Manycore Algorithms for Batch Scalar and Block Tridiagonal Solvers

ENDRE LÁSZLÓ and MIKE GILES, University of Oxford
JEREMY APPLEYARD, NVIDIA Corporation

Engineering, scientific, and financial applications often require the simultaneous solution of a large number of independent tridiagonal systems of equations with varying coefficients. Since the number of systems is large enough to offer considerable parallelism on manycore systems, the choice between different tridiagonal solution algorithms, such as Thomas, Cyclic Reduction (CR) or Parallel Cyclic Reduction (PCR) needs to be reexamined. This work investigates the optimal choice of tridiagonal algorithm for CPU, Intel MIC, and NVIDIA GPU with a focus on minimizing the amount of data transfer to and from the main memory using novel algorithms and the register-blocking mechanism, and maximizing the achieved bandwidth. It also considers block tridiagonal solutions, which are sometimes required in Computational Fluid Dynamic (CFD) applications. A novel work-sharing and register blocking–based Thomas solver is also presented.

CCS Concepts: • **Mathematics of computing → Solvers**; **Mathematical software performance**

Additional Key Words and Phrases: Scalar tridiagonal solver, block tridiagonal solver, CPU, GPU, MIC, Xeon Phi, CUDA, vectorization

ACM Reference Format:
Endre László, Mike Giles, and Jeremy Appleyard. 2016. Manycore algorithms for batch scalar and block tridiagonal solvers. ACM Trans. Math. Softw. 42, 4, Article 31 (June 2016), 36 pages.
DOI: http://dx.doi.org/10.1145/2830568

INTRODUCTION

The numerical approximation of multidimensional PDE problems on regular grids often requires the solution of multiple tridiagonal systems of equations. In computational finance, such problems arise frequently due to the alternating direction implicit (ADI) time discretization favored by many in the community (see Dang et al. [2010]). The ADI method requires the solution of multiple tridiagonal systems of equations in each dimension of the problem (see Peaceman and Rachford [1955], Craig and Sneyd [1988], Douglas and Rachford [1956], and Douglas and Gunn [1964]). The requirement also arises when using line-implicit smoothers as part of a multigrid solver [Douglas et al. 1998], and when using high-order compact differencing [Dring et al. 2014; Karaa and

The research at the Oxford e-Research Centre has been partially supported by the ASEArch project on Algorithms and Software for Emerging Architectures, funded by the UK Engineering and Physical Sciences Research Council. The research has also been supported by the TÁMOP-4.2.1./B-11/2/KMR-2011-002, TÁMOP - 4.2.2./B-10/1-2010-0014 projects at PPCU - University of National Excellence. The authors acknowledge the FP7 funded CRISP project, grant agreement number 283745, the Square Kilometre Array (SKA) Telescope Project, the Department of Physics and the Oxford e-Research Centre (OeRC) at the University of Oxford, for use of their system resources and personnel.
Authors' addresses: E. László, e-Research Centre, University of Oxford, (Current address) Faculty of Information Technology and Bionics, Pázmány Pter Catholic University; email: laszlo.endre@itk.ppke.hu; M. Giles, Mathematical Institute, University of Oxford; email: mike.giles@maths.ox.ac.uk; J. Appleyard, NVIDIA Corporation; email: jappleyard@nvidia.com.

DOI: http://dx.doi.org/10.1145/2830568

Table I. Number of Parallel Systems Increases Rapidly as Dimension d is Increased

d	N	# parallel systems
2	27386	27386
3	908	824464
4	165	4492125

Note: N is chosen to accommodate an N^d single-precision problem domain on the available 12GB of an NVIDIA Tesla K40 GPU.

Zhang 2004]. In most cases, the tridiagonal systems are scalar, with one unknown per grid point, but this is not always the case. For example, computational fluid dynamics applications often have systems with block-tridiagonal structure up to 8 unknowns per grid point [Pulliam 1986]. This article is divided into two parts. The first part considers a scalar tridiagonal system of equations; the second part considers a block tridiagonal system of equations. Both parts cover methods suitable for parallel, multi- and manycore architectures. The conclusions for the scalar and block tridiagonal solvers are given separately at the end of each part.

SCALAR TRIDIAGONAL SOLVERS ON SIMD AND SIMT ARCHITECTURES

1. INTRODUCTION

Let us start the discussion of the scalar tridiagonal solvers with a specific example of a regular 3D grid of size 256^3. An ADI time-marching algorithm will require the solution of a tridiagonal system of equations along each line in the first coordinate direction, then along each line in the second direction, and then along each line in the third direction. There are two important observations to make here. The first is that there are 256^2 separate tridiagonal solutions in each phase, which can be performed independently and in parallel; that is, there is plenty of natural parallelism to exploit on manycore processors. The second is that a data layout that is optimal for the solution in one particular direction might be far from optimal for another direction. This will lead us to consider using different algorithms and implementations for different directions. Due to this natural parallelism in batch problems, the improved parallelism of Cyclic Reduction (CR) and Parallel Cyclic Reduction (PCR) are not necessarily advantageous, and the increased workload of these methods might not necessarily pay off. If we take the example of one of the most modern accelerator cards, the NVIDIA Tesla K40 GPU, we may conclude that the 12GB device memory is suitable to accommodate an N^d multidimensional problem domain with $N^d = 12\,\text{GB}/4\,\text{arrays} = 0.75 \times 10^9$ single precision grid points. This means that the length N along each dimension is $N = \sqrt[d]{0.75 \times 10^9}$ for dimensions $d = 2{-}4$, as shown in Table I.

Before discussing the specific implementations on different architectures, we review a number of different algorithms for solving tridiagonal systems, and discuss their properties in terms of the number of floating-point operations and the amount of memory traffic generated. Research has been conducted by Zhang et al. [2010] and Chang et al. [2012] to solve a tridiagonal system of equations on GPUs. The Recursive Doubling algorithm has been developed by Stone [1973]. Earlier work by Wang [1981], van der Vorst [1987], Bondeli [1991], Mattor et al. [1995], and Spaletta and Evans [1993] has decomposed a larger tridiagonal system into smaller ones that can be solved independently, in parallel. The algorithms described in the following sections are the key building blocks of a high-performance implementation of a tridiagonal solver. We introduce the tridiagonal system of equations, as shown in Equation (1), or in its matrix form, shown in Equation (2).

$$a_i u_{i-1} + b_i u_i + c_i u_{i+1} = d_i, \quad i = 0, 1, \ldots, N-1 \tag{1}$$

$$\begin{pmatrix} b_0 & c_0 & 0 & 0 & \cdots & 0 \\ a_1 & b_1 & c_1 & 0 & \cdots & 0 \\ 0 & a_2 & b_2 & c_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{N-1} & b_{N-1} \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{N-1} \end{pmatrix} = \begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{N-1} \end{pmatrix}, \tag{2}$$

where $a_0 = c_{N-1} = 0$.

2. TRIDIAGONAL ALGORITHMS

In this section, the standard algorithms—such as Thomas, PCR, and CR—are presented along with a new hybrid Thomas-PCR algorithm, which is the combination of the Thomas and the PCR algorithms.

2.1. Thomas Algorithm

The Thomas algorithm [Thomas 1949] is a sequential algorithm that is described in Press et al. [2007]. It is simply a customization of Gaussian elimination to the case in which the matrix is tridiagonal. The algorithm has a forward pass in which the lower diagonal elements a_i are eliminated by adding a multiple of the row above. This is then followed by a reverse pass to compute the final solution using the modified c_i values. In both passes, the d_i is also modified according to the row operations.

The full Thomas algorithm with in-place solution—right-hand side (RHS) vector is overwritten—is given in Algorithm 1. Note that this does not perform any pivoting; it is assumed that the matrix is diagonally dominant, or at least sufficiently close to diagonal dominance so that the solution is well conditioned. Floating-point multiplication and addition, as well as Fused Multiply and Add (FMA) have the same instruction throughput on almost every hardware. Also, the use of FMA is considered to be a compiler optimization feature; therefore, optimistic calculations based on this instruction give a good lower estimate on the number of actual instruction uses. In the following discussion, the FMA instruction is used as the basis of estimating the work complexity of the algorithm. The computational cost per row is approximately three FMA operations: one reciprocal and two multiplications. If we treat the cost of the reciprocal as being equivalent to five FMAs (which is the approximate cost on a GPU

ALGORITHM 1: Thomas Algorithm

```
Require: $thomas(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})$
 1: $d_0^* := d_0 / b_0$
 2: $c_0^* := c_0 / b_0$
 3: for $i = 1, \ldots, N-1$ do
 4:     $r := 1 / (b_i - a_i\, c_{i-1})$
 5:     $d_i^* := r\,(d_i - a_i\, d_{i-1})$
 6:     $c_i^* := r\, c_i$
 7: end for
 8: for $i = N-2, \ldots, 0$ do
 9:     $d_i := d_i^* - c_i^*\, d_{i+1}$
10: end for
11: return $\mathbf{d}$
```

ALGORITHM 2: PCR Algorithm

```
Require: $pcr(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})$
 1: for $p = 1, \ldots, P$ do
 2:   $s := 2^{p-1}$
 3:   for $i = 0, \ldots, N-1$ do
 4:     $r := 1\,/\,(1 - a_i^{(p-1)}\, c_{i-s}^{(p-1)} - c_i^{(p-1)} a_{i+s}^{(p-1)})$
 5:     $a_i^{(p)} := -r\, a_i^{(p-1)}\, a_{i-s}^{(p-1)}$
 6:     $c_i^{(p)} := -r\, c_i^{(p-1)}\, c_{i+s}^{(p-1)}$
 7:     $d_i^{(p)} := r\,(d_i^{(p-1)} - a_i^{(p-1)}\, d_{i-s}^{(p-1)} - c_i^{(p-1)}\, d_{i+s}^{(p-1)})$
 8:   end for
 9: end for
10: return $\mathbf{d}^{(P)}$
```

for a double precision reciprocal), then the total cost is equivalent to 10 FMAs per grid point.

On the other hand, the algorithm requires at minimum the loading of a_i, b_i, c_i, d_i, and the storing of the final answer d_i. Worse still, it may be necessary to store the c_i^*, d_i^* computed in the forward pass, then read them back during the reverse pass. This shows that the implementation of the algorithm is likely to be memory-bandwidth limited, because there are very few operations per memory reference. Thus, when using the Thomas algorithm it will be critical to ensure the maximum possible memory bandwidth.

2.2. PCR and CR Algorithms

PCR [Gander and Golub 1997] is an inherently parallel algorithm that is ideal when using multiple execution threads to solve each tridiagonal system.

If we start with the same tridiagonal system of equations, but normalized so that $b_i = 1$,

$$a_i u_{i-1} + u_i + c_i u_{i+1} = d_i, \quad i = 0, 1, \ldots, N-1,$$

with $a_0 = c_{N-1} = 0$; then, subtracting the appropriate multiples of rows $i \pm 1$ and renormalizing, gives

$$a_i' u_{i-2} + u_i + c_i' u_{i+2} = d_i', \quad i = 0, 1, \ldots, N-1,$$

with $a_0' = a_1' = c_{N-2}' = c_{N-1}' = 0$. Repeating this by subtracting the appropriate multiples of rows $i \pm 2$ gives

$$a_i'' u_{i-4} + u_i + c_i'' u_{i+4} = d_i'', \quad i = 1, 2, \ldots, N,$$

with $a_j'' = c_{N-1-j}'' = 0$ for $0 \le j < 4$.

After P such steps, where P is the smallest integer such that $2^P \ge N$, then all of the modified a and c coefficients are zero (since, otherwise, the value of u_i would be coupled to the nonexistent values of $u_{i \pm 2^P}$); thus, we have the value of u_i.

The PCR algorithm is given in Algorithm 2. Note that any reference to a value with index j outside the range $0 \le j < N$ can be taken to be zero. It is multiplied by a zero coefficient; thus, as long as it is not an IEEE exception value (such as an NaN), then any valid floating-point value can be used. If the computations within each step are performed simultaneously for all i, then it is possible to reuse the storage so that $a^{(p+1)}$ and $c^{(p+1)}$ are held in the same storage (e.g., the same registers) as $a^{(p)}$ and $c^{(p)}$. The computational cost per row is approximately equivalent to 14 FMAs in each step; thus, the total cost per grid point is $14 \log_2 N$. This is clearly much greater than the cost of

ALGORITHM 3: First Phase of Hybrid Algorithm

```
Require: $hybridFW(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})$
 1: $d_0 := d_0/b_0$
 2: $a_0 := a_0/b_0$
 3: $c_0 := c_0/b_0$
 4: for $i = 1, \dots, M-2$ do
 5:    $r := 1\,/\,(b_i - a_i\,c_{i-1})$
 6:    $d_i := r\,(d_i - a_i\,d_{i-1})$
 7:    $a_i := -r\,a_i\,a_{i-1}$
 8:    $c_i := r\,c_i$
 9: end for
10: for $i = M-3, \dots, 1$ do
11:    $d_i := d_i - c_i\,u_{i+1}$
12:    $a_i := a_i - c_i\,a_{i+1}$
13:    $c_i := -c_i\,c_{i+1}$
14: end for
```

the Thomas algorithm with 10 FMA, but the data transfer to/from the main memory may be lower if there is no need to store and read back the various intermediate values of the a and c coefficients.

CR is a slightly different algorithm, in which the p^{th} pass of the algorithm presented earlier is performed only for those i for which $i \bmod 2^p = 0$. This gives the forward pass of the PCR algorithm; there is then a corresponding reverse pass that performs the back solve. This involves fewer floating-point operations overall. However, there is less parallelism, on average, and it requires twice as many steps, thus it is slower than PCR when there are many active threads. We thus have not found it to be helpful in the work presented here.

2.3. Hybrid Thomas-PCR Algorithm

The hybrid algorithm is a combination of modified Thomas and PCR algorithms. Hybrid algorithms have been developed over the years for different applications. Wang [1981] decomposed tridiagonal systems in an upper tridiagonal form which is useful when multiple right hand sides are available. The slides of Sakharnykh [2010] show that the PCR algorithm can be used to divide a larger system into smaller ones, which can be solved using the Thomas algorithm. In this article, it is shown that a larger system can be divided into smaller ones using a modified Thomas algorithm and the warp-level PCR can be used to solve the remaining system. The complete computation with the presented algorithms can be done in registers using the *__shfl()* instructions introduced in the NVIDIA Kepler GPU architecture. In the case of the Thomas solver intermediate values, c_i^* and d_i^* had to be stored and loaded in main memory. With the hybrid algorithm, the input data is read once and output is written once without the need to move intermediate values in global memory. Therefore, this algorithm allows an optimal implementation.

