# GPU BASED IMPLEMENTATION OF RECURSIVE DIGITAL FILTERING ALGORITHMS

*Dong-hwan Lee and Wonyong Sung*

Department of Electrical Engineering and Computer Science, Seoul National University
Gwanak-gu, Seoul 151-744 Korea
Email: ldh@dsp.snu.ac.kr, wysung@snu.ac.kr

## ABSTRACT

Recursive filtering is widely used for many signal processing applications. Speeding-up the computation of recursive filtering using many processing elements is difficult because of the dependency problem. In this paper, massively parallel computation of recursive filtering algorithms using GPGPUs (General Purpose Graphics Processing Units) is studied. The proposed method uses the multi-block parallel processing algorithm, where each thread executes one block of data as independently as possible. To resolve the dependency among threads, we develop a fast look-ahead method that shows high efficiency even when thousands of threads are used. The developed method has been implemented using Nvidia GTX 285 GPU and shows over 15 times of speed-up when compared to sequential CPU based implementations.

***Index Terms***— Recursive filtering, graphics processing unit (GPU), parallel computation, look-ahead method

## 1. INTRODUCTION

General purpose graphics processing units (GPGPUs), such as Nvidia's GTX series hardware, contain more than one hundred programmable processing elements on a chip, and hence they are used for many computation intensive scientific and signal processing applications. GPU based implementations typically employ thousands of threads not only to utilize all the processing elements but also to overcome the delay of accessing slow global memory. As a result, parallel algorithms developed for small scale multi-core processors do not always show good efficiency in GPGPUs [1].

Recursive filtering, such as $y[n] = a \cdot y[n-1] + x[n]$, is represented in a sequential manner, and parallel computation of multiple output samples is not obvious because computing $y[n+1]$ needs the previous output sample, $y[n]$. This is often called the data dependency problem. Although the dependency problem hinders the parallel computation of recursive filters with any parallel architecture, such as SIMD and multi-core computers, this problem is much more severe with GPUs because the degree of parallelism demanded is much higher.

In this research, we develop a GPU based massively parallel computing program for an $M$th order recursive filtering equation. In order to exploit thousands of threads available, the multi-block parallel processing algorithm is utilized [2]. Especially, we propose a fast look-ahead method to increase the scalability of the multi-block processing and obtain good efficiency even when the number of threads is very large. In this algorithm, the input data is divided into multiple blocks, and each block is assigned to each thread. Then, each thread computes the particular solutions simultaneously. Note that the particular solutions do not need the initial condition of the assigned block. After obtaining the particular solutions, the initial condition for each block is produced using a look-ahead method, and after then the complete solutions for each block are computed independently by each thread. The conventional look-ahead method consumes the time that is proportional to the number of processing elements or threads, which does not degrade the efficiency much in SIMD and multi-core architecture but becomes a critical problem in GPGPUs where thousands of threads need to be utilized. The developed fast look-ahead method demands the time that is only proportional to $\log_2 P$, where $P$ is the number of processing elements. In GPU based implementations, the communication among thread-blocks demands much time. The fast look-ahead algorithm is implemented to minimize the number of inter-thread-block communication operations.

This paper is organized as follows. Section 2 discusses the multi-block parallel processing algorithm including the proposed fast look-ahead method. The implementation procedure using a GPU is explained in Section 3. The experimental results using the Nvidia GTX 285 Tesla architecture is shown in Section 4. Concluding remarks are given in Section 5.

## 2. MULTI-BLOCK PROCESSING ALGORITHM WITH FAST LOOK-AHEAD

There have been quite many previous researches on the parallel computation of recursive equations, and their efficiency depends on the data size and the number of processing elements. When the number of data approximately equals to that of processing el-

This work was supported by the Brain Korea 21 Project and the National Research Foundation of Korea (NRF) grants funded by the Ministry of Education, Science and Technology (MEST), Republic of Korea (No. 2012R1A2A2A06047297). Dong-hwan Lee was also financially supported by the NRF of Korea under the Global Ph.D. Fellowship project.

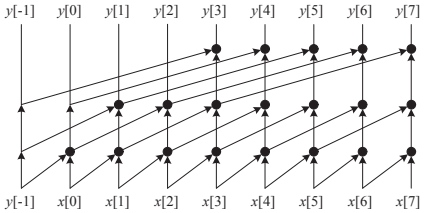


**Fig. 1**. Recursive doubling of 8 data samples.

| | | | |
|---|---|---|---|
| $x[0]$ | $x[L]$ | | $x[(P\text{-}1)L]$ |
| $x[1]$ | $x[L+1]$ | | $x[(P\text{-}1)L+1]$ |
| ⋮ | ⋮ | ⋯ | ⋮ |
| $x[L\text{-}1]$ | $x[2L\text{-}1]$ | | $x[PL\text{-}1]$ |

**Fig. 2**. Data layout for multi-block processing algorithm.

ements, the recursive doubling method shown in Fig. 1 is very efficient [3]. This method computes the output of $N$ data samples in $\log_2 N$ steps using $N$ processing elements. When the number of data is very large compared to that of processing elements, we need to apply the recursive doubling many times until all the input data are processed, which can hardly be efficient because the number of operations for each data increases with the number of processing elements. The multi-block processing method that assigns one block of data to each processor is much more efficient when the number of data is very large.

