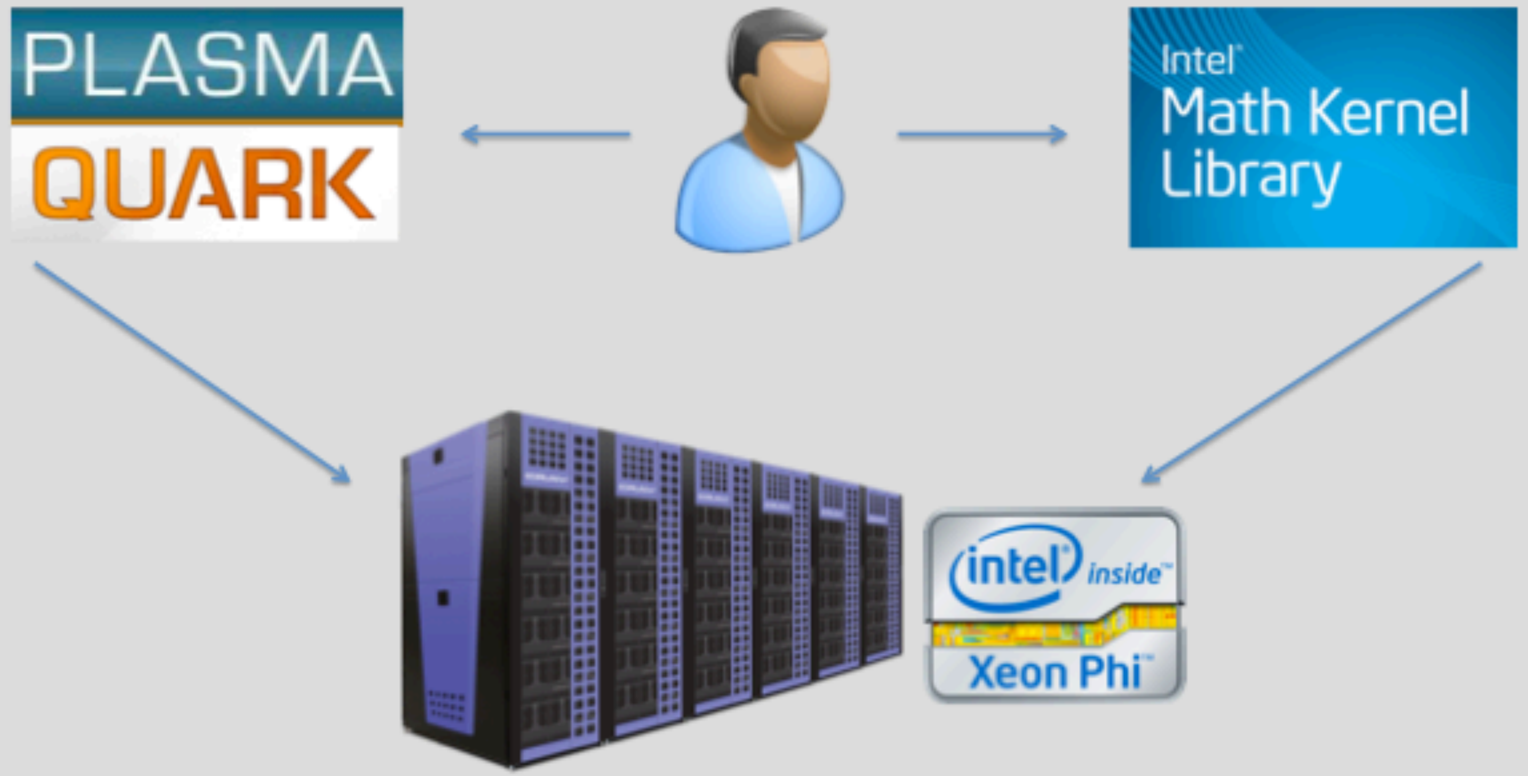
Computational Science for Undergraduate Research Experience (CSURE) 2014

RUNTIME SYSTEMS AND OUT-OF-CORE CHOLESKY FACTORIZATION ON THE INTEL XEON PHI SYSTEM

Students: Allan Richmond Razon Morales (GWU) and Chong Tian (CUHK)
Mentors: Dr. Kwai Wong (UT), Dr. Eduardo D'Azevedo (ORNL)
Collaborators: Dr. Shiquan Su (NICS), Dr. Asim YarKhan (UT), Ben Chan

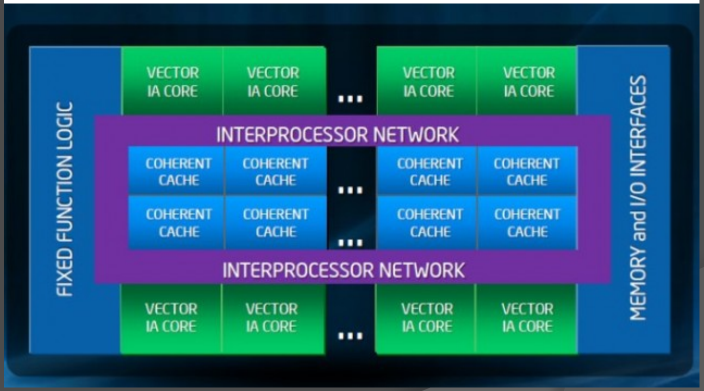
Visual Overview



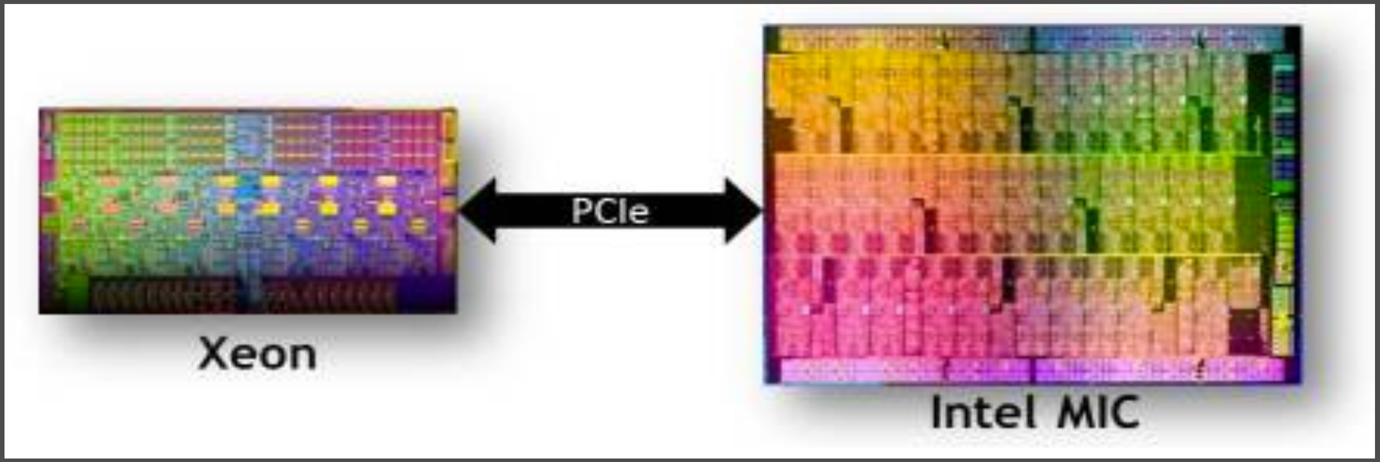
- **PLASMA** – dense algebra algorithms
- **QUARK** – multithreading and task management
- **Intel MKL Library** – optimized math library for comparison

Runtime Systems Overview

- Basic Specifications:
  - Beacon: 157.3 TFLOPS (peak)
  - 48 compute nodes
  - Each node has access to four Intel Xeon Phi co-processors 5110P (MIC) and two 8-core Intel Xeon E5-2670 processors
- Goals:
  - Compare different runtime systems that can be run on the Intel Xeon Phi System
  - Utilize QUARK within this system
  - Optimize the QUARK performance tests to see if the program can be scaled efficiently
- Performance testing was conducted on the host processor and its coprocessors through its normal execution.



