

A Distributive Linear Algebra Approach for Scalable Computing Platform with Accelerator

Shiquan Su, University of Tennessee, USA;

Ki Sing Chan, The Chinese University of Hong Kong, Hong Kong

Ed D'Azevedo, Oak Ridge National Laboratory, USA;

Kwai L. Wong, University of Tennessee and Oak Ridge National Laboratory, USA;

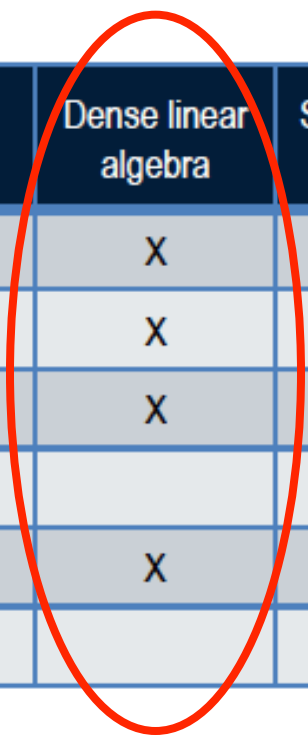
**Research sponsors: the U.S. National Science Foundation
the U.S. Department of Energy**



U.S. DEPARTMENT OF
ENERGY

Office of
Science

Algorithmic Coverage of Apps



Code	FFT	Dense linear algebra	Sparse linear algebra	Particles	Monte Carlo	Structured grids	Unstructured grids
S3D		X	X	X		X	
CAM	X	X	X	X		X	
LSMS		X					
LAMMPS	X			X			
Denovo		X	X	X	X	X	
NRDF			X				X (AMR)

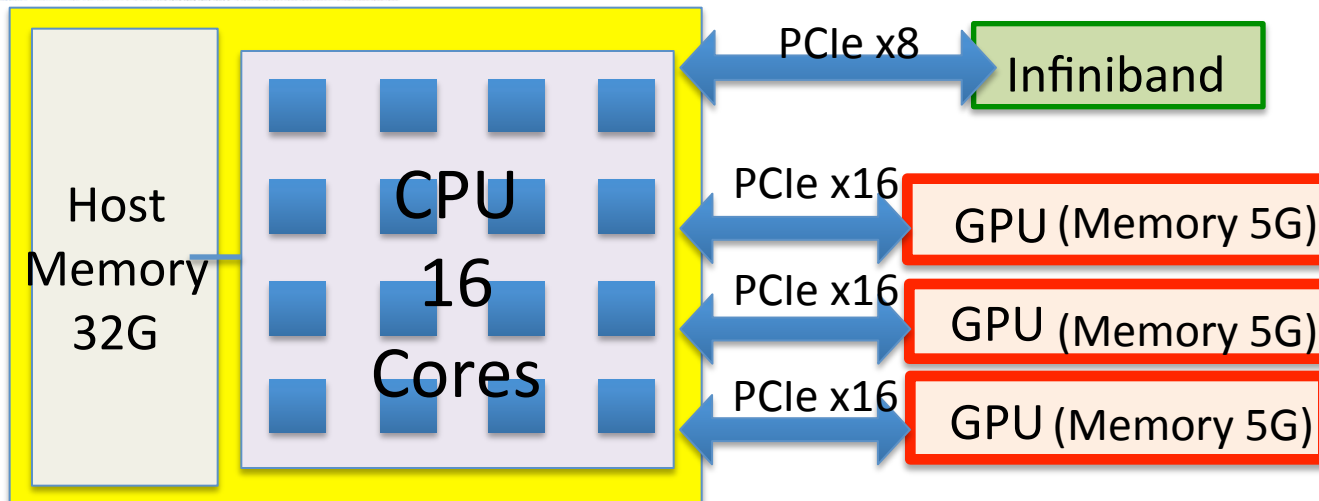
In the following, we use an example of performing the Cholesky decomposition to demonstrate how to take the advantage of accelerator (GPU/MIC) to solve dense linear algebra problem.

Keeneland System:

The Keeneland project partners : GATech, NICS, ORNL, UTK.
The Keeneland consists of **264** HP SL250G8 compute nodes.



- **two** 8-core Intel Sandy Bridge (Xeon E5) processors,
- **three** NVIDIA M2090 GPU accelerators,
- with Mellanox FDR **InfiniBand** interconnect,
- full **PCIeG3 x16** bandwidth to all GPUS.



MIC on XSEDE resources: Stampede



Dell PowerEdge C8220 Cluster with Intel Xeon Phi coprocessors

Peak Performance: 2+ PF (compute cluster), 7+ PF (coprocessors)

WORLD RECORD! "Beacon" at NICS

Intel® Xeon® + Intel Xeon Phi™
Cluster

First to Deliver
2.499 GigaFLOPS / Watt
71.4% efficiency
#1 on current Green500

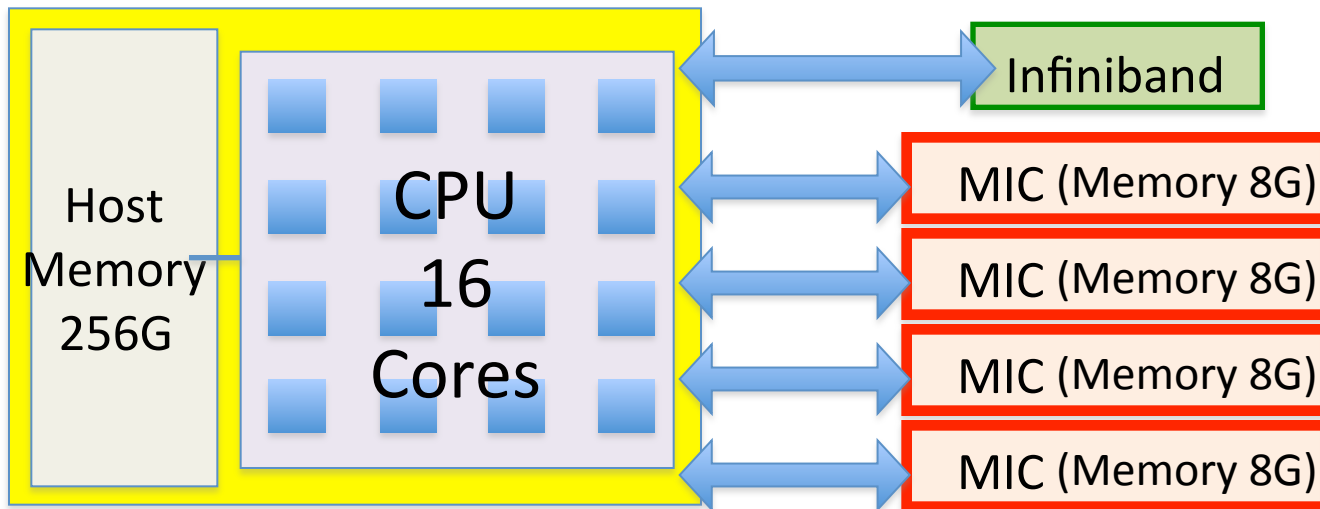


Beacon System:

The Beacon project partners : UTK, NICS, APPRO, INTEL.

The Beacon consists of **48** Appro GreenBlade GB824M compute nodes.

- **two** 8-core Intel Xeon E5-2670 processors,
- **four** Intel Xeon Phi Coprocessor 5110P,
- **256G** CPU memory,
- with FDR **InfiniBand** interconnect,
- full access to all MICs from each core.



General Steps of Cholesky Decomposition:

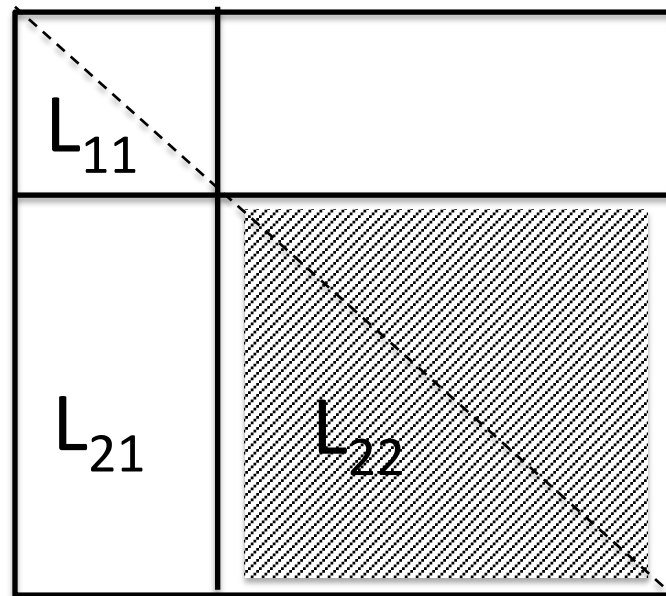
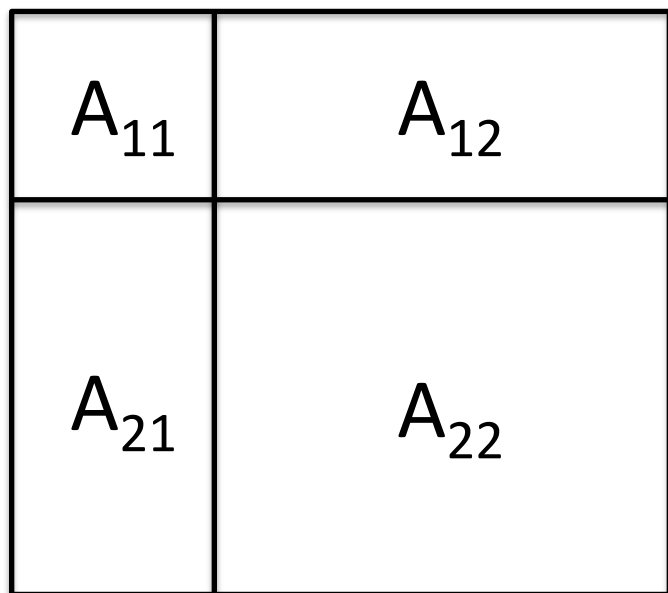
Right-looking Update:

Step 1: $L_{11} \leftarrow \text{cholesky}(A_{11})$, cholesky factorization

Step 2: $L_{21} \leftarrow A_{21} / L_{11}'$, triangular solve using dtrsm()

Step 3: $A_{22} \leftarrow A_{22} - L_{21} * L_{21}'$, symmetric rank-k update using dsyrk()
(Right-looking)

Step 4: $L_{22} \leftarrow \text{cholesky}(A_{22})$, cholesky factorization.



