Internship
at the Idaho State University
in co-operation with the University of Applied Sciences Jena

Topic:

Development of a DFT algorithm
using CUDA

Student         Sebastian Schwabe (926443)
Advising Prof.  Steve C. Chiu, Ph.D.
Due Date        October/17/2011
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>I</td>
</tr>
<tr>
<td>List of Figures</td>
<td>II</td>
</tr>
<tr>
<td>List of Tables</td>
<td>III</td>
</tr>
<tr>
<td>1 Preliminary</td>
<td>1</td>
</tr>
<tr>
<td>2 DFT - Algorithm</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Implementation for a normal CPU (serial code) and for OpenMP (parallel code)</td>
<td>5</td>
</tr>
<tr>
<td>2.1.1 Porting CPU code to OpenMP code</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Implementation for CUDA (parallel code)</td>
<td>7</td>
</tr>
<tr>
<td>3 Proving the Algorithm</td>
<td>14</td>
</tr>
<tr>
<td>4 Speed Up</td>
<td>20</td>
</tr>
<tr>
<td>5 Conclusion</td>
<td>22</td>
</tr>
<tr>
<td>A FFT on CPU</td>
<td>VI</td>
</tr>
<tr>
<td>B FFT on GPU</td>
<td>VII</td>
</tr>
</tbody>
</table>
List of Figures

2.1 CPU Single-Core Code ........................................... 7
2.2 CPU Multi-Core Code ........................................... 7
2.3 Porting a function to a CUDA kernel ...................... 8
2.4 Memory Allocation and Deallocation ...................... 9
2.5 Summation Reduction of a Vector [1, P. 80] ............ 10
2.6 Theoretical synchronization between Blocks and Threads . 11
2.7 Port of the function calcAbs(...) .......................... 13

3.1 The separated signals .......................................... 15
3.2 Signal used to prove the algorithm ......................... 16
3.3 Reference Spectrum created with MATLAB ............... 16
3.4 MATLAB code to generate the signal values and the refer-
   ence spectrum .................................................. 17
3.5 MATLAB code to read the values calculated by the DFT
   algorithm ....................................................... 18
3.6 DFT result calculated on CPU ............................... 18
3.7 DFT result calculated on CPU using OpenMP ............ 19
3.8 DFT result calculated on TESLA C2050 with CUDA ... 19
List of Tables

2.1 DFT values ....................................................... 3

4.1 Time Measurement for 100 frequency values ............. 21
Chapter 1

Preliminary

At the Department of Electrical Engineering and Computer Science of Idaho State University a new research group with the topic Massive Parallel Computing was established. The group around Prof. Steve C. Chiu investigates the computation of physics, nuclear and other algorithms with the aim of speeding up the computations. There are different ways to achieve a Speed Up. First one, is the usage of more powerful server architectures which parallelize the computation tasks. The problem in using server architectures is the limited amount of money in the sector of researches. That is why a second possibility came up in the last years. The usage of graphic cards for parallel computing, now also investigated by the research group at the Idaho State University. The power of the graphic cards is based on their multi-core Floating-Point-Unit (FPU). Graphic cards from the manufacturer nVIDIA with the FERMI architecture (see [2]) uses FPUs with a huge amount of CUDA cores depending on the type of the graphic card. This graphic cards offer a future-oriented possibility in Massive Parallel Computing. They are a lot cheaper and more powerful than expanding or setting up a server architecture.

Using the graphic cards from nVIDIA for Massive Parallel Computing utilizes the CUDA API. If you wish to take a closer look to the nVIDIA graphic cards architecture and the CUDA API the documents [4] and [1]
are useful.

The idea of this document was to compare the Speed Up of an algorithm calculated on a single CPU, a multi-core system using OpenMP and on the nVIDIA graphic card TESLA C2050. Using OpenMP was interesting because it is the cheapest way of speeding up the computation of an algorithm on a multi-core machine. To use OpenMP the organization with the same name offers a programmer an API to use all cores of a CPU (see [5]). For the researches around the group at the Idaho State University the TESLA C2050 was used and is specified in the document [3].

