Implementation and Evaluation of Recurrence Equation Solvers on GPGPU systems using Rearrangement of Array Configurations

Akiyoshi Wakatani*
* *Faculty of Intelligence and Informatics*
Konan University
Higashinada, Kobe, 658-8501, Japan
wakatani@konan-u.ac.jp

***Abstract*—The recurrence equation solver is used in many numerical applications and other general-purpose applications, but it is inherently a sequential algorithm, so it is difficult to implement the parallel program for it. Recently, GPGPU (General Purpose computing on Graphic Processing Unit) attracts a great deal of attention, which is used for general-purpose computations like numerical calculations as well as graphic processing. In this paper, we implement a parallel and scalable algorithm for solving recurrence equations on GPUs by using CUDA (Compute Unified Device Architecture) and evaluate its effectiveness. The algorithm was originally implemented for MIMD parallel computers like a PC cluster and an SMP system by the authors and we modify the algorithm suitable for the GPGPU system by rearranging arrays configurations.**

***Keywords*-multithreading; tridiagonal solver; GPU; multi-core; CUDA**

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 like GeForce 8 type of NVIDIA 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.

Parallel applications having less data dependencies can be easily implemented on GPGPU systems, but complicated data dependencies prevent an optimal implementation of applications on GPGPU systems because we must carefully select which data should be kept in a small but fast memory. 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 [11], [12].

In this paper, we modify the parallel algorithm of recurrence equations "P-scheme" suitable for GPGPU system and we evaluate the performance comparison of our methods on GPU and CPU. Note that P-scheme has been developed for distributed memory computers by the authors [2].

The rest of this paper is organized as follows: Section 2 presents the P-scheme algorithm and Section 3 summarizes the prior arts related to our method. Section 4 presents the experimental method and discusses the results and Section 5 concludes this paper with a summary.

II. RECURRENCE EQUATIONS

A. Tridiagonal system of equations

P-scheme is an algorithm that solves a recurrence equation in parallel. Our purpose is to parallelize a solver for the following tridiagonal system of equations of $A \times x = c$ where A is a tridiagonal matrix with $N \times N$ elements and x and c are vectors with N elements. The system is given by

$$x_0 = c_0 \quad (1)$$

$$-b_i \cdot x_{i-1} + a_i \cdot x_i - b_i \cdot x_{i+1} = c_i \ (1 \le i \le N-2) \quad (2)$$

$$x_{N-1} = c_{N-1} \quad (3)$$

where arrays a and b are elements of matrix A and array c is given in advance, array x is an unknown variable and N is the number of elements of the arrays.

B. P-scheme

It is known that the system given by Equations (1), (2) and (3) can be deterministically solved by Gaussian elimination, which utilizes two auxiliary arrays p and q.

$$p_0 = 0,\ q_0 = c_0 \quad (4)$$

$$p_i = \frac{b_i}{a_i - b_i \cdot p_{i-1}},\ q_i = \frac{c_i + b_i \cdot q_{i-1}}{a_i - b_i \cdot p_{i-1}} \quad (5)$$

$$x_i = x_{i+1} \cdot p_i + q_i \quad (6)$$

The above procedure consists of a forward substitution (Equation (5)) and a backward substitution (Equation (6)). However, on parallel computers, the procedure cannot be straightforwardly parallelized due to the data dependency

 ISBN: 978-1-61208-217-2

that resides on both the forward and backward substitutions. Suppose that the number of processor is P, $N = P*M+2$ and arrays are block-distributed. Processor k $(0 \le k \le P-1)$ is in charge of M elements of arrays from $k*M+1$ to $k*M+M$, thus p_{k*M+1} can be calculated on processor k only after $p_{(k-1)*M+M}$ is calculated on processor $k-1$. Meanwhile, x_{k*M+M} can be calculated on processor k only after $x_{(k+1)*M+1}$ is calculated on processor $k+1$. These data-dependencies completely diminish the possibility of parallel computing.

