SIMD Implementation of a Multiplicative Schwarz Smoother for a Multigrid Poisson Solver on an Intel Xeon Phi Coprocessor

Masatoshi Kawai[1,2], Takeshi Iwashita[3,4]
and Hiroshi Nakashima[3]

[1] Graduate School of Informatics, Kyoto University, Japan
[2] JSPS Research Fellow
[3] ACCMS Kyoto University, Japan
[4] JST CREST

Abstract. In this paper, we discuss an efficient implementation of the three-dimensional multigrid Poisson solver on a many-core coprocessor, Intel Xeon Phi. We have used the modified block red-black (mBRB) Gauss-Seidel (GS) smoother to achieve sufficient degree of parallelism and high cache hit ratio. We have vectorized (SIMDized) the GS steps in the smoother by introducing a partially SIMDizing technique based on loop splitting. Our numerical tests demonstrate that our implementation performs 35.5% better than the conventional mBRB-GS smoother implementation on Xeon Phi.

1 Introduction

Discrete Poisson equation problems often appear in various computational science simulations. When the problem is associated with spatially varying diffusion coefficients, it is commonly solved using the finite difference method. The finite difference discretization results in a linear system of equations, that can require a large amount of computational effort, especially for a large-scale simulation problem. Consequently, there is a demand for a fast linear solver for the discrete Poisson equation problem.

A (geometric) multigrid[2] solver is one of the most popular linear iterative solvers for a linear system derived from the finite difference discretization of the Poisson equation. It has a convergence property suitable for large-scale problems. The multigrid solver's convergence rate is independent from the problem size when it is applied to the linear system derived from the homogeneous discrete Poisson equation. Consequently, we have developed a fast geometric multigrid Poisson solver. In this paper, we have investigated the performance of our solver on an Intel Xeon Phi coprocessor[7], which is a recently developed processor.

The Intel Xeon Phi coprocessor is based on Intel MIC architecture, and includes many relatively lower performance cores in its package. The current version of the processor, which we have used in our research, consists of 60 cores. Its peak performance reaches 1TFlops (DP). Moreover, the Xeon Phi coprocessor

has a good performance per watt ratio[3], and its programming model is easier than that for GPU. Various useful applications and tools developed for general purpose multi-core processors (such as MPI and OpenMP) can be used, because each processing core in Xeon Phi is based on X86 architecture. Xeon Phi is currently used as the accelerator for the host CPU[4], but standalone CPU models will be developed in the future. Because of these features, it is predicted that Xeon Phi will play an important role in future computational science.

To efficiently implement a multigrid Poisson solver on the Xeon Phi, we should consider the following key issues.

1. Large degree of thread parallelism: Xeon Phi has larger numbers of cores than a general multi-core processor, and it can simultaneously execute 240 threads using Intel Hyper-Threading technology.
2. Data locality: The processor uses a general cache based memory architecture. Therefore, data locality is important for attaining a high cache hit ratio.
3. Convergence rate: Similarly to implementations on other processors, the convergence rate of the solver has a significant effect on its performance.
4. Vectorization ($SingleInstructionMultipleData$ (SIMD) instructions): Xeon Phi has a relatively wide SIMD engine. Therefore, SIMD instructions should be effectively used in the analysis to let the processor achieve its full potential.

In this paper, we mainly discuss the parallel smoother in the multigrid solver, paying special attention to these key issues. The other components of the multigrid solver can be straightforwardly parallelized and vectorized using a domain decomposition approach.

In VECPAR 2012, we reported a parallel smoother called the modified block red-black Gauss-Seidel (mBRB-GS) smoother[6]. It is a multiplicative Schwarz smoother. The Schwarz smoother is parallelized by applying red-black ordering to cuboid blocks of the problem domain[5], and multiple Gauss-Seidel (GS) iterations are performed in each red or black block. Because the second or later GS iterations in the block are performed on-cache, high data locality is achieved in the smoothing step. Moreover, analytical investigation and numerical tests showed that the smoother attains a sufficient degree of thread parallelism and fast convergence. Accordingly, the mBRB-GS smoother has desirable characteristics in three out of the four key issues mentioned above, and it can be regarded as a candidate for a parallel smoother for the Xeon Phi coprocessor. However, the innermost loop of the smoother consists of sequential GS steps, and it cannot be straightforwardly vectorized.

