Research Article

Performant Automatic BLAS Offloading on Unified Memory Architecture with OpenMP First-Touch Style Data Movement

Junjie Li[1]

1. Texas Advanced Computing Center, The University of Texas at Austin, USA

BLAS is a fundamental building block of advanced linear algebra libraries and many modern scientific computing applications. GPU is known for its strong arithmetic computing capability, and highly suited for BLAS operations. However, porting code to GPUs often requires significant effort especially for large complex codes or legacy codes, even for BLAS heavy applications. While various tools exist to automatically offload BLAS to GPU, they are often impractical due to the high costs associated with mandatory data transfers. The advent of unified memory architectures in recent GPU designs, such as the NVIDIA Grace-Hopper, allows cache-coherent memory access across all types of memory for both CPU and GPU, potentially eliminating the bottlenecks faced in conventional architectures. This breakthrough paves the way for innovative application developments and porting strategies. In this paper, building on my preliminary work[1] demonstrating the possibility of performant automatic *gemm offload, I extend the framework to all level-3 BLAS operations, and present SCILIB-Accel[2], a novel tool for automatic BLAS offload . SCILIB-Accel leverages the cache-coherent NVLink C2C interconnect in Grace-Hopper and introduces a Device First-Use data movement policy. This policy, inspired by the OpenMP First-Touch approach in multi-socket CPU programming, minimizes CPU-GPU data transfers for typical scientific computing codes. Additionally, utilizing the dynamic binary instrumentation technique, the tool intercepts BLAS symbols directly from a CPU binary, requiring no code modifications or recompilation. SCILIB-Accel has been evaluated using multiple quantum physics codes on up to a few hundred GPU nodes, yielding promising speedups. Notably, for the LSMS method in the MuST suite, a 3x speedup was

achieved on Grace-Hopper compared to Grace-Grace. SCILIB-Accel is the first tool to deliver practical, high-performance automatic BLAS offload for scientific applications.

Corresponding author: Junjie Li, nicejunjie@gmail.com

1. Introduction

The Basic Linear Algebra Subprograms (BLAS) serves as a building block for many scientific computing applications and forms the foundation for advanced linear algebra libraries such as LAPACK and ScaLAPACK. These libraries are extensively used in mathematical software like Mathematica and MATLAB, as well as in data science packages such as NumPy, and in computational chemistry and physics applications. Notably, BLAS is heavily utilized in quantum chemistry and quantum physics codes, as linear algebra is the natural language of quantum mechanics.

Modern general-purpose Graphics Processing Units (GPUs) are known for their exceptional arithmetic compute power. Their raw FP32 and FP64 compute capabilities significantly outpace those of CPUs, making GPUs an ideal platform for running BLAS-intensive applications. While all major GPU manufacturers provide highly optimized BLAS libraries, such as cuBLAS for NVIDIA GPUs and rocBLAS for AMD GPUs, these GPU libraries have slightly different interfaces and are not drop-in replacements for CPU BLAS libraries. More critically, using these GPU BLAS libraries, like any other GPU porting task, requires developers to manually manage data movement for optimal performance. Consequently, porting large codes, legacy codes and codes with complex workflow to GPU isn't trivial, sometimes daunting and requires significant investment of manpower. Furthermore, supercomputers are becoming increasingly GPU-centric due to the rapid advancement of GPUs and the surge of AI applications, this trend generates a pressing needs to port more scientific codes to GPU, but many researchers lack the expertise for GPU porting and face challenges securing funding for pure code-porting efforts. Given BLAS' central role and its extensive GPU support, there have been numerous attempts to automate GPU usage, as outlined in Section 2.2. However, limitations inherent to conventional GPU architectures often necessitate frequent data transfers between main memory and GPU memory, resulting in overheads that are unacceptable for practical use.

In recent years, GPU manufacturers have introduced highly innovative architectures featuring unified memory connected via cache-coherent interconnects, such as AMD's MI250X and MI300X GPUs with

Infinity Fabric and NVIDIA's Grace-Hopper with NVLink-C2C. Additionally, designs like AMD's MI300A Accelerated Processing Unit (APU) integrate CPUs and GPUs with a single type of memory. These innovations eliminate the constraints of conventional architectures and inspire new programming approaches that may make automatic offloading feasible.

In our previous work[1][2], we presented a proof-of-concept framework for doing symbol interception and replacement with Dynamic Binary Instrumentation (DBI), and preliminary test of automatically offload the *gemm (general matrix multiplication) routines, proofing that performat automatic BLAS offload is achievable. In this paper, I further extend the implementation to all level-3 BLAS calls, and focus on discussing a novel data management strategy named Device First-use policy. Inspired by the OpenMP First-Touch memory management approach in multi-socket CPU or NUMA programming, The Device First-Use policy is designed for CPU-GPU systems where data is moved to GPU memory (or general device memory) upon its first use by a GPU kernel. This approach is both simple and effective, minimizing data transfers in practical BLAS use cases. Tests across several BLAS-intensive scientific applications on up to hundreds of GPU nodes demonstrate significant speedups. Although all tests are done on the NVIDIA Grace-Hopper system, the methodology is generic for any CPU-GPU system with cache coherency.

The remainder of this paper is organized as follows: Section 2 reviews the NVIDIA Grace-Hopper unified memory architectures and related BLAS offloading work. Section 3 details the implementation of SCILIB-Accel. In Section 4, we apply the tool to two BLAS-intensive scientific computing codes on up to 200 Grace-Hopper nodes and discuss the results along with performance issues of the NVIDIA Grace-Hopper system. Finally, Section 5 concludes the paper.

2. Background and Related Work

2.1. Coherent memory in NVIDIA Grace-Hopper

In conventional architectures, GPU and host memory exist in separate memory space, preventing direct access between CPU and GPU memory. To partially address this limitation, CUDA 6 introduced managed memory, allowing a single memory address space that is accessible from any GPU or CPU. This system operates through implicit page migration triggered by page faults managed by the CUDA runtime. The NVIDIA Grace-Hopper superchip takes a step further by featuring closely integrated CPU and GPU units along with LPDDR5x and HBM3e memory subsystems, connected by the high-bandwidth and cache-coherent NVLink Chip-2-Chip (C2C) interconnect[3]. While Grace-Hopper maintains compatibility with traditional GPU memory management where GPU memory is exclusively managed by the GPU's memory management unit, it more importantly implements a single system-managed page table where both CPU and GPU can cache-coherently access all memory subsystems without page movement. In this unified memory architecture, the two types of memories appear as two NUMA domains, similar to memories in a two-socket CPU system.

