Parallel Simulation of FIR Adaptive Filters on nVIDIA GeForce Graphics Processing Unit

Akihiro HIRANO Kenji NAKAYAMA
Kanazawa University

Abstract This paper discusses a fast execution of multiple simulations for an FIR adaptive filter on nVIDIA GeForce family GPU with an easy programming. For simplicity, only one shader processor per filter is used. In order to overcome a long latency of off-chip memory access, multi-word memory accesses, software-based data caches, and address assignment in the same order as the memory accesses are introduced. Dedicated functions for memory accesses hide the complicated code for acceleration from programmers. The performance comparisons shows that GeForce 8800 GPU is almost four times faster than Core 2 CPU. Even the Atom/ION platform is almost as fast as Core 2 CPU.

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]–[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 order to exploit the performance of GPU's for signal processing, computationally efficient implementations of adaptive filters on GPU's have been reported[2], [7]. Adaptive filters on GPU's outperforms those on CPU's especially for a multiple simulation case[7]. However, a source code of an efficient program for GPU's might become complicated: it should support multi-thread execution, tree adders, and vector load/store operations. Such a complicated programming might be a barrier for GPU computing.

This paper discusses a fast execution of multiple simulations for an FIR adaptive filter on nVIDIA GeForce family GPU with an easy programming. Section 2 describes the FIR adaptive filter with the normalized least mean squares (NLMS) algorithm[8]. GeForce family GPU and "CUDA" software development environment are 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[8], 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[9] 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 Fermi family GPU's[10]; 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.

- Unified shader architecture
- Large number of shader processors (SP's):
  - $16 \sim 128$ SP's per chip.
  - 8 SP's execute the same instruction.

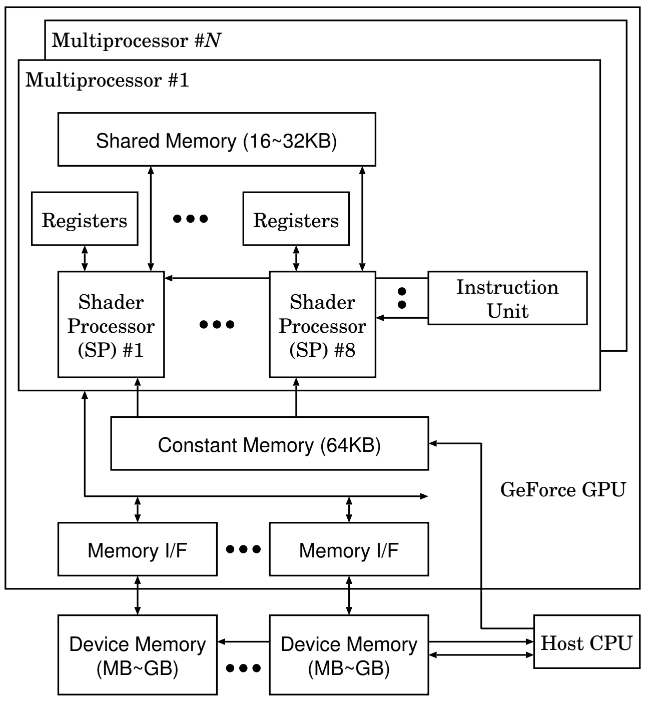


Figure 1: Computation model of GeForce GPU

  - 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 SP per filter is used for simplicity. This implementation focuses on

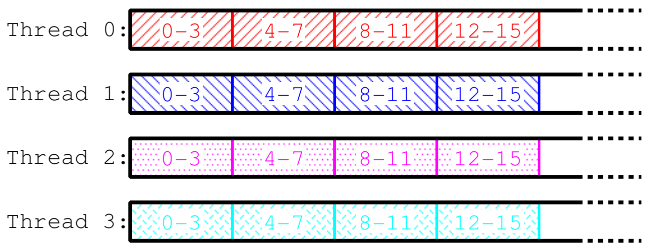
1. Multi-word access[2] and cache
2. Address assignment in the same order as memory access
3. Memory assignment
4. Multiple simlations.

In order to hide a complex program code for 1 and 2, memory accesses for the filter coefficients and the input signals $x(n)$ are performed via specified functions.

