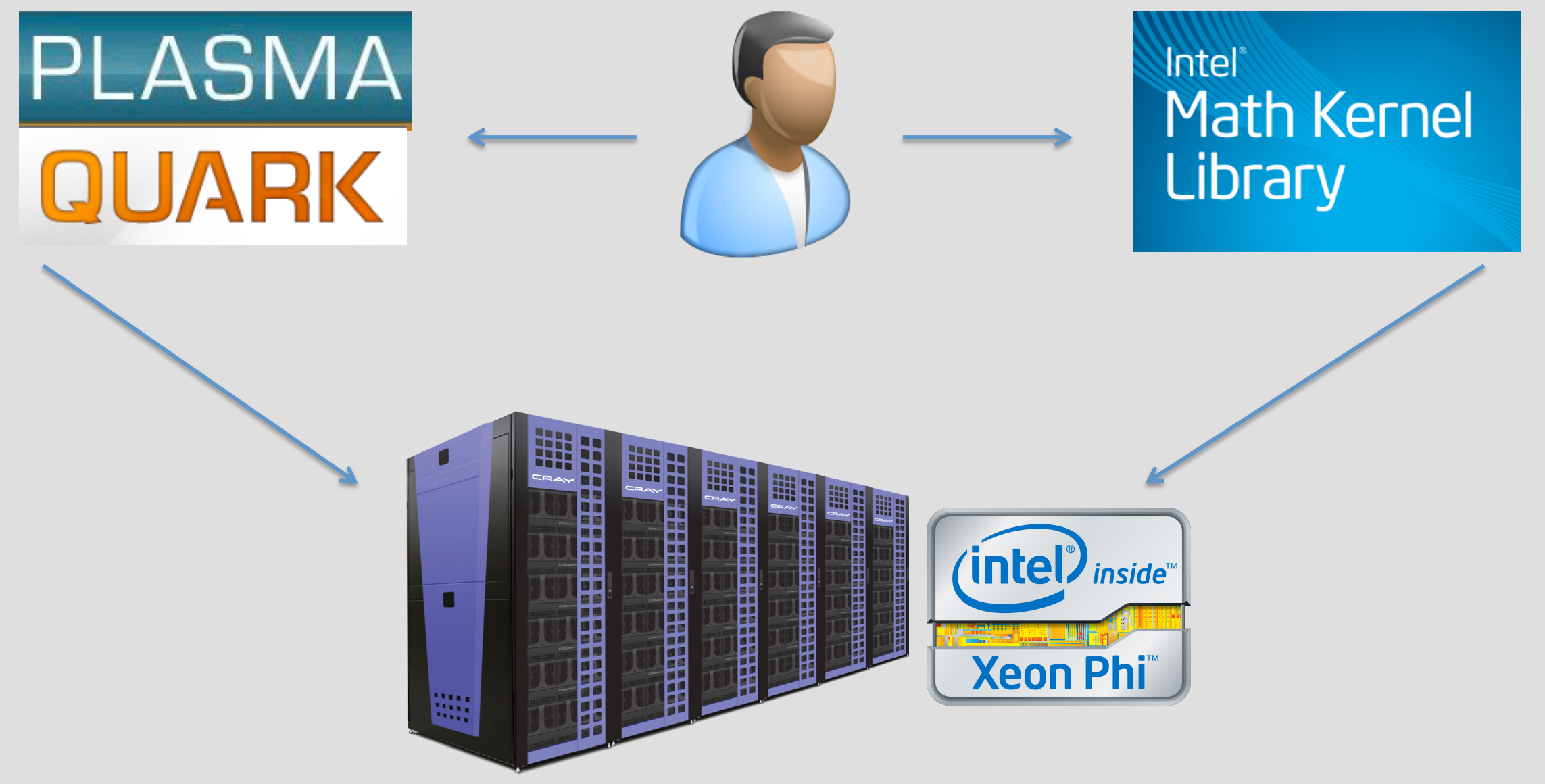
Runtime Systems and Out-of-Core Cholesky Factorization on the Intel Xeon Phi System

Authors: Allan Richmond Razon Morales (The George Washington University), Tian Chong (The Chinese University of Hong Kong) **Mentors:** Dr. Kwai Wong (UTK), Dr. Eduardo D'Azevedo (ORNL)

OBJECTIVE

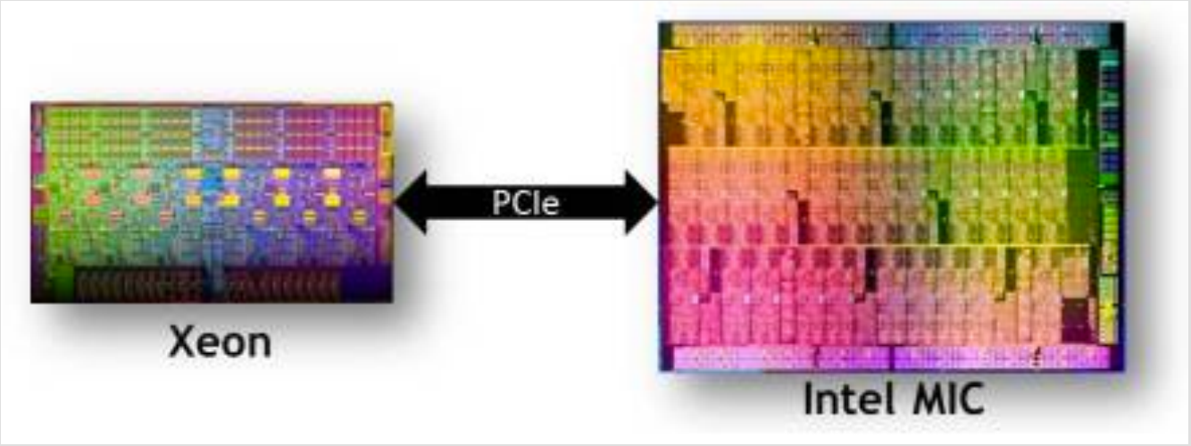
We will explore how different runtime systems can be implemented on the Intel Xeon Phi System on Beacon. This coprocessor does have its own Intel MKL library that implements BLAS and LAPACK functionality. For this research, we will first explore how to utilize PLASMA for handling dense linear algebra computations and QUARK for task management and added parallelism to figure out the dependencies between the tasks and the scheduler. Once accomplished, these algorithms will be rigorously tested on the Beacon's MIC card for performance analysis and comparison with the standard Intel MKL implementation. 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.