Suppose that the tridiagonal system is broken into a number of sections of size M (shown in Figure 1), each of which will be handled by a separate thread. Within each of these pieces, using local indices ranging from 0 to $M-1$, a slight modification to the Thomas algorithm (shown in Algorithm 3) operating on Rows 1 to $M-2$ enables one to obtain an equation of the form

$$a_i\,u_0 + u_i + c_i\,u_{M-1} = d_i, \quad i = 1, 2, \dots, M-2 \tag{3}$$

expressing the central values as a linear function of the two end values, u_0 and u_{M-1}. The forward and backward phases of the modified Thomas algorithm are shown in

$$\left(\begin{array}{cccc|cccc|cccc}
b_0 & c_0 & & & & & & & & & & \\
a_1 & b_1 & c_1 & & & & & & & & & \\
 & a_2 & b_2 & c_2 & & & & & & & & \\
 & & a_3 & b_3 & c_3 & & & & & & & \\
\hline
 & & & a_4 & b_4 & c_4 & & & & & & \\
 & & & & a_5 & b_5 & c_5 & & & & & \\
 & & & & & a_6 & b_6 & c_6 & & & & \\
 & & & & & & a_7 & b_7 & c_7 & & & \\
\hline
 & & & & & & & a_8 & b_8 & c_8 & & \\
 & & & & & & & & a_9 & b_9 & c_9 & \\
 & & & & & & & & & a_{10} & b_{10} & c_{10} \\
 & & & & & & & & & & a_{11} & b_{11}
\end{array}\right)
\left(\begin{array}{c}
u_0\\ u_1\\ u_2\\ u_3\\ \hline u_4\\ u_5\\ u_6\\ u_7\\ \hline u_8\\ u_9\\ u_{10}\\ u_{11}
\end{array}\right)
=
\left(\begin{array}{c}
d_0\\ d_1\\ d_2\\ d_3\\ \hline d_4\\ d_5\\ d_6\\ d_7\\ \hline d_8\\ d_9\\ d_{10}\\ d_{11}
\end{array}\right)$$

Fig. 1. Starting state of the equation system. 0 values outside the bar are part of the equation system, and are needed for the algorithm and implementation.

$$\left(\begin{array}{cccc|cccc|cccc}
1 & c_0 & & & & & & & & & & \\
a_1^* & 1 & c_1 & & & & & & & & & \\
a_2^* & & 1 & c_2 & & & & & & & & \\
a_3^* & & & 1 & c_3 & & & & & & & \\
\hline
 & & & a_4^* & 1 & c_4 & & & & & & \\
 & & & & a_5^* & 1 & c_5 & & & & & \\
 & & & & a_6^* & & 1 & c_6 & & & & \\
 & & & & a_7^* & & & 1 & c_7 & & & \\
\hline
 & & & & & & & a_8^* & 1 & c_8 & & \\
 & & & & & & & & a_9^* & 1 & c_9 & \\
 & & & & & & & & a_{10}^* & & 1 & c_{10} \\
 & & & & & & & & a_{11}^* & & & 1
\end{array}\right)
\left(\begin{array}{c}
u_0\\ u_1\\ u_2\\ u_3\\ \hline u_4\\ u_5\\ u_6\\ u_7\\ \hline u_8\\ u_9\\ u_{10}\\ u_{11}
\end{array}\right)
=
\left(\begin{array}{c}
d_0^*\\ d_1^*\\ d_2^*\\ d_3^*\\ \hline d_4^*\\ d_5^*\\ d_6^*\\ d_7^*\\ \hline d_8^*\\ d_9^*\\ d_{10}^*\\ d_{11}^*
\end{array}\right)$$

Fig. 2. State of the equation system after the forward sweep of the modified Thomas algorithm.

$$\left(\begin{array}{cccc|cccc|cccc}
\mathbf{1} & & & \mathbf{c_0^*} & & & & & & & & \\
a_1^* & 1 & & c_1^* & & & & & & & & \\
a_2^* & & 1 & c_2^* & & & & & & & & \\
\mathbf{a_3^*} & & & \mathbf{1} & \mathbf{c_3^*} & & & & & & & \\
\hline
 & & & \mathbf{a_4^*} & \mathbf{1} & & & \mathbf{c_4^*} & & & & \\
 & & & & a_5^* & 1 & & c_5^* & & & & \\
 & & & & a_6^* & & 1 & c_6^* & & & & \\
 & & & & \mathbf{a_7^*} & & & \mathbf{1} & \mathbf{c_7^*} & & & \\
\hline
 & & & & & & & \mathbf{a_8^*} & \mathbf{1} & & & \mathbf{c_8^*} \\
 & & & & & & & & a_9^* & 1 & & c_9^* \\
 & & & & & & & & a_{10}^* & & 1 & c_{10}^* \\
 & & & & & & & & \mathbf{a_{11}^*} & & & \mathbf{1}
\end{array}\right)
\left(\begin{array}{c}
\mathbf{u_0}\\ u_1\\ u_2\\ \mathbf{u_3}\\ \hline \mathbf{u_4}\\ u_5\\ u_6\\ \mathbf{u_7}\\ \hline \mathbf{u_8}\\ u_9\\ u_{10}\\ \mathbf{u_{11}}
\end{array}\right)
=
\left(\begin{array}{c}
\mathbf{d_0^*}\\ d_1^*\\ d_2^*\\ \mathbf{d_3^*}\\ \hline \mathbf{d_4^*}\\ d_5^*\\ d_6^*\\ \mathbf{d_7^*}\\ \hline \mathbf{d_8^*}\\ d_9^*\\ d_{10}^*\\ \mathbf{d_{11}^*}
\end{array}\right)$$

Fig. 3. State of the equation system after the backward sweep of the modified Thomas algorithm. Bold variables show the elements of the reduced system that is to be solved with the PCR algorithm.

Figures 2 and 3, respectively. Using Equation (3) for $i = 1, M-2$ to eliminate the u_1 and u_{M-2} entries in the equations for Rows 0 and M–1 leads to a reduced tridiagonal system (shown in Figure 3) of equations involving the first and last variables within each section. This reduced tridiagonal system can be solved using PCR using Algorithm 4; then, Algorithm 5 gives the interior values.

ALGORITHM 4: Middle Phase: Modified PCR Algorithm

```
Require: $pcr2(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})$
 1: for $i = 1 : 2 : N-1$ do
 2:     $r = 1/(1 - a_i^{(0)} c_{i-1}^{(0)} - c_i^{(0)} a_{i+1}^{(0)})$
 3:     $d_i^{(1)} = r\,(d_i^{(0)} - a_i^{(0)} d_{i-1}^{(0)} - c_i^{(0)} d_{i+1}^{(0)})$
 4:     $a_i^{(1)} = -r\, a_i^{(0)} a_{i-1}^{(0)}$
 5:     $c_i^{(1)} = -r\, c_i^{(0)} c_{i+1}^{(0)}$
 6: end for
 7: for $p = 1, \ldots, P$ do
 8:     $s := 2^p$
 9:     for $i = 1 : 2 : N-1$ do
10:         $r = 1/(1 - a_i^{(p-1)} c_{i-s}^{(p-1)} - c_i^{(p-1)} a_{i+s}^{(p-1)})$
11:         $d_i^{(p)} = r\,(d_i^{(p-1)} - a_i^{(p-1)} d_{i-s}^{(p-1)} - c_i^{(p-1)} d_{i+s}^{(p-1)})$
12:         $a_i^{(p)} = -r\, a_i^{(p-1)} a_{i-s}^{(p-1)}$
13:         $c_i^{(p)} = -r\, c_i^{(p-1)} c_{i+s}^{(p-1)}$
14:     end for
15: end for
16: for $i = 1 : 1 : N-1$ do
17:     $d_i^{(P)} = d_i^{(P-1)} - a_i^{(P-1)} d_i^{(P-1)} - c_i^{(P-1)} d_{i+1}^{(P-1)}$
18: end for
19: return $\mathbf{d}^{(P)}$
```

ALGORITHM 5: Last Phase: Solve for the Remaining Unknowns

```
Require: $hybrid1b(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})$
 1: for $i = 1, \ldots, M-1$ do
 2:     $d_i := d_i - a_i\, r\, d_0 - c_i\, d_{M-1}$
 3: end for
```

In the last phase, the solution of the interior unknowns using the outer $i = 0$ and $i = M - 1$ values needs to be completed with Algorithm 5.

When there are 32 CUDA threads and 8 unknowns per thread, the cost is approximately 14 FMAs per point, about 40% more than the cost of the Thomas algorithm. The data-transfer cost depends on whether these intermediate coefficients a_i, c_i, d_i need to be stored in the main memory. If they do, it is again more costly than the Thomas algorithm; if not, it is less costly. We need to add this to the cost of the PCR solution. The relative size of this depends on the ratio $(\log_2 N)/M$. We will discuss this in more detail later.

The hybrid Thomas-PCR solver is validated against the Thomas solver by computing the Mean Square Error (MSE) between the two solutions. The results show MSE error in the order of 10^{-9} in single precision and 10^{-18} in double precision.

3. DATA LAYOUT IN MEMORY

Knowledge of the data layout and the access patterns of the executed code with various algorithms is essential to understanding the achieved performance. Taking the example of a 3D application shown in Figure 4, each of the coefficient arrays is a 3D indexed array that is mathematically of the form $u_{i,j,k}$. That is, each array **a**, **b**, **c**, **d**, and **u** is stored in a separate cubic data array.

However, each is stored in memory as a linear array; thus, we require a mapping from the index set (i, j, k) to a location in memory. If the array has dimensions (N_x, N_y, N_z),

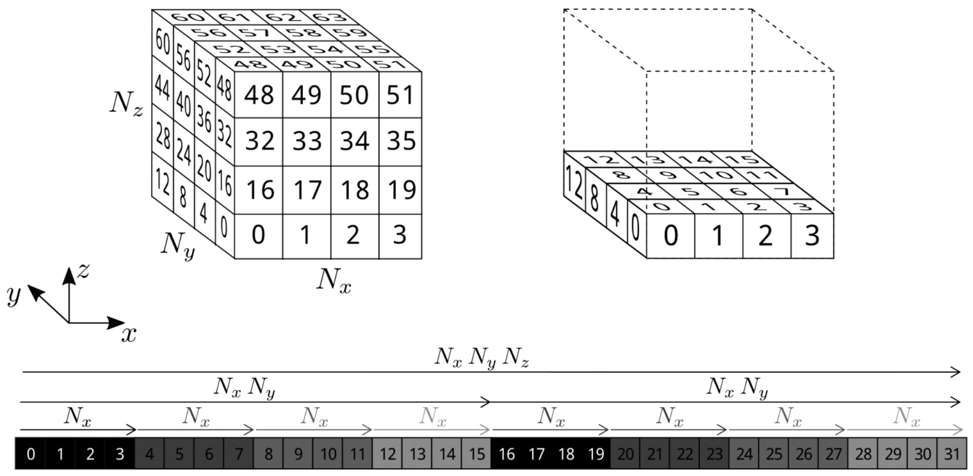


Fig. 4. Data layout of 3D cube data-structure.

we choose to use this mapping:

$$\text{index} = i + j\,N_x + k\,N_x\,N_y.$$

This means that if we perform a Thomas algorithm solution in the first coordinate direction, then consecutive elements of the same tridiagonal system are contiguous in memory. On the other hand, in the second coordinate direction, they are accessed with a stride of N_x, and in the third direction, the stride is $N_x\,N_y$. This data layout extends naturally to applications with a higher number of dimensions.

To understand the consequences of this layout and access pattern, it is also necessary to understand the operation of the cache-memory hierarchy. The memory bus that transfers data from the main memory (or graphics memory) to the CPU (or GPU) does so in cache lines that have a size usually in the range of 32B to 128B. These cache lines are held in the cache hierarchy until they are displaced to make room for a new cache line. One difference between CPUs and GPUs is that CPUs have a lot more cache memory per thread; thus, cache lines are usually resident longer.

The effectiveness of caches depends on two kinds of locality: (1) *temporal locality*, which means that a data item that has been loaded into the cache is likely to be used again in the near future; and (2) *spatial locality*, which means that when a cache line is loaded to provide one particular piece of data, then it is likely that other data items in the same cache line will be used in the near future. An extreme example of spatial locality can occur in a GPU when a set of 32 threads (known as a *warp*) each load one data item. If these 32 items form a contiguous block of data consisting of one or more cache lines, then there is perfect spatial locality, with the entire line of data being used. This is referred to as a *coalesced* read or write. See NVIDIA [2015a] for further details.

Aligned memory access for efficient data transfer [Intel 2012a, 2012b; NVIDIA 2015a], Translation Lookaside Buffer (TLB) [Intel 2012a; Wong et al. 2010], and hit rate reduction for lower cache latency are among the techniques used to reduce execution time. Avoiding cache and TLB cache thrashing [Intel 2012a] can be done by proper padding, which is the responsibility of the user of the library.

4. DETAILS OF CPU, MIC, AND GPU HARDWARE

Processors used in this study include a high-end, dual-socket Intel Xeon server CPU, Intel Xeon Phi coprocessor, and NVIDIA K40m GPU. Processor specifications are listed in Appendix A. CPUs operate with fast, heavyweight cores, with large cache, out-of-order execution, and branch prediction. GPUs, on the other hand, use lightweight, in-order hardware threads with moderate, but programmable, caching capabilities and without branch prediction.

The CPU cores are very complex out-of-order, shallow-pipelined, low-latency execution cores, in which operations can be performed in a different order to that specified by the executable code. This on-the-fly reordering is performed by the hardware to avoid delays due to waiting for data from the main memory, but is done in a way that guarantees thatthe correct results are computed. Each CPU core also has an AVX vector unit. This 256b unit can process 8 single-precision or 4 double-precision operations at the same time, using vector registers for the inputs and output. For example, it can add or multiply the corresponding elements of two vectors of 8 single-precision variables. To achieve the highest performance from the CPU, it is important to exploit the capabilities of this vector unit, but there is a cost associated with gathering data into the vector registers to be worked on. Each cores owns an L1 (32KB) and an L2 (256KB) level cache and they also own a portion of the distributed L3 cache. These distributed L3 caches are connected with a special ring bus to form a large, usually 15MB to 35MB L3 or Last Level Cache (LLC). Cores can fetch data from the cache of another core.

The Xeon Phi or Many Integrated Core (MIC) is Intel's HPC coprocessor aimed at accelerating compute-intensive problems. The architecture is composed of 60 individual simple, in-order, deep-pipelined, high-latency, high-throughput CPU cores equipped with 512b-wide floating-point vector units, which can process 16 single-precision or 8 double-precision operations at the same time, using vector registers for inputs and output. This architecture faces mostly the same programmability issues as the Xeon CPU, although, due to the in-order execution, the consequences of inefficiencies in the code can result in more server performance deficit. The MIC coprocessor uses a similar caching architecture as the Xeon CPU, but has only two levels: L1 with 32KB and the distributed L2 with a 512KB/core.

The Kepler generation of NVIDIA GPUs has a number of Streaming Multiprocessor – eXtended (SMX) functional units. Each SMX has 192 relatively simple in-order execution cores. These operate effectively in warps of 32 threads; thus, they can be thought of as vector processors with a vector length of 32. To avoid poor performance due to the delays associated with accessing the graphics memory, the GPU design is heavily multithreaded, with up to 2048 threads (or 64 warps) running on each SMX simultaneously. While one thread in a warp waits for data, execution can switch to a different thread in the warp that has the data it requires for its computation. Each thread has its own share (up to 255 32b registers) of the SMX's registers (65536 32b registers) to avoid the context-switching overhead usually associated with multithreading on a single core. Each SMX has a local L1 cache as well as a shared memory that enables different threads to exchange data. The combined size of these is 64KB, with the relative split controlled by the programmer. There is also a relatively small 1.5MB global L2 cache that is shared by all SMX units.

5. CPU AND MIC SOLUTION

In this section, the CPU and MIC solutions are described. The MIC implementation is essentially the same as the CPU implementation with some minor differences in terms of the available Instruction Set Architecture (ISA). The ZMM vector registers are 512b wide; thus, they allow handling 8x8 double-precision or 16x16 single-precision data transposition.

To have an efficient implementation of the Thomas algorithm running on a CPU, compiler auto-vectorization and single-thread, sequential execution is not sufficient.

To exploit the power of a multicore CPU, the vector instruction set together with the multithreaded execution needs to be fully utilized. Unfortunately, compilers today are not always capable of making full use of the vector units of a CPU even though the instruction set would make it feasible. Often, algorithmic data dependencies and execution flow can prevent the vectorization of a loop. Usually, these conditions are related to the data alignment, layout, and data dependencies in the control flow graph. Such conditions are live-out variables, interloop dependency, nonaligned array, noncontiguous data-access pattern, array aliasing, and so on. If the alignment of arrays cannot be proven at compile time, then special vector instructions with unaligned access will be used in the executable code, which leads to a significant performance deficit. In general, the largest difference between the vector-level parallelism of the CPU and GPU are the differences in how the actual parallelism is executed. In the case of the CPU and MIC code, the compiler decides how a code segment can be vectorized with the available instruction set; this is called Single Instruction Multiple Data (SIMD) parallelism. The capabilities of the ISA influences the efficiency of the compiler to accomplish this task. GPUs, on the other hand, leave the vector-level parallelization for the hardware; this is called Single Instruction Multiple Threads (SIMT). It is decided in runtime by the scheduling unit whether an instruction can be executed in parallel or not. If not, then the threads working in a group (called a warp in CUDA terms) are divided into a set of idle and active threads. When the active threads complete the task, another set of idle threads is selected for execution. This sequential scheduling goes on while there are unaccomplished threads left. A more detailed study on this topic in the case of unstructured grids can be found in Reguly et al. [2016].

Since the Thomas algorithm needs huge data traffic due to the varying coefficients, one would expect the implementation of the algorithm to be limited by memory bandwidth. If an implementation is bandwidth limited and can be implemented with a nonbranching computation stream, the GPU is supposed to be more efficient than the CPU due to the 2 to 4 times larger memory bandwidth. However, this is not true for CPUs with out-of-order execution, large cache, and sophisticated caching capabilities. A single-socket CPU in Appendix A has 20MB of LLC/L3 per socket, 8 x 256KB L2 and 8 x 32KB L1 Cache per socket, which is in total 40MB L3, 4MB L2, and 512KB L1 cache total. This caching/execution mechanism makes the CPU efficient when solving a tridiagonal algorithm with the Thomas algorithm. The two temporary arrays ($\mathbf{c}^*$ and $\mathbf{d}^*$) can be held in this cache hierarchy and reused in the backward sweep. Therefore, input data ($\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{d}$) passes through the DDR3 memory interface only once when it is loaded and result array ($\mathbf{u}$) passes once when it is stored.