Getting a good comparison of the speed up presupposes an easy to understand algorithm but also complex enough in computation. Therefore, the algorithm of a Discrete Fourier Transformation (DFT) was implemented in C. The programming language C was chosen because the CUDA API is also written in C code. In the signal theory the DFT has an important role to analyze the spectrum of a signal. The DFT is easy enough to understand and to implement but needs a lot of steps to be calculated. The written algorithm was executed on the three different hardware platforms: CPU, a multi-core system using OpenMP and the TESLA C2050. At the end their execution times were compared to get the Speed Up with each architecture in comparison to the execution on a single CPU core.
Chapter 2

DFT - Algorithm

To get a good comparison in Speed Up the classical DFT given by the formula [2.1] was implemented in C code.

\[
X(m) = \sum_{k=0}^{L-1} x(k) \cdot e^{-j2\pi m \frac{k}{L}} \tag{2.1}
\]

\[
X(m) = \sum_{k=0}^{L-1} x(k) \cdot \left[ \cos(2\pi m \frac{k}{L}) - j \cdot \sin(2\pi m \frac{k}{L}) \right] \tag{2.2}
\]

\[
m = 0 \ldots L
\]

\[
m = \frac{f}{f_P} \cdot L
\]

\[
k = 0 \ldots L
\]

<table>
<thead>
<tr>
<th>x(k)</th>
<th>discrete time domain signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>X(m)</td>
<td>discrete frequency domain signal</td>
</tr>
<tr>
<td>L</td>
<td>length of the discrete time domain signal</td>
</tr>
<tr>
<td>f</td>
<td>frequency for a specific m</td>
</tr>
<tr>
<td>f_P</td>
<td>signal sample frequency</td>
</tr>
<tr>
<td>k, m</td>
<td>indexes for the computing of the DFT</td>
</tr>
</tbody>
</table>

Table 2.1: DFT values
To achieve an easy port from serial code, executed on a single CPU, to parallel code, executed on a multi-core system, the DFT algorithm was designed for parallel execution. Therefore, the job given by the equation 2.2 was split into four jobs and is computed by four functions in C code. The four jobs are described in the next enumeration:

1. Calculation of the product

\[ 2\pi m \frac{k}{L} \]

and storing the values in an array named MK (with dimension L)

2. Calculation of the real part of the DFT using the results from 1. and storing in an array (with dimension L)

\[ REAL = x(k) \cdot \cos(MK) \]

3. Calculation of the imaginary part of the DFT using the results from 1. and storing in an array (with dimension L)

\[ IMAG = -x(k) \cdot \sin(MK) \]

4. Calculation of the frequency domain value for one specific m using the results from 1., 2. and 3.

\[ X(m) = \sqrt{\sum_{k=0}^{L-1} REAL^2 + \sum_{k=0}^{L-1} IMAG^2} \]

The frequency for one specific m can be calculated by the formula

\[ f = \frac{m \cdot f_P}{L} \]

The functions are used to compute the frequency value for one specific
m. These functions are called in a loop to calculate all the frequency values. At the Nyquist frequency the DFT is mirrored, that is why the job given by the equation 2.2 is repeated until this frequency. The Nyquist frequency is given by the formula 2.3.

\[ f = \frac{f_P}{2} \]  

(2.3)

2.1 Implementation for a normal CPU (serial code) and for OpenMP (parallel code)

The DFT algorithm was designed to run on a single CPU and a multi-core CPU using OpenMP. This was done by simply adding a command line parameter which indicates which hardware architecture should be used. The code section A shows the implementation of the DFT algorithm. In the main function it is proved if enough RAM is available, the signal values from a given file are read and the DFT is calculated like described above and the amplitude of each frequency value is stored back to a file. The application uses four arrays in the floating point format to calculate the DFT. Each floating point value needs a size of 4 bytes in memory. The application was limited to use a maximum of 4GB RAM. That means, each array can contain a maximum of 250,000 elements, thus the signal size is limited to 250,000 sample points. The signal values are stored in a file which must be in the format of the following manner:

- the signal values are in the floating point format
- one line contains one signal value
- a line ends on one newline character (ASCII Code 0x0a)

After one amplitude of a frequency value was calculated it is written back
into a file. The section defines how to interpret this file correctly.