### 2.1. Multi-block processing

The multi-block processing algorithm utilizes the fact that the solution of a linear recursive equation consists of the particular and the transient solutions. In this parallel computation scheme, the data is distributed to a two dimensional $L$ by $P$ array as shown in Fig. 2, where $P$ is the number of processing units and $L$ is the length of one data block.

The $n$th output of $k$th data block is computed as follows:

$$y[kL+n] = a \cdot y[kL+n-1] + x[kL+n], \qquad (1)$$

where $0 \le k \le P-1$ and $0 \le n \le L-1$. In this first order recursive filtering equation, $x[n]$ and $y[n]$ are the input and output, respectively, and $a$ is the constant coefficient. We can reformulate this equation as follows:

$$y[kL+n] = a^{n+1} \cdot y[kL-1] + z[kL+n] \qquad (2)$$

and

$$z[kL+n] = a \cdot z[kL+n-1] + x[kL+n] \qquad (3)$$

with $z[kL-1] = 0$. Note that $y[kL+n]$ consists of two terms: particular solution $z[kL+n]$ and transient solution $a^{n+1} \cdot y[kL-1]$.

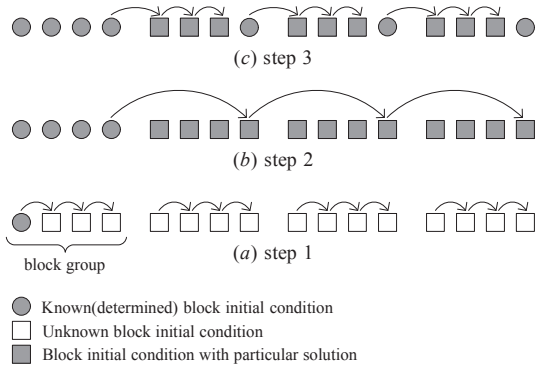


**Fig. 3**. Multi-block processing algorithm for recursive filtering.

The particular solution can be computed by evaluating Eq. (3) while assuming the initial condition $z[kL-1]$ to zero. Since the initial conditions for all data blocks are assumed to be zero, there exists no data dependency among the data blocks and the particular solutions of multiple data blocks can be computed in parallel. In order to obtain the solutions with exact initial conditions or complete solutions, we need to know the initial conditions for data blocks. Note that the initial condition of the first data block is initially given. Since we already obtained the particular solutions $z[k]$, we can compute the initial conditions, $y[kL-1]$ for $k = 1$ to $P-1$, sequentially by using the following equation, which is often called the look-ahead algorithm [2].

$$y[kL-1] = a^L \cdot y[(k-1)L-1] + z[kL-1], \qquad (4)$$

for $k = 1$ to $P-1$. The parallel processing steps can be summarized as follows [2, 4].

**Step 1 (particular solution)** Compute the particular solutions by using Eq. (1) with zero initial conditions except the first data block. This computation is conducted in parallel for all the data blocks, but sequentially within a data block. It demands approximately $L$ time steps.

**Step 2 (look-ahead)** Calculate the initial condition of each data block using Eq. (4). This computation is conducted sequentially from the first to the last data block. This step takes approximately $P$ time steps.

**Step 3 (complete solution)** Compute the complete solutions by executing the recursive filtering equation sequentially with the input data and the block initial condition determined in Step 2. Note that this step is conducted for every block independently and takes $L$ time steps.

The computation procedure is illustrated in Fig. 3. The computation time can be modeled as $2L+P$. Since the execution time for the look-ahead increases in proportion to the number of processing elements, Step 2 becomes the bottleneck as $P$ increases.

### 2.2. Multi-block processing with fast look-ahead

The Step 2 of the conventional multi-block processing, which is the serious bottleneck in GPU based computation, can be represented as follows:

$$\begin{aligned} y[L-1] &= a^L \cdot y[-1] + z[L-1] \\ y[2L-1] &= a^L \cdot y[L-1] + z[2L-1] \\ &\cdots \\ y[PL-1] &= a^L \cdot y[(P-1)L-1] + z[PL-1]. \end{aligned} \tag{5}$$

We can notice that the above computation corresponds to evaluating a recursive equation of $P$ data samples with the constant coefficient of $a^L$. Since the number of data is the same with the number of processing elements, it is efficient to apply the recursive doubling algorithm shown in Fig. 1. As a result, the time step for the look-ahead is now reduced to $\log_2 P$. Unless the block length, $L$, is very small, the time consumed at the Step 2 is not a bottleneck anymore.

