GPU Skillpill Problem Set

James Schloss
Irina Reshodko

Outline

You are not intended to do all of these problems. Instead, please pick one that suits your interests and go from there. The final session of the GPU skillpill will be exclusively devoted to starting (but maybe not completing) one of these problems.

1. **Vector Addition in OpenCL:** For GPU computing, CUDA's major competitor is OpenCL; however OpenCL works much differently under the hood and is more powerful in many ways. This power comes at a cost of more cumbersome boilerplate code, but that's worth it in a few cases!

2. **Matrix Transposition in CUDA:** This one is deceptively complicated and involves taking a matrix and switching the x and y indices. The simple way is just to mess around with gridding, but the hard (and efficient) way uses a shared memory tile, which is pretty cool!

3. **Using the cuFFT library:** This is a simple example of the cufft library and how it works

4. **Sobel filter with FFT (or not):** Basically, finish what we started last Thursday. If you want, use the cuFFT library!

5. **Parallel Reduction:** this is also sometimes called *parallel summation.* The idea's simple: we have an array and we want to add all the elements together, leaving the sum of all elements in the first position `a[0]`.

Vector Addition in OpenCL

If you have ever played video games, you might be aware of the heated debate between DirectX and OpenGL for rendering graphics. The former is Windows-specific and designed by Microsoft, while the latter is cross-platform and designed by the Khronos group. In the early days of GPU computing, rather than using compute-specific languages, programmers used OpenGL to do computation. Now, CUDA is king, but similar to how DirectX only works on Windows machines, CUDA only works on nVidia cards. If you have an AMD card, you are out of luck, unless you use CUDA's competitor: OpenCL!

OpenCL stands for the Open Computing Language and is meant to allow users to access any type of hardware with the same language. When using GPU hardware with OpenCL, you can expect similar (sometimes faster) performance with OpenCL when compared to CUDA; however, it comes at a cost of bulky boiler-plate code. The extra code is mainly associated with choosing appropriate devices to work on and with and managing the memory of those devices. In addition, it is clear that the OpenCL API is significantly influenced by the same design philosphy of OpenGL, so if you want to make games later, this might be a good exercise to go through!

First things first, when compiling OpenCL code, the compiler will be your standard c++ compiler `g++` or `clang++`. Because of this, you need to make sure you have the appropriate libraries installed (which your local supercomputer should!). On Tombo, this means using the same `cuda\7.5.18` module you have been using. Compilation is a little different, though:

```
g++ code.cpp -I/apps/free/cuda/7.5.18/include
    -L/apps/free/cuda/7.5.18/lib64 -lOpenCL
```

Note that we are using the `-I` and `-L` flags to include the appropriate directory. In most programs, these flags will be found in the `makefile`.

Now that we know how to compile, let's talk about how OpenCL differs from CUDA. Firstly, the headers:

```
#define __CL_ENABLE_EXCEPTIONS

#include <CL/cl.hpp>
#include <iostream>
#include <vector>
#include <math.h>
```

Here, we are using `__CL_ENABLE_EXCEPTIONS` so we may play some tricks later. Outside of the `CL/cl.hpp` heading, though, everything should be good.

NOTE: We will be using the C++ API to OpenCL instead of C. Ultimately, the C++ API calls the C API, so it's the same thing. C++ just allows us to use some more interesting features of the language.

