Evaluation of Directive-Based GPU Programming Models on a Block Eigensolver with Consideration of Large Sparse Matrices

Fazlay Rabbi[1](✉), Christopher S. Daley[2], Hasan Metin Aktulga[1], and Nicholas J. Wright[2]

[1] Michigan State University, East Lansing, MI 48823, USA
{rabbimd,hma}@msu.edu

[2] Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA
{csdaley,njwright}@lbl.gov

Abstract. Achieving high performance and performance portability for large-scale scientific applications is a major challenge on heterogeneous computing systems such as many-core CPUs and accelerators like GPUs. In this work, we implement a widely used block eigensolver, Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG), using two popular directive based programming models (OpenMP and OpenACC) for GPU-accelerated systems. Our work differs from existing work in that it adopts a holistic approach that optimizes the full solver performance rather than narrowing the problem into small kernels (*e.g.*, SpMM, SpMV). Our LOPBCG GPU implementation achieves a 2.8×–4.3× speedup over an optimized CPU implementation when tested with four different input matrices. The evaluated configuration compared one Skylake CPU to one Skylake CPU and one NVIDIA V100 GPU. Our OpenMP and OpenACC LOBPCG GPU implementations gave nearly identical performance. We also consider how to create an efficient LOBPCG solver that can solve problems larger than GPU memory capacity. To this end, we create microbenchmarks representing the two dominant kernels (inner product and SpMM kernel) in LOBPCG and then evaluate performance when using two different programming approaches: tiling the kernels, and using Unified Memory with the original kernels. Our tiled SpMM implementation achieves a 2.9× and 48.2× speedup over the Unified Memory implementation on supercomputers with PCIe Gen3 and NVLink 2.0 CPU to GPU interconnects, respectively.

Keywords: Sparse solvers · Performance optimization · Performance portability · Directive based programming models · OpenMP 4.5 · OpenACC

Electronic supplementary material The online version of this chapter (https://doi.org/10.1007/978-3-030-49943-3_4) contains supplementary material, which is available to authorized users.

1 Introduction

There is a pressing need to migrate and optimize applications for execution on GPUs and other accelerators. Future planned systems for the Department of Energy Office of Advanced Scientific Computing Research (DOE ASCR) include Perlmutter at NERSC (AMD CPU + NVIDIA GPU nodes), Aurora at ALCF (Intel CPU + Intel Xe accelerator nodes), and Frontier at OLCF (AMD CPU + AMD GPU nodes). The full capability of these systems can only be realized by making efficient use of the accelerators on the compute nodes. Most efforts to use accelerators to date have involved scientists using the CUDA programming language to target NVIDIA GPUs. The success of these efforts, the expected marginal gains in general-purpose CPU performance, and the understanding that special purpose accelerators are the best way to obtain significant performance gains within a fixed financial and power budget convinced DOE ASCR to invest in accelerator-based systems. However, CUDA alone is not an appropriate method to target accelerators produced by different vendors, e.g. NVIDIA, AMD, Intel, Xilinx, although there are efforts by AMD to use the HIP framework to convert CUDA to a more portable style of C++ [4].

In recent years, OpenACC and OpenMP have emerged as portable, base-language independent, and an increasingly robust and performant way to target accelerators. These directive-based methods have lowered the barrier of entry for application developers to target accelerators and are anticipated to be a key enabler for DOE users to efficiently use forthcoming supercomputers. However, there needs to be wider testing of OpenMP and OpenACC in scientific applications to address any shortcomings in the language specifications, improve the robustness and performance of vendor compilers, and continue to refine our understanding of best practices to migrate applications to accelerators. At the same time, the most efficient way to use accelerators is often achieved using optimized math and scientific libraries, e.g. cuBLAS and Tensorflow. Therefore, it will frequently be the case that non-trivial applications will increasingly need to mix optimized library calls with directives to obtain highest performance for the full application.

In this paper, we port and optimize a block eigensolver for GPUs using a combination of directives and optimized library calls. Sparse matrix computations (in the form of eigensolvers and linear solvers) are central to several applications in scientific computing and data analytics, from quantum many-body problems to graph analytics to machine learning. In the context of eigensolvers, performance of traditional sparse matrix-vector multiplication (SpMV) based methods are essentially limited by the memory system performance [33]. As such, block solver alternatives that rely on higher intensity operations such as sparse matrix-matrix multiplication (SpMM) and multiplication of vector blocks (*i.e.*, tall skinny matrices) have garnered the attention of several groups [7,29]. We adopt the Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG) [19,20] algorithm to represent block eigensolvers. Given that LOBPCG is a relatively popular method and requires a fairly complex implementation, it represents a suitable choice for our purposes.

An important issue in large scientific computing and data analysis workloads is that applications' data usage often exceeds the available device memory space. For instance, Many Fermion Dynamics - nuclei (MFDn), which is a quantum many-body code based on the configuration interaction model, is a "total memory-bound" application, *i.e.*, scientific studies using this code typically utilize all memory (DRAM) space available, thus easily exceeding the total device memory available [5,24]. As such, our evaluation extends into such scenarios and we present remedies for the significant performance degradations observed due to large data transfers between host and device memories.

Our contributions in this study can be summarized as follows:

– We demonstrate that a complex block eigensolver can be implemented efficiently using a mix of accelerator directives (in both OpenMP and OpenACC frameworks) and optimized library functions. We obtain up to a 4.3× speedup over a well optimized CPU implementation.
– We show that the performance of the Unified Memory version of SpMM, the dominant kernel in LOBPCG, depends on the supercomputer used and apparently the underlying CPU to GPU interconnect, when application working set exceeds GPU memory capacity. We measure a 13.4× performance loss when migrating from a supercomputer with a PCIe Gen3 CPU to GPU interconnect to one with NVLink 2.0.
– We address the Unified Memory performance portability issue by tiling the dominant kernels in LOBPCG. This obtains the highest performance on both supercomputers which have different CPU to GPU interconnects.

The paper is organized as follows. In Sect. 2, we describe the related work on efforts to port LOBPCG solvers to GPUs, application experience using OpenMP and OpenACC directives, and the use of Unified Memory to simplify porting applications to GPUs. In Sect. 3, we describe the kernel steps in the LOBPCG solver, the baseline OpenMP version of the LOBPCG solver including the library dependencies, and the steps we took to port the LOBPCG solver to GPUs. It also describes our tiling method for expensive kernels in the LOBPCG algorithm when a problem exceeds the GPU memory capacity. Finally, it describes the Cori-GPU and Summit platforms used to evaluate the performance of our directive based LOBPCG implementation and tiled microbenchmarks. In Sect. 4, we present performance results obtained on the Cori-GPU and Summit supercomputers. Section 5 discusses the key lessons and aims to provide advice for application developers based on our observations. Finally, Sect. 6 summarizes our conclusions and plans for future work.

2 Background and Related Work

Sparse Matrix Operations (SpMV/SpMM) on GPUs: Sparse matrix-vector multiplication (SpMV) and sparse matrix-matrix multiplication (SpMM) are the main kernels of many iterative solvers [19,22], machine learning techniques and other scientific applications. Several optimization techniques have been proposed for SpMV on GPUs [8,9,15,35]. However, performance of SpMV is bounded by memory bandwidth [33]. The main appeal of block eigensolvers (i.e. LOBPCG algorithm) is their high arithmetic intensity which is especially important to reap the full benefits of GPUs. The main computational kernels involved in block iterative solvers are the multiplication of a sparse matrix with multiple vectors and level-3 BLAS operations on dense vector blocks. Optimizing the SpMM kernel on GPUs has been studied in several research works. Yang et al. [34] propose two novel algorithms for SpMM operation on GPUs that take the sparse matrix input in compressed-sparse-row (CSR) format and focus on latency hiding with instruction-level parallelism and load-balancing. They find out a memory access pattern that allows efficient access into both input and output matrices which is the main enabler for their excellent performance on SpMM. A common optimization strategy of SpMM is to rely on a special sparse matrix representation to exploit the nonzeros efficiently. Most commonly used sparse matrix storage variants other than CSR format are ELLPACK called ELLPACK-R [27] and a variant of Sliced ELLPACK called SELL-P [7]. Hong et al. [16] separates the sparse matrix into heavy and light rows in order to perform dynamic load-balancing. They process the heavy rows by CSR and the light rows by doubly compressed sparse row (DCSR) in order to take advantage of tiling. However, these special matrix storage formats incur some additional computational and format conversion cost in the full computational pipeline.