General Steps of Cholesky Decomposition:

Left-looking Update:

Step 1: $A_{n:} \leftarrow A_{n:} - \sum_k (L_{k:} * L_{k:}')$, (Left-looking)

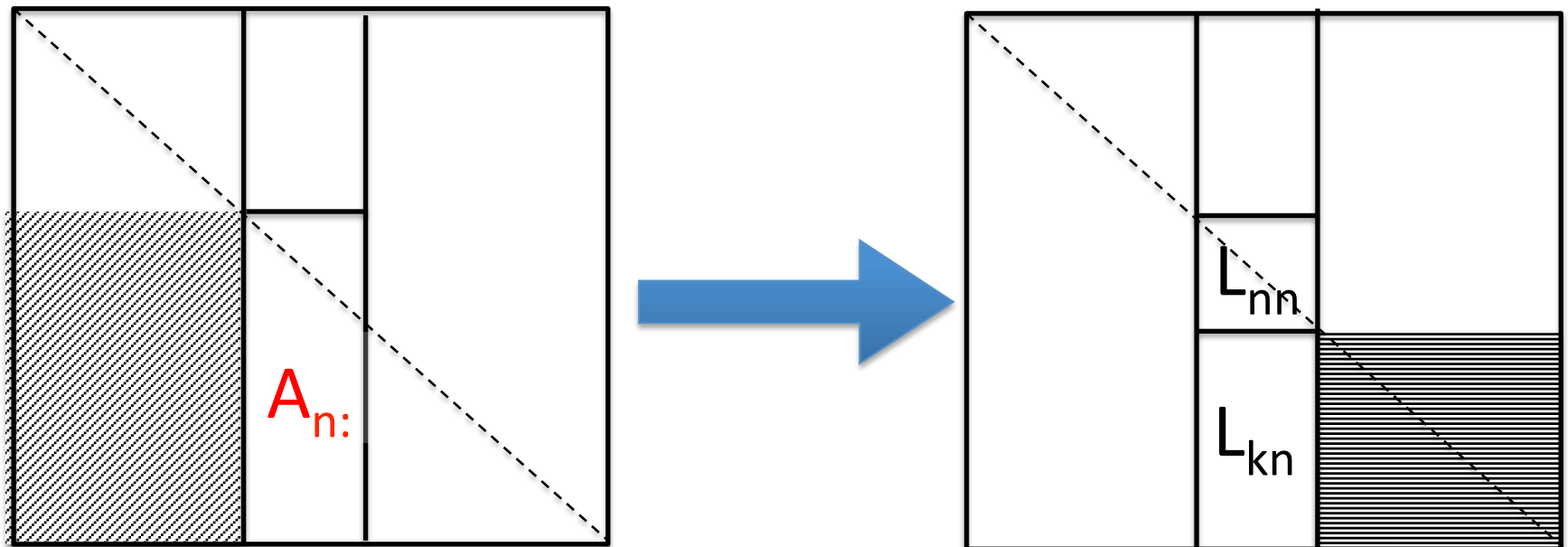
diagonal part: symmetric rank-k update using dsyrk()

off-diagonal part: matrix multiplication update using dgemm()

Step 2: $L_{nn} \leftarrow \text{cholesky}(A_{nn})$, cholesky factorization

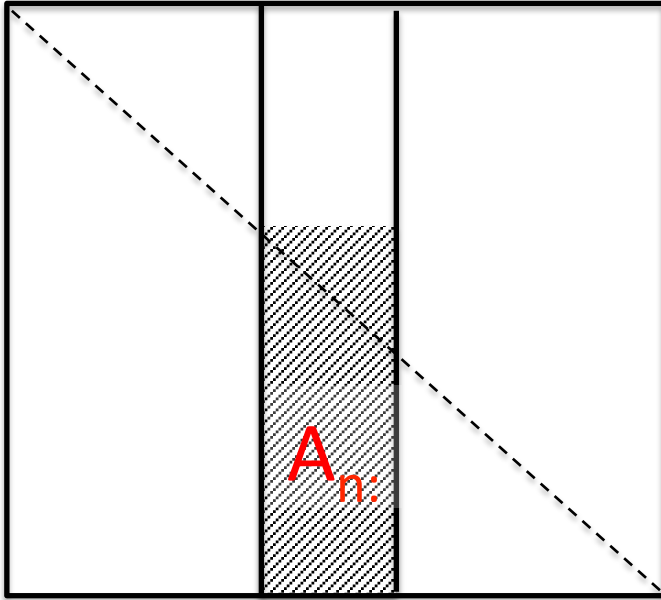
Step 3: $L_{kn} \leftarrow A_{kn} / L_{nn}'$, triangular solve using dtrsm()

Step 4: $L_{n+1:} \leftarrow \text{cholesky}(A_{n+1:})$, cholesky factorization.

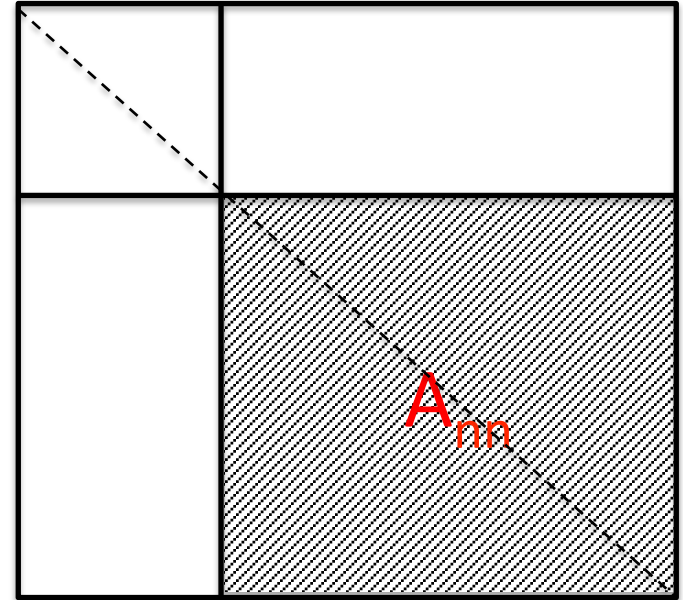


Locality vs. Parallelism

left looking update vs. right looking update



- Left-looking: the update is localized within the current tile of the matrix.
- Increase locality,
Reduce communication



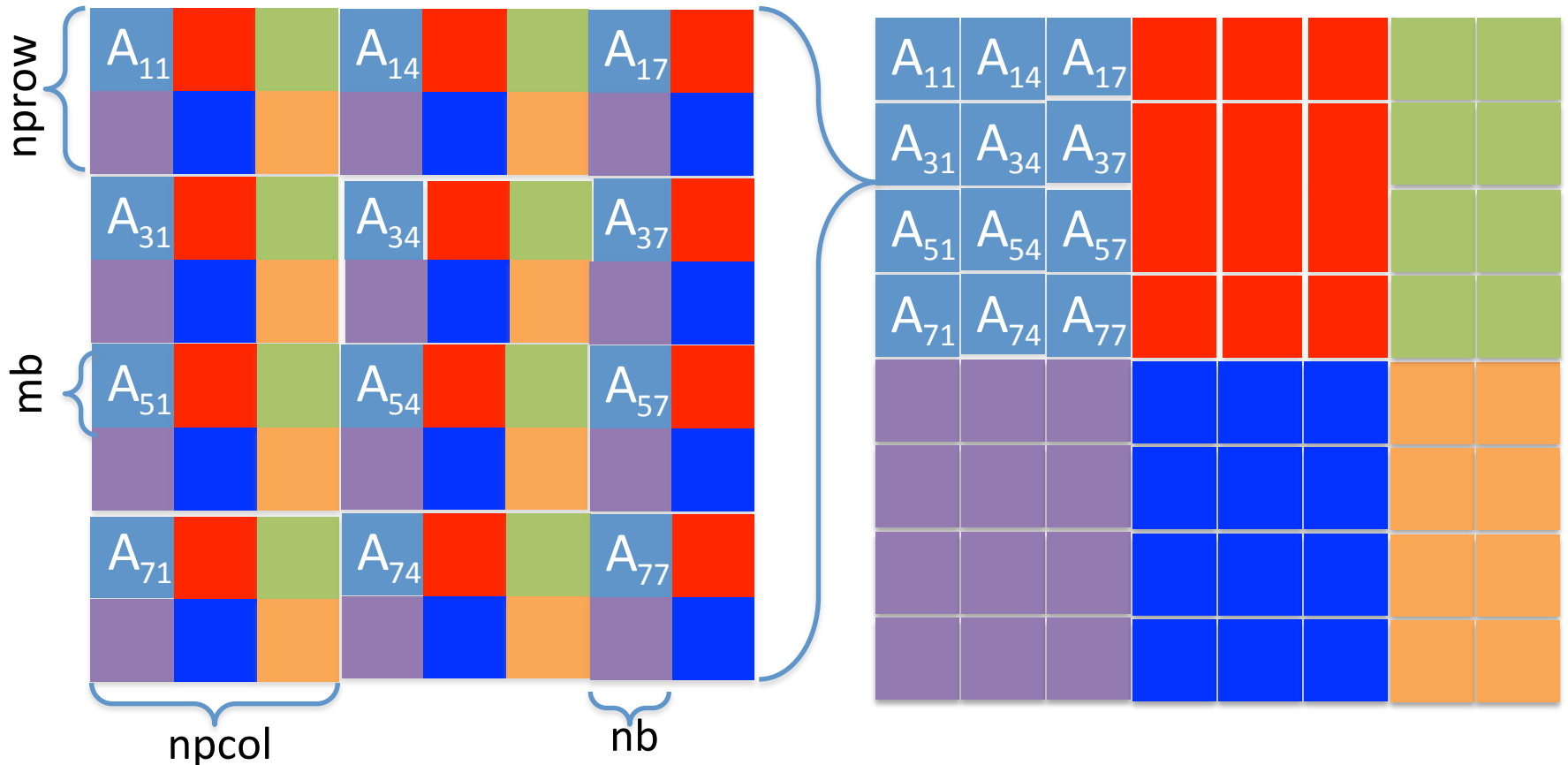
- Right-looking: the update can be performed in parallel
- Increase parallelism
Increase Flops

We combine both advantages in our scheme.