Our solution to this problem is a partial SIMDization (vectorization) of the GS loop which we split into six simpler loops. Five of the loops are made do-all and thus SIMDizable. The loop-splitting itself is a classic technique for vector processors[1]. However, our revisit has various new aspects such as its application to the SIMD mechanism and the cache-awareness that is essential for scalar many-core processors. We conducted numerical tests on the Xeon Phi coprocessor to compare the effectiveness of the developed solver with the solver based on the conventional gridpoint-wise red-black GS smoother that is naturally vectorized.

2 Parallelized Multigrid Solver for the Three-Dimensional Poisson Equation

2.1 Poisson Equation Problem and Multigrid Solver

We used a 7-point finite difference scheme to solve the three-dimensional Poisson equation. This leads to the following linear system of equations.

$$\boldsymbol{A}\boldsymbol{\phi} = \boldsymbol{\rho}, \tag{1}$$

where $\boldsymbol{\rho}$ is the discretized given source, $\boldsymbol{\phi}$ is the unknown vector, and $\boldsymbol{A}$ is the coefficient matrix. The row of the linear system of (1) corresponding to the grid-point (i, j, k) is written as

$$\begin{aligned} a_{i,j,k} * \phi_{i,j,k-1} + b_{i,j,k} * \phi_{i,j-1,k} + c_{i,j,k} * \phi_{i-1,j,k} + \phi_{i,j,k} \\ + e_{i,j,k} * \phi_{i+1,j,k} + f_{i,j,k} * \phi_{i,j+1,k} + g_{i,j,k} * \phi_{i,j,k+1} = \rho_{i,j,k} \end{aligned} \tag{2}$$

where (i, j, k) represents the grid coordinates. We use the geometric multigrid method to solve the linear system.

The multigrid method consists of the smoother, residual calculator, restriction and prolongation operators and coarsest grid solver. Among these components, we have focused our analysis on the smoother. When we consider the parallelization of the multigrid solver, the residual calculation, the restriction, and the prolongation can be straightforwardly parallelized using domain decomposition. However, it is difficult to parallelize some smoothers. For example, the GS smoother cannot be naturally parallelized. Moreover, a smoother has a significant impact on the performance of the multigrid solver. It greatly affects the convergence of the solver, and its total computational effort is larger than the other components. Consequently, this paper mainly discusses the smoother and its vectorization for the many-core processor.

2.2 Modified Block Red-Black Gauss-Seidel Smoother

In our analysis, we have used the mBRB-GS smoother, which is a multiplicative Schwarz smoother. In this smoother, the entire grid is decomposed into subdomains based on block red-black (BRB) ordering. The entire grid is divided into multiple blocks, and then the red-black ordering is applied to the blocks. Each red or black block is treated as a subdomain in the Schwarz smoother. Multiple sequential GS steps are performed in each subdomain (red/black block), which is smaller than the cache size. Consequently, the second and subsequent GS steps in each subdomain are executed on-cache, which results in high cache hit ratio (good data locality). The degree of parallelism of the smoother is given by the number of blocks of each color. In general, the size of the entire grid is larger than the cache size, and the degree of parallelism is expected to be sufficiently large. Moreover, it was reported in [6] that the multigrid solver using mBRB-GS converges more quickly than when using hybrid Jacobi and GS, or red-black GS smoothers. Consequently, the mBRB-GS smoother is considered to be a promising parallel smoother candidate for the multigrid solver on the Intel Xeon Phi coprocessor.

3 Efficient Implementation of the GS Smoother on Xeon Phi

To develop an efficient multigrid Poisson solver on the Xeon Phi coprocessor, we should consider the vectorization (SIMDization) of the smoother in addition to the parallelization. This is because of the relatively wide SIMD engine of the coprocessor compared with general multi-core processors. However, the mBRB-GS smoother uses GS iterations, which cannot be naturally vectorized. We now introduce an implementation method that makes a compiler generate a partially SIMDized binary code.

