

International Conference on Computational Science, ICCS 2013

Parallelizing Alternating Direction Implicit Solver on GPUs

Zhangping Wei^{a,*}, Byunghyun Jang^b, Yaoxin Zhang^a, Yafei Jia^a^aNational Center for Computational Hydrosience & Engineering, The University of Mississippi, University, MS 38677 U.S.A.^bDepartment of Computer & Information Science, The University of Mississippi, University, MS 38677 U.S.A.

Abstract

We present a parallel Alternating Direction Implicit (ADI) solver on GPUs. Our implementation significantly improves existing implementations in two aspects. First, we address the scalability issue of existing Parallel Cyclic Reduction (PCR) implementations by eliminating their hardware resource constraints. As a result, our parallel ADI, which is based on PCR, no longer has the maximum domain size limitation. Second, we optimize inefficient data accesses of parallel ADI solver by leveraging hardware texture memory and matrix transpose techniques. These memory optimizations further make already parallelized ADI solver twice faster, achieving overall more than 100 times speedup over a highly optimized CPU version. We also present the analysis of numerical accuracy of the proposed parallel ADI solver.

Keywords: GPGPU; ADI; PCR; Performance Optimization; Parallel Computing

1. Introduction

Fast simulation of a large volume of data is becoming increasingly important in modern scientific and engineering researches and developments. Time integration is normally categorized into explicit and implicit methods. On the question of whether one of these methods is preferable the last word has not yet been said. In general, an explicit method requires impractically small time steps to keep the error in the result bounded and an implicit method takes much less computational time with larger time steps by solving the equations of the state of the system in time domain. From the parallel computation point of view, an implicit method is more difficult to parallelize than an explicit method because the solution at a point is dependent on those in the entire domain. In this study, we parallelize one of the most important implicit methods, Alternating Direction Implicit (ADI) method [1], on modern Graphics Processing Units (GPUs).

ADI is a finite difference numerical analysis method for solving parabolic, hyperbolic, and elliptic partial differential equations in multiple dimensions, and it has been widely used in scientific and engineering fields [2, 3, 4, 5]. In ADI method, each numerical step is split into several sub-steps based on the spatial dimension of the problem, and the linear equation system is solved implicitly in one direction while treating information in the other direction(s) explicitly. With this alternating calculations, ADI method is unconditionally stable and second order in time and space [6]. Another favorable property of the ADI method is that in each sub-step the equations to be solved have a tridiagonal structure and can be solved efficiently with Tri-Diagonal Matrix Algorithm (TDMA) [7]. As solution of tridiagonal equation systems is found in many disciplines, several parallel alternatives of serial

*Corresponding author. Tel.: +1-662-915-7788; fax: +1-662-915-7796.

E-mail address: zpwei@ncche.olemiss.edu.

TDMA have been developed. Cyclic Reduction (CR) [8], Recursive Doubling (RD) [9], Parallel Cyclic Reduction (PCR) [10] are popular parallel choices.

General-Purpose computing on GPUs (GPGPU) provides an excellent platform to accelerate such parallel alternatives. Its data-parallel execution model perfectly fits into these algorithms where data sharing is localized into blocks and loop-carried dependencies are elegantly eliminated. GPU achieves its high performance by executing more than thousands of threads simultaneously and each of them processes different sets of data. Multiple programming languages are available to program GPUs such as CUDA C, CUDA FORTRAN, Direct-Compute, OpenCL and OpenACC [11, 12]. There have been many successful GPU applications reported in several research areas, such as medical imaging analysis [13] and computational fluid dynamics [14, 15]. Well optimized application presents several orders of magnitude speedup over its highly optimized serial version on high-end CPUs.

In this paper, we parallelize ADI solver using PCR on modern GPUs and demonstrate its efficiency with a two-dimensional heat conduction simulation example. Our implementation significantly improves existing GPU-based PCR solvers by addressing their limitations. First, we propose thread mapping schemes that remove the dependency of the amount of shared memory used. This shared memory limitation restricts existing solvers to increase the domain sizes up to only 1024×1024 which is too small to model real world phenomena [16, 17]. Our implementation no longer has this domain size limitation. Second, our implementation optimizes the usage of limited yet performance-critical hardware resources for better hardware utilizations. Third, our proposed ADI solver also improves the memory accesses efficiency of sweep directions by using hardware texture memory, and its performance is compared with an existing matrix transpose approach. Finally, the performance of our four different versions of ADI solver are evaluated using a heat conduction simulation with large domain sizes. The results show that the proposed ADI solver is very efficient and suitable for real-world scientific simulations.

