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. For a 4096-tap case, a GPU program is almost three times faster than a CPU program.

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]–[9] and independent component analysis (ICA)[10].

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

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

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

$$e(n) = d(n) - y(n). \quad (2)$$

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

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

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

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

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

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

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

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

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

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

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

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

3 nVIDIA GeForce GPU and CUDA

In this implementation, nVIDIA GeForce 8000 family or later GPU's are assumed. Though GeForce 8800 GTS is used as a benchmark platform, the results could be applied for other GPU's. Exceptions might be latest GeForce GT400 family or later GPU's; they are equipped with L1 and L2 data cache memories and therefore, different optimization could be applied. Main features of GeForce 8000 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.

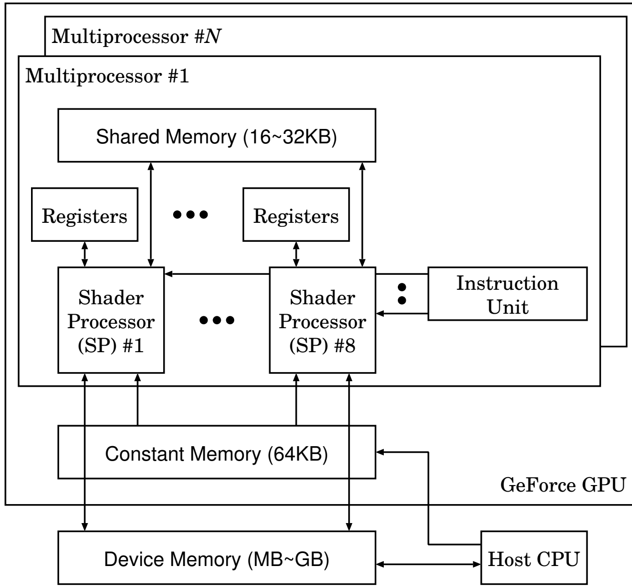


Figure 1: Computation model of GeForce GPU

- 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 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 Adaptive Filters Based on RLS Algorithm

In this implementation, only one SIMD processor array 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 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 $i = 1, \cdots, N$:

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

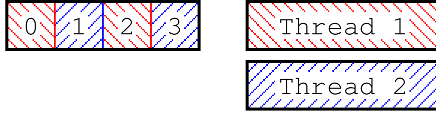
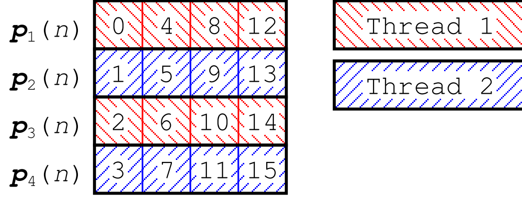
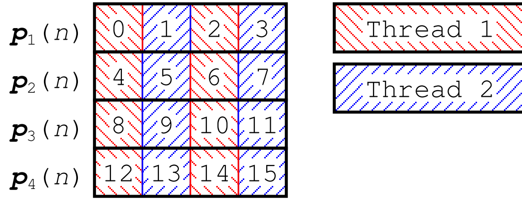


Figure 2: Thread assignments for vectors



(a) Thread per vector



(b) Thread per segment

Figure 3: Thread assignments for matrix

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

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[11] 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 $j = 0, 1, \cdots, N_{tap}/N_{th}$. Figure 2 demonstrates a two-thread and four-dimensional vector case. 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 [9], the same effect is achieved by a different way. It changes the data address assignments.

4.3 Task assignments for multiple threads

For vector operations, each vectors are divided into multiple segments as in Fig. 2. For a scalar product operation, a tree adder is used to accumulate the segment outputs, which is similar to the LMS case[8].

Two task assignments for calculations of (11) and (12) have been compared in this implementation. Though GeForce GPU can handle up to three-dimensional thread and the RLS algorithm requires N_{tap}^2-order computations, only one-dimensional thread is used. First assignment is based on $\boldsymbol{p}_i(n)$. Each thread handles whole vector $\boldsymbol{p}_i(n)$. Figure 3 (a) depicts a two-thread and 4×4-matrix case. The data address is assigned "row-first" basis so that combining accesses as in 4.2 can be used.

Second assignment shown in Fig. 3 (b) divides $\boldsymbol{p}_i(n)$ into multiple segments. In this case, the data address is assigned "column-first" basis. The efficiency of these assignments will be compared in the performance comparison.