Alg. 1 shows the ordinary Fortran implementation of the GS method. When it is used in the mBRB-GS smoother, NX, NY and NZ correspond to the block sizes along the x, y and z axes, respectively.

In the program code, the innermost loop has a loop carried dependence caused by the term $c(i,j,k) * phi(i-1,j,k)$ (highlighted in red), which usually prevents the compiler from SIMDizing the loop.

Alg 1. Ordinary implementation of the GS method

```
1  do k = 1, NZ
     do j = 1, NY
       do i = 1, NX !This loop is not SIMDized
         phi(i,j,k)= rho(i,j,k)                                          &
                   + a(i,j,k)*phi(i,j,k-1) + b(i,j,k)*phi(i,j-1,k) &
6                  + c(i,j,k)*phi(i-1,j,k) + e(i,j,k)*phi(i+1,j,k) &
                   + f(i,j,k)*phi(i,j+1,k) + g(i,j,k)*phi(i,j,k+1) )
       enddo
     enddo
   enddo
```

Although this dependence is essential to the GS method, it does not necessarily inhibit the SIMDization of the whole loop. In fact, as shown in Alg. 2, if we apply loop-splitting so that we have six separated loops for each of the additive terms referencing *phi*, five loops out of the six are free from the loop carried dependence and thus easily and well SIMDized. Note that the loop-splitting does not always improve the loop performance because of additional operations, which in this case are load/store operations of the scratchpad array *tmp*. However, the negative impact is expected to be minor and the following positive effects are also possible.

First, the cost of the additional accesses to *tmp* is minimized by tuning the size NX so that *tmp* is always resident in the first level cache. For the mBRB-GS smoother that has an inherent blocking feature, this tuning is almost automatic and does not require further cache-aware blocking. Second, each of the five dependence-free loops is so simple that all compilers can easily grasp the structure of the loop body, so it is strongly expected that the loop body is efficiently SIMDized, even with the restricted SIMD architecture of Xeon Phi. That is, the relatively small number of streams (four for the first and three

for others) makes it feasible for the compilers to exploit SIMD load/store and alignment instructions, while the common right-hand side structure of $x + y * z$ perfectly fits to the fused multiply-add instructions that are the other source of the high peak performance of Xeon Phi.

Alg 2. Partially SIMDized implementation of the GS method

```
do k = 1, NZ
  do j = 1, NY
    !$DEC SIMD
    do i = 1, NX
      tmp(i) = rho(i,j,k) + a(i,j,k)*phi(i,j,k-1)
    enddo
    !$DEC SIMD
    do i = 1, NX
      tmp(i) = tmp(i) + b(i,j,k)*phi(i,j-1,k)
    enddo

!There are SIMDized phi(i+1,j,k), phi(i,j+1,k) and phi(i,j,k+1) loop.

    do i = 1, NX !This loop is not SIMDized
      phi(i,j,k) = tmp(i) + c(i,j,k)*phi(i-1,j,k)
    enddo
  enddo
enddo
```

4 Numerical Tests

4.1 Numerical Test Conditions

We examined the performance of the multigrid Poisson solver with mBRB-GS using an implementation method for partial SIMDization on an Intel Xeon Phi 7120 coprocessor. The fundamental specifications are listed in Table 1. On the coprocessor, up to 240 threads can run on 60 cores. The program code was written in Fortran compiled by Intel Composer 14.0.0 with the options -O3 -openmp -mmic -no-opt-prefetch. It was run on Xeon Phi using its native execution mode. The test problem had 512^3 grid points. The multigrid solver has converged when the relative residual norm was less than 10^{-7}. In our numerical tests, we evaluated the performance of the multigrid Poisson solver with red-black GS (RB-GS), mBRB-GS based on the implementation methods in Alg. 1 and 2. The block size of the mBRB-GS smoother $NX \times NY \times NZ$ was $512 \times 2 \times 2$, and there were two GS iterations for each block in a smoothing step.

