ECE 572 CUDA Project: FFT / Convolution

Ryan Albright, Chris Kirkpatrick

December 9, 2010
Introduction
This report will cover CUDA, a parallel programming tool that utilizes the unique multi-core architecture of an nVidia GPU. The report will then cover how CUDA works, and why it is useful. Then I will go over an example that utilizes CUDA vs a standard sequential programming interface and reflect on my results.

About Nvidia CUDA
Due to the limitations of “Central Processing Units” due to power consumption, speed, or size computing is drifting away from the single-processor scheme. It is instead moving towards “co-processing” where the computational load is shared between a CPU and GPU (Graphical Processing Unit). Nvidia recognized this need for parallel computing and invented the CUDA parallel computing architecture and is implementing this in all current Graphics Cards (GeForce, ION, Quadro, and Tesla). This is significant because Nvidia is a major part of the Graphics industry and already has a good foothold on the market. CUDA allows for the CPU to request computational support from the GPU which has many individual cores that can each perform simple instructions allowing for rapid processing of large quantities of data.

How CUDA works
CUDA creates a means for rapid parallel computing on the GPU instead of overloading and waiting on a CPU. Almost every programming application can benefit from having the ability parallelize the load. One of the more basic examples of this would be simple matrix multiplication. In this application, the processor must perform many multiplications and additions for each element of the matrix. These simple instructions can easily be passed to the GPU and done in parallel.

This ability to parallelize computations is easy to see given the architecture of the GPU vs th CPU. The GPU is comprised of many (100’s - 1000’s) of cores and is increasing this number every generation of Graphics Cards. CPU’s have been doubling their parallel abilities every few years moving from single core to dual core, and now up to 6 or even 8 cores (Cell Processor) but this doesn’t really compare. In the example later in this report, I used a single consumer grade GPU priced at <$200 that comes equipped with 192 Cores.

CPU/GPU Architecture Comparison

![CPU and GPU Architecture Diagram]

This works because the CPU can load information into the GPU’s memory, request a computation, and then ask for the result back. This is done in the following manner:
FFT Example: Convolution

Convolution Background Information:

One simple example of how the GPU can help speed up processing time is using the CUDA instructions for FFT (Fast Fourier Transform). FFT is a faster implementation of the DFT (Discrete Fourier Transform) which is used extensively in signal processing. The DFT definition is as follows:

\[ X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi k \frac{n}{N}} \quad k = 0, \ldots, N - 1 \]

Evaluating this definition requires \( O(N^2) \) operations. FFT produces the same result but with \( O(n \log(n)) \) operations.

Convolution can be described with the following form:

\[ (f \ast g)[n] = \sum_{m=-\infty}^{\infty} f[m] g[n - m] \]

This requires multiple multiplications making this an ideal example for CUDA!

Experiment:

The hardware used in this example is the Nvidia GTS450. This GPU has 192 cores and 1GB or 128bit GDDR5 memory. This is a Fermi GPU.
The programming language used was “C”. In one program I was able to time a sequential “C” convolution.

**Convolution Results:**

The results of this example are pretty significant. In the table, N is the number of bytes that the signal to be convolved is comprised of.

<table>
<thead>
<tr>
<th>N</th>
<th>Sequential C (us)</th>
<th>CUDA (us)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>65</td>
<td>169</td>
</tr>
<tr>
<td>256</td>
<td>138</td>
<td>165</td>
</tr>
<tr>
<td>512</td>
<td>284</td>
<td>172</td>
</tr>
<tr>
<td>1024</td>
<td>584</td>
<td>308</td>
</tr>
<tr>
<td>2048</td>
<td>1196</td>
<td>370</td>
</tr>
<tr>
<td>4096</td>
<td>2409</td>
<td>387</td>
</tr>
<tr>
<td>8192</td>
<td>4812</td>
<td>407</td>
</tr>
<tr>
<td>16384</td>
<td>9464</td>
<td>743</td>
</tr>
<tr>
<td>32768</td>
<td>19221</td>
<td>734</td>
</tr>
</tbody>
</table>

It is easy to see that for large signals, the CUDA implementation is much faster. The sequential C implementation grows linearly with the sample size and the CUDA implementation is roughly a step function that doubles the time it takes about every time there are 192 times the data to process. This shows that the parallelism of the CUDA enabled GPU is working. For very small sample sizes however, the sequential version is better because it doesn’t require the long delay between memory reads/writes to the GPU.