VISUAL OF THE OVERVIEW

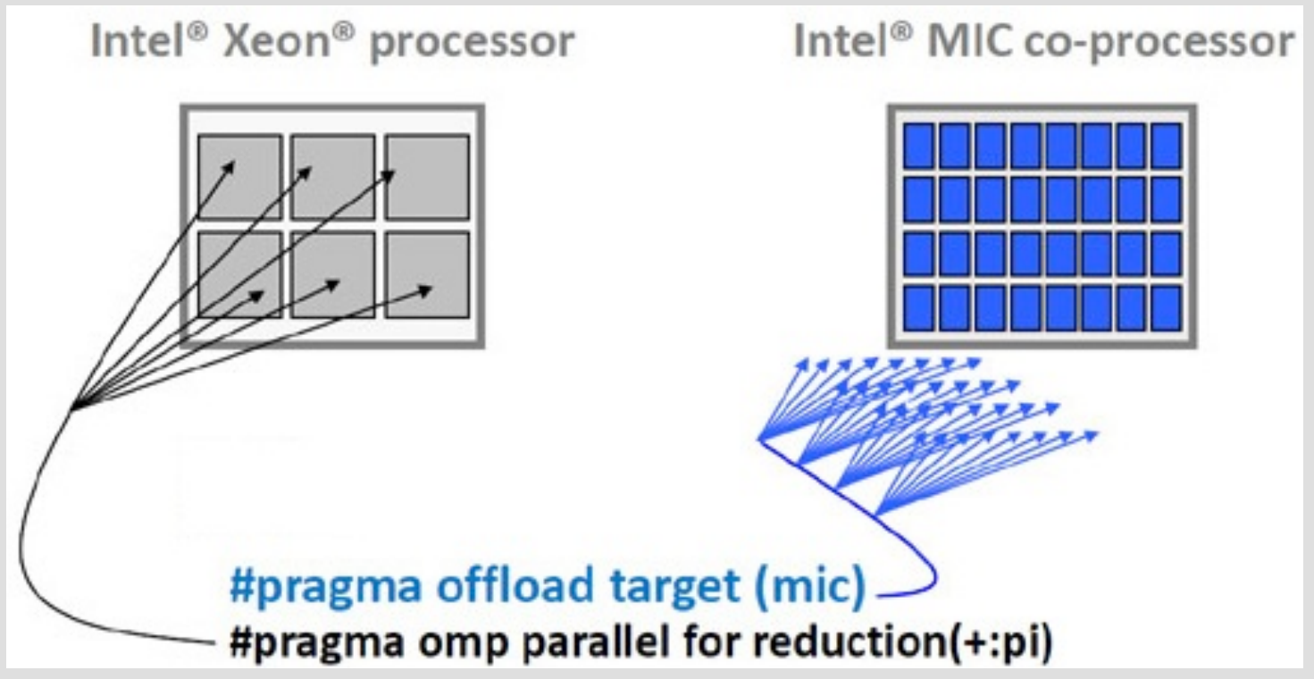


- **PLASMA** – dense algebra algorithms
- **QUARK** – multithreading and task management
- **Intel MKL Library** – optimized math library for comparison