Anzt et al. [7] optimize the performance of SpMM using ELLPACK format [6] and compare the performance of their CPU-GPU implementation with the multithreaded CPU implementation of LOBPCG provided in the BLOPEX [21] package. All of their kernels were written in CUDA 5.5 and they evaluated the performance experiment on two Intel Sandy Bridge CPUs and one NVIDIA K40 GPU. Dziekonski et al. [13] implement LOBPCG method with an inexact nullspace filtering approach to find eigenvalues in electromagnetics analysis.

Most of the prior work focused on optimizing either the SpMV or the SpMM operation on GPUs with the ultimate goal of accelerating the iterative solver used in a scientific application. A distinguishing aspect of this paper is that we adopt a holistic approach that includes all computational kernels required for the LOBPCG solver. We use directive based programming models to achieve portability. We also investigate the scenario where the total memory footprint exceeds the device memory capacity and propose a solution that addresses performance degradations seen with NVIDIA's generic "Unified Memory" approach (see below).

OpenMP/OpenACC: OpenMP and OpenACC are two directive-based methods to parallelize serial applications. Both languages enable a programmer to run application kernels on a GPU. Multiple compilers support these directives and can generate GPU code. The quality of GPU support in OpenMP and OpenACC compilers is evaluated in [23] on a suite of 4 mini applications. Here, the authors find issues with all compilers as well as challenges in creating a single portable code which compiles and executes efficiently for all compilers. The interoperability of CUDA and OpenACC is evaluated in [32]. The author successfully combines hand-written CUDA with OpenACC when using the PGI compiler. Our work evaluates the performance of OpenMP and OpenACC implementations of a block eigensolver, as well the interoperability of these runtime systems with optimized CUDA libraries for 3 different compilers.

Unified Memory: Unified Memory (UM) is a programming feature which provides a single memory address space accessible by all processors in a compute node. It greatly simplifies GPU programming because the same single pointer to data can be used on both CPU and GPU. The NVIDIA Volta V100 GPU provides a page migration engine to move memory pages between CPU and GPU when the page is not in the memory of the processor accessing the data. NVIDIA evaluated UM performance using the PGI OpenACC compiler in [12]. The authors created UM versions of the OpenACC applications in the SPEC ACCEL 1.2 benchmark suite. They ran the applications on the Piz-Daint supercomputer and found that the UM versions ran at 95% of the performance of the original explicit data management versions. In [28], the NVIDIA presenter shows that the Gyrokinetic Toroidal Code (GTC) has almost identical performance on a x86+V100 system whether OpenACC data directives are used or not. Our work also compares UM against explicit data management, but additionally considers problems whose memory requirements are significantly over the device memory capacity. The performance of oversubscribing UM is evaluated in [18]. The authors find that UM can be up to 2× slower than explicit data management in several applications on an x86+V100 system. Our work considers performance on both x86 and Power GPU-accelerated systems.

3 Methodology

In this section, we provide an overview of the LOBPCG algorithm, our baseline CPU implementation, and the steps we took to port and optimize the CPU implementation to run efficiently on GPU-accelerated systems using OpenMP and OpenACC. We then describe our pathfinding activities for creating an efficient LOBPCG algorithm which can operate on matrices exceeding the device memory capacity. In particular, we discuss how we tiled the two most expensive kernels in LOBPCG and created microbenchmarks that enable performance comparison of programmer-controlled and system-controlled (i.e. Unified Memory) data movement schemes between the CPU and GPU. Finally, we describe the experimental platforms used for evaluating the performance of our LOBPCG and microbenchmark implementations on GPU-accelerated systems.

3.1 The LOBPCG Algorithm

LOBPCG is a commonly used block eigensolver based on the sparse matrix multiple vector multiplication kernel [19]. It is designed to find a prescribed number of the largest (or smallest) eigenvalues and the corresponding eigenvectors of a

Algorithm 1: LOBPCG Algorithm (for simplicity, without a preconditioner) used to solve $\hat{H}\Psi = E\Psi$

```
   Input: $\hat{H}$ , matrix of dimensions $N \times N$
   Input: $\Psi_0$, a block of randomly initialized vectors of dimensions of $N \times m$
   Output: $\Psi$ and $E$ such that $\|\hat{H}\Psi - \Psi E\|_F$ is small, and $\Psi^T\Psi = I_m$
1  Orthonormalize the columns of $\Psi_0$
2  $P_0 \leftarrow 0$
3  for $i$ = 0, 1, ..., until convergence do
4      $E_i = \Psi_i^T \hat{H} \Psi_i$
5      $R_i \leftarrow \hat{H}\Psi_i - \Psi_i E_i$
6      Apply the Rayleigh–Ritz procedure on span$\{\Psi_i, R_i, P_i\}$
7      $\Psi_{i+1} \leftarrow \underset{S \in \text{span}\{\Psi_i . R_i, P_i\},\ S^T S = I_m}{\text{argmin}} \text{trace}(S^T \hat{H} S)$
8      $P_{i+1} \leftarrow \Psi_{i+1} - \Psi_i$
9      Check convergence
10 end
11 $\Psi \leftarrow \Psi_{i+1}$
```

symmetric positive definite generalized eigenvalue problem $H\Psi = EB\Psi$ for a given pair (H, B) of complex Hermitian or real symmetric matrices, where the matrix B is also assumed positive-definite. Here, E is a diagonal matrix of the sought eigenvalues and Ψ is the corresponding block of eigenvectors. Algorithm 1 shows the pseudocode of the LOBPCG algorithm for the standard eigenvalue problem $H\Psi = E\Psi$. LOBPCG comprises high arithmetic intensity operations (SpMM and Level-3 BLAS). In terms of memory, while the $\widehat{H}$ matrix takes up considerable space, when a large number of eigenpairs are needed (e.g., in dimensionality reduction, spectral clustering or quantum many-body problems), memory needed for the block vector Ψ can be comparable to or even greater than that of $\widehat{H}$. In addition, other block vectors (residual R, preconditioned residual W, previous direction P), block vectors from the previous iteration and the preconditioning matrix T must be stored (not shown in Algorithm 1 for simplicity), and accessed at each iteration.

3.2 Baseline CPU Implementation

We implemented the baseline CPU version of LOBPCG using OpenMP and OpenACC directives in C/C++. We adopted the Compress Sparse Row (CSR) format to store the sparse matrix and used the `mkl_dcsrmm` routine from Intel MKL library for the SpMM kernel. We also implemented a custom SpMM kernel in both OpenMP and OpenACC, again based on the CSR format, and used it with the PGI and IBM compilers. For all LAPACK and BLAS routines needed, we used Intel MKL, the PGI-packaged LAPACK and BLAS libraries, and IBM ESSL for Intel, PGI and IBM compilers, respectively.

3.3 A GPU Implementation of LOBPCG

The most expensive kernels in the baseline CPU version are the SpMM operation and the inner product of vector blocks ($X^T Y$). The cuSPARSE [26] and cuBLAS CUDA libraries provide tuned versions of these kernels. We used `cusparseDcsrmm` for the SpMM operation and replaced `cblas_dgemm` routine with `cublasDgemm` for the vector block operations. We allocated the device data for these routines using `cudaMalloc`. We ported the remaining application kernels using OpenMP and OpenACC offloading pragmas. The application kernels are grouped together inside device data regions to avoid data movement between successive application kernels. However, the performance of this implementation was still poor because significant time was spent moving data between CPU and GPU. This happened because the application and library kernels were operating on distinct data on the GPU.

