Efficient Implementation of RLS-Based Adaptive Filterson nVIDIA GeForce Graphics Processing Unit

Efficient Implementation of RLS-Based Adaptive Filters on nVIDIA GeForce Graphics Processing Unit

Akihiro HIRANO Kenji Nakayama
Kanazawa University

Abstract This paper presents efficient implementation of RLS-based adaptive filters with a large number of taps on nVIDIA GeForce graphics processing unit (GPU) and CUDA software development environment. Modification of the order and the combination of calculations reduces the number of accesses to slow off-chip memory. Assigning tasks into multiple threads also takes memory access order into account. Multiple shader processor arrays are used to handle a large matrix. For a 8192-tap case, a GPU program is almost 30-times faster than a CPU program. Real-time processing is possible for an 8kHz-sampling and 512-tap case by using 32 shader processors, which is only 25% of GeForce 8800GTS.

1 Introduction

Echo cancellers are used to reduce echoes in a wide range of applications, such as teleconference systems and hands-free telephones. As adaptation algorithms used in echo cancellers, least mean square (LMS) family algorithms[1], [2] are widely used because of their low computational complexity. However, the convergence speed of the LMS algorithms is slow for colored signals such as speech signals.

As a candidate of a fast convergence algorithm, a recursive least squares (RLS) algorithm[3] is well known. The drawback of the RLS algorithm is its huge amount of computation which is proportional to the square of the filter length. For acoustic echo cancellers (AEC's), the number of taps is from several hundreds to several thousands. Therefore, using the RLS algorithm in real-time AEC's is extremely difficult.

Recent years, PC-based communication systems such as Skype and Messenger becomes very popular. 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. Therefore, audio/speech processing on GPU's has been studied for AEC's[7]–[11] and independent component analysis (ICA)[12].

GeForce GPU by nVIDIA used in previously reported AEC implementation consists of multiple single-instruction multiple-data (SIMD) processor arrays. However, only one SIMD array per filter is used these AEC's [8]–[11]. A reason is a difficulty in synchronization between SIMD arrays. For implementation of RLS algorithm, using multiple SIMD arrays is considered because of its heavy computations.

In this paper, computationally efficient implementation of adaptive filters with the RLS algorithm on nVIDIA GeForce family GPU and CUDA is discussed. Section 2 describes the adaptive filter with the RLS algorithm. 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 Adaptive Filter Based on RLS Algorithm

From the filter coefficient vector $\boldsymbol{w}(n)$ and the input signal vector $\boldsymbol{u}(n)$ at the time index n, the filter output $y(n)$ is generated by

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

The superscript ^{T} denotes the transpose of a matrix or a vector. The error signal $e(n)$ between the desired response $d(n)$ and the filter output $y(n)$ is calculated by

$$e(n) = d(n) - y(n). \tag{2}$$

Using the inverse correlation matrix $\boldsymbol{P}(n)$, the gain vector $\boldsymbol{k}(n)$ is given by

$$\boldsymbol{k}(n) = \frac{\lambda^{-1}\boldsymbol{P}(n-1)\boldsymbol{u}(n)}{1+\lambda^{-1}\boldsymbol{u}^T(n)\boldsymbol{P}(n-1)\boldsymbol{u}(n)}. \tag{3}$$

The filter coefficients $\boldsymbol{w}(n)$ is updated by

$$\boldsymbol{w}(n) = \boldsymbol{w}(n-1) + \boldsymbol{k}(n)e(n), \tag{4}$$

followed by the update of $\boldsymbol{P}(n)$ by

$$\boldsymbol{P}(n) = \lambda^{-1}\boldsymbol{P}(n-1) - \lambda^{-1}\boldsymbol{k}(n)\boldsymbol{u}^T(n)\boldsymbol{P}(n-1). \tag{5}$$

By introducing a vector $\boldsymbol{v}(n)$ defined by

$$\boldsymbol{v}(n) = \boldsymbol{P}(n-1)\boldsymbol{u}(n), \tag{6}$$

equations (3) and (5) can be rewritten as

$$\boldsymbol{k}(n) = \frac{\lambda^{-1}\boldsymbol{v}(n)}{1+\lambda^{-1}\boldsymbol{u}^T(n)\boldsymbol{v}(n)} \tag{7}$$

$$\boldsymbol{P}(n) = \lambda^{-1}\boldsymbol{P}(n-1) - \lambda^{-1}\boldsymbol{k}(n)\boldsymbol{v}^T(n). \tag{8}$$

For N_{tap}-tap case, computations for (6) and (8) require N_{tap}^2-order computations.

