Title

A Hybrid Method for Solving Tridiagonal Systems on the GPU

ISBN

9780123859631

Authors

Zhang, Y

Cohen, J

Davidson, AA

et al.

Publication Date

2012-12-01

DOI

10.1016/B978-0-12-385963-1.00011-3

A Hybrid Method for Solving Tridiagonal Systems on the GPU

Yao Zhang Jonathan Cohen* John D. Owens
University of California, Davis *NVIDIA

1 Background

Tridiagonal linear systems are of importance to many problems in numerical analysis and computational fluid dynamics, as well as to computer graphics applications in video games and computer-animated films. Typical applications require solving hundreds or thousands of tridiagonal systems, which takes a majority part of total computation time. Fast parallel solutions are critical to larger scientific simulations, interactive computations of special effects in films, and real-time applications in video games.

In this article, we study the performance of multiple tridiagonal algorithms on a GPU. We design a novel hybrid algorithm that combines a work-efficient algorithm with a step-efficient algorithm in a way well-suited for a GPU architecture. We also develop a differential technique to measure and analyze the performance of GPU programs. The technique allows us to estimate the cost of critical performance factors such as bank conflicts, computation, synchronization overhead.

We hope this article will benefit parallel computing researchers, computer graphics developers, and general GPU programmers by (1) a detailed description of how an efficient tridiagonal solver is implemented on the GPU, (2) our discussion of hybrid parallelism, which improves overall performance when work-efficient and step-efficient algorithms are both available, and (3) our performance analysis technique that can be applied to other applications as well.

2 Problem Statement

We wish to solve a system of n linear equations of the form $Ax = d$, where A is a tridiagonal matrix

$$A = \begin{pmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & 0 & \\ & a_3 & b_3 & c_3 & & \\ & & \ddots & \ddots & \ddots & \\ & 0 & & \ddots & \ddots & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix}.$$

A sequential Gaussian elimination algorithm requires $O(n)$ computation steps to solve this system. Since the 1960s, a variety of parallel algorithms have been developed to solve a tridiagonal system in $O(\log_2 n)$ steps on vector supercomputers. Notable among these are cyclic reduction (CR) [3], parallel cyclic reduction (PCR) [3], recursive doubling [10], and partition methods [11].

We notice that CR enjoys linear algorithm complexity but suffers from more algorithmic steps and bank conflicts, while PCR has fewer algorithmic steps but do more work each step. To combine the merits of the two algorithms, we propose a hybrid CR-PCR algorithm tailored for the GPU architecture. We also provide detailed performance analysis of all three solvers: CR, PCR, and the hybrid solver, to gain insights into why a certain solver performs better or worse.

3 Related Work

The tridiagonal solver was first implemented on a GPU by Kass et al. [4] to perform efficient depth-of-field blurs. Their solver was based on CR and was written in a GPU shading language. Sengupta et al. implemented CR with CUDA, and applied it to real-time shallow water simulation [5, 7]. Sakharnykh simulated 3D viscid incompressible fluid on the GPU by using the serial Thomas algorithm to solve thousands of tridiagonal systems in parallel [6]. His solver achieves high performance when there are a large number of systems available. Göddeke et al. developed a bank-conflicts-free GPU implementation of CR at the expense of 50% more on-chip storage, and employed it as a line relaxation smoother in a multigrid solver [2]. Egloff developed a GPU-based PCR solver to solve one dimensional PDEs with finite difference schemes [1]. The work by Shishkovtsov et al [9] is most similar to ours in nature. They designed a hybrid solver based on the Thomas algrithm and PCR, and applied it to the simulation of depth-of-field effects. In this article, we expand on and extend our previous work on hybrid tridiagonal solvers [12].

4 Algorithm Review

In this section, we review the classic sequential algorithm, the parallel cyclic reduction algorithm and its variant parallel cyclic reduction.

4.1 The Thomas Algorithm

The Thomas algorithm is Gaussian elimination in the tridiagonal matrix case. The algorithm has two phases, forward elimination and backward substitution. In the first phase, we eliminate the lower diagonal by

$$c'_1 = \frac{c_1}{b_1}, c'_i = \frac{c_i}{b_i - c'_{i-1}a_i}, i = 2, 3, \ldots, n-1$$

$$d'_1 = \frac{d_1}{b_1}, d'_i = \frac{d_i - d'_{i-1}a_i}{b_i - c'_{i-1}a_i}, i = 2, 3, \ldots, n-1$$

The second phase solves all unknowns from last to first:

$$x_n = d'_n, x_i = d'_i - c'_i x_{i+1}.$$

The algorithm is simple, but inherently serial and takes $2n$ computation steps, because the calculation of c'_i, d'_i, and x_i depends on the result of the immediately preceding calculation of c'_{i-1}, d'_{i-1}, and x_{i+1}.

