2013 IEEE 27th International Symposium on Parallel & Distributed Processing

Design and Implementation of the Linpack Benchmark for Single and Multi-Node Systems Based on Intel® Xeon Phi™ Coprocessor

Alexander Heinecke[*], Karthikeyan Vaidyanathan[†], Mikhail Smelyanskiy[‡], Alexander Kobotov[§], Roman Dubtsov[§], Greg Henry[¶], Aniruddha G Shet [‖], George Chrysos[‖], Pradeep Dubey[‡]

[*]*Department of Informatics, Technische Universität München, Munich, Germany*
[†]*Parallel Computing Lab, Intel Corporation, Bangalore, India*
[‡]*Parallel Computing Lab, Intel Corporation, Santa Clara, USA*
[§]*Software and Service Group, Intel Corporation, Novosibirsk, Russia*
[¶]*Software and Service Group, Intel Corporation, Hillsboro, USA*
[‖]*Intel Architecture Group, Intel Corporation, Hillsboro, USA*

***Abstract*—Dense linear algebra has been traditionally used to evaluate the performance and efficiency of new architectures. This trend has continued for the past half decade with the advent of multi-core processors and hardware accelerators.**

In this paper we describe how several flavors of the Linpack benchmark are accelerated on Intel's recently released Intel® Xeon Phi™[1] co-processor (code-named Knights Corner) in both native and hybrid configurations. Our native DGEMM implementation takes full advantage of Knights Corner's salient architectural features and successfully utilizes close to 90% of its peak compute capability. Our native Linpack implementation running entirely on Knights Corner employs novel dynamic scheduling and achieves close to 80% efficiency — the highest published co-processor efficiency. Similarly to native, our single-node hybrid implementation of Linpack also achieves nearly 80% efficiency. Using dynamic scheduling and an enhanced look-ahead scheme, this implementation scales well to a 100-node cluster, on which it achieves over 76% efficiency while delivering the total performance of 107 TFLOPS.

***Keywords*-HPL; SIMD; TLP; LU factorization; panel factorization; hybrid parallelization; Xeon Phi**

I. INTRODUCTION

A. Introduction

The current fastest computer in the TOP500 is the Titan system which delivered 17.59 PFLOPS on the Linpack benchmark in November 2012 [19]. The HPC community expects to deploy the first 100 PFLOPS machine by 2014, paving the way for the first ExaFLOP system at the end of the decade [13]. To fuel such growth in computational power, the current trend is to couple commodity processors with various types of computational accelerators, which offers dramatic increases in both compute density and energy efficiency.

In this paper, we describe the implementations and tuning of several flavors of the Linpack benchmark for Intel's recently announced Intel® Xeon Phi™ coprocessor "Knights Corner". Achieving high Linpack performance on modern hybrid many-core system requires careful tuning of BLAS sub-routines, hiding communication latency and balancing the load across devices of variable processing capabilities. To this end, this paper makes the following contributions:

- We propose a Knights Corner-friendly matrix format which enables our DGEMM kernel to achieve 89.4% efficiency, which corresponds to 944 GFLOPS of performance.
- We extend the dynamic scheduling technique proposed in Buttari et al. [4] to scale to the large number of cores on Knights Corner. Using this extension as well as highly optimized panel factorization, our native Linpack implementation achieves 78.8% efficiency – the highest published coprocessor efficiency running Linpack directly on the card.
- We enhance advanced offload DGEMM as well as the advanced look-ahead scheme proposed by Bach et al. [3] to improve performance and efficiency of our hybrid Linpack implementation. With the help of this scheme, our hybrid implementation running on both Intel® Xeon® Processor E5-2670 (formerly code-named Sandy Bridge) and Knights Corner achieves 79.8% efficiency on a single node, and scales up to 107 TFLOPS on a 100-node cluster, which corresponds to 76.1% efficiency.

The rest of the paper is organized as follows. Section II introduces the hardware architectures used in this study: the Intel® Xeon® Processor E5-2670 and the Intel® Xeon Phi™ coprocessor. Section III describes our optimized implementation, results and analysis of DGEMM. In Section IV, we present implementation details and performance of the native Linpack, which runs entirely on Knights Corner. We describe our hybrid Linpack implementation and its performance in Section V. Our methods and results

[1]Intel, Xeon, and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.

	Intel® Xeon® E5-2670	Intel® Xeon Phi™ co-processor
Sockets × Cores × SMT	2 × 8 × 2	1 × 61 × 4
Clock (GHz)	2.6	1.1
Single Precision GFLOPS	666	2148
Double Precision GFLOPS	333	1074
L1 / L2 / L3 Cache (KB)	32 / 256 / 20,480	32 / 512 / -
DRAM	128 GB	8 GB GDDR
STREAM Bandwidth [14]	76 GB/s	150 GB/s
PCIe Bandwidth	6 GB/s	
Compiler Version	Intel® v13.0.1.117	
MPI Version	Intel® v4.1.0.027	

Table I
SYSTEM CONFIGURATIONS OF SANDY BRIDGE EP AND KNIGHTS CORNER. NOTE L1 AND L2 CACHE SIZES ARE LISTED PER CORE.

are compared with previous work in Section VI, and we conclude in Section VII.

II. HARDWARE PLATFORMS

Our experimental test-bed consists of a dual-socket Intel® Xeon® E5-2670 and Intel® Xeon Phi™ coprocessor described next.

A. Hardware Platforms

Intel® Xeon® E5-2670 "Sandy Bridge EP": This is an x86-based multi-core server architecture featuring a superscalar, out-of-order micro-architecture supporting 2-way hyper threading. In addition to scalar units, it has a 256 bit-wide SIMD unit that executes the AVX instruction set. Separate multiply and add ports allow for the execution of one multiply and one addition instruction (each 4-wide in double-precision, or 8-wide in single-precision) in a single cycle.

Intel® Xeon Phi™ coprocessor "Knights Corner": This architecture features many in-order cores [2] on a single die; each core has 4-way hyper-threading support to help hide memory and multi-cycle instruction latency. To maximize area and power efficiency, these cores are less aggressive, i.e., they have lower single-threaded instruction throughput than Sandy Bridge EP cores and run at a lower frequency. However, each core has 32 vector registers, 512 bits wide, and its vector unit executes 8-wide double-precision SIMD instructions in a single clock. Each core further has two levels of cache: a single-cycle access 32 KB first level data cache (L1) and a larger 512 KB second level cache (L2), which is globally coherent via directory-based MESI coherence protocol.

Knights Corner is physically mounted on a PCIe slot and has dedicated GDDR memory. Communication between the host CPU and Knights Corner is therefore done explicitly through message passing. However, unlike many other co-processors, it runs a complete Linux-based operating system, with full paging and virtual memory support, and features a shared memory model across all threads and hardware cache coherence. Thus, in addition to common programming models for co-processors, such as OpenCL, Knights Corner supports more traditional multiprocessor programming models such as pthreads and OpenMP. Additional details can be found in Table I.

Knights Corner has a rich ISA that supports many flavors of scalar and vector operations, including fused multiply-add. Most vector operations can take one of their operands from memory; this reduces instruction footprint. In addition, Knights Corner has a dual-issue pipeline which allows prefetches and scalar instructions to co-issue with vector operations in the same cycle. Such a feature removes these instructions from the critical path, especially in loops with limited unrolling, and is particularly useful in DGEMM operation.

Examples of several flavors of Knights Corner instructions used in our DGEMM implementation (see Section III) are demonstrated in Figure 1a and Figure 1b. Figure 1a shows fused vector multiply-add operations with the second operand broadcast from memory. There are two types of broadcasts; the *1to8* broadcast takes a single double-precision element and replicates it eight times, while the *4to8* broadcast, shown in the Figure 1a, replicates four double-precision elements twice. In addition, a register operand can be swizzled in-flight. $SWIZZLE_i$ replicates the ith element of the 4-element lane four times in each lane. Figure 1b shows example of $SWIZZLE_2$.

