Implementation of stereophonic acoustic echo canceller on nVIDIA GeForce graphics processing unit

著者	Hirano Akihiro, Nakayama Kenji
journal or publication title	ISPACS 2009 - 2009 International Symposium on Intelligent Signal Processing and Communication Systems, Proceedings
number	5383842
page range	303-306
year	2009-01-01
URL	http://hdl.handle.net/2297/24447

Implementation of Stereophonic Acoustic Echo Canceller on nVIDIA GeForce Graphics Processing Unit

Akihiro Hirano and Kenji Nakayama
Kanazawa University, Japan
E-mail: {hirano,nakayama}@t.kanazawa-u.ac.jp Tel: +81-76-234-4897

***Abstract*—This paper presents an implementation of a stereophonic acoustic echo canceller on nVIDIA GeForce graphics processor and CUDA software development environment. For efficiency, fast shared memory has been used as much as possilbe. A tree adder is introduced to reduce the cost for summing thread outputs up. The performance evaluation results suggest that Even a low-cost GPU's with a small number of shader processor greatly helps the echo cancellation for low-cost PC-based teleconferencing.**

I. Introduction

Echo cancellers are used to reduce echoes in a wide range of applications, such as teleconference systems and hands-free telephones. To realistic teleconferencing, multi-channel audio, at least stereophonic, is essential. For stereophonic teleconferencing, stereophonic acoustic echo cancellers (SAEC's) [1], [2], [3] have been studied.

Recent years, PC-based communication systems such as Skype and Messenger becomes very popular. PC-based systems are useful not only for personal communications, but also for business systems such as teleconferencing. For realistic and comfortable teleconferencing, SAEC's should be implemented on PC-based systems.

Modern processors for PC's have powerful instruction set for multimedia processing. Intel IA-32 architectures[4] have MMX (Multi Media eXtension) and also SSE (Streaming Single instruction multiple data Extension, Streaming SIMD Extension). Four-way vector operations are supported for 32-bit floating-point (FP) data. Implementation of SAEC on IA-32 processor has also been reported[5].

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[6], [7], [8] and have been used for computer simulations. Latest GPU's have computation performance over tera floating-point operations per second (TFLOPS). 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 SAEC's on nVIDIA GeForce family GPU and CUDA is discussed. Section II describes SAEC's. GeForce family GPU and CUDA is briefly described in Sec. III. The proposed implementation is shown by Sec. IV. Section V compares the performance.

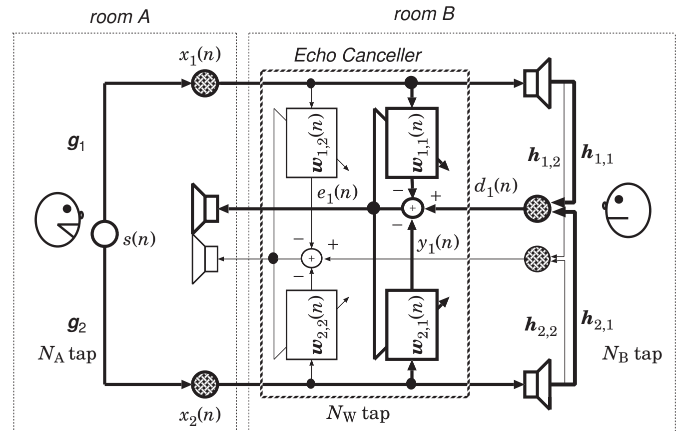

Fig. 1. Teleconferencing using SAEC

II. Stereophonic Acoustic Echo Canceller

Figure 1 shows a teleconferencing using an SAEC. This echo canceller consists of four adaptive filters corresponding to four echo paths from two loudspeakers to two microphones. Each adaptive filter estimates the corresponding echo path.

The far-end signal $\boldsymbol{x}_i(n)$ in the i-th channel at time index n is generated from a talker speech $s(n)$ by passing room A impulse response $\boldsymbol{g}_i$ from the talker to the i-th microphone. $\boldsymbol{x}_i(n)$ passes an echo path $\boldsymbol{h}_{i,j}$ from the i-th loudspeaker to the j-th microphone and become an echo $d_j(n)$. Similarly, adaptive filters $\boldsymbol{w}_{i,j}(n)$ generates an echo replica $y_j(n)$. $\boldsymbol{w}_{i,j}(n)$ is so updated as to reduce the residual echo $e_j(n)$.