Now for the kernel. In OpenCL, kernels are compiled at runtime, which might sound like an awful idea! On the other hand, it means that we can change the kernels on-the-fly without needing to recompile the code to use different functions. That's nice in it's own way. Ultimately, kernels are read in as strings (usually in separate files):

```
// OpenCL kernel
const char *kernelSource =                                 "\n" \
"#pragma OPENCL EXTENSION cl_khr_fp64 : enable              \n" \
"__kernel void vecAdd( __global double *a,                  \n" \
"                      __global double *b,                  \n" \
"                      __global double *c,                  \n" \
"                      const unsigned int n){               \n" \
"    // Global Thread ID                                    \n" \
"    int id = get_global_id(0);                             \n" \
"                                                           \n" \
"    // Remain in bounds...                                 \n" \
"    if (id < n){                                           \n" \
"        c[id] = a[id] + b[id];                             \n" \
"    }                                                      \n" \
"}                                                          \n" \
                                                           "\n" ;
```

Overall, the code is quite similar to the CUDA kernel, except that `get_global_id(0)` is a little different than `blockIdx.x * blickDim.x + threadIdx.x`. For OpenCL, 0, 1, and 2 are x, y, and z.

Now for the main code...

```
    // length of vectors
    unsigned int n = 1000000;

    // host vectors
    double *h_a, *h_b, *h_c;

    h_a = new double[n];
    h_b = new double[n];
    h_c = new double[n];
```

```
// Setting all elements to 1
for (int i = 0; i < n; ++i){
    h_a[i] = 1;
    h_b[i] = 1;
}

// device vectors
cl::Buffer d_a, d_b, d_c;

cl_int err = CL_SUCCESS;
```

Here there are some obvious differences with initialization. For one, we can use the C++ keyword `new` which helps because we do not need to `free()` the variables later. In addition, it's clear that OpenCL calls everything on the GPU a `cl::Buffer`, to differentiate between what's on the CPU and GPU.

Now for the bulk of the code. Note we will have most of it in a `try` / `catch` conditional. This is another pitfall of OpenCL: debugging's a pain!

Firstly, we query for platforms and select the GPU.

```
try{

    // First, Query for platforms
    std::vector<cl::Platform> platforms;
    cl::Platform::get(&platforms);
    if (platforms.size() == 0){
        std::cout << "Platform size 0\n";
        return -1;
    }

    // Get list of devices on default platform and create context
    cl_context_properties properties[] =
        { CL_CONTEXT_PLATFORM,
            (cl_context_properties)(platforms[0])(), 0};

    cl::Context context(CL_DEVICE_TYPE_GPU, properties);
    std::vector<cl::Device> devices =
        context.getInfo<CL_CONTEXT_DEVICES>();
```

Next, we create a `commandQueue`, which is basically the queue of commands to implement. The first command is writing the buffers.

```
    // Create command queue on first device
    cl::CommandQueue queue(context, devices[0], 0, &err);
```

```
// Create device memory buffers
d_a = cl::Buffer(context, CL_MEM_READ_ONLY,
    n*sizeof(double));
d_b = cl::Buffer(context, CL_MEM_READ_ONLY,
    n*sizeof(double));
d_c = cl::Buffer(context, CL_MEM_WRITE_ONLY,
    n*sizeof(double));

// Bind memory buffers
queue.enqueueWriteBuffer(d_a, CL_TRUE, 0, n*sizeof(double),
    h_a);
queue.enqueueWriteBuffer(d_b, CL_TRUE, 0, n*sizeof(double),
    h_b);
```

Finally, we build the kernel, set the arguments, run it and get the output.

```
// Build kernel from source string
cl::Program::Sources source(1,
    std::make_pair(kernelSource,strlen(kernelSource)));
cl::Program program_ = cl::Program(context, source);
program_.build(devices);

// Create kernel object
cl::Kernel kernel(program_, "vecAdd", &err);

// bind kernel arguments to kernel
kernel.setArg(0, d_a);
kernel.setArg(1, d_b);
kernel.setArg(2, d_c);
kernel.setArg(3, n);

// Number of work items in each work group
cl::NDRange localSize(64);

// Number of total work items -- localsize must be a divisor
cl::NDRange globalSize((int)(ceil(n/(float)64)*64));

// Enqueue kernel
cl::Event event;
queue.enqueueNDRangeKernel(
    kernel,
    cl::NullRange,
    globalSize,
    localSize,
        NULL,
        &event
    );

    // Wait until kernel completion
    event.wait();

    // Read back d_c
    queue.enqueueReadBuffer(d_c, CL_TRUE, 0, n*sizeof(double),
        h_c);
}
```

Finally, the catch statement looks like:

```
catch(cl::Error err){
    std::cerr << "ERROR:
        "<<err.what()<<"("<<err.err()<<")"<<std::endl;
}
```

At this point, you should have the vector c ready to read, so do whatever you like to output it's contents and check that you did things correctly. Don't forget to delete the appropriate arrays!