4.2 Cyclic Reduction (CR)

CR consists of two phases, forward reduction and backward substitution. The forward reduction phase successively reduces a system to a smaller system with half the number of unknowns, until a system of 2 unknowns is reached. The backward substitution phase successively determines the other half of the unknowns using the previously solved values.

In each step of forward reduction, we update all even-indexed equations in parallel with equation i of the current system as a linear combination of equations i, $i+1$ and $i-1$, so that we derive a system of only even-indexed unknowns. Equation i has the form $a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i$. The updated values of a_i, b_i, c_i and d_i are

$$a'_i = -a_{i-1}k_1, b'_i = b_i - c_{i-1}k_1 - a_{i+1}k_2$$

$$c'_i = -c_{i+1}k_2, d'_i = d_i - d_{i-1}k_1 - d_{i+1}k_2$$

$$k_1 = \frac{a_i}{b_{i-1}}, k_2 = \frac{c_i}{b_{i+1}}$$

In each step of backward substitution, we solve all odd-indexed unknowns x_i in parallel by substituting the already solved x_{i-1} and x_{i+1} values to equation i,

$$x_i = \frac{d'_i - a'_i x_{i-1} - c'_i x_{i+1}}{b'_i}.$$

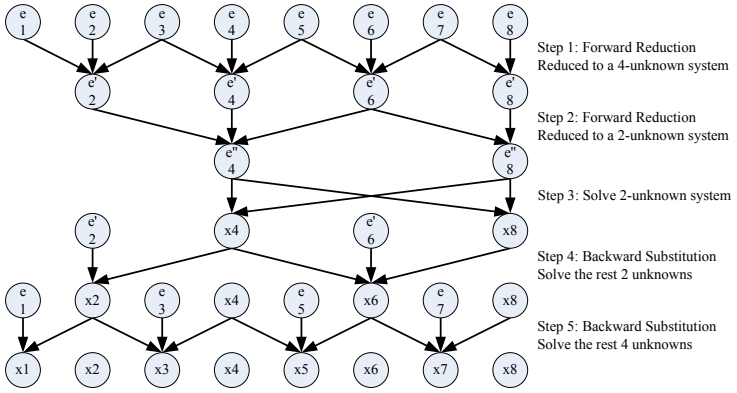

Figure 1: Communication pattern for CR in the 8-unknown case, showing the dataflow between each equation, labeled $e1$ to $e8$. Letters e' and e'' stand for updated equation.

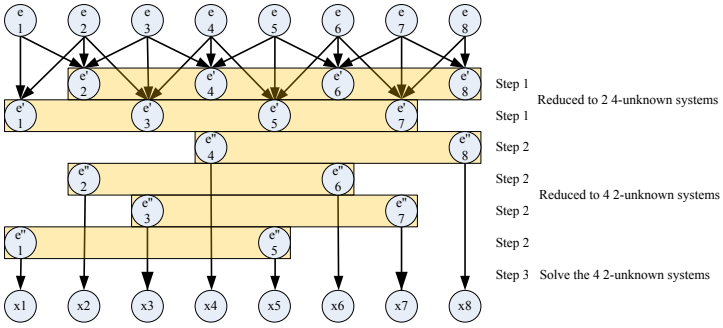

Figure 2: Communication pattern for PCR in the 8-unknown case, showing the dataflow between each equation, labeled $e1$ to $e8$. Letters e' and e'' stand for updated equations. Equations in a yellow rectangle form a system. We omit the arrows in step 2 for clarity.

Note that for the sake of simplicity, in the above description, we disregard the special treatment of the last equation and the first unknown respectively in the two algorithm phases. Also, we solve a 2-unknown system between the two algorithm phases. Figure 1 shows the communication pattern of the algorithm for an 8-unknown system.

Both the parallel CR algorithm and the serial Thomas algorithm perform a number of operations that are linear in the number of unknowns. The Thomas algorithm performs $8n$ operations while CR performs $17n$ operations. However, on a parallel computer with n processors, CR requires $2\log_2 n$ steps while the Thomas algorithm requires $2n$ steps.

