Pease FFT의 GPU 가속

권오영*, 오세창**
*한국기술교육대학교 컴퓨터공학부
**세종사이버대학교 정보통신학과
e-mail:*oykwon@kut.ac.kr, **scoh713@gmail.com

GPU Accelating of Pease FFT

Oh-Young Kwon*, Se-Chang Oh**
*School of Computer Science and Engineering, Koreatech
**Dept. of Information and Communication, Sejong Cyber University

요 약

영상처리, 음성처리, 물리학, 천문학, 응용 수학등 다양한 분야에 FFT가 널리 사용되고 있다. FFT의 중요성 때문에 많은 연구가 이루어졌고, 최근 고속처리를 위하여 GPU를 활용한 FFT 알고리즘들이 개발되고 있다. 본 논문은 FFT알고리즘의 변형중 하나인 Pease FFT알고리즘을 GPGPU의 하드웨어 구성을 반영하여 최적화시킨 FFT 가속알고리즘을 제안한다. 실험결과 제안된 알고리즘은 CUFFT에 비하여 3% ~ 43%까지 우수한 성능을 보였다.

1. 서론

현대 사회에는 수많은 디지털 데이터가 존재한다. 이러한 디지털 정보들의 처리에 FFT 알고리즘이 많이 사용된다. 특히 디지털화된 영상이나 음성정보의 처리에 FFT가 널리 사용되고 있다. 그 외에도 물리학, 천문학, 응용 수학등 다양한 분야에 사용되고 있다[1]. FFT의 중요성 때문에 많은 연구가 이루어졌고, Cooley-Tukey 알고리즘에 기반한 다양한 방법들이 FFT 구현에 사용되었다[2].

FFT는 고속의 연산처리를 필요로 하고 있다. 최근 GPU는 단순 그래픽 처리를 넘어서 저가의 고속 병렬 연산 장치로써 주목을 받고 있다[3,4]. GPGPU기술을 이용하여 FFT의 성능을 향상시키려는 시도가 있었으며[5,6], NVIDIA사의 경우 FFTW[7]에 기반한 CUFFT[8]를 제공하고 있다.

본 논문은 FFT알고리즘의 변형중 하나인 Pease FFT알고리즘을 GPGPU의 하드웨어 구성을 반영하여 최적화시킨 FFT 가속알고리즘을 제안한다. 실험결과 제안된 알고리즘은 CUFFT에 비하여 3% ~ 43%까지 우수한 성능을 보였다. 본 논문의 2장에서 Pease FFT와 CUDA환경에 대하여 기술하고, 3장에서 가속알고리즘을 제안하고, 4장에서 실험 결과를 기술하고, 5장에 결론을 기술한다. .

2. Pease FFT 및 CUDA 환경

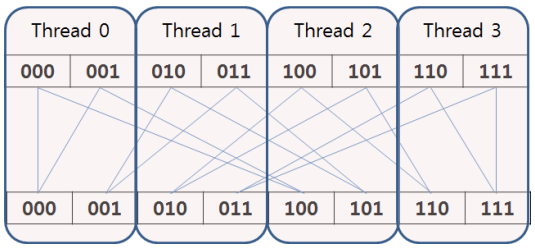
2.1 Pease FFT[9]

DFT(Discrete Fourier Transform)은 다음과 같이 정의된다.

$$y_k = \sum_{l=0}^{N-1} \omega_N^{kl} x_l, \quad 0 \le k < N, \qquad (1)$$

여기서 $\omega_N = e^{\pm \frac{2\pi i}{N}}$ 이다.

DFT를 고속으로 해결하기 위하여 다양한 방법들이 제시되었다. 그중 Pease FFT는 원소의 수가 $N = 2^n$이라는 가정하에 iterative FFT 방법을 변형한 알고리즘으로 프로그램의 control flow가 계산 스테이지와 독립적으로 구성되는 장점이 있다[2]. 즉 짝수(even)항과 홀수(odd)항을 짝(pair)를 이루어 계산하고, 그 결과를 퍼펙트셔플 네트워크 형태로 데이터를 전송하고, 이 과정의 매 스테이지마다 동일하게 반복된다. 그러므로, 병렬화나 적은 규모의 하드웨어 구성에 적합하다.