Although the two memory subsystems can be cache coherently accessed by both CPU and GPU, the bandwidth varies significantly across different access patterns, as shown in Table 1. When the CPU accesses its local LPDDR5X memory, it achieves a descent bandwidth of over 400 GB/s. The GPU accessing its local HBM3 memory delivers even more impressive performance, reaching 3.6 TB/s. The NVLink-C2C interconnect provides 450 GB/s of bandwidth in each direction, which adequately supports the full bandwidth of LPDDR5X, allowing GPU access to remote LPDDR5X memory at 400+ GB/s. However, CPU access to HBM3 memory is substantially slower, achieving only approximately 140 GB/s. These bandwidth disparities indicate that data locality remains crucial for Grace-Hopper systems. Developers must carefully optimize data movement patterns to maximize performance, similar to conventional GPU architectures, rather than relying solely on coherent unified memory access for production workloads.

		LPDDR5	HBM3
CPU	Copy	446.46	145.56
	Mul	438.58	145.50
	Add	435.28	141.94
	Triad	418.22	141.94
GPU	Copy	406.69	3364.55
	Scale	406.65	3364.55
	Add	610.34	3668.78
	Triad	610.43	3679.50

Table 1. STREAM Bandwidth (GB/s)
on GH200 (120GB LPDDR5X model)

2.2. *Previous automatic BLAS offload attempts*

Numerous attempts have been made to automatically accelerate CPU BLAS calls since the early adoption of GPUs in HPC. Cray LIBSCI_ACC[4], available for over a decade, was deployed on the Titan supercomputer for NVIDIA Tesla K20 GPU, supporting selected BLAS, LAPACK, and ScaLAPACK routines for offload when the library module is loaded. Similarly, IBM ESSL is capable of automatically offload selected BLAS, LAPACK, and FFTW calls but requires the accelerated version of math library, libesslsmpcuda, is linked. NVIDIA's NVBLAS[5] serves as a drop-in replacement for CPU BLAS calls, allowing users to configure host BLAS libraries and selected routines for offload. By $LD\_PRELOAD$ NVBLAS, dynamically linked CPU BLAS is replaced without relinking. Unfortunately, the NVBLAS tool is heavily over-engineered, it uses the cuBLASXT as the backend instead of cuBLAS and has an acceptable overhead[1].

These tools make offload decisions at runtime based on workload sizes and handles data movement automatically. Overall, these libraries are tailored for conventional GPU architectures, and frequent data movement is unavoidable, therefore suffers poor performance for small and medium sized matrix math in real workloads.

3. Performant Automatic BLAS Offload

A basic workflow of automatic BLAS offload tool is illustrated in Figure 1 where an dgemm call in the caller code is intercepted and redirected to a BLAS wrapper that manages data movement and makes the GPU BLAS call. Conceptually this workflow contains two tasks: 1) intercept BLAS symbols and replace it with a BLAS wrapper where GPU BLAS is called, 2) manage data movement between CPU and GPU resident memories. In the following content, we will discuss how these tasks are implemented in SCILIB-Accel, most critially how the data movement can be optimized using a novel data movement strategy inspired by the OpenMP First-Touch data placement policy.

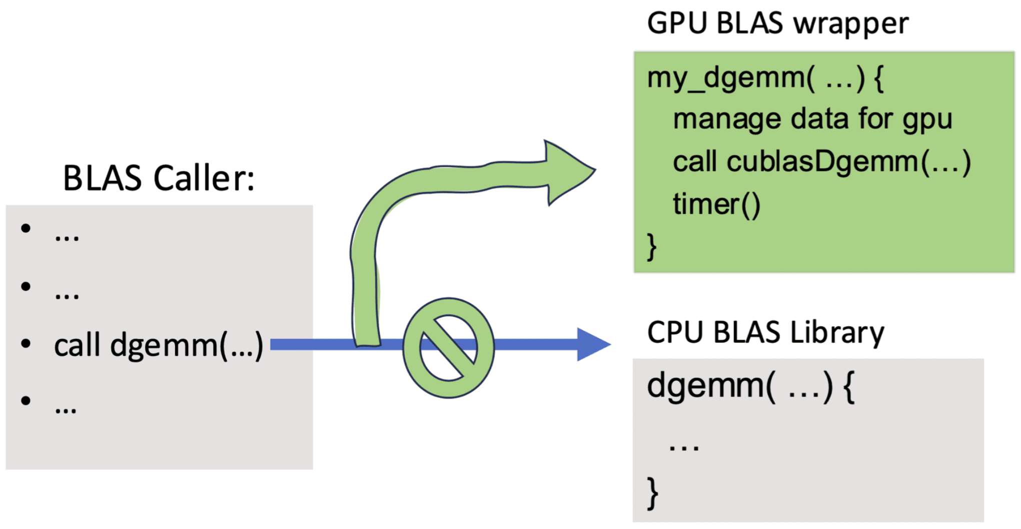


Figure 1. Workflow of Automatic BLAS Offload

3.1. Symbol Interception

Symbol interception is achieved via a trampoline-based Dynamic Binary Instrumentation (DBI) approach: a small piece of assembly code is inserted into the original function, enabling it to jump to a trampoline function. This trampoline function preserves the overwritten bytes by the extra jump instruction and executes customized code before returning to the original program.

For automatic BLAS offload, we intercept BLAS calls where the trampoline function (BLAS wrapper function) maintains the same signature as the original function. This results in minimal overhead, as the register data for function arguments remain undisturbed. This mechanism finds extensive use in profilers, and here we use the PEAK[6] lightweight profiler we have developed as a DBI framework, ensuring portability across various architectures including but not limited to x86 and ARM. Inside the DBI framework, the FRIDA-GUM[7] binary instrumentation backbend is chosen for simplicity and performance.