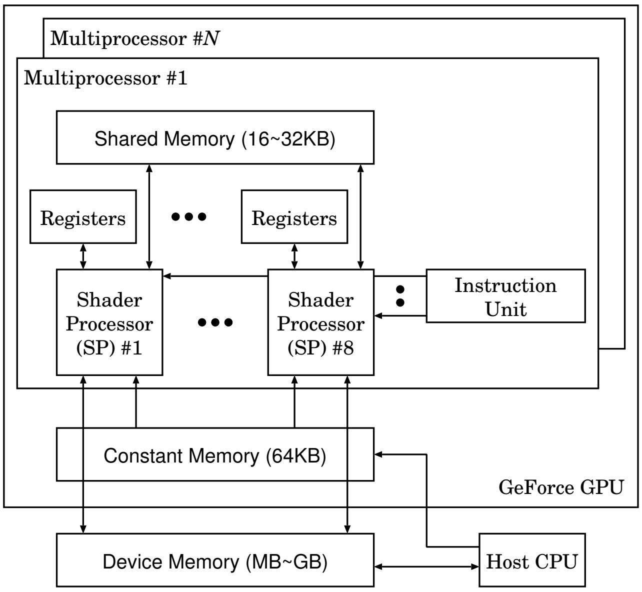

Figure 1: Computation model for Tesla architecture

3 nVIDIA GeForce GPU and CUDA

In this implementation, nVIDIA GeForce family GPU's are assumed. Both Tesla architecture (GeForce 8000 through GT 300) and Fermi architecture (GT 400 through GT 500) are used as benchmark platforms. For the latest Kepler architecture (GT 600 or later), different optimizations would be necessary.

Figure 1 shows the computation model for Tesla architecture. Main features of GeForce GPU's with Tesla architecture are listed below.

- Unified shader architecture
- Large number of shader processors (SP's):
  - 16 ∼ 480 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 or more 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): ∼ 1GB.
    Very slow: Latency is 400 ∼ 600 clocks.
- Compiler support

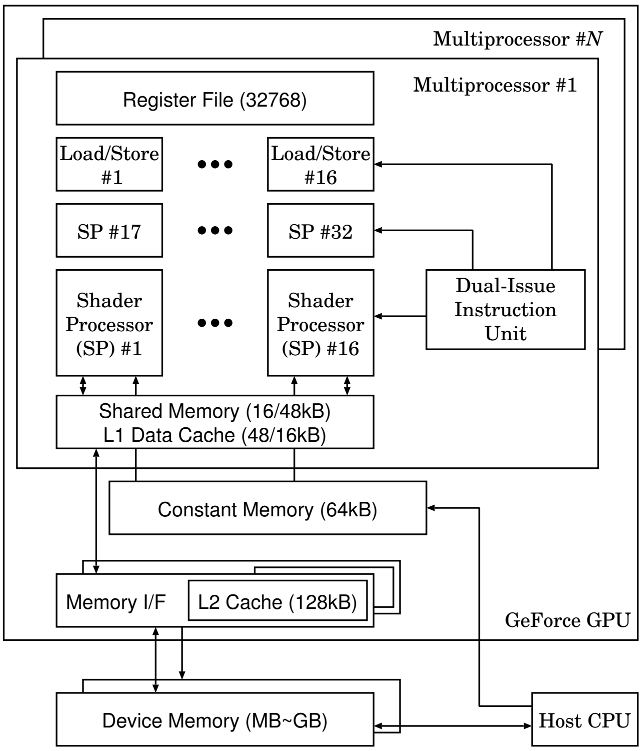

Figure 2: Computation model for Fermi architecture

As a programmable processor, GPU's with Tesla architecture can be regarded as multiple sets of 8-way SIMD processor arrays. 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 ∼ 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.

The computation model for Fermi architecture is depicted in Fig. 2. Major differences between two architectures are listed below.

- Shader multiprocessors (SM's)
  - SM consists of 32 or 48 SP's and 16 load/store units.
  - 16 SP's execute the same instruction.
  - Dual- ore tripple-issue instruction unit executes two or three independent sets of threads simultaneously.
  - Improved performance for 64-bit FP.
  - Configurable 64kB data memory for shared memory and L1 data cache.
- Data cache for GPU computing
  - 16 or 48kB L1 data cache per SM.
  - 128kB L2 data cache per memory interface.

