Performance Portability of an SpMV Kernel Across Scientific Computing and Data Science Applications

Stephen L. Olivier
Sandia National Laboratories
Albuquerque, NM
slolivi@sandia.gov

Nathan D. Ellingwood
Sandia National Laboratories
Albuquerque, NM
ndellin@sandia.gov

Jonathan Berry
Sandia National Laboratories
Albuquerque, NM
jberry@sandia.gov

Daniel M. Dunlavy
Sandia National Laboratories
Albuquerque, NM
dmdunla@sandia.gov

***Abstract*—Both the data science and scientific computing communities are embracing GPU acceleration for their most demanding workloads. For scientific computing applications, the massive volume of code and diversity of hardware platforms at supercomputing centers has motivated a strong effort toward performance portability. This property of a program, denoting its ability to perform well on multiple architectures and varied datasets, is heavily dependent on the choice of parallel programming model and which features of the programming model are used. In this paper, we evaluate performance portibility in the context of a data science workload in constrast to a scientific computing workload, evaluating the same sparse matrix kernel on both. Among our implementations of the kernel in different performance-portable programming models, we find that many struggle to consistently acheive performance improvements using the GPU compared to simple one-line OpenMP parallelization on high-end multicore CPUs. We show one that does, and its performance approaches and sometimes even matches that of vendor-provided GPU math libraries.**

***Index Terms*—performance portability, SpMV, sparse matrix operations, graph computations**

I. Introduction

In the past decade, high-end GPUs have taken an increasingly prominent role in scientific computing, machine learning, and data science. Among the world's five most powerful supercomputers in 2020, two are accelerated by NVIDIA Volta GPUs and the other by NVIDIA Ampere GPUs [1]. Of the first three planned scale supercomputers in the United States, two are set to be accelerated by AMD GPUs, and the other will use Intel Xe GPUs. Thus, recent and upcoming flagship supercomputers include three different vendors, as well as different generations of hardware by the same vendor. Given this diversity of target platforms, the scientific computing community has expended substantial effort on *performance portability*, i.e., the ability to run the same program with little or no modification across different architectures at an acceptable level of performance [2], [3].

In scientific computing, performance portability typically involves the choice of a programming model with a corresponding implementation that supports multiple GPUs (and CPUs). Kokkos [4] and RAJA [5] provide performance-portable frameworks in the form of libraries built on the latest C++ standards. OpenMP [6] and OpenACC [7] use compiler directives to annotate C, C++, and Fortran code for execution on CPUs and GPUs. OpenCL [8] is a C library with nearly universal support among CPU, GPU, and FPGA vendors, but provides a low level of abstraction that serves better as a layer under other parallel programming models than as a direct API.

Considerable time and attention have been devoted to vetting performance of programming model options in the realm of classic modeling and simulation applications and math libraries (see, for example, [9]–[11]). However, it is unclear if the level of performance portability observed on scientific computing workloads also extends to problems in the data analytics space, even when using the same basic math kernels. In this paper, we investigate the performance portability of a sparse matrix-vector (SpMV) multiplication kernel, motivated by its use in iterative linear solvers in data science applications. For example, efficient SpMV kernels are critical in solving the seed-set expansion problem for large graphs used in social network analysis [12]. Although the SpMV kernel is used extensively in traditional scientific computing applications as well, the structure of the matrix nonzeros is often dramatically different for applications in scientific computing (e.g., relatively few nonzeros per row, mostly concentrated in blocks around the diagonal in highly structured patterns) and data science (e.g., multiple orders of magnitude different nonzeros per row, unstructured distribution of nonzeros across rows).

The main contributions of this paper are as follows:

- Demonstration of the path from serial to performance-portable implementation of the SpMV kernel on CPUs and GPUs using the OpenMP and Kokkos programming models.
- Comparison of performance portability of the SpMV kernel in the context of scientific computing (finite element method) and data science (graph seed-set expansion).

II. Background

A. An Overview of GPGPU Programming

General purpose GPU (GPGPU) systems and their execution models present a dense web of vendor-specific terminology. However, many basic concepts are common across vendors, and we shall use generic terms to give a high-level view of GPGPU programming. The GPU is composed of groups of processing units and a global memory accessible by all processing units. Each group of processing units has a local memory and register sets that allow it to keep multiple teams of

TABLE I
A PERFORMANCE-PORTABILITY HIERARCHY

Source of Parallelism	Source of Portability	Skill Level	Performance Expectation	This paper
Vendor	None	Serial programmer	Maximum	CPU, NVIDIA
Descriptive language features	Vendor/compiler/open-source	Basic parallel programming	Near-maximum	CPU, NVIDIA
Prescriptive language features	Vendor/compiler/open-source	Basic to intermediate	Good	CPU, NVIDIA, AMD
User-level library	Library-dependent	Serial programmer to expert	Good to near-maximum	CPU, NVIDIA, AMD

threads resident and to swap frequently between those teams to maximize utilization. Programming models that target a GPU must thus decompose application kernels into teams of threads, and the GPU runtime system schedules those teams of threads onto the groups of processing units.