The SCILIB-Accel automatic offload library can be attached to user application by $LD\_PRELOAD$ the library. The SCILIB-Accel initialization function is placed into the .init_array section of Linux ELF to search and replace BLAS symbols along with other initialization tasks such as setting GPU memory pool, initialize cuBLAS, etc. Similarly, the SCILIB-Accel finalization function is inserted into the .fini_array section of the ELF to collect statistics and handle clean up tasks.

Note that the DBI approach applies to both dynamically and statically linked BLAS, while other tools like LIBSCI_ACC[4] and NVBLAS[5], which works by resolving runtime library dependency, only work for dynamically linked BLAS.

As DBI is widely used in profilers, the DBI symbol interception in SCILIB-Accel can cause conflicts when doing profiling. To be profiler friendly, an implementation of SCILIB-Accel using dlsym() to dynamically resolve shared library symbols to intercept BLAS calls is also provided. This approach works by defining the wrapper function to be the same name as the function to be intercepted, and by prepending the SCILIB-Accel library in $LD\_PRELOAD$, the symbols in the wrappers get used and the original function symbol can be obtained by looking up the next available symbol using dlsym(). This dlsym-based version have no issue being used with profilers, but can only intercept dynamically linked BLAS.

3.2. Data Movement Strategies

Managing data movement is often the most critical part of GPU porting as data transfer speed has been a limiting factor. It is even more so for developing automatic offload tool as the tool deals with pure CPU code that is totally untuned for GPU. Here, we discuss three data movement strategies that utilize different features of Grace-Hopper and it helps us better understand the challenge and opportunities for doing automatic offload on unified memory architecture.

*3.2.1. Strategy 1, **Mem-Copy***

This is the most intuitive strategy and used by other tools. The pseudocode is listed below. Upon interception of a BLAS call and code is redirected to a BLAS wrapper, the input matrices are copied from host memory to GPU memory, then resultant matrix is copied back after cuBLAS execution. then resultant matrix is copied back after cuBLAS execution. This strategy works on all GPUs including the conventional PCIe-based cards without needing unified memory capability, This strategy works on all GPUs including the conventional PCIe-based cards without needing unified memory capability, but at the cost of frequent data movement. This strategy can be effective for codes handling very large matrices where the compute time far exceeds the data transfer cost, but won't be useful for most codes that only runs small to medium sized matrices in practical use. This policy is studied here mostly for helping us understand the limitation of the conventional automatic BLAS offload approach in all other existing tools.

```
void my_dgemm(...) {
...
cudaMalloc(&device_A, size_A);
cudaMalloc(&device_B, size_B);
cudaMalloc(&device_C, size_C);

cudaMemcpy(device_A, host_A, size_A, cudaMemcpyDefault);
cudaMemcpy(device_B, host_B, size_B, cudaMemcpyDefault);

cublasDgemm(handle, TransA, TransB, m, n, k,
            &alpha, device_A, lda, device_B, ldb, &beta,
            device_C, ldc);

cudaMemcpy(host_C, device_C, size_C, cudaMemcpyDefault);
...
}
```

Listing 1. Pseudocode: Mem-Copy data movement policy

3.2.2. *Strategy 2, **counter-based migration***

Since CPU resident memory (LPDDR5X) and the GPU resident memory (HBM3) are physically unified with cache-coherent NVLink C2C, CPU matrix pointers can be passed directly to cuBLAS calls. One can also use numactl to force all memory resident on the HBM, but given the poor bandwidth of CPU access HBM, a performance penalty is expect. A new feature in Grace-Hopper designed to better serve the unified memory is the access counter on Hopper GPU, the CUDA runtime will automatically move memory page from LPDDR5X to HBM3 based on remote memory access detected by the counter. This counter-based migration mechanism can serve automatic offload when we pass CPU resident matrices to GPU kernel. A pseudocode example is shown below.

```
void my_dgemm(...) {
...
cublasDgemm(handle, TransA, TransB, m, n, k,
       &alpha, host_A, lda, host_B, ldb, &beta,
       host_C, ldc);
...
}
```

Listing 2. Pseudocode: counter-based data migration policy

*3.2.3. Strategy 3, **Device First-Use** policy*

One alternative way to look at the Grace-Hopper superchip is that is operates as a heterogeneous dual-socket system, with one socket being a CPU and the other a GPU. Its NUMA configuration also mirrors that of a dual-socket CPU system, where CPU-resident memory is assigned to NUMA 0 and GPU-resident memory to NUMA 1. The data management challenge here in CPU-GPU superchip is a reminiscence of the OpenMP First-Touch data placement policy used in CPU-CPU NUMA programming. While malloc calls can theoretically be intercepted to allocate memory directly for GPU access, these calls originate from the CPU binary, making it impractical to identify which memory regions will later be used by the GPU. As a result, implementing a GPU-first-touch policy is not feasible. To address this, I propose a **GPU First-Use policy**, where data is moved to the GPU the first time it is accessed by a CUDA kernel. More generally, this concept can be extended to a **Device First-Use policy**, applicable to any accelerator device with cache-coherent memory access. Table 2 summarizes the features of Device First-Use and highlights its similarities to OpenMP First-Touch.

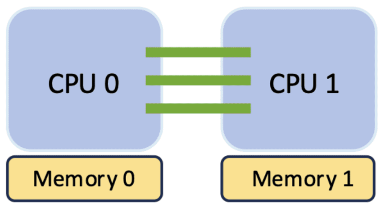
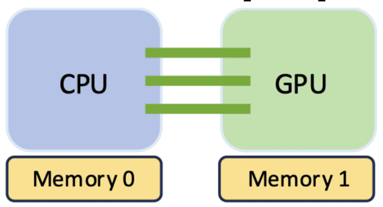
OpenMP First-Touch	**Device First-Use**
for dual socket CPU	for CPU-GPU superchip
	
allocate space on local memory of OpenMP threads upon initialization	migrate data from CPU memory to device memory upon first use by device (GPU) kernel
assume remote memory access is trivial (e.g. CPU 0 accessing CPU 1's memory)	assume remote memory access is trivial (e.g. CPU accessing GPU's memory)

Table 2. OpenMP First-Touch vs Device First-Use

Since the BLAS wrapper used to replace the CPU BLAS calls only knows the memory addresses of the matrices, implementing Device First-Use policy requires move data from the CPU resident memory to device (GPU) resident memory without reallocation or disrupting to the virtual memory address used by the CPU binary. This can be achieved by relocating the physical memory page and updating the page table to remap the virtual memory to new physical memory locations. Figure 2 illustrates the working of virtual memory in modern operation system and how physical memory page can be dynamically reassigned.