The SAEC generates the echo replica $y_j(n)$ by

$$y_j(n) = \boldsymbol{w}_{1,j}^T(n)\boldsymbol{x}_1(n) + \boldsymbol{w}_{2,j}^T(n)\boldsymbol{x}_2(n). \quad (1)$$

the residual echo $e_j(n)$ is calculated by

$$e_j(n) = d_j(n) - y_j(n). \quad (2)$$

Assuming the Normalized Least Mean Squares (NLMS) algorithm[9], the filter coefficient vector $\boldsymbol{w}_{i,j}(n)$ is updated by

$$\boldsymbol{w}_{i,j}(n+1) = \boldsymbol{w}_{i,j}(n) + \frac{\mu e_j(n)\boldsymbol{x}_i(n)}{|\boldsymbol{x}_1(n)|^2 + |\boldsymbol{x}_2(n)|^2} \quad (3)$$

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

978-1-4244-5016-9/09/$25.00 ©2009 IEEE

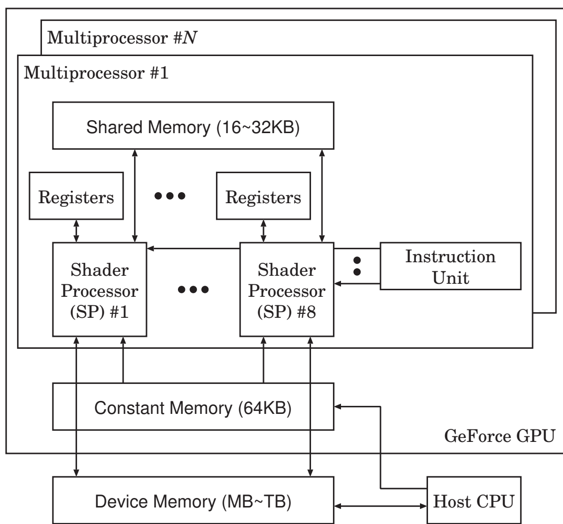

Fig. 2. Computation model of GeForce GPU

III. 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 can be applied for other GPU's. Main features are listed below.

- Unified shader architecture
- Large number of shader processors (SP's):
  - $16 \sim 240$ 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 (slower than 32-bit).
- 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.

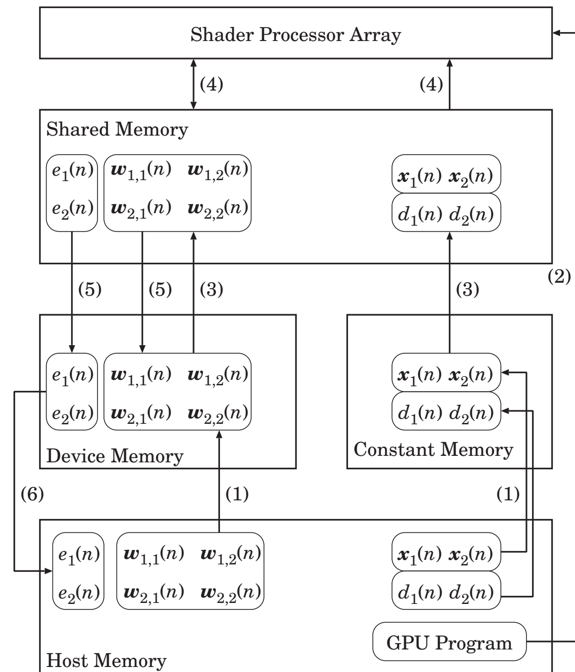

Fig. 3. Memory assignments and dataflow.

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"[6], [7] 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.

IV. Implementation of SAEC

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 SAEC and the other for graphics and video. Another reason is to avoid synchronization and communication between multiple SIMD arrays. This implementation focuses on

- Memory assignment
- Reduction of memory access

- Division of adaptive FIR filters into multiple threads.

A PC-based communication is carried out as a block-based processing, a block with several hundred samples is assumed.

A. Memory assignment

Since the cost for the memory access might restrict the performance of an adaptive FIR filter with the NLMS algorithm, a faster memory is used if available. Figure 3 demonstrates the memory assignments and the flow of data. The input signals, which are not modified, are stored into constant memory and then cached in shared memory. The filter coefficients, which are modified, are stored into device memory. They are loaded into shared memory at the beginning of a block, and restored into device memory at the end of the block.

The procedure of data transfer and signal processing is shown below.

1) Copy signals from CPU to GPU
2) Copy program from CPU to GPU, and execute
3) Copy signals and coefficients from device/constant memory to shared memory
4) Computation for a block of input signals
5) Copy results and coefficients from shared memory to device memory
6) Copy results from GPU to CPU

In the actual implementaion, steps 3 and 4 are performed in the NLMS process for the first sapmle of the block. Also, the NLMS process for the last sample carried steps 4 and out.