Constraints on the GPU execution model vary by vendor and hardware generation, but some general principles apply. First, it is not guaranteed that all outstanding teams of threads are scheduled onto the GPU at once, and frequently they are not. In this way, the GPU runtime scheduler can accommodate kernels of any size on a GPU with fixed execution resources. Thus, synchronization operations such as barriers that span the whole GPU are either expensive or prohibited, though atomic operations on memory locations are allowed. However, within a single team of threads, synchronization is available and efficient. Finally, there is the issue of data transfer between the GPU and its host CPU. Some systems present a common address space across both CPU and GPU memories. Some even allow data objects to be automatically moved between the two memories. To maximize portability and avoid any hidden costs of implicit data transfers, explicit data transfer operations between CPU and GPU memories should be used.

B. Performance Portability Hierarchy

In Table I, we summarize our experience working toward performance-portable data science software. Performance portability requires a source of parallelism and a source or mechanism for portability. These determine expectations for level-of-effort and performance. The rows of the table address inflexible but performant vendor products designed to exploit hardware and keep customers within their product ecosystem, runtime systems that support various developer and application communities, and user-level libraries that provide features not yet in languages and runtimes.

NVIDIA's *cuSPARSE* library of basic linear algebra subroutines[1] exemplifies a performant but vendor-specific option for leveraging GPU parallelism. It is designed to extract maximum performance from NVIDIA devices. We have no reasonable expectation of beating its performance since NVIDIA has proprietary knowledge of their own systems and any published algorithmic advances would soon be incorporated into it.

Rather than committing to only one vendor and depending on that vendor to incorporate algorithmic advances, the first step down our hierarchy considers *descriptive language features*. They allow users to incorporate algorithmic changes themselves with reduced dependence on vendor timelines. A current canonical example is the `loop` directive of OpenMP 5.0. This primitive is descriptive in the sense that it provides an interface to describe a goal such as "parallelize this loop nest" without requiring the programmer to make decisions about load balancing or multi-level parallelism. Furthermore, it provides access to any hardware, CPU or GPU, that supports it. Currently, only NVIDIA provides GPU-level support for the primitive. However, with that support, OpenMP programs may be able to exploit NVIDIA devices near-optimally with no exposure to NVIDIA's lower level CUDA API. If other vendors make the effort to provide this support, we could achieve a high level of performance portability via descriptive runtime features. However, we note that this task for vendors is relatively hard compared to other methods that share the burden for providing the performance with the programmer.

Since supporting descriptive runtime features is challenging for vendors, we next consider *prescriptive language features*. These include the familiar APIs that parallel programmers have long been using on CPU systems. The canonical exemplar for the CPU is OpenMP's `pragma omp parallel for`, along with the extensive ecosystem of related features that provide flexibility and parallelization alternatives to the OpenMP programmer. The programmer now controls such details as load balancing and matching the application's parallelism to the hardware, and the level-of-effort for a vendor to support prescriptive language features is greatly reduced. However, extensions for GPU support are more recent and immature than capabilities for CPU parallelism.

The explosion of heterogeneous computing alternatives has put great stress on the U.S. Department of Energy's (DOE) modeling and simulation mission, which features large, mature codes developed over decades. Concerns about sustainability have precluded dependence on particular parallel programming systems, so it has been necessary to create performance-portable options at the user level, at least for C++ applications. A canonical example of these is *Kokkos*, a thin layer of software that has enabled exploitation of heterogeneous architectures by DOE codes and at the same time has influenced the evolving international C++ standards. For example, Kokkos has directly influenced both the C++ 20 (`atomic ref`) and C++ 23 (`basic_mdspan`, executors) standards [13]. Kokkos also features a growing library called KokkosKernels [14] that encapsulates common linear algebra operations and can access vendor libraries such as *cuSPARSE*. Therefore, savvy Kokkos programmers can achieve maximum or near-maximum performance on some architectures with codes that still run with good performance on other architectures.

[1] https://docs.nvidia.com/cuda/cusparse/

While this paper addresses some available options for C++ code, solutions for other languages could bring parallelism to the masses. For example, the data science community has welcomed the emergence of *numba*, a Python package that generalizes access to runtimes such as OpenMP and NVIDIA/AMD GPU libraries beyond that already available in special purpose frameworks aimed at machine learning such as TensorFlow [15] and PyTorch [16]. In our experience, enabling impressive CPU parallelism with *numba* is easy, but achieving GPU performance require higher levels of expertise.

III. Developing Performance Portable Code

A. CPU-Only: The Starting Point (Serial SpMV)

The sequential sparse matrix vector (SpMV) product algorithm is implemented for compressed row storage (CRS) format matrices and serves as the starting point for our study. The associated CRS matrix arrays storing the data are named `rowmap`, `entries`, and `values`. The `rowmap` stores extents of rows used as offsets for indexing into the `entries` and `values` arrays; `entries` stores the column indices for non-zero matrix entries (in sorted order per row), and `values` stores corresponding the non-zero values.