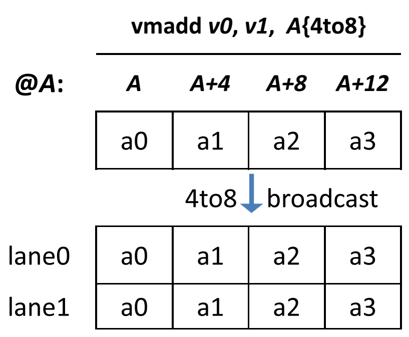
The L1 cache has two ports: one for read and the other for write. As a result, a vector instruction with one memory operand and a vector store can be co-issued in the same cycle. Knights Corner provides prefetches for both levels caches: L1 and L2. The high-level behavior of L1 prefetch issued for the line located in L2 cache but missing in L1 is shown in Figure 1c. As a cache line arrives from L2 cache, a victim cache line has to be evicted from L1 cache and a new line will be filled in. This operation requires both L1 ports. If in a given cycle, one of the ports is busy, which can happen, for example, when another instruction (e.g., vector multiply-add with a memory operand) is accessing L1, the fill cannot happen and is deferred until the next cycle. The cache port availability is checked every cycle. If the port becomes available, prefetch operation completes. If the port is not available after a certain number of threshold cycles, the core pipeline is stalled for a few cycles to let the prefetch operation complete.

III. NATIVE DGEMM

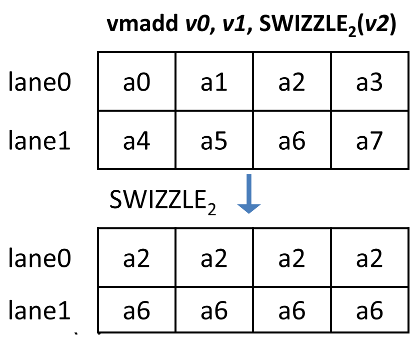
A. Implementation

In this section, we describe our design and implementation of the double-precision general matrix-matrix multiplication

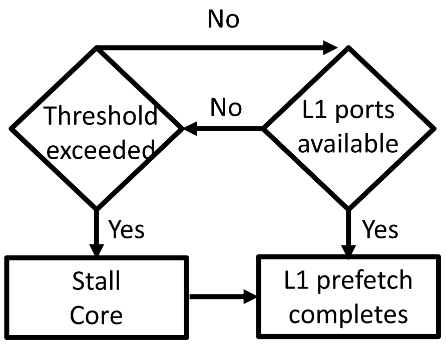
[2]Last core is always reserved by the operating system and is typically not used for computation. To understand inherent hardware efficiency, in the case of native DGEMM and native HPL (Section III-A and IV, respectively), we report efficiency with respect to peak performance of all but the last core. For offload DGEMM and hybrid HPL (Section V), we report efficiency with respect to all available cores



(a) 4to8 memory broadcast from @A into vector register $v0$.

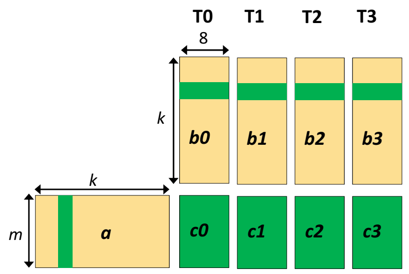


(b) Swizzling a second element of each 4-element lane within $v2$.



(c) L1 prefetch high level behavior.

Figure 1. Introduction to operations supported by Knights Corner.



(a) Shared memory parallelization using four hardware threads per core.

```
for i=0 to k do
  vload v31, &b[i][0];                vprefetch1 &b[i+1][0];
  vmadd v0, v31, &a[i][0]{1to8};
  ...
  vmadd v3, v31, &a[i][3]{1to8};      vprefetch1 &a[i+1][0];
  vmadd v4, v31, &a[i][4]{1to8};      vprefetch1 &a[i+1][8];
  vmadd v5, v31, &a[i][5]{1to8};      vprefetch1 &a[i+1][16];
  vmadd v6, v31, &a[i][6]{1to8};      vprefetch1 &a[i+1][24];
  ...
  vmadd v30, v31, x &a[i][30];
endfor
```

(b) **Basic Kernel 1**: multiplies column of a by row of b.

```
for i=0 to k do
  vbroadcast64x4 v30, &a[k][0];
  v31=vload &b[i][0];                   vprefetch1 &b[i+1][0];
  vmadd v4, v31, &a[i][4]{1to8};        vprefetch1 &a[i+1][0];
  vmadd v0, v31,  SWIZZLE_0(v30);       vprefetch1 &a[i+1][8];
  vmadd v5, v31, &a[i][5]{1to8};        vprefetch1 &a[i+1][16];
  vmadd v6, v31, &a[i][6]{1to8};        vprefetch1 &a[i+1][24];
  vmadd v1, v31, SWIZZLE_1(v30) ;
  vmadd v2, v31, SWIZZLE_2(v30);
  vmadd v3, v31, SWIZZLE_3(v30);
  ...
  vmadd v29, v31, &a[i][29];
endfor
```

(c) **Basic Kernel 2**: relieves L1 pressure at the expense of broadcast.

Figure 2. Basic DGEMM kernel on Knights Corner.

(DGEMM) kernel, $C = \alpha AB + \beta C$, where A, B, and C are $M \times K$, $K \times N$, and $M \times N$ matrices, respectively, while α and β are scalars. DGEMM is part of the "Basic Linear Algebra System" (BLAS), which is a common interface for matrix/vector operations [1]. Our DGEMM kernel assumes that all three matrices are in row-major format [3]. Our implementation breaks this general DGEMM kernel into a sequence of outer products (aka rank-k updates): $C = \alpha \sum_{i=0}^{K/k} A_i B_i + \beta C$. Here A_i is $M \times k$ column block sub-matrix of A, while B_i is $k \times N$ row block sub-matrix of B. In the rest, we describe our optimized implementation of this outer product. While our focus is on DGEMM, we apply the same optimizations to SGEMM as well.

1) Cache Blocking: Most optimized DGEMM implementations block the matrices to fit into one or more levels of caches on a given architecture. The objective is to reduce the bandwidth requirements to be under the limit the architecture can deliver. Our implementation blocks matrices in each core's private 512 KB L2 cache, as described next.

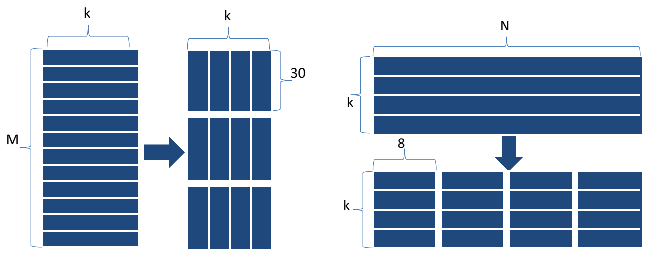
We conservatively require all three matrix blocks, *Ab*, *Bb*, and *Cb* of dimensions, $m \times k$, $k \times n$ and $m \times n$, respectively, to fit into L2 cache. which results in the following inequality: 8 bytes $\cdot\, (m \cdot n + m \cdot k + k \cdot n) <$ 512KB. Note that better approximations exist [21], but the one above is sufficient for our purposes. To compute $m \times n$ block *Cb* each cores requires a minimum of $m{\cdot}n{\cdot}k/(8\ \mathit{vmadds/cycle})$ cycles, and fetches 8 bytes$\cdot(2{\cdot}m{\cdot}n{+}m{\cdot}k{+}k{\cdot}n)$ bytes of data from main memory to bring in all three blocks into L2 cache. The factor 2 in front of $m{\cdot}n$ is due to the fact that $m{\times}n$ output *Cb* block is both read and written. The required memory bandwidth is the ratio between the memory traffic and compute time and is equal to $64 \cdot (2/k + 1/n + 1/m)$ bytes/cycles per core. This bound is conservative, as it accounts for no cache sharing among cores. If this sharing is taken into account, further bandwidth reductions are possible. Similar to Goto et al. [10], we choose m and k such that $m \times k$ block *Ab* occupies the largest fraction of L2 cache, while leaving some room for *Bb* and *Cb* blocks. In practice, there are additional considerations that factor into choosing a value of k, as shown later. For large values of N, the overhead of bringing *Ab* into L2 cache is amortized. As a result, required memory bandwidth does not depend on n and can be approximated as $64 \cdot (2/k + 1/m)$ bytes/cycles per core. For example, choosing m=120, n=32 and k=240, results in 1.1 bytes/cycle of bandwidth per core, or 74 GB/s on our system with 60 cores at 1.1GHz. This is well within the limits of Knights Corner's achievable STREAM bandwidth of 150 GB/s (see

[3]Column-major (CM) DGEMM is easily derived from row-major (RM) DGEMM by transposing both sides of the equality $C(CM) = A(CM) \cdot B(CM)$, to get $C(RM) = B(RM) \cdot A(RM)$

Table I).

