KoHPCG – High-Performance Conjugate Gradient Benchmark Program on Kokkos Performance Portability Ecosystem

Muhammad Rizwan[†], Jaeyoung Choi[†*], Yoonhee Kim[‡]
[†]School of Computer Science and Engineering, Soongsil University, Seoul, South Korea
[‡]Department of Computer Science, Sookmyung Women's University, Seoul, South Korea
Email: mrizwan@soongsil.ac.kr, choi@ssu.ac.kr, yulan@sookmyung.ac.kr

***Abstract*—KoHPCG is a performance-portable HPCG benchmark program developed using the Kokkos programming model. The Reference HPCG benchmark is constrained by memory-bound kernels that restrict performance across various architectures. This paper details a thorough implementation of core HPCG kernels, including Dot Product (DDOT), WAXPBY, Sparse Matrix Vector Multiplication (SpMV), Symmetric Gauss-Seidel (SymGS), and Multigrid (MG), utilizing the Kokkos::Views, parallel constructs, execution and memory space abstraction.**

Evaluated on Intel Xeon Phi (KNL) and Xeon Skylake (SKL) processors with a maximum of 16 nodes, KoHPCG achieves significant performance improvement up to 11.7× acceleration in MG on SKL, alongside a 17.3× MG improvement on KNL. The overall HPCG performance increases by as much as 3.2× on SKL and attains 4.1× on KNL compared to the Reference HPCG implementation, on problem sizes of 192^3 and 160^3 on SKL and KNL, respectively. Furthermore, on a larger problem size of 320^3, KoHPCG attains an overall HPCG performance improvement of 5.1× on SKL.

In addition to the overall performance improvement in HPCG, we reported the kernel-level performance that exposes the little and no performance improvement in SpMV and WAXPBY respectively, and increased memory consumption by using the Kokkos data structure, thereby highlighting further optimisation opportunities for future work.

***Index Terms*—High-Performance Conjugate Gradient (HPCG), Kokkos, Performance Portability, Symmetric Gauss-Seidel (SymGS), Multigrid (MG), High-Performance Computing (HPC)**

I. Introduction

The High-Performance LINPACK benchmark (HPL) [1] for distributed memory computers has been widely used since the 1990s to evaluate supercomputer performance. HPL has been a benchmark for years, but real-world applications have revealed its limitations. HPL measures only the computational capability and shows the peak performance. The High-Performance Conjugate Gradient (HPCG) is a new benchmark to evaluate and compare the performance of modern supercomputers [2–5]. HPCG evaluates modern applications by testing not just the computational capability of the systems but also their ability to solve complex problems. HPCG tests memory bandwidth, computation power, interconnect network performance, and overall system synergy. HPCG is a benchmark that more accurately represents the performance of modern, practical, and real-world applications. HPCG suffers significant performance bottlenecks due to memory-bound kernels such as Sparse Matrix-Vector (SpMV) and Symmetric Gauss-Seidel (SymGS) operations.

Kokkos [6] is a C++ library designed to facilitate performance portability across various hardware architectures, including CPUs, GPUs, and emerging platforms. It offers abstractions for parallel execution and data management, enabling developers to create code that is efficient and portable without engaging with hardware-specific complexities. Kokkos supports various backends, including CUDA, HIP, SYCL, OpenMP, and C++ threads, enabling seamless transitions among diverse execution environments. Kokkos Kernels provides a collection of performance-optimized routines, covering sparse and dense linear algebra, batched operations, and graph algorithms, all constructed based on the Kokkos programming model. This combination enables developers to create high-performance applications that meet the evolving paradigm of high-performance computing.

For the HPCG benchmark optimization, many architecture-aware optimization strategies have been used to resolve these performance bottlenecks for the targeted hardware platforms. Park and Smelyanskiy [7], and Park et al. [8] suggested optimizations using SELLPACK, asynchronous scheduling, and Gauss-Seidel (GS) smoother tuning for Intel CPU architectures. Phillips & Fatica [9] and Phillips et al. [10] developed CUDA-based kernels with multi-color reordering and overlapping communication and computation, on GPU systems to improve SpMV and SymGS performance. For hybrid CPU-MIC systems, Y. Liu et al. [11, 12] introduced inner-outer domain decomposition and asynchronous data transfer, Red-Black relaxation, forward-backward sweeps fusion for CPUs and Many Integrated Cores (MICs). Using kernel-to-kernel streaming, memory tuning, and load balancing in Xilinx Alveo FPGA devices, Zeni et al. [13] and Steiger [14] showed performance improvements on FPGAs. For Sunway architectures, Zhu et al. [15]

979-8-3315-5693-8/25/$31.00 ©2025 IEEE

and Ao et al. [16] proposed hierarchical blocking, layout transformations, and kernel fusion techniques to exploit many-core parallelism. These studies highlight the need for hardware-aware optimizations for the HPCG benchmark, but they lack generality and portability.

Z. Bookey showcased the development of KHPCG [17, 18], the first attempt of creating a performance-portable version of the HPCG benchmark utilizing the Kokkos library [6, 19, 20], which provides hardware abstraction. The Kokkos-based KHPCG variant, but is constrained by limited performance, supports for only single MPI rank, and numerical instability in multicolor SymGS implementation. The report [21] highlighted the primary challenges in the practical execution of the KHPCG. These challenges emphasize the necessity for a comprehensive new implementation of a performance-portable HPCG benchmark capable of functioning across various platforms.