4.2 Performance Evaluation of Proposed Implementation Method

Fig.1 shows the relative speedup of calculation time of the entire mulgitrid solver compared with the sequential mBRB-GS (Alg. 1). These results confirm that

Table 1. Specifications of Xeon Phi

Processor	Model	7120(KnightsCorner)
	Number of cores	61
	Clock frequency	1.24GHz
	L1D-cache size	32KByte/core
	L2-cache size	512KByte/core
Memory	Technology	GDDR5
	Size	16GByte

mBRB-GS outperforms RB-GS on numerical tests conducted on the Xeon Phi coprocessor. In our analysis, RB-GS is implemented using stride memory access. This is one of the most popular implementation methods for RB-GS, because it does not require the array to be reordered for the unknowns. This is convenient for other multigrid components such as the restriction. However, stride memory access is not advantageous in terms of the cache hit ratio when compared with the contiguous method. Consequently, RB-GS is inferior to mBRB-GS in terms of performance, although it can be naturally vectorized. This result is similar to the numerical results on a general multi-core processor [6].

Next, we compared the two implementations of mBRB-GS. Although the partially SIMDized GS implementation (Alg. 2) outperforms the ordinary method (Alg. 1) on the numerical test using 240 threads, it is inferior to the ordinary implementation when using 120 or less threads. Using the Intel Vtune Amplifier, we found that Alg. 2 suffers from two types of processor stalls. One is the VPU_STALL_REG event detected when a read-after-write hazard stalls the SIMD instruction pipeline. The other is the PIPELINE_AGI_STALL that corresponds to the stall of a load/store instruction. It is caused by the latency of the corresponding instruction to provide (a source of) the address to be accessed. We consider that the number of stalls increases because of the reduced number of instructions involved in a loop in Alg. 2. The hyper-threading technology reduces the impact of these stalls on the performance, by interleaving instructions from multiple threads to hide the latency between the pair of instructions. Thus, Alg. 2 performs 19.1% better than Alg. 1 in the case of 240 threads.

The other important observation obtained from our analysis of Alg. 2 using the Vtune Amplifier is that the ratio of SIMD instructions to all executed instructions is only 25.9%. This is much smaller than we expected for the five simple SIMDizable loops. We examined the object code for the loops and found that a significantly large portion of instructions in the loop body is occupied by calculations of the addresses of each element of the three or four arrays referred to in each loop. That is, Intel Composer generates fairly redundant codes for address calculations of three-dimensional array elements. This potential inefficiency might be hidden under the powerful out-of-order superscalar mechanism of ordinary Xeons with multiple integer units which simultaneously works together with SIMD floating-point units. However, this inefficiency is revealed when it is executed on Xeon Phi, because of its in-order two-way superscalar mechanism. This means that only one integer instruction stream can be processed when it

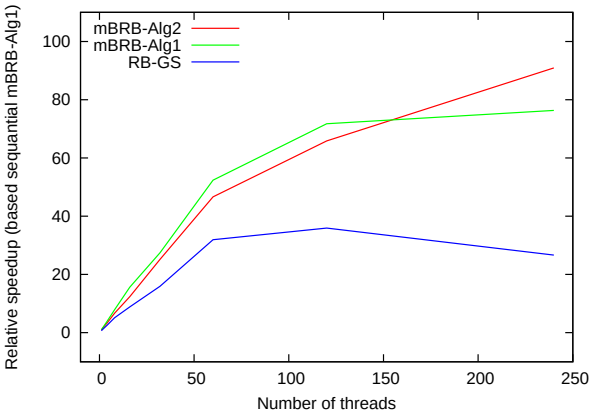

Fig. 1. Comparison of RB and two implementations of the mBRB-GS smoother

has instructions tightly dependent on each other and/or the SIMD floating-point unit is in use.

To reduce the overhead of the address calculation, we made the three-dimensional arrays one-dimensional by applying the well-known array flattening technique. Then, most operations are made loop-invariant and are explicitly moved outside the loop body. This handmade optimization significantly reduced the number of non-SIMD instruction executions for address calculations, resulting in a much higher SIMD instruction ratio of 53.8%. Then, the improvement of the SIMD instruction ratio directly effected the higher performance of three mBRB-GS implementations, as shown in Fig.3. As the figure clearly shows, Alg. 2 with array flattening improves the performance of the 240-thread case and outperforms Alg. 1 by 35.5%. It also improves performance when using 120 threads or less, and is the best of the three implementations in all cases.

