A Scalable and Portable Approach to Accelerate Hybrid HPL on Heterogeneous CPU-GPU Clusters*

Rong Shi[1], Sreeram Potluri[1], Khaled Hamidouche[1], Xiaoyi Lu[1], Karen Tomko[2], and Dhabaleswar K. (DK) Panda[1]

[1] *Department of Computer Science and Engineering, The Ohio State University*
{shir, potluri, hamidouc, luxi, panda}@cse.ohio-state.edu

[2] *Ohio Supercomputer Center*
ktomko@osc.edu

***Abstract*—Accelerating High-Performance Linkpack (HPL) on heterogeneous clusters with multi-core CPUs and GPUs has attracted a lot of attention from the High Performance Computing community. It is becoming common for large scale clusters to have GPUs on only a subset of nodes in order to limit system costs. The major challenge for HPL in this case is to efficiently take advantage of all the CPU and GPU resources available on a cluster. In this paper, we present a novel two-level workload partitioning approach for HPL that distributes workload based on the compute power of CPU/GPU nodes across the cluster. Our approach also handles multi-GPU configurations. Unlike earlier approaches for heterogeneous clusters with CPU and GPU nodes, our design takes advantage of asynchronous kernel launches and CUDA copies to overlap computation and CPU-GPU data movement. It uses techniques such as process grid reordering to reduce MPI communication/contention while ensuring load balance across nodes. Our experimental results using 32 GPU and 128 CPU nodes of Oakley, a research cluster at Ohio Supercomputer Center, shows that our proposed approach can achieve more than 80% of combined actual peak performance of CPU and GPU nodes. This provides 47% and 63% increase in the HPL performance that can be reported using only CPU nodes and only GPU nodes, respectively.**

***Keywords*-HPL, GPU, CUDA, Heterogeneity**

I. INTRODUCTION

It is becoming increasingly common for High Performance Computing clusters to use accelerators such as NVIDIA GPUs to push their peak compute capabilities. This trend is evident in the most recent (November 2012) TOP500 list [1] where 62 systems make use of accelerator technology. With GPUs being expensive and specialized compute resources, it is also common for large scale clusters to have GPUs on only a subset of their nodes, to limit system costs. The Blue Waters supercomputer with Cray XE6 (CPU nodes) and Cray XK7 (GPU nodes) is an example. Systems with such configurations have two levels of heterogeneity: intra-node heterogeneity (between CPU and GPU within a node) and inter-node heterogeneity (between nodes with GPUs and nodes without GPUs).

The High Performance Linpack (HPL) benchmark has been the yardstick to measure the performance of high-end supercomputing systems [2]. Optimizing performance of HPL on heterogeneous clusters with GPUs has attracted a lot of attention from the research community. Most of the research carried out has focused on optimizing the compute kernels on the GPUs and on addressing the intra-node heterogeneity between CPU and GPU. The version of HPL available from NVIDIA is an example of such an effort [3]. However, there is no public version of HPL available that can handle inter-node heterogeneity with CPU and GPU nodes. Hence, several clusters with heterogeneous CPU and GPU nodes report their HPL performance using either a pure CPU node configuration or a pure GPU node configuration; whichever yields the better performance. The Oakley cluster at the Ohio Supercomputer Center is an example of this.

Motivation: To demonstrate the true potential of a compute cluster, the HPL benchmark should be able to efficiently take advantage of all CPU and GPU resources that are available. The work distribution has to be balanced based on the compute power of each CPU and GPU node in order to achieve maximum performance. The latest version of NVIDIA GPUs and CUDA libraries provide features such as asynchronous kernel execution and DMA-based CUDA memory copies that provide efficient overlap between computation and data movement. An efficient design in HPL should take advantage of these features to boost HPL peak performance. The design should also consider other factors such as optimal MPI process placement and process to grid mapping in order to minimize communication overhead and load imbalance.

Earlier work on HPL for GPU clusters has not addressed many of these challenges. The version of HPL from NVIDIA provides efficient work distribution between CPU and GPU within a node, and takes advantage of the asynchronous CUDA features [3]. But, it does not address inter-node heterogeneity and does not address issues such as process to grid mapping. We refer to this version as NVIDIA's version of HPL in the rest of the paper. Work by Endo et. al. discusses a technique for inter-node heterogeneity but does not take advantage of asynchronous operations offered by CUDA for efficient overlap between computation and data movement. They reserve a CPU core per process to progress communication, thus leading to wasted CPU resources [4].

*This research is supported in part by National Science Foundation grants #OCI-0926691, #OCI-1148371 and #CCF-1213084. We would like to thank Doug Johnson (Ohio Supercomputer Center), Mark Arnold (The Ohio State University), Massimiliano Fatica and Everett Phillips (NVIDIA) for their inputs and support during this work.
978-1-4799-0898-1/13/$31.00 ©2013 IEEE.

