GPU Acceleration of Communication Avoiding Chebyshev Basis Conjugate Gradient Solver for Multiphase CFD Simulations

Yussuf Ali, Naoyuki Onodera, Yasuhiro Idomura
Center for Computational Science and e-Systems
Japan Atomic Energy Agency
Kashiwa, Chiba 227-0871, Japan

Takuya Ina, Toshiyuki Imamura
Center for Computational Science
RIKEN
Kobe, Hyogo 650-0047, Japan

***Abstract*—Iterative methods for solving large linear systems are a common part of computational fluid dynamics (CFD) codes. The Preconditioned Conjugate Gradient (P-CG) method is one of the most widely used iterative methods. However, in the P-CG method, global collective communication is a crucial bottleneck especially on accelerated computing platforms. To resolve this issue, communication avoiding (CA) variants of the P-CG method are becoming increasingly important. In this paper, the P-CG and Preconditioned Chebyshev Basis CA CG (P-CBCG) solvers in the multiphase CFD code JUPITER are ported to the latest V100 GPU. All GPU kernels are highly optimized to achieve about 90% of the roofline performance, the block-Jacobi preconditioner is re-designed to benefit from the high computing power of the GPU and the remaining bottleneck of halo data communication is resolved by overlapping communication and computation. The overall performance of the P-CG and P-CBCG solvers is determined by the competition between the CA properties of the global collective communication and the halo data communication, indicating an importance of the inter-node interconnect bandwidth per GPU. The developed GPU solvers are accelerated up to 2x if compared to the CPU solvers on KNLs, and excellent strong scaling is achieved up to 7,680 GPUs on Summit.**

***Index Terms*—Communication avoiding Krylov subspace method, CFD, GPU**

I. Introduction

Computational Fluid Dynamics (CFD) codes are common tools in science and engineering. In nuclear engineering, CFD codes are used in critical issues such as the development of new reactor concepts, severe accident analysis, nuclear safety analysis, and the environmental dynamics of radioactive substances released in nuclear accidents and nuclear terrorism. Iterative methods are common components of these CFD codes. They occupy a significant part of the overall computing cost. Furthermore, on large scale distributed computing platforms, inter-node communication introduces an additional cost to the iterative methods. This is especially true for accelerated computing platforms, where the computing cost is dramatically reduced, and the remaining communication cost becomes relatively important. Global collective communication is one of the most crucial parts in the iterative methods, since all computing nodes need to synchronize. For this reason, global collective communication is commonly seen as a crucial bottleneck in exascale computing.

Krylov Subspace Methods (KSMs) are a widely used class of iterative methods. For example, the Preconditioned Conjugate Gradient (P-CG) method contains two global collective communication method calls per iteration. To overcome this bottleneck, communication avoiding (CA) KSMs were developed[1]. Although they are mathematically equivalent to their ordinary non-CA counterparts, the behavior of CA KSMs inside CFD codes still remains an active field of research, because CA KSMs are more sensitive to numerical errors. In addition, most practical applications become ill-conditioned problems, which require proper preconditioning. Preconditioners in CA KSMs are still a challenging issue on GPUs.

GPUs are promising candidates of exascale computing platforms. A single compute node can contain several GPUs, and the number of GPUs per node tends to increase because of facility requirements, leading to high density packaging. Although intra-node communication can be processed by utilizing high bandwidth interfaces like NVLINK, an increasing number of GPUs per node leads to less inter-node communication bandwidth, and the need for CA KSMs on GPUs is increasing. However, in order to benefit from the high computing power of the GPU, computing kernels need to provide extreme concurrency in addition to correctly handling the complex memory hierarchy of the GPU architecture. This is a challenging issue for CA KSMs, because they contain a wide range of different computing kernels.

In this work, we implemented the P-CG and Preconditioned Chebyshev Basis CA CG (P-CBCG) methods on the V100 GPU. All main computing kernels were implemented using CUDA. The block-Jacobi (BJ) preconditioner was re-designed in order to take optimal advantage of the GPU hardware. We applied both methods to the pressure Poisson solver in the multiphase CFD code JUPITER [2]. The developed solvers were tested in a cross platform comparison on the AI Bridging Cloud Infrastructure (ABCI) [3], Summit [4], and Oakforest-PACS [5] clusters using up to 2,048 GPUs/CPUs. On Summit, strong scaling was further evaluated with up to 7,680 GPUs.

The rest of the paper is organized as follows: Related research is reviewed in Sec.II. In Sec.III, we briefly describe (CA) KSMs used in JUPITER. In Sec.IV, we explain the GPU implementation using CUDA. In Sec.V, we present numerical experiments on ABCI and Summit. Finally, a summary is given in Sec.VI.

II. Related work

The CA-CG method is based on the so-called s-step CG method, in which the data dependency between SpMV and inner product operations in the standard CG method is removed. Van Rosendale [6] first developed a s-step version of the CG method. Chronopoulos and Gear [7] called their own variant of the CG method as the s-step CG method. However, the above works did not change SpMV operations for generating the s-step basis. Joubert and Carey[8] optimized the SpMV operations by reducing communications between levels of the memory hierarchy and between processors, while their version worked only for 2-D regular meshes. Toledo optimized the computation of the s-step basis in the s-step CG method [9]. His s-step basis generation algorithm reduced the number of words transferred between levels of the memory hierarchy. On the other hand, the CACG method by Hoemmen [1] reduced communication between levels of memory and between processors by a matrix power kernel (MPK) [10], in which multiple SpMVs are computed without communication.

