Climbing the Summit and Pushing the Frontier of Mixed Precision Benchmarks at Extreme Scale

Hao Lu
Oak Ridge National Laboratory
Oak Ridge, USA
luh1@ornl.gov

Michael Matheson
Oak Ridge National Laboratory
Oak Ridge, USA
mathesonma@ornl.gov

Vladyslav Oles
Oak Ridge National Laboratory
Oak Ridge, USA
olesv@ornl.gov

Austin Ellis
Oak Ridge National Laboratory
Oak Ridge, USA
ellisja@ornl.gov

Wayne Joubert
Oak Ridge National Laboratory
Oak Ridge, USA
joubert@ornl.gov

Feiyi Wang
Oak Ridge National Laboratory
Oak Ridge, USA
fwang2@ornl.gov

***Abstract*—The rise of machine learning (ML) applications and their use of mixed precision to perform interesting science are driving forces behind AI for science on HPC. The convergence of ML and HPC with mixed precision offers the possibility of transformational changes in computational science. The HPL-AI benchmark is designed to measure the performance of mixed precision arithmetic as opposed to the HPL benchmark which measures double precision performance. Pushing the limits of systems at extreme scale is nontrivial —little public literature explores optimization of mixed precision computations at this scale. In this work, we demonstrate how to scale up the HPL-AI benchmark on the pre-exascale Summit and exascale Frontier systems at the Oak Ridge Leadership Computing Facility (OLCF) with a cross-platform design. We present the implementation, performance results, and a guideline of optimization strategies employed for delivering portable performance on both AMD and NVIDIA GPUs at extreme scale.**

***Index Terms*—Parallel programming, High performance computing, Exascale computing, Linear algebra.**

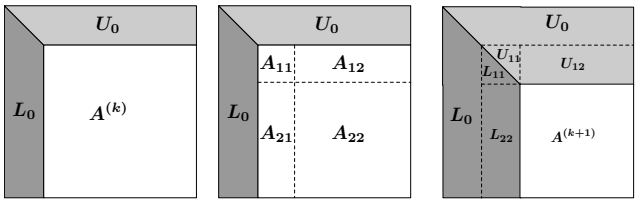
I. Introduction

Mixed precision computing is critically important in the modern data center as machine learning (ML) has shown an explosive growth due to the innovative use of the significant c omputational p ower a vailable a t l ower p recision on graphics processing units (GPUs). These applications have demonstrated that acceptable results can be obtained at reduced precision. The motivation to use mixed precision is due to hardware having at least five t imes t he fl oating point performance in lower precision blue [1]. However, traditional HPC workloads are dominated by double precision, 64-bit floating p oint (FP64) c omputations t o d eliver t he required accuracy. The convergence of HPC and ML offers the opportunity to develop new techniques that exploit mixed precision capabilities to enable new science. The High Performance LINPACK (HPL) benchmark [2] has long played a key role in tracking the performance of the world's top supercomputers. The double precision math reflects the precision used in typical HPC applications. Most of the ML models seen in centers train using mixed precision operations, which are fundamentally different from those in HPL.

The High-Performance LINPACK benchmark for Accelerator Inspection (HPL-AI) [3] was created to provide a benchmark that measures the performance potential of mixed precision (FP16/FP32) on newly deployed supercomputers. Together these benchmarks are important in the acquisition of large leadership class supercomputers given the cost and resources necessary to design, deploy, and maintain these systems. The combined benchmarks yield valuable insight into the double precision and mixed precision performance possible on systems covering a wide range of use cases.

Only a handful of systems in the world are capable of exascale performance, and just a limited number of applications are able to sustain it. The present work helps pave the way for more into the future. The main contributions of this paper are:

- We summarize the efforts and accomplishments of developing a leadership class cross-platform implementation of the HPL-AI benchmark for two of the world's fastest supercomputers at the OLCF, both the NVIDIA-based Summit and the newly launched AMD-based exascale system, Frontier. This is the **first** known code that runs on different GPU enabled systems and delivers exascale performance on both. We report sustained performance of 1.411 EFLOPS on Summit and 2.387 EFLOPS on approximately 40% of Frontier. Our Summit result achieved 9.5 times the performance of HPL demonstrating the value of mixed precision.
- We propose a performance model based on measured floating point operation (flop) rate for key compute ker-



(a) (b) (c)

Fig. 1: The kth step of block LU factorization. (a) The layout right before the kth step. (b) Partitioning of the trailing matrix. (c) The outcome of the kth step.

nels, and we explore and report on the general algorithmic strategies and optimization processes to exploit system hardware in a portable way. We discuss the holistic optimization of communication, computation, memory placement, and algorithm design for complex codes. We benchmark each component and optimization technique in HPL-AI both separately and as a whole so that results can be utilized to help deploy applications at scale.

- We share lessons learned and practical advice that are applicable to other development efforts on leadership class systems. Observations are derived from experience with the newly deployed supercomputer Frontier. The subsequent insight can enable one to achieve sustainable performance at scale in the newly minted exascale era.

The remainder of the paper is organized as follows. We begin by providing the background of theoretical and numerical aspects of the HPL and HPL-AI benchmarks, with a review of the past literature. We continue with the design and implementation of our distributed, GPU-centric HPL-AI algorithm. We outline our approach for achieving and sustaining state-of-the-art performance. Focus is given to practical issues such as algorithms, communication strategies, data layouts, parallelization and cross-platform development. We discuss and contrast critical performance tuning and optimization strategies. Finally, we share the insights gained and best practices discovered through this experience to help the broader computational community.

II. Background and Related Work

The HPL-AI benchmark [3] is concerned with finding the unique solution to a dense system of linear equations $\boldsymbol{Ax} = \boldsymbol{b}$, where $\boldsymbol{A} \in \mathbb{R}^{N\times N}$ is a full rank matrix and $\boldsymbol{x}, \boldsymbol{b} \in \mathbb{R}^N$ are the solution and right-hand side vectors, respectively. The solution employs Gaussian elimination to transform a generic system of linear equations into a triangular form [4]. It applies elementary row transformations to zero out entries below each diagonal element of $\boldsymbol{A}$ until the resulting matrix $\boldsymbol{U}$ is upper triangular. For every diagonal element $\boldsymbol{A}_{k,k}$, such a set of transformations can be represented as a left multiplication by a unit lower triangular matrix $\boldsymbol{L}_k$, whose only nonzero off-diagonal entries are in the kth column. Once the LU factorization of $\boldsymbol{A} = \boldsymbol{LU}$ is obtained, the solution to $\boldsymbol{Ax} = \boldsymbol{b}$ can be efficiently obtained by solving two triangular systems of linear equations.

In contrast to the HPL benchmark, HPL-AI allows for the input matrix to have an appropriate condition number to omit the pivoting step during the LU factorization [5]. More importantly, it allows the use of a mixed precision solution process to obtain lower precision $\tilde{\boldsymbol{L}}$ and $\tilde{\boldsymbol{U}}$ factors. The solution $\tilde{\boldsymbol{x}}$ in HPL-AI is corrected to higher precision by using an FP64 iterative refinement method.

Block LU factorization: Gaussian elimination at scale utilizes a distributed matrix $\boldsymbol{A}$ which is partitioned into $B \times B$ blocks that are distributed across the processes. The block size B is chosen to balance communication and computation. Consequently, transforming $\boldsymbol{A}$ into its LU factorization occurs in blocks, that is, each step of the Gaussian elimination computes B columns of $\boldsymbol{L}$ and B rows of $\boldsymbol{U}$ [6], [7].

At the kth step of the process, the first $(k-1)B$ rows and columns of $\boldsymbol{A}$ are finalized and store the corresponding nontrivial entries of $\boldsymbol{L}$ and $\boldsymbol{U}$, see part (a) of Figure 1. To factor the remaining $(N - kB + B) \times (N - kB + B)$ submatrix $\boldsymbol{A}^{(k)}$ as $\boldsymbol{L}^{(k)}\boldsymbol{U}^{(k)}$, we represent as

$$\boldsymbol{A}^{(k)} = \begin{bmatrix} \boldsymbol{A}_{1,1} & \boldsymbol{A}_{1,2} \\ \boldsymbol{A}_{2,1} & \boldsymbol{A}_{2,2} \end{bmatrix},$$

where $\boldsymbol{A}_{1,1}$, $\boldsymbol{A}_{1,2}$, $\boldsymbol{A}_{2,1}$, and $\boldsymbol{A}_{2,2}$ are of sizes $B\times B$, $(N-kB)\times B$, $B \times (N - kB)$, and $(N - kB) \times (N - kB)$, respectively (part (b) of Figure 1). Under the analogous partition of $\boldsymbol{L}^{(k)}$ and $\boldsymbol{U}^{(k)}$, the LU factorization can be viewed as the matrix equations

$$\begin{aligned} \begin{bmatrix} \boldsymbol{A}_{1,1} & \boldsymbol{A}_{1,2} \\ \boldsymbol{A}_{2,1} & \boldsymbol{A}_{2,2} \end{bmatrix} &= \begin{bmatrix} \boldsymbol{L}_{1,1} & \boldsymbol{0} \\ \boldsymbol{L}_{2,1} & \boldsymbol{L}_{2,2} \end{bmatrix} \begin{bmatrix} \boldsymbol{U}_{1,1} & \boldsymbol{U}_{1,2} \\ \boldsymbol{0} & \boldsymbol{U}_{2,2} \end{bmatrix} \\ &= \begin{bmatrix} \boldsymbol{L}_{1,1}\boldsymbol{U}_{1,1} & \boldsymbol{L}_{1,1}\boldsymbol{U}_{1,2} \\ \boldsymbol{L}_{2,1}\boldsymbol{U}_{1,1} & \boldsymbol{L}_{2,1}\boldsymbol{U}_{1,2} + \boldsymbol{L}_{2,2}\boldsymbol{U}_{2,2} \end{bmatrix}, \end{aligned}$$

where $\boldsymbol{L}_{1,1}$ and $\boldsymbol{L}_{2,2}$ are unit lower-triangular and $\boldsymbol{U}_{1,1}$ and $\boldsymbol{U}_{2,2}$ are upper-triangular. The kth step is then carried out by the following steps (see part (c) of Figure 1):