(그림 1) Pease FFT 수행 패턴

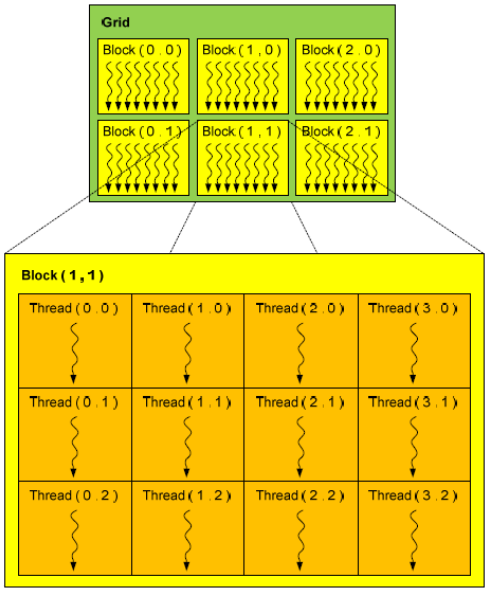
2.2 CUDA[10]

CUDA는 NVIDIA사의 GPGPU를 이용하여 병렬처리를 하기 위한 것으로 C언어 인터페이스를 제공하여 프로그래머블한 프로세서들로 구성된 GPGPU상에서 SIMD 구조의 문제들을 해결하는데 적합하다.

그림2는 GPGPU의 쓰레드 블록들과 그리드를 보여주고 있다. 하나의 그리드는 다수의 쓰레드 블록들을 가지고 있으며 각 쓰레드 블록들은 다수의 쓰레드들을 가질 수 있

다. 프로그래머는 쓰레드 블록과 각 쓰레드 블록 내의 쓰레드의 수를 설정해주어야 하고, 하드웨어와 드라이버들은 쓰레드 블록들을 GPGPU내의 프로세서들에 매핑시켜 준다. CUDA 프로그램은 쓰레드 블록내의 쓰레드 병렬성과 쓰레드 블록들 간의 병렬성을 같이 고려해야한다.

쓰레드 블록 내의 쓰레드들은 쉐어드 메모리를 통해 서로 통신할 수 있으며, 블록 내부의 쓰레드들간의 동기화가 가능하다. 하지만 블록들간의 동기화를 하는 방법을 제공하지 않아서 다수의 블록을 사용하는 경우 동기화에 주의하여야 한다. 블록간의 동기화를 위해서는 호스트(CPU)에서 커널함수(GPU)를 호출하여야 한다. 블록간의 동기화를 제공한 방법[11]이 제시되었지만 논리적인 블록의 수가 물리적인 SM의 수를 넘어가는 경우 동기화를 하지 못하는 단점이 있다.



(그림 2) GPGPU내의 쓰레드블럭과 그리드

CUDA의 메모리 모델은 텍스쳐 메모리, 콘스탄트 메모리, 글로벌 메모리, 공유 메모리, 레지스터, 로컬 메모리들로 이루어져 있으며, 각 메모리들의 엑세스 타임과 읽기, 쓰기 권한은 각각 다르다. 레지스터와 쉐어드 메모리는 고속으로 동작하는 반면 글로벌 메모리는 메모리 접근 지연시간이 길다. 프로그램의 성능을 높이기 위해서는 메모리 접근 형태에 대한 고려도 함께 이루어져야한다.

3. GPU 가속

한 쓰레드에서 2개의 원소를 이용하도록 단위 함수를 작성하면, 모든 쓰레드가 균등한 부하로 작업을 할 수 있다.

