Comput Sci Res Dev (2011) 26: 211–220
DOI 10.1007/s00450-011-0169-x

SPECIAL ISSUE PAPER

Designing and dynamically load balancing hybrid LU for multi/many-core

Michael Deisher · Mikhail Smelyanskiy · Brian Nickerson · Victor W. Lee · Michael Chuvelev · Pradeep Dubey

Published online: 14 April 2011

Abstract Designing high-performance LU factorization for modern hybrid multi/many-core systems requires highly-tuned BLAS subroutines, hiding communication latency and balancing the load across devices of variable processing capabilities. In this paper we show how single-precision LU factorization is accelerated on Intel® MIC(Many Integrated Core) architecture in both native and hybrid (Intel® Xeon® processor and Intel MIC) configurations. Our SGEMM implementation delivers close to 1 Tflop/s on Intel's first implementation of Intel MIC architecture [codenamed Knight's Ferry (KNF)] silicon platform. Our implementation takes full advantage of multiple levels of memory hierarchy on MIC, and successfully utilizes up to 80% of its peak compute capability. Our LU factorization performance exceeds 570 Gflop/s including matrix transfer overhead when executed entirely on a KNF coprocessor. Our hybrid implementation, which offloads parts of LU processing to a dual-socket multi-core Intel Xeon processor X5680 host, delivers up to 772 Gflop/s. The novel aspect of our implementations is dynamic resource partitioning to improve load balance across the entire system.

Keywords High performance computing · Hybrid architecture · LU factorization · Intel MIC architecture · SGEMM · Many-core architecture · Dense linear algebra · Panel factorization · Partial pivoting · Right looking

M. Deisher (✉)
Intel Labs, Hillsboro, OR, USA
e-mail: michael.deisher@intel.com

M. Smelyanskiy · V.W. Lee · P. Dubey
Intel Labs, Santa Clara, CA, USA

M. Smelyanskiy
e-mail: mikhail.smelyanskiy@intel.com

V.W. Lee
e-mail: victor.w.lee@intel.com

P. Dubey
e-mail: pradeep.dubey@intel.com

B. Nickerson
Intel Architecture Group, Santa Clara, CA, USA
e-mail: brian.r.nickerson@intel.com

M. Chuvelev
Software and Solutions Group, Nizhny Novgorod, Russia
e-mail: michael.chuvelev@intel.com

1 Introduction

The dense LU factorization, which represents matrix A as a product of lower and upper triangular matrices, L and U, is a central kernel commonly used in many linear algebra operations, such as solving non-symmetric systems of linear equations or inverting a matrix.

Modern many-core architectures deliver from several hundreds of Gflop/s to several Tflop/s of floating-point single-precision performance and from tens of GB/s to over a hundred GB/s memory bandwidth. Hybrid systems which consist of one or several multi-core architectures enhanced with one or several co-processors, connected via slow links, are also becoming common [10]. To achieve high LU performance on such systems requires careful tuning and optimization of BLAS, hiding communication latency and balancing the load across devices of variable processing capabilities. In this paper we demonstrate how LU factorization can be accelerated on Intel® MIC (Many Integrated Core) architecture in both native and hybrid (Intel Xeon processor and Intel MIC) configurations. We restrict our discussion to single-precision implementations of LU, even though the observations noted and insights derived are applicable to more common double precision implementations as well.

We consider the factorization of matrices that reside in the CPU memory in column-major layout, and whose factorization overwrites the original data. We make the following contributions:

High performing single coprocessor SGEMM: Our highly tuned matrix-matrix multiply (SGEMM) implementation takes full advantage of multiple levels of memory hierarchy present on KNF, including a register file and two levels of caches to satisfy the bandwidth constraints between each level. Our implementation delivers close to 1 Tflop/s on an Intel MIC coprocessor [codenamed Knight's Ferry (KNF)] for matrices in row-major or column-major format. This corresponds to almost 80% of the KNF compute peak.

High performing single coprocessor LU: Our optimized native LU implementation, which runs entirely on a KNF system, delivers up to 650 Gflop/s of performance without data transfer overhead and up to 574 Gflop/s of performance when transfer time is included. This implementation employs a novel dynamic core allocation mechanism to partition cores between panel factorization and trailing sub-matrix update.

High performing hybrid LU: Our hybrid implementation, which offloads parts of LU processing to a dual-socket multi-core Intel Xeon processor X5680 host, achieves up to 772 Gflop/s. Similar to other implementations, our hybrid approach offloads panel factorization and portions of SGEMM to CPU cores. Here we use a new dynamic core allocation and matrix partitioning scheme for better load balance.

The rest of this paper is organized as follows. In Sect. 2 we present the standard LU factorization algorithm and our experimental environment. Section 3 describes our optimized implementation of SGEMM, provides results, and presents their analysis. Sections 4 and 5 present tuning and performance analysis of native and hybrid LU implementations. Results are compared with previous work in Sect. 6. We conclude and present our future work in Sect. 7.

2 Background

2.1 LU factorization

The LU factorization algorithm decomposes a matrix A into a product of a lower-triangular matrix L and an upper triangular matrix U. The blocked LU formulation is shown schematically in Fig. 1. 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 until the entire matrix is factorized. At each step, a portion of the column panel of L consisting of D and L_i is first factored. Then a portion of the row panel U_i is updated using a triangular forward solve. The trailing sub-matrix A_i is 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. Pivots are computed within a single column, and individual rows, not blocks, are swapped during pivoting. After this process is completed, the solution of $Ax = b$ can be obtained by forward and back substitution with L and U. A more detailed treatment of the algorithmic issues can be found in [6].

