九州大学学術情報リポジトリ
Kyushu University Institutional Repository

A Study on the Performance and Usability of Managed Memory and Unified Memory for Accelerating Numerical Calculation Program

Ohshima, Satoshi
Research Institute for Information Technology, Kyushu University

Ida, Akihiro
Research Institute for Value-Added-Information Generation, Japan Agency for Marine-Earth Science and Technology

Kawai, Masatoshi
Tohoku University, Cyberscience Center

Fukaya, Takeshi
Hokkaido University, Information Initiative Center

他

https://hdl.handle.net/2324/7420563

出版情報：IEEE International Symposium on Embedded Multicore Socs (MCSoC), pp.41-48, 2025-12-31. IEEE
バージョン：
権利関係：© 2025 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

KYUSHU UNIVERSITY

A Study on the Performance and Usability of Managed Memory and Unified Memory for Accelerating Numerical Calculation Program

Satoshi Ohshima
Research Institute for Information Technology
Kyushu University
Fukuoka, Japan
ORCID 0000-0003-4073-5688

Akihiro Ida
Research Institute for Value-Added-Information Generation
Japan Agency for Marine-Earth Science and Technology
Kanagawa, Japan
ORCID 0000-0001-7751-1093

Masatoshi Kawai
Cyberscience Center
Tohoku University
Miyagi, Japan
ORCID 0009-0003-1454-202X

Takeshi Fukaya
Information Initiative Center
Hokkaido University
Hokkaido, Japan
ORCID 0000-0003-1217-6444

Rio Yokota
Institute of Integrated Research
Institute of Science Tokyo
Tokyo, Japan
ORCID 0000-0001-7573-7873

***Abstract*—**
This paper presents a comparative study of NVIDIA Managed Memory and Unified Memory for GPU-accelerated numerical computing using CUDA Fortran. We focus on Block Low-Rank QR factorization implemented with CUDA Fortran, evaluating two approaches: memory control via compiler option and manual specification using variable qualifiers. Experiments on GH200 and H100 systems reveal that manual memory placement reduces runtime by up to 68% compared to compiler-option-based control. Unified Memory consistently outperforms Managed Memory on GH200, highlighting its suitability for integrated CPU/GPU architectures. Our contributions are: (1) a systematic evaluation of memory strategies for CUDA Fortran applications, (2) insights into productivity-performance trade-offs, and (3) practical guidelines for auto-tuning memory placement in HPC environments.

***Index Terms*—CUDA Fortran, Managed Memory, Unified Memory, BLR-QR**

I. INTRODUCTION

The deployment of high-performance computing systems utilizing GPUs is advancing, driven partly by the proliferation of technologies based on deep learning, such as Generative AI. Many GPU systems are now featured in the top rankings of the TOP500 list [1]. Even the supercomputer "Fugaku", which was a rare exception in the top rankings for not including a GPU, has announced plans to incorporate NVIDIA GPUs in its next-generation system [2], further emphasizing the importance of GPU utilization. Conversely, numerous programs have yet to be migrated to GPUs, making GPU porting a critical shared challenge across computational science and computer science.

In numerical calculation programs (scientific computing and numerical simulation), Fortran still holds substantial demand. Although Fortran support in GPU development environments is progressing, the importance of techniques and use cases that easily facilitate GPU optimization for Fortran programs continues to rise. For adapting CPU-oriented numerical programs to GPUs, optimizing major time-consuming calculation kernels is undoubtedly important. However, many programs spend a significant amount of time executing standardized numerical libraries like the BLAS and the LAPACK. For such programs, leveraging libraries such as cuBLAS allows for easy exploitation of GPU performance.

Nevertheless, programs with complex data structures, such as tree structures created through dynamic memory allocation, require significant effort for managing CPU-GPU data transfers, which also negatively impacts performance. To reduce this burden of memory management, NVIDIA GPUs offer mechanisms such as Managed Memory and Unified Memory, which can be utilized in conjunction with libraries like cuBLAS. Although sufficient documentation and samples for Managed Memory and Unified Memory are available, there is a pervasive perception that their performance degrades compared to manual memory control that does not utilize Managed Memory or Unified Memory. Consequently, there is an observable lack of shared evaluation and discussion concerning their application in real systems and applications. Studies addressing its impact on Fortran applications with complex data structures are important for computational science.

This study reports an instance where Managed Memory and Unified Memory were applied to a numerical calculation program (QR factorization using block low-rank matrices) employing MPI and CUDA Fortran. We evaluate the performance differences by testing multiple implementation methods for each memory type and summarize the associated programming challenges. Considering performance and productivity, the choice of memory type is also vital from the perspective of automatic tuning.

The remainder of this paper is structured as follows: Sec-

tion II summarizes Managed Memory, Unified Memory, and their usage methods as background technologies. Section III describes the target problem and the implementation details in existing work. Section IV compares program descriptions and execution times, providing evaluation and discussion. Section V mentions related work, and Section VI concludes the paper.