This means that a system in single precision with 3 coefficient arrays (an RHS array and 2 temporary arrays) can stay in L1 cache up to system length 1365 if no hardware hyper-threading (HTT) is used or half the size, namely, 682, if HTT is used. The efficiency of the solver still remains very good above this level, and a gradual decrease can be observed as L2 and L3 get a more significant role in caching.

Thomas Solver in X Dimension. The data layout described in Section 3 does not allow for natural parallelism available with AVX vector instructions. The loading of an 8-wide Single Precision (SP) or 4-wide Double Precision (DP) chunk of array into a register can be accomplished with the use of *_mm256_load_p{s,d}* intrinsic or the *vmovap{s,d}* assembly instruction if the first element of the register is on an aligned address; otherwise, the *_mm256_loadu_p{s,d}* intrinsic or *vmovup{s,d}* assembly instruction needs to be used.

Unfortunately, since the data in the vector register belongs to one tridiagonal system, it needs to be processed sequentially. When the algorithm acquires a value of a coefficient array, the consecutive 7 SP (or 3 DP) values will also be loaded into the L1

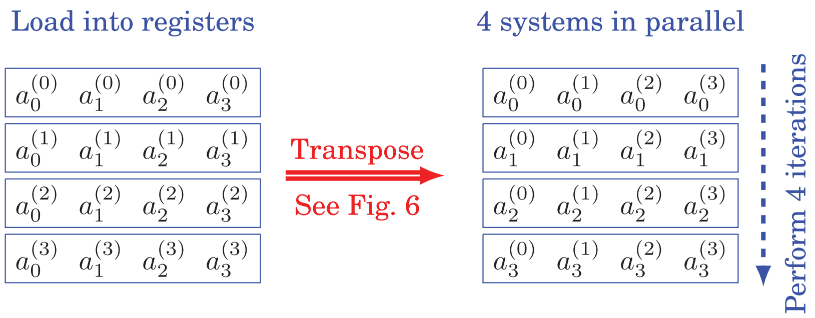


Fig. 5. Vectorized Thomas solver on CPU and MIC architectures. Data is loaded into M-wide vector registers in M steps. On AVX, $M = 4$ for double precision and $M = 8$ for single precision. On IMCI, $M = 8$ for double precision and $M = 16$ for single precision. Data is then transposed within registers, as described in Figure 6; after that, M number of iterations of the Thomas algorithm is performed. When this is done, the procedure repeats until the end of the systems is reached.

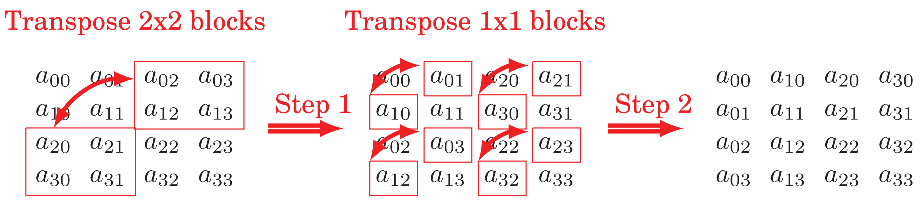


Fig. 6. Local data transposition of an $M \times M$ two-dimensional array of M consecutive values of M systems, where a_{ti} is the ith coefficient of tridiagonal system t. Transposition can be performed with $M \times \log_2 M$ AVX or IMCI shuffle-type instructions such as swizzle, permute, and align, or their masked counterpart. On AVX, $M = 4$ for double precision and $M = 8$ for single precision. On IMCI, $M = 8$ for double precision and $M = 16$ for single precision.

cache. The L1 cache line size is the same as the vector register width; thus, no saving on the efficiency of the main memory traffic can be achieved. According to Appendix A, this memory can be accessed with 4 clock cycle latency. On a Xeon server processor, this latency is hidden by the out-of-order execution unit if data dependencies allow it. When the instructions following the memory load instruction are independent from the loaded value—for example, the instructions to calculate the index of the next value to be read—the processor can skip waiting for the data to arrive and continue on with the independent computations. When the data arrives or the execution flow reaches an instruction that depends on the loaded value, the processor enters an idle state until the data arrives. L1 caching and out-of-order execution on the CPU therefore enables the Thomas algorithm in the X dimension to run with high efficiency, although some performance is still lost due to nonvectorized code.

Vectorization and efficient data reuse can be achieved by applying local data transposition. This local transposition combines the register or cache-blocking optimization with butterfly transposition. If the code is written using intrinsic operations, the compiler decides where the temporary data is stored, either in registers or cache. The advantage of this approach is that it allows for the use of vector load/store and arithmetic instructions and vector registers (or L1 cache), which leads to highly efficient execution.

The process in the case in double-precision coefficient arrays is depicted in Figure 6. As the AVX YMM registers are capable of holding 4 DP values, 4 tridiagonal systems can be solved in parallel. The first 4 values of the 4 systems are loaded and stored in

4 separate vector registers for each coefficient array **a**, **b**, and so on. This procedure allows for perfect cache-line utilization and is depicted in Figure 6 for the first 4x4 array case. The same procedure is applied for the 8x8 single-precision case with one additional transposition of 4x4 blocks. The process in the case of 4x4 of array **a** is the following. Load the first 4 values of the first tridiagonal system into the YMM0 register, load the first 4 values of the second tridiagonal system into the YMM1 register, and so on. Once data is in the register, perform transposition according to Figure 6 and perform 4 steps of the Thomas algorithm. Do the same for the next 4 values of the system. Repeat this procedure until the end of the systems are reached.

Thomas Solver in Y and Z Dimension. The data layout described in Section 3 suggests natural parallel execution with *_mm256_load*{*u*}*_p*{*s,d*} loads. Unfortunately, the compiler is not able to determine how to vectorize along the these dimensions, since it cannot prove that the neighboring solves will access neighboring elements in the nested *for* loops of the forward and backward phases. Not even Intel's elemental function and array notation is capable of handling this case correctly.

Since solving tridiagonal problems is inherently data parallel and data reads fit the AVX load instructions, manual vectorization with AVX intrinsic functions is used to vectorize the method. Compared to a scalar, multithreaded implementation, using the explicit vectorization gives a 2.5 times speedup in dimension Y in single precision.

The vectorized code reads 8/4 consecutive SP/DP scalar elements into YMM registers (*__m256* or *__m256d*) from the multidimensional coefficient array. The implementation is straightforward with *_mm256_load_p*{*s,d*} intrinsic and the *vmovap*{*s,d*} assembly instruction if the first element of the register is on an aligned address or with *_mm256_loadu_p*{*s,d*} intrinsic and the *vmovup*{*s,d*} assembly instruction when data is not aligned. In the case in which the length along the X dimension is not a multiple of 8/4, padding is used to achieve maximum performance. Arithmetic operations are carried out using *_mm256_*{*add,sub,mul,div*}*_p*{*s,d*} intrinsics.

When stepping in the Z dimension to solve a system, one may hit the NUMA penalty. The consequences of cache-coherent NUMA (ccNUMA) are two-fold. First, the LLC miss rate increases since the values of **d** are still in the cache where they were used the last time. The coherent cache is implemented through the QPI bus, which introduces an extra access latency when the cache line needs to be fetched from a different socket. LLC cache miss rate can go up to a level of 84.7% of all LLC-cache access in the Z dimension. The second consequence is the TLB miss penalty, which occurs when elements in an array are accessed with large stride. A TLB page covers only a 4KB range of the dynamically allocated memory. Once the data acquired is outside this range, a new page-frame pointer needs to be loaded into the TLB cache. The latency of doing this is the time taken to access the main memory and time taken by the Linux kernel to give the pointer and permission to access that particular page.

6. GPU SOLUTION

The GPU implementation of the tridiagonal solver is not as straightforward as the CPU solver with the Thomas algorithm. Regardless of the underlying algorithms (Thomas or Thomas-PCR hybrid), solving tridiagonal systems is usually considered to be memory bandwidth limited, since the ratio of the minimal amount of data that needs to be loaded ($4N$) and stored (N) and the minimal amount of FLOPs need to be carried out (with the Thomas algorithm $9N$) is $\frac{9N FLOP}{(4+1)N\, values}$. This figure is called the compute ratio and implies that, for every loaded value, this amount of compute needs to be performed. This figure also depends on the SP/DP floating-point data and processing unit throughput. Thus, the required compute ratio for single and double precision is $0.45\frac{FLOP}{byte}$ and $0.225\frac{FLOP}{byte}$, respectively. An algorithm is theoretically expected to be memory-bandwidth limited if the compute ratio capability of the specific hardware exceeds the compute ratio of the algorithm. For the K40 GPU, these figures are $16.92\frac{FLOP}{byte}$ and $14.89\frac{FLOP}{byte}$, respectively, which suggests that solving the problem with the most compute-efficient algorithm—the Thomas algorithm—is memory bound. In the forthcoming discussion, the aim is to improve this bottleneck and achieve high memory utilization with suitable access patterns and reduced memory traffic.

Global memory load on the GPU poses strict constraints on the access patterns used to solve systems of equations on a multidimensional domain. The difference with regard to CPUs comes from the way that SIMT thread-level parallelism provides access to data in the memory. Solvers using unit-stride access pattern (along the X dimension) and the long-strided access pattern (along Y and higher dimensions) need to be distinguished. The former is explained in Section 6.1; the latter is explained in Section 6.2.

The memory-load instructions are issued by threads within the same warp. In order to utilize the whole 32B (or 128B) cache, line threads need to use all the data loaded by that line. At this point, we need to distinguish between the two efficient algorithms discussed in this article for solving tridiagonal problems along the X dimension, because the two need different optimization strategies. These algorithms are

(1) the Thomas algorithm with optimized memory access pattern, detailed in Section 6.1
(2) the Thomas-PCR hybrid algorithm, detailed in Section 6.3

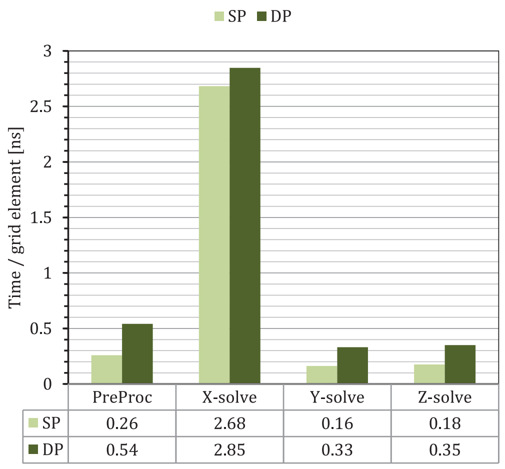
6.1. Thomas Algorithm with Optimized Memory Access Pattern

The optimization is performed on the Thomas algorithm, which is detailed in Section 2.1. The problem with the presented solver is two-fold: (1) the access pattern along dimension X is different from the pattern along dimensions Y and higher; (2) in X dimension, the actual data transfer is more than the theoretically required lower limit. In this section, it is shown how to overcome problem (1) and how to optimize the effect of (2) on a GPU. In order to make the discussion unambiguous, the focus of the following discussion is put on the single-precision algorithm. The same argument applies for the access pattern of double precision.

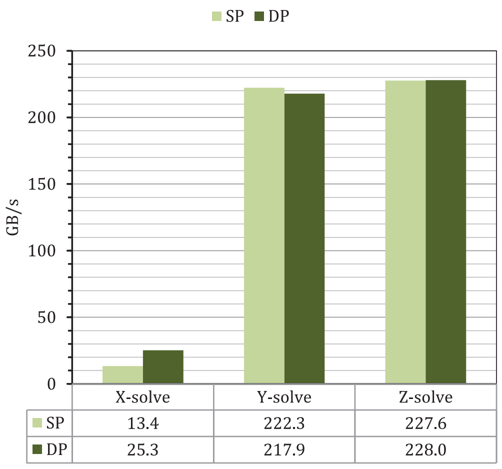
The first problem becomes obvious when one considers execution time along the three dimensions of an ADI solution step (see Figure 7). The X-dimensional execution time is more than one order of magnitude lower than the solution along the Y and Z dimension. This is due to two factors: (1) high cache underutilization and (2) high TLB thrashing.

The high cache line underutilization can be explained with the access pattern of coefficient a_i in Algorithm 1. A whole 32B cache line with 8 SP values is loaded into the L2/L1 or noncoherent cache, but temporarily only one value is used by the algorithm before the cache line is subsequently evicted; this results in poor cache-line utilization. The other values within the cache line are used in a different time instance of the solution process, but at that point the cache line will already be flushed from the cache. The same applies to arrays $\mathbf{b}$, $\mathbf{c}$, $\mathbf{d}$, $\mathbf{u}$ as well. To overcome this issue, one needs to do local data transposition with cache or register blocking similar to that discussed in Section 5. Two solutions to perform this optimization are introduced in Sections 6.1.1 and 6.1.2. The idea of these optimizations is to increase cache-line utilization by reading consecutive memory locations with consecutive threads into a scratch memory (shared memory or registers), then redistributing them as needed.

The issue with TLB thrashing in the X-dimensional solution relates to the mapping of threads to the problem. One might map a 2-dimensional thread block on the Y-Z plane



	PreProc	X-solve	Y-solve	Z-solve
🟩 SP	0.26	2.68	0.16	0.18
🟩 DP	0.54	2.85	0.33	0.35



	X-solve	Y-solve	Z-solve
🟩 SP	13.4	222.3	227.6
🟩 DP	25.3	217.9	228.0

Fig. 7. ADI kernel execution times on K40m GPU, and corresponding bandwidth figures for X-, Y-, and Z-dimensional solves based on the Thomas algorithm. Single-precision X-dimensional solve is slower $\times 16.75$ than the Y-dimensional solve. High bandwidth in the case of Y- and Z-dimensional solve is reached due to local memory caching and read-only (texture) caching.

of the 3-dimensional ADI problem, and process the systems along the X dimension. In this case, the start index (*ind* in $a[ind]$) calculation is easy, according to the following formula:

ALGORITHM 6: The Common Way of Mapping CUDA Threads on a 3D Problem

```
1: y = threadIdx.x
2: z = threadIdx.y
3: ind = z*NX*NY + y*NX
```

The problem with this mapping is that it introduces significant TLB thrashing. TLB is an important part of the cache hierarchy in every processor with virtual memory. A virtual-memory page covers a certain section of the memory, usually 4KB in the case of systems with relatively low memory and 2MB "huge memory page" on systems with large memory. Since NVIDIA does not reveal the details of its cache and TLB memory subsystem, further details on the supposed TLB structure for the older GT200 architecture can be found in Wong et al. [2010]. Note that significant architecture changes have been made since GT200, but the latency is expected to be dominated by DRAM access latency, which did not change significantly in the last few years. The problem with TLB thrashing arises as z changes and when the different coefficient arrays are accessed. This access pattern induces reads from $NX \times NY \times 4B$ distance and even larger when reading different arrays, which are $NX \times NY \times NZ$ apart. For a sufficiently large domain, this requires the load of a new TLB page table pointer for every z. The TLB cache can handle only a handful of page pointers; thus, in such a situation, thrashing is more significant. Depending on the level of TLB page misses, the introduced latency further increases. Explaining the in-depth effects of TLB is out of the scope of this article; the reader is referred to Wong et al. [2010] for this information.

The solution for this problem is simple. One needs to preserve data locality when accessing the coefficients of the systems by mapping the threads to always solve the

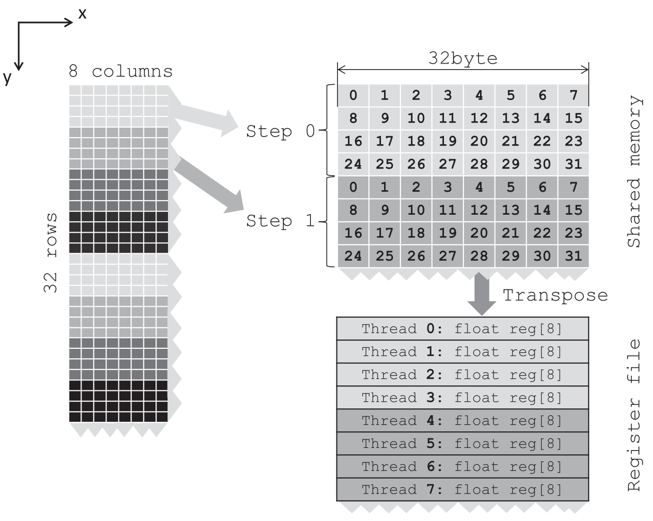


Fig. 8. Local transpose with shared memory.

closest neighboring system. One may map the threads with a 1-dimensional thread block according to:

ALGORITHM 7: Mapping Threads to Neighboring Systems

```
1: tid = threadIdx.x + blockIdx.x*blockDim.x
2: ind = tid*NX
```

The inefficiency due to noncoalesced memory access has also been shown in the slides of Sakharnykh [2009], and a global transposition solution has been given. This essentially means that the data has been transposed before performing the execution of the tridiagonal solve; therefore, data is read and written unnecessarily during the transposition. In this article, we give two local data transposition solutions, both of which keep the data in registers for the time of the tridiagonal solution. These solutions work in a register-blocking manner, thus avoid the need of the load and store of the global transposition, resulting in much higher efficiency.

