# Evaluation of P-Scheme/G Algorithm for Solving Recurrence Equations

Akiyoshi Wakatani

***Abstract*—A parallel algorithm called P-scheme/G is proposed for solving recurrence equations on GPGPU systems. This is based on P-scheme algorithm that has been originally developed for distributed memory multicomputers. In order to achieve a high performance computation on GPGPU systems, our method alleviates branch divergences by reducing the stride data accesses. We also illustrate the effectiveness of the optimal thread configuration for the recurrence equation. Our experiments with GTX 590 show that the implementation of the rearrangement using the shared memory improves the performance by 200\% to 300\% and the validity of the policy of the thread configuration is confirmed for both the constant and the non-constant parameter cases. We achieve the speedup of around 400 as a recurrence equation solver with non-constant parameters.**

***Index Terms*—CUDA, GPU, multithreading, tridiagonal matrix solver.**

## I. Introduction

Recently, the peak performance of GPU (Graphic Processing Unit) has increased very much and outperforms that of general-purpose processors. Since past GPUs consisted of special-purpose hardware, they were used only for graphic processing and image processing. However, recent GPUs are composed of general-purpose unified shaders, so by using CUDA (Compute Unified Device Architecture) [1], they are used for general purpose processing like numerical calculations as well as graphic processing.

Linear first-order recurrence equations are expressed as $w_i=s_i \times w_{i-1}+t_i$, but these cannot be parallelized straightforwardly by dividing domains because the value of $w_i$ is determined by using $w_{i-1}$. The recurrence equations are used frequently on many applications like Gauss elimination, the tridiagonal matrix solver and DPCM (Differential Pulse-Code Modulation) codec, so it is very important to implement the recurrence equation solver on GPGPU systems in order to achieve a high performance [2]-[5]. Then, we modify the parallel algorithm of a recurrence equation solver (called "Pscheme") so that it is suitable for GPGPU system and we evaluate the performance comparison of our improved method (called "P-scheme/G") on GPU and CPU. Note that P-scheme has been originally developed for distributed memory computers by the authors [6].

In this paper, we show the effectiveness of the rearrangement of array configurations that improves the efficiency of accesses to the global memory by using coalesced communications. We also illustrate the effectiveness of the optimal thread configuration for the recurrence equation with constant parameter and non-constant parameter cases.

The rest of this paper is organized as follows: Section II presents the P-scheme algorithm for the recurrence equations and Section III summarizes the two optimization methods. Section IV presents the experimental method and discusses the results and Section V concludes this paper with a summary.

Manuscript received February 20, 2013; revised June 22, 2013. This work was supported in part by MEXT, Japan.

Akiyoshi Wakatani is with Faculty of Intelligence and Informatics, Konan University, Kobe, Japan (e-mail: wakatani@konan-u.ac.jp).

## II. Recurrence Equations

The following system is considered:

$$w_0 = C \tag{1}$$

$$w_i = s_i \times w_{i-1} + t_i \quad (1 \le i \le N) \tag{2}$$

P-scheme method is a recurrence equation solver suitable for parallel processing [6]-[8]. The P-scheme consists of three phases. Here $N$ (the size of array) is assumed to be a multiple of $P$ (the number of threads), $M$ is defined as $N/P$ and each thread $k\,(0 \le k < P)$ is in charge of the computation of array element $w_i$ $(k \times M + 1 \le i \le (k+1) \times M)$.

First of all, $w_{k \times M}$ is assumed to be zero. Then (2) is calculated and $a_i=s_{k \times M+1} * s_{k \times M+2} * \ldots * s_i$ is also calculated. This phase is called "pre-computation" and it can be carried out independently. The computational complexity of the pre-computation phase is $O(N/P)$. As mentioned, since $w_{k*M}$ is assumed to be zero, $w_i$ should be corrected. The element $w_i$ is corrected by using the relation of $w_i=w_i+a_i*w_{k \times M}$, so the elements $w_M$, $w_{2M} \ldots w_{(P-1)M}$ can be corrected with the cost of $O(P)$. This phase is called "propagation." Finally, each thread corrects other elements by using the corrected $w_{k \times M}$ independently. This phase is called "determination" and the complexity of this phase is $O(N/P)$. It should be noted that this method can be used for linear recurrence equations as well as non-linear recurrence equations.

