Optimizing High-Performance Linpack for Exascale Accelerated Architectures

Noel Chalmers
Advanced Micro Devices Inc.
Austin, Texas, USA
noel.chalmers@amd.com

Jakub Kurzak
Advanced Micro Devices Inc.
Oak Ridge, Tennessee, USA
jakub.kurzak@amd.com

Damon McDougall
Advanced Micro Devices Inc.
Austin, Texas, USA
damon.mcdougall@amd.com

Paul T. Bauman
Advanced Micro Devices Inc.
Austin, Texas, USA
paul.bauman@amd.com

***Abstract*—We detail the performance optimizations made in rocHPL, AMD's open-source implementation of the High-Performance Linpack (HPL) benchmark targeting accelerated node architectures designed for exascale systems such as the Frontier supercomputer. The implementation leverages the high-throughput GPU accelerators on the node via highly optimized linear algebra libraries, as well as the entire CPU socket to perform latency-sensitive factorization phases. We detail novel performance improvements such as a multi-threaded approach to computing the panel factorization phase on the CPU, time-sharing of CPU cores between processes on the node, as well as several optimizations which hide MPI communication. We present some performance results of this implementation of the HPL benchmark on a single node of the Frontier early access cluster at Oak Ridge National Laboratory, as well as scaling to multiple nodes.**

***Index Terms*—Accelerators, Exascale, GPU, Linpack**

I. Introduction

In June of 2022, the Frontier supercomputer housed at Oak Ridge National Laboratory (ORNL) debuted on the Top500 list of supercomputers in the top position with a High-Performance Linpack (HPL) [1] score of 1.1 ExaFLOPS [2]. Over twice the score of the previous top supercomputer, Frontier was the first supercomputer ever to achieve over one ExaFLOPS in HPL, marking it as the first true exascale machine. Shortly afterwards, AMD's optimized implementation of HPL, named `rocHPL` [3], was made open-source and freely available. A variant of this HPL implementation, with optimized communication performance provided by Hewlett Packard Enterprise (HPE), was run on Frontier to achieve over one ExaFLOPS of overall performance. In this paper, we detail some of the performance optimizations which helped achieve this score with the expectation that these performance optimizations provides useful information for users optimizing HPL on heterogeneous architectures.

HPL is one of many benchmarks designed to measure some aspects of a computer system. Other common benchmarks include the High-Performance Conjugate Gradients (HPCG) benchmark which stresses the system's main memory bandwidth and system-wide all-reduce performance, and the High Performance Linpack - Mixed Precision (HPL-MxP) benchmark which stresses the system's computational throughput of mixed- and lower-precision math operations. As with these other benchmarks, HPL effectively stresses several aspects of a computer system including the 64-bit floating-point computation rate, network bandwidth, and network latency for extended periods of time while drawing essentially the peak amount of power the system can use. This makes HPL an incredibly useful stress test for validating a new computer system's reliability and overall performance.

The high FLOP rate in HPL on Frontier is owed almost entirely to its GPU-accelerated node architecture and high-speed network. The presence of accelerators is a growing trend in high-performance computing. Indeed, as of June 2022 seven of the top ten supercomputers on the Top500 list have GPU-accelerated node architectures. In Frontier's case, each node is comprised of a single 64-core AMD EPYC CPU, four AMD Instinct MI250X GPU accelerators. The EPYC CPU and MI250X GPUs all leverage AMD's advanced packaging as Multi-Chip Modules (MCMs). The CPU socket is comprised of eight 8-core Core Complex Dies (CCDs) and an IO die, while each MI250X GPU is comprised of two Graphics Compute Dies (GCDs). The GCDs are connected to one another, and to the CPU socket, via AMD Infinity Fabric. With this architecture, the MI250X accelerators in each Frontier node contribute over 98% of the node's peak FLOPS rate.

The computation performed in the HPL benchmark is the solution of an $N \times N$ random linear system of equations with a blocked Gaussian elimination method with partial pivoting. The $N \times N$ matrix, A, is distributed into a $P \times Q$ MPI process grid via a 2D block-cyclic distribution for load balancing. Leveraging the high FLOP rate of the GPU accelerators in HPL requires careful consideration of the implementation of its four main phases, each with different computational character. These phases are:

- Panel factorization (FACT) - The leading NB columns of A are LU factorized. This involves NB small collectives among the P processes performing the factorization in order to find the pivot rows.
- Panel broadcast (LBCAST) - The trailing matrix below the current $NB \times NB$ diagonal block is broadcast to all other processes in the grid.
- Row-swapping (RS) - The NB pivots determined during FACT are applied to the remaining columns of A
- Trailing update (UPDATE) - A rank NB update is applied to the trailing submatrix of A. This phase consists of

a computationally demanding triangular inverse, a.k.a. DTRSM, and a matrix-matrix multiplication routine, a.k.a. DGEMM.

Classically, the vast majority of time in the HPL benchmark is spent inside DGEMM routines inside the trailing update phase. These routines are often highly optimized on different hardware which enables HPL scores to achieve significant fractions of the hardware's peak floating point computation rate.

As accelerators became more prevalent over the past decade, there have been several works which studied leveraging them in HPL. A natural approach is to use accelerators to improve the performance of the large DGEMM computations in HPL, keeping the matrix in the CPU memory and offloading the DGEMM operations to the accelerator by dividing it into smaller pieces and processing piece by piece, interleaved with data transfers. Endo & Matsuoka [4] first studied using Clear-Speed SIMD accelerators to improve DGEMM performance in HPL in a heterogeneous cluster where only some nodes contained accelerators. This idea of offloading the DGEMM work to accelerators was also applied for HPL on the Road-runner supercomputer by Kistler et al. [5] which used the IBM PowerXCell 8i accelerators. Fatica [6] described this approach for GPGPUs using CUDA, wherein they describe a pipe-lining strategy for moving sections of the input/output matrices to/from the GPU to accelerate both DGEMM and DTRSM routines and hiding this data motion behind the computation time on the GPGPU. Demonstrations of scaling of this pipe-lining strategy to full clusters were described by Wang et al. [7] and Rohr et al. [8]. Other authors considered similar approaches for other programming models such as OpenCL [9], and other accelerators such as the Intel Xeon Phi [10], and even clusters which mixed different types of accelerators [11].