Step 1) Gaussian elimination for $\boldsymbol{A}_{1,1}$ to find $\boldsymbol{L}_{1,1}$ and $\boldsymbol{U}_{1,1}$;
Step 2) Compute $\boldsymbol{L}_{2,1} = \boldsymbol{A}_{2,1}\boldsymbol{U}_{1,1}^{-1}$;
Step 3) Compute $\boldsymbol{U}_{1,2} = \boldsymbol{L}_{1,1}^{-1}\boldsymbol{A}_{1,2}$;
Step 4) Compute $\boldsymbol{A}^{(k+1)} \stackrel{\text{def}}{=} \boldsymbol{L}_{2,2}\boldsymbol{U}_{2,2} = \boldsymbol{A}_{2,2} - \boldsymbol{L}_{2,1}\boldsymbol{U}_{1,2}$.

The above steps illustrate the high-level algorithmic steps for block-based LU factorization. For state-of-the-art results the algorithmic design must be broken down much further—data distribution, dependencies and potential data movement, the critical execution path, communication strategy and the delivered floating point performance. We detail our design and implementation using pseudocode in Section III.

Iterative refinement: Even when $\boldsymbol{A}$ is well-conditioned, computing its LU factorization suffers from precision limitations of floating point arithmetic, especially when working in mixed precision. As a result, the equality $\boldsymbol{A} = \boldsymbol{LU}$ holds only approximately for the computed $\boldsymbol{L}$ and $\boldsymbol{U}$. To indicate the discrepancy between the practical and theoretical results of Gaussian elimination, we use the notation $\tilde{\boldsymbol{L}}$ and $\tilde{\boldsymbol{U}}$ for the numerically-obtained matrices, implying that $\tilde{\boldsymbol{A}} \stackrel{\text{def}}{=} \tilde{\boldsymbol{L}}\tilde{\boldsymbol{U}} \approx \boldsymbol{A}$. In a similar vein, we use $\tilde{\boldsymbol{x}}$ to denote the approximate solution

to $\boldsymbol{Ax} = \boldsymbol{b}$ obtained by solving $\tilde{\boldsymbol{A}}\tilde{\boldsymbol{x}} = \tilde{\boldsymbol{L}}(\tilde{\boldsymbol{U}}\tilde{\boldsymbol{x}}) = \boldsymbol{b}$ as two triangular systems.

The accuracy of $\tilde{\boldsymbol{x}}$ can be improved by a procedure called iterative refinement (IR) [8], based on reusing the LU factorization of $\tilde{\boldsymbol{A}}$. Performing IR amounts to repeating the following steps until the required solution accuracy is reached:

Step 1) Compute the residual $\boldsymbol{r} \stackrel{\text{def}}{=} \boldsymbol{b} - \boldsymbol{A}\tilde{\boldsymbol{x}}$ in higher precision;
Step 2) Find an approximation $\tilde{\boldsymbol{d}}$ of solution discrepancy $\boldsymbol{d} \stackrel{\text{def}}{=} \boldsymbol{x} - \tilde{\boldsymbol{x}}$ by solving $\tilde{\boldsymbol{A}}\tilde{\boldsymbol{d}} = \boldsymbol{r}$ (note that $\boldsymbol{Ad} = \boldsymbol{r}$);
Step 3) Refine the approximate solution $\tilde{\boldsymbol{x}}$ by assigning $\tilde{\boldsymbol{x}} \leftarrow \tilde{\boldsymbol{x}} + \tilde{\boldsymbol{d}}$.

Every iteration of the above steps adjusts $\tilde{\boldsymbol{x}}$ closer to the actual solution $\boldsymbol{x}$. Typically IR is stopped once the approximated solution discrepancy $\tilde{\boldsymbol{d}}$ gets below some predefined threshold. Computationally, it is a relatively inexpensive process to recover the lost accuracy and thus worth the tradeoff.

Related Work: The 2006 seminal work of Kurzak and Dongarra [9] solved $\boldsymbol{Ax} = \boldsymbol{b}$ in mixed precision followed by iterative refinement. In 2010, Wang et al. offered a GPU-accelerated version of the algorithm for the first time and demonstrated its performance on several heterogeneous architectures [10]. The dense linear algebra library ScaLAPACK [11] supported distributed computing, and the MAGMA library enabled GPU support [12], [13]. In 2017, Haidar et al. added mixed precision variants to MAGMA [14], [15]. In 2019, a library called SLATE [16] supported the capability in multiple precisions.

The Fugaku HPL-AI code [17], [18] broke the exascale barrier in 2020 with a CPU only implementation supporting ARM processors and optimized for the proprietary Tofu interconnect. As in our paper, their HPL-AI benchmark used a linear congruential generator for the entries of $\boldsymbol{A}$, allowing regenerating them from any place in the code at low cost. Our design and implementation has been heavily influenced by these prior works, including using the Fugaku code as a baseline, but have taken it further on model-based performance tuning, optimization strategies, GPU support and cross-platform portability.

III. GPU-centric Parallel Design and Implementation

This section gives a comparative overview of Summit and Frontier architectures in the context of a cross-platform GPU implementation. We provide detailed algorithmic steps for our work. We also present design rationales on data distribution and data movement, along with considerations of matrix initialization and iterative refinement.

A. Summit and Frontier architectures

The OLCF currently hosts two different leadership class supercomputers: Summit and Frontier. We provide a brief overview of the key architectural specifications (cf. Table I).

Certain features directly impact the choice of parameters, optimization, and ultimately the performance of the HPL-AI code at scale. The GPU memory dictates the maximum size of the problem that can be solved. The graphics complex die (GCD) count per node impacts the number of MPI ranks per node. Contrasting the two systems, Frontier's AMD GPUs can solve larger problems with 4× memory per GCD over Summit.

	Summit	**Frontier**
Number of Nodes	4608	9408
Processor	Power9	3rd Gen EPYC
CPU memory (Node)	512 GB	512 GB
GPU / # of GCDs (Node)	NVIDIA V100 / 6	AMD MI250X / 8
per GPU / per Node memory	16 / 96 GB	128 / 512 GB
GPU Interconnect	NVLINK	Infinity Fabric
GPU Interconnect B/W	50+50 GB/s	50+50 GB/s
FP16/FP64 TFLOPS (GCD)	125 / 7.8	298 / 54.5
FP16 TFLOPS (Node)	750	1192
# of NICs	2x Mellanox EDR IB	4x Slingshot-11
NIC B/W (node)	12.5+12.5 GB/s	25+25 GB/s

TABLE I: Key architectural specifications for Summit and Frontier

Most importantly, GCD performance is highly correlated with the performance of HPL-AI. Frontier has 1.58× per-node performance in half precision and 2×+ the number of nodes over Summit. Frontier is expected to see about 3× HPL-AI performance improvement when compared to Summit at full scale. Further, Frontier will be 8× more powerful than Summit in double precision.

Considering communication costs, the time required for the exchange of the matrix is directly tied to the number of network interface cards (NICs) and their performance. The Frontier NIC is directly connected to the GPU, which in general enhances the performance of using GPU-aware MPI, as the data can be transferred directly between GPUs.

B. Cross-platform Design Considerations

The OLCF hosts two leadership supercomputers with two different GPU architectures, creating strong motivation for a cross-platform capability. NVIDIA CUDA libraries [19] and the AMD ROCm libraries [20] provide the building blocks for accelerator-based application development. To enhance compatibility and provide a migration path for existing CUDA-based applications, AMD provides a suite of tools based on the Heterogeneous-Compute Interface for Portability (HIP) [21], which greatly aids in developing portable software.