The pre-computation and determination phases can be completely parallelized. Meanwhile the propagation phase is still sequential but the cost of the propagation phase is in proportion to the number of threads. Thus the total execution time is estimated by $O(N/P)+O(P)+O(N/P)$. It is general that $O(P)$ is absolutely less than $O(N)$, because $P$ is supposed to be much less than $N$.

Although the pre-computation and determination phases

can be parallelized, the computational complexity is larger than the original substitution. Execution times of the original substitution and P-scheme on PC (Pentium III (1 GHz), 1 GB memory, GCC 4.1.1 with the option of O3) are shown in Fig. 1. The graph shows that the execution time of P-scheme is about twice slower than that of the original substitution.

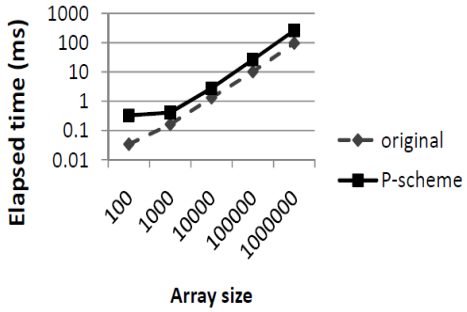


Fig. 1. Comparison of execution times.

It should be noted here that we consider three cases of $s_i$: light computation (case L), medium computation (case M) and heavy computation (case H). Namely, $s_i$ in the case L is a constant parameter, so no floating point calculations are required. However, it is assumed that the calculation of $s_i$ for the case M and case H requires 400 floating point operations per data 1000 floating point operations per data, respectively.

## III. Optimization

### A. Rearrangement of Array Configurations

As mentioned before, the pre-computation and determination phases can be completely parallelized between threads, but the array assignment determines whether the global memory accesses can be done by the coalesced communication or not. On the P-scheme algorithm for distributed memory computers, the *i*-th thread is in charge of the computations between $w_{i\times(N/P)}$ and $w_{(i+1)\times(N/P)-1}$ when $w_i$ is distributed into *P* threads. On GPGPU systems, $w_{0+k}$, $w_{(N/P)+k}$, $w_{2\times(N/P)+k}$, $w_{3\times(N/P)+k}$, ... are concurrently accessed at the *k*-th step since calculations on GPUs are in principle SIMD calculations. However, these are accessed using the non-coalesced communication, so the access cost is very large.

In order to cope with this difficulty, array elements that are accessed simultaneously should be rearranged so that they are adjacent to each other.

$$w'_{i\times P+j} = w_{j\times s+i} \quad (0 \le i \le s-1, \quad 0 \le j \le P-1)$$

where $s = N/P$ . Namely, this rearrangement is equal to the transposition of a $P \times s$ two-dimensional array into a $s \times P$ two-dimensional array.

The computation of the P-scheme utilizes the adjacent data (previous data), then the so-called SOA (Structure Of Array) transformation works very well [9]. However, our method (the rearrangement of array configurations) differs from the SOA in the point described later.

We call the P-scheme method with the rearrangement of array configuration P-scheme/G. Note that our rearrangement method requires an array with the size of $N+P$ instead of $N+1$.

Fig. 2 shows a simple SOA (Structure Of Array) transformation. At first, thread 0 gets the value of $w_0$ that is stored at the location $w_1$-1 while thread 1 get the value of $w_M$ that is stored at the location $w_{M+1}+P\times(M-1)-1$. This situation causes the branch divergence. For the rest of the steps, each thread gets the value that is stored at (the current location -*P*). Therefore, the simple SOA results in many branch divergences because some calculations of this parallel scheme need adjacent data.