In more recent works, several authors have noted the increases in computation rates of accelerators have out-paced the bandwidth improvements in the host-accelerator links. Indeed, some modern GPU accelerators, including the MI250X GPUs, also include specialized hardware units which further accelerate the compute rate of matrix-matrix multiplications. This has made the pipe-lining strategy described by Fatica [6] for accelerating DGEMMs and other routines in HPL impractical. In order to hide the data motion between host and accelerator, the amount of computation done in each kernel must be dramatically increased, usually leading to unreasonably large blocking parameters in HPL which induces bottlenecks in other phases. To alleviate this, Tan et al. [12] and Kim et al. [13] present HPL implementations on modern GPU accelerators where the entire problem is stored in the GPUs' memory, rather than host DDR, an idea that originally appeared in Kistler et al. [5]. This has the benefit of removing the need to move data for large computational routines to/from the accelerator. Several large MPI communication phases can then also leverage GPU-aware MPI routines to move data directly between different GPUs' HBM and leverage fast hardware links between GPUs on-node when available.

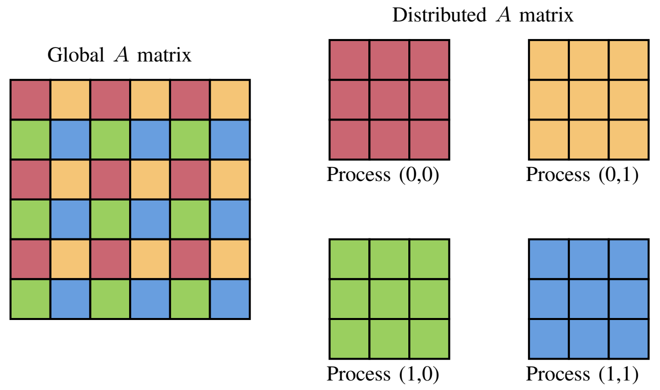
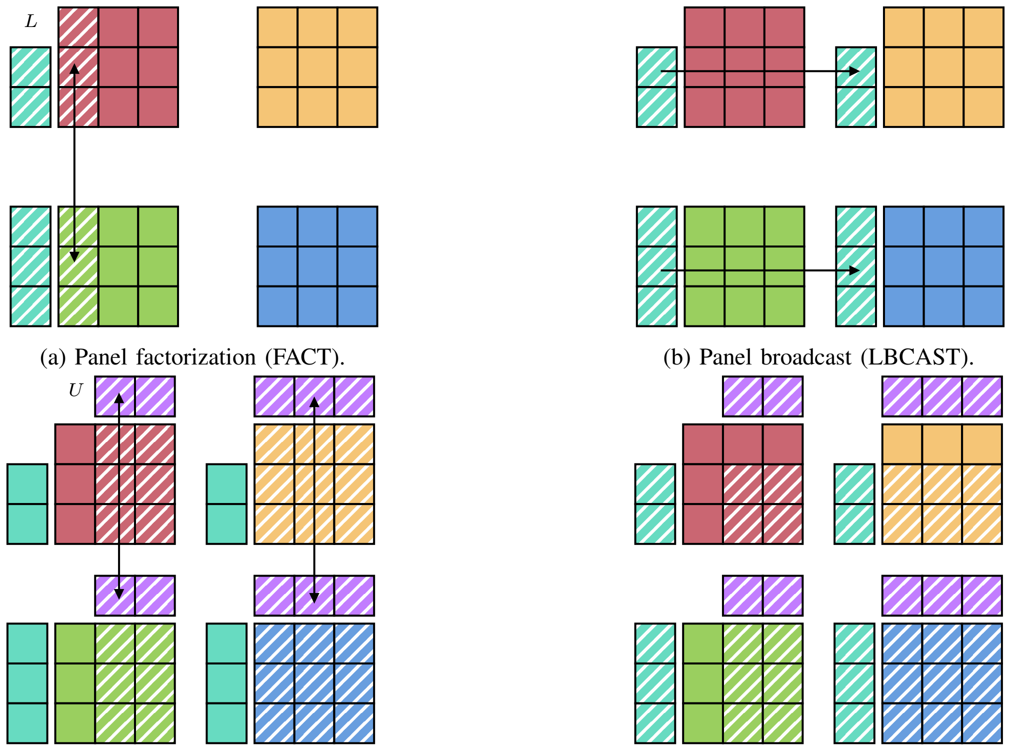


Fig. 1: 2D block cyclic distribution of a matrix. The global $N \times N$ matrix is blocked into $NB \times NB$ panels, which are distributed among a $P \times Q$ grid of processes. Figure shows an example of a distribution into a 2×2 process grid.

Some complications in this implementation arise, however. The panel factorization remains a complex communication- and latency-sensitive phase in HPL, which is not well-suited for fine-grain parallelism on accelerators. Furthermore, MPI communications must still be coordinated by the host process and overlapping these communications with useful work on the accelerator can be challenging.

Both Tan et al. [12] and Kim et al. [13] opt to utilize the host CPU to perform the panel factorization, transferring only the needed data from/to the GPU at each iteration. This reduced amount of data motion allows for the FACT and LBCAST phases to easily overlap with trailing update computation on the GPU. Communication required to perform the row-swapping phase then introduces GPU idle time, which both studies resolve by splitting the row-swapping and trailing update into several smaller pieces and pipelining to hide communication by smaller trailing updates. Tan et al. implement even further pipelining using multiple CPU threads to advance the panel broadcast phase while panel factorization progresses. This approach, which potentially causes some congestion of the network interfaces, is not used by Kim et al. who instead use NVIDIA's NCCL communication library for GPU-direct communication which uses GPU kernels for data motion, making overlapping with other communication impractical.

In this paper, we detail some of the optimizations we have implemented for HPL to improve performance on GPU-accelerated node architectures such as Frontier. In particular, we detail a multi-threading strategy for extracting data parallelism in the inherently serial panel factorization phase, overlapping CPU and GPU computation, and hiding GPU-GPU communication time via a split trailing update formulation. We then present some performance results of this implementation of the HPL benchmark on the Crusher cluster at ORNL showing the efficacy of our optimizations in hiding MPI communication time and demonstrating good scaling performance to multiple nodes, and afterwards give some concluding remarks.



(a) Panel factorization (FACT).

(b) Panel broadcast (LBCAST).

(c) Row-swapping (RS).

(d) Trailing update (UPDATE).

Fig. 2: The four main phases at each iteration of HPL's factorization algorithm. The figures show the phases on an example 2×2 process grid. In each phase, we show with a patterned fill what panels of each process's local matrix are accessed. We also show with arrows what processes communicate in each phase.

II. HPL Overview

The HPL benchmark begins by generating a distributed $N \times N$ double-precision matrix A on a two-dimensional grid of $P \times Q$ processes. For load balancing, the global matrix A is blocked into $NB \times NB$ sized panels, and these panels are distributed to the process grid in a 2D block-cyclic fashion. An example of this distribution is depicted graphically in Figure 1 for a 2×2 process grid. A length N right-hand-side vector $\mathbf{b}$ is also generated and appended to A to form an $N \times (N+1)$ augmented system.