3.1 CUDA code

Pease FFT를 구현한 커널함수가 그림 3에 표시되어 있다. 16 ~ 29 라인까지가 그림 1에 표현된 연산을 구현한 부분이고, 이 과정을 n(=logN)번 반복한다. 30 라인은 [11]의 lock-free 동기화 방법을 직접 구현한 함수이다. 31 라인은 Pease FFT가 N개의 원소에 대하여 $2N$개 원소를 갖고 src와 dst를 번갈아 사용하므로 계산 스테이지가 끝날 때 마다 포인터를 이용하여 src와 dst를 교환하는 부분이다.

```
__global__ void
pease_kernel(Complex *dst, Complex *src, int logN, int dir)
{
1:  int t, r;
2:  int stride;
3:  int step;
4:  Complex w;
5:  Complex s0, s1;
6:  Complex d0, d2n;
7:  Complex *tmpsrc, *tmpdst, *tmp;
8:  float theta, tfcos, tfsin;

9:  unsigned int idx  = blockIdx.x * blockDim.x + threadIdx.x;
10: unsigned int idx2 = (idx << 1);

11: if (idx == 0) ArrInit();

12: __syncthreads();

13: tmpsrc = src; tmpdst = dst;
14: stride = NX >> 1;

15: for (step = 0; step < logN; step++) {
16:     s0 = tmpsrc[idx2];      // even
17:     s1 = tmpsrc[idx2+1];    // odd

18:     theta = dir*M_PI/(1 << step);
19:     r = idx/(1 << (logN-1-step));
20:     theta = theta*r;
21:     tfcos = __cosf(theta);
22:     tfsin = __sinf(theta);

23:     w.x = s1.x * tfcos - s1.y * tfsin;
24:     w.y = s1.y * tfcos + s1.x * tfsin;

25:     d0.x = s0.x + w.x;     d0.y = s0.y + w.y;
26:     d2n.x = s0.x - w.x;    d2n.y = s0.y - w.y;

27:     __syncthreads();

28:     tmpdst[idx] = d0;
29:     tmpdst[idx+stride] = d2n;

30:     __gpu_lfsync(1, ArrIn, ArrOut);

31:     tmp = tmpsrc; tmpsrc = tmpdst; tmpdst = tmp;

32:     __syncthreads();
    }
}
```

(그림 3) CUDA code for Pease FFT

3.2 단일 블록 처리

병렬성을 높이려면 다수의 블록을 사용하는 것이 바람직하다. 하지만 이럴 경우 블록동기화가 필요하고, 글로벌 메모리의 참조가 많아지므로 좋은 FFT의 좋은 성능을 기대할 수 없다[11]. 그러므로 본 논문에서는 하나의 블록에서 최적으로 처리할 수 있는 쓰레드 수와 메모리의 수를 구하고 이를 바탕으로 Pease FFT를 가속하였다.

그림 1과 그림 3의 구현에서 보듯이 하나의 쓰레드가 2개

의 원소를 처리하고 있다. 그리고, 쓰레드당 명시적으로 26개의 변수 선언에 함수 내부에서 임시 사용이나 함수호출등에 사용될 수 있는 레지스터를 고려하면 약 35개 정도의 레지스터가 필요할 것으로 추정된다. 블록에서 사용할 수 있는 레지스터수의 양은 고정이 되어있으므로 하나의 블록에서 최대한 사용가능한 쓰레드 수는 (2)와 같이 제한된다.

$$T_{reg} < \frac{R}{V} \qquad (2)$$

여기서, T_{reg}은 쓰레드의 수, R는 32-bit 크기의 레지스터 수이고, V는 32-bit 크기로 환산한 변수의 개수이다. 만일 블록당 최대 사용 레지스터수가 32,768개 이면, $T_{reg} < 32{,}768/35 = 963.2$ 으로 각 블럭에서 최대 963개의 쓰레드 활용이 가능하다.

Pease FFT는 $N(=2^n)$개의 원소에 대하여 $2N$개의 메모리를 필요로 한다. CUDA의 메모리 계층구조를 효과적으로 이용하기 위해서는 블럭내의 공유메모리나 L1 캐쉬를 충분히 활용하여야 한다. 그러므로 한 블럭에서 효과적으로 처리할 수 있는 FFT의 크기는 공유메모리나 캐쉬의 크기에 의해 제한된다. 결국 메모리의 크기에 의해 제한되는 쓰레드의 수는 (2)와 같다.

$$T_{mem} < \frac{M_s}{4C} \qquad (2)$$

여기서, T_{mem}은 메모리에 의해 제한되는 쓰레드의 수, M_s는 L1 캐쉬나 공유메모리의 크기, C는 사용된 complex변수의 바이트 단위 크기이다. 각 쓰레드가 2개의 원소를 처리하므로 쓰레드당 4개의 원소가 필요함을 의미하는 상수이다. 공유 메모리가 49,152바이트인 경우 T_{mem}=1,536이 된다.

한 블록을 이용해서 효과적으로 처리할 수 있는 FFT의 크기는 한 블록에서 사용할 수 있는 쓰레드로 결정된다. 즉 $\lfloor \log_2 \min(T_{reg}, T_{mem}) \rfloor$을 만족하는 값으로 쓰레드 수를 정한다. 위의 예제로 사용한 값들을 이용하면 한 블록에서 처리할 수 있는 최대 쓰레드의 수는 512이다.