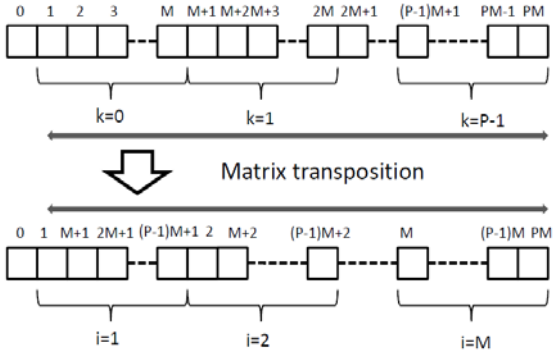


Fig. 2. Simple SOA transformation.

However, by duplicating the elements of the last index, the branch divergences can be alleviated. As shown in Fig. 3, $w_{i-1}$ is always stored at the location $w_i$-*P* so the program can be created easily with less branch divergences.

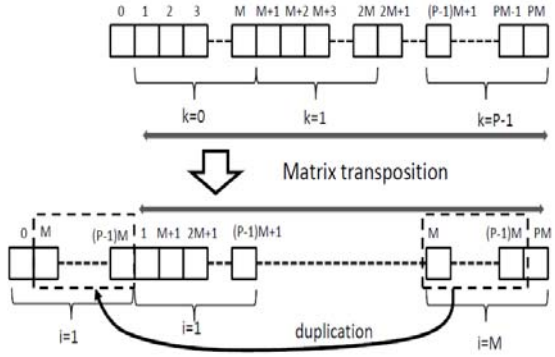


Fig. 3. Rearrangement of array configurations.

In order to rearrange the array located in the global memory, the array should be fetched and stored, but one of the memory accesses is non-coalesced communication. But, using the shared memory in the intermediate form and rearranging the array within the shared memory, both memory accesses can be done in a coalesced communication way.

### B. Optimal Thread Configurations

We discuss the optimal combination of *G* and *T* when the rearrangement of array configurations is used. *G* is the number of thread blocks and *T* is the number of threads in a thread block.

As shown in previous sections, the execution times of all the phases are roughly estimated as follows:

$$\text{pre} - \text{comp} = \alpha \cdot \frac{N}{P} \tag{3}$$

$$\text{propagation} = \beta \cdot P \tag{4}$$

$$\text{determination} = \gamma \cdot \frac{N}{P} \tag{5}$$

where $\alpha$, $\beta$ and $\gamma$ are the execution costs per data for the pre-computation phase, the propagation phase and the determination phase, respectively.

On our experimental environment, the values of $\alpha$, $\beta$ and $\gamma$ are measured as follows:

$$\alpha : \beta : \gamma = \begin{cases} 1.6 : 1.0 : 1.1 \text{ (case L)} \\ 20.0 : 1.0 : 1.1 \text{ (case M)} \\ 91.7 : 1.0 : 1.1 \text{ (case H).} \end{cases} \tag{6}$$

The parallelism is in proportion to *P*, so, in order to achieve the maximum speedup by increasing *P*, the following policy should be applied: 1) *G* should be maximized first and 2) the optimal *T* should be selected.

By using (3), (4), and (5), the execution time *t* and the optimal parallelism $P_{opt}$ with the minimum execution time are determined as follows:

$$t = \alpha \cdot \frac{N}{P} + \beta \cdot P + \gamma \cdot \frac{N}{P}$$
$$P_{opt} = \sqrt{\frac{(\alpha + \gamma)N}{\beta}} \tag{7}$$

When *N* is $10^6$ and the computation of $s_i$ is light (constant case), $P_{opt}$ is around 1643.17, so *T* should be 128, 64, 32 and 16 for the cases with the value of *G* of 16, 32, 64 and 128, respectively. Moreover, when *N* is $10^6$ and the computation of $s_i$ is medium, $P_{opt}$ is around 4593.5, so *T* should be 256, 128, 64 and 32 for the cases with the value of *G* of 16, 32, 64 and 128, respectively. Finally, when *N* is $10^6$ and the computation of $s_i$ is heavy, $P_{opt}$ is around 9633.3, so *T* should be 512, 256, 128 and 64 for the cases with the value of *G* of 16, 32, 64 and 128, respectively.