OpenMP and OpenACC provide a clause to enable the application kernels to operate on data already resident on the device. The clause is named `is_device_ptr` in OpenMP and `deviceptr` in OpenACC. We used the pointer returned by `cudaMalloc` in our OpenACC implementation. This approach caused a run time error in our OpenMP implementation compiled with LLVM/-Clang. We therefore replaced `cudaMalloc` with `omp_target_alloc` in our OpenMP implementation because the OpenMP 5.0 specification [2] states that "Support for device pointers created outside of OpenMP, specifically outside of the `omp_target_alloc` routine and the `use_device_ptr` clause, is implementation defined.". Figure 1 shows an example of the structure of most of our application kernels after using this clause. It enabled us to remove multiple OpenMP/OpenACC data regions and thus considerable data movement between the CPU and GPU[1].

All kernels run on the GPU except for some LAPACK routines, i.e., `LAPACKE_dpotrf` and `LAPACKE_dsygv` which are not available in the CUDA toolkit math libraries. This causes 10 small matrices to move between CPU and GPU in each iteration of the LOBPCG method. As the sizes of those matrices are very small, we find that the overhead associated with these data movements are insignificant compared to the total execution time.

3.4 Tiling LOBPCG Kernels to Fit in GPU Memory Capacity

The LOBPCG GPU implementation described in Sect. 3.3 allocated the tall skinny matrices and the sparse matrix in GPU memory. This approach is limited to cases where the aggregated matrix memory footprint is less than the GPU memory capacity. However, a major challenge in many scientific domains [5,25,30] (such as configuration interaction in MFDn) is the massive size of the sparse

[1] Alternatively, we could have copied the data to the device using OpenMP/OpenACC and then passed the device pointer to the CUDA library functions using OpenMP's `use_device_ptr` clause or OpenACC's `use_device` clause. We did not use this approach because we wanted the option to use `cudaMallocManaged` to allocate data in managed memory.

Before using `is_device_ptr`

```
// d_R and other device arrays allocated with omp_target_alloc
cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, b, numrows, b,
  &cudaAlpha, d_lambda, b, d_X, b, &cudaBeta, d_R, b);

// Copy output array d_R to the host array R
omp_target_memcpy(R, d_R, R_size * sizeof(double), 0, 0, h, t);

// Copy host array R to the device in OpenMP target data region
#pragma omp target data map(tofrom: newX[0 : X_size])\
  map(to: X[0 : X_size], R[0 : R_size])
{
  mat_mult(X, R, newX, numrows, b);
}

void mat_mult(double *src1, double *src2, double *dst,
              int row, int col)
{
#pragma omp target teams distribute parallel for collapse(2)
  for(int i = 0; i < row ; i++)
    for(int j = 0 ; j < col ; j++)
      dst[i * col + j] = src1[i * col + j] * src2[i * col + j];
}
```

After using `is_device_ptr`

```
// d_R and other device arrays allocated with omp_target_alloc
cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, b, numrows, b,
  &cudaAlpha, d_lambda, b, d_X, b, &cudaBeta, d_R, b);

// Pointers to device arrays passed into mat_mult function
mat_mult(d_X, d_R, d_newX, numrows, b);

void mat_mult(double *src1, double *src2, double *dst,
              int row, int col)
{
  // Use is_device_ptr because data is already on the device
#pragma omp target is_device_ptr(src1, src2, dst)
#pragma omp teams distribute parallel for collapse(2)
  for(int i = 0; i < row ; i++)
    for(int j = 0 ; j < col ; j++)
      dst[i * col + j] = src1[i * col + j] * src2[i * col + j];
}
```

Fig. 1. The use of `is_device_ptr` to avoid memory copies. Error checking is omitted for brevity.

matrix, which can have several billions of rows and columns and the total number of nonzeros can easily exceed trillions. In this subsection, we explain how we tiled the SpMM and inner product kernels (X^TY) to operate on problems larger than the GPU memory capacity. We extracted each kernel into a standalone microbenchmark to check for correctness and enable performance evaluation. Although not described in this paper, we have also implemented and evaluated the linear combination kernel (XY) which has similar characteristics to the inner product kernel (X^TY), but involves the multiplication of a tall-skinny vector block (X) with a small square matrix (Y).

SpMM Kernel: The SpMM kernel is typically the most expensive operation in LOBPCG. Figure 2 shows the tiling idea for the SpMM kernel for cases when the LOBPCG data is too large to fit into the GPU memory. For a given tile size β, we divide the sparse matrix into block of rows. Algorithm 2 describes the steps in our tiled SpMM kernel. In short, we copy the Y matrix to the GPU at the beginning and it resides there until all sparse matrix tiles are processed. Then, we extract the CSR format of each of the tiles and copy that to GPU memory. Then we apply the `cusparseDcsrmm` routine on the sparse matrix block and Y.

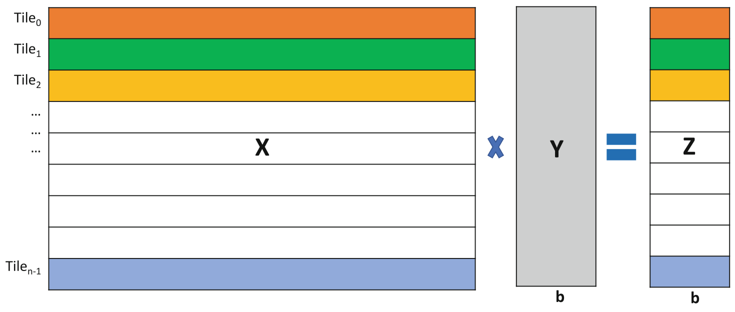

Fig. 2. Overview of tiling SpMM operation.

Algorithm 2: Tiled SpMM (`cusparseDcsrmm`) kernel

```
  Input: X($m \times m$) sprase matrix in CSR format (val, rowPtr, colIndex),
         Y($m \times b$), $\beta$(tile size)
  Output: Z($m \times b$)
1 $nrowblk = \lceil \frac{m}{\beta} \rceil$
2 for $i = 0$ to $nrowblk$ - 1 do
      // extract_CSR_tile() method extracts the CSR format of the i-th
         tile from the given sparse matrix
3     [rowPtrTile, colIndxTile, valTile, nnz_Tile] = extract_CSR_tile(val, rowPtr,
      colIndex, i)
4     cusparseDcsrmm($\beta$, b, m, nnz_tile, 1.0, valTile, rowPtrTile, colIndxTile, R,
      m, 0.0, AR, $\beta$)
5     cudaDeviceSynchronize()
6     cudaMemcpy(Z[i-th tile], AR, cudaMemcpyDeviceToHost)
7     cudaDeviceSynchronize()
8 end
```

This produces the corresponding row blocks of the final output matrix Z. After processing each tile, we copy back the partial output to the corresponding tile of the Z matrix.

Inner Product Kernel: One of the most frequently invoked and expensive kernels in LOBPCG is the inner product operation ($Z = X^T Y$) between two tall skinny matrices. Hence, a well performing tiled inner product kernel is crucial for large problem sizes. Figure 3 shows the overview of the matrix tiling idea for the inner product kernel. X and Y are of size $m \times b$ where $m \gg b$. Both matrices are partitioned into $n = \lceil \frac{m}{\beta} \rceil$ tiles. In our custom inner product kernel, we transfer each tile of X and Y from CPU to GPU and apply `cublasDgemm` routine on each tile. We keep accumulating the partial output to a $b \times b$ matrix on the GPU. After processing all tiles, we copy back the final result to Z. Algorithm 3 gives an overview of our custom inner product kernel.

