loop is gone: this will now be executed in parallel. The original loop had NUMOUTPUT iterations; we will thus execute this kernel using NUMOUTPUT threads. The actual functionality is identical to the C# code, thanks to our data reorganization efforts. Like the C# code, the kernel calls a few functions to find data in the arrays. These functions are also identical to their C# counterparts:

```
int outputNeuron( int i ) { return i + 960; }
int expected( int i, int n ) { return i * 800 + 788 + n; }
int errorGradientOutput( int i ) { return 127 * 1024 + 800 + i; }
```

There is also a call to GetOutputErrorGradient. This function, too, is identical to the C# code:

```
float GetOutputErrorGradient( float desiredValue, float outputValue )
{ return outputValue * (1.0f - outputValue) * (desiredValue - outputValue); }
```

We can now test the partial port. The safest (albeit slow) way to do this is like this:

```
if (true)
{
    kernelBackprop1.SetArgument( 0, entryIdx );
    kernelBackprop1.SetArgument( 1, gpuND );
    kernelBackprop1.SetArgument( 2, gpuTD );
    kernelBackprop1.SetArgument( 3, gpuDG );
    gpuND.CopyToDevice();
    gpuTD.CopyToDevice();
    gpuDG.CopyToDevice();
    kernelBackprop1.Execute( NUMOUTPUT );
    gpuND.CopyFromDevice();
    gpuTD.CopyFromDevice();
    gpuDG.CopyFromDevice();
}
else
{
    // modify deltas between hidden and output layers
    for( int i = 0; i < NUMOUTPUT; i++ )
    {
        // get error gradient for every output node
        DG[errorGradientOutput( i )] = GetOutputErrorGradient(
           TD[expected( entryIdx, i )], ND[outputNeuron( i )] );
    }
}
```

In other words: transfer a copy of the current data in RAM to the GPU; execute the kernel; copy the data back. After this, the situation should be identical to the result of executing the C# code (in the 'false' branch). In the above code, we can swap effortlessly between OpenCL and C#, until we are sure the OpenCL kernel works as intended. And, we do this for a tiny portion of the code, which allows us to port in small chunks.

This does however make the neural net horribly slow. Although this is not really a problem right now, there are a few things that we can do to alleviate this. First of all: training data never changes. So, after setting up `TD` and `gpuTD` we can copy it once, and skip the copies after that. Secondly, the above code doesn't change any data in `ND`, so we don't have to copy that back either. And finally, arguments 1, 2 and 3 are always the same: we can thus set these once. The 'true' branch then becomes:

```
    kernelBackprop1.SetArgument( 0, entryIdx );
    gpuND.CopyToDevice();
    gpuDG.CopyToDevice();
    kernelBackprop1.Execute( NUMOUTPUT );
    gpuDG.CopyFromDevice();
```

The port of the second loop proceeds in the same way. The kernel code:

```
__kernel void Backpropagate2( __global float* ND, __global float* DG )
{
    int id = get_global_id( 0 ); // get thread id, will be 0..NUMHIDDEN
    // modify deltas between input and hidden layers
    for( int k = 0; k < NUMOUTPUT; k++ )
    {
        // calculate change in weight
```

```
        DG[deltaHiddenOutput( id, k )] =
           LEARNINGRATE * ND[hiddenNeuron( id )] * DG[errorGradientOutput( k )] +
           MOMENTUM * DG[deltaHiddenOutput( id, k )];
    }
    // get error gradient for every hidden node
    DG[errorGradientHidden( id )] = GetHiddenErrorGradient( id, ND, DG );
    // for all nodes in input layer and bias neuron
    for( int j = 0; j <= INPUTSIZE; j++ )
    {
        // calculate change in weight
        DG[deltaInputHidden( j, id )] =
           LEARNINGRATE * ND[inputNeuron( j )] * DG[errorGradientHidden( id )] +
           MOMENTUM * DG[deltaInputHidden( j, id )];
    }
}
```

