Out-of-Core Cholesky Factorization Algorithm on GPU and the Intel MIC Co-processors

Ben Chan, Nina Qian (Chinese University of Hong Kong)
Mentors: Ed D'Azevedo (ORNL), Shiquan Su (UTK), Kwai Wong (UTK)

Motivation: Radiosity Problem

Introduction

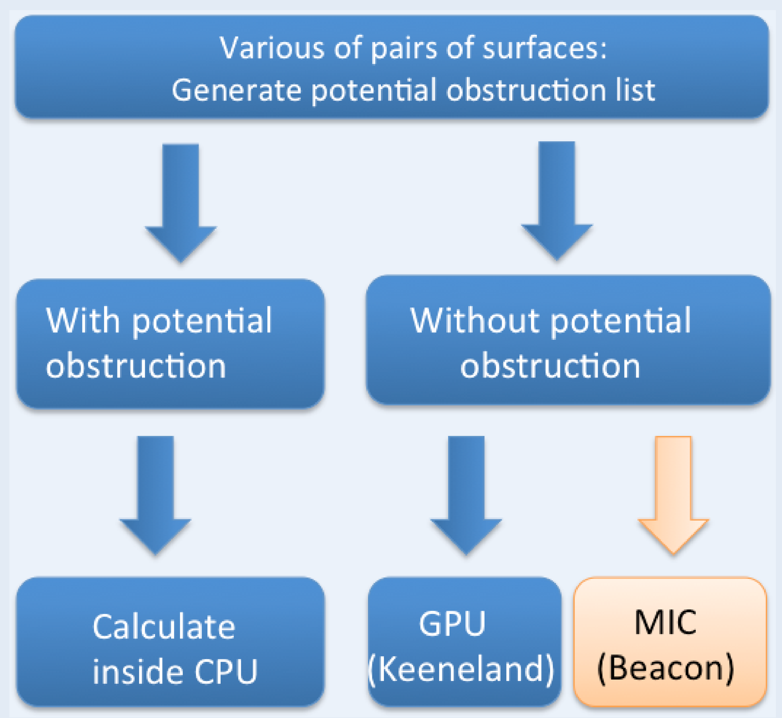
- Stepthen-Boltzmann's equation indicates the relation between the object temperature and emitted radiation which can presented by view factor matrix F. It can transform to radiosity matrix G.

$$G = \begin{pmatrix} \frac{A_1}{\phi_1} - A_1F_{11} & -A_2F_{12} & \cdots & -A_NF_{1N} \\ -A_1F_{21} & \frac{A_2}{\phi_2} - A_2F_{22} & \cdots & -A_NF_{2N} \\ \vdots & & & \vdots \\ -A_1F_{N1} & -A_2F_{N2} & \cdots & \frac{A_N}{\phi_N} - A_NF_{NN} \end{pmatrix}$$

Structure of radiosity matrix G: an SPD matrix

Parallel View3D Program

- Parallel generation of the view factor matrix F based on Host-Device architecture.
- The Device for Keeneland and Beacon are GPU and MIC respectively.



Implementation from GPU to MIC

- MIC implementation: Assign one MIC card to each core; Use implicit offload model (offload with shared virtual memory)

GPU	MIC
Allocation of data in Device	
`cudaMalloc(&DEV_CUDA_srf,size);` `cudaMalloc(&DEV_ans,size);`	`DEV_MIC_srf= _Offload_shared_malloc(size);` `DEV_ans= _Offload_shared_malloc(size);`
Unobstructed calculation in MIC	
`cudaComp(DEV_CUDA_srf, DEV_ans,…);`	`_Cilk_spawn _Cilk_offload_to (rank%num_devices) MIC_Comp(…);`
Obstructed calculation in Host at the same time	
`View3D(srf,possibleObstr,…);`	`View3D(srf,possibleObstr,…);`
Synchronize	
`cudaThreadSychronize();` `cudaMemcpy(HOST_ans,DEV_ans,…);`	`_Cilk_sync;`

Performance Comparison

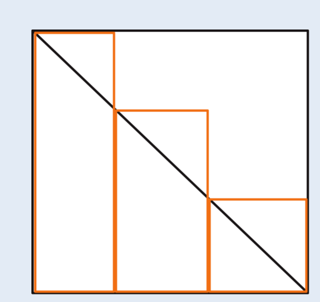
- Case: L-shape with number of surfaces = 20000
- Processor grid: 6x6, NB=64

Determine possible obstruction		Calculation of unobstructed cases	
Keeneland	Beacon	Keeneland	Beacon
1.795 secs	2.149 secs	6.507 secs	111.09 secs