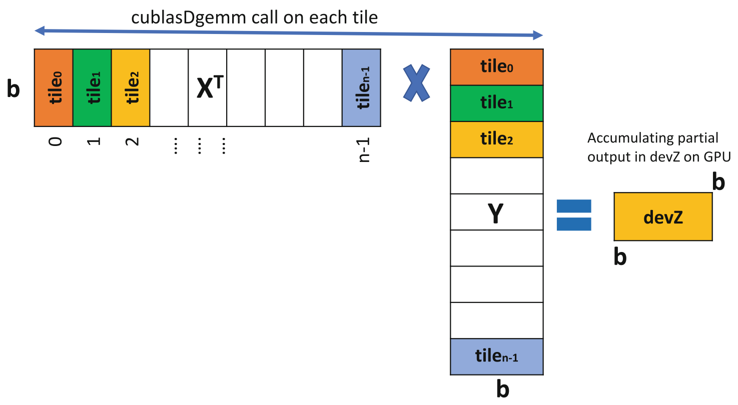
cublasDgemm call on each tile
b
tile0
tile1
tile2
XT
tilen-1
0
1
2
n-1
tile0
tile1
tile2
Y
tilen-1
b
Accumulating partial output in devZ on GPU
b
devZ
b

Fig. 3. Overview of tiling Inner Product kernel

Algorithm 3: Tiled Inner Product (`cublasDgemm`) Kernel

```
   Input: X($m \times b$), Y($m \times b$), $\beta$(tile size)
   Output: Z($b \times b$)
1  $nrowblk = \lceil \frac{m}{\beta} \rceil$
2  cudaMemset(devZ, 0.0, b*b*sizeof(b))
3  for $i$ = 0 to nrowblk - 1 do
4      cudaMemcpy(devX, X[i-th block], $\beta$ * b, cudaMemcpyHostToDevice);
5      cudaMemcpy(devY, Y[i-th block], $\beta$ * b, cudaMemcpyHostToDevice);
6      cudaDeviceSynchronize();
7      cublasDgemm(b, b, $\beta$, 1.0, devY, $\beta$, devX, $\beta$, 1.0, devZ, $\beta$);
8      cudaDeviceSynchronize()
9  end
10 cudaMemcpy(Z, devZ, b * b, cudaMemcpyDeviceToHost);
```

3.5 Hardware and Software Environment

We conducted all of our experiments on the Cori-GPU testbed at the National Energy Research Scientific Computing Center (NERSC) [1] and the Summit supercomputer at the Oak Ridge Leadership Computing Facility (OLCF) [3]. Cori-GPU is a Cray CS-Storm 500NX consisting of 18 compute nodes. Each compute node has two 20-core Skylake processors clocked at 2.4 GHz and 8 NVIDIA Tesla V100 "Volta" GPUs with 16 GBs of HBM per GPU. The V100 GPU model has a peak double precision performance of 7.0 TFLOP/s. There is a total of 384 GB DDR4 DRAM space on each node. The CPUs are connected to the GPUs via four PCIe 3.0 switches and the GPUs are connected to each other via NVIDIA's NVLink 2.0 interconnect. The Summit supercomputer is an IBM AC922 system consisting of 4608 compute nodes [31]. Each compute node has two 22-core IBM Power9 processors clocked at 3.1 GHz and 6 NVIDIA Tesla V100 "Volta" GPUs with 16 GBs of HBM per GPU. The V100 GPU model is based on the SXM2 form factor and has a peak double precision performance of 7.8

TFLOP/s. There is a total of 512 GB DDR4 DRAM space per node. Unlike Cori-GPU, the CPUs and GPUs in a Summit compute node are all connected with the high bandwidth NVLink 2.0 interconnect. This also provides cache coherence between CPUs and GPUs and enables system-wide atomics. The theoretical peak uni-directional bandwidth between 1 CPU and 1 GPU is 16 GB/s on Cori-GPU and 50 GB/s on Summit. However, the highest pageable bandwidth we measured from CPU to GPU was 5.2 GB/s on Cori-GPU and 25.0 GB/s on Summit.

The Cori-GPU and Summit supercomputers provide extensive software environments to compile OpenMP and OpenACC programs. Here, we list the software environment used in this paper. The software used on the Cori-GPU system were Intel Compiler v19.0.3 (OpenMP for CPU), LLVM/Clang compiler v9.0.0-git (OpenMP for GPU), and PGI compiler v19.5 (OpenACC for CPU and GPU). We used Intel MKL with the Intel and LLVM/Clang compilers and PGI's version of LAPACK with the PGI compiler. The GPU accelerated libraries were cuSPARSE and cuBLAS provided with CUDA v10.1.168. The software used on Summit were IBM XLC Compiler v16.1.1-3 (OpenMP for CPU and GPU) and PGI compiler v19.5 (OpenACC for CPU and GPU). We used IBM ESSL with the IBM XLC Compiler and PGI's version of LAPACK with the PGI compiler. Once again, the GPU accelerated libraries were cuSPARSE and cuBLAS provided with CUDA v10.1.168.

3.6 Experiments

In this section we explain the experiments conducted. The first set of experiments are used to evaluate the LOBPCG GPU implementation. The second set of experiments are used to evaluate our microbenchmarks on problems exceeding the GPU memory capacity.

Performance of the LOBPCG Solver: The CPU and GPU implementations of LOBPCG are evaluated using a series of real-world matrices with different sizes, sparsity patterns and application domains as shown in Table 1. The first 2 matrices are from the SuitSparse Matrix Collection [11] and the **Nm7** and **Nm8** matrices are extracted from two very large Hamiltonian matrices that arise in nuclear structure calculations with MFDn. Note that the test matrices have millions of rows and hundreds of millions of nonzeros. The memory footprint of these matrices vary from 2 GB to 7.8 GB using the CSR matrix format.

Table 1. Test matrices.

Matrix	Rows	Columns	Nonzeros	Size (GB)	Domain
Queen_4147	4,147,110	4,147,110	166,823,197	2.018	3D strctural problem
HV15R	2,017,169	2,017,169	283,073,458	3.405	Computational fluid dynamics
Nm7	4,985,422	4,985,422	647,663,919	7.792	MFDn
Nm8	7,579,303	7,579,303	592,099,416	7.136	MFDn

We measured the runtime of the LOBPCG CPU implementation on a single CPU socket on Cori-GPU and Summit nodes. The configurations used 1 thread per core and used the appropriate `slurm`, `jsrun` and OpenMP/OpenACC environment variables to bind the process and child threads. We did not use hyperthreading/SMT because our kernels are memory bandwidth bound. We measured the runtime of the LOBPCG GPU implementation on a single CPU socket and one GPU on Cori-GPU and Summit nodes. Our configurations only ever used a single CPU socket to avoid potential performance issues associated with non-uniform memory access time. We evaluated the compiler combinations described in Sect. 3.5 and measured runtime with application timers.

Performance of X^TY and SpMM Kernels for Large Matrices: Our next experiment evaluated the X^TY microbenchmark and SpMM microbenchmark on input problems exceeding GPU memory capacity on Cori-GPU and Summit. This experiment is designed to inform our future sparse solver implementations. We tested the tiled versions of the microbenchmarks so that we could easily separate how much time is spent in computation versus data movement between the CPU and GPU. If more time is spent in computation then data movement costs can potentially be hidden. In the X^TY microbenchmark, we chose to multiply two matrices of size $67,108,864 \times 48$ leading to a memory footprint of 51.54 GB. We set the tile size (β) to $131,072$ for the X^TY microbenchmark and $2,597,152$ for the SpMM microbenchmark as this gives us the best performance. The tile size (β) is an optimization parameter and one can vary it as long as the memory footprint required to process a single tile is less than GPU memory capacity. In the SpMM microbenchmark, we used a synthetic input matrix of 24 GB, leading to a memory footprint of 35.1 GB. The dimension of the synthetic sparse matrix is $14,957,833 \times 14,957,833$ with $1,946,671,770$ nonzeros. We multiplied this sparse matrix with a dense matrix of dimension $14,957,833 \times 48$. We used a multi-threaded RMAT graph generator [17] to generate our synthetic sparse matrix. We measured compute and data movement time using the `nvprof` profiler.