Fig. 1 LU factorization

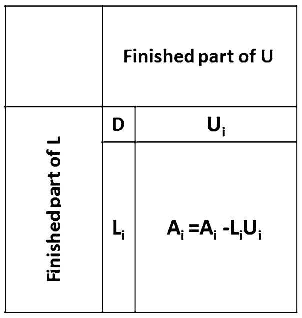


Table 1 Architectural characteristics of Intel Xeon processor X5680 and KNF coprocessor

	Xeon	KNF
Sockets	2	1
Cores/socket	6	32
Core Frequency, GHz	3.3	1.2
SIMD Width	4	16
SIMD Regs/core	16	32
L1 cache, KB	32	32
L2 cache, KB	256	256
L3 cache, MB	12	n/a
Peak bandwidth, GB/s	64	115
Peak SP, Gflop/s	316	1,228

2.2 Modern many-core architectures

Our experimental testbed consists of a dual-socket Intel® Xeon® processor with a KNF coprocessor attached to it via PCIe 2.0 × 16 interface. We next provide the background on each of these architectures. Their key features are summarized in Table 1.

Intel Xeon processor: The Intel Xeon processor X5680 is based on an x86-based multi-core architecture which provides six cores on the same die. Each core is running at up to 3.3 GHz. The architecture features a super-scalar out-of-order micro-architecture supporting 2-way hyper-threading and 4-wide SIMD. Each core is backed by a 32 KB L1 and a 256 KB L2 cache, and all six cores share a 12 MB L3 cache. Six-core CPUs each deliver a peak 158 Gflop/s of single-precision, as well as 32 GB/s of peak main memory bandwidth. To benchmark SGEMM and LU factorization on this platform, we used highly optimized BLAS and LA-

PACK implementations from the Intel® Math Kernel Library (MKL) 10.2.

Intel Knight's Ferry: KNF is an Intel MIC architecture platform. Intel MIC, whose schematic diagram is shown in Fig. 2, is an x86-based many-core processor architecture based on small in-order cores that uniquely combines the full programmability of today's general purpose CPU architecture with compute throughput and memory bandwidth capabilities of modern GPU architectures. Each core is a general purpose processor, which has a scalar unit based on the Intel® Pentium® processor design, as well as a vector unit that supports 16 32-bit float or integer operations per clock. KNF has a peak Gflop/s of 1,228 and peak bandwidth of 115 GB/s. It has two levels of cache: a low latency 32 KB L1 data cache and a larger globally coherent L2 cache that is partitioned among the cores, each core having a 256 KB partitioned L2 cache. As a result, widely-used multiprocessor programming techniques, such as Pthreads or OpenMP apply to KNF. To further hide latency, each core is augmented with 4-way multi-threading. The KNF pipeline is dual-issue: scalar instructions as well as pre-fetch instructions can pair and issue in the same cycle as vector instructions.

3 Matrix-matrix multiply

In this section we describe our design and implementation of single-precision matrix-matrix multiply (SGEMM), $C = \alpha AB + \beta C$. To avoid inefficient strided memory access during pivoting, our matrix is laid out in row-major order. Our implementation can be trivially extended to handle column-major format.

3.1 Implementation

Basic kernel: Our basic SGEMM kernel, shown in Fig. 3(a) multiplies a 6×4 sub-block of A by a 4×64 sub-block of B and stores the result into a 6×64 sub-block of C. The outermost loop iterates over the rows of sub-blocks of A. Each 4-element row is multiplied by the entire sub-block of B and the result is aggregated into a corresponding row of a sub-block of C. Our implementation performs register blocking to reduce pressure on the $L1$ cache. Specifically, 6×64 sub-blocks of C are pre-loaded into 24 16-wide vector registers, $vc_{i,j}$, which are re-used across multiple invocations of the basic kernel to amortize the overhead of filling and spilling these registers from and to memory.

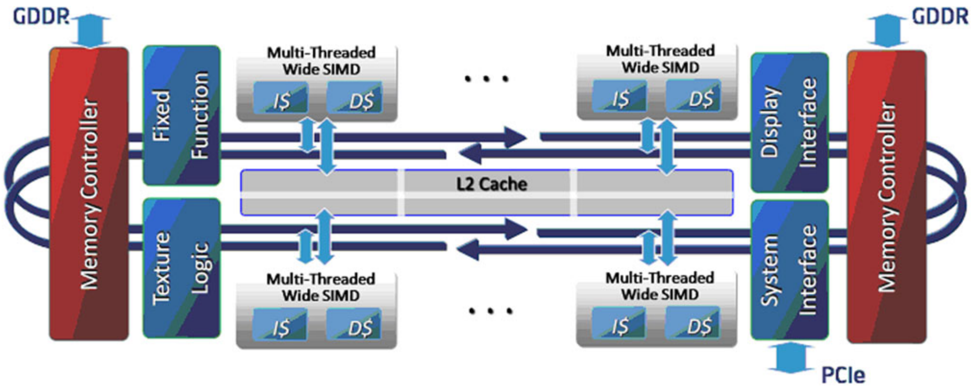
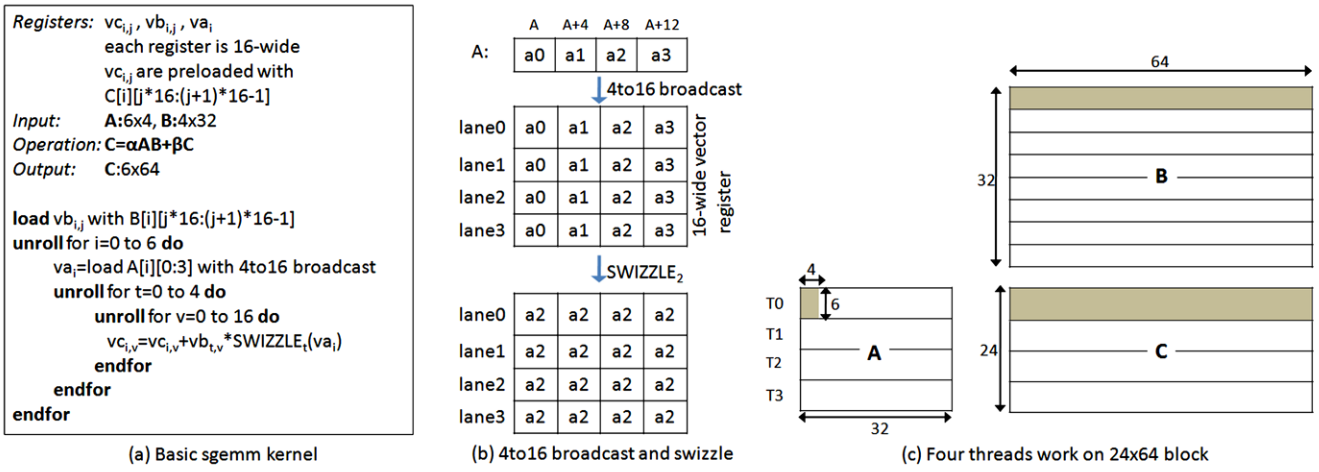