**Conclusion:**

In summary, CUDA allows for massive parallelism of computations allowing for much faster processing time for large number of operations. For processes such as matrix multiplication or in this case convolution it is perfect. CUDA is simple to implement in parallel with sequential programming. For small sets of instructions use the individually faster CPU and when you need to do lots of computation for large data sets, offload the process to the GPU. CUDA comes with an easy to use and decently documented toolkit for both Windows and Linux and with Nvidia GPU’s becoming more prevalent in most systems, there is going to be even more applications where CUDA will greatly aid processing.
Code:

```c
/* Example showing the use of CUFFT for fast 1D-convolution using FFT. */

#include <stdio.h>
#include <cufft.h>
#include <string.h>
#include <math.h>
#include <unistd.h>
#include <sys/time.h>
#include <cutil_inline.h>
#include <cu_fft.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

// Complex data type
typedef float2 Complex;
static __device__ __host__ inline Complex ComplexAdd(Complex, Complex);
static __device__ __host__ inline Complex ComplexScale(Complex, float);
static __device__ __host__ inline Complex ComplexMul(Complex, Complex);
static __global__ void ComplexPointwiseMulAndScale (Complex*, const Complex*, int, int);

// Filtering functions
void Convolve(const Complex*, int, const Complex*, int, Complex*);
long fft (Complex*, Complex*, int, int, Complex*);
void fft_test (int);
long fftSeq (Complex*, Complex*, int, int, Complex*);

// Padding functions
int PadData (const Complex*, Complex**, int, const Complex*, Complex**, int);

// declaration, forward
void runTest (int argc, char** argv);

// The filter size is assumed to be a number smaller than the signal size
#define SIGNAL_SIZE 65536
#define FILTER_SIZE 128

int main (int argc, char* argv, char* argv+1)
{
    printf("Size [bits] CPU CUFFTtook (us) | Sequential FFT Took (us)]: n^%d \n", n);
    runTest (argc, argv);
    // int i;
    // long size = i;R
    // Run with an i bit number
    // ran(i = 1; i < 36; i++)
    // size = size + 2;
    // printf("%ld R",size);
    // fft_test(size);
    // cuutilExit(argc, argv);
}

// Run a simple test for Convolve() function
void fft_test (int signal_size)
{
    Complex h_signal = (Complex*) malloc(sizeof(Complex) * signal_size);
    Complex o_signal = (Complex*) malloc(sizeof(Complex) * signal_size);
    long time1, time2;
    // initialize the memory for the signal
    for (unsigned int i = 0; i < signal_size; i++)
    {
        h_signal[i].x = rand() / (float)RAND_MAX;
        h_signal[i].y = rand() / (float)RAND_MAX;
    }
    time1 = fftSeq (h_signal, o_signal, 1, SIGNAL_SIZE);
    //printf("Sequential FFT Took: %ld us, time1: %ld \n", n, time1);
    time2 = fftSeq (h_signal, o_signal, 1, SIGNAL_SIZE);
    //printf("CUFFT Took: %ld us, time2: %ld \n", n, time2);
}

long fftSeq (Complex* input_signal, Complex* output_signal, int dir, int run_times)
{
    struct timeval start, end;
    long seconds, useconds, mtime;
    gettimeofday(&start, NULL); // Start Recording time
    Complex signal_buffer;
    Complex* working_signal = input_signal;
    Complex* working_signal = input_signal;
    int size = sizeof(input_signal) / m is the size
    n, m, l, k, i1, i2, 1, 0, 1, 2, w, x;
    double e1, e2, t1, t2, u1, u2, u3;
    long iteration;
    
    for (iteration = 0; iteration < run_times; iteration++)
    {
        // Create an intermediate Complex for bit inversion
        // Calculate the number of points */
```
n = 1;
for (i = 0; i < size; i++)
    n *= 2;

/* Do the bit reversal */
12 = n >> 1;
for (i = 0; i < 12; i++) {
    if (i < j) {
        signal_buffer.x = working_signal[i].x;
        signal_buffer.y = working_signal[i].y;
        working_signal[i].x = working_signal[j].x;
        working_signal[i].y = working_signal[j].y;
    }
}
k = 12;
while (k < j) {
    j = k;
    k <<= 1;
}