Contributions: We propose a novel approach to run the HPL benchmark on heterogeneous CPU-GPU clusters. We present a two-level workload partitioning scheme that efficiently balances the workload across CPU and GPU nodes. Our approach also takes into consideration node configurations with multiple GPUs. We base our design on the version of HPL from NVIDIA [3], thus taking advantage of asynchronous CUDA operations to overlap computation and data movement. We propose a novel process grid rordering approach to minimize communication and load imbalance. Experiments using 32 GPU and 128 CPU nodes of Oakley Cluster show that our proposed designs can achieve more than 80% of the combined actual peak performance of CPU and GPU nodes. This provides up to 47% and 63% increase in the HPL performance compared to that using pure CPU node and pure GPU node configurations, respectively. We show the flexiblity and scalability of our approach by using different heterogeneous node configurations. In this paper, we make the following key contributions:

1) Propose a novel two-level workload partitioning approach that enables the HPL benchmark to take advantage of CPU and GPU nodes on a heterogeneous cluster.
2) Introduce a process grid reordering technique in our hybrid HPL to reduce communication overheads and improve load balancing.
3) Present detailed analysis of performance, efficiency, and scalability of our hybrid HPL design across different clusters with diverse configurations.

The rest of the paper is organized as follows. In Section II, we provide required background and present related work. We then discuss our proposed designs for a hybrid HPL on heterogeneous clusters in Section III. In Section IV, we present experimental results and detailed analysis. We conclude the paper in Section V.

II. Background and Related Work on HPL

The Linpack Benchmark [2] is widely used to measure the peak performance of supercomputer systems. The benchmark solves a dense system of linear equations, $Ax = b$, by applying LU factorization with partial pivoting followed by backward substitution. The overall workload of the benchmark can be estimated as: $(2/3)N^3 + 2N^2 + O(N)$. The LU factorization [5] [6] contributes to most of the execution time. Two kernels, DTRSM and DGEMM, dominate the time spent in LU factorization.

High Performance Linpack (HPL) has attracted a lot of attention from researchers over the years. LINPACK, provided by Netlib at UTK [2], has formed the basis for several optimized versions of the benchmark. Several researchers have also presented designs of HPL for GPU clusters. Fatica et. al. [3] implemented the first version of CUDA-based HPL for clusters with NVIDIA GPUs. The author proposed a host library that intercepts the calls to DGEMM and DTRSM and executes two operations simultaneously on both GPU accelerators and CPU multi-core hosts. Further, they utilize asynchronous memory copy and kernel launch operations to overlap computation and data movement. M. Bach et. al. [7] presented an optimized HPL version targeted at AMD GPU clusters. However, these versions handle only intra-node heterogeneity. T. Endo et. al. [4] utilizes an approach of launching a varying number of MPI processes on CPU and GPU nodes to handle inter-node heterogeneity. However, their work only parallelizes the DGEMM kernel on the GPU and reserves one CPU core for each MPI process to progress communication. The approach presented in this paper overcomes the limitations of the above approaches and handles inter-node heterogeneity on clusters with CPU and GPU nodes while utilizing all CPU and GPU resources for computation.

There has also been extensive research focusing on scheduling strategies for the HPL benchmark [8–14]. Using frameworks like StarPU [12] and OmpSs [10], which support asynchronous parallelism and heterogeneity, researchers have compared static and dynamic scheduling of HPL tasks. In this paper, we resort to static workload partitioning at the inter-node level and adopt dynamic workload partitioning at the intra-node level.

III. Proposed Framework for a Hybrid HPL

In this section, we first give an overview of the framework for our hybrid HPL benchmark. We then discuss the two-level hybrid workload partitioning that includes inter-node static and intra-node dynamic partitioning strategies. Finally, we describe the process grid reordering technique.

A. Proposed design for Hybrid HPL

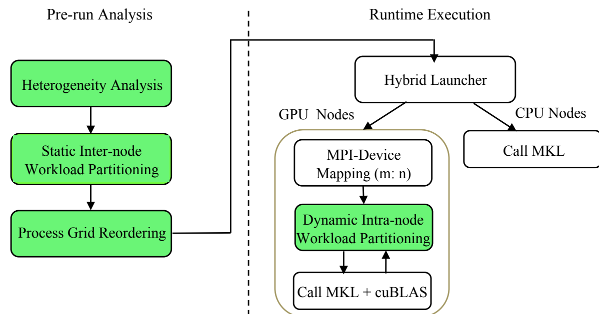


Figure 1. Overview of the Hybrid HPL Design

Our framework consists of two major parts: Pre-run Analysis and Runtime Execution. The first step in pre-run analysis is to identify the heterogeneity of a cluster. We detect three scenarios: “pure CPU nodes,” “pure GPU nodes,” and “heterogeneous CPU nodes + GPU nodes.” Note that we use “CPU nodes” to refer to compute nodes with only multi-core host processors and we use “GPU nodes” to refer to nodes with multi-core host processors and GPU accelerators.

The second step in the framework is to distribute data across nodes, taking the heterogeneity into consideration. In the case of a pure GPU node configuration, our design follows the model used in NVIDIA's HPL implementation [3]. We launch one MPI process per GPU and we use the "MPI+OpenMP" model to utilize all of the CPU cores. In the case of pure CPU nodes, our design uses the hybrid "MPI+OpenMP" model to flexibly select the number of MPI processes per node and the number of OpenMP threads per process to achieve maximum performance. In both cases, each node gets equal portion of the workload. To maximize utilization in a heterogeneous configuration, it is required that the workload distribution is proportional to the compute power of CPU and GPU nodes. We achieve this using a static MPI process-scaling based workload distribution. We explain this in detail in Section III-B. Then, our framework applies compute capacity-aware process grid reordering to generate an efficient node topology. This completes the pre-run analysis and the MPI launcher spawns processes on GPU nodes and CPU nodes accordingly. The runtime execution part of the framework decides the MPI process to GPU Device mapping and intranode CPU-GPU workload distribution on each of the "GPU nodes." The intranode CPU-GPU workload distribution is handled in a dynamic manner as is explained in later sections.