Intel Xeon Architecture



2 x Intel Xeon Processor E5-2670

- 16 cores per node (8 per processor)
- 2.600 GHz Clock Speed
- 256 GB RAM

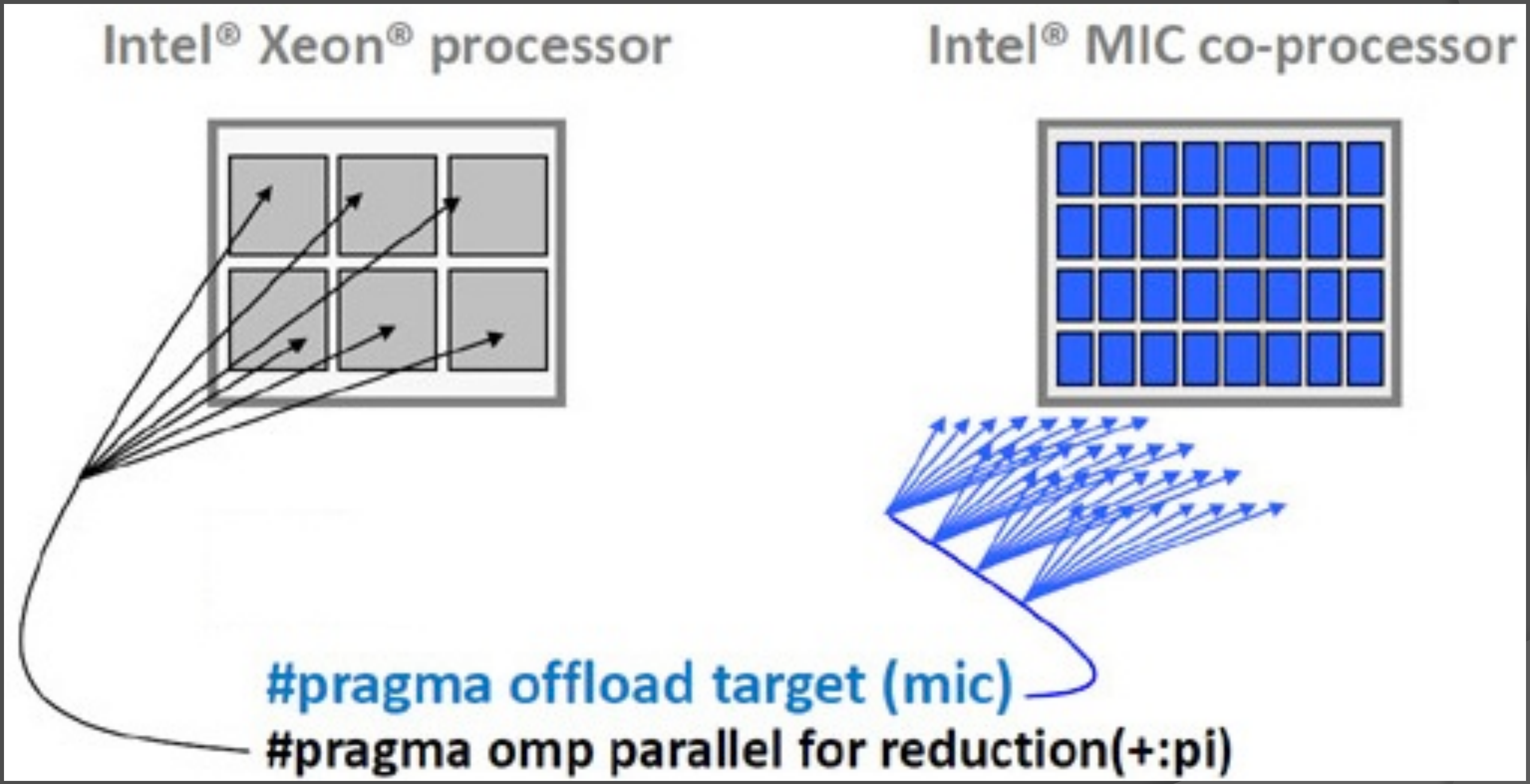
4 x Intel Xeon Phi Coprocessor 5110P

- 60 cores
- 1.053 GHz Clock Speed
- 8 GB RAM

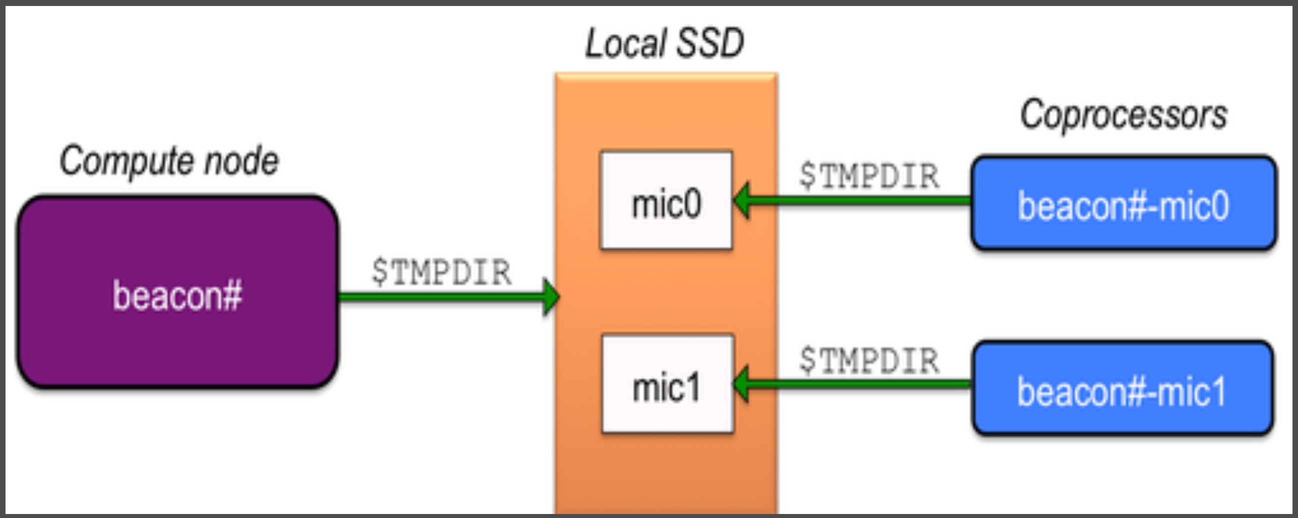
- **Pro**: More Memory (8x more)
- **Con**: Less Computational Power

- **Pro**: More Computational Power (more cores)
- **Con**: Less Memory

Intel Xeon Architecture (Offload Comparison Example)



Beacon: Modes of Execution



- Host: Normal Execution through host processor (compute node)
- Native: Execution runs only directly on the co-processor (MIC)
- Offload: Run on the host processor and then "offloads" dense calculations to the co-processor (Ideal for the OOC algorithm)

Proposed Methodology

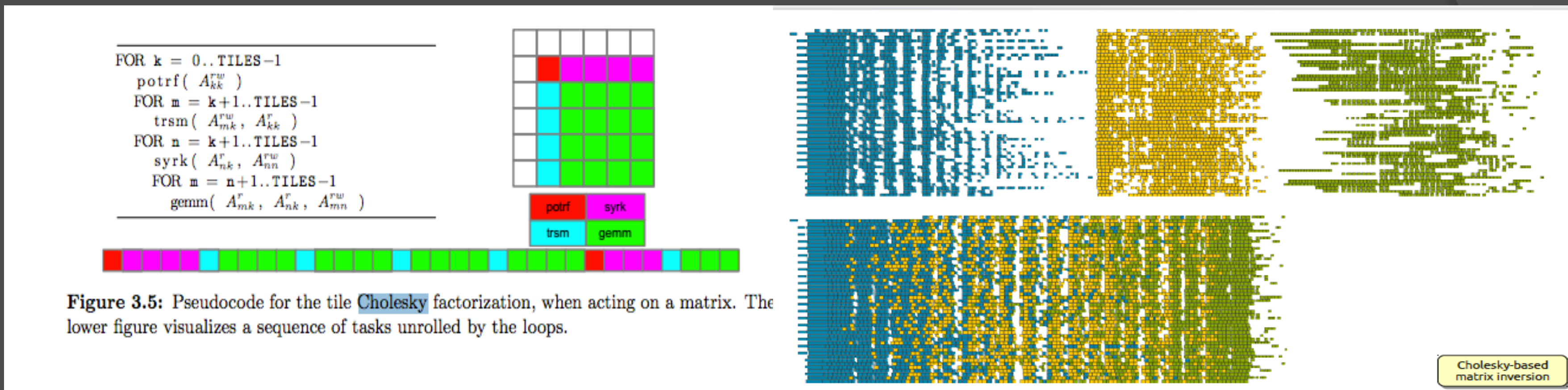


Figure 3.5: Pseudocode for the tile Cholesky factorization, when acting on a matrix. The lower figure visualizes a sequence of tasks unrolled by the loops.

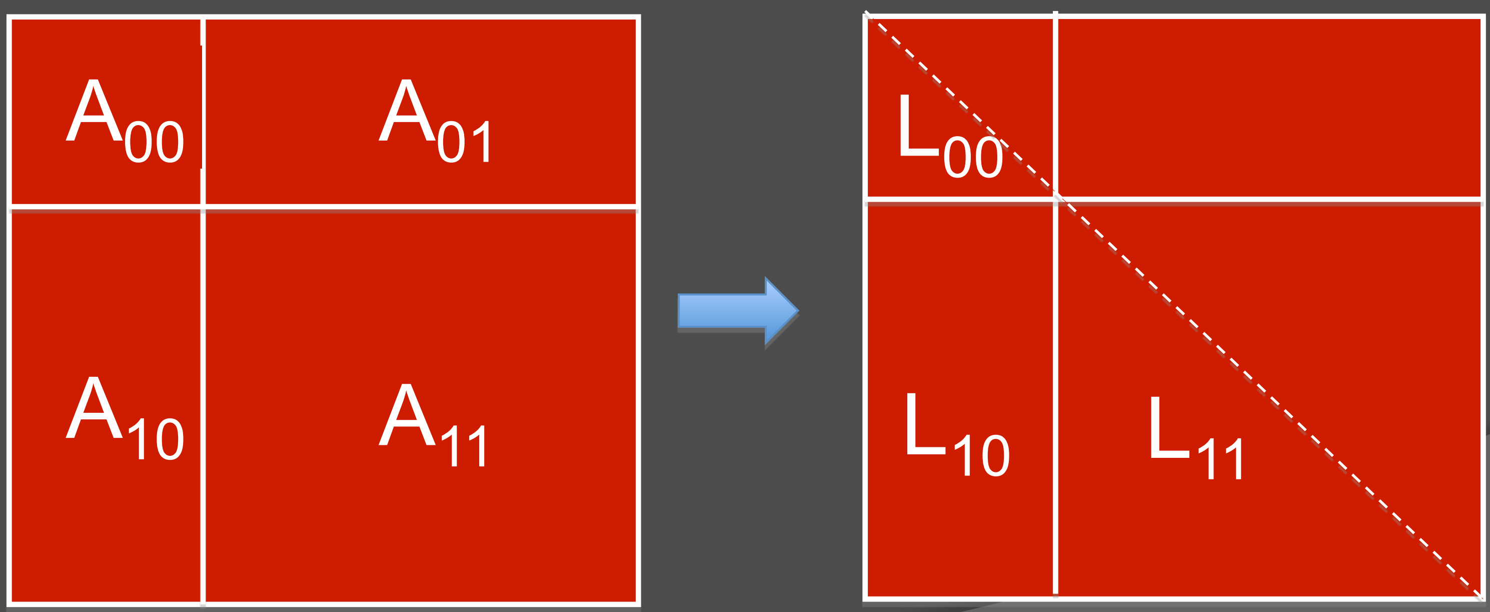
- Runtime Systems
  - Understanding each programming environment
    - Nested-For Loop Matrix Multiplication (MM) – QUARK
    - DGEMM – PLASMA, Intel MKL
    - Cholesky – Intel MKL
  - All modes of executions were considered and tested