It's important to note here that you may need to look up the error code for OpenCL to figure out what the error means, which brings me back to my point before: OpenCL is a pain to debug!

Still, it's open-source and can work on all hardware, so a few hassles here and there are not too big of a deal when compared to CUDA.

Matrix Transposition in CUDA

This problem seems straightforward: Take a matrix $A[i, j]$ and turn it into $A[j, i]$. In principle, this can be naively by just creating two separate index distributions and copying the contents of one array appropriately into the next array; however, when compared to a direct copy (no transpose), there will be a *huge* drop in performance, and this is because you lose a property known as *coalescence*. Take a look at the following code:

```
...
int main(){
    double *a, *b;
    int n = 10;
    int index_a, index_b;

    a = (double *)malloc(sizeof(double)*n*n);
    b = (double *)malloc(sizeof(double)*n*n);

    for (int i = 0; i < n; ++i){
        for (int j = 0; j < n; ++j){
            index_a = j + i*n;
            index_b = i + j*n;
            a[index_a] = index_a;
            b[index_b] = index_b;
        }
    }
}
...
```

Here, we have two arrays `a` and `b` and we are populating them with two different indices `index_a = j + i*n` and `index_b = i * j*n`. Both indices will appropriately fill the arrays; however, `index_a` will do so much, much faster because it needs to access successive elements of our psuedo-2D array, which `index_b` has to skip over `n` elements every `j` iteration.

In this way, `index_a` is coalesced, while `index_b` is not.

Now, imagine a simple CPU-based transpose:

```
...
int main(){
    // Initialize a as above
    for (int i = 0; i < n; ++i){
        for (int j = 0; j < n; ++j){
            index_a = j + i*n;
            index_b = i + j*n;
            b[index_b] = a[index_a];
        }
    }
}
...
```

It might appear that no matter what we do, our second index will not be coalesced! However, we can get around this with a shared-memory tile. Now, here's where I get a little lazy. Everything from this point has been properly described here: `https://devblogs.nvidia.com/parallelforall/efficient-matrix-transpose-cuda-cc/`, so here's the problem:

- Perform a simple matrix transpose in CUDA (like the CPU code above, just on the GPU)
- Go here: `https://devblogs.nvidia.com/parallelforall/efficient-matrix-transpose-cuda-cc/`. Make your code efficient

Using the cuFFT library

One operation that has become incredibly efficient on GPU's is the FFT (Fast Fourier Transform). This algorithm lies at the heart of almost every field of research that requires signal processing. For this problem, we assume that you already have an understanding of how the Fourier Transform works and ultimately just want to implement it in code. For a more in-depth description, feel free to look at: `https://leios.gitbooks.io/algorithm-archive/content/chapters/computational_mathematics/FFT/cooley_tukey.html`.

For the most part, the cuFFT library follows syntactically from the FFTW library and will require similar steps. In principle, the library allows for simple n-dimensional FFT's; however, depending on what you need, you may have to play some tricks. For this exercise, we will be focusing exclusively on 1D transforms, for 2D transforms, see the exercise on implementing a Sobel Filter with FFT's. For more complicated transforms, the full documentation for the cuFFT library can be found here: `http://docs.nvidia.com/cuda/cufft/index.html`.