KoHPCG aims to bridge this gap by offering improved parallelism, multi-rank support, and a Kokkos-based implementation. KoHPCG is developed with the following considerations in mind:

- HPCG optimizations done so far are closely associated with specific architecture and require architecture-specific tuning.
- The data dependencies and sequential nature of SymGS render it highly challenging to parallelize.
- HPCG with the Kokkos abstraction lacks a practical, scalable, and optimized version.
- Acquisitions of HPC systems increasingly rely on benchmarks such as HPCG that reflect realistic workloads. Consequently, both evaluators and developers can gain advantages from a portable and efficient implementation.

This paper presents the KoHPCG, a performance-portable HPCG implementation optimized using the Kokkos programming model. A complete re-implementation of the HPCG benchmark core kernels, such as DOT Product, WAXPBY, SpMV, and SymGS utilizing the Kokkos library for performance portability. KoHPCG demonstrates significant performance improvement on Intel Xeon Phi (KNL) and Xeon Skylake (SKL) processors with up to 16 nodes, achieving performance improvements by 3.2× on SKL and 4.1× on KNL compared to the Reference HPCG implementation, on problem sizes of 192^3 and 160^3 on SKL and KNL respectively. While HPCG showed overall performance improvement, the kernel-level performance of SpMV and WAXPBY showed little and no performance improvement, respectively, and increased memory consumption by using the Kokkos data structure, thereby highlighting further optimization opportunities for future work.

II. HPCG

The HPCG [22, 23] benchmark uses a 27-point stencil and a preconditioned conjugate gradient solver to simulate a 3D discretized partial differential equation. The Preconditioned Conjugate Gradient (PCG) algorithm 1 is composed of core computational kernels which are allowed to optimize in HPCG, and are as follows: Dot Product (DDOT), WAXPBY (scaled vector update), Sparse Matrix-Vector Multiply (SpMV), Symmetric Gauss-Seidel (SymGS), and a Multigrid preconditioner (MG). Irregular memory access and inherent data dependencies constrain SpMV and SymGS, thereby significantly restricting HPCG performance.

Algorithm 1 Preconditioned Conjugate Gradient (PCG)

Input: Matrix A, vector b, initial guess x_0, tolerance ϵ, max iterations $k_{\max}$
Output: Approximate solution x
$r_0 \leftarrow b - Ax_0$
$p_0 \leftarrow r_0$
$\rho_0 \leftarrow \|r_0\|_2$
for $k = 0$ to $k_{\max} - 1$ **do**
  $z_k \leftarrow \mathrm{MG}(A, r_k)$
  $\delta_k \leftarrow r_k^\top z_k$
  $q_k \leftarrow Ap_k$
  $\alpha_k \leftarrow \delta_k / (p_k^\top q_k)$
  $x_{k+1} \leftarrow x_k + \alpha_k p_k$
  $r_{k+1} \leftarrow r_k - \alpha_k q_k$
  if $\|r_{k+1}\|_2 / \rho_0 < \epsilon$ **then**
    break
  end if
  $\beta_k \leftarrow \delta_{k+1} / \delta_k$
  $p_{k+1} \leftarrow z_{k+1} + \beta_k p_k$
end for

HPCG solves the linear system of equation $Ax = b$, where A is a sparse, symmetric, and positive-definite matrix. The PCG method in HPCG uses several essential computational kernels to derive the solution.

- **Dot Product (DDOT):** $\alpha = x \cdot y$, where x and y are vectors. It calculates the scalar result of two input vectors x and y, each of length n.
- **WAXPBY:** $w = \alpha \cdot x + \beta \cdot y$, where α and β are the scalar values, x and y are input vectors, and w is the resultant vector. It is the weighted addition of two vectors x and y (scaled vector αx plus a scaled vector βy), also known as vector vector coefficient multiplication [13].
- **Sparse Matrix-Vector Multiply (SpMV):** $y = Ax$, It computes the product of a sparse matrix A with a vector x to produce the resultant vector y.
- **Symmetric Gauss–Seidel (SymGS):** An iterative method for solving $A\mathbf{x} = \mathbf{r}$, where $A = L + D + U$ with L, D, and U being the lower, diagonal, and upper parts of A, respectively. It performs a forward sweep $(L + D)\mathbf{x}^{(k+1)} = \mathbf{r} - U\mathbf{x}^{(k)}$ and a backward sweep $(U + D)\mathbf{x}^{(k+1)} = \mathbf{r} - L\mathbf{x}^{(k)}$. This routine is inherently sequential with limited parallelism, consuming a significant amount of computational time and becoming the performance bottleneck in HPCG.
- **Multigrid V-cycle (MG):** A hierarchical solver that accelerates convergence by using the pre- and post-smoother, coarse-grid restriction, coarse-grid

correction, and prolongation back to fine grids. It effectively lowers both high- and low-frequency errors using the SymGS as the smoother.

III. Methodology

Kokkos [6], a C++ library for performance portability, was developed by Sandia National Laboratories. It provides abstractions for data layouts (LayoutLeft, LayoutRight), memory domains (e.g., Host, CudaSpace, UVM), and execution environments (e.g., CUDA, OpenMP). Kokkos enables developers to target multiple architectures with minimal code modifications. Using constructs such as `parallel_for`, `parallel_reduce`, and `Kokkos::Views`, it facilitates scalable parallel execution and memory management. The Kokkos ecosystem extends beyond the core library to include optimized kernels (dense and sparse linear algebra, graph algorithms), remote spaces for distributed memory models, and profiling/autotuning tools, making it a comprehensive platform for portable HPC applications. We port HPCG core computational kernels to the Kokkos performance-portability framework, naming the resulting implementation as KoHPCG.