We built a thin shim layer using a macro approach to support both GPU architectures instead of using HIP insofar as it did not allow for a seamless transition for all required library calls. In the future, we will likely use a wholly HIP implementation. Our code leverages many optimized and tuned GPU libraries such as cuBLAS/rocBLAS and cuSOLVER/rocSOLVER, but there are instances where differences in CUDA/ROCM library API signatures require a non-HIP implementation. A concrete example is the GETRF call, for which cuSOLVER requires a separate step to compute the additional workspace memory needed, `cusolverDnSgetrf_bufferSize`, unlike rocSOLVER which supports a single call.

The major basic linear algebra subprograms (BLAS) library functions used in our implementation are shown in Table II. GEMM is a general matrix-matrix product, TRSM/TRSV is a triangular solve with matrix/vector right-hand sides, and

GETRF is a general triangular factorization. The naming is defined in the BLAS Technical Forum standard [22].

BLAS Mapping	Summit	Frontier
GEMM	`cublasSgemmEx`	`rocblas_gemm_ex`
TRSM	`cublasStrsm`	`rocblas_strsm`
GETRF	`cusolverDnSgetrf`	`rocsolver_sgetrf`
TRSV	`openBLAS`	`openBLAS`

TABLE II: Cross-platform BLAS library functions

C. GPU-centric Distributed Implementation

Pseudocode for the distributed GPU-enabled HPL-AI algorithm is shown in Algorithm 1. Our implementation takes four input parameters B, N, P_r and P_c, which are the block size, total matrix size, processor count down rows, and processor count across columns, respectively. We begin by generating the global matrix $\boldsymbol{A}$, right-hand side vector $\boldsymbol{b}$ and solution vector in double precision (FP64) on the CPU. The size of $\boldsymbol{A}$ is determined by N and adjusted to a multiple of P_r, P_c and B. For generating the entries of $\boldsymbol{A}$, we use the 64-bit linear congruential generator (LCG), which yields random numbers in a sequence based on a chosen random seed. Importantly, LCG can jump start the sequence at low computational cost and is capable of generating a number n steps ahead of the current one in $O(\log n)$ time. This means LCG can quickly generate any $A_{i,j}$ as a function of i, j, making it easily parallelizable and also allowing each process to access any part of $\boldsymbol{A}$ by regenerating it on the fly. The ability to regenerate the entries of $\boldsymbol{A}$ alleviates the need to permanently store the matrix.

The responsibility for generating and hosting the global matrix A is partitioned across the total processor count P using a 2D block cyclic decomposition with process grid dimensions $P = P_r \times P_c$ and blocksize B. Each MPI rank is mapped with an index P_{ir}, P_{ic} representing its location in the process grid and allocates a contiguous local matrix with size $N_{Lr} = \frac{N}{P_r}$ by $N_{Lc} = \frac{N}{P_c}$ that represents a part of the global matrix. Since the local matrix is stored contiguously, the leading dimension LDA of the local matrix $\boldsymbol{A'}$ is fixed for the whole run.

The full local matrix is then converted to single precision (FP32) and copied to the GPU as input for LU factorization. We did not use data pipelining between CPU and GPU during the factorization. Making the whole matrix resident on the GPU removes the performance lost by coping data back and forth between CPU and GPU. The relative speeds of transfers versus mixed precision operations would require a block size B that is too large to amortize transfer costs.

Our GPU-based right looking recursive LU factorization is labeled to match the **Block LU factorization** algorithm described in Section II with the iterator $k = 1...\frac{N}{B}$ and is broken into three subproblems. The **Diagonal Update** subproblem solves $\boldsymbol{L}_{1,1}$ and $\boldsymbol{U}_{1,1}$ for $\boldsymbol{A}_{1,1}$ in FP32; **Panel Update** computes $\boldsymbol{L}_{2,1} = \boldsymbol{A}_{2,1}\boldsymbol{U}_{1,1}^{-1}$ and $\boldsymbol{U}_{1,2} = \boldsymbol{L}_{1,1}^{-1}\boldsymbol{A}_{1,2}$ in FP32; and **Update Trailing Matrix** computes $\boldsymbol{A}^{(k+1)} = \boldsymbol{A}_{2,2} - \boldsymbol{L}_{2,1}\boldsymbol{U}_{1,2}$ in mixed precision where ($\boldsymbol{L}$ and $\boldsymbol{U}$ are FP16 and $\boldsymbol{A}$ is FP32).

In the following, we explain HPL-AI iteration details with line numbers referring to the Algorithm 1. On Line 7-10, iteration k begins with the owner process factoring the diagonal block $\boldsymbol{A}_{k,k} = \boldsymbol{L}_{k,k}\boldsymbol{U}_{k,k}$ using BLAS operation ***GETRF*** on the GPU. The factors $\boldsymbol{L}$ and $\boldsymbol{U}$ are then broadcast to the processes that own the kth block rows and kth block columns . Subsequently, on Line 11-13/20-23, the processes that own the kth block row or column solve for the remaining $\boldsymbol{L}_{k+1:,k}$ and $\boldsymbol{U}_{k,k+1:}$ panels using the received factors. The BLAS operation ***TRSM_[R|L]_[UP|LOW]***, performed on the GPU, solves the triangular matrix equations, where ***[R|L]*** represents which side the triangular matrix is on and ***[UP|LOW]*** represents whether the matrix is upper or lower triangular.

Once the panels are solved, on Line 15-16/24-25, $\boldsymbol{L}$ is converted to half precision (FP16) in the ***CAST*** phase, and $\boldsymbol{U}$ is conveniently transposed and cast simultaneously using ***TRANS_CAST***. After the cast, on Line 16/25, the panel-owner processes broadcast the $\boldsymbol{L}_{:,k}$ and $\boldsymbol{U}_{k,:}$ to the processes that reside on the same process rows and process columns.

Once the panels are transferred, on Line 29, the rest of the matrix is updated with a mixed precision ***GEMM*** BLAS operation $\boldsymbol{A}_{k+1:,k+1:} = \boldsymbol{A}_{k+1:,k+1:} - \boldsymbol{L}_{k+1:,k}\boldsymbol{U}_{k,k+1:}$ to update FP32 $\boldsymbol{A}_{k+1:,k+1:}$ with FP16 $\boldsymbol{L}_{k+1:,k}$ and $\boldsymbol{U}_{k,k+1:}$. The GEMM phase is at the heart of the HPL-AI benchmark and enables accelerated performance through the mixed precision calculation.

Once the **Block LU factorization** is fully completed, we transfer the factored matrix $\boldsymbol{A}$ back to the CPU for **Iterative Refinement** from Line 33-49. To recover the accuracy lost in the mixed precision ***GEMM***, on Line 33-43, we first regenerate the entries of $\boldsymbol{A}$ in FP64 on the fly and compute the residual $\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A}\tilde{\boldsymbol{x}}$ with $\boldsymbol{b}$ in FP64. The residual is calculated using a parallel ***GEMV*** matrix-vector product on the CPU with minimal communication. Taking advantage of the ease of regenerating $\boldsymbol{A}$, each process owning a diagonal block $\boldsymbol{A}(k,k)$ regenerates the entries in the block-column $\boldsymbol{A}(:,k)$ and multiplies it by $\boldsymbol{x}(k)$. The only communication needed for the procedure is a single MPI Allreduce at the end that sums the resulting $N \times 1$ vectors to $\boldsymbol{Ax}$. Once we obtain the residual,on Line 47-48, the solution discrepancy $\tilde{\boldsymbol{d}}$ is solved with mixed precision (FP32/FP64) and stored in double, then the solution x is updated. The discrepancy $\tilde{\boldsymbol{d}}$ is calculated using both ***TRSV_[UP|LOW]*** for two triangular matrix-vector solves on the CPU. The **Iterative Refinement** step is complete when the discrepancy converges below a threshold providing a solution converged to double precision accuracy.

Finding 1. *The Frontier CPU and GPU memory ratio is 1:1, but total available GPU memory is actually larger than the available CPU memory (over 30GB) due to CPU memory containing the OS, cached files, and libraries. This makes a traditional pipelined offload scheme for accelerators less useful. Moreover, codes should attempt to run as much as possible on GPUs given the performance advantages over CPUs and the larger high bandwidth memory.*

IV. Strategies for Tuning and Optimization

The theoretical constraints derived from hardware, software and algorithms can provide a ceiling and general guidance for performance tuning and optimizations. In this section, we