B. Reduction of memory access

The number of memory access can be reduce by similar manner as in[5]. The data load can be reduced by changing the order of (1) and (3). Calculating

$$\begin{aligned} w_{i,j,k}(n) &= w_{i,j,k}(n-1) \\ &+ \delta_j(n-1)x_i(n-k-1) \end{aligned} \tag{4}$$

and

$$sum_j(n) = sum_j(n) + w_{i,j,k}(n)x_i(n-k) \tag{5}$$

in the descending order of the tap index k could reduce the number of load operation for both $w_{i,j,k}(n)$ and $x_i(n-k)$. In (4) , $\delta_j(n-1)$ is defined by

$$\delta_j(n-1) = \frac{\mu e_j(n-1)}{|\boldsymbol{x}_1(n)|^2 + |\boldsymbol{x}_2(n)|^2}. \tag{6}$$

$w_{i,j,k}(n)$ is a k-th element of $\boldsymbol{w}_{i,j}(n)$. The load operation for $w_{i,j,k}(n)$ is reduced because $w_{i,j,k}(n)$ calculated in (4) is also used for convolution (5) just after (4). The number of load operation for $x_i(n-k)$ can be reduced because $x_i(n-k)$ in (5) can be re-used in (4) for the next $k = k-1$. In SAEC, treating both channels in the same loop also reduces the memory access cost for the input signals.

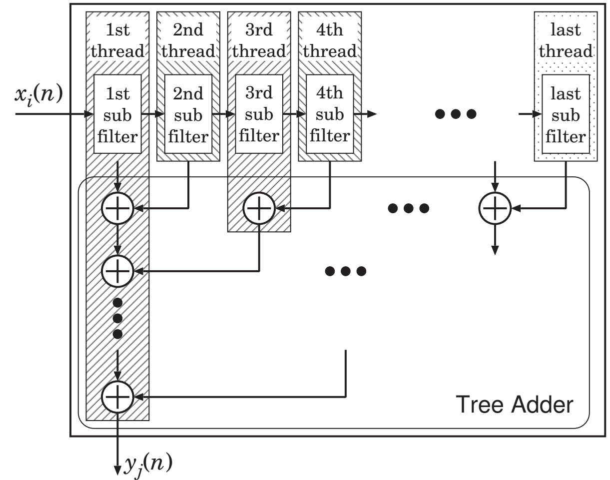

Fig. 4. Multi-thread implementation of adaptive FIR filter

C. Multi-thread implementation of adaptive FIR filter

In this implementation, each adaptive filter is divided simply into short sub filters. Figure 4 shows the implementation of adaptive FIR filters. Each thread processes small segments from all of four adaptive filters from $\boldsymbol{w}_{1,1}(n)$ through $\boldsymbol{w}_{2,2}(n)$. This is because the memory access reduction shown in IV-B requires successive processing of adjacent filter taps and also simultaneous processing of four adaptive filters. This division also simplifies thread division.

A problem specific to GeForce GPU is the computational cost for summing all sub filter outputs up. If this summing-up process is carried out by a thread, it requires larger amount of computations than that for the sub filters. This is because the optimum number of threads is very large. The evaluation results show that the optimum number of threads is 128 for 512-tap SAEC. Therefore, each sub filters have only four taps, while the summing-up process have to sum 128 results up.

In order to reduce the summing-up cost, a tree adder is introduced. Figure 4 depicts the tree adder. For 128-thread case, seven-stage adder is used. The first stage consists of 64 adders by 64 threads; each thread performs one addition.

D. Maximum number of taps

For this implementation, the maximum number of taps is restricted by the shared-memory size. The capacity of shared memory for 32-bit word is only 4096 words. Therefore, the maximum number of taps is about 512, which might be enough for 8kHz sampling and for a small room.

E. Memory assignment for larger number of taps

The maximum number of taps can be increased by slightly degrading the performance. The memory assignments and the data flow are shown in Fig. 5. The procedure is shown below.