In KoHPCG implementation, we rewrite all computational kernels using Kokkos constructs, preserving the original algorithmic structure of HPCG.

- Used `Kokkos::View` for managing multi-dimensional arrays across execution spaces.
- Used `Kokkos::parallel_for` and `Kokkos::parallel_reduce` to enable thread-level and data-level parallelism.
- Used `Kokkos::fence` and memory traits to ensure consistency across host and device execution.
- For now, we used default execution and memory space using `Kokkos::DefaultExecutionSpace` and `ExecSpace::memory_space`.

A. Kokkos-Based Implementation

DDOT

```
Kokkos::parallel_reduce("DDOT", n,
  KOKKOS_LAMBDA(const local_int_t i, double &update) {
    update += x.values(i) * y.values(i);
}, local_result);
```

Kokkos kernels also offer this operation as `kokkosBlas::dot(...)`. In our current results, we utilized KokkosBlas operation because Kokkos kernels internally employ the architecture-tuned implementation for enhanced portability.

WAXPBY

```
Kokkos::parallel_for("WAXPBY", Kokkos::RangePolicy<>(0, n),
  KOKKOS_LAMBDA(const int i) {
    w(i) = alpha * x(i) + beta * y(i);
});
```

SpMV

```
Kokkos::parallel_for("SpMV", Kokkos::RangePolicy<>(0, nrow),
  KOKKOS_LAMBDA(const local_int_t i) {
    double sum = 0.0;
    int nnz = (int)(nonzerosInRow_d(i));
    for (int j = 0; j < nnz; j++) {
      sum += matrixValues_d(i, j) * xv(mtxIndL_d(i, j));
    }
    yv(i) = sum;
});
```

Kokkos kernels also provide this operation as `Kokkos Sparse::spmv(...)`. In our current implementation, we utilized the Compressed Sparse Row (CSR) based SpMV offered by KokkosSparse, as it provides us the relatively good HPCG performance when combined with our optimized SymGS kernel. We noted that the Block Sparse Row (BSR) format exhibited superior SpMV performance compared to CSR when evaluated only on SpMV.

SymGS

```
Kokkos::View<double*, Layout, MemorySpace> tmpX("tmpX", size);
deep_copy(tmpX, xv);

Kokkos::parallel_for("Forward_Sweep",
  Kokkos::RangePolicy<ExecSpace>(0, nrow),
  KOKKOS_LAMBDA(const local_int_t i) {
    double sum = rv(i);
    int cur_nnz = nnzInRow(i);
    double diag_val = diag(i);
    double x_i = xv(i);
    for (int j = 0; j < cur_nnz; j++) {
      const local_int_t col = indices(i, j);
      sum -= values(i, j) * xv(col);
    }
    sum += x_i * diag_val;
    tmpX(i) = sum / diag_val;
});
Kokkos::fence("Forward_Sweep");

Kokkos::parallel_for("Backward_Sweep",
  Kokkos::RangePolicy<ExecSpace>(0, nrow),
  KOKKOS_LAMBDA(const local_int_t idx) {
    const local_int_t i = nrow - 1 - idx;
    double sum = rv(i);
    int cur_nnz = nnzInRow(i);
    double diag_val = diag(i);
    for (int j = 0; j < cur_nnz; j++) {
      const local_int_t col = indices(i, j);
      sum -= values(i, j) * tmpX(col);
    }
    sum += tmpX(i) * diag_val;
    xv(i) = sum / diag_val;
});
Kokkos::fence("Backward_Sweep");
```

Given the inherently sequential nature of the SymGS operation, we optimized and parallelized it by employing a temporary vector to eliminate the dependency. This straightforward modification requires reading and writing to different memory locations to avoid conflicts arising from partial updates, thereby facilitating the parallelization of the SymGS operation. This approach permits concurrent forward and backward execution, data dependencies are not violated, and improves the performance of the SymGS operation, which overall contributes in the performance improvement of HPCG.

MG

Multigrid (MG) is composed of SymGS, SpMV, Prolongation, and Restriction operations. We modified SpMV and SymGS to the Kokkos programming model, so we also update the Prolongation and Restriction operations with required modifications using `Kokkos::parallel_for`.