Fig. 2 Intel MIC architecture diagram



```
Registers:  vc_{i,j} , vb_{i,j} , va_i
            each register is 16-wide
            vc_{i,j} are preloaded with
            C[i][j*16:(j+1)*16-1]
Input:      A:6x4, B:4x32
Operation:  C=αAB+βC
Output:     C:6x64

load vb_{i,j} with B[i][j*16:(j+1)*16-1]
unroll for i=0 to 6 do
    va_i=load A[i][0:3] with 4to16 broadcast
    unroll for t=0 to 4 do
        unroll for v=0 to 16 do
          vc_{i,v}=vc_{i,v}+vb_{t,v}*SWIZZLE_t(va_i)
        endfor
    endfor
endfor
```

(a) Basic sgemm kernel

Fig. 3 Basic SGEMM kernel

Elements of sub-blocks of B are loaded into $vb_{i,j}$ registers for each invocation of the kernel, but they are reused across each row of C. To multiply 4-element rows of sub-blocks of A by the entire sub-block of B, we load the four elements into va using the 4-to-16 broadcast instruction shown in Fig. 3(b). This instruction replicates four elements in all four lanes. The inner loop multiplies each element of a row of a sub-block of A by a row of a sub-block of B. To do this, we replicate the required element across the entire vector using in-register swizzle, as shown in Fig. 3(c). The swizzle is fused with the multiply-add instruction and has no additional penalty.

Exploiting data locality: Each of four threads on a single core calls the basic kernel eight times to multiply a 6×32 block of A by a 32×64 block of B, and store the result into a 6×64 block of C. Conceptually, a core works on a 24×64 block of C, as shown in Fig. 3(c). Note that a 32×64 block of B fits into $L1$ cache and is therefore shared among four threads. Furthermore, blocks of A and B are combined into panels. A panel of A is some number, pw, of horizontally adjacent 24×32 blocks of A, while a panel of B is the same number of vertically adjacent 32×64 blocks of B. Multiplying these panels together produces a 24×64 block of C. Panel size pw is selected to fit well into the 256 K $L2$ cache. To decrease the byte:flop ratio of SGEMM, each core multiplies ph vertically adjacent panels of A ($24ph \times 32pw$ block), by the same panel of B (32×64 block), which stays reused in the $L2$ cache, to produce a $24ph \times 64$ block of C. We seek ph and pw such that:

- Panels of A and B fit into the $L2$ cache partition. This requires the following condition to be true: $4(24 * 32pw + 32pw * 64) < \text{L2partsize}_{KNF}$.
- The byte:flop ratio of multiplying pw panels of A by a panel of B is less than the byte:flop ratio of KNF. This results in the second condition:

$$\frac{4(24ph32pw + 32pw64 + 24ph64)\text{ bytes}}{(24ph32pw64 * 2)\text{ flops}} < \text{byte} : \text{flop}_{KNF}$$

There are many values of ph and pw which satisfy these two conditions. Using auto-tuning we discovered that for $ph = 5$ and $pw = 18$, SGEMM achieves best performance.

Memory latency optimization: To hide memory latency, our implementation uses software pre-fetching. KNF has two types of pre-fetch instructions: an instruction that pre-fetches data from DRAM into $L2$ cache, and an instruction that pre-fetches data from $L2$ into $L1$ cache. When we compute a 24×64 block of C, we issue $L2$ pre-fetches for the data required by the next block. This reduces $L2$ warm-up misses as we move across blocks. Due to the fact that each block takes at least $24 * 32 * 64/16 = 3000$ cycles of computation, $L2$ pre-fetches have enough time to complete. Each basic SGEMM kernel issues $L1$ pre-fetches for the next 6×4 sub-block of A and the next 4×64 sub-block of B. Finally, on KNF pre-fetches are queued up in pre-fetch buffers. There is limited number of these buffers. Hence, special care is taken to space-out $L1$ and $L2$ pre-fetches so as not to exceed the number of these buffers. Exceeding the number of pre-fetch buffers results in extra pipeline stalls.

Thread-level parallelism: Super-blocks of C are divided evenly between cores to achieve good load balance. To ensure that four threads run in sync and thus have constructive sharing of data in the core's $L1$ and $L2$ caches, we deploy a low overhead synchronization mechanism. Specifically, each core has an array of four counters, collocated into a single cache line, with each core having its own cache line. At intervals during the processing kernel, each thread increments its own counter and then compares against all counters of the core. When unequal, the thread issues a DELAY instruction for some tunable number cycles; when equal, the thread continues processing its block. Without this synchronization mechanism between threads on the same core, we observed performance degradation of almost $2\times$. The degradation comes from threads getting out of sync, and as a result working on different parts of B, which, in turn, results in a large working set that does not fit into the $L1$ and $L2$ caches.

