A Portable Sparse Solver Framework for Large Matrices on Heterogeneous Architectures

Fazlay Rabbi*, Christopher S. Daley‡, Ümit V. Çatalyürek†, Hasan Metin Aktulga*
*Computer Science & Engineering, Michigan State University
‡ Lawrence Berkeley National Laboratory
†Amazon Web Services§ and School of Computational Science & Engineering, Georgia Institute of Technology
rabbimd@msu.edu, csdaley@lbl.gov, umit@gatech.edu, hma@msu.edu

***Abstract*—Programming applications on heterogeneous systems with hardware accelerators is challenging due to the disjoint address spaces between the host (CPU) and the device (GPU). The limited device memory further exacerbates the challenges as most data-intensive applications will not fit in the limited device memory. CUDA Unified Memory (UM) was introduced to mitigate such challenges. UM improves GPU programmability by supporting oversubscription, on-demand paging, and migration. However, when the working set of an application exceeds the device memory capacity, the resulting data movement can cause significant performance losses. We propose a tiling-based task-parallel framework, named *DeepSparseGPU*, to accelerate sparse eigensolvers on GPUs by minimizing data movement between the host and device. To this end, we tile all operations in a sparse solver and express the entire computation as a directed acyclic graph (DAG). We design and develop a memory manager (MM) to execute larger inputs that do not fit into GPU memory. MM keeps track of the data on CPU and GPU, and automatically moves data between them as needed. We use OpenMP target offload in our implementation to achieve portability beyond NVIDIA hardware. Performance evaluations show that *DeepSparseGPU* transfers 1.39x-2.18x less host to device (H2D) and device to host (D2H) data, while executing up to 2.93x faster than the UM-based baseline version.**

***Index Terms*—sparse solvers, task parallelism, performance optimization, directive-based GPU programming, performance portability.**

I. Introduction

Graphics Processing Units (GPUs) have been employed in more than a quarter of TOP500 [28] supercomputers due to their massive peak performance and power efficiency. GPUs have been used to accelerate applications in various domains, such as graph analytics, machine learning, computational finance, climate modeling, multimedia. Given NVIDIA's dominance in GPU computing, most GPU acceleration efforts to date have focused on CUDA, which is an NVIDIA-hardware specific programming model. Considering the increasing number of computing systems with GPUs from different vendors, choosing a portable programming model that allows applications to adapt to the diversity in heterogeneous computing is crucial from software maintenance and sustainability perspectives. OpenMP 4.5+ and OpenACC have emerged as portable directive-based programming models to provide a unified programming model for CPUs and GPUs. Higher-level abstraction adopted in these directive-based programming models yields application portability and developer productivity. Recent studies show that OpenMP and OpenACC are robust and able to provide good performance on different hardware [8], [12], [27].

Regardless of whether an application uses a device-specific (e.g., CUDA) or a portable (e.g., OpenMP) programming model, the amount of data moved between CPU and GPU is a key concern in heterogeneous systems due to the wide gap between computational performance and data movement. For best performance, programmers have to explicitly control the data movement between the host and the device, but this comes at the expense of programmer productivity. As a general solution, Unified Virtual Memory (UVM) has become available since CUDA 8.0 and the Pascal architecture (2016). The key idea behind UVM is that programmers no longer need to think about GPU and CPU memory as two distinct memory spaces. Instead, UVM creates a pool of managed memory that is shared between the CPU and GPU, and is accessible from both with memory pages being migrated on-demand by the CUDA runtime system automatically. Even though UVM dramatically reduces developer effort in regards to managing data movement, it can cause a significant hit in performance.

While the size of data needed by scientific applications and machine learning/data analytics workloads increase at a rapid pace, the memory available on GPUs increase at a much more modest pace. For example, NVIDIA's Tesla V100 "Volta" GPUs have only 16 GBs of device memory available; one could have up to 80 GBs of memory on the newest generation NVIDIA A100s, but even 80 GBs is not enough for all application use cases. To illustrate this issue, take for instance the Many Fermion Dynamics - nuclei (MFDn) code, which is a quantum many-body code based on the configuration interaction model. MFDn 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 [2], [23]. When an application's working set size exceeds the device memory size, the resulting data movement becomes a critical design and performance bottleneck [32].

In this paper, we present a tiling-based task-parallel sparse linear algebra framework, named DeepSparseGPU, which

§This publication describes work performed at the Georgia Institute of Technology and is not associated with Amazon.

aims to make it easy to develop sparse solvers for large problems on GPU-accelerated architectures. Due to the irregular data access patterns and low arithmetic intensities associated with sparse matrix computations, achieving a high percentage of the peak processor performance, especially on GPUs, is challenging. These challenges are further exacerbated due to the increasing number of hardware accelerators by different vendors (i.e., NVIDIA, AMD, Intel, etc.). The main goal of the DeepSparseGPU framework is therefore to ease the development of performant and portable sparse solvers. Hence, building upon the DeepSparse framework [1] which runs on CPUs, DeepSparseGPU leverages OpenMP's target offload functionality. As we aim to optimize DeepSparseGPU for GPUs, several changes were made to the original DeepSparse framework, especially to support applications with memory requirements that exceed the available device memory capacity. We adopt the same task-based tiling approach which serves the double purpose of enabling data locality optimizations and facilitates the management of application data so that it can be processed in batches that fit into the device memory. For this, DeepSparseGPU automatically generates and expresses the entire computation as a task dependency graph (TDG), which is then executed using OpenMP's tasking and target offloading functionalities.

A memory manager developed as a part of DeepSparseGPU keeps track of the data residing on host and device memories, and automatically migrates data between the two whenever needed. Given the data dependencies between computational tasks, DeepSparseGPU employs a topological-sorting based heuristic to minimize the data movement and increase application performance.

The paper is organized as follows. In Section II, we describe the related work on efforts to manage the GPU memory efficiently in different application domains, application experience using directive-based programming models, and accelerating LOBPCG solver using GPUs. In Section III, we describe both the design of our proposed tile-based framework and the main data structures used in our memory manager with an illustrative example. In Section IV, we present performance results obtained on the Cori-GPU platform. Finally, Section V summarizes our conclusions and plans for future work.

II. Related Work and Our Contribution

Given the importance of sparse solvers in scientific computing and machine learning, several optimization techniques have been proposed for sparse matrix-vector multiplication (SpMV) on GPUs [6], [7], [14], [31]. However, the performance of SpMV is bounded by memory bandwidth [29]. Since sparse matrix-matrix multiplication (SpMM) has a much higher arithmetic intensity than SpMV and can efficiently leverage the performance benefits of GPUs, SpMM-based solvers have recently drawn significant interest in scientific computing in the form of block linear solvers and eigensolvers [2]. As such several groups have studied the optimization of the SpMM kernel on GPUs [5], [15], [26], [30]. On the solver side, Anzt et. al [5] optimize the performance of SpMM using ELLPACK format [4] and compare the performance of their implementation with the multithreaded CPU implementation of LOBPCG provided in the BLOPEX [20] package. Dziekonski et. al [13] implement LOBPCG method to find eigenvalues in electromagnetics analysis. They use an inexact nullspace filtering approach in their implementation.