6.1.1. Thomas Algorithm with Local Transpose in Shared Memory. Local data transpose can be performed in shared memory available on most GPU devices. The optimization is based on warp-synchronous execution; thus, there is no need for explicit synchronization. Although a warp size of 32 is assumed in the following, the implementation uses macros to define warp size, allowing for changes in the future. Threads within a warp cooperate to read data into shared memory. The data is then transposed via shared memory and stored back in registers. The threads perform 8 steps of the Thomas algorithm with the data in registers, then read the next 8 values, and so on. Meanwhile, the updated coefficients and intermediate values $\mathbf{c}^*$, $\mathbf{d}^*$ are stored in the local memory, which automatically creates coalesced and cached load/store memory transactions. The algorithm is shown in Algorithm 8 and further detailed in Figure 8.

The shared memory on an NVIDIA GPU can be accessed through 32 banks of width 32b or 64b, the latter applicable only from NVIDIA Compute Capability 3.0 and higher. Since the width of the block that is stored in shared memory is 8, this leads to shared

ALGORITHM 8: Thomas Algorithm with Shared Memory Transpose

```
Forward pass:
  1: Wrap a warp (32 threads) into $4 \times 8$ blocks to perform noncaching (32B) loads.
  2: Load $32 \times 8$ size tiles into shared memory: in 8 steps of $4 \times 8$ block loads.
  3: Transpose data by putting values into *float a[8]* register arrays.
  4: Perform Thomas calculation with the 8 values along the $X$ dimension.
  5: Repeat from Step 2 until end of $X$ dimension is reached.
Backward pass:
  1: Compute backward stage of Thomas in chunks of 8.
  2: Transpose results and store in global memory.
```

memory bank conflicts in the 3rd step of Algorithm 8. In the first iteration, the first 4 threads will access banks 0,8,16,24, the next 4 threads will again access 0,8,16,24, and so on. To overcome this problem, the leading dimension (stride) in shared memory block is padded to 9 instead of 8. This common trick helps to get rid of most of the bank conflicts. The first 4 threads will access banks 0,9,18,27, the next 4 threads will access banks 1,10,19,28, and so on. The amount of shared memory utilized for the 4 register arrays is $9 * 32 * 4 * 4\, bytes/warp = 4608\, bytes/warp$.

6.1.2. Thomas Algorithm with Local Transpose using Register Shuffle. Local data transposition can also be performed in registers exclusively. This solution fits recent NVIDIA GPU architectures, starting with the Kepler architecture with Compute Capability 3.0. The optimization is again based on warp-synchronous execution with the use of *__shfl()* register shuffle intrinsic instructions. Although a warp size of 32 is assumed in the following, the implementation uses C macros definitions to set warp size; thus, it allows for future architecture change. Threads within a warp cooperate to read a 32×16 SP block or 32×8 DP block of the x-y plane, that is, threads cooperate to read 16 SP or 8 DP values of 32 neighboring systems. Every 16 or 8 threads within the warp initiate a read of two cache lines (2×32B). They store the data into their register arrays of size 16 for SP and 8 for DP. These local arrays will be kept in registers for the same reasons that are discussed in Section 6.1.1. Once data is read, threads cooperatively distribute values with the XOR (Butterfly) transpose algorithm using the *__shfl_xor()* intrinsic in two steps. Once every thread has the data, each performs 16 SP or 8 DP steps of the Thomas algorithm with the data in registers, then reads the next 16 SP or 8 DP long array, and so on. Meanwhile, the updated coefficients and intermediate values $\mathbf{c}^*$, $\mathbf{d}^*$ are stored in the local memory, which automatically creates coalesced, cached load/store memory transactions. The algorithm is shown in Algorithm 9 and further detailed in Figure 9.

The use of the register shuffle instruction increases the register pressure compared to the shared memory version. However, since the data is kept in registers, all the operations within a 16 SP or 8 DP solve are performed in the fastest possible way; thus, the overall gain is significant.

6.2. Thomas Algorithm in Higher Dimensions

The Thomas algorithm gives itself to natural parallelization and efficient access pattern in the dimensions Y and above. The memory access patterns fit the coalesced way of reading and storing data. Therefore, the Thomas algorithm is efficient in these directions; the only limiting factor is the bandwidth throughput requirement posed by reading 3 coefficients and 1 RHS arrays, writing and later reading the 2 temporary arrays, and, finally, writing out the solution array.

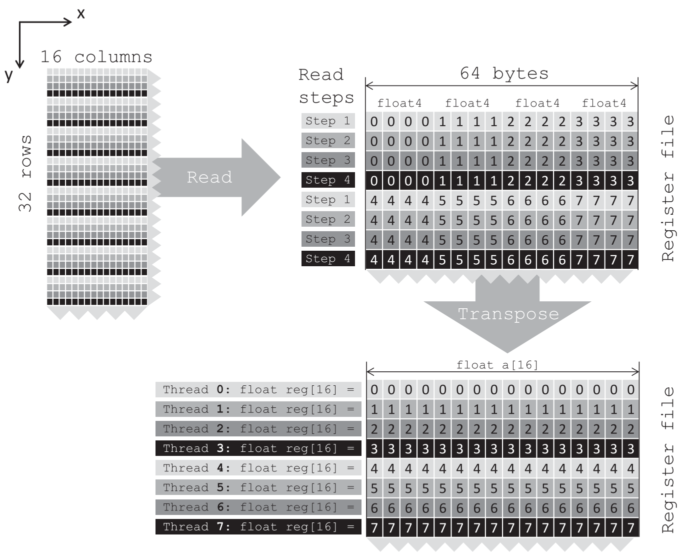

x
y
16 columns
32 rows
Read
Read steps
64 bytes
float4
float4
float4
float4
Step 1
Step 2
Step 3
Step 4
Register file
Transpose
float a[16]
Thread 0: float reg[16] =
Thread 1: float reg[16] =
Thread 2: float reg[16] =
Thread 3: float reg[16] =
Thread 4: float reg[16] =
Thread 5: float reg[16] =
Thread 6: float reg[16] =
Thread 7: float reg[16] =
Register file

Fig. 9. Local transpose with registers.

ALGORITHM 9: Thomas Algorithm with Register Shuffle Transpose

```
Forward pass:
 1: Wrap 32 threads into 8 × 4 blocks to perform 4× float4 vector loads.
 2: Load 32 × 16 size tiles into registers:
 3:    4 threads read 4 consecutive float4 vectors = 64B.
 4:    Do this 4 times for rows under each other.
 5: Transpose data within 4 threads:
 6:    4 threads exchange data on a 4 × 4 2D array with __shfl(float4).
 7:    Each element in the 2D array is a float4 vector.
 8: Perform Thomas calculation with the 16 values along X dimension.
 9: Repeat from 2 until end of X dimension is reached.
Backward pass:
 1: Compute backward stage of Thomas in chunks of 8.
 2: Transpose results and store in global memory.
```

6.3. Thomas-PCR Hybrid Algorithm

The Thomas-PCR hybrid algorithm introduced in Section 2.3 is implemented using either shared memory or register shuffle instructions. The advantage of the hybrid algorithm is that it allows for storing the entire system in the registers of a warp. Every thread in the warp is responsible for part of the system of size M; for example, if the system size is 256, then $M = 8$ and each thread stores an 8-long chunk of the 3 coefficients, the RHS, and the solution vector in its registers. This storage method puts a high pressure on register allocation, and it stays efficient only up to a system size that allows the subarrays to be stored in registers. Every thread individually performs the first phase of the modified Thomas algorithm, then switches to PCR to cooperatively solve the second phase of the hybrid algorithm. Meanwhile, the intermediate, modified coefficients and RHS values are kept in registers. Once the PCR finished, its solution can be used to solve the subsystems with the recently computed boundary values. In the third phase, the back-substitution of the modified Thomas algorithm is performed.

In the case of the X-dimensional solve, threads of a warp need contiguous chunks of size M of the whole array. This poses the same need for reading and transposing data in the X dimension, just like in Sections 6.1.1 and 6.1.2.

Data transposition can be performed using *shared memory* or register shuffles. To achieve good efficiency, warp-synchronous implementation is used in the solver. Threads of a warp read consecutive values in a coalesced way, in multiple steps of 32 values, and store data in shared memory. Then, each thread reads its own M values from shared memory.

ALGORITHM 10: Thomas-PCR Hybrid in X Dimension

```
1: Read $N$ values with 32 threads in a coalesced way.
2: Transpose array data so that every thread has its own $M$ interior values.
3: For each thread, compute interior $M$ elements in terms of outer 2.
4: The new tridiagonal system of size 2*32 is solved by PCR, using shuffles or data exchange
   through shared memory.
5: Perform back-substitution for interior elements in terms of outer 2 values.
```

In Y and higher dimensions, the access pattern fits the algorithm better, as in the case of the standard Thomas algorithm. In this case, there is no need for data transposition, but there is need for thread block synchronization. The solution along these dimensions is the most efficient with few restrictions. Algorithm 11 is similar to Algorithm 10 in terms of the underlying hybrid algorithm, with the significant difference that warps cooperate to solve multiple systems. Each thread within a warp solves an M size chunk. The chunks within a warp do not necessarily contribute to the same system. At a certain phase, warps share the necessary information to connect the matching systems. Sharing data is done through shared memory with thread block synchronization, but the solution of the reduced system is done with PCR the same way as in Algorithm 10.

ALGORITHM 11: Thomas-PCR Hybrid in Dimension Y and Above

```
1: Threads within a warp are grouped into $(W/G) \times G$ blocks, where $G$ is the number of
   systems solved by a warp.
2: The first $G$ threads read the first $M$ values of $G$ systems, the second $G$ threads read the
   second $M$ values, and so on.
3: Data of $G$ systems are now loaded into registers.
4: Every thread computes an independent $M$ size system: thread 0 computes the first $M$
   values of problem 0, thread $G$ computes the second $M$ values, thread $2G$ computes the
   second $M$ values, and so on.
5: Threads cooperate through shared memory to each reduced system into one warp.
6: Every warp computes an independent reduced system independently.
7: Reduced solution is propagated back to threads to solve interior systems.
8: Data is stored the same way as it was read.
```

7. PERFORMANCE COMPARISON

The performance of the solvers presented are discussed in terms of scalability on different system sizes and along different dimensions on all three architectures: NVIDIA GPU, Intel Xeon CPU, and Xeon Phi coprocessor. Later in this article, the GPU will also be referred to as SIMT architecture and the CPU and Xeon Phi collectively will also be referred to as SIMD architecture.

SIMD (CPU and Xeon Phi) measurement results are heavily contaminated with system noise. Therefore, significantly longer integration time had to be used for averaging execution time. However, there is a significant recurring spike in the execution time of the Xeon Phi results, which is worth discussing. In single precision, almost every system size with a multiple of 128 and almost every system size in double precision with a multiple of 64 results in a significant, more than $\times 2$ slowdown. This is the consequence of cache-thrashing. Cache-thrashing happens when cache lines on 2^n bytes boundaries with the same n are accessed frequently. These memory addresses have different tag IDs, but the same address within a tag. This means that threads accessing the two cache lines of such boundaries contaminate the shared L3 (or LLC) level cache; that is, they thrash the shared cache. This problem can be overcome if the cache architecture uses a set-associative cache with high associativity—at least as many threads are sharing the cache. This is true in the case of a modern processor. The size of the L2 cache is so low in the case of the Xeon Phi coprocessor that only $512KB/4threads = 128KB/thread$ is available, which might not allow for an efficient set associativity.

The scaling measurements in the following sections are run with fixed, $256^2 = 65536$ number of systems. The large number of systems ensures enough parallelism even for the Thomas solver, which by nature contains the most sequential computations, in the case of the GPU. Since GPUs need a tremendous amount of work to be done in parallel to hide memory latency, the 65536 parallel work units are enough to saturate the CUDA cores and, more important, the memory controllers. The length of the systems along the different dimensions are changed by extruding the dimension under investigation from 64 up to 1024, with steps of size 4. The resulting execution times are averaged over 100 to 200 iterations to integrate out system noise. The execution time reported is the proportion of the total execution time to one element of the three-dimensional cube, and does not include the data transfer to the accelerator card. This gives good reference to compare the different solvers, since it is independent of the number and length of the systems to be solved. It is expected that the execution time per element be constant for a solver, since the execution time is limited by the memory transfer bottleneck. The execution time differences of the algorithms presented in this section relate only to the memory transfer and memory access pattern of the particular algorithm. All implementations presented here utilize a whole cache line except the SIMD X solvers and the naïve GPU solver. The dependence on system size relates to running out of scratch memory (registers caching, L1/L2/L3 cache, TLB cache) for large systems and having enough workload for efficiency in the case of small systems. These dependences are discussed in the following in the discussion of the corresponding solver. The results are also compared against the *dtsvb()* function of the Intel Math Kernel Library 11.2 Update 2 for Linux library [Intel 2015] and the *gtsv()* function of NVIDIA's cuSPARSE library [NVIDIA 2015b]. The *dtsvb()* solver is a diagonally dominant solver that, according to the Intel documentation, is two times faster than the partial pivoting–based *gtsv()* solver. Details of the hardware used are discussed in Appendix A. Since the implementations for X and higher dimensions differ, we also need a separate discussion for these cases.

In a realistic scenario in high dimensions, the data transfer between the NUMA nodes is unavoidable since a NUMA-efficient layout for the X-dimensional solve is not efficient for the Y-dimensional solve and vice versa. Therefore, no special attention was made to handle any NUMA-related issues in the 2-socket CPU implementation for any dimensions.

7.1. Scaling in the *X* Dimension

Figures 10 and 11 show the performance scaling in the X dimension for single and double precision for all the architectures studied in this article. Figure 13 compares the solvers for a specific setup.

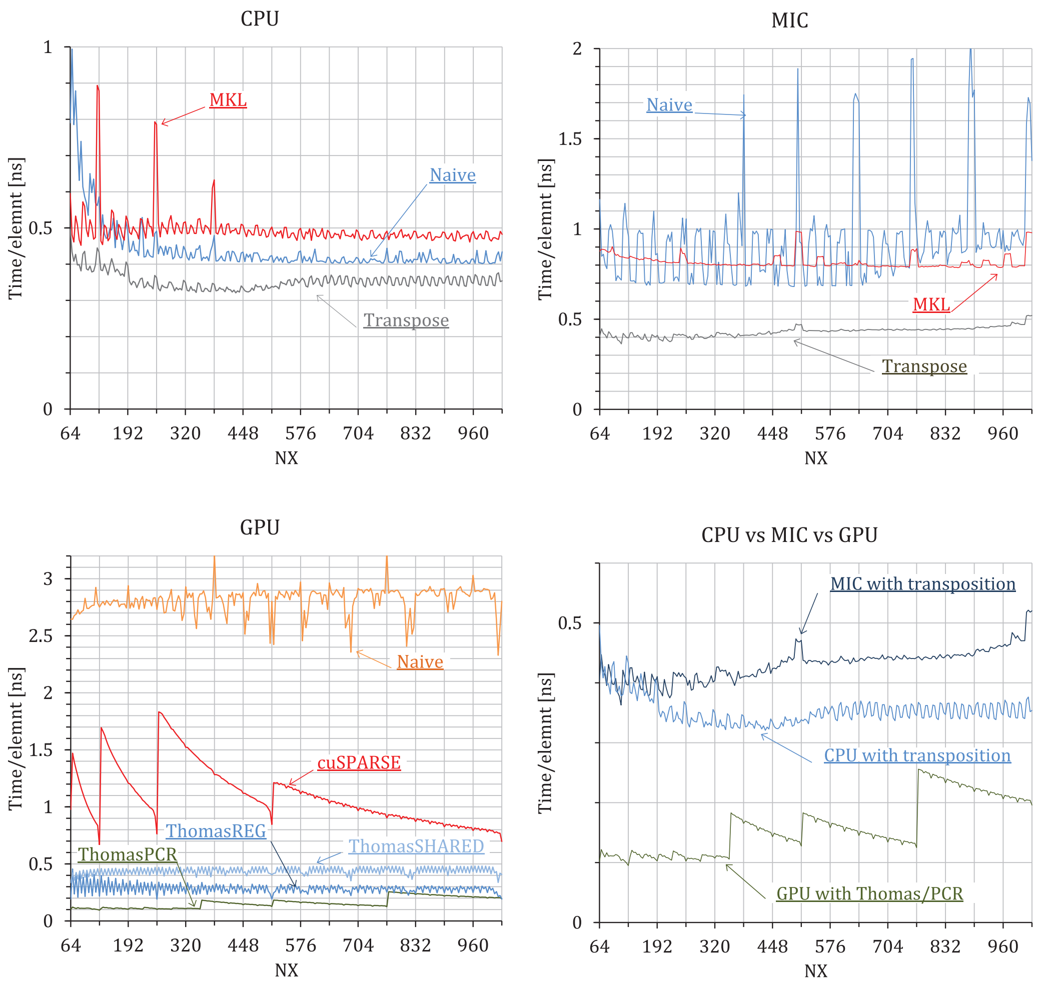


Fig. 10. Single-precision execution time per grid element. 65536 tridiagonal systems with varying *NX* length along the X dimension are solved. CPU, MIC, and GPU execution times, along with the comparison of the best solver on all three architectures, are shown.