We have proposed a parallel and scalable algorithm, called "*P-scheme*" [2], [3], [4]. We focus on array p on Equation (5) and explain how P-scheme works for it. Note that the equation for array p is a non-linear recurrence equation and the equations for arrays q and x are linear recurrence equations. So, our method can be easily extended to the equations q and x. Then we assume that p_{i-j} and p_i can be expressed by the following equation:

$$\frac{\beta_j + \gamma_j \cdot p_i}{\delta_j + p_i} = (-1)^j \cdot p_{i-j} \ (j > 0), \tag{7}$$

where β_j, γ_j and δ_j are auxiliary arrays that are defined below. By substituting Equation (5) to Equation (7), the following relation can be found.

$$\begin{aligned} & \frac{\frac{\delta_j + (-1)^{j+1} \cdot \frac{a_{i-j}}{b_{i-j}} \cdot \beta_j}{\gamma_j} + \frac{1 + (-1)^{j+1} \cdot \frac{a_{i-j}}{b_{i-j}} \cdot \gamma_j}{\gamma_j} \cdot p_i}{\frac{\beta_j}{\gamma_j} + p_i} \\ = \ & (-1)^{j+1} \cdot p_{i-(j+1)} \end{aligned} \tag{8}$$

Thus, β_j, γ_j and δ_j can be determined by the following system of equations:

$$\beta_1 = 1, \ \gamma_1 = -\frac{a_i}{b_i}, \ \delta_1 = 0 \tag{9}$$

$$\beta_{j+1} = \frac{1}{\gamma_j}(\delta_j + (-1)^{j+1} \cdot \frac{a_{i-j}}{b_{i-j}} \cdot \beta_j) \tag{10}$$

$$\gamma_{j+1} = \frac{1}{\gamma_j}(1 + (-1)^{j+1} \cdot \frac{a_{i-j}}{b_{i-j}} \cdot \gamma_j) \tag{11}$$

$$\delta_{j+1} = \frac{\beta_j}{\gamma_j} \tag{12}$$

It should be noted that β_j, γ_j and δ_j are independent of p_j. Hence p_i can be determined by using only p_0 as follows:

$$p_i = \frac{-\beta_i + (-1)^i \cdot p_0 \cdot \delta_i}{\gamma_i - (-1)^i \cdot p_0} \tag{13}$$

Therefore, if β_i, γ_i and δ_i are calculated in advance, p_i can be directly determined just after p_0 is determined.

By using the above relation, we proposed a scheme called *P-scheme(Pre-Propagation scheme)*, which consists of three phases. First of all, every processor simultaneously starts its calculation of β_i, γ_i and δ_i. This is called *pre-computation* phase. After that, processor 0 can directly determine p_M from p_0, β_M, γ_M and δ_M and sends p_M to processor 1. After receiving it, processor 1 can directly determine p_{2*M} from p_M, β_M, γ_M and δ_M and sends p_{2*M} to processor 2 and then processor 2 can directly determine p_{3*M} from received p_{2*M} and its auxiliary arrays and sends p_{3*M} to processor 3 and so on. This is called *propagation* phase. It should be noted that processor k can determine $p_{(k+1)*M}$ without calculating p_{k*M+i} $(1 \le i \le M-1)$. Finally, all processors can determine p_{k*M+i} $(1 \le i \le M-1)$ by using received data p_{k*M}. This is called *determination* phase.

The pre-computation and determination phases can be completely parallelized. Meanwhile the propagation phase is still sequential but the data to be exchanged is very slight, like just one data, so the communication cost is also expected to be very slight. The cost of the propagation phase is in proportion to the number of processors. Thus the total execution time is estimated by $O(\frac{N}{P}) + O(P) + O(\frac{N}{P})$. 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 given by Equation (5). Since the original contains 1 multiplication, 1 division, 1 addition, 3 loads and 1 store and P-scheme contains 4 multiplications, 3 divisions, 3 additions, 6 loads and 4 stores, P-scheme must carry out over twice more computation than the original substitution. Execution times of the original substitution and P-scheme on PC (Pentium III (1 GHz), 1GB memory, GCC4.1.1 with O3) are shown in Figure 1. The graph shows that the execution time of P-scheme is about twice slower than that of the original substitution.