2. Related Work

With GPU becoming a viable alternative to CPU for parallel computing, aforementioned parallel tridiagonal solvers and other hybrid methods have been implemented on GPUs [16, 17, 18, 19, 20, 21, 22, 23]. Zhang et al. [16] first implemented PCR and then proposed a CR-PCR hybrid algorithm. A hybrid of PCR-Thomas method was proposed by Sakharaykh [22], and it was also studied by Zhang et al. [16]. Although they demonstrated the benefit of using GPUs, their implementations can solve only small size systems as they store entire input system in shared memory which is a small limited memory space. This observation motivates us to address the shared memory limitation of previous PCR implementation and to redesign it scalable up to realistic large systems. Several researchers have also worked on hybrid PCR methods to handle large input sizes [18, 21].

As each step of ADI method combines solution of tridiagonal equation systems, GPU tridiagonal solvers allow entire ADI solver to be straightforwardly implemented on GPUs [15, 23]. With multi-dimensional data in the same application, an usual strategy is to map them to one-dimensional array and store them linearly in memory space. In this way, memory access is coalesced in one direction but not for the rest directions. To improve the code performance in uncoalesced directions, an existing solution is to first transpose the uncoalesced tridiagonal input array to be coalesced, and then solve it with tridiagonal solver, finally the results are transposed back to original directions [23]. In this scheme, all directions are solved by the same solver and the performance is highly dependent on the amount and efficiency of matrix transpose, and the shape of multi-dimensional arrays.

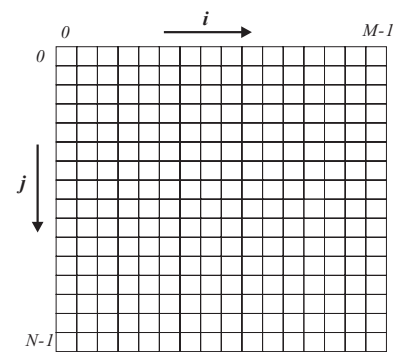


Fig. 1. Discretized two-dimensional computation domain.

3. Alternating Direction Implicit Method

In this section, we show how ADI method is used to solve a two-dimensional heat conduction equation. We also introduce TDMA and PCR algorithms which are underlying building blocks of ADI method.

3.1. Heat Conduction Equation Discretization

Heat conduction equation is a special and simplified case of a mass transport equation or Navier-Stokes Equations in computational fluid dynamics [7]. Given the same numerical methods, the discretized heat conduction equation has the same algebraic equation structure with other more advanced equations. In this way, our proposed ADI solver tested by the heat conduction simulation can be easily adopted for solving other equation systems.

In heat conduction equation, the computation domain is discretized in multiple dimensions. Fig. 1 shows a two-dimensional discretization where i and j are nodal indices in horizontal and vertical directions, respectively, and M and N are the total number of nodes (i.e., domain size) in horizontal and vertical directions, respectively. A two-dimensional unsteady heat conduction equation [24] is expressed as

$$\rho c \frac{\partial T}{\partial t} = k \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) \quad (1)$$

where T is temperature, ρ is the density of conduction media, c is specific heat and k is diffusion coefficient. For the sake of simplicity, $\frac{k}{\rho c} = 1$ is assumed and the above equation is rewritten as

$$\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \quad (2)$$

Dirichlet boundary condition is specified along the domain boundaries, Equation (2) is implicitly discretized by finite difference method in the rectangular domain and, at a typical node (i, j), an algebraic equation is formed as

$$A_P T_{i,j}^{n+1} + A_E T_{i+1,j}^{n+1} + A_W T_{i-1,j}^{n+1} + A_N T_{i,j+1}^{n+1} + A_S T_{i,j-1}^{n+1} = T_{i,j}^n \quad (3)$$

with coefficients at internal nodes expressed as

$$A_P = \left(1 + 2 \frac{\Delta t}{\Delta x^2} + 2 \frac{\Delta t}{\Delta y^2} \right), \quad A_W = A_E = -\frac{\Delta t}{\Delta x^2}, \quad A_N = A_S = -\frac{\Delta t}{\Delta y^2}$$

where n is time level, Δt is time step, Δx and Δy are mesh sizes in horizontal and vertical directions, respectively. Since the temperature value is constant at boundary, coefficients are given as

$$A_P = 1, \quad A_W = A_E = A_N = A_S = 0$$

3.2. Heat Conduction Solution by ADI

Equation (3) then can be efficiently solved by ADI method [1] in two sub-steps. At the first sub-step, Equation (3) is solved in i direction:

$$a_i T_{i-1,j}^{n+1/2} + b_i T_{i,j}^{n+1/2} + c_i T_{i+1,j}^{n+1/2} = d_i \quad (4)$$

where $a_i = A_W$, $b_i = A_P$, $c_i = A_E$ and $d_i = T_{i,j}^n - A_N T_{i,j+1}^n - A_S T_{i,j-1}^n$. For the next sub-step, Equation (3) is solved in j direction:

$$a_j T_{i,j-1}^{n+1} + b_j T_{i,j}^{n+1} + c_j T_{i,j+1}^{n+1} = d_j \quad (5)$$

where $a_j = A_S$, $b_j = A_P$, $c_j = A_N$ and $d_j = T_{i,j}^{n+1/2} - A_E T_{i+1,j}^{n+1/2} - A_W T_{i-1,j}^{n+1/2}$.