### 2.3. Multi-block processing for $M$th order recursive filtering

The first order case can be extended to the $M$th order linear recurrence equation as follows:

$$y[n] = \sum_{i=1}^{M} a_i \cdot y[n-i] + x[n], \tag{6}$$

where $x[n]$, $y[n]$, and $a_i$ are the input data, output data, and coefficients, respectively. This equation can be represented in a vector form [5]:

$$Y[n] = A \cdot Y[n-1] + X[n], \tag{7}$$

where

$$Y[n] = \begin{bmatrix} y[n] \\ y[n-1] \\ \vdots \\ y[n-M+1] \end{bmatrix}, \; X[n] = \begin{bmatrix} x[n] \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{8}$$

and

$$A = \begin{bmatrix} a_1 & a_2 & \cdots & a_{M-1} & a_M \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix} \tag{9}$$

Similar to the first order case, the $n$th output of the $k$th data block can be represented in a vector form as follows:

$$Y[kL+n] = A \cdot Y[kL+n-1] + X[kL+n], \tag{10}$$

where $0 \le k \le P-1$ and $0 \le n \le L-1$. We can manipulate this equation similar to Eq. (2) and (3), thus we have

$$Y[kL+n] = A^{n+1} \cdot Y[kL-1] + Z[kL+n] \tag{11}$$

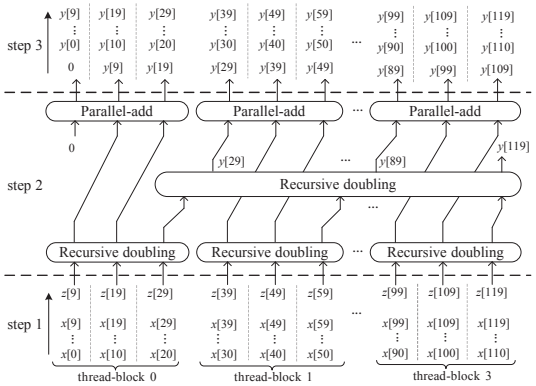


**Fig. 4**. Implementation of parallel processing algorithm in GPUs.

and

$$Z[kL+n] = A \cdot Z[kL+n-1] + X[kL+n]. \tag{12}$$

Since $Y[kL+n]$ also consists of the particular solution $Z[kL+n]$ and the transient solution $A^{n+1} \cdot Y[kL-1]$, the multi-block processing can be applied in a similar way to the first order case.

## 3. GPU BASED PARALLEL COMPUTATION OF RECURSIVE FILTERING

We use the Nvidia GTX series architecture and CUDA based development environment. Nvidia GTX 285 GPU contains 30 SMs (streaming multiprocessors), and each SM has 8 CUDA cores. Since a GPU employs multi-thread scheduling in order to hide the memory access latency and resolve pipelining hazards, the number of thread-blocks in a parallel program is usually much larger than that of SMs. In GPU based implementations, the number of thread-blocks is usually more than 100, and the number of threads in each thread-block is usually between 32 and 128. The communication within each thread-block is very fast, however that among the thread-blocks is relatively slow [1]. Thus, we need to minimize the number of inter-thread-block communication operations.

The input data of the length $N$ is divided into blocks, and each block is assigned to a single thread. In this paper, we denote the number of thread-blocks as $n_{tb}$ and assume that each thread-block contains $n_t$ threads. Thus, the block length $L$ is equal to $N/(n_{tb} \cdot n_t)$. For example, the GPU based implementation with $n_{tb} = 4$ and $n_t = 3$ is illustrated in Fig. 4. At the first stage of computation, each thread needs to read $L$ input samples from the global memory to compute the particular solutions. For the coalesced memory access, we assign a half warp (16 threads in GTX 285) of threads to read one data block as shown in Fig. 5. During the first iteration, 16 threads (threads $0 \sim 15$ in Fig. 5) read the first data block ($x[0] \sim x[15]$), and the other 16 threads (thread $16 \sim 31$) access the second data block ($x[L] \sim x[L+15]$). This results in only two DRAM access transactions for reading 32 samples. Note that consecutive data samples in DRAM can be

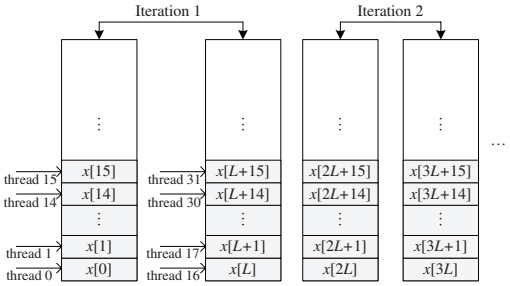


**Fig. 5**. Global memory access for Steps 1 and 3.

accessed quite efficiently. Since the current kernel needs to be terminated for thread synchronization, the data to use later should be stored in the global memory.