2) Basic Matrix-Matrix Multiply Kernels: To get the highest efficiency, our basic matrix-matrix multiply kernel is hand-coded in assembly language. Figure 2a shows the matrix decomposition used by the kernel. Specifically, each of four hardware threads on a given core multiplies $31 \times k$ matrix a by $k \times 8$ matrix b and stores the result into $m \times 8$ matrix c. Note that a, stored in column-major format, and b, stored in row major formats, are the packed tiles of the original matrices. Packing is described in Section III-A3. a is shared between four threads, while each thread accesses its own b and c. Sharing a between four threads provides reuse in L1 cache, since a line of a accessed by one of the threads is likely to remain in $L1$ for the other three threads, as long as all threads are synchronized. To ensure this, we enforce frequent fast inter-thread synchronization, which keeps all four hardware threads coherent. Figure 2b shows the main loop of the basic kernel (**Basic Kernel 1**), which iterates k times. Each iteration multiplies a 31-element column of a, by an 8-element column of b and stores the result into 31 intermediate registers $v0 - v30$. Specifically, we first load a row of b into a temporary register $v31$, followed by 31 vector multiply-adds of $v31$ with corresponding element of a, which is *1to8* broadcast from memory. When the loop completes, we update 31 rows of the original matrix C (not shown) with the values stored in the registers $v0 - v30$. Recall that Knights Corner has 32 vector registers (see Section II). Blocking 31 rows of c rows in all but one registers, amortizes overhead of loading a row of b. As the result, this kernel has the theoretical efficiency of $96.9\%(= 31/32)$, because there are 32 vector instructions in the loop iteration, out which 31 are vector multiply-adds. A small overhead comes from updating C with the values in $v0 - v30$. This overhead gets amortized and decreases linearly with k. For example, for $k = 240$ it is less than 0.5%.

Figure 2b also shows the L1 prefetches required to hide L1 cache miss latency. We also insert L2 prefetches, not shown, to bring the data into the core's local L2. Since local L2 cache hit latency is under 25 cycles, we prefetch for the next iteration of the loop, $i + 1$. In our case, since k is large (to both reduce memory bandwidth and to reduce overhead of updating C, as described earlier), neither a nor b fits into L1 and thus are streamed from L2. Note that each thread accesses five cache lines per loop iteration: one line to access 8-element row of b and four lines to access 31-element column of a. Since a is shared among four threads, the four lines are only brought in only once from L2 into L1 by one of the threads. Therefore, on average, each iteration of the kernel requires two cache lines to be brought from L2 into L1. Since each vector operation in the inner loop of Figure 2b accesses memory every cycle, two L1 prefetches issued for these two lines will cause L1 port conflict and result in core stalls, as described in Section II. As few as two stall cycles in the tight inner-loop will reduce overall efficiency down to $91\% = (31/(32 + 2$ cycles).



(a) packing A_i into $30 \times k$ tiles: each tile is in column-major format.
(b) packing B_i into $k \times 8$ tiles: each tile is in row-major format.

Figure 3. Packing A_i and B_i matrices into tiles.

To address this challenge, we propose an enhanced **Basic Kernel 2**, shown in Figure 2c. Namely, we add broadcast instruction, which load-broadcasts first four elements of a into $v30$ (see Figure 1a). Furthermore, instead of broadcasting elements 0, 1, 2 and 3 of column of a from memory, as was done in **Basic Kernel 1**, these elements are swizzled out of the vector register $v30$, as shown by four highlighted vmadd instructions. As these instructions do not access memory, this creates four "holes" for every thread in the pipeline, during which L1 ports can be exclusively accessed by L1 prefetch. Given that each thread only brings on an average two cache lines from L1, as explained earlier, four "holes" are sufficient to significantly reduce core stalls due to port conflicts. With this approach, the peak theoretical efficiency of **Basic Kernel 2** is $93.7\%(= 30/32)$, because there are 32 vector instructions in the loop, out which 30 are vector multiply-adds, compared to 31 vector multiply-adds in **Basic Kernel 1**. While the broadcast instruction, which uses one extra register ($v30$), reduces the number of vector multiply-adds in the inner loop, it also enables conflict-free access to L1 and as a result improves overall efficiency.

3) Packing into Knights Corner-friendly Data Layout: Multiplying matrices stored in row or column-major format may result in performance degradation, due to TLB pressure and cache associativity conflicts, especially when these matrices have large leading dimensions [10]. As a result, most DGEMM implementations pack matrices into a special tiled format, best suited for a given architecture [12]. Small leading dimensions of the tiles mitigates the above problems. If well optimized, packing has small overhead because its quadratic complexity (w.r.t. matrix dimension) is amortized by the computation that has cubic complexity. Thus prior to performing an outer product, we pack both A_i and B_i matrices into a novel Knights Corner-friendly format, using a temporary storage. Specifically, as shown in Figure 3a, A_i is packed into block row-major format that consists of $30 \times k$ tiles; each tile is stored in column-major. The latter allows contiguous access to each column of a in the basic kernel as discussed earlier, and simplifies address calculation for prefetching, as discussed in Section III-A2(a) and (b). Matrix B_i is also packed into block row-major format that consists of $8 \times k$ tiles, where each tile is stored row-major, as shown in Figure 3b. Our packing routines are highly optimized and achieve bandwidth-bound performance for medium to large size matrices.

4) Other Optimizations: Other optimizations, such as tuning of L2 prefetch distance, fast inter-thread synchronization and parallelization are similar to how they are done by Deischer et al [5].

Finally, while our design and implementation of DGEMM kernel is based on [10], there are number of differences. First, we transpose packed tiles of A_i to spread out prefetches more uniformly. Second, we choose tile dimensions based on Knights Corner's specific architectural and micro-architectural considerations, such as register file size, cache sizes and bandwidth between different levels of memory hierarchy.

B. Performance Results

k	120	180	240	300	340	400
SGEMM						
Efficiency	88.3	89.3	90.1	90.4	90.6	90.8
Performance	1866	1886	1902	1910	1914	1917
DGEMM						
Efficiency	86.7	88.6	89.1	89.4	89.3	88.9
Performance	915	935	941	944	943	943

Table II
SGEMM AND DGEMM PERFORMANCE AND EFFICIENCY AS FUNCTION OF k FOR $N = M = 28,000$.

Table II shows performance and efficiency for both DGEMM and SGEMM, as we vary k from 120 to 400, while M and N remain fixed at $28,000$. We see that the efficiency improves as k increases for both kernels. This is due to the fact that the basic kernel overhead of updating c decreases as k increases. However, as k becomes larger the improvements diminish. There is even a slight degradation in performance in case of DGEMM for $k = 340$ and $k = 400$. This is due the fact that as k increases, L2 block sizes also increase and eventually falls out of $L2$ cache. Overall, in case of SGEMM the best efficiency of 90.8% (1,917 TFLOPS) is achieved for $k = 400$, while in the case of DGEMM the best efficiency of 89.4% (944 GFLOPS) is achieved for $k = 300$. We see that DGEMM is 4% below its projected efficiency of 93.7%. The 4% loss is due to the three overheads unaccounted by the performance projection: (i) updating c tile, (ii) packing matrix into the tiles (see Figure 4), and (iii) scalar instructions overhead required to drive DGEMM parallel distribution of work.