Algorithm 1 Distributed GPU HPL-AI

Input: N, B, P_r, P_c
Fill global matrix $\boldsymbol{A}$ with random numbers.
(1) Block LU factorization
On each MPI process p_{id} do in parallel:
for $k = 1,\ 2,\ 3 \ldots\ n_b$ **do**
  Synchronize all processes
  $P_{ir}, P_{ic} \leftarrow processmapping(k)$ // $\boldsymbol{A}_{k,k}$ owner process index
  (1a) Diagonal Update
  if $p_{id} == P(P_{ir}, P_{ic})$ **then**
    $\boldsymbol{A}(k,k) \leftarrow \boldsymbol{GETRF}(\boldsymbol{A}(k,k))$
    Broadcast $\boldsymbol{A}(k,k)$ to $P(P_{ir}, :)$ and $P(:, P_{ic})$
  end if
  (1b) Panel Update
  if $p_{id} \in P(P_{ir}, :)$ **then**
    Receive $\boldsymbol{A}(k,k)$
    $\boldsymbol{A}(k, k+1:n) \leftarrow$
      $\boldsymbol{TRSM\_L\_LOW}(\boldsymbol{A}(k,k), \boldsymbol{A}(k, k+1:n))$
    $\boldsymbol{U} \leftarrow \boldsymbol{TRANS\_CAST}(\boldsymbol{A}(k, k+1:n))$
    Broadcast $\boldsymbol{U}$ to processes in $P(:, p_{ic})$
  else
    Receive $\boldsymbol{U}$
  end if
  if $p_{id} \in P(:, P_{ic})$ **then**
    Receive $\boldsymbol{A}(k,k)$
    $\boldsymbol{A}(k+1:n, k) \leftarrow$
      $\boldsymbol{TRSM\_R\_UP}(\boldsymbol{A}(k,k), \boldsymbol{A}(k+1:n, k))$
    $\boldsymbol{L} \leftarrow \boldsymbol{CAST}(\boldsymbol{A}(k+1:n, k))$
    Broadcast $\boldsymbol{L}$ to processes in $P(p_{ir}, :)$
  else
    Receive $\boldsymbol{L}$
  end if
  (1c) Update Trailing Matrix
  $\boldsymbol{A}(k+1:n, k+1:n) \leftarrow \boldsymbol{GEMM}(\boldsymbol{L}, \boldsymbol{U}, \boldsymbol{A}(k+1:n, k+1:n))$
end for
$\boldsymbol{A}_{cpu} \leftarrow \boldsymbol{A}$
(2) Iterative Refinement
$x \leftarrow b/\mathrm{diag}(\boldsymbol{A})$
for $i = 1,\ 2,\ 3 \ldots\ i_{\max}$ **do**
  for $k = 1,\ 2,\ 3 \ldots\ n_b$ **do**
    $P_{ir}, P_{ic} \leftarrow processmapping(k)$
    if $p_{id} == P(P_{ir}, P_{ic})$ **then**
      Broadcast $x(k)$ to $P(:, P_{ic})$
      $r \leftarrow \boldsymbol{GEMV}(generate(\boldsymbol{A}_{cpu}(:,k)), -x(k), b)$
    else
      $r \leftarrow 0$
    end if
  end for
  Sum r across all processes using Allreduce
  if $\|r\|_\infty < 8N\epsilon(2\|\mathrm{diag}(\boldsymbol{A})\|_\infty\|v\|_\infty + \|b\|_\infty)$ **then**
    break
  end if
  $d \leftarrow \boldsymbol{U}^{-1}(\boldsymbol{L}^{-1}r)$ // using ***TRSV_LOW*** and ***TRSV_UP***
  $x \leftarrow x + d$
end for

propose and structure such a performance model to facilitate the following tuning and optimization discussion, with the goal of rationalizing and highlighting the performance implications and improvements. Importantly, the model we constructed is used solely as a guideline for tuning and is not a complete model since it is incapable of representing the total complexity. Thus one cannot directly backsolve for the optimal parameters from the model.

As described in the previous section, the HPL-AI benchmark can be divided into four main parts: Diagonal Update (GETRF), Panel Update (TRSM), Update Trailing Matrix (GEMM) and the post-factorization Iterative Refinement. We begin by particularly focusing on the LU factorization part, as it is asymptotically the most time consuming portion, $O(N^3)$. The total runtime is approximated as

$$\begin{aligned} T(LU) = T(GETRF) + T(BCAST\_DIAG) + \\ T(TRSM) + T(BCAST\_PANEL) + T(GEMM). \end{aligned} \tag{1}$$

The LU runtime is estimated by the relationship of each parameter, including problem size N and block size B, as well as library kernel performance and network performance. This guideline is then used in a parameter search to yield the highest performance at scale. The serial upper bound runtime per iteration in terms of each subproblem is

$$\frac{B^3}{GETRF\_fr(B)} + \frac{2 \times N \times B^2}{TRSM\_fr(B)} + \frac{N^2 \times B}{GEMM\_fr(N,B)}. \tag{2}$$

Let $GETRF\_fr$ denote the ***GETRF*** flop rate for the triangular factorization of the diagonal block (FP32), $TRSM\_fr$ for the panel solve on the upper and lower panel (FP32), and $GEMM\_fr$ for the matrix-matrix multiply (FP16/FP32) update. The performance of each subproblem is not trivially known as they all have a complex dependency on the block size B due to vendor library implementation decisions.

We construct a **projected upper bound** for the total runtime by multiplying (2) by the N/B iterations and including the process grid dimension (P_r, P_c) and parallel data transfer time yielding

$$\begin{aligned} T(\text{parallel}) = \frac{N \times B^2}{GETRF\_fr(B)} + \frac{N^2 \times B}{P_r \times TRSM\_fr(B)} \\ + \frac{N^2 \times B}{P_c \times TRSM\_fr(B)} + \frac{2 \times N^2}{P_r \times NBB} \\ + \frac{2 \times N^2}{P_c \times NBB} + \frac{N^3}{P_r \times P_c \times GEMM\_fr(N_L, B)} \end{aligned} \tag{3}$$

where N_L is local matrix dimension, $2 \times N^2$ is the size of each FP16 panel, and NBB is the network broadcast bandwidth. NBB encapsulates the size of transfer, distance of transfer, iteration effects, memory bandwidth and network bandwidth per node. The diagonal broadcast time and latency can be safely ignored in the final model because it is relatively small in comparison to the total runtime of the other portions.

A. Computation Optimization

We can see in the performance model that problem size N and block B critically influence the overall runtime, and so we discuss their optimal selection and the rationales below.

B selection: The size of B not only determines the number of FP32 operations but also impacts the performance of each individual subproblem. Most computational kernels favor the larger sizes of B. However, to achieve the maximum algorithmic parallel performance, it is necessary to determine a B such that kernels on the critical path (i.e. GETRF) do not degrade the overall performance. The best performance comes

from minimizing the time of each iteration step and not by the maximizing the flop rates for each subproblem.

The GPU-enabled BLAS libraries have distinct performance behaviors with respect to B on different systems and even with different versions of the same library. The general strategy is to search the parameter space of B and plot the performance of each subproblem to estimate for the total runtime. We thoroughly discuss B tuning results in Section V for both Summit and Frontier.

N selection: In HPL-AI, one has the luxury of choosing any N that delivers the highest performance. Due to the complex interactions between parameters and subproblems, choosing the largest N does not always yield the highest performance. In our study, we consider local square blocks, or $N_{Lr} = N_{Lc}$, and $P_r = P_c$ with the local matrix size (N_L) being a multiple of B; therefore, $N = N_{Lr} \times P_r = N_{Lc} \times P_c$. The reasoning is that this creates a matrix of full blocks without needing padding on any node. We record the N_L tuning for Frontier in Section V.

B. Communication Optimization

As performance increases through reduced precision computations, the fraction of time taken in network communication becomes a dominant bottleneck. In this section, we provide comprehensive optimization information on how to reduce communication costs by overlapping the communication and computations not on the critical path and how to improve the broadcast performance.

Look-ahead: Look-ahead is a standard optimization included in most HPL and HPL-AI implementations. The idea is not to perform the full **Update Trailing Matrix** phase in the current iteration but instead to update the trailing matrix only during the next iteration's **Diagonal Update** and **Panel Update**. The procedure completes the next iteration's **Diagonal Update** and **Panel Update**, proceeds with the panel broadcast, then finishes the full **Update Trailing Matrix** phase from the previous iteration. This optimization overlaps the panel broadcast communication and ***GEMM*** computation, changing the last two terms of (1) to $max[T(BCAST\_PANEL), T(GEMM)]$. HPL-AI can be communication bound depending on the performance of the network and BLAS kernel, and so finding the parameters that balance the two overlapped terms becomes important.

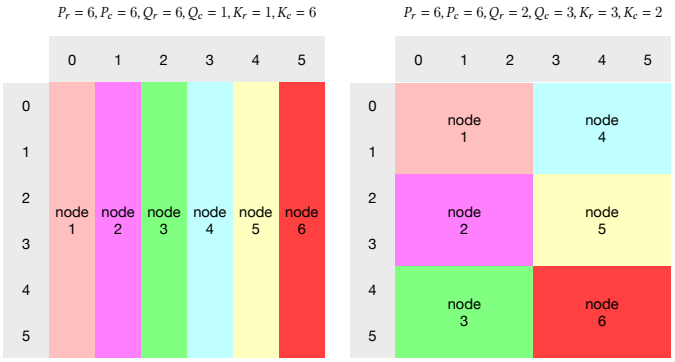
Node local grid tuning: We assume each node on the system contains Q GCDs and the binding of GCDs to MPI-ranks is using a node local grid $Q_r \times Q_c$, where $Q = Q_r \times Q_c$. A 2D partition with this node local grid results in a node layout $K_r \times K_c$, where $K_r = \frac{P_r}{Q_r}$, $K_c = \frac{P_c}{Q_c}$. The amount of data that one node has to transfer through the network (cross-node NICs) is

$$Data\_Size = \left(\frac{2 \times N^2}{K_r} + \frac{2 \times N^2}{K_c} \right). \quad (4)$$

Since the inter-node communication goes through the NICs, its bandwidth is significantly smaller than the bandwidth of intra-node communication. Letting NBN be the network bandwidth per node, which is easier to quantify, we now redefine our communication time to consider the shared NICs effect on inter-node communication, obtaining

$$T(\text{inter-node comm.}) = \left(\frac{2 \times N^2 \times Q_r}{P_r \times NBN} + \frac{2 \times N^2 \times Q_c}{P_c \times NBN} \right). \quad (5)$$

This model captures the impact of NICs being shared by processes on the same node. To minimize the communication volume and the total time we suggest having $K_r \approx K_c$, $P_r \approx P_c$ and $Q_r \approx Q_c$. We provide an example of two node-level grid configurations in Figure 2 and present numerical results with various node local grid tunings in Section V.