Performance of Tiled and Unified Memory Versions of SpMM: Our final experiment evaluated the Unified Memory version of the SpMM microbenchmark. The Unified Memory version was written in OpenACC and compiled with the PGI compiler and the compiler option `-ta:tesla:managed` to replace regular system memory allocations with managed memory allocations. We compared runtime against the tiled version of SpMM on Cori-GPU and Summit for two input matrices. The first input matrix is Nm7 (see Table 1) and leads to a microbenchmark memory footprint of 11.7 GB. The second input matrix is the synthetic sparse matrix ($14,957,833 \times 14,957,833$ with $1,946,671,770$ nonzeros) and leads to a microbenchmark memory footprint of 35.1 GB. The matrices are chosen to create problems less than GPU memory capacity and greater than GPU memory capacity. In both cases, we multiplied these sparse matrices with a dense matrix of 48 vector blocks. We set the tile size (β) to $2,597,152$ for both of matrices as it is the highest tile size that we can use without overflowing the

GPU memory and it gives the best performance. The `nvprof` profiler is used to collect compute time, data movement time, and Unified Memory data movement and page fault time.

4 Results

In this section we show performance results on the Cori-GPU and Summit supercomputers. Section 4.1 shows the performance of the CPU and GPU versions of LOBPCG when parallelized with either OpenMP or OpenACC. We then consider how we could use the LOBPCG solver on matrices larger than GPU memory capacity. Section 4.2 shows performance results when tiling the dominant X^TY and SpMM kernels so that each tile fits within GPU memory capacity. Finally, Sect. 4.3 compares the performance of the tiled implementation of the SpMM kernel against a naive Unified Memory implementation.

4.1 Performance of the LOBPCG Solver

We compared the performance of the LOBPCG solver when using a suite of different compilers. The compilers can all generate code for the host CPU and sometimes also for the GPU. In the following sentences, we place CPU or GPU in parenthesis to indicate whether we used the compiler to generate code for the CPU or GPU. The OpenMP compilers were Intel (CPU) and Clang (GPU) on Cori-GPU and IBM (CPU and GPU) on Summit. The OpenACC compiler was always PGI (CPU and GPU). In all cases we used a hand-written portable SpMM kernel except for our Intel compiler experiment which used `mkl_dcsrmm` from Intel MKL. We did this to obtain the best possible CPU time to more transparently show the value of our GPU implementation. The performance results for the Nm7 matrix are shown in Fig. 4. The execution time of the LOBPCG solver is averaged over 10 iterations.

The results show that the execution time of our GPU implementation is almost independent of directive based programming model and evaluation platform. Our reasoning is that the OpenMP and OpenACC configurations use the same GPU math libraries, the GPUs are nearly identical in Cori-GPU and Summit (different V100 models), and that our LOBPCG implementation has been highly tuned to minimize data movement between CPU and GPU. The best GPU performance is 3.05x faster than the best CPU performance for Nm7 matrix. The CPU versions show more variable performance for different combinations of compilers and math libraries used on Cori-GPU and Summit. The highest performance is obtained with the OpenMP version when compiled with Intel compiler on Cori-GPU. The performance differences can mostly be attributed to the host CPU and SpMM performance: `mkl_dcsrmm` is 1.4$\times$ faster than our hand-written SpMM kernel in OpenMP and the hand-written SpMM kernel is 1.5–3.0$\times$ faster when using OpenMP rather than OpenACC. We did not investigate the host CPU performance in any more detail because it is not the focus of our work.

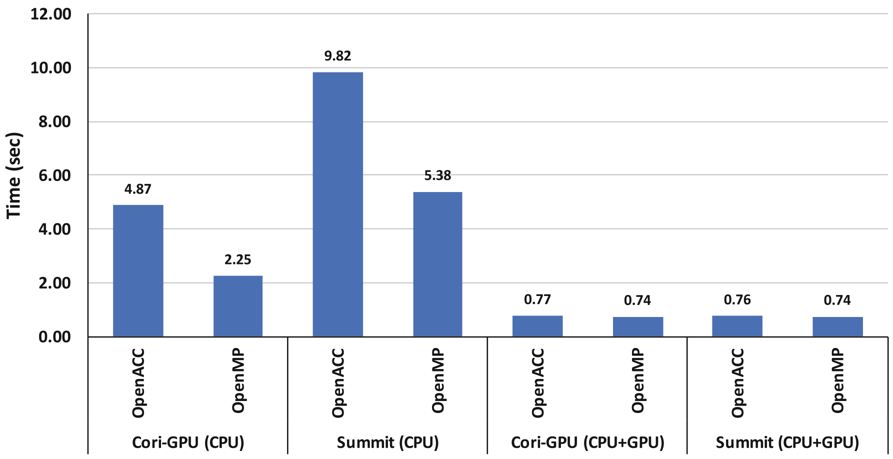

Fig. 4. The time spent in LOBPCG on Cori-GPU and Summit when using various compilers with either OpenMP or OpenACC

Figure 5 shows how time is spent in the best configurations on CPU and GPU when using the Nm7 matrix. Execution time is divided into library time, application kernel time, and unaccounted CUDA API time. The library time is spent in cuBLAS and cuSPARSE in the GPU implementation and Intel MKL in the CPU implementation. The application kernel time is spent in user defined functions in both the CPU and GPU implementations. The CUDA API time includes GPU data allocation and data movement between CPU and GPU and is calculated by subtracting time spent in application and library kernels from the total run time. The library and application kernels speedup by 3.7× and 5.0×, respectively, when using GPUs. Application kernel time is a relatively small fraction of total run time on GPU. However, the offload is a key optimization step needed to keep total run time low. Total run time would be significantly slower if we decided to use host application kernels because of unnecessary data movement between CPU and GPU.

Figure 6 shows GPU speedup over the best LOBPCG CPU implementation for all the test matrices in Table 1. The LOBPCG GPU implementation achieves 2.8×–4.3× speedup over the best CPU implementation. The GPU implementation therefore performs well over a range of matrices from different domains with different sparsity patterns.

4.2 Performance of X^TY and SpMM Kernels for Large Matrices

Figure 7 shows the time spent in the inner product (X^TY) kernel on Cori-GPU and Summit when total memory footprint is 51.54 GB. The tile size is 131,072. The total time is divided into host-to-device (HtoD) data transfer time and computation time in the inner product kernel (device-to-host (DtoH) data transfer times are negligible for this kernel). We measured data transfer and computa-

