GPU Accelerated Fast Fourier Transform

Matt McCarthy

Christopher Newport University
matthew.mccarthy.12@cnu.edu

December 2015

Abstract We empirically investigate the performance benefits of parallel fast Fourier transform running on the GPU over a sequential version running on the GPU.

1 Background

1.1 Discrete Fourier Transform

The discrete Fourier transform is a mathematical transformation that takes a set of Complex-valued signals and outputs a set of Complex-valued frequencies. For an n-dimensional Complex-valued vector $\mathbf{X}$, the discrete Fourier transform $\mathbf{Y} = \mathcal{F}(X)$ is given by

$$Y_j = \sum_{k=0}^{n} x_k \omega^{jk}$$

where ω is the n-th root of unity, $e^{2\pi i/n}$. Since $\mathbf{Y}$ is an n-dimensional, Complex-valued vector, we can see that the discrete Fourier transform has a complexity of $\Theta(n^2)$.

1.2 Fast Fourier Transform

Furthermore, we can split the discrete Fourier transform into even and odd sums for $n = 2m$, yielding

$$Y_j = \sum_{k=0}^{m} x_{2k} \omega^{2jk} + \omega^j \sum_{k=0}^{m} x_{2k+1} \omega^{2jk}$$

which is two seperate discrete Fourier transforms. Suppose $n = 2^k$. If we iterate this process, we get the following algorithm called the one-dimensional, unordered radix 2, fast Fourier transform.

```
1: function R-FFT(X,Y,n,ω)
2:     if n=1 then
3:         y_0 = x_0
4:     else
5:         Let Q = 0, T = 0 ∈ C^n
6:         Let X_e = (x_0, x_2, ..., x_{n-2})
7:         Let X_o = (x_1, x_3, ..., x_{n-1})
8:         R-FFT(X_e,Q_e,n/2,ω^2)
9:         R-FFT(X_o,T_o,n/2,ω^2)
10:        for all j ∈ {0, 1, ..., n − 1} do
11:            y_j = q_{j mod n/2} + ω^i t_{j mod n/2}
12:        end for
13:    end if
14: end function
```

1.2.1 Cooley Tukey

Furthermore, we have an iterative formulation of the prior algorithm, called the Cooley Tukey algorithm for one-dimensional, unordered radix 2, fast Fourier transforms.

```
1: function I-FFT(X,Y,n)
2:     t := lg n
3:     R = X
4:     for m = 0 to t − 1 do
5:         S = R
6:         for l = 0 to n − 1 do
7:             Let (b_0 b_1 ... b_{t−1}) be the binary expansion of l
8:             j := (b_0 ... b_{m−1} 0 b_{m+1} ... b_{t−1})
9:             k := (b_0 ... b_{m−1} 1 b_{m+1} ... b_{t−1})
10:            r_i := s_j + s_k ω^{(b_m b_{m−1} ... b_0 0 ... 0)}
11:        end for
12:    end for
13:    Y := R
14: end function
```

From this pseudo-code, we can determine the complexity of fast Fourier transform. We begin by noting that we iterate through the outer loop precisely $\lg n$ times and the inner loop n times. Therefore our com-

plexity is $T_1 = \Theta(n \lg n)$.

1.3 Parallelization

For our parallelization, we use a simplified version of the binary exchange algorithm, a parallelization of the Cooley Tukey algorithm designed for use on a hypercube. Since our implementation runs on a single GPU, any thread can access any memory location via a pointer. However, this also complicates the matter by introducing a potential for data races. We solve this by modifying the algorithm to work as follows.

1: **function** PAR-FFT($\mathbf{X}$,$\mathbf{Y}$,n)
2: $t := \lg n$, $BLK := n/p$
3: $\mathbf{R} = \mathbf{X}$
4: $\mathbf{S} = \mathbf{0}$
5: **for** $m = 0$ to $t - 1$ **do**
6: Swap pointers $\mathbf{R}$ and $\mathbf{S}$
7: **spawn process for** $l = 0$ to $BLK - 1$ **do**
8: **for** $c = l \cdot BLK$, to $l \cdot (BLK + 1)$ **do**
9: Let $(b_0 b_1 \dots b_{t-1})$ be the binary expansion of c
10: $j := (b_0 \dots b_{m-1} 0 b_{m+1} \dots b_{t-1})$
11: $k := (b_0 \dots b_{m-1} 1 b_{m+1} \dots b_{t-1})$
12: $r_i := s_j + s_k \omega^{(b_m b_{m-1} \dots b_0 0 \dots 0)}$
13: **end for**
14: **end spawn**
15: **sync**
16: **end for**
17: $\mathbf{Y} := \mathbf{R}$
18: **end function**