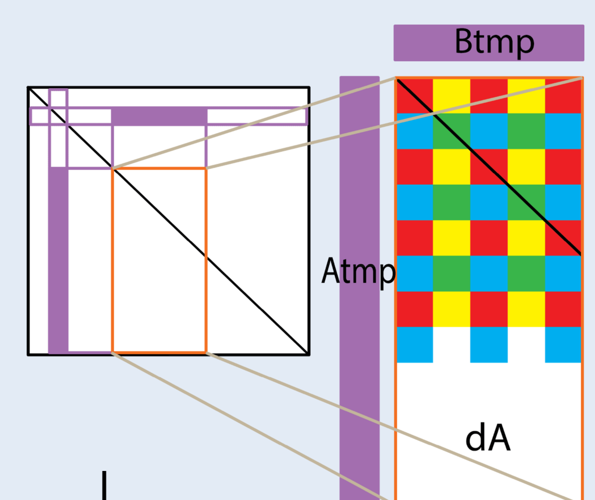
OOC Cholesky Factorization Overview

- To factorize a large dense SPD matrix in the Out-of-Core approach of Chloesky Factorization.
- Combine two standard procedures together: right-looking method and left-looking method.
- Store the problem matrix in host CPU memory (out-of-core)
  - ☐ High amount of memory storage
- Perform heavy matrix calculation on device (in-core)
  - ☐ High computational power
  - ☐ GPU and MIC

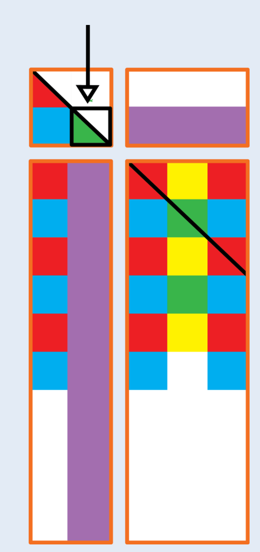
General Procedure



Chop the matrix into panels
- The panel is transferred to core
- Size of panel is limited to core memory
- Different ways of determining each panel size can fine-tune the performance



For each panel, compute **left-looking updates**:
- With every factorized block-columns on the left, update *dA <- dA - Atmp x Btmp* by subroutine `Cpdsyrk_hhd`
- Involve host-host-device data transfer
- Compute by calling `dsyrk` (diagonal part) and `dgemm` (off-diagonal part)

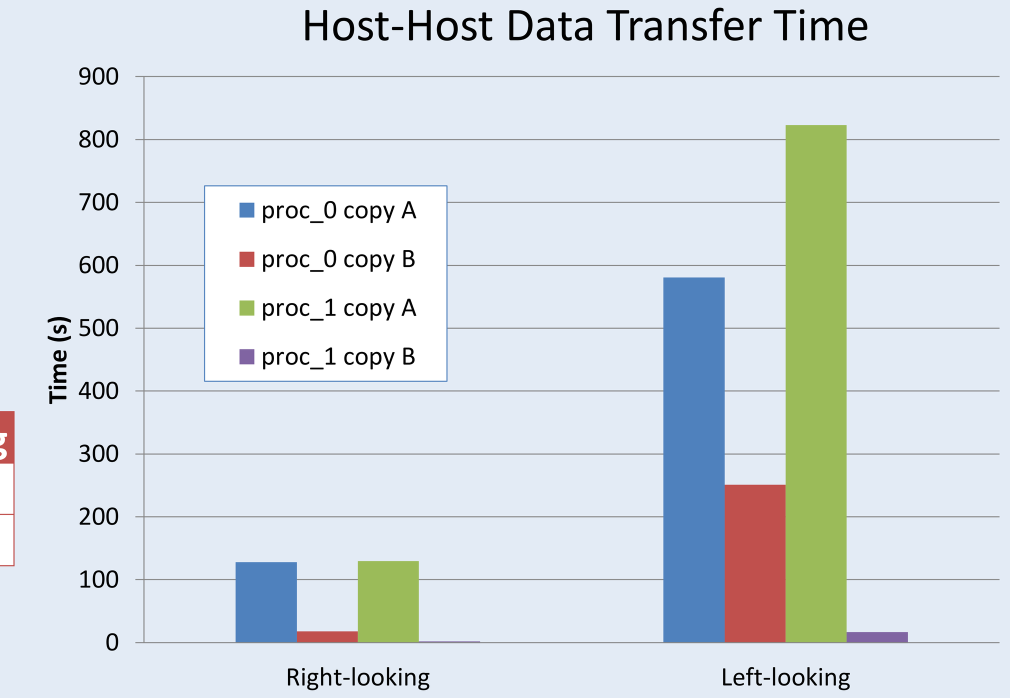


Followed by **right-looking updates**:
For every block along the diagonal,
- call `pdpotrf` to obtain its lower triangular factor;
- call `pdtrsm` to update the submatrix under the block;
- call `Cpdsyrk_hhd` to update the trailing matrix with the updated column