As can be seen, most prior work on accelerating iterative solvers for GPUs has focused only on optimizing the SpMV/SpMM kernels with a few exceptions. In this work, we present a holistic framework that includes all computational kernels required for block eigensolvers (LOBPCG is used as a case study). Another distinguishing aspect of our work compared to the work reviewed above is the support we provide for applications with memory demands significantly larger than the available device memory capacity.

In addition to the support for large applications, we adopt a directive-based programming model to achieve portability. OpenMP and OpenACC have recently emerged as directive-based programming models to accelerate applications on GPUs. Several existing work studied the efficacy of the GPU offload support in OpenMP and/or OpenACC in the context of individual kernels [8], [12], [27], mini-applications [21] or proxy applications [9]. In this regard, our evaluation of OpenMP's offloading support in the context of a real eigensolver applied to matrices from several domains also constitutes a niche. In doing so, for the benefit of the community, we also discuss the compiler support issues we faced, as OpenMP offloading implementations are constantly evolving and are advertised as having only partial support in different compilers.

In evaluating DeepSparseGPU, we compare against a baseline Unified Virtual Memory (UVM) implementation using GPU-accelerated cuSPARSE and cuBLAS library kernels. UVM was introduced to provide a single, unified virtual address space to applications for accessing CPU and GPU memory. Data can be automatically migrated at an individual page level between host memory and device memory. As we aim to do for sparse solvers, UVM greatly simplifies general-purpose GPU programming because the same pointer to data can be used on both the host and the device side. The current data communication strategy in UVM is full page migration. Oversubscription of GPU memory and system-wide automatic memory operations are enabled by full-page migration [17], [32]. There are some performance risks in UVM regarding large page migration. NVIDIA evaluated UVM performance using the PGI OpenACC compiler in [11] by creating UVM versions of OpenACC applications in the SPEC ACCEL 1.2 benchmark suite. They found that the UVM versions ran at 95% of the performance of the original explicit data management versions when running the applications on the Piz-Daint supercomputer. Our work also compares UVM against explicit data management and considers problems whose memory requirements significantly exceed the device's memory capacity. The performance of oversubscribing UM is evaluated in [18]. The authors find that UM can be up to 2x slower than explicit data management in several applications on an x86+V100 system.

Consequently, our contributions can be summarized as follows:

- We demonstrate that a complex block eigensolver can be implemented efficiently using OpenMP target offload directives by minimizing data movement between CPU and GPU. We obtain up to 2.93x speedup and up to 2.18x less data transfer between CPU and GPU over a well-optimized UVM based implementation.
- We designed and developed a Memory Manager (MM) that automatically keeps track of data on both CPUs and GPUs, and migrates it whenever needed. Memory manager helps to execute problem sizes that exceed device memory.
- We use topological sort on the DAG to get a custom schedule of the computing tasks. Empirically, we find that topological sort helps minimize data movement compared to the pain baseline schedule.
- We share and discuss our experiences and issues that we faced during the development process so that the community could benefit from it.

III. DeepSparseGPU Overview

Figure 1 illustrates the architectural overview of the DeepSparseGPU framework. DeepSparseGPU consists of two major components: i) Primitive Conversion Unit (PCU), which provides a front-end to domain scientists to express their application, such as the LOBPCG solver, at a high-level and generates the task dependency graph (TDG); and ii) Task Executor (TE), which receives a DAG from the PCU and performs a topological sort on the DAG to get a task schedule. The task executor then launches proper kernels that correspond to computational nodes in the TDG according to the task schedule.

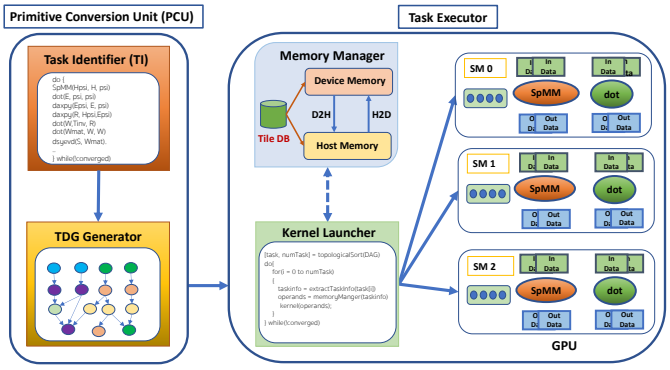


Fig. 1: Schematic overview of DeepSparseGPU framework.

A. Primitive Conversion Unit (PCU)

The Primitive Conversion Unit (PCU) is composed of two parts: i) Task Identifier and ii) Task Dependency Graph (TDG) Generator.

1) Task Identifier (TI): DeepSparseGPU provides an application programming interface (API) for developers, which is a combination of the GraphBLAS interface [16] for sparse matrix related operations, the BLAS/LAPACK interface [3], [22] for vector and occasional dense matrix related computations and custom kernels. While the GraphBLAS interface is implemented as tiled operations in the DeepSparseGPU library, optimized BLAS/LAPACK libraries (which are typically available on HPC systems; otherwise needs to be provided by the user) are used for dense vector and matrix operations. As DeepSparse [1] and DeepSparseGPU are used to implement new solvers, we extend the library with support for necessary operations not found in GraphBLAS, BLAS or LAPACK. Through this interface, developer can express their algorithms at a high level without having to worry about architectural details (e.g., memory hierarchy) or parallelization considerations (e.g., determining the tiles, tasks resulting from tiling and their scheduling).

Task identifier parses the given application code to identify the specific BLAS/LAPACK and GraphBLAS function calls and the input/output of each function call. It then passes this information to the local task dependency graph generator. The parsed data is maintained in TI as an unordered map of (`Key`, `Value`) pairs in a data structure named ParserMap. Each `Key` is defined by three pieces of information: The operation code (`opCode`), the operation id (`id`) to distinguish between multiple calls to the same function in different parts of the code, and a relative ordering info (`timestamp`) used to infer the input/output dependencies between different operations. ParserMap uses two helper data structures called Keywords and idTracker to uniquely identify all kernels in a given solver code. The `Value` object corresponding to each `Key` stores the input and output variable information along with their sizes for that function call.

2) Task Dependency Graph Generator (TDGG): The output of Task Identifier (TI) is a dependency graph at a very coarse-level, i.e., at the function call level. Tasks must be generated at a much finer granularity for efficient parallel execution of larger problems on the GPU by carefully planning the data movement between the CPU and GPU. This is accomplished by the Task Dependency Graph Generator (TDGG), which goes over the input/output data information generated by TI for each function call and starts decomposing/tiling these kernels and data structures.

In DeepSparseGPU, the decomposition into finer granularity tasks starts with the first function call involving the sparse matrix (or matrices) in the solver code, which is typically an SpMV, SpMM, or SpGEMM operation. For example, SpMM is the main sparse matrix kernel in the LOBPCG solver. We use compressed sparse row (CSR) matrix format to store the sparse matrix. First, the TDGG decomposes the sparse matrix using a 1D decomposition. Then, the main sparse matrix kernel decomposition induces the decomposition of all kernels above and below (i.e., the full solver) the sparse matrix operation. TDGG also generates the dependencies between individual fine-granularity tasks by examining the function call

dependencies determined by TI as part of the task dependency graph generation procedure. The resulting task dependency graph generated by TDGG is a directed acyclic graph (DAG) representing the data flow in the solver code where vertices denote computational tasks, incoming edges represent the input data, and outgoing edges represent the output data for each task. TDGG uses an instance of *Taskinfo* [listing 1] as the name of each vertex in TDG to store the necessary information to execute each task in the DAG. TDGG also labels the vertices in the TDG with the estimated computational cost of each task, and the directed edges with the name and size of the corresponding data, respectively. During execution, such information can be used for load balancing and/or ensuring that active tasks fit in the available GPU memory.