The algorithm computes $y = Ax$ (see Listing 1). It consists of an outer loop over the rows of the matrix, which contains an inner loop over the offsets into the `entries` and `values` arrays for the given row to compute the product of the (sparse) matrix row by vector `x`, storing the result in `y`.

```
for (size_type row = 0; row < nrows; ++row) {
  values_t sum = 0.0;
  rmap_t row_start = rowmap[row];
  rmap_t row_end = rowmap[row+1];
  for (rmap_t idx = row_start; idx < row_end; ++idx) {
    entries_t col = entries[idx];
    sum += values[idx]*x[col];
  }
  y[row] = sum;
}
```

Listing 1. Sequential Version of SpMV Kernel

B. CPU-Only: Legacy OpenMP (`parallel for`)

Since 1997, OpenMP has provided programmers with compiler directives for incremental parallelizations of their loops that are supported across many vendor and open-source compilers. Listing 2 shows the one additional line (Line 1) required to transform the SpMV example from a serial to a parallel implementation using OpenMP. This line combines two constructs, `parallel` to create a team of threads and `for` to divide the loop iterations among the threads for execution. The `schedule(dynamic)` clause instructs the runtime to assign the iterations at execution time. This behavior provides load balancing—e.g., in the case of SpMV if some rows have many more nonzeros than others. For this version of the code in our study, we set `OMP_NUM_THREADS` to the total number of hardware threads. Using more than one hardware thread per core did not help in all cases, but it never degraded performance compared to using one thread per core. Parallel initialization of data is optional but recommended, and our evaluation uses it for all versions discussed from this point.

```
1  #pragma omp parallel for schedule(dynamic)
2  for (size_type row = 0; row < nrows; ++row) {
3    values_t sum = 0.0;
4    rmap_t row_start = rowmap[row];
5    rmap_t row_end = rowmap[row+1];
6    for (rmap_t idx = row_start; idx < row_end; ++idx) {
7      entries_t col = entries[idx];
8      sum += values[idx]*x[col];
9    }
10   y[row] = sum;
11 }
```

Listing 2. CPU-Only OpenMP Version of SpMV Kernel

C. GPU-Only: Vendor libraries (cuSPARSE/rocSPARSE)

Unfortunately, `omp parallel for` is not a performance-portable construct for GPUs, though we will see later that OpenMP support for GPUs is possible through additional directives. For users seeking optimal GPU performance on their kernels with little effort, they may find that an implementation of the kernel is available in a vendor library. These libraries are typically not portable between different GPU vendors. NVIDIA provides SpMV for their GPUs in the *cuSPARSE* library and AMD provides it in the *rocSPARSE* library. Moreover, NVIDIA changed their API for SpMV in *cuSPARSE* between versions of their CUDA 10 and CUDA 11 SDK. Nonetheless, the expected high performance of these vendor libraries makes them valuable for our study as comparison points. A recent study of SpMV performance using *cuSPARSE* and *rocSPARSE* illustrates a comparison of these tools on a wide variety of application matrices [17].

D. Performance-Portable OpenMP (`target ...`)

Version 4.0 of the OpenMP specification introduced directives to enable offloading of data and computation to accelerators. In keeping with OpenMP's vendor agnostic philosophy, the specification's "device"-related directives are not intended to be specific only to GPUs. Moreover, the specification does not require any particular correspondence between the parallelism expressed in the directives and the levels of the accelerator hardware, as long as the semantics are correct.

For our SpMV example, the effort to use OpenMP's GPU support portably comprises code for data transfer between CPU and GPU and offload of the computation onto the GPU. Data transfer code is shown in Listing 3. The `target data` construct defines a scope in which the data is available on the GPU. The `map` clause specifies each data item to be moved and the direction of the transfer. In the example, the 0-initialized output must be moved from CPU to GPU before the computation and from GPU to CPU after the computation, hence the map type `tofrom`. The input vector and matrix are unchanged by the computation and do not need to be moved back onto the CPU, hence the map type `to`.

```
#pragma omp target data map(tofrom: y[0:nrows])    \
                        map(to: x[0:ncols],        \
                                rowmap[0:nrows+1], \
                                entries[0:nnz],    \
                                values[0:nnz])

// Call compute kernel here
```

Listing 3. Data Transfer for OpenMP Target

The code for GPU execution of the loop is placed inside the scope of the `target data` construct, but for clarity we show it separately in Listing 4. The directive used is longer than the simple CPU-only code discussed previously, but it is still only one line. First, `target` instructs the OpenMP implementation to offload execution to the GPU. Then, `teams distribute` creates work teams and divides the loop iterations among those teams. Finally, the familiar `parallel for` creates threads within each team and further divides that team's assigned iterations among those threads.

```
#pragma omp target teams distribute parallel for
for (size_type row = 0; row < nrows; ++row) {
  values_t sum = 0.0;
  rmap_t row_start = rowmap[row];
  rmap_t row_end = rowmap[row+1];
  for (rmap_t idx = row_start; idx < row_end; ++idx) {
    entries_t col = entries[idx];
    sum += values[idx]*x[col];
  }
  y[row] = sum;
}
```