### 2.1.1 Porting CPU code to OpenMP code

To advise the C compiler that some code should be executed on multiple CPUs using OpenMP the \texttt{#pragma} directives are used. An exactly documentation how to use OpenMP can be found in [5]. The illustrations 2.1 and 2.2 shows how to port some code to OpenMP. Illustration 2.1 shows the C code which will be interpreted by the compiler to run on a single CPU. The loop can be parallelized and calculated on multiple CPUs using OpenMP as shown in illustration 2.2. Therefore, the OpenMP directive \texttt{#pragma omp parallel for if(openMPenabled) reduction(+: real, imag)} was add. The directive \texttt{#pragma omp parallel for} defines that the loop coming up should be executed in parallel. With the help of the parameter \texttt{openMPenabled} the compiler can be controlled. Means, if this parameter is set on command line the code is compiled using OpenMP and without the parameter it is compiled for single CPUs. The last clause of the directive: \texttt{reduction(+: real, imag) "... specifies an operator and one or more list items. For each list item, a private copy is created in each implicit task, and is initialized appropriately for the operator. After the end of the region, the original list item is updated with the values of the private copies using the specified operator". ( [5, P. 103]).}
2.2 Implementation for CUDA (parallel code)

For porting the CPU code to CUDA code a lot more things must be done than porting the code to OpenMP code. Therefore, a new file listed in B was created. The main function of this file does the same like the one from the CPU code. The difference is that the functions, called by the main function, for calculating the DFT are now CUDA kernels. This is the first port, the functions and their calls must be migrated. Illustration 2.3 shows the port from a normal function to a CUDA kernel.
The next difference to the CPU code is the memory handling. Now, the application uses the memory of the graphic card to store the four arrays needed by the DFT. This is understandable because all signal operations are calculated on the graphic card. Using the memory of the graphic card saves a lot of IO operations. This important point brings up one challenge: the input parameters, especially the pointers to the data, of the origin functions must be changed to point on positions in the memory of the graphic card. Also the allocation and deallocation of the memory must be changed (shown in illustration 2.4).

The most difficult part of porting the code to CUDA was the redesign of the function `calcAbs(...)`. This function is used to calculate the absolute value for one frequency of the DFT. Problem is, that the summation reduction of one vector must be calculated. Usually this is done by:

```
for (k=0; k<L; k++)
    VectorReduction += Vector[k]
```

With the help of CUDA this could be done like written in the source code:

```c
__global__ void calcImag( const long uLengthOfDFT, const float* pDev_mkArr,
                          const float* pDev_imagArr, const float* pDev_signalVals )
{
    int nTid = threadIdx.x + blockDim.x * blockIdx.x;
    if( nTid < uLengthOfDFT )
        pDev_imagArr[nTid] = -__sin((float)pDev_mkArr[nTid]) * pDev_signalVals[nTid];
}
```
Figure 2.4: Memory Allocation and Deallocation

Sourcecode 2.2: Summation Reduction using CUDA

```c
long nTid = threadIdx.x + blockDim.x * blockIdx.x;
long nLength = uLengthOfVector; // length of the vector for calculating the reduction
if( nTid < nLength/2 )
{
    pDev_Vector[nTid] += pDev_Vector[nTid+nLength/2];
    nLength /= 2;
}
```

The illustration 2.5 shows what the code from source code section 2.2 is doing. This brings up the problem of synchronizing the summation reduction between blocks and threads. Imagine the following simple example:

- a vector has 16 elements
- notional we limit the amount of threads per block offered by the graphic card to four (usually the graphic card can handle 1024
Figure 2.5: Summation Reduction of a Vector [1, P. 80]

threads per block)

- the kernel doing the synchronization is called once and will calculate
  the reduction as shown in illustration 2.6

As seen, two kinds of synchronizations must be handled by the graphic

- synchronization between blocks

- synchronization between threads

Without the synchronization one block or thread could be finished before

another one. Then the graphic card will calculate the wrong summation

reduction result of the vector. The following code of a CUDA kernel
could do the job as described above:

Sourcecode 2.3: Summation Reduction using thread synchronization

```c
long nTid = threadIdx.x + blockDim.x * blockIdx.x;
long nLength = uLengthOfVector; //length of the
  //vector for calculating the reduction

while ( nLength != 0 )
{
    if( nTid < nLength/2 )
        pDev_Vector[nTid] += pDev_Vector[nTid+nLength/2];

    __syncthreads();
    nLength /= 2;
}
```