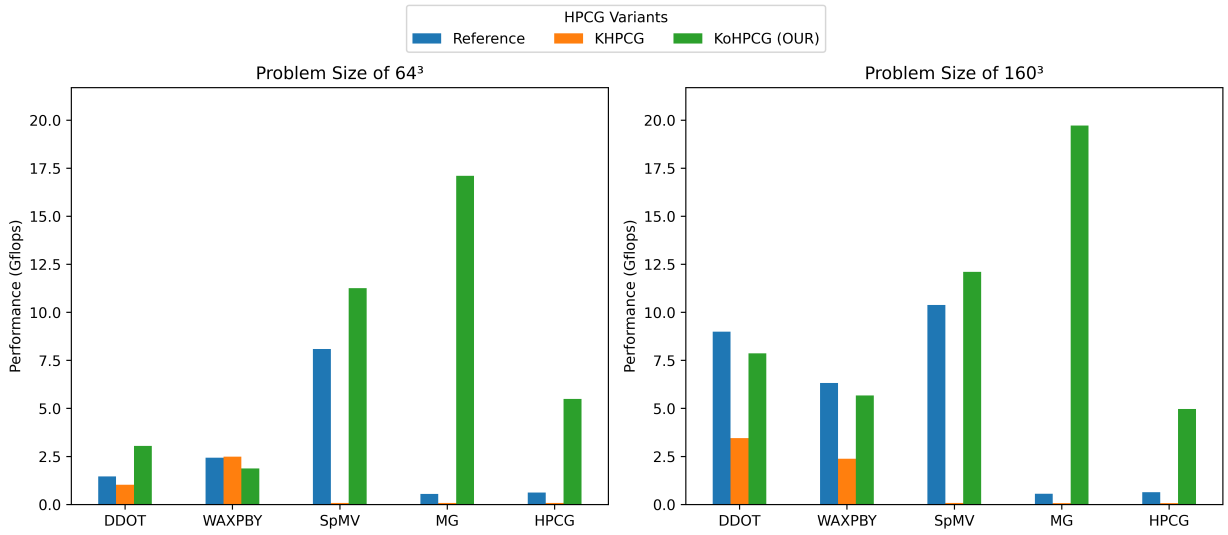

Fig. 1. Comparison of performance (Gflops) across different HPCG variants for problem sizes 64^3 and 160^3, on single node and 1 MPI process with 68 threads. The **Reference** version corresponds to the original HPCG v3.1 implementation. **KHPCG** [18] is an early Kokkos-based but it is limited in scalability and stability. **KoHPCG (OUR)** is our Kokkos-based optimized version of HPCG with the kernels (DDOT, WAXPBY, SpMV, SymGS, and MG) ported to the Kokkos programming model, designed for performance portability.

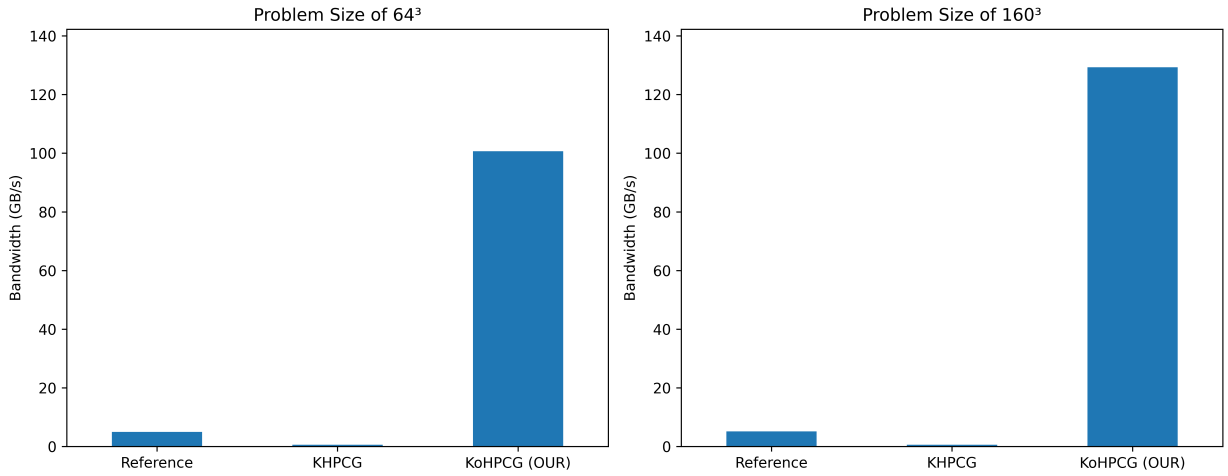

Fig. 2. Comparison of memory bandwidth (GB/s) across different HPCG variants for problem sizes 64^3 and 160^3, on single node and 1 MPI process with 68 threads.

IV. Experiments and Results

A. Experimental Setup

By using Kokkos and Kokkos Kernels, we developed the Kokkos-based variant of HPCG, named KoHPCG. We configured the Kokkos and Kokkos Kernels packages using Trilinos with CMake 3.26.2 and C++17 with Intel MKL, MPI, and OpenMP support.

We conducted experiments on two different hardware platforms on the Korean Supercomputer Nurion system [24], Intel Xeon Phi 7250 (KNL) and Intel Xeon Gold 6148 (SKL) processors. All experiments employed an MPI+OpenMP parallelization strategy. The number of OpenMP threads assigned per MPI process was determined using the relation:

$$\text{OpenMP Threads per MPI Process} = \frac{\text{Total Number of Cores}}{\text{Number of MPI Processes}}.$$

The KNL processor features 68 cores per node, along with 96 GB DDR4 and 16 GB high-bandwidth memory (HBM). The SKL processor consists of 40 cores per node with 192 GB of memory. For example, when two MPI processes are launched on KNL and SKL, each process is assigned 34 and 20 OpenMP threads, respectively. This threads per MPI process mapping scheme was consistently applied across all test cases unless stated otherwise.