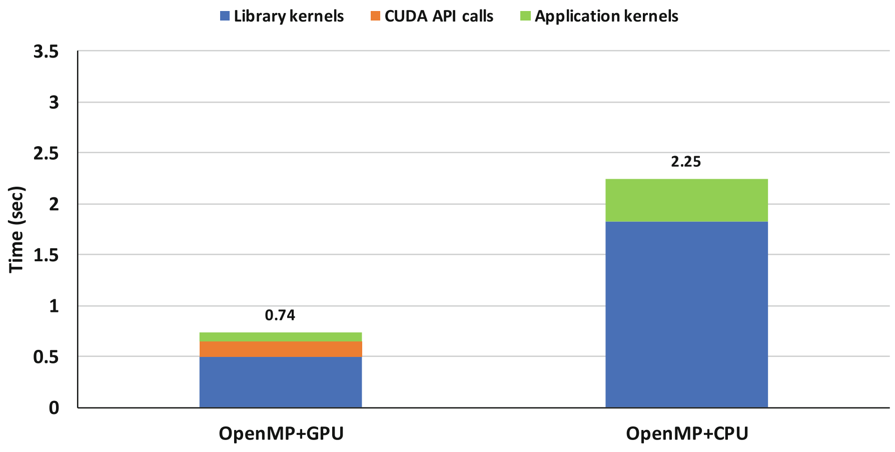

Fig. 5. The time spent in LOBPCG on Cori-GPU when using matrix Nm7

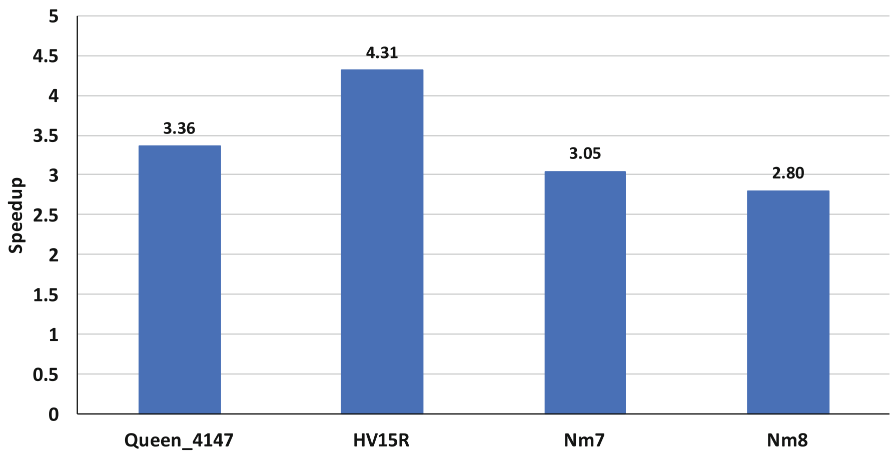

Fig. 6. LOBPCG GPU speedup on Cori-GPU for each test matrix

tion time using `nvprof`. The results show that total run time is dominated by data transfers. Run time is lower on Summit because of the high bandwidth NVLink 2.0 interconnect. We obtained data transfers of 4 GB/s on Cori-GPU and 13 GB/s on Summit in this kernel. Results indicate that data transfer time cannot be hidden behind computation when the matrix exceeds the GPU memory capacity.

Figure 8 shows the time spent in the SpMM kernel. The input sparse matrix is 24 GB and the total memory footprint is 35.1 GB. This time, results show that computation time is greater than the data movement time. This indicates that data movement time could be completely hidden behind computation. It would therefore be possible to obtain nearly the same computational throughput as one

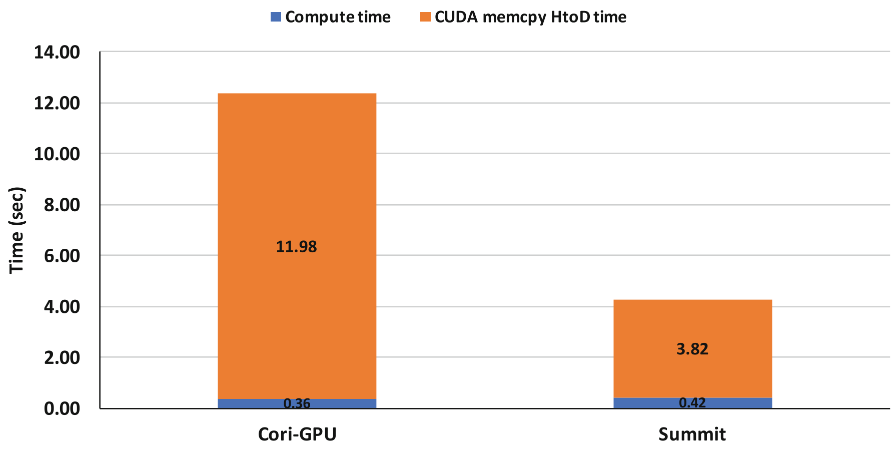

Fig. 7. Time spent in X^TY kernel on Cori-GPU and Summit when the memory footprint exceeds GPU memory capacity.

would get using matrices completely resident in the GPU memory. However, an actual block eigensolver alternates between SpMM and vector block operations, so this may not be easy to realize in practice.

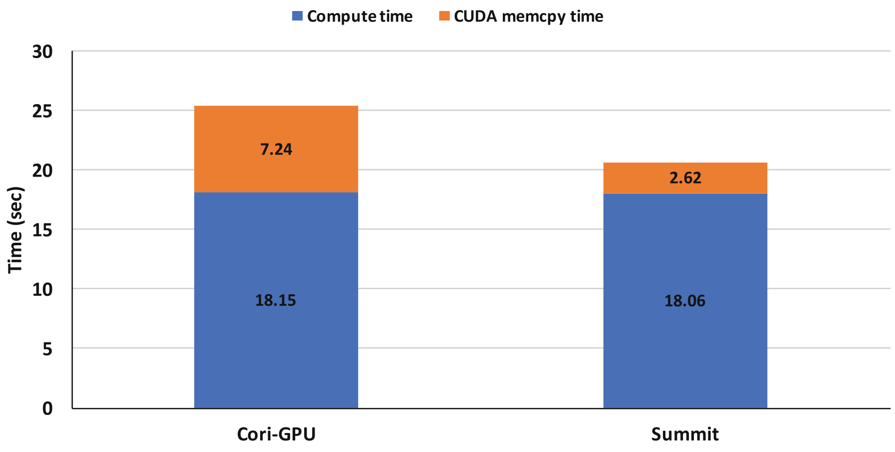

Fig. 8. Time spent in SpMM kernel on Cori-GPU and Summit when the memory footprint exceeds GPU memory capacity

4.3 Performance of Tiled and Unified Memory Versions of SpMM

Figure 9 shows the performance of the tiled SpMM kernel compared to the Unified Memory version of the SpMM kernel when the memory footprint is less than GPU memory capacity. The total memory footprint of this experiment is 11.7 GB. The tiled version is fastest on both platforms. `nvprof` shows that the tiled version is faster on Summit because of less time in CUDA memcpy. Interestingly, the Unified Memory version performs similarly on both platforms.

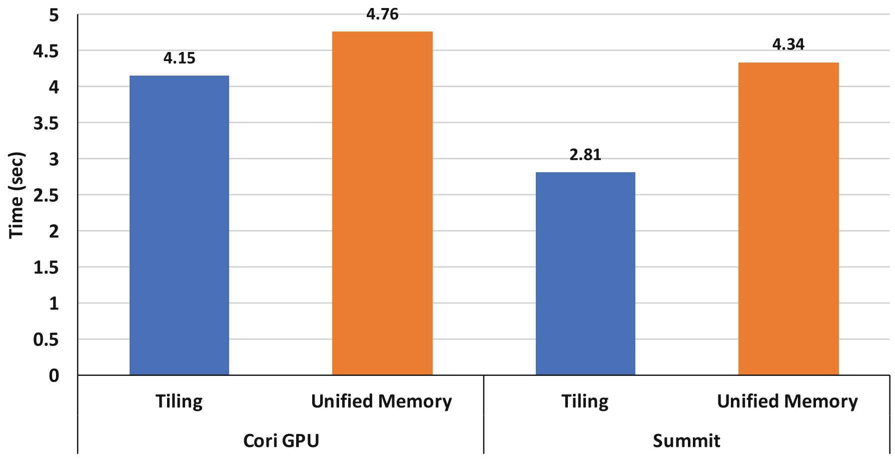

Fig. 9. Time spent in tiled and Unified Memory versions of the SpMM kernel on Cori-GPU and Summit. The memory footprint is less than GPU memory capacity.

Figure 10 shows the performance of the two SpMM kernels when the memory footprint exceeds GPU memory capacity. We used the same tile size (β) for the tiled experiments in Figs. 9 and 10. There are now significant differences between the performance of the tiled and Unified Memory versions. The most surprising result is the 48.2× performance difference between tiled and Unified Memory versions on Summit. This is a performance difference of 13.4× between Cori-GPU and Summit when using Unified Memory on different machines. This is unexpected given the high bandwidth NVLink 2.0 interconnect and hardware managed cache coherency on the Summit IBM system. Although not shown, there is a similar performance difference on Summit for the X^TY and XY kernels. Unified Memory performance is therefore poor and depends on the machine used.

Figure 11 shows `nvprof` output for the Unified Memory version of the XY kernel on Cori-GPU and Summit. The results show that the total count of page faults and the total data moved is the same on both systems. As expected, the data transfer is 3× faster on Summit according to the bandwidth of the CPU to GPU interconnect. However, the metric named "Gpu page fault groups" takes 30× more time on Summit compared to Cori-GPU for unknown reasons. This explains the poor performance on Summit. We observed similar performance difference without `nvprof` (`nvprof` added a performance overhead of about 10% on both machines). We are currently in contact with OLCF and NVIDIA staff to understand our performance observations.

