Comparative Study of High Performance Computing Using Multi-core Parallel Systems

Hyo Jong Lee[1,2], **Hyeon Kyu Kim**[1]
[1] Division of Computer Science and Engineering
[2] Center for Advanced Image & Information Technology
Chonbuk National University, Jeonju Korea
hlee@jbnu.ac.kr, hkskyp@gmail.com

Abstract - Multi-core based high performance computing systems are available with a reasonable price. Parallel programming paradigm needs to be adjusted to an individual system. Parallel computing systems were compared in this paper. Electroencephalography signals were collected in order to measure performance of parallel computing for CPU and GPU based systems. A CPU based system showed better performance for smaller data set, while a GPU system showed better performance for larger data set. GTX580 processor, which has 512 CUDA cores, showed consistent speedup as input data was increased continuously. However, CPU has a limited speedup due to the lack of parallelism. For the FIR filter computation, GPU showed a good scalability, while a CPU system did not. The performance of GPU was better than CPU system slightly.

Keywords: CPU; GPU; multi-core; High Performance Computing; parallel FIR filter;

1 Introduction

Recently the high performance computing systems are taking a direction to multi-core based microprocessor since the power demands of increased clock speeds cannot be managed efficiently. Owing to the advanced VLSI and CPU design technology, multi-core systems are becoming popular satisfying customers' needs. Some scientists expect the core counts per chip would rise as the number of transistor increase according to Moore's law [1]. It is experienced that high performance computing (HPC) system is easy to build up by using basic building block of multi-core chips sharing memory in a single node. Clustering those basic computing blocks with a high speed network would be a convenient way to construct more powerful computing systems.

Programming models for parallel systems require different approaches. Traditionally it was common to develop parallel programming models for heterogeneous systems equipped with hybrid memory systems. Parallel library developers take advantage of the shared memory within a single node, but tried to optimize the inter-connected communications. Thus, a user may get good performance for his parallel applications using a single standardized programming model. A programming model for shared memory system is easy to control, but must overcome increased memory bandwidth for a large scale problem. Jin et al. [1] proposed a hybrid method to program multi-core based HPC systems combining standardized programming models. They also extended the OpenMP model with new data locality extensions to better match the more complex memory system.

GPU (Graphics Processing Units) system is a typical many-core system. In the beginning they were used for graphics processing only. Since NVIDIA released CUDA, the GPU becomes GPGPU (general-purpose computing on graphics processing units) and more applications became much accessible on it. Although programming on GPU requires optimization to utilize the maximum potential of the CPU, its performance was proved to be worth to pay the cost for optimization. For example, 2D Discrete Cosine Transform (DCT) problem for a 256x256 grey image took 10 seconds on a CPU, but just 48 milliseconds on a CPU using an optimized implementation [2].

There are other criteria besides performance to evaluate parallel programming models, such as scalability and the cost to performance ratio. The scale of some problems is not big to require an expensive supercomputer, although parallel programming approach would help users. Thus, it is important to investigate performance characteristics considering the cost to performance ratio. The low priced parallel computing system may be beneficial to a small scale parallel computing.

In this study we compare the performance of four different systems, a common desktop PC with a quad-core, a medium level work station, a low and a high priced GPU system. They are all easy to purchase depending on a user's various circumstance. A target problem was selected from a bioinformatics area called analysis of electroencephalography (EEG) signals. The same problem was implemented for each system for performance measurement. Each implementation did not require any special skill or serious programming time. That is, the cost for implementation would be considered as similar level.

2 Background

2.1 Electroencephalography signal

A target problem adopted in this study is to select correct bandwidth from electroencephalography (EEG). It is required to understand the concept of EEG. EEG is the recording of electrical activity along the scalp. Electrical recordings from the surface of the brain or even from the outer surface of the head demonstrate that there are continuous electrical activities in the brain. Both the intensity and the patterns of this electrical activity depend on the level of excitation of different parts of the brain resulting from sleep, wakefulness, or brain diseases such as epilepsy or even psychoses. The undulations in the recorded electrical potentials are known as brain waves, and the entire record is called an EEG [3]. Intensity of EEG recording range from 0 to 200 microvolt on the surface of the scalp, and their frequency ranges from once every few seconds to 50 or more per second. The characteristics of the waves are dependent on the degree of activity in respective parts of the cerebral cortex. The waves change markedly between the states of emotions. Much of the time, the brain waves are irregular, and no specific pattern can be discerned in the EEG.