We conducted experiments on 1, 2, 4, 8, and 16 nodes. In the results, the notation TP:40_N:2_MN:20_T:3 signifies that the result pertains to 20 MPI processes with 3 OpenMP

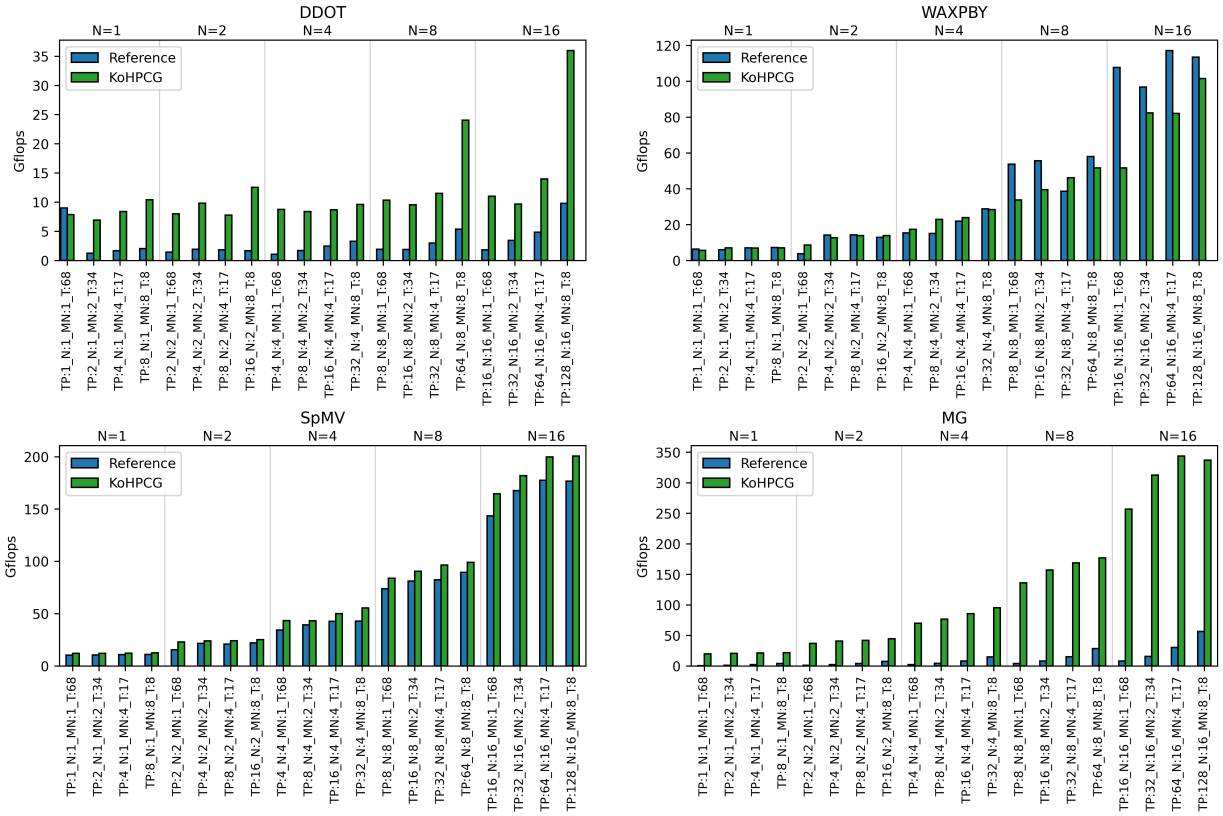

Fig. 3. Comparison of Gflops performance for DDOT, WAXPBY, SpMV and MG on Intel KNL for a problem size of 160^3 across different multi-node configurations. Each configuration is represented in the format `TP:X_N:Y_MN:Z_T:W`, where `TP` signifies the total number of MPI processes, `N` indicates the number of nodes, `MN` represents the number of MPI processes per node, and `T` denotes the number of OpenMP threads per process. The notation `TP:64_N:2_MN:32_T:2` signifies 64 MPI processes allocated over 2 nodes, with 32 MPI processes per node and 2 OpenMP threads per process. Results are sorted in ascending order based on node count and MPI processes per node from left to right.

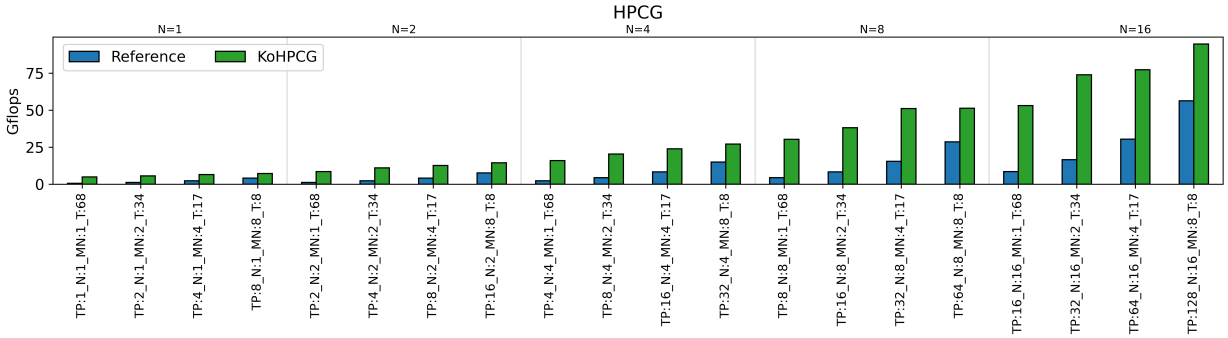

Fig. 4. Comparison of Gflops performance for HPCG on Intel KNL for a problem size of 160^3 across different multi-node configurations.

threads per process across 2 nodes, totaling 40 MPI processes. TP denotes the total MPI processes, N signifies the number of nodes, MN represents the MPI processes per node, and T refers to the OpenMP threads utilized per process. The results are organized in ascending order based on the number of nodes and the MPI processes per node.