Listing 1: TaskInfo Structure

```
1 struct TaskInfo
2 {
3   int opCode; //type of computational kernel
4   int numParamsCount;
5   int *numParamsList; //tile id, matrix dimensions etc.
6   int strParamsCount;
7   char **strParamsList; //i.e. buffer name
8   int taskID; //which operation of a specific kernel
9 }
```

Listing 2: An example pseudocode

```
1 dmmadd(E, F, G, m, n); // G = E + F
2 SpMM(X, G, D, m, n); // D = X * G
3 cblas_dgemm(CblasRowMajor, CblasTrans,
      CblasNoTrans, n, n, m, 1.0, D, n, G, n, 0, R,
      n); // R = D'G
```

3) Illustrative Example of PCU: We use the simple code snippet provided in Listing 2 to demonstrate how the PCU operates. As TI parses the sample solver code, it discovers that the `dmmadd` call on the first line corresponds to a matrix addition operation (available as a custom kernel in the DeepSparseGPU library), the second line is a sparse matrix vector block multiplication (`SpMM`) that is part of GraphBLAS and the `cblas_dgemm` at the end is an inner product of two vector blocks (X^TY) as defined in the BLAS library. These function calls, their parameters as well as dependencies are captured by TI in the ParserMap, Keywords, and idTracker data structures as shown in Table I.

Data Structure	Content
ParserMap	<{dmmadd, 1, 1},{<E, F>, <G>, <m, n, k>}>
	<{SpMM, 1, 2},{<X, G>, <D>, <m, m, n>}>
	<{XTY, 1, 3},{<D, G>, <R>, <m, n, n>}>
keyword	<dmmadd, SpMM, XTY>
idTracker	<1, 1, 1>

TABLE I: Major data structures after parsing third line.

Based on the parsed data from TI, TDGG determines the tiling of operand data structures for each kernel (based on a tile size chosen by the user), as well as their origins (where each data tile is coming from). TDGG then builds the DAG of each computational kernel individually and appends it to the global TDG with proper dependencies. In the resulting TDG, each vertex is named using an encoding of its corresponding TaskInfo structure and tile id. Figure 2 shows the task dependency graph of the solver code in Listing 2, assuming m = 100, k = 8, n = 8, $tilesize$ = 50, $m/tilesize$ = 2, so each input matrix is partitioned into 2 chunks.

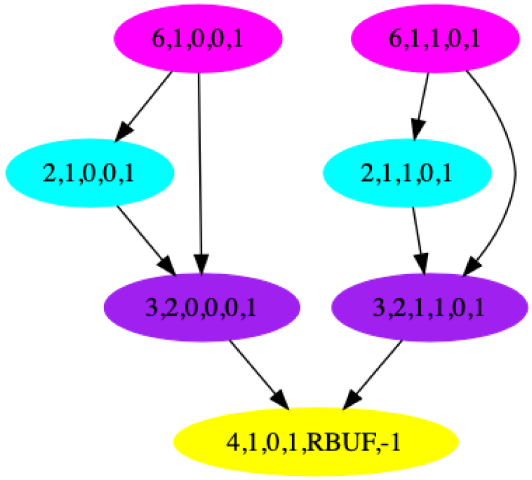


Fig. 2: Task graph for the psudocode in listing 2.

B. Task Executor (TE)

The Task Executor (TE) is composed of two parts as well: i) Kernel Launcher, and ii) Memory Manager.

1) Kernel Launcher (KL): The KL receives a DAG from the PCU that represents the full execution flow of the given application. Then, KL performs a depth-first topological sort based on the dependencies in the DAG to get an execution schedule of tasks and saves the tasks in an array of TaskInfo structures according to the task schedule. The topological ordering of the DAG respects the data dependencies and guarantees the numerical correctness of the given application. Furthermore, it is expected that the depth-first strategy used in the topological ordering would help minimize the data movement between the CPU and GPU by placing tasks with data dependencies together in the task schedule.

Once the topological ordering of the DAG is done, the KL picks each node from the task array one by one and launches a kernel for them on the GPU. While GPUs are the preferred target in DeepSparseGPU, we realize that some kernels may not be suitable for GPU acceleration or they simply may not be available in existing GPU libraries. DeepSparseGPU is designed to be flexible in such cases, and can execute kernels which do not have a corresponding GPU implementation on the CPU.

The KL first extracts the task information from the task name. Then, it consults with the Memory Manager (MM) to check if the operands of the task are ready on the GPU memory (or on the CPU memory). Once all operands are ready, the kernel can be launched.

As shown in listing 3, we use `target teams distribute parallel for` directives to offload com-

putations to the GPU. The `nowait` clause is used for asynchronous execution on the GPU. Whenever possible, the `collapse` clause is used to merge nested loops to obtain higher degrees of parallelism in the kernels. All directives use the `depend` clause to respect the input/output dependencies among tasks.

Listing 3: Kernel Launcher skeleton code in OpenMP

```
// main.cpp
[task, numTask] = topologicalSort(DAG);
#pragma omp parallel
{
  #pragma omp master
  {
    while(!converged)
    {
      for(i = 0 ; i < numTask ; i++)
      {
        // extract task information
        taskinfo = extractTaskInfo(task[i]);
        // preparing operands on gpu (or cpu) using
        // Memory Manager
        operands = memoryManger(taskinfo);
        // launch the proper kernel on GPU
        // (or CPU) based on taskinfo
        kernel(operands);
      }
    }
  }
}

// an example gpu application kernel, dst = src1 + src2
void dmmadd(double *device_memory, int offset1 /*src1*/,
      int offset2 /*src2*/, int offset3 /*dst*/, int
      blksz, int col)
{
  int sz = blksz * col;
  #pragma omp target is_device_ptr(device_memory)\
  depend(in: device_memory[offset1:sz], device_memory[
        offset2:sz])\
  depend(out: device_memory[offset3:sz]) nowait
  #pragma omp teams distribute parallel for collapse(2)
  for(int i = 0; i < row ; i++)
    for(int j = 0 ; j < col ; j++)
      device_memory[offset3 + i * col + j] =
            device_memory[offset1 + i * col + j] +
            device_memory[offset2 + i * col + j];
}
```

2) Memory Manager (MM): The memory manager (MM) is a low-overhead runtime system which is responsible for keeping track of all data tiles on the host and device memory. As illustrated in Fig. 1, the KL consults with MM before launching each kernel on the GPU (or CPU). MM makes sure that the latest copy of each of the operands of a kernel is available on GPU (or CPU) before launching the kernel. MM moves data tiles between host and device automatically whenever needed and implements a First-In-First-Out (FIFO) eviction policy to evict blocks when the device memory becomes full.