Remarkably, this complex process of physical page movement and remapping can be easily carried out using the Linux move_page() system call. This system call allows specifying a group of pages to move along along with their target NUMA destination (NUMA 1 for GPU on Grace-Hopper), simplifying the implementation significantly. Pseudocode of this implementation is presented below.

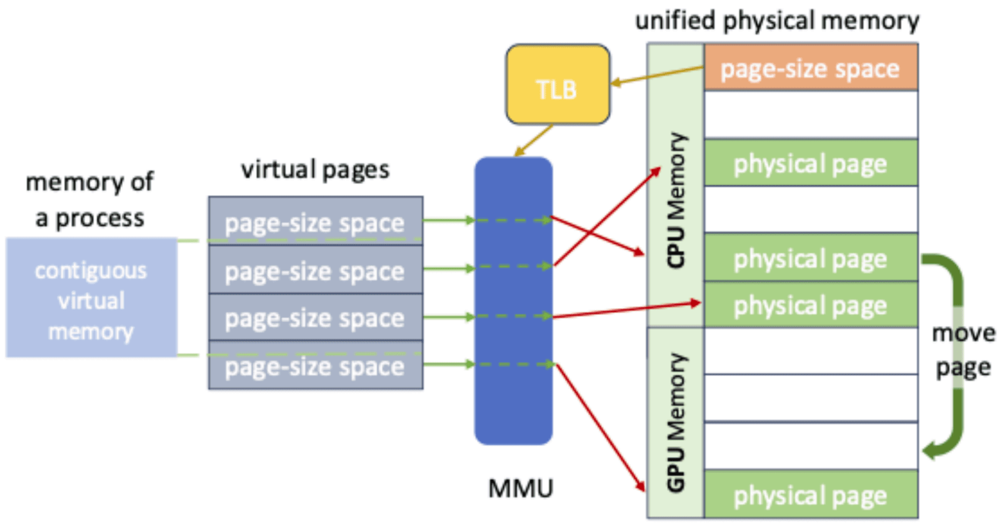


Figure 2. Virtual Memory and Physical Page Migration

```
void my_dgemm(...) {
...
if (host_A on NUMA 0) move_pages (host_A, to NUMA 1) ;
if (host_B on NUMA 0) move_pages (host_B, to NUMA 1) ;
if (host-C on NUMA 0) move_pages (host_C, to NUMA 1) ;

cublasDgemm(handle, TransA, TransB, m, n, k,
      &alpha, host_A, lda, host_B, ldb, &beta,
      host_C, ldc);
...
}
```

Listing 3. Pseudocode: Device First-Use policy

How does this policy improve data reusability for the GPU? To answer this, we need to examine how BLAS is typically used in scientific applications. In most cases, scientific problems are not solved with a single BLAS call. Instead, they involve a sequence of BLAS operations, such as $C = A \times B$ followed by $E = D \times C$, and so on. In these workflows, intermediate matrices (e.g., C) are frequently reused in subsequent operations. Data reuse is also particularly common in block matrix multiplications, where each block of a matrix is multiplied by multiple blocks from another matrix. Additionally, many scientific codes adopts an iterative approach, such as the self-consistent field process to solve partial differential equations in quantum chemistry or physics, the same matrices and memory pointers are re-used across all the iterations. All the data only need to be moved to the GPU once and can be re-used by subsequent iterations. All these common use cases conceptually justifies the appropriateness of Device First-Use policy.

3.3. Usage Instructions

The automatic BLAS offload wrappers are compiled and linked together as a shared library file (.so). All users need to do is to load the SCILIB-Accel library by $LD\_PRELOAD$ the .so file as shown below, then run their CPU binary as normal.

DBI version for general use:

```
export LD_PRELOAD=/path/scilib-dbi.so
run your CPU binary
```

DLSYM-based version when using with profilers:

```
export LD_PRELOAD=/path/scilib-dl.so
run your CPU binary
```

Several optional environmental variables can be set to tweak offload behaviors including:

1. data management strategies with Device First-Use as default
2. minimum matrix size that will allow a BLAS call to be offloaded, if matrices are small, then the BLAS call will stay on CPU. The default threshold is $N_{avg} > 500$ where N_{avg} is the average matrix size, the definition of N_{avg} is routine dependent. For general matrix multiplication routines C=AxB, $N_{avg} = (MNK)^{1/3}$ where dimensions of matrices A, B and C are $M \times K$, $K \times N$ and $M \times N$. The default threshold is a safe lower-bound based on preliminary dgemm testings on Grace-Hopper and can certainly be further fine-tuned for different kernels or precisions.
3. debug output levels.

4. Performance Testings and Discussion

In this section, I perform application tests using production HPC codes. Since quantum chemistry/physics are known to heavily relying on BLAS operations as linear algebra is the natural language of quantum mechanics, two codes, MuST and PARSEC, from this scientific domain are chosen, and they also part of the Characteristic Science Application (CSA)[8] efforts funded the

National Scientific Foundation (NSF) as representative workloads for the Leadership Class Computing Facility.

Different offload strategies are extensively tested to showcase the performance of Device First-Use data movement policy, and understand limitations of data movement approaches. In all test cases, the offload threshold is $N_{avg} > 500$ (see Section 3.3 for details) will be offloaded which is proven to be appropriate for these applications on Grace-Hopper.