Figure 4 shows DGEMM performance comparison on Sandy Bridge EP using Intel MKL 11.0 BLAS [2] and Knights Corner using our implementation for a range of matrix sizes. As bottom curve shows, Sandy Bridge EP

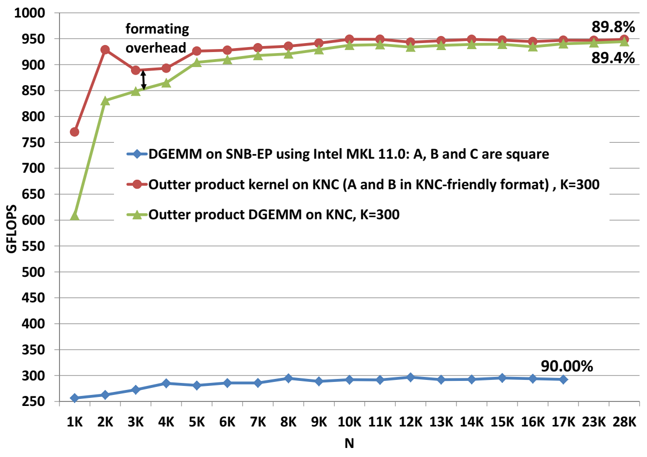


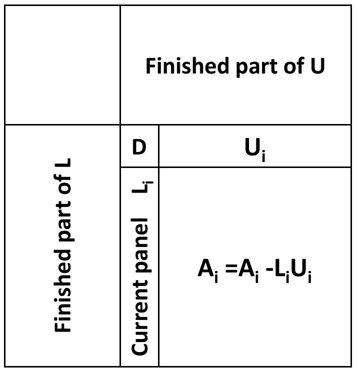
Figure 4. Native DGEMM performance comparison on Sandy Bridge EP and Knights Corner for different problem sizes.

achieves up to 90% efficiency. For Knights Corner we show several results. The middle curve shows performance of the outer product kernel for k=300 using Knights Corner-friendly format – no overhead of packing is included. This is the key kernel in our hybrid Linpack implementation, which performs data packing on CPU, as discussed in Section V. As shown in Table II, k=300 results in the best DGEMM efficiency. We see that kernel performance is high even for sizes as small as 5K for which it reaches 88% efficiency. The top curve shows performance of the DGEMM which executes the same outer-product, but includes overhead of packing the input matrices into our Knights Corner-friendly format. We see that this overhead decreases from 15% for 1K matrices down to less than 0.4% for matrices larger than 17K The packing overhead is under 2% starting from 5K matrices.s Note also that the performance of square matrices (not shown) is similar to outer-product DGEMM. This is expected – as described in the beginning of this section, square DGEMM is composed of the sequence of outer-products.

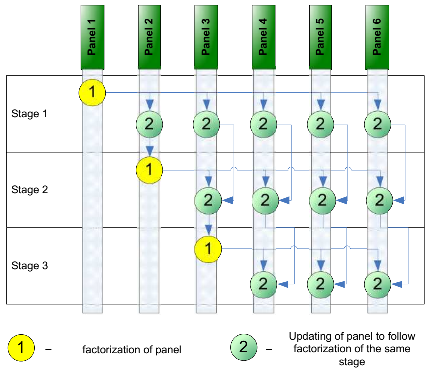
IV. Native Linpack

In this section, we describe design and implementation of native Linpack, which runs entirely on Knights Corner. Native Linpack spends majority of its execution time in LU factorization.

The LU factorization algorithm decomposes a matrix A into a product of a lower-triangular matrix L and a unit upper triangular matrix U. The blocked LU formulation is shown schematically in Figure 5a. In this algorithm, the lower triangular part of the input matrix is overwritten with L and the upper triangular part with U. The algorithm proceeds from left to right in block steps (aka stages) until the entire matrix is factorized. At each stage i, a portion of column panel of L, $[DL_i]$, is first factored, a block of matrix rows are swapped based on the pivot vector, produced by



(a) Structure of LU factorization.


Panel 1
Panel 2
Panel 3
Panel 4
Panel 5
Panel 6
Stage 1
Stage 2
Stage 3
1 – factorization of panel
2 – Updating of panel to follow factorization of the same stage

(b) Directed acyclic graph (DAG) of dependencies within LU factorization.

```
init DAG
while (DAG.NotFinished()) in parallel
   critical section
      Task = DAG.AvailableTask()
   end critical
   case( Task )
      Task1: dgetrf(Task.panel, Task.stage);
      Task2: {dlaswp;
                dtrsm;
                dgemm}(Task.panel,Task.stage);
   DAG.Commit( Task )
```

(c) Dynamic load balancing within LU factorization.

Figure 5. Knights Corner native LU Factorization in ith stage.

panel factorization, and a portion of row panel of U, U_i is updated using a forward solver. The trailing sub-matrix A_i is then updated with the matrix-matrix product of L_i and U_i. Panel factorization and trailing matrix update are the two most critical LU kernels. When the current stage completes, next stage performs the same sequence of operations on updated sub-matrix A_i. After the last stage, original matrix is factorized, and the solution of $Ax = b$ is obtained by forward and back substitutions using L and U factors. A more detailed treatment of the algorithmic issues can be found in Golub and Van Loan [9].

Our implementation is based on dynamic scheduling scheme of a Data Acyclic Graph (DAG), originally proposed by Buttari et al [4] for a multi-core architecture with a small number of cores. We extend this idea to scale on an architecture with large number of cores, such as Knights Corner.

A. Implementation

Figure 5b demonstrates DAG-based approach for a matrix divided into six panels. Each node represents computational task which involves a panels. There are two categories of tasks: panel factorization Task1 and a composite task Task2, which is comprised of pivoting, forward solve and trailing update. The edges enforce dependencies between these tasks. For example, tasks within the j-th row cannot be performed until the respective factorization of the j-th panel is completed. However all Task2 tasks within a row can be executed in parallel. Finally, the rows of the DAG represent stages of computation, as described in the beginning of this section.

To reduce storage requirements of the required DAG, we represent it as one dimensional array of the length equal to the number of panels. Each element of the array stores the current stage of the panel, among other information. As a task completes, its stage is incremented. Threads access and update the DAG array dynamically in order to keep track of the stage of the computation, exploit available parallelism and satisfy the dependencies. The parallel code is shown in Figure 5c. Task distribution is encapsulated in the DAG object. To obtain a new task, each thread calls DAG.AvailableTask() atomically. This function searches the DAG for a new task from the current stage, and only proceeds to the next stage when all current stage tasks are finished or in progress by other threads. The only exception is panel factorization (Task1) from the next stage: this task is immediately performed when the corresponding panel is updated in the current stage by Task2. To see if this dependency is satisfied, DAG.AvailableTask() checks the stage number of the corresponding panel. This operation is known as look-ahead [17], which effectively allows tasks from the previous stage to be overlapped with panel factorization from the next stage. Likewise, Task2 from stage i can only start when the corresponding panel factorization in the same stage is completed: its stage number is greater or equal to i. When task completes, a commit function increments its stage. This increment does not require critical section, because it is always performed by the same thread which completed the task.

In addition to compact storage representation of the DAG, this dynamic scheduling of LU factorization tasks has several advantages: (i) it avoids global barrier synchronization between consecutive stages, (ii) it improves resource utilization: threads that finish one task can immediately switch to other available tasks.