B. Results on Knights Landing (KNL)

We evaluated the performance of HPCG variants as shown in Fig. 1, Reference, KHPCG, and KoHPCG (OUR), focusing on kernels performance in Gflops, and overall HPCG performance, on problem sizes 64^3 and 160^3 on KNL single node and 1 MPI process with 68 threads. Evaluated on just 1 MPI process because of the limitation of KHPCG for fair performance comparison. The Reference version, representing the original HPCG v3.1, shows very bad MG performance, under 0.6 Gflops, because of the SymGS performance, which gives a low total HPCG performance of 0.62 Gflops. KHPCG shows a significant performance drop, especially in SpMV and MG, where it falls below 0.08 Gflops, and also the low HPCG performance of 0.08 Gflops, indicating it as ill-suited for the evaluated architecture. Fig. 2 shows limited bandwidth use by KHPCG in GB/s.

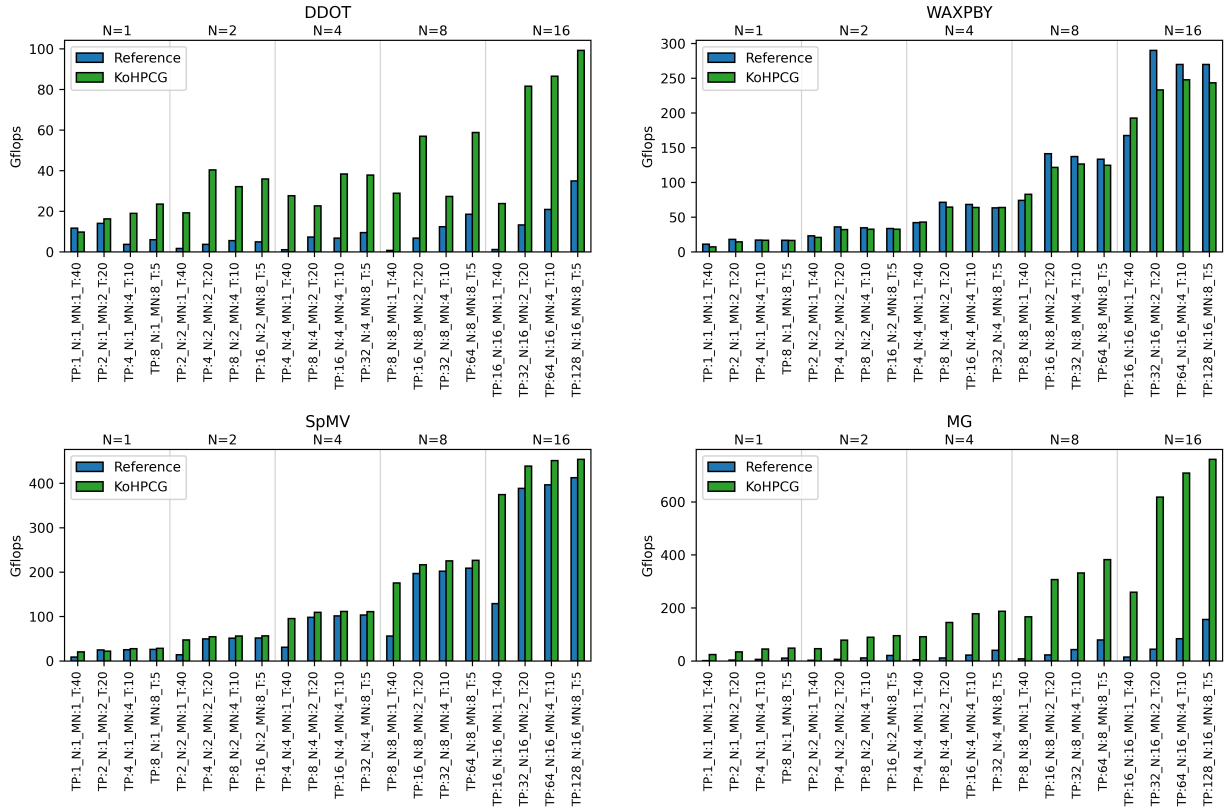

Fig. 5. Comparison of Gflops performance for DDOT, WAXPBY, SpMV and MG on Intel SKL for a problem size of 192^3 across different multi-node configurations. Each configuration is represented in the format `TP:X_N:Y_MN:Z_T:W`, where `TP` signifies the total number of MPI processes, `N` indicates the number of nodes, `MN` represents the number of MPI processes per node, and `T` denotes the number of OpenMP threads per process. The notation `TP:40_N:4_MN:10_T:4` signifies 40 MPI processes allocated over 4 nodes, with 10 MPI processes per node and 4 OpenMP threads per process. Results are sorted in ascending order based on node count and MPI processes per node from left to right.

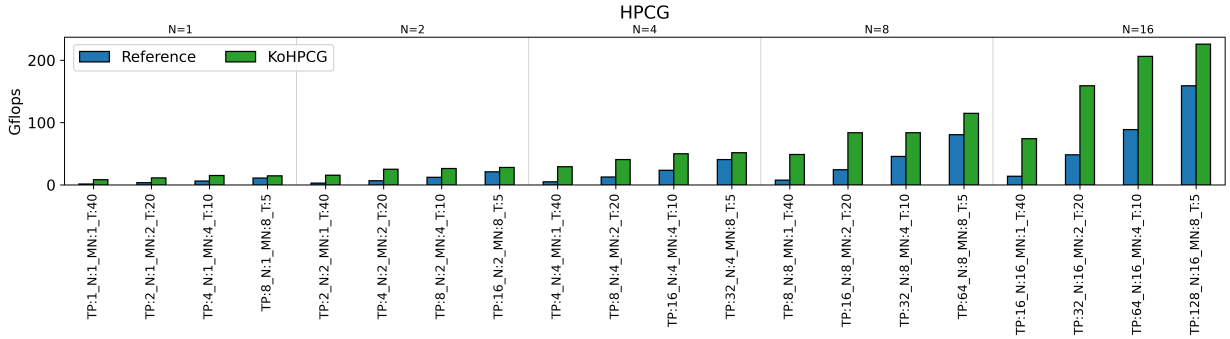

Fig. 6. Comparison of Gflops performance for overall HPCG on Intel SKL for a problem size of 192^3 across different multi-node configurations.