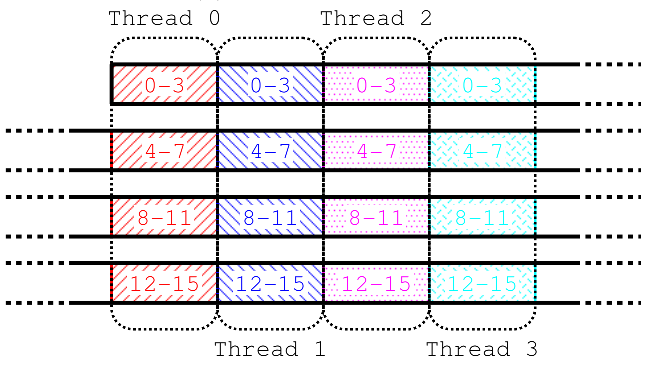
4.1 Multi-word access and cache

The number of memory read operations from the slow device memory can be reduced by reading multiple data simultaneously. Using four-word load/store operations, twice faster computation speed than that for a scalar program can be achieved[7]. The drawback of this technique is a complicated programming for both vector operations on a scalar processor and data accesses avoiding the misalignment problem[11].

In this implementation, a simple cache mechanism using the fast shared memory is introduced. For simple programming, the multi-word memory access and the cache operations are hidden in load/store functions. Only one cache entry for the filter coefficients and also one for the input signals are prepared. For LMS-family



(a) Simple assignment



(b) Same order as memory access

Figure 2: Address assignments

adaptive filters, single-entry cache memories achieves hight cache-hit rate because of their simple sequential memory access. The effect os the cache size will be examined later.

4.2 Address assignment in same order as memory access

The number of memory access operations can further be reduced by changing the data address assignments for the filter coefficients $w_i(n)$ and the delay line $x(n-i)$. The data addresses are assigned in the same order as the memory accesses. For multiple simulations of the same-order adaptive FIR filters, such assignment is simply achieved by gathering data with the same index i for multiple threads into successive data address. Exceptions would be sparse-tap adaptive FIR filters[12], [13] in which the order of memory access depends both on the input signals and also on the adaptation parameters.

Figure 2 (a) shows a simple address assignment. In this figure, "0-3" means $w_0(n)$ through $w_3(n)$ for the filter coefficients, or, $x(n)$ through $x(n-3)$ for the delay line. Data for each thread is stored in a memory block, followed by data for the next block. In this manner, multiple data transfers should be required only for one-word data, e.g. $w_0(n)$. This is because multiple threads are simultaneously executed in a SIMD array. Accesses for multiple $w_0(n)$, one per thread, will occur at a time. Since these data might be stored in separate data address, a large number of data transfers should be required.

Such multiple memory access operations can be gathered by changing the data address based on the access order. An address assignment for a four-word access

```
tx = ...;  /* Thread Index */
dp = ...;  /* Delay Pointer */
cp = ...;  /* Coefficient Pointer */
out = 0;

for (tap = 0; tap < TAPNUM;
     dp--, cp--, tap++) {

  /* Read coefficient */
  c = ReadCoef(tx, coef, cp);

  /* Update coefficient */
  c += delta * in;

  /* Read new input */
  in = ReadDelay(tx, delay, dp);

  /* Convolution */
  out += c * in;

  /* Store coefficient */
  StoreCoef(tx, coef, cp, c);
}
```

Figure 3: Example of source code for LMS adaptive filter

and four-thread case is demonstrated by Fig. 2 (b). Four threads will request data access for their $w_0(n)$ through $w_3(n)$, which result in sixteen-word access in total. These sixteen-word data are stored into successive data address. Therefore, a smaller number of block transfer might sufficient compared with assignment in Fig. 2 (a).

4.3 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 by GPU, are stored into constant memory.

4.4 Code example

Figure 3 is an example of implemented source program for (1) and (5) of LMS-family algorithms. This program calculates

$$w_k(n) = w_k(n-1) + \delta(n-1)x(n-k-1) \qquad (6)$$

and

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

in the descending order of the tap index k. In (6) , $\delta(n-1)$ is defined by

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

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)

Table 2: Computation time per filter for multple GPU programs

Type	Simple	With cache	Cache + assign
Time [sec]	5.830	2.927	1.152

Table 3: Optimum computation time per filter for 4096-tap, 10 seconds data

Type	Core 2	GeForce 8800	Atom	ION
Time [sec]	2.370	0.584	5.664	2.655

The number of memory access can be reduced by data reuse[1], [2]. In order to hide the complicated code for acceleration from programmers, memory accesses are performed by dedicated functions.

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 CPU's and GPU's, 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 cache size 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. Simulations for multiple FIR adaptive filters with different step-size have been performed simultaneously. All filters uses the same input signals. The processing time per filter for 10-second data have been compared.