An important change would be the introduction of the data cache for GPU computing. This change would affect the optimization for slow device 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 Adaptive Filters Based on RLS Algorithm

4.1 Reduction of memory accesses for matrix $\boldsymbol{P}(n)$

In order to reduce the number of the memory accesses for the matrix $\boldsymbol{P}(n)$, the computation order of the equations (1) through (8) is modified as shown below;

$$y(n) = \boldsymbol{w}^T(n)\boldsymbol{u}(n) \tag{9}$$

$$e(n) = d(n) - y(n) \tag{10}$$

$$\boldsymbol{P}(n-1) = \lambda^{-1}\boldsymbol{P}(n-2) - \lambda^{-1}\boldsymbol{k}(n-1)\boldsymbol{v}^T(n-1) \tag{11}$$

$$\boldsymbol{v}(n) = \boldsymbol{P}(n-1)\boldsymbol{u}(n) \tag{12}$$

$$\boldsymbol{k}(n) = \frac{\lambda^{-1}\boldsymbol{v}(n)}{1 + \lambda^{-1}\boldsymbol{u}^T(n)\boldsymbol{v}(n)} \tag{13}$$

$$\boldsymbol{w}(n) = \boldsymbol{w}(n-1) + \boldsymbol{k}(n)e(n). \tag{14}$$

The calculations in (11) and (12) are further combined. The matrix $\boldsymbol{P}(n)$ is divided into a set of raw vectors as

$$\boldsymbol{P}(n) = \begin{bmatrix} \boldsymbol{p}_1(n) \\ \vdots \\ \boldsymbol{p}_N(n) \end{bmatrix}, \tag{15}$$

where $\boldsymbol{p}_i(n)$ is an i-th row vector of $\boldsymbol{P}(n)$. Computations in equations (11) and (12) can be performed by repeating the following two equations for $i = 1, \cdots, N$:

$$\boldsymbol{p}_i(n-1) = \lambda^{-1}\boldsymbol{p}_i(n-2) - \lambda^{-1}k_i(n-1)\boldsymbol{v}^T(n-1) \tag{16}$$

$$v_i(n) = \boldsymbol{p}_i(n-1)\boldsymbol{u}(n) \tag{17}$$

where $k_i(n)$ and $v_i(n)$ are the i-th element of vectors $\boldsymbol{k}(n)$ and $\boldsymbol{v}(n)$, respectively. In this manner, the number of the memory accesses for $\boldsymbol{P}(n)$ can be minimized; only one read and one write per element. Please note that a double-buffer operation is necessary for $\boldsymbol{v}(n)$.

4.2 Coping with slow off-chip memory

Multiple techniques are required for avoiding the performance degradation caused by the slow off-chip memory. Vector load/store operations reduce the number of memory accesses. Techniques avoiding the misalignment problem[13] caused by vector load/sore operations are required for the input signal vector $\boldsymbol{u}(n)$. The delay line in the off-chip memory uses a multiple-delay-line approach. The number of the delay lines is same as the vector load/store size.

In order to combine multiple accesses for the off-chip memory into one, the i-th thread handles the $(i + j \times N_{th})$-th elements where N_{th} is the number of threads and $j = 0, 1, \cdots, N_{tap}/N_{th}$. This assignment results in the successive memory accesses to the successive addresses. The memory controller will combine these memory accesses into a multi-word read/write operation for the SDRAM. In [10], the same effect is achieved by a different way. It changes the data address assignmens.

4.3 Task assignments for multiple SIMD array

In this implementation, calculations for (11) and (12), which requires N_{tap}^2-order computations, are divided into multiple SIMD arrays. This division is based on $\boldsymbol{p}_i(n)$. For an N_{sm} SIMD arrays case, $\boldsymbol{p}_1(n)$ through $\boldsymbol{p}_{N_{tap}/N_{sm}}$ are handled by first SIMD array.

Other operations are not divided and are executed by a single SIMD array. This is because synchronization between multiple SIMD arrays through the slow off-chip memory degrades the performance.

4.4 Implementation for a small number of taps

If the number of taps N_{tap} is small, the vectors $\boldsymbol{w}(n)$, $\boldsymbol{u}(n)$, $\boldsymbol{v}(n)$ and $\boldsymbol{k}(n)$ can be stored into the shared memory. The matrix $\boldsymbol{P}(n)$ is stored into the off-chip memory because of its N_{tap}^2 size. An exception would be a very small N_{tap} such as 32. Figure 3 depics memory allocation.