The linear system is solved via a blocked Gaussian elimination algorithm, with partial pivoting. By treating A as an augmented system, the linear system $A\mathbf{x} = \mathbf{b}$ is essentially transformed into the upper triangular system $U\mathbf{x} = \hat{\mathbf{b}} = L^{-1}P^{-1}\mathbf{b}$, where $A = PLU$ is the LU-factorization of A with row-pivoting. After this transformation, the solution vector $\mathbf{x}$ is readily found by applying U^{-1}.

The blocked algorithm proceeds iteratively along the diagonal of A. Each iteration then consists of four main phases which themselves consist of varying levels of computation and communication between processes. To begin, at each iteration the block of NB columns at the current position along the diagonal of A are LU factored, applying row pivoting only within these NB columns and leaving the rest of the matrix unchanged. This phase, called the 'panel factorization' (FACT) stage, is shown graphically in Figure 2a for a 2×2 process grid. Only the leftmost block of panels is accessed, as shown in the patterned panels in the figure, and only in the processes which own sections of these NB columns. The processes participating in the panel factorization must frequently communicate in order to determine the NB row pivots to apply as the factorization progresses. Communication is indicated in the figure by arrows between these processes. Each communication to determine the distinct row to pivot is essentially a collective all-reduce operation involving all the processes in this process column, as the processes must collectively determine the pivot row and receive a copy of this row. At the end of this phase, the column of panels will have been pivoted and LU factored, yielding an $N \times NB$ lower triangular matrix L distributed between the column of processes.

Once panel factorization is completed, each of the processes which participated in the factorization packs their section of the L matrix, along with some index data conveying the pivoting information, into a buffer and broadcasts this buffer to all other processes in their row of the process grid. This step is graphically shown in Figure 2b. No computation is performed in this step, and only the data in the buffer holding L is accessed or modified. As the L matrix is typically large for the majority of the HPL benchmark, the performance of this phase is heavily dependent on the amount of bandwidth available for

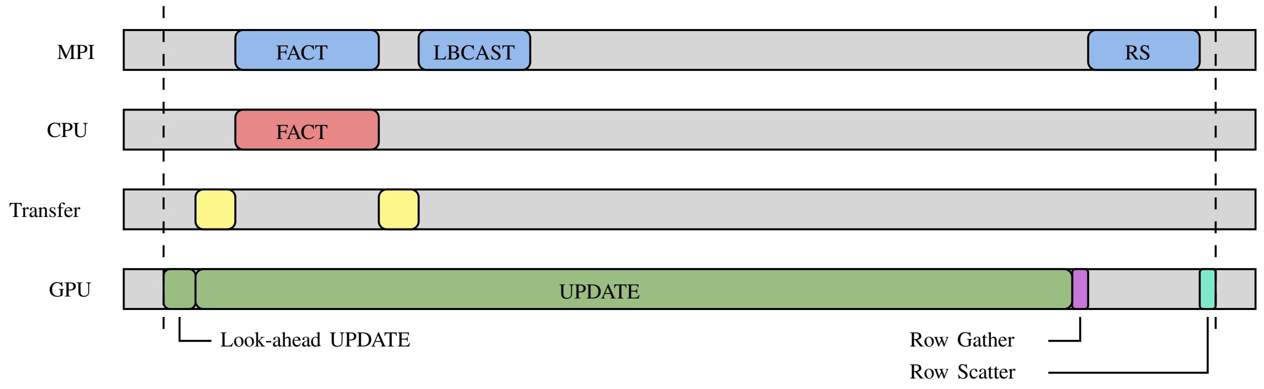


Fig. 3: Diagram of the execution of a single iteration in HPL's factorization. Diagram shows utilization of both CPU and GPU, as well as the data transfer between them. MPI activity in different phases is also represented on the timeline. When the walltime of the UPDATE phase is large, we observe that the computation on the GPU can effectively hide all phases except for RS.

inter-process communication, as well as the efficiency of the broadcast algorithm used.

With the L matrix and pivoting information broadcast to all processes, the final major communication phase in each process is to apply all the row pivots determined in FACT to the remainder of the A matrix on each process, and collectively construct the U matrix which resides to the right of the $NB \times NB$ currently factored panel. Since the full set of NB pivots are known in this phase, we can perform the required communication in bulk via routines equivalent to `MPI_Scatterv` followed by |`MPI_Allgatherv`|. Each process first assembles their rows to be communicated into buffers, which requires an irregular access of their local A matrix. Each process in the process row containing the currently factored panel then scatter the NB source rows to their destination processes in each process column via a `Scatterv` communication. Following this, all processes in a column collectively assemble their section of the distributed $NB \times N$ matrix, U, via an `Allgatherv` communication. The data accessed and the communication directions are shown graphically in Figure 2c.

The final phase of the iteration is the most computationally demanding but requires no inter-process communication. The pivoted rows of the global matrix have been assembled into the U matrix, and the computations from FACT are extended to these rows and applied as a single DTRSM routine, using the low-triangular piece of the factored diagonal panel. With the L and U matrices constructed and duplicated along the process rows and columns, respectively, the last computation is to apply a rank NB update to the trailing sub-matrix of A distributed among all the processes. This computation is a distributed $N \times N \times NB$ DGEMM which subtracts the product LU from the trailing sub-matrix of A. The data accessed for this computation is shown graphically in Figure 2d.

III. rocHPL Design

AMD's implementation of the HPL benchmark, named `rocHPL` [3], is based on the open-source HPL implementation hosted on Netlib [14]. This reference HPL code is parallelized with MPI, but otherwise contains no other parallel programming model. Our modifications to this HPL implementation involved adding GPU support via AMD's ROCm platform, runtime, and toolchains. The `rocHPL` code is written using the HIP programming language and leverages linear algebra routines highly optimized for AMD's latest discrete GPUs via the rocBLAS math library.

As noted above, the recent works of Tan et al. [12] and Kim et al. [13] both argue that the computational throughput of modern accelerators is so large that the entire matrix A must be stored in the accelerators' high-bandwidth memory (HBM), as moving data from/to CPU memory would be too costly. As the AMD Instinct MI250X accelerators contain specialized hardware accelerating the crucial DGEMM computations, computational throughput has been even further increased beyond even what these works consider. We must therefore follow a similar design in `rocHPL`, storing the matrix A across each of the MI250X's 128 GB HBM capacity.