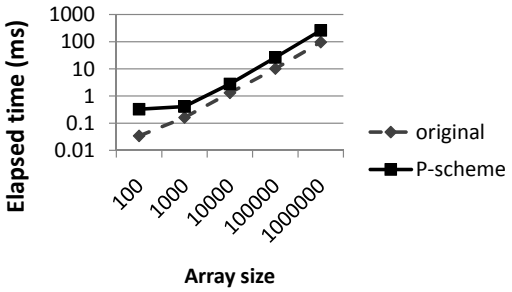


Figure 1. Comparison of execution times

III. RELATED WORKS

It is known that the first-order recurrence equation cannot be parallelized straightforwardly since the i-th element can be determined by using the $(i-1)$-th element. CR (Cyclic Reduction) and RD (Recursive Doubling) are recurrence equation solvers which can be directly applied on parallel computers [5], [6], [7], [8], [9].

 ISBN: 978-1-61208-217-2

A tridiagonal matrix can be solved by using recurrence equations and several tridiagonal matrix solvers have been implemented on GPUs. Kass et al. used ADI method for an approximate depth-of-view computation and solved the tridiagonal matrix by using CR method on a GPU [10]. Zhang et al. applied four methods (CR, parallel CR, RD and hybrid) to the tridiagonal matrix solver on the GPU and evaluated the performances to find that the hybrid method achieved the best performance [11]. Goddeke and Strzodka proposed mixed precision iterative solvers using CR method and implemented it on the GPU. They found that the resulting mixed precision schemes are always faster than double precision alone schemes, and outperform tuned CPU solvers [12].

When the size of the matrix is $N \times N$, the computational complexity of CR is $O(N)$ but it requires $2\log_2 N$ synchronizations between processors. Meanwhile, parallel CR requires only $\log_2 N$ synchronizations but its total computational complexity is $O(N \times \log_2 N)$. On the other hand, since the sequential algorithm (Gaussian elimination) consists of two recurrence equations (a forward substitution and a backward substitution) and both of the recurrence equations can be parallelized by using our method, those computational complexities are $O(N/P)$, $O(P)$ and $O(N/P)$, respectively. Our method also requires only two synchronizations for each recurrence equation. Then, we implement our method for recurrence equations on GPUs and evaluate the parallelism and the effectiveness of the rearrangement of array configurations in order to utilize the coalesced communication.

IV. Rearrangement of array configurations

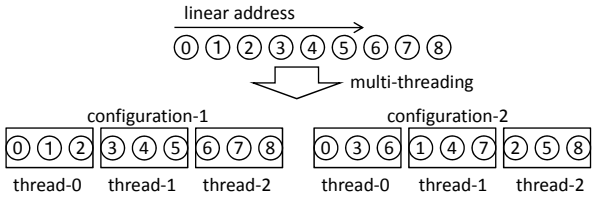


Figure 2. Rearrangement of array configurations

As mentioned before, the pre-computation and determination phases can be completely parallelized between threads, but the global memory accesses are done in either the coalesced communication or the non-coalesced communication, which depends on the array assignment. 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}, $w_{(N/P)+k}$, $w_{2*(N/P)+k}$, $w_{3*(N/P)+k}\cdots$ 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.

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

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

Figure 2 shows an example of the rearrangement of array configurations. Suppose that the size of an array is 9 and the array should be divided into three parts. In an ordinary parallel computer like a PC cluster or an SMP system, the array should be just divided simply, so thread 0 is in charge of array elements 0, 1 and 2, thread 1 is in charge of array elements 3, 4 and 5, and thread 2 is in charge of array elements 6, 7 and 8, because this configuration (configuration 1) can enhance the locality of memory accesses and the efficiency of the cache memory. On the other hand, in a GPGPU system having NVIDIA's GPUs, the array should be rearranged (configuration 2) in order to utilize the coalesced communication, that is, thread 0 is in charge of array elements 0, 3 and 6, thread 1 is in charge of array elements 1, 4 and 7, and thread 2 is in charge of array elements 2, 5 and 8. So, at the first step, the threads access array elements 0, 1 and 2, and at the second step, the threads access array elements 3, 4 and 5, and so on. Thus threads can always coalesce their memory accesses into one memory request.

In the following subsections, we will evaluate the effectiveness of the rearrangement of arrays empirically by using the experiments.

V. Experiment and discussion

A. Experimental environment

Our experiments are carried out on the GPGPU system that consists of AMD Phenom II X4 945 (3.0 GHz), 4.0 GB memory and Tesla C1060 GPU (30 MPs and compute capability 1.3) under Windows 7 Ultimate and CUDA 3.0.