Distance of row broadcast	node 1 to 6	node 1 to 4
Distance of column broadcast	node 1	node 1 to 3
Transfer data size	$2N^2 + 0$	$\frac{2N^2}{3} + \frac{2N^2}{2}$

Fig. 2: Example of different node local grid, and the metric that impact the communication time.

Communicator Choice: In order to extract the best broadcast bandwidth of the system, we have implemented and tested several broadcast strategies for both systems. We consider the library based MPI broadcast (Bcast), MPI nonblocking broadcast (IBcast), single ring-based broadcast (Ring1), modified-ring (Ring1M), and modified double-ring (Ring-2M). Ring-based broadcasts are designed to overlap the communication into a pipeline, which increases the total effective network bandwidth at the cost of higher broadcast latency and can potentially reduce the length of the critical path.

The critical path is the sequence of the diagonal blocks that must be factored and distributed. Four synchronized MPI library broadcasts are needed to send the required data to the next diagonal block. Ring-based broadcasts effectively shrink the communication latency of the critical path, by breaking the MPI library broadcast into point-to-point sends and receives. The potential downside of a ring-based broadcast strategy is that it may increase the end-to-end latency of the full communication. Rings may also preclude potential vendor optimizations in their MPI implementation. The details of ring broadcast can be found in HPL [23], [24]. We describe the performance implications of different communication strategies on both Summit and Frontier in Section V.

V. Evaluation and Analysis

A. Methodology

Measurement: The units of performance used are GFLOPS/GCD (effective GFLOP rate per GCD). A Summit V100

GPU is shown as one GCD, while a Frontier MI250X GPU has two GCDs. Each MPI process (rank) is mapped to a single GCD. For distributed performance, HPL-AI average GFLOPS/GCD is thus calculated as $\frac{(2/3)N^3+(3/2)N^2}{P \times Run\_time}$, with flop count based on the HPL-AI submission rules. The performance we present in the figures are the highest performing runs, and run variability is discussed in Section VI-B.

Problem sizes: We set N near to the maximum the system can hold in GPU memory, excluding any strong scaling, since the HPL-AI benchmark's objective is to maximize sustained performance. We select the local matrix size per GCD, N_L = 61440 for Summit and, N_L = 119808 for Frontier, leading to $N = N_L \times P_r$. These values N_L correspond to approximately 14GB and 53GB of single precision matrix storage, and the rest of GPU memory contains the diagonal block (FP32), panel blocks (FP16) and additional look-ahead buffers (FP16). Frontier's N_L is smaller than the MI250X GCD capacity due to available CPU memory being smaller than the combined GPU memory as well as performance issues (discussed in Section V-D).

B. rocBLAS Performance

HPL-AI depends heavily on performance of the mixed precision FP16/FP32 ***GEMM*** matrix product. Figure 3 shows performance of this operation on a MI250X GCD for $\boldsymbol{C} = \boldsymbol{A}^T\boldsymbol{B}$ where $\boldsymbol{A}$ is $k \times m$ and $\boldsymbol{B}$ is $k \times n$. Noting that m (= n) is equivalent to blocksize B, one sees that highest performance (red) is not uniformly achievable across all matrix sizes encountered in a typical HPL-AI run. In particular, the optimal B of 3072 would generate highest performance only for a few matrix sizes. This is in agreement with our findings with other GPU vendors' BLAS libraries, that performance can depend in complex ways on the sizes of the input matrices.

Finding 2. *BLAS library behaviors affect performance of the full code, and applications heavily using BLAS operations can benefit from evaluating a performance heat map. Future tuning of libraries by vendors is expected to improve performance.*

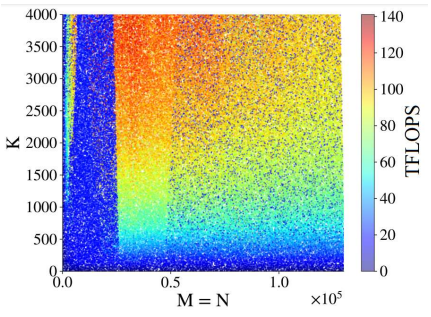


Fig. 3: rocBLAS GEMM flop rate as a function of matrix size.

C. Computation Tuning with Block Size B

To tune the block size B of the LU factorization, we examine the performance of each BLAS component on the two systems. In Figure 5, we plot the single cuBLAS V100 GPU performance of each computational component for all steps of the LU factorization. The x-axis represents the trailing matrix size, and the y-axis is performance, with each line representing a different block size B. Figure 6 shows the same set of plots with rocBLAS on one MI250X GCD.

Finding 3. *We observed that rocBLAS will require additional tuning of GEMM kernel parameters to achieve more uniform performance across the range of matrix sizes. The critical path includes rocsolver_getrf which has lower performance than expected. We anticipate these issues to be updated in later versions of rocBLAS.*

In Figure 5 and 6, the data shows that the flop rate for each operation grows with the block size B, as expected. However, as described in Section IV, maximizing the performance of each subproblem may result in lower total performance due to the increased number of single precision operations and the greater time spent in the critical path (***GETRF***). From the plot, we chose the optimal B based on the smallest B able to deliver an acceptable performance in ***GEMM***, ***GETRF***, and ***TRSM***. We also limited the runtime of ***GETRF*** to be less than 5% of the ***GEMM*** to better project the best B at scale. The results show B = 768 or 1024 for Summit's V100s and B = 3072 for Frontier's MI250Xs provide optimized performance.

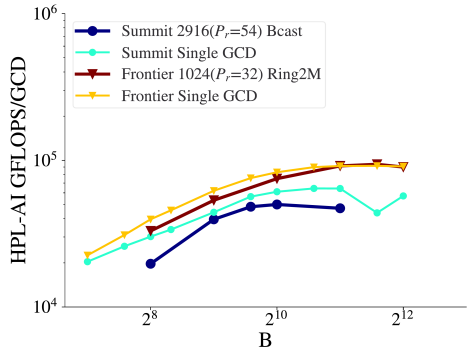


Fig. 4: Total performance relative to B with distinct communication layouts and scale.

In Figure 4, we verify the selected B values were near optimal by examining the flop rate per GCD in a distributed setting. The number of GCDs used was 2916 (P_r=54) on Summit and 1024 (P_r=32) on Frontier.

Finding 4. *The optimal value of B is crucial for total performance and depends on the problem size and BLAS library. This must be carefully chosen by benchmarking and evaluating imbalance issues caused by changes in the critical path's serial workload.*

D. Computation Tuning of N_L

We considered various N_L at 64 (P_r=8), 256 (P_r=16) and 1024(P_r=32) GCDs scale, and found that N_L = 119808 provides better performance over N_L = 122880. The Frontier performance drop at larger N_L is due to peculiarities in BLAS performance. The local matrix is stored on the GCD with the leading dimension equal to the initial problem size $LDA = N_L$. The trailing matrix update of each factorization step is performed on the submatrix without changing LDA. In Figure 7, we plot the ***GEMM*** flop rate from a single GCD.

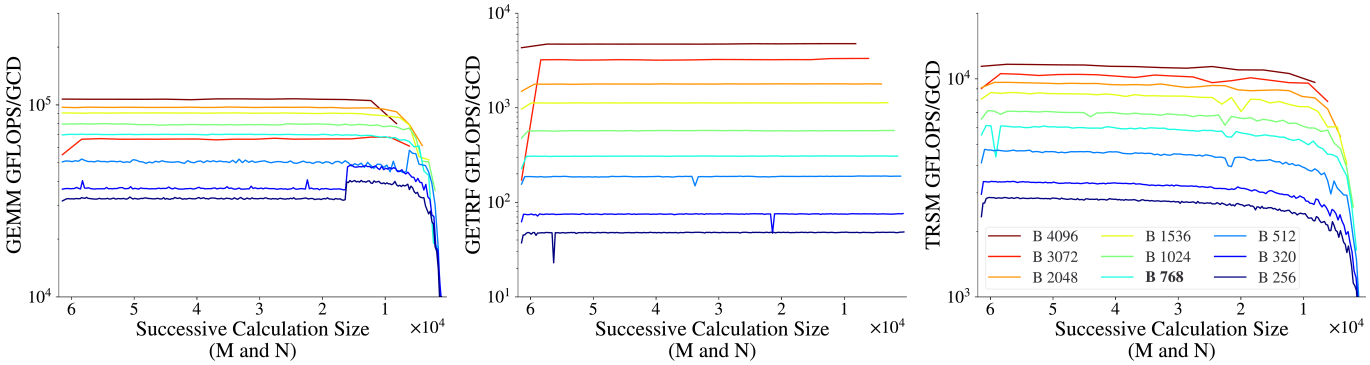


Fig. 5: Per iteration LU-Factorization performance of the ***GEMM***, ***GETRF***, and ***TRSM*** library kernel on a V100 GPU. Each color represents a different block size B. The trailing problem size proceeds from left to right on the x-axis.