It then becomes natural to consider whether all phases of the HPL benchmark should be performed on the accelerator, with the host process only serving to coordinate MPI communication. The UPDATE phase is, of course, a natural fit for the accelerator's high computational throughput. Likewise, the LBCAST and RS phases map easily to the accelerators as the required local data motion for row-swapping is accelerated using the GPUs' high memory bandwidth, and MPI communications can leverage both the high-bandwidth Infinity Fabric links between GPUs as well as the direct connection of the network interface cards (NICs) to the GPUs on node. The FACT phase, however, remains a challenge to execute on the accelerator. While it is true that many of the individual BLAS computations in FACT would be accelerated on the GPU, the communications required for row-pivoting would require frequent host-device synchronizations and would consequently introduce significant amounts of GPU idle time due to kernel launch latency. We therefore follow a similar approach to that of Tan et al. and Kim et al. and transfer necessary data back to the host processes in order for the FACT computations to be performed on the CPU before sending needed data back to the accelerators.

Fortunately, performing the FACT phase on the CPU leads to a relatively simple way to hide some necessary MPI communication by local computation using the ‘look-ahead’ mechanism in the HPL benchmark. By noting that the FACT phase of each iteration requires only the next NB columns of the matrix, the look-ahead works by splitting the UPDATE phase on each process which will be performing the FACT phase in the next iteration. These processes first perform the UPDATE phase on only the leading NB columns, and then immediately begin transferring these columns to the host for factorization while completing the UPDATE phase on the remaining local matrix. This approach leads to an iteration whose timeline of execution looks similar to that shown in Figure 3. When the UPDATE phase begins, the computations are performed on just the look-ahead first so that this section of columns can be transferred to the CPU, and then transferred back after the FACT phase. Once the FACT data arrives back on the GPU, the LBCAST communication can be done all while the remaining trailing UPDATE is being completed on the GPU. Processes which do not participate in the FACT phase simply wait in the LBCAST phase. After the UPDATE phase is completed, the row pivots computed in FACT are applied which requires a GPU kernel to gather the rows to be communicated, followed by MPI communication, and a GPU kernel to scatter the received rows back into A.

A. Multi-threaded Panel Factorization

At the beginning of the HPL benchmark, the computational work on the accelerator in each iteration can effectively hide both transfers to and from the host for the FACT computation, as well as the LBCAST communication. But as the benchmark progresses, the amount of work being performed in the UPDATE phase decreases until it is no longer able to hide these other phases. In order to maximize the duration of the benchmark where communication and factorization are hidden by UPDATE, and to spend the minimal amount of time without the UPDATE phase on the critical path, it is crucial to perform the FACT phase as fast as possible on the CPU.

The design in `rocHPL` is to let every MPI process manage one and only one GPU device. In the case of MI250X GPUs, where each GCD of the module presents to the OS as a distinct GPU, each MPI rank manages a unique GCD. Assuming each MPI rank is also bound to a distinct CPU core, this leaves potentially many unused CPU cores which can be leveraged in the FACT phase through multi-threading. While many CPU BLAS libraries offer multi-threaded implementations of computationally expensive BLAS routines, such as the DGEMMs and DTRSMs needed in FACT, we opt instead to multi-thread the entirety of the FACT phase by manually distributing the computation among CPU threads.

The matrix being LU factored in the FACT phase is tall and skinny. It consists of only NB columns, but potentially many thousands of rows. This makes it amenable to parallelization by distributing chunks of rows between threads on the host, an approach similar to the technique of Parallel Cache Assignment (PCA) by Castaldo et al. [15] and the work on parallel panel factorization by Dongarra et al. [16] and Kurzak et al. [17]. At the beginning of the FACT phase, we create an OpenMP parallel region of T threads and distribute work between threads by blocking the tall and skinny matrix into tiles of NB rows, assigning blocks in a round-robin fashion to each thread, as shown graphically in Figure 4. We choose square tile sizes purely out of convenience as this way the first tile, which will contain the upper-triangular factor as well as all the source rows during pivoting, is guaranteed to be assigned to the main thread.

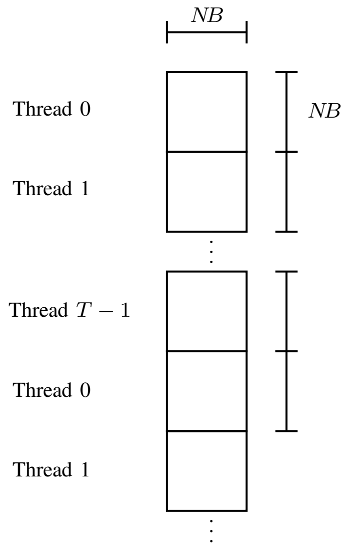


Fig. 4: Graphical depiction of the multi-threading strategy used in the FACT computation on CPU. The ensemble of NB columns is split into $NB \times NB$ tiles, and the tiles are round-robined between T CPU threads.

The original Netlib HPL code on which `rocHPL` is based contains several serial and blocked LU factorization methods, including left-looking, right-looking and Crout factorizations. Each of these is directly parallelizable with the tiling strategy. The determination of the pivot row is implemented as a parallel reduction over all OpenMP threads, after which the main thread calls MPI to complete the reduction across all processes in the process column. The main thread then applies the row pivot and synchronizes with the remaining threads so that the rank-1 update to the trailing sub-matrix can be applied in a parallel fashion using all the threads. For the blocked factorization methods, a similar idea is applied where the main thread performs the DTRSM updates to the upper-triangular factor, after which each thread uses the result to perform their section of the trailing update. With this approach, the data in each tile is accessed by only one thread, with the exception of any accesses by the main thread when applying row pivots. The data can therefore be kept resident in the CPU caches near

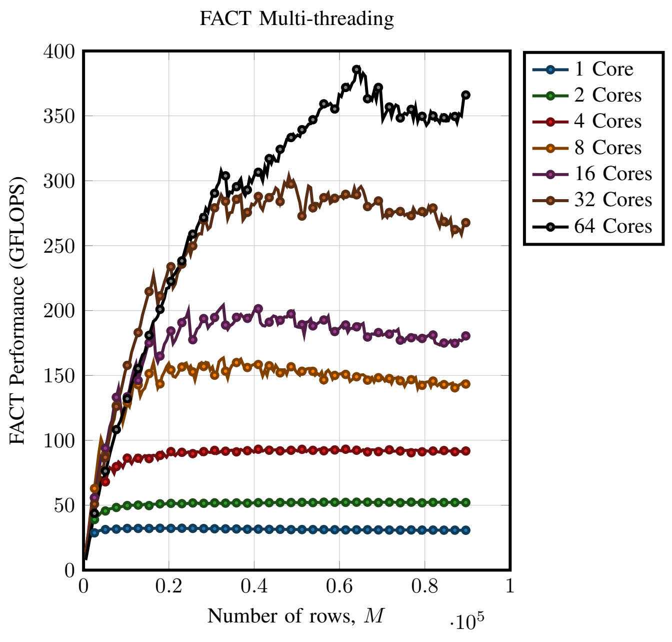


Fig. 5: Multi-threading performance test for FACT phase. Performance in GFLOPS is measured when factoring an $M \times NB$ matrix on a single process for $NB = 512$ and M a range of multiples of NB. Different curves show different numbers of CPU cores used in powers of 2 from 1 to 64.

the physical core to which that thread is bound. In addition, using the 64-core AMD CPUs on Frontier the entirety of the data accessed during the FACT phase typically remains resident in the L3 cache.