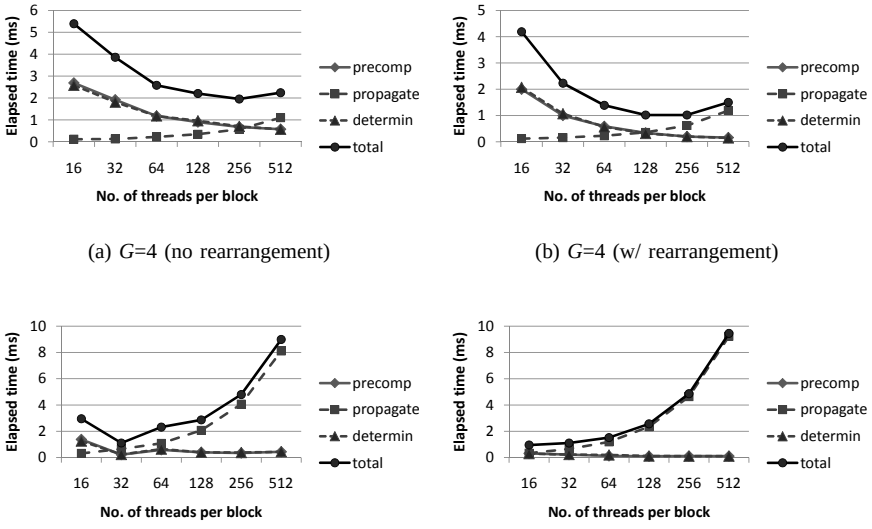
In order to evaluate our approach on the GPGPU system, we focus on the following linear recurrence equation:

$$\begin{aligned} w_0 &= C \\ w_i &= scale \times w_{i-1} + offset\ (1 \le i \le N) \end{aligned}$$

where C, $scale$ and $offset$ are constant. The value of N is set to 2^{18} (small arrays) and 2^{20} (large arrays), 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 SPs 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. Therefore we do not care the bank conflict of the shared memory. The

 ISBN: 978-1-61208-217-2



(a) G=4 (no rearrangement)

(b) G=4 (w/ rearrangement)

(c) G=32 (no rearrangement)

(d) G=32 (w/ rearrangement)

Figure 3. Results of $N = 256K(= 2^{18})$

global synchronization is implemented by invoking different 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, it does not affect the total performance.

B. Occupancy and threads

Occupancy is one of performance metrics that predict the effective performance on GPU execution. As mentioned earlier, one MP has 8 SPs and each SP executes one thread, so the efficiency of the MP does not reach 100% unless there are at least 8 threads. Due to the difference between the clock frequencies of the SP (shader clock) and that of the MP (core clock), at least 4 threads should be concurrently executed on one SP in order to keep the instruction pipeline of the MP full. Therefore, it is recommended that at least 32 threads should be placed on each MP and this size (32) is called "warp." Moreover, since the access latency to the global memory is large, the large size of the thread group is preferable for hiding the latency.

In the CUDA execution environment, G thread blocks are assigned to MPs and T threads are assigned to SPs within each MP. As mentioned before, the number of SPs within a MP is 8, but at least 32 threads, namely the size of the warp, must be assigned to one MP in order to maintain the efficient execution. Moreover, more threads should be assigned to each MP in order to improve the occupancy. On the other hand, the GPGPU system that is used for our experiment has 30 MPs, so G should be around 30 but it may be less than 30 for the optimal P. Since P is $T \times G$ and the execution times of the pre-computation and determination phases are in inverse proportion to P, they decrease when T and G increase. However, the execution time of the propagation phase increases when T and G increase, so the total execution time may be worsen. Therefore, one of purposes of our experiments is to confirm the contribution of T and G to the execution time.

C. Experiment 1 (small arrays)

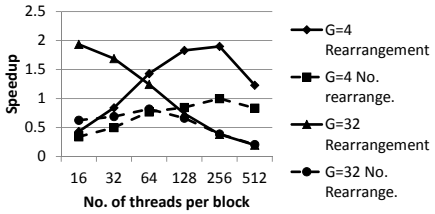
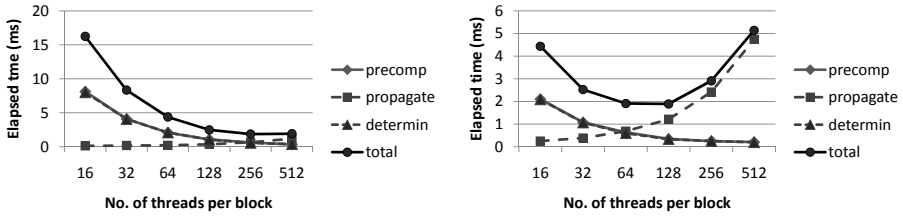