II. Background: Managed Memory and Unified Memory

This section outlines Managed Memory and Unified Memory and their utilization methods in CUDA Fortran. Both Managed Memory and Unified Memory are memory technologies in CUDA that enable transparent handling of device memory and host memory.

A. Managed Memory

Managed Memory is a memory management mechanism introduced in CUDA 6 and later. Memory regions allocated using Managed Memory are transparently accessible by both the host (CPU) and the device (GPU). Upon memory access, an on-demand page fault occurs, and the necessary page is transferred between the host and the device by the driver. This eliminates the need for explicit data transfer descriptions between CPU and GPU, contributing to program simplification. However, the overhead of page migration can sometimes lead to performance degradation. Nevertheless, automatic prefetching is performed during data transfer, and manual prefetch transfers using APIs are also possible. It is generally stated that performance comparable to manual data transfer can be achieved if the memory access pattern is simple.

B. Unified Memory

Unified Memory is a mechanism that unifies the virtual address space between the CPU and GPU and maintains memory coherence. In this study, we specifically focus on Unified Memory in processors where the CPU and GPU are integrated (Grace Hopper and Grace Blackwell architectures). In the Grace Hopper architecture (GH200), the CPU and GPU share a single virtual address space, enabling transparent memory access via a cache-coherent connection using NVLink-C2C. This eliminates the explicit data transfer between CPU and GPU previously required, allowing developers to write GPU programs without worrying about memory management. Furthermore, the ability for the GPU to directly access host memory greatly enhances the productivity of programs handling memory sizes that cannot be placed entirely in the GPU's device memory. When using Unified Memory, memory access performance varies depending on whether the target memory is on the host or the device, and whether it is accessed by the CPU or the GPU. Program optimization based on controlling the memory access order, similar to first-touch optimization for Non-Uniform Memory Access (NUMA) architectures in CPUs, is possible and can be managed via prefetch processing.

```
program main
  ...
  integer, managed :: marray(10)           ! Managed Memory
  real, unified, allocatable :: uarray(:)  ! Unified Memory
  ...
end program main
```

Fig. 1. Example of the Variable Qualifiers which indicate the kind of memory.

C. Usage in CUDA Fortran

There are generally two major methods for utilizing Managed Memory and Unified Memory in CUDA Fortran.

The first approach involves enabling the functionality by providing compiler options. Specifically, appending `mem:managed` to the `-gpu` option of the `nvfortran` compiler (included in the NVIDIA HPC SDK) automatically treats dynamically allocated memory as Managed Memory. Similarly, appending `mem:unified` treats all memory as Unified Memory. Additionally, options like `mem:unified:managedalloc` and `mem:unified:nomanagedalloc` exist; although official documentation does not detail them, setting `mem:unified:nomanagedalloc` appears to ensure that only Unified Memory is utilized (as noted in `nvfortran -help` output).

The second method is to explicitly define the memory attribute within the code using variable qualifiers. As illustrated in Figure 1, specifying the `managed` attribute allows for the explicit use of Managed Memory for arrays or variables, while the `unified` attribute enables the explicit use of Unified Memory in `nvfortran`. Standard Fortran `allocate/deallocate` functions can be used for dynamic memory allocation and deallocation of Managed Memory and Unified Memory. It is important to note that these attributes can be used even when compiler options are applied, and the explicitly specified attribute information takes precedence over the compiler options. Optimization through explicit memory specification can be an advantage of CUDA Fortran over OpenACC and OpenMP.

When Managed Memory is employed, data transfer is fundamentally triggered by page faults, but prefetch functions such as `cudaMemPrefetchAsync` can be used to transfer data in advance, which is effective for improving performance. While page faults do not occur when Unified Memory is used, prefetch functions can still explicitly place the target data into device memory, thereby helping to improve GPU memory access performance.

III. Target problem and prior research

A. BLR-QR

This study uses the QR factorization of Block Low-Rank (BLR) matrices (BLR-QR) as the target problem.

While there is significant demand for handling large dense matrices, the advancement in memory capacity and speed in modern computing systems has lagged behind computational performance. Therefore, approximation calculation methods

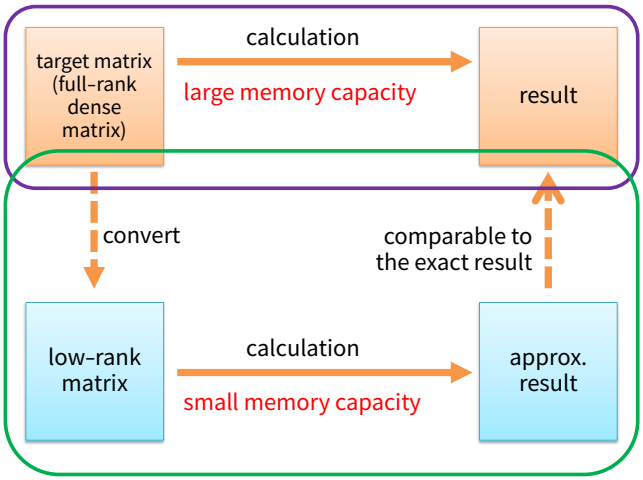


Fig. 2. Example of Low-rank approximation matrix calculation. Low-cost computations on approximated low-rank matrices yield reasonable solutions at low cost.

