Performance Improvement of Iterative Methods using a Bit-Representation Technique for Coefficient Matrices

Kenji Ono[1,2,3], Shuichi Chiba[4], Shunsuke Inoue[1], and Kazuo Minami[1]

[1] RIKEN, Advanced Institute for Computational Science, 7-1-26, Minatojima-minami-machi, Chuo-ku, Kobe, 650-0047, Japan
[2] Kobe University, Graduate School of System Infomatics
[3] The University of Tokyo, Institute of Industrial Science
[4] Fujitsu Limited
{keno,inoue.shunsuke,minami_kaz}@riken.jp
shuc@jp.fujitsu.com
http://labs.aics.riken.jp/ono.html

Abstract. Practical simulators require high-performance iterative methods and efficient implementations of various boundary conditions, which reflect real-world effects. A nobel *Bit-representation technique* is proposed to enhance the performance of such a code. This technique is applied to the implementation of iterative kernels that treat various boundary conditions. The first advantage of this approach is that it reduces memory traffic and effectively utilizes Single Instruction Multiple Data stream (SIMD) units with cache. Secondly, the proposed implementation can replace if-branch statements with mask operations using a bit expression. This promotes the optimization of code during compilation and run-time. Consequently, the proposed approach achieves 3.5 times faster than a naïve implementation on both *Intel* and *Fujitsu Sparc* architectures.

Keywords: Sparse matrix, Boundary condition, SIMD, Bit operation

1 Introduction

Progress in computational capabilities allows us to increase the scale of problems to obtain more reliable solutions. Computational fluid dynamics simulations, which are commonly used to design industrial products, involve large-scale linear systems with a sparse matrix from the pressure Poisson equation or implicit time integration. Iterative methods for such large-scale sparse matrices are a crucial building block for high-performance physical simulations. Recent computer architectures have been constructed with a deep memory hierarchy, which demands effective utilization of the small but high-speed cache for user programs. If a straightforward implementation of an iterative method is applied to such architectures, the code encounters memory-bandwidth limitations owing to the

large amount of load/store instructions relative to floating-point operations in a loop of the source code.

Williams[1] proposed the *Roofline model*, which provides useful information to analyze the performance of codes. This *Roofline* analysis demonstrates that high operational intensity in an algorithm brings high performance. Thus, one promising approach is to reduce memory traffic from the main memory to cache. To reduce memory traffic, many studies have investigated the Compressed Sparse Row (CSR) data format, run length encoding of CSR[2], and bit-representation schemes[3].

Because engineers and researchers have a strong requirement to predict realistic phenomena and simultaneously reduce the time cost, we must consider various boundary conditions that reflect real-world effects in simulations as well as high-performance computation. However, the calculation of boundary conditions is generally complicated, and the source code cannot be easily optimized owing to if-branch statements, table references, and indirect memory access.

This paper presents an efficient implementation of iterative methods and boundary conditions that employs a *bit-representation technique* to reduce memory traffic. This expresses both the coefficient of a sparse matrix and a mask function that replaces if-branch statements using a bit sequence. The performance of the proposed approach is investigated on several architectures.

2 Basic equations and bit-representation

We consider a pressure Poisson equation derived from the incompressible Navier —Stokes equations.

$$\nabla\,(\nabla p)\;=\;div\,(\frac{\partial \boldsymbol{u}}{\partial t})\;\equiv\,\phi\,, \tag{1}$$

where p, $\boldsymbol{u}$, and ϕ represent pressure, velocity vector, and a source term, respectively. This equation is discretized by finite-difference or finite-volume methods on Cartesian grid system. Hence, the Laplace operator of the equation is approximated by a 7-point stencil, and generates a linear system that has a large-scale sparse matrix. Practical simulators must treat various kinds of boundary conditions in addition to arbitrary positions in computational domain to reproduce various real-world situations. Basically, almost all pressure boundary conditions can be reduced to Dirichlet and Neumann boundary conditions. For instance, we can substitute Neumann boundary conditions for ∇p in eq. (1). Thus, introducing the Heaviside function H, the pressure variable can be written as eq. (3). This expression is a mask function that replaces an if-branch statement in eq. (2), and will promote optimization during compilation.