4.3 Parallel Cyclic Reduction (PCR)

PCR is a variant of CR. In contrast to CR, PCR only has the forward reduction phase. Although the reduction mechanism and formula are the same as those of CR, in each reduction step, PCR reduces each of the current systems to two systems of half size. For example, for an 8-unknown system, we reduce it to two 4-unknown systems in step 1 (see Figure 2), then further reduce the two 4-unknown systems to four 2-unknown systems in step 2, and finally solve the four systems in step 3. PCR takes $12n\log_2 n$ operations and $\log_2 n$ steps to finish. PCR requires fewer algorithmic steps than CR but does asymptotically more work per step.

5 A Hybrid Solver

First, we call CR a work-efficient algorithm and PCR a step-efficient algorithm, because CR requires $17n$ arithmetic operations in $2\log_2 n$ algorithmic steps, while PCR requires $12n\log_2 n$ operations in $\log_2 n$ steps. CR has inefficient steps

(a) CR: $2\log_2 8 = 6$ steps.
The vector unit is partially idle during the middle three steps.
Less work per step, more steps.

(b) PCR: $\log_2 8 = 3$ steps.
The vector unit is fully utilized across all steps.
More work per step, fewer steps.

(c) Hybrid: $2 + \log_2 4 = 4$ steps.
The vector unit is fully utilized across all steps.
Less work per step than PCR, fewer steps than CR.

Figure 3: Comparison between the CR, PCR and hybrid algorithms in terms of algorithmic steps and work per step for solving an 8-unknown system. A dot stands for a unit of work, and a row of dots stands for an algorithmic step. We assume the length of vector unit is 4 in this example.

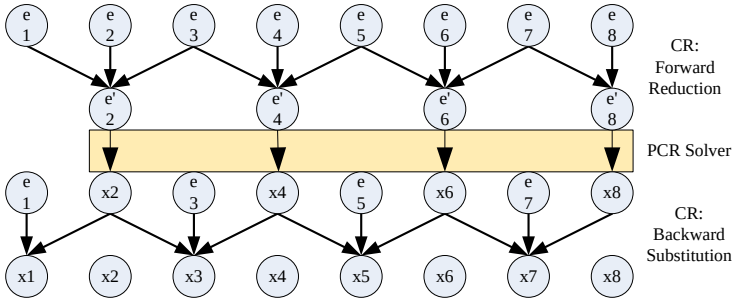

Figure 4: Hybrid algorithms for an 8-known system. Letter e stands for equation, and e' stands for updated equation.

where it lacks parallelism to keep the vector unit fully utilized (the amount of parallelism needs to be at least the warp size 32 in the CUDA case); PCR takes longer time per step, although it always keeps the vector unit in full occupancy. This observation motivates a hybrid method. By switching from CR to PCR when there is insufficient parallelism, we are able to reduce both the amount of work and the inefficient algorithmic steps as shown in Figure 3. Sengupta et al. use a similar idea to combine a step-efficient scan algorithm with a work-efficient one for implementation on a GPU [8].

The hybrid algorithm first reduces the system to a certain size using the forward reduction phase of CR, then solves the reduced (intermediate) system with PCR. Finally, it substitutes the solved unknowns back into the original systems using the backward substitution phase of CR. Figure 4 shows the hybrid method schematically. In this example, we switch to PCR before the forward reduction has reached a 2-unknown system. PCR enables us to finish the inefficient middle steps more quickly, because they have fewer algorithmic steps than CR.

6 Implementation

To solve hundreds of tridiagonal systems simultaneously, we map the solver to the GPU's two-level hierarchical architecture with systems mapped to blocks and equations mapped to threads. We use shared memory to keep most communication between threads on chip. Although all parallel algorithms are general for any system size, our solvers only handle a power-of-two system size, which makes thread numbering and address calculation simpler. In the following subsections, we will walk through the implementation details of the tridiagonal solvers.

6.1 Cyclic Reduction

We first setup the kernel execution parameters as shown in Listing 1. The number of CUDA blocks is the number of systems, since every single system is mapped to one CUDA block. The number of threads per block is half the number of equations per system, since we start with updating only even-indexed equations in the first algorithmic step. The template parameter T allows us to choose either single-precision or double-precision floating point data type.