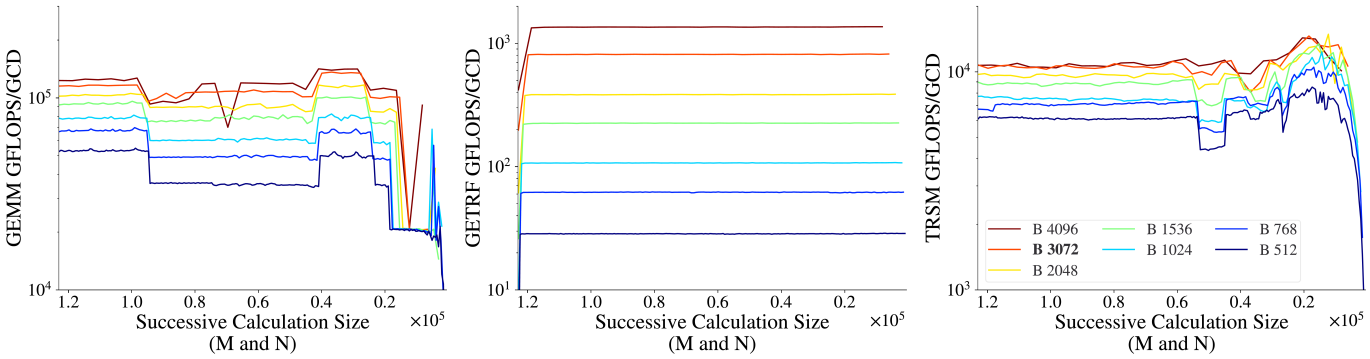


Fig. 6: Per iteration LU-Factorization performance of the ***GEMM***, ***GETRF***, and ***TRSM*** library kernel on a MI250X GCD. Each color represents a different block size B. The trailing problem size proceeds from left to right on the x-axis.

The x-axis represents the ***GEMM*** size in each iteration and the y-axis represents the flop rate that the ***GEMM*** kernel achieves. The legends shows the LDA of the local matrix, we observe that the rocBLAS ***GEMM*** performance was impacted by the leading dimension of the local matrix. LDA = 122880 has a significantly lower performance compared with the others.

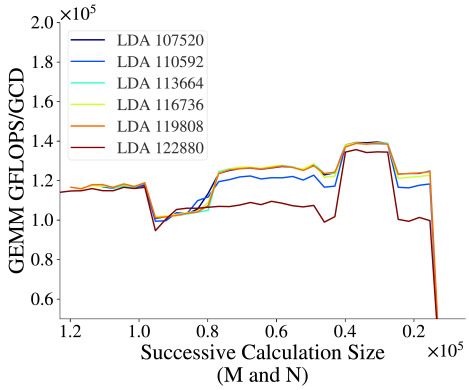


Fig. 7: Single MI250X GCDs GEMM performance on different GEMM size with different leading dimensions (LDA) with each color represents a different LDA.

E. Communication Tuning

Selecting a communication scheme and a local node grid also has considerable impact on performance. We denote the MPI library synchronized broadcast as Bcast and asynchronized broadcast as IBcast. The ring communicators are built with MPI point-to-point send and receives. In Figure 8, we plot the GFLOPS/GCD against different communications optimization strategies. The experiment was conducted with 2916 (P_x=54) GCDs for Summit and 1024 (P_x=32) for Frontier.

Port Binding: A Summit node contains two NICs that are each connected to a socket. Given that HPL-AI sends $B \times (N - k \times B)$ data per iteration, where k is the iteration number of the factorization step, we see the application having bandwidth bounded communication. Port binding to increase effective bandwidth yields a 35.6% to 59.7% overall performance improvement across different communication strategies on Summit with only a minor increase in latency.

Finding 5. *Port Binding features should be considered for bandwidth bounded communication for better performance.*

Ring Broadcast: The ring broadcasts are designed to decompose the synchronized broadcast into smaller point-to-point communications that can be pipelined to increase the effective bandwidth. We see a 20.0% to 34.4% overall performance improvement using different rings over MPI broadcast on the Frontier architecture, with ring2M as the best. However, we see a 2.3% to 11.5% decrease of performance on Summit. It is reasonable to assume that the Summit broadcast is highly optimized for the underlying fat tree network, whereas a ring

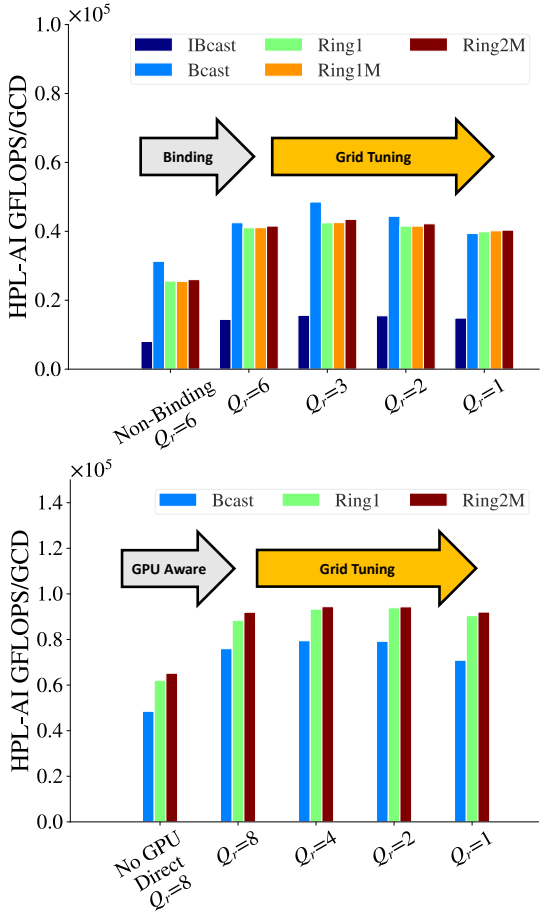


Fig. 8: Per GCD performance of distinct communication techniques and local node grid ($Q_r \times Q_c$)

broadcast may not significantly improve the bandwidth but may still increase the message latency.

Finding 6. *Ring broadcast currently outperforms the optimized MPI broadcast on Frontier. For applications that have similar communication patterns to LU factorization, alternative broadcast algorithms such as ring broadcast should be investigated as an option.*

GPU-aware MPI: GPU-aware MPI is designed to remove data transfer overhead for distributed GPU applications. On Frontier, NICs are directly attached to the GPUs, and we see a 40.3% to 56.6% overall performance improvement across all settings by sending diagonal and panel data directly from GPU memory.

Finding 7. *GPU-aware MPI has the potential to significantly improve performance of data transfers, and we verify this at scale on Frontier. Applications should make the problem resident on GPUs especially if they have both heavy computation and communication. The Frontier NIC is directly connected to the GPU which means CPU-based communication is less emphasized and node level network bandwidth may change based on the GPU binding. We note that there are temporary limitations with MPI on Frontier, such as not allowing a single MPI rank to control all 8 GCDs and utilize all 4 NIC ports.*

Node Local Grid Tuning: Optimizing the MPI rank mapping to the physical GCDs allows the broadcast to reduce inter-node communication volume and balance the impact of sharing the NICs on one side of the communication. It can also balance the communication distance (hops across network) between the process row and process column. Shown in Figure 8, Summit MPI broadcast obtains a 14.1% improvement with the $Q_r = 3$ over $Q_r = 6$, and Frontier Ring2M sees a 2.7% improvement with $Q_r = 2$ over $Q_r = 8$. The effect of grid tuning tends to be more observable as the scale increases as shown in the weak scaling study in Figure 9.

Finding 8. *Grid tuning has shown improvement across both systems, and we recommend similar 2D-partitioned applications to investigate a grid-based mapping that considers the physical GPU layout for optimal performance. Column-major node level mapping that uses a 3x2 grid on Summit and a 2x4 grid on Frontier appeared to work best. We observed that the grid tuning benefit on Frontier is not as strong as on Summit and that reduced improvement may be due to the unique node NIC architecture. Our performance model on a node local grid does not account for this behavior.*

In conclusion, Summit achieves the highest performance with (Broadcast, 2×3 Grid, **B**=768) which is a 603% improvement over the poorest settings. Frontier's highest performance configuration is (Ring2M, 4×2 Grid, **B**=3072) which achieves 94.6% improvement over the poorest settings. The high improvement rate on Summit is due to the asynchronous broadcast having extremely low performance with the current MPI library, IBM's Spectrum MPI.