CA KSMs showed promising results for a wide range of test problems [11], [12], provided that preconditioners are not needed or a simple algorithm such as the point-Jacobi preconditioner is sufficient. However, in most of the practical issues, more sophisticated preconditioners are needed, and they require additional communication on distributed computing platforms. This issue was addressed by Yamazaki et al. [13]. They proposed a CA preconditioner based on an underlap approach, in which BJ preconditioning is modified by applying point-Jacobi preconditioning in the surface part of each subdomain to avoid additional communication. The underlap approach was tested on the pressure Poisson solver in JUPITER, and it was shown that the underlap approach leads to significant convergence degradation [14]. To avoid such convergence degradation, a hybrid CA approach was proposed. In this approach, CA is only applied to the global collective communication and SpMVs are computed with halo data communication, which allowed overlap of communication and computing kernels.

Another issue in CA KSMs is their numerical stability in ill-conditioned problems. In Ref. [14], it was also shown that for ill-conditioned problems in JUPITER, the P-CACG method [1] is numerically stable only within a few CA steps $s \leq 3$ even with the original BJ preconditioning. This issue is attributed to the monomial basis vectors, which tend to become more and more linearly dependent with an increasing s. To resolve this issue, Suda et al. [15] proposed the P-CBCG method, which was tested with point-Jacobi preconditioning on the K-computer [12]. In Ref. [16], the P-CBCG method with the BJ preconditioner was applied to JUPITER, and the above numerical stability issue was resolved for $s > 10$.

GPU implementations of the CA-GMRES method [1] were discussed in Refs. [13], [17], [18]. In Ref. [17], several tall-skinny QR (TSQR) factorization methods were optimized on GPUs, and the performance of TS matrix operations was improved using a batched GEMM approach. In Ref. [13], MPK with CA preconditioner was implemented on GPUs. Although these works used MPK, comparisons between MPK and the hybrid CA approach in Ref. [18] showed higher performance with the latter. In this work, we present a GPU implementation of the P-CBCG method based on the hybrid CA approach.

III. Krylov solvers in JUPITER

A. Multiphase CFD code JUPITER

JUPITER [2] models materials inside nuclear reactors using the equation of continuity, Navier-Stokes, and energy assuming Newtonian and incompressible viscous fluids. The dynamics of the gas, liquid, and solid phases of several components including the fuel pellets, fuel cladding, the absorber, internal reactor components and the atmosphere are described by an advection equation of the volume of fluid (VOF) function.

The main computational cost ($\sim 90\%$) comes from computation of the pressure Poisson equation, which is discretized by the second order accurate centered finite difference scheme (seven stencils) in the Cartesian grid system (x, y, z). The linear system of the Poisson equation, which is a symmetric block diagonal sparse matrix, is solved using KSMs. Among the wide range of KSMs, in terms of convergence property and computational cost, the best results were obtained using the P-CG method and its CA variants with BJ preconditioning. The Poisson solver is parallelized using a three dimensional (3D) domain decomposition method in (x, y, z).

The CPU version of JUPITER takes advantage of the computational resources by using a hybrid MPI+OpenMP approach. In this work, JUPITER is extended by adding GPU Poisson solvers, which are implemented using CUDA, resulting in a new MPI+OpenMP+CUDA hybrid code.

B. P-CG method

The P-CG method is shown in Algorithm 1. Each iteration consists of SpMV (line 3), the BJ preconditioner (line 7), AXPYs (lines 5,6,9), and inner product operations (lines 4,8). The communication cost per single iteration consists of one halo data exchange for the SpMV and two global collective communication methods calls (Allreduce) for the inner product operations.

C. P-CBCG method

The P-CBCG method [15] is shown in Algorithm 2. The preconditioned Chebyshev basis vectors are generated at line 10, which is further described in Algorithm 3. The basis vector generation in this work does *not* exploit MPK, in which halo data for multiple SpMVs is communicated only once, because

Algorithm 1 Preconditioned Conjugate Gradient (P-CG) method

Input: $Ax = b$, Initial guess x_1
Output: Approximate solution x_i
1: $r_1 := b - Ax_1, z_1 = M^{-1}r_1, p_1 := z_1$
2: **for** $j = 1, 2, ...$ until convergence **do**
3: Compute $w := Ap_j$
4: $\alpha_j := \langle r_j, z_j \rangle / \langle w, p_j \rangle$
5: $x_{j+1} := x_j + \alpha_j p_j$
6: $r_{j+1} := r_j - \alpha_j w$
7: $z_{j+1} := M^{-1} r_{j+1}$
8: $\beta_j := \langle r_{j+1}, z_{j+1} \rangle / \langle r_j, z_j \rangle$
9: $p_{j+1} := z_{j+1} + \beta_j p_j$
10: **end for**

the BJ preconditioner requires additional halo data communication at each iteration. Therefore, SpMV is implemented in a straightforward manner including the BJ preconditioner as in the hybrid CA approach [14]. The basis vectors are generated using Chebyshev polynomials $T_j(AM^{-1})$ with T_j being the j-th Chebyshev polynomial scaled and shifted within the minimum and maximum eigenvalues $[\lambda_{min}, \lambda_{max}]$. Here, λ_{min} is approximated as zero, while λ_{max} is computed using the power method, which typically runs a hundred iterations before the main loop starts. In the monomial basis, with an increasing s the generated basis vectors are aligned to the eigenvector with λ_{max}. If the basis vector is not orthogonalized at each iteration, the other eigen components become relatively small and are hidden by round off errors. This may cause the basis vectors to become linearly dependent, leading to breakdown of KSMs. However, in the Chebyshev basis, the minmax property of the Chebyshev polynomials helps to keep the basis vectors linearly independent. This property makes the Chebyshev basis numerically more robust, and enables us to select a larger s than the monomial basis.