To demonstrate the performance benefits of this multi-threading strategy in the FACT phase, we show the results of a performance test in Figure 5 on a single Frontier node using BLIS v4.0 as the CPU BLAS library. The performance of the FACT phase when factoring an $M \times NB$ matrix is measured for $NB = 512$ and M taken to be various multiples of NB. We run this test with a single process to eliminate time which would be spent by the main thread determining and exchanging pivots with MPI. The factorization algorithm used is the recursive right-looking with two subdivisions in the recursion and a base block size of 16. On the base block, the factorization algorithm used is a right-looking factorization. We execute the FACT computation across these problem sizes using different numbers of CPU cores. From the figure, we see that the performance of the FACT phase is considerably improved through multi-threading and that using large numbers of CPU cores benefits performance for even the relatively small problem sizes.

B. CPU Core Time Sharing

With the multi-threading strategy for the FACT phase, an important consideration is where to place CPU threads to maximize the performance in each FACT computation. Consider the example of the Frontier node architecture; as there are eight GCDs present in the node, we launch eight MPI processes and bind each process to the CCD that is nearest to the GCD it will manage (c.f. the node topology diagram in [18]). A natural choice then is to have each process create seven additional OpenMP threads when it enters the FACT phase, so that the process can leverage all eight CPU cores in its CCD.

However, it is often possible for a process to utilize even more CPU cores than would be available through a simple partitioning of all available cores. Consider a 2D process grid of $P \times Q = 2 \times 4$ on the Frontier node architecture. At any given iteration of the computation in HPL only two processes will coordinate on computing the panel factorization while the other six processes are waiting to receive the LBCAST. If both processes performing the panel factorization each use eight CPU cores while the remaining six MPI processes each use a single CPU core, that leaves 42 idle CPU cores on the socket during this iteration. As the benchmark proceeds through iterations, the 16 total CPU cores being used during each FACT phase will cycle between different CCDs, but we will still always observe 42 total idle CPU cores in every iteration. With this observation, we consider a generic way to leverage all CPU cores in each HPL iteration by over-subscribing OpenMP threads to physical CPU cores.

In the general case of launching a node-local $P \times Q$ process grid to a node with C CPU cores, we bind each of the processes to a distinct root core and consider the remaining $\bar{C} = C - PQ$ cores as a pool of resources. This pool is partitioned into P non-overlapping groups, each with $\frac{\bar{C}}{P}$ cores, and each group is assigned to a distinct process row. Every MPI rank in each process column then uses OpenMP bindings to specify a total of $T = 1 + \frac{\bar{C}}{P}$ OpenMP threads and binds them to its root core and its process row's partition of the pool. In this way, every FACT phase will leverage a total of $PT = P + \bar{C}$ CPU cores on the node. In the extreme case of a $P \times 1$ local process grid on the node, this core binding reduces to a simple partitioning of available CPU cores, as all processes on the node must participate in the FACT phase simultaneously. At the opposite extreme of a $1 \times Q$ local process grid on the node, the amount of CPU core sharing is maximized since at most one process on the node will ever be computing the FACT phase at any given time.

In `rocHPL` we have implemented a generic wrapper script to compute these OpenMP bindings when launching the benchmark. The CPU core time sharing, as well as the decomposition of the global problem among the compute nodes, uses input from the user which describes the local process grid configuration desired on each node.

C. Split Update

Examination of the timeline view of execution in Figure 3 shows that the division of work between the host CPU and accelerator allows us to effectively hide the FACT and LBCAST phases behind the local UPDATE computation. However, the communication time required to perform the RS phase still leads to idle time on the accelerator. It is therefore

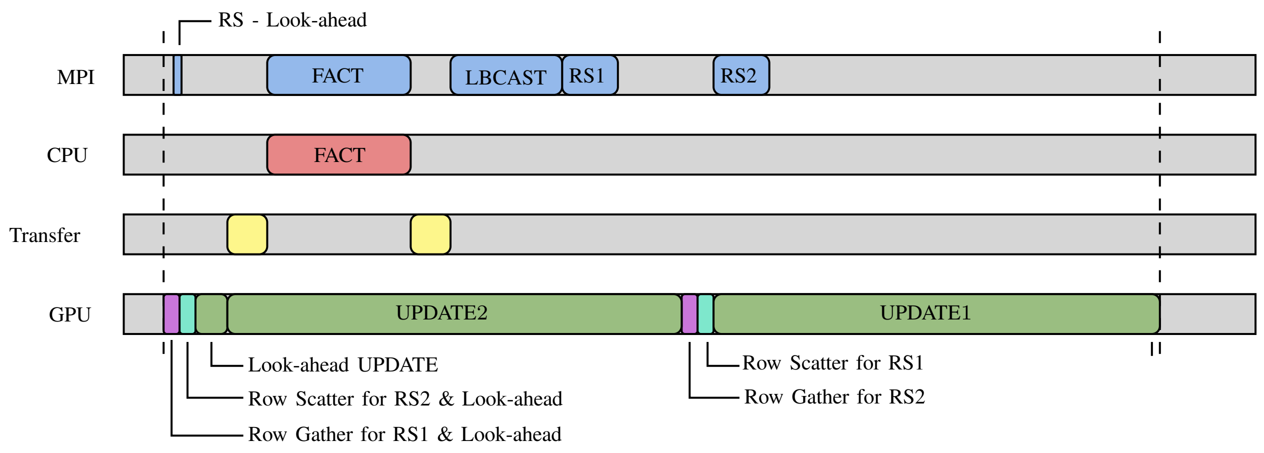


Fig. 6: Diagram of the execution of a single iteration in HPL's factorization with the split update formulation. When the execution time of the UPDATE phase is large, we observe that computation on the GPU can effectively hide all other phases in the HPL iteration.

advantageous to consider ways to hide this communication time with local computation as well.

A simple way to accomplish this communication hiding would be to divide the RS and UPDATE phases column-wise into several smaller chunks and apply a pipe-lining strategy. In this way, the local computation to perform the UPDATE phase on a chunk can hide the communication time for the RS phase for the next chunk. This strategy is what is applied in Tan et al. [12] and Kim et al. [13], who both use a multi-threaded implementation to coordinate the different chunks and different phases. Such a multi-threaded strategy is costly in our HPL implementation, however, as using multiple threads to pipe-line different phases utilizes CPU cores that could otherwise be used in FACT.

