Implementation of large-scale FIR adaptive filters on nVIDIA GeForce graphics processing unit

2010 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS 2010) December 6-8, 2010
TA1.2.2

IMPLEMENTATION OF LARGE-SCALE FIR ADAPTIVE FILTERS ON NVIDIA GEFORCE GRAPHICS PROCESSING UNIT

Akihiro Hirano and Kenji Nakayama

Kanazawa University
Kakuma-Machi, Kanazawa, 920-1192, Japan

ABSTRACT

This paper presents implementations of an FIR adaptive filter with a large number of taps on nVIDIA GeForce graphics processing unit (GPU) and CUDA software development environment. In order to overcome a long access latency for slow off-chip memory access, reduction of memory accesses by re-ordering and vector load/store operations and an increase of the number of threads are introduced. A tree adder is introduced to reduce the cost for summing thread outputs up. A simultaneous execution of multiple filters are also examined. On low-cost platform such as an Atom/ION nettop, GPU will accelerates the computation by almost three times. For simultaneous multiple simulations such as an ensemble averaging, a GPU with a large number of processing elements outperforms a dual-core CPU; almost six times faster for 16 runs.

1. INTRODUCTION

Echo cancellers are used to reduce echoes in a wide range of applications, such as teleconference systems and hands-free telephones. For acoustic echo cancellers (AEC's), the number of taps is very large; from several hundreds to several thousands. Therefore, efficient implementation of AEC's has been studied[1, 2]. In research of AEC's, optimization of adaptation parameters requires multiple simulations. Thousands of simulations for ensemble averaging might be necessary in order to confirm a convergence analysis[3]. Parallel simulations might be useful for these cases.

Recent years, PC-based communication systems such as Skype and Messenger becomes very popular. PC-based systems are also useful for simulations because they have powerful CPU's over giga floating-point operations per second (FLOPS) performance. Recent PC's are also equipped with powerful graphics processing units (GPU's). These GPU's are also capable of numerical computations by using C/C++ language[4, 5, 6] and have been used for computer simulations. Latest GPU's have computation performance over tera FLOPS. Even some low-cost chipsets consist of programmable GPU's. An example is ION platform by nVIDIA for Intel Atom processor.

In this paper, an implementation of an FIR adaptive filter on nVIDIA GeForce family GPU and CUDA is discussed. Section 2 describes the FIR adaptive filter with the normalized least mean squares (NLMS) algorithm[7]. GeForce family GPU and CUDA is briefly described in Sec. 3. The proposed implementation is shown by Sec. 4. Section 5 compares the performance.

2. FIR ADAPTIVE FILTER BASED ON NLMS ALGORITHM

The adaptive FIR filter generates its output signal $y(n)$ from the input signal $x(n)$ and the filter coefficient $w_k(n)$ by

$$y(n) = \boldsymbol{w}^T(n)\boldsymbol{x}(n) \tag{1}$$

$$\boldsymbol{x}(n) = [x(n)\ x(n-1)\ \cdots\ x(n-N+1)]^T \tag{2}$$

$$\boldsymbol{w}(n) = [w_0(n)\ w_1(n)\ \cdots\ w_{N-1}(n)]^T, \tag{3}$$

where N is the number of taps, $[\cdots]^T$ is a transpose of a vector $[\cdots]$. The error signal $e(n)$ between the desired signal $d(n)$ and the filter output $y(n)$ is calculated by

$$e(n) = d(n) - y(n). \tag{4}$$

By using the NLMS algorithm[7], the filter coefficient vector $\boldsymbol{w}(n)$ is updated by

$$\boldsymbol{w}(n+1) = \boldsymbol{w}(n) + \frac{\mu e(n)\boldsymbol{x}(n)}{|\boldsymbol{x}(n)|^2} \tag{5}$$

where a positive constant μ is a step-size parameter.

3. NVIDIA GEFORCE GPU AND CUDA

In this implementation, nVIDIA GeForce 8000 family or later GPU's are assumed. Though GeForce 8800 GTS and GeForce 9400M in ION chipset are used as a benchmark platform, the results could be applied for other GPU's. Exceptions might be latest GeForce GT200 family GPU's; they are equipped with L1 and L2 data cache memories and therefore, different optimization could be applied. Main features of GeForce 8000 or 9000 family GPU's are listed below.