$$H\;=\begin{cases}0 & (Boundary\,Condition)\\ 1 & (Fluid)\end{cases} \tag{2}$$

$$p\;=\;p\,H+(1-H)\,p^{\,BC} \tag{3}$$

A semi-discrete form of eq. (1) can be combined with eqs. (2) and (3) to give

$$\sum_l \left(\nabla p\, H^N\right)_l \boldsymbol{n}_l = h\,\phi - \sum_l \left(1 - H_l^N\right) \nabla p_l^{BC} \boldsymbol{n}_l \tag{4}$$

where h, l, $\boldsymbol{n}$ and H^N denote the grid width, cell face location, outside normal vector at the cell face, and Heaviside function for the Neumann boundary condition, respectively. We can also introduce a Dirichlet boundary condition into eq. (4) by replacing the pressure gradient term ∇p with a Heaviside function H^D. Eq. (5) represents an example of a Dirichlet condition on the face of cell i as shown in Fig. 1.

$$\left(\nabla p\, H^N\right)_e = \frac{1}{h}\left\{ p_{i+1} H_e^D + \left(1 - H_e^D\right) p_{i+1}^{BC} - p_i \right\} H_e^N \tag{5}$$

Finally, we obtain

$$\begin{aligned} &\sum_l \left(p\, H^D\, H^N\right)_l - p \sum_l H_l^N \\ &= h^2\,\phi - h \sum_l \left(1 - H_l^N\right) \nabla p_l^{BC} \boldsymbol{n}_l - \sum_l \left(1 - H_l^D\right) p_l^{BC} H_l^N \boldsymbol{n}_l \end{aligned} \tag{6}$$

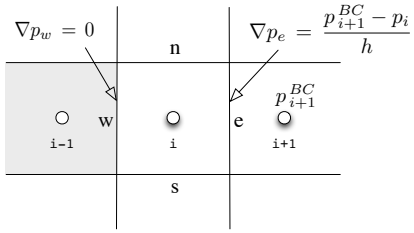


Fig. 1: Neumann and Dirichlet boundary conditions for cell i in two dimensions. A Neumann BC is applied at the west cell face, which is solidly shaded. A Dirichlet BC is employed at the east cell face, where the boundary value is given by the pressure p_{i+1}^{BC}.

The contribution of all boundary conditions is included in the RHS of eq. (6), which is computed at the start of each iteration. Although Heaviside functions only output values of one or zero, to distinguish between normal and boundary cell states, the boundary values can be exactly reproduced in the RHS. In eq. (6), the second and third terms of the RHS can include any values of the Neumann and Dirichlet conditions. It is obvious that the coefficients of non-diagonal component $p\, H_l^D\, H_l^N$ are either 1 or 0, and diagonal component $\sum H_l^N$ ranges from 0 to 6. Therefore, the coefficients of the non-diagonal components can be expressed by 1 bit, and those of the diagonal components can be expressed by 3 bits. That is, 9 bits are sufficient to represent all coefficients in the update of the pressure value of a cell. These coefficients are encoded in a 4-byte integer

array, as shown in Fig. 2, with other functional flags. For instance, `BC_Diag` and `BC_Ndag_x` express the diagonal and non-diagonal elements of the coefficient in the sparse matrix A, where `x` denotes the six outward directions of the cell faces by the notation of `W`, `E`, `S`, `N`, `B`, and `T`. The two types of boundary conditions for each direction are encoded as, e.g., `BC_N_x` and `BC_D_x`. This bit sequence is prepared before the flow calculation. Encoding bits into an array `b` can be performed by a simple shift operation, e.g., `b|=(0x1<<BC_Ndag_S)` for the south direction of a cell. Unlike the approach developed by Tang[3], the proposed method does not require any extra calculations, as we do not use compression.

Here, two implementations of the conventional Red-Black SOR (Successive Over-Relaxation) method are shown. One is the proposed bit-representation (hereafter, *bit-reps*), and the other is a *naïve* code that stores all coefficients in floating-point arrays, where `p, b, bp, pn` denote pressure, the RHS of a derived linear system $A\boldsymbol{x} = \boldsymbol{b}$, a utility array of the mask function for a cell (indicates active or inactive), and the coefficients of matrix A for the *naïve* method, respectively. In the case of *bit-reps*, in particular, the array `bp` includes compressed coefficient information. The loop in both codes contains 27 floating-point arithmetic operations, whereas 9 loads/1 store and 16 loads/1 store can be found in the code for the *bit-reps* and the *naïve* system, respectively. In the *naïve* code, memory access for the variable array `pn` is set to consecutive, so that the SIMD units work effectively.