Porting UpdateWeights

The final code snippet in the Backpropagate function is a call to `UpdateWeights`. The C# code contains two loops:

```
void UpdateWeights()
{
    // input -> hidden weights
    Parallel.For( 0, INPUTSIZE + 1, i => { for( int j = 0; j <= NUMHIDDEN; j++ )
        ND[weightInputHidden( i, j )] += DG[deltaInputHidden( i, j )];
    } );
    // hidden -> output weights
    Parallel.For( 0, NUMHIDDEN + 1, i => { for ( int j = 0; j < NUMOUTPUT; j++ )
        ND[weightHiddenOutput( i, j )] += DG[deltaHiddenOutput( i, j )];
    } );
}
```

Based on earlier experience, we could split this into two kernels. However: the two loops can run concurrently; they operate on different data. This reveals an important problem that we ignored so far: our kernels run 10 threads if they parallelize a loop over the output neurons, 150 for the hidden neurons, and 784 for the input neurons. None of these numbers is even close to what we need to keep the GPU occupied. Recall that a high-end GPU is designed to run 3,840 threads in parallel, chosen from 61,440 threads. What is worse: it *needs* those numbers to hide latencies arising from memory access. Looking at the kernels we have so far it is clear that execution is dominated by memory access: there is not a lot of calculation going on, but we do loop over massive arrays.

Consider the following OpenCL port of `UpdateWeights`:

```
__kernel void UpdateWeights( __global float* ND, __global float* DG )
{
    int id = get_global_id( 0 ); // get thread id, will be 0..INPUTSIZE
    // input -> hidden weights
    for( int j = 0; j <= NUMHIDDEN; j++ )
        ND[weightInputHidden( id, j )] += DG[deltaInputHidden( id, j )];
    // hidden -> output weights
    if (id <= NUMHIDDEN) for ( int j = 0; j < NUMOUTPUT; j++ )
```

```
        ND[weightHiddenOutput( id, j )] += DG[deltaHiddenOutput( id, j )];
}
```

Here, we execute this kernel using 784 threads (to suit the first loop). For the second loop, we disable all threads beyond 150 using a simple if-statement. This means that a substantial number of threads will be idling while 150 threads execute the second loop. This is not an issue: we have *thousands* of idling threads anyway. Combining the two loops in one kernel at least saves on kernel invocation overhead, making this the faster option. Note that this is *only* possible here because the two loops are independent!

Finishing the OpenCL Port

The goal of the remainder of the port is to move all functionality in `RunEpoch` to the GPU. Once this is done, we can get rid of most of the transfers; the only thing we really need back after training with one image is the updated neural network. This will not be described in detail here; the steps are similar to what we did for Backpropagate. You can see the result in the NN_OpenCL project.

Porting the `Evaluate` function brings up one interesting issue. `Evaluate` contains this loop:

```
for( int j = 0; j <= NUMHIDDEN; j++ ) for( int i = 0; i < NUMOUTPUT; i++ )
// can't (easily) do in parallel; fights for outputNeuron[i]
{
    // get weighted sum of pattern and bias neuron
    ND[outputNeuron( i )] += ND[hiddenNeuron( j )] * ND[weightHiddenOutput( j, i
)];
}
```

The comment points out an issue: if we run the outer loop in parallel, multiple threads may add to `ND[outputNeuron( i )]` at the same time. This is only safe if we can do this atomically. On the CPU, atomics are expensive. On the GPU however, they are very efficient: in fact, an atomic write can be done at virtually the same cost as a regular write to global memory.

OpenCL provides a broad set of atomic functions:

- atomic_add / sub for addition and subtraction;
- atomic_inc / dec for atomic increment and decrement;
- atomic_and / or / xor for atomic logical operations;
- atomic_min / max for atomically determine minima and maxima;
- atomic_xchg / cmpxchg for (conditionally) swapping numbers.

All these functions have one thing in common: they operate on integers. This exposes a major flaw of OpenCL: on NVidia GPUs, atomic operations on floats are supported natively, but OpenCL does not expose this. In CUDA we can simply use atomicAdd on floats. In OpenCL we need a hack.