3.3 다중 블록 처리

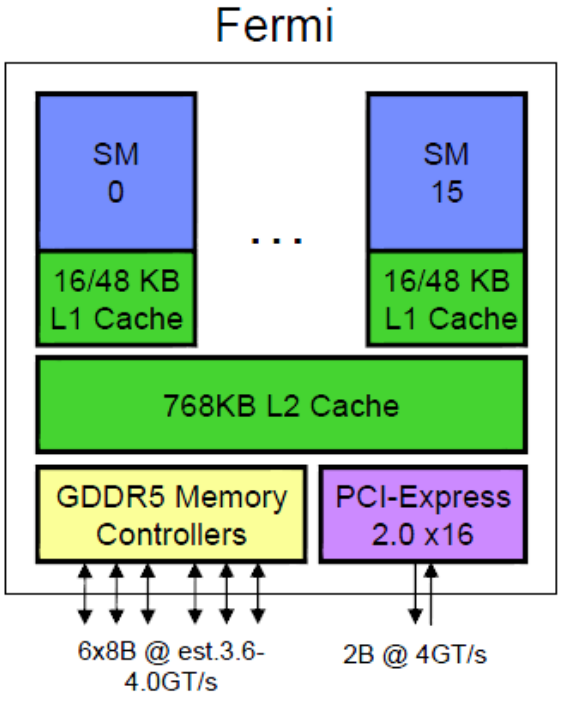
병렬성을 최대한 이용하는 것보다는 한 블록으로 처리가 가능하면 한 블록으로 처리하는 것이 바람직하다. 하지만 FFT를 수행해야할 크기가 한 블록의 계산 범위를 넘을 경우 블록동기화 문제가 발생한다. 하지만 [11]의 방법을 사용하면 물리적으로 제공하는 SM(Streaming Multiprocessor)의 개수까지 블록간의 그림 3의 코드를 그대로 활용할 수 있다. 본 논문에서는 SM의 개수보다 많은 블록을 요구하는 크기의 FFT는 고려하지 않았다. 그 이상의 FFT는 CUFFT등 다른 방법으로 해결할 수 있다. [1]에서 사용한 Four step FFT도 좋은 대안이 된다. Four step FFT는 일차원 배열을 이차원 배열 $n_1 n_2 (=N)$으로 분할하여 다음과 같이 계산한다.

1. n_1개의 행에 대하여 각각 독립적으로 FFT 수행
2. 배열원소 a_{ij}에 twiddle factor $e^{\pm\frac{2\pi ij}{N}}$ 를 곱하고 $n_1 n_2$ 배열을 $n_2 n_1$ 배열로 transpose
3. n_2개의 행에 대하여 각각 독립적으로 FFT 수행
4. $n_2 n_1$배열을 $n_1 n_2$배열로 transpose

4단계중에 1, 3단계에 본 논문에서 제안한 방법을 적용할 수 있다.

4. 실험결과

본 논문에서 제안방법은 Fermi 구조를 사용하는 GeForce GTX 480을 사용하여 실험하였다. CPU는 Intel(®) Core™ 2 Quad CPU Q9400 @ 2.66Ghz이고, 메인 메모리는 3GB, 운영체제는 Windows 7 64-bit 버전이다. GPGPU의 CUDA 런타임환경은 4.2버전이고, 32개의 코어를 갖는 15개의 SM으로 이루어져 있고, 블록당 최대 1024개의 쓰레드를 사용하고, 48Kb의 공유 메모리를 갖고 있으며, 32K개의 32-bit 레지스터를 가지고 있다. 그림 4는 페르미 구조를 간략하게 보여주고 있다[12]. Fermi 구조에서 SM 내의 L1 cache와 공유 메모리는 같은 계층구조에 있고 동일한 영역을 분할해서 사용하므로 명시적으로 공유메모리를 선언하여 사용할 필요가 없다.



(그림 4) 페르미 구조

식 (1)과 (2)를 사용하여 한 블록에서 사용할 수 있는 최적으로 쓰레드 수를 구해본 결과 512개의 쓰레드를 활용하여 1024개의 원소에 FFT를 수행하는 것이 적합함을 알 수 있다. GTX 480의 경우 한 블럭당 최대 1024개의 쓰레드가 허용된다. 하지만 1024개의 쓰레드를 모두 사용할 경

우 논리적으로는 한 블럭이지만 물리적인 SM블럭에서 사용할 레지스터 수가 부족하게 되어 2개의 SM을 사용하게 되므로, 오히려 성능의 감소를 초래한다.