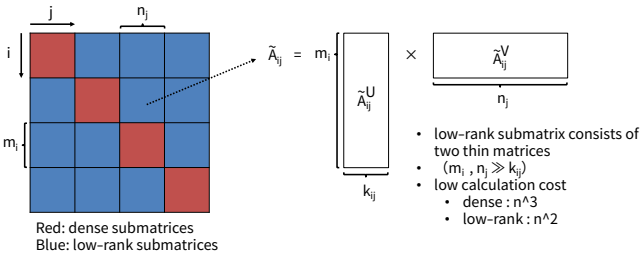


Fig. 3. Example of BLR matrix. It is a block matrix, and whether it is a dense matrix or a low-rank matrix is determined for each block.

(Figure 2) that approximate dense matrices with smaller memory footprint matrices and perform calculations on them are attracting attention. BLR matrices (Figure 3) is one such type of approximation matrix, characterized by a simple structure suitable for parallel computation. Prior research on BLR-QR includes Ida et al.'s proposal for MPI parallelization using CPUs, implemented using Fortran90 [3]. Ohshima et al. further proposed accelerating the main calculation parts using GPUs via CUDA Fortran with Managed Memory [4].

The BLR-QR algorithm used in this study employs the Modified Block Gram-Schmidt (MBGS) method (Algorithm 1) [5]. The basic parallelization strategy involves assigning calculations to MPI processes cyclically by column block. Further parallelization within column blocks is expected to yield high performance on CPU cores (threads) or GPUs. The total time required for QR factorization is largely dominated by matrix multiplication and addition (contained within the product-sum operations on the approximation matrices) and rank reduction calculations (Algorithm 2). In prior research [3], the matrix multiplication and addition was implemented using the DGEMM function of the BLAS library. Similarly, the rank reduction calculation was implemented using the dgeqrf and dgesvd functions of the LAPACK library. The implementations of these numerical functions utilized the Intel oneAPI MKL library.

Algorithm 1 MBGS algorithm, where N_b is the number of blocks in a column (or row) of matrices A, Q, and R. The subscripts of the matrices denote the row and column indices of the lattice. The symbol $*$ represents all blocks, e.g., $A_{*,j}$ is the j-th block column which is a tall-skinny matrix. The function TSQR($A_{*,j}$) returns the QR factorization of the tall-skinny matrix $A_{*,j}$ such that $Q_{*,j}^T Q_{*,j} = I$ and $Q_{*,j}, R_{j,j} = A_{*,j}$, where $Q_{*,j} \in \mathbb{R}^{N \times l}$ and $R_{j,j} \in \mathbb{R}^{l \times l}$.

Input: A with $N_b \times N_a$ blocks
Output: Q, R with $N_b \times N_a$ blocks such that $QR = A$

```
for j = 1, 2, ··· , N_b do
    [Q_{*,j}, R_{j,j}] := TSQR(A_{*,j})
    for k = j + 1, j + 2, ··· , Nb do
        R_{j,k} := Q^T_{*,j} A_{*,k}
        A_{*,k} := A_{*,k} − Q_{*,j} R_{j,k}
    end for
end for
```

Algorithm 2 Rank reduction algorithm for matrices with low-rank representation, where $(Q^U \mu)\sigma(\nu Q^V)$ is the Singular Value Decomposition (SVD) of UV and the corresponding truncated SVD with the $k^{'}$ largest singular values gives the best rank $k^{'}$ approximation of UV.

Input: $U \in \mathbb{R}^{m \times k}$, $V \in \mathbb{R}^{k \times n}$ composing $A := UV$
Output: $U^{'} \in \mathbb{R}^{m \times k^{'}}$, $V^{'} \in \mathbb{R}^{k^{'} \times n}$ such that $U'V'$ approximates UV

1: $[Q^U, R^U] := \mathrm{TSQR}(U)$, $[Q^V, R^V] := \mathrm{TSQR}(V)$
2: $[\mu, \sigma, \nu] := \mathrm{SVD}(R^U R^V)$
3: Truncate σ at $k^{'}$: $\sigma \in \mathbb{R}^{k \times k} \rightarrow \sigma^{'} \in \mathbb{R}^{k^{'} \times k^{'}}$
4: Set $U^{'}$ as the first $k^{'}$ columns of $Q^U \mu \sigma^{'}$
5: Set $V^{'}$ as the first $k^{'}$ rows of νQ^V

The structure of the BLR-QR program, which abstracts the computational content, is shown in Figure 4. Considering it as a program that repeatedly performs computations provided by the BLAS and the LAPACK, along with simple matrix and vector operations, the implementation method and performance trends established in this research are expected to be useful references for various numerical computation programs.