3.2 Performance comparison and analysis

Figure 4 shows the performance of SGEMM on an Intel Xeon processor X5680 and KNF coprocessor. The bottom two curves show SGEMM performance on single and dual socket Xeon processors, respectively. Single socket achieves a maximum performance of 150 Gflop/s, which corresponds to 94% efficiency, while dual socket achieves 290 Gflop/s,

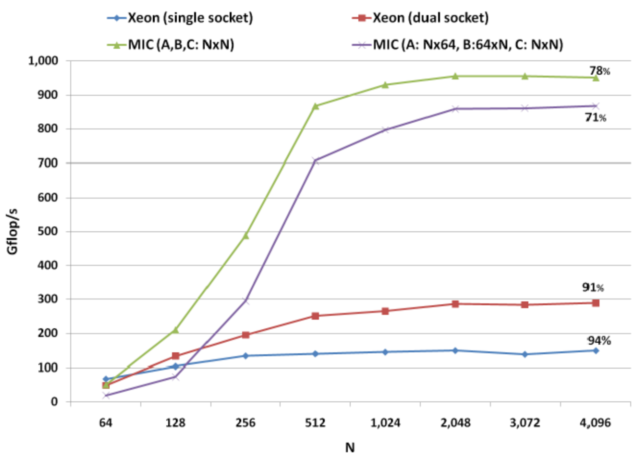


Fig. 4 Performance of SGEMM on Xeon X5680 and KNF

which corresponds to 91% efficiency. The top curve shows KNF performance on a general SGEMM, where all three matrices have equal numbers of rows and columns, N. KNF achieves a maximum performance of 956 Gflop/s, which corresponds to 78% efficiency.

The second curve from the top shows performance of a special SGEMM which does rank-64 update. As discussed in Sect. 2, this is the key operation of our blocked LU implementation. We see that rank-64 SGEMM achieves a lower efficiency of 71%, compared to the general SGEMM efficiency of 78%. The loss in efficiency is due to the following reasons. First, for rank-64 update, panel width, pw, must be 64 which exposes the instruction overhead of moving block C between memory and registers. Second, the number of vertically adjacent panels of A (a block of $24ph \times 32pw$ elements) must also be 64, which is not a multiple of 6. As the result, in contrast to the general SGEMM where each call to a basic kernel works on 6 rows of A, in the rank-64 SGEMM some calls to the basic kernel work only on 5 rows of A to make the total work add up to 64. This reduces basic kernel SIMD efficiency.

4 LU Factorization: KNF implementation

In this section, we describe our implementation of LU, explain design trade-offs, provide details of optimization of its modules, and compare its performance on two hardware architectures. Our experiments are done for randomly generated matrices.

Following [12] and [15] we adopt a right-looking implementation of LU because it exposes the maximum amount of data- and thread-level parallelism in the SGEMM routine. Our implementation also uses a static schedule and overlaps column panel LU with trailing sub-matrix update using a look-ahead technique. However, in contrast to [15] and [2] that perform panel LU on the host CPU and trailing sub-matrix update on a GPU we perform both operations on a KNF coprocessor.

The main operations of LU are SGEMM, block inverse, row swapping, and panel LU. Matrix-matrix multiply was discussed in the previous section. The block inverse (which, along with an SGEMM, takes the place of triangular solve) was implemented in a single-threaded section. Sufficient performance was obtained by implementing it using KNF vector instructions. Overall it only accounted for 2% of LU performance for the 8192 × 8192 problem to 13% of LU for the 1024 × 1024 problem after optimization. The row swapping operation was implemented using vector load and store instructions and was parallelized by assigning each block column of the matrix to a thread. Row major layout makes row swapping bandwidth friendly. Overall row swapping only accounts for 2.7% of LU performance for the 1024 × 1024 problem to 4.5% of LU performance for the 8192 × 8192 problem after optimization. Since panel LU is critical for good performance we treat it in greater detail in the following sub-sections.

4.1 Panel LU performance target

To balance processing resources among the two parallel operations (panel LU and SGEMM), consider the number of floating point operations required for each. The size of the panel is $(n-k+1) \times v$ while the size of the partial previous trailing sub-matrix is $(n-k+1) \times (n-k-v+1)$. The number of floating point operations required for panel LU is $(n-k+1)v^2 - v^3/3 - v^2/2 + 5v/6$ while the number of flops to update the partial previous trailing sub-matrix is $2(n-k+1)(n-k-v+1)v$. Let F_P denote the performance of panel LU and let F_S denote the performance of the partial previous trailing sub-matrix update. Both are measured in Gflop/s. To balance the two operations we require that

$$\frac{(n-k+1)v^2 - \frac{v^3}{3} - \frac{v^2}{2} + \frac{5v}{6}}{F_P} = \frac{2(n-k+1)(n-k-v+1)v}{F_S}$$

leading to

$$F_P = F_S \frac{(n-k+1)v^2 - \frac{v^3}{3} - \frac{v^2}{2} + \frac{5v}{6}}{2(n-k+1)(n-k-v+1)v}$$

Figure 5 shows a plot of F_P versus panel height for panel widths of 64, 96, and 128, and measured values of F_S on KNF using the SGEMM described in Sect. 3. It shows the required performance of panel LU in each of these cases to balance the load with the partial previous trailing sub-matrix update. For shorter panels higher performance is required to maintain load balance. For example, when panel width is 64, 17.3 Gflop/s are needed at a panel height of 384 while

