Runtime Systems and Out-of-Core Cholesky Factorization on the Intel Xeon Phi System

Allan Richmond R. Morales, Chong Tian, Kwai Wong, Eduardo D'Azevedo

The George Washington University, The Chinese University of Hong Kong,
University of Tennessee, Oak Ridge National Laboratory

Abstract

We will explore how different runtime systems can be implemented on the Intel Xeon Phi System in Beacon. First, we will explore how to utilize PLASMA for handling dense linear algebra computations and QUARK for task management and added parallelism to figure out the dependencies between the tasks and the scheduler. These algorithms will then be tested on the Beacon's MIC for performance analysis and comparison with the Intel MKL implementation. The end goal is to have an optimized runtime system that incorporates QUARK threading and management with the MKL BLAS routines. Another goal is to implement a hybrid Out-of-Core algorithm for Cholesky factorization that can be used in conjunction with the PLASMA/QUARK implementation to see if its performance is efficient and scalable.

I. Background

Beacon is a supercomputer at the Oak Ridge National Laboratory (ORNL) that has 48 compute nodes. Each node consists of the following: two 8-core 2.6 GHz Intel Xeon E5-2670 Processors with 256 GB RAM and four 1.053 GHz 60-core Intel Xeon Phi 5110P Coprocessors with 8 GB RAM, which has a Many Integrated Core (MIC) architecture. Each has its own tradeoffs, but the former has 8x more memory and the latter has more computational power since it has more cores than the processors. All these coprocessors are connected to an Intel Xeon processor (the host) via a PCI bus.

Within Beacon, there are different modes of execution for Beacon. Host mode is the standard execution through the host processor. Native mode utilizes the coprocessor as an independent compute node and runs the execution only on the MIC card. Any libraries used must be recompiled for native mode with the compiler flag "-mmic". To achieve parallelism across cores, threads will be used. And in offload mode, code will run on the host. If there are specified sections of code for parallelism, pragmas and directives can be used to offload into the MIC.

The Intel Xeon Phi system support a number of runtime systems written in C, C++, and Fortran. A runtime system is an integral part for program execution since every programming language has some form of this system. Therefore, one purpose of this research is to focus on a compiled language (C) as well as application programming interface (API) calls to QUARK, PLASMA, and the Intel MKL libraries to provide performance analysis on Beacon.

To test the performance of these runtime systems, some routine must be implemented that can generate a large number of floating operations per second (GFLOPS/sec). Therefore, matrix manipulation using dense linear algebra algorithms is ideal since matrices are efficient in calculating and storing data. Some routines that will tested include nested matrix multiplication, DGEMM, and Cholesky Factorization. Intel's MKL library has been optimized on the Xeon Phi System; therefore, the data collected will be used as the benchmark for comparison with the other tested runtime systems.

Another purpose of this research is to explore an Out-of-Core (OOC) algorithm for Cholesky Factorization using QUARK, which will later be discussed in the report. From a hardware perspective, this is an optimized al-

gorithm that will take full advantage of the Intel Xeon architecture. With more memory in the processor, larger matrices can be supported. Then breaking the matrices into sub-matrices (tiles), they can be offloaded into the MIC, which will have more computational power.

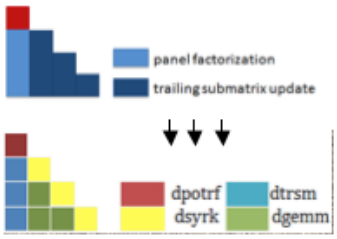


Figure 2: Tile Representations of Cholesky steps

II. Tile Cholesky Factorization

Cholesky factorization is the decomposition of a positve-definite matrix into the product of a lower triangular matrix and its conjugate transpose, as shown in Figure 1. This dense linear algebra algorithm can be broken down into the following steps:

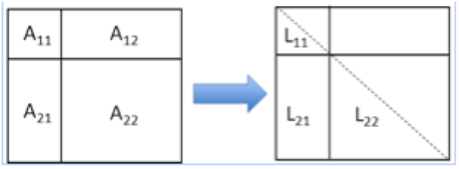


Figure 1: Visual Representation of Steps

- Step 1: L11 <– cholesky(A11)
- Step 2: L21 <– A21 / L11' <Panel Factorization>
- Step 3: A22 <– A22 – (L21 * L21') <Trailing Sub-matrix Update>
- Step 4: L22 <– cholesky(A22)

Solving linear system Ax=b is vital is scientific computing. When A is symmetric positive definite, Cholesky factorization can be applied in which A = L* L', where is L is a lower triangular matrix. Then using forward and backward substitution on L * L' * x = b can make it easier to solve Ax=b. For a large-scale matrix, implementing Cholesky factorization will operate on many sub-matrix blocks of "A".