We opt instead for an alternative way to hide the communication time in the RS phase that requires no addition multi-threading, which we call a 'split update'. Let us denote by n the number of columns in the local section of A on a process at the start of the HPL benchmark. Note that due to the 2D block-cyclic distribution of A, n will be the same for all processes in each process column. Consider splitting the local matrix column-wise into two pieces with n_1 and n_2 columns which we call the 'left' and 'right' sections of the local A matrix, respectively. We select n_1 such that it is a multiple of NB. We denote by UPDATE1 and UPDATE2, the application of the UPDATE phase on the left and right sections, respectively, and likewise for the RS phase. The idea of the split update is to use the UPDATE computation on one section to hide the MPI communication of the RS phase of the other section. The key observation is that in order to do so, the needed rows for the RS stage on one section must be gathered before the UPDATE of the other section is started.

The split update formulation leads to a timeline of execution that resembles the one shown in Figure 6. At the start of an iteration, we assume that RS2 communication has already been completed. That is, we assume that the rows in the right section of the local A matrix have been communicated, though not necessarily scattered back into A. We begin the iteration by gathering the needed rows for communication from both the look-ahead and left section and scattering the communicated rows from the right section back into A. While the rows are scattered, the communication of rows for only the look-ahead is performed, and the received rows are written into the look-ahead. The iteration then proceeds as it does without the split update, with the UPDATE phase being performed on the look-ahead and the result copied back to the host for the FACT phase and subsequent LBCAST. While the transfers, FACT, and LBCAST are executed, the UPDATE2 phase is computed on the accelerator. However, since the rows of the left section of A have already been gathered for communication, the RS1 communication can also be performed at this point and be hidden by UPDATE2. Following UPDATE2, the rows for the next iteration in the right section of A are gathered to prepare for the RS2 communication. The UPDATE1 phase can then be queued, first scattering the communicated rows back into A, and the RS2 communication can be hidden by this local computation. Note that if this process performed the FACT phase, then the number of columns updated in UPDATE1 will be $n_1 - NB$.

Because of the staggered fashion in which the left and right updates are performed, interleaved with their respective row gathering and scattering, we must keep the number of columns in the right section of the local A matrix, n_2, on each process fixed for each iteration while n_1 decreases. Since we pick n_1 to be a multiple of NB, eventually n_1 will equal NB, and the look-ahead will then eventually become the entirety of the remaining left section. After this occurs, there is no longer a split update formulation, and the iterations fall back to the form shown in Figure 3 where the RS communication is not hidden by UPDATE.

For the split update formulation to effectively hide all communication time, the right section of the local A matrix must be at least large enough to hide the data transfers to and from the host, as well as the FACT, LBCAST, and RS1 phases. It is then natural to ask: if the UPDATE2 phase can initially hide all time spent in these phases, will it continue to hide them as the benchmark progresses? To determine this, note that since n_2 remains fixed while n_1 decreases, the UPDATE2 phase is always updating the same number of columns of A in each iteration for as long as n_1 remains non-zero. In terms of computational cost, while n_2 remains fixed the UPDATE2 computation scales linearly with the number of rows, denoted by m, in the local piece of A being updated on this process. Likewise, the other phases hidden by UPDATE2, except for the RS1 phase, share this linear scaling with m. Indeed, the data transfers to/from the host, the FACT phase, and the LBCAST phase each have a linear complexity in the number of local rows of A being updated in each iteration. The row-swapping communication in RS1, on the other hand, has a complexity which is linear in n_1, and is therefore decreasing as n_1 decreases. That rate is roughly the same rate as the local number of rows of A decreases. We can therefore conclude that if the UPDATE2 can initially hide each of these components, it will continue to hide them up until the point when the left section has decreased to zero columns and the RS2 communication can no longer be hidden. In practice, we have observed that this split update formulation is able to hide all MPI communication by UPDATE phases for approximately 75% of the execution time of the HPL benchmark on a single Frontier node.

Since it is crucial that n_2 be chosen only large enough to hide the FACT, LBCAST, and RS1 phases, its selection at the beginning of the benchmark is a key consideration in the split update formulation. While some performance models could be used to estimate the optimal size of n_2, we instead allow the user to input a ‘split fraction’ parameter to `rocHPL` to indicate what percentage of columns should be in the right section of A and leave this input as a tuning parameter. For HPL runs on a single Frontier node, we typically find that splitting the local A matrix in half between the left and right sections works optimally.

IV. Performance Results

In this section we present some performance results of the `rocHPL` benchmark on a single node, and scaled to multiple nodes, of the Crusher system at the Oak Ridge Leadership Compute Facility (OLCF). Crusher is a Frontier early access cluster and therefore shares the same node architecture as Frontier. Crusher is an HPE Cray EX supercomputer system with each node consisting of a single socket optimized 3rd Gen EPYC 64 core processor, four AMD Instinct MI250X accelerators, and four HPE Slingshot 200Gbps network interfaces, each directly attached to a distinct MI250X GPU. For all performance results below, we use GCC v11.2.0 as our C++ compiler, ROCm v5.4.0 to compile HIP kernels, and use the rocBLAS v2.46 GPU BLAS library distributed with ROCm v5.4.0. For the CPU BLAS library, we use BLIS v4.0. Finally, we use Cray-MPICH v8.1.17 as the MPI implementation.

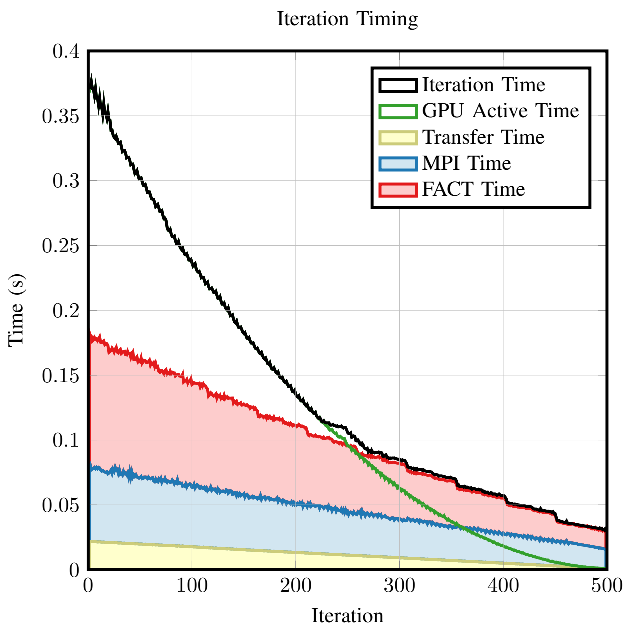


Fig. 7: Timing breakdown of each iteration in a run of the HPL benchmark on a single node of Crusher. The black line shows the total time at each iteration and the green line shows the time the GPU was active during the iteration. The stacked areas of red, blue, and yellow show the FACT computation time, the MPI communication time, and the host-device data transfer time at each iteration, respectively.