OOC Cholesky Using Dynamic Scheduling

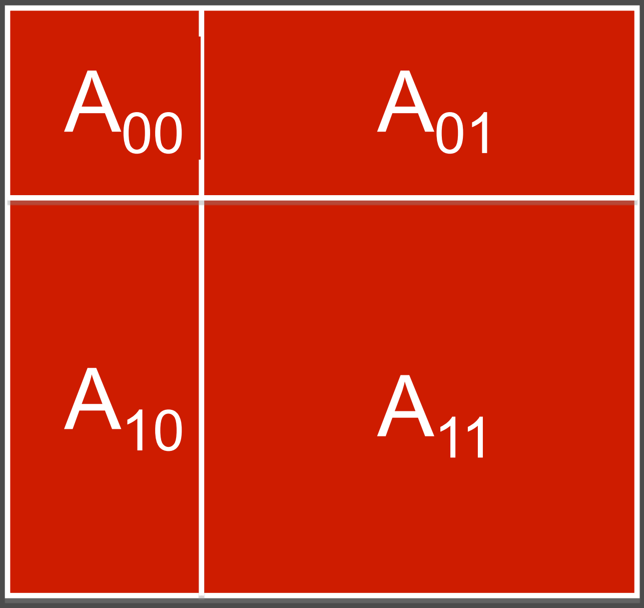
- Cholesky Factorization
- Task DAG and QUARK
- OOC algorithm
- Further goals

General Cholesky Steps

- Step 1: L_{11} <-- cholesky(A_{11}) ,
  Step 2: L_{21} <-- A_{21} / L_{11}^T, <Panel factorization>
  Step 3: A_{22} <-- $A_{22} - L_{21} * L_{21}^T$, <Trailing submatrix update>
  Step 4: L_{22} <-- cholesky(A_{22}),

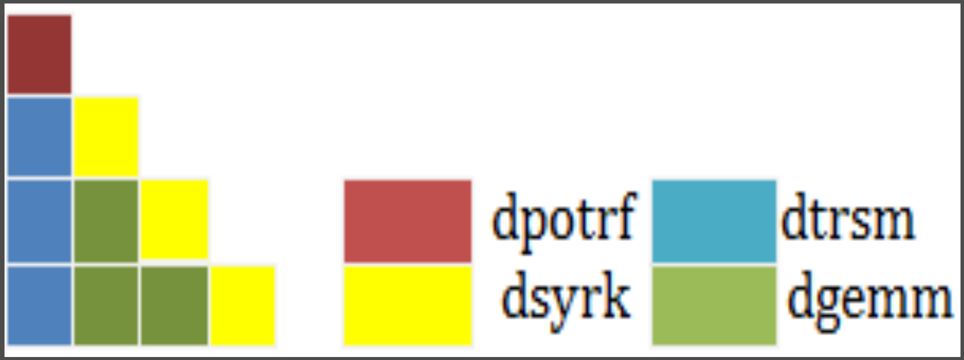


General Cholesky Using Tile Operations



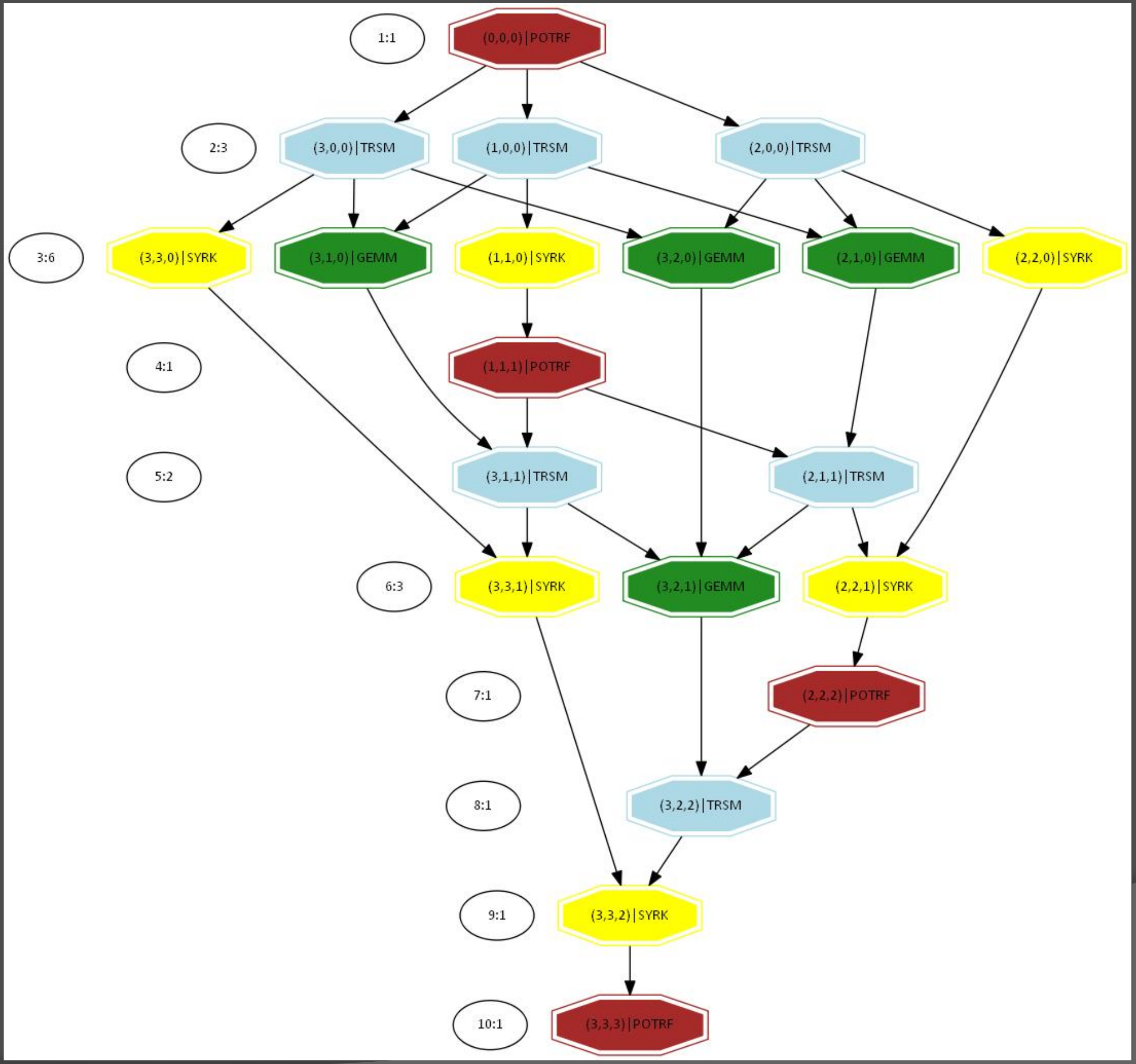
A_{00}	A_{01}	…	A_{0k}	…	A_{0n}
A_{10}	A_{11}	…	…	…	…
…		…			
A_{k0}			A_{kk}		
…				…	
A_{n0}					A_{nn}

General Cholesky Factorization Pseudo Code




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

Task Directed Acyclic Graph (DAG)



Code Generating the Cholesky DAG

```
/*1. dpotrf type:(k,k,k)*/
if((j==k)&&(i==j))
{
    list[count].Node=assignlabel(i,j,k);/*Assign node labels*/
    list[count].node=(i+1+j*n)+k*n*n;
    list[count].type='F';
    fprintf(fp,"%ld[label=\"(%ld,%ld,%ld)|POTRF\",color=brown];\n",list[count].node,i,j,k);

    /*in-nodes*/
    if(k>0) list[count].in[0]=assignlabel(i,j,k-1);
    /*Assign data dependencies,i.e. edges*/
    for(q=0;q<3;q++)
    {
        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);
    }

    /*out-nodes*/
    if(k<n-1)
    {
        for(q=1;q<n-k;q++) list[count].out[q-1]=assignlabel(k+q,k,k); /*to (k+q,k,k)*/

    }

    /*Assign the rank*/
    fprintf(fp,"{rank=same;depth%ld %ld}\n",(3*k+1),list[count].node);  /*if (i,j,k) is a type F,*/
                                                                        /*then it's on depth 3k+1*/
```

Screenshot of Result

```
8
NB=?
3
Original A=

2.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 2.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 2.000000 1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 2.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 2.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000 2.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 2.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 2.000000
```

```
L:

1.414214 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.707107 1.224745 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.707107 0.408248 1.154701 0.000000 0.000000 0.000000 0.000000 0.000000
0.707107 0.408248 0.288675 1.118034 0.000000 0.000000 0.000000 0.000000
0.707107 0.408248 0.288675 0.223607 1.095445 0.000000 0.000000 0.000000
0.707107 0.408248 0.288675 0.223607 0.182574 1.080123 0.000000 0.000000
0.707107 0.408248 0.288675 0.223607 0.182574 0.154303 1.069045 0.000000
0.707107 0.408248 0.288675 0.223607 0.182574 0.154303 0.133631 1.060660
```