Listing 4. OpenMP Target Version of SpMV Kernel

Compilation of this code results in a "fat binary" containing generated machine code for both CPU and GPU. The `OMP_TARGET_OFFLOAD` environment variable can be set to `MANDATORY` to force GPU execution (terminating if a GPU is not available) or `DISABLED` to force CPU execution. Absent explicit instruction from the user, the compilers we used for this study choose to create only one team of threads for the CPU, so for CPU testing we added `schedule(dynamic)` for load balancing. The compilers generated many teams for the GPU, such that load balance between teams is ensured naturally by the GPU's runtime scheduler, so we omitted the clause for GPU testing. Our CPU executions used a team size equal to the number of hardware threads, and our GPU executions used a number of threads determined to be optimal for each GPU platform based on a priori testing of prospective candidate values ranging in powers of two up to 2048 using a test input. However, relying on the OpenMP compiler to pick a default thread count resulted in performance not far behind those using the optimal value.

E. Descriptive OpenMP (`loop`)

Since its inception, OpenMP has tended to provide mostly prescriptive directives, i.e., telling the compiler how to parallelize the code. However, more recent versions of the standard have increasingly provided more descriptive options, i.e., instructing the compiler that code should be parallelized but not how to do the parallelization. OpenMP 5.0 introduced the descriptive `loop` directive, which simply informs the compiler that loop iterations are independent and invites the compiler to choose any way it pleases to make the loop fast. Listing 5 shows use of this directive for our SpMV example. It is simply nested within a `target teams` construct to specify offload to work teams on the GPU. What occurs in the work teams is entirely up to the compiler and runtime. As a newer feature that puts the burden of performance largely on the OpenMP implementation, it is not yet as widely implemented as the older prescriptive `target teams distribute parallel for` alternative. Data transfer as shown earlier is still required.

```
#pragma omp target teams loop
for (size_type row = 0; row < nrows; ++row) {
  values_t sum = 0.0;
  rmap_t row_start = rowmap[row];
  rmap_t row_end = rowmap[row+1];
  for (rmap_t idx = row_start; idx < row_end; ++idx) {
    entries_t col = entries[idx];
    sum += values[idx]*x[col];
  }
  y[row] = sum;
}
```

Listing 5. OpenMP Version of SpMV Kernel Using `loop` Construct

F. Kokkos I: The Low-Hanging Fruit (`RangePolicy`)

The Kokkos Core programming model provides abstractions for parallel execution patterns, policies to allow users to select how and where those patterns execute, and data structure abstractions that support architecture-aware memory layouts of the data necessary to achieve performance portability, as well as policies for user control of memory layout and where memory allocation occurs. Support is available for CPU backends (Serial, OpenMP, HPX) and GPU backends (CUDA for NVIDIA, Hip for AMD, SYCL for Intel and NVIDIA, and OpenMPTarget for NVIDIA, AMD, and Intel).

```
using namespace Kokkos;

// Types for Kokkos views to be used
using rowmap_view_t = View<rowmap_t*>;
using entries_view_t = View<entries_t*>;
using values_view_t = View<values_t*>;

// Kokkos views to be used
rowmap_view_t rowmap_v("rowmap", nrows+1);
entries_view_t entries_v("entries", nnz);
values_view_t values_v("values", nnz);
values_view_t x("x", ncols);
values_view_t y("y", nrows);

// Create host-side "mirror" views for matrix
auto hrowmap_v = create_mirror_view(rowmap_v);
auto hentries_v = create_mirror_view(entries_v);
auto hvalues_v = create_mirror_view(values_v);

// Load data to host views...

// Explicity move data from host to device
deep_copy(rowmap_v, hrowmap_v);
deep_copy(entries_v, hentries_v);
deep_copy(values_v, hvalues_v);
deep_copy(x, values_t(1.0));
deep_copy(y, values_t(0.0));

// Call compute kernel here
```

Listing 6. Data Transfer for Kokkos

GPU data access requires the Kokkos `View` data structure. Views are templated on data type and dimension, abstract away memory allocation and clean up, and provide multi-dimensional access operators to access data. The pattern for portable use, demonstrated in Listing 6, involves first constructing views with label and dimension sizes for device (`rowmap_v`, `entries_v`, `values_v`) and host (`hrowmap_v`, `hentries_v`, `hvalues_v`). To ensure views have compatible memory layouts for data transfer,

the `create_mirror_view` function creates host views.[2] After view construction/memory allocation, the `deep_copy` function provides data transfer between host and device.

```
using namespace Kokkos;
parallel_for (nrows,
  KOKKOS_LAMBDA (const size_type row) {
    values_t sum = 0.0;
    rmap_t row_start = rowmap_v[row];
    rmap_t row_end = rowmap_v[row+1];
    for (rmap_t idx = row_start; idx < row_end; ++idx) {
      entries_t col = entries_v[idx];
      sum += values_v[idx]*x[col];
    }
    y[row] = sum;
} );
```

Listing 7. Initial Kokkos Version of SpMV Kernel

Listing 7 demonstrates a Kokkos implementation for SpMV requiring the minimum amount of programming model knowledge to achieve parallel execution and portability across all supported architectures. The `parallel_for` execution pattern parallelizes the outer loop over the number of rows of the matrix. The default execution policy, called `RangePolicy`, employs flattened parallelism in which the total work is specified and results in assigning each matrix row-vector product to a hardware thread for computation. In this case, parallelism is limited by the number of rows in the matrix, thus making it inefficient for use with matrices of small dimensions.