BEACON ARCHITECTURE: INTEL XEON PHI

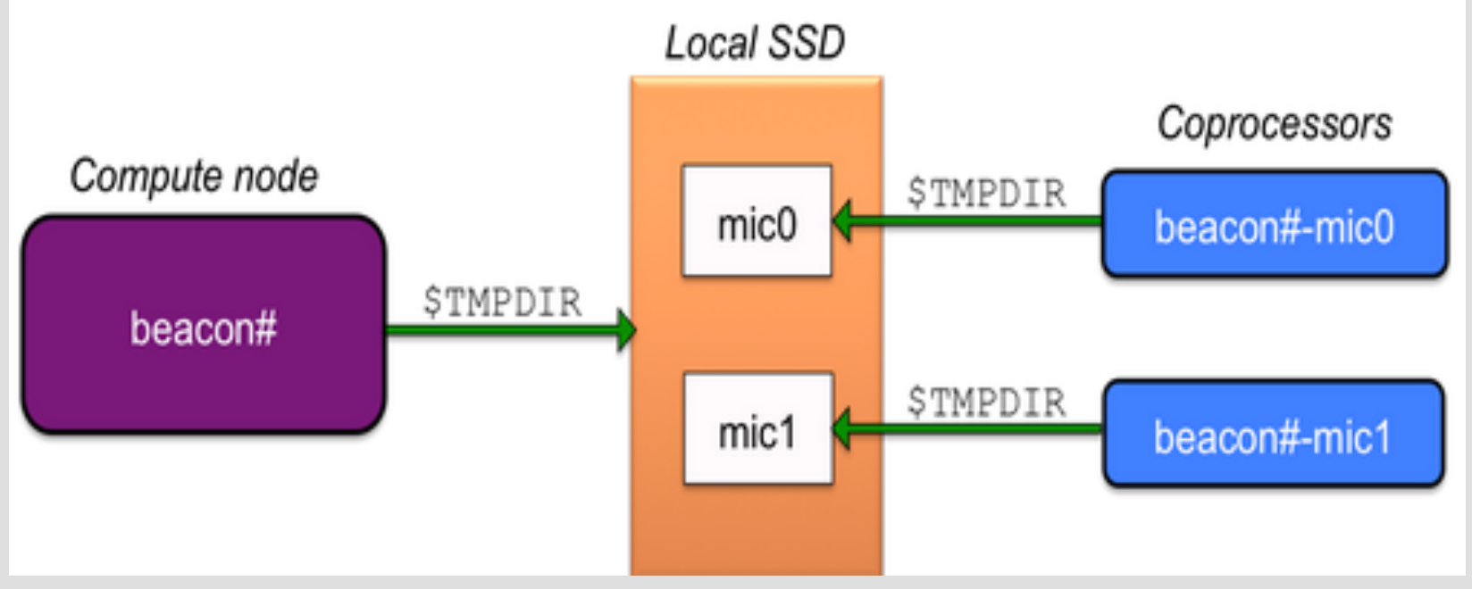


<u>Intel Xeon Processor E5-2670</u>	<u>4 x Intel Xeon Phi Coprocessor 5110P</u>
2 x 8 cores (16 in total per node)	60 cores
2.600 GHz Clock Speed	1.053 GHz Clock Speed
256 GB RAM	8 GB RAM



```
#pragma offload target (mic)
#pragma omp parallel for reduction(+:pi)
```

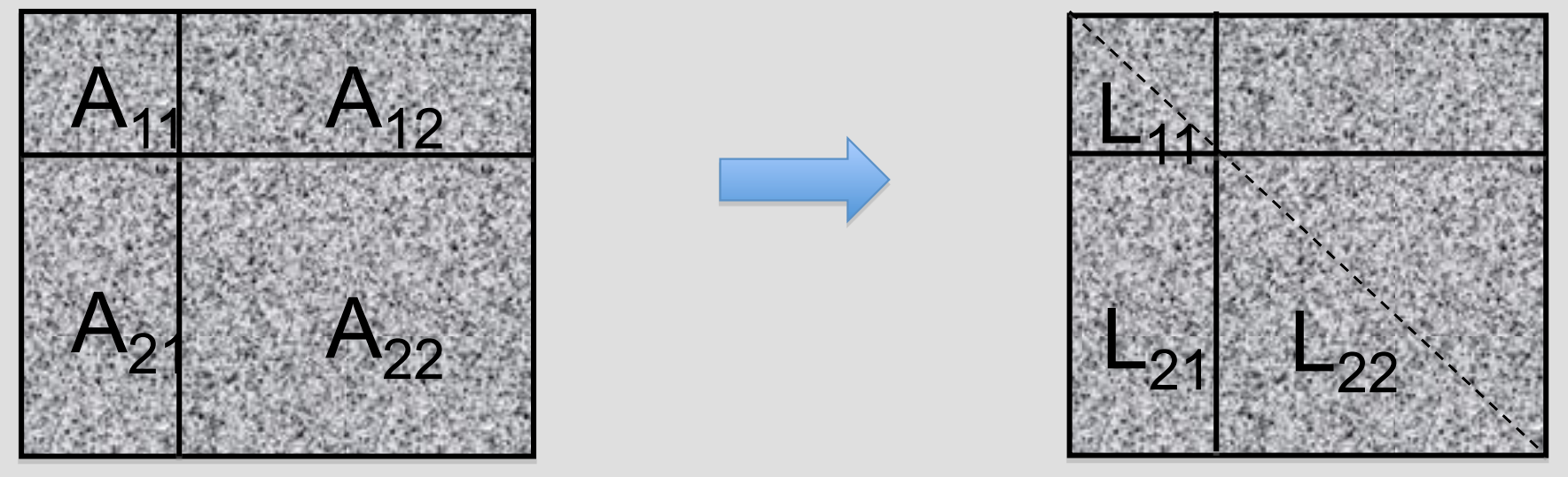
MODES OF EXECUTION



There are two primary modes of execution for Beacon: Native and Offload. The former runs executables directly into the co-processor (MIC). The goal for further optimization is using Offload Mode, which will run on the host processor and "offload" the dense calculations to the co-processor.

CHOLESKY FACTORIZATION

Cholesky Factorization: $A=LL^T$.