Although each SpMV requires halo data communication as in the P-CG method, global collective communication is reduced to $1/s$. During every s iterations, one global collective communication method call (Allreduce) is needed at line 5, transferring the upper triangular part of $Q_k^* A Q_k$ which counts for $s(s+1)/2$ elements plus s elements from $Q_k^* r_{sk}$ and one element needed for the norm of the residual vector. Another Allreduce is needed at line 11 where the square matrix $Q_k^* A S_{k+1}$ of size s^2 elements is reduced.

IV. Implementation of P-CG and P-CBCG on the GPU

A. Sparse matrix format

The choice of the sparse matrix format is crucial for efficient memory access. This is especially true for GPUs, since threads on the GPU need to access the device memory in a coalesced manner in order to utilize the full device memory bandwidth. Sparse matrix formats like the compressed sparse row (CSR) format do not provide coalesced memory access, because of indirect memory accesses. For this reason, we use the diagonal (DIA) format (sometimes also referred to as the compressed diagonal storage format (CDS)), which provides perfect coalesced memory access. The same format is also used in the CPU version of JUPITER [2], [16]. The DIA format is the natural choice for JUPITER, since the matrix has a block diagonal structure with seven stencils.

Algorithm 2 Preconditioned Chebyshev Basis communication avoiding Conjugate Gradient (P-CBCG) method

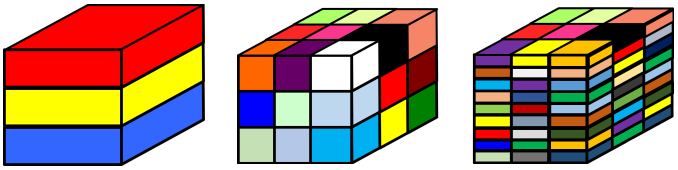
Input: $Ax = b$, Initial guess x_0
Output: Approximate solution x_i
1: $r_0 := b - Ax_0$
2: Compute S_0 $(T_0(AM^{-1})r_0, ..., T_{s-1}(AM^{-1})r_0)$
3: $Q_0 = S_0$
4: **for** $k = 0, 1, 2, ...$ until convergence **do**
5: Compute $Q_k^* A Q_k$
6: Compute $Q_k^* r_{sk}$
7: $a_k := (Q_k^* A Q_k)^{-1} Q_k^* r_{sk}$
8: $x_{s(k+1)} := x_{sk} + Q_k a_k$
9: $r_{s(k+1)} := r_{sk} - A Q_k a_k$
10: S_{k+1} $(T_0(AM^{-1})r_{s(k+1)}, ..., T_{s-1}(AM^{-1})r_{s(k+1)})$
11: Compute $Q_k^* A S_{k+1}$
12: $B_k := (Q_k^* A Q_k)^{-1} Q_k^* A S_{k+1}$
13: $Q_{k+1} := S_{k+1} - Q_k B_k$
14: $AQ_{k+1} := AS_{k+1} + AQ_k B_k$
15: **end for**

Algorithm 3 Preconditioned Chebyshev basis

Input: r_{sk}, Approximate minimum/maximum eigenvalues of AM^{-1}, $\lambda_{\min}, \lambda_{\max}$
Output: S_k $(\tilde{z}_0, \tilde{z}_1, ..., \tilde{z}_{s-1})$, AS_k $(A\tilde{z}_0, A\tilde{z}_1, ..., A\tilde{z}_{s-1})$
1: $\eta := 2/(\lambda_{\max} - \lambda_{\min})$
2: $\zeta := (\lambda_{\max} + \lambda_{\min})/(\lambda_{\max} - \lambda_{\min})$
3: $z_0 := r_{sk}$
4: $\tilde{z}_0 := M^{-1} z_0$
5: $z_1 := \eta A \tilde{z}_0 - \zeta z_0$
6: $\tilde{z}_1 := M^{-1} z_1$
7: **for** $j = 2, 3, ..., s$ **do**
8: $z_j := 2\eta A \tilde{z}_{j-1} - 2\zeta z_{j-1} - z_{j-2}$
9: $\tilde{z}_j := M^{-1} z_j$
10: **end for**

B. Preconditioning

Preconditioning is an essential kernel in the pressure Poisson solver in JUPITER. The CPU solver uses the BJ preconditioning, in which the incomplete LU factorization (ILU) is applied to each block. Porting the CPU version as it is would result in a very inefficient GPU preconditioner. The CPU preconditioner divides the subdomain on each MPI process along the z-axis into smaller blocks. Each block is then processed by a single OpenMP thread, because ILU in each block cannot be parallelized. The 1D block decomposition in the z-axis was chosen because it showed the best convergence property and provided sufficient concurrency for a CPU with several tens of cores. However, the concurrency of this implementation



(a) Division among the z-axis (b) Division among the xyz-axes (c) Division among the zxy-axes

Fig. 1: Block decomposition methods for the block-Jacobi preconditioner. (a) shows the 1D block decomposition for CPUs, (b) shows the 3D block decomposition for GPUs which does not allow coalesce memory access, and (c) shows the 3D block decomposition with finer tiles which allows coalesce memory access.

is not sufficient for GPUs which require a several orders of magnitudes larger number of threads. In addition, the CPU implementation would lead to a non-coalesced memory access pattern with large strides. To resolve these issues, we re-designed the BJ preconditioner for the GPU.