/* Compute the FFT */
c1 = -1.0;
c2 = 0.0;
d2 = 1.0;
for (i = 0; i < size; i++) {
    l1 = l2 << 1;
    l2 = l1 >> 1;
    t1 = t2 = 1.0;
    u1 = u2 = 0.0;
    for (j = 0; j < l1; j++) {
        working_signal[j].y = signal_buffer[j].y;
        working_signal[j].x = signal_buffer[j].x;
        working_signal[j].y += t1 * working_signal[j].y;
        working_signal[j].x += t1 * working_signal[j].x;
    }
}

c2 = sqrt((1.0 - c1) / 2.0);
if (dir == 1) {
    c2 = -c2;
}

/* Scaling for forward transform */
if (dir == 1) {
    for (i = 0; i < size; i++) {
        working_signal[i].x /= n;
        working_signal[i].y /= n;
    }
}

gettimeofday(&endTime, NULL);
// Print how long the sequential version took
seconds = end.tv_sec - start.tv_sec;
useconds = end.tv_usec - start.tv_usec;
mt ime = (seconds * 1000000 + useconds) + 0.5;
return mt ime;

int f t t (Complex* input_signal, Complex* output_signal, int run_times) {
    struct timeval Start, end;
    long seconds, useconds, mt ime;
    cudaMemcpy((void**)&device_input_signal, size, cudaMemcpyHostToDevice);
    CUFFT plan;
    cu fftHandle plan;
    //Do the fft run_times number of times
    for (iteration = 0; iteration < run.times; iteration++) {
        cu fftSafeCall(cu fftPlan domain, plan, size, CUFFT_FORWARD);
    }
    cudaMemcpy(output_signal, device_input_signal, size, cudaMemcpyDeviceToHost);
    // Send Back timing data
    gettimeofday(&endTime, NULL);
    seconds = end.tv_sec - start.tv_sec;
    useconds = end.tv_usec - start.tv_usec;
    mt ime = (seconds * 1000000 + useconds) + 0.5;
    cudaFree(device_input_signal);
    cu fftDestroy(plan);
`return mtimetime;`  

```c
void runTest(int argc, char** argv) {
    struct timeval start, end;
    long seconds, useconds, mtimetime;
    if (cutCheckCmdLinePing(argc, (const char**) argv, "device") ||
        cutDeviceInit(argc, argv))
        return;
    else
cu
    
    cudaSetDevice(cuGetMaxGflopsDeviceId());
    // Allocate host memory for the signal
    Complex* h_signal = (Complex*) malloc(sizeof(Complex) * SIGNAL_SIZE);
    // Initialize the memory for the signal
    for (unsigned int i = 0; i < SIGNAL_SIZE; i++)
        h_signal[i].x = rand() / (float) RAND_MAX;
    h_signal[i].y = 0;
    // Allocate host memory for the filter
    Complex* h_filter_kernel = (Complex*) malloc(sizeof(Complex) * FILTER_KERNEL_SIZE);
    // Initialize the memory for the filter
    for (unsigned int i = 0; i < FILTER_KERNEL_SIZE; i++)
        h_filter_kernel[i].x = rand() / (float) RAND_MAX;
    h_filter_kernel[i].y = 0;
    // Pad signal and filter kernel
    Complex h_padded_signal;
    Complex h_padded_filter_kernel;
    int new_size = Padded(h_signal, SIGNAL_SIZE, h_padded_signal, SIGNAL_SIZE, h_padded_filter_kernel, FILTER_KERNEL_SIZE);
    int mem_size = sizeof(Complex) * new_size;
    // Allocate device memory for signal
    cututilMallocCUDA((void**)&d_signal, mem_size);
    // Copy host memory to device
    cututilMallocCUDA((void**)&d_signal, mem_size);
    cuMallocHostToDevice;
    // Allocate device memory for filter kernel
    Complex d_filter_kernel;
    cututilMallocCUDA((void**)&d_signal, mem_size);
    // Copy host memory to device
    cututilMallocCUDA((void**)&d_signal, mem_size);
    cuMallocHostToDevice;
    // CUFFT plan
    cuFFT_Handle plan;
    cuFFTMalloc1D(plan, mem_size, CUFFT_C2C, 1);
    gettimeofday(&start, NULL);
    // Transform signal and kernel
    cuFFT_SUCCESS cuFFTPlan1D(plan, (cuFFTComplex*) d_signal, cuFFT_Success cuFFT_C2C, cuFFT_FORWARD);
    cuFFT_SUCCESS cuFFTPlan1D(plan, (cuFFTComplex*) d_signal, cuFFT_Success cuFFT_FORWARD);
    // Multiply the coefficients together and normalize the result
    Complex PointwiseMulAndScale((Complex) d_signal, d_filter_kernel, new_size, 1.0f / new_size);
    // Check if kernel execution generated and error
    cututilCheckMsg("Kernel execution failed", Complex PointwiseMulAndScale);
    // Transform signal back
    cuFFT_SUCCESS cuFFTPlan1D(plan, (cuFFTComplex*) d_signal, cuFFT_Success cuFFT_INVERSE);
    // Copy device memory to host
    Complex* h_conved_signal = h_padded_signal;
    cututilMallocCUDA((void**)&d_signal, mem_size, d_signal, mem_size, cuMallocHostToDevice);
    gettimeofday(&start, NULL);
    seconds = end.tv_sec - start.tv_sec;
    useconds = end.tv_usec - start.tv_usec;
    mtimetime = (seconds) * 1000000 + useconds + 0.5;
    printf("CUDA version took %d ms \n", mtimetime);
    // Allocate host memory for the convolution result
    Complex* h_conved_signal_ref = (Complex*) malloc(sizeof(Complex) * SIGNAL_SIZE);
    // Convolve on the host
    Convolve(h_signal, SIGNAL_SIZE, h_filter_kernel, FILTER_KERNEL_SIZE, h_conved_signal_ref);
    gettimeofday(&start, NULL);
    seconds = end.tv_sec - start.tv_sec;
    useconds = end.tv_usec - start.tv_usec;
    mtimetime = (seconds) * 1000000 + useconds + 0.5;
    printf("Convolution took %d ms \n", mtimetime);
    // check results
    CUTFBoolean res = cutCheck(sample_t, h_conved_signal, (float*) h_conved_signal_ref, (float*) h_conved_signal, (float*) h_conved_signal_ref, 1e-5f);
    printf("%s\n", (res == res) ? "PASSED" : "FAILED");
    printf("%s\n", (res == res) ? "PASSED" : "FAILED");
}
```
// Pad data
cost Complex signal, Complex* padded_signal, int signal_size,
const Complex* filter_kernel, int filter_kernel_size;
{
    int minRadius = filter_kernel_size / 2;
    int maxRadius = filter_kernel_size - minRadius;
    int new_size = signal_size + MaxRadius;
    // Pad signal
    Complex* new_data = (Complex*)malloc(sizeof(Complex) * new_size);
    memset(new_data + signal_size, 0, (new_size - signal_size) * sizeof(Complex));
    *padded_signal = new_data;
    // Pad filter
    new_data = (Complex*)malloc(sizeof(Complex) * new_size);
    memset(new_data + new_size - filter_kernel_size, 0, filter_kernel_size * sizeof(Complex));
    *padded_filter_kernel = new_data;
    return new_size;
}