In its original implementation, threads (from one or more cores) are partitioned into groups, such that each group works on the same task. The assignment is fixed. This scheme has two disadvantages. The first disadvantage is potential contention over the critical section; while the overhead of such contention may be acceptable for small number of cores, it limits scalability on many-core architectures, such as Knights Corner. To address this problem, only a single "master" thread within a group accesses the critical section to obtain a new task, while the remaining threads

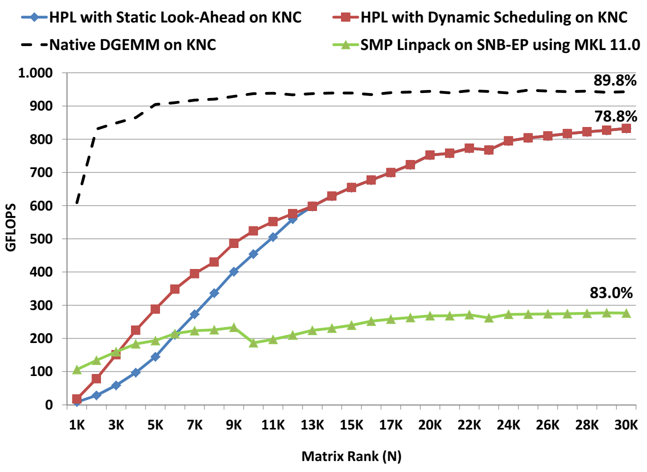
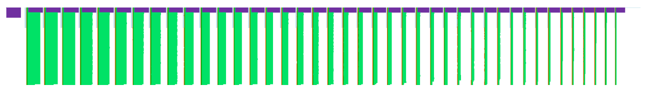
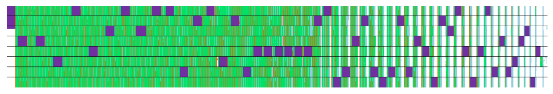


Figure 6. Native Linpack performance comparison on Sandy Bridge EP and Knights Corner for different problem sizes.



(a) static look-ahead



(b) dynamic scheduling

Figure 7. Gantt chart of LU execution profile for 5K problem. Light blue: DLASWP, orange: DTRSM, violet: DGETRF, green: DGEMM, white: barrier, black lines separate thread groups.

wait on the local group barrier for the “master” thread to return with a new task, at which point the entire group starts computing the task. Allowing only the “master” threads to access the critical section, significantly reduces contention.

The second disadvantage is that using static thread partitioning for all stages creates load imbalance and exposes panel factorization overhead. For example, while using four threads in a group may be sufficient to hide panel factorization during early stages dominated by large trailing matrix updates, later stages which work on smaller matrices require more threads to hide the panel. To address this load imbalance, we extend the original approach by breaking LU factorization into **super-stages**. Each super-stage consists of some number of stages. Within each super-stage, the thread grouping is fixed, so that each group has enough threads to hide panel factorization overhead in each stage within the super-stage. After a super-stage is complete, and before factorizing a smaller matrix, we perform a global barrier synchronization, followed by thread re-grouping, which increases the number of threads assigned to a panel to speed up panel factorization. While this approach requires global barrier synchronization, the barrier is executed infrequently, at the end of the super-stage. This amortizes its overhead.

B. *Performance Results*

Figure 6 shows a comparison of Linpack performance between Sandy Bridge EP using Intel MKL 11.0 SMP Linpack [2] and Knights Corner using the above implementation, as we vary problem size from 1K to 30K, which is the largest problem that fits into 8 GB memory on Knights Corner. We see that for this problem, Sandy Bridge EP achieves 277 GFLOPS which corresponds to 83% efficiency. This is within 7% from its native DGEMM performance (Figure 4). For Knights Corner we show two curves: one with static look-ahead scheme and the other with dynamic scheduling. The static look-ahead implementation uses global barrier synchronization between stages [5]: at each stage it assigns the minimum required number of threads to each panel factorization to achieve the best load-balance with trailing update. We see that up to 8K, dynamic scheduling outperforms static look-ahead scheme. This is expected. As Figure 7a shows, for smaller problem sizes significant amount of time is spent in panel factorization and global barrier synchronization. As Figure 7b shows, dynamic scheduling significantly reduces the time spent in both of these regions and as the result performs better overall. However, as the problem size increases, the fraction of time spent in panel factorization and global barrier synchronization becomes smaller and the performance of the static look-ahead scheme approaches that of the dynamic scheduling scheme. For the 30K problem, both schemes achieve 832 GFLOPS, which corresponds to ≈79% efficiency. This is within 12% of the native DGEMM efficiency, shown by the upper curve and within 5% from Sandy Bridge EP efficiency as shown by the lower curve.

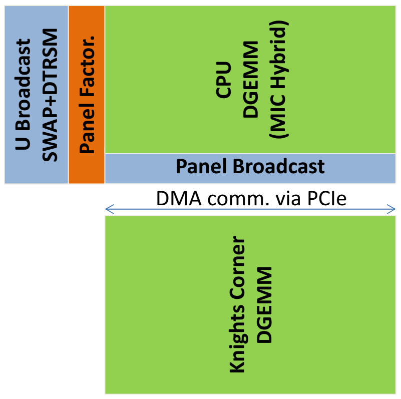
V. Hybrid Linpack

While our native Linpack implementation on Knights Corner achieves nearly 79% efficiency, this implementation limits the problem size to the 8 GB of memory available on Knights Corner. In this section, we describe our hybrid Linpack implementation, which runs on one or more hybrid nodes, where each node consists of a Sandy Bridge EP host and one or two Knights Corner co-processors. In contrast to native, our hybrid implementation takes full advantage of the large memory capacity, available on the host.

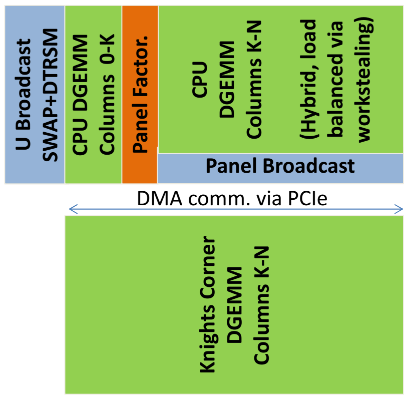
A. *Implementation*

Our implementation of Linpack for this hybrid system is based on the standard open-source implementation, High Performance Linpack (HPL) [15], originally designed for homogeneous clusters. Similar to Bach et al. [3], our implementation uses Knights Corner only to accelerate DGEMM, and uses Sandy Bridge EP to run remaining HPL tasks.

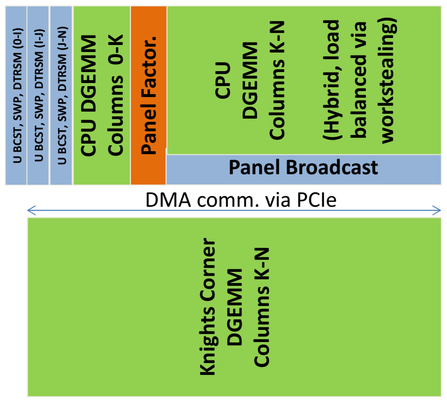
The key challenge in such a hybrid implementation, is to utilize Knights Corner as highly as possible. As Table I shows, two Knights Corner cards can deliver more roughly



(a) *No lookahead*: no overlap between DGEMM on Knights Corner and the rest of the kernels on Sandy Bridge EP.



(b) *Basic lookahead*: DGEMM on Knights Corner is overlapped with panel factorization on Sandy Bridge EP.



(c) *Pipelined lookahead*: overlaps DGEMM on Knights Corner and the rest of the kernels on Sandy Bridge EP.

Figure 8. Three implementations of hybrid HPL. DGEMM is offloaded to Knights Corner and performs panel factorization, swapping, DTRSM and U broadcast on Sandy Bridge EP.