Our Kokkos-based implementation, KoHPCG, outperforms other variants, especially in MG performance and the performance drop recorded in WAXPBY. MG peaks at 19.7 Gflops, with 5.49 Gflops on 64^3 and 4.97 Gflops on 160^3 in HPCG overall performance, KoHPCG performs better in overall HPCG and achieves memory bandwidth utilization over 129 GB/s. These results highlight the efficacy of the Kokkos programming model for performance portability and the major impact of the SymGS kernel in HPCG performance.

The Figs. 3 and 4 shows a comparative performance analysis of Reference HPCG and KoHPCG for computational kernels (DDOT, WAXPBY, SpMV, and MG), and the overall HPCG benchmark, respectively, quantified in Gflops across multi-node different configurations. HPCG required that at least one fourth memory of the system should be used, so on KNL, we evaluated the results on 160^3 problem size on multi-node environment for 1, 2, 4, 8 and 16 nodes for different MPI + OpenMP settings. The graphs illustrate the significant performance improvements of KoHPCG over the Reference implementation.

The performance improvement of KoHPCG relative to the Reference HPCG for a problem size of 160^3 on Intel KNL demonstrates the improvement across the computational kernels. The MG exhibits performance improvement of

17.3×, whereas SpMV, although a critical and generally memory-bound kernel, demonstrates a relatively modest improvement of 1.17×. Similarly, the WAXPBY kernel exhibits negligible improvement in some configurations, but on 8 and 16 nodes, its performance dropped, indicating that it is further required to optimize. The MG consistently surpasses other kernels in improvement, with DDOT and overall HPCG demonstrating significant improvement. The little and no performance improvement of SpMV and WAXPBY suggests that further optimization may be required. KoHPCG provides a substantial and scalable performance improvement compared to the Reference implementation of HPCG.

C. Results on Skylake Scalable Processor (SKL)

Figs. 5 and 6 shows a performance comparison of Reference HPCG and KoHPCG for DDOT, WAXPBY, SpMV, MG, and the overall HPCG, respectively, on problem size of 192^3, For the DDOT kernel, the Reference implementation reaches a peak of around 66.5 Gflops on 16 nodes, whereas KoHPCG achieves 99.29 Gflops. The performance improvement increases with the number of processes, due to the use of the `kokkosBlas::dot` operation, which internally leverage highly optimized BLAS routine and offers good scalability. In contrast, WAXPBY performance in KoHPCG is slightly lower at larger process counts, indicating the need for further optimization in this kernel. For the SpMV kernel, KoHPCG consistently outperforms the Reference, with small improvement in performance. The most significant improvements are observed in the MG kernel, where KoHPCG demonstrates speedups of up to 3.2×, particularly at higher node counts such as `TP:128_N:16`, where performance increases from 156.5 Gflops to 760.1 Gflops. For the overall HPCG performance, KoHPCG continues to deliver substantial improvements across different configurations, often doubling the performance of the Reference implementation.

The parallelization strategy for SymGS significantly reduces data dependency bottlenecks, but it experiences additional overhead from increased use of memory. The current implementation of KoHPCG requires additional memory to transform the original HPCG data structures into Kokkos-compatible formats. The original data structures must remain unaltered to ensure appropriate benchmarking of the reference implementations of the core computational kernels. Duplicate data structures are required. This limitation affects our capacity to execute KoHPCG with increased MPI process counts on larger problem sizes due to memory constraints of the system.

This is ongoing work, and future efforts will focus on improving memory efficiency and extending evaluation to GPU-based systems. Our evaluation revealed that systems with just 96 GB of DDR memory performing better than the system which using an extra 16 GB of High Bandwidth Memory (HBM) in HPCG performance. Although the higher bandwidth of HBM, did not yield improved performance for memory bound operations such as SpMV and SymGS. However, the additional memory facilitated execution on a slightly higher number of MPI processes.

The performance of the WAXPBY kernel is suboptimal in some configurations, indicating a need for additional tuning. So far, KoHPCG has been primarily tested on Intel architectures, comprehensive validation on diverse platforms, including GPUs, is part of our planned future work to evaluate its full portability and performance potential.

V. Conclusion

This study presented KoHPCG, an optimized and performance-portable implementation of the HPCG benchmark utilizing the Kokkos programming model. KoHPCG addresses significant performance bottlenecks, especially in the Symmetric Gauss-Seidel (SymGS) operation, by parallelizing inherently sequential algorithm through using simple extra buffer vector. Experimental evaluations on Intel Xeon Phi (KNL) and Skylake (SKL) architectures demonstrated substantial performance improvement compared to the original reference HPCG implementation. The results highlighted the capabilities of Kokkos to provide performance optimization. In MG and DDOT kernels, we gain substantial performance improvements. KoHPCG demonstrates effective memory bandwidth utilization, affirming its utility as a benchmark tool that accurately reflects reasonable high-performance computing workloads. This work is on going and in near future we will configure Kokkos for GPU based system and evaluate KoHPCG performance on GPUs by setting up DefaultExecutionSpace to Cuda and MemorySpace to CudaSpace or CudaUVMSpace.