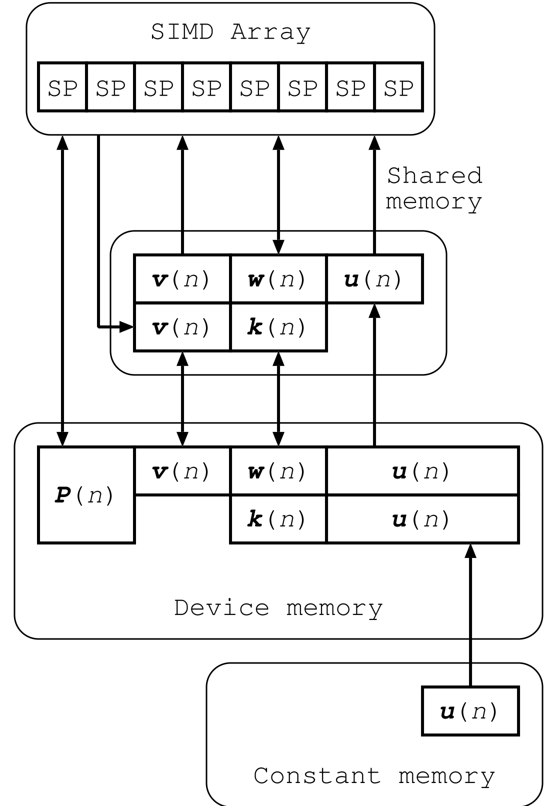


Figure 4: Memory assignments for small filter

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. Figure 4 shows the memory assignments for this case. 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.

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 (device memory) and written into the shared memory. These vectors are written back to the off-chip memory in the last sample of the signal block.

Table 1: Specifications of Platform

CPU	Core 2 Duo E8200
Physical cores	2
Logical cores	2
CPU clock	2.66GHz
GPU	GeForce 8800 GTS
SPs	8 × 16
SP clock	1.62GHz
OS (bits)	Linux (64bit)

Table 2: Specifications of GPU Programs

Name	GPU1	GPU2	GPU3	GPU4
Vectors	Ext.	Ext.	Int.	Int.
$\boldsymbol{P}(n)$	(a)	(b)	(a)	(b)

In the beginning of a signal block, the host CPU stores the signals $u(n)$ and $d(n)$ into the constant memory. The GPU copies $u(n)$ to multiple delay lines in the device memory. Larger-size vectors are used for simplification of the circular buffer operations. 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. Table 1 depicts the specifications of the platform. For both CPU and GPU, 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 1600-sample signals have been compared. The CPU time less than two seconds means real-time processing for an 8kHz sampling case.

Table 2 shows the combination of the techniques. Programs "GPU3" and "GPU4" store the vectors into the shared memory, while "GPU1" and "GPU2" stores all vectors and matrix into the off-chip memory. The matrix $\boldsymbol{P}(n)$ is handled "thread per vector" basis in "GPU1" and "GPU3."

Figure 5 compares the computation time in seconds. For large number of taps over 128, all GPU programs are faster than the CPU program. For 4096-tap case, "GPU2" program reduces the CPU time by almost 67%. By using the shared memory, the computation speed becomes up to three times faster. However, the number of taps is limited by the memory size. "GPU2" program is faster than "GPU1" program if the number of taps is 2048 or above.

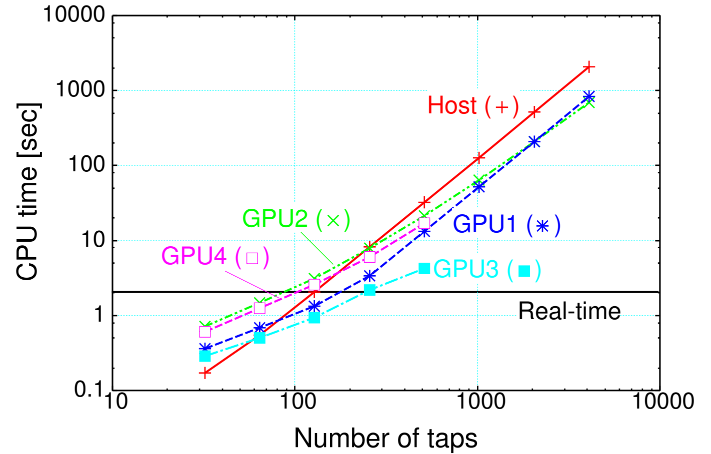


Figure 5: Performance comparison

For an 8kHz-sampling and a 256-tap case, "GPU3" program is slightly slower for real-time processing. This program uses only one SIMD array or 8 SP's, which is a minimum configuration of GeForce 8000 family or later.

6 Conclusion

RLS-based adaptive filters with a large number of taps has been implemented on nVIDIA GeForce GPU. This implementation focuses on a single SIMD array case. In order to reduce accesses to slow off-chip memory, the order and the combination of calculations has been modified. Task assignment to multiple threads takes memory access order into account. If the number of taps is 256 or more, the GPU programs reduces the computation time up to 67% compared with the CPU program.

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 nvidia geforce graphics processing unit," *Proc. of ISPACS 2009*, pp. 303–306, Dec. 2009.

[8] 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.

[9] 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.

[10] 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.

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