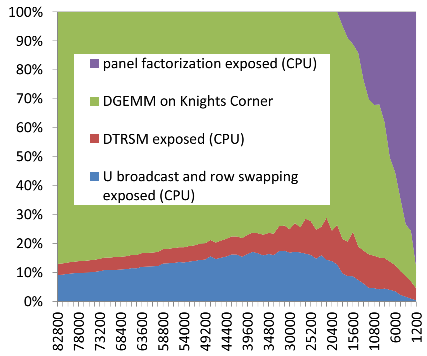
six times flops compared to Sandy Bridge EP. Therefore, when Knights Corner is idle, the system performs six times less work, compared to when Sandy Bridge EP is idle for same amount of time. Hence, a simple HPL extension in which Sandy Bridge EP simply offloads a larger portion of DGEMM to Knights Corner, while performing the panel factorization, swapping, DTRSM and U broadcast, will significantly expose Knights Corner idle time. This naive, *no look-ahead*, scheme is illustrated in Figure 8a.

A more advanced optimization extends a simple look-ahead scheme to the hybrid implementation. In our new scheme, the panel factorization from the next stage is done on Sandy Bridge EP and is overlapped with the current stage trailing update performed on Knights Corner. Specifically, as soon as DTRSM completes, Knights Corner starts executing offload DGEMM (see Section V-B) while Sandy Bridge EP updates (frees-up) the left-most panel and then proceeds to panel factorization. As soon as Sandy Bridge EP finishes the panel factorization, it immediately starts working on the rest of the trailing update together with Knights Corner. To improve load balance, this scheme is implemented using dynamic work-stealing (see Section V-B for further details). Overlapping panel factorization and trailing update in this fashion, reduces idle time on Knights Corner. This scheme, which we call *basic look-ahead*, is shown in Figure 8b.

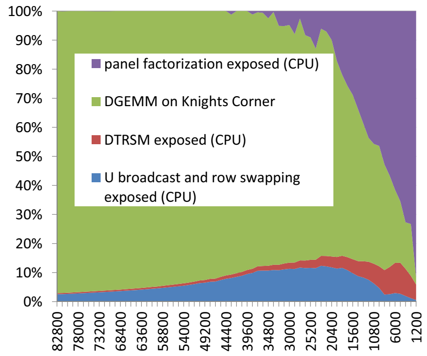
While overlapping panel in Sandy Bridge EP with DGEMM on Knights Corner reduces a portion of the Knights Corner's idle time, the coprocessor still remains idle during three steps: U broadcast, row swapping and DTRSM used to update the U panel. Swapping, constrained by both DRAM and interconnect bandwidth, exposes a larger fraction of Knights Corner's idle time compared to DTRSM, which is compute-bound. To address this challenge, we pipeline these three steps, as shown in Figure 8c. This scheme, which we call *pipelined look-ahead*, applies each of three steps to a subset of columns at a time, instead of all the columns. As soon as Sandy Bridge EP finishes the first subset of columns, Knights Corner immediately starts the trailing matrix update, which overlaps with the next set of columns on Sandy Bridge EP. By pipelining these three steps, *pipelined look-ahead* hides their overhead, further reducing Knights Corner's idle time, compared to the *basic look-ahead* scheme.

To illustrate the effect of pipelining, Figure 9 shows HPL execution time profile for $N = 84$K with and without pipelining for a 2×2 multi-node run. Execution time is broken into four regions. The bottom (green) region shows time during which Knights Corner executes DGEMM. The other three regions show exposed time during which Sandy Bridge EP executes the remaining kernels, while Knights Corner remains idle. We see that without pipelining (Figure 9a) Knights Corner is idle at least 13% of the time during which U broadcast, swapping and DTRSM are exposed. With pipelining (Figure 9b), Knights Corner idle time is reduced to less than 2.5%. However for the later stages, as the matrix becomes smaller, panel factorization gets exposed more than in the case without pipelining. This due to the fact that *pipelined look-ahead* breaks U broadcast, swapping and DTRSM into multiple steps which incur extra overhead that delays panel factorization. However, due to the fact that HPL spends most of its time in the earlier execution stages, this overhead is small and is further mitigated by our dynamic load-balancer described in Section V-B. This effect can be clearly seen in Figure 9c, as the swapping pipeline reduces the iteration time by up to 11% in the early and most time-consuming iterations.

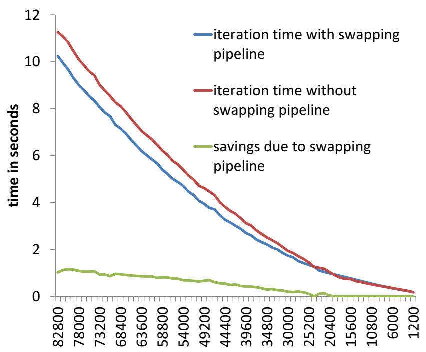
Note that the static overlap of swapping and DTRSM with DGEMM was originally proposed by Bach et al. [3] in the context of hybrid HPL running on AMD Cypress GPU and 24-core Magny-Cours CPUs. However, their implementation



(a) *basic look-ahead*: roughly 13% of each iteration time is spent in U broadcast, swapping and DTRSM in case of basic lookahead.



(b) *pipelined look-ahead*: less than 3% is spent in these routines when employing the proposed swapping pipeline.



(c) *runtime comparison*: comparison for an execution with two co-processors. Up to 11% can be saved per iteration due to swapping pipeline.

Figure 9. Execution profile of multi-node (2x2) hybrid HPL with and without pipelining. The x-axis gives the problems size per iteration whereas the y axis denotes the (relative) execution time per iteration.

resulted in only 0.5% improvement, while our implementation enjoys up to 11% gain, as shown in Section V-C. Such large improvement is due to pipelining of U broadcast, whose exposed time hurts Knights Corner more than AMD Cypress GPU, which is slower. Bach et al. [3] mentions pipelining the U broadcast in the future work section.

B. *Offload DGEMM*

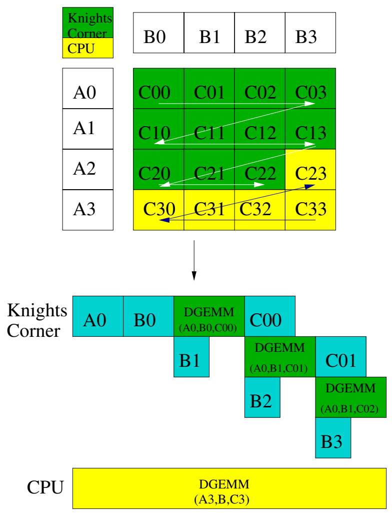
The offload DGEMM kernel, which offloads the trailing matrix update from Sandy Bridge EP to Knights Corner, is the key part of our hybrid HPL implementation. In this section, we describe the design and implementation of this kernel and show how it addresses the following two challenges, paramount to achieving high performance: (i) choosing optimal block size, such that data transfer overhead is hidden, while Knights Corner DGEMM efficiency is maximized, (ii) dynamically load-balancing computation between Sandy Bridge EP and Knights Corner.

Figure 10 shows the high-level design of offload DGEMM. Sandy Bridge EP divides and copies large input matrices into smaller tiles (eg., *A0*, *B2*) and sends them to Knights Corner. Knights Corner, in turn, locally computes DGEMM on these tiles and sends the result C tile (eg., *C02*) to Sandy Bridge EP, which accumulates it back into original C matrix. Due to the fact that copying input tiles is memory bound, Sandy Bridge EP combines 'copy' operation with packing data into our Knights Corner-friendly format (see Section III-A3). To hide the overhead of transferring C tiles back to the host, we choose tile size as follows. To compute one tile on Knights Corner takes $(2 \cdot Mt \cdot Nt \cdot Kt)/P_{dgemm}$, where Mt, Nt, and Kt, are tile dimensions, and P_{dgemm} is Knights Corner's achievable DGEMM performance. The PCIe transfer time, T_{pcie}, of the corresponding $Mt \times Nt$ output tile is $T_{\text{pcie}} = 8 \cdot Mt \cdot Nt / B_{\text{pcie}}$, where B_{pcie} is PCIe transfer bandwidth. To hide the PCIe transfer overhead requires the ratio of compute to transfer be greater than 1.0, which results in the following lower-bound on Kt: $Kt > 4 \cdot P_{\text{dgem}} / B_{\text{pcie}}$. In our case, BW_{pcie} is ≈ 4 GB/s [4] and P_{dgm} is ≈ 950 GFLOPS. As a result, the panel width Kt should at least be 950. To further account for transferring input tiles (which result in a much smaller portion of PCIe traffic compared to output tile) and the fact that the best performing Knights Corner DGEMM requires $k = 300$ (see Section III-B), we use $Kt = 1200$ in our experiments.