Data Structures used in MM: The MM maintains a few data structures to fulfill its operations. Figure 3 shows the important data structures that MM maintains to manage all data tiles between the host and device. The purposes of these data structures are as follows:

- ***device_memory:*** This is an array of double-precision numbers that spans the total available device memory in our target machine. We reserve a small fraction of the device memory to store some matrices if the computational model requires it. The rest of the device memory is allocated as this bigger array. We divide the *device_memory* array into blocks of memory chunks according to user-provided computational granularity. If the size of the *device_memory* array is m bytes and memory granularity is β bytes, then we divide the full *device_memory* array into $n = \lceil \frac{m}{\beta} \rceil$ memory blocks. We use *omp_target_alloc* function in our OpenMP implementation to allocate the needed memory on device.
- ***memory_view:*** This is an array of n elements (*<data structure name, tile id>* pair). It contains name and id of the data tiles stored in the i-th memory block of *device_memory* array.
- ***memory_map:*** This is an unordered map that holds detailed information on the data tiles stored in *device_memory* in the form of (Key, Value) pairs. The *Key* is a *<data structure name, tile id>* pair so that each tile of all data structures associated with the computation has a unique key. The *Value* is an array of 4 numbers *(<index, # of occupied blocks, size, isModified>)*. The first number in the Value field is the *index* of *device_memory* where the data block (*<data structure name, tile id>*) is stored on the device. The second number in the array is the number of memory blocks occupied on the device by the stored tile. Depending on the size of the tile, it may require more than one device memory block for storage. The third element of the array is the actual size of the tile that is stored at $index$ location on *device_memory*. The fourth element of the array indicates whether the tile data is modified on the device or not. If *isModified = 0*, then the matrix tile data is not modified on the device. If *isModified = 1*, then it indicates that the matrix tile data is modified on the device. MM always checks the *isModified* field before evicting a matrix tile from the device memory. If tile data is modified, the MM copies the latest matrix tile data to the proper host location before evicting it from the device memory.
- ***evictionQueue:*** The MM implements a First-In-First-Out

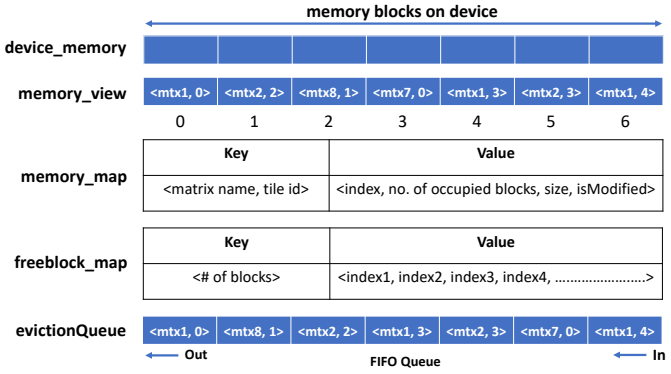
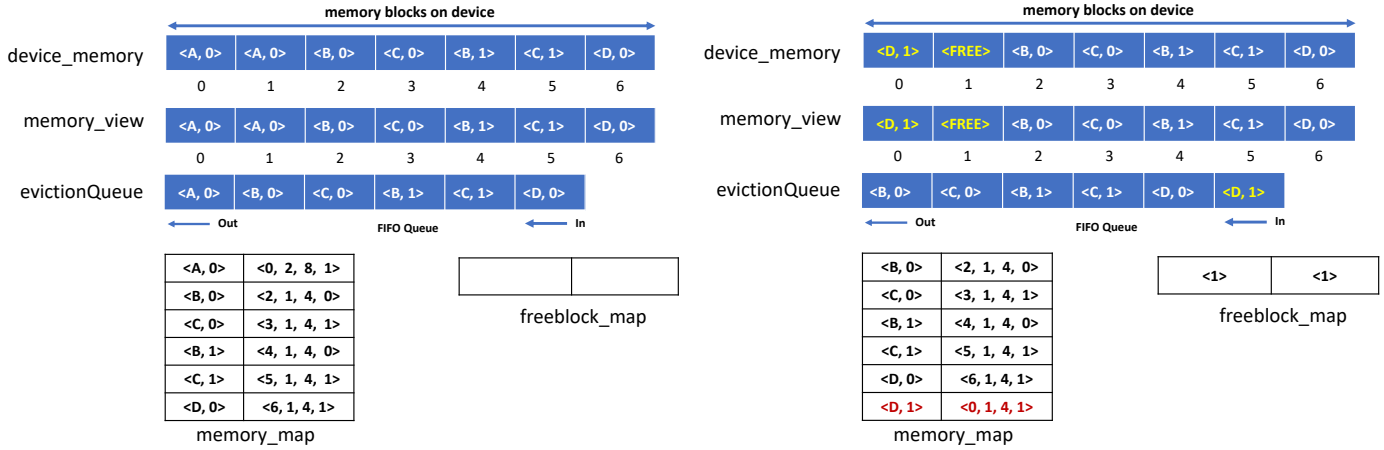


Fig. 3: Data structures used in managing tile information in Memory Manager.



(a) Device memory becomes full

(b) Evicting $<A,0>$ and copying $<D,1>$

Fig. 4: Illustrative example of how MM works

(FIFO) eviction policy using *evictionQueue* to execute larger sparse matrix problems that exceed device memory. As MM moves a tile from host to device, it adds the $<$*data structure name, tile id*$>$ of that block to the end of *evictionQueue*. MM always tries to evict from the beginning of *evictionQueue*.

- ***freeblock_map:*** This is an unordered map that keeps track of the free memory blocks on the *device_memory* array. The *Key* of the *freeblock_map* is the number of contiguous free memory blocks on the *device_memory* array. The *Value* of the *freeblock_map* is an array of start indices of free memory blocks on the *device_memory* array. The MM starts tracking the free memory blocks once the eviction policy is activated, and MM always tries to copy a matrix block from the host to the free memory blocks on the device. This helps minimize memory fragmentation on the device.

Eviction/Replacement Policy: Data movement between the host and device becomes the main performance bottleneck when application working sets exceed the physical memory capacity of the device. The MM uses a software-managed First-In-First-Out (FIFO) eviction policy to efficiently manage data movement between the host and device algorithm. From the beginning of the execution of the application, MM adds a matrix block at the end of the *evictionQueue* whenever it moves a matrix block from host to device; an eviction is performed when the device memory becomes full. MM evicts the front element of the *evictionQueue* to make space for newly required data on the device. If the data tile at the front of *evictionQueue* is an operand of the current operation, it is removed from the front and reinserted at the back of *evictionQueue*.

List of MM methods: MM has multiple methods to perform its functionality. These methods are internal to DeepSparseGPU and are not to be used by a user. Here are the list of important methods of MM:

- ***isOnDevice(mtx, tile_id)***: This function returns true if the *tile_id*-th block of *mtx* matrix is available on device, otherwise it returns false.
- ***copyToDevice(mtx, tile_id)***: This function is used to copy the *tile_id*-th block of *mtx* matrix from host to the suitable available memory blocks on device memory. It throws an exception if there is not enough space or any suitable memory blocks on the device. We use *omp_target_memcpy* function to copy data between host and device in DeepSparseGPU.
- ***reserveOnDevice(mtx, tile_id)***: This function reserves spaces for the *tile_id*-th block of *mtx* on the device. This function is beneficial when any operation produces new data. In such cases, we do not need to copy the output matrix tile from host to device; reserving sufficient device space for that matrix tile is enough in this case. This function throws an exception if there is not enough space or no suitable memory blocks exist on the device.
- ***copyToHost(mtx, tile_id)***: This function is used to copy the *tile_id*-th block of *mtx* matrix from the device to the host. It throws an exception, if it is unable to copy the matrix block. We use *omp_target_memcpy* function to copy data between from device to host.

