OpenCL Fast Fourier Transform

Ruobing Li
New York University

December 13, 2012

1 Introduction

Fast Fourier Transform is one of the most important numerical algorithms in history. It has wide range of applications: audio signal processing, medical imaging, image processing, pattern recognition, computational chemistry, error correcting codes and spectral methods for PDE's. The goal of this project is to implement an OpenCL based FFT algorithm that has comparable performance with existing open source and vendor libraries.

1.1 Discrete Fourier Transform

To make this report more self-contained, first let's give a brief introduction of Discrete Fourier Transform (DFT). The following refers the Wikipedia[6].
The sequence of N complex numbers $x_0, ..., x_{n-1}$ is transformed into another sequence of N complex numbers according to the DFT formula:

$$\forall 0 \leq k \leq N-1 : \left(x_k = \sum_{n=0}^{N-1} (x_n e^{-(i2\pi \frac{k}{N} n)}) \right)$$

DFT can be expensive to compute directly using the above formula. For each k, we must execute: N complex multiplies and $N-1$ complex adds. The total cost of direct computation of an N-point DFT is N^2 complex multiplies and $N(N-1)$ complex adds. The overall runtime complexity is $O(N^2)$.

1.2 Fast Fourier Transform

The FFT provides us with a much more efficient way of computing the DFT. The FFT requires only $O(NlogN)$ computations to compute the N-point DFT[7].
The Cooley-Tukey algorithm is by far the most commonly used FFT algorithm. The idea is to build a DFT out of smaller and smaller DFTs by decomposing the input into smaller and smaller subsequences. It exploits the symmetries of the complex exponentials $W_N^{kn} = e^{-(i2\pi \frac{k}{N} n)}$. W_N^{kn} are called *twiddle factors*.
Assume $N = 2m$ (a power of 2). The N-point DFT can be written as:

$$\forall 0 \leq k \leq N-1 : \left(x_k = E(k) + W_N^k O(k) \right)$$

where $\frac{N}{2}$-point DFT of even samples is:

$$E(k) = \sum_{n=0}^{N/2-1} (x_{2k} W_{\frac{N}{2}}^{kr})$$

and $\frac{N}{2}$-point DFT of odd samples is:

$$O(k) = \sum_{n=0}^{N/2-1} (x_{2k+1} W_{\frac{N}{2}}^{kr})$$

The Cooley-Tukey algorithm is to divide the transform into two pieces of size $\frac{N}{2}$ at each step, and is therefore limited to power-of-two sizes, but any factorization can be used in general. These are called the radix-2 and mixed-radix cases. Although the basic idea is recursive, most traditional implementations rearrange the algorithm to avoid explicit recursion.

2 Related work

There are many existing papers discussing this problem[4][5][3][2]. Many high performance open source and vendor FFT libraries are widely used by research and industry. CUFFT is NVidia's implementation of an FFT solver on their CUDA architecture. CUFFT employs a radix-n algorithm, and operates by taking a user-defined plan as input which specifies the parameters of the transform. It then optimally splits the transform into more manageable sizes if necessary. These sub-FFT's are then farmed out to the individual blocks on the GPU itself which will handle the FFT computation.
FFTW[1] is a library of FFT routines which will provide optimized code for a given transform. FFTW was the interface from which CUDA was derived as it also creates a plan for a given transform and can then continually execute it. FFTW achieves its competitive performance by actually processing the transform initially when the plan is created. It checks through a series of optimized codes to see which one performs best for the given transform on the current architecture. These optimized routines are known as *codelets* and are chosen at runtime. The user can also create a series of codelets for FFTW to choose from. The best performance for FFT on any architecture necessitates some form of specialized codes for a given subset of problem sizes.
Apple FFT is an OpenCL based FFT library that uses similar planning techniques described above. It's implementation is based on this paper[5]. It generates optimized code for current platform and device (CPU or GPU) on the fly and achieved competitive performance.