In the GPU preconditioner, a 3D block decomposition along the xyz-axes was introduced, and ILU in each block is processed by a single GPU thread. Table I shows the block size dependency of the GPU preconditioner for a small JUPITER matrix with $N = N_x \times N_y \times N_z = 256 \times 128 \times 512$, which corresponds to a typical problem size per GPU. By scanning different block divisions, an appropriate block shape was found (see Fig.1). The block size on the GPU preconditioner has to be three orders of magnitudes smaller than the CPU preconditioner. The use of finer blocks changes the quality of the BJ preconditioning, leading to a significant convergence degradation. The block size scan with the original xyz data structure (t_1) shows the minimum cost when the block size in the inner most x direction is one or two and the GPU can use coalesced memory accesses for all threads inside a warp. On the other hand, it is also shown that in terms of the number of iterations, the best convergence is obtained when the block size becomes fine in the z-axis. In order to satisfy these two requirements, the same block size scan is repeated by transposing the data structure from xyz to zyx (t_2), and the best performance is obtained with $8 \times 8 \times 1$, which shows a $\sim$ 64% increase in the number of iterations, and a $5\times$ speedup against the CPU preconditioner. This result shows a trade off between mathematical and computational properties in the current preconditioner design. It is noted that the data transpose is a one time cost before and after KSMs.

C. Tall-skinny matrix operations

The kernels described here are only used inside the P-CBCG algorithm. In each iteration of the P-CBCG method, a matrix of the size $n \times s$ is generated with $n \gg s$, a tall-skinny (TS) matrix. Here, $n = n_x \times n_y \times n_z$ is the size of subdomain on each MPI process. The P-CBCG method consists of computing kernels involving TS matrices. In the CPU version, high performance was achieved by using a cache blocking approach and by fusing several TS matrix operations into a single kernel in order to increase the arithmetic intensity.

TABLE I: Block size dependency of the BJ preconditioner on a single CPU/GPU is shown for a small JUPITER matrix with $N = N_x \times N_y \times N_z = 256 \times 128 \times 512$. The number of iterations and the elapse time are summarized, where t_1 and t_2 being the time before and after the data transpose from xyz to zyx, respectively. Case 1 uses Broadwell with 14 OpenMP threads, and all other cases use V100.

Case	x	y	z	# of iterations	t_1	t_2
1. (CPU)	256	128	36	106	34.60	-
2.	16	32	4	142	17.49	9.45
3.	16	16	4	145	38.62	7.64
4.	8	8	16	156	13.04	39.44
5.	16	16	1	161	48.69	7.27
6.	8	8	1	174	47.34	6.87
7.	4	4	4	185	22.83	22.83
8.	2	4	64	210	8.70	80.67
9.	1	8	8	263	10.79	69.13

The lines 5 and 11 in Algorithm 2 multiply two TS matrices ($s \times n$ and $n \times s$) to obtain a square matrix of the size $s \times s$. In Ref. [17], a single BLAS GEMM operation performed poorly on TS matrices, and high performance was achieved using a batched GEMM approach. We implement this approach using the batched GEMM routine in the NVIDIA cuBLAS library. In the current problem, the JUPITER data contains halo data regions, which must be skipped to avoid extra computation. This can be treated by a pointer to pointer interface of this routine. The TS matrix is divided into n/b smaller matrices by the specified batch size b, which is adjusted depending on the size of each subdomain. In this work, we chose $b = 128$. The batched GEMM routine outputs n/b matrices, which are reduced to obtain the final square matrix.

Other TS matrix operations used inside the P-CBCG method are GEMV at the line 6 ($s \times n$ and $n \times 1$), GEMM at the lines 13 and 14 ($n \times s$ and $s \times s$), and GEMV at the lines 8 and 9 ($n \times s$ and $s \times 1$). Although the batched GEMM routine can be applied also to these kernels, we refactored the kernels into a single kernel, in which the small matrix and vector data are loaded into the shared memory, from which they are loaded and multiplied many times with a part of the TS matrix data.

D. Refactoring GPU kernels

We apply kernel fusion to the P-CG method in Algorithm 1 in order to improve the arithmetic intensity inside each kernel. The two AXPY kernels in line 5 and 9 are merged into a single kernel, since the vector p_j is accessed in both kernels. The SpMV kernel in line 3 is merged with the inner product in line 4 because p_j and w are accessed in both kernels. The second inner product in line 8 is merged into the preconditioner kernel in line 7 to share z_{j+1} and r_{j+1}.

On the other hand, refactoring GPU kernels in the P-CBCG method in Algorithm 2 is more complicated. In Fig.2, the left figure shows the main GPU kernels as they were in the original algorithm. In the right figure, they were refactored into four

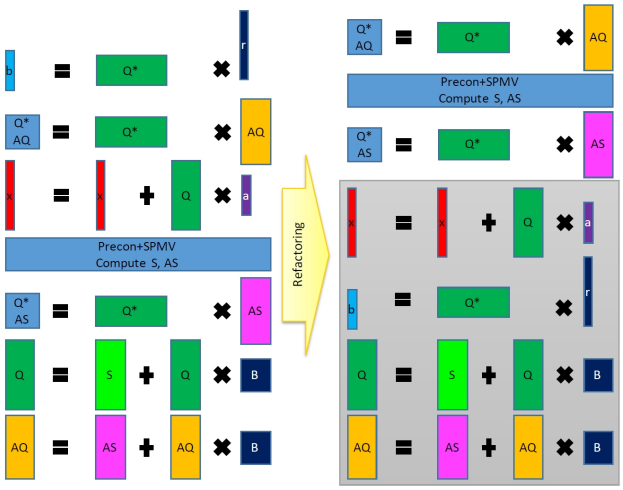


Fig. 2: Refactoring GPU kernels in the P-CBCG method. Left shows main computing kernels as in Algorithm 2. Right shows refactored kernels.