G. Kokkos II: Increased Parallelism (`TeamPolicy`)

```
using namespace Kokkos;
// user-defined team_size, vector_size, rows_per_team
int nleagues = (nrows+rows_per_team-1) / rows_per_team;
using team_policy_t = TeamPolicy<execution_space,
     Schedule<Dynamic>>;
using member_t = typename team_policy_t::member_type;

parallel_for(
  team_policy_t(nleagues, team_size, vector_size),
  KOKKOS_LAMBDA (const member_t & member) {
    rowmap_t t_start_row =
      member.league_rank()*rows_per_team;
    const bool full_range =
      t_start_row+rows_per_team < nrows;
    rowmap_t t_end_row =
      full_range ? t_start_row+rows_per_team : nrows;
    parallel_for(
      TeamThreadRange(member, t_start_row, t_end_row),
      [=] (const size_t row) {
         values_t sum = 0.0;
         rowmap_t boffset = rowmap_v[row];
         rowmap_t eoffset = rowmap_v[row+1];
         parallel_reduce(
           ThreadVectorRange(member, boffset, eoffset),
           [=] (const rowmap_t offset, values_t& tmp) {
             entries_t col = entries_v[offset];
             tmp += values_v[offset]*x[col];
           },sum);
         y[row] = sum;
      });
  });
```

Listing 8. Hierachical Parallelism Kokkos Version of SpMV Kernel

Improvement of the Listing 7 Kokkos implementation requires exposing more parallelism for the hardware. We achieve this through use of nested parallel execution patterns which explicitly define thread teams to exploit the hierarchical parallelism present in the computation. Instead of simply parallelizing over the number of rows of the matrix, we also group multiple rows together for parallel computation by a thread team, and parallelize the inner `for` loop using Kokkos' `parallel_reduce` execution pattern to protect concurrent writes to `sum`. The code is given in Listing 8.

The use of thread teams, rather than simply specifying total amount of work, results in an execution policy change from the previous default of Listing 7. The policies corresponding to the three levels of parallelism (outer-most to inner most) are: `TeamPolicy` (enables use of thread teams), `TeamThreadRange` (allows intra-team work), and `ThreadVectorRange` (sub-warp level parallelism of GPU). For three-level parallelism, we specify three numbers - the number of teams (`nleagues`), the number of threads in a team (`team_size`), and the vector length (`vector_size`) which is limited to the size of a warp for CUDA and a wavefront for Hip. The use of nested parallelism introduces more complexity and requires more advanced knowledge of Kokkos, but results in improved performance.

IV. Experimental Methodology

As an exercise in performance portability, we tested complete implementations of SpMV as presented in the previous section on two CPUs and three GPUs.

A. Program

The evaluation program consists of several phases:

- Reading in the matrix from disk
- Data transfer to GPU (unless using CPU only)
- 100 repeated SpMV operations
- Data transfer back to CPU (unless using CPU only)
- Verification checks

The SpMV phase is timed, and performance is measured in seconds per SpMV (total time for that phase divided by 100).

While the results exclude data transfer times, those operations are separately timed for reference. In our experiments, the vast majority of observed transfer times are a few tenths of a second. However, a few outliers take up to five seconds. Efficient use of a GPU requires sufficient reuse of the data: running only a few SpMV operations would not adequately amortize the costs of the data transfer, but after thousands of repeated SpMV operations, as are often needed in many scientific computing and data science applications, the data transfer costs are justified by faster kernel execution times.

B. Data sets

We consider two canonical matrices from scientific computing and data science. The former, labeled here as *Brick3D*, arises from a 27-point difference stencil for the Laplace operator on a 3D hex mesh with 320^3 mesh points and has been in research of parallel multigrid solvers [18], [19]. Figure 1 (left) illustrates the regular pattern of a small number of nonzero entries per row; this is typical in many scientific computing applications. The latter, labeled here as *Twitter*, is a variant of the Twitter-2010 dataset available from SNAP [20]–[23].

[2]The view code used in the study takes a slightly different form for compatibility with an existing matrix reader implementation.

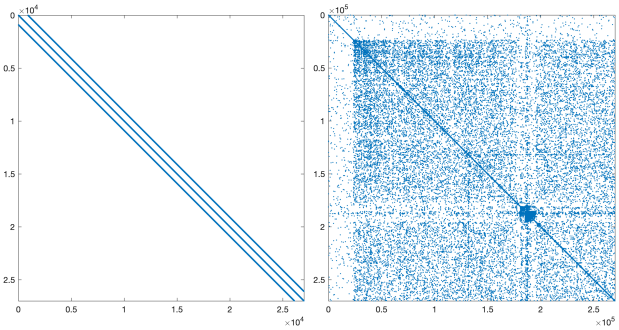

Fig. 1. Patterns of nonzero entries (i.e., blue dots) in the Brick3D (left) and Twitter (right) matrices. The data shown here illustrate submatrices of the full Brick3D (27,000 rows) and Twitter (270,000 rows) matrices.