3 Algorithm Design

3.1 Stockham's FFT algorithm

Our algorithm is based on Stockham's FFT algorithm described in this paper [3]. The original algorithm maps well on SIMD processors, but it does not take advantage of cache system, since its memory access pattern is not good. Therefore, we will do some optimization (will be described in the next section) on the original algorithm to improve memory access pattern.
The following illustrates the psudeo-code of radix-2 Stockham's FFT:

Data: input array in, output array out, size n
for $t = 1;\ t \leq logn;\ t = t + 1$ **do**
  for $j = 0;\ j \leq \frac{n}{2^t};\ j = j + 1$ **do**
    for $k = 0;\ k \leq 2^{t-1};\ k = k + 1$ **do**
      $theta = -\frac{2\pi k}{2^t}$
      $out[j2^t + k] = in[j2^{t-1} + k] + e^{i\ theta} in[j2^{t-1} + k + \frac{n}{2}]$
      $out[j2^t + k + 2^{t-1}] = in[j2^{t-1} + k] - e^{i\ theta} in[j2^{t-1} + k + \frac{n}{2}]$
    end
  end
  swap(in, out);
end

Algorithm 1: Stockham's radix-2 FFT algorithm

This pseudo-code illustrates the nested loop implementation of Stockham FFT algorithm. The FFT algorithm proceeds in $logn$ steps and during each step j, the output array is conceptually divided into data chunks of size $2j$. Similarly, the input chunks are conceptually divided into data chunks of size $2j-1$ and two input data chunks are mapped onto the appropriate output chunks. Therefore, the time complexity of this algorithm is $O(nlogn)$, and space complexity is $O(n)$.

3.2 Problem scale

We plan to implement this algorithm for both CPU and GPU. For CPU, our experimental platform has 24GB memory per node, so the maximum problem size is 2^{30} (16 bytes of memory per element for single precision). For GPU, the platform available to us (Tesla M2070) has 6GB of global memory, which limits our problem size to 2^{28}. Typically we have no minimum scale for CPU, since data transfer overhead is cheap. But for GPU, data transfer and command queue is the bottleneck when the problem size is small. We found that it's more economical to use CPU if the problem size is below 2^{18}.

4 Implementation

4.1 OpenCL kernel

We have implemented three different kinds of kernels: radix-2, 4 and 8. We will see an improving trend of performance as radix increases. Higher radix makes better use of private memory to processes several iterations per kernel, hence reduce the need of global communication. However, even higher radix does not necessarily improve performance, as they may exceed the size of private memory and have to use much slower local memory.
Below is the code snippet of a radix-2 kernel:

```
#define TWOPI 6.28318530718

__kernel void fft_radix2(__global float2* src, /*input array*/
                         __global float2* dst, /*output array*/
                         const int p,          /*block size*/
                         const int t) {        /*number of threads*/

    const int gid = get_global_id(0);
    const int k = gid & (p - 1);
    src += gid;
    dst += (gid << 1) - k;

    const float2 in1 = src[0];
    const float2 in2 = src[t];

    const float theta = -TWOPI * k / p;
    float cs;
    float sn = sincos(theta, &cs);
    const float2 temp = (float2) (in2.x * cs - in2.y * sn, in2.y * cs + in2.x * sn);

    dst[0] = in1 + temp;
    dst[p] = in1 - temp;
}
```

This kernel directly implements the inner two loops of Algorithm 1 described above. The host program provides the outermost loop. The kernel is called by host program $logN$ times with $N/2$

threads. At the end of every outermost iteration, the input and output array pointer is swapped, and block size increases by twice. We use a vector type `float2` to represent a complex number, in order to realize more contiguous memory accesses. We also use `sincos()` OpenCL function to calculate twiddle factor using only one call, which is faster than seperate `sin()` and `cos()` calls. We extend this kernel to a radix-4 kernel by merging two consecutive iterations. Two iterations of the FFT radix-2 algorithm will combine 4 sub-sequences of length p into one sequence of length $4p$, corresponding to a radix-4 algorithm. The kernel is called by host program $(logN)/2$ times with $N/4$ threads. The block size increases by 4 times after every iteration. Similarly, a radix-8 kernel can be developed. It is called by host program $(logN)/3$ times with $N/8$ threads. The block size increases by 8 times after every iteration.