How MM works: We provide an example in Figure 4 to demonstrate the operations of MM. Let us assume we have have 4 matrices (*A, B, C and D*) associated with a computation and we have a total of 7 memory blocks on the device. Each matrix is tiled into 2×1 blocks in this example. Each tile of matrix A requires 2 memory blocks on the device, whereas tiles for other matrices require 1 memory block on the device. Due to the nature of the computation, let us assume that the order of moving matrix tiles to device is - $\{<A, 0>, <B, 0>, <C, 0>, <B, 1>, <C, 1>, <D, 0>, <D, 1>\}$. As shown in Figure 4a, device memory becomes full before moving the last block ($<D, 1>$) in due to the limited device memory capacity. Figure 4a shows the status of all the data structures

of MM at this stage and MM activates the eviction policy, as device memory is full now. The MM needs to move *<D, 1>* tile to device. In order to do so, MM evicts the front of *evictionQueue* which is *<A, 0>* matrix tile. *(<A, 0>)* occupies 2 memory blocks on device and *<D, 1>* requires only 1 memory block to be stored on device. So MM copies *<D, 1>* at 0 index of *device_memory* array and adds index 1 in the *freeblock_map*. MM also updates *memory_view*, *map_map* and *evictionQueue* accordingly. As shown in Figure 4a, *isModified = 1* in *memory_map* for *<A, 0>* which means *<A, 0>* is modified on the device. MM copies the latest value of *<A, 0>* from device to host before evicting it in Figure 4b. As memory manager mostly performs insertion and look up operations on an unordered map, an array or a FIFO queue that typically take O(1) time on average, it incurs a negligible execution overhead.

IV. Performance Evaluation

A. Experimental Setup

We conducted all of our experiments on the Cori-GPU testbed at the National Energy Research Scientific Computing Center (NERSC) [24]. Each compute node has two 20-core Skylake processors clocked at 2.4 GHz and 8 NVIDIA Tesla V100 "Volta" GPUs with 16 GB 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. Cori-GPU provides extensive software environments to compile OpenMP (and OpenACC) programs. We used the Cray Compiler Environment (CCE) at version 9.1.0 to compile our software. Our work used the classic CCE compiler as opposed to the newer LLVM/Clang-based compiler that is also available in the package. The GPU accelerated cuSPARSE and cuBLAS libraries provided with CUDA v11.1.1 are used in our baseline implementation. We used thread affinity to bind threads to cores and we use 20 CPU threads and 1 GPU (1 CPU socket + 1 GPU) for our experiments to avoid NUMA issues. We take a full Cori-GPU node for our experiments in order to avoid noisy environment created by sharing the same node with other users.

1) Benchmark Application: We demonstrate the performance of the DeepSparseGPU framework on the LOBPCG algorithm which is an important eigensolver for large-scale scientific computing applications [19]. We give the pseudocode for the LOBPCG solver in Algorithm 1. It involves kernels with high arithmetic intensities such as SpMM and several level-3 BLAS kernels. The total memory needed for block vectors Ψ, R, Q and others can easily exceed the space matrix $\hat{H}$ takes up. Figure 5 shows a sample task dependency graph for the LOBPCG algorithm for a toy problem generated by TDGG where the sparse matrix is decomposed into 2×1 tiles. Clearly, minimizing the data movement between CPU and GPU to obtain an efficient LOBPCG implementation is non-trivial.

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: Ψ_0, a block of vectors of dimensions of $N \times m$
Output: Ψ and E such that $\|\hat{H}\Psi - \Psi E\|_F$ is small, and $\Psi^T\Psi = I_m$

1 Orthonormalize the columns of Ψ_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 $\Psi \leftarrow \Psi_{i+1}$

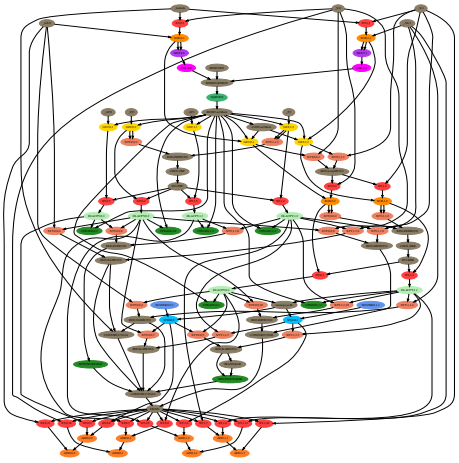


Fig. 5: A sample task graph for the LOBPCG algorithm using a small sparse matrix.

2) Dataset: We selected 11 square matrices with varying sizes, sparsity patterns, and domains from the SuiteSparse Matrix Collection in addition to the Nm7 matrix, which is from a nuclear shell model code (see Tab. II and III) [10]. Matrices in Table II are used to evaluate bigger problem sizes that do not fit in GPU memory. The bigger problem sizes range from 32.06 GB to 84.54 GB. Matrices in Table III are used to evaluate problem sizes that fit in the GPU memory. The smaller problem sizes range from 2.64 GB to 13.87 GB. Performance data for LOBPCG is averaged over five iterations.

3) Baseline Implementation: We compare the performance of DeepSparseGPU with a baseline implementation that we call *Libcsr_UM*, which is an implementation of the LOBPCG solver using GPU accelerated cuSPARSE and cuBLAS libraries for SpMM (with CSR storage of the sparse matrix), inner product, and linear combination kernels. The application

TABLE II: Matrices used to evaluate problem size $>$ 16 GB.

Matrix	#Rows	#Non-zeros	Problem Size (GB)
Nm7	4,985,944	648,890,590	32.06
nlpkkt200	16,240,000	448,225,632	44.98
nlpkkt240	27,993,600	760,648,352	77.32
it_2004	41,291,594	1,150,725,436	64.18
sk_2005	50,636,154	1,949,412,601	54.38
webbase_2001	118,142,155	1,019,903,190	84.54

TABLE III: Matrices used to evaluate problem size $<$ 16 GB.

Matrix	#Rows	#Non-zeros	Problem Size (GB)
inline_1	503,712	36,816,342	2.64
dielFilterV3real	1,102,824	89,306,020	6.07
Flan_1565	1,564,794	117,406,044	5.22
HV15R	2,017,169	281,419,743	8.29
Bump_2911	2,911,419	127,729,899	8.63
Nm7	4,985,944	648,890,590	13.87
nlpkkt160	8,345,600	229,518,112	12.94

kernels are executed on the GPU using OpenMP directives. Application kernels include tall skinny matrix operations such as addition, subtraction, element-wise multiplication, etc. We use NVIDIA's unified memory technology (using the `cudaMallocManaged` function) in this implementation. As such, the runtime system automatically takes care of the data movement between the host and device.