단일 블록으로 쓰레드 128, 256, 512개를 사용하여 각각 256, 512, 1024 크기의 FFT를 수행하였다. 실험은 FFT와 Inverse FFT를 번갈아 2번 수행한 시간을 밀리세컨드 단위로 측정하였다. 이 실험을 5번 반복하여 평균을 기록하였다.

(표 1) 단일 블록 성능

(단위: ms)

방법 \ 원소수	256	512	1024
CUFFT	0.99	0.96	1.01
pFFT	0.56	0.65	0.73

제안한 방법이 CUFFT에 비하여 28% ~ 43%의 성능 개선 효과가 있었다. 다중 블록도 같은 방법으로 시간을 측정하였다. SM의 개수가 15개이므로 최대 8개의 블록만을 사용할 수 있었다. 즉 FFT의 크기가 8192까지 허용한다.

(표 2) 다중 블록 성능

(단위: ms)

방법 \ 원소수	2048	4096	8192
CUFFT	1.12	1.04	1.12
pFFT	0.93	0.96	1.082

다중 블록을 사용할 경우 lock-free 동기화의 부담 때문에 성능개선이 3% ~ 17%정도만 이루어졌다. 블록간 동기화를 위하여 atomicCAS 연산을 사용한다. 이 연산의 오버헤드가 커서 블록의 수가 많아질수록 성능 개선이 미비한 것으로 추정된다.

5. 결론

과학 및 공학분야에서 FFT는 널리사용되는 알고리즘중의 하나이다. FFT 알고리즘에 대하여 많은 연구가 되어왔고 Pease FFT알고리즘의 경우 구조의 단순성 때문에 적은 규모의 하드웨어 구성에 많이 사용되었다.

본 논문에서는 Pease FFT를 GPU의 특성에 맞게 가속하는 방법을 제안하였다. 단일 블록에서 최적으로 처리할 수 있는 쓰레드 수를 구하고, 이 쓰레드 수보다 적은 쓰레드를 요구하는 FFT는 단일 블록으로 처리하고, 다중 블록을 사용하는 경우는 블록간 동기화 방법을 사용하여 물리적인 SM의 개수까지 다중 블록으로 FFT를 수행하는 방안을 제시하였다. 제안한 방법을 GTX 480 그래픽카드를 이용하여 실험한 결과 CUFFT보다 약 3%~43%까지 성능개선이 이루어졌다. 단일 블록일 경우 성능개선의 더 많이 이루어졌다. 향후 효과적인 동기화 알고리즘 개발과 다중 블록의 한계를 극복할 수 있는 GPU가속 방안 개발이 필요할 것이다.

참고문헌

[1] Naga K. Govindaraju, et. al., “High performance discrete fourier transforms on graphics processors,” in Proc. Supercomputing (SC), 2008, pp. 1 - 12.

[2] Franz Franchetti, et. al., "Discrete Fourier Transform on Multicores," IEEE Signal Processing Magazine, Vol. 26, No. 6, pp. 90-102, 2009. (http://users.ece.cmu.edu/~franzf/papers/spmag09.pdf)

[3] J. Nickolls, et. al, “Scalable parallel programming with CUDA”, Queue, vol. 6, no. 2, pp. 40-53, Mar/Apr 2008.

[4] NVIDIA, “GPU Computing SDK”, https://developer.nvidia.com/gpu-computing-sdk

[5] Vasily Volkov and Brian Kazian, "Fitting FFT onto the G80 Architecture," CS258 final project, UC Berkeley, 2008.

[6] Yuri Dotsenko, et. al., "Auto-tuning of Fast Fourier Transform on Graphics Processors," PPoPP'11. Feb. 2011, Texas, USA.

[7] M. Frigo and S. G. Johnson, “The design and implementation of FFTW3,” Proceedings of the IEEE, vol. 93, no. 2, pp. 216 - 231, 2005. (http://www.fftw.org)

[8] NDVIDA, "CUFFT Library," https://developer.nvidia.com/cufft

[9] J.R. Johnson, "Pease FFT Algortihm," Technical Report DU-MCS-98-01, Dept. of Math. and CS., Drexel University, USA, Nov. 1998

[10] David B. Kirk and Wen-mei W. Hwu, *Programming Massively Parallel Processors*, Morgan Kaufmann, 2010.

[11] Shucai Xiao and Wu-chun Feng, "Inter-Block GPU Communication via Fast Barrier Synchronization," IPDPS, pp. 1-12, April 2010.

[12] http://www.realworldtech.com/fermi/3/