VI. HPL-AI Performance at Extreme Scale

The design, performance model, optimization and tuning strategies and evaluations conclude in this section, as we present scaling results along with the *exascale achievement runs*, the best performance results obtained, conditioned on resource availability and system readiness.

A. Strong and Weak Scaling of HPL-AI

Analyzing the Summit results that use a column-major grid, we see that application performance under strong scaling is communication bound when performed at scale. The strong scaling behavior matches the communication and panel solve part of the performance model. We do not include the chart due to limited space.

Figure 9 illustrates the memory weak scaling characteristics of our implementation. Unlike a traditional weak scaling chart, we keep the required memory size for each GCD constant instead of the number of operations. The problem size is limited by GPU memory, and so memory-size weak scaling provides a better projection of the whole system's capability. Another difference is that we plot GFLOPS/GCD against the scaled number of GCDs, as opposed to the time to solution against GCDs, to track any degradation in parallel efficiency.

Weak memory scaling may result in an increase in GFLOPS/GCD at the beginning of the scaling plot due to the increased GPU workload and smaller fraction of run time in the serial

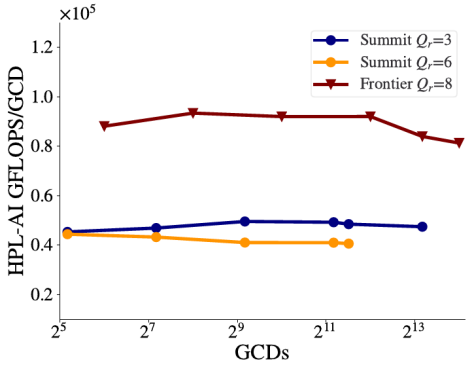


Fig. 9: Weak scaling with regard to memory size on both systems, with different node local grid settings

section. Eventually weak scaling flattens and then decreases as network communication overhead becomes more significant.

On Summit, we used 36 GCDs as the baseline to measure the parallel efficiency,

$$Parallel_{eff} = \frac{FLOPS/GCD \text{ on } \#\ of GCDs}{FLOPS/GCD \text{ on } 36\ GCDs}.$$

The column-major process mapping achieves 91.4% parallel efficiency on 2916 GCDs and a 3×2 Grid mapping achieves 104.6% parallel efficiency; the superlinear scaling behavior is due to the effects described above for weak memory scaling. On Frontier, we used 64 GCDs as the baseline and saw column-major achieving 92.2% parallel efficiency on 16384 GCDs. We suspect that the drop seen on Frontier at higher GCD counts is due to the interconnect fabric and will improve as the newly deployed system further stablizes.

Finding 9. *From the weak memory scaling data, we observe that tuning the process mapping can potentially improve the communication and scalability (measured by parallel efficiency) by as much as 10%, thus making it a valuable optimization technique.*

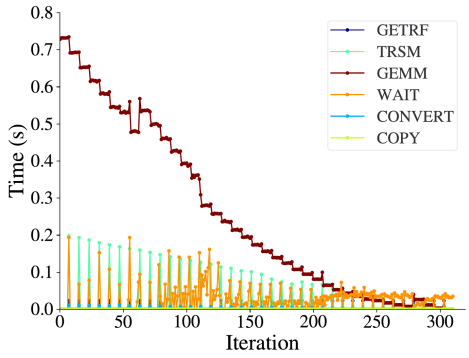


Fig. 10: Timing Breakdown of components per iteration on Frontier with 64 GCDs.

In Figure 10, we recorded the per iteration runtime for each kernel and the communication wait time. The data is based on MPI rank 0. We observed that the HPL-AI benchmark is computational bounded until the final trailing iterations.

B. Exascale Achievement Runs and Best Practices

In the end, as shown in Figure 11, we obtained our best number on Summit with a column major 3×2 grid, $P_r = P_c =$

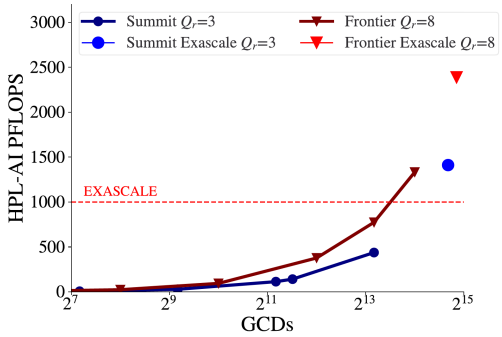


Fig. 11: Greater than Exascale performance on both of the OLCF leadership platforms, Summit and Frontier, with separate node local grid settings

162, $B = 768$, $N = 1368570$ at **1.411** EFLOPS. On just a fraction of the Frontier system, we obtained **2.387** EFLOPS with $N = 20606976$, $B = 3072$, $P_r = P_c = 172$, with Ring2M as the communication technique. It is important to note the disparity between the problem sizes solved. Even using less than half of Frontier, the N is over 20M compared with the 14M for Summit showing the potential to compute much larger problems on Frontier. We share the following best practices applicable to leadership system runs:

Identify slow nodes: Large systems such as Summit and Frontier contain tens of thousands of GPUs, and the performance of each GPU in such systems can vary due to manufacturing variability and nonuniformity of power/thermal management. On new systems we suggest to identify and exclude those nodes when running for top performance, since a single slow GPU can severely worsen total performance by stalling the pipeline. Using a mini-benchmark code, we scan through the GCDs, and thereby whole nodes, to exclude them from scaling runs. The mini-benchmark code is implemented with a single GPU *LU* factorization and an MPI aggregator to identify the slow GCDs. We observed approximately 5% maximum variation between GCDs on Frontier. We believe this difference will be minimized as Frontier moves to production.

Warm up: The system may require a warm up to achieve the expected performance. We have observed a performance difference larger than 20% between warm up and non-warm up runs on Summit. In the case of HPL-AI at scale, the total run times could be in the multiple hours range, so we developed a small warm up mini-benchmark to run prior to the HPL-AI run to improve the performance of the HPL-AI benchmark. In Figure 12, six full HPL-AI runs are launched consecutively within the same batch job, keeping allocated nodes constant. On Summit, we observe the first whole run is 20% slower than subsequent runs, which cap at a 0.12% performance discrepancy. An interesting Summit observation is that all compute kernels and communication are slower throughout the entire first run, not just the first few iterations. On the other hand, we observe on Frontier that the first two runs achieve higher performance than subsequent runs, which cap at a 0.34% performance discrepancy. We suspect the degradation of performance over runs is due to power, frequency and

thermal controls on the GPU. This is likely to be minimized as stability work continues on Frontier.

Finding 10. *In view of the data in Figure 12, the suggested strategy to warm up Summit is with a full run of the mini-benchmark to improve potential file system caching issues for binaries and dynamic libraries. Conversely, the strategy to warm up Frontier, if one has to, is to embed the small GEMM kernels at the beginning of the run, to produce the best performance. We suggest examining possible degradation behaviors due to warmup issues if this is a suspected problem for any specific application.*

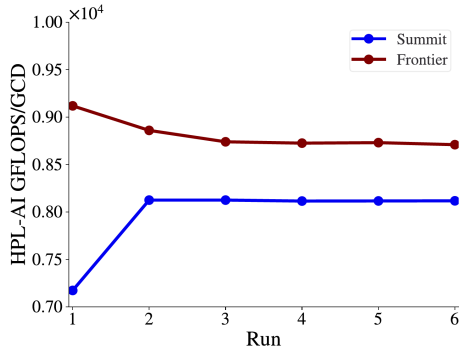


Fig. 12: Variability of performance with 2916 GCDs

Progress monitoring: Large runs at full scale are always at the peril of process and node failures and overall system instability. Since runs at scale may require hours to complete, without carefully monitoring the progress, one could inefficiently use these valuable node hours. It is therefore prudent to have built-in mechanisms to track and report the calculation's progress, and be able to terminate abnormal runs.

Our benchmark code has a detailed progress report for each component at definable iterations. We compare each component's performance to our previously recorded data in Figures 5 and 6. Other practices include monitoring the power utilization, which in our experience has often revealed early problems. We quickly terminate runs that incur a significant slowdown in performance. An example of progress output is given in Figure 10. We observed several fabric hangs during this Frontier run which could have been shutdown by our early termination mechanism to save system resources. We expect this to become less of a concern on Frontier as the system becomes more stable prior to production launch.

VII. Discussion and Lessons Learned

In this section we summarize lessons learned from porting, optimizing and running the HPL-AI on Frontier and Summit that are broadly relevant to leadership-scale HPC applications. HPL-AI achieves over 9× performance of HPL on Summit, and further performance on just a fraction of Frontier substantially exceeds peak double precision performance of the full Frontier system. This has potential impact on the performance of many science applications relying on distributed multiplication of dense matrices. While training deep learning models [25] is likely the most salient example, other applications include quantum chemistry [26], [27], nuclear physics [28], [29], image processing [30], and numerical methods ranging from solving linear systems to inverting a matrix [31] or using Newton's method for optimization problems [32], [33].