B. Two-Level Hybrid Workload Partitioning

The two-level workload partitioning consists of: static inter-node distribution during the pre-run analysis and dynamic intra-node work distribution during the runtime processing.

1) Inter-node Static Workload Partitioning: Figure 2(a) shows the standard cyclic distribution used in HPL with a 2×2 process grid and a configuration of 2 GPU and 2 CPU nodes. It shows the case when there is one process launched per node. Since work is distributed uniformly across all nodes, the overall performance is bound by that of the lower compute-capacity CPU nodes.

One way to handle this shortcoming is to proportionally distribute the workload between processes running on CPU nodes and processes running on GPU nodes based on the ratio of their computation capacities. This involves hybrid block-sizes as depicted in Figure 2(b). However, this approach is not very applicable for the HPL benchmark. All panel factorization happens across the diagonal of the matrix. With this approach, firstly, it is difficult to control computation and communication order with an asymmetric workload partitioning. Secondly, the computation of irregular sub-blocks in the update stage will incur too much overhead.

MPI Process-scaling Based Workload Partitioning: To avoid the overheads and complexities of hybrid block-size based partitioning, we use an MPI process-scaling based approach as shown in Figure 2(c)). Rather than partitioning

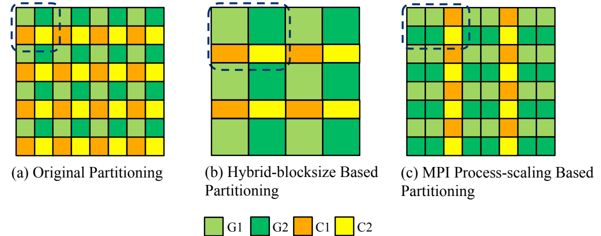


Figure 2. Strategies of Inter-node Workload Partitioning

the matrix into various sizes, this strategy conforms to the original strategy with the identical block size. To increase the workload ratio between GPU nodes and CPU nodes, this strategy schedules more MPI processes on GPU nodes to exploit their compute power while each MPI process (either CPU or GPU) get equal workload. First, based on general MPI binding, this strategy allows launching various number of MPI processes on CPU and GPU nodes. This flexibility is crucial to the portability of our partitioning strategy. With different cluster configurations, this strategy can adaptively adjust the number of MPI processes on CPU and GPU nodes.

The dashed rectangles in Figure 2 denote the process grid. Both original and hybrid block-size strategies use 2×2 grids even though hybrid block-size based approach adopts variable sized sub-blocks. The MPI process-scaling based strategy extends the grid to 2×3 by launching two MPI processes on each GPU node. Launching more MPI processes on GPUs will help boost the performance, but launching too many processes on a node is undesirable. With prior knowledge of actual peak performances of a single GPU node and a single CPU node, the number of MPI processes per GPU node (MPI_G) can be calculated as:

$$Ratio = Actual_GPU_Peak / Actual_CPU_Peak \quad (1)$$

$$MPI_G = Ratio + \delta \quad (2)$$

where $\delta \in \mathbb{Z}$ that makes $Num_CPU_Cores \bmod MPI_G = 0$.

In order to equally split the CPU cores across all MPI processes on each GPU node, our design tunes the value of δ and binds each MPI process on a GPU node with one GPU device and k CPU cores on host, where $k = Num_CPU_Cores / MPI_G$. Assuming $Ratio$ equals 5 and CPU host has 12 cores, the framework will adjust δ to "-1" and pessimistically launch 4 MPI processes on each GPU node accordingly. If we simply launch 5 MPI processes in this case, it will waste two cores in computation. Or in order to fully utilize all CPU cores, we may need to make two out of five processes have one more core. This case will cause computation imbalance. As shown in Section IV-B4, our design can adaptively adjust the number of MPI processes on each GPU node.

2) Intra-node Dynamic Workload Partitioning: The intra-node partitioning happens on GPU nodes only. Each MPI process splits the workload and offloads a larger portion of compute to GPU devices while leaving a smaller portion for

the CPU. NVIDIA's original design supports 1:1 mapping between MPI processes and GPU devices. In contrast, our design maps multiple MPI processes onto one GPU Node to ensure load balancing between CPU nodes and GPU nodes. This requires us to re-evaluate the ratio of intra-node workload partitioning between CPU and GPU. First, splitting the CPU cores across all of the MPI processes on each GPU node will reduce the CPU compute capacity associated with each MPI process. Further, the overhead resulting from sharing the GPU device by multiple MPI processes cannot be ignored. We have to increase the ratio to offset the first factor yet decrease the ratio to counteract the second one. We empirically adjust the initial CPU-GPU split to offset these overheads. NVIDIA's HPL version dynamically adjusts the CPU-GPU split as the execution progresses until it reaches the optimal balance. Our adjustments enable HPL to reach this equilibrium faster and thus allow for better performance.