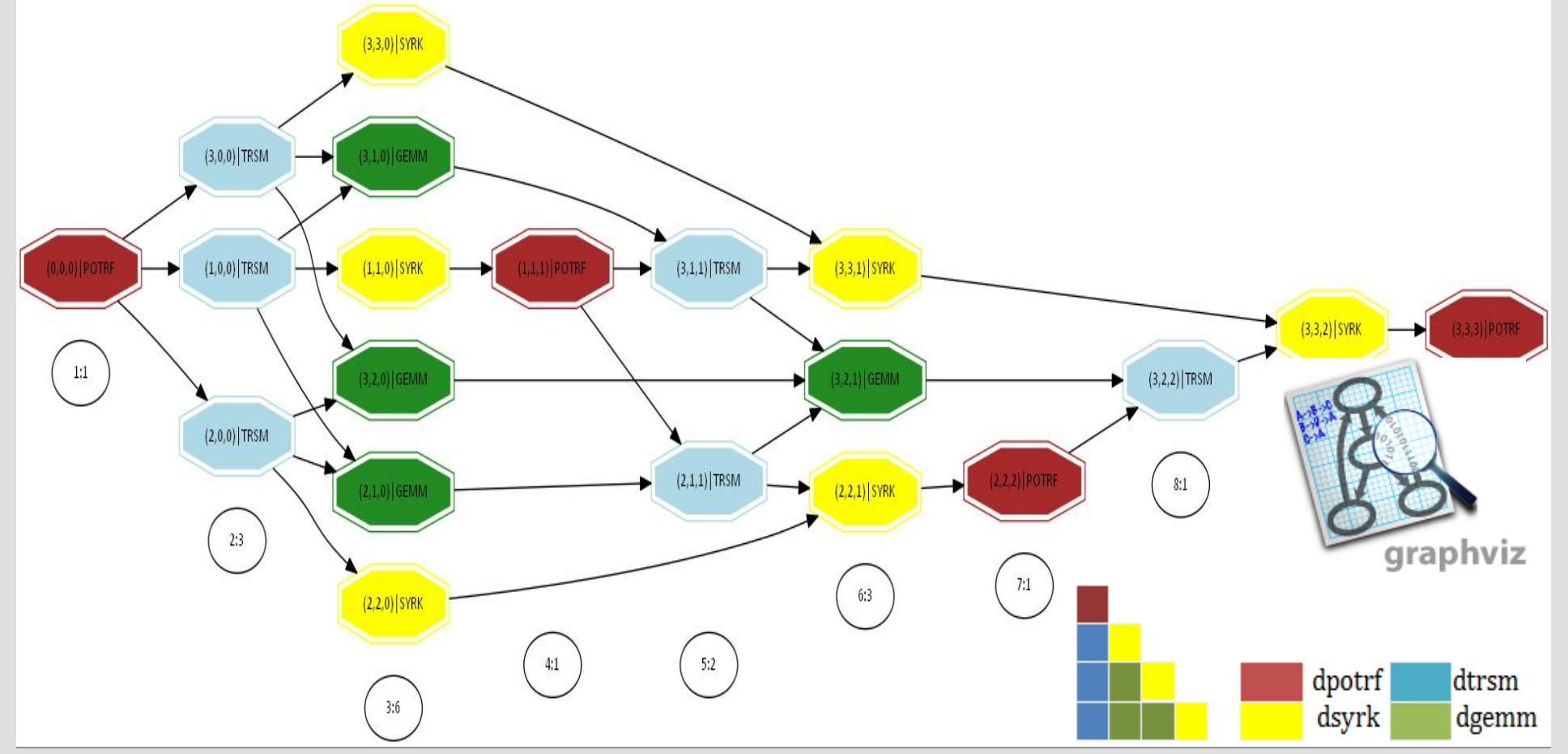


Cholesky steps on matrix blocks

- **Step 1:** L_{11} <-- cholesky(A_{11}) ,potrf()
- **Step 2:** L_{21} <-- A_{21} / L_{11}^T, trsm()
- **Step 3:** A_{22} <-- A_{22} – L_{21} * L_{21}^T, syrk() and gemm()
- **Step 4:** L_{22} <-- cholesky(A_{22}), potrf()

TASK DIRECTED ACYCLIC GRAPH (DAG)

- Tasks in Cholesky factorization depend on previous tasks if they use the same tiles of data. If we use a node to represent an operation on a tile and use an edge to represent a data dependency, then a DAG is formed.
- Once the DAG is produced and fed into the QUARK runtime system, tasks can be scheduled asynchronously and independently as long as the dependencies are not violated. (Eg.4 by 4 case)



Pseudocode for DAG:

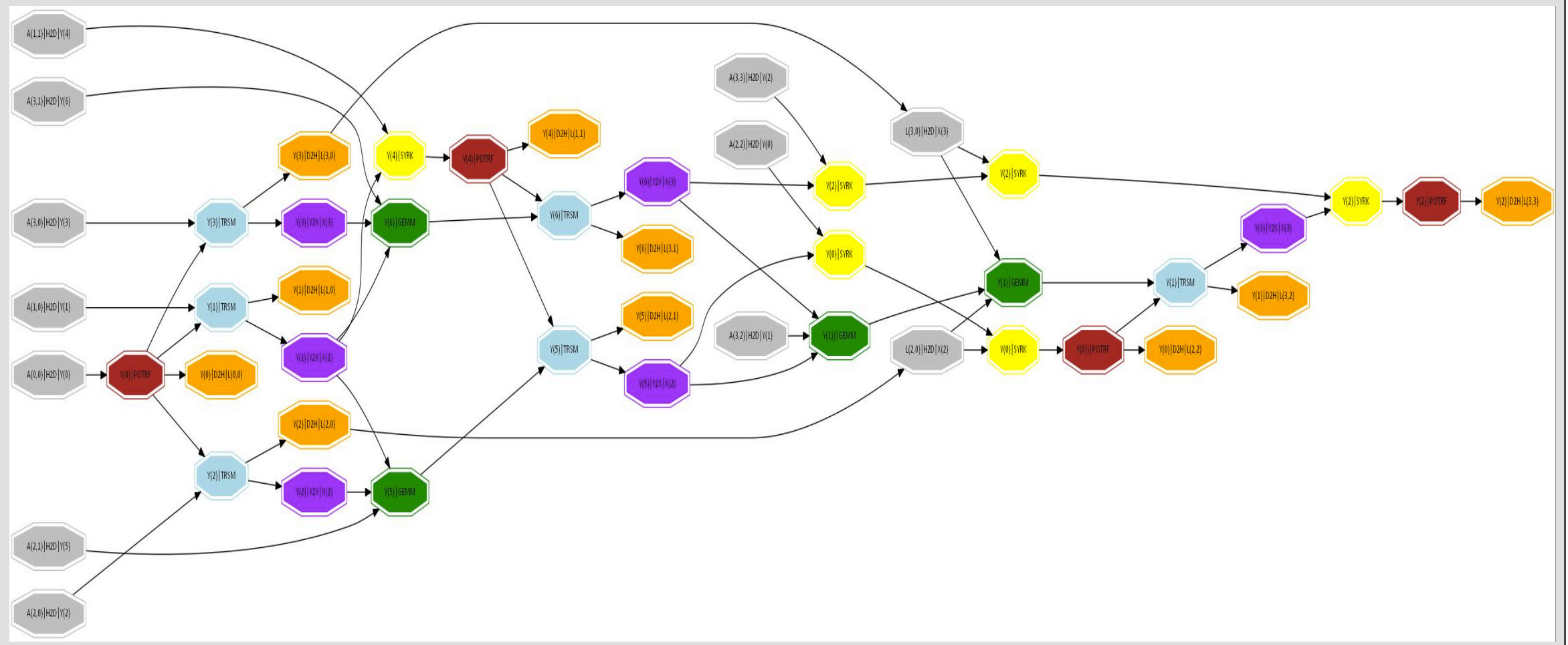
```
for k=0…n-1
   for j=k…n-1
      for i=j…n-1 {
            if (i=j&&j=k)  potrf (A(i,j,k-1)^r, A(i,j,k )^w)
            if (i>j&&j=k)  trsm (A(i,j,k-1)^r, A(k,k,k)^r, A(i,j,k)^w)
            if (i=j&&j>k)  syrk (A(i,j,k-1)^r, A(i,k,k)^r, A(i,j,k)^w)
            if (i>j&&j>k)  gemm (A(i,j,k-1)^r, A(i,k,k)^r,A(j,k,k)^r,A(i,j,k)^w) }
```

OUT-OF-CORE ALGORITHM (OOC)

- OOC stores most data on CPU memory and brings small pieces of data into coprocessors for computation, and then write them back.
- **CPU vs coprocessors(GPU, MIC, etc.)**:GPU is much faster and more energy efficient than CPU but has limited amount of device memory.

OOC STRUCTURE

- The **out-of-core** part loads parts of the matrix. For example, matrix panels,to device memory,and applies the "left-looking" update from the parts already factorized and written back.
- The **In-core** part factorizes the parts residing on device memory in which "right-looking" update is involved.
- Out-of-core Cholesky DAG: (Eg.4 by 4 case)



PROPOSED METHODOLOGY

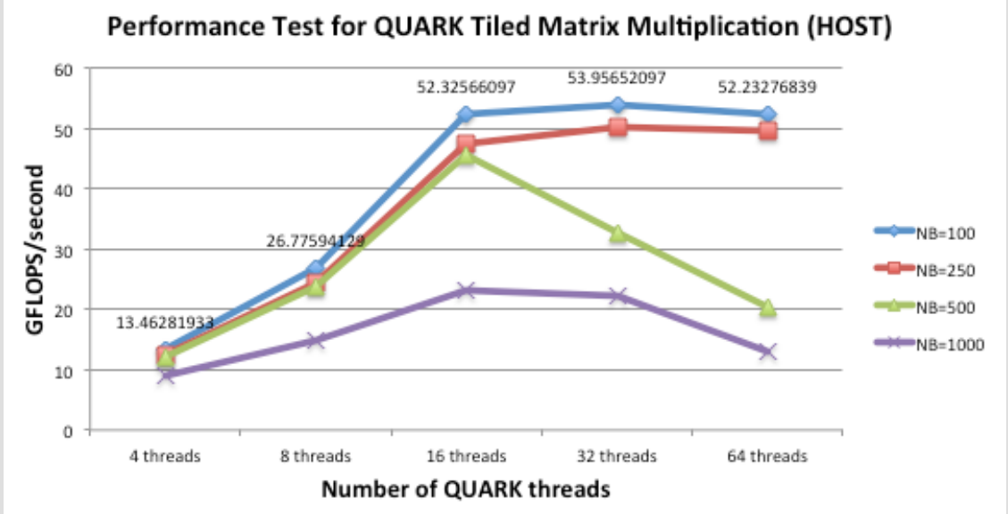
Performance Testing in seconds, GFLOPS, GLOPS/sec
("Giga Floating Operations Per Second")

1. **Nested-For Loop Matrix Multiplication (MM) - QUARK**
2. **DGEMM - PLASMA, Intel MKL**
3. **Cholesky - Intel MKL**