Equations (4) and (5) are linear equation systems of the form $[A]\{T\} = \{d\}$ as shown in Equation (6). The readers should be advised that the coefficients a , b and c are constant for all internal nodes in this specific case and, therefore, Equation (6) can be further simplified and solved easily. However, these coefficients may vary at

each time step for more complex applications. From solver parallelization point of view, we keep the original system for generality.

$$\begin{pmatrix} b_0 & c_0 & & & \\ a_1 & b_1 & c_1 & & 0 \\ & a_2 & b_2 & c_2 & \\ & & \ddots & \ddots & \ddots \\ 0 & & & \ddots & \ddots & c_{M-2} \\ & & & & a_{M-1} & b_{M-1} \end{pmatrix} \begin{pmatrix} T_0 \\ T_1 \\ T_2 \\ \vdots \\ T_{M-2} \\ T_{M-1} \end{pmatrix} = \begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{M-2} \\ d_{M-1} \end{pmatrix} \quad (6)$$

3.3. Thomas Algorithm

Thomas Algorithm (TDMA) is an efficient tridiagonal matrix solver for Equation (6) and it has two sweeps: forward elimination and backward substitution. In the first sweep, the lower diagonal is eliminated by

$$c'_0 = \frac{c_0}{b_0}, c'_i = \frac{c_i}{b_i - c'_{i-1}a_i}, i = 1, 2, \dots, M-1 \quad (7)$$

$$d'_0 = \frac{d_0}{b_0}, d'_i = \frac{d_i - d'_{i-1}a_i}{b_i - c'_{i-1}a_i}, i = 1, 2, \dots, M-1 \quad (8)$$

The second sweep solves unknowns from the last point of the domain to the first as

$$T_{M-1} = d'_{M-1}, T_i = d'_i - c'_i T_{i+1}, i = M-2, M-3, \dots, 0 \quad (9)$$

This algorithm is serial in nature. In order to calculate c'_i , d'_i and T_i , their immediately preceding terms c'_{i-1} , d'_{i-1} and T_{i+1} have to be calculated ahead.

3.4. Parallel Cyclic Reduction Algorithm

Parallel Cyclic Reduction (PCR) method [10] is one of the most powerful parallel alternatives of TDMA algorithm for the tridiagonal linear equation system. PCR only has forward reduction calculation and, in each step, it reduces the current system into two systems with half size, and finally it solves M (domain size) one-equation systems simultaneously. PCR requires $\log_2(M)$ algorithmic steps and M units of work per step [17]. The data flow process has been explained in detail in references [16, 21].

4. Parallel ADI Method

This section discusses the parallelization of ADI solver on GPUs using NVIDIA CUDA. We begin with an existing GPU implementation of PCR solver which is a key component of our proposed ADI solver. We then present our improved implementation that addresses the limitations of existing implementations. We detail several optimization techniques to further improve the performance. For all implementations, we copy data once from CPU to GPU and keep all data in GPU memory until we copy the final result back to CPU.

4.1. Existing PCR Implementation

In Zhang's GPU implementations [16, 17], PCR kernel is used to solve hundreds of tridiagonal systems simultaneously. Each equation system is mapped to one thread block, and then each thread inside the block solves one equation. This kernel is divided into three components as follows.

1. Data is preloaded into shared memory. In their implementations, there are five arrays (three for the matrix diagonals, one for right-hand-side term, and one for solution vector).
2. From the first reduction step to step $\log_2(M/2)$, the coefficients for calculating the M one-equation systems are updated.
3. In step $\log_2(M)$, one-equation systems are solved concurrently. Results are temporarily stored in shared memory, and finally copied to global memory.

In this design, a system size more than 512 would exceed the size of shared memory on current hardware as pointed out in [16]. Zhang later implemented hybrid PCR solvers that support system sizes up to 1024×1024 [17], but the restriction of shared memory size is still unsolved.

4.2. Improved PCR Implementation

In this section, we present our approach to eliminate the domain size limitation of the existing PCR implementation. We also present how we reduce the amount of shared memory used.

4.2.1. Mapping strategy

To make the kernel independent of limited hardware resources on GPUs, the scalable two-level thread mapping model should be fully utilized. In Zhang's implementations [16, 17], the whole system is mapped to one thread block as shown in Fig. 2(a). Because of limited shared memory size, this mapping method can not handle a large domain size. To address this issue, we propose two flexible mapping methods as follows.