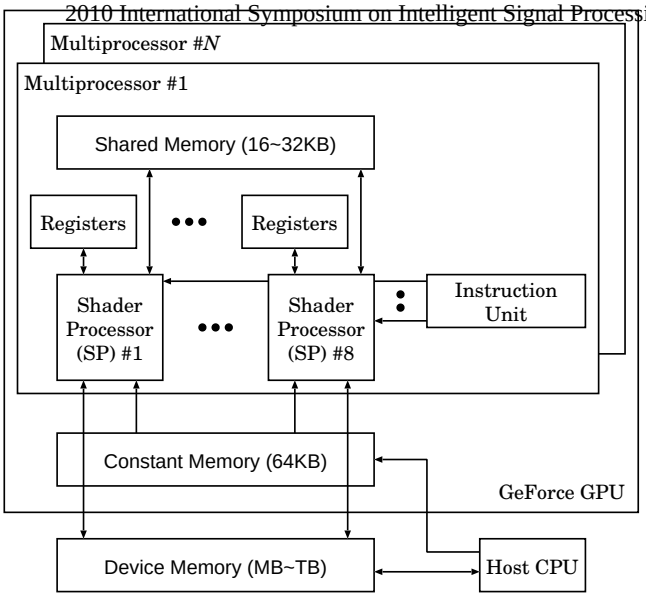

Fig. 1. Computation model of GeForce GPU

- Unified shader architecture
- Large number of shader processors (SP's):
  - $16 \sim 128$ SP's per chip.
  - 8 SP's execute the same instruction.
  - The same instruction are executed in four successive instruction cycles.
  - 32 threads are executed simultaneously by 8-SP block.
  - 8192 data registers per 8 SP's.
- Floating-Point (FP) support
  - 32-bit FP multiply-add.
  - Four-clock latency for 32-bit FP multiply-add.
  - Some newer GPU's support 64-bit FP.
- Multiple data memories
  - Shared memory: 16KB or 32KB read/write RAM per 8 SP's.
    Access latency is 4 instruction cycles.
  - Constant memory: 64KB read-only RAM per chip.
  - Device memory (off-chip RAM): $\sim$ 1GB.
    Very slow: Latency is $400 \sim 600$ clocks.
- Compiler support

As a programmable processor, GeForce GPU's can be regarded as multiple sets of 8-way SIMD (single-instruction multiple-data) processor array. In order to cover a four-cycle latency for most operations, each SP repeats a single instruction by four times. Therefore, a set of 32 threads is executed by a set of 8 SP's. A synchronization mechanism is prepared between threads in a SIMD processor array, while there are no synchronization mechanisms between different SIMD processor arrays.

There are some classes for data memories on GeForce GPU's: shared memory, constant memory, texture memory and device memory. 8 SP's in the same group can access shared memory. Though shared memory is the fastest memory, special care is required for its lifetime. Shared memory is prepared at the beginning of thread and is removed at the end. Users have to save data which will be used after the end of thread into device memory (off-chip memory).

Device memory is a large off-chip memory. The problem of device memory is a very long access latency which is $400 \sim 600$ instruction cycles. To hide this latency, multiple groups of threads are commonly used; another thread starts when a thread is interlocked by slow memory access. Constant memory is an intermediate-speed memory. From GPU, constant memory is a read-only memory, while host CPU can read/write this memory.

"CUDA"[4, 5] is a software development tools and drivers for GeForce family GPU's, which is an abbreviation of "Compute Unified Device Architecture." Programs for both CPU and GPU can be written in a single source file. Some extensions to C/C++ language support parallel processing and multiple memory classes.

4. IMPLEMENTATION OF FIR ADAPTIVE FILTERS BASED ON NLMS ALGORITHM

In this implementation, only one SIMD processor array per filter is used. An implementation with one SIMD array is useful for low-cost GPUs with only two SIMD arrays; one for the adaptive filter and the other for graphics and video. Another reason is to avoid synchronization and communication between multiple SIMD arrays.

4.1. Memory assignment

Since the number of taps is assumed to be very large, vectors $\boldsymbol{x}(n)$ and $\boldsymbol{w}(n)$ will be stored in the device memory. It distinguishes this implementation from that reported in [2]. The input signals and the desired inputs, which are not modified, are stored into constant memory.

4.2. Reduction of memory access

The number of memory access can be reduced by similar manner as in[1, 2]. The data load can be reduced by chang-

ing the order of (1) and (5). Calculating

$$\begin{aligned} w_k(n) &= w_k(n-1) \\ &+ \delta(n-1)x(n-k-1) \end{aligned} \quad (6)$$

and

$$sum(n) = sum(n) + w_k(n)x(n-k) \quad (7)$$

in the descending order of the tap index k could reduce the number of load operation for both $w_k(n)$ and $x(n-k)$. In (6) , $\delta(n-1)$ is defined by

$$\delta(n-1) = \frac{\mu e(n-1)}{|\boldsymbol{x}(n)|^2}. \quad (8)$$

The load operation for $w_k(n)$ is reduced because $w_k(n)$ calculated in (6) is also used for convolution (7) just after (6). The number of load operation for $x(n-k)$ can be reduced because $x(n-k)$ in (7) can be re-used in (6) for the next $k = k-1$.