// Compute convolution on the host
void Convolve(const Complex signal, int signal_size,
const Complex* filter_kernel, int filter_kernel_size,
const Complex* padded_signal, int filter_kernel_size,
Complex* filter_kernel, Complex* filtered_signal)
{
    int minRadius = filter_kernel_size / 2;
    int maxRadius = filter_kernel_size - minRadius;
    // Loop over output elements
    for (int i = 0; i < signal_size; ++i) {
        filtered_signal[i].x = filtered_signal[i].y = 0;
        // Loop over convolution indices
        for (int j = -maxRadius + 1; j <= minRadius + 1; j += 1)
        {
            int k = j + i;
            if (k >= 0 && k < signal_size)
            {
                filtered_signal[i] = ComplexAdd(filtered_signal[i], ComplexMul(signal[k], filter_kernel[minRadius - j]));
            }
        }
    }
}

// Complex addition
static __device__ __host__ inline Complex ComplexAdd(Complex a, Complex b)
{
    Complex c;
    c.x = a.x + b.x;
    c.y = a.y + b.y;
    return c;
}

// Complex scale
static __device__ __host__ inline Complex ComplexScale(Complex a, float s)
{
    Complex c;
    c.x = a.x * s;
    c.y = a.y * s;
    return c;
}

// Complex multiplication
static __device__ __host__ inline Complex ComplexMul(Complex a, Complex b)
{
    Complex c;
    c.x = a.x * b.x - a.y * b.y;
    c.y = a.x * b.y + a.y * b.x;
    return c;
}

// Complex pointwise multiplication
static __global__ void ComplexPointwiseMulAndScale(Complex* a, const Complex* b, int size, float scale)
{
    const int numThreads = blockDim.x * gridDim.x;
    const int threadID = blockIdx.x * blockDim.x + threadIdx.x;
    for (int i = threadID; i < size; i += numThreads)
    {
        a[i] = ComplexScale(ComplexMul(a[i], b[i]), scale);
    }
}