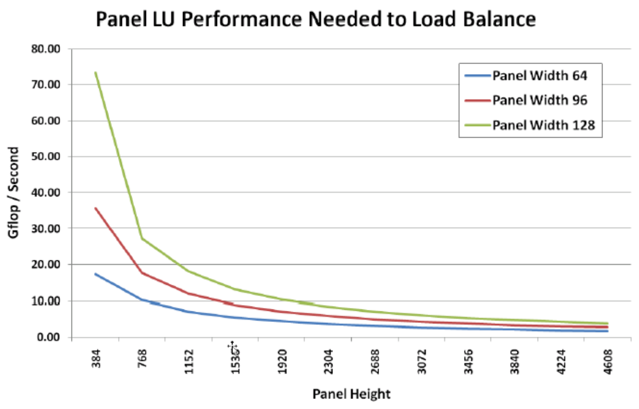


Fig. 5 Performance of panel LU needed to balance trailing sub-matrix update for panel widths of 64, 96, and 128

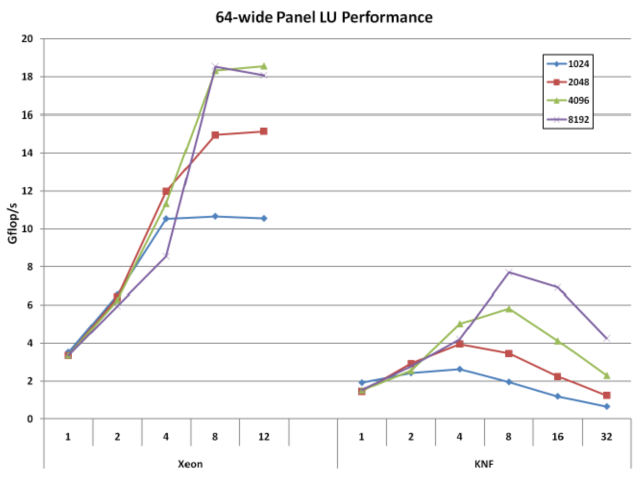


Fig. 6 Performance of optimized panel LU on Xeon X5680 and KNF

1.0 Gflop/s are needed at a panel height of 7680. For our LU implementation we choose a panel width of 64 and use the performance target given by Fig. 5 with the goal of improving panel LU performance to match that of the previous partial trailing sub-matrix update in all cases.

4.2 Optimization of panel LU

We devoted considerable attention to the optimization of panel LU since it is a performance bottleneck of LU factorization. To optimize panel LU, we first combined the pivot search with the previous iteration of the normalization/update loop. The combined inner loop was parallelized by dividing rows below the diagonal among the available threads. SIMD vector instructions were used to exploit data parallelism within the combined loop. Finally, the outer loop was split into v/w sections where w is the SIMD width. Since the number of columns processed decreases by one for each iteration of the outer loop, splitting the outer loop into v/w sections allowed us to decrease the number of instructions in the inner loop of each subsequent section. Finally, software pre-fetching was used to hide memory latency.

Figure 6 shows the panel LU performance achieved on an Intel KNF coprocessor compared with that obtained using a similar optimization approach on an Intel Xeon processor X5680.[1] The KNF coprocessor achieves nearly 8 Gflop/s while the Xeon processor gets more than 18 Gflop/s. Comparing with Fig. 5, neither meets the requirement to perfectly load balance at smaller panel heights. However, load balance is easily achieved at larger panel heights. The panel width of 64 was a good choice since narrower panels introduce communication bottleneck in SGEMM while wider panels are harder to load balance.

[1]Due to lack of row major panel factorization in MKL, a custom row major panel factorization routine was used.

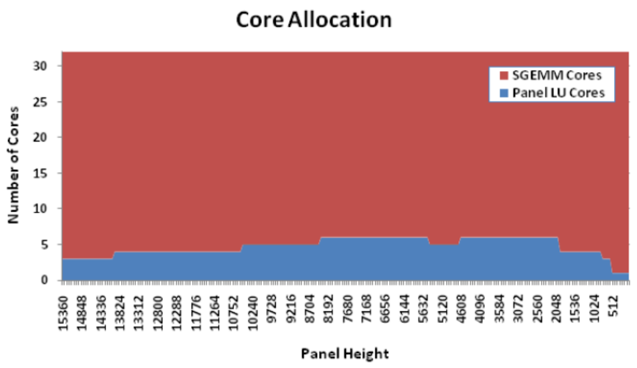


Fig. 7 Cores are allocated between panel LU and trailing sub-matrix update as a function of panel height

4.3 Dynamic core allocation

Both panel LU and trailing sub-matrix efficiencies and performance are a function of panel height. Therefore, the distribution of cores between panel LU and the previous partial trailing sub-matrix update that provides maximum LU performance is also a function of panel height. Since the first panel LU cannot be overlapped with a partial previous trailing sub-matrix update, all cores are available to it. However, maximum performance on KNF is achieved for 7 cores since inter-core communication and synchronization overhead prevents further improvement as more cores are added. This issue is addressed in the next section by offloading panel LU. For subsequent panels, the dynamic core allocation schedule shown in Fig. 7 was developed by measuring overall performance on KNF as panel height is stepped from 64 to 15360 in increments of 64 and number of cores assigned to panel LU N_{panel_cores} is stepped from 1 to the total number of cores, N_{cores}, minus one. The allocation resulting in maximum overall LU performance was recorded for each panel height.

4.4 Overall LU results

Figure 8 shows performance of LU factorization running entirely on an Intel Xeon processor X5680 and entirely on a KNF coprocessor prototype. The data is assumed to be in column-major format in both implementations. This figure shows KNF performance with and without matrix transfer. The KNF implementation achieves nearly 575 Gflop/sec (650 Gflop/sec if transfer is not required) while the Intel Xeon processor implementation achieves about 275 Gflop/sec. Although matrix transfer overhead will diminish as the problem size increases, we reach the memory limit of the hardware before that trend can be seen here. For smaller matrix sizes the Intel Xeon processor outperforms the KNF coprocessor due to lower inter-core communication overhead. Performance on KNF is limited by panel LU time, especially for smaller panel heights. Therefore, we next investigate performance when panel LU is offloaded.