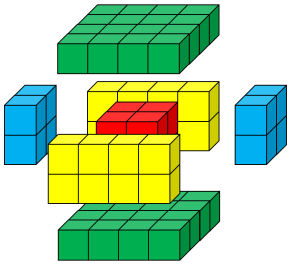


Fig. 3: Decomposing the subdomain into surface and core parts while maximizing the coalesce memory access.

TABLE II: Communication overlap in the P-CG solver.

Step	Stream name	operation
1	default	Surface parts of AXPY
2a	default	Halo data communication using Isend/Irecv
2b	calc	Core parts of AXPY and SpMV
3	default	Wait for Isend/Irecv to finish
4	default	Surface parts of SpMV

GPU kernels, the preconditioner kernel, the SpMV kernel, the TS GEMM kernels using the batched GEMM routine, and the remaining TS GEMM/GEMV kernel optimized using the above implementation. In the TS GEMM/GEMV kernel, the small matrix B_k and vector a_k are loaded into the shared memory, and the large TS matrices Q_k and AS_{k+1} are reused to reduce memory access.

E. Communication overlap

Overlapping communication and computation is an important method to hide the latency in the data exchange. In JUPITER, we overlap the halo data communication with computation. In order to implement the communication overlap, the subdomain must be decomposed into surface and core parts, where only the surface part needs halo data. The core part can be computed independently of the halo data, and thus, can be simultaneously processed during the halo data communication. The surface-core decomposition normally introduces a performance penalty. This is especially true for GPUs, because the device memory access needs to be coalesced. For this reason, the surface-core decomposition was designed to maximize the coalesced memory access. Figure 3 shows the surface-core decomposition, in which the bottom, top, front, and back surfaces are maximized in the coalesced memory access direction, and the left and right surfaces are minimized in the non-coalesced directions.

Table II shows the communication overlap in the P-CG solver. Firstly, the surface part of the AXPY kernel is computed, and then, the halo data communication and the core part computation of the AXPY and SpMV kernels are executed simultaneously. Finally, the surface part of SpMV is computed. It is noted that the preconditioner kernel would also be a candidate for the communication overlap. However, we found that dividing the preconditioner kernel into surface and core parts resulted in a large performance degradation, and thus, the preconditioner kernel was not used. In the P-CBCG solver, the AXPY kernel is replaced with the TS GEMM kernel, and s-SpMVs are processed at once. Therefore, we use only the SpMV kernel for the communication overlap. Since the P-CG method can use two computing kernels, the P-CG method is able to hide more communication than the P-CBCG method. The communication overlap is implemented using multiple CUDA streams. The *default* stream is used for the surface part computation and the halo data communication, which is implemented using GPU direct communication and CUDA-aware MPI. On the other hand, the core part computation is processed in a separate stream called *calc*.

V. Numerical Experiments

A. Kernel performance on V100

In order to evaluate the quality of our GPU implementation, we compare the achieved performance against the modified roofline model [19], in which a theoretical processing time of each kernel is estimated by the sum of costs for floating point operations and memory accesses, $t_{RL} = f/F + b/B$. Here, f and b are the numbers of floating point operations and memory accesses of the kernel, and F and B are the peak performance and the memory bandwidth of the GPU. We compute a small JUPITER matrix with $N = N_x \times N_y \times N_z = 256 \times 128 \times 512$ using a single V100 GPU, which has F=7,800GFlops and B=835GBytes/s (memory bandwidth obtained using the BabelStream benchmark[20]). Table III shows the kernel performance analysis for each kernel in the P-CBCG method. The dominant computational costs come from the SpMV and preconditioner kernels, which are memory bound $f/b \sim 0.16$ and therefore benefit from the high memory bandwidth of the V100. On the other hand, TS matrix kernels have relatively high arithmetic intensity $f/b \sim 1$. The result shows that all computing kernels are fully optimized with a roofline ratio of $\sim 90\%$.

TABLE III: Kernel performance analysis: Floating point operation f [Flop/grid], Memory access b [Byte/grid], Arithmetic intensity f/b, Peak performance F [Flops], memory bandwidth B [Byte/s], Roofline time $t_{RL} = f/F + b/B$ [ns/grid], Elapse time t [ns/grid], Sustained performance P [GFlops].

Kernel	f	b	f/b	t	P	t_{RL}/t
SpMV	15	92	0.16	0.12	135	0.90
Preconditioner	20	128	0.16	0.18	128	0.84
AXPY	3	24	0.13	0.03	103	0.85
batched GEMM	48	35	1.38	0.05	1002	0.95
GEMV/GEMM	54	59	0.91	0.10	694	0.81
Total	141	338	0.42	0.46	332	0.92

TABLE IV: Hardware metrics of the GPU/CPU platforms.

	ABCI	Summit	Oakforest-PACS
GPU/CPU	V100	V100	KNL
GPUs/CPUs per node	4	6	1
Peak performance [GFlops]	7,800	7,800	3,046
STREAM BW [GByte/s]	835	835	480(MCDRAM)
Interconnect BW/node [GByte/s]	25	25	12.5

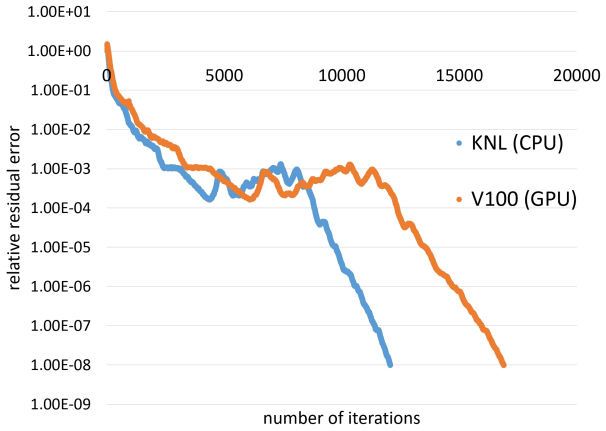


Fig. 4: Convergence property of the P-CG solvers using the CPU and GPU preconditioner. JUPITER matrix size with $N = N_x \times N_y \times N_z = 1,280 \times 1,280 \times 4,608$ processed using 2,048 KNLs and GPUs. The CPU and GPU solvers converged with 12,058 and 16,897 iterations, respectively.