We use five global arrays to store three matrix diagonals (a: lower diagonal, b: main diagonal, c: upper diagonal), one right-hand side d, and one solution vector x. These five arrays store the data of all systems continuously, with the data of the first system stored at the beginning of the arrays, followed by the second system, the third system, and so on. The amount of shared memory we allocate for each CUDA block or each tridiagonal system is systemSize * 5 * sizeof(T) bytes.

Listing 1: Kernel setup for CR

```
dim3  grid(numSystems, 1, 1);
dim3  threads(sizeSystem/2, 1, 1);
crKernel<<< grid, threads, systemSize * 5 * sizeof(T) >>>(a, b, c, d, x);
```

The forward reduction phase (Listing 2) takes $\log_2(sizeSystem/2)$ algorithmic steps. All five arrays a, b, c, d, and x are shared memory arrays with data loaded from global memory. The stride starts with 1, and double its value every algorithmic step. The number of active threads decreases by half every algorithmic step. We always enable contiguously ordered threads as active threads so that we do not have unnecessary divergent branches within a SIMD unit.

To unify the special treatment to the last equation, we set iRight to systemSize - 1 if iRight is equal or larger than systemSize. This technique takes the advantage of the fact that c[systemSize - 1] = 0, and more and more zeros are introduced to the array c along the algorithmic steps.

Listing 2: Forward reduction in CR

```
int stride = 1;
for (int j = 0; j < numSteps; j++)
{
    __syncthreads();
    stride *= 2;
    int delta = stride / 2;
    if (threadIdx.x < numThreads)
    {
        int i = stride * threadIdx.x + stride - 1;
        int iLeft  = i - delta;
        int iRight = i + delta;
        if (iRight >= systemSize) iRight = systemSize - 1;

        T tmp1 = a[i] / b[iLeft];
        T tmp2 = c[i] / b[iRight];
        b[i] = b[i] - c[iLeft] * tmp1 - a[iRight] * tmp2;
        d[i] = d[i] - d[iLeft] * tmp1 - d[iRight] * tmp2;
        a[i] = -a[iLeft] * tmp1;
        c[i] = -c[iRight] * tmp2;
    }

    numThreads /= 2;
}
```

At the beginning of the backward substitution phase (Listing 3), we solve a 2-unknown system produced at the final stage of forward reduction. Then we have $\log_2(sizeSystem/2)$ algorithmic steps to solve all rest unknowns. We double the number of active threads every algorithmic step.

Listing 3: Backward substitution in CR

```
if (threadIdx.x < 2)
```

```
{
    int addr1 = stride - 1;
    int addr2 = 2 * stride - 1;
    T tmp3 = b[addr2] * b[addr1] - c[addr1] * a[addr2];
    x[addr1] = (b[addr2] * d[addr1] - c[addr1] * d[addr2]) / tmp3;
    x[addr2] = (d[addr2] * b[addr1] - d[addr1] * a[addr2]) / tmp3;
}

numThreads = 2;
for (int j = 0; j < numSteps; j++)
{
    int delta = stride/2;
    __syncthreads();
    if (threadIdx.x < numThreads)
    {
        int i = stride * threadIdx.x + stride / 2 - 1;
        if(i == delta - 1)
            x[i] = (d[i] - c[i] * x[i + delta]) / b[i];
        else
            x[i] = (d[i] - a[i] * x[i - delta] - c[i] * x[i+delta]) / b[i];
    }
    stride /= 2;
    numThreads *= 2;
}
```

6.2 Parallel Cyclic Reduction

As in CR, we set the number of CUDA blocks to the number of systems. But in contrast to CR, we set the number of threads per block to the number of equations per system, since in PCR we update all equations, rather than half equations, during an algorithmic step (Listing 4). Furthermore, the number of active threads keeps constant as the number of equations through all algorithmic steps (Listing 5).

To unify the special treatment to the first equation and the last equation, we set iRight to systemSize - 1 if iRight is equal or larger than systemSize, and iLeft to 0 if iLeft is equal or less than 0 (Listing 5). This technique takes the advantage of the fact that a[0] = 0 and c[systemSize - 1] = 0, and more and more zeros are introduced to the array a and c along the algorithmic steps.