7.1.1. SIMD Solvers. The SIMD solvers rely on the regular Thomas algorithm with or without local data transposition. The detailed description of these solvers is in Section 5. The MKL *dtsvb()* solver for diagonally dominant tridiagonal systems was chosen as the baseline solver. In the following comparison, all the presented SIMD implementations take advantage of multithreading with OpenMP. Threads using *KMP_AFFINITY=compact* were pinned to the CPU and MIC cores to avoid unnecessary cache reload when threads are scheduled to another core. Other runtime optimization approaches using *numactl* and *membind* were also considered, but they did not provide any significant speedup.

As seen in Figures 10 and 11, the naïve Thomas algorithm with OpenMP outperforms the MKL *dtsvb()* solver and the transposition-based Thomas solver further increases the speedup on a 2-socket Xeon processor. The naïve Thomas solver in the X dimension is not capable of utilizing any AVX instruction due to the inherent nature of vector units on CPUs; however, the transposition-based solvers are capable of taking advantage of

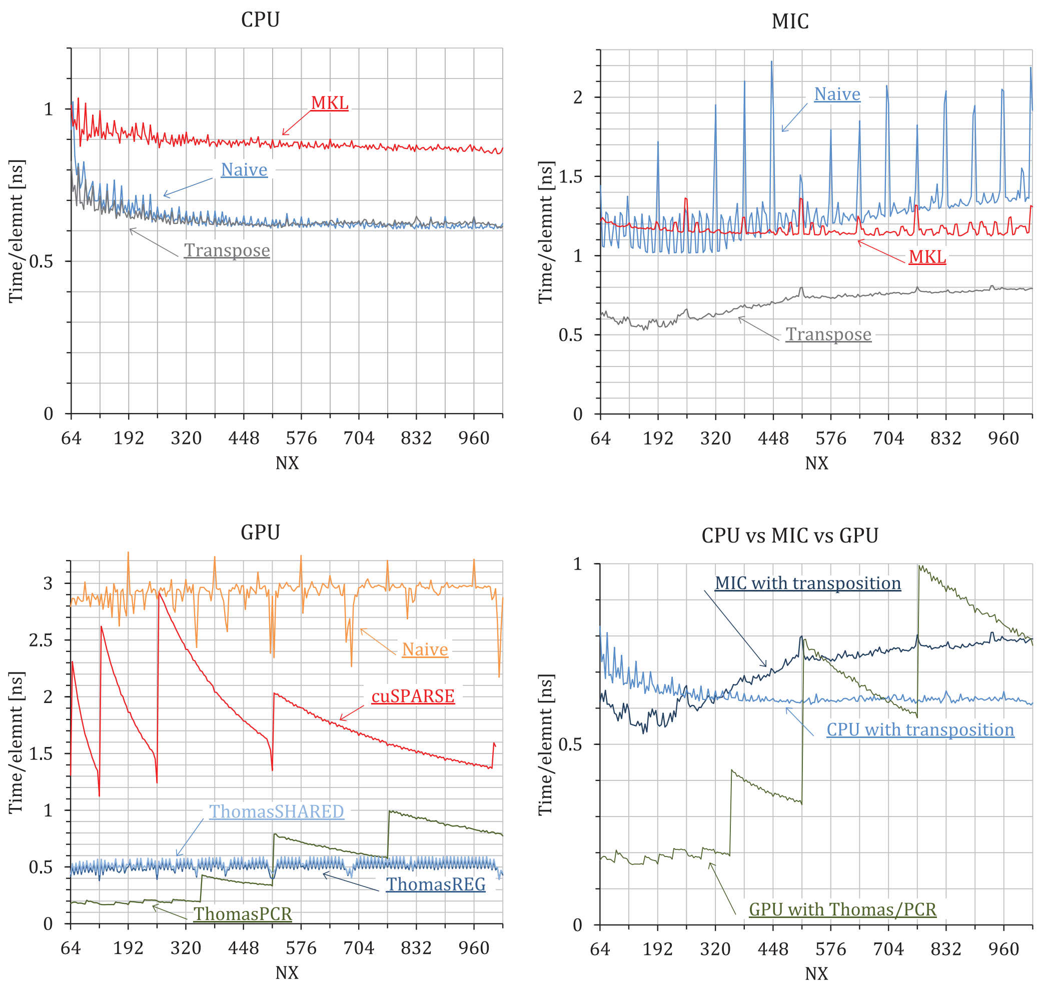


Fig. 11. Double-precision execution time per grid element. 65536 tridiagonal systems with varying NX length along the X dimension are solved. CPU, MIC, and GPU execution times, along with the comparison of the best solver on all three architectures, are shown.

the vector units as described in Section 5. The execution time declines and saturates in both single and double precision as the system size increases due to the workload necessary to keep threads busy on the CPU. Forking new threads with OpenMP has a constant overhead that becomes less significant when threads have more work. Above a certain system size, the CPU provides stable execution time. The efficiency of the CPU relies on the cache and out-of-order execution performance of the architecture. The size of the temporary arrays during the solution is small enough to fit into the L1 cache. The out-of-order execution is capable of hiding some of the 4 clock cycle latency of accessing these temporary cached values, which, in total, results in high efficiency.

The performance of the naïve Thomas algorithm on the Xeon Phi coprocessor is far from the expected. Due to the difficulty in vectorizing in the X dimension described in Section 5, the coprocessor needs to process the data with a scalar code. Since the scalar code is more compute limited than a vectorized code, and since the clock rate of the Xeon Phi is almost a third of the Xeon processor, the efficiency of the code drops to about a third of the vectorized code. However, a significant x2 increase in speedup is reached on the MIC architecture both in single and double precision when using the transposition-based Thomas solver. The performance increase due to vectorization would imply an 8 or 16 times speedup in single or double precision, respectively; however, the underlying execution architecture, compiler, and caching mechanism is not capable of providing this speedup. The overall performance of the Xeon Phi with the transposition-based Thomas algorithm is comparable to the CPU for small system size and becomes slower than the CPU as the system size increases above 200 to 300. Significant spikes in execution time of the naïve solver can be seen in both single and double precision on almost every 512B step, either in steps of 128 in single precision or in steps of 64 in double precision. Thread pining with the *KMP_AFFINITY=compact* option prevents thread migration and improves the performance of the coprocessor significantly. To understand the difference between the CPU and the MIC architecture, the reader is suggested to study the ISA manuals [Intel 2012a] and [Intel 2012b] of the two architectures. The two architectures are radically different. The CPU works at high clock rates with complex control logic, out-of-order execution, branch prediction, low latency instructions, and large, low latency distributed cache per thread. The MIC (Knights Corner) architecture is the opposite in many of these properties. It works at low clock rate, has simple control logic, in-order execution, no branch prediction, higher latency instructions, and small, higher-latency cache per thread. These facts suggest that, when the input data is fetched into the cache, the heavyweight CPU cores utilize this data much more efficiently than the lightweight MIC cores. One may note that the upcoming Intel Knights Landing architecture, with its out-of-order execution unit, may significantly improve the MIC performance.

7.1.2. SIMT Solvers. One may notice that the worst performance is achieved by the naïve GPU solver—even worse than any SIMD implementation. The inefficiency comes from the poor cache-line utilizations discussed in Section 6. The measurements are contaminated with deterministic, nonstochastic noise that may come from cache efficiency, cache thrashing, and so on. This noise cannot be attenuated by longer integration time. Due to the low cache-line utilization, the implementation is fundamentally latency limited. This is supported by the fact that the execution time is almost the same for the single- and double-precision cases. Every step of the solver requires the load of a new cache line, and since cache lines on GPU architectures are not prefetched, both single- and double-precision versions move the same amount of data (32B cache line) through the memory bus.

The cuSPARSE v7.0 solver provides better performance than the naïve solver, but it also has a significant recurring dependence on system size. The *cusparse{S,D}gtsvStridedBatch()* solver performs the best when the system size is a power of two in both single- and double-precision cases. The performance can vary approximately by two times speed difference in both single and double precision. This dependence is due to the internal implementation of the hybrid CR/PCR algorithm that is used inside the solver [NVIDIA 2015b]. The performance of this solver is even worse than the MKL *dtsvb()*library function on the CPU in both single and double precision. The slowdown of cuSPARSE versus the MKL solver is only valid for the regime of short systems in the order of thousands of length. For larger systems, this tendency changes for the advantage of the cuSPRASE library. Since the practical applications detailed in the introductory section involve the solution of systems below the thousand size regime, systems above this limit are not the scope of the article; thus, we do not elaborate on these differences further.

The transposition-based Thomas solvers perform better than cuSPARSE by a factor of $2.5 - 3.1$ in the case of transposition in shared memory and by $4.3 - 3.3$ in the case of

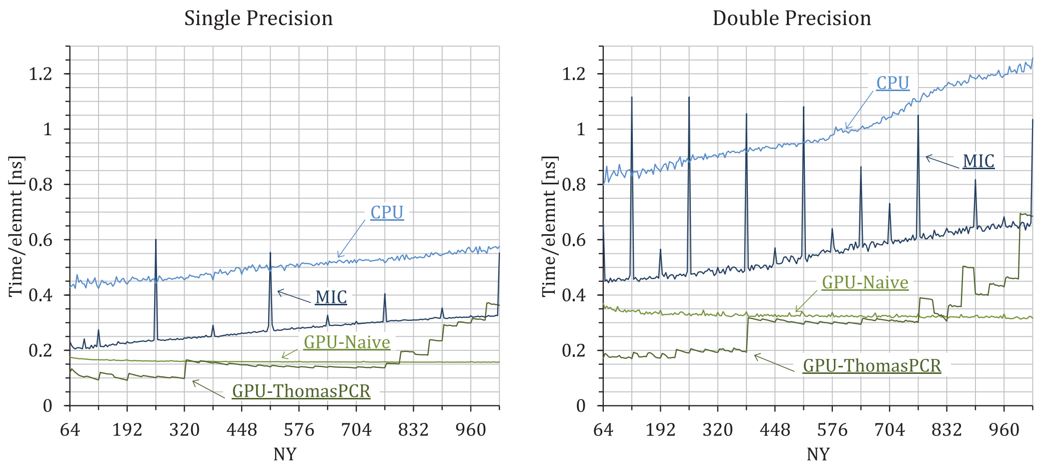


Fig. 12. Single- and double-precision execution time per grid element. 65536 tridiagonal systems with varying NY length along the Y dimension are solved. CPU, MIC, and GPU execution times, along with the comparison of the best solver on all three architectures, are shown.

register shuffle depending on floating-point precision and system size. The advantage of making an extra effort for improving the cache-line utilization is obvious. The achieved speedup compared to the naïve Thomas solver is $6.1 - 7.2$ in the case of single and double precision, respectively. The efficiency of the Thomas solver with transposition remains constant as the system size increases; that is, there is no significant fluctuation in performance.

One may notice that the padding is important for the register transposition so that aligned *float4* or *double2* loads may be done, which cannot always be ensured. There is a factor 2 speed difference between the shared memory transposition and the register shuffle transposition in single precision. This difference is negligible in double precision since 64b-wide shared memory bank access is used to access the scratch memory instead of 32b in the case of single precision. The wider bank access has been introduced in the Kepler architectures and effectively doubles the bandwidth of the shared memory when 64b access can be ensured. In the case of shared memory, a read throughput of 1211GB/s and store throughput of 570GB/s is measured with NVVP (NVIDIA Visual Profiler) on a 256^3 single-precision cube domain. Also, the transposition using shared memory happens by reading 32B at a time, whereas, in the case of the register shuffle-based transposition, 64B are read from the global memory at a time.

The hybrid Thomas-PCR algorithm outperforms every solver in the X dimension in single precision. In double precision, however, the performance drops significantly beyond system size 512. The efficiency is due to the register-blocking nature of the algorithm. Each system that is solved is completely resident in registers; therefore, only the input data is read once and the output is stored once. This results in the minimum amount of data transfers and leads to the best possible performance.

7.2. Scaling in Higher Dimensions

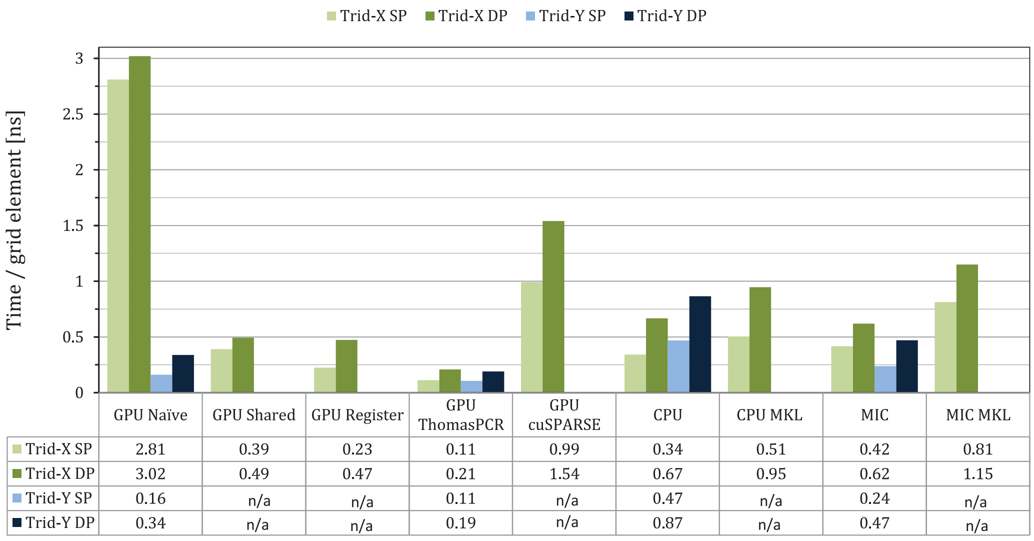
Figure 12 shows the performance scaling in the Y dimension for single and double precision for all the architectures studied in this article. Since the solution of tridiagonal systems with nonunit stride is not implemented in any library up to date, the higher-dimensional execution-time benchmarks do not contain any standard library execution times. Transposing the data structure would allow for the use of the standard solvers, but it has not been done for two reasons: (1) the efficiency of transposing the whole data structure would involve further optimization of implementations attached to standard library code, and the overall performance would be influenced by the extra programming effort; and (2) even with a highly optimized transposition, the overall execution time would be higher than in the X dimension.

7.2.1. SIMD Solvers. The SIMD solvers rely on the regular Thomas algorithm. The detailed description is in Section 5. Both the CPU and the MIC SIMD architectures are able to utilize the 256B- and 512B-wide AVX and IMCI vector units to solve a tridiagonal system. This means that 8(4) and 16(8) systems can be solved by a single thread in parallel in single(double) precision on the CPU and MIC, respectively. The efficiency of the CPU relies on the cache and out-of-order execution performance of the architecture. The size of the temporary arrays for the systems solved in parallel is small enough to fit into the L1 cache and the L2 cache. The out-of-order execution is capable of hiding some of the 4 clock cycle latency of accessing these temporary in the L1 cache. In Figure 12, it can be seen that the CPU and MIC performance changes in parallel. Although the scaling should be constant, it is changing almost linearly in the case of single precision and super linearly in double precision. The reason for the latter is that the cores are running out of L1 cache and the L2 cache performance starts to dominate. Two arrays (c^* and d^*) need to be cached for good performance. For instance, system size $N = 1024$ needs $1024 element * 8 bytes/element * 2 arrays * 4 parallel systems = 64KB$ to be cached, which is twice the size of the L1 cache. The single-precision solver remains linear in this regime, because even 1024-long system fits into the L1 cache. The order of the two for loops (X and Z) iterating on the 65526 systems have set so that better data locality is achieved; thus, a better TLB hit rate is reached. The Thomas solver on the dual socket Xeon CPU is the slowest among all the solvers. The MIC architecture is by a factor $1.8 - 2$ faster than the CPU. The SIMD architectures require aligned loads for best performance, which can be ensured with padding; otherwise, the performance is hit by unaligned data loads and stores. The vector operations can still be performed; the nonalignment hits only data transfer.