There are mainly five types of Brain waves: Delta waves (0.5-4 Hz) which are considered to be related to the deep sleep [4] in the adults or premature babies. It is usually found in the frontal region of brain in adults and posterior region in children. A common Theta wave (4-8 Hz) which occurs in children and adults when they are in emotional stress or they have deep midline disorders. It is found in parietal and occipital region. Another type of theta waves is named frontal midline theta. The theta waves exist during the various tasks which need the correlation of the increased mental effort and sustained concentration [5]. Alpha wave (8-13 Hz), which occurs in quiet resting state but not sleep, is found in the occipital region. Alpha waves can reflect the relaxation level a person is having. They are also believed to be responsible for the movement related brain activity. Another role of Alpha rhythms is to handle a perceptual processing, memory tasks, and emotions [4]. Beta wave (13-30 Hz) occurs in active and busy concentration or anxious thinking state. It is found in the frontal and parietal region and is related to the concentration level of people [6]. An increase in a beta power may reflect the increase of the arousal level of an emotional state [5]. Gamma wave (30-100 Hz) which occurs in certain cognitive or motor functions. It is often used for diagnosis of the certain brain illness [4].

2.2 Finite Impulse Response filter

A finite impulse response (FIR) is a digital filter whose impulse response is of finite duration in signal processing. This is in contrast to infinite impulse filters, which may have internal feedback and may continue to respond indefinitely. The output y of a linear time invariant system is determined by convolving its input signal x with its impulse response h. Figure 1 displays a discrete-time FIR filter of order M. For a discrete-time FIR filter, the output is a weighted sum of the current and a finite number of previous values of the input marked as h in Figure 1.

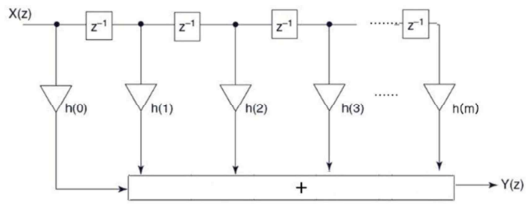

Fig 1. Diagram of discrete-time FIR filter design

The operation of FIR filter is described by Equation (1), which defines the output sequence Y(z) in terms of its input sequence X(z).

$$y[z] = \sum_{i=0}^{M} b_i x[z-i] \tag{1}$$

One property of the FIR filter is not to require feedback. That is, any rounding errors are not compounded by summed iterations. The same relative error occurs in each calculation. Another property is that the filter is inherently stable. This is due to the fact that all the poles are located at the origin and thus are located within the unit circle. Generally speaking, it is easy to design to be linear phase by making the coefficient sequence symmetric. Selection of coefficients is a key step in designing of filters. Most of the time filter specifications refer to the frequency response of the filter.

Applying the FIR filter to EEG signals, desired bandwidth of brain waves may be selected. First of all, FIR filter can eliminate unwanted artifact signals from the raw EEG signals. Figure 2 shows the original EEG signals for each channel. Each channel is affected by electrical noises or eye blinking noise. Since the magnitude of EEG signal is very low compared to those artifact signals, those unwanted signal must be removed.

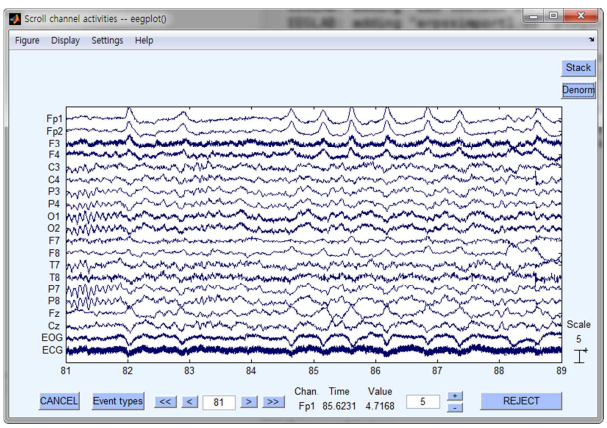

Fig 2. Example of original raw EEG signals

Figure 3 shows the result signals after applying FIR filter to eliminate electrical noise. The part of artifact signals were cleanly removed by selecting band of 4~50Hz. The

electrical noise is distributed around 60Hz, the FIR filter was able to remove noise components.