B. GPU Implementation of BLR-QR

GPU-oriented numerical libraries such as cuBLAS and cuSOLVER are widely utilized for GPU optimization of numerical programs. Since many of these libraries offer APIs similar to existing CPU libraries such as the BLAS and the LAPACK, high performance can be achieved with minimal effort. The prior BLR-QR implementation [4] achieved high performance by replacing MKL libraries with cuBLAS (e.g., cublasDgemm function) and cuSOLVER (e.g., cusolverDnDgeqrf function). However, the individual matrices and vectors involved in BLR-QR calculations are typically small, which limits the ability to fully exploit GPU performance. To address this, the prior study utilized the Multi-Instance GPU (MIG) feature to partition the GPU into sub-GPUs, ensuring that multiple

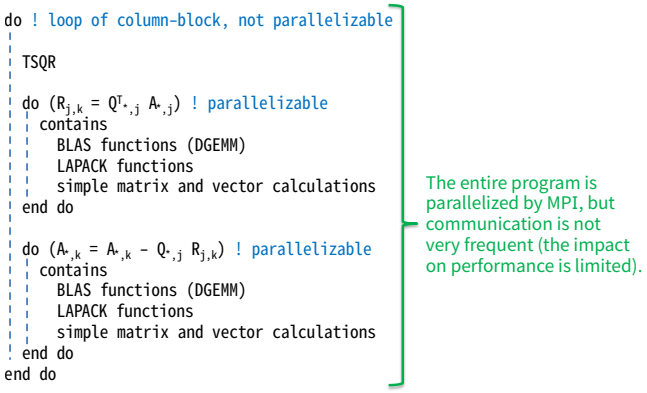


Fig. 4. Outline of the Target Program. Most of the execution time is consumed by library functions, and there are many parts that can be parallelized. This is not unusual for a matrix computation program.

library functions executed smoothly without interfering with one another, thereby improving performance.

The construction of the BLR matrix, the target of the QR factorization, is performed on the CPU prior to these accelerated steps. Since the data structure is tree-like and involves user-defined data types with dynamically allocated member variables, data transfer between CPU and GPU is not straightforward. To circumvent the complexities of implementing explicit data transfers, Managed Memory was employed in the prior work. Furthermore, CUDA Streams were utilized to enhance GPU utilization, which confirmed the realization of high-speed BLR-QR calculations on an A100 GPU. Batch processing [6] is a powerful method for accelerating numerous small-scale computations. However, due to the data structure and computational sequence involved in this calculation, applying batch processing is difficult.

IV. Implementation and Performance Comparison

This study compares Managed Memory and Unified Memory for BLR-QR, focusing on two aspects: the simplicity of program description and the resulting execution performance.

The experiments were conducted on two supercomputer systems:

1) Miyabi-G: Installed at JCAHPC. Equipped with the NVIDIA GH200 processor (Grace CPU and Hopper GPU are integrated), supporting both Managed Memory and Unified Memory.
2) Genkai-B: Installed at Kyushu University. Equipped with NVIDIA H100 GPUs and Intel Xeon 8490H CPUs, supporting only Managed Memory.

The node configurations are detailed in Table I (Miyabi-G) and Table II (Genkai-B).

A. Comparison of Program Description

This subsection summarizes the similarities and differences between Managed Memory and Unified Memory in the BLR-QR implementation.

TABLE I
Compute node of Miyabi-G (accelerated compute node of supercomputer Miyabi) installed at JCAHPC

CPU	
Processor	NVIDIA Grace
Architecture	Grace
Amount of FP64 cores	72
Clock speed	3.00 GHz
Peak performance (DP)	3,456 GFLOPS
Memory	LPDDR5X, 120 GiB, 512 GB/s
Amount of processor	1
GPU	
Processor	NVIDIA H100
Architecture	Hopper
Amount of FP64 cores	8,448 (64 cores × 132 SMs)
Clock speed	Up to 1,980 MHz
Peak performance (DP)	33.5 TFLOPS
Peak performance (DGEMM by TensorCore)	66.9 TFLOPS
Memory	HBM3, 96 GiB, 4,020 GB/s
Amount of processor	1
Interconnect network, GPU Driver, and software	
Network	InfiniBand NDR 200bps
GPU driver	550.163.01
Compiler	nvfortran 24.9
MPI	Open MPI 4.1.7a1

As discussed in Section II, Managed Memory and Unified Memory can be utilized using either compiler options or variable qualifiers.

1) Compiler Option Implementations: The compiler option approach is expected to allow GPU computation with minimal explicit programming related to device memory.

First, consider the case where the compiler option `mem:unified` (or `mem:unified:managedalloc` or `mem:unified:nomanagedalloc`) is specified. When using `mem:unified` options, memories allocated by using standard Fortran `allocate` function can be directly passed to the cuBLAS/cuSOLVER functions and utilized in GPU kernel calculations using CUF Kernel Directives (`cuf kernel do` directive). It is also deallocated by standard Fortran `deallocate` function. However, caution is required when using CUDA Streams. While CUDA Fortran allows linking an `allocate` process to a Stream via an option for Device Memory, this is not available for Unified Memory. Instead, the `cudaMemPrefetchAsync` function can be used to guarantee the execution order between memory allocation and subsequent computation. It was confirmed that a correctly executable program could be created without using any variable qualifiers. We refer to these implementation as **mem:unified**, **mem:unified(ma)**, and **mem:unified(nma)** respectively.