C. Process Grid Reordering

Given the fixed number of MPI processes, there are different permutations of the two dimensional process grid. The optimal process grid on different clusters depends on the architecture of the physical interconnect. Based on the interconnect of our experiment platforms, an HPL benchmark test using T MPI processes and with a $P \times Q$ process grid will achieve peak performance if [15]:

$$\begin{cases} P \times Q = T, \text{ where } P \leq Q \text{ and } P, Q \in \mathbb{N} \\ \nexists P', Q' \in \mathbb{N}, \text{ that } P' \times Q' = T \text{ and } P < P' \leq Q' < Q \end{cases} \tag{3}$$

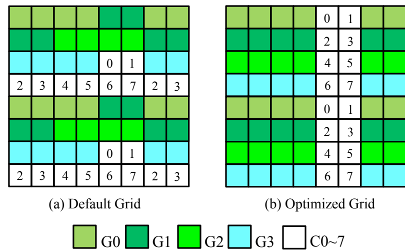


Figure 3. Workload Partitioning with Process Grid Reordering

Figure 3(a) illustrates an example of the default grid. We consider a 4×6 process grid for a 12-node configuration (4 GPU nodes and 8 CPU nodes), in which we launch 4 MPI processes on each GPU node and 1 MPI process on each CPU node. The default process grid does not provide optimal performance because of the inconsistent broadcast pattern across grid rows, and disproportional load balancing across GPU and CPU nodes. To solve these problems, this paper proposes a new process grid reordering method. Algorithm 1 shows how to generate an optimal grid with a given sets of parameters. First, we read the GPU and CPU hostfiles and calculate the total number of MPI processes. Then, we factorize this number and choose the initial $P \times Q$ grid. Based on the relationship between the number of GPU nodes and initial value of P, we either adjust the process grid or the placement of MPI processes for each GPU node. Then, GPU nodes are given higher priority and we place MPI processes on GPU nodes in top-down manner. Finally, CPU nodes are filled in the remaining slots.

In contrast to the default grid, the optimized grid exhibits better load balancing between GPU nodes and CPU nodes. In addition, panel broadcasts across MPI processes within one GPU node take advantage of the shared memory. Further, our method can be easily extended to support different cluster configurations. For instance, let's assume that a cluster has three types of nodes: GPU nodes with 2 GPU devices (2G-Nodes), GPU nodes with 1 GPU device (1G-Nodes), and pure CPU nodes. The inter-node partitioning strategy launches 6, 4, and 1 MPI processes on each of the three types of nodes, respectively. Then, the reordering method follows the "capacity-Priority" criterion to first place 2G-Nodes in the grid, followed by 1G-Nodes, and pure CPU nodes in order.

Algorithm 1: PROCESS GRID REORDERING

Input: g_hosts, c_hosts, mpi_g, $threshold\ \alpha = 0.5$
Output: Reordered Process Grid
1 $g(c)_num \leftarrow$ Calc GPU/CPU numbers by $g(c)_hosts$
  $t_mpi \leftarrow g_num * mpi_g + c_num$
2 $P, Q \leftarrow$ Choose initial P, Q based on Formula 3;
3 **if** $g_num * mpi_g < P$ **then**
    goto $label_rest$;
  end
  if $g_num * mpi_g * \alpha < P$ **then**
    Choose another pair of P' and Q' satisfying:
    no M exists that $P' < M <= g_num < P$
    in which $M * N = P' * Q' = t_mpi$
  end
  if $g_num < P < g_num * mpi_g * \alpha$ **then**
    while $i \in [2, mpi_g]$ *and* $g_num < P$ **do**
      $mpi_g \leftarrow mpi_g / i$;
      $g_num \leftarrow g_num * i$;
    end
  end
4 $gpu_per_row = g_num / P$;
  for $row \in [0, P)$ **do**
    Place $mpi_g * gpu_per_row$ GPU MPI processes;
  end
  label_rest: $rg_iter \leftarrow g_num \% P$;
  for $row \in [0, rg_iters)$ **do**
    Place mpi_g GPU MPI processes;
  end
5 Place CPU MPI processes top-down in remaining slots

IV. PERFORMANCE EVALUATION

In this section, we present experimental evaluation of our hybrid HPL benchmark and an analysis of the results.

A. Experimental Setup

We have used two different clusters in our experiments. Their specifications are in Table I. To demonstrate the

flexibility and portability of our design, we have conducted experiments using two different configurations on the Oakley cluster: "1G-Config", where each GPU node has one GPU accelerator, and "2G-Config", where each GPU node has two GPU accelerators. Cluster A only has the "1G-Config" configuration.

Specifications	Cluster A	Oakley Cluster
CPU Processor Type	Intel Xeon E5630	Intel Xeon X5650
CPU Clock	2.53GHz	2.66GHz
Node Type	two quad-core sockets	two 6-core sockets
CPU Memory	11.6 GB	46 GB
CPU Theo.peak (double)	80.96 Gflops	127.68 Gflops
GPU Processor Type	NVIDIA Tesla C2050	NVIDIA Tesla M2070
GPU Theo.peak (double)	515 Gflops/GPU	515 Gflops/GPU
BLAS Lib	MKL 10.3/cuBLAS	MKL 10.3/cuBLAS
Compilers	Intel Compilers 11.1	Intel Compiler 11.1
MPI Lib	MVAPICH2 1.9	MVAPICH2 1.9
OS	RHEL 6.1	RHEL 6.3
Interconnect	Mellanox IB QDR	Mellanox IB QDR

Table I
EXPERIMENTAL ENVIRONMENT

B. Analysis of Design Enhancements

In this part, we evaluate the performance impact of different aspects of our design. Figures 4 and 5 show the results with "1G-Config" on cluster A and "2G-Config" on Oakley, respectively. We used three different node combinations (4 GPU nodes with 4, 8, or 16 CPU nodes) on both clusters. We launch 4 and 6 MPI processes per GPU node on cluster A and Oakley, respectively. The performance impact of various aspects of our design is compared with that of the earlier design proposed by Endo et. al. [4]. In the graphs, we refer to our design as "OSU-HYBRID" and we refer to the simulation of design by Endo et. al. as "SIMU-ENDO."