Screenshot of Result

```
===========
Before Factorization------(0,0) block:

2.000000 1.000000 1.000000
1.000000 2.000000 1.000000
1.000000 1.000000 2.000000
===========


===========
Before Factorization------(0,1) block:

1.000000 1.000000 1.000000
1.000000 1.000000 1.000000
1.000000 1.000000 1.000000
===========


===========
Before Factorization------(0,2) block:

1.000000 1.000000 0.000000
1.000000 1.000000 0.000000
1.000000 1.000000 0.000000
===========


===========
Before Factorization------(1,0) block:

1.000000 1.000000 1.000000
1.000000 1.000000 1.000000
1.000000 1.000000 1.000000
===========
```

```
===========
After Factorization------(0,0) block:

1.414214 1.000000 1.000000
0.707107 1.224745 1.000000
0.707107 0.408248 1.154701
===========


===========
After Factorization------(0,1) block:

1.000000 1.000000 1.000000
1.000000 1.000000 1.000000
1.000000 1.000000 1.000000
===========


===========
After Factorization------(0,2) block:

1.000000 1.000000 0.000000
1.000000 1.000000 0.000000
1.000000 1.000000 0.000000
===========


===========
After Factorization------(1,0) block:

0.707107 0.408248 0.288675
0.707107 0.408248 0.288675
0.707107 0.408248 0.288675
===========
```

General Cholesky Code Using QUARK

```
void CORE_incore_dpotrf( Quark *quark )

void QUARK_incore_dpotrf( Quark *quark,Quark_Task_Flags *task_flags, int uplo,
int n, double *A,int nb )

……

/*1. dpotrf type:(k,k,k)*/
        if((j_==k_)&&(i_==j_))
        {
          /*set task flags*/
          Quark_Task_Flags tflags=Quark_Task_Flags_Initializer;
          QUARK_Task_Flag_Set(&tflags,TASK_PRIORITY,3);
          /*Insert the dpotrf task*/
          QUARK_incore_dpotrf(quark,&tflags,(int)'L',NB,A2[i_][j_],NB);
          continue;
        }
```

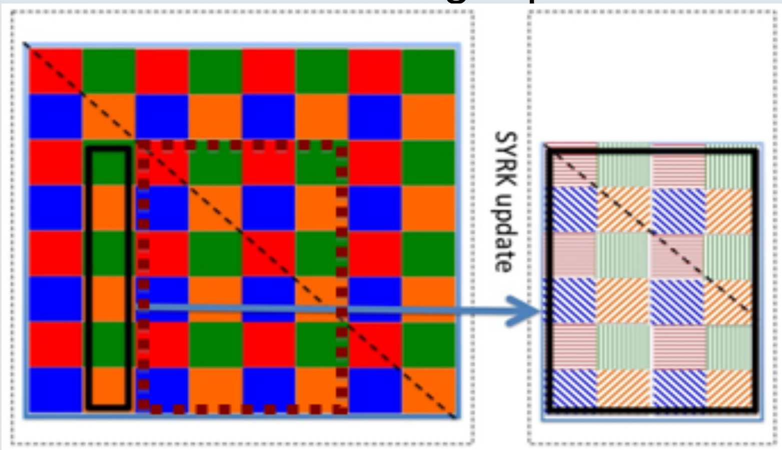
Out-of-Core Algorithm (OOC)

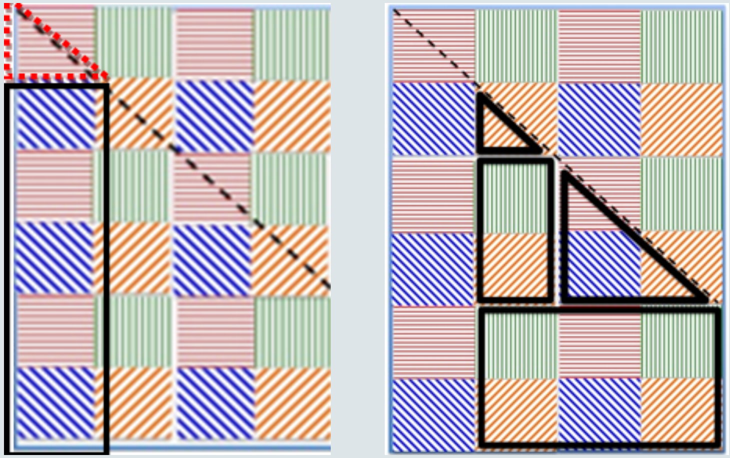
- Motivation: CPU vs Coprocessor

CPU	Coprocessors(GPU,Intel MIC,etc.)
Slower than Coprocessors for some certain computations like dense matrix multiplication. Larger memory size	Much faster and more energy efficient Limited amount of device memory Data movement is expensive

Out-of-Core Structure

Out-of-core part	In-core Part
1. Loads Y panels to the device memory 2. Apply the update from the part already factorized.,which is called “left-looking” update.	Factorize the sub-matrix residing on device memory,in which “right-looking” update is involved.





OOC Cholesky pseudo code

*/*Out of core part:(starting from the A(k,k) tile)*/*

*/*O1.Send in Y-panel*/*

```
for j=k:1:k+sizeY-1
```

*/*Expected optimization 0:find the optimal Y size*/*

```
    for i=j:1:n
        H2D_Copy A(i,j) -> Y(i,j)
```

*/*Expected optimization 1:Since the right part of the lower part of "A" shrinks. for the same amount of space dedicated to the Y panel, we may use a wider Y-panel to store as many tiles as possible*/*

OOC Cholesky pseudo code

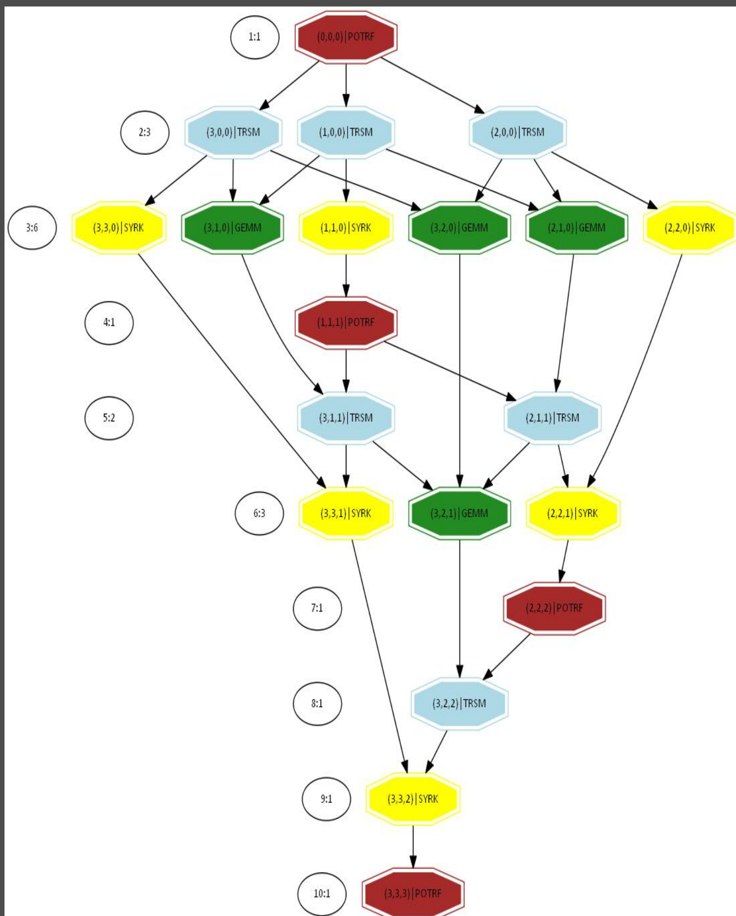
```
/*O2.Left looking update,if not the first Y-panel*/
    /*Send factorized columns into X panel*/
    for i=1:1:k-1
    {
        for j=k:1:n
            H2D_Copy L(i,j)->X(j)
/*Expected optimization 2:Some factorized panels can be copied into X panel immediately before written back to CPU */
        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:Use double buffering—while one X panel is doing dgemm,the other can be reading data concurrently*/
```

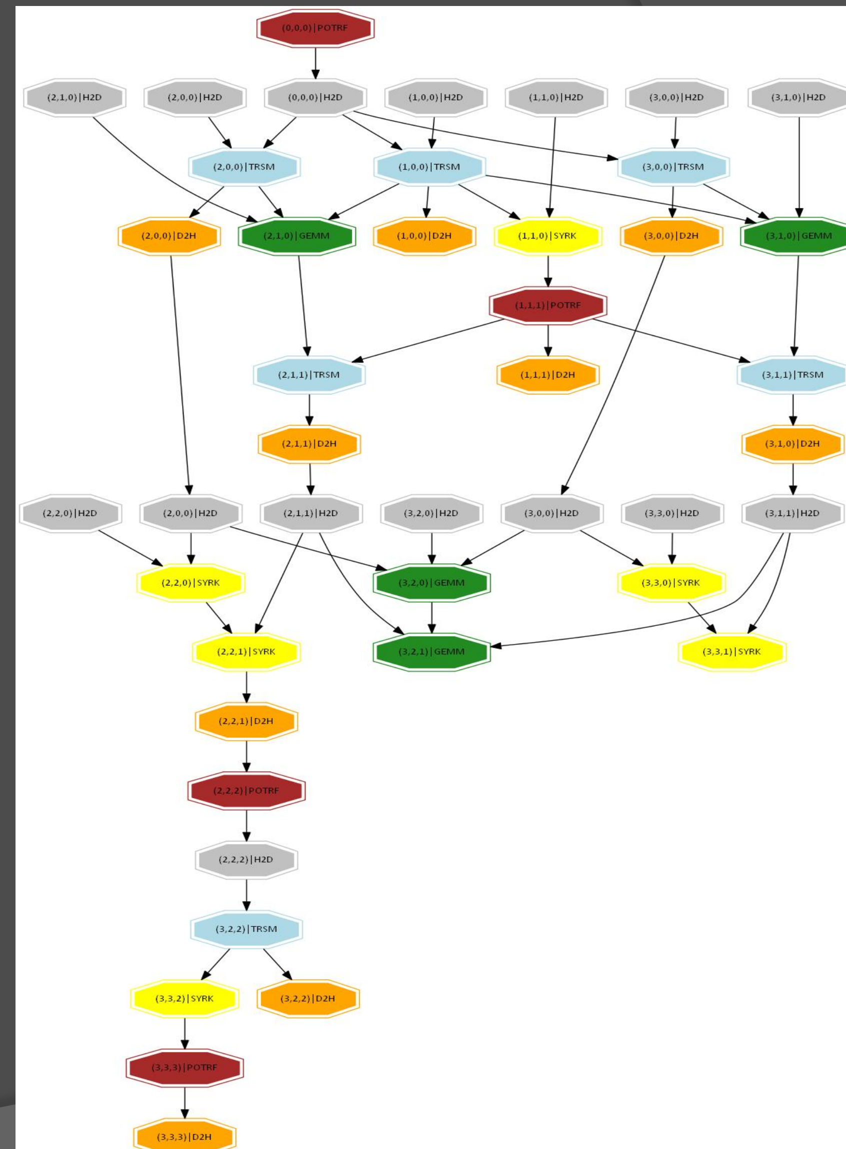
OOC Cholesky pseudo code

*/*In core part : similar to the general Cholesky factorization,except there are extra data movements,especially from Y panel to X panel or to CPU*/*

/*Expected optimization 5:Perform all dpotrf() operations on CPU*/

Simple 4*4 OOC Cholesky DAG





Further Goals

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

Expectations from Runtime Results

- Which mode of execution is the most scalable?