TABLE V: Cost distribution (msec/iteration) with and without communication overlap. A large JUPITER matrix with $N = N_x \times N_y \times N_z = 1,280 \times 1,280 \times 4,608$ is computed using 1,024 KNLs and GPUs. The total cost is shown with and without the communication overlap.

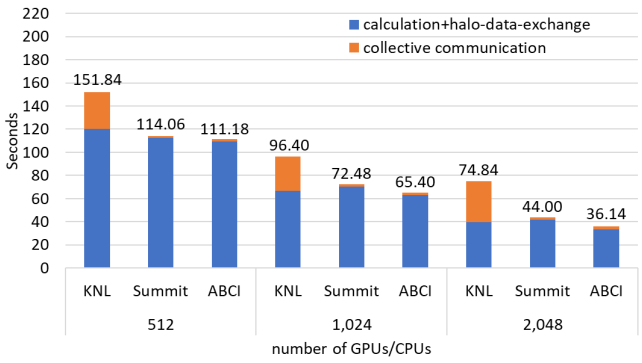
	No Overlap			Overlap	
msec/iteration	KNL	ABCI	Summit	ABCI	Summit
P-CG					
Reduce	2.46	0.20	0.14	0.14	0.14
Halo	0.78	1.48	2.18	1.52	1.98
Precon	2.81	1.61	1.63	1.61	1.63
SpMV	1.21	1.19	1.28	1.21	1.32
AXPY	0.69	0.36	0.38	0.38	0.40
Total w/o overlap	7.95	4.84	5.61	4.86	5.47
Total w/ overlap	-	-	-	3.87	4.29
P-CBCG					
Reduce	0.91	0.03	0.03	0.03	0.04
Halo	1.05	1.37	2.12	1.37	2.08
Precon	2.32	1.44	1.44	1.44	1.44
SpMV	1.71	0.93	0.93	0.96	0.96
Matrix	2.75	1.09	1.10	1.09	1.10
Total w/o overlap	8.74	4.86	5.63	4.87	5.62
Total w/ overlap	-	-	-	4.35	4.92

B. Computing platforms

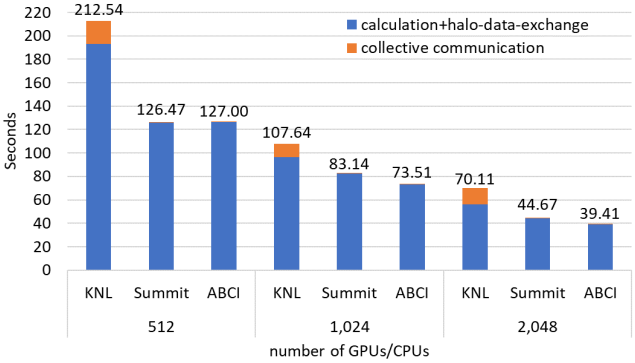
Large scale numerical experiments were conducted on ABCI, Summit, and the Oakforest-PACS clusters using a large JUPITER matrix with $N = N_x \times N_y \times N_z = 1,280 \times 1,280 \times 4,608$. Table IV shows the hardware specifications of these platforms. Both ABCI and Summit are based on the NVIDIA V100 GPU and a dual-rail EDR InifiniBand interconnect, while Oakforest-PACS consists of the Intel Xeon Phi 7250 processor (KNL) and a Omni Path interconnect. Although the interconnect bandwidth on ABCI and Summit is doubled compared to Oakforest-PACS, it is shared by four and six GPUs, and the interconnect bandwidth per GPU becomes 1/2 and 1/3, respectively. We use CUDA 9.2 and the gcc 4.8.5 (-O3) together with OpenMPI 2.1.6 on ABCI and IBM-Spectrum MPI on Summit. On Oakforest-PACS, we use the Intel compiler 17.0.4 with the Intel MPI library 2017 (-O3 -qopenmp -xMIC-AVX512), and the CPU solver results were obtained by using the MCDRAM in a flat mode. Each MPI rank was assigned to a single GPU/CPU, and we only used 64 threads on KNL with 68 cores. In the P-CBCG solver, we used $s = 12$ for both the CPU and GPU solvers. As discussed in Sec.IV-B, the CPU and GPU solvers use different preconditioner implementations. Fig 4 shows a comparison between both implementations, the re-designed GPU preconditioner increases the number of iterations by $\sim 40\%$.