The utility of blocking promotes data locality and reuse in the faster and smaller levels of the memory hierarchy. However, the panel factorization step consists of matrix-vector operations (level 2 BLAS operations), which cannot be parallelized efficiently in shared memory machines. The trailing matrix update operations (matrix-matrix operations or level 3 BLAS operations),which are highly parallelizable, must wait for completion of the panel factorization. Thus, the behavior of Cholesky factorization can be described as a sequence of serial operations followed by parallel ones periodically. This results in a fork-join style execution,where scalability is limited. Therefore, to gain more parallelism, we choose to perform tile operations on the large-scale matrix, which breaks the panel factorization and trailing sub-matrix update steps into smaller tasks.Then these tasks can be scheduled dynamically by the runtime system. In such an asynchronous execution, sequential tasks can be hidden behind parallel ones.

From this abstraction, the following pseudocode conveys the behavior of such blocking:

```
for k=0...n-1
    for j=k...n-1
        for i=j...n-1 {
            if (i=j&&j=k)   dpotrf (A(i,j)INOUT)
            if (i>j&&j=k)   dtrsm (A(k,k)IN, A(i,j)INOUT)
            if (i=j&&j>k)   dsyrk (A(j,k)IN, A(i,j)INOUT)
            if (i>j&&j>k)   dgemm (A(i,k)IN,A(j,k)IN,A(i,j)INOUT)
        }
```

To perform this tiled routine, the following four BLAS subroutines are required:

- **dpotrf** - Cholesky factorization

- **dtrsm** - triangular solve:
  Solve op(A)*X = alpha*B, or X*op(A) = alpha*B

- **dsyrk** - symmetric rank k operations:
  C = alpha*A*A′ + beta*C

- **dgemm** - general matrix-matrix operations: C = alpha*op(A) * op(B) + beta*C

Given the logic of this pseudocode, the next approach is using the appropriate runtime system to begin implementing the tile Cholesky factorization. For this research, the goal is using QUARK and the following snippet of code utilizes the QUARK API to configure DGEMM:

```
void CORE_dgemm_quark(Quark *quark); //body omitted
void QUARK_CORE_dgemm(Quark *quark, Quark_Task_Flags *task_flags, PLASMA_enum transA,
PLASMA_enum transB,int m, int n, int k, int nb,double alpha, const double *A, int lda,const double
*B, int ldb,double beta, double *C, int ldc); //body omitted
.....
    if((j>k)&&(i>j)) //dgemm type:(i,j,k),where(i>j>k
    {
        Quark_Task_Flags  tflags=Quark_Task_Flags_Initializer; //initallize the task
        QUARK_Task_Flag_Set(&tflags,TASK_PRIORITY,1);//set the task attributes like priority

QUARK_CORE_dgemm(quark,&tflags,CblasNoTrans,CblasTrans,NB,NB,NB,NB,-1.0,&A2(0,0,i,k),NB,&
A2(0,0,j,k),NB,1.0,&A2(0,0,i,j),NB);  // pass the arguments ,where data dependencies are implied
        continue; }
```

III. Task Directed Acyclic Graph

The next step is creating a directed acyclic graph (DAG), a visual representation using Graphviz in which a node can represent an operation on a matrix tile and an edge can represent the data dependency.

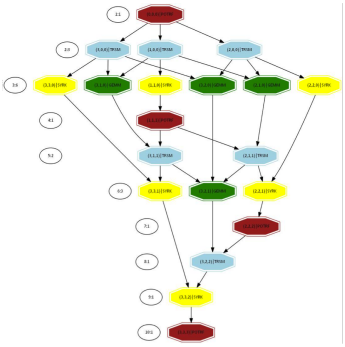


Figure 3: 4x4 Cholesky Factorization Case

Once the DAG is produced and fed into the runtime system QUARK, tasks should be scheduled asynchronously and independently as long as the dependencies are not violated. In the DAG, a critical path can be identified by connecting nodes having more outgoing edges. And the tasks on the critical path are the most important. Once it is done, it will free up more available parallel work. Within the programming, DPOTRF can be assigned higher priority within the QUARK runtime system.