4) Incremental Implementation of DeepSparseGPU: We incrementally implemented DeepSparseGPU. We implemented two intermediate working versions of our tile-based DeepSparseGPU framework. We adopted two different data movement schemes in these two intermediate versions, which helped us design and develop our memory manager. We include the performance data from these intermediate versions to show the margin of improvement using the latest version of DeepSparseGPU:

- *DeepSparse_UM:* Like DeepSparseGPU, we tile all computational kernels and express them as DAG in DeepSparse_UM. But instead of using MM, we rely on using unified memory for automatic data movement between the host and the device. All associated matrices are allocated using `cudaMallocManaged`. They are accessed using `is_device_ptr` in target offload pragmas which allows the kernels to treat the data tiles as device pointers, and the runtime system automatically makes the data available on the GPU whenever needed.
- *DeepSparse_MAP:* This version relies on OpenMP `map(to: <list>)`, `map(tofrom: <list>)` and `map(from: <list>)` clauses for transferring data between host and device. Each offload pragma moves the necessary data tiles on-the-fly right before launching its GPU kernel.

B. LOBPCG Evaluation

Our performance comparison criteria include the amount of Host-to-Device (H2D), Device-to-Host (D2H) data transfer, and average execution time per iteration among Libcsr_UM, DeepSparse_UM, DeepSparse_MAP and DeepSparseGPU implementations for the LOBPCG solver. The Memory Manager quantifies and keeps track of the H2D and D2H data transfer in the DeepSparseGPU framework. We use the NVIDIA *nvprof* profiler tool to measure the amount of H2D and D2H data transfer while using unified memory. In DeepSparse_MAP, we manually measure the total H2D and D2H data transfer based on the data mentioned in the *map* clauses. The performance of DeepSparseGPU, DeepSparse_UM, and DeepSparse_MAP depends on the tile size. Choosing a small tile size creates a large number of fine granularity tasks. This means we have to launch more kernels on GPU. There are overheads associated with launching kernels on GPU. Also, a smaller tile size may lead to poor H2D and D2H transfer rates. Increasing the tile size reduces GPU execution overhead as GPU execution heavily depends on data parallelism. Therefore, we use the tile size as an optimization parameter based on matrix dimensions and sparsity patterns. We experimented with different tile sizes of 32K, 64K, 128K, and 256K. We report the results of the best-performing tile size for DeepSparseGPU, DeepSparse_UM, and DeepSparse_MAP. In Libcsr_UM, we are not required to tile the input matrices as we used unified memory with cuSPARSE and CUBLAS library kernels from CUDA and OpenMP target offloaded application kernels.

1) Data Movement between Host and Device: Figure 6 shows the H2D data transferred for all four versions of the LOBPCG algorithm. LOBPCG is a complex algorithm, as it has several different kernel types, and its task graph results in a vast number of tasks to be launched on GPU for each iteration. As can be seen in Figure 6, DeepSparseGPU always transfers less data from the host to the device compared to the other three versions. DeepSparseGPU transfers 1.18x - 1.94x less H2D data compared to Libcsr_UM version except for the sk_2005 matrix. DeepSparseGPU also transfers 1.25x - 2.59x less H2D data compared to DeepSparse_UM version.

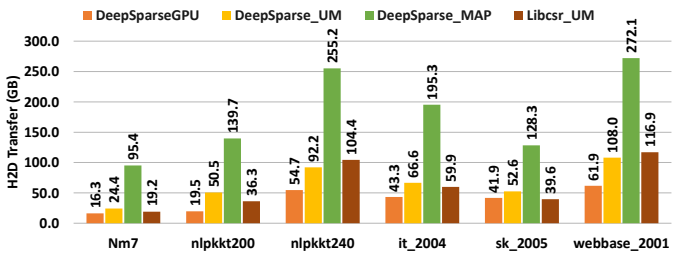


Fig. 6: Average H2D data transfer per iteration.

Figure 7 shows the D2H data transferred for all four versions of the LOBPCG algorithm. As can be seen in Figure 7, DeepSparseGPU always outperforms the other three versions when it comes to the amount of D2H data transfer. DeepSparseGPU transfers 1.84x - 2.69x less D2H data compared to Libcsr_UM, 2.39x - 3.70x less D2H data compared to DeepSparse_UM and 1.95x - 3.12x less D2H data compared to DeepSparse_MAP. Considering the total amount of data transfer (H2D + D2H), DeepSparseGPU transfers 1.39x - 2.18x less data compared to Libcsr_UM, 1.89x - 2.79x less

data compared to DeepSparse_UM and 2.92x - 5.01x less data compared to DeepSparse_MAP.

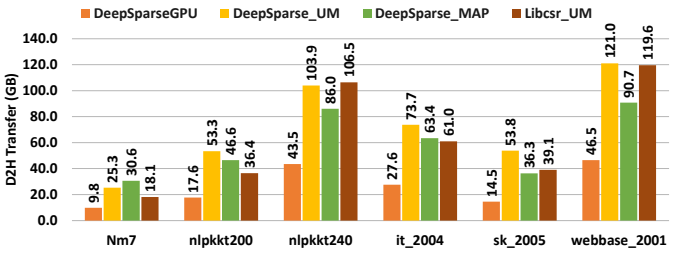


Fig. 7: Average D2H data transfer per iteration.

From Figure 6 and 7, we can see that the main reason why DeepSparseGPU transfers significantly less data is due to the explicit data management by the Memory Manager based on a good task scheduling heuristic. The task scheduling heuristic helps to maximize the utilization of the data while it resides in GPU memory. DeepSparse_MAP is the worst regarding H2D and D2H data transfer performance among all four versions. This is expected because DeepSparse_MAP moves all inputs and outputs of a kernel in both H2D and D2H directions during each kernel launch. Figure 6 and 7 show that the design of our task scheduling scheme and memory manager is robust and helps to minimize the data movement between host and device.

2) Execution Time: As can be seen in Figure 8, the significant reduction in H2D and D2H data transfer of DeepSparseGPU over the other three versions leads to a significant execution time speedups in general. In particular, DeepSparseGPU achieves 1.21x - 1.38x speedup compared to Libcsr_UM version except for Nm7, nlpkkt200 and sk_2005 matrices. We further investigated the reasons for the slower execution of DeepSparseGPU for these matrices. We found that our CSR format-based custom SpMM runs significantly slower than the highly optimized `cusparseSpMM` (a cuSPARSE library routine) used for SpMM operation in the Libcsr_UM version. To be specific, our custom SpMM routine runs 1.47x - 2.58x slower in case of these 3 matrices compared to the `cusparseSpMM` routine in Libcsr_UM.

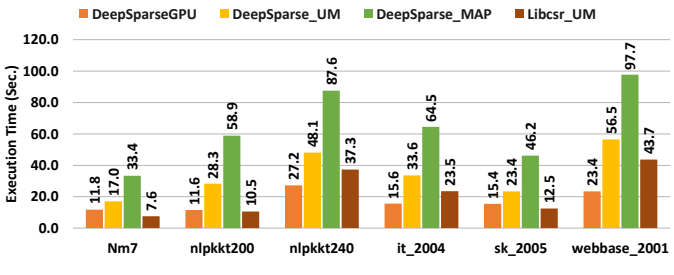


Fig. 8: Average execution time per iteration.