Cyclic Decomposition of the Distributed Matrix:

Matrix A is decomposed by a grid of blocks (left figure).

Consider a $2 \times 3 = \text{nproW} \times \text{ncol}$ processor grid, NB is the size of the block used in scalapack, assigned by input, usually 512 on our machine.

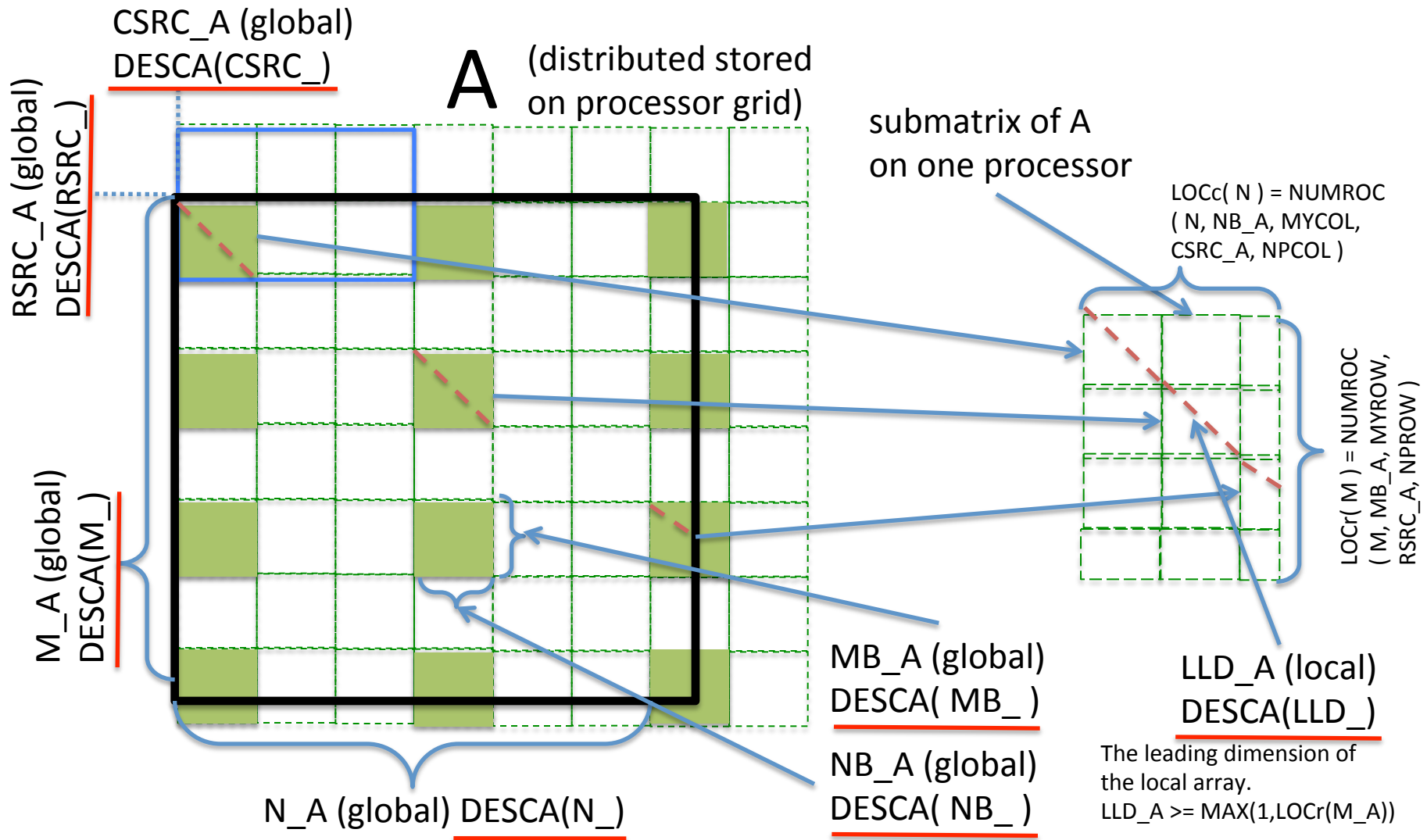


- Each core has a coordinate $(\text{myproW}, \text{mypcol})$ in the processor grid, and owns an A_{color} matrix. Also a piece of MIC memory is assigned to each core.

Matrix description vector:

DTYPE_A(global) DESCA(DTYPE_)The descriptor type. In this case, DTYPE_A = 1.

CTXT_A (global) DESCA(CTXT_) The BLACS context handle, indicating the BLACS process grid A is distributed over. The context itself is global, but the handle (the integer value) may vary.



OOC: “Out of Core” Approach:

Goal:

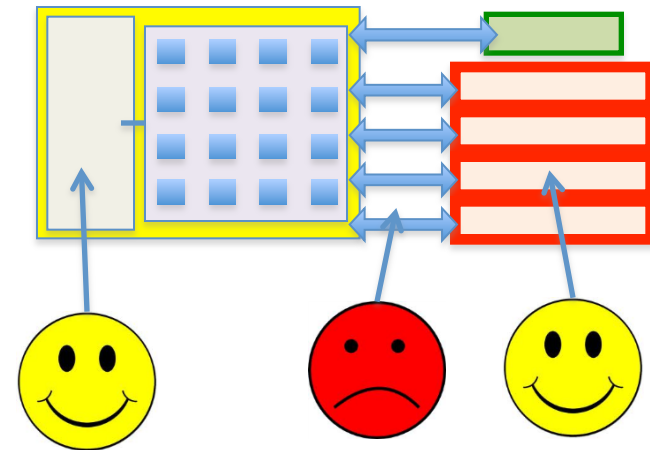
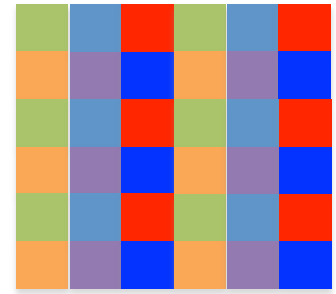
- Cholesky factorize of a large SPD matrix, which is larger than the MIC memory size.

Features:

- Fully utilize all the memory resources on CPU to solve the large size matrix.
- Perform most of the heavy computation load on the high throughput MIC device.
- Minimize the data transfer between CPU and MIC memory.

Structure:

- *Out-of-core Part*: load part of the matrix to the device memory, apply the update from the part already factorized. (left-looking)
- *In-core Part*: factorize the sub-matrix residing on device memory. (right-looking)
- *Host-Host-Device Update*: broadcast the source matrix on CPU memory via MPI, DSYRK update the target matrix on MIC memory.



Function Call of the OOC Approach Package (GPU version):

pdpotrf_ooc2 (args. list + GPU memsize)

(The API is very similar to pdpotrf (args. list) in scalpack)

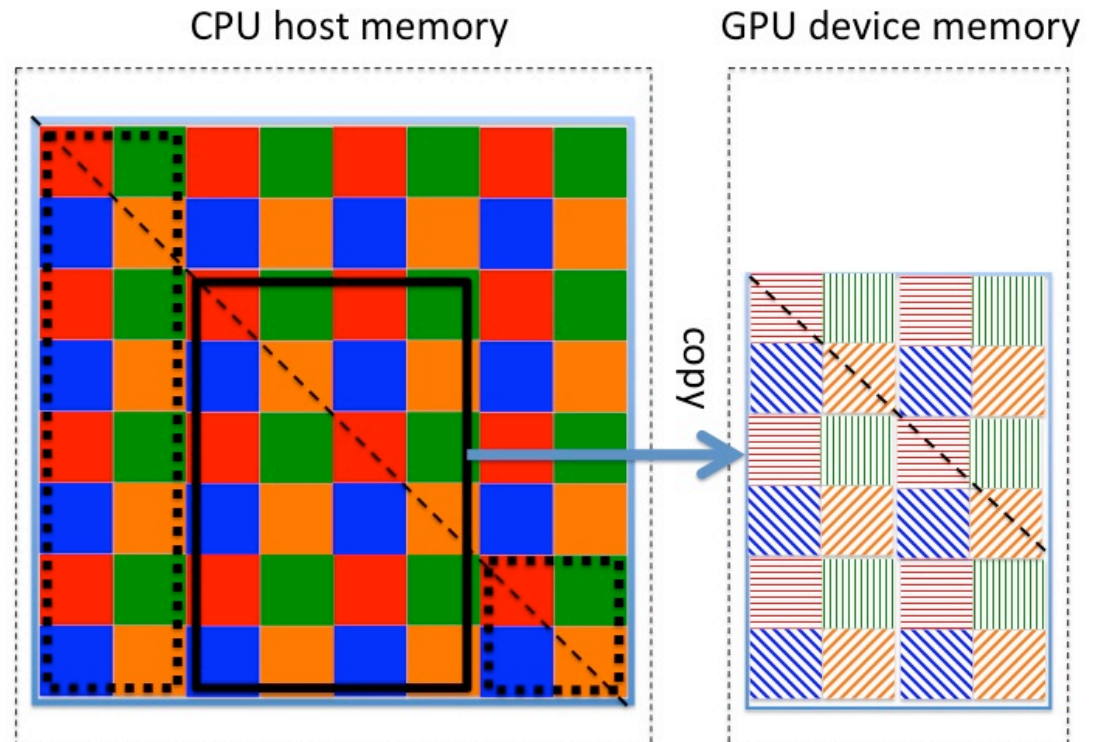
- Memory arrangement (CUDA subroutines)
- Implement BLOCK CYCLIC decomposition for the matrix (BLACS subroutines)
- Cholesky factorize in core submatrix (pdpotrf_gpu2)
- Host-Host-Device update the matrix (Cpdsyrk_hhd)