At this point the calculation using CUDA runs into trouble, because

CUDA does not offer a synchronization between blocks. This is more
Figure 2.6: Theoretical synchronization between Blocks and Threads

than understandable because for a synchronization object an additional shared memory section is needed. The nVIDIA graphic cards, using the FERMI architecture, only offer shared memory for threads. That is the reason why a synchronization between threads but not between blocks is possible. So, what is the solution for synchronization of blocks? The answer is quite simple: This must be done by the Host in calling the kernel multiple times! All this lead to the port of the function as shown in illustration 2.7. Before, the code called the `calcAbs(...) function di-
rectly. Now, the kernel is called by a helper function with the name
\textit{abs\_device(...)} which reduces the code changes in calling the origin func-
tion from the main. Right now this code is not perfect, but it does a
synchronization between blocks. This is done by calling the kernel again
after each reduction. But the code also calls the kernel multiple times
if only one block is needed. This brings a lot of overhead in function
calls which also includes temporary saving of registers and loading the
values back. To solve this problem, a second kernel must be written with
the code described in code listing \textbf{2.3}. This new kernel should be called
when the remainder of vector elements, after a summation reduction, has
reached the number of threads per block.
float calcAbs( const long uLength0fDST, const float* realArr, const float* imgArr )
{
    long k;
    long L = uLength0fDST;
    float abs = 0.0f;
    float real = 0.0f;
    float imag = 0.0f;

    for(k=0; k<L; k++)
    {
        real += realArr[k];
        imag += imgArr[k];
    }

    abs = (float)sqrt((float)((real*real) + (imag*imag)));

    return abs;
}

void calcAbs( const long uLength0fArray, float* pDev_realArr, float* pDev_imgArr )
{
    //calculate the threadId
    long nTid = threadIdx.x * blockDim.x * blockIdx.x;

    if( nTid < uLength0fArray )
    {
        pDev_realArr[nTid] += pDev_realArr[nTid+uLength0fArray];
        pDev_imgArr[nTid] += pDev_imgArr[nTid+uLength0fArray];
    }
}

float abs_device( const long uLength0fArray, float** pDev_realArr, float** pDev_imgArr,
                   const int nBlocks, const int nThreads )
{
    long i=uLength0fArray/L;
    float real, imag, abs;  
    real = imag = abs = 0;

    //calculate the real and imag part of the arrays
    while( i > 0 )
    {
        calcAbs<<<nBlocks, nThreads>>>( i, pDev_realArr, pDev_imgArr );
        i /= 2;
    }

    //get the real and imag value
    HANDLE_CUDA_ERROR( cudaMemcpy( &real, *pDev_realArr, sizeof(float), cudaMemcpyDeviceToHost ) );
    HANDLE_CUDA_ERROR( cudaMemcpy( &imag, *pDev_imgArr, sizeof(float), cudaMemcpyDeviceToHost ) );

    abs = sqrt((float)((real*real) + (imag*imag)));

    return abs;
}

Figure 2.7: Port of the function calcAbs(...)
Chapter 3

Proving the Algorithm

MATLAB was used to prove the implemented algorithm. With the help of this program the signal

\[
\text{signal} = 0.7 \cdot \sin(2\pi \cdot 50Hz \cdot t) + \sin(2\pi \cdot 120Hz \cdot t)
\]

was created to prove the algorithm. This signal was sampled with a frequency of 1kHz. So, the sine with \(f=50\)Hz consists of 20 sampling points and the sine with \(f=120\)Hz consists of 8 sampling points. To get a reference for comparing the results of the DFT algorithm the created signal was transformed into the frequency spectrum with the help of MATLAB. The transformed signal is shown in illustration 3.3. The MATLAB source code shown in illustration 3.4 was used to create the signal, write the signal values into a file (which can be read by the DFT algorithm) and to transform the signal into the frequency spectrum. After creation of the reference signal and spectrum the DFT was called with the signal created by MATLAB. The DFT writes the result into a file which then was read by MATLAB to compare the results with the reference spectrum. To read the values back and plot them the MATLAB code shown in illustration 3.5 was used. The spectrum calculated by the DFT algorithm is the same like the one from MATLAB. The only difference is, that the
Figure 3.1: The separated signals

written algorithm is less accurate than the FFT from MATLAB.
Figure 3.2: Signal used to prove the algorithm

Figure 3.3: Reference Spectrum created with MATLAB
Fs = 1000;
T = 1/Fs;
L = 2^10;
t = (0:L-1)*T;
x = 0.7*sin(2*pi*50*t) + sin(2*pi*120*t);

% write the signal to a file
fid = fopen( 'signal.dat', 'wt' );
fprintf( fid, '%f\n', x );
fclose(fid);