Anca Hamuraru and Vincent Hindriksen propose the following solution on streamhpc.com:

```
inline void AtomicAddFloat( volatile __global float* source, const float operand )
{
	union { unsigned int intVal; float floatVal; } newVal;
	union { unsigned int intVal; float floatVal; } prevVal;
	do {
		prevVal.floatVal = *source;
		newVal.floatVal = prevVal.floatVal + operand;
```

```
        }
        while (atomic_cmpxchg( (volatile __global unsigned int*)source,
               prevVal.intVal, newVal.intVal ) != prevVal.intVal);
}
```

Although this is nowhere near as fast as the NVidia / CUDA solution, at least this allows us to run that particular loop in parallel.

Evaluation

The full OpenCL port of the neural network can be found in project NN_OpenCL. When we run it, it turns out that the GPU executes the training process slightly faster than a single CPU core, which is at least something. But, it is not faster than the C# version with Parallel.For. What is wrong?

Several things:

- As discussed, we don't have enough work. Invoking a kernel with just 10 threads is a waste of time: although the kernel will be executed in the blink of an eye, the overhead of the invocation is not nearly worth it. Running these tiny kernels on the CPU is not an option however: this would require constant synchronization of data between GPU and CPU. So, an inefficient kernel invocation may actually be worth it.
- Many loops have dependencies. Backpropagate was split in three kernels (one for UpdateWeights); Evaluate was split in no less than four kernels. Training with 4000 images thus requires 4000 * 7 kernel invocations, plus data transfer. Bringing down the number of kernel invocations would help.
- The AtomicAdd is inefficient. The proposed hack works well for occasional atomic adds, because in that case the condition for the while will fall through at the first attempt. We are however hammering an array.

The solution is simple: *do more work*. Although the port is weak for a small network, the code will run at virtually the same speed if we double the neuron counts. Training with tens of thousands of input neurons and similar counts for the hidden layer is not feasible on the CPU, but perfectly possible on the GPU.

In the final part of this document we will investigate how training can be made more efficient *without* increasing the workload.

PART 3 – SPEED

In part 2 we converted a C# application to OpenCL. Sadly, the result was disappointing in terms of performance. A number of causes for this were identified. In this final part, we will improve the performance of the code, without increasing the amount of work. This will provide additional insight in factors that determine OpenCL application performance.

Establishing Baseline

Before we start the optimization process it is good to record our current performance level. On my machine (Intel i3, Titan XP Pascal) a single Epoch takes about 2960 milliseconds for the NN_OpenCL project.

Function `Evaluate` has been reduced to four kernel invocations. No data is copied back to the CPU; `Backpropagate` operates on the same data and needs this on the GPU as well. Function `Backpropagate` invokes three kernels. It concludes with a transfer of the updated neural net to the host. This data is needed for the remainder of the RunEpoch function, which estimates the mean square error (MSE).

Reducing Transfer

The remaining `CopyFromDevice` in `Backpropagate` transfers 128KB of data. This is a tiny amount, so the transfer cost is expected to be dominated by transfer overhead. As mentioned, the transfer is needed to update the MSE in RunEpoch:

```
// check all outputs from neural network against desired values
bool resultCorrect = true;
for( int j = 0; j < NUMOUTPUT; j++ )
{
    if (ND[clampedOutput( j )] != TD[expected( entryIdx, j )])
        resultCorrect = false;
    float delta = ND[outputNeuron( j )] - (int)TD[expected( entryIdx, j )];
    MSE += delta * delta;
}
if (!resultCorrect) incorrectEntries++;
```

This is a loop that should not benefit from running on the GPU: we can spawn only 10 threads, and on top of that, adding `delta * delta` to `MSE` again requires an atomic addition. But, if this code runs on the GPU, it operates on data on the GPU, which saves us the remaining `CopyFromDevice`.