In each iteration, we only write to $\mathbf{R}$ and only read from $\mathbf{S}$. Since we wait until each thread is complete before moving on to the next iteration, we avoid the potential to use old or incorrect data.

If we inspect the parallelization, we see that the outer loop runs $\lg n$ times. However, the inner loop is ran on p processes, each of which handle n/p iterations. Therefore, the computation cost is $\Theta(n/p \lg n)$. Moreover, for communication we simply do two reads from VRAM on each iteration of the inner loop, however some values will be in cache so we may get better performance than that. Ergo, the communication cost is $O(n/p \lg n)$. Thus, the parallel runtime will be $T_p = O(n/p \lg n)$ and is cost optimal if and only if $p \leq n$.

2 Experimental Design

The goal of the experiment is to empirically determine the effect of parallelization on the runtime of the Cooley Tukey algorithm. To this end we want to measure the runtime of fast Fourier transform using differing sizes of n, the dimension of our Complex valued vector, and t, the number of threads. Let $\mathcal{N}$ be the set of vector dimensions we will test and let $\mathcal{T}$ be the set of thread counts we will test. Furthermore, we specify that each $n \in \mathcal{N}$ and each $t \in \mathcal{T}$ be powers of two.

For $t = 1$, we will run the Cooley Tukey algorithm on random Complex valued vectors of dimension n for each $n \in \mathcal{N}$. Otherwise, we run our parallelization of the Cooley Tukey algorithm on Complex valued vectors of dimension n for each $n \in \mathcal{N}$. Moreover, before and after each run, we copy the input to the GPU and the output from the GPU. We do not check the output for correctness during the test in order to save time, but instead checked the correctness of the algorithm beforehand.

3 Test Environment

3.1 Test System

The machine used to run the test has an Intel i7 4770k running at 4.2GHz, 16GB of RAM, and a Nvidia GTX 970 with a core clock of 1342MHz and memory clock of 7000MHz. The computer was running Arch Linux on Linux kernel 4.2.5 with Nvidia driver version 358.16 and CUDA 7.5.

3.2 Test Program

Firstly, all code can be located at `https://github.com/matt-mccarthy/fast-fourier-transform` in the `fast-fourier` folder. We also include a version of the code in the Appendix.

All code was compiled with the CUDA toolkit 7.5 and GCC 5.2.0 with the command `nvcc -std=c++11 -rdc=true -arch=compute_52 -code=sm_52 -O3`. The library source file is `src/fast-fourier.cu`, and the sequential and parallel main functions are located in `sequential-fft.cu` and `parallel-fft.cu` respectively.

In both the sequential fast Fourier and parallel fast Fourier, we ran into issues with dynamically allocating memory in a CUDA kernel. Namely, if we allocated and freed often enough, we ran out of memory. To avoid this issue entirely, we made all allocations of giant arrays occur in main once and then reused those arrays. As a note, the `binary_stor` array is where we store the binary representation of our index l for

each thread. While this design is suboptimal (ideally, each thread would create its own so others can not touch it), it works and the performance impact should be neglible for large enough n.

For our experiment, we let $\mathcal{N} = \{2^{12}, 2^{13}, \ldots, 2^{18}\}$ and $\mathcal{T} = \{1, 4, 16, 64, 256, 1024\}$. Furthermore, we chose each $t \in \mathcal{T}$ such that they are all perfect squares. This allows us to tell the kernel to spawn $\sqrt{t}$ blocks with $\sqrt{t}$ threads each yielding a total of t threads. Lastly, we measured $T_t(n)$ 1000 times each and at the end of the program, found the average runtime and standard deviation.

4 Results

4.1 Linear Speedup

5 Conclusion

Appendix