1) Copy signals from CPU to GPU
2) Copy program from CPU to GPU, and execute

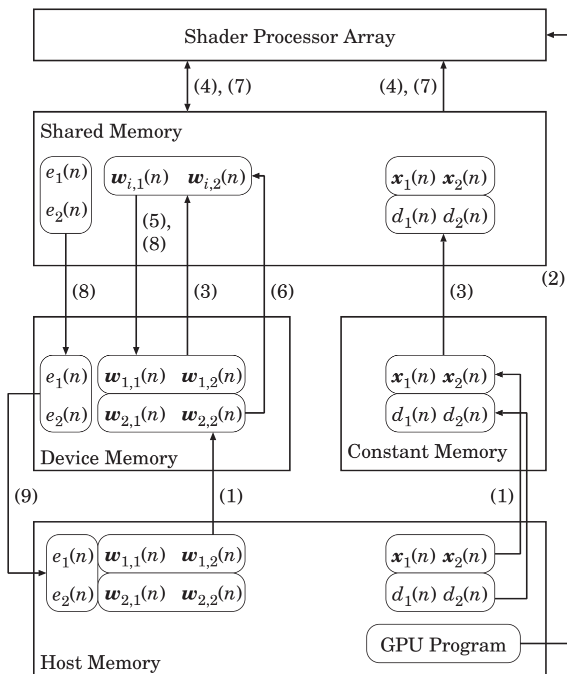

Fig. 5. Dataflow

TABLE I
SPECIFICATIONS OF PLATFORMS

Type	Server	Notebook	Netbook	GPU
CPU	Core 2 Extreme	Core 2 Duo	Atom	GeForce
Model	X9650	T7700	N270	8800 GTS
Total cores	4	2	1	128
Cores used	1	1	1	8
Core Clock	3GHz	2.4GHz	1.6GHz	1.6GHz
OS (bits)	Linux (64bit)	WinXP (32bit)	WinXP (32bit)	—

3) Copy signals and coefficients from device/constant memory to shared memory
4) Computation for 1st channel
5) Copy coefficients from shared memory to device memory
6) Copy coefficients from device/constant memory to shared memory
7) Computation for 2nd channel
8) Copy results and coefficients from shared memory to device memory
9) Copy results from GPU to CPU

In this implementation, the first and the second channels are processed separately. The maximum nuber of taps is almost 768, which is 50% larger than the previous implementation.

V. PERFORMANCE COMPARISON

The standard SAEC has been implemented and tested on two different platforms. Table I 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 has been manually optimized for the speed.

TABLE II
COMPUTATION TIME FOR 512-TAP, 16000-SAMPLE CASE

Type	Server	Notebook	Netbook	GPU 1	GPU 2
Time [msec]	126.57	228.60	698.01	159.60	163.30

TABLE III
COMPUTATION TIME FOR DIFFERENT FILTER SIZES

Tap	128	256	384	512	640	768
Time	111.68	125.03	126.48	162.11	140.45	163.46

Table II compares the processing time for 16000 sample of inputs. The results suggests that all processors are capable of real-time processing for 512-tap SAEC at 16kHz sampling. The difference between GPU 1 and 2 is the memory assignments shown in IV-A and IV-E. Though parallel processing is used and manual optimization have also been carried out, the performance of GPU's is slightly less than Core 2 Extreme server. Furthermore, almost four times speed-up has been achieved by introduction of vector processing for Intel Core 2 family CPU's[5].

On the other hand, the performance of the GPU's is superior to those of notebook CPU's. This result suggests that even a low-cost GPU's with a small number of shader processor greatly helps the echo cancellation for low-cost PC-based teleconferencing.

Table III examines the influence of the number of taps on the performance of GPU 2. The result suggests that there might be a large overhead because a large offset exists, which is not proportional to the number of taps. Without the tree adder, the perfomance is degraded by almost 50%.

VI. CONCLUSIONS

This paper presents an implementation of an SAEC on nVIDIA GeForce GPU and CUDA. For efficiency, fast shared memory has been used as much as possilbe. Introducing the tree adder improves the performance by almost 33%. Even a low-cost GPU's with a small number of shader processor greatly helps the echo cancellation for low-cost PC-based teleconferencing.

REFERENCES

[1] M. M. Sondhi and D. R. Morgan, "Stereophonic acoustic echo cancellation — an overview of the fundamental problem," *IEEE SP Letters*, vol. 2, no. 8, pp. 148–151, Aug. 1995.
[2] A. Sugiyama, Y. Joncour, and A. Hirano, "A stereo echo canceler with correct echo-path identification based on an input-sliding technique," *IEEE Trans. SP*, vol. 49, no. 11, pp. 2577–2587, Nov. 2001.
[3] A. Hirano, K. Nakayama, and K. Watanabe, "Convergence analysis of stereophonic echo canceller with pre-processing — relation between preprocessing and convergence —," *Proc. of ICASSP '99*, pp. 861–864, Mar. 1999.
[4] "Intel 64 and IA-32 architectures software developer's manual volume 1: Basic architecture," May 2007.
[5] 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.
[6] "NVIDIA CUDA compute unified device architecture reference manual," Nov. 2008.
[7] "NVIDIA CUDA programming guide," Dec. 2008.
[8] "ATI stream computing user guide," Mar 2009.
[9] J. Nagumo and A. Noda, "A learning method for system identification," *IEEE Trans. AC*, vol. 12, no. 3, pp. 282–287, Mar. 1967.