We take the original graph, remove non-reciprocated edges, and extract the largest connected component; the result has 21,297,772 vertices and 265,025,545 edges. Figure 1 (right) shows that this social network graph lacks the regular structure of nonzero entries of Brick3D. As we will see, it is important to consider both types of data when making decisions about performance portability.

C. Platforms

Five different platforms are included in this study. Their specifications are given in Table II. The compiler and runtime systems vary based on the programming model, implementation, and hardware platform, as shown in Table III.

D. Kokkos Runtime Parameters

With performance-portable libraries like Kokkos, we are able to write a single implementation for SpMV capable of execution on supported platforms by compiling the source code with appropriate configuration options for the given architecture. Achieving best performance requires careful selection of kernel parameters, which can be sensitive to architecture, as well as input data such as characteristics of the matrix in the case of SpMV. For our GPU runs, performance was most sensitive to the `vector_size` kernel parameter. An optimal `vector_size` of 8 on NVIDIA Ampere architectures for the Brick3D matrix resulted in a 2.4X loss in performance for the Twitter matrix; likewise, the optimal `vector_size` of 32 for Twitter led to a 1.3X loss in performance for Brick3D.

Development of heuristics capable of supporting regular matrix structures typically encountered in scientific computing, as well as the more irregular structures found in data science, is a challenging task. Ongoing autotuning efforts toward this end may be the most promising path forward.

V. Experimental Results

Figure 2 shows the results of our primary study. Of the many interesting aspects of these plots, we first note that *the choice of dataset has a profound effect on both performance and portability*. For example, the descriptive runtime solution (OpenMP_Loop_PP) dominates the prescriptive runtime solution (OpenMP_Target_PP) when Twitter is run on GPU, but this result is inverted when Brick3D is run on GPU.

As expected, the vendor library implementations (*cuSPARSE* and *rocSPARSE*) offer excellent performance on their respective architectures. Of the performance-portable options, only a finely-tuned application of the Kokkos user-level library (Kokkos_Team_PP) preserves near-optimal performance on both GPU and CPU for both datasets. Other performance-portable methods fail to offer consistently superior performance compared to the CPU-only OpenMP implementation (OpenMP_CPU) when all combinations of dataset and GPU architecture are considered. Twitter is so challenging for the other methods that, in several instances, their performance significantly lags OpenMP_CPU. OpenMP_Target_PP at least matches OpenMP_CPU in all but one case and is the only fully performance-portable implementation other than those using Kokkos. Some data do appear to be anomalies—such as Volta outperforming Ampere on Brick3D using OpenMP_Loop_PP—and require further investigation.

Several caveats apply to this study. First, although our OpenMP-based solutions did not produce a variant competitive with Kokkos_Team_PP, there may be others that could. Our intent is to show the performance of straightforward OpenMP solutions that make reasonable use of the prescriptive and descriptive features currently available. Second, the AMD software stack for GPGPU, including its AOMP compiler, is a work in progress. Its current development focus is on feature completeness, but performance improvements are expected to accelerate as delivery of the Frontier supercomputer nears. And lastly, our rudimentary implementation of SpMV does not reflect the state of the art in algorithmic and data structure optimizations for SpMV on GPU. For example, one survey of such optimizations enumerates over 70 of them, though many are implemented in low-level CUDA or OpenCL code [24].

VI. Conclusions

In this paper, we used an SpMV kernel as a vehicle to investigate the use of performance portable programming models on five CPU and GPU architectures on both a data science workload and a scientific computing workload. We showed that the options vary widely in the code changes and knowledge required, with resulting performance varying based on workload and architecture. The most consistent message bases on our evaluation is that, at present, a code developer seeking to exceed the performance of a high-end multicore CPU for diverse datasets and on diverse architectures faces the choice to accept the code transformation and tuning effort required to use a method like Kokkos' `RangePolicy` that is tuned to the GPU, or hope that their particular kernel is available in vendor libraries. The situation may change as descriptive OpenMP GPU runtime support becomes more widely available and prescriptive OpenMP runtime support matures. Performance portability is not an entirely elusive goal, but the path to it is a currently narrow one.