7.2.2. SIMT Solvers. The naïve GPU solver provides stable execution time and is up to 3.6(3.8) times faster than the dual socket Xeon CPU and 2.1(2.5) times faster than the Xeon Phi in single(double) precision. The GPU implementation is capable of solving 32 systems within a warp using coalesced memory access. The performance is therefore predictable and very efficient. The only drawback of the solver is that it is moving more data than the Thomas-PCR hybrid solver (detailed in Section 2.3), which caches the data in registers. Therefore, the Thomas-PCR hybrid algorithm is up to 1.5(1.8) faster than the naïve GPU solver in the case of single(double) precision. Compared to the highly optimized 2-socket CPU solver, the Thomas-PCR solver is 4.3(4.6) faster in single(double) precision. Compared to the highly optimized MIC, the speedup is $2.2 - (2.5)$. These are significant differences that can be maintained until there is enough register to hold the values of the processed systems. Once the compiled code runs out of register, the effect of register spill becomes obvious, since the execution time jumps by more the 50%; this happens with system size 320 in single precision and 384 in double precision. In the case of register spill, the advantage of the Thomas-PCR over the naïve solver is negligible, and above certain system size, it is even worse.

8. CONCLUSION FOR SCALAR TRIDIAGONAL SOLVERS

In the past, many algorithms have been introduced to solve a system of tridiagonal equations, but only a few took advantage of the fact that, in certain cases (for example, an ADI solver), the problem to be solved contains a large number of small systems to solve and the access pattern of the system elements might change in data structures with 2 dimensions and above. It has been shown that, in the X dimension, the standard



	GPU Naïve	GPU Shared	GPU Register	GPU ThomasPCR	GPU cuSPARSE	CPU	CPU MKL	MIC	MIC MKL
🟩 Trid-X SP	2.81	0.39	0.23	0.11	0.99	0.34	0.51	0.42	0.81
🟩 Trid-X DP	3.02	0.49	0.47	0.21	1.54	0.67	0.95	0.62	1.15
🟦 Trid-Y SP	0.16	n/a	n/a	0.11	n/a	0.47	n/a	0.24	n/a
⬛ Trid-Y DP	0.34	n/a	n/a	0.19	n/a	0.87	n/a	0.47	n/a

Fig. 13. Grid element execution times per grid element on a 240 × 256 × 256 cube domain. X dimension is chosen to be 240 to alleviate the effect of caching and algorithmic inefficiencies that occur on sizes of power of 2, as can be seen in Figures 10, 11, and 12. Missing bars and n/a values are due to nonfeasible or nonadvantageous implementations in the Y dimension.

Thomas algorithm with modified access pattern using local data transposition on the CPU, MIC, and GPU can outperform library quality, highly optimized code, such as (1) *dtsvb()* MKL diagonally dominant solver running on multiple threads on a dual socket Xeon processor and (2) the PCR-CR hybrid algorithm implemented in the cuSPARSE library provided by NVIDIA. If the system size allows caching in registers, then a new proposed Thomas-PCR hybrid algorithm on the GPU can be used to solve the problem even more efficiently with a speedup of about 2 compared to the Thomas algorithm with local data transposition. It has been shown that in higher dimensions (Y, Z, and so on) the naïve solver allows for coalesced (or almost coalesced) access pattern; therefore, there is no need for transposition and the performance is high. The Thomas-PCR for X dimension is modified to handle systems in higher dimensions by using more warps to load and store the data required in the computation. The register and shared memory pressure is higher in these cases, and register spills occur above system size 320 in single and 384 in double precision. The Thomas algorithm performs better for above these system sizes. Figure 13 summarizes the execution times for the X and Y dimensions in the case of a 240 × 256 × 256 domain.

The conclusion is that the Thomas algorithm with modified access pattern is advantageous up to relatively large system sizes in the order of thousands. The Thomas-PCR hybrid gives better performance in the X dimension in this regime. In the Y dimensions and above, the Thomas-PCR is the best performing up to system size 320(384) in single(double) precision, but above this size the Thomas regains its advantage due to its simplicity.

BLOCK TRIDIAGONAL SOLVER ON SIMD AND SIMT ARCHITECTURES

1. INTRODUCTION

In many real-world CFD and financial applications, the multidimensional PDEs have interdependent state variables. The state variable dependence creates a block structure in the matrix used in the implicit solution of the PDE. In certain cases, these matrices

ALGORITHM 12: Block Thomas Algorithm

```
Require: $block\_thomas(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{d})$
    Forward pass
 1: $\mathbf{C_0}^* = \mathbf{B_0^{-1}C_0}$
 2: $\mathbf{d_0}^* = \mathbf{B_0^{-1}d_0}$
 3: for $i = 1, \ldots, N-1$ do
 4:     $\mathbf{C_i}^* = (\mathbf{B_i} - \mathbf{A_i C^*_{i-1}})^{-1}\mathbf{C_i}$
 5:     $\mathbf{d_i}^* = (\mathbf{B_i} - \mathbf{A_i C^*_{i-1}})^{-1}(\mathbf{d_i} - \mathbf{A_i d^*_{i-1}})$
 6: end for
    Backward pass
 7: $\mathbf{u_{N-1}} = \mathbf{d^*_i}$
 8: for $i = N-2, \ldots, 0$ do
 9:     $\mathbf{u_i} = \mathbf{d^*_i} - \mathbf{C^*_i u_{i+1}}$
10: end for
11: return u
```

are formed to be tridiagonal, with block matrices in the diagonal and off-diagonals. The M^2 block sizes are usually in the range of $M = 2, \ldots, 8$; therefore, the tridiagonal matrix takes the forms shown in Equations (4) and (5),

$$\mathbf{A_i u_{i-1}} + \mathbf{B_i u_i} + \mathbf{C_i u_{i+1}} = \mathbf{d_i} \tag{4}$$

$$\begin{pmatrix} \mathbf{B_0} & \mathbf{C_0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{A_1} & \mathbf{B_1} & \mathbf{C_1} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{A_2} & \mathbf{B_2} & \mathbf{C_2} & \cdots & \mathbf{0} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{A_{N-1}} & \mathbf{B_{N-1}} \end{pmatrix} \begin{pmatrix} \mathbf{u_0} \\ \mathbf{u_1} \\ \mathbf{u_2} \\ \vdots \\ \mathbf{u_{N-1}} \end{pmatrix} = \begin{pmatrix} \mathbf{d_0} \\ \mathbf{d_1} \\ \mathbf{d_2} \\ \vdots \\ \mathbf{d_{N-1}} \end{pmatrix}, \tag{5}$$

where $\mathbf{A_i}, \mathbf{B_i}, \mathbf{C_i} \in \mathbb{R}^{MxM}$, $\mathbf{u}_i, \mathbf{d}_i \in \mathbb{R}^M$ and $M \in [2, 8]$.

Algorithms for solving a block-tridiagonal system of equations have been previously developed in Hirshman et al. [2010], Seal et al. [2013], and van der Vorst [1987]. Stone et al. [2011] gave a GPU-based solution using the block PCR algorithm, motivating their choice by the inherent parallelism given by the algorithm and the demand for high parallelism by the GPU. The overall arithmetic complexity of the PCR is known to be higher than that of the Thomas algorithm, as detailed in Section 2. As many CFD applications consider block sizes of $M = 2, \ldots, 8$, the motivation of our work is to make use of the computationally less expensive algorithm—namely, the Thomas algorithm—and exploit parallelism in the block-matrix operations. In other words, the overall arithmetic complexity is kept low by the Thomas algorithm and the parallelism is increased by the work-sharing threads that solve the block-matrix operations.

Stone et al. [2011] used a highly interleaved Structure of Arrays (SOA) storage format to store the coefficients of the systems in order to achieve a coalesced memory access pattern suitable for the GPU. In that approach, the data on the host is stored in the Array of Structures (AOS) format, which had to be converted into the SOA format to suit the needs of the GPU. In our approach, the work-sharing and register-blocking solution alleviates the need for AOS-SOA conversion, that is, the data access efficiency remains high.

2. BLOCK-THOMAS ALGORITHM

The block structure introduces matrix operations such as (1) block matrix inversion with $O(M^3)$; (2) block matrix-matrix multiplication with $O(M^3)$ and addition with $O(M^2)$; and (3) block matrix-vector multiplication with $O(M^2)$. On the other hand, the

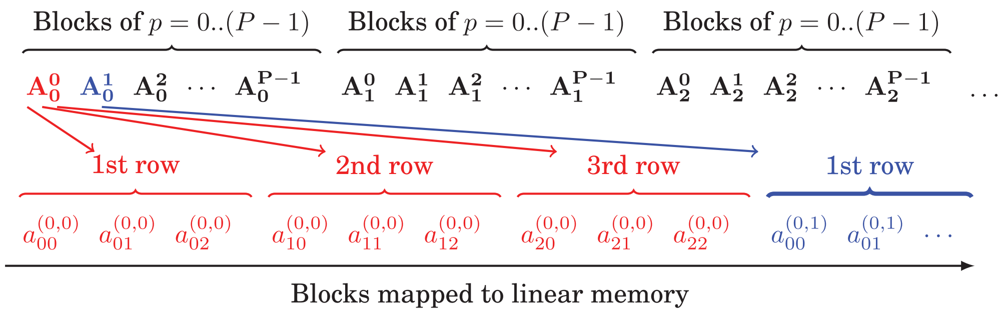

Blocks of $p = 0..(P-1)$
Blocks of $p = 0..(P-1)$
Blocks of $p = 0..(P-1)$
$\mathbf{A_0^0}$ $\mathbf{A_0^1}$ $\mathbf{A_0^2}$ $\cdots$ $\mathbf{A_0^{P-1}}$ $\mathbf{A_1^0}$ $\mathbf{A_1^1}$ $\mathbf{A_1^2}$ $\cdots$ $\mathbf{A_1^{P-1}}$ $\mathbf{A_2^0}$ $\mathbf{A_2^1}$ $\mathbf{A_2^2}$ $\cdots$ $\mathbf{A_2^{P-1}}$ $\cdots$
1st row
2nd row
3rd row
1st row
$a_{00}^{(0,0)}$ $a_{01}^{(0,0)}$ $a_{02}^{(0,0)}$ $a_{10}^{(0,0)}$ $a_{11}^{(0,0)}$ $a_{12}^{(0,0)}$ $a_{20}^{(0,0)}$ $a_{21}^{(0,0)}$ $a_{22}^{(0,0)}$ $a_{00}^{(0,1)}$ $a_{01}^{(0,1)}$ $\cdots$
Blocks mapped to linear memory

Fig. 14. Data layout within coefficient array **A**. This layout allows nearly coalesced access pattern and high cache hit rate when coalescence criteria are not met. Here, $\mathbf{A_n^p}$ is the nth block (i.e., nth block-row in the coefficient matrix) of problem p and $p \in [0, P-1]$, $n \in [0, N-1]$. Notation for the scalar elements in the nth block of problem p is shown in Equation (6). The bottom part of the figure shows the mapping of scalar values to the linear main memory.

data transfer is only $O(M^2)$ per block matrix. As the matrix operations are dominated by $O(M^3)$ versus the $O(M^2)$ data transfer, the solution of the block tridiagonal system becomes compute limited as the block size increases. Once the data is read and stored in scratch memory, the cost of making the matrix operations is the bottleneck, both because arithmetic complexity and control flow complexity are significant. Let us define the O complexity in terms of block matrix operations with the arithmetic complexity states presented earlier, and define the total work as the complexity of solving a single system times the number of systems to be solved on a given number of parallel processors. When solving N-long systems on N processors, the Thomas algorithm has $N\,O(N)$ total work complexity versus the $N\,O(N \log N)$ total work complexity of the PCR algorithm. This significant difference establishes the use of the Thomas algorithm over the PCR in the block tridiagonal case.

The block Thomas algorithm is essentially the same as the scalar algorithm, assuming that scalar operations are exchanged with matrix operations. The lack of commutative property of the matrix multiplication, the order of these matrix operations have to be maintained throughout the implementation. See Algorithm 12 for details.

3. DATA LAYOUT FOR SIMT ARCHITECTURE

The data layout is a critical point of the solver, since this influences the effectiveness of the memory access pattern. Coalesced memory access is needed to achieve the best theoretical performance; therefore, SOA data structures are used in many GPU applications. In this section, an AOS data storage is presented in which blocks of distinct tridiagonal system are interleaved. Block coefficients $\mathbf{A_n^p}$, $\mathbf{B_n^p}$, $\mathbf{C_n^p}$, $\mathbf{d_n^p}$ are stored in separate arrays. The data layout of **A**, **B**, **C**, and $\mathbf{C}^*$ coefficient block arrays are the same. Within the array of blocks, the leading dimension is the row of a block; that is, blocks are stored in row major format. The block of system p is followed by the block of system block $p+1$ in the array; that is, the blocks in array **A** are stored by interleaving the nth blocks of problems $p = 0, \ldots, P-1$ in the way shown in Figure 14.

$$\mathbf{A_n^p} = \begin{pmatrix} a_{00}^{(n,p)} & a_{01}^{(n,p)} & a_{02}^{(n,p)} \\ a_{10}^{(n,p)} & a_{11}^{(n,p)} & a_{12}^{(n,p)} \\ a_{20}^{(n,p)} & a_{21}^{(n,p)} & a_{22}^{(n,p)} \end{pmatrix} \tag{6}$$

Vectors **d**, $\mathbf{d}^*$, and **u** are stored in a similar interleaved fashion, as depicted in Figure 14. Subvectors in array **d** are stored by interleaving the nth subvector of problems

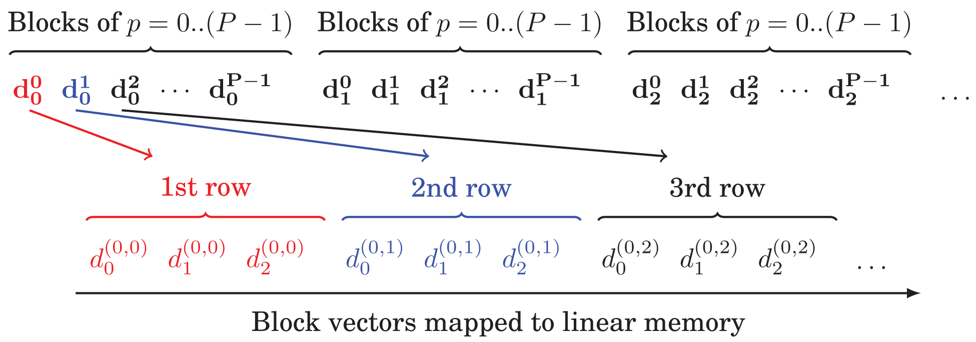

Blocks of $p = 0..(P-1)$
Blocks of $p = 0..(P-1)$
Blocks of $p = 0..(P-1)$
$\mathbf{d}_0^0$ $\mathbf{d}_0^1$ $\mathbf{d}_0^2$ $\cdots$ $\mathbf{d}_0^{P-1}$
$\mathbf{d}_1^0$ $\mathbf{d}_1^1$ $\mathbf{d}_1^2$ $\cdots$ $\mathbf{d}_1^{P-1}$
$\mathbf{d}_2^0$ $\mathbf{d}_2^1$ $\mathbf{d}_2^2$ $\cdots$ $\mathbf{d}_2^{P-1}$
1st row
2nd row
3rd row
$d_0^{(0,0)}$ $d_1^{(0,0)}$ $d_2^{(0,0)}$ $d_0^{(0,1)}$ $d_1^{(0,1)}$ $d_2^{(0,1)}$ $d_0^{(0,2)}$ $d_1^{(0,2)}$ $d_2^{(0,2)}$ $\dots$
Block vectors mapped to linear memory

Fig. 15. Data layout of **d** as block vectors. This layout allows nearly coalesced access pattern and high cache hit rate when coalescence criteria are not met. Here, $\mathbf{d}_\mathbf{n}^\mathbf{p}$ is the nth block (i.e., nth block-row in the coefficient matrix) of problem p and $p \in [0, P-1]$, $n \in [0, N-1]$. Notation for the scalar elements in the nth block of problem p are shown in Equation (7). The bottom part of the figure shows the mapping of scalar values to the linear main memory.