Implemented Fortran code of Red-Black SOR method

```
[Bit-reps code]                          | [Naive code]
                                         |
do color=0,1                             | do color=0,1
do k=1,kx                                | do k=1,kx
do j=1,jx                                | do j=1,jx
do i=1+mod(k+j+color,2), ix, 2           | do i=1+mod(k+j+color,2), ix, 2
  idx = bp(i,j,k)                        |
  c_e = real(ibits(idx, NDAG_E, 1))      |   c_w = pn(i,j,k,1)
  c_w = real(ibits(idx, NDAG_W, 1))      |   c_e = pn(i,j,k,2)
  c_n = real(ibits(idx, NDAG_N, 1))      |   c_s = pn(i,j,k,3)
  c_s = real(ibits(idx, NDAG_S, 1))      |   c_n = pn(i,j,k,4)
  c_t = real(ibits(idx, NDAG_T, 1))      |   c_b = pn(i,j,k,5)
  c_b = real(ibits(idx, NDAG_B, 1))      |   c_t = pn(i,j,k,6)
  d0 = real(ibits(idx,bc_diag+0, 1))     |   ndg = pn(i,j,k,7)
  d1 = real(ibits(idx,bc_diag+1, 1))     |
  d2 = real(ibits(idx,bc_diag+2, 1))     |
  dd = 1.0 / (d2*4.0 + d1*2.0 + d0)      |   dd = 1.0 / ndg
  pp = p(i,j,k)                          |   pp = p(i,j,k)
  ss = c_e * p(i+1,j  ,k  )              |   ss = c_e * p(i+1,j  ,k  )
     + c_w * p(i-1,j  ,k  )              |      + c_w * p(i-1,j  ,k  )
     + c_n * p(i  ,j+1,k  )              |      + c_n * p(i  ,j+1,k  )
     + c_s * p(i  ,j-1,k  )              |      + c_s * p(i  ,j-1,k  )
     + c_t * p(i  ,j  ,k+1)              |      + c_t * p(i  ,j  ,k+1)
     + c_b * p(i  ,j  ,k-1)              |      + c_b * p(i  ,j  ,k-1)
  dp = (dd*ss + b(i,j,k)-pp) * omg       |   dp = (dd*ss + b(i,j,k)-pp) * omg
  p(i,j,k) = pp + dp                     |   p(i,j,k) = pp + dp
  res = res + dble(dp*dp)                |   res = res + dble(dp*dp)
    * dble(ibits(idx,Active,1))          |   * dble(ibits(bp(i,j,k),Active,1))
end do                                   | end do
```

The above code clearly shows the number of arrays required in both implementations, e.g., ten scalar arrays are required for the *naïve* implementation, whereas *bit-reps* needs three. Table 1 shows the memory footprint of both im-

plementations for the cases evaluated in Section 3. It is obvious that the *bit-reps* implementation demands one-third of the memory capacity needed by the *naïve* implementation. This has a significant impact on the performance of cache-based architectures.

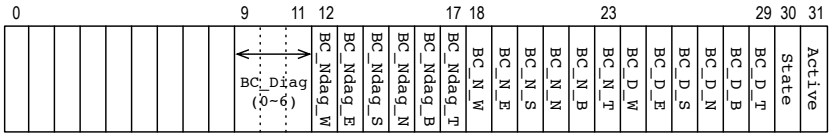


Fig. 2: Bit representation. Several bits required for the bit-representation are encoded into this array. This example includes diagonal(BC_Diag), non-diagonal (BC_Ndag_x), Neumann boundary(BC_N_x), Dirichlet boundary (BC_D_x), cell state (State), and activeness (Active) of a cell. Other bits are used for more complicated processes.

Table 1: Comparison of the memory footprint (MB) required for *naïve* and *bit-reps* implementation. These memory capacity include the size of the halo region.