It is important to emphasize that all performance comparisons presented here are based on an equal number of nodes: Grace-Grace nodes (two CPUs with 144 cores) and Grace-Hopper nodes (one CPU and one GPU). This approach ensures both simplicity and fairness in the analysis. Power and cost are also key factors in performance study. The power consumption of a Grace-Hopper node is approximately twice that of a Grace-Grace node under full load. Additionally, for each node-hour, the Service Unit (SU) charged to users for Grace-Hopper nodes is three times higher than that for Grace-Grace nodes. This charge rate reflects the costs associated with acquiring, maintaining, and supporting these different types of nodes.

4.1. Test Environment

All tests were conducted on the Vista[9] supercomputer at the Texas Advanced Computing Center (TACC). The system comprises 560 Grace-Hopper (GH200) nodes and 180 Grace-Grace nodes, configured as follows:

- **Grace-Hopper Nodes:** Each node is equipped with one 72-core Grace CPU (120 GB LPDDR5X memory) and one H100 GPU (96 GB HBM3 memory). The Grace-Hopper superchip is power-capped at 900W.
- **Grace-Grace Nodes:** Each node features two 72-core Grace CPUs with a combined Thermal Design Power (TDP) of 500W.

Notably, the 120 GB Grace CPU model has approximately 30% higher memory bandwidth compared to the 480 GB Grace model tested previously[1].

The nodes are connected with Infiniband interconnects in non-blocking fat-tree topology:

- Grace-Hopper nodes utilize full HDR (400 Gbps) configuration.
- Grace-Grace nodes are connected via split HDR (200 Gbps).

The software environment and configuration include:

- **GPU Driver:** Version 560.35.03
- **CUDA:** Version 12.6
- **Infiniband Firmware:** Version 28.41.1000
- **NVHPC Compiler Suite:** Version 24.9 (latest at the time of testing)
- **MPI:** HPCX (based on OpenMPI 4.1.7a1) provided with the NVHPC compiler suite
- **OS:** Rocky Linux 9.3
- **Linux Kernel:** Version 5.14.0-362.24.1.el9_3.aarch64+64k

For application testing, all CPU binaries were linked to the NVIDIA Performance Library (NVPL), which provides optimized BLAS, LAPACK, and ScaLAPACK routines. The SCILIB-Accel auto-offload tool utilized cuBLAS for GPU-accelerated operations. Both NVPL and cuBLAS were from the NVHPC 24.9 compiler suite.

4.2. Application Test 1: MuST

MuST (Multiple Scattering Theory)[10][11] is a package designed to perform electronic structure calculations, it solves the Kohn-Sham equation by solving the Green's function. The LSMS calculation method is designed for large systems with linear scalability to the system size. This method is won the 2009 Gordon Bell prize[12]. The code has a heavy dependency on BLAS operations, mostly zgemm and ztrsm, which often exceeds 80% of runtime on CPU. A major portion of the BLAS calls are from LAPACK routines zgetrs and zgetrf. Most matrices are squared or near-square shaped. MuST natively supports GPU offload, the native method implemented in MuST offloads the matrix inverse onto a GPU by calling cuSOLVER.

The test workload calculates energy of a CoCrFeMnNi supercell alloy using the LSMS method. Total atom number in the supercell is 5600 and concentration of each element is identical. The number of energy grid is 32. The calculation is limited to 3 self-consistent steps to reduce benchmark cost.

MuST is thoroughly tested at large scale on CPU, on GPU through automatic offload and native CUDA port. Table 3 summarizes the performance comparison of different run strategies on 50 Grace-Grace CPU nodes or 50 Grace-Hopper nodes. As mentioned before, the code strongly relying BLAS operations, the two major BLAS routines zgemm and ztrsm consumes about 2080s out of the total 2318s runtime. Using the native CUDA port from the developers, about 1.4x speedup is achieved comparing to CPU run. Surprisingly, all auto offload strategies are faster than the native CUDA code illustrating the complexity and challenge of CUDA programming, the developers will need spend substantial more amount porting efforts to polish their CUDA code. With the most basic data management method that copies (cudaMemcpy) matrices to/from GPU for every cuBLAS call, the total runtime reduces to 1098s, but 292s are spent in just moving the data around. This reaffirms that optimizing data movement is still critical on Grace-Hopper even with the fast NVLink-C2C interconnect. The counter-based migration approach works okay, total runtime is better than doing frequent cudaMemcpy. Note that the page migration time is included in the BLAS call time as the counter-based does the data movement automatically behind the scene with GPU kernel. The novel Device First-Use policy is substantially better other approaches, total runtime is reduced to 824s, about 2.8x faster than the CPU run, the total data movement time is reduced to just 4.8s. More analysis shows that under Device First-Use policy, each matrix that gets migrated to the GPU resident memory gets reused 780 times by subsequent BLAS calls. Such high level of data reusage is the key factor of the good performance we achieve here. Also note that the BLAS (zgemm and ztrsm) time in Device First-Use is much longer than the corresponding time in Mem-Copy policy, this is due to a performance issue of CUDA kernel accessing GPU memory allocated by system malloc, more details are discussed in Section 4.4.3. If such performance issue is resolved by NVIDIA, performance of automatic offload can be further improved.

Hardware	**Setup**	**Total runtime (s)**	**zgemm+ztrsm (s)**	**Data movement (s)**
CPU: Grace-Grace	CPU binary linked to NVPL	2318.4	2079.2	0
GPU: Grace-Hopper	native CUDA port	1685	N/A	N/A
	auto offload: Mem-Copy	1098	439.8	291.7
	auto offload: counter-based migration	858	616.0	included in BLAS
	auto offload: **Device First-Use**	**824**	**580.0**	**4.8**†

Table 3. MuST: Performance on GPU vs CPU using 50 Nodes

[†] *Matrices migrated to GPU resident memory are reused 780 times.*

Large scale strong scaling tests were also performed for this code. Table 4 shows the runtime of CPU run, GPU run with native CUDA code, and automatic offload using Device First-Use policy. Scaling range goes from 25 nodes to 200 nodes. Throughout the tests, the automatic offload approach is consistent 2x faster than the native CUDA code, and up to 3x faster than the CPU run, break even with the extra node-hour charge rate for GPU so users running on GPU not only gets faster time-to-solution but also more efficient in terms of energy consumption and cost. The strong scaling data is visualized in Figure 3, both the CPU run and automatic offload GPU run have excellent scability, very close to linear scaling at this wide test range.