Out-of-core Part I:

The matrix A is divided in multiple Y panels, each Y panel can fit in the GPU device memory. The Y panel on the device memory is distributively stored on multiple GPUs device memory across the whole processor grid .

Call `cublasSetMatrix` to copy from CPU to GPU

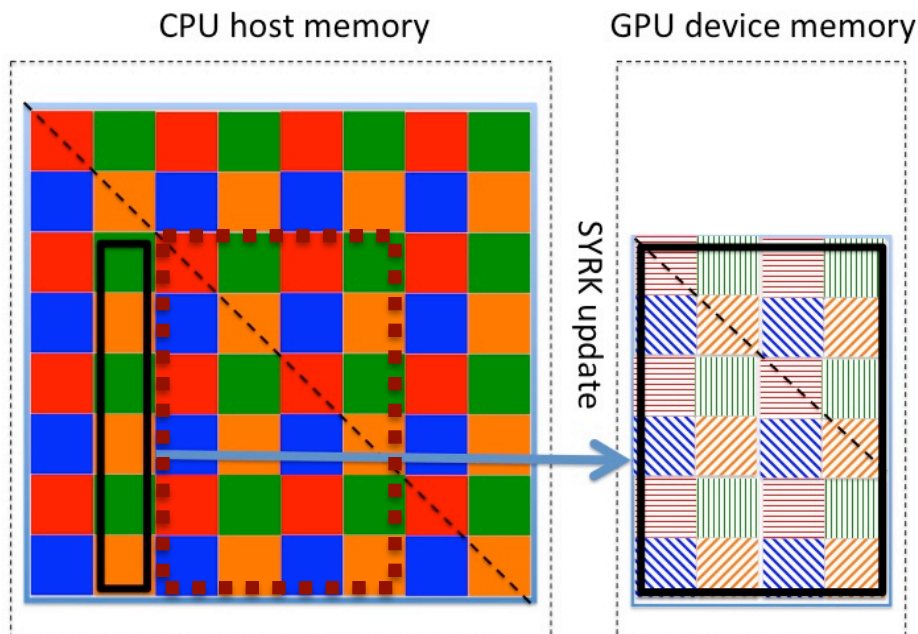
If the data is not aligned, call the subroutines in scalapack to redistribute the data on different MPI tasks.



Out-of-core Part II:

- We use a left-looking scheme. Apply the update from the factorized part L of matrix A to the current Y panel.
- The factorized part L is divided into X stripe, labeled by k, the width of X stripe is the block width NB.
- The update is applied stripe by stripe. We use a Host-Host-Device version of DSYRK update.

- The m, n is the index of matrix A, within the range of current Y panel.
- The k labels the X stripe in the factorized part.
- The l runs over the width of the X stripe.



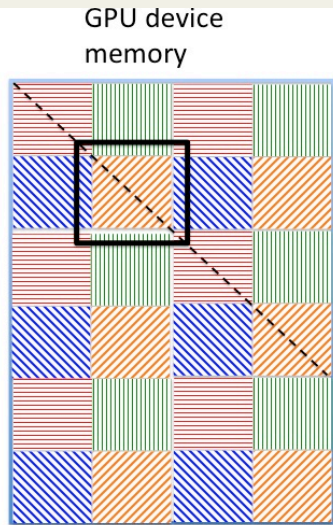
compared to left-looking general step 3:

$$A_{22} \leftarrow A_{22} - L_{21} * L_{21}'$$

$$\underline{A}_{mn} = A_{mn} - \sum_k \left(\sum_l L_{ml} * L_{ln}^T \right)_k$$

In-core Part I:

Now we factorize the panel residing on the GPU memory. The panel is divided into X stripe, the width is the block size NB. We factorize stripe by stripe.

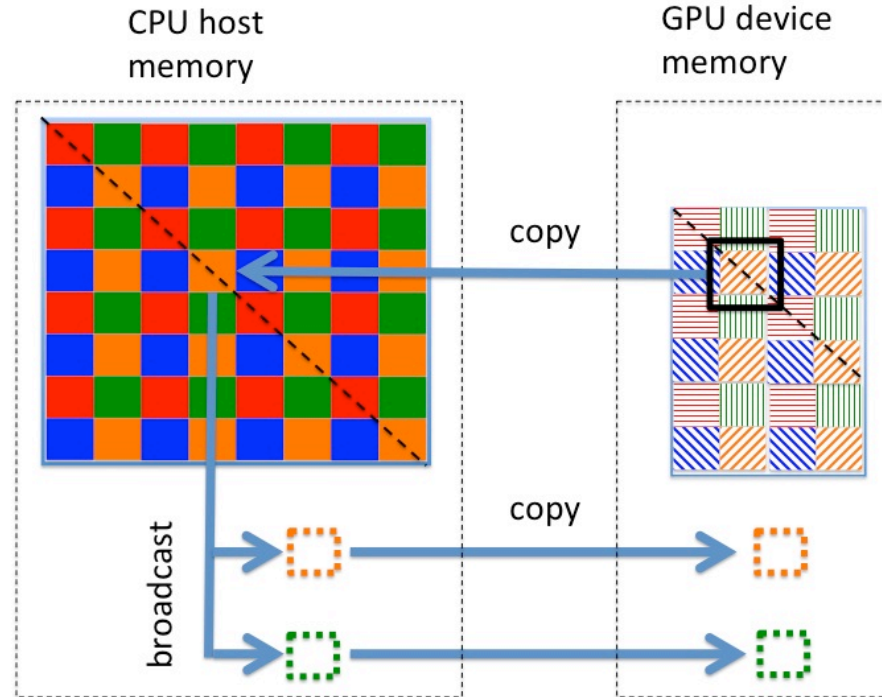


$$A_{nn} = L_{nn} * L_{nn}^T$$

compared to right-looking general step 1:

$$L_{11} \leftarrow \text{cholesky}(A_{11})$$

Call the *dpotrf* subroutine in MAGMA to Cholesky factorize diagonal block belong to the current stripe.



Call *cublasGetMatrix* to copy the L_{nn} from GPU to CPU.
Call *pdgeadd* to broadcast the L_{nn} to all the cores on the same column of the processor grid.
Call *cublasSetMatrix* to copy the L_{nn} to the GPU device memory associate to to the core.

In-core Part II:

Triangular solve the rectangle matrix under the diagonal block.

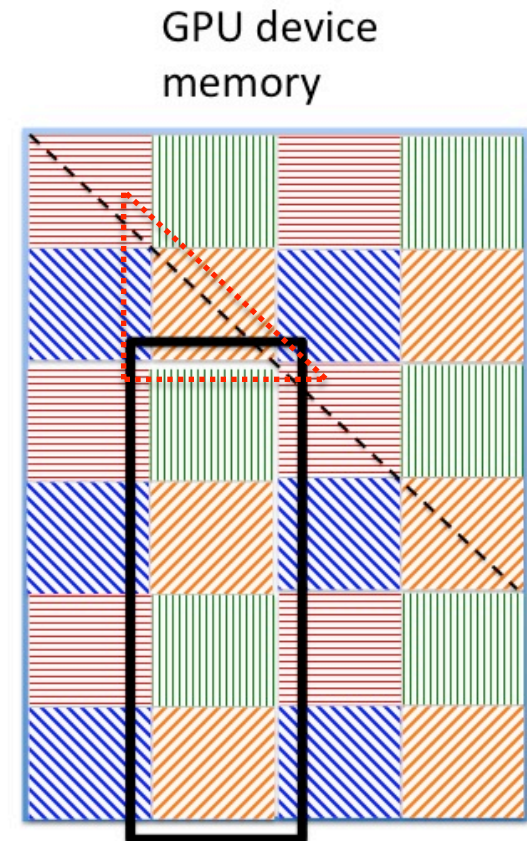
We call the *DTRSM* subroutine from MAGMA to perform the triangular solve.

$$A_{mn} = \sum_{l=1}^{NB} \underline{L_{ml}} * L_{ln}$$

compared to right-looking
general step 2:

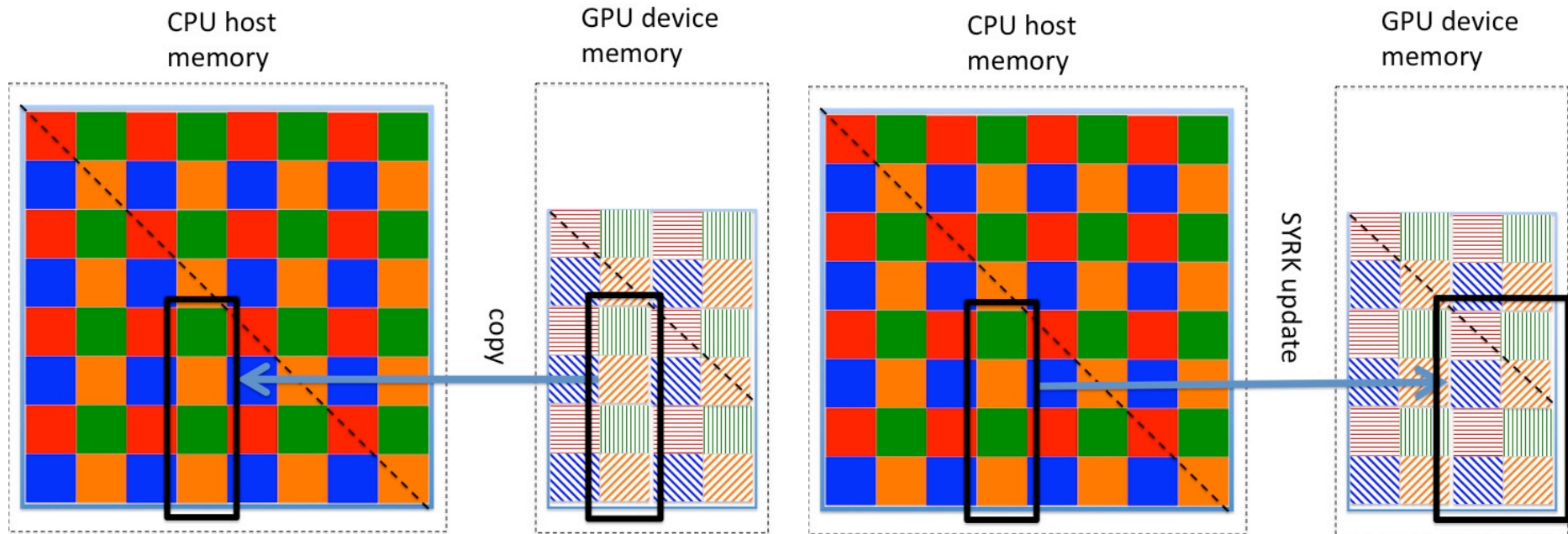
$$L_{21} \leftarrow A_{21} / L_{11}'$$

- The m, n is the index of matrix A , within the range of rectangle matrix under the diagonal block..
- The l runs over the width of the X stripe.



In-core Part III:

- We follow a right-looking scheme to update the rest of the panel.
- We use a Host-Host-Device version of DSYRK update.



compared to right-looking general step 3:

$$(A_{22} \leftarrow A_{22} - L_{21} * L_{21}')$$

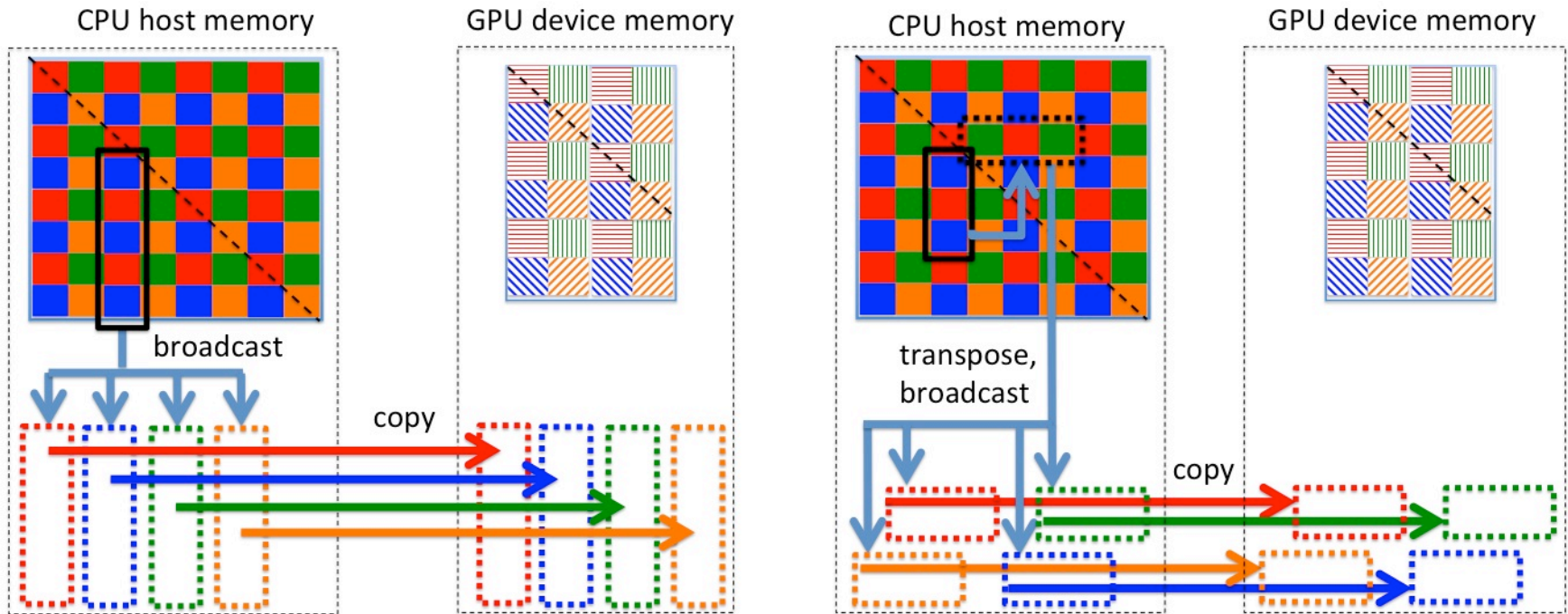
$$\underline{A}_{mn} = A_{mn} - \sum_l L_{ml} * L_{ln}^T$$

Call the *cudaGetMatrix* subroutine to copy the rectangular matrix from GPU to CPU.

- The m, n is the index of matrix A, within the range of the rest of the panel.
- The l runs over the width of the X stripe.

Host-Host-Device Update I:

- The source matrix is sitting on host memory and the target matrix is sitting on device memory.
- We need to use MPI on the CPU to distribute the source matrix to the correct cores in the processor grid.



Call *pdgeadd* to broadcast the L_{m1} to every core in the processor grid.

Call *cublasSetMatrix* to copy the L_{m1} to the GPU device memory associate to the core.

Call *pdgeadd* to transpose and broadcast L_{In} to every core in the processor grid.

Call *cublasSetMatrix* to copy the L_{In} to the GPU device memory associate to the core.

Host-Host-Device Update II:

Recursively, divide the matrix A into triangular matrix along the diagonal and rectangular matrix at the off-diagonal area.

Call the DSYRK in CUBLAS to calculate the diagonal part, labeled by the black triangular.

$$\underline{A}_{nn'} = A_{nn'} - \sum_l L_{nl} * L_{ln'}^T$$

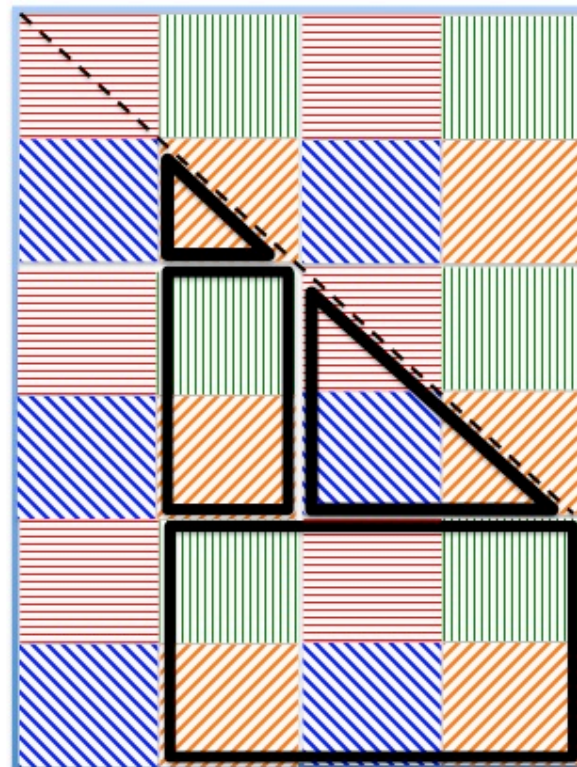
Call the DGEMM in CUBLAS to calculate the off-diagonal part, labeled by the black rectangular.

$$\underline{A}_{mn} = A_{mn} - \sum_l L_{ml} * L_{ln}$$

compared to right-looking general step 3:

$$A_{22} \leftarrow A_{22} - L_{21} * L_{21}'$$

GPU device memory



Allocate/free MIC memory

- Convert pointer type to and from unsigned integer type. Store memory address on device as an unsigned integer.
- Allocate and free memory within pragma offload region using memalign() and free().