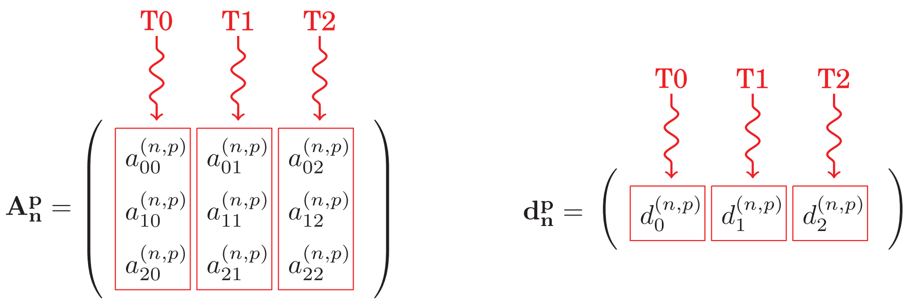

T0
T1
T2
$\mathbf{A}_\mathbf{n}^\mathbf{p} =$
$a_{00}^{(n,p)}$ $a_{01}^{(n,p)}$ $a_{02}^{(n,p)}$
$a_{10}^{(n,p)}$ $a_{11}^{(n,p)}$ $a_{12}^{(n,p)}$
$a_{20}^{(n,p)}$ $a_{21}^{(n,p)}$ $a_{22}^{(n,p)}$
T0
T1
T2
$\mathbf{d}_\mathbf{n}^\mathbf{p} =$
$d_0^{(n,p)}$ $d_1^{(n,p)}$ $d_2^{(n,p)}$

Fig. 16. Shared storage of blocks in registers. Arrows show load order. Thread $T0$, $T1$ and $T2$ stores the first, second, and third columns of block $\mathbf{A}_\mathbf{n}^\mathbf{p}$ and first, second, and third values of $\mathbf{d}_\mathbf{n}^\mathbf{p}$.

$p = 0, \dots, P-1$ in the way shown in Figure 15. The notation of the scalar values of $\mathbf{d}_\mathbf{n}^\mathbf{p}$ can be seen in Equation (7).

$$\mathbf{d}_\mathbf{n}^\mathbf{p} = \left(d_0^{(n,p)} \ d_1^{(n,p)} \ d_2^{(n,p)} \right)^T \tag{7}$$

This data layout allows the threads to load data through texture cache from global memory with efficient access pattern and cache-line utilization. A block is read in M steps. In each step, a row of the block is read. The scalar values are stored in the registers, as shown in Figure 16. In the second step, threads read the second row in the same manner, and so on.

This storage format allows for high texture cache hit rate by ensuring that most of the data that is read is within a 32B cache line. Let us take the case of $M = 3$ in single precision when reading A_0^0. In the first step, only the first row of the block is read; that is, 12B of the 32B cache line is utilized. Once a texture cache line is loaded, it will be reused immediately in the following few instructions or by the neighboring thread. In the next step, when the second row of the block is read, the cache line is already in the texture cache; this time, 12B are directly read from the texture cache. In the third step, when the third row is read, the 12B are again in the cache, since 8B out of the 12B are in the same cache line as the first two rows, and the remaining 4B are in the cache line read by the next group of 3 threads that read the block A_0^1. All this is done in parallel by $\lfloor 32/3 \rfloor = 30$ threads. The total amount of data that is read by the warp (i.e., 10 groups of threads) is $10 \times 9 \times 4 = 360 bytes$, which fits into 12 cache lines. The probability of a cache line being evicted is low, since the cache lines are reused by the threads in the same warp. Since the sequence of instructions of loading a block does not contain data dependency, there is no reason for the scheduler to idle the active threads that started loading the data.

4. COOPERATING THREADS FOR INCREASED PARALLELISM ON SIMT ARCHITECTURE

SIMT architectures are inherently sensitive to parallelism; that is, they need enough running threads to hide the latency of accessing the memory and filling up the arithmetic pipelines efficiently. Also, the implemented solver has to consider the constraints of the processor architecture being used. The Kepler GK110b architecture has 48KB shared memory and 64K 32b registers available for use within a thread block with up to 255 registers/thread. Computing block matrix operations with a single thread is not efficient due to the following reasons: (1) the limited number of registers and shared memory does not allow for temporary storage of block matrices and reloading data from global memory is costly; and (2) in order to utilize coalesced memory access, a high level of input data interleaving would be required, which is not useful in a real application environment. Consequently, the problem would become memory bandwidth limited rather than compute limited. The workload of computing a single system therefore needs to be shared among threads, so that block matrix data is distributively stored in the registers (and shared memory) of threads. This means that both workload and data storage is distributed among the cooperating threads.

We propose using M number of threads to solve the M-dimensional block matrix operations, so that every thread stores one column of a block and one scalar value of a vector that is being processed (see Figure 16 for details). Subsequent M threads are computing a system and a warp computes $\lfloor 32/M \rfloor$ number of systems. This means that $\lfloor 32/M \rfloor * M$ threads are active during the computation. The rest are idle. With this work distribution in the $M = 2, .., 8$ range, the worst case is $M = 7$ when 4 out of 32 threads are inactive; thus, 87.5% is the actual computation performance that can be reached.

The communication to perform matrix operations is carried out using either shared memory or register shuffles (*__shfl()* intrinsic). Just like in the case of the scalar solver, register and shared memory is not enough to store the intermediate $\mathbf{C}^*$, $\mathbf{d}^*$ block arrays; for this purpose, local memory is used. This is an elegant way of getting coalesced memory access.

In the following, the essential block matrix operations are discussed using Figure 16. Note that all the implementation with register shuffle is written in a way that the local arrays storing block columns are held in registers. Two criteria need to be satisfied to achieve this: (1) local array indexing needs to be known at compile time, and (2) local array size cannot exceed a certain size defined internally in the compiler.

Matrix-Vector Multiplication. When performing matrix-vector multiplication, threads have the column of a block and the corresponding scalar value to perform the first step of matrix-vector multiplication—namely scalar-vector multiplication. This is done in parallel, independently by the threads. The second step is to add up the result vectors of the previous step. In this case, threads need to share data, which can either be done through shared memory or using register shuffle instructions. In the case of *shared memory,* the result is stored in shared memory. It is initialized to 0. In the mth step (where $m = 0, \dots, M-1$), thread m adds its vector to the shared vector and thread block synchronizes. In the case of *register shuffle*, the multiplication is also done in M steps. In the mth step, the M threads compute a scalar product of the mth row and

$$\overbrace{\mathbf{A_0^0}\ \mathbf{A_1^0}\ \mathbf{A_2^0}\ \cdots\ \mathbf{A_{N-1}^0}}^{\text{Blocks of problem } p=0}\ \overbrace{\mathbf{A_0^1}\ \mathbf{A_1^1}\ \mathbf{A_2^1}\ \cdots\ \mathbf{A_{N-1}^1}}^{\text{Blocks of problem } p=1}\ \overbrace{\mathbf{A_0^2}\ \mathbf{A_1^2}\ \mathbf{A_2^2}\ \cdots\ \mathbf{A_{N-1}^2}}^{\text{Blocks of problem } p=2}\ \ \cdots$$

Fig. 17. Data layout within coefficient array $\mathbf{A}$ suited for better data locality on a CPU and MIC. Here, $\mathbf{A_n^p}$ is the nth block (i.e., nth block-row in the coefficient matrix) of problem p and $p \in [0, P-1]$, $n \in [0, N-1]$. Notation for the scalar elements in the nth block of problem p are shown in Equation (6).

shuffle the computed values around (in round-robin) to accumulate these values. The actual addition of the accumulation is done by the mth thread.

Matrix-Matrix Multiplication. Matrix-matrix multiplication needs to communicate the $M \times M$ values of one of the blocks. This can either be done through shared memory or register shuffle. In the case of shared memory, this approach uses M^2 number of *__syncthreads()*, which would suggest a heavy impact on performance, but the results are still satisfying. The register shuffle approach does not require synchronization; thus, it is expected to work faster than the shared memory approach.

Gauss-Jordan Elimination. The solution of block systems in Lines 4 and 5 in Algorithm 12 is done by immediately solving these systems when performing a Gauss-Jordan elimination; that is, the systems are solved without composing the explicit inverse of matrix $\mathbf{B_i} - \mathbf{A_i}\mathbf{C^*_{i-1}}$. The Gauss-Jordan elimination is computed cooperatively by using either shared memory of register shuffle instructions. Both versions have high compute throughput.

5. OPTIMIZATION ON SIMD ARCHITECTURE

The approach for an efficient solution in the case of SIMD architectures is different in terms of data storage and execution flow: data layout is changed to get better data locality when each independent system is solved by an individual CPU thread. This approach is a more suitable way of performing computation on multicore systems. As each thread solves an independent system, better cache locality is achieved when the blocks of array A are reordered to store the blocks $A_0^p, A_1^p, \ldots, A_N^p$ next to each other. Figure 17 depicts this in more detail.

In these systems, threads are heavyweight with out-of-order execution and have enough L1 cache to efficiently solve a block tridiagonal system. The C++ code is optimized to take advantage of the vector units. Loops are written so that the requirements for auto-vectorization are satisfied. Where needed, *#pragma ivdep* or *#pragma simd* is used to help and force the compiler to vectorize loops. Dimension sizes are passed through template variables to make better use of compile-time optimizations such as loop unrolling, vectorization, and static indexing. Thread-level parallelism is provided by OpenMP and threads solving independent systems are scheduled with guided scheduling to balance workload and provide an overall better performance.

6. PERFORMANCE COMPARISON AND ANALYSIS

Performances of the implemented solvers are compared in terms of memory bandwidth, computation throughput in the case of GPUs, and speedups of GPU and CPU implementations compared to the banded solver *gbsv()*. The size of the problem is always chosen to be such that it saturates both the CPU and the GPU with enough work so that these architectures can provide the best performance; that is, as the block size is increased the length of the system to be solved is decreased so that the use of the available memory is kept close to maximum. Tables III and IV show the selected length of a system (N) and the number of systems to be solved (P), respectively.

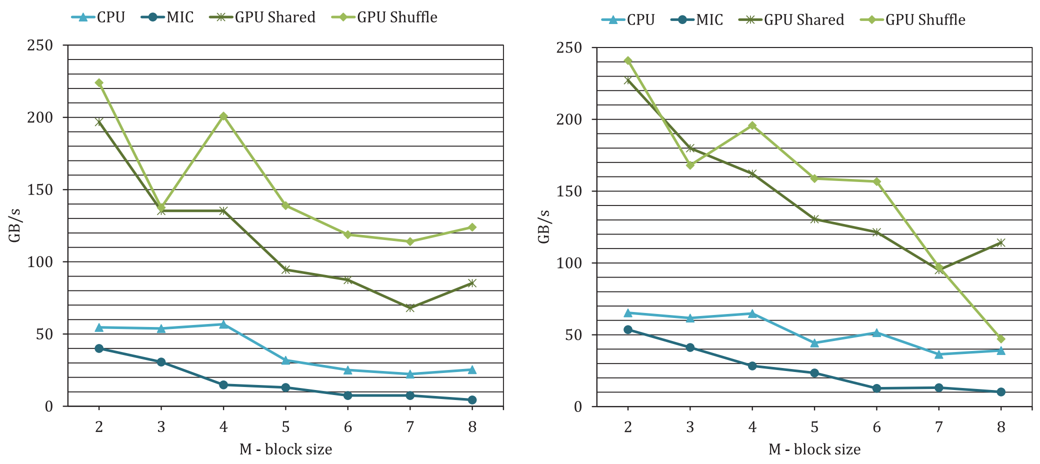


Fig. 18. Single (left) and double (right) precision effective bandwidth when solving block tridiagonal systems with varying M block sizes. Bandwidth is computed based on the amount of data to be transferred. Caching in L1 and registers can make this figure higher than the achievable bandwidth with a bandwidth test.

The Intel Math Kernel Library 11.2 Update 2 library [Intel 2015] was chosen with its *LAPACKE_{s,d}gbsv_work()* function to perform the banded solution. The library function was deploy using OpenMP to achieve the best performance that MKL can provide. According to the MKL manual [Intel 2015], this routine solves for X the linear equations $AX = B$, where $A \in \mathbb{R}^{n\times n}$ band matrix with *kl* subdiagonals and *ku* superdiagonals. The columns of matrix B are individual RHSs, and the columns of X are the corresponding solutions. This function is based on LU decomposition with partial pivoting and row interchanges. The factored form of A is then used to solve the system of equations $AX = B$. The solution of a system is done in two steps. First, a partial solve is done with the upper-triangular U matrix, then a solve with the lower-triangular L matrix is performed. This is an efficient approach when many RHSs exist. In the present case, there is always one RHS, that is, $X \in \mathbb{R}^{n\times 1}$ and $B \in \mathbb{R}^{n\times 1}$. As the systems that are solved are defined with diagonally dominant matrices, pivoting is not performed during execution time. Moreover, the *_work* version of the solver neglects any data validity check, thus provides a fair comparison. The scalar elements of diagonal block matrix arrays A_n^p, B_n^p, and C_n^p are mapped to band matrix A and the scalar elements of diagonal block vector arrays of d_n^p and u_n^p are mapped to X and B accordingly. The performance of the routine is expected to be high as the triangular submatrices are small enough to reside in the L1, L2, or L3 cache. Comparing the solutions of the banded solver with the block tridiagonal solver by taking the differences between the corresponding scalar values shows a Mean Square Error (MSE) in the order of 10^{-4}. The MSE does vary but stays in the order of 10^{-4} as floating-point precision, block size, system size, or the number of systems is changed.

Figures 18 and 19 show the effective bandwidth and computation throughput. The term *effective* is used to emphasize the fact that these performance figures are computed on the basis of the actual data needed to be transferred and actual floating-point arithmetic needed to be performed. Any caching and FMA influencing these figures are therefore implicitly included. In general, the register shuffle-based GPU solver outperforms the shared memory version, with one major exception in double precision with $M = 8$ block size. In this case, the register pressure is too high and registers get spilled into local memory.

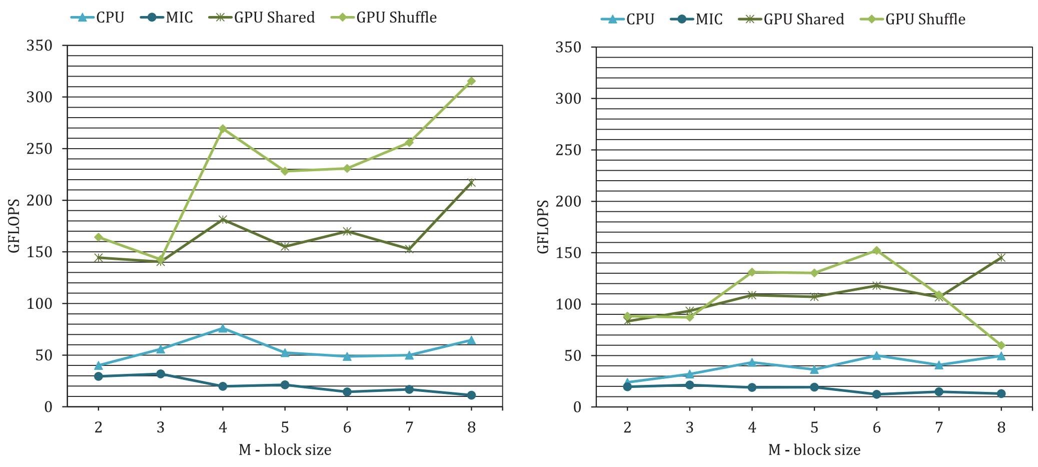


Fig. 19. Single (left) and double (right) precision effective computational throughput when solving block tridiagonal systems with varying M block sizes. GFLOPS is computed based on the amount of floating-point arithmetic operations needed to accomplish the task. Addition, multiplication, and division are considered as floating-point arithmetic operations; that is, FMA is considered as two floating-point instructions.