Listing 4: Kernel setup for PCR

```
dim3  grid(numSystems, 1, 1);
dim3  threads(sizeSystem, 1, 1);
pcrKernel<<< grid, threads, systemSize * 5 * sizeof(T) >>>(a, b, c, d, x);
```

Listing 5: PCR

```
for (int j = 0; j <numSteps; j++)
{
    int i = threadIdx.x;
    int iRight = i+delta;
    if (iRight >= systemSize) iRight = systemSize - 1;
    int iLeft = i-delta;
    if (iLeft < 0) iLeft = 0;

    T tmp1 = a[i] / b[iLeft];
    T tmp2 = c[i] / b[iRight];
    bNew = b[i] - c[iLeft] * tmp1 - a[iRight] * tmp2;
    dNew = d[i] - d[iLeft] * tmp1 - d[iRight] * tmp2;
    aNew = -a[iLeft] * tmp1;
    cNew = -c[iRight] * tmp2;
```

```
    __syncthreads();
    b[i] = bNew;
    d[i] = dNew;
    a[i] = aNew;
    c[i] = cNew;
    delta *=2;

    __syncthreads();
}

if (threadIdx.x < delta)
{
    int addr1 = threadIdx.x;
    int addr2 = threadIdx.x + delta;
    T tmp3 = b[addr2] * b[addr1] - c[addr1] * a[addr2];
    x[addr1] = (b[addr2] * d[addr1] - c[addr1] * d[addr2]) / tmp3;
    x[addr2] = (d[addr2] * b[addr1] - d[addr1] * a[addr2]) / tmp3;
}
```

6.3 The Hybrid Solver

The implementation of hybrid solver is straightforward once we have the implementations of CR and PCR. To prepare for the PCR stage, we copy the data of the intermediate system to another contiguous space in shared memory. The copy takes little time and extra storage space for the intermediate system, but makes the solver more modular, so that we can directly plug the PCR solver into the intermediate system.

7 Performance Results

Our test platform uses a 2.5 GHz Intel Core 2 Q9300 quad-core CPU, a GTX 280 graphics card with 1 GB video memory, CUDA 2.0 and the CentOS 5 Linux operating system.

We experimented with different sizes of intermediate systems to find the sweet spot for the hybrid solver to switch from CR to PCR as shown in Figure 5. The hybrid solver outperforms CR and PCR in all cases with 256 as the best switch point.

Figure 6 compares the results of various GPU and CPU solvers. The CPU solvers include a Gaussian elimination tridiagonal solver without pivoting (GE), a multi-threaded GE solver (MT), and a Gaussian elimination tridiagonal solver with pivoting (GEP). The MT solver is an OpenMP implementation developed by us with multiple threads solving multiple systems simultaneously. We use four threads for the MT solver with each thread running on one CPU core.

The CPU-GPU data transfer takes a significant amount of time as shown by the yellow bar in Figure 6. Whether or not we should include the cost of the CPU-GPU data transfer depends on the application scenario. Transfer time can be ignored if the GPU solver is used locally as a part of a GPU program (for example, GPU fluid simulation [7] and depth-of-field effects [4]). It should be considered if the GPU solver is used as an accelerator for a part of a CPU program.

8 Performance Analysis

We use a differential technique to measure the time for each part of the algorithm. We first comment out the whole code, then uncomment it incrementally in program order and measure execution time. Finally, we calculate the time difference between all neighboring timing results. This way we obtain the time spent on each algorithmic step. To estimate the overhead of bank conflicts, we replace the memory access pattern with a bank-conflicts-free pattern, and then measure the timing difference.

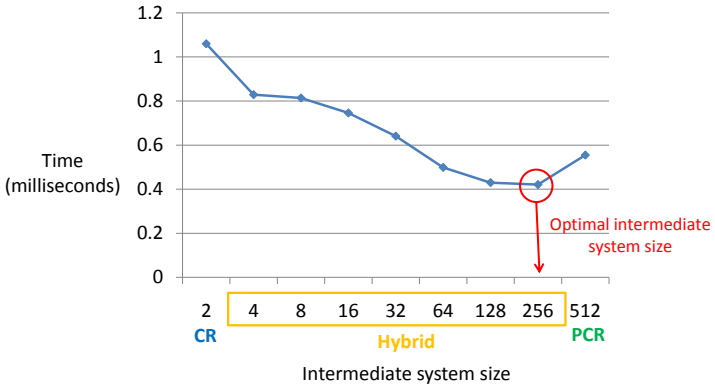

Figure 5: Timings for the hybrid solver with various intermediate system sizes. Problem size: 512 512-unknown systems. Endpoints mark non-hybrid implementations; each point in the middle is a hybrid implementation.