We experimented with OpenMP target offload and CUDA implemented versions of the most expensive kernels. Figure 9 shows the performance comparison between OpenMP target offload implementation and CUDA implementation (cuBLAS, cuSPARSE) of the most expensive kernels including inner product ($X'Y$), linear combination (XY), SpMM and column-wise matrix reduction. The input size of the matrices used in all of these kernels was less than 16 GB. As we can see, the CUDA versions of these kernels are 1.67x - 12.79x faster compared to the OpenMP target offload versions. We use *nvprof* to get the execution time of cuBLAS and cuSPARSE library kernels.

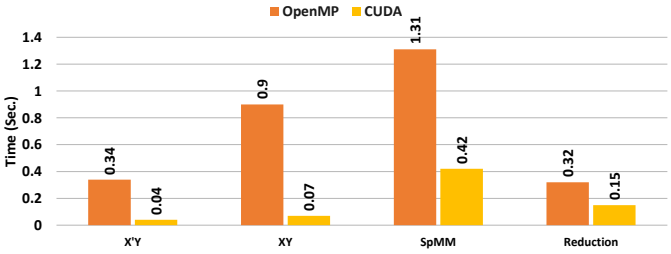


Fig. 9: OpenMP vs CUDA execution time comparison for most expensive kernels.

3) Effect of Pinned Memory: It is important to note that even if our DeepSparseGPU transfers less H2D and D2H data, the overall H2D data transfer rate in DeepSparseGPU is 4.49 GB/sec, whereas the average H2D rate is 6.34 GB/sec in Libcsr_UM for all test matrices. The overall D2H data transfer rates are more striking, Libcsr_UM (12.13 GB/sec) achieves 2.82x higher data transfer rates DeepSparseGPU (4.29 GB/sec). Despite this lower data transfer rate and the worse SpMM performance mentioned above, DeepSparseGPU runs faster than Libcsr_UM with the Unified Memory. We had used pageable memory with DeepSparseGPU for the results shown in Figure 6, 7 and 8, therefore we also experimented with pinned memory to achieve a better bandwidth and see its effect on execution time. Figure 10 shows the execution time performance comparison of DeepSparseGPU using pinned memory with the results shown in Figure 8. As can be seen, we achieve 1.45x - 1.98x execution time speedup using pinned memory with DeepSparseGPU compared to its pageable memory version and up to 2.93x speedup compared to Libcsr_UM.

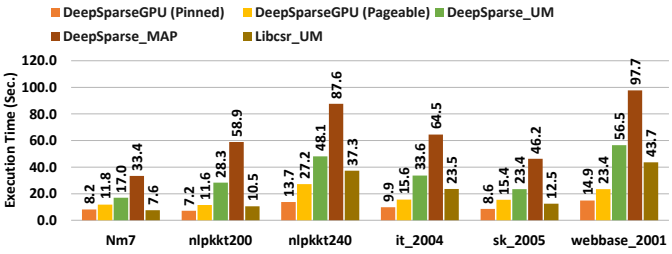


Fig. 10: Performance of DeepSparseGPU using pinned memory

4) Comparison with a CPU version: Figure 11 shows the execution time performance comparison between DeepSparseGPU (using pinned memory) and the baseline version running on CPU for bigger problem sizes. The CPU baseline implementation uses thread-parallel Intel MKL Library calls (including SpMM) with CSR storage of the sparse matrix. As we can see, DeepSparseGPU is running slower in comparison. However, it should be noted that even with the pinned memory optimization above, we get only about 12 GB/sec bandwidth between the CPU and GPU using a PCIe express 3.0 interconnect. In contrast, the bandwidth between the CPU and DRAM is approximately 90 GB/sec.

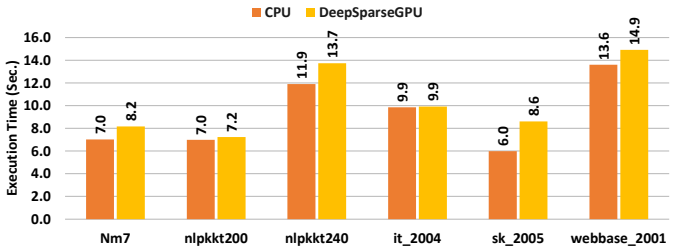


Fig. 11: CPU vs DeepSparseGPU execution time comparison (problem size $>$ 16 GB)

Figure 12 shows the theoretical execution time of DeepSparseGPU with different types of interconnect such as PCIe-Gen3 (16 GB/Sec), PCIe-Gen4 (32 GB/Sec), PCIe-Gen5 (64 GB/Sec), and NVLINK-4 (450 GB/Sec) [25]. For this experiment, we took the total compute time of V100 from Figure 11 and assumed it remains the same for each configuration. We calculated the data movement time by dividing the total amount of data transfer in Figure 11 by 80% of the peak per-direction bandwidth for each type of interconnect and added it to the total compute time to get the total theoretical execution time. As we can see from Figure 12, the theoretical execution time with PCIe-Gen3 is still on par with the CPU time. However, we can see that the theoretical time of DeepSparseGPU with PCIe-Gen4, PCIe-Gen5 and NVLINK-4 beat the CPU time.

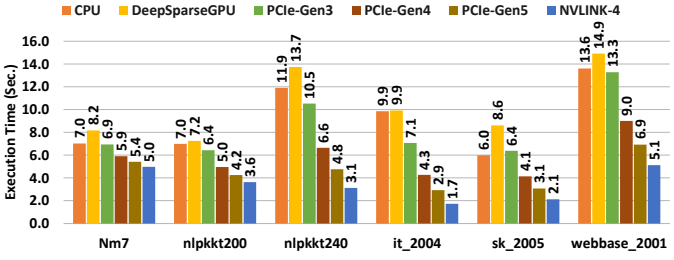


Fig. 12: CPU vs DeepSparseGPU execution time comparison with different CPU-GPU interconnect

The above simulation indicates that with the availability of platforms with faster interconnects (some such platforms currently exist, but those hardware have not been available for us to experiment with), DeepSparseGPU would have real merit in accelerating actual machine learning or scientific computing workloads. Figure 13 which shows the execution time performance comparison between DeepSparseGPU when the application working set fits into GPU memory (Table III) provides further evidence in this direction.

As can be seen, DeepSparseGPU achieves 1.77x - 4.87x speedup compared to the CPU baseline implementation when the total memory footprint is less than 16GB, i.e., when data movement is not a big bottleneck.

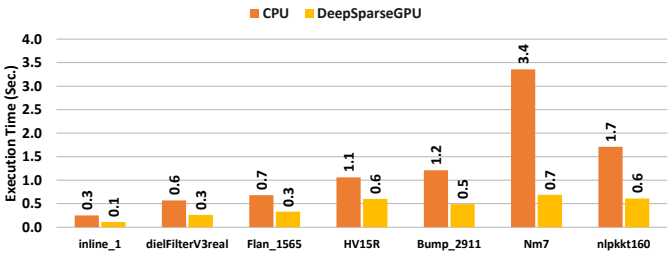


Fig. 13: CPU vs DeepSparseGPU execution time comparison using problem size $<$ 16 GB

V. Conclusion and Future Work

In this work, we introduced a tiling-based sparse solver framework for heterogeneous architectures that aims to minimize data transfer between host and device to achieve better performance. We showed that our framework transfers significantly less data between host and device. Our framework also improves the execution time over the UM-based baseline implementation using pinned memory. In our future work, we will focus on improving the efficiency of data transfers in DeepSparseGPU and optimizing the performance of sparse matrix kernels. To this end, we plan to design and implement a DAG partitioner that would help minimize the data movement between CPU and GPU. We also plan to use 2D decomposition of the SpMM operation, which would expose more parallelism in the computation. We also plan to explore and experiment with CUDA Graphs, which seems a good fit for our design.