Finally, we briefly compared the performances of Xeon Phi and an ordinary multi-core HPC server node of dual Xeon E5-2670 SandyBridge processors. We measured the server node performance using Alg. 1 and Alg. 2 and found that the differences between them are insignificant. This is most likely because of the narrower 256-bit wide SIMD mechanism and the powerful out-of-order superscalar mechanism. On the other hand, an important observation is that the single Xeon Phi coprocessor outperforms the dual-Xeon server using 16 threads by 34.1%. This demonstrates its high potential, even for hardly-SIMDizable kernels. It also shows the importance of architecture-aware code tuning, which we expect to be incorporated into automated optimizations of future compilers for many-core processors with wider SIMD mechanisms.

5 Conclusion

In this paper, we discussed an efficient three dimensional multigrid Poisson solver, working on an Intel Xeon Phi coprocessor. To effectively use the SIMD instructions of Xeon Phi, we introduced the partially SIMDized method for the GS iterations in the multigrid solver, using the mBRB-GS smoother. In our

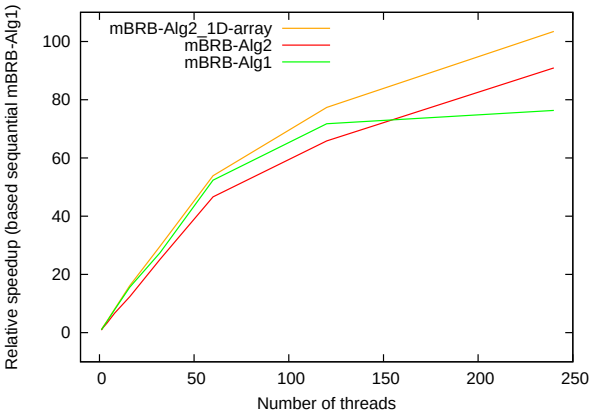

Fig. 2. Comparison of three implementations of the mBRB-GS smoother for the multigrid Poisson solver

implementation, the innermost loop is split into six loops, each of which corresponds to one additive term in a 7-point finite difference scheme. Because five of these loops are free from loop carried dependence, they can be SIMDized. The loop-splitting itself is a classic technique for vector processors. However, our revisit has various new aspects such as its application to the SIMD mechanism and the cache-awareness that is essential for scalar many-core processors. Moreover, using detailed performance profiling and analysis, we found that the reduction of address calculations in the loop body by using array flattening significantly improved performance. Overall, the partially SIMDized implementation attained a 35.5% better performance than the conventional implementation of the mBRB-GS smoother in the 240-thread execution on Xeon Phi.

References

1. Allen, R., Kennedy, K.: Automatic translation of fortran programs to vector form. ACM Trans. Program. Lang. Syst. 9(4), 491–542 (Oct 1987)
2. Briggs, W., Henson, V., McCormick, S.: A Multigrid Tutorial Second Edition. SIAM, Philadelphia , PA (2000)
3. Dokulil, J., et al: High-level support for hybrid parallel execution of c++ applications targeting intel® xeon phi coprocessors. Procedia Computer Science 18, 2508–2511 (2013)
4. Heinecke, A., et al: Design and implementation of the linpack benchmark for single and multi-node systems based on intel® xeon phi coprocessor. In: Parallel & Distributed Processing. pp. 126–137. IEEE (2013)
5. Iwashita, T., Shimasaki, M.: Block red-black ordering: a new ordering strategy for parallelization of ICCG method. Int. J. Parallel Prog. 31, 55–75 (2003)
6. Kawai, M., Iwashita, T., Nakashima, H., Marques, O.: Parallel smoother based on block red-black ordering for multigrid poisson solver. High Performance Computing for Computational Science - VECPAR 2012 (2013)
7. Reinders, J.: An overview of programming for intel® xeon processors and intel® xeon phi coprocessors (2012)