4.2 Performance expectation

The algorithm is very easy to parallelize, due to the nature of its triple loop structure and independent memory access. Floating point computation is the most time-consuming operations in the kernel. Each kernel involves one `sincos()` function call and several floating point addition and multiplications.
Global memory access is the secondary time-consuming step. A radix-2, 4 and 8 kernel has 4, 8, 16 global memory reads+writes in each thread, respectively. Therefore, the total number of global memory accesses is $2N$ for each iteration. At the first few iterations, write operations are contiguous, since p is small. However after that, write operations are no longer contiguous.
This algorithm is neither a computation bound or memory bound algorithm. They are somewhat balanced based on our observation. Improving either of them will get a significant performance boost. It is not expected to achieve or beat the performance of existing libraries like FFTW, CUFFT or Apple FFT. Because we are neither using the best algorithm so far, nor generating optimized code on the fly to get good performance on all problems and platforms.

4.3 Source code

The source code is available on the Github repository: `https://github.com/miracle2121/hpc12-fp-rl1609`. It can be compiled and run on NYU HPC clusters with CPU and GPUs. The program takes two arguments: a problem size, and the prefix of device name you would like to run on. It uses a hard-coded array as input, in which all complex numbers are (1.0, 1.0). Then it outputs the result and performance metrics to the standard output.

5 Results

Our experimental platform is NYU HPC Bowery cluster. One computer node consists of an Intel Xeon X5650 six-core processor at 2.66GHz, 24GB of memory, and Nvidia Tesla M2070, which has 448 CUDA cores and 6GB memory. We test the performance of radix-2, 4 and 8 kernels on size $N = 2^{24}$ on both CPU and GPU. The measured time is the wall clock time in the host program, and we have turned off kernel profiling to get the best running times. The Gflop/s count is computed assuming a total number of real add+mul operations of $5Nlog_2(N)$.

	Xeon X5650	Tesla M2070
radix-2	3.38 Gflop/s	25.66 Gflop/s
radix-4	9.42 Gflop/s	39.15 Gflop/s
radix-8	11.12 Gflop/s	47.28 Gflop/s

Table 1: Performance (in Gflop/s) of kernels on size $N = 2^{24}$

Apparently radix-8 kernel is the fastest since it uses more private memory and reduces the need of global communication. However, radix-8 kernel only support those sizes N which $logN$ is divisible by 3. In practice we can combine different kernels to support all sizes and achieve the best performance. Now let's test the scalability of this algorithm using radix-8 kernel, by using different problem sizes.

	Xeon X5650	Tesla M2070
$logN = 15$	1.22 Gflop/s	18.76 Gflop/s
18	5.21 Gflop/s	39.19 Gflop/s
21	7.75 Gflop/s	56.53 Gflop/s
24	11.12 Gflop/s	47.28 Gflop/s
27	10.86 Gflop/s	50.05 Gflop/s

Table 2: Performance (in Gflop/s) of radix-8 kernels on different sizes ($logN$)

As we can see, the algorithm scales very well on both CPU and GPU.

References

[1] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[2] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High performance discrete fourier transforms on graphics processors. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 2:1–2:12, Piscataway, NJ, USA, 2008. IEEE Press.

[3] Naga K. Govindaraju and Dinesh Manocha. Cache-efficient numerical algorithms using graphics hardware, 2007.

[4] Kenneth Moreland and Edward Angel. The fft on a gpu. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '03, pages 112–119, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[5] Vasily Volkov and Brian Kazian. Fitting FFT onto the G80 Architecture. May 2008.

[6] Wikipedia. Discrete fourier transform — wikipedia, the free encyclopedia, 2012. [Online; accessed 5-December-2012].

[7] Wikipedia. Fast fourier transform — wikipedia, the free encyclopedia, 2012. [Online; accessed 5-December-2012].