To generate the structure in Figure 3, the following code configures the node and color assignment:

```
struct Label{long I;long J;long K;};
struct List{long node;label Node;char type;label in[3];label out[n-1];};
.....
if((j>k)&&(i>j)) //dgemm type:(i,j,k),where(i>j>k
{
    list[count].Node=assignlabel(i,j,k);    list[count].node=(i+1+j*n)+k*n*n;    list[count].type='M';
    fprintf(fp,"%ld[label=\"(%ld,%ld,%ld)|GEMM\",color=forestgreen];\n",list[count].node,i,j,k);
    //assign node attributes like label,color and so on
    for(q=0;q<3;q++)//Traverse the in-nodes and specify the data dependencies by edges
    {
        if (!((list[count].in[q].I==-1)||(list[count].in[q].J==-1)||(list[count].in[q].K==-1)))
        fprintf(fp,"%ld->%ld;",(list[count].in[q].I+1+list[count].in[q].J*n+list[count].in[q].K*n*n),
    list[count].node);
     }
        fprintf(fp,"{rank=same;depth%ld %ld}\n",(3*k+3),list[count].node); //mark the depth
        .....
}
```

IV. Out-of-Core (OOC) Algorithm

Given the standard Cholesky factorization, the OOC algorithm is a hybrid method to perform Cholesky factorization. It distributes the left-looking part to CPU (out-of-core) and the right-looking part to GPU/MIC (in-core). The OOC algorithm has only been considered in recent times because more research has discovered that coprocessors such as the GPU and MIC are much faster and energy efficient when performing high-level computations such as matrix-matrix multiplication.

However, there are some tradeoffs. First, they have relatively small device memory, which restricts the problem size. To fix this problem, the idea of OOC algorithm is to bring in small pieces of matrix and perform most of the heavy computation loads on the coprocessor and then write the data back. So it takes

advantage of the computational efficiency of hardware accelerators without limiting the size of the matrix problem. Second, the data movement between CPU and coprocessor is expensive, which means we need to optimize the amount of data we bring into the device each time.

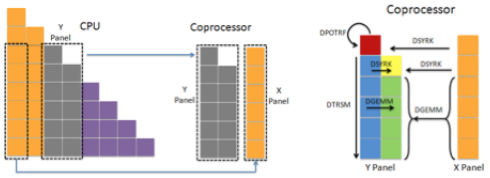


Figure 4: OOC Structure of a 8 x 8 example (left: out-of-core; right: in-core)

The structure of this hybrid algorithm can be broken down into the *out-of-core* and *in-core* part, as shown in Figure 4. For the out-of-core part, the load part of the matrix is sent to the device memory(Y panel) while the update is applied from the part already factorized. Second, the in-core part will factorize the sub-matrix residing on device memory. As a result, combining the OOC algorithm and general Cholesky factorization can yield a theoretical DAG like Figure 5.

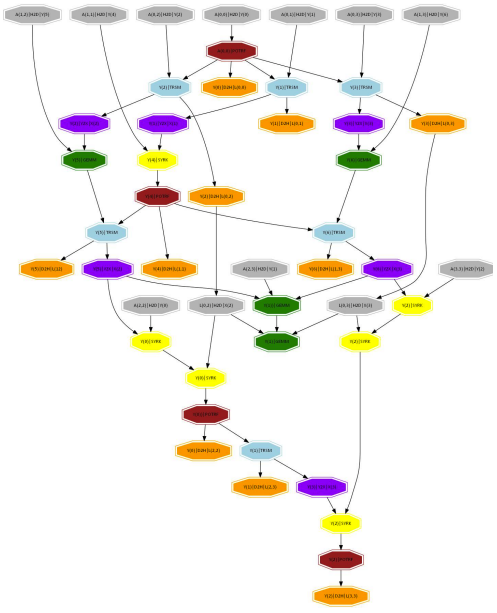


Figure 5: OOC DAG

For a better abstraction, the pseudocode can be stated as the following:

```
/*Out of core part:(starting from the A(k,k) tile)*/
    /*O1.Send in Y-panel*/
    for j=k:1:k+sizeY-1
        for i=j:1:n
            H2D_Copy A(i,j) -> Y(i,j)   /*Expected optimization 1 */
    /*O2.Left looking update,if not the first Y-panel*/
        /*Send factorized columns into X panel*/
        for j=1:1:k-1
        {
            for j=k:1:n
                H2D_Copy L(i,j)->X(j)
            /*Expected optimization 2*/
            for q=k:1:k+sizeY-1
                    for p=q:1:n
                            if(p==q) dsyrk(Y(p,q),X(p))
                            else dgemm(Y(p,q),X(p),X(q)^T)}
            /*Expected optimization 3*/
/*In core part :basically similar to the general Cholesky
factorization,except there are extra data movements,
especially from Y panel to X panel or to CPU*/
/*Expected optimization 4 */
```