In order to avoid data sharing via the slow off-chip memory, each SIMD array holds whole vectors rather than holding a partial vectors. Furthermore, all SIMD arrays perform all calculations for (9), (10), (13) and (14), which cause almost no performance penalties.

In a first sample of the signal block, the vectors $\boldsymbol{w}(n)$, $\boldsymbol{v}(n)$ and $\boldsymbol{k}(n)$ are read from the off-chip memory and

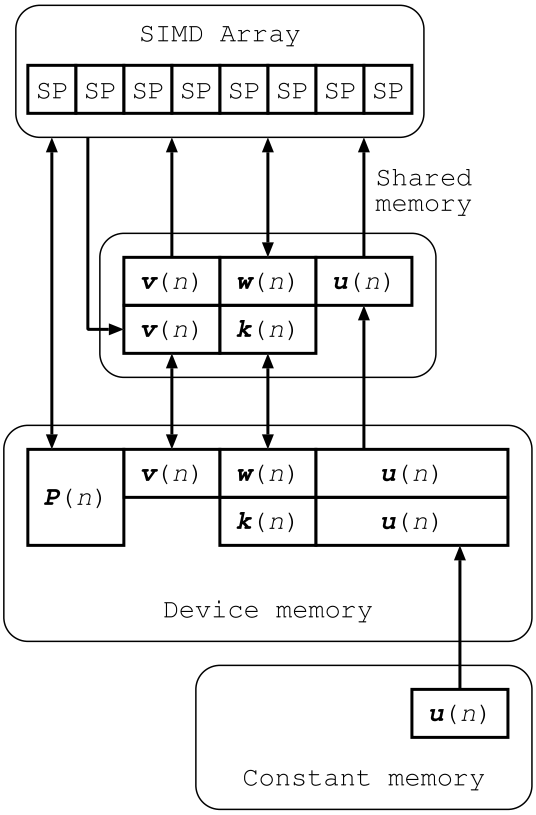
SIMD Array
SP SP SP SP SP SP SP SP
Shared memory
$\boldsymbol{v}(n)$ $\boldsymbol{w}(n)$ $\boldsymbol{u}(n)$
$\boldsymbol{v}(n)$ $\boldsymbol{k}(n)$
$\boldsymbol{P}(n)$ $\boldsymbol{v}(n)$ $\boldsymbol{w}(n)$ $\boldsymbol{u}(n)$
$\boldsymbol{k}(n)$ $\boldsymbol{u}(n)$
Device memory
$\boldsymbol{u}(n)$
Constant memory

Figure 3: Memory assignments for small filter

written into the shared memory. These vectors are written back to the off-chip memory in the last sample of the signal block. This write-back operation should be carried out by only one SIMD array. In order to reduce the data size, N_{tap}-th order vector is prepared in the shared memory as a cache. The input signals are stored in the cache in (9). The other operations read $\boldsymbol{u}(n)$ from the cache.

5 Performance Comparison

The FIR adaptive filters with the RLS algorithm have been implemented and tested. GPU-based RLS programs with a single multiprocessor (MP, i.e. SIMD array) [11] are also compared. Table 1 depicts the specifications of the platforms. For all 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 threads have been manually optimized for the speed. The computation time for 16000-sample signals have been compared. The CPU time less than two seconds means realtime processing for an 8kHz-sampling case.

Figure 4 compares the computation time in seconds. For large-scale filters over 200 taps, all GPU programs are faster than CPU programs. For a 4096-tap case, GeForce GTS 450 is 30-times faster than Core i5 CPU.

Table 1: Specifications of Platform

CPU	Core 2 Duo E8200	Core i5 2405S
Physical cores	2	4
Logical cores	2	4
CPU clock	2.66GHz	2.5GHz
GPU	GeForce 8800 GTS	GeForce GTS 450
SPs	$128 = 8 \times 16$	$192 = (16 \times 3) \times 4$
SP clock	1.62GHz	1.56GHz
OS (bits)	Linux (64bit)	Linux (64bit)