At the second stage of computation, the fast look-ahead routine that employs the recursive doubling is conducted in two steps. At the first step, the recursive doubling is performed within each thread-block and yields the last output samples. This step is conducted by every thread-block, and the number of output samples is now reduced to $n_{tb}$. At the second step, the $n_{tb}$ output samples are combined by recursive doubling. This step needs a separate kernel, and uses only one thread-block that employs $n_{tb}$ threads.

At the third stage of computation, the complete solutions can be obtained with the global memory access scheme shown in Fig. 5. Finally, all the output samples are written back to the global memory.

## 4. EXPERIMENTAL RESULTS

We use the Nvidia GTX 285 running at 1,476 MHz [6]. The global memory size of the GTX 285 is 1 GByte and the memory bandwidth that employs a 512 bit bus can be as high as 159.0 GB/sec. CUDA (Compute Unified Device Architecture) of Nvidia is used for C language based programming of the GPU [1]. For the purpose of comparison, a few CPU-based versions were also implemented, which included sequential (single-core non-SIMD) and parallel (four core with 4-way SIMD) programs. A desktop PC with an Intel Core2 Quad CPU (Q9550) running at 2.83 GHz clock was used. During the experiments, 4 Mega data samples are used. We found the optimum parameters for $n_{tb}$ and $n_t$ from the experiments. Usually, $n_t$ of 32 yielded the best results, while the other parameters that resulted in the best performance varied with the order of the recursive equation.

Recursive filtering with orders of 1, 2, 4, and 8 were implemented in parallel and their performances in the CPU and the GPU are compared. Table 1 summarizes the implementation results, and the developed method shows significant speed-ups when compared to the sequential and parallel CPU implementations. The speed-up with respect to the sequential CPU implementation is over 2,000 % when the filer order is higher than 4. Also, we can find that the speed-up of the GPU-based implementations increases as the order of the equation goes up. This result occurs because recursive filtering consumes more arithmetic operations per sample as the order of the equation becomes higher, and GPU-based parallel processing architecture is more efficient in handling arithmetic-intensive problems.

**Table 1**. Comparison of CPU and GPU implementations when 4 Mega samples are used (time is in ms, parentheses show speed-up w.r.t. sequential CPU)

| | $M = 1$ | $M = 2$ | $M = 4$ | $M = 8$ |
|---|---|---|---|---|
| Sequential on a CPU | 11.079 | 15.318 | 28.56 | 46.266 |
| Parallel on a CPU | 11.076 | 11.247 | 11.504 | 11.657 |
| Parallel on a GPU | 0.728 (1,522%) | 0.902 (1,698%) | 1.317 (2,169%) | 2.058 (2,248%) |

## 5. CONCLUDING REMARKS

We have implemented recursive filtering equations using GPUs (Graphics Processing Units). Although recursive filtering incurs a dependency problem and GPU based implementations demand a large degree of parallelism, it was possible to obtain a significant speed-up by using a multi-block processing algorithm with fast look-ahead. This method can be extended to the parallel computation of adaptive filters.

## 6. REFERENCES

[1] NVIDIA Corporation., "NVIDIA CUDA (Compute Unified Device Architecture) programming guide," [Online]. Available: http://developer.nvidia.com/object/cuda.html.

[2] W. Sung, S.K. Mitra, and B. Jeren, "Multiprocessor implementation of digital filtering algorithms using a parallel block processing method," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 1, pp. 110–120, Jan. 1992.

[3] P. Kogge and H. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Transactions on Computers*, vol. C-22, no. 8, pp. 786–793, Aug. 1973.

[4] J. Ahn, H. Chang, J. Cho, and W. Sung, "SIMD processor based implementation of recursive filtering equations," in *Proc. of IEEE Workshop on Signal Processing Systems (SiPS)*, Oct. 2009, pp. 087–092.

[5] C. Burrus, "Block implementation of digital filters," *IEEE Transactions on Circuit Theory*, vol. 18, no. 6, pp. 697–701, Nov. 1971.

[6] NVIDIA Corporation., "NVIDIA GeForce GTX 285," [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-285.