- Is there a threshold or condition where the performance begins to remain constant or even fails?

Breakdown for Testing Approach

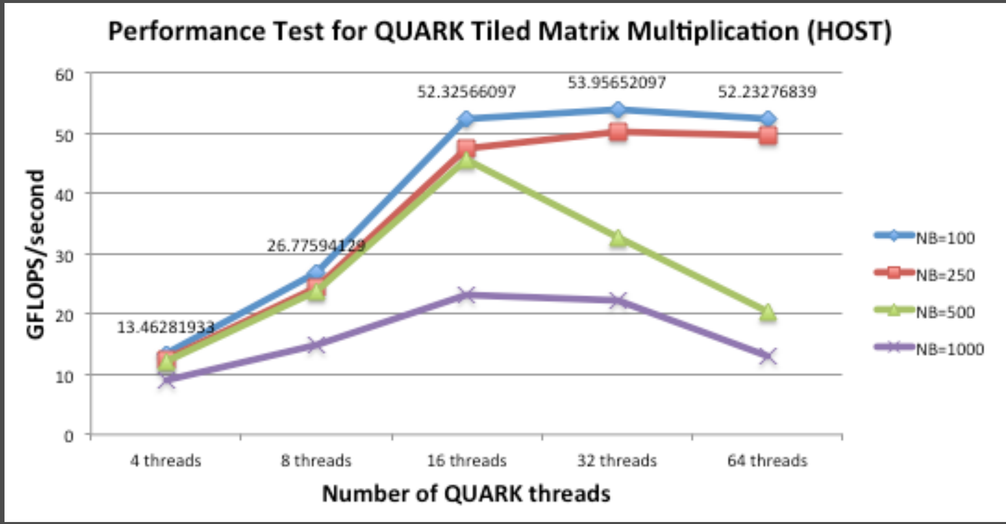
Testing Routines:

1. QUARK MM
2. PLASMA DGEMM Tiled
3. Intel MKL DGEMM
4. Intel MKL SPOTRF (Cholesky Factorization)

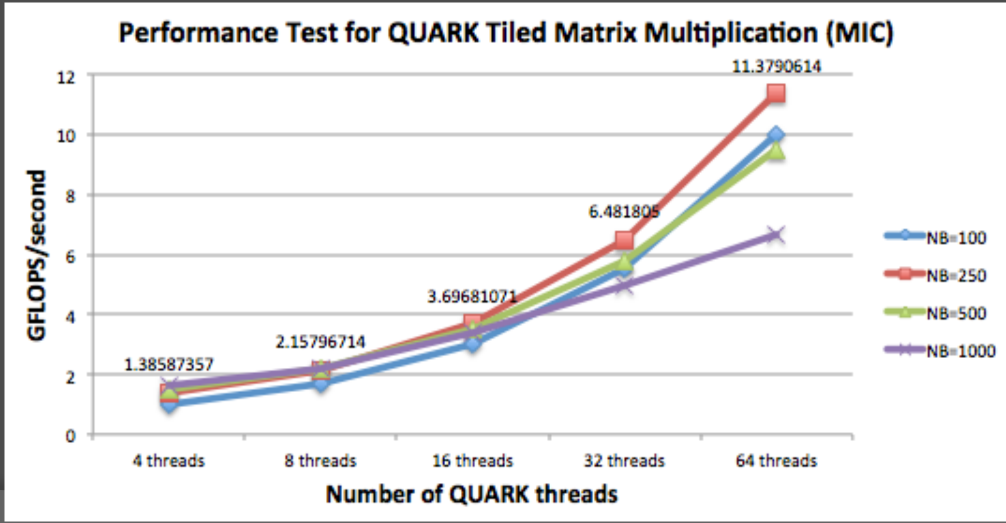
Measuring GFLOPS/s : ("Giga" Floating Operations per second)

1. For matrix multiplication and DGEMM: $$\frac{2n^3}{10^9 * time\_avg}$$
2. For Cholesky Factorization (SPOTRF): $$\frac{\frac{1}{3}n^3}{10^9 * time\_avg}$$

QUARK Matrix Multiplication: Multi-threaded Tiled Routine

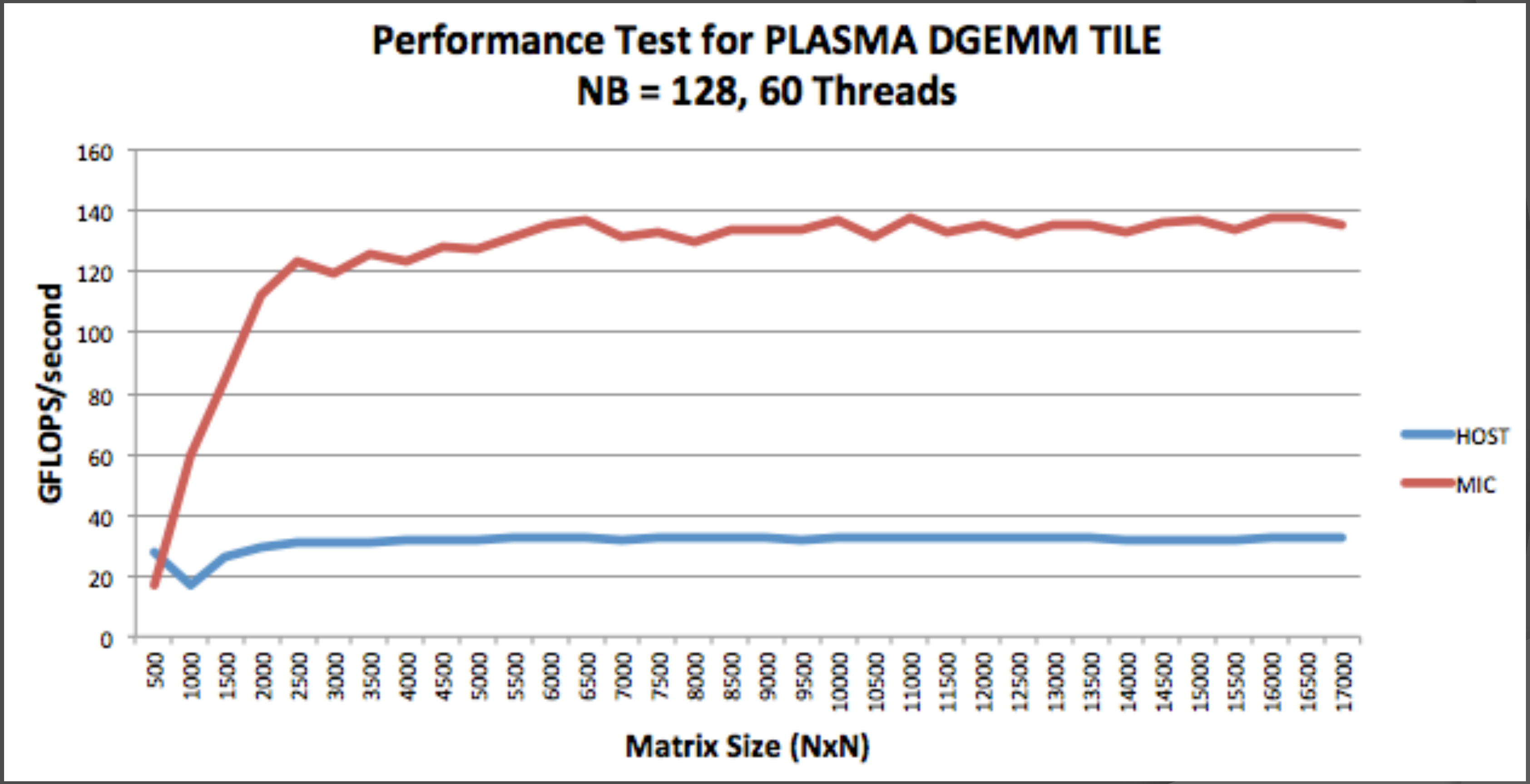


	NB=100	NB=250	NB=500	NB=1000
4 threads	13.46281933	12.5576024	12.173018	9.01319071
8 threads	26.77594129	24.3421548	23.656976	14.8945193
16 threads	52.32566097	47.4777321	45.76333371	23.1449664
32 threads	53.95652097	50.2472455	32.62076229	22.262155
64 threads	52.23276839	49.5421097	20.25220514	13.0737957