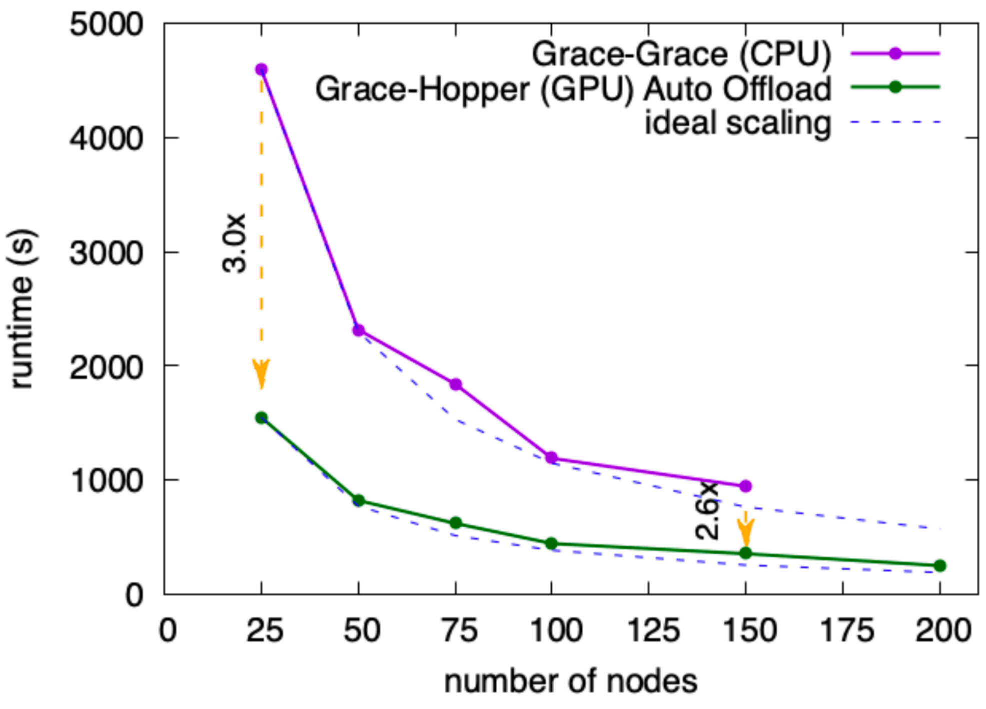


Figure 3. MuST: Strong Scaling Test for CPU Run and Automatic GPU Offload Run

Node Count	Total runtime (s)			Best GPU/CPU speedup
	CPU (Grace-Grace)	GPU: Native CUDA port	GPU, auto offload: Device First-Use	
25	4598.1	3223.3	1550.9	3.0x
50	2318.4	1685.2	823.8	2.8x
75	1842.6	1244.7	623.1	3.0x
100	1192.2	903.9	446.8	2.7x
150	947.0	673.6	357.5	2.6x
200	N/A †	493.9	253.3	N/A †

Table 4. MuST: Strong Scaling Performance on CPU vs GPU

† *Not enough CPU nodes available.*

4.3. Application Test 2: PARSEC

PARSEC (Pseudopotential Algorithm for Real-Space Electronic Calculations)[13][14] is a package designed to perform Density Functional Theory (DFT) calculations of solids and molecules. It solves the Kohn–Sham equations directly in real space, avoiding the use of explicit basis sets. Our benchmark case calculates energy of a Silicon nanocrystal $Si_{1947}H_{604}$, boundary sphere radius is set to 50 bohr, grid spacing is 0.9 bohr, the calculation is limited to two self-consistent field steps to reduce benchmark cost, but performance characteristics of a fully converged calculation is identical.

Historically, PARSEC has been a CPU-only code, relying heavily on ScaLAPACK. In typical use cases, dgemm calls from ScaLAPACK can account for over 50% of the runtime. With the help of SCILIB-Accel automatic offload, PARSEC runs on GPU for the first time with good performance. Tests were conducted to evaluate all offload strategies alongside a CPU-only baseline. All tests were performed on a single node: Grace-Grace for CPU runs and Grace-Hopper for GPU runs. The results are summarized in Table 5.

Hardware	Setup	Total runtime (s)	dgemm (s)	Data movement (s)
CPU: Grace-Grace	CPU binary linked to NVPL	415.1	270.1	0
GPU: Grace-Hopper	auto offload: Mem-Copy	425.7	12.4	220.7
	auto offload: counter-based migration	470.0	234.0	included in BLAS
	auto offload: **Device First-Use**	**220.3**	**29.1**	**1.3**[†]

Table 5. PARSEC: Performance on GPU vs CPU on single node

[†] *Matrices migrated to GPU resident memory are reused 570 times.*

In these tests, the Mem-Copy data policy resulted in runtimes slower than dual-CPU execution. The cudaMemcpy operations consumed 220 seconds, accounting for more than 50% of the total runtime — a significantly higher proportion than observed in the MuST test case. This is due to the fact that most matrices used in PARSEC are long skinny matrices rather than squared shape ones. For example, a common dgemm input in PARSEC is transA='T', transB='N', M=32, N=2400, K=93536, the extreme skinny shapes make the total byte size of the matrices much bigger than if square matrices are used in a calculation with equivalent computational workload. a calculation of identical compute workload but with all square matrices. This outcome again demonstrates that the conventional data movement strategy for automatic offload is impractically useful even on Grace-Hoper where NVLink-C2C transfer rate is 450 GB/s per direction, not to mention the PCIe based cards where PCIe Gen5 x16 can only do 64 GB/s. The counter-based data migration strategy performs even worse due to incompetent migration algorithm, see complete discussions of NVIDIA's migration issues later in Section 4.4.1.

Finally, the Device First-Use policy is able to efficiently manage the data movement, enabling significant performance gains. The total runtime is nearly 2x faster than the CPU run, with dgemm component achieving nearly 10x speedup compared to CPU run. Data transfer overhead is minimal, totaling just 1.3 seconds. The speedup is able to offset the extra cost from power hungry GPU. Again, data reuse is counted and for every matrix that is migrated to GPU resident memory, it is reused on average 570 times by subsequent dgemm calls.