Interface:

```
intptr_t offload_Alloc(size_t size){
    intptr_t ptr;
    #pragma offload target(mic:MYDEVICE) out(ptr)
    {
        ptr = (intptr_t) memalign(64, size);
    }
    return ptr;
}

void offload_Free(void* p){
    intptr_t ptr = (intptr_t)p;
    #pragma offload target(mic:MYDEVICE) in(ptr)
    {
        free((void*)ptr);
    }
}
```

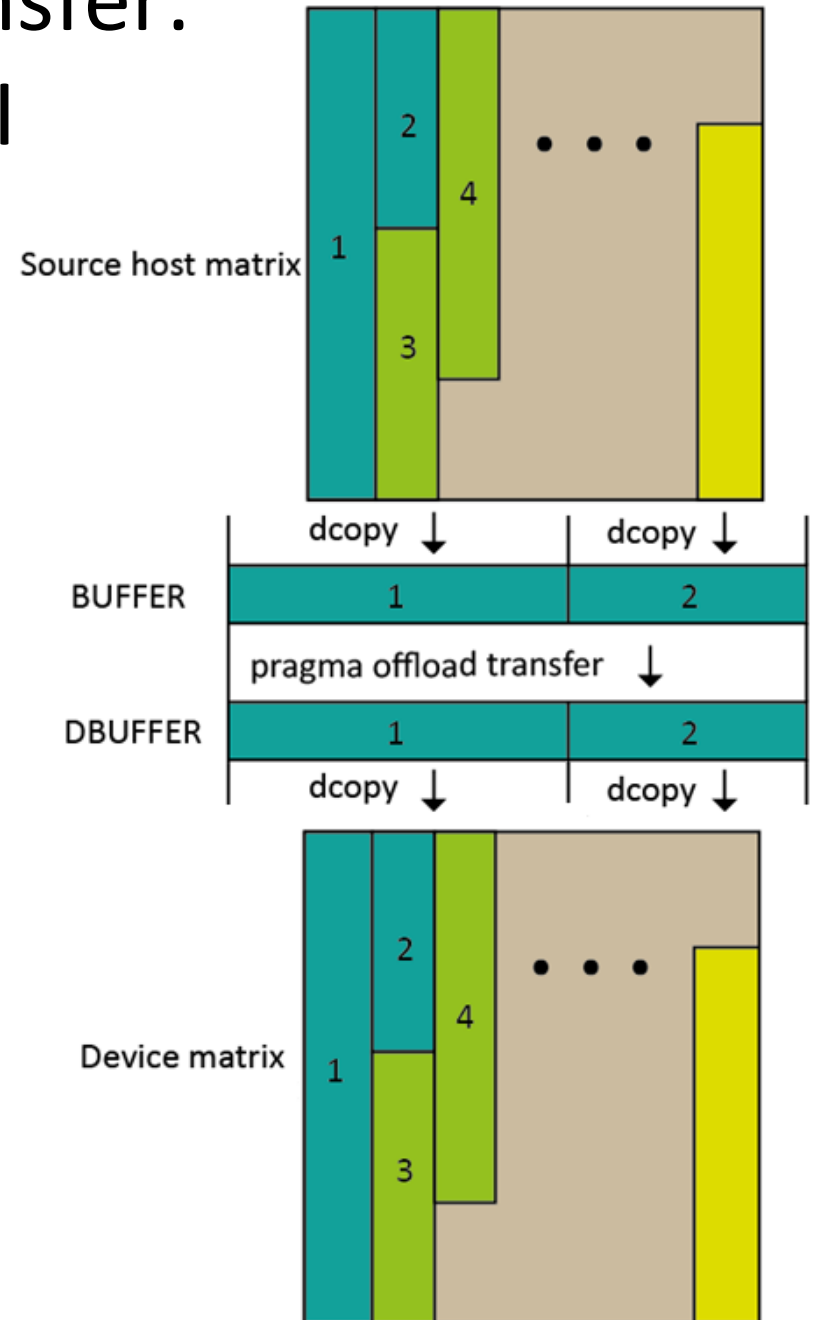
usage:

```
#ifdef USE_MIC
    dY = (double*) offload_Alloc(ysizeY*elemSize);
#else
    cublasAlloc( izeY, elemSize, (void **) &dY );
#endif
...
if (dAtmp != 0) {
    #ifdef USE_MIC
        offload_Free(dAtmp);
    #else
        CUBLAS_FREE( dAtmp );
    #endif
    dAtmp = 0;
};
```

Host-device data transfer: ooc_dSetMatrix I

➤ Allocate fixed size buffers on host and on device. (16 MB)

1. Divide the source matrix in smaller continuous memory blocks.
2. Call dcopy in MKL to pack the data into host buffer
3. Pragma offload transfer the packed data into device buffer.
4. Unpack buffer into target device matrix.



BLAS level 2,3 function calls

```
CUBLAS_DGEMM(  
    CUBLAS_OP_N, CUBLAS_OP_N, mm, nn, kk,  
    zalpha, (double *) dA(lrA1,lcA1), ldAtmp,  
            (double *) dB(lrB1,lcB1), ldBtmp,  
    zbeta, (double *) dC(lrC1,lcC1), ldc );
```



```
offload_dgemm("N", "N", &mm, &nn, &kk,  
    &zalpha, (double *) dA(lrA1,lcA1), &ldAtmp,  
            (double *) dB(lrB1,lcB1), &ldBtmp,  
    &zbeta, (double *) dC(lrC1,lcC1), &ldc );
```



```
void offload_dgemm(const char *transa, const char *transb, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k,  
    const double *alpha, const double *a, const MKL_INT *lda, const double *b, const MKL_INT *ldb,  
    const double *beta, double *c, const MKL_INT *ldc){  
/*  
 * perform dgemm on the device. a,b,c pre-exist on the device  
 */  
    intptr_t aptr = (intptr_t)a;  
    intptr_t bptr = (intptr_t)b;  
    intptr_t cptr = (intptr_t)c;  
    #pragma offload target(mic:MYDEVICE) in(transa,transb,m,n,k:length(1)) \  
        in(alpha,lda,ldb,beta,ldc:length(1))  
    {  
        dgemm(transa,transb,m,n,k,alpha,(double*)aptr,lda,(double*)bptr,ldb,beta,(double*)cptr,ldc);  
    }  
}
```

Numerical Results for MIC (Beacon)

- Typical Performance
- Effect of Different Block Size
- Arrangement of (P*Q)
- Performance of Increasing Matrix Size
- Strong Scaling Study
- Successful Attempt for Largest Matrix Size

Multithreading in MIC enabled

```
export MIC_ENV_PREFIX=MIC
export MIC_USE_2MB_BUFFERS=2M
export MIC_KMP_AFFINITY=explicit,granularity=fine,proclist=[1-236:1]
export KMP_AFFINITY=granularity=fine,compact,1,0
```

Performance shown in double precision GFLOPS

Typical Performance

- Matrix size close to host CPU memory
- 1 MPI task per MIC card

Processor Grid	Problem Size	Panels	Block Size	Nodes	Host Memory per Node (GB)	Performance per MIC (GFlops)
1x1	180,224	26	512	1	242.00	365.81
4x1	178,176	7	512	1	250.32	273.58
2x2	178,176	7	512	4	250.32	258.53
8x2	327,680	6	512	4	200.00	278.68
8x2	655,360	22	512	16	200.00	341.04

Effect of Different Block Size NB:

Problem size: 147456 x 147456
4 nodes, 40.5 GB per node,
4 MPI tasks per node,
4 x 4 processor grid,
2 panels

NB	Performance per MIC
128	10.67
256	37.49
512	90.09
1024	109.91
2048	81.81

Problem size: 147456 x 147456
9 nodes, 18 GB per node,
4 MPI tasks per node,
6 x 6 processor grid,
1 panels

NB	Performance per MIC
128	7.28
256	23.32
512	55.75
1024	69.47
2048	50.93

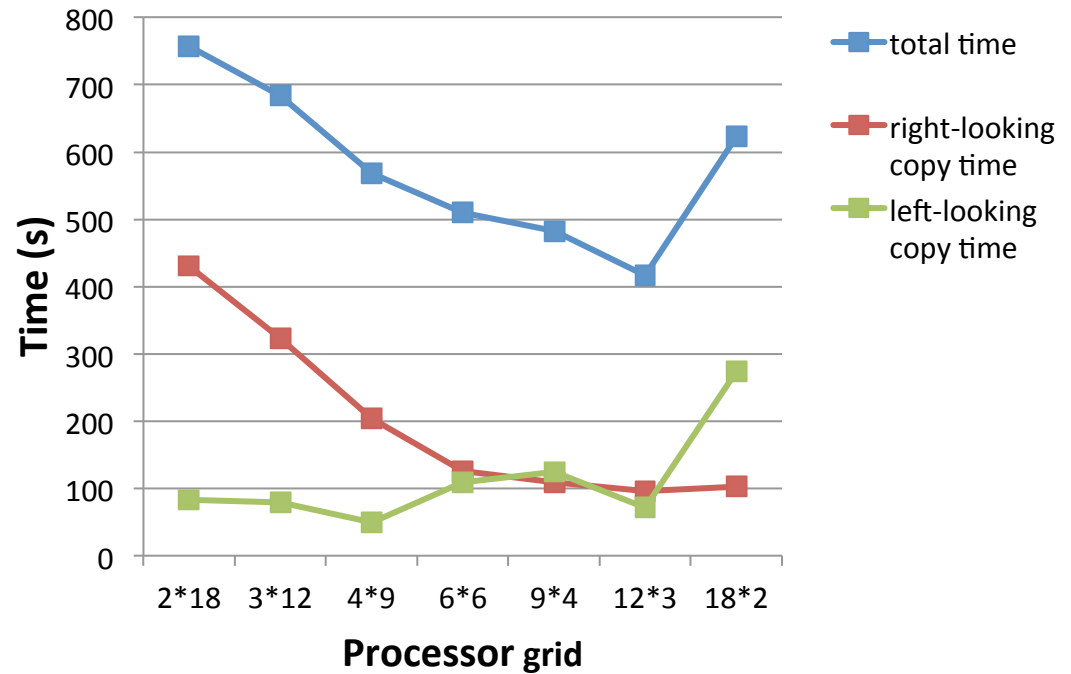
512 and 1024 are close to optimal.