The kernel looks like this:

```
__kernel void UpdateMSE( int entryIdx, __global float* ND, __global float* TD )
{
    int id = get_global_id( 0 );
    // check all outputs from neural network against desired values
    bool resultCorrect = true;
    for( int j = 0; j < NUMOUTPUT; j++ )
    {
```

```
        if (ND[clampedOutput( j )] != TD[expected( entryIdx, j )])
            resultCorrect = false;
        float delta = ND[outputNeuron( j )] - TD[expected( entryIdx, j )];
        AtomicAddFloat( &ND[MSE()], delta * delta );
    }
    if (!resultCorrect) ND[incorrectEntries()] += 1.0f;
}
```

For this to work, the local variables `MSE` and `incorrectEntries` used in RunEpoch were moved to `ND`. Once this final part of the loop over 4000 images runs on the GPU, transferring back the trained network is only needed after this loop, which saves us 3999 transfers.

Despite the terrible kernel there is a positive impact on performance: an epoch now takes 2420 milliseconds, a 22% reduction.

Better Hardware Utilization

Due to the small workload, the GPU hardware is severely under-utilized when running the kernels. In fact, when running just 10 threads, these will be assigned to a single SM, leaving all other SMs to idle. This is a shame: each SM has its own cache and memory interface, so in a memory-intensive application like this it could be beneficial to spread the work over more SMs.

When we dispatch some work to OpenCL, the system automatically takes care of the division of work over SMs. We can however influence this behaviour, by creating *work groups*. Each work group is guaranteed to run on a single SM. Normally, due to the 32-wide SIMD nature of the GPU, a workgroup consists of at least 32 threads. In our case, it could be better to have much smaller work groups. For this, a version of the `Execute` function exists that takes two parameters: the size of the workload, and a work group size.

Using the two-argument version of Execute comes with a caveat: the size of the workload *must* be a multiple of the work group size. This is handled internally (see the implementation of `Execute` in `opencl.cs`), but it does mean that some threads may be started that are outside the intended range. To counter this, a small addition must be made to each kernel. Here is the Evaluat4 kernel:

```
__kernel void Evaluate4( __global float* ND )
{
    int id = get_global_id( 0 ); // get thread id, 0..NUMOUTPUT are valid
    if (id >= NUMOUTPUT) return;
    // apply activation function and clamp the result
    ND[outputNeuron( id )] = SigmoidActivationFunction( ND[outputNeuron( id )] );
    ND[clampedOutput( id )] = ClampOutputValue( ND[outputNeuron( id )] );
}
```

Right after fetching the thread id, any thread that is outside the valid range simply terminates.

We can now start each kernel with a specific work group size. For work groups of 8 threads, the performance improves to 1633 milliseconds, 67% of the time it took without this optimization. On my hardware, just 2 threads per work group seems optimal; performance improves to 1467 milliseconds, which is only 61% of the previous time. The GPU now outperforms the multi-threaded CPU code, albeit by a tiny margin.

Reducing Kernel Invocations

Kernels `Evaluate4` and `Backpropagate1` are executed right after each other, without any other logic between them. Both run with `NUMOUTPUT` threads. The only real reason that the code in these kernels is split over two kernels is the original code architecture, where these loops where located in separate functions.

This allows us to investigate kernel invocation overhead: obviously, each kernel takes no time at all to complete, so joining them should get rid of the overhead, without adding any work. `Backpropagate1` now becomes:

```
__kernel void Backpropagate1( int entryIdx, __global float* ND,
                              __global float* TD, __global float* DG )
{
    int id = get_global_id( 0 ); // get thread id, 0..NUMOUTPUT are valid
    if (id >= NUMOUTPUT) return;
    // apply activation function and clamp the result (originally: Evaluate4)
    ND[outputNeuron( id )] = SigmoidActivationFunction( ND[outputNeuron( id )] );
    ND[clampedOutput( id )] = ClampOutputValue( ND[outputNeuron( id )] );
    // get error gradient for every output node
    DG[errorGradientOutput( id )] = GetOutputErrorGradient(
       TD[expected( entryIdx, id )], ND[outputNeuron( id )] );
}
```

The runtime of an epoch now goes down to 1440 milliseconds. Not the magic bullet we hoped for, but still a reduction.

Conclusion

Comments, questions? Mail me:

bikker.j@gmail.com

Houten, September 7th, 2017.