4.4. Performance Issues with Grace-Hopper Relevant to Auto Offload

In this part of the paper, we discuss a few performance issues observed in the above application test, these issues reflect the immaturity of the software or hardware design in the current Grace-Hopper system, and could be fixed in the future to further improve performance and usability of automatic offload.

4.4.1. ***Counter-based page migration***

The Hopper GPU has an access counter monitoring remote memory access, and will migrate memory page from CPU resident memory to GPU resident memory. Due to lack of access counter on the Grace CPU, data on GPU will not be migrated back to CPU. The details of the migration criteria is unknown.

Since the data migrated to GPU will not be migrated back, automatic offload with counter-based migration should work very similar to manually implemented Device First-Use policy, but we've already seen the slow performance of the counter-based migration in application tests. Here I present a few simple dgemm test cases with different matrix sizes, and provide a deeper understanding of the issues with counter-based migration. In the following test, matrices A(M,K), B(K,N) and C(M,N) are allocated by malloc and initialized on CPU resident memory, then multiplication $C = A \times B$ is performed at least 5 times by passing these matrices to cublasDgemm, and the access-counter should allow the matrices to be migrated to GPU resident memory. The NUMA locations of the matrices are reported after each cublasDgemm call, and runtime for each call is reported. From the NUMA location, we can infer if the data is on the CPU resident memory (NUMA 0) or GPU resident memory (NUMA 1). Four sets of inputs with different matrix sizes and shapes are tested, all use FP64 precision.

When M=N=K=1000 are used, the sizes of A, B and C are all 8.0MB. All of them are successfully migrated to the GPU resident memory upon the first cublasDgemm call.

When the matrix dimensions becomes M=N=K=5000, the size of all the matrices are 200.0MB, The migration becomes unstable and inconsistent from run-to-run. The tests were run many times. In most cases, only matrices A and B are migrated to GPU in the first cublasDgemm call, while C stays on the CPU no matter how many more cublasDgemm call iterations are added. Occasionally, matrices A and B stays on the CPU side in the first cycle, and only get migrated to GPU after the second call. In all runs, matrix C is never migrated to GPU.

If the square matrix dimension is further increased to M=N=K=20000, matrix size becomes 3200.0MB each, only matrix A is migrated to GPU, while B and C are always on the CPU memory no matter how many more cublasDgemm cycles are added.

The case gets strange for non-square matrix dimensions. When M=32, N=2400 and K=93536, a matrix size commonly used in my PARSEC workload, the matrix size becomes 24.0MB for A, 1795.9MB for B, and 0.6MB for C. Throughout my test, only matrix A gets migrated to GPU, while B and C are always on the CPU. This is very counter-intuitive as one would expect the bigger matrix B should at least be moved as it generates the most amount of remote memory access, but NVIDIA's counter-based migration algorithm doesn't agree.

From these tests, we can see that NVIDIA's algorithm tends to migrate smaller data to GPU, this could be that the algorithm only makes the decision based on a single CUDA kernel call, i.e. moving the big data is costly comparing to that single CUDA kernel runtime, so it decides not to migrated disregard the fact that the big data gets accessed by GPU many more times later on. The current counter-based migration is unpredictable and inconsistent, NVIDIA should provide a simple way to disable it by users instead of requiring unload a kernel model by root.

4.4.2. *Impact of page sizes*

The Grace-Hopper platform, similar to all other ARM-based platform, supports two base page sizes 4KB and 64KB. Most of the NVIDIA's internal tests are done on 64KB page size which is also the recommended page size. At TACC, we have setup both page sizes for testings, and performance issues with the 64KB page size are revealed by comparing the test results.

Here we again run simple dgemm tests to understand the performance and issues. Notice that the aforementioned counter-based page migration mechanism doesn't work with 4KB page size, so I can measure the performance of GPU kernel running on LPDDR5X, while I can't do this when page size is 64KB as matrices are partially migrated to GPU as explained in Section 4.4.1.

Test results are summarized in Table 6. It can be seen that CPU accessing HBM3 memory is substantially slower under 64KB page than 4KB page for both problem sizes. There is a substantial difference even for CPU accessing LPDDR5 for the second workload, runtime with 64KB page is 15.8ms, much slower than the 10.9ms under 4KB page.

Page Size	Memory Type	CPU (72C)	GPU
		(dgemm)	(cublasDgemm)
Workload: M=2000, N=2000, K=2000; 96MB total			
4KB	LPDDR5X	5.1 ms	9.0 ms
	HBM3	5.3 ms	0.37 ms
64KB	LPDDR5X	5.1 ms	N/A†
	HBM3	10.0 ms	0.39 ms
Workload: M=32, N=2400, K=93536; 1820MB total			
4KB	LPDDR5X	10.9 ms	18.1 ms
	HBM3	15.5 ms	0.95 ms
64KB	LPDDR5X	15.8 ms	N/A†
	HBM3	23.2 ms	0.94 ms

Table 6. DGEMM Runtime with Unified Memory

† *Part of the data gets migrated to HBM3 and can't do full LPDDR5X run.*

4.4.3. GPU kernel accessing system allocated HBM

As seen above in application tests, the BLAS runtime under Device First-Use policy is noticeably slower than the BLAS running on the cudaMalloc memory created in the Mem-Copy policy. Further investigation shows CUDA kernel is slow when running on system malloc allocated HBM, unless the matrices are aligned to the page. Table 7 shows the difference with page size being 64 KB. All matrices are allocated by malloc, and pinned to HBM by using numactl -m 1 ./exe. When the matrices are aligned to page size, cublasDgemm speed can be up to nearly 50% faster than if data isn't aligned. The impact is more significant memory bandwidth bound kernels. The aligned performance is identical to the same CUDA kernel executing on cudaMalloc memory.

Application tests reveal that BLAS runtime performance under the Device First-Use policy is noticeably slower compared to BLAS operating on cudaMalloc memory used in the Mem-Copy policy.