The number of memory access can further be reduced by introducing a vector data type "float4", which is 4-dimensional (4D) vector. By using this data type, the number of access from/to slow device memory can be reduced by factor of 1/4. Please note that the vector operation can be applied only to memory access. This is because the SP is a scalar processor.

The misalignment problem[8] also appears in this implementation. Thanks to the scalar-processor architecture, the influence of the misalignment is very limited. By using two 4D vectors for $\boldsymbol{x}(n)$ and selecting four samples from the eight, this problem can easily solved.

4.3. Multi-thread implementation of adaptive FIR filter

In this implementation, each adaptive filter is divided simply into short sub filters, which is almost same implementation as in [2]. Figure 2 shows the implementation of adaptive FIR filters. Each thread processes small segments from all of four adaptive filters. This is because the memory access reduction shown in 4.2 requires successive processing of adjacent filter taps. This division also simplifies thread division.

A problem specific to GPU computing is the computational cost for summing all sub filter outputs up. An implementation of an SAEC in [2] has introduced a tree adder in Fig. 2 in order to reduce the summing-up cost. For a 4096-tap FIR filter case, a tree adder is more effective than a single-thread adder if the number of threads is more than 64.

4.4. Multiple Simulations on Multiple SIMD array

Execution of multiple simulations with independent $x(n)$, $d(n)$ and μ have also been examined. Applications of such a parallel processing are parameter optimization and ensemble averaging.

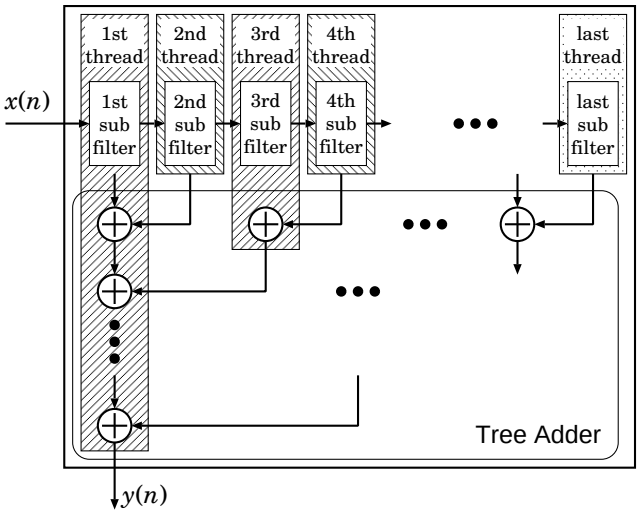

Fig. 2. Multi-thread implementation of adaptive FIR filter

In this configuration, one SIMD array per filter is used. Multiple SIMD array are assigned for parallel processing. Handling of multiple $x(n)$, $d(n)$ and μ are main difference between the single-filter case and the multiple-filter case.

5. PERFORMANCE COMPARISON

The FIR adaptive filter with NLMS algorithm has been implemented and tested on two different platforms. Table 1 depicts the specifications of the platforms. For both CPUs and GPUs, programs in C language is used. The CPU program has been optimized by the compiler. For the GPU programs, the tunable parameters such as the number of thread and the adder type have been manually optimized for the speed. An 4096-tap FIR adaptive filter and a 16kHz sampling are assumed, which is applicable 250msec reverberation time.

Figure 3 demonstrates the influence of the number of threads, the adder type, and the load/store type The processing time by the Core 2 CPU programs and GeForce 8800 GPU programs for 10-second data have been compared. The GPU program with the vector load/store is twice faster than that with the scalar load/store. If the number of threads is larger than 64, the tree adder is faster.