A. *Single Node Performance*

Beginning with a single node run on Crusher, we execute the `rocHPL` benchmark by launching eight processes in a $P \times Q = 4 \times 2$ process grid to a single node. We use a global problem size of $N = 256,000$ which, combined with necessary workspace buffers, effectively fills the HBM capacity of each of the four MI250X GPUs on the node. As is typical of HPL implementations, the choice of blocking factor NB is an important balance of computation and communication performance. The block size NB should be chosen at least large enough that the large DGEMM computations reach a high percentage of peak performance on the device, while choosing NB to be as small as possible allows for maximal overlap of communication and computation in each iteration. For the Frontier node architecture, we typically choose $NB = 512$ to strike this balance. At $NB = 512$ the DGEMMs required in HPL typically achieve 49 TFLOPS of performance on each MI250X GPU using the highly tuned DGEMM kernels available in rocBLAS. With this NB value, we also utilize a 50-50 left-right split in the split update formulation to hide MPI communication for row-swapping. With these parameters, the single node execution of the `rocHPL` benchmark on Crusher achieves on average 153 TFLOPS of performance overall.

We show in Figure 7 the timing breakdown of each iteration in the single node run of the `rocHPL` benchmark. For each iteration, the process which owns the current diagonal panel records several timers. We record the overall iteration time along with several other components, namely: the total time the GPU spent actively computing in this iteration, the total time spent sending data to and from the host, the total time spent in MPI communication, and the total time spent computing the FACT phase on the CPU. We plot both the per-iteration time along with the total GPU active time in this figure. The remaining three timers are shown in the figure as stacked lines to show the critical path of execution near the end of the benchmark.

From the behavior of the per-iteration time in Figure 7 we see two distinct regimes during the HPL benchmark execution. At the beginning of the benchmark, the per-iteration time precisely corresponds to the total GPU time in each iteration. This demonstrates that all other phases, including panel factorization and all MPI communication, are entirely hidden by GPU actively. Furthermore, 95% of the GPU active time in each iteration is typically spent inside DGEMM computations. We therefore achieve a high percentage of the achievable computational throughput of the node in this regime. Indeed, as each large DGEMM computation achieves 49 TFLOPS of performance, we have an absolute limit of $4 \times 49 = 196$ TFLOPS of computational throughput in this regime. The `rocHPL` benchmark prints a variety of performance metrics during execution, from which we typically see the running throughput in this regime achieve 90% of this limit, or 175 TFLOPS.

Around iteration 250, the left section in the split update is too small to adequately hide the RS2 communication and the per-iteration time can be seen to be slightly above the GPU active time. Soon after this, the left section of the split update becomes zero, and the right section shrinks until GPU activity is no longer on the critical path at all. From the stacked line plots of the host-device transfer time, MPI communication time, and FACT computation time in Figure 7 we see that these combined phases become the critical path of execution for the remainder of the benchmark execution. It is in this tail section that the running computational throughput of the benchmark decreases substantially to its final value, as the time per iteration in this regime is no longer compute-bound, but rather latency and communication bound. Nevertheless, using the optimization described above, the HPL implementation in `rocHPL` still achieves an overall performance of 78% of the achievable $NB = 512$ DGEMM computation rate of 49 TFLOPS per MI250X on a single Crusher node.

B. Multi-node Scaling

When weak scaled to multiple nodes, we expect the per-iteration time breakdown of the HPL run to follow a similar trend to that shown in Figure 7, albeit with more total iterations as the node count grows. However, while the computational work in each GPU and CPU socket remains the same as the problem is scaled, the MPI communication time is expected to grow compared to the single node results. This is due to two factors. First, the inter-node bandwidth uses the network

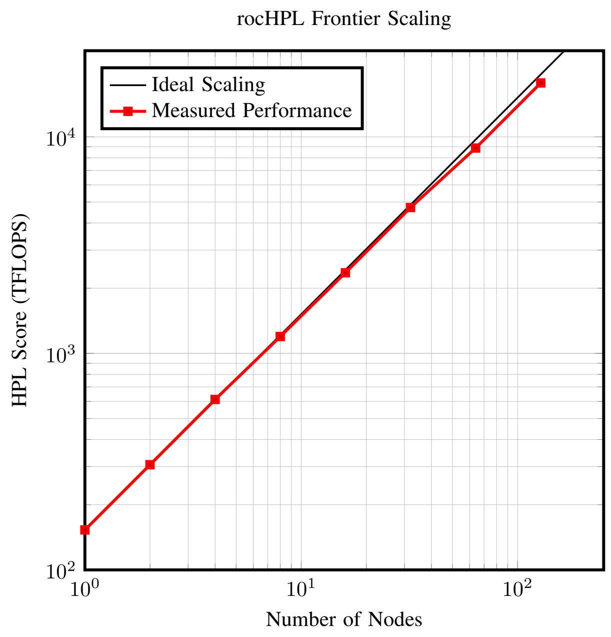


Fig. 8: Measured HPL score on multiple nodes of Crusher using the rocHPL benchmark. Benchmark is executed on $1, 2, 4, 8, \ldots, 128$ nodes.

interfaces which have less peak bandwidth than the Infinity Fabric links between the GCDs on a single node. This affects bandwidth-sensitive communication operations like those in the LBCAST and RS phases. Second, the latency cost of communications is expected to increase as the node count grows. This is important in latency-sensitive communications like the individual row pivots in the FACT phase, which are essentially MPI collectives across an entire process column.

We show in Figure 8 the performance of the `rocHPL` benchmark measured on $1, 2, 4, 8, \ldots, 128$ nodes of Crusher. We also show the ideal perfect weak scaling from the single node performance. For each node count, we keep the $P \times Q$ process grid square, or a grid with a 2:1 ratio of P to Q. For the node-local process grid, which determines the amount of CPU core time sharing that we can perform, we maximize the number of process columns on-node. That is, once Q is at least 8, we select the node-local process grid to be 1×8. We scale the global problem size, N, to again fill the GPUs' HBM capacity, and hold NB fixed at 512, and the left-right split at 50%, for all tests. We see in the figure that the `rocHPL` benchmark scales very well to multiple nodes, achieving over 90% weak-scaling efficiency from the single node score of 153 TFLOPS to the score of 17.75 PFLOPS on 128 nodes. Despite the relatively small node count for this test, this score would rank 38th on the November 2022 Top500 list.