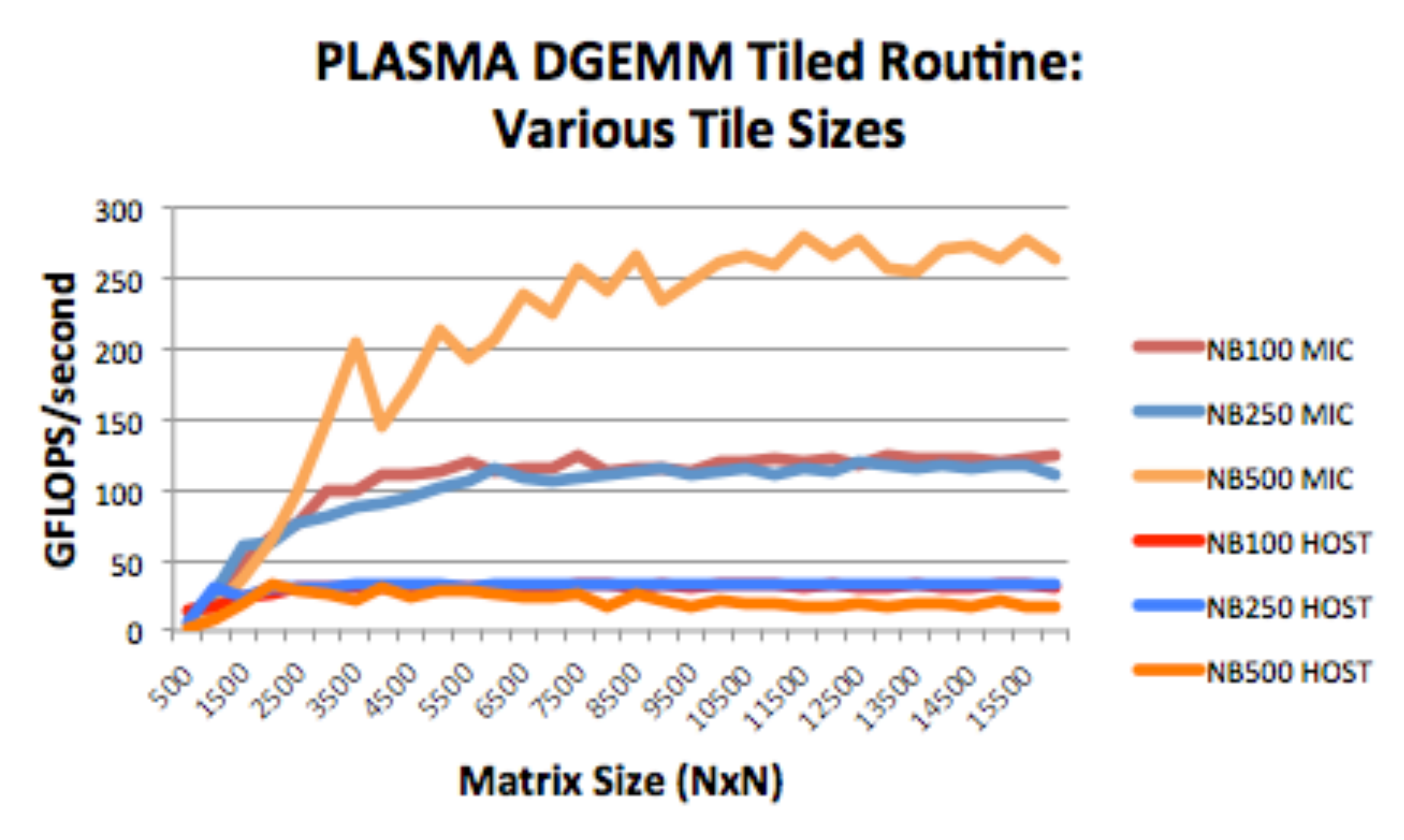


	NB=100	NB=250	NB=500	NB=1000
4 threads	0.99663077	1.38587357	1.496806	1.63284111
8 threads	1.70272846	2.15796714	2.21691933	2.22034778
16 threads	3.03245308	3.69681071	3.537532	3.36262222
32 threads	5.5189	6.481805	5.77946333	4.94149778
64 threads	9.96874769	11.3790614	9.487648	6.68409111

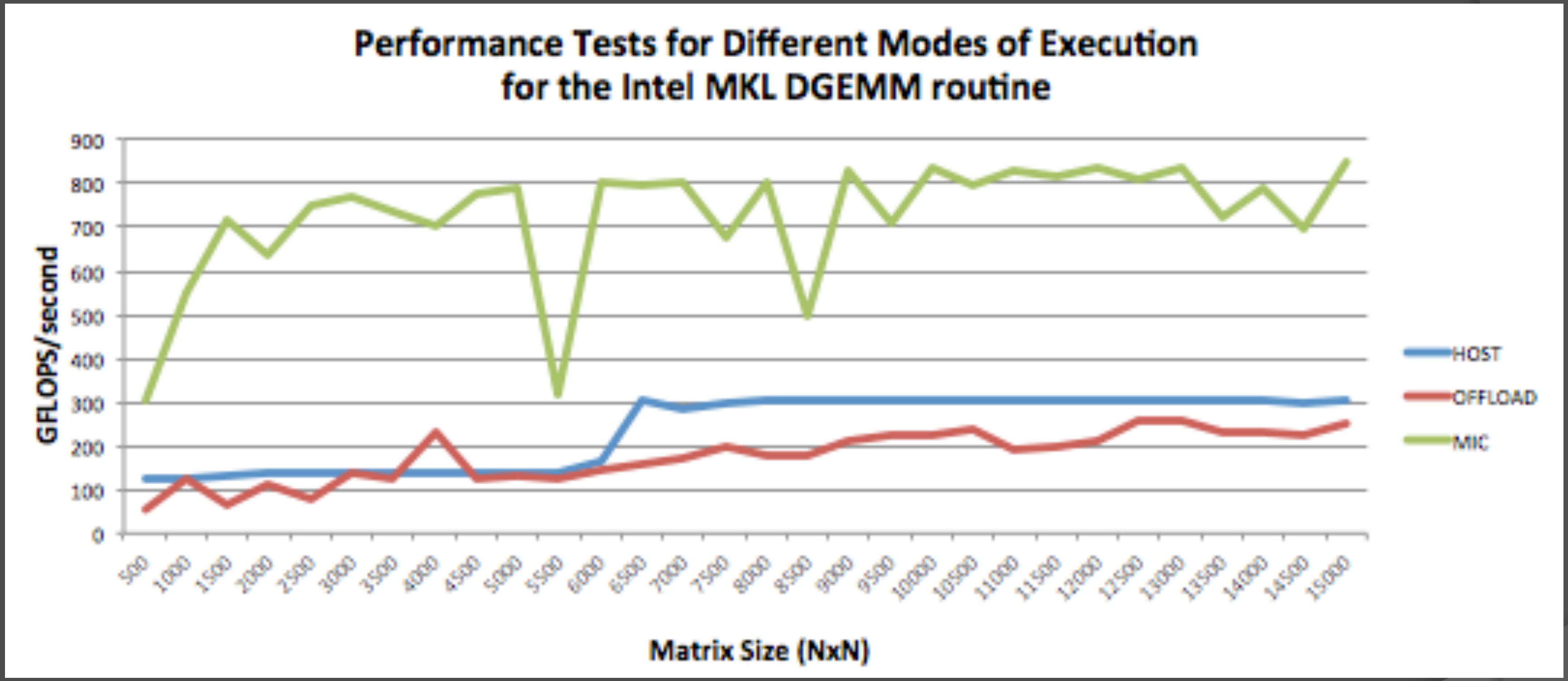
PLASMA DGEMM Tiled Routine: MIC vs HOST



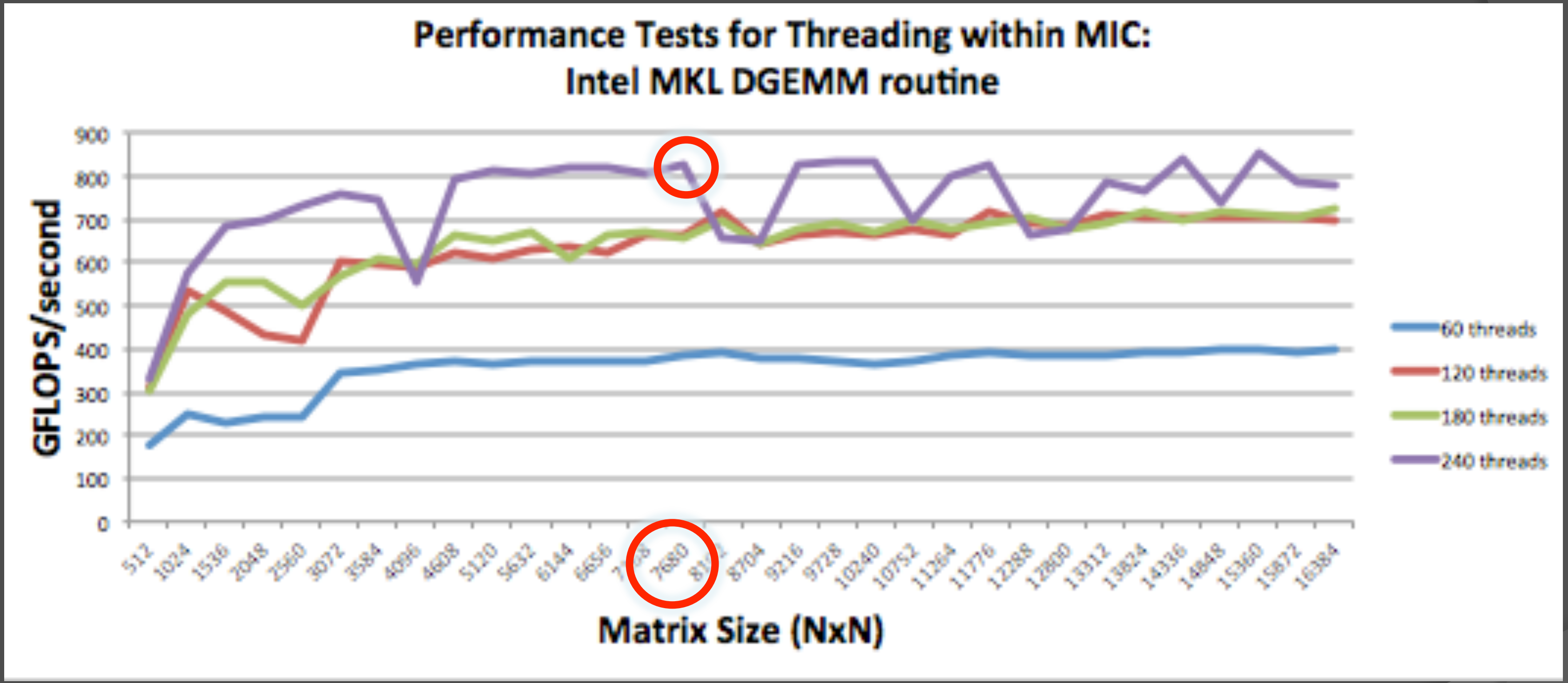
PLASMA DGEMM Tiled Routine: Different Tile Sizes