1) Parallel DTRSM: The hybrid HPL design proposed by Endo et. al executes the DGEMM kernel on the GPU while using only the CPUs to execute DTRSM [4]. Based on NVIDIA's design, our implementation parallelizes DTRSM using both CPU and GPU. As shown in Figure 4(a) and Figure 5(a), DTRSM parallelization provides around 2-3% and 3-5% increase in overall HPL performance on Cluster A and Oakley, respectively. The benefits from parallelizing DTRSM are bounded by the percentage of HPL runtime spent in DTRSM and the overhead of creating and managing CUDA streams and events.

2) CPU Resource Usage: Each MPI process on a GPU node is assigned a subset of CPU cores and one shared GPU accelerator. Earlier design by Endo et. al. dedicates one CPU core per MPI process to process data movement between CPU and GPU. With the help of asynchronous memory copy and kernel launches, our design takes advantage of all of the CPU cores available for an MPI process. In Figures 4(b) and 5(b) we show the impact of fully utilizing CPU resources on the overall performance compared to a case when one core per MPI process is dedicated for data movement. Here, we achieve 8-10% and 13-15% improvement in performance with full utilization of CPU resources, compared to that using "SIMU-ENDO."

3) Process Grid Reordering: We demonstrate the impact of process grid reordering using a 4×6 process grid on the node configuration with 4 GPU nodes and 8 CPU nodes, using a problem size $N = 81,920$ and $NB = 512$. A portion of the grid with default and reordered mapping of processes is shown in Figure 3. Table II presents the workload distribution among processes with the default and optimized process ordering. Clearly, optimized process grid provides better load balancing, giving equal work to all processes running on CPU nodes and equal work to all processes running on the GPU nodes. In addition, Table III shows the decomposed time for panel factorization (rfact), broadcast (bcast), and sub-matrix update (update). We see that the optimized grid improves the execution time of all three parts due to improved computation and communication.

Process Grid	Number of Blocks				
	g0,3	g1,2	c0,1,6,7	c2,3,4,5	Total
Default	4320	4240	1040	1080	25600
Optimized	4320		1040		25600

Table II
SUB-BLOCK WORK DISTRIBUTION BETWEEN PROCESSES ON CPU AND GPU NODES (N = 81,920, NB = 512)

Process Grid	Total	Max rfact	Max bcast	Max update
Default	267.6	17.5	82.8	229.3
Optimized	245.6	12.4	72.1	222.8

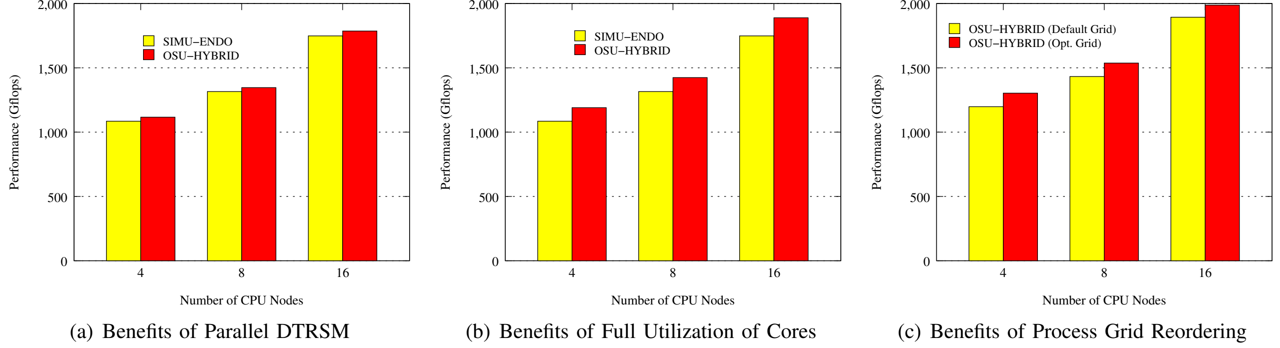
Table III
DECOMPOSED TIMING OF ALL PARTS

4) MPI Processes Scaling: Here, we present the performance results with varying number of MPI processes per GPU node on both Cluster A and Oakley. The goal is to achieve the best load balancing possible between the CPU and GPU nodes, as described in Section III. We focus on the performance of the update phase that dominates the overall execution time.

As shown in Figure 6, DTRSM takes a small portion of the overall runtime and it increases consistently with more number of MPI processes per GPU node. We attribute this fact to the context switching costs and scheduling overheads on GPU device. However, with DGEMM, where the kernel execution dominates the overall time, we see that increasing the number of processes on GPU nodes improves the load balancing between CPU and GPU nodes, thus improving performance. From Figure 6, we see that the best balance is achieved by launching 4 MPI processes per GPU node on Cluster A and on Oakley with "1G-Config," even though these clusters have different CPU specifications. Also, the best balance is achieved by launching 6 MPI processes on Oakley with "2G-Config." We use these configurations in the rest of our experiments.