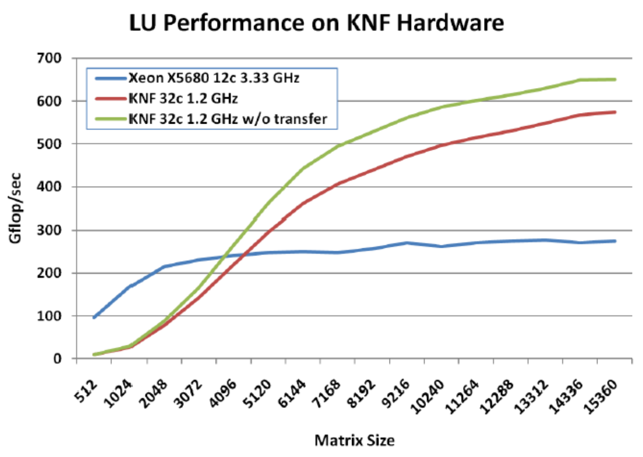


Fig. 8 Performance of LU on Xeon X5680 and KNF

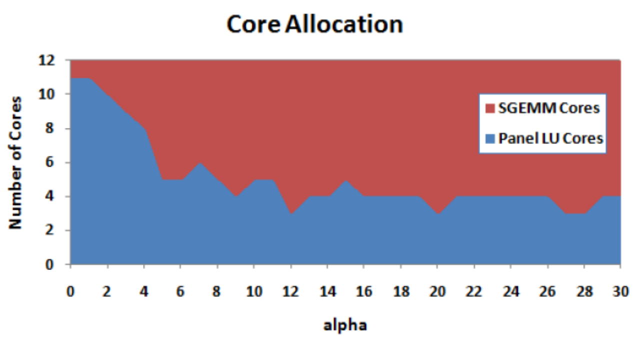


Fig. 9 Core allocation for hybrid LU

5 Hybrid LU implementation

As suggested by Fig. 6 and the findings of others (e.g., [15]), offloading panel LU to the CPU could improve performance provided data transfer overhead is small. Moreover, in [12] further performance improvement was achieved by retaining some number of columns on the right side of the matrix and utilizing free CPU cores to process them. Here, we introduce a load-splitting hybrid LU algorithm with dynamic core allocation and matrix partitioning.

Our algorithm is described as follows. Let α be the number of rightmost $N \times 64$ block-columns retained on the CPU. The first panel of the column major input matrix is factored in place and the block inverse is computed. Then the left $N \times (N - 64\alpha)$ portion of the matrix is transferred to the card along with the pivot vector and block inverse. There the matrix is transposed to convert to row-major format. Row swapping and the SGEMM required to complete the triangular solve operation are carried out for the left and right portions of the matrix on the host CPU and KNF card, respectively. The next panel is updated on the card and transmitted to the host CPU. At this point, α is updated for best load balance (see discussion below) and if necessary the required columns are transferred to the card. Next, panel LU, block inverse, and partial previous trailing sub-matrix update (right part) are performed in parallel on the host CPU while the partial previous trailing sub-matrix update (left part) is performed on the card. Then the factored panel is transmitted to the card. If the card has received new columns due to a change in α then they are updated before proceeding. Once the factored panel has been received by the card, the process is repeated. Once the left side of the matrix is complete, the right side is processed on the host, the matrix is converted to column-major on the card, and the left portion transferred back to the host CPU.

There are two load balance issues in our hybrid LU algorithm. First, on the CPU panel LU and block inverse must be balanced with the partial previous trailing sub-matrix update of the right side of the matrix. Second, CPU processing must be balanced with processing on the card. The first is addressed by allocation of CPU cores between panel LU and SGEMM while the second is addressed by choice of α, the partition size. The analysis presented in Sect. 4.1 can be used determine the core allocation that maximizes overall LU performance. For a panel height m and width 64, panel LU and block inverse require $4096m + 168672$ floating point operations while SGEMM requires $8192\alpha m$ floating point operations. Since SGEMM width is fixed, load balance requires only that $2F_p \approx F_s/\alpha$ for sufficiently large m. Therefore, if performance of panel LU and SGEMM scale linearly with m, then for each α one particular allocation of cores between panel LU and SGEMM is optimal for all panel heights. But since α is chosen dynamically to balance processing on the CPU and card, it is necessary to dynamically change core allocation as a function of α. Figure 9 shows the core allocation determined by exhaustive search. The value of α for each problem size was also determined by a search over measured performance data.

Figure 10 illustrates selection of starting α. There is up to 66 Gflop/s performance difference at the 15360×15360 problem size depending on the choice of initial partition size and core allocation. To choose α at each subsequent panel height, a dynamic programming (Viterbi) search [14] of previously measured performance data was used. The Viterbi search was an attractive choice because it avoids an exhaustive search, allows for easy application of constraints, and produces a smoothed state trajectory. We define the number of cycles required to complete all LU steps for a particular panel height as the noisy observed event. The partitioning α plays the role of hidden state. State transition penalty is defined as the number of cycles required to transfer the needed block-columns to/from the host. Performance at each panel height was measured for $\alpha = 0, \ldots, 30$. We observed that the best α tends to decrease with panel height. By constraining α to decrease monotonically with panel height, implementation was also simplified since block-columns are only transferred in one direction. The search yields the sequence