The first proposed mapping method [15] still maps the whole system to one thread block. Unlike previous, however, this mapping method solves the whole system by iterating equally-chunked sections as shown in Fig. 2(b). Since the thread block satisfies the shared memory size requirement by reusing the shared memory, large system sizes can be solved. Although this mapping method can be applied to massive data computation, one potential limitation would be that the total number of thread blocks may be too small to fully utilize the hardware. For example, given the domain sizes in Fig. 1 and the current sub-step of ADI solver sweeps horizontally, the total number of thread blocks would be N . Unless N is big, the kernel wouldn't be able to fully utilize the hardware pipelines and resources.

A more flexible and scalable mapping method is to map the whole system to multiple blocks as Fig. 2(c). In this way, the thread block size and the amount of shared memory used is not affected by the system size. As a result, the system size can be increased as long as the graphics memory size allows. This scheme allows GPU resources to be equally well utilized at all times.

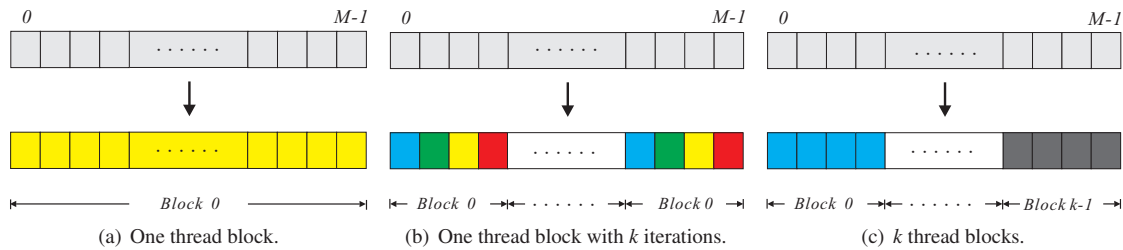


Fig. 2. Three different thread mapping techniques. A whole system is mapped to (a) one thread block, (b) one thread block with multiple iterations, (c) multiple thread blocks.

4.2.2. Shared memory reduction

On-chip shared memory provides a lot higher bandwidth and lower latency than global memory and, therefore, any opportunity to preload data from global memory to this memory space should be exploited. At the same time, as the usage of shared memory influences the hardware occupancy, one should not overuse. In general, low shared memory usage gives higher occupancy and better performance by allowing more threads launch when the shared memory is a limiting factor.

We observed that Zhang's implementations [16, 17] allocate one extra array in shared memory to store temporary solution. However, this array is just a temporary buffer which is not associated with any computation in later part of the kernel but just copied to global memory. In our implementation, following the proposal by Zhang and Jia [15] we remove this unnecessary shared memory allocation and directly write solution to global memory. With this, the total amount of shared memory used is reduced by 20% over time. We label our proposed PCR solvers as shown in Table 1, which summarizes the configurations of four possible combinations between this shared memory reduction and newly proposed thread mappings described in Section 4.2.1.

4.3. Data Access Analysis

Data access pattern is one of the most important performance factors on data-parallel GPUs and we briefly discuss the memory access patterns of both serial and parallel ADI solvers in this section.

Table 1. Four improved PCR solver configurations.

| Configurations | Descriptions |
|----------------|---|
| C1 | Large shared memory is used (5 arrays) and the whole system is mapped to one block. |
| C2 | Large shared memory is used (5 arrays) and the whole system is mapped to multiple blocks. |
| C3 | Small shared memory is used (4 arrays) and the whole system is mapped to one block. |
| C4 | Small shared memory is used (4 arrays) and the whole system is mapped to multiple blocks. |

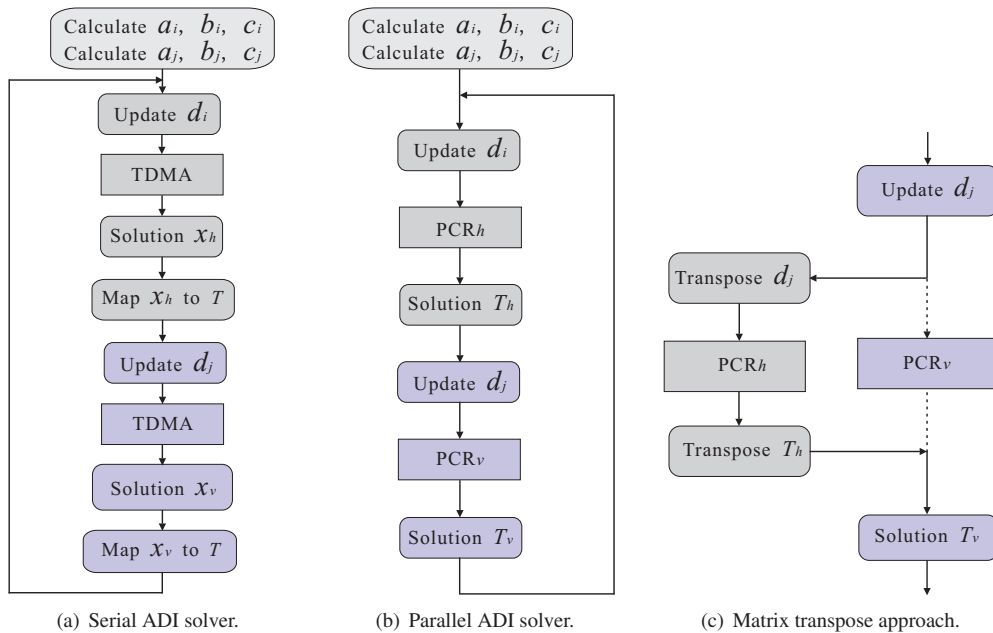


Fig. 3. Data flow diagrams.