Next, let's consider an implementation using Managed Memory by specifying the `mem:managed` option. Similar to Unified Memory, memories allocated by using standard Fortran `allocate` function can be directly passed to the cuBLAS functions and deallocated by standard `deallocate`

TABLE II
COMPUTE NODE OF GENKAI-B (NODE GROUP B OF SUPERCOMPUTER GENKAI) INSTALLED AT KYUSHU UNIVERSITY

CPU	
Processor	Intel Xeon 8490H
Architecture	Sapphire Rapids
Amount of FP64 cores	60 (HTT is disabled in this system)
Clock speed	1.90 GHz (Up to 3.50 GHz)
Peak performance (DP)	3,648 GFLOPS
Amount of processor	2
Main Memory	
Specification	DDR5-4800
Capacity	256 GiB (per CPU socket)
Bandwidth	307.20 GB/s (per CPU socket)
GPU	
Processor	NVIDIA H100
Architecture	Hopper
Amount of FP64 cores	8,448 (64 cores × 132 SMs)
Clock speed	Up to 1,980 MHz
Peak performance (DP)	33.5 TFLOPS
Peak performance (DGEMM by TensorCore)	66.9 TFLOPS
Memory	HBM2e, 94 GiB, 2,396 GB/s
Amount of processor	4
Interconnect network, GPU Driver, and software	
Network	InfiniBand NDR 400bps × 2
GPU driver	580.82.07
Compiler	nvfortran 25.5
math library	Intel oneAPI MKL 2024.1
MPI	Open MPI 4.1.7rc1

function. However, when using the `mem:managed` option without variable qualifiers, an attribute mismatch error occurred when passing declared arrays to cuSOLVER functions (e.g., `cusolverDnDgeqrf`) and CUF Kernel Directives. A simple solution was to apply variable qualifiers to the problematic arrays. Since some cuSOLVER functions require sub-matrices composing the BLR matrix, the `managed` qualifier had to be added across a broad range of the program's data structure declarations. Similar to Unified Memory, `allocate` functions cannot be linked to Stream information, so `cudaMemPrefetchAsync` was used solely to ensure processing order. We refer to this implementation as **mem:managed**.

2) Variable Qualifier Implementations: Next, we consider an implementation using the method of writing variable qualifiers (without adding any `mem:` options as compiler options). This approach requires explicit program description for device memory, increasing implementation effort. However, it offers the potential for higher performance than relying solely on compiler options, due to optimized memory placement. In these implementations, temporary variables known to be used only on GPU were declared with the `device` attribute, while arrays such as the BLR matrix itself, which are accessed by both CPU and GPU, were declared with the `managed` or `unified` attribute. Unlike the compiler option approach, this method explicitly uses Device Memory, enabling the use of CUDA Streams without prefetch functions to guarantee processing order.

TABLE III
TARGET MATRICS

Matrix name	50ts	100ts
Number of lines (= degree of freedom)	50,864	101,250
Maximum rank	268	371
Average rank	36	43
Number of blocks	79 * 79	79 * 79
Memory amount of low-rank matrix (MB)	1,654	4,755
Memory ratio (%) †	8.38	6.08

† Memory amount ratio of BLR matrix to dense matrix.

We refer to programs in which memory types are specified manually as **unified-handed** and **managed-handed** in the following discussion.

The implementation variations prepared are summarized as follows:

- **mem:unified**, **mem:unified(ma)**, **mem:unified(nma)**: Using Unified Memory via compiler options. No variable qualifiers used. GH200 exclusive.
- **mem:managed**: Using Managed Memory via compiler options, but variable qualifiers were additionally required due to limitations of cuSOLVER and CUF Kernel Directives.
- **unified-handed**: Using Unified Memory specified manually. Device Memory was aggressively used for GPU-only variables. GH200 exclusive.
- **managed-handed**: Using Managed Memory specified manually. Device Memory was aggressively used for GPU-only variables.

B. Comparison of Execution Time

We executed BLR-QR using the implementations described above and compared the execution times. The main calculation part of the QR factorization of BLR matrices corresponds to lines 3 through 6 of Algorithm 1. The total execution time for these parts was used for comparison, specifically utilizing the longest time among all MPI processes. While first-touch optimization is possible when using Unified Memory, we have not considered applying it in this problem setting because QR decomposition is performed on matrices created by the CPU. Two matrix configurations (small: 50ts, large: 100ts) were used (Table III). Both matrices originated from an electromagnetic field analysis problem concerning simple spherical substances.