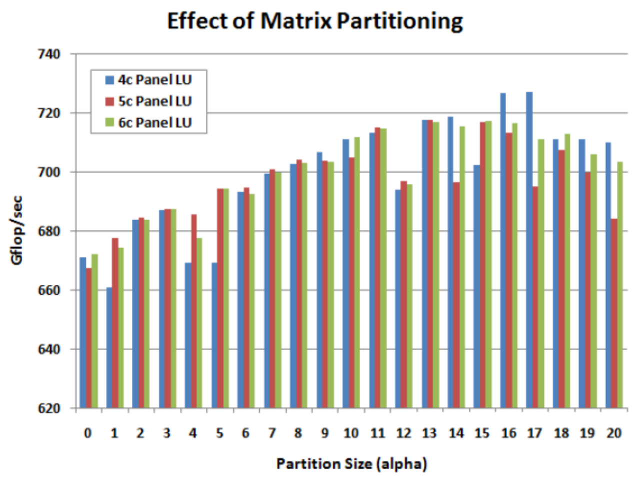


Fig. 10 Performance of 15360 × 15360 LU factorization over initial α and number of panel LU cores

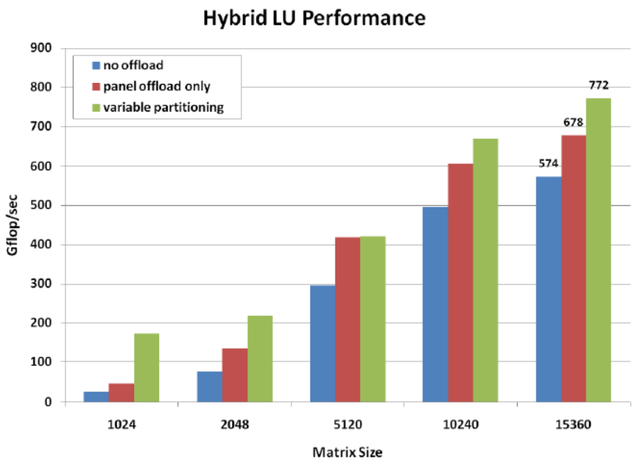


Fig. 11 Performance of hybrid LU on KNF

of α that minimizes the total number of cycles required for LU.

Figure 11 shows measured performance results comparing non-hybrid performance with hybrid performance. The first bar shows performance of the KNF-only LU factorization with dynamic core allocation. The second bar shows performance when panel LU is offloaded to the CPU. The third bar shows performance when both panel LU and partial SGEMM are offloaded. Here, α (and consequently core allocation) is chosen dynamically for each panel height to maximize overall LU performance. The hybrid LU with variable matrix partitioning and core allocation achieves up to 772 Gflop/s. This is 14% better than the algorithm that only offloads panel LU, and 34% better than the KNF-only algorithm.

Table 2 Comparison of system configurations

	Current	Ref. [11]
Host CPU	Xeon X5680	Core2 Q9300
Peak SP Gflop/s	316	40
Peak BW GB/s	64	13
Coprocessor	KNF	Tesla C2050
Peak SP Gflop/s	1228	1075
Peak BW GB/s	115	144

6 Comparison with previous work

Optimization of blocked LU factorization has been studied for many years [5, 6]. Very well-optimized libraries are available for modern multi-core CPUs. The Intel MKL is one such library [9]. Open source libraries are also available demonstrating the latest research techniques [1, 7, 13].

Recently, there have been several papers on hybrid CPU-GPU implementations where panel LU and possibly a portion of trailing sub-matrix update SGEMM are offloaded from the accelerator card to the CPU [2, 8, 13, 15]. The highest reported single-node hybrid LU performance is reported in [11]. In this work, a single socket Quad-Core Intel® Core™2 processor Q9300@2.50 GHz together with a single Fermi C2050 card deliver close to 460 Gflop/s for 10 K matrices in single precision (see Table 2). [8] delivers up to 350 Gflop/s on the single precision problem of the same size.

None of the previous work attempts to optimize full LU factorization entirely on the accelerator. This is in contrast to our work, which delivers 574 Gflop/s of LU performance running entirely on a KNF coprocessor. This native (all-KNF), non-hybrid performance is almost 25% faster than the hybrid performance reported in [11]. Furthermore, our hybrid implementation for the same problem running on a comparable platform of 12 cores and a single KNF coprocessor delivers up to 772 Gflop/s, which is nearly 1.7× faster than the highest performance reported in [11] (or 1.5× faster for 10 K matrices, the largest size in [11]).

Furthermore, existing hybrid implementations do not attempt to dynamically adjust size of SGEMM partition executed on CPU, while our implementation does.

Panel factorization is a key performance limiter to LU factorization. It has limited amount of parallelism and is typically constrained by memory bandwidth, especially on architectures with small last level caches. While it is typically overlapped with trailing sub-matrix update, it can still create sequential dependence between stages of LU. While accelerator-based implementations, including ours, offload panel factorization to the faster CPU cores, there exist alternative implementations which reorganize LU to use smaller granularity task which breaks the sequential dependencies and results in larger amount of parallelism. Current examples include work on LU and QR factorizations, in particular in the so called tiled [3] and communication avoiding [4] algorithms.

7 Conclusion

This paper shows how single precision LU factorization is accelerated on the Intel MIC parallel architecture in both native and hybrid processing configurations. Our implementations of SGEMM and LU factorization take full advantage of the architectural features of the KNF coprocessor and successfully utilize a high fraction of its peak compute capabilities. Our highly tuned SGEMM implementation achieves close to 1 Tflop/s performance on a single KNF coprocessor. Our LU factorization, which employs a dynamic load balancing mechanism, achieves nearly 575 Gflop/s when executed entirely on a KNF coprocessor and up to 772 Gflop/s when executed in hybrid mode. This is faster than previously published results on comparable systems.

Acknowledgements The authors would like to thank Jim Jeffers, Alexander Kobotov, and Steve Sylvester for helpful feedback and advice on KNF performance tuning.