The effects of the optimizing techniques are compared by Tab. 2. 512 simulations have been carried out simultaneously by using 64 threads $\times$ eight SIMD arrays. The cache size is eight-word for GPU program with multi-word access and cache (With cache), while four-word is used for program with both cache and address assignment(Cache + assign). The cache enhances the speed by almost two times. Further three times improvements has been achieved by the address assignment optimization.

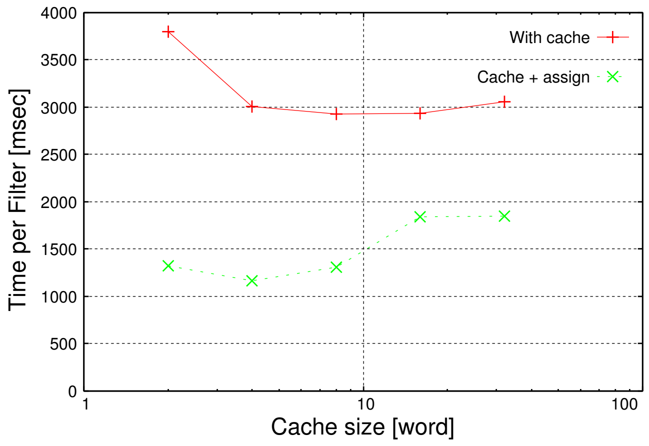


Figure 4: Computation time vs. cache size

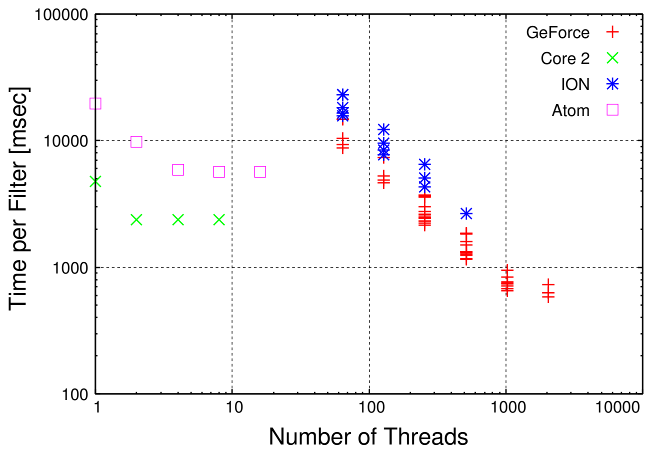


Figure 5: Computation time vs. number of filters

The influence of the cache size is depicted by Fig. 4. 512 simulations have been carried out simultaneously by using 32 threads $\times$ sixteen SIMD arrays. The optimum cache size for most configurations is four words. Though eight-word cache is better for some configurations, the difference between the four-word cache and the eight-word cache is small.

Table 3 compares processing time per filter for different platforms. The fastest parameters have been manually selected. The GPU programs use multi-word access, cache and address assignment. In both platforms, GPU's are faster than the corresponding CPU's: almost four times faster for GeForce 8800 GPU. The Atom/ION platform is almost as fast as Core 2 CPU.

Figure 5 compares processing time per filter for different paramters. The results of GPU's are plotted for multiple combination of the tunable parameters: the number of SIMD arrays, the number of threads per SIMD array, and the cache size. Though the performance of GPU program depends on tunable parameters, the total number of threads is the most important factor.

6 Conclusion

A fast execution of multiple simulations for an FIR adaptive filter on nVIDIA GeForce family GPU with an easy programming has been discussed. Multi-word memory accesses, software-based data caches, and address assignment in the same order as the memory accesses overcome a long latency of off-chip memory access. In order to hide the complicated code for acceleration from programmers, memory accesses are performed by dedicated functions. A simple user program achieves almost four times faster simulations by GeForce 8800 GPU than Core 2 CPU.

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, Dec. 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] A. Hirano and K. Nakayama, "Implementation of large-scale fir adaptive filters on nvidia geforce graphics processing unit," *to be presented at ISPACS 2010*, Dec. 2010.

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

[9] "NVIDIA FeForce 8800 GPU architecture overview," Nov. 2006.

[10] "Tuning CUDA applications for Fermi," Apr. 2010.

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

[12] S. Kawamura and M. Hatori, "A tap selection algorithm for adaptive filters," *Proc. of ICASSP '86*, pp. 2979–2982, 1986.

[13] S. Ikeda and A. Sugiyama, "A fast convergence algorithm for sparse-tap adaptive fir filters for an unknown number of multiple echoes," *Proc. of ICASSP '94*, pp. 41–44, 1994.