V. Expected Optimization Analysis

Expected optimization 1 - Squeeze more data into Y: Note that right bottom of the lower part of A shrinks in each new iteration(the purple part in Figure 5 left). For the same amount of space dedicated to the Y panel,which is usually N*isize Y, we may send more tile columns into device each time.This requires a much more complicated correspondence between A and Y than simply copying A(i,j) to Y(i,j).

Expected optimization 2 - Fast copy form Y to X: On device memory, copying one array to another is fast. For the last tile column on Y panel, it can be directly copied to X instead of writing it back first.

Expected optimization 3 - Double buffering: Use two X panels for update steps:while one panel is doing DGEMM(),the other can be reading data concurrently.If the time needed for DGEMM operations is very close to the time needed for transferring data, the benefit can be substantial because the total time is reduced to about the half.

Expected optimization 4 - Perform DPOTRF() on CPU: When doing irregular computations like small scale Cholesky factorization, the single core on MIC is slower than the CPU. So it is wiser to let CPU perform the DPOTRF() but perform most of the regular large scale matrix-matrix operations on the device.

VI. Future Goals for OOC Implementation

As of now, the process of combining the OOC and the general Cholesky factorization to be used with QUARK is incomplete. Nevertheless, there is enough fundamental data and abstractions for this process to be finished in the near future. Another future goal would be extending the current single MPI process code to multiple MPI processes version. And once this OOC algorithm has been successfully tested for correctness and optimization in its execution, these principles and methodologies can be applied to other dense linear algebra algorithms such as LU factorization with pivoting and QR factorization.

VII. Overview of Runtime Systems

Besides QUARK implementation of Cholesky on the Intel Xeon Phi System, another area to explore is how well other runtime systems perform on the Intel Xeon Phi system. By measuring the GFLOPS/second of various matrix-multiplication routines, the performance of programming environments such as QUARK, PLASMA, and the Intel MKL library can be tested and compared.

QUARK stands for QUeuing And Runtime for Kernels. QUARK provides a number of libraries with sets of instructions to enable dynamic execution of tasks with data dependencies in a multi-core and multi-socket shared-memory environment to attain a high utilization of available resources. QUARK is scalable for large numbers of cores, which is ideal for high performance computing. The goal with using QUARK is to coordinate a sequence of tasks that are submitted to a set of resources, i.e. the MIC cards. This task can be a BLAS or MKL routine. Next, the tasks will be assigned to a QUARK worker thread, which will send it to a MIC and then wait for the completion of the tasks. After a task is completed, the QUARK worker thread will pick up the next available task to send to its MIC. These tasks could be a BLAS/MKL routine.

PLASMA stands for Parallel Linear Algebra for Scalable Multicore Architectures. PLASMA provides a scalable and efficient environment for dense linear algebra applications and high performance computing. Within its libraries are a number of function calls such as Cholesky (potrf) and matrix multiplication (gemm) that have defined algorithms to manipulate matrices. The latest version (2.6.0) has been installed as a module onto Beacon. With QUARK, a number of PLASMA-defined linear algebra function calls will be wrapped and then implemented directly on the Beacon MIC cards.

And the Intel Math Kernel Library (MKL) includes functionalities from BLAS, LAPACK, SacLAPACK, and other dense linear algebra algorithms. Because these MKL functions have been optimized for the Intel architecture, they can serve as benchmarks for normal and optimized test cases.

VIII. Breakdown of Performance Testing

The following routines were tested in their respective programming environment:

- QUARK Multi-threaded Tiled Matrix Multiplication
- PLASMA Tiled DGEMM
- Intel MKL DGEMM
- Intel MKL SPOTRF (Cholesky Factorization)

For matrix multiplication and DGEMM, the equation in Figure 6 will be used. For Cholesky factorization, the equation in Figure 7 will be used.