C. Impact of communication overlap

Table V shows the detailed cost distributions of the CPU and GPU solvers for 1,024 CPUs/GPUs with and without the communication overlap. In the measurement, each kernel is synchronized and the communication overlap is turned off with keeping the surface-core decomposition of related computing kernels. The CPU solvers do not use communication overlap, because the surface-core decomposition of the computing kernels introduced a significant performance degradation on KNL. The cost of global collective communication on Oakforest-PACS is an order of magnitude larger than on the GPU platforms, while the P-CBCG solver reduces it to $\sim 1/3$ and $\sim 1/5$ on these platforms, respectively. On the other hand, the halo data communication on ABCI and Summit is respectively $\sim 2\times$ and $\sim 3\times$ slower than on Oakforest-PACS, reflecting the interconnect bandwidth per CPU/GPU. The GPU preconditioner is $\sim 1.7\times$ faster than the CPU preconditioner, when comparing the computational cost per iteration. The SpMV kernel in the P-CBCG solver is $\sim 1.8\times$ faster on



(a) P-CG cross platform results



(b) P-CBCG cross platform results

Fig. 5: Cross platform comparisons using ABCI, Summit, and Oakforest-PACS. Strong scaling tests up to 2,048 KNLs/GPUs. JUPITER matrix of size $N = N_x \times N_y \times N_z = 1,280 \times 1,280 \times 4,608$.

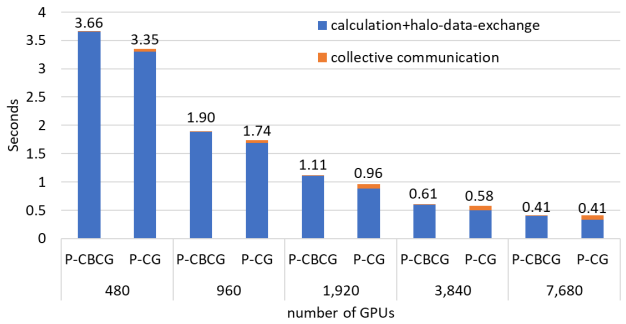


Fig. 6: Strong scaling test on Summit up to 7,680 GPUs. JUPITER matrix of size $N = 1,280 \times 1,280 \times 4,608$. Number of iterations fixed to 480 SpMVs both in the P-CG and P-CBCG solver.

the V100, while the P-CG solver shows the similar SpMV performance between KNL and the V100. The remaining AXPY and matrix kernels are $\sim 1.8\times$ and $\sim 2.5\times$ faster on the V100, and the overall speedup of computing kernels in the P-CG and the P-CBCG solvers becomes $\sim 1.5\times$ and $\sim 2\times$, respectively. The SpMV and AXPY kernels show almost the same costs with and without the communication overlap, indicating only a small overhead due to the surface-core decomposition.

The cost distribution clearly shows that the halo data communication is the remaining critical bottleneck, especially on GPUs. To resolve this issue, we implemented communication overlap. In the P-CG solver, the SpMV and AXPY kernels are overlapped with halo data communication. On ABCI, the cost of the former computing kernels becomes comparable to the latter communication kernel. On the other hand, on Summit, the latter exceeds the former, and the communication cost cannot be fully masked. In fact, when the communication overlap is turned on, ABCI is $\sim 10\%$ faster than Summit, while the total cost reduction exceeds more than 25% on both GPU platforms. In the P-CBCG solver, only the SpMV kernel is used for the communication overlap, and the total cost reduction is limited to $10\% \sim 15\%$. At 1,024 GPUs, an impact of the communication overlap exceeds that of CA on the global collective communication, and the P-CG solver is faster than the P-CBCG solver.

D. Strong scaling test

At first, we compare the strong scaling of the P-CG and P-CBCG solvers up to 2,048 GPUs/CPUs. The strong scaling of the P-CG solver is shown in Figure 5a. On Oakforest-PACS, the P-CG solver shows good scaling up to 1,024 CPUs. However, for the 2,048 CPU case, the global collective communication accounts for almost $\sim 50\%$ of the total cost. On ABCI and Summit, the P-CG solver keeps low global collective communication costs, and shows good scaling up to 2,048 GPUs. The performance gain of the GPU solver over the CPU solver ranges from $1.3 \sim 1.7\times$ on Summit and $1.3 \sim 2.0\times$ on ABCI, although executing 40% more iterations.

Figure 5b shows the strong scaling of the P-CBCG solver. All solvers show good scaling up to 2,048 GPUs/CPUs. On Oakforest-PACS, the P-CBCG solver keeps better scaling, because the CA property reduces the global collective communication to $\sim 20\%$ of the total cost. On both GPU platforms, the global collective communication is negligible, and the computing kernels with the halo data communication becomes dominant. The performance gain of the GPU solver over the CPU solver ranges from $1.2 \sim 1.6\times$ on Summit and $1.4 \sim 1.7\times$ on ABCI.

Finally, we present strong scaling up to 7,680 GPUs on Summit. In this test, we fix the number of iterations to 480 in order to save computational resources. Both the P-CG and P-CBCG solvers show good strong scaling up to 7,680 GPUs. The P-CBCG solver has a $\sim 3\times$ and $\sim 1.25\times$ larger computation and memory access cost than the P-CG solver [16]. In addition, the communication overlap provides larger cost reduction in the P-CG solver. Because of these advantages, the P-CG solver outperforms the P-CBCG solver

up to 3,840 GPUs. However, at 7,680 GPUs, both solvers become comparable. Here, the cost of global collective communication in the P-CG solver increases up to $\sim 20\%$ of the total cost, while it is suppressed to $\sim 1.3\%$ in the P-CBCG solver, showing almost ideal cost reduction $\sim 1/s$.

VI. Conclusion

In this work, we extended the JUPITER CFD code with GPU Poisson solvers based on the P-CG and P-CBCG methods. All main kernels were implemented using CUDA. By using the DIA format, coalesced memory access was achieved inside the SpMV kernel. The BJ preconditioner was redesigned in order to take full advantage of the GPU hardware. A new 3D block decomposition with finer tiles and data transpose allowed the preconditioner kernel to utilize a several orders of magnitude larger number of threads while providing coalesced device memory access, however the quality of the preconditioner decreased resulting in an increase of iterations by $\sim 40\%$. The TS matrix operations were optimized using the batched GEMM routine in the NVIDIA cuBLAS library. The other TS matrix kernels were refactored and optimized using the shared memory. Overlap of communication and computation was implemented using multiple CUDA streams and by dividing the computing kernels into core and surface parts. Our GPU implementation achieved $\sim 90\%$ of the theoretical performance limit based on the roofline model.