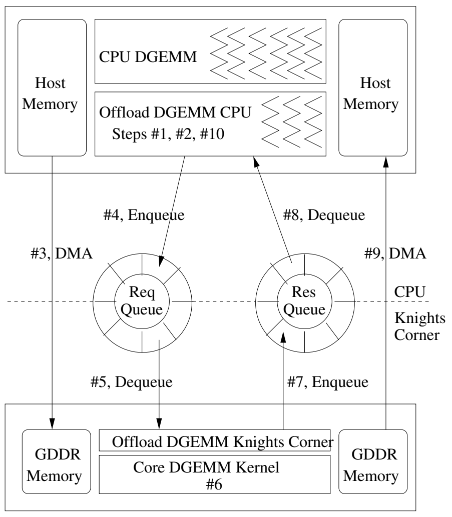
Figure 10b shows the detailed steps involved in implementing DGEMM. We use several designated Sandy Bridge EP cores to pack input tiles, initiate DMA requests and copy result tile back into original matrix (Steps 1, 2 and 10). All communication between Knights Corner and host happens via memory-mapped queue: Sandy Bridge EP inserts a Knights Corner DGEMM into a queue (Step 4), while Knights Corner constantly polls this queue (Step 5) for new requests and starts the DGEMM kernel when the request arrives (Step 6). The remaining Sandy Bridge EP cores are free to do other work, such as panel factorization, as well as work in parallel with Knights Corner on the rest of the DGEMM. To achieve good load-balance and reduce idle time on both Sandy Bridge EP and Knights Corner, we use dynamic work-stealing (mentioned in Section V-A), as follows: Knights Corner starts with the first tile in the upper-left corner of the matrix (*C00* in Figure 10a), and continues forward in column-major order, stealing one tile at a time. When Sandy Bridge EP finishes other tasks and is ready to work on the trailing update, it starts with the last tile in the lower-right corner of the matrix (*C33* in Figure 10a) and continues backwards also stealing one tile at a time. Both Knights Corner and Sandy Bridge EP continue in this fashion, until there are no more tiles to steal.

As Figure 10a shows, overlapping communication and computation in offload DGEMM, exposes the overhead of

[4]While 5.5 GB/s bandwidth is possible to achieve, in our case, PCIe transfers compete for memory bandwidth with swapping and DGEMM, executed on Sandy Bridge EP



(a) Matrix is divided into tiles and distributed across Knights Corner and Sandy Bridge EP.



(b) Detailed diagram of DGEMM offload.

Figure 10. Design of offload DGEMM.

processing the first and last tile on Knights Corner. Note that this overhead is higher with bigger tile sizes, as there are fewer tiles to amortize it, and lower with smaller tile sizes, as there are more tiles to alleviate it. On the other hand, smaller tiles result in lower DGEMM efficiency on Knights Corner. To address this challenge, for each matrix size, *Mt* and *Nt*, we pre-compute the best tile sizes that maximize overall offload DGEMM efficiency, and dynamically pick the best tile size at run-time.

We also performed several other optimizations to further improve offload DGEMM efficiency. First, if the matrix size is not a multiple of the tile size, partial (smaller) tiles will exist and processing them on Knights Corner can expose data transfer overheads. To alleviate this overhead, we merge the last two tiles (one complete tile and one partial tile) at the end of each row or column and process them together. Second, to effectively utilize the memory bandwidth available from two CPU sockets, it is critical to distribute load uniformly between the sockets. For example, if there are two PCIe operations or two copy operations in-flight, we need to ensure that they go to different sockets. We achieve this by explicitly partitioning the matrices and distributing these partitions across the two sockets in an interleaved manner.

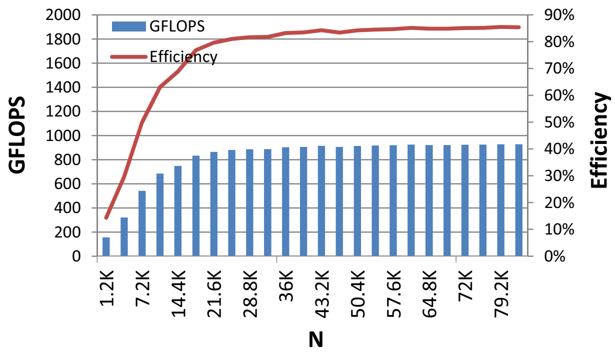
Lastly, as presented in Section V-A, *pipelined look-ahead* software pipelines swapping within HPL to overlap swapping on the host with a portion of the trailing update on Knights Corner. This requires a synchronization mechanism between swapping threads and DGEMM offload threads on Sandy Bridge EP to assure that offloaded DGEMM tiles have already been swapped. This synchronization mechanism along with dynamic work stealing and dynamic tile size selection are our main contributions to the design of offload DGEMM. The rest of the design is similar to the offload DGEMM design described by Bach et al. [3] Philips [6], and Yang et al. [20].

Figure 11a shows the performance of offload DGEMM using a single Knights Corner for varying matrix sizes. For 82K matrix it achieves ≈917 GFLOPS, resulting in 85.4% efficiency. This is 4% lower compared to native DGEMM efficiency, which achieves 89.4% (see Section III-B). The reason for 4% efficiency loss is as follows: first, during offload DGEMM, one of the cores on Knights Corner is used for communication with the Sandy Bridge EP. This results in 1.5% efficiency loss. The remaining 2.5% efficiency loss is due to the exposed overhead of transferring first and last tiles. Overall, efficiency degrades slowly with decreasing matrix sizes. We observe similar performance trends when using two Knights Corner cards, as shown in Figure 11b. The achieved peak offload DGEMM performance for dual Knights Corner systems is 1785 GFLOPS, resulting in 83% efficiency. We see that the efficiency degrades much faster as compared to the single Knights Corner system. In the dual Knights Corner system, each Knights Corner is only solving half the problem size as compared to a single Knights Corner system; thus, the first and last tile processing contribute to a larger extent, resulting in overall performance degradation.

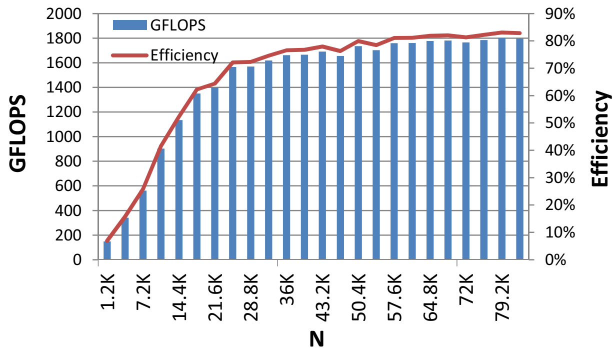
C. Hybrid HPL Performance Results

In this section we present the single-node and cluster-level results of our hybrid HPL implementation. Each node has a peak performance of 1.4 TFLOPS with a single Knights Corner card and 2.48 TFLOPS with two Knights Corner cards, respectively. Cluster nodes are connected with a single rail FDR Infiniband network.

The results of HPL run are shown in Table III, which is divided into four sections. The first section shows HPL results on a CPU-only cluster featuring the host system's



(a) 1 coprocessor accelerated offload-dgemm.



(b) 2 co-processors accelerated offload-dgemm.

Figure 11. Offload DGEMM performance for matrices occurring during trailing update: $M = N$ and $Kt = 1200$.