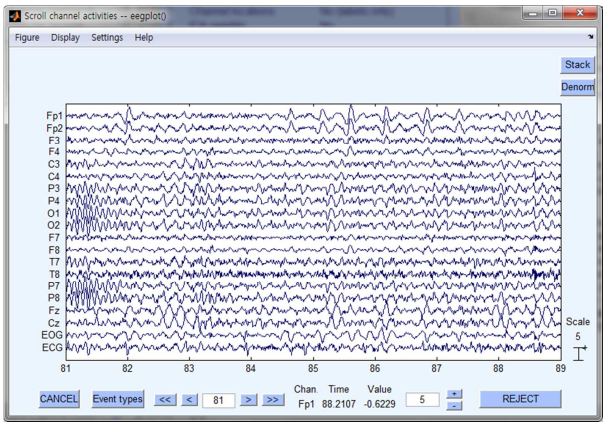

Fig 3. Result signals of Fig. 2 after applying FIR filter of 4~50HZ

2.3 CUDA

Since NVIDIA introduced CUDA, a general purpose parallel computing and programming model became available to developers as an ideal tool to solve many complex computational problems in a more efficient way than a traditional model on a CPU. CUDA comes with a programming environment that developers can use C programming language as a native to a GPU system. The proposed programming model exploits the maximum power of multi-core CPU or GPU system by deploying parallel technique. The challenge of multi-core system is to develop application software that transparently scales its parallelism to leverage the increasing number of processor cores. The CUDA programming model is designed to overcome this problem.

In CUDA programming tasks are handled by kernel function call. A kernel function is the function being executed in GPU and consists with blocks and grids. The computational grid consists of a grid of thread blocks. Each thread executes the kernel. Figure 4 shows the relationship between CPU and GPU.

The parallel kernel in Figure 4 can be executed on CPU either synchronously or asynchronously. Thus, GPUs are multi-thread computational engines based on stream computing. They can execute hundreds of threads simultaneously. That is, a CUDA process, which constructs the multiple of eight streams multiprocessor, executes kernel functions. A single stream multiprocessor consists of 32 or 48 CUDA processors in the Fermi architecture. A single stream multiprocessor can execute 1,536 threads at maximum concurrently [7]. The construction of threads in a unit block is critical to the performance of CUDA programming. It is important to make sure the optimal construction for the best performance.

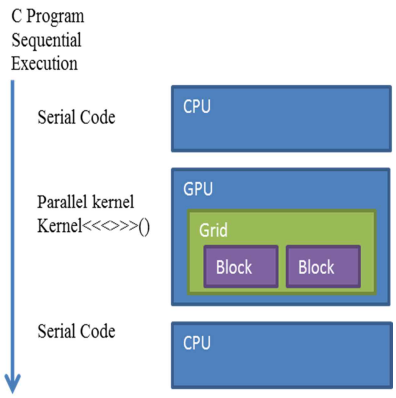

Fig 4. Relationship between CUDA and CPU

3 Experiment

Data processing time of EEG data is obviously proportional to the number of channels and experimental duration. The size of data is 17Mbytes (for double precision) approximately for one hour collection from a single channel. In this experiment the international standard 10-20 system was used to measure EEG signals using19 channels. The international 10-10 system has 71 electrodes to cover a whole brain. Assuming one hour of experiment duration, data size would be 260Mbytes and 973Mbyes approximately for 10-20 and 10-10 system, respectively. In this paper data collection time was controlled so that 24.7Mbytes, 49.4Mbytes, 98.8Mbytes, and 197.6Mbytes of data were collected.

The FIR filter function has been implemented for both CPU and GPU system. Two main systems were used for this experiment. The first system is a desktop PC equipped with Intel Core i5-2500 3.30GHz, which has a quad-core. This system also has NVIDIA GeForce GTX550 Ti 1GiB installed. This system runs on Window7 Enterprise 64-bit. The other system is a middle class server, which has a Dual Intel Xeon X5650 2.67GHz. The system has total 12 physical CPUs or 24 logical CPUs permitting the Hyper Threading technology. This system also has a NVIDIA GeForce GTX 580 3GiB installed. The system runs on CentOS 6.3 64-bit. Both system installed Version 5.0 of CUDA driver.

Different parallel approaches were applied for a CPU system based on multicores and GPU system based on many cores. Total number of electrodes were distributed to each core similarly (or equally) for a CPU system. In this case a single core processed all data distributed to the core by deploying FIR filter. In the case of a GPU system data collected by a single channel is processed on to a single kernel function. That is, a single thread handled a single input data stream. For example, a single core of a quad-core CPU system will process data collected from 5 channels, while a single thread processes a single channel data by calling a kernel function 20-times.