The developed solvers showed good strong scaling up to 7,680 GPUs on Summit, where the P-CG solver outperformed the P-CBCG solver up to 3,840 GPUs. It was shown that the communication overlap has a larger impact on the P-CG solver, while the global collective communication becomes $1/s$ in the P-CBCG solver. The total performance is determined by the competition between these two CA properties, and the latter becomes significant beyond several thousands of GPUs. In a cross platform comparison, the GPU solvers on ABCI and Summit were compared with the CPU solvers on Oakforest-PACS using $512 \sim 2,048$ CPUs/GPUs. The speedup of the GPU solvers ranged from $1.3 \sim 2.0\times$ in the P-CG solver and $1.3 \sim 1.7\times$ in the P-CBCG solver. It was also shown that ABCI is $\sim 10\%$ faster than Summit, because of a larger interconnect bandwidth per GPU with less GPU density per node.

Acknowledgment

This work is supported by the MEXT (Grant for Post-K priority issue No.6: Development of Innovative Clean Energy) and "Large-scale HPC Challenge" Project, the Joint Center for Advanced High Performance Computing (JCAHPC). Computations were performed on the Oakforest-PACS (JCAHPC), the ABCI (AIST), the Tsubame3.0 (Titech, hp190073, jh190050), and the ICEX (JAEA). This research also used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

References

[1] M. Hoemmen, "Communication-avoiding Krylov subspace methods," Ph.D. dissertation, University of California, Berkeley, 2010.

[2] S. Yamashita, T. Ina, Y. Idomura, and H. Yoshida, "A numerical simulation method for molten material behavior in nuclear reactors," *Nuclear Engineering and Design*, vol. 322, pp. 301 – 312, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0029549317303035

[3] [Online]. Available: http://abci.ai/

[4] [Online]. Available: http://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/

[5] [Online]. Available: http://jcahpc.jp/eng/ofp_intro.html

[6] J. V. Rosendale, "Minimizing inner product data dependencies in conjugate gradient iteration," 1983.

[7] A. T. Chronopoulos and C. W. Gear, "s-step iterative methods for symmetric linear systems," *Journal of Computational and Applied Mathematics*, vol. 25, no. 2, pp. 153–168, 1989.

[8] W. D. Joubert and G. F. Carey, "Parallelizable restarted iterative methods for nonsymmetric linear systems. II: parallel implementation," *International journal of computer mathematics*, vol. 44, no. 1-4, pp. 269–290, 1992.

[9] S. A. Toledo, "Quantitative performance modeling of scientific computations and creating locality in numerical algorithms," Ph.D. dissertation, Massachusetts Institute of Technology, 1995.

[10] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick, "Avoiding communication in sparse matrix computations," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, April 2008, pp. 1–12.

[11] E. C. Carson, "Communication-Avoiding Krylov Subspace Methods in Theory and Practice," Ph.D. dissertation, University of California, Berkeley, 2015.

[12] Y. Kumagai, A. Fujii, T. Tanaka, Y. Hirota, T. Fukaya, T. Imamura, and R. Suda, "Performance Analysis of the Chebyshev Basis Conjugate Gradient Method on the K Computer," in *International Conference on Parallel Processing and Applied Mathematics*. Springer, 2015, pp. 74–85.

[13] I. Yamazaki, S. Rajamanickam, E. G. Boman, M. Hoemmen, M. A. Heroux, and S. Tomov, "Domain decomposition preconditioners for communication-avoiding krylov methods on a hybrid CPU/GPU cluster," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 933–944.

[14] A. Mayumi, Y. Idomura, T. Ina, S. Yamada, and T. Imamura, "Left-preconditioned communication-avoiding conjugate gradient methods for multiphase cfd simulations on the k computer," in *2016 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, Nov 2016, pp. 17–24.

[15] R. Suda, L. Cong, and D. Watanabe, "Communication-avoiding CG method: New direction of krylov subspace methods towards exa-scale computing," RIMS Kôkyûroku, vol. 1995, pp. 102–111, 2016.

[16] Y. Idomura, T. Ina, A. Mayumi, S. Yamada, and T. Imamura, "Application of a preconditioned chebyshev basis communication-avoiding conjugate gradient method to a multiphase thermal-hydraulic cfd code," in *Supercomputing Frontiers*, R. Yokota and W. Wu, Eds. Cham: Springer International Publishing, 2018, pp. 257–273.

[17] I. Yamazaki, H. Anzt, S. Tomov, M. Hoemmen, and J. Dongarra, "Improving the performance of ca-gmres on multicores with multiple gpus," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 382–391.

[18] K. Matsumoto, Y. Idomura, T. Ina, A. Mayumi, and S. Yamada, "Implementation and performance evaluation of a communication-avoiding gmres method for stencil-based code on gpu cluster," *Journal of Supercomputing*, to appear.

[19] T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T. Endo, A. Nukada, N. Maruyama, and S. Matsuoka, "An 80-fold speedup, 15.0 tflops full gpu acceleration of non-hydrostatic weather model asuca production code," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2010, pp. 1–11.

[20] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "Evaluating attainable memory bandwidth of parallel programming models via babelstream," *International Journal of Computational Science and Engineering (special issue)*, 2017.