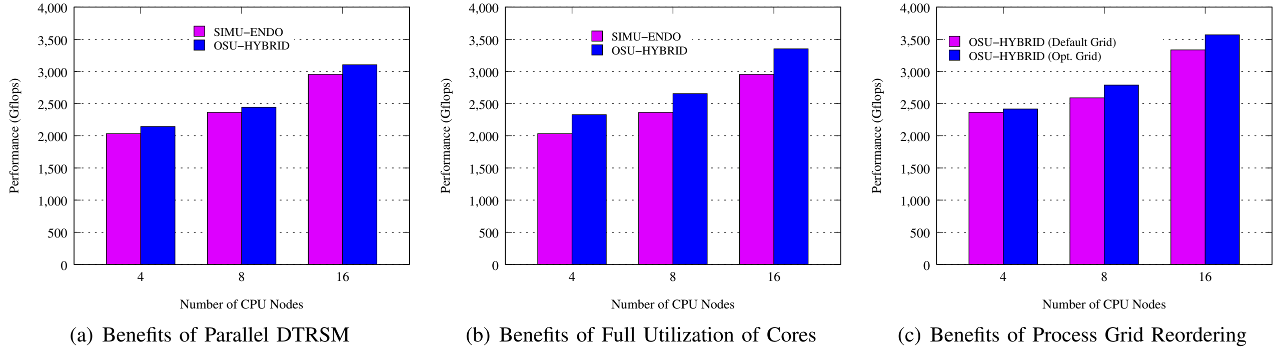
C. Evaluation of Peak Performance

In Figure 7, we present a comparison of peak performance achieved using different versions of HPL on "16 GPU



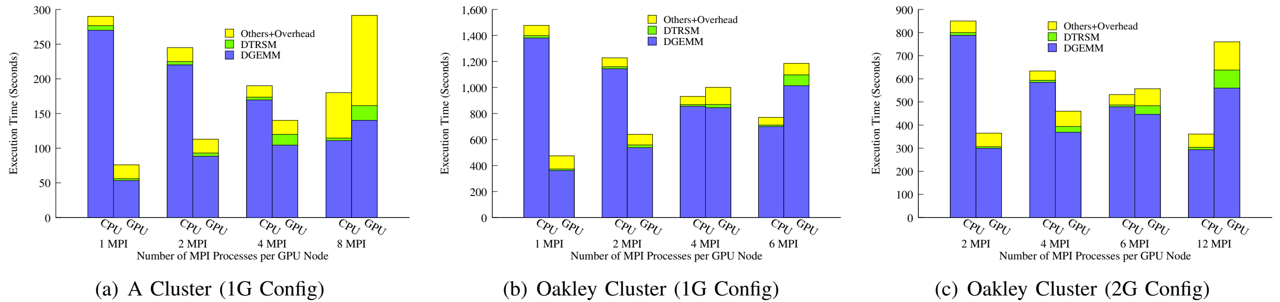
(a) Benefits of Parallel DTRSM (b) Benefits of Full Utilization of Cores (c) Benefits of Process Grid Reordering

Figure 4. Performance Evaluation on Cluster A with 4 GPU nodes and varying number of CPU nodes



(a) Benefits of Parallel DTRSM (b) Benefits of Full Utilization of Cores (c) Benefits of Process Grid Reordering

Figure 5. Performance Evaluation on Oakley Cluster with 4 GPU nodes and varying number of CPU nodes



(a) A Cluster (1G Config) (b) Oakley Cluster (1G Config) (c) Oakley Cluster (2G Config)

Figure 6. Update Time Decomposition Analysis with 4 GPU nodes and 16 CPU nodes

nodes + 64 CPU nodes" of the Oakley cluster. "Netlib-MPI-CPU" refers to the standard version of HPL from UTK that was used for reporting Top500 HPL performance of a pure CPU configuration of 64 CPUs on Oakley. "MPI+OpenMP-CPU" is NVIDIA's version of HPL which is run on the same pure CPU configuration. "MPI+OpenMP-GPU" is NVIDIA's version run on pure GPU configuration of 16 GPUs. NVIDIA's version of HPL requires the number of MPI processes per node to be equal to the number of GPUs per node. It uses OpenMP threads to utilize all the CPU cores. "SIMU-ENDO" and "OSU-HYBRID" are as described in Section IV-A. From Figure 7(a), we see that "OSU-HYBRID" can deliver better performance compared to the "SIMU-ENDO" in both "1G-Config" and "2G-Config" configurations. Figure 7(b) shows the percentage of combined real peak of pure CPU and pure GPU configurations achieved by the hybrid versions of HPL. We see that "OSU-HYBRID" achieves more than 80% of the peak in both "1G-Config" and "2G-Config" configurations. The loss in performance compared to the peak is primarily due to the imbalance in memory and compute power on Oakley nodes. We take advantage of the higher compute power of GPU nodes by scheduling more MPI processes on them while scheduling only one MPI process per CPU node. Since both CPU and GPU nodes on Oakley have the same amount of memory and data distribution among MPI processes is uniform, the processes running on the CPU node cannot take advantage of all the memory available. This limits the peak performance achieved on the CPU nodes, when running in the hybrid manner. On clusters which have

memory resources proportionate to the compute power of GPU nodes, this performance loss can be avoided using our design.