System	N	P	Q	TFLOPS	Eff.
Sandy Bridge EP, 64GB	84K	1	1	0.29	86.4
Sandy Bridge EP, 64GB	168K	2	2	1.10	82.8
no pipeline, 1 card, 64GB	84K	1	1	0.99	71.0
pipeline, 1 card, 64GB	84K	1	1	1.12	79.8
no pipeline, 1 card, 64GB	168K	2	2	3.88	69.1
pipeline, 1 card, 64GB	168K	2	2	4.36	77.6
no pipeline, 1 card, 64GB	825K	10	10	95.2	67.7
pipeline, 1 card, 64GB	825K	10	10	107.0	76.1
no pipeline, 2 cards, 64GB	84K	1	1	1.66	68.2
pipeline, 2 cards, 64GB	84K	1	1	1.87	76.6
no pipeline, 2 cards, 64GB	166K	2	2	6.36	65.0
pipeline, 2 cards, 64GB	166K	2	2	7.15	73.1
no pipeline, 2 cards, 64GB	822K	10	10	156.5	64.0
pipeline, 2 cards, 64GB	822K	10	10	175.8	71.9
pipeline, 1 card, 128GB	242K	2	2	4.42	79.6

Table III
ACHIEVED PERFORMANCE ON NODE AND CLUSTER LEVEL FOR DIFFERENT KNIGHTS CORNER CONFIGURATION AND HOST MEMORY CONFIGURATIONS. THE NUMBER OF USED NODES CAN BE DERIVED BY MULTIPLYING P (# PROCESS-ROWS) AND Q (# PROCESS COLUMNS).

processor (Intel® Xeon® E5-2670), using Intel MKL MP Linpack [2]. This implementation achieves 82.8% efficiency on 2x2 cluster, 4% degradation, compared to single node efficiency.

The second and third sections show hybrid results with one and two Knights Corner cards per node, respectively. First, we observe that *pipelined look-ahead* improves hybrid HPL efficiency by 7%-9%, as it reduces exposed idle time on Knights Corner. Second, the efficiency loss due to a second Knights Corner card is 4.2%. This is due to the lower efficiency of the dual card offload DGEMM as well as higher penalty of Knights Corners' idle time. Note that similar to the Sandy Bridge EP-only result, performance degradation of multi-node implementation, compared to a single node is 4%. Finally, as the fourth section shows, doubling available node memory to 128 GB, the dual card cluster-level efficiency increases by 3.5% to 79.6 %.

Overall, using dynamic scheduling and enhanced look-ahead scheme, our HPL implementation scales well to a 100-node cluster, on which it achieves over 76% efficiency while delivering the total performance of 107 TFLOPS.

VI. Related Work

There is a large body of related work in the area of exploiting multi-core processors and hardware accelerators, such as GPUs and Cell B.E., to accelerate a variety of dense linear algebra routines.

As the result, there are many highly optimized DLA libraries for these architectures. [11, 2, 22, 18, 16] are some more recent examples of these libraries. Our design and implementation of DGEMM and Linpack kernels builds upon ideas found in these libraries, and extends them in the context of Knights Corner.

High Performance Linpack [15] has traditionally been a benchmark of choice for ranking top 500 fastest supercomputers [19]. Recent work has focused on speeding up HPL using GPUs [3, 8, 7, 6]. To overcome limitted memory capacity of GPUs, these implementations let the entire problem reside in CPU memory, and offload critical computations, such as DGEMM and DTRSM, to GPU. Our HPL implementation using Knights Corner, builds upon several ideas found in these approaches, such as work splitting between CPU and GPU, task co-scheduling and software pipelining to hide PCI-Express communication overhead. We enhance these techniques with dynamic work stealing, and run-time adaptive tile size selection, as well as advanced pipelined look-ahead schemes, to further improve HPL efficiency on Knights Corner.

VII. Conclusion

We have described the design and implementation of several dense linear algebra kernels on single- and multi-node systems, based on the recently released Intel® Xeon Phi™ coprocessor. These implementations take advantage of our highly tuned DGEMM, which achieves close to 90% efficiency on Knights Corner architecture. Using our novel *pipelined look-ahead* scheme, our hybrid Linpack implementation achieves 76.1% efficiency. This is 11% improvement in efficiency over the corresponding entry that we submitted to TOP500 list in June'12, which achieved 118 TFLOPS (on 140 nodes) and ranked number 150 [19], but did not use *pipelined look-ahead*.

There are several drawbacks of our hybrid Linpack implementation. First, limited PCIe bandwidth imposes a lower-bound on block size, which slows panel factorization, relative to the rest of Linpack. To hide panel factorization requires increased amount of host memory. Second, the fact that Sandy Bridge EP is several times slower than Knights Corner, but consumes comparable power, makes hybrid implementation less energy efficient compared to the fully-native multi-node implementation that only uses Knights Corners.

Our fully native 79% efficient single-node Linpack implementation on Knights Corner is a first step in the direction of running the Linpack directly on a cluster of Knights Corners, while CPU cores are put into a deep sleep state to significantly reduce their energy. This is the focus of our future work.

ACKNOWLEDGMENT

The authors would like to thank Andrey Moskalev, Konstantin Arturov, and Michael Chuvelev for their help in designing and optimizing various portions of the native Linpack implementation. We would also like to thank Catherine Djunaedi, Ravi Murty and Susan Meredith for their continuous support in building, debugging and analyzing the systems. Finally, we thank Jason Sewall for carefully proof-reading the paper and providing useful comments and suggestions.

REFERENCES

[1] Basic Linear Algebra techical forum standard, August 2001.

[2] Intel Math Kernel Library (Intel MKL) 11.0, 2012.

[3] M. Bach, M. Kretz, V. Lindenstruth, and D. Rohr. Optimized HPL for AMD GPU and multi-core CPU usage. *Comput. Sci.*, 26(3-4):153–164, June 2011.

[4] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. The impact of multicore on math software. In *Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing*, PARA'06, 2007.

[5] M. Deisher, M. Smelyanskiy, B. Nickerson, V. W. Lee, M. Chuvelev, and P. Dubey. Designing and dynamically load balancing hybrid LU for multi/many-core. *Comput. Sci.*, 26(3-4), June 2011.

[6] E. Philips and M. Fatica. CUDA Accelerated Linpack on Clusters. In *GPU Technology Conference*, 2010.

[7] T. Endo, A. Nukada, S. Matsuoka, and N. Maruyama. Linpack evaluation on a supercomputer with heterogeneous accelerators. In *IPDPS*, pages 1–8. IEEE, 2010.

[8] M. Fatica. Accelerating linpack with CUDA on heterogenous clusters. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, page 4651. ACM, 2009.

[9] G. H. Golub and C. F. V. Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996.

[10] K. Goto and R. A. v. d. Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, May 2008.

[11] K. Goto and R. Van De Geijn. High-performance implementation of the level-3 blas. *ACM Trans. Math. Softw.*, 35(1):4:1–4:14, July 2008.

[12] Greg Henry. BLAS Based on Alternate Data Structures. In *Cornell Theory Center Technical Report Number 89*, 1992.

[13] HPC Advisory Council. Toward Exascale computing, 2010.

[14] J. D. McCalpin. The STREAM Benchmark.

[15] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers.

[16] F. Song and J. Dongarra. A scalable framework for heterogeneous gpu-based clusters. In *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures*, SPAA '12, pages 91–100, New York, NY, USA, 2012. ACM.

[17] P. Strazdins. A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization, 1998.

[18] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proceedings of IPDPS 2010: 24th IEEE International Parallel and Distributed Processing Symposium*, 2010.

[19] www.top500.org. TOP500 list, June 2012 release. 2012.

[20] C. Yang, F. Wang, Y. Du, J. Chen, J. Liu, H. Yi, and K. Lu. Adaptive optimization for petascale heterogeneous cpu/gpu computing. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pages 19 –28, sept. 2010.

[21] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[22] F. G. V. Zee, E. Chan, R. A. v. d. Geijn, E. S. Quintana-Orti, and G. Quintana-Orti. The libflame library for dense matrix computations. *IEEE Des. Test*, 11(6):56–63, Nov. 2009.