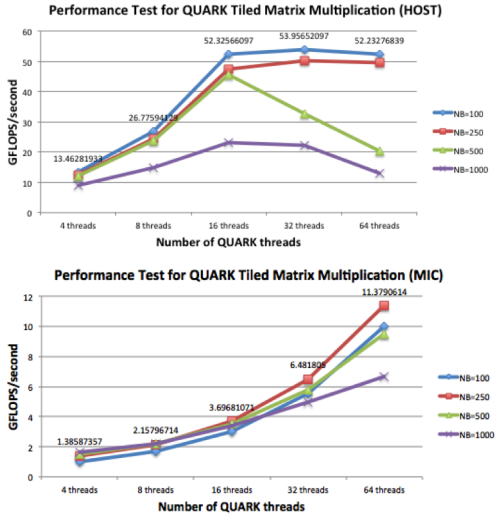
$$\frac{2n^3}{10^9 * time\_avg} \qquad \frac{\frac{1}{3}n^3}{10^9 * time\_avg}$$

Figures 6 + 7: Respective GFLOPS/second Equations

For "n", an assumption is made that matrices A, B, and C are symmetric, i.e. n = m = k. And using the average time provides more accuracy by running multiple iterations of the routine, summing their execution times, and then divide by the number of iterations.

IX. Results

First, QUARK was implemented to test a multi-threaded tiled routine for matrix multiplication on the host and the MIC. The matrix was divided into tiles and the calculations were done in parallel based on the number of available QUARK threads.



Figures 8 + 9: QUARK Performance Test for Host and MIC

	NB=100	NB=250	NB=500	NB=1000
4 threads	13.46281933	12.5576024	12.173018	9.01319071
8 threads	26.77594129	24.3421548	23.656976	14.8945193
16 threads	52.32566097	47.4777321	45.76333371	23.1449664
32 threads	53.95652097	50.2472455	32.62076229	22.262155
64 threads	52.23276839	49.5421097	20.25220514	13.0737957

	NB=100	NB=250	NB=500	NB=1000
4 threads	0.99663077	1.38587357	1.496806	1.63284111
8 threads	1.70272846	2.15796714	2.21691933	2.22034778
16 threads	3.03245308	3.69681071	3.537532	3.36262222
32 threads	5.5189	6.481805	5.77946333	4.94149778
64 threads	9.96874769	11.3790614	9.487648	6.68409111

Tables 1 + 2: Numerical Data from QUARK Performance Tests

The range for the matrix size was tested between 500 and 15000. The data collected proved to be relatively constant throughout all matrix sizes; therefore, these values were averaged and the performance graph was based on the number of QUARK threads.

First, the data collected from the host demonstrates that using a smaller number of threads can yield a better performance output. Though 32 threads is favored at smaller tile sizes, the difference in performance between 16 and 32 threads is small; thus, it can be concluded that optimal performance when increasing the tile sizes can be attained using 16 threads. Using too many threads will reduce the performance output due to more blocking and overhead. Another important point is the inverse relationship between the performance and tile sizes. Given 16 threads, the best output of approximately 54 GFLOPS/second is collected from the smallest tile size of 100. However, as the tile size increases, the performance decreases. And this trend exists over all trials of different number of QUARK threads.

As for the data collected on the MIC, a different trend exists. Optimal performance is achieved across different tile sizes using the maximum number of QUARK threads defined, and these performance readings are significantly decreased compared to those on the host with the maximum output being five times less than that on the host.

One final observation is this QUARK Tiled Matrix Multiplication routine is not optimized for either the Intel Xeon processor as well as the MIC architecture as the maximum performance output is only a small percentage to the peak performance expected (approximately 1011 GFLOPS/second); nevertheless, preliminary tests provided by the University of Tennessee's Star1 cluster only yielded approximately 1 GFLOPS/second, given significantly less computational power and memory. Therefore, one approach for improvement can be utilizing the offload functionality to take advantage of the host's memory and the MIC's computational power. Another approach worth testing is taking advantage of the hyper threading given that Beacon supports 4 MIC per compute node, each having 60 cores and thus providing a total of 240 threads.

Given that the nested-for loop matrix multiplication approach, the next step is finding a more optimized approach. And according to Intel, the typical benchmark for performance testing is with the DGEMM routine. Therefore, PLASMA is first tested since it has already been installed as a module in the Intel MIC architecture. For comparison purposes, another PLASMA programming environment was installed on the host processor.