Prior research confirmed that MIG and CUDA Stream are effective for BLR-QR calculation. Miyabi-G allows execution in a 4MIG configuration (1 GPU divided into 4 sub-GPUs), while Genkai-B allows a 7MIG configuration (1 GPU divided into 7 sub-GPUs). For Miyabi-G, measurements included execution using 4MIG within one node (1node, 4MIG) and execution using 4 nodes without MIG (4node, 4GPU). For Genkai-B, measurements included execution using 7MIG within one

node (7MIG) and execution using 4 GPUs within the node without MIG (4GPU). Furthermore, for each configuration, we measured execution times both with CUDA Stream used for 4-way parallelism within one process (4Stream) and without it (1Stream). Although the number of processes differs between the execution environments, the calculation assigned to the processes column-cyclically means that the target problem itself is common across both environments. The compilers used are shown in Tables I and II. The optimization option `-fast` was applied during compilation.

The measurement results are presented in Figure 5 (50ts) and Figure 6 (100ts).

First, focusing on the performance difference between *compiler option implementations* and *variable qualifier implementations*, the overall trend indicates that *variable qualifier implementations* executes in a shorter time. For Unified Memory on Miyabi-G, **unified-handed** reduced execution time by a maximum of 42.7% compared to **mem:unified** in the 100ts, 1node, 4MIG, 4Stream configuration. For Managed Memory on Genkai-B, **managed-handed** reduced execution time by a maximum of 48.4% compared to 7MIG, 4Stream. However, for Managed Memory on Miyabi-G, the execution time of **mem:managed** was extremely long, and **managed-handed** resulted in a maximum execution time reduction of 68.0%. Although performance improved slightly with a newer CUDA 12.9, it still took about twice as long as **managed-handed**. Since the matrix construction process (outside the measured range) where the CPU processes Managed Memory showed no significant difference compared to **mem:unified** or **unified-handed**, it is surmised that the GH200's control of Managed Memory may have inefficiencies that were exposed in this calculation. To identify the cause, we used the Nsight profiler to examine the breakdown of execution time. While we observed a significant increase in data transfer time from the CPU to the GPU, we were unable to pinpoint the specific process responsible. The majority of this program's execution time is consumed by library functions, within which numerous functions are executed, making performance analysis difficult. Analyzing a simpler program may be more helpful in identifying the cause.

Comparing the execution times for *variable qualifier implementations* between Miyabi-G and Genkai-B, the difference in execution time was small. Notably, when using MIG, Genkai-B was faster. This result may seem odd since Miyabi-G appears more powerful based solely on simple GPU theoretical performance. However, the MIG configurations differ across GPUs. Notably, Miyabi-G's 4MIG configuration is heterogeneous, with one sub-GPU performing lower than the others. Additionally, the program itself is not simple enough to fully leverage the computational and memory performance of the hardware. Therefore, this result is not particularly unusual.

Finally, considering the relationship between programming cost and computational performance. Programming using compiler option memory control is easy, and utilizing Unified Memory is particularly effective for cost reduction. However, given that manually optimizing memory placement can sometimes reduce execution time by over 40%, the significance of manual optimization remains substantial. Clarifying which specific processes offer the greatest potential for performance improvement through manual optimization is a future task, as this would make it easier to balance programming cost and computational performance. Regarding Managed Memory support via compiler options, the implementation faced difficulties due to compatibility issues with the cuSOLVER and CUF Kernel Directives. Moreover, we observed performance degradation on the GH200. Furthermore, since Managed Memory performed worse than Unified Memory on the GH200 even after manual optimization using variable qualifiers, our conclusion is that the Unified Memory should be prioritized for GH200.

V. RELATED WORK

Related studies concerning Managed Memory and Unified Memory include the work by Schieffer et al. [8], they performed performance evaluation of the Grace Hopper processor, primarily focusing on memory aspects and including a comparison of Managed Memory and Unified Memory. Gayatri et al. [9] evaluated the performance of Managed Memory and unified memory systems on NVIDIA Volta GPUs. These studies are more detailed than our study regarding the sheer performance evaluation of Managed Memory and Unified Memory. However, our study is unique in its focus on the productivity and performance of a CUDA Fortran-based numerical calculation application (not simple benchmarks).

The BLR-QR problem addressed in our study involves MPI communication. However, given the ratio of execution time, we concentrated solely on the primary calculation parts, and communication time was excluded from the discussion. It is possible that communication time needs to be considered depending on the problem setting. For MPI communication performance involving unified memory systems, the research by Karthik et al. [10] is relevant.

Research on approximation matrix calculation methods has been conducted from various perspectives [11]–[13]. Authors themselves also published many related works [3], [4], [14]. No prior research specifically focusing on BLR-QR, CUDA Fortran, and Managed Memory/Unified Memory in combination has been identified, establishing the originality of our study.

VI. CONCLUSION

This study evaluated Managed Memory and Unified Memory in the context of optimizing the Block Low-Rank QR factorization (BLR-QR) numerical computation application, implemented using CUDA Fortran. We compared the simplicity of program description and the resulting execution performance between compiler-driven memory control and manual optimization utilizing variable qualifiers.