Table IV reports the peak performance numbers using our hybrid HPL on Cluster A and Oakley. We define peak performance efficiency (PPE) and theoretical performance efficiency (TPE) as follows: PPE is the hybrid performance (R_{peak-h}) as a percentage of combined real peak achieved using pure CPU node (R_{peak-c}) and pure GPU node (R_{peak-g}) configurations while TPE is the hybrid performance as a percentage of combined theoretical peak of pure CPU node and GPU nodes configurations.

$$PPE = H_Peak/(CPU_Peak + GPU_Peak) \quad (4)$$

$$TPE = H_Peak/(CPU_t_Peak + GPU_t_Peak) \quad (5)$$

We see that our hybrid version can achieve up to 86.3% PPE for the largest run on Oakley. It achieves over 50% TPE. Our version delivers up to 47% improvement in the performance compared to that reported using a pure CPU configuration (160 CPU nodes). Note that when computing the PPE, the compute power of multi-core processors of the GPU nodes has to be deducted to factor out the overlap.

Config	R_{peak-c}	R_{peak-g}	R_{peak-h}	PPE(%)	TPE (%)
1G-Config-A	2479	2911	3888	80.7%	52.8%
1G-Config-Oakley	18428	11722	22040	83.2%	59.7%
2G-Config-Oakley	18428	16670	27110	86.3%	50.8%

Table IV
PEAK PERFORMANCE ACHIEVED USING OUR HYBRID HPL ON CLUSTER A(8G+32C) AND OAKLEY(32G+128C). PERFORMANCE IS IN GFLOPS

D. Performance Scalability

In Figure 8, we show the strong and weak scaling of our hybrid HPL on Cluster A. For strong scaling, we fix the HPL problem size to 80,000 and 110,000 for 4 and 8 GPUs, respectively, while varying the number of CPU nodes used. In the experiments for weak scalability, we keep the memory usage of GPUs around 80%. We keep the number of GPU nodes constant at 4 and 8 while we increase the number of CPU nodes. This is to show the scalability while our code takes advantage of abundant CPU resources available on a cluster. Clearly, both curves achieve sustained scalability.

Figure 9 shows the scalability of PPE achieved by our code with different heterogeneous node configurations on Oakley. PPE is calculated as mentioned above. In Figure 9(a), we keep the number of CPU nodes constant while we increase the number of GPU nodes. In Figure 9(b), we keep the number of GPU nodes constant while we increase the number of CPU nodes. In Figure 9(c), we scale the number of CPU and GPU nodes proportionally. We see constant PPE as we vary the GPU nodes only or when we vary CPU and GPU nodes proportionately. We see a drop when the number of CPU nodes are increased while keeping number of GPU nodes constant. This is due the loss of efficiency on CPU nodes due to memory-compute imbalance as explained in Section IV-C. We still get above 80% efficiency for both configurations.

V. CONCLUSION AND FUTURE WORK

This paper proposes a novel approach to enable the HPL benchmark to efficiently utilize all computing resources on heterogeneous CPU-GPU clusters. With a two-level workload partitioning strategy and process grid reordering, our design achieves above 80% of the combined real peak performance of pure CPU and pure GPU node configurations. The test results exhibit sustained scalability, and experiments across two platforms with three configurations validate the portability of our approach. In future work, we will explore our design on heterogeneous clusters with Intel MIC.

REFERENCES

[1] Top500 Site, http://www.top500.org.

[2] J. J. Dongarra, L. S. Duff, D. C. Sorensen, and H. A. V. Vorst, *Numerical Linear Algebra for High Performance Computers*. Society for Industrial and Applied Mathematics, 1998.

[3] M. Fatica, "Accelerating Linpack with CUDA on Heterogenous Clusters," in *Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, 2009.

[4] T. Endo, A. Nukada, S. Matsuoka, and N. Maruyama, "Linpack evaluation on a supercomputer with heterogeneous accelerators," in *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

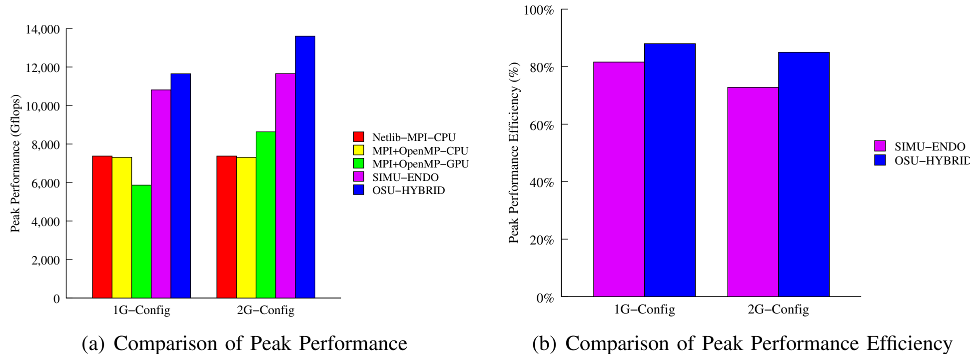
[5] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, J. Langou, H. Ltaief, and S. Tomov, "LU Factorization for Accelerator-based systems," in *Proceedings of the 9th International Conference on Computer Systems and Applications (AICCSA)*, 2011.

[6] J. Kurzak, P. Luszczek, M. Faverge, and J. Dongarra, "LU Factorization with Partial Pivoting for a Multicore System with Accelerators," *IEEE Transactions on Parallel and Distributed Systems*, vol. 99, no. PrePrints, p. 1, 2012.