4.3.1. Data access in serial solver

The overall data flow of the serial ADI solver is shown in Fig. 3(a). All arrays are linearly stored in physical memory space in both horizontal and vertical ADI sweeps. The solution variable T is updated in each sweep and it is immediately used to calculate the new linearly stored right-hand-side term in the other sweep (See Equations (4) and (5)), so the linear solution result x from TDMA is mapped back to the two-dimensional array T for simplicity. Since arrays are stored in row-major order in C/C++ program, this strategy fits well into the calculation of vertical right-hand-side term, but not so well for that in the horizontal direction.

4.3.2. Data access in parallel solver

On GPU-accelerated platforms where computation intensive portions of application are offloaded onto GPU over PCI express bus, application should minimize data transfer between the host (CPU) and the device (GPU) [11]. Therefore, the best scenario is that the data is only copied to the device once at the beginning, and it is kept until the end of simulation. More importantly, coalesced memory access should be always desired, as accessing off-chip DRAM is very expensive. Initially, the computation flow shown in Fig. 3(a) was implemented on GPU with just TDMA replaced with PCR but it gave poor performance because of the slow data copies between one-dimensional and two-dimensional arrays. Therefore, a new computation flow shown in Fig. 3(b) is adopted, in which all arrays are linearly defined horizontally for both sweeps. For the horizontal sweep, although the discrete global memory access for solution variable T still exists for calculation of right-hand-side term d_i (See Equation (4)), the global memory access in kernel PCR_h and memory access for calculation of vertical right-hand-side term d_j are fully coalesced (See Equation (5)). However, in the vertical sweep, global memory prefetching at the beginning of the

kernel PCR_v is still not coalesced. After prefetched, shared memory is used for the rest of calculation and the performance penalty from uncoalesced access has been alleviated. This initial parallelized ADI solver is named as *baseline* implementation compared with two optimized versions, hereafter.

4.4. ADI Memory Access Optimizations

As mentioned in previous section, the baseline implementation suffers from uncoalesced memory access in one of two sweeps. As a result, there is a significant performance unbalance between two sweeps (this is clearly shown in Section 5.1). To address this performance bottleneck, we propose and experiment two techniques: leveraging texture memory and matrix transpose.

4.4.1. Texture memory

Designed around graphics processing, GPUs have a special hardware block called *texture unit* to efficiently access two-dimensional texture images. Off-chip memory access via this texture unit is referred to as texture memory access, which offers several benefits such as hardware address calculation, filtering, and higher uncoalesced memory access performance insensitivity. In our case, we take advantage of higher uncoalesced memory access performance insensitivity which is, when memory access pattern is uncoalesced, texture memory gives better performance than default global memory.

In the vertical sweep of ADI solver, two neighbor threads in the same warp access global memory addresses with stride of domain size in the horizontal direction. This memory access pattern results in multiple not-fully-utilized memory transactions as opposed to one large fully-utilized transaction in coalesced case. Texture memory provides better cache efficiency as its two-dimensional spatial locality aware access works better in this strided case, resulting in better overall performance. In our implementation, coefficient arrays a, b, c, d , and solution T are bound with texture memory, and all accesses to these arrays go through texture unit. Once properly bound, kernel requires only small modifications yet its performance return is not trivial.

4.4.2. Matrix transpose

One of the widely used techniques to handle strided access patterns on GPUs is to change data layout. A downside of this approach is an inevitable overhead associated with data transformation. In parallel ADI solver, simple matrix transpose between vertical and horizontal directions is sufficient to resolve inefficient (i.e., uncoalesced) memory accesses. Specifically, four input coefficient arrays a_j, b_j, c_j and d_j are firstly transposed to a_i, b_i, c_i and d_i horizontally. Then the same solver PCR_h as indicated in Fig. 3(b) will solve the tridiagonal linear equation system for vertical direction instead of PCR_v with coalesced memory access. Finally the solution T_h is transposed back to the original T_v . In general, there are five times matrix transpose operations for two-dimensional application, however, in our specific testcase with uniform mesh size as presented in Section 5, coefficient arrays a_j and a_i have same values in memory space (See Equations (4) and (5)), this relation is also observed between b_j (c_j) and b_i (c_i). Therefore, a data flow as shown in Fig. 3(c) with twice matrix transpose operations is implemented in this research.