Figure 4. Speedups of $N = 256K(= 2^{18})$

The experimental results with the array size of $256K(= 2^{18})$ are shown in Figure 3. In the figure, the execution times of the pre-computation, propagation and determination phases are illustrated when G is 4 and 32 and T is varied from 16 to 512. It should be noted that the elapsed times of the pre-computation and the determination are same on the whole, and thus these lines are almost overlapped.

For both cases with no rearrangement and with the rearrangement of array configurations, the execution times of these phases decrease as P increases. For example, when G is 4 and the rearrangement is used, the execution times of the pre-computation phase with T of 16, 32, 64 and 128 are 2.07

 ISBN: 978-1-61208-217-2



(a) Elapsed time (G=4) (b) Elapsed time (G=16)

Figure 5. Results of $N = 1M(= 2^{20})$ (w/ rearrangement)

msec, 1.07 msec, 0.59 msec and 0.34 msec, respectively. When the rearrangement is not used, the execution times of the pre-computation phase with T of 16, 32, 64 and 128 are 2.69 msec, 1.92 msec, 1.18 msec and 0.9 msec, respectively. When T is 64 and the rearrangement is used, the execution times of the pre-computation phase with G of 4 and 32 are 0.59 msec and 0.13 msec, respectively. When the rearrangement is not used, the execution times of the pre-computation phase with G of 4 and 32 are 1.18 msec and 0.59 msec, respectively. As P increases, the area where each thread is in charge is getting smaller, so the overhead like a thread creation increases relatively. Note that the speedup seems to be flat when T is over 8, because the number of SPs per MP is 8, but, as T increases, the occupancy increases until the number of threads reaches the warp size (The occupancy is 1.0 when a MP of Tesla C1060 has 128 threads). Therefore, by increasing T (over 8), the performance can be improved. It should be also noted that, by rearranging the array configuration and using the coalesced communication, the performance can be enhanced from 2 to 10 times, which depends on the value of G.

On the other hand, as P increases, the execution time of the propagation phase increases. For example, when G is 4 and the rearrangement is used, the execution times of the pre-computation phase with T of 16, 32, 64 and 128 are 0.25 msec, 0.38 msec, 0.69 msec and 1.21 msec, respectively. Since the propagation phase is carried out on one SP, there is no difference between the execution time using the rearrangement and that without the rearrangement.

The comparisons of the speedups with a variety of parameter settings based on the execution time of the CPU are shown in Figure 4.

On the whole, the speedups using the rearrangement outperform those without the rearrangement. When the rearrangement is used, the difference of the speedups is small since the change of the value of T does not result in the difference of the execution time so much. However, when the rearrangement is used, the value of T and P decide whose phase should be dominant among the execution times of three phases, so an optimal value of T and P results in the largest speedup. For example, when G is 4, the maximum speedup is 1.9 with the value of T of 256. As mentioned below, $G \times T$ is constant when the combination of G and T results in the maximum speedup.

Moreover, when G and T are large, the value of the speedup using the rearrangement is almost identical to that without the rearrangement. For example, this is true when $G = 32, T = 128, 256, 512$. The reason is that the propagation phase is dominant among three phases when G and T are large. But there is no difference between the execution time using the coalesced communication and that using the non-coalesced communication, since the propagation phase is carried out on one SP.

D. Experiment 2 (large arrays)

A part of the experimental results with the array size of 1 M $(= 2^{20})$ are shown in Figure 5. The results of this case are almost equal to those of the case with the size of 256 K. Namely, as P increases, the execution times of the pre-computation and the determination phases decrease. It is also found that the performance can be enhanced from 2 to 10 times by using the rearrangement of array configurations and reducing the access cost to the global memory. As P increases, the execution time of the propagation phase increases. The difference from the case with the size of 256 K is that the absolute time of the pre-computation and the determination phases increases due to the increase of the array size and then the execution time of the propagation phase is relatively smaller than the 256 K case. Therefore, the optimal value of P is larger than the 256 K case.