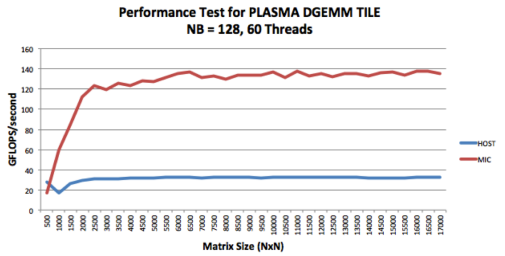


Figure 10: PLASMA Tiled DGEMM Performance Test

First, running the default PLASMA test given a tile size of 128 and 60 threads will provide a benchmark for performance on the MIC and on the host. From this data, it can be concluded that the MIC performs more than 4x better than on the host (138.14 and 32.32 GFLOPS/second respectively). Given this benchmark, the next approach is testing what factors can be changed in order to gain more performance output. One consideration was testing the impact of increased number of threads; however, the data proved to be insignificant as the performance readings on both the host and MIC did not change significantly; especially on larger matrix size, the output yields a similar threshold. Therefore, the other option was changing the tile sizes, and this data proves to provide some insight.

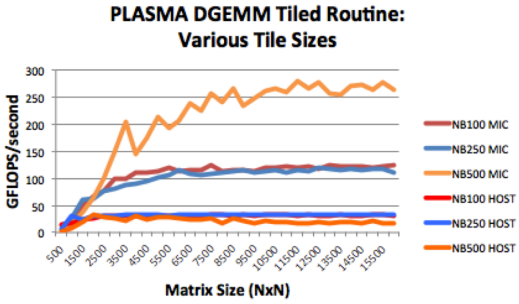


Figure 11: PLASMA Tiled DGEMM Test with Different Tile Sizes

On the host, changing the tile size (NB) on the host proves to be insignificant. Though it is important to note that the performance slightly decreases. However, the opposite trend exists on the MIC. Changing the tile size to the max defined does provide the most performance boost. From a numerical standpoint, the MIC implementation is more than eight times faster than the Host implementation (279.89 and 32.88 GFLOPS/second respectively). With this approach, this implementation of DGEMM was able to reach 30 percent of the Intel MIC's theoretical peak performance (1011 GFLOPS/second).

Given this experimental data, it can be compared with the Intel MKL benchmark results for DGEMM. Because this routine has been optimized for the Intel architecture, it has been advertised that DGEMM can reach up to 833 GFLOPS/second and greater. This performance test was recreated. Before doing so, a generic performance test based on the modes of execution was performed.

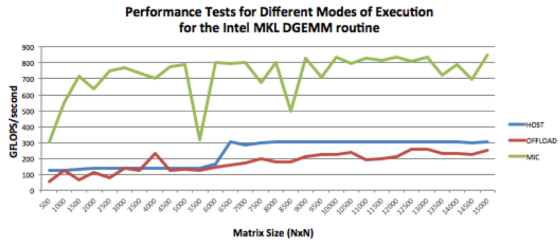


Figure 12: Intel MKL DGEMM Performance Test: Modes of Execution

According to these results, the MIC yields the greatest performance output overall, which is expected since the MIC provides more computational power. And the host only reaches a threshold of approximately 300 GFLOPS/second. The data that proves questionable is the Offload execution, which barely peaks past approximately 250 GFLOPS/second, because this mode should take advantage of both the host and MIC architecture. Therefore, a theoretical plot for Offloading should yield a performance output greater than that from the host processor. Nevertheless, these results convey that the MIC can performance values even greater than 833 GFLOPS/second.

Given this data, the next approach is reaching this proposed performance output, which states that at a matrix size of 7680, the output can reach 833 GFLOPS/second. Before running these tests, certain MIC environment variables need to be configured. The first variable is OMP NUM THREADS, which is based on the hyper-threading incorporated within the MIC architecture. Each MIC has 60 cores; and since Beacon has four per node, the maximum number of threads available is 240. The second variable is KMP AFFINITY, which is the scheduling and organization of the hyper-threads. There are three different assignments: compact, balanced, and scatter.