Firstly, the coefficient *b* from Equation 1 was implemented based on EEGLab [9] implementation. Then, FIR filter was parallelized separately for both of CPU system and GPU system. A loop unrolling technique has been used for optimization for both CPU and CUDA. OpenMP 2.0 was used for CPU parallel implementation, which required minimal code changes for a shared memory system. A SSE (Streaming SIMD Extensions) 2.0 operations were also used for CPU parallelism. The pseudo code of FIR filter for a CPU system is depicted in Figure 5. The usage of directive was prohibited in order to minimize variable spaces, which increases due to the directives under loop unrolling. At line 07 from Figure 5 (0:6) means that seven variables from zero to six were used. (0:2:120 represents that variable 0 to 12 were used for only even numbered variables. Since the data was double type, two SSE instructions were processed at lines between 07 and 12 from Figure 5.

The FIR filter was translated to CUDA version of C language. . In the CUDA, add and multiply operators called in a FIR filter were translated into a single MAD (multiply / add) operator. Figure 6 shows a pseudo code of FIR filter for CUDA. The function fir_filter was executed on a CPU system and a kernel function was executed on a GPU system. The bandwidth of data communication between GPU and GPU was efficiently reduced by using OpenMP. The line 07 was optimized by substitution of MAD instructions after CUDA compilation options.

The source codes were compiled with Visual Studio 2010 SP1 with an O3 option for code optimization. For CentOS system, GCC 4.6.3 was used for compilation. A binary code was created with the same level of optimization – O3. The CUDA compiler nvcc was used for both systems with the same compiler option level.

```
01    #pragma omp parallel for
02    for( size_t i = 0; i < eeg_data.size(); ++i ) {
03     size_t j = 0;
04     size_t size = (Y.size()/14)*14;
05     for( ; j < size; j+=14 ) {
06      for( size_t k = 0; k < b.size(); ++k ) {
07       __m128d sb = _mm_set_pd(b[k], b[k]);
08       __m128d sY(0:6) = _mm_load_pd(&Y[j+(0:2:12)]);
09       __m128d sX(0:6) = _mm_load_pd(&X[j+(0:2:12)+k]);
10       sX(0:6) = _mm_mul_pd(sb, sX(0:6));
11       sY(0:6) = _mm_add_pd(sY(0:6), sX(0:6));
12       _mm_store_pd(&Y[j+(0:2:12)], sY(0:6));
13      }
14     }
15     for( ; j < Y.size(); ++j ) {
16      for( size_t k = 0; k < b.size(); ++k ) {
17       Y[j] += b[k]*X[j+k];
18      }
19     }
20    }
```

Fig 5. Pseudocode of FIR filter for CPU parallelism

```
01    __global__ void kernel(b, X, Y) {
02     int gid = blockDim.x*blockIdx.x + threadIdx.x;
03     if( gid < Y_size ) {
04      double sum = 0.0;
05      #pragma unroll 8
06      for( size_t i = 0; i < b_size; ++i ) {
07       sum = b[i]*X[gid+i]+sum;
08      }
09      Y[gid] = sum;
10     }
11    }
12    function fir_filter(b, X, Y) {
13     #pragma omp parallel for
14     for( size_t i = 0; i < eeg_data.size(); ++i ) {
15      memcpy host to device;
16      kernel<<<blocks, threads>>>(b, X[i], Y[i]);
17      memcpy device to host;
18     }
19    }
```

Fig 6. Pseudocode of FIR filter for CUDA

4 Result And Discussion

The FIR filter function has been successfully implemented and compiled as described in Section III. In order to measure correct execution times, four different executable files were created for a desktop PC, a server, and two GPU programs for both systems. Table I and II summarized execution times for both systems.

Table 1. Performance measurement of CPU and GPU for FIR filter for a desktop (unit: milli-second)

Data length x Multiples	**CPU(Intel i5-2500)**			**GPU**
	Thread 1	***Thread 4***	***Thread 8***	***GTX 550 Ti***
161,890x1	1,405	379	533	351
161,890x2	2,189	617	753	531
161,890x4	4,021	1,133	1264	976
161,890x8	7,429	2,175	2262	1,517

Table 2. Performance Measure of CPU and GPU In FIR filter for a server (unit: milli-second)

Data length x Multiples	**CPU(Dual Xeon X5650)**					**GPU**
	Thread 1	***Thread 4***	***Thread 8***	***Thread12***	***Thread24***	***GTX 580***
161,890x1	1,632	423	280	208	270	339
161,890x2	2,747	713	470	357	429	393
161,890x4	5,058	1301	851	625	759	522
161,890x8	9,431	2446	1601	1,197	1,422	820