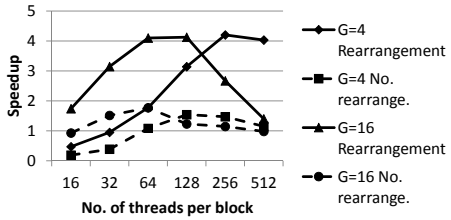
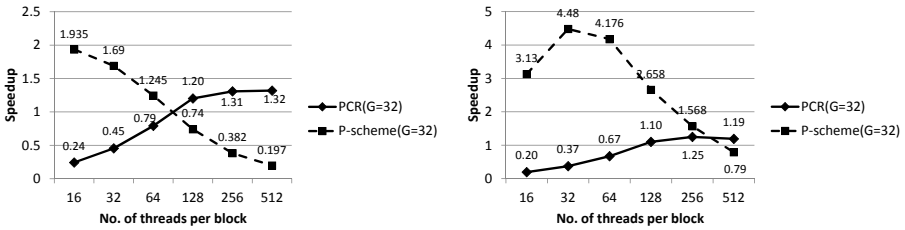


Figure 6. Speedups of $N = 1M(= 2^{20})$



(a) N=256K (b) N=1M

Figure 7. Comparison with PCR (Parallel Cyclic Reduction)

The comparisons of the speedups with a variety of parameter settings based on the execution time of the CPU are also shown in Figures 5 and 6 .

The trend of the results of the speedups are also similar to that of the 256 K case except that the value of T for the maximum speedup is smaller than the 256 K case.

E. Comparison with PCR

We compare the performance of our methods with PCR (parallel cyclic reduction) method [11] and show the experimental results in Figure 7. The maximum speedup of the P-scheme is larger than that of the PCR method for both cases because the computational complexities of the PCR method and the P-scheme are $O(N \cdot \log N)$ and $O(N)$, respectively. However, the speedup depends on the size of the thread block very much, so the tuning parameter for GPGPU programs must be carefully selected in order to achieve the maximum speedup.

F. Discussion

We discuss the optimal combination of G and T when the coalesced communication is used. The speedups using the rearrangement of array configurations when N is 256 K and 1 M are shown in Figure 8,

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

$$pre-comp = \alpha \cdot \frac{N}{P} \quad (14)$$

$$propagation = \beta \cdot P \quad (15)$$

$$determination = \gamma \cdot \frac{N}{P} \quad (16)$$

where α, β and γ are the execution costs per data for the pre-computation phase, the propagation phase and the determination phase, respectively.

The computational complexities of these three phases are almost identical. However, while both the pre-computation phase and the determination phase are executed on T threads (over 1 warp) within thread blocks, the propagation phase is carried out only on one thread. Thus, since the core clock is 4 times slower than the shader clock, we assume the computational complexities of all the phases as follows:

$$\alpha : \beta : \gamma \simeq 1 : 4 : 1. \quad (17)$$

The parallelism is in proportion to G when G increases until the number of MPs, but it is almost flat when G is over the number of MPs. Since Tesla C1060 has 30 MPs, we evaluate the cases with the value of G of up to 32 in our experiments. Each MP has 8 SPs, so the parallelism is in proportion to T until T reaches 8 and it is in quasi-proportion to T until the warp size (32). When T is more than 32, the occupancy is getting close to 1.0 but the increase of the parallelism is almost flat. Therefore, 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 Equations (14), (15) and (16), the execution time t and the optimal parallelism P_{opt} are determined as follows:

$$t = \alpha \cdot \frac{N}{P} + \beta \cdot P + \gamma \cdot \frac{N}{P}$$

$$P_{opt} = \sqrt{\frac{(\alpha+\gamma)N}{\beta}} = \sqrt{\frac{N}{2}}. \quad (18)$$

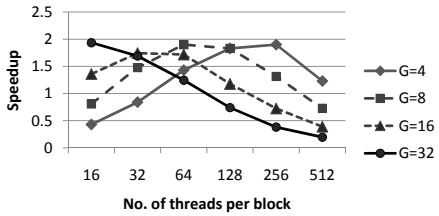
When N is 1 M, P_{opt} is nearly equal to 720, so T should be 180, 90, 45 and 22.5 for the cases with the value of G of 4, 8, 16 and 32, respectively. According to Figure 8-(a), T for the maximum speedup is 256, 128, 64 and 32 when G is 4, 8, 16 and 32. Thus, the estimation and the experimental results are identical. Moreover, When N is 256 K, P_{opt} is nearly equal to 360, so T should be 90, 45, 22.5 and 11.25 for the cases with the value of G of 4, 8, 16 and 32, respectively. According to Figure 8-(a), T for the maximum speedup is 256, 64, 32 and 16 when G is 4, 8, 16 and 32. Thus, the estimation and the experimental results are almost identical for this case as well.