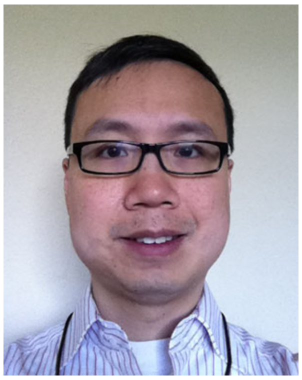
References

1. Agullo E, Demmel J, Dongarra J, Hadri B, Kurzak J, Langou J, Ltaief H, Luszczek P, Tomov S (2009) Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects. J Phys 180(1)
2. Barrachina S, Castillo M, Igual FD, Mayo R, Quintana-Ort ES (2008) Solving dense linear systems on graphics processors. In: Proc Euro-par conference on parallel processing, pp 739–748
3. Buttari A, Langou J, Kurzak J, Dongarra J (2007) A class of parallel tiled linear algebra algorithms for multicore architectures. In: LAPACK working note 191, pp 1–19
4. Demmel J, Grigori L, Xiang H (2010) CALU: a communication optimal lu factorization algorithm
5. Dongarra JJ, Duff IS, Sorensen DC, van der Vorst HA (1987) Numerical linear algebra for high-performance computers. Society for Industrial Mathematics, Philadelphia
6. Golub GH, Loan CFV (1996) Matrix computations. The Johns Hopkins University Press, Baltimore
7. Gunnels JA, Gustavson FG, Henry GM, van de Geijn RA (2001) FLAME: formal linear algebra methods environment. ACM Trans Math Softw 27(4):422–455
8. Humphrey JR, Price DK, Spagnoli KE, Paolini AL, Kelmelis EJ (2010) CULA: hybrid GPU accelerated linear algebra routines. In: Society of photo-optical instrumentation engineers (SPIE) conference series, vol 7705
9. Intel (2009) Intel(R) Math kernel library reference manual. Intel Corporation
10. McIntosh-Smith S, Irwin J (2007) The best of both worlds: delivering aggregated performance for high-performance math libraries in accelerated systems. In: Proc 2007 international supercomputing conference
11. Tomov S (2011) MAGMA 1.0—LAPACK for GPUs. ICL Lunch Talk. http://tinyurl.com/68rz3qk
12. Tomov S, Dongarra J, Baboulin M (2008) Towards dense linear algebra for hybrid GPU accelerated manycore systems. http://www.netlib.org/lapack/lawnspdf/lawn210.pdf
13. Tomov S, Nath R, Du P, Dongarra J (2010) MAGMA version 1.0rc2. http://icl.cs.utk.edu/magma
14. Viterbi A (1967) Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. IEEE Trans Inf Theory 13(2):260–269
15. Volkov V, Demmel JW (2008) Benchmarking GPUs to tune dense linear algebra. In: Proc ACM/IEEE conf supercomputing, pp 1–11

Michael Deisher joined Intel's Parallel Computing Lab as a research scientist in 2009. His research interests include speech signal processing and applied mathematics. He holds M.S. and Ph.D. degrees in Electrical Engineering from Arizona State University, and a B.S. in Computer Engineering from Valparaiso University. Mike serves on the scientific review committees for ICASSP, InterSpeech, and ICSLP. He has served as chair of the IEEE Signal Processing Society's Standing Committee on Industry DSP, chair of the ICASSP Industry Technology Track, and guest co-editor of Signal Processing Magazine. Mike has authored numerous publications, holds 9 US patents, and has 8 patents pending. Mike is a Senior Member of the IEEE.

Mikhail Smelyanskiy (Member, IEEE) received the Ph.D. degree from the University of Michigan, Ann Arbor. He is a Senior Research Scientist with the Corporate Technology Group, Intel Corporation, Santa Clara, CA. His research focus is on building and analyzing parallel emerging workloads to drive the design of next-generation parallel architectures.

Brian Nickerson is a 23-year Intel veteran. Brian has worked in the flow analysis, code scheduling, and register allocation phases of Intel compilers, and during his tenure at Intel has written around a million lines of highly-tuned assembly code for Intel architectures spanning all generations of X86 architecture starting with the 80386, as well as Intel's 80960 and Itanium architectures. Focus areas have been video encoding, decoding, and post-processing, semantic video analysis, and scientific kernels.



Victor W. Lee is a Senior Staff Researcher at the Parallel Computing Lab, Intel Corporation. He is a lead researcher in developing Intel's new Many Integrated Core architecture and new applications that take advantages of the many-core architectures. Prior to the current research, he was involved in other Intel processor designs such as Pentium and Itanium processors.

Michael Chuvelev has graduated Moscow Institute of Physics and Technology in 1993 with MS degree. Michael has joined Intel (R) in 2003 as a member Intel (R) Math Kernel Library (Intel (R) MKL) team. Michael has great experience in Linear Algebra software optimizations, he has publications for Intel (R) Technological Journal (2007), Sobolev's International Conference, Novosibirsk (2008). Currently Michael is a member of Intel (R) MPI team.

Pradeep Dubey is a Senior Principal Engineer and Director of Parallel Computing Lab (PCL), part of Intel Labs. His research focus is computer architectures to efficiently handle new application paradigms for the future computing environment. Dubey previously worked at IBM's T.J. Watson Research Center, and Broadcom Corporation. He was one of the principal architects of the AltiVec* multimedia extension to Power PC* architecture. He also worked on the design, architecture, and performance issues of various microprocessors, including Intel® i386TM, i486TM, and Pentium® processors. He holds over 35 patents and has published extensively. Dr. Dubey received a BS in electronics and communication engineering from Birla Institute of Technology, India, an MSEE from the University of Massachusetts at Amherst, and a Ph.D. in electrical engineering from Purdue University. He is a Fellow of IEEE.