Intel MKL Routine DGEMM: Modes of Execution



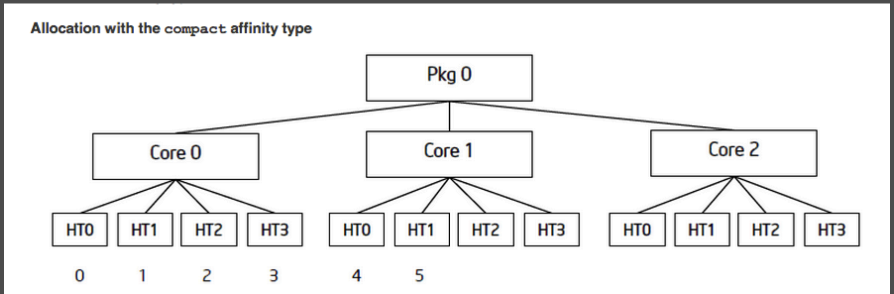
Intel MKL Routine DGEMM: Threading within MIC

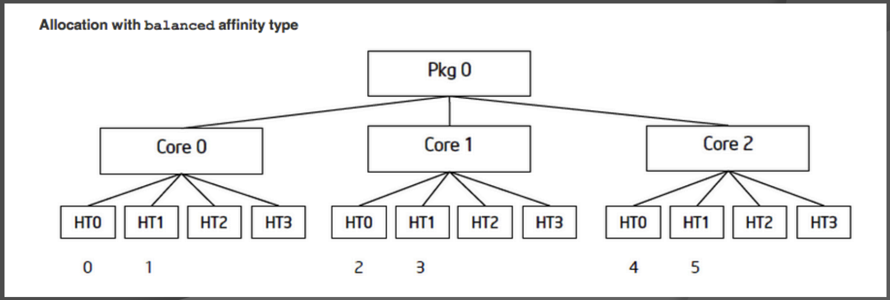


MIC Environment Variables

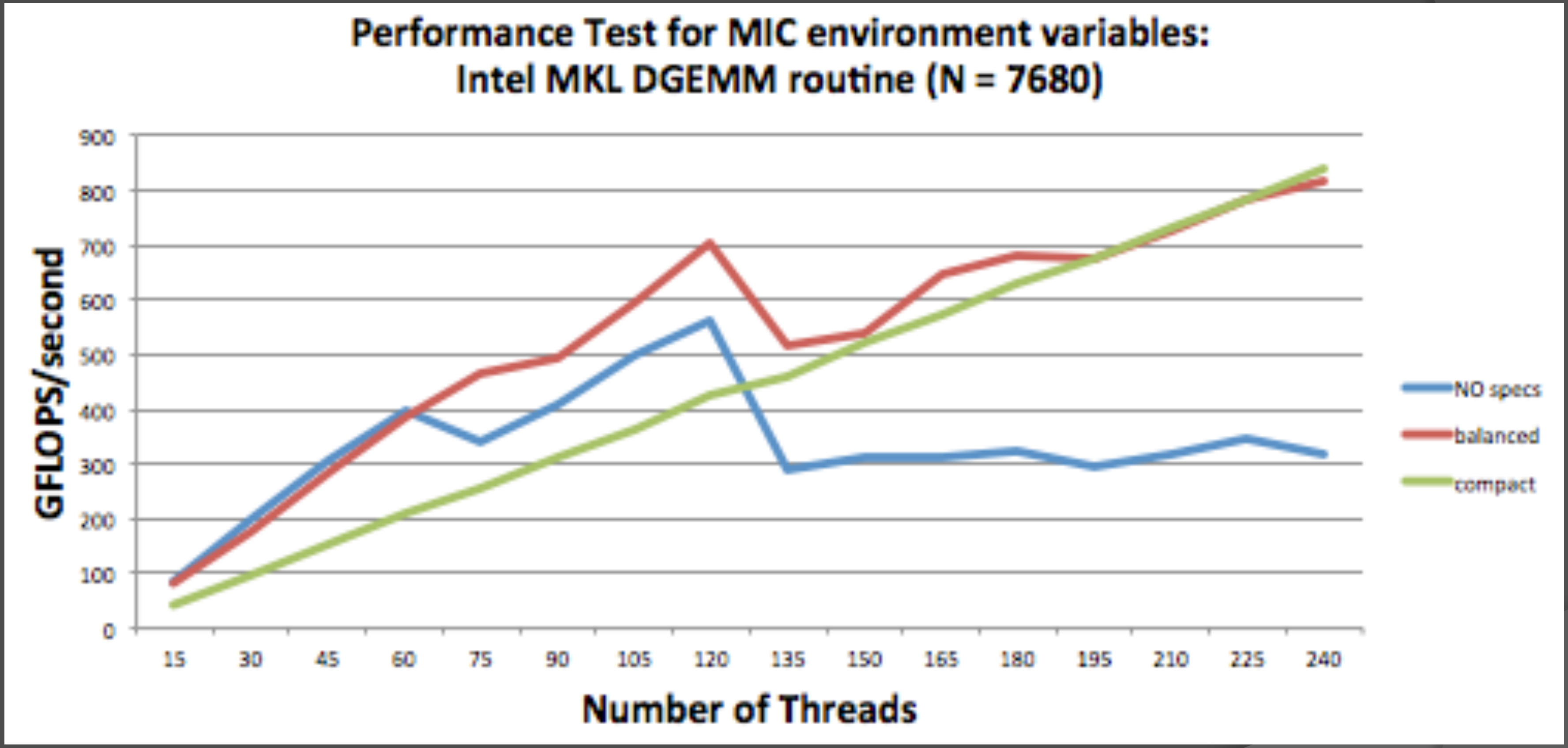
- OMP_NUM_THREADS:
  - Each coprocessor has 60 cores
  - Beacon has 4 per node.
  - Therefore, maximum value is 240.

- KMP_AFFINITY:
  - Compact: Sequential Queuing
  - Balanced: Threads allocated evenly among cores