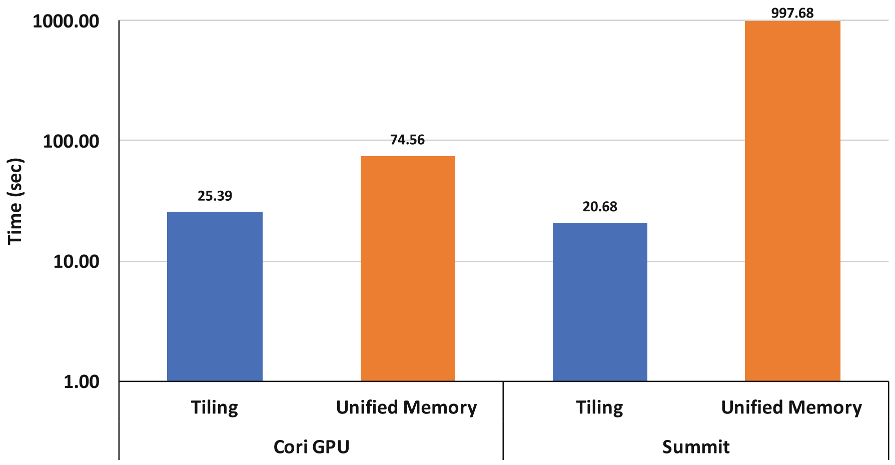

Fig. 10. Time spent in tiled and Unified Memory versions of the SpMM kernel on Cori-GPU and Summit. The memory footprint exceeds GPU memory capacity. We use a logarithmic scale on the *Time (sec)* axis to capture the slow run time for the Unified Memory configuration on Summit.

Cori-GPU

```
Device "Tesla V100-SXM2-16GB (0)"
   Count   Avg Size   Min Size   Max Size    Total Size    Total Time   Name
  196608   170.67KB   4.0000KB   0.9961MB   32.00000GB     3.326868s   Host To Device
    8526   1.9993MB   4.0000KB   2.0000MB   16.64655GB     1.368811s   Device To Host
   98304          -          -          -            -    10.668444s   Gpu page fault groups
Total CPU Page faults: 98305
```

Summit

```
Device "Tesla V100-SXM2-16GB (0)"
   Count   Avg Size   Min Size   Max Size    Total Size    Total Time   Name
  163840   204.80KB   64.000KB   960.00KB   32.00000GB     1.078612s   Host To Device
    8525   1.9998MB   64.000KB   2.0000MB   16.64850GB   396.9533ms   Device To Host
   98304          -          -          -            -   313.43688s   Gpu page fault groups
    8524   2.0000MB   2.0000MB   2.0000MB   16.64844GB            -
Remote mapping from device
Total CPU Page faults: 98305
Total remote mappings to CPU: 8524
```

Fig. 11. Unified Memory nvprof profile of the *XY* microbenchmark on Cori-GPU (top) and Summit (bottom).

5 Discussion

In this section we discuss the key learnings from the results in Sect. 4.

The results show that we have successfully ported the LOBPCG solver to NVIDIA GPUs using directives and optimized CUDA library calls. We obtained similar performance for the OpenMP implementation using Clang and XLC compiler as we did for the OpenACC implementation using the PGI compiler. The quality of OpenMP compilers for GPUs have often been criticized over the past few years [23], however, our experience provides evidence that OpenMP compilers are becoming more robust and are capable of generating high performance code.

We found that the key enabler of performance was to keep data resident on the GPU between calls to optimized CUDA math functions. We were able to do this trivially by adding OpenMP/OpenACC accelerator directives to the large number of kernels in the LOBPCG solver. In the past, this would have been much more challenging and time-consuming because the remaining application kernels would need to be ported to CUDA. Our related work section shows that earlier attempts to port a LOBPCG solver to GPUs by other scientists was generally focused on optimizing the SpMM kernel only on GPU whereas we focus on optimizing the full solver on GPU. This highlights the productivity gains from using directives and the importance of interoperability between the code generated by the OpenMP/OpenACC compilers and CUDA. This interoperability is not required in the OpenMP specification and is only recommended as a note to implementors in the OpenACC specification. However, we have highlighted the importance of interoperability, and believe that the HPC community should strongly request this support from compilers as we have done for LLVM/Clang (https://bugs.llvm.org/show_bug.cgi?id=42643).

We have shown that our LOBPCG microbenchmarks can be tiled to solve problems larger than GPU memory capacity. We found that the time spent in `cublasDgemm` for the inner product (X^TY) microbenchmark is shorter than the time spent moving data to and from the GPU. This indicates that it is not possible to write a tiled `cublasDgemm` for larger problems which achieves the same computational throughput as a problem which fits in GPU memory capacity. The tiled `cublasDgemm` performance was mostly determined by the bandwidth of the CPU to GPU interconnect. This will remain a challenge in many CPU+GPU systems in the coming years because PCIe Gen4 has lower bandwidth than NVLink 2.0. The SpMM microbenchmark showed the opposite to X^TY in that more time was spent in computation than data movement. This indicates that data movement costs could be hidden, i.e., computation on one tile could occur concurrently with the data movement for the next tile. The full LOBPCG solver includes X^TY and SpMM operations. Therefore, the amount of computation on the GPU relative to data movement between CPU and GPU is more than what is shown in our microbenchmarks. This indicates that it should be possible to write an efficient LOBPCG solver for GPUs which can solve problems larger than the GPU memory capacity.

We had mixed success when using a Unified Memory implementation of the SpMM kernel. The performance was a little worse than the tiled implementation when the memory footprint was less than GPU memory capacity. This could be acceptable to many application programmers because we obtained this performance with much simpler code. This would be a huge productivity win for the application programmer because there is no need to manage separate host and device copies of data; there is just a single pointer to the data which can be used on both host and device. We found that the performance of the Unified Memory implementation was much worse than the tiled implementation when the memory footprint exceeded GPU memory capacity. It was so bad on Summit that it would have been more efficient to use a CPU implementation and leave the GPUs idle. We are still working to understand why Unified Memory performance was so poor on Summit. However, our early experience serves as a warning to application programmers that they should not rely on Unified Memory when application memory footprint is larger than GPU memory capacity. It is also useful information to HPC system providers that the success of their users strongly depends on purchasing GPUs with sufficient memory capacity.

We recommend that tiling be used in large memory footprint applications on CPU+GPU systems. This can deliver both high performance and predictable performance across different CPU+GPU systems. However, it can be a significant amount of work to tile and overlap data transfers with computation in an application. This may become easier in future with enhancements to the OpenMP standard providing directive-based partitioning and pipelining [10]. Alternatively, middleware for sparse solvers on GPUs could abstract away these programming challenges.

6 Conclusions

In this paper, we have described our approaches to mix CUDA library calls with OpenMP/OpenACC offloading pragmas in order to implement and optimize the full LOBPCG eigensolver on GPU-accelerated systems. We successfully used both OpenMP and OpenACC and achieved a speedup of 2.8×–4.3× over a baseline CPU implementation. Our experiments with SpMM and inner product microbenchmarks showed that tiling is the preferred approach for larger problem sizes. We found that a naive Unified Memory implementation had worse performance than a tiled implementation by up to an order of magnitude depending on the target supercomputing platform. Our future work will go in the direction of tiling the full LOBPCG solver and attempting to overlap computation with data movement.

Acknowledgments. This work was supported in part by the US Department of Energy, Office of Science under the award DE-SC0018083 (NUCLEI SciDAC-4 collaboration) and the National Science Foundation under the award OAC-1845208. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231. 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. The authors would like to thank Brandon Cook for helpful discussion about MFDn application requirements and useful research directions for this project.

Data Availability Statement.