Data and Code Availability

The source code used in this study has been made publicly available to facilitate reproducibility and further research. The repository is accessible at [25]. Researchers are encouraged to use, adapt, and extend the code.

Acknowledgment

This work was partially supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government (MSIT) (RS-2023-00321688), and partially supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government (MSIT) (No.RS-2025-24534879). Also this work was supported by the National Supercomputing Center with supercomputing resources including technical support (No. TS-2024-RE-0029).

References

[1] J. J. Dongarra, "The linpack benchmark: An explanation," in *International Conference on Supercomputing*. Springer, 1987, pp. 456–474.

[2] J. Dongarra, P. Luszczek, and M. Heroux, "HPCG technical specification," *Sandia National Laboratories, Sandia Report SAND2013-8752*, 2013.

[3] J. J. Dongarra, P. Luszczek, and M. Heroux, "HPCG: one year later," *ISC14 Top500 BoF*, vol. 818, 2014.

[4] J. Dongarra, M. A. Heroux, and P. Luszczek, "High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems," *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 3–10, 2016.

[5] J. J. Dongarra, M. A. Heroux, and P. Luszczek, "A new metric for ranking high-performance computing systems," *National Science Review*, vol. 3, no. 1, pp. 30–35, 2016.

[6] Kokkos Team, "Kokkos: The manycore performance portability programming model," https://kokkos.org/, 2020, accessed: 2025-04-22.

[7] J. Park and M. Smelyanskiy, "Optimizing gauss–seidel smoother in hpcg," in *ASCR HPCG Workshop*, 2014.

[8] J. Park, M. Smelyanskiy, K. Vaidyanathan, A. Heinecke, D. D. Kalamkar, X. Liu, M. M. A. Patwary, Y. Lu, and P. Dubey, "Efficient shared-memory implementation of high-performance conjugate gradient benchmark and its application to unstructured matrices," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 945–955.

[9] E. Phillips and M. Fatica, "A cuda implementation of the high performance conjugate gradient benchmark," in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, 2014, pp. 68–84.

[10] Phillips and Fatica, "Performance analysis of the high-performance conjugate gradient benchmark on gpus," *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 28–38, 2016.

[11] Y. Liu, X. Zhang, C. Yang, F. Liu, and Y. Lu, "Accelerating HPCG on tianhe-2: a hybrid cpu-mic algorithm," in *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2014, pp. 542–551.

[12] Y. Liu, C. Yang, F. Liu, X. Zhang, Y. Lu, Y. Du, C. Yang, M. Xie, and X. Liao, "623 tflop/s HPCG run on tianhe-2: Leveraging millions of hybrid cores," *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 39–54, 2016.

[13] A. Zeni, K. O'Brien, M. Blott, and M. D. Santambrogio, "Optimized implementation of the HPCG benchmark on reconfigurable hardware," in *European Conference on Parallel Processing*. Springer, 2021, pp. 616–630.

[14] R. Steiger, "HPCG for fpgas: A data-centric approach," B.S. thesis, ETH Zurich, 2022.

[15] Q. Zhu, H. Luo, C. Yang, M. Ding, W. Yin, and X. Yuan, "Enabling and scaling the HPCG benchmark on the newest generation sunway supercomputer with 42 million heterogeneous cores," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–13.

[16] Y. Ao, C. Yang, F. Liu, W. Yin, L. Jiang, and Q. Sun, "Performance optimization of the HPCG benchmark on the sunway taihulight supercomputer," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 15, no. 1, pp. 1–20, 2018.

[17] Z. Bookey, "Performance portable high performance conjugate gradients benchmark," All College Thesis, College of Saint Benedict/Saint John's University, 2016, accessed: 2025-03-14. [Online]. Available: https://digitalcommons.csbsju.edu/honors_thesis/12

[18] "KHPCG 3.0," https://github.com/zabookey/KHPCG3.0, 2016, accessed: 2025-02-19.

[19] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of parallel and distributed computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[20] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez *et al.*, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2021.

[21] C. Ulmer, "Sand2021-1220: Benchmarking the nvidia a100 graphics processing unit for high-performance computing and data analytics workloads," Sandia National Laboratories, Tech. Rep., 2021, accessed: 2024-08-14. [Online]. Available: https://www.craigulmer.com/data/2021/SAND2021-1220_uur.pdf

[22] "HPCG benchmark," https://www.hpcg-benchmark.org/, 2013, accessed: 2025-05-15.

[23] J. Dongarra, M. Heroux, and P. Luszczek, "HPCG benchmark," 2019, accessed: 2025-03-21. [Online]. Available: https://github.com/hpcg-benchmark/hpcg

[24] KISTI, "National supercomputing center," 2018, accessed 5 May 2025. [Online]. Available: https://www.ksc.re.kr/eng/resources/nurion

[25] M. Rizwan, "KoHPCG: Kokkos-based High Performance Conjugate Gradient Benchmark," GitHub repository, 2025, accessed: Aug. 19, 2025. [Online]. Available: https://github.com/MRizwanSoongsil/KoHPCG