## IV. Experiments and Discussion

### A. Experimental Environment

Our experiments are carried out on the GPGPU system that consists of Intel Core i7 875K, 8.0 GB memory and GTX 590 GPU under Yellow Dog Linux and CUDA 4.1. It should be noted that the GTX 590 consists of two sets of 16 multi processors having 32 cores, but we utilize only one of two sets by using one process, so the number of CUDA cores used in the experiment is 512. Regarding the CPU, we use nvcc compiler for compilation of CPU codes in order to keep the fairness of the quality of the optimization. The CPU codes do not use multithreading and the SSE extension.

In order to evaluate our approach on the GPGPU system, we focus on the linear recurrence equation shown in (1) and (2) and we construct G thread blocks having T threads and execute them in parallel on GPUs, that is, the total number of threads is $P = T \times G$.

The elements of arrays are fetched from the global memory to registers of cores and the results are directly stored to the global memory, so the shared memory is not used because no data is repeatedly used in our method. Thus, the usage of the shared memory does not affect the performance very much. Therefore we do not care the bank conflict of the shared memory. The global synchronization is implemented by invoking individual kernels. Since our method consists of three phases, only two global synchronizations are required between the phases. So, since the overhead of the synchronization is quite small, the usage of plural kernels does not affect the total performance.

### B. Effectiveness of the Rearrangement of Array Configurations

In order to illustrate the effectiveness of the rearrangement of array configurations, Fig. 4 shows the speedups with and without the rearrangement when N is 256K and the number of thread blocks (*G*) is 32. The speedup is the ratio of elapsed times of CPU computation and GPU computation. The GPU1 shows the speedup without the rearrangement and the GPU2 shows that with the rearrangement. When the computation of $s_i$ is medium (case M), the maximum speedup is achieved at *T* of 128 and the result of the GPU2 is 1.45 (=92.91/63.68) times larger than the GPU1. Moreover, when the computation of $s_i$ is heavy (case H), the maximum speedup is achieved at *T* of 64 and the result of the GPU2 is 1.23 (=233.58/189.87) times larger than the GPU1. Therefore, since the effectiveness of the rearrangement of array configurations is obviously confirmed, we only consider the cases with the rearrangement hereafter.

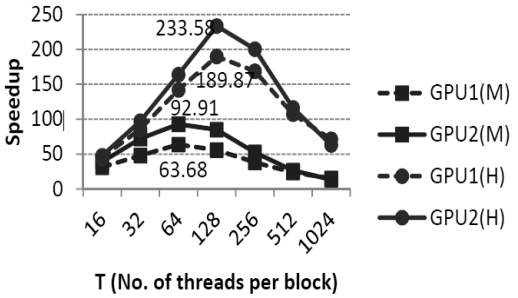


Fig. 4. Effectiveness of the rearrangement of array configurations.

In addition to the rearrangement, we consider the overhead of matrix transposition, by which the rearrangement of array configurations is implemented. The matrix transposition can be simply implemented by using the memory accesses within the global memory, but its memory access cost is large because half of the memory accesses are non-coalesced communication. In order to alleviate the memory access cost, we should utilize the shared memory and all the memory accesses to the global memory can be done in the coalesced communication. First of all, the matrix is divided into the sub matrix with the size of $32 \times 32$ and the sub matrix is fetched into the shared memory by using the coalesced communication in a row orientation. Then, the sub matrix is fetched from the shared memory in a column orientation and it is stored in the global memory by using the coalesced communication.