X = fft(x, L);
f = Fs/2*linspace(0,1,L/2+1);
absX = abs(X);
fftVal = 20 * log10(absX(1:L/2+1));
plot(f, fftVal);
xlabel('Frequency (Hz)');
ylabel('|Y(f)|');

% write the result of the fft in a file
fid = fopen( 'result.dat', 'wt' );
fprintf( fid, '%f\n', fftVal );
fclose(fid);

Figure 3.4: MATLAB code to generate the signal values and the reference spectrum
\[ L = 1024; \]
\[ Fs = 1000; \]

\[ \text{fid} = \text{fopen( 'result\_sinus\_sum\_cuda.dat', 'r' );} \]
\[ \text{fftVals} = \text{fscanf( fid, '%f\n' );} \]
\[ \text{fclose(fid);} \]

\[ f = Fs/2*\text{linspace(0,1,L/2+1);} \]
\[ \text{plot(f, fftVals);} \]
\[ \text{xlabel('Frequency (Hz)');} \]
\[ \text{ylabel('|Y(f)|');} \]

Figure 3.5: MATLAB code to read the values calculated by the DFT algorithm

Figure 3.6: DFT result calculated on CPU
Figure 3.7: DFT result calculated on CPU using OpenMP

Figure 3.8: DFT result calculated on TESLA C2050 with CUDA
Chapter 4

Speed Up

To calculate the Speed Up of the algorithm using OpenMP and CUDA a long signal was created with MATLAB. This was the same signal described in section 3 but with $2^{23}$ values which is also the length of the DFT. The Calculation of the DFT for this kind of a long signal will take a long time and is therefore good for measuring the Speed Up. After 100 calculated frequency values the time the algorithm needed on CPU, with OpenMP and CUDA was measured. This time indicates how long it would take to calculate the spectrum of the whole DFT. A signal with $2^{23}$ values would create an output of $m=2^{22}$ frequency values. When $x=100$ it would take a time of

$$T = \frac{2^{22}}{100} \cdot t_{\text{calculation for } x}$$

to calculate the spectrum of the whole DFT. The following table shows the calculation times in seconds for 100 frequency values on CPU, using OpenMP and CUDA (5 measurements were accomplished).
<table>
<thead>
<tr>
<th>CPU without OpenMP</th>
<th>CPU using OpenMP</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>86.706s</td>
<td>11.935s</td>
<td>1.432s</td>
</tr>
<tr>
<td>86.961s</td>
<td>11.821s</td>
<td>1.568s</td>
</tr>
<tr>
<td>86.990s</td>
<td>11.748s</td>
<td>1.402s</td>
</tr>
<tr>
<td>87.017s</td>
<td>11.945s</td>
<td>1.378s</td>
</tr>
<tr>
<td>87.030s</td>
<td>11.943s</td>
<td>1.315s</td>
</tr>
</tbody>
</table>

Table 4.1: Time Measurement for 100 frequency values

This results in the following operating run-times for the hole DFT and the Speed Up.

Using CPU:

- the CPU used was a 8 core Intel(R) Xeon(R) CPU on 3.16 GHz
- estimated operating time: \( T = \frac{2^{22}}{100} \cdot \frac{87}{3600} = 1013.62h \)

Using CPU with OpenMP:

- the CPU used was a 8 core Intel(R) Xeon(R) CPU on 3.16 GHz
- estimated operating time: \( T = \frac{2^{22}}{100} \cdot \frac{12}{3600} = 139.81h \)
- Speed Up: 7.25

Using CUDA:

- the graphic card TESLA C2050 was used
- estimated operating time: \( T = \frac{2^{22}}{100} \cdot \frac{1.4}{3600} = 16.31h \)
- Speed Up: 62.15
Chapter 5

Conclusion

The document highlighted the enormous potential in Massive Parallel Computing using the CUDA environment. With the help of a self implemented algorithm it was shown how a migration from serial code to parallel code can be arranged and how much the Speed Up can be. The calculated Speed Up is not an indicator for each application because this depends on how much code can be parallelized in an application. Nevertheless, this document shows the advantage using parallel code in developing algorithms. Especially if the algorithm must compute large vectors or matrices then the usage of CUDA is more than reasonable. Migrating an existing application to CUDA could be a big problem, particularly if the code was not designed for parallel execution. But at least developers should think about the potential of CUDA and how much time could be saved in executing large computations.
Bibliography


Appendix A

FFT on CPU

See file dft.c
Appendix B

FFT on GPU

See file dft.cu