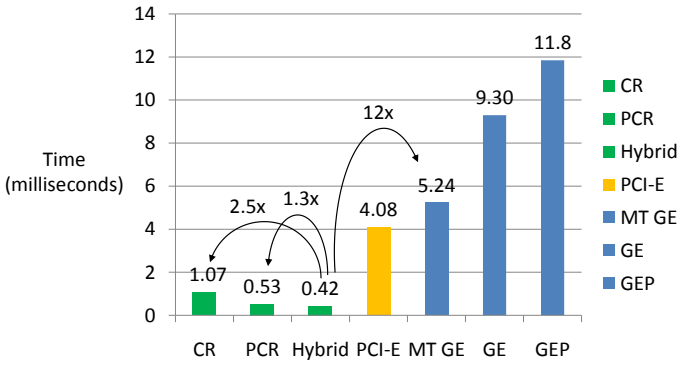

Figure 6: Performance comparison of various GPU and GPU solvers. Problem size: 512 512-unknown systems. PCI-E: CPU-GPU PCI-Express data transfer. MT GE: a four-threaded CPU Gaussian elimination solver. GE: a CPU Gaussian elimination solver. GEP: a CPU Gaussian elimination solver with pivoting from LAPACK. All CPU and GPU solvers use single-precision floating point arithmetic.

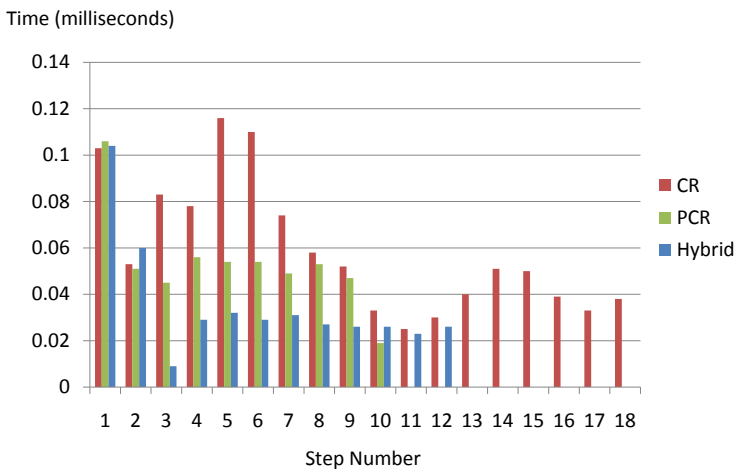

Figure 7: Runtime breakdown by algorithmic steps. Problem size: 512 512-unknown systems. CR requires 18 steps including including 1 step for global memory access, 8 forward reduction steps and 9 backward substitution steps. PCR requires 9 steps including 1 step for global memory access and 8 forward reduction steps. The hybrid solver requires 12 steps including 1 step for global memory access, 1 CR forward reduction step, 1 step for data copy of the intermediate system, 8 PCR step, and 1 CR backward substitution step. (For simplicity, we count the global memory read and write as one step)

8.1 Runtime Breakdown

Figure 7 shows how much time is spent on each step of CR, PCR and the hybrid solver. PCR is twice as fast as CR, mainly because it has half the number of steps of CR. Although PCR's theoretical amount of work per step more than CR's, PCR takes less time per step in reality, which results from the fact that CR suffers from bank conflicts and control overhead, as will be discussed in the next subsection.

The hybrid solver achieves the best runtime by making a balance between the number of steps and the amount of work per step. It first reduces the system size by half by CR's forward reduction, then solve the reduced-size system by PCR, and in the end compute the other half of unknowns by CR's backward substitution. The hybrid solver takes much less time per step than PCR and CR, because it mostly works on the reduced-size system using PCR, which is free from bank conflicts and has fewer algorithmic steps and thus less control overhead. The system size reduction is at the cost of more CR steps, but overall saves the total computation time.

8.2 Bank Conflicts and Control Overhead

Bank conflicts in shared memory access turn out to have a significant impact on the performance of CR. Figure 8 shows that the penalty of bank conflicts can be as high as 4.8x in the forward reduction phase of CR. Apart from bank conflicts, we found that the synchronization overhead included in each algorithmic step is substantial. In the forward reduction phase of CR, we reduce the amount work by half every step, but the execution time almost remains same (blue bars in Figure 8), which means the overall execution time depends primarily on the control and synchronization overhead included in algorithmic steps rather than the amount of work per step. The significant overhead of bank conflicts and synchronization makes the CR-to-PCR switching in the hybrid solver even more preferable besides the work-efficiency and step-efficiency point of view.