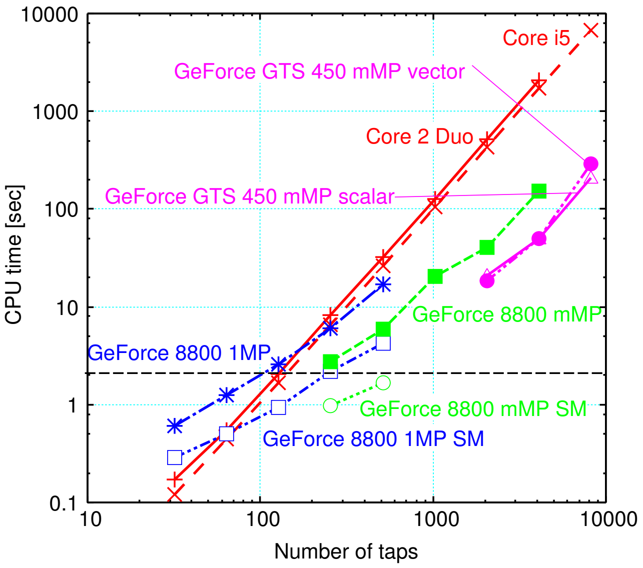

Figure 4: Computation time

In order to examine the effects of techniques, four types of programs for GeForce 8800 GTS have been compared. The program “GeForce 8800 mMP,” which uses multiple MP's and does not store vectors into the shared memory (SM), is as fast as the program “GeForce 8800 1MP SM,” which uses a single MP and stores the vectors into the SM. By using multiple SIMD arrays, the performance degradation caused by slow off-chip memory can be reduced. The optimization without storing vectors into the SM is applicable for larger number of taps such as 4096.

The program “GeForce 8800 mMP SM” stores the vectors into the SM. By using the SM, the computation speed becomes almost three times faster. However, the number of taps is limited by the memory size as 512-tap. For an 8kHz-sampling and a 512-tap case, “GeForce 8800 mMP SM” program is capable of real-time processing. This program uses four SIMD arrays or 32 SP's, which is only 25% of GeForce 8800 GTS GPU. Please note that 32-SP is only an “entry class” products in recent GeForce family.

The performance of GeForce GTS 450 and 8800 GTS are also compared for large-scale filters. Though the clock speed is comparable, GeForce GTS 450 (Fermi) is almost twice as fast as GeForce 8800 GTS (Tesla). A possible reason is the structure of SIMD array. The Fermi architecture is 16-way SIMD while the Tesla architecture is 8-way SIMD. Another reason might be the data cache.

The effect of the data cache has been examined by programs with and without the vector load/store operations. The vector load/store operation improves performance for GPU's without data cache[11]. On the other hand, he vector load/store operations seems to have almost no improvements for GeForce GTS 450 with data cache.

6 Conclusion

RLS-based adaptive filters with a large number of taps has been implemented on nVIDIA GeForce GPU's. In order to reduce accesses to slow off-chip memory, the order and the combination of calculations has been modified. The matrix operations are handled by multiple SIMD arrays, while the vector operations are not divided. Task asignment to multiple threads takes memory access order into account. The GPU program is up to 30 times faster than the CPU program. For an 8kHz-sampling and 512-tap case, Only 25% of SP's in GeForce 8800GTS GPU is capable of real-time processing.

References

[1] B. Widrow and S. D. Stearns, "Adaptive noise canceling: Principles and applications," *Proc. of IEEE*, vol. 63, no. 12, pp. 1692–1716, Dec. 1975.

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

[3] S. Haykin, *Adatptive Filter Theory, Third Edition*, Prentice Hall, 1996.

[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 stereophonic acoustic echo canceller on intel IA-32 processors with SIMD capability," *Proc. of 22nd SIP symposium*, Nov. 2007.

[8] 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.

[9] A. Hirano and K. Nakayama, "Implementation of large-scale FIR adaptive filters on nVIDIA GeForce graphics processing unit," *Proc. of ISPACS 2010*, pp. 269–272, Dec. 2010.

[10] A. Hirano and K. Nakayama, "Parallel simulation of FIR adaptive filters on nVIDIA GeForce graphics processing unit," *Proc. of 25th SIP Symposium*, pp. 98–102, Nov. 2010.

[11] A. Hirano and K. Nakayama, "Implementation of RLS-based adaptive filters on nVIDIA GeForce graphics processing unit," *Proc. of 26th SIP Symposium*, pp. 477–481, Nov. 2011.

[12] R. Mazur and A. Mertins, "A CUDA implementation of independent component analysis in the time-frequency domain," *Proc. of 19th EUSIPCO*, pp. 511–514, Aug. 2011.

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