The desktop PC supported up to four cores. Table I shows the execution time decreased as the number of cores increased linearly. The GTX 550 Ti supports 192 CUDA cores. According to Table I the execution on the GPU system

is slightly better than quad-core system. Table II summarized the result from a server processor, which provides 24 HT cores including 12 physical cores. The execution time of a server processor showed decreased linearly as the number of threads increased until 12-thread. However, the execution time of 24-thread is longer than the 12-thread due to the limited ALU functionality. The HT technique allows two logical cores for a single physical core; however, those two logical cores must share a single ALU. This restriction becomes speedup bottle neck for computing intensive problems. Since a FIR filter is belongs to a computing intensive problem, it experienced slow-down for more logical cores. GPU system installed on a server was GTX 580, which has 512 CUDA cores. Its performance was slightly better than the GPU installed on a desktop PC. It also shows slightly better than six-core case, but 60% slower than 12-core for small problem cases (161,890x1x20 and 161,890x2x20). This delay was caused by the same reason as shown in a desktop PC. However, as the problem size gets larger, GPU upbeats the server system.

Figure 7 displays the speedups of each system based on the Xeon X5650 single-thread, which showed the slowest processing time. In the case of CPU parallel implementation, a speedup reached up to 8-fold. However, both systems did not demonstrate that their performances would be better than the 8-fold. A GPU system on a server showed about 11-fold faster speedup. As the size of problem is increased, the speedup is also increased because more computing available threads are available. The speedups of GTX are higher compared to CPU for larger problem domains. The reason for this better performance is caused by wider memory bandwidth in a GPU processor. In the case of GTX 580, the speedup is worse than CPU due to the increased communication overhead.

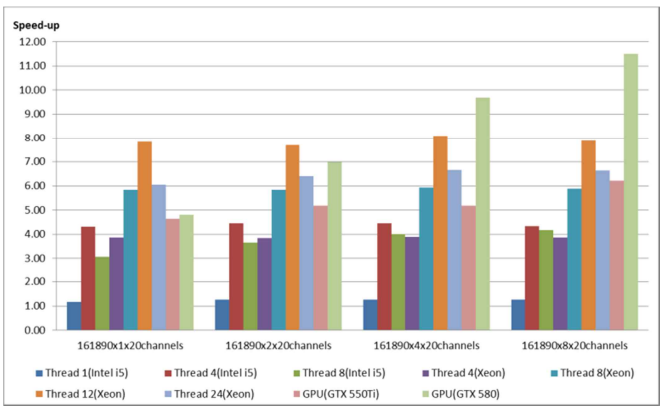

Fig 7. Speedups for GPU and CPU based on Xeon CPU

5 Conclusions

In this paper the performances of FIR filter execution were compared on various machines. The input data were collected through EEG system, of which sample rate was 500Hz. The EEG signals were assumed to be collected for 5min, 10min, 20min and 40 minutes. CPU showed better performance for smaller data set, which were collected for 5 min and 10 min, while GPU showed better performance for larger data set, which were collected for 20 min and 40 min. GTX580 processor, which has 512 CUDA cores, shows consistent speedup as input data is increased continuously. However, CPU has a limited speedup due to lack of parallelism. For the FIR filter computation, GPU showed a good scalability, while CPU did not. The performance of GPU was better than CPU, however, the difference was not significant due to small problem size of a FIR filter.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2012R1A2A2A03).

References

[1] H. Jin, D. Jespersen, P. Mehrotra, R. Biswas, L. Huang, B. Chapman, "High performance computing using MPI and OpenMP on multi-core parallel systems", Parallel Computing Vol 37, pp. 562-575, 2011.

[2] M. Nixon and A. Aguado, Feature Extraction & Image Processing, Newnes, 2002.

[3] Guyton,A.C.,Hall J.E, Textbook Of Medical Physiology, Elsevier Inc., Philadelphia, 2005.

[4] S. Sanei and J. Chambers, EEG signal processing. Chichester, England; Hoboken, NJ: John Wiley & Sons, 2007.

[5] D. Sammler, M. Grigutsch, T. Fritz, and S. Koelsch, “Music and emotion: Electrophysiological correlates of the processing of pleasant and unpleasant music,” Psychophysiology, vol. 44, pp. 293-304, 2007.

[6] Larsen, R. J., & Buss, D. M. Personality psychology. NewYork: McGraw-Hill, 2002

[7] NVIDIA. (2009). FERMI Compute Architecture White Paper (v1.1ed.). http://www.nvidia.com/

[8] Kennett R. Modern electroencephalography. J Neurol 2012;259: 783-9

[9] D. A and M. S., "EEGLAB: an open source toolbox for analysis of single-trial EEG dynamics including independent component analysis," Journal of Neurosci Methods, vol. 134, pp. 9-21, 2004.