TABLE I: Elapsed Time of the Rearrangement of Array Configurations

| | Without the shared mem. | With the shared mem. |
|---|---|---|
| *N=256K* | 0.33 ms | 0.11 ms |
| *N=1M* | 0.751 ms | 0.376 ms |

It should be noted that the size of the sub matrix is limited by the size of the shared memory. The elapsed times of the matrix transposition with the value of N of $256\times10^3$ and $10^6$ are shown in Table I. As shown in the table, by using the shared memory, the rearrangement can be carried out two to three times faster than that without the shared memory.

### C. Effectiveness of the Optimal Thread Configurations

1) Light computation case: When $s_i$ is the light computation case (constant case), Fig. 5 shows the speedups for several combinations of *G* and *T*. As described in the previous section, the best speedup can be achieved at $G\times T$ of about 2048. For example, the best speedup of 7.1 is achieved at *T* of 32 when *G* is 64, and the best speedup of 6.84 is achieved at *T* of 16 when *G* is 128. So, the validity of the estimation of the optimal number of threads is confirmed when the parameter of the recurrence equation is constant.

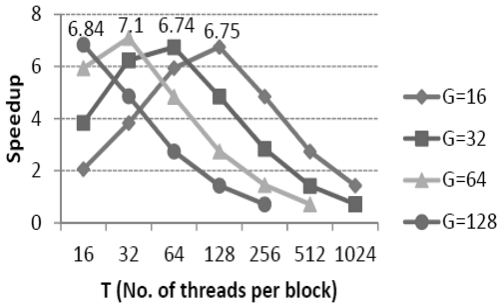


Fig. 5. Results in the case L.

2) Medium and heavy computation cases: When $s_i$ is non-constant cases (the case M and the case H), Fig. 6 and Fig. 7 show the speedups for several combinations of *G* and *T*.

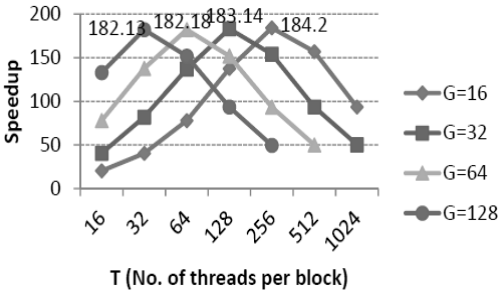


Fig. 6. Results in the case M.

As described in the previous section, when the parameter calculation is medium (the number of single precision multiplications and divisions is 200 each), $\alpha:\beta:\gamma=20.0:1.0:1.1$, so $P_{opt}$ is $4.582\sqrt{N}$ according to (7). Thus, when $N = 10^6$, $P_{opt}$ is calculated as 4691.9.

As shown in the figure, whatever the value of *G* is, the best speedup is achieved at $P(=G\times T)$ of 4096. However, when *G* = *128*, the effectiveness of the parallelism is slightly degraded and then the value of the speedup is smaller than others.

The value of the speedups for the case M (around 183) is larger than that for the case L (around 7), because the principal term of the elapsed time is the floating point operations instead of the global memory accesses and so the effectiveness of the parallelization is enhanced.

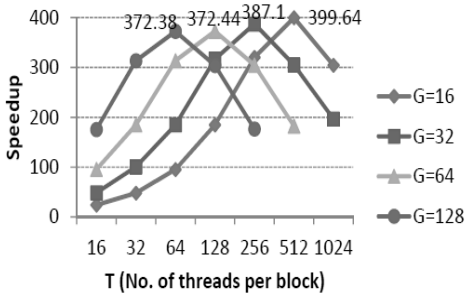


Fig. 7. Results in the case H.

Moreover, when the parameter calculation is large (the number of single precision multiplications and divisions is 500 each), $\alpha:\beta:\gamma=91.0:1.0:1.1$, so $P_{opt}$ is $9.633\sqrt{N}$ according to (7). Thus, when $N = 10^6$, $P_{opt}$ is calculated as 9864.5.