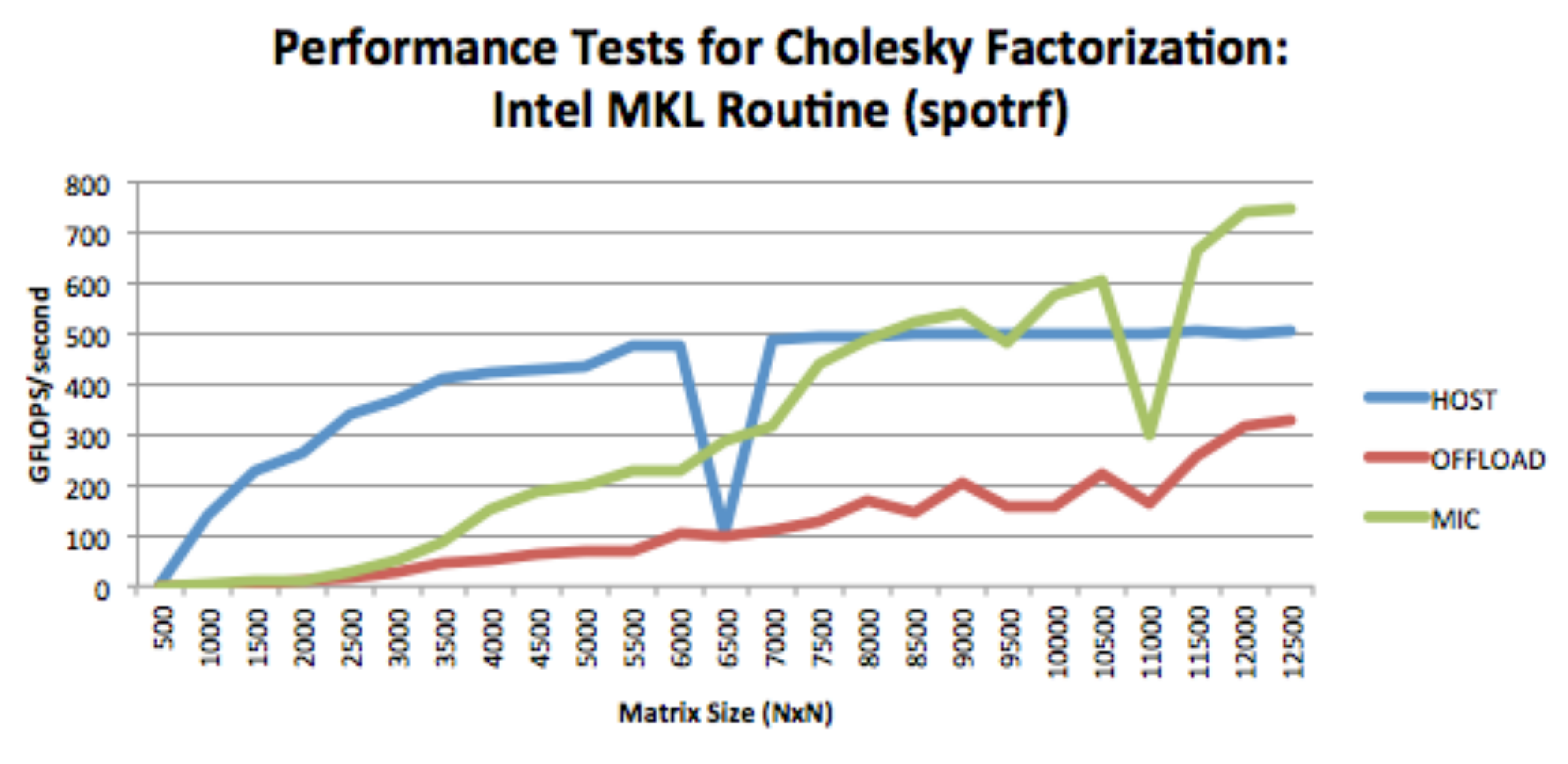




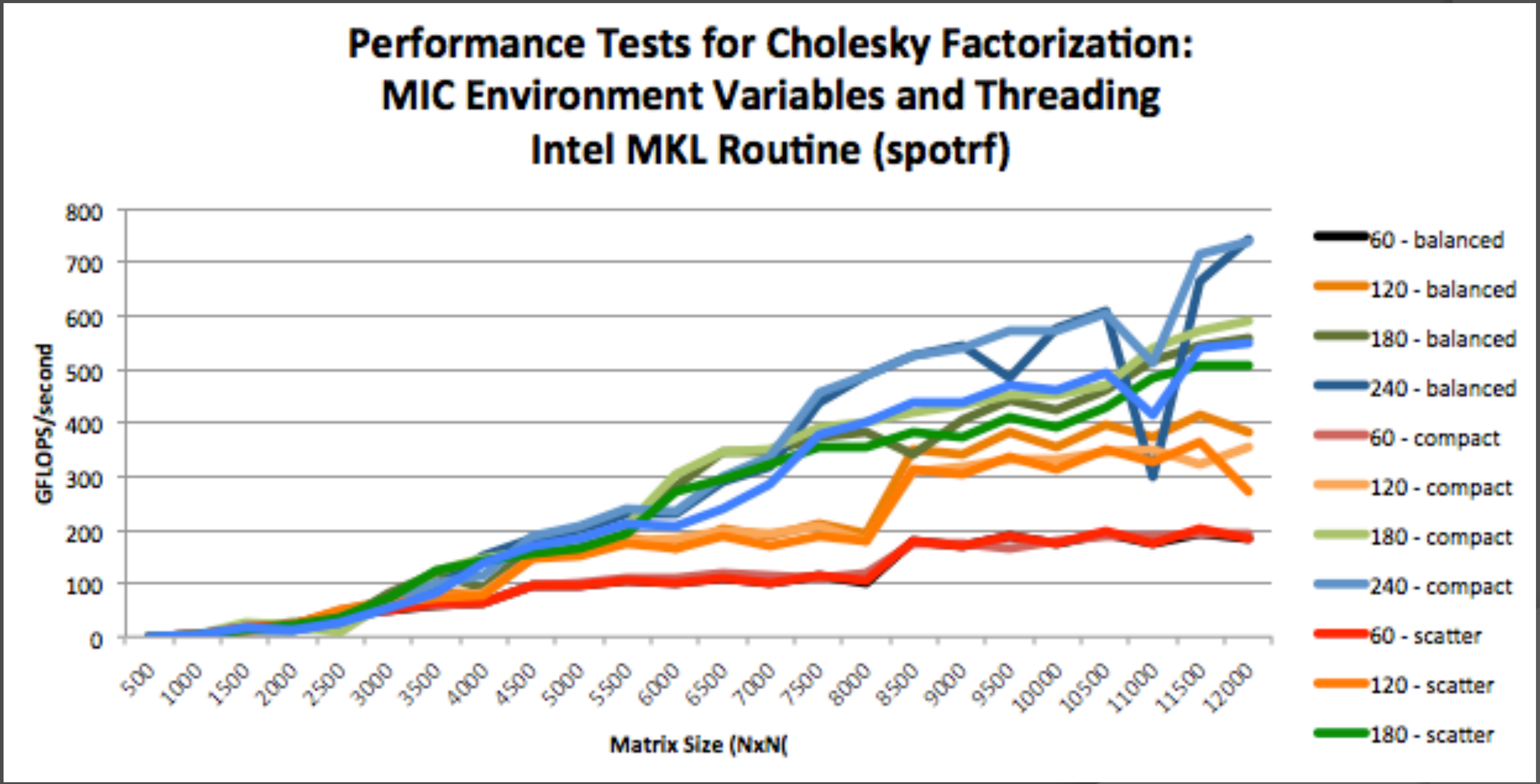
Intel MKL Routine DGEMM: MIC Environment Variables



Intel MKL Routine SPOTRF (Cholesky Factorization) Modes of Execution



Intel MKL Routine SPOTRF (Cholesky Factorization) Modes of Execution



Other Potential SPOTRF tests

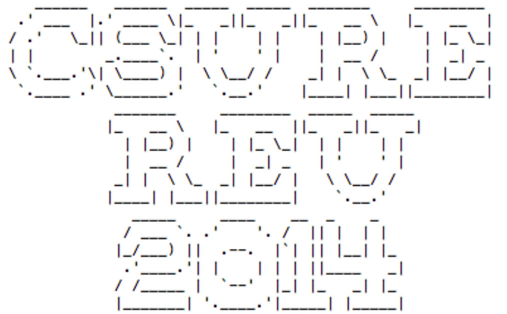
- Serial and OpenMP– results barely suffice for comparison
  - ~0.6 GFLOPS/second for Serial
  - ~0.6 GFLOPS/second for OpenMP (MIC)
  - ~1.2 GFLOPS/second for OpenMP (Host)

Future Goals

- 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.

Thinking about the Future: Documentation (In Progress)

```
********************************************************************************
```




```
Computational Science for Undergraduate Research (CSURE)
Joint Institute of Computational Sciences (JICS)
National Institute for Computational Sciences (NICS) at
Oak Ridge National Laborator

Student Researchers: Allan Richmond Morales, Tian Chong
Mentors: Dr. Kwai Wong, Dr. Eduardo D'Azevedo
Collaborators: Dr. Shiquan Su, Dr. Asim YarKhan, Ben Chan

Title: "Runtime Systems and the Out-of-Core Cholesky Algorithm
                On the Intel Xeon Phi System"

http://www.jics.utk.edu/csure-reu/csure14

********************************************************************************

***IMPORTANT TO NOTE***
I have modified, reformatted, and added code to make results more readable or
add more functionality, but I cannot claim ownership. Original code
has been provided by Dr. Asim YarKahn (QUARK + PLASMA),
the Help XSEDE troubleshooting, and Intel MKL examples.
***********************


Table of Contents
=================
- Introduction
- Special Features
- Methodology
- General Execution
- QUARK + details
- PLASMA + details
- Results
- References
- Team Info
```

```
Methodoloy:
----------

[1] Basic Matrix Multiplication (3 nested for loops)

- QUARK implementation (almost done)
        > host + MIC
        > three different NB = 100,250,500
        > matrix size are increments of 500
- Standard C / MKL
        > in progress

[2] DGEMM

- PLASMA tiled
        > host + MIC
- Intel MKL
        > host, offload, and MIC
        > MIC environment variables are crucial for optimization

[3] CHOL (spotrf)

- Intel MKL
        > host, offload, and MIC
        > same deal with MIC environment variable

[4] In Progress --- Optimize Quark


General Execution:
------------------
There are a number of ways to run a program:

[1] bash script

The first method is self-explanatory and can be used to
configure the environment variables. These directories
use plenty of them, which end in ".sh".

With bash scripts, a basic knowledge of the
Portable Batch System (PBS) documentation in order
for further configurations such as the duration
```

Conclusions

- QUARK implementation needs to be optimized to better utilize the MIC's computational power.
- Given the range of 500:15000 at steps of 500 for the PLASMA DGEMM trial, increasing the tile size yielded better performance but increasing the number of threads proved insignificant.
- As expected, Intel's optimized MKL performs 2.96x better than PLASMA's DGEMM on the MIC:

  828.96038 and 279.89 GFLOPS/s respectively
- After running a number of stress tests for the Intel MKL Cholesky factorization, the best result at <u>741.18587 GFLOPS/s</u> was attained by using the maximum number of available cores (OMP_NUM_THREADS=240) and organizing these cores in a compact manner (KMP_AFFINITY=compact).

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
- Dongarra, Jack, et al. *PLASMA Users' Guide Version 2.3*. Sept. 2010
- Kurzak, Jakub. PLASMA/QUARK and DPLASMA/PaRSEC tutorial: ICL UT Innovative Computing Laboratory.
- 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 were derived from Google Images or their respective source (i.e., Intel)

Acknowledgements

- Dr. Kwai Wong + Dr. Christian Halloy, the JICS CSURE REU coordinators
- University of Tennessee for access to Star1 and the NICS conference rooms
- Oak Ridge National Laboratory for remote access to Beacon
- Ben Chan, Dr. Kwai Wong (UT), Dr. Asim YarKhan (UT), Dr. Eduardo D'Azevedo (ORNL), Dr. Shiquan Su (NICS), and XSEDE Help for all the documentation and help to progress in this research
- The National Science Foundation (NSF) for funding the CSURE Program

Questions?

- For questions about QUARK or PLASMA, please contact Dr. YarKhan
- For questions about the CSURE program, please contact Dr. Wong.
- For questions about how this research was conducted, please contact Allan Richmond (arrm93@gwu.edu) or Terrence (tc92321@hotmail.com)
- For fast troubleshooting help for Beacon, the Intel Xeon Phi System, or general supercomputing tips, contact XSEDE help email “help@xsede.org”