The performance of this approach significantly depends on the degree of matrix transpose optimization. Matrix transpose is not a GPU-friendly algorithm in nature. NVIDIA CUDA SDK provides matrix transpose examples [25] which are highly tuned for their underlying hardware architecture. We use their final version that implements the highest level of optimizations which includes memory coalescing, shared memory bank conflict, partition capping, and tiling.

5. Performance Analysis

In this section, we evaluate our proposed parallel ADI solver using a two-dimensional heat conduction simulation with large domain sizes. Our test machine is configured with an Intel Core i7-3770K CPU running at 3.5GHz, 16GB system memory, a NVIDIA GTX 680 GPU with 2GB graphics memory and 48KB per-multiprocessor shared memory (Kepler architecture), CUDA 4.2, and Ubuntu 11.10 operation system.

Simulation parameters are configured as follows. Mesh size is uniform in both directions with $\Delta x = \Delta y = 0.01$ m and numerical time step Δt is 0.01 s, and heat conduction time is 1.0 s, therefore the total numerical time

Table 2. Speedup comparison of parallel ADI solver over serial solver for four domain sizes with thread block size of 128.

| Sizes | Baseline | | | | Texture memory | | | | Matrix transpose | | | |
|-------------------|----------|------|------|------|----------------|-------|------|-------|------------------|-------|-------|-------|
| | C1 | C2 | C3 | C4 | C1 | C2 | C3 | C4 | C1 | C2 | C3 | C4 |
| 1024 ² | 33.3 | 77.2 | 33.4 | 78.8 | 45.0 | 103.1 | 45.8 | 103.0 | 112.3 | 118.0 | 109.6 | 119.6 |
| 2048 ² | 21.3 | 59.4 | 21.4 | 60.4 | 24.9 | 58.9 | 25.8 | 58.4 | 97.1 | 100.1 | 96.1 | 102.2 |
| 4096 ² | 15.7 | 51.0 | 15.7 | 51.9 | 18.0 | 63.4 | 18.3 | 63.2 | 78.9 | 80.1 | 77.8 | 82.8 |
| 8192 ² | 12.2 | 43.8 | 12.1 | 44.3 | 13.8 | 52.2 | 14.1 | 51.4 | 68.7 | 69.4 | 67.4 | 71.5 |

step is 100. To present more realistic data which was not possible with other existing parallel solvers, we tested four cases with large domain sizes of 1024×1024, 2048×2048, 4096×4096, and 8192×8192 with five different thread block configurations (32, 64, 128, 256, and 512). Thread block configuration plays an important role in determining hardware occupancy/utilization and memory performance. In all implementations, single-precision floating point data type was used.

5.1. Baseline Implementation

The first large column in Table 2 shows the speedup of baseline parallel ADI solver over the serial ADI for four domain sizes with the thread block size of 128, which gave overall best performance over other block sizes in our experiment. We have several observations from the results. 1) there is a significant performance difference between two thread mapping methods. Multi-block mappings (C2 and C4) outperform one-block mappings (C1 and C3) by a large margin (approximately 250%). 2) shared memory reduction (C3 and C4) does not have any notable impact on the performance. This, however, does not mean that reducing shared memory usage is useless. Shared memory is an important on-chip hardware resource that can limit the number of thread blocks that can be launched simultaneously. Although shared memory is not a limiting factor in our experiment, this can be a big issue on GPUs that have less shared memory (e.g., earlier generation or low-end GPUs). 3) ADI solver's performance decreases as the domain sizes increase for all configurations. This degradation is caused by the impact/contribution of less efficient memory access patterns in the vertical sweep. PCR_v accounts for nearly 70% of total execution time (Fig. 4(a)), as a consequence, overall performance is significantly affected by this module. PCR_v loads two vertically consecutive data which are in two different horizontal lines with the stride of horizontal domain size. As the horizontal domain size (i.e., the stride) increases, this access pattern causes more serious performance degradation.

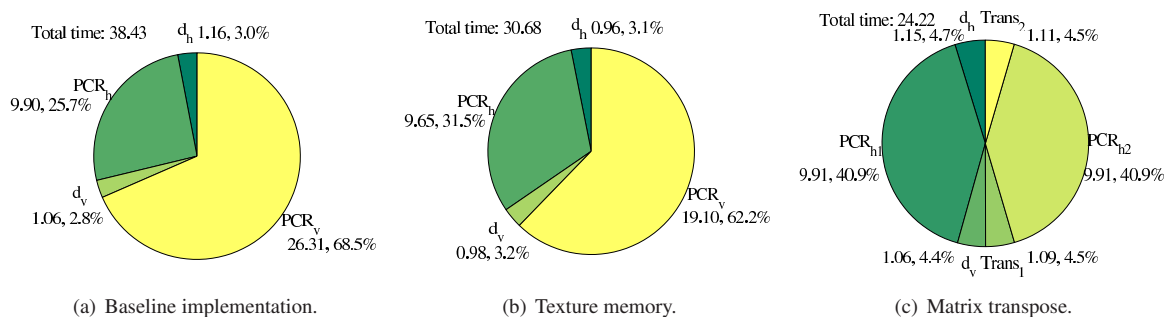


Fig. 4. Time breakdown of three implementations of ADI solver (domain sizes: 4096² and thread block size: 128). Time in milliseconds.