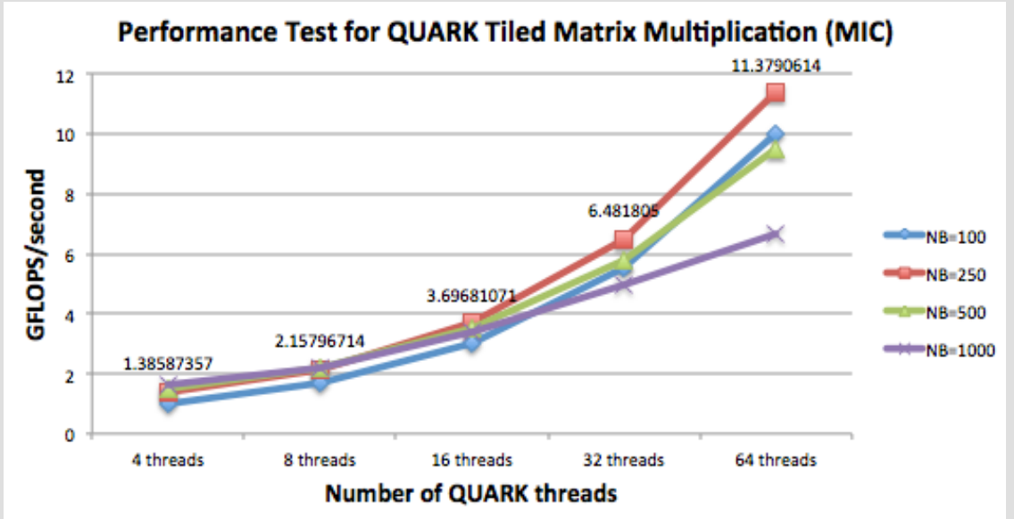
- Both Native and Offload Execution were taken into consideration

NESTED FOR-LOOP MATRIX MULTIPLICATION RESULTS

- I have modified example code from Dr. Asim YarKhan for a QUARK-multithreaded, tiled-routine matrix multiplication driver that will measure the performance in seconds and GFLOPS and print this data in a user-friendly manner to be used on any graphing software.
- To generate GFLOPS/sec, under the assumption that C = A * B where A,B,C are symmetric matrices (n by n), then the general formula would be: $\frac{2n^3}{10^9 * time\_avg}$



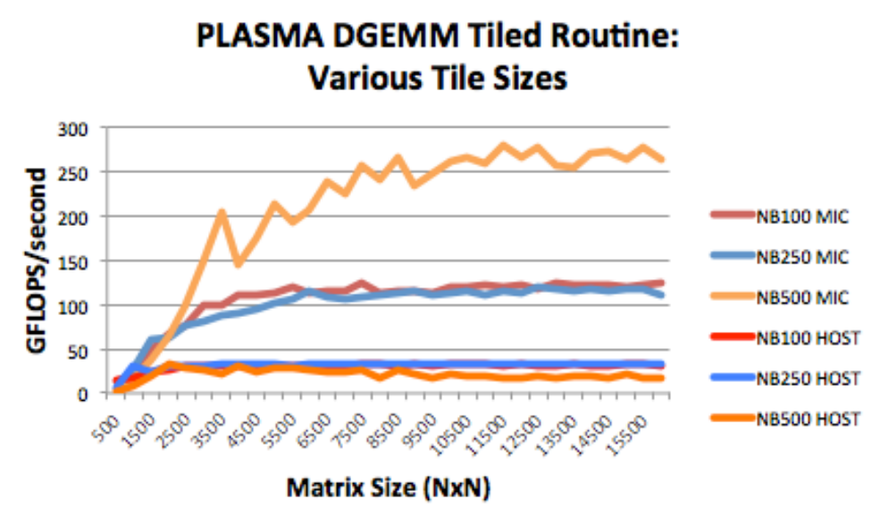
	NB=100	NB=250	NB=500	NB=1000
4 threads	13.46	12.56	12.17	9.01
8 threads	26.78	24.34	23.66	14.89
16 threads	52.33	47.48	45.76	23.14
32 threads	53.96	50.25	32.62	22.62
64 threads	52.23	49.54	20.25	13.07



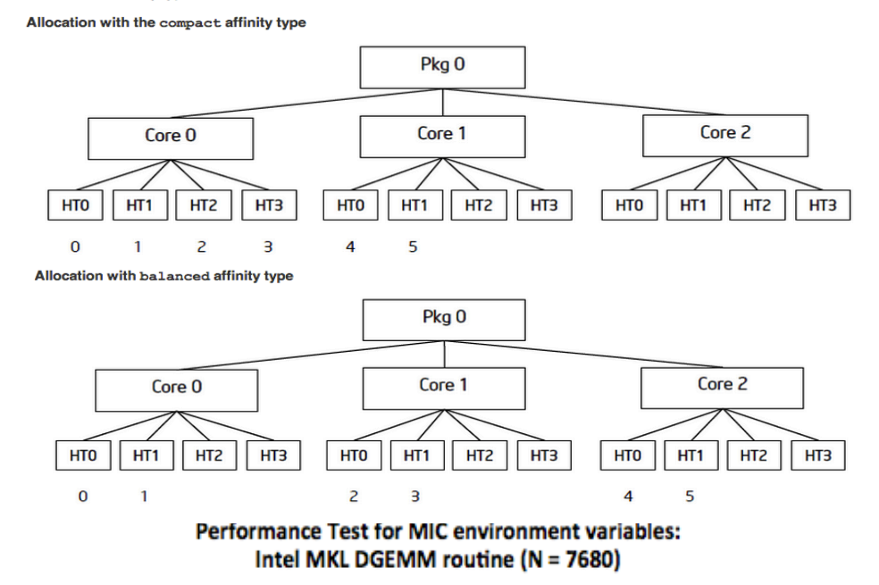
	NB=100	NB=250	NB=500	NB=1000
4 threads	1.00	1.39	1.50	1.63
8 threads	1.70	2.16	2.22	2.22
16 threads	3.03	3.70	3.54	3.36
32 threads	5.52	6.48	5.78	4.94
64 threads	9.97	11.38	9.49	6.68

- The general trend for the HOST shows optimal performance at 16 threads; though at smaller tile sizes, this threshold can be 32 threads.
- The general trend for the MIC shows that optimal performance can be attained at 64 threads, and the data proves to be scalable; however, the actual performance is significantly slower than that on the HOST.
- The performance is still poor (~50 GFLOPS/sec on HOST and ~10 GFLOPS/sec on MIC) but there is possibility for increased performance through offloading and added parallelism.