Further investigation indicates that CUDA kernel performance is suboptimal when operating on system-allocated HBM3 via malloc unless the matrices are aligned to the page. Table 7 highlights the performance differences. The tests use 64KB page size. All matrices are allocated using malloc and pinned to HBM3 with the command numactl -m 1 ./exe. When matrices are page-aligned, the performance of cublasDgemm improves by nearly 50% compared to cases where the data is not aligned. This improvement is particularly pronounced for memory bandwidth-bound kernels. In the page-aligned case, the performance on HBM3 allocated by malloc matches that of the same CUDA kernel executing on cudaMalloc memory. The reason of such behavior is unknown, and it partially defeats the advantage of unified memory architecture.

problem size	unaligned	aligned
M=2000, N=2000, K=2000	0.39 ms	0.29 ms
M=32, N=2400, K=93536	0.94 ms	0.64 ms

Table 7. Impact of Memory Alignment on cublasDgemm Performance[†]

[†] *Tests were performed on 64KB page size, all memory allocations are by malloc.*

5. Conclusion

In this paper, we report a new tool that can intercept Level 3 BLAS symbols in a CPU code, and automatically perform GPU offload using GPU-enabled BLAS along with several choices of data management strategies, especially a newly proposed first-touch type of matrix migration scheme. The tool is fine-tuned to achieve minimum overhead and take full advantage of the new unified memory architecture with cache coherent NVLink C2C. Performance tests on BLAS heavy codes show significant speedup comparing to CPU code, and the tool far outperforms NVBLAS auto-offload tool provided by NVIDIA on not only Grace-Hopper but also on the conventional PCIe-based GPU. The tool is useful for users and code developers to quickly explore the potential benefits of using GPU and have a quick start on the new architecture.

In this paper, a previous proof-of-concept BLAS auto-offload prototype tool is further optimized and extended to all level-3 BLAS operations. The limitations and advantages of the memory subsystem in

NVIDIA Grace-Hopper architecture are outlined. To overcome the data transfer bottleneck between CPU and GPU, an OpenMP first-touch type of data management strategy (GPU first-use policy) is introduced and discussed in detail, the new policy minimizes the data transfer between CPU and GPU in practical scientific computing applications resulting in production level GPU performance for several quantum chemistry codes. Such data management policy can be universally applied for any GPU architecture allowing cache-coherent access to the GPU memory from CPU, eliminating the necessity of integrating a single small piece of HBM in an APU.

Acknowledgements

The author J. Li thanks Robert Henschel for teaching him about the automatic offload capability in Cray libsci library and OpenMP First-Touch about seven years ago when Li switched his career from quantum chemistry to HPC. This work is supported by the National Science Foundation through awards OAC-2402542, OAC-1854828, and OAC-2139536.

References

1. a, b, c, d *Li J, Wang Y, Liang X, Liu H (2024). "Automatic BLAS Offloading on Unified Memory Architecture: A Study on NVIDIA Grace-Hopper". In: Practice and Experience in Advanced Research Computing 2024: Human Powered Computing (PEARC '24). New York, NY, USA: Association for Computing Machinery. Article 47, 5 pages. doi:10.1145/3626203.3670561.*
2. a, b *Li J, Wang Y (2024). SCILIB-accel: automatic BLAS offload tool. Available from: https://github.com/nicejunjie/scilib-accel.*
3. ^ *NVIDIA (2023). NVIDIA GH200 Grace Hopper Superchip Architecture. Available from: https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper.*
4. a, b *Poxon H (2013). Introduction to the Cray Accelerated Scientific Libraries. https://www.olcf.ornl.gov/wp-content/uploads/2013/01/Scientific_Libs.pdf.*
5. a, b *NVIDIA (2024). NVBLAS documentation. Available from: https://docs.nvidia.com/cuda/nvblas.*
6. ^ *Wang Y, Li J (2023). "PEAK: a Light-Weight Profiler for HPC Systems" (SC-W '23). Association for Computing Machinery, New York, NY, USA, 677–680. doi:10.1145/3624062.3624143.*
7. ^ *Frida. Frida Gum [Internet]. n.d. Available from: https://github.com/frida/frida-gum. Accessed: 2024-12-27.*

8. ^*National Science Foundation (2021). Characteristic Science Applications for the Leadership Class Computing Facility. Available from: https://www.nsf.gov/awardsearch/showAward?AWD_ID=2139536&HistoricalAwards=false.*
9. ^*Texas Advanced Computing Center (2024). VISTA System at TACC. Available from: https://tacc.utexas.edu/systems/vista.*
10. ^*Wang Y, Stocks GM, Shelton WA, Nicholson DMC, Szotek Z, Temmerman WM (1995). "Order-N Multiple Scattering Approach to Electronic Structure Calculations". Phys. Rev. Lett.. **75** (15): 2867–2870. doi:10.1103/PhysRevLett.75.2867.*
11. ^*Eisenbach M, Larkin J, Lutjens J, Rennich S, Rogers JH (2017). "GPU acceleration of the Locally Selfconsistent Multiple Scattering code for first principles calculation of the ground state and statistical physics of materials". Computer Physics Communications. **211**: 2–7. doi:10.1016/j.cpc.2016.07.013.*
12. ^*Eisenbach M, Zhou C-G, Nicholson DM, Brown G, Larkin J, Schulthess TC (2009). "A scalable method for ab initio computation of free energies in nanoscale systems". In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09). New York, NY, USA: Association for Computing Machinery. Article 64, 8 pages. doi:10.1145/1654059.1654125.*
13. ^*Kronik L, Makmal A, Tiago ML, Alemany MMG, Jain M, Huang X, Saad Y, Chelikowsky JR (2006). "PARSEC – the pseudopotential algorithm for real-space electronic structure calculations: recent advances and novel applications to nano-structures". physica status solidi (b). **243** (5): 1063–1079. doi:10.1002/pssb.200541463.*
14. ^*Chelikowsky JR, Troullier N, Saad Y (1994). "Finite-difference-pseudopotential method: Electronic structure calculations without a basis". Phys. Rev. Lett.. **72** (8): 1240–1243. doi:10.1103/PhysRevLett.72.1240.*

Declarations

Funding: This work is supported by the National Science Foundation through awards OAC-2402542, OAC-1854828, and OAC-2139536.

Potential competing interests: No potential competing interests to declare.