V. Discussion

We have presented `rocHPL`, AMD's open-source implementation of the HPL benchmark targeting accelerated node architectures designed for exascale systems. As with other recent works on leveraging modern accelerators in HPL, `rocHPL` holds the entire problem in the accelerators' high-bandwidth memory and moves panels to the CPU only to perform the small latency-sensitive panel factorization. We detailed some performance optimizations used in `rocHPL` including a multi-threading strategy for improving panel factorization performance on the CPU, a method for time-sharing CPU core resources between different processes on the same node, and a split update strategy that can effectively hide communication time required for performing row-pivoting.

Detailed timing of the execution of `rocHPL` on a single node of the Crusher system shows that our optimizations are able to entirely hide MPI communications and CPU work behind GPU compute activity for the first 50% of the iterations in the benchmark. This, combined with the high performance DGEMM routines in rocBLAS, allows the benchmark to achieve a high percentage of the effective DGEMM computational throughput on each accelerator. Towards the end of the benchmark, the performance of MPI communications and the FACT phase on the CPU become the critical components. Our multi-threading strategy for the FACT phase helps to reduce the time spent in this regime.

The scaling performance for this HPL implementation is observed to be over 90% efficient when weak scaling to from a single Crusher node to 128 nodes. Full scale runs on machines such as Frontier of course require efficient scaling far beyond 128 nodes. For these large-scale runs, however, careful consideration of the performance of the MPI routines in HPL is required. It is likely that specialized communication algorithms, which optimize for the system's network topology, would be required to maintain efficient scaling, which is a topic outside the scope of this paper. Such optimizations are not present in our general implementations of these routines in `rocHPL`, but the code is designed to be modular so that users can easily implement their own custom routines and further optimize for their target systems/architectures.

The steady progression of generational leaps in computational throughput on accelerated node architectures continues to pose a challenge for benchmarks such as HPL which mix compute, network bandwidth, and latency sensitive phases. As the improvement of computational throughput outpaces inter-process communication performance, the performance bottlenecks shift away from being bound by computation rate and lowers overall performance, as measured by efficiency of peak computational throughput. Future works will have to address such shifts and carefully consider how accelerators can or cannot be further leveraged in the latency- and communication-dominated tail regime of the HPL benchmark.

Acknowledgment

The authors gratefully acknowledge the performance team at Hewlett Packard Enterprise and, in particular, Steve Whalen, Norm Troullier, and John Baron for their frequent discussions and insights. We also gratefully acknowledge the rocBLAS library team at AMD: Alex Brown, Andrew Chapman, Henry Ho, Raman Jana, Carson, Liao, Alex Liu, Wasiq Mahmood, Daine Mcniven, Koji Nakajima, Braga Natarajan, Benjamin Ulmer, Yoichi Yoshida, and Torre Zuk for all their work optimizing crucial BLAS routines on AMD Instinct GPUs.

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] 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.

[2] Top500.org. (2022). "June Top500 list," [Online]. Available: https://www.top500.org/lists/top500/2022/06/ (visited on 06/01/2022).

[3] [SW Rel.] N. Chalmers, *rocHPL - High Performance Linpack for Next-Generation AMD HPC Accelerators* version 6.0, 2022, Advanced Micro Devices Inc., URL: https://github.com/ROCmSoftwarePlatform/rocHPL.

[4] T. Endo and S. Matsuoka, "Massive supercomputing coping with heterogeneity of modern accelerators," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, IEEE, 2008, pp. 1–10.

[5] M. Kistler, J. Gunnels, D. Brokenshire, and B. Benton, "Petascale computing with accelerators," *ACM Sigplan Notices*, vol. 44, no. 4, pp. 241–250, 2009.

[6] M. Fatica, "Accelerating Linpack with CUDA on heterogenous clusters," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, 2009, pp. 46–51.

[7] F. Wang, C.-Q. Yang, Y.-F. Du, J. Chen, H.-Z. Yi, and W.-X. Xu, "Optimizing Linpack benchmark on GPU-accelerated petascale supercomputer," *Journal of Computer Science and Technology*, vol. 26, no. 5, pp. 854–865, 2011.

[8] D. Rohr, M. Bach, M. Kretz, and V. Lindenstruth, "Multi-GPU DGEMM and high performance Linpack on highly energy-efficient clusters," *IEEE Micro*, vol. 31, no. 5, pp. 18–27, 2011.

[9] G. Jo, J. Nah, J. Lee, J. Kim, and J. Lee, "Accelerating LINPACK with MPI-OpenCL on clusters of multi-GPU nodes," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 7, pp. 1814–1825, 2014.

[10] A. Heinecke, K. Vaidyanathan, M. Smelyanskiy, A. Kobotov, R. Dubtsov, G. Henry, A. G. Shet, G. Chrysos, and P. Dubey, "Design and implementation of the Linpack benchmark for single and multi-node systems based on Intel® Xeon Phi coprocessor," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IEEE, 2013, pp. 126–137.

[11] T. Endo, S. Matsuoka, A. Nukada, and N. Maruyama, “Linpack evaluation on a supercomputer with heterogeneous accelerators,” in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, IEEE, 2010, pp. 1–8.

[12] G. Tan, C. Shui, Y. Wang, X. Yu, and Y. Yan, “Optimizing the LINPACK algorithm for large-scale PCIe-based CPU-GPU heterogeneous systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 9, pp. 2367–2380, 2021.

[13] J. Kim, H. Kwon, J. Kang, J. Park, S. Lee, and J. Lee, “SnuHPL: High performance LINPACK for heterogeneous GPUs,” in *Proceedings of the 36th ACM International Conference on Supercomputing*, 2022, pp. 1–12.

[14] [SW REL.] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary, *HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers* version 2.3, 2018, Innovative Compute Laboratory, URL: https://netlib.org/benchmark/hpl/.

[15] A. M. Castaldo and R. C. Whaley, “Scaling LAPACK panel operations using parallel cache assignment,” *ACM Sigplan Notices*, vol. 45, no. 5, pp. 223–232, 2010.

[16] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek, “Achieving numerical accuracy and high performance using recursive tile LU factorization with partial pivoting,” *Concurrency and Computation: Practice and Experience*, vol. 26, no. 7, pp. 1408–1431, 2014.

[17] J. Kurzak, M. Gates, A. Charara, A. YarKhan, I. Yamazaki, and J. Dongarra, “Linear systems solvers for distributed-memory machines with GPU accelerators,” in *Euro-Par 2019: Parallel Processing: 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26–30, 2019, Proceedings 25*, Springer, 2019, pp. 495–506.

[18] Oak Ridge Leadership Computing Facility. (2023). “Crusher Quick-Start Guide,” [Online]. Available: https://docs.olcf.ornl.gov/systems/crusher_quick_start_guide.html (visited on 03/01/2023).