DGEMM

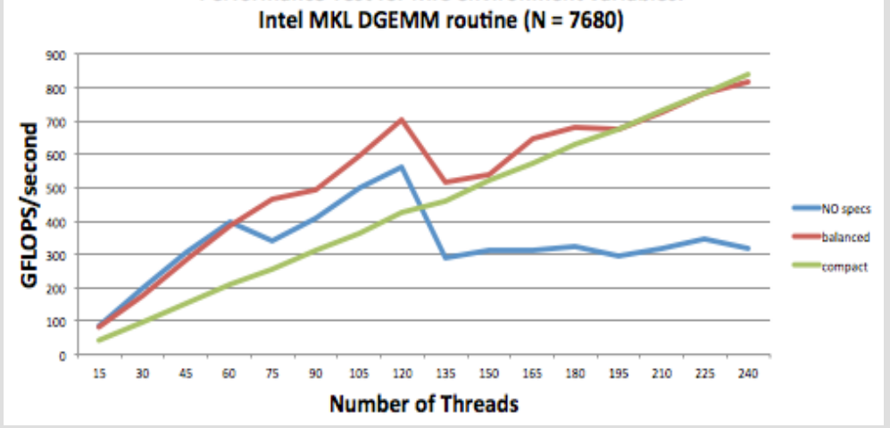


- PLASMA is installed as a module within Beacon, and a separate environment was installed on the HOST for comparison data The routine is optimized through a tiled routine similar to the QUARK MM.

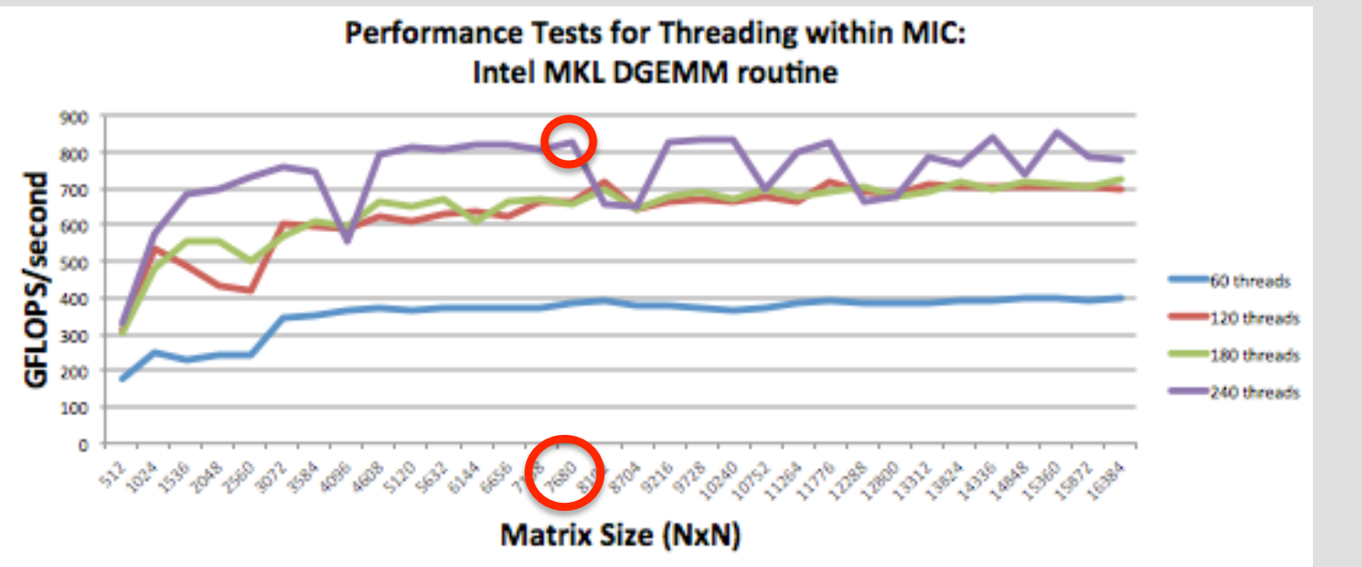


<u>MIC Environment Variables:</u>

- OMP_NUM_THREADS:
  - In Beacon, each node has 4 MIC, each with 60 cores (MAX VALUE = 240).
- KMP_AFFINITY:
  - Compact: Sequential Queuing
  - Balanced: Threads allocated evenly among cores