Acknowledgment

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineer-

TABLE II
HARDWARE PLATFORMS USED

Architecture	Kind	Model	Processor configuration	OS/driver
Intel Skylake	CPU	8160	2-socket, 24 cores/socket, 2 HW threads/core (96 total), 192GB DDR memory	CentOS 7.9
AMD EPYC Zen2	CPU	7452	2-socket, 32 cores/socket, 2 HW threads/core (128 total), 256GB DDR memory	RHEL 7.6
NVIDIA Volta	GPU	V100	5120 CUDA cores, 32GB HBM2 memory, in NVIDIA DGX with 2X Intel Broadwell CPUs	CUDA 10.2
NVIDIA Ampere	GPU	A100	6912 CUDA cores, 40GB HBM2 memory, in rack node with 2X AMD Zen2 CPUs	CUDA 11.2
AMD Radeon	GPU	MI50	3840 stream processors, 16GB HBM2 memory, in rack node with 2X AMD Zen1 CPUs	ROCm 4.2

TABLE III
COMPILER AND RUNTIME SYSTEMS USED

Implementation	Intel Skylake	AMD Zen2	NVIDIA Volta	NVIDIA Ampere	AMD MI50
Serial	Intel icpx2021.2	Clang/LLVM 11.0	N/A	N/A	N/A
OpenMP_CPU	Intel icpx 2021.2	Clang/LLVM 11.0	N/A	N/A	N/A
cuSPARSE_GPU	N/A	N/A	GCC 6.4, nvcc 10.2	GCC 7.5, nvcc 11.2	N/A
rocSPARSE_GPU	N/A	N/A	N/A	N/A	GCC 8.2, hipcc 4.2
OpenMP_Loop_PP	NVIDIA nvc++ 21.5	NVIDIA nvc++ 21.3	NVIDIA nvc++ 20.11	NVIDIA nvc++ 21.3	not yet available
OpenMP_Target_PP	Intel icpx 2021.2	Clang/LLVM 11.0	Clang/LLVM 11.0	Clang/LLVM 11.0	ROCm/aomp 4.2
Kokkos_Range_PP	GCC 9.2	GCC 7.5	GCC 6.4, nvcc 10.2	GCC 7.5, nvcc 11.2	GCC 8.2, hipcc 4.2
Kokkos_Team_PP	GCC 9.2	GCC 7.5	GCC 6.4, nvcc 10.2	GCC 7.5, nvcc 11.2	GCC 8.2, hipcc 4.2