5.2. Optimized Implementations

On top of baseline implementation, two memory optimization techniques (texture memory and matrix transpose) were further implemented. The second and third large columns in Table 2 show their performance, respectively. Compared with the baseline implementation, the optimization techniques improve the performance

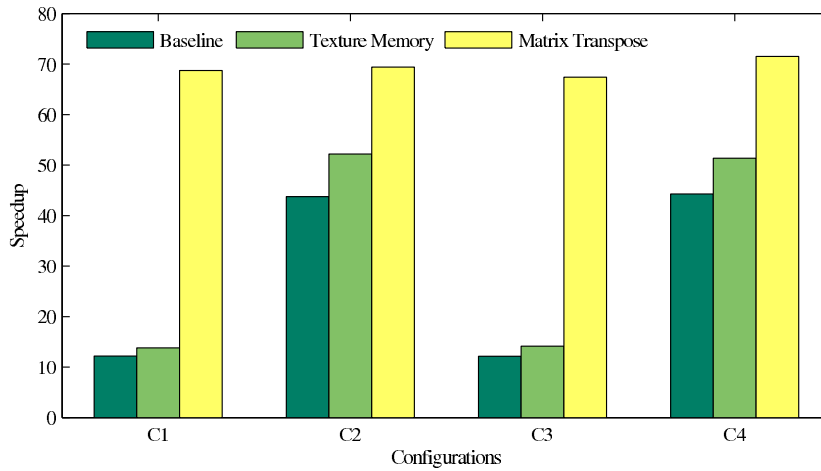


Fig. 5. Performance comparison among the proposed configurations (domain sizes: 8192^2 and thread block size: 128).

significantly. Comparing two techniques, matrix transpose approach shows better performance than texture memory in our experiment. However, this can change if more arrays need to be transposed (please note that our specific experiment does not require the transpose of the rest of arrays other than two, d_j and T_h , see Fig. 3(c)). Also three-dimensional domain would require more matrix transpose operations which would make this approach less attractive. Therefore, these approaches should be carefully chosen based on the target system configuration.

Fig. 4(b) and Fig. 4(c) present the detailed time breakdown of two optimization techniques. It clearly shows that texture fetch improves the performance of PCR_v kernel in particular. On the other hand, matrix transpose replaces the bad performing PCR_v with PCR_h , making their contributions the same. Note also that the approach introduces the overhead of twice matrix transpose operations (denoted as $Trans_1$ and $Trans_2$ in Fig. 4(c)).

Finally, a side-by-side comparison of solver performance with large domain sizes 8192×8192 is shown in Fig. 5. This result demonstrates that our proposed parallel ADI solvers (C2) and (C4) are very efficient even with large domain sizes.

5.3. Numerical Accuracy Analysis

Scientific computation requires a certain level of numerical accuracy. We check the accuracy of our proposed GPU solver using norm error L^2 which is defined as

$$Error = \sqrt{\frac{\sum_{i=1}^M (T_{GPU}(i) - T_{CPU}(i))^2}{\sum_{i=1}^M (T_{CPU}(i))^2}} \quad (10)$$

where M is total number of nodes, $T_{CPU}(i)$ and $T_{GPU}(i)$ are the temperature values computed by CPU and GPU ADI solvers, respectively.

With single-precision floating point data type used in both versions, the truncated error is expected to be about 10^{-7} . Table 3 shows the error analysis result of configuration C4 with thread block size of 128. As the mesh size is kept uniform in our testcase, the norm error of our GPU ADI solver increases slightly as the domain sizes become larger, but the norm error is still smaller than 3.0×10^{-6} with 100 time steps. We believe the accuracy of GPU ADI solver is within the acceptable range in most scientific computation researches.

Table 3. Norm errors of configuration C4 with thread block size of 128.

| Domain sizes | 1024 ² | 2048 ² | 4096 ² | 9182 ² |
|-----------------------------|-------------------|-------------------|-------------------|-------------------|
| Errors (x10 ⁻⁶) | 2.87 | 2.90 | 2.92 | 2.93 |

6. Conclusion

In this paper, we presented an improved parallel ADI solver that harnesses modern GPUs for solving the two-dimensional heat conduction equation. Our improvements include 1) new thread mappings which address the hardware resource constraints, 2) reduced shared memory usage which helps launch more threads, 3) two memory optimization techniques which improve the efficiency of off-chip memory accesses. With these improvements, our proposed parallel ADI solver demonstrates a significant speedup across large computation domain sizes. Our future work includes developing more advanced GPU optimization techniques and applying our improved ADI solver to more complex systems.