Acknowledgment

This work was in part supported by the NSF under awards CCF-1822932, OAC-1845208 and CCF-1919021, as well as the US Department of Energy, Office of Science under the award DE-SC0018083 (NUCLEI SciDAC-4 Collaboration). 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.

References

[1] Md Afibuzzaman, Fazlay Rabbi, M Yusuf Özkaya, Hasan Metin Aktulga, and Umit V Çatalyürek. Deepsparse: A task-parallel framework for sparsesolvers on deep memory architectures. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 373–382. IEEE, 2019.

[2] Hasan Metin Aktulga, Aydin Buluç, Samuel Williams, and Chao Yang. Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1213–1222. IEEE, 2014.

[3] Edward Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, S Hammerling, Alan McKenney, et al. Lapack users' guide, vol. 9. *Society for Industrial Mathematics*, 39, 1999.

[4] Hartwig Anzt, Stanimire Tomov, and Jack Dongarra. Implementing a sparse matrix vector product for the sell-c/sell-c-σ formats on nvidia gpus. *University of Tennessee, Tech. Rep. ut-eecs-14-727*, 2014.

[5] Hartwig Anzt, Stanimire Tomov, and Jack Dongarra. Accelerating the lobpcg method on gpus using a blocked sparse matrix vector product. In *Proceedings of the Symposium on High Performance Computing*, pages 75–82. Society for Computer Simulation International, 2015.

[6] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis*, page 18. ACM, 2009.

[7] Jee W Choi, Amik Singh, and Richard W Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *ACM sigplan notices*, volume 45, pages 115–126. ACM, 2010.

[8] Christopher Daley, Hadia Ahmed, Samuel Williams, and Nicholas Wright. A case study of porting hpgmg from cuda to openmp target offload. In *International Workshop on OpenMP*, pages 37–51. Springer, 2020.

[9] Joshua Hoke Davis, Christopher Daley, Swaroop Pophale, Thomas Huber, Sunita Chandrasekaran, and Nicholas J Wright. Performance assessment of openmp compilers targeting nvidia v100 gpus. *arXiv preprint arXiv:2010.09454*, 2020.

[10] Tim Davis, Yifan Hu, and Scott Kolodziej. The suitesparse matrix collection. http://faculty.cse.tamu.edu/davis/suitesparse.html, 2018.

[11] Sebastien Deldon, James Beyer, and Douglas Miles. OpenACC and CUDA Unified Memory. In *Cray User Group (CUG)*, May 2018.

[12] Jose Monsalve Diaz, Swaroop Pophale, Kyle Friedline, Oscar Hernandez, David E Bernholdt, and Sunita Chandrasekaran. Evaluating support for openmp offload features. In *Proceedings of the 47th International Conference on Parallel Processing Companion*, pages 1–10, 2018.

[13] A Dziekonski, M Rewienski, Piotr Sypek, A Lamecki, and Michał Mrozowski. Gpu-accelerated lobpcg method with inexact null-space filtering for solving generalized eigenvalue problems in computational electromagnetics analysis with higher-order fem. *Communications in Computational Physics*, 22(4):997–1014, 2017.

[14] Michael Garland. Sparse matrix computations on manycore gpu's. In *Proceedings of the 45th annual Design Automation Conference*, pages 2–6. ACM, 2008.

[15] Changwan Hong, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Jinsung Kim, Süreyya Emre Kurt, Israt Nisa, Shivani Sabhlok, Ümit V Çatalyürek, Srinivasan Parthasarathy, and P Sadayappan. Efficient sparse-matrix multi-vector product on gpus. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 66–79. ACM, 2018.

[16] Jeremy Kepner, David Bade, Aydın Buluç, John Gilbert, Timothy Mattson, and Henning Meyerhenke. Graphs, matrices, and the graphblas: Seven good reasons. *arXiv preprint arXiv:1504.01039*, 2015.

[17] Youngsok Kim, Jaewon Lee, Donggyu Kim, and Jangwoo Kim. Scalegpu: Gpu architecture for memory-unaware gpu programming. *IEEE Computer Architecture Letters*, 13(2):101–104, 2013.

[18] Marcin Knap and Paweł Czarnul. Performance evaluation of unified memory with prefetching and oversubscription for selected parallel cuda applications on nvidia pascal and volta gpus. *The Journal of Supercomputing*, pages 1–21, 2019.

[19] Andrew V Knyazev. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM journal on scientific computing*, 23(2):517–541, 2001.

[20] Andrew V Knyazev, Merico E Argentati, Ilya Lashuk, and Evgueni E Ovtchinnikov. Block locally optimal preconditioned eigenvalue xolvers (blopex) in hypre and petsc. *SIAM Journal on Scientific Computing*, 29(5):2224–2239, 2007.

[21] V. G. Vergara Larrea, R. Budiardja, R. Gayatri, C. Daley, O. Hernandez, and W. Joubert. Experiences porting mini-applications to OpenACC and OpenMP on heterogeneous systems. In *Cray User Group (CUG)*, May 2019.

[22] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[23] Pieter Maris, H Metin Aktulga, Mark A Caprio, Ümit V Çatalyürek, Esmond G Ng, Dossay Oryspayev, Hugh Potter, Erik Saule, Masha Sosonkina, James P Vary, et al. Large-scale ab initio configuration interaction calculations for light nuclei. In *Journal of Physics: Conference Series*, volume 403, page 012019. IOP Publishing, 2012.

[24] Cori-gpu system configuration. https://docs-dev.nersc.gov/cgpu/.

[25] Nvlink-4 bandwidth. https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper.

[26] Gloria Ortega, Francisco Vázquez, Inmaculada García, and Ester M Garzón. Fastspmm: An efficient library for sparse matrix matrix product on gpus. *The Computer Journal*, 57(7):968–979, 2014.

[27] Fazlay Rabbi, Christopher S Daley, Hasan Metin Aktulga, and Nicholas J Wright. Evaluation of directive-based gpu programming models on a block eigensolver with consideration of large sparse matrices. In *International Workshop on Accelerator Programming Using Directives*, pages 66–88. Springer, 2019.

[28] Top500 supercomputers. http://www:top500:org.

[29] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2009.

[30] Carl Yang, Aydın Buluç, and John D Owens. Design principles for sparse matrix multiplication on the gpu. In *European Conference on Parallel Processing*, pages 672–687. Springer, 2018.

[31] Xintian Yang, Srinivasan Parthasarathy, and Ponnuswamy Sadayappan. Fast sparse matrix-vector multiplication on gpus: implications for graph mining. *Proceedings of the VLDB Endowment*, 4(4):231–242, 2011.

[32] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W Keckler. Towards high performance paged memory for gpus. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 345–357. IEEE, 2016.