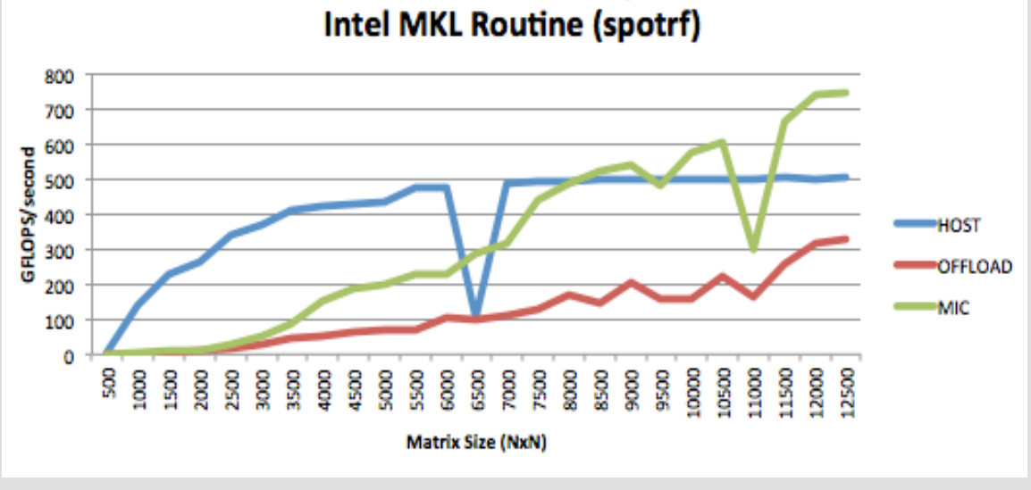


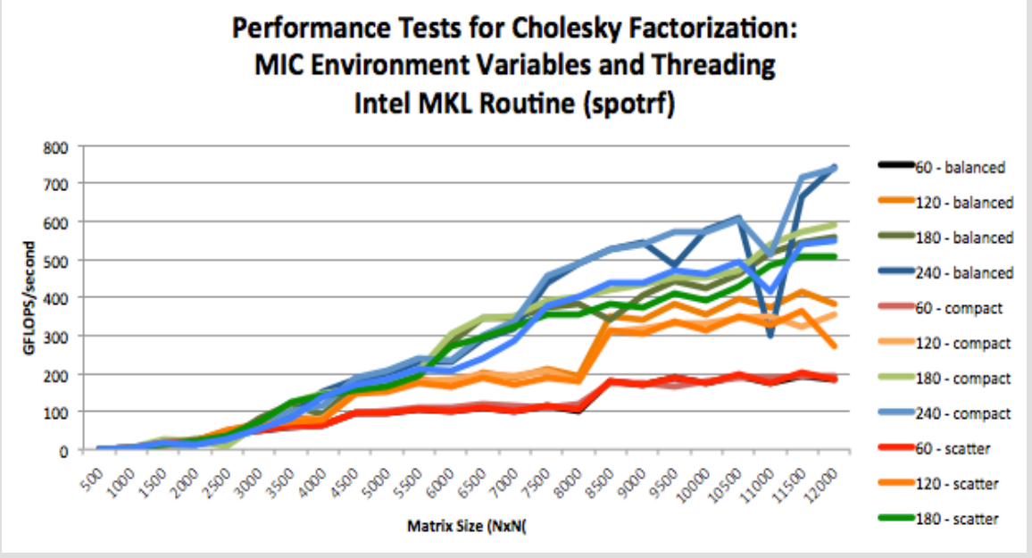
- Intel's MKL Library has been advertised to have its functions optimized (i.e., DGEMM = 833 GFLOPS/s); therefore, this test was recreated.



- The test was successful. Given the maximum number of threads and setting the core organization to balanced, the results matched.

SPOTRF (CHOLESKY FACTORIZATION MKL)





- Formula for GFLOPS/s: $\frac{\frac{1}{3}n^3}{10^9 * time\_avg}$
- Single Precision Cholesky Factorization was tested on different modes of execution.
- MAX GFLOPS/sec was achieved at ~745 within the MIC.
- Given the MIC environment variables, a stress test was implemented to see what were the ideal conditions for getting a similar performance output.
- Best overall performance was attained from using 240 threads and organizing in a compact manner.

CODE GENERATING DAG&CODE USING QUARK

```
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),wherei>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 atrributes 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
   ……
   }
```

```
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),wherei>j>k*
   {
      Quark_Task_Flags  tflags=Quark_Task_Flags_Initializer; //initailize the task
      QUARK_Task_Flag_Set(&tflags,TASK_PRIORITY,1);  //set 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; }
```

EXPECTED GOALS

<u>Runtime Systems</u>

- Optimize QUARK implementations (matrix multiplication, DGEMM) with additional OpenMP and Offloading directives to produce better performance.
- Incorporate the OOC Cholesky Factorization into QUARK and implement onto Beacon.

<u>OOC Cholesky Factorization</u>:

- Complete the code combining OOC algorithm and general Cholesky factorization.
- Extend to multiple MPI processes case.
- Extend to LU factorization with pivoting and QR factorization.

REFERENCES

- Betro, Vincent. *Beacon Quickstart Guide at AACE/NICS*
- Betro, Vincent. *Beacon Training: Using the Intel Many Integrate Core (MIC) Architecture: Native Mode and Intel MPI*. March 2013
- D'Azevedo, Eduardo, Shiquan Su, and Kwai Wong. *A Performance Study of Solving a Large Dense Matrix for Radiation Heat Transfer*.
- Intel. https://software.intel.com/en-us/mic-developer
- YarKhan, Asim. *Dynamic Task Execution on Shared and Distributed Memory Architectures*. Dec. 2012.
- YarKhan, Asim, Jakub Kurzak, and Jack Dongarra. *QUARK Users' Guide*. April 2011
- Images are provided by Google Images, their respective websites, or generated using software

TEAM INFO

- **Authors:** Allan Richmond Razon Morales and Tian Chong
- **Mentors:** Dr. Kwai Wong and Dr. Eduardo D'Azevedo
- **Collaborators:** Dr. Shiquan Su, Dr. Asim YarKhan, and Ben Chan