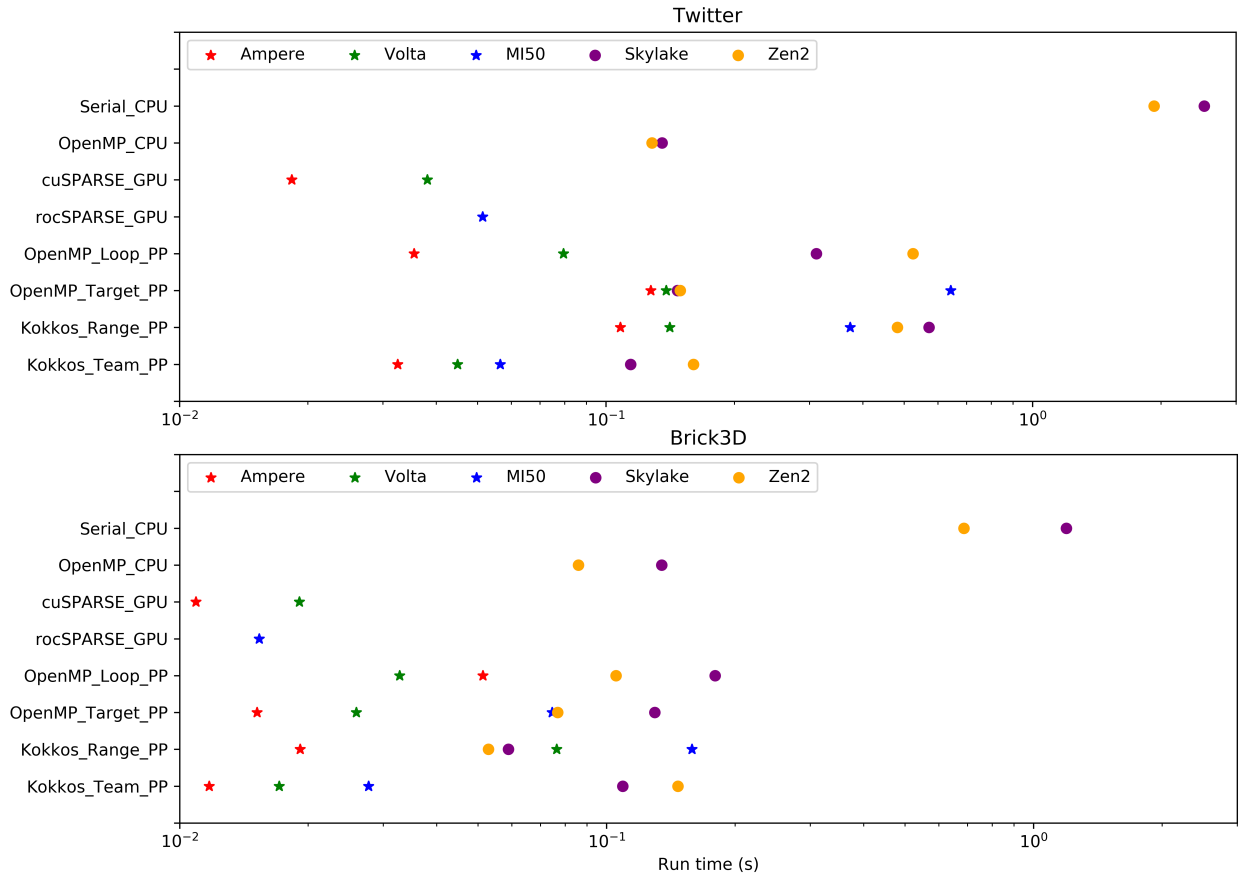

Fig. 2. Performance Portability Comparison Plots depicting SpMV run times for CPU-only approaches, GPU-only approaches, and performance-portable (PP) approaches. Our implementations using both the descriptive runtime solution (OpenMP_Loop_PP) and the prescriptive runtime solution (OpenMP_Target_PP) struggle on some combinations of architecture and dataset. Only our implementation using the tuned user-level PP approach (Kokkos_Team_PP) effectively exploits all architectures for both data science (Twitter) and scientific computing (Brick3D) workloads.

ing Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

REFERENCES

[1] top500.org, "Top 500 supercomputers: November 2020 list," https://www.top500.org/lists/top500/2020/11/, Nov 2020.

[2] T. P. Straatsma, K. B. Antypas, and T. J. Williams, *Exascale Scientific*

Applications: Scalability and Performance Portability, 1st ed. Chapman & Hall/CRC, 2017.

[3] "Portability across DOE Office of Science HPC facilities," https://performanceportability.org/, accessed: 2021-06-10.

[4] H. Edwards, C. 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.

[5] D. Beckingsale, et al., "RAJA: Portable performance for large-scale scientific applications," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 71–81.

[6] OpenMP Architecture Review Board, "OpenMP Application Programming Interface, Version 5.1," https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf, November 2020.

[7] OpenACC-Standard.org, "OpenACC Application Programming Interface, Version 3.1," https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.1-final.pdf, November 2020.

[8] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science & Engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[9] V. G. Vergara Larrea, R. D. Budiardja, R. Gayatri, C. Daley, O. Hernandez, and W. Joubert, "Experiences in porting mini-applications to OpenACC and OpenMP on heterogeneous systems," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 20, p. e5780, 2020.

[10] V. Artigues, K. Kormann, M. Rampp, and K. Reuter, "Evaluation of performance portability frameworks for the implementation of a particle-in-cell code," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 11, p. e5640, 2020.

[11] J. Eichstaedt, M. Vymazal, D. Moxey, and J. Peiro, "A comparison of the shared-memory parallel programming models OpenMP, OpenACC and Kokkos in the context of implicit solvers for high-order FEM," *Computer Physics Communications*, vol. 255, p. 107245, 03 2020.

[12] A. H. Foss, R. B. Lehoucq, W. Z. Stuart, J. D. Tucker, and J. W. Berry, "A deterministic hitting-time moment approach to seed-set expansion over a graph," arXiv:2011.09544, 2020.

[13] C. R. Trott, "Kokkos DeepDive," https://www.osti.gov/biblio/1648833, May 2019.

[14] S. Rajamanickam, S. Acer, L. Berger-Vergiat, V. Q. Dang, N. D. Ellingwood, E. Harvey, B. Kelley, C. R. Trott, J. Wilke, and I. Yamazaki, "Kokkos kernels: Performance portable sparse/dense linear algebra and graph kernels," *CoRR*, vol. abs/2103.11991, 2021. [Online]. Available: https://arxiv.org/abs/2103.11991

[15] M. Abadi, et al., "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[16] A. Paszke, et al., "Pytorch: An imperative style, high-performance deep learning library," arXiv:1912.01703, 2019.

[17] Y. M. Tsai, T. Cojean, and H. Anzt, "Sparse linear algebra on AMD and NVIDIA GPUs – the race is on," in *High Performance Computing*, P. Sadayappan, B. L. Chamberlain, G. Juckeland, and H. Ltaief, Eds., Cham, 2020, pp. 309–327.

[18] L. Berger-Vergiat, C. A. Glusa, J. J. Hu, M. Mayr, A. Prokopenko, C. M. Siefert, R. S. Tuminaro, and T. A. Wiesner, "MueLu user's guide," Sandia National Laboratories, Tech. Rep. SAND2019-0537, 2019.

[19] J. J. Elliott and C. M. Siefert, "Low thread-count gustavson: A multi-threaded algorithm for sparse matrix-matrix multiplication using perfect hashing," in *2018 IEEE/ACM 9th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (scalA)*, 2018, pp. 57–64.

[20] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[21] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th International Conference on World Wide Web*, 2010, pp. 591–600.

[22] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proceedings of the 13th International Conference on World Wide Web*, 2004, pp. 595–602.

[23] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th International Conference on World Wide Web*, 2011, pp. 587–596.

[24] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, "Sparse matrix-vector multiplication on GPGPUs," *ACM Trans. Math. Softw.*, vol. 43, no. 4, Jan. 2017.