As shown in the figure, whatever the value of *G* is, the best speedup is achieved at $P(=G\times T)$ of 9192. As well as the case M, the larger *G* is, the smaller the value of the speedup is. The value of the speedups for the case H (over 370) is much larger than the other cases because the main part of the elapsed time is the parameter calculation and so the effectiveness of the parallelism is determined by the number of active cores. We achieve the speedup of up to 400 for this case on GTX 590 GPU.

## V. Conclusion

We implemented a parallel recurrence equation solver on GPGPU systems by using CUDA and evaluated the effectiveness of the rearrangement of array configuration in order to utilize the coalesced communication. We also proposed a policy to decide the optimal number of threads per thread block and the optimal number of thread blocks in order to maximize the efficiency of parallelism and empirically confirmed the validity of the policy by our experiments. Then we achieve the speedup of around 400 by using two optimization methods. In the near future, we will implement a non-linear recurrence equation on GPGPU in order to solve the tridiagonal matrix equations by using the Thomas method. We will also try to implement our approach using OpenCL that recently attracts a lot of attention and evaluate it on other accelerators.

## References

[1] D. Kirk and W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Massachusetts: Morgan Kaufmann, 2010.
[2] Y. Zhang, L. Cohen, and J. D. Owens, "Fast tridiagonal solvers on the GPU," in *Proc. PPoPP 2010*, Bangalore, 2010, pp. 10.
[3] D. Goddeke and R. Strzodka, "Cyclic reduction tridiagonal solvers on GPUs applied to mixed precision multigrid," *IEEE Trans. on Parallel and Distributed Systems*, vol. 22, pp. 22-32, 2011.
[4] D. Lee and W. Sung, "Multi-core and SIMD architecture based implementation of recursive digital filtering algorithms," in *Proc. ICASSP 2010*, Dallas, 2010, pp. 1550-1553.
[5] E. Dekker and L. Dekker, "Parallel minimal norm method for tridiagonal linear systems," *IEEE Trans. on Computer*, vol. 44, pp. 942-946, 1995.
[6] A. Wakatani, "A parallel and scalable algorithm for ADI method with pre-propagation and message vectorization," *Parallel Computing*, vol. 30, pp. 1345-1359, 2004.

[7] A. Wakatani, "A parallel scheme for solving a tridiagonal matrix with pre-propagation," in *Proc. 10th Euro PVM/MPI Conference*, Venice, 2003, pp. 222-226.
[8] A. Wakatani, "A parallel and scalable algorithm for calculating linear and non-linear recurrence equations," in *Proc. Int'l Conf. Parallel and Distributed Computing and Networks*, Las Vegas, 2004, pp. 446-451.
[9] J. A. Stratton *et al*., "Algorithm and data optimization techniques for scaling to massively threaded systems," *IEEE Computer*, vol. 45, no. 8, pp. 26-32, 2012.

**Akiyoshi Wakatani** was born in Osaka City, Osaka Pref., Japan, on February 3, 1962. He received the B. Eng. degree from the Department of Applied Mathematics and Physics, Faculty of Engineering, Kyoto University, Kyoto, Japan, in 1984. He received the M. Eng. degree from the Division of Applied Systems Science, Faculty of Engineering, Kyoto University in 1986. He also received the Dr. Eng. degree from the Division of Information Engineering, Faculty of Engineering, Kyoto University in 1996.

He was with Matsushita Electric Industrial (currently Panasonic) from 1986 to 2000, as a researcher and a senior researcher. From 1992 to 1994, he was a visiting scholar of Oregon Graduate Institute of Science and Technology, OR, USA. From 2000 to 2006, he was an associate professor of Department of Information Science and Systems Engineering, Faculty of Science and Engineering, Konan University. From 2006 to 2008, he was a Full Professor of the same department in the same university. Since 2008, he has been a Full Professor of Department of Intelligence and Informatics, Faculty of Intelligence and Informatics, Konan University, Kobe, Japan. His research interests include parallel processing and parallel architecture.

Prof. Wakatani is a member of the IEEE, the Information Processing Society of Japan, and the IEICE of Japan.