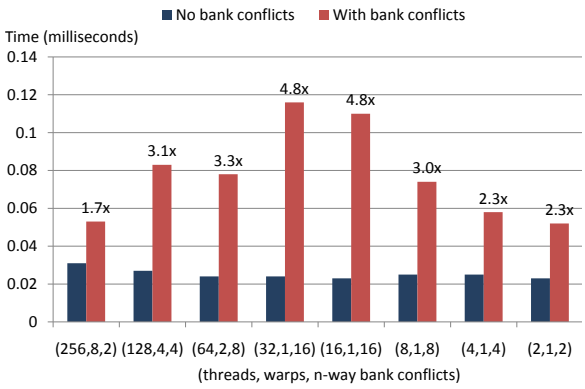

Figure 8: Bank conflicts' impact on performance in the forward reduction phase. Problem size: 512 512-unknown systems.

9 Additional Work

If the editors are interested, we would like to add to the article two on-going extensions of this work, (1) performance auto-tuning, and (2) the PCR-Thomas hybrid algorithm.

(1) Performance auto-tuning: we would like to generate the optimal configuration of CUDA block size and hybrid algorithm switch point, according to different tridiagonal system sizes and GPU models.

(2) The PCR-Thomas hybrid algorithm: the work-efficient Thomas algorithm and step-efficient PCR can produce another hybrid algorithm. Our experiment shows that the hybrid PCR-Thomas solver achieves similar performance with the hybrid CR-PCR solver. The PCR-Thomas solver was used in the video game Metro 2033 for the simulation of depth-of-field effects [9].

10 Summary

We have discussed hybrid parallelism in the GPU context and demonstrated an efficient hybrid tridiagonal solver that achieves 2.5x, 1.3x and 12x speedup over CR, PCR, and a multi-threaded CPU solver respectively. We develop a differential analysis technique for performance analysis. By applying this technique to tridiagonal solvers, we have shown how and why the superior performance can be achieved with a hybrid solver.

Source Code

We have integrated the solvers into CUDA Data Parallel Primitives Library (CUDPP). The source code is available at:

`http://cudpp.googlecode.com/svn/branches/tridiagonal/cudpp/`

References

[1] Daniel Egloff. High performance finite difference PDE solvers on GPUs, February 2010. `http://download.quantalea.net/fdm_gpu.pdf`.

[2] Dominik Göddeke and Robert Strzodka. Cyclic reduction tridiagonal solvers on GPUs applied to mixed precision multigrid. *IEEE Transactions on Parallel and Distributed Systems, Special Issue: High Performance Computing with Accelerators*, March 2010.

[3] R. W. Hockney and C. R. Jesshope. *Parallel Computers*. Adam Hilger, Bristol, 1981.

[4] Michael Kass, Aaron Lefohn, and John D. Owens. Interactive depth of field using simulated diffusion. Technical Report 06-01, Pixar Animation Studios, January 2006.

[5] Michael Kass and Gavin Miller. Rapid, stable fluid dynamics for computer graphics. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, pages 49–57, August 1990.

[6] Nikolai Sakharnykh. Tridiagonal solvers on the GPU and applications to fluid simulation, September 2009. `http://www.nvidia.com/content/GTC/documents/1058_GTC09.pdf`.

[7] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In *Graphics Hardware 2007*, pages 97–106, August 2007.

[8] Shubhabrata Sengupta, Aaron E. Lefohn, and John D. Owens. A work-efficient step-efficient prefix sum algorithm. In *Proceedings of the 2006 Workshop on Edge Computing Using New Commodity Architectures*, pages D–26–27, May 2006.

[9] Oles Shishkovtsov and Ashu Rege. DX11 effects in Metro 2033, March 2010. `http://developer.download.nvidia.com/presentations/2010/gdc/metro.pdf`.

[10] Harold S. Stone. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *Journal of the ACM*, 20(1):27–38, January 1973.

[11] H. H. Wang. A parallel method for tridiagonal equations. *ACM Trans. Math. Software*, 7:170–183, 1981.

[12] Yao Zhang, Jonathan Cohen, and John D. Owens. Fast tridiagonal solvers on the GPU. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010)*, pages 127–136, January 2010.