Table 2 compares processing time for different platforms. The fastest parameters have been manually selected. The tree adder and the vector load/store are used for GPU programs. The number of threads is 128 for both GPU's. Core 2 CPU and GeForce 8800 and ION GPUs can be used for real-time processing. In both platforms, GPU's are faster than the corresponding CPU's. The acceleration by GPU is

Table 1. Specifications of Platforms

Platform	Server	Nettop
CPU	Core 2 Duo E8200	Atom N330
Physical cores	2	2
Logical cores	2	4
CPU clock	2.66GHz	1.6GHz
GPU	GeForce 8800 GTS	GeForce 9400M (ION chipset)
SPs	8×16	8×2
SP clock	1.62GHz	1.1GHz
OS (bits)	Linux (64bit)	Linux (64bit)

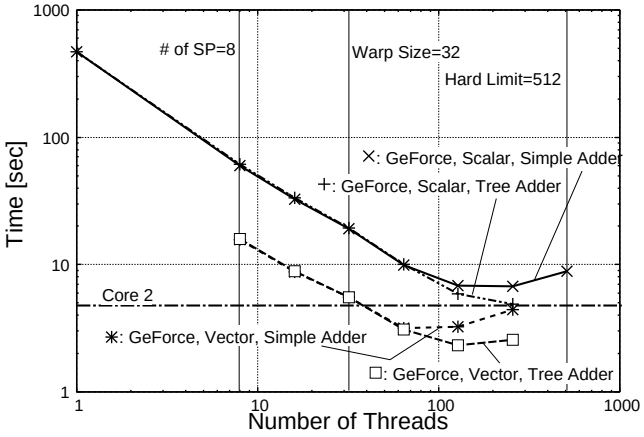

Fig. 3. Influence of number of threads

larger for Atom/ION platform.

The elapsed times for 16 independent simulations for a 4096-tap filter, 16kHz sampling and 10-second data case are compared by Fig. 4. The tree adder and 256 threads have been chosen for GPU programs. GeForce 8800 GPU program with the vector load/store outperforms the other programs. The computation time is well reduced until eight SIMD arrays are used. Almost no performance improvements has been achieved by using sixteen SIMD arrays, possibly because of the limitation on the memory bandwidths.

6. CONCLUSION

Efficient implementations of a large-scale FIR adaptive filter on nVIDIA GeForce GPU's have been discussed. Reduction of memory accesses including vector load/store and multi-thread code help to overcome the influence of slow off-chip memory access. A tree adder is introduced to reduce the cost for summing thread outputs up. Two or three times acceleration for a single-filter case and six times acceleration for a 16-filter case have been achieved by GPU computing.

Table 2. Computation time for 4096-tap, 10 seconds data

Type	Core 2	GeForce 8800	Atom	ION
Time [sec]	4.75	2.32	19.60	6.51

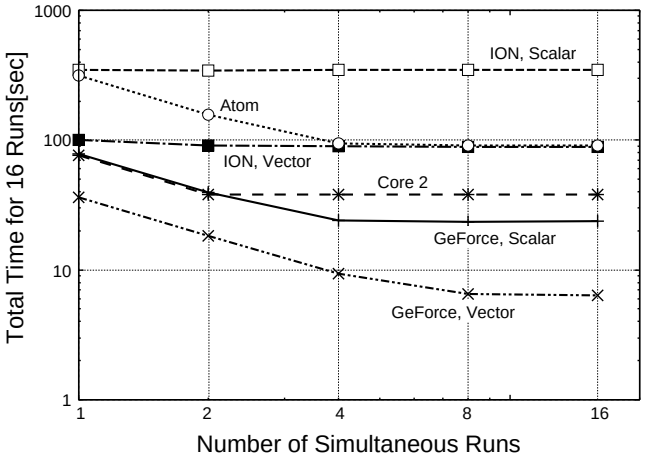

Fig. 4. Performance for multiple simulations

7. REFERENCES

[1] A. Hirano and K. Nakayama, "Implementation of stereophonic acoustic echo canceller on intel IA-32 processors with SIMD capability," *Proc. of 22nd SIP symposium*, Nov. 2007.

[2] A. Hirano and K. Nakayama, "Implementation of stereophonic acoustic echo canceller on nvidia geforce graphics processing unit," *Proc. of ISPACS 2009*, pp. 303–306, Dec. 2009.

[3] S. Koike, "Performance analysis of least mean modulus-newton algorithm," *Proc. of ISPACS 2009*, pp. 413–414, Doc. 2009.

[4] "NVIDIA CUDA compute unified device architecture reference manual," Nov. 2008.

[5] "NVIDIA CUDA programming guide," Dec. 2008.

[6] "ATI stream computing user guide," Mar 2009.

[7] J. Nagumo and A. Noda, "A learning method for system identification," *IEEE Trans. AC*, vol. 12, no. 3, pp. 282–287, Mar. 1967.

[8] B. Juurlink A. Shahbahrami and S. Vassiliadis, "Performance impact of misaligned accesses in SIMD extensions," *Proc. of ProRISC 2006*, pp. 334–342, 2006.