Frontier is a world-leading system deploying many new features, and, like its predecessors Jaguar, Titan and Summit, requires considerable time and effort to stabilize. Only part of Frontier was available for our use; thus our results are preliminary. Even so, we have found Frontier to be a capable, powerful system able to run HPL-AI effectively at record-achieving speed. For this work, portability layers like HIP were found helpful for cross-platform application development but do not capture all idiosyncrasies of library APIs from different vendors, thus requiring custom code in the form of macros, for example. This is true with respect to NVIDIA versus AMD GPUs and expected to be the case also for Intel GPUs.

Kernel benchmarks, though not fully representing performance of an operation as used *in situ* in an application, were nonetheless useful for tuning. Individual GPUs can have slight performance differences as the inevitable result of small manufacturing variances. If needed, the user can map these out of an application run to increase performance.

Certain features of Frontier enabled high performance of HPL-AI, such as NICs directly attached to the GPUs enabling efficient communication from GPU memory as well as the huge high bandwidth memories of the GPUs enabling high efficiencies for the GEMM computations. Also the large GPU memory with respect to CPU memory size obviates the need for double- and triple-buffering for memory management.

Codes like this have different performance regimes over the course of a run, requiring special considerations for optimization—in this case, compute-bound early in the run and communication-bound later. The node architecture of both systems has nonuniform interconnect speeds between the GCDs on the node, an important consideration for performance.

Accelerated applications must continue the trend of moving more computations to the GPU and, when appropriate, communicating directly between the system's GPUs. In addition to improving the performance, this would also reduce the associated carbon footprint by decreasing the execution time.

VIII. Conclusion

We have developed a cross platform implementation of HPL-AI that delivers state-of-the-art exascale performance on two of the world's fastest supercomputer systems, Summit and Frontier. We demonstrate how mixed precision can be utilized effectively to deliver defined double precision accuracy.

As we embarked on developing and optimizing this benchmark for Summit and Frontier, we focused on the flop rates of core BLAS routines such as ***GEMM***, ***TRSM***, ***GETRF***. As important as these key subproblems are, the performance sweetspot for parameters at this scale is an at-times elusive moving target, with shifting performance characteristics in each phase. Ultimately, no kernel particularly emerged as a singular bottleneck. The aforementioned techniques, such as reducing variability, overlapping compute and communication

as much as possible, judicious design of data placement and movement, combined with an architecturally well balanced system made this effort possible.

Further, we have reason to believe that full scale Frontier runs will be able to achieve 5 EFLOPS and that these developments demonstrate future opportunities for massively parallel applications on GPUs. The mixed precision routines can serve as a model for new techniques to be developed that enable new science to be conducted efficiently and effectively. Of great interest would be investigating how mixed precision operations effects the energy profile required for various calculations. One would expect that the improvements seen in performance would translate directly to energy utilization and sustainable computing which are ultimately critical for all data centers.

IX. Acknowledgment

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] AMD, "AMD Instinct MI250 Accelerator," Accessed Apr. 21, 2021. [Online]. Available: https://www.amd.com/en/products/serveraccelerators/instinct-mi250/

[2] TOP-500, "TOP-500 Benchmark," Accessed Apr. 21, 2021. [Online]. Available: https://www.top500.org/

[3] ICL, "HPL-AI Mixed-Precision Benchmark," Accessed Aug. 1, 2021. [Online]. Available: https://hpl-ai.org/

[4] G. Strang, *Introduction to Linear Algebra*, 5th ed. Wellesley, MA: Wellesley-Cambridge Press, 2016.

[5] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. USA: Society for Industrial and Applied Mathematics, 2002.

[6] R. Schreiber, "Block algorithms for parallel machines," in *Numerical Algorithms for Modern Parallel Computer Architectures*, M. Schultz, Ed. New York, NY: Springer US, 1988, pp. 197–207.

[7] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh, "Parallel algorithms for dense linear algebra computations," *SIAM Rev.*, vol. 32, no. 1, p. 54–135, Mar. 1990. [Online]. Available: https://doi.org/10.1137/1032002

[8] J. H. Wilkinson, *Rounding Errors in Algebraic Processes*. USA: Dover Publications, Inc., 1994.

[9] J. Kurzak and J. Dongarra, "Implementation of the mixed-precision high performance linpack benchmark on the cell processor," *University of Tennessee Computer Science Tech Report*, no. UT-CS-06-580, LAPACK Working Note #177, 2006.

[10] W. Lei, Z. Yunquan, Z. Xianyi, and L. Fangfang, "Accelerating linpack performance with mixed precision algorithm on CPU+GPGPU heterogeneous cluster," in *2010 10th IEEE International Conference on Computer and Information Technology*, 2010, pp. 1169–1174.

[11] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers," in *The Fourth Symposium on the Frontiers of Massively Parallel Computation*, 1992, pp. 120–121.

[12] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects," vol. 180, no. 1, p. 012037, 2009.

[13] C. Brown, A. Abdelfattah, S. Tomov, and J. Dongarra, "Design, optimization, and benchmarking of dense linear algebra algorithms on amd gpus," in *2020 IEEE High Performance Extreme Computing Conference*, 2020, pp. 1–7.

[14] A. Haidar, H. Bayraktar, S. Tomov, J. Dongarra, and N. J. Higham, "Mixed-precision iterative refinement using tensor cores on GPUs to accelerate solution of linear systems," *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 476, no. 2243, p. 20200110, 2020. [Online]. Available: https://royalsocietypublishing.org/doi/abs/10.1098/rspa.2020.0110

[15] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, "Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 603–613.

[16] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra, "SLATE: Design of a modern distributed and accelerated linear algebra library," in *SC19:Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–18.

[17] S. Kudo, K. Nitadori, T. Ina, and T. Imamura, "Implementation and numerical techniques for one EFlop/s HPL-AI benchmark on Fugaku," in *2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, 2020, pp. 69–76.

[18] RIKEN-RCCS, "HPL-AI implementation for Fugaku," Accessed Apr. 21, 2021. [Online]. Available: https://github.com/RIKEN-RCCS/hpl-ai

[19] NVIDIA, "NVIDIA CUDA toolkit documentation," Accessed Apr. 21, 2021. [Online]. Available: https://docs.nvidia.com/cuda/index.html

[20] AMD, "AMD ROCm platform portal," Accessed Oct. 21, 2021. [Online]. Available: https://rocmdocs.amd.com/en/latest/

[21] AMD, "AMD HIP programming guide," Accessed Oct. 21, 2021. [Online]. Available: https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-GUIDE.html

[22] J. Dongarra, "Basic linear algebra subprograms technical (blast) forum standard ii," *IJHPCA*, vol. 16, pp. 1–111, 05 2002.

[23] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: past, present and future," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 9, pp. 803–820, 2003. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.728

[24] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[25] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–43, 2019.

[26] E. Epifanovsky, M. Wormit, T. Kuś, A. Landau, D. Zuev, K. Khistyaev, P. Manohar, I. Kaliman, A. Dreuw, and A. I. Krylov, "New implementation of high-level correlated methods using a general block tensor library for high-performance electronic structure calculations," *Journal of Computational Chemistry*, vol. 34, no. 26, pp. 2293–2309, 2013.

[27] E. Solomonik, D. Matthews, J. R. Hammond, J. F. Stanton, and J. Demmel, "A massively parallel tensor contraction framework for coupled-cluster computations," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3176–3190, 2014.

[28] C. Lau, E. Jaeger, N. Bertelli, L. Berry, D. Green, M. Murakami, J. Park, R. Pinsker, and R. Prater, "Aorsa full wave calculations of helicon waves in diii-d and iter," *Nuclear Fusion*, vol. 58, no. 6, p. 066004, 2018.

[29] P. Du, P. Luszczek, and J. Dongarra, "High performance dense linear system solver with resilience to multiple soft errors," *Procedia Computer Science*, vol. 9, pp. 216–225, 2012.

[30] S. Eswar, K. Hayashi, G. Ballard, R. Kannan, R. Vuduc, and H. Park, "Distributed-memory parallel symmetric nonnegative matrix factorization," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–14.

[31] X. He, M. Holm, and M. Neytcheva, "Parallel implementation of the sherman-morrison matrix inverse algorithm," in *International Workshop on Applied Parallel Computing*. Springer, 2012, pp. 206–219.

[32] S. Wang, F. Roosta-Khorasani, P. Xu, and M. W. Mahoney, "Giant: Globally improved approximate newton method for distributed optimization," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18, 2018, p. 2338–2348.

[33] E. Wei, A. Ozdaglar, and A. Jadbabaie, "A distributed newton method for network utility maximization–i: Algorithm," *IEEE Transactions on Automatic Control*, vol. 58, no. 9, pp. 2162–2175, 2013.