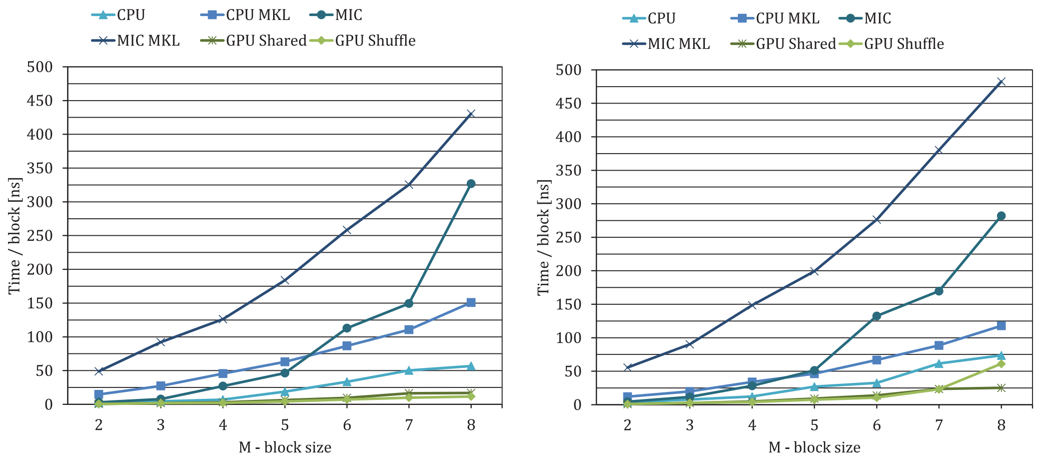


Fig. 20. Single (left) and double (right) precision block-tridiagonal solver execution time per block matrix row for varying block sizes.

One may notice that, as the block size increases, the effective bandwidth decreases in Figure 18 and the effective compute throughput increases at the same time in Figure 19. Therefore, it is implied that the problem is becoming compute limited rather than bandwidth limited, as discussed in Section 2, due to the increasing difference between the $O(M^3)$ compute and $O(M^2)$ memory complexity of a single block.

Execution time per block row is shown in Figure 20. The relative execution time measures efficiency, which is independent of problem size. The total execution time of the solver is divided by NP, where N is the length of a system and P is the number of systems that are solved. Execution time drastically increases for $M = 8$ in the double-precision shuffle version. This is due to register spilling and local memory allocation; that is, data can no longer fit into registers, therefore it is put into local memory. Also, the small local arrays that are supposed to be allocated in registers due to compiler

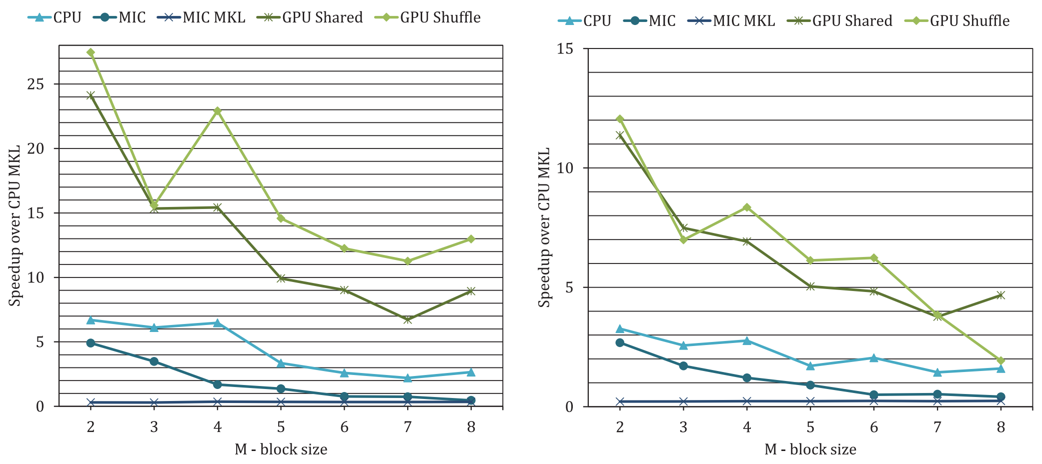


Fig. 21. Single (left) and double (right) precision block tridiagonal solver speedup over the CPU MKL banded solver.

considerations are put into local memory as well. This shows that the presented GPU approaches have very high efficiency for $M = 2, \ldots, 8$ block sizes.

An NVIDIA Tesla C2070 GPU card has been used to compare the results against the PCR solver presented in Stone et al. [2011], in which an NVIDIA Tesla C2050 was used. The two cards contain identical GPUs with a different amount of global memory (see NVIDIA [2010]). 3GB is available on the C2050 and 6GB is available on the C2070. The execution times were measured for the single-precision case on block size $M = 4$, on $P = 512$ number of problems, each of which is $N = 128$ long. The block PCR-based solver completes in 10.5ms and the (shared memory–based) block Thomas solver completes in 2.42ms. This is a $\times 4.3$ speed difference for the sake of the Thomas solver. $\times 8.3$ and $\times 9.8$ improvement is achieved if the execution time per block metrics are compared and the number of systems to be solved is increased to $P = 4096$ or $P = 32768$.

The SIMD solution presented in this article performs well on the CPU, but performs poorly on the MIC architecture. On both architectures, the compute-intensive loops were vectorized as reported by the Intel compiler. Both the MKL banded solver and the presented block tridiagonal solver run more efficiently on the CPU.

Figure 21 presents the speedup of the block-tridiagonal solvers on a GPU over the MKL banded solvers and the CPU- and MIC-based block tridiagonal solvers. This proves the benefit of the presented GPU-based solutions. Also, the highly efficient CPU and MIC implementations show the benefit of using a block tridiagonal solver over a banded solver for the range of block sizes $M = 2, \ldots, 8$.

Significant speedup against the CPU MKL banded solver is reached with the GPU-based solver, up to $\times 27$ in single and $\times 12$ in double precision. The multithreaded CPU code provides $\times 2 - 6$ speedup in single and $\times 1.5 - 3$ speedup in double precision. The multithreaded MIC performance of the block solver is better than the CPU MKL, but the MKL banded solver perform poorly on the MIC. The efficiency of the CPU relies on the cache and out-of-order execution performance of the architecture. The size of the temporary blocks and arrays of blocks is small enough to fit into the L1 and L2 cache. The out-of-order execution is capable of hiding some of the 4 clock cycle latency of accessing these temporary data structures in the L1 cache. As the MIC lacks the out-of-order execution and the cache size per thread is much smaller, the performance is also worse than the CPU. Moreover, the *gbsv()* banded solver of the MKL library does not perform well on the MIC architecture, as it shows 3 times lower performance than the CPU MKL version.

The advantage of doing block tridiagonal solve in the range of $M = 2, \ldots, 8$ instead of a banded solve is obvious. It is important to note that, as the block size M increases, the computations involved in performing block-matrix operations make the problem compute limited instead of memory bandwidth limited. The total execution time of computing a system is composed of loading the blocks of data and performing matrix linear algebra on the blocks. The former scales as $O(NM^2)$ and the latter as $O(NM^3)$, where N is the system size and M is the block size. As the block size increases, the computational part becomes more dominant in the execution time and the memory access time becomes less significant. This can be read from Figures 18 and 19: as M increases, the bandwidth decreases and the GFLOPS increases.

7. CONCLUSION FOR BLOCK TRIDIAGONAL SOLVERS

In this article, it has been shown that solving block-tridiagonal systems with $M = 2, \ldots, 8$ block sizes with the Thomas algorithm pays off over the Intel MKL banded solver. It has been shown that the advantage of the block Thomas algorithm is the computational complexity over the CR or PCR algorithm, and that parallelism can be increased by exploiting the properties of the block matrix operations. The superior performance of the GPU relies on the low arithmetic complexity of the Thomas algorithm and the efficiency of the parallel block matrix operations allowed by work sharing and the register-blocking capabilities of the GPU threads. Since the work complexity (i.e, number of block-matrix operations) of the CR/PCR algorithms is significantly higher than the Thomas algorithm, the CR/PCR has no advantage in block-tridiagonal solvers. Significant speedup is reached with the GPU-based solver with up to $\times 27$ in single and $\times 12$ in double precision. The multithreaded CPU code provides $\times 2-6$ speedup in single and $\times 1.5 - 3$ times speedup in double precision against the MKL banded solver.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

The authors would like to acknowledge the use of the University of Oxford Advanced Research Computing (ARC) facility in carrying out this work. http://dx.doi.org/10.5281/zenodo.22558. The authors also acknowledge the Oxford University CUDA Center of Excellence (CCOE) for providing GPU resources. We are grateful to James Whittle and Catherine Hastings for their initial work on the block-tridiagonal solver, and thankful for the support of Péter Szolgay at PPCU-FIT, Hungary.

REFERENCES

AMD. 2013. AMD64 Architecture Programmer's Manual Volumes 1-5. Retrieved May 17, 2006 from http://developer.amd.com/resources/documentation-articles/developer-guides-m anuals/#manuals.

Stefan Bondeli. 1991. Divide and conquer: A parallel algorithm for the solution of a tridiagonal linear system of equations. *Parallel Computing* 17, 45, 419–434. DOI:http://dx.doi.org/10.1016/S0167-8191(05)80145-0

Li-Wen Chang, John A. Stratton, Hee-Seok Kim, and Wen-Mei W. Hwu. 2012. A scalable, numerically stable, high-performance tridiagonal solver using GPUs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*. IEEE Computer Society Press, Los Alamitos, CA, Article 27, 11 pages. http://dl.acm.org/citation.cfm?id=2388996.2389033.

I. J. D. Craig and A. D. Sneyd. 1988. An alternating-direction implicit scheme for parabolic equations with mixed derivatives. *Computers and Mathematics with Applications* 16, 4, 341–350. DOI:http://dx.doi.org/10.1016/0898-1221(88)90150-2

Duy M. Dang, Christina Christara, and Kenneth R. Jackson. 2010. Parallel implementation on GPUs of ADI finite difference methods for parabolic PDEs with applications in finance. *Social Science Research Network Working Paper Series.* http://ssrn.com/abstract=1580057.

Craig C. Douglas, Sachit Malhotra, and Martin H. Schultz. 1998. Parallel Multigrid with ADI-like Smoothers in Two Dimensions. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.36.3502&rep=rep1&type=pdf.

J. Douglas and H. H. Rachford. 1956. On the numerical solution of heat conduction problems in two and three space variables. *Transactions of the American Mathematical Society* 82, 421–489.

Jim Douglas, Jr. and James E. Gunn. 1964. A general formulation of alternating direction methods. *Numerische Mathematik* 6, 1, 428–453. DOI:http://dx.doi.org/10.1007/BF01386093

B. Dring, M. Fourni, and A. Rigal. 2014. High-order ADI schemes for convection-diffusion equations with mixed derivative terms. In *Spectral and High Order Methods for Partial Differential Equations (ICOSAHOM'12)*, Mejdi Azaez, Henda El Fekih, and Jan S. Hesthaven (Eds.). Lecture Notes in Computational Science and Engineering, Vol. 95. Springer International Publishing, 217–226. DOI:http://dx.doi.org/10.1007/978-3-319-01601-6_17

Walter Gander and Gene H. Golub. 1997. Cyclic reduction - history and applications. In *Proceedings of the Workshop on Scientific Computing*.

S. P. Hirshman, K. S. Perumalla, V. E. Lynch, and R. Sanchez. 2010. BCYCLIC: A parallel block tridiagonal matrix cyclic solver. *Journal of Computational Physics* 229, 18, 6392–6404. DOI:http://dx.doi.org/10.1016/j.jcp.2010.04.049

Intel. 2012a. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel. Retrieved May 17, 2006 from http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf.

Intel. 2012b. *Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual*. Intel. Retrieved May 17, 2006 from https://software.intel.com/sites/default/files/forum/278102/327364001en.pdf.

Intel. 2015. *Math Kernel Library*. Retrieved May 17, 2006 from http://software.intel.com/en-us/articles/intel-mkl/.

Samir Karaa and Jun Zhang. 2004. High order {ADI} method for solving unsteady convection diffusion problems. *Journal of Computational Physics* 198, 1, 1–9. DOI:http://dx.doi.org/10.1016/j.jcp.2004.01.002

Nathan Mattor, Timothy J. Williams, and Dennis W. Hewett. 1995. Algorithm for solving tridiagonal matrix problems in parallel. *Parallel Computing* 21, 11, 1769–1782. DOI:http://dx.doi.org/10.1016/0167-8191(95)00033-0

NVIDIA. 2010. *TESLA C2050 / C2070 GPU Computing Processor*. Retrieved May 17, 2006 from http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf.

NVIDIA. 2015a. *CUDA C Programming Guide*. Retrieved May 17, 2006 from http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3aTPUq4Jo.

NVIDIA 2015b. *CUSPARSE LIBRARY v7.0*. Retrieved May 17, 2006 from http://docs.nvidia.com/cuda/cusparse/index.html#axzz3aTPUq4Jo.

D. W. Peaceman and H. H. Rachford, Jr. 1955. The numerical solution of parabolic and elliptic differential equations. *Journal of the Society for Industrial and Applied Mathematics.* 3, 1, 28–41. http://www.jstor.org/stable/2098834

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 2007. *Numerical Recipes: The Art of Scientific Computing* (3rd ed.). Cambridge University Press, New York, NY.

Thomas H. Pulliam. 1986. Implicit solution methods in computational fluid dynamics. *Applied Numerical Mathematics* 2, 6, 441–474. DOI:http://dx.doi.org/10.1016/0168-9274(86)90002-4

I. Reguly, E. Laszlo, G. Mudalige, and M. Giles. 2016. Vectorizing unstructured mesh computations for many-core architectures. *Concurrency and Computation: Practice and Experience* 28, 2, 557–577. http://dx.doi.org/10.1002/cpe.3621.

Subhash Saini, Johnny Chang, and Haoqiang Jin. 2015. *Performance Evaluation of the Intel Sandy Bridge Based NASA Pleiades Using Scientific and Engineering Applications*. Technical Report. NASA Advanced Supercomputing Division, NASA Ames Research Center, Moffett Field, CA. https://www.nas.nasa.gov/assets/pdf/papers/NAS_Technical_Report_NAS-2015-05.pdf.

Nikolai Sakharnykh. 2009. Tridiagonal solvers on the GPU and applications to fluid simulation. Presented at the GPU Technology Conference, San Jose, CA. Retrieved May 17, 2006 from http://www.nvidia.com/content/gtc/documents/1058_gtc09.pdf.

Nikolai Sakharnykh. 2010. Efficient tridiagonal solvers for ADI methods and fluid simulation. Presented at the GPU Technology Conference, San Jose, CA. Retrieved May 17, 2006 from http://on-demand.gputechconf.com/gtc/2010/presentations/S12015-Tridiagonal-Solvers-ADI-Methods-Fluid-Simulation.pdf.

Sudip K. Seal, Kalyan S. Perumalla, and Steven P. Hirshman. 2013. Revisiting parallel cyclic reduction and parallel prefix-based algorithms for block tridiagonal systems of equations. *Journal of Parallel and Distributed Computing* 73, 2, 273–280. DOI:http://dx.doi.org/10.1016/j.jpdc.2012.10.003

G. Spaletta and D. J. Evans. 1993. The parallel recursive decoupling algorithm for solving tridiagonal linear systems. *Parallel Computing* 19, 5, 563–576. DOI:http://dx.doi.org/10.1016/0167-8191(93)90006-7

Christopher P. Stone, Earl P. N. Duque, Yao Zhang, David Car, John D. Owens, and Roger L. Davis. 2011. GPGPU parallel algorithms for structured-grid CFD codes. In *Proceedings of the 20th AIAA Computational Fluid Dynamics Conference*, Vol. 3221.

Harold S. Stone. 1973. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *Journal of the ACM* 20, 1, 27–38. DOI:http://dx.doi.org/10.1145/321738.321741

L. H. Thomas. 1949. *Elliptic Problems in Linear Differential Equations Over a Network*. Technical Report. Columbia University, New York, NY.

Henk A. van der Vorst. 1987. Large tridiagonal and block tridiagonal linear systems on vector and parallel computers. *Parallel Computing* 5, 12, 45–54. DOI:http://dx.doi.org/10.1016/0167-8191(87)90005-6 Proceedings of the International Conference on Vector and Parallel Computing-Issues in Applied Research and Development.

H. H. Wang. 1981. A parallel method for tridiagonal equations. *ACM Transactions on Mathematical Software* 7, 2, 170–183. DOI:http://dx.doi.org/10.1145/355945.355947

H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. 2010. Demystifying GPU microarchitecture through microbenchmarking. In *IEEE International Symposium on Performance Analysis of Systems Software (ISPASS'10)*. 235–246. DOI:http://dx.doi.org/10.1109/ISPASS.2010.5452013

Yao Zhang, Jonathan Cohen, and John D. Owens. 2010. Fast tridiagonal solvers on the GPU. *SIGPLAN Notices* 45, 5, 127–136. DOI:http://dx.doi.org/10.1145/1837853.1693472

Received October 2014; revised May 2015; accepted September 2015