Summary of the Experiments Reported

We conducted all of our experiments on the Cori-GPU testbed at the National Energy Research Scientific Computing Center (NERSC) and the Summit supercomputer at the Oak Ridge Leadership Computing Facility (OLCF) using Intel Compiler v19.0.3 (OpenMP for CPU), LLVM/Clang compiler v9.0.0-git (OpenMP for GPU), and PGI compiler v19.5 (OpenACC for CPU and GPU), CUDA v10.1.168, IBM XLC Compiler v16.1.1-3 (OpenMP for CPU and GPU) as described in the paper. Our software and dataset are publicly available at `10.6084/m9.figshare.11636067` [14]. The repository includes necessary instructions and scripts to run our software. Interested individuals can contact the authors if they need they help to run the codebase.

Artifact Availability

Software Artifact Availability. All author-created software artifacts are maintained in a public repository under an OSI-approved license.

Hardware Artifact Availability. All author-created hardware artifacts are maintained in a public repository under an OSI-approved license.

Data Artifact Availability. All author-created data artifacts are maintained in a public repository under an OSI-approved license.

Proprietary Artifacts. None of the associated artifacts, author-created or otherwise, are proprietary.

List of URLs and/or DOIs Where Artifacts are Available. 10.6084/m9. figshare. 11636067.

The details of the baseline experimental setup, and modifications made for the paper are also available at https://github.com/fazlay-rabbi/WACCPD_2019_Artifact [14].

References

1. Cori-GPU system configuration. https://docs-dev.nersc.gov/cgpu/
2. Openmp specification. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf
3. Summit system configuration. https://www.olcf.ornl.gov/summit/
4. HIP : Convert CUDA to Portable C++ Code (2019). https://github.com/ROCm-Developer-Tools/HIP. Accessed 4 Sept 2019
5. Aktulga, H.M., Buluç, A., Williams, S., Yang, C.: Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pp. 1213–1222. IEEE (2014)

6. Anzt, H., Tomov, S., Dongarra, J.: Implementing a sparse matrix vector product for the SELL-C/SELL-C-σ formats on nvidia gpus. University of Tennessee, Technical report. ut-eecs-14-727 (2014)
7. Anzt, H., Tomov, S., Dongarra, J.: Accelerating the LOBPCG method on GPUs using a blocked sparse matrix vector product. In: Proceedings of the Symposium on High Performance Computing, pp. 75–82. Society for Computer Simulation International (2015)
8. Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. p. 18. ACM (2009)
9. Choi, J.W., Singh, A., Vuduc, R.W.: Model-driven autotuning of sparse matrix-vector multiply on GPUs. ACM SIGPLAN Not. **45**, 115–126 (2010)
10. Cui, X., Scogland, T.R.W., de Supinski, B.R., Feng, W.: Directive-based partitioning and pipelining for graphics processing units. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 575–584, May 2017. https://doi.org/10.1109/IPDPS.2017.96
11. Davis, T., Hu, Y., Kolodziej, S.: The suitesparse matrix collection (2018). http://faculty.cse.tamu.edu/davis/suitesparse.html
12. Deldon, S., Beyer, J., Miles, D.: OpenACC and CUDA unified memory. Cray User Group (CUG), May 2018
13. Dziekonski, A., Rewienski, M., Sypek, P., Lamecki, A., Mrozowski, M.: GPU-accelerated LOBPCG method with inexact null-space filtering for solving generalized eigenvalue problems in computational electromagnetics analysis with higher-order fem. Commun. Comput. Phys. **22**(4), 997–1014 (2017)
14. Rabbi, F., Daley, C.S., Aktulga, H.M., Wright, N.J.: Evaluation of directive-based GPU programming models on a block eigensolver with consideration of large sparse matrices (waccpd 2019 paper's artifact). https://doi.org/10.6084/m9.figshare.11636067, https://github.com/fazlay-rabbi/WACCPD_2019_Artifact
15. Garland, M.: Sparse matrix computations on manycore GPU's. In: Proceedings of the 45th annual Design Automation Conference, pp. 2–6. ACM (2008)
16. Hong, C., et al.: Efficient sparse-matrix multi-vector product on GPUs. In: Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing, pp. 66–79. ACM (2018)
17. Khorasani, F., Gupta, R., Bhuyan, L.N.: Scalable SIMD-efficient graph processing on GPUs. In: 2015 International Conference on Parallel Architecture and Compilation (PACT), pp. 39–50. IEEE (2015)
18. Knap, M., Czarnul, P.: Performance evaluation of unified memory with prefetching and oversubscription for selected parallel CUDA applications on NVIDIA Pascal and Volta GPUs. J. Supercomput. **75**, 1–21 (2019)
19. Knyazev, A.V.: Toward the optimal preconditioned eigensolver: locally optimal block preconditioned conjugate gradient method. SIAM J. Sci. Comput. **23**(2), 517–541 (2001)
20. Knyazev, A.V., Argentati, M.E.: Implementation of a preconditioned eigensolver using hypre (2005)
21. Knyazev, A.V., Argentati, M.E., Lashuk, I., Ovtchinnikov, E.E.: Block locally optimal preconditioned eigenvalue xolvers (BLOPEX) in HYPRE and PETSc. SIAM J. Sci. Comput. **29**(5), 2224–2239 (2007)
22. Lanczos, C.: An Iteration Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operators. United States Government Press Office, Los Angeles (1950)

23. Larrea, V.G.V., Budiardja, R., Gayatri, R., Daley, C., Hernandez, O., Joubert, W.: Experiences porting mini-applications to OpenACC and OpenMP on heterogeneous systems. In: Cray User Group (CUG), May 2019
24. Maris, P., et al.: Large-scale ab initio configuration interaction calculations for light nuclei. J. Phys.: Conf. Ser. **403**, 012019 (2012)
25. Maris, P., Sosonkina, M., Vary, J.P., Ng, E., Yang, C.: Scaling of ab-initio nuclear physics calculations on multicore computer architectures. Procedia Comput. Sci. **1**(1), 97–106 (2010)
26. Naumov, M., Chien, L., Vandermersch, P., Kapasi, U.: cuSPARSE library. In: GPU Technology Conference (2010)
27. Ortega, G., Vázquez, F., García, I., Garzón, E.M.: FastSpMM: an efficient library for sparse matrix matrix product on GPUs. Comput. J. **57**(7), 968–979 (2014)
28. Sakharnykh, N.: Everything You Need To Know About Unified Memory. Presented at GPU Technology Conference (GTC) (2018). http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf. Accessed Mar 2018
29. Shao, M., Aktulga, H.M., Yang, C., Ng, E.G., Maris, P., Vary, J.P.: Accelerating nuclear configuration interaction calculations through a preconditioned block iterative eigensolver. Comput. Phys. Commun. **222**, 1–13 (2018)
30. Sternberg, P., et al.: Accelerating configuration interaction calculations for nuclear structure. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, p. 15. IEEE Press (2008)
31. Vazhkudai, S.S., et al.: The design, deployment, and evaluation of the coral pre-exascale systems. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, p. 52. IEEE Press (2018)
32. Wang, Y.: Research on matrix multiplication based on the combination of OpenACC and CUDA. In: Xie, Y., Zhang, A., Liu, H., Feng, L. (eds.) GSES 2018. CCIS, vol. 980, pp. 100–108. Springer, Singapore (2019). https://doi.org/10.1007/978-981-13-7025-0_10
33. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for floating-point programs and multicore architectures. Technical report, Lawrence Berkeley National Lab (LBNL), Berkeley, CA, USA (2009)
34. Yang, C., Buluç, A., Owens, J.D.: Design principles for sparse matrix multiplication on the GPU. In: Aldinucci, M., Padovani, L., Torquati, M. (eds.) Euro-Par 2018. LNCS, vol. 11014, pp. 672–687. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96983-1_48
35. Yang, X., Parthasarathy, S., Sadayappan, P.: Fast sparse matrix-vector multiplication on GPUs: implications for graph mining. Proc. VLDB Endow. **4**(4), 231–242 (2011)