Problem size (/w halo=2)	16^3	32^3	64^3	128^3	256^3
Naïve	0.3	1.8	12.0	87.7	670.5
Bit-reps	0.1	0.5	3.6	26.3	201.1

3 Evaluation of proposed method and discussion

The performance of *bit-reps* and the *naïve* implementation was examined on *Intel* and *Sparc* architectures (see Table 2). Three-dimensional unsteady cavity flows were computed in single precision to evaluate the performance. The *Performance Monitor library (PMlib)*[4] was employed to measure the GFlops/s attained by each code. The *PMlib* is designed to calculate performance statistics based on the user's declaration of the Flop count and the measured timing between marked sections in the source code. This approach is not perfect, but can offer portability for many platforms, even with no means of accessing the hardware performance counter. Fig. 3 shows the serial measured performance for different problem sizes. In the *Intel* architectures, we can see that the performance of the *naïve* code worsens when the problem size exceeds the cache, i.e., beyond 64^3 in this case. On the other hand, the performance of the *bit-reps* code is maintained. Because the *bit-reps* code has a lower memory requirement, the degradation in performance as the problem size increases seems to be delayed. The performance results on the *Sparc* architecture exhibit different behavior from those on the *Intel*, as the

performance tends to improve as the problem size increases. It was found that the performance of the *bit-reps* code is superior to that of the *naïve* code on *Sparc VIIIfx*. In particular, the performance of *Sparc VIIIfx* is much greater than that of *Sparc IXfx*. The assembler code is responsible for this phenomenon. That is, the compiler on *Sparc VIIIfx* is optimized to decode a shift operation in order to issue the SIMD instructions, but this is not the case for the *Sparc IXfx* compiler.

Table 2: Specification of evaluation machines. TRIAD scores are measured by the STREAM benchmark[5].

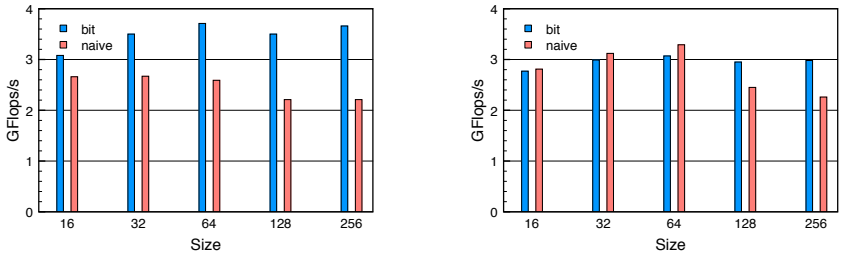
Architecture	Clock (GHz)	Core	CPU	Peak (GFlops/s)	Cache (MB)	Memory (GB)	Theoretical BW (GB/s)	TRIAD (GB/s)
Westmere	2.66	6	2	127.7	12	16	64	22
SandyBridge	2.6	8	2	166.4	20	64	102	59
Sparc VIIIfx	2.0	8		128.0	6	16	64	36
Sparc IXfx	1.85	16		236.5	12	32	85	50

The measured thread performance is shown in Fig. 4. The size of the calculation was chosen as 256^3, so that the data is placed out of cache. Fig. 4 indicates that memory-bound behavior occurs for the *naïve* code with more than four threads (*Intel*) and six threads (*Sparc VIIIfx*), but this is not observed on *Sparc IXfx*. Although the performance of the *naïve* code on *Sparc IXfx* seems to have good scalability, this is because of the unoptimized code described above. In contrast, the *bit-reps* code shows a significant effect from suppressing the memory traffic, and achieves remarkable performance gain compared to the *naïve* implementation in all cases. In particular, the highest performance is mostly obtained with the maximum number of threads owing to the designed bit-compression scheme.

Thus, the *bit-reps* code exhibits higher performance than the *naïve* code for all evaluated architectures, and we have found that the *bit-reps* code significantly improves performance and achieves over 17% of the theoretical peak performance on both *Intel* and *Sparc* architectures. One of the reasons that higher performance is attained by the *Intel* CPUs is that they have SIMD computing units for integer arithmetic in addition to floating-point arithmetic, and the optimized compiler enables both to work well.

We have described how to implement the *bit-reps* code, and demonstrated its effectiveness in a classical iterative method for the Poisson equation derived from the incompressible Navier—Stokes equations. Usually, flow fields are solved by coupling the Poisson equation with other momentum equations, although our *bit-reps* implementation can only be applied to the Poisson equation. Of course, the same idea can be introduced for other equations, e.g., momentum equations, energy equations, and the equation of continuity, and a similar effect will be ob-

served. The proposed *bit representation technique* can also be applied in modern iterative methods like the Krylov subspace method, because our implementation effectively accelerates the computation of the matrix-vector product $A\boldsymbol{x}$. We can apply the *bit-reps* implementation for the calculation of $A\boldsymbol{x}$ in Krylov methods. However, there are some limitations of our method in terms of preconditioners. For instance, the incomplete Cholesky decomposition cannot be employed, because this type of preconditioner changes the coefficients in the matrix. In addition, scaling, which is one of the simplest preconditioning techniques, cannot be used for the same reason. Thus, only Gauss—Seidel smoothers can be applied for preconditioning.