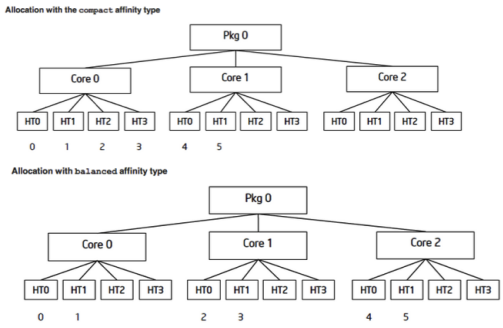


Figure 13: KMP AFFINITY variables

Setting KMP AFFINITY to "compact" allows for sequential queueing as shown in Figure 13 in which an example is given with three cores and 6 threads allocated. Setting KMP AFFINITY to "balanced" allows for threads to be allocated evenly among the cores as shown in Figure 13. With this setting, cache utilization is enabled so that there is much less overhead for memory transfer. And setting the KMP AFFINITY to "scatter" allows for threads to be allocated in an arbitrary manner. Setting these MIC environment variables are crucial in order to achieve any speedup. Hence, the first performance test involved changing the values for the OMP NUM THREADS variable.

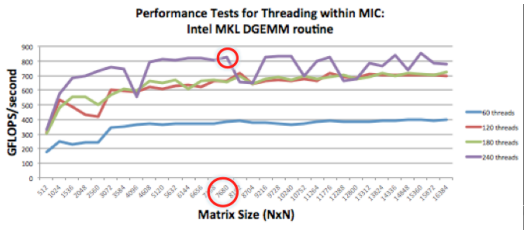


Figure 14: Advertised Intel MKL DGEMM Performance Test

From this data, it can be concluded that using the maximum value available on a single compute node (240 threads) will yield the best output; and when the matrix size is 7680, the performance output reaches a value very similar to the advertised value (833 GFLOPS/second). Therefore, the next step is modifying the distribution of the hyper-threads to confirm if the same results can be attained.

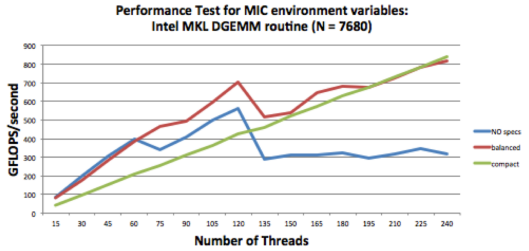


Figure 15: Intel MKL DGEMM MIC Environment Variable Performance Test

In Figure 15, the test focused on the matrix size of 7680, and the steps were determined based on the number of threads in order to create plots that reflect the value of the KMP AFFINITY variable. As a result, it can be concluded that this variable must be configured in order to reach this optimal performance output. With no specifications, the performance can reach up to 550 GFLOPS / second, but the overall performance reaches only half the expected optimal performance output. Once the KMP AFFINITY variable was set to "compact" or "balanced", the expected output was achieved, though their plot lines differ. For the former, the plot is linear and conveys a direct relationship between the number of threads and the performance output. For the latter, the performance plot resembles that from the

plot with no specifications; however, it yielded greater values at smaller threads that that from the the "compact". Therefore, in order to fully utilize the MIC architecture to get this advertised value of roughly 833 GFLOPS / second, the environment variables must be set so that OMP NUM THREADS=240 and KMP AFFINITY=balanced.

Understanding how to test the Intel MKL routines, the same testing environment will be applied to test Cholesky factorization in order to provide a benchmark for future testing, specifically for the QUARK implementation and the Out-of-Core algorithm to optimize this routine. For this test, the single precision routine for Cholesky factorization (SPOTRF) was tested. The first performance test will compare the available modes of execution.

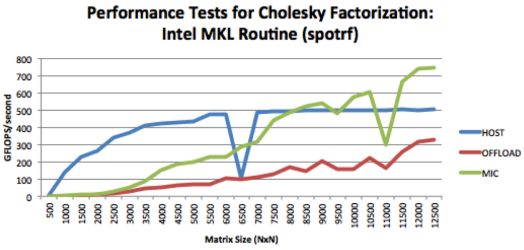


Figure 16: Cholesky Factorization (SPOTRF) Performance Test

Given these results, the MIC architecture proves to provide the most performance output at approximately 745 GFLOPS/second. For the host execution, it reaches a threshold of approximately 500 GFLOPS/ second once the matrix size reaches 6000 and more. For the offload execution, the data reaches a maximum output of 300 GFLOPS /sec but the overall plot proves to be questionable. This flaws plot may result from incorrectly timing the offloading or an incorrect implementation of the offloading routine. Nevertheless, it is important to note that the MIC only reaches greater performance output values when the plot reaches certain intervals of 9000 and 10,000 and 11500 onward. Therefore, it can be concluded that at smaller matrix sizes up to 8000, the host implementation performs better. At larger matrix sizes past 8000, the MIC implementation performs better. Nevertheless, because the MIC can reach a higher maximum performance output than the host, the next approach is testing how modifying the MIC environment variables can affect the performance output.