Acknowledgements

This work is supported in part by the USDA Agriculture Research Service under the Specific Research Agreement No. 58-6408-1-609 monitored by the USDA-ARS National Sedimentation Laboratory and The University of Mississippi, and the National Science Foundation under award number EPS-0903787.

References

- [1] D. Peaceman, H. Rachford Jr, The numerical solution of parabolic and elliptic differential equations, *Journal of the Society for Industrial & Applied Mathematics* 3 (1) (1955) 28–41.
- [2] V. Casulli, R. Cheng, Semi-implicit finite difference methods for three-dimensional shallow water flow, *International Journal for numerical methods in fluids* 15 (6) (2005) 629–648.
- [3] N. Ellner, E. Wachspress, Alternating direction implicit iteration for systems with complex spectra, *SIAM journal on numerical analysis* 28 (3) (1991) 859–870.
- [4] I. Lindemuth, J. Killeen, Alternating direction implicit techniques for two-dimensional magnetohydrodynamic calculations, *Journal of Computational Physics* 13 (2) (1973) 181–208.
- [5] T. Namiki, A new FDTD algorithm based on alternating-direction implicit method, *Microwave Theory and Techniques, IEEE Transactions on* 47 (10) (1999) 2003–2007.
- [6] J. Douglas Jr, On the numerical integration of $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \frac{\partial^2 u}{\partial t^2}$ by implicit methods, *Journal of the Society for Industrial & Applied Mathematics* 3 (1) (1955) 42–65.
- [7] W. Wu, *Computational river dynamics*, CRC, 2007.
- [8] R. Hockney, A fast direct solution of Poisson's equation using Fourier analysis, *Journal of the ACM* 12 (1) (1965) 95–113.
- [9] H. Stone, An efficient parallel algorithm for the solution of a tridiagonal linear system of equations, *Journal of the ACM* 20 (1) (1973) 27–38.
- [10] R. Hockney, C. Jesshope, *Parallel computers*. 1981, Adam Hilger.
- [11] Nvidia, *CUDA C programming guide v4. 2*, Nvidia Corp.
- [12] J. Sanders, E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*, Addison-Wesley Professional, 2010.
- [13] B. Jang, D. Kaeli, S. Do, H. Pien, Multi GPU implementation of iterative tomographic reconstruction algorithms, in: *Biomedical Imaging: From Nano to Macro, 2009. ISBI '09. IEEE International Symposium on*, 2009, pp. 185–188.
- [14] Y. Zhang, Y. Jia, Parallelization of implicit CCHE2D model using CUDA programming techniques, in: *EWRI World Environment & Water Resources Congress 2013*, EWRI, Cincinnati, USA, 2013.
- [15] Y. Zhang, Y. Jia, Parallelized CCHE2D model with CUDA Fortran on graphics process units, *Computers and Fluids* (under review).
- [16] Y. Zhang, J. Cohen, J. Owens, Fast tridiagonal solvers on the GPU, *ACM Sigplan Notices* 45 (5) (2010) 127–136.
- [17] Y. Zhang, J. Cohen, A. Davidson, J. Owens, A hybrid method for solving tridiagonal systems on the GPU, *GPU Computing Gems Jade Edition* (2011) 117.
- [18] A. Davidson, J. Owens, Register packing for cyclic reduction: a case study, in: *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ACM, 2011, p. 4.
- [19] A. Davidson, Y. Zhang, J. Owens, An auto-tuned method for solving large tridiagonal systems on the GPU, in: *Parallel & Distributed Processing Symposium (IPDPS)*, 2011 IEEE International, IEEE, 2011, pp. 956–965.
- [20] D. Goddeke, R. Strzodka, Cyclic reduction tridiagonal solvers on GPUs applied to mixed-precision multigrid, *Parallel and Distributed Systems, IEEE Transactions on* 22 (1) (2011) 22–32.
- [21] H. Kim, S. Wu, L. Chang, W. Hwu, A scalable tridiagonal solver for GPUs, in: *Parallel Processing (ICPP)*, 2011 International Conference on, IEEE, 2011, pp. 444–453.
- [22] N. Sakharmykh, Tridiagonal solvers on the GPU and applications to fluid simulation, in: *GPU Technology Conference*, 2009.
- [23] N. Sakharmykh, Efficient tridiagonal solvers for ADI methods and fluid simulation, in: *NVIDIA GPU Technology Conference*, 2010.
- [24] S. Patankar, *Numerical heat transfer and fluid flow*, Taylor & Francis, 1980.
- [25] G. Ruetsch, P. Mickevicius, Optimizing matrix transpose in CUDA, *NVIDIA CUDA SDK Application Note*.