GPU on Keeneland

Data Transfer Case Study
- Matrix Size = 518400
- Total time = 1366 s
- Data transfer time = 1050 s
- Data transfer amount (TB):

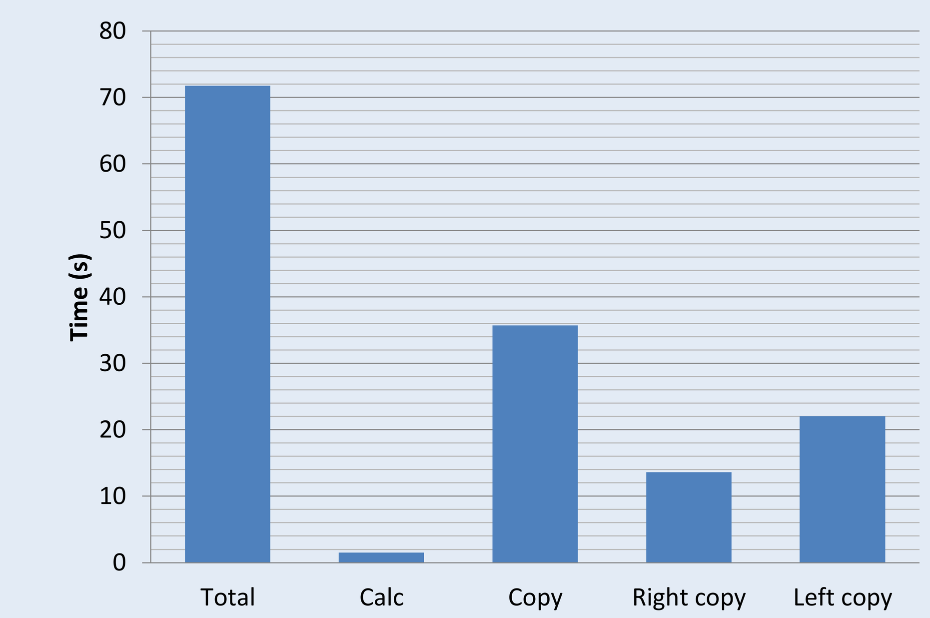
	Right-looking	Left-looking
copy A	25.4	96.5
copy B	2.5	22.9



GPU on Keeneland

Best performance case:
- Matrix size N = 72912
- Processor grid 3x3
- NB=128

Total time: 71.79 secs
Performance: 191 GFLOPS/C



Extend Support to MIC Architecture

Allocate device memory

GPU
- Call **CUBLAS** library

`cublasAlloc(…);`

MIC
- Allocate a memory block in offload region
- The address value is sent back to host

```
#pragma offload target(mic) out(ptr)
    ptr = (intptr_t) memalign(64, size);
```

- intptr_t type is used like a void pointer

Free device memory

GPU
- Call **CUBLAS** library

`cublasFree(…);`

MIC
- Free the pointer in offload region

```
#pragma offload target(mic) in(ptr)
    free((void*)ptr);
```

Data transfer

GPU
- Call **CUBLAS** library

```
cublasSetMatrix(…);
cublasGetMatrix(…);
```

MIC
- Use a buffer to hold the data being copied
- Memory for buffer is allocated on both host and MIC

```
double *buffer=(double*)malloc(n*sizeof(double));
#pragma offload_transfer target(mic) \
  nocopy(buffer:length(n) alloc_if(1) free_if(0));
```

- Then transfer data in buffer to MIC

```
#pragma offload target(mic) in(buffer:…)
```

- Copy data from buffer to destination

Device calculation

GPU
Call **CUBLAS** library

`Cublas_Dgemm(…);`

MIC
Copy argument list to MIC, call **MKL** routine

```
#pragma offload target(mic) in(arg1,arg2,…)
    dgemm(arg1,arg2,…);
```

- Test case of matrix size 368640 on **Beacon**
  Run across 64 compute nodes of Beacon, using 4 MICs per node
  Performance = 47.10 GFLOPS/C
  (peak performance of MIC = 1 TFLOPS)