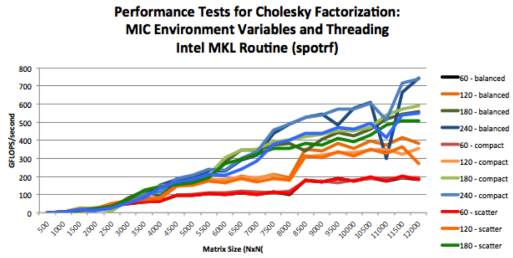


Figure 17: Cholesky Factorization (SPOTRF) Performance Test: Comparison of Different MIC configurations

For this graph, the plots are based on the number of hyper threads and changing the organization of the cores. Given this data, the ideal configurations to get the optimal performance output of approximately 745 GFLOPS/second is setting the OMP NUM THREADS=240 and KMP AFFINITY=compact. The second best setting proved to be OMP NUM THREADS=240 and KMP AFFINITY=balanced. These two plots are very similar and overlap up to a matrix size of 9000. When the matrix size is increased past this value, there are fluctuations that cause the balanced setting to decrease in its performance. These fluctuations may be caused by overhead from checking all 240 cores that may not be completely occupied rather than checking through fewer cores that will be completely occupied. Nevertheless, these results convey that the all the available resources within the MIC are being utilized in order to get the best performance output.

X. Future Goals for the Runtime System

A number of performance tests can have been conducted based on certain matrix manipulation routines on the MIC and host; but for this particular research, there are a number of goals that were not accomplished but can be explored in the future. Having more QUARK

and PLASMA performance tests would provide more stable data for comparison, specifically QUARK DGEMM as well as PLASMA DPOTRF, DTRSM, and DSYRK. A greater exploration of the offloading mode would help verify if it is an efficient mode of execution across a number of routines. And as for the end goal, we were unable to incorporate the OOC algorithm for Cholesky Factorization with QUARK to implement within Beacon. However, we do hope that our preliminary research can provide future interns and researchers a foundation to explore more possibilities of optimizing runtime systems with the Intel Xeon Phi System.

XI. Thinking about the Future: Documentation

One of the major obstacles in this research is the lack of documentation for inexperienced researchers and interns. As a result, a substantial amount of time was spent setting up these programming environments rather than coding and testing routines. Therefore, I have provided some documentation that will provide newcomers with all pertinent information about execution with Beacon, PBS implementation, how to set up a particular environment for Host or MIC performance testing, sample code, and the current progress of the research. In providing this documentation, it will allow the users to quickly configure their programming environment and begin testing code at a faster pace.

XII. Acknowledgements

This research was conducted during the summer research program known as the Computing Science for Undergraduate Research Experiences (CSURE) in Knoxville, Tennessee. This summer program is run by the National Institute of Computational Science (NICS), which is located at the Oak Ridge National Laboratory. We would like to extend our gratitude to our mentors Dr. Eduardo D'Azevedo (ORNL) and Dr. Kwai Wong (UT). Throughout the duration of the program, we have also received assistance from a number of collaborators including Dr. Asim YarKhan (UT), Dr. Shiquan Su (NICS), Ben Chan, and the XSEDE Troubleshooting Support Service.

XIII. References

1. Beacon User Guide. *https://www.nics.tennessee.edu/beacon*
2. Betro, Vincent. *Beacon Quickstart Guide at AACE/NICS.*
3. Betro, Vincent. *Beacon Training: Using the Intel Many Integrated Core (MIC) Architecture: Native Mode and Intel MPI.* March 2013
4. Dongarra, Jack, et al. *PLASMA Users' Guide Version 2.3.* Sept. 2010
5. Hebenstreit, Michael. *Faces of Parallelism: Porting Programs to the Intel Many Integrates Core Architecture.*
6. Hulgain, Ryan. *Intro to Beacon and Intel Xeon Phi Coprocessors.*
7. Kurzak,Jakub. *PLASMA/QUARK and DPLASMA/PaRSEC tutorial: ICL UT Innovative Computing Laboratory.*
8. Oertel, Klaus-Dieter. *ScicomP 2013 Tutorial: Intel Xeon Phi Product Famiy Programming Model*
9. Wong, Kwai. *Parallel Computing: An Oveview, Supercomputers, MPI, OpenMP, and More...*
10. YarKhan, Asim, Jakub Kurzak, and Jack Dongarra. *QUARK Users' Guide.* April 2011