The evaluation confirmed that utilizing compiler options for memory control is straightforward, and specifically, the utilization of Unified Memory is effective for cost reduction. However, given that manually optimizing memory placement

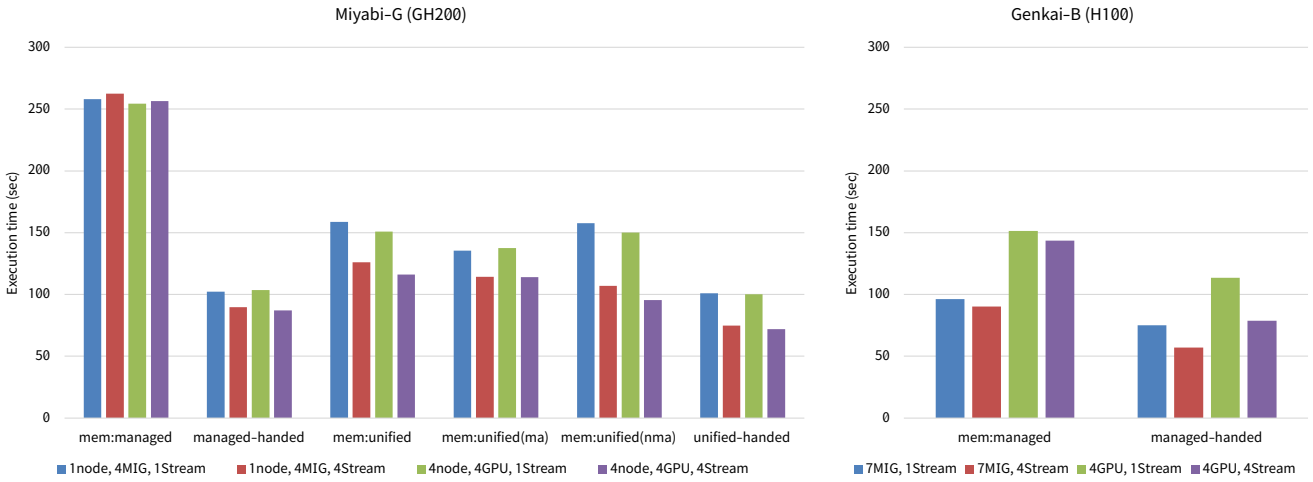


Fig. 5. Execution time comparison for BLR-QR (50ts configuration) across memory strategies and MIG/Stream settings.

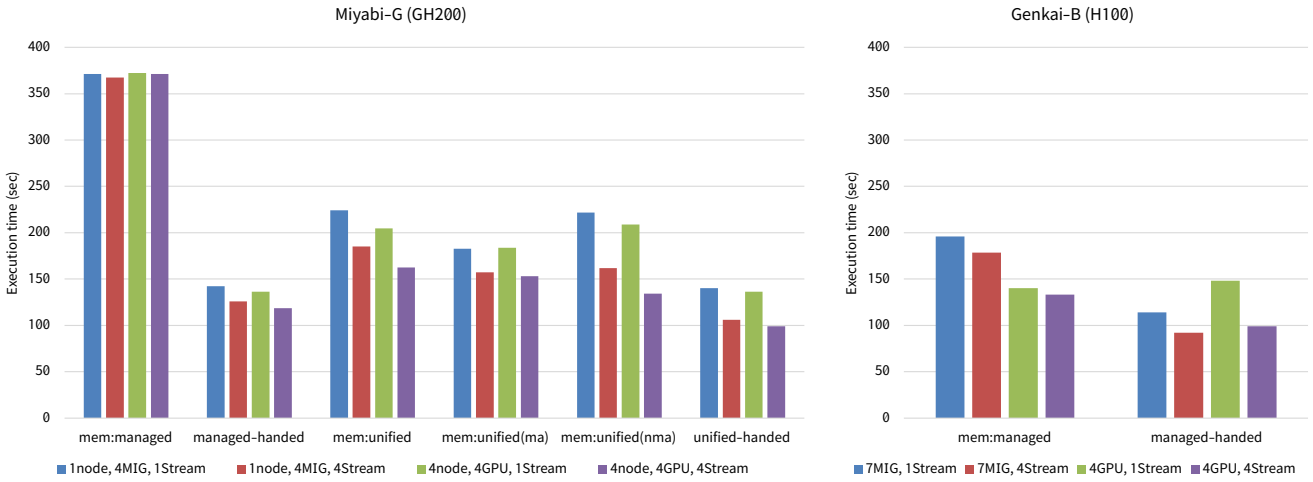


Fig. 6. Execution time comparison for BLR-QR (100ts configuration) across memory strategies and MIG/Stream settings.

can result in a reduction of execution time by over 40%, the significance of manual optimization remains substantial.