(a) Intel Westmere X5650.

(b) Intel SandyBridge E5-2670.

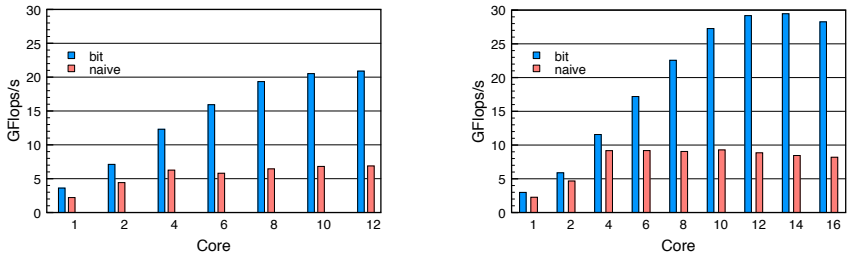
(c) Fujitsu Sparc Venus VIIIfx.

(d) Fujitsu Sparc Venus IXfx.

Fig. 3: Comparison of serial performance of each machine. The problem size varies ranging from 16^3 to 256^3.

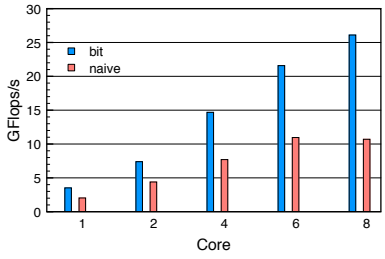
4 Concluding Remarks

We have described a novel and effective implementation of the classical Red-Black SOR iterative method to fully exploit recent cache-based architectures. The developed *bit-reps* method represents the coefficients of a large-scale sparse matrix compactly by a bit sequence. This approach enables the effective reduction of memory traffic, which is directly associated with performance. The performance of the *bit-reps* code was compared to a *naïve* code on several architectures. Consequently, it was found that the performance was improved by a

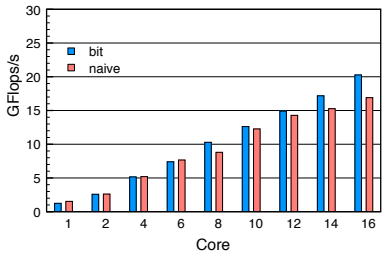


(a) Intel Westmere X5650.

(b) Intel SandyBridge E5-2670.



(c) Fujitsu Sparc Venus VIIIfx.



(d) Fujitsu Sparc Venus IXfx.

Fig. 4: Comparison of thread parallel performance of each machine. The problem size is chosen to 256^3 so that the data resides in main memory.

factor of up to 3.5 times, and 17% of the theoretical peak was achieved on both *Intel* and *Fujitsu Sparc* architectures. In addition to the performance gain, the *bit-reps* approach allows us to handle various arbitrary boundary conditions and arbitrary positions in a computational domain without significant performance degradation. This feature is particularly favorable for practical simulations. Although we exploited the Red-Black SOR method as an iterative solver in this paper, the *bit-reps* technique can be applied to the other modern iterative methods discussed in Section 3.

Acknowledgments. We thank the RIKEN Advanced Institute for Computational Science for allowing us to use the K computer to obtain our results.

References

1. S. Williams, S., Waterman, A. and Patterson, D.: Roofline; An Insightful Visual Performance Model for Multicore Arch. *Commun. ACM*, Vol.**52** No.**4** (2009) 65–76
2. Willcock, J. and Lumsdaine, A.: Accelerating sparse matrix computations via data compression. Proc. 20th Annual ICS '06 (2006) 307–316
3. Tang, W. T., et al.: Accelerating Sparse Matrix-vector Multiplication on GPUs Using Bit-representation-optimized Schemes. Proc. of SC13 **26** (2013) 1–12
4. `https://github.com/avr-aics-riken/PMlib`
5. `http://www.cs.virginia.edu/stream`