[7] M. Bach, M. Kretz, V. Lindenstruth, and D. Rohr, "Optimized HPL for AMD GPU and multi-core CPU usage," *Comput. Sci.*, vol. 26, no. 3-4, pp. 153–164, Jun. 2011.

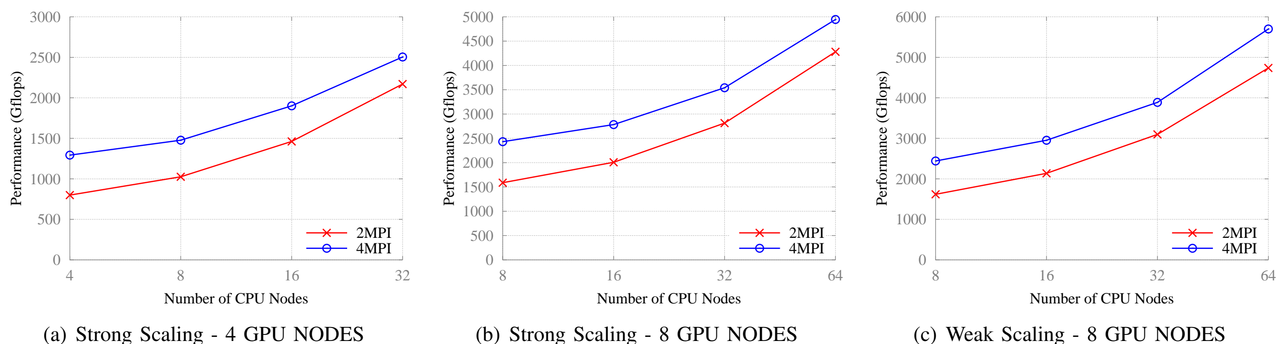
[8] T. Gautier, F. Lementec, V. Faucher, and B. Raffin, "X-Kaapi: A Multi Paradigm Runtime for Multicore Architectures," INRIA, Rapport de recherche RR-8058, Feb. 2012.

[9] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.



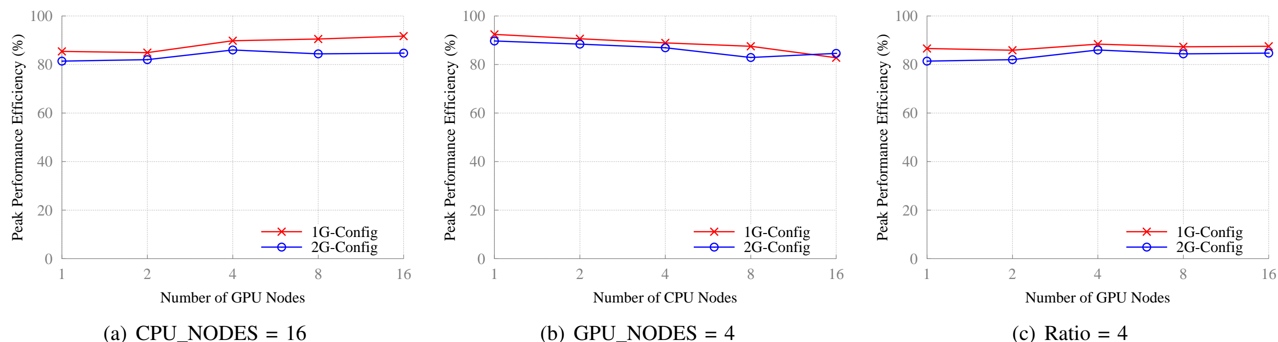
(a) Comparison of Peak Performance (b) Comparison of Peak Performance Efficiency

Figure 7. Comparison of peak performance and efficiency across different HPL versions on 16 GPU nodes and 64 CPU nodes



(a) Strong Scaling - 4 GPU NODES (b) Strong Scaling - 8 GPU NODES (c) Weak Scaling - 8 GPU NODES

Figure 8. Strong Scalability and Weak Scalability on Cluster A with 2 and 4 MPI processes launched on each GPU node



(a) CPU_NODES = 16 (b) GPU_NODES = 4 (c) Ratio = 4

Figure 9. Performance Efficiency Scalability on Oakley

[10] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta, “Productive Cluster Programming with OmpSs,” in *Proceedings of the 17th international conference on Parallel processing (Euro-Par)*, 2011.

[11] T. R. W. Scogland, B. Rountree, W.-C. Feng, and B. R. de Supinski, “Heterogeneous Task Scheduling for Accelerated OpenMP,” in *Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS)*, 2012.

[12] C. Augonnet, O. Aumage, N. Furmento, R. Namyst, and S. Thibault, “StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators,” in *Proceedings of the 19th European conference on Recent Advances in the Message Passing Interface (EuroMPI)*, 2012.

[13] F. Song, S. Tomov, and J. Dongarra, “Enabling and Scaling Matrix Computations on Heterogeneous Multi-core and Multi-GPU Systems,” in *Proceedings of the 26th ACM international conference on Supercomputing (ICS)*, 2012.

[14] F. Song and J. Dongarra, “A Scalable Framework for Heterogeneous GPU-based Clusters,” in *Proceedings of the 24th ACM symposium on Parallelism in algorithms and architectures (SPAA)*, 2012.

[15] Netlib FAQ, http://www.netlib.org/benchmark/hpl/faqs.html.