Regarding Managed Memory, implementation challenges arose because compiler option support could not fully address compatibility issues with the cuSOLVER interface and CUF Kernel Directives. Furthermore, Managed Memory exhibited specific instances of extreme performance degradation when controlled by compiler options on the GH200 architecture. Since Managed Memory performed worse than Unified Memory on the GH200 even after manual optimization, our conclusion is that the Unified Memory should be prioritized for GH200 architecture.

Future work should focus on the following tasks to better balance programming cost and computational performance: 1) Clarifying the specific processes where manual optimization yields the largest potential for performance improvement and identifying the variable qualifiers that significantly impacted performance. 2) Evaluating the performance and utility of Managed Memory and Unified Memory for problems other than BLR-QR.

Addressing these challenges will allow developers to select optimal memory management strategies while maintaining high productivity.

ACKNOWLEDGMENTS

This work was partially supported by JSPS KAKENHI Grant Numbers 24K02949. This work is supported by “Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures (JHPCN)” in Japan (Project ID: jh250032, jh250083). This research was conducted using the Supermicro ARS-111GL-DNHR-LCC and FUJITSU Server PRIMERGY CX2550 M7 (Miyabi) at Joint Center for Advanced High Performance Computing (JCAHPC) The computation was carried out using the computer resource offered under the category of JHPCN Joint Research Projects by Research Institute for Information Technology, Kyushu University.

REFERENCES

[1] TOP500 https://top500.org/ Accessed 2025-10-15

[2] RIKEN launches international initiative with Fujitsu and NVIDIA for "FugakuNEXT" development https://www.riken.jp/en/news_pubs/news/2025/20250822_1/index.html Accessed 2025-10-15
[3] Ida, A., Nakashima, H., Hiraishi, T., Yamazaki, I., Yokota, R. and Iwashita, T., "QR Factorization of Block Low-rank Matrices with Weak Admissibility Condition", Journal of Information Processing, Vol. 27, pp. 831–839 (2019). https://doi.org/10.2197/ipsjjip.27.831
[4] Satoshi Ohshima, Akihiro Ida, Rio Yokota, Ichitaro Yamazaki, "QR Factorization of Block Low-Rank Matrices on Multi-Instance GPU", Parallel and Distributed Computing, Applications and Technologies, Springer Nature Switzerland, pp.359-369 (2023) https://doi.org/10.1007/978-3-031-29927-8_28
[5] Jalby, W. and Philippe, B., "Stability Analysis and Improvement of the Block Gram–Schmidt Algorithm", SIAM Journal on Scientific and Statistical Computing, Vol. 12, No. 5, pp. 1058–1073 (1991).
[6] Batched BLAS https://icl.utk.edu/bblas/ (Accessed 2025-02-07)
[7] NVIDIA Multi-Instance GPU User Guide r560 https://docs.nvidia.com/datacenter/tesla/mig-user-guide/ (Accessed 2025-10-15)
[8] Schieffer, Gabin and Wahlgren, Jacob and Ren, Jie and Faj, Jennifer and Peng, Ivy, "Harnessing Integrated CPU-GPU System Memory for HPC: a first look into Grace Hopper", Proceedings of the 53rd International Conference on Parallel Processing, pp.199-209 (2024) https://doi.org/10.1145/3673038.3673110
[9] Gayatri, Rahulkumar and Gott, Kevin and Deslippe, Jack, "Comparing Managed Memory and ATS with and without Prefetching on NVIDIA Volta GPUs", 2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), pp.41-46 (2019) https://doi.org/10.1109/PMBS49563.2019.00010
[10] Vadambacheri Manian, Karthik and Chu, Ching-Hsiang and Ahmad Awan, Ammar and Shafie Khorassani, Kawthar and Subramoni, Hari and Panda, D. K, "OMB-UM: Design, Implementation, and Evaluation of CUDA Unified Memory Aware MPI Benchmarks", 2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), pp.82-92 (2019) https://doi.org/10.1109/PMBS49563.2019.00015
[11] Hackbusch, W., "A Sparse Matrix Arithmetic Based on H-Matrices", Part I:Introduction to H-Matrices, Computing, Vol. 62, pp. 89–108 (1999). https://doi.org/10.1007/s006070050015
[12] Benner, P. and Mach, T., "On the QR Decomposition of H-matrices", Computing, Vol. 88, No. 3-4, pp. 111-129 (2010) https://doi.org/10.1007/s00607-010-0087-y
[13] Baboulin, Marc and Donfack, Simplice and Kaya, Oguz and Mary, Theo and Robeyns, Matthieu, "Mixed Precision Randomized Low-Rank Approximation with GPU Tensor Cores", Euro-Par 2024: Parallel Processing: 30th European Conference on Parallel and Distributed Processing, pp.31-44 (2024) https://doi.org/10.1007/978-3-031-69583-4_3
[14] Apriansyah, M. Ridwan and Yokota, Rio, "Parallel QR Factorization of Block Low-rank Matrices", ACM Transactions on Mathematical Software, Vol.48, No.3, pp.1-28 (2022) https://doi.org/10.1145/3538647