Therefore, the maximum speedup is achieved when G is 32. So it is indicated that the optimization policy described above is rational. Namely, G should be maximized first and

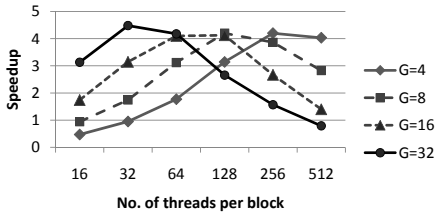
 ISBN: 978-1-61208-217-2

then the optimal T should be selected in order to achieve the maximum speedup.

It should be noted that P_{opt} is $\sqrt{\frac{N}{2}}$ and thus $\frac{N}{P_{opt}}$ is $\sqrt{2N}$. Therefore the computational complexity of the propagation phase is $O(P_{opt}) = O(\sqrt{N})$ and it is not larger than the computational complexities of other phases ($O(\frac{N}{P_{opt}}) = O(\sqrt{N})$).



(a) N=256K



(b) N=1M

Figure 8. Comparison of speedups

VI. 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.

In the near future, we will apply the recurrence equation solver to real applications. We will also try to implement our approach using OpenCL that recently attracts a lot of attention.

Acknowledgment

This work was supported in part by MEXT, Japan.

References

[1] D. Kirk, and W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann, Massachusetts, 2010.

[2] A. Wakatani, “A Parallel and Scalable Algorithm for ADI Method with Pre-propagation and Message Vectorization,” Parallel Computing, vol. 30, pp. 1345-1359, 2004.

[3] A. Wakatani, “A Parallel Scheme for Solving a Tridiagonal Matrix with Pre-propagation,” Proc. 10th Euro PVM/MPI Conference, Venice, 2003, pp. 222-226.

[4] A. Wakatani, “A Parallel and Scalable Algorithm for Calculating Linear and Non-linear Recurrence Equations,” Proc. Int’l Conf. Parallel and Distributed Computing and Networks, Las Vegas, 2004, pp. 446-451.

[5] R. Hockney and C. Jesshope, Parallel Computer 2. Taylor & Francis, London, 1988.

[6] J. Lopez and E. Zapata, “Unified Architecture for Divide and Conquer Based Tridiagonal System Solver,” IEEE Transactions on Computer, vol. 43, pp. 1413-1425, 1994.

[7] O. Eğecioğlu, et al., “A Recursive Doubling Algorithm for Solution of Tridiagonal Systems on Hypercube Multiprocessors, ” J. of Comput. and Applied Mathematics, vol. 27, pp. 95-108, 1989.

[8] E. Dekker and L. Dekker, “Parallel Minimal Norm Method for Tridiagonal Linear Systems,” IEEE Transactions on Computer, vol. 44, pp. 942-946, 1995.

[9] D. Lee and W. Sung, “Multi-core and SIMD architecture based implementation of recursive digital filtering algorithms, ” Proc. IEEE International Conference on Acoustics Speech and Signal Processing (ICASSP), 2010, pp. 1550-1553.

[10] M. Kass, A. Lefohn and J.D. Owens, “Interactive Depth of Field Using Diffusion,” Technical report 0601,Pixar Animation Studios, 2006, pp. 1-8.

[11] Y. Zhang, L. Cohen and J.D. Owens, “Fast Tridiagonal Solvers on the GPU,” Proc. PPoPP 2010, Bangalore, 2010, 10 pages.

[12] D. Goddeke and R. Strzodka, “Cyclic Reduction Tridiagonal Solvers on GPUs Applied to [Mixed Precision Multigrid,” IEEE Transactions on Parallel and Distributed Systems, vol. 22, pp. 22-32, 2011.

 ISBN: 978-1-61208-217-2