The cuFFT library has a special type called `double2` or `cufftDoubleComplex`, which is simply a complex number for FFT operations. For these values, `number.x` will be the real component and `number.y` will be the imaginary component. In principle, the cuFFT library works like this:

1. include the `<cufft.h>` header file from the your cuda installation. Note that on Sango / Tombo, you may need to make sure that you have used `module load cuda/7.5.18`

2. Create a plan with the `cufftPlan1D()` function. `cufftPlan2D()` and `cufftPlan3D()` are also allowed here, however, if you need more complicated transforms, you might need to use `cufftPlanMany()`, which will not be covered here.

3. Execute plan with the `cufftExecC2C()` function. Note that in this case, the C means that the transform will use a complex to complex transformation. There are other transformations available:

    - `cufftExecR2C` Will execute a transformation from real to complex (usually for a forward transform)
    - `cufftExecC2R` Will execute a transformation from complex to real (usually for an inverse transform)

- For double precision, `cufftExecC2C` = `cufftExecZ2Z`, `cufftExecC2R` = `cufftExecZ2D`, and `cufftExecR2C` = `cufftExecD2Z`

Here is some example code to see how this might look:

```
#include <cufft.h>

int main(){
    // Initializing variables
    int n = 1024;
    cufftHandle plan1d;
    double2 *h_a, *d_a;

    // Allocation / definitions
    h_a = (double2 *)malloc(sizeof(double2)*n);
    for (int i = 0; i < n; ++i){
        h_a[i].x = sin(2*M_PI*i/n);
        h_a[i].y = 0;
    }

    cudaMalloc(&d_a, sizeof(double2)*n);
    cudaMemcpy(d_a, h_a, sizeof(double)*n, cudaMemcpyHostToDevice);
    cufftPlan1d(&plan1d, n, CUFFT_Z2Z, 1);

    // Executing FFT
    cufftExecZ2Z(plan1d, d_a, d_a, CUFFT_FORWARD);

    //Executing the iFFT
    cufftExecZ2Z(plan1d, d_a, d_a, CUFFT_INVERSE);

    // Copying back
    cudaMemcpy(h_a, d_a, sizeof(double)*n, cudaMemcpyDeviceToHost);

}
```

Exercise: Use cuFFT to do a 2D FFT of some 2D sinusoidal field you have created.

Sobel Filter with FFT

Last time on *GPU SkillPill X: The Final Pill*, we implemented a basic Sobel filter with a convolution using a window method. If using this method on an $m \times n$ image with a $p \times q$ filter, the complexity is $\sim \Theta(mnpq)$. By using the Fourier convolutional theorem, we can do the same convolution with a complexity of $\sim \Theta(mn(\log(m) + \log(n)))$, which is better for large filters. Also, using the convolutional theorem is better for data with dimensions higher than 2.

Parallel Reduction

This one is sometimes called “Parallel Summation”, but it ultimately doesn’t matter. Your job is to do the following:

```
int main(){
    double a[10];

    // Initialize a here

    // Summing a
    double sum = 0;
    for (int i = 0; i < n; ++i){
        sum += a[i]
    }
}
```

This seems simple enough on the CPU; however, on the GPU it is not obvious how parallelization will help at all! Simply put, this is a sequential operation, so the parallelized version will still require a for loop; however, it ultimately boils down to a bunch of parallel vector additions, something like this:

```
int main(){
    // Initialize a as above
    // Summing a
    // Note, if n is not a power of 2, just add a bunch of 0's
    for (int num = n/2; num >= 1; num *= 0.5){
        for (int i = 0; i < num; ++i){
            a[i] += a[i+num];
        }
    }
}
```

In this case, the for loop can be quire easily parallelized on the GPU, while the outer for loop on the outside cannot. It should be clear that by performing this operation, the sum will appear in the first element of the array. Your job is to implement this summation / reduction on the GPU. Good luck!

More information can be found here: `http://www.drdobbs.com/architecture-and-design/parallel-pattern-7-reduce/222000718`