Most runs use NB=512, unless mention explicitly.

Effect of Different P * Q:

Problem size: 147456 x 147456
9 nodes, 18 GB per node,
4 MPI tasks per node,
2 panels

P	Q	Performance per MIC
2	18	36.48
3	12	40.45
4	9	48.70
6	6	54.24
9	4	57.24
12	3	66.35
18	2	44.39



(Time perceived as (0,1) in the processor grid)

- Right-looking copy is more time-consuming when P is small.
- Left-looking copy is more time-consuming when Q is small.
- Computation time is not sensitive to P & Q when number of MPI tasks is unchanged.

Effect of Matrix Size N:

4 nodes, 1 MPI tasks per node
2 x 2 processor grid

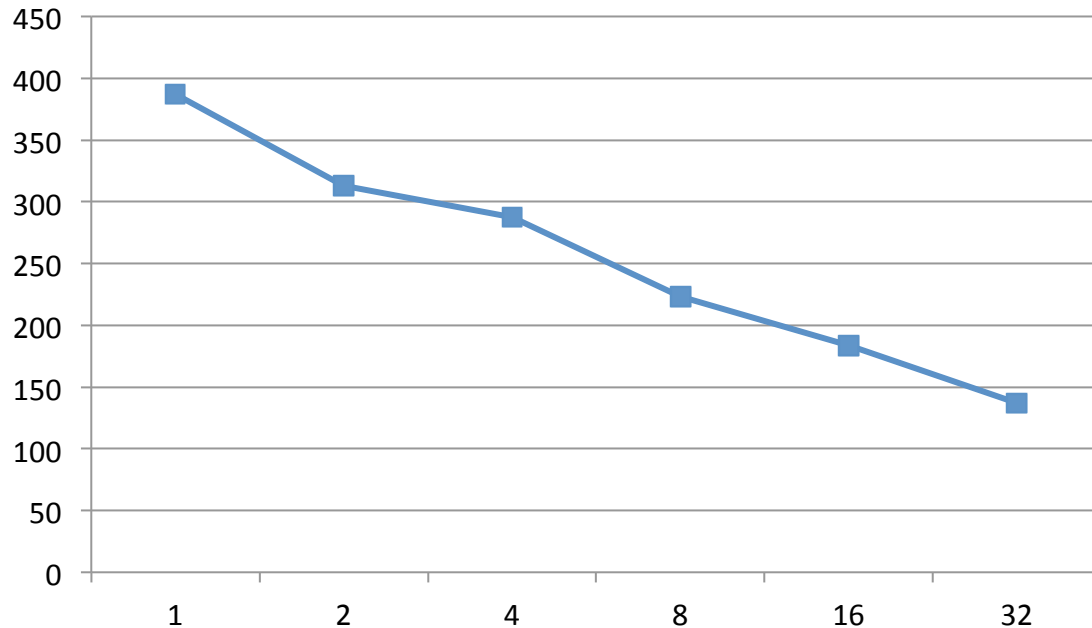
N	Host memory per node	Performance per MIC
92160	15.82	63.80
147456	40.50	134.98
192512	69.03	164.80
245760	112.50	197.15
348160	225.78	260.16

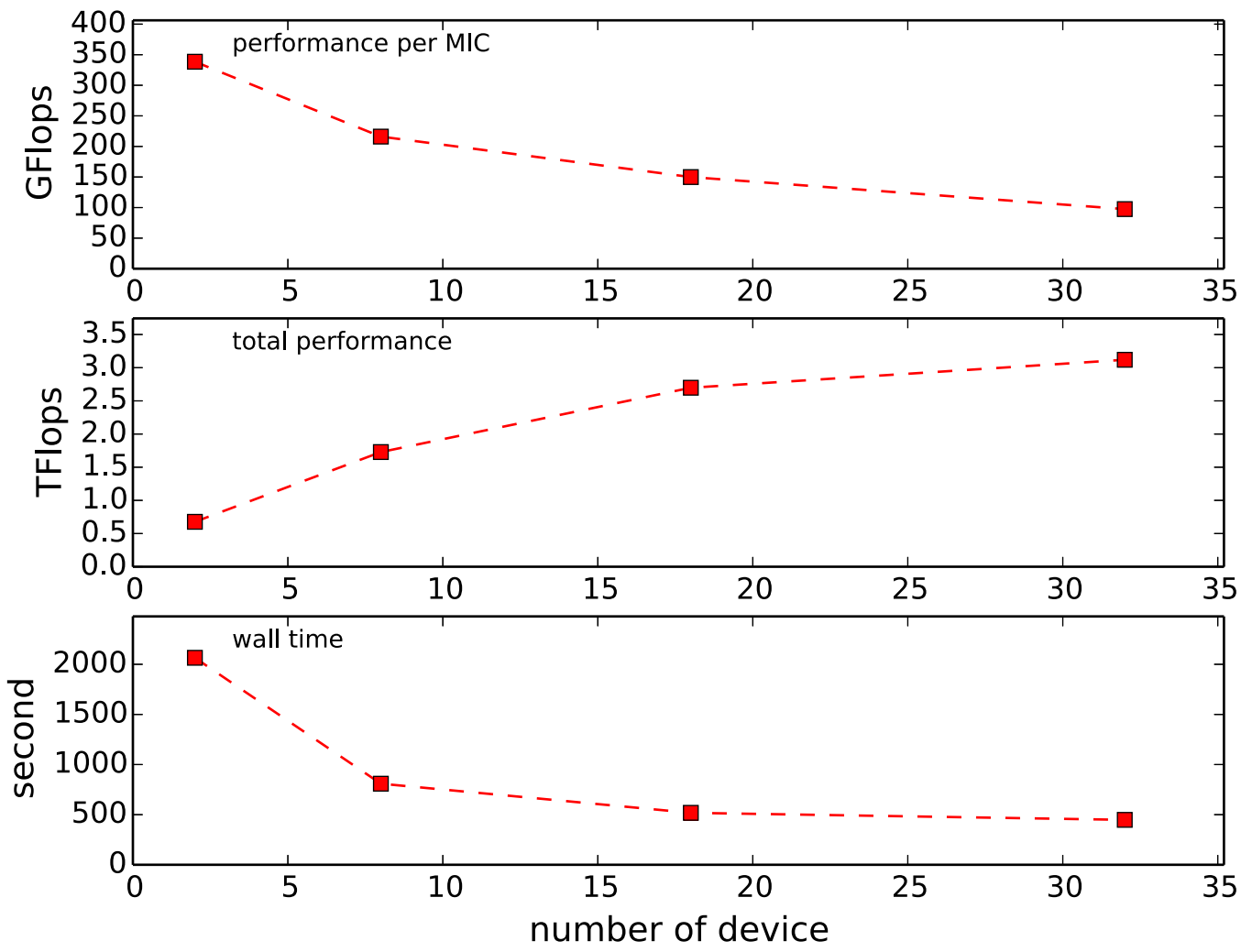
Obtain better performance as the matrix fill up more of the host CPU memory.

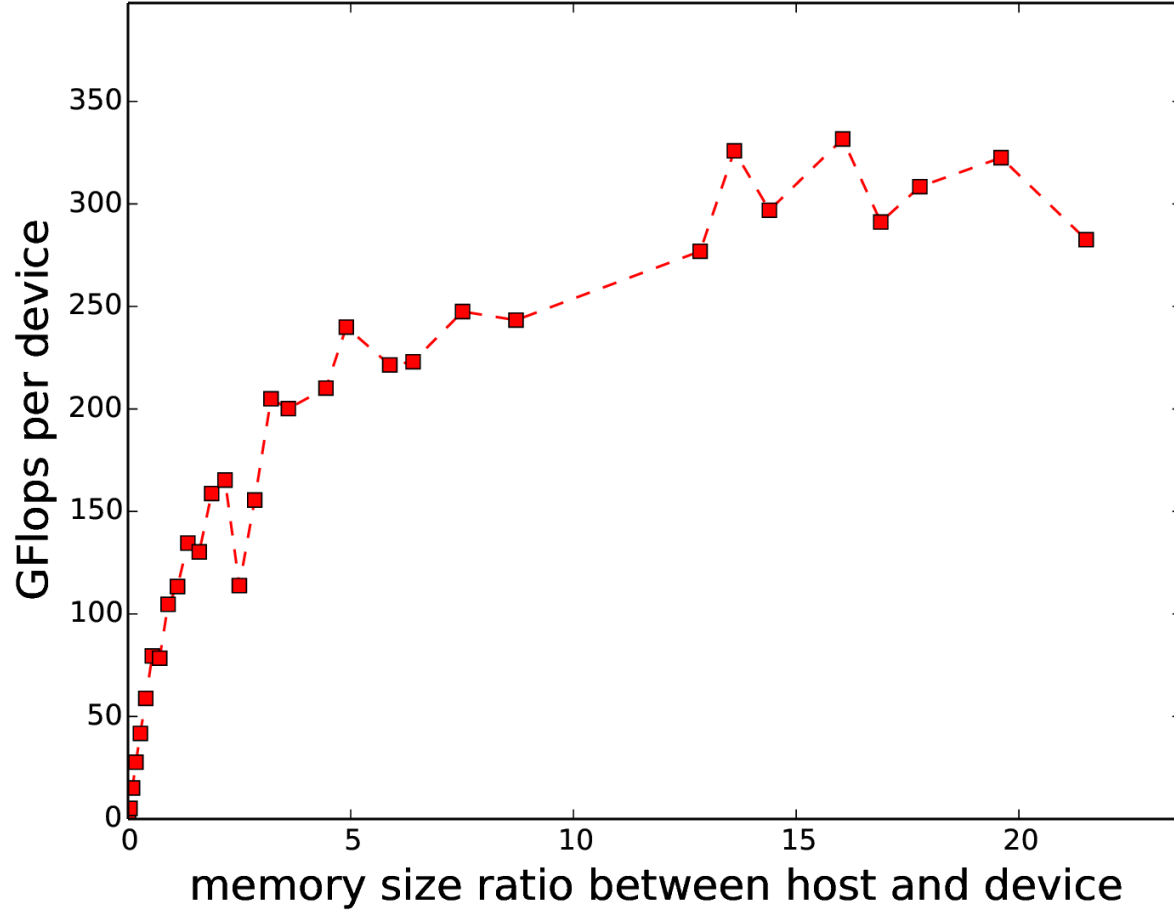
Strong Scaling Study:

Problem size: 178126 x 178126
1 MPI task per node,
occupy around 230GB host
memory per node

Nodes	Processor Grid	Panels	Time	Performance per MIC
1	1x1	22	4379.44	387.30
2	2x1	11	2712.65	312.64
4	4x1	7	1473.45	287.79
8	4x2	4	950.04	223.17
16	8x2	2	578.30	183.31
32	8x4	2	386.45	137.16







Successful Attempt for Largest Matrix Size:

Problem size: **983040 x 983040**
32 nodes, 225GB host memory

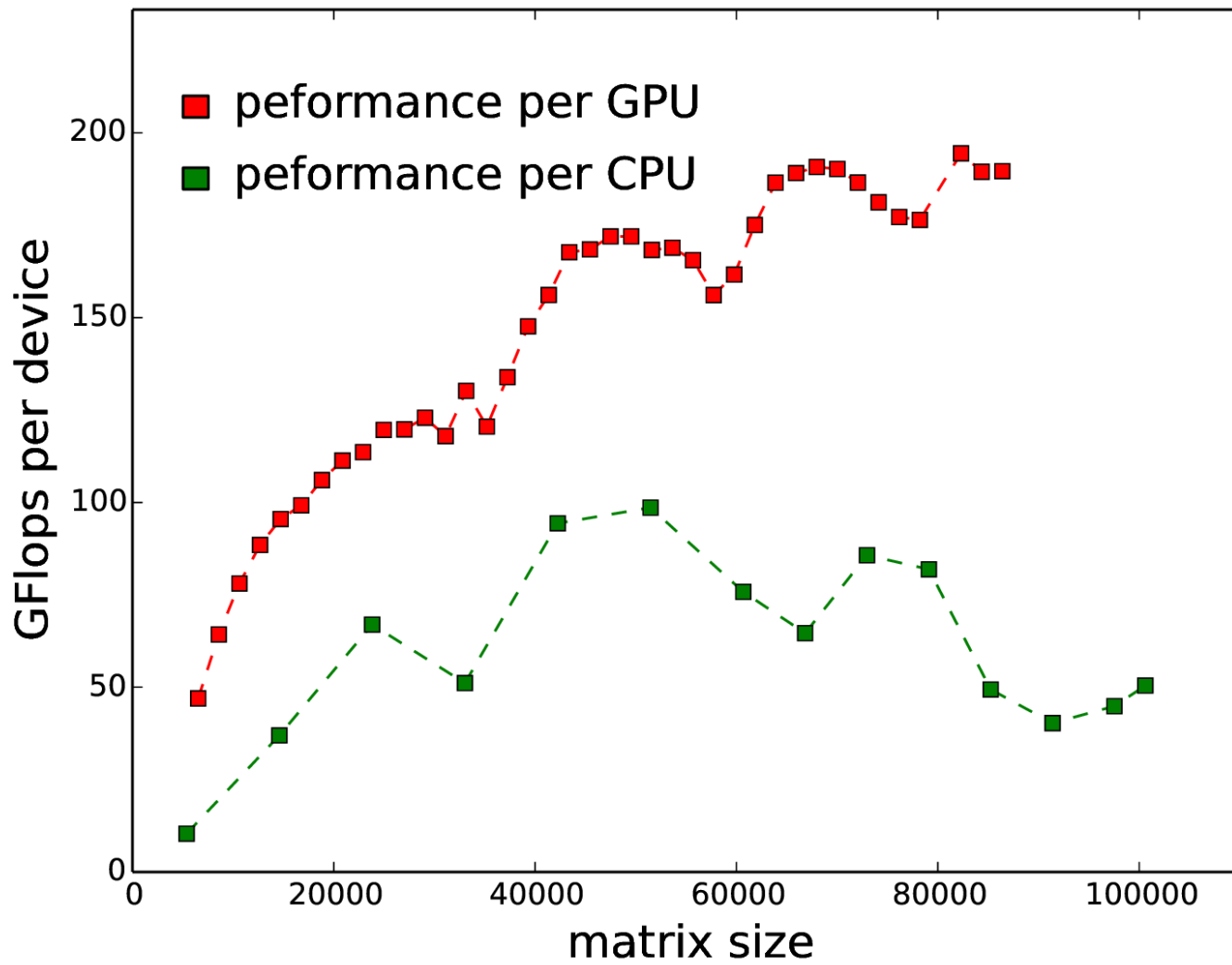
ppn	Processor Grid	Panels	Performance per MIC
2	16 x 4	12	246.42

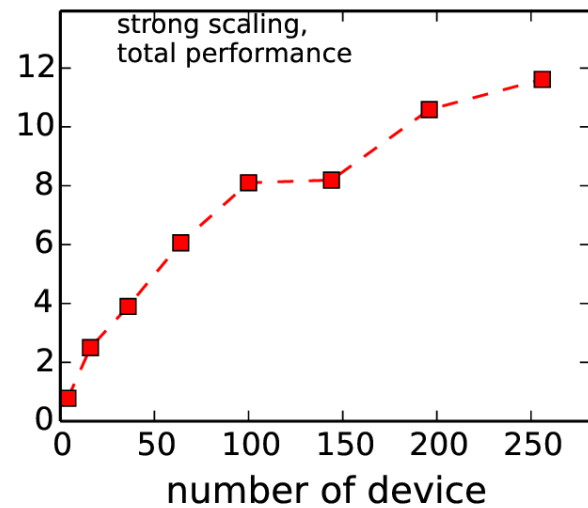
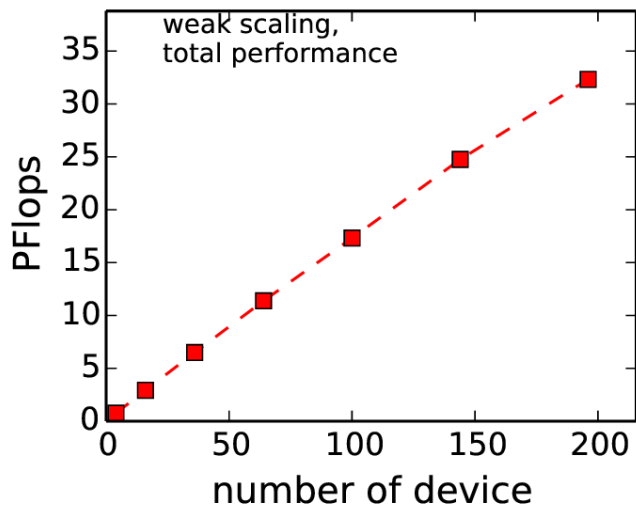
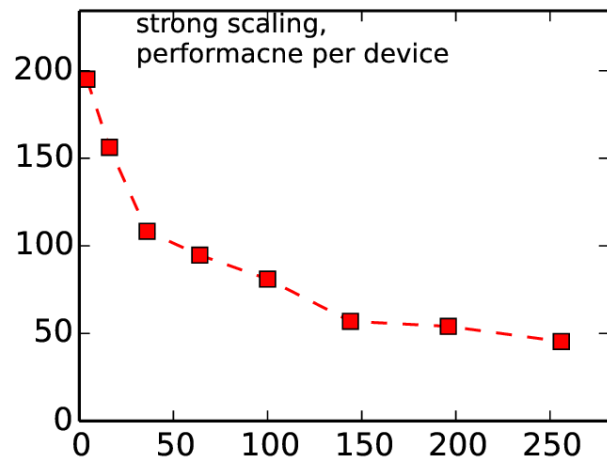
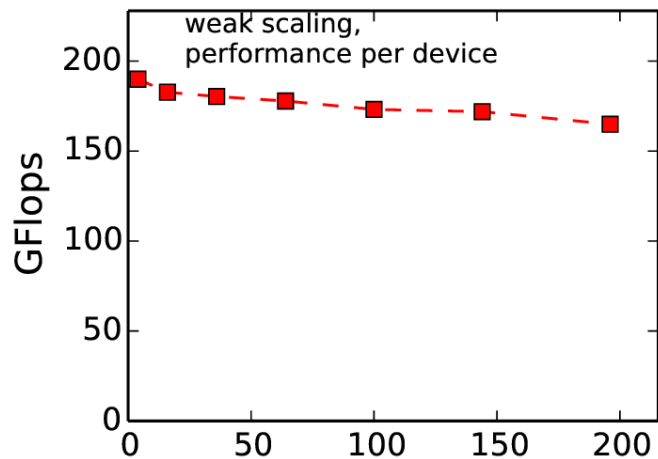
Summary for Obtaining Better Performance

For the best performance, we suggest:

- 512 or 1024 as block size
- Processes per node should not exceed the number of available MIC cards per node
- Processor grid $P : Q = 4 : 1$, it varies at different memory ratio
- For the same problem, try to use fewer number of node

Numerical Results for GPU (Keeneland)





Summary

- Parallel Cholesky factorization takes advantage of multiple accelerator device and is compatible with Scalapack.
- Out-of-core approach solves large problems and to minimize data movement between CPU and MIC.
- *DGEMM* is a major computational kernel and data movement and communication are major bottlenecks.