

# Multi GPU Performance of Conjugate Gradient Algorithm

Lattice 2010, Italy

Hyung-Jin Kim, Weonjong Lee  
Department of physics and Astronomy  
Seoul National University



# Outline

- CG(Conjugate Gradient) Method & Dirac Operator
- CG Implementation on GPU using CUDA & CUBLAS
- Multi GPU implementation using MPI
- Conclusion

# Conjugate Gradient algorithm

- Iterative method for solving linear algebraic equations of the following form

$$Ax = b$$

- $A : n \times n$  positive definite ( $\forall x \neq 0, x^\dagger Ax > 0$ ) Hermitian matrix
- $x, b : n$  dimensional complex vectors
- $A$  and  $b$  are known,  $x$  is unknown.

# CUBLAS & CG

- Conjugate gradient operation

$$r = b - Ax$$

$$d = r$$

$$\delta_{\text{new}} = r^T r$$

$$\delta_0 = \delta_{\text{new}}$$

} Initial Condition

r : residual vector

d : directional vector

$\epsilon$  : tolerance

Ax(or Ad): Dirac operation

for (i = 0; i < N<sub>dim</sub> and  $\delta_{\text{new}} > \epsilon^2 \delta_0$ ; i++) {

$$\alpha = \delta_{\text{new}} / d^T Ad$$

$$x = x + \alpha d$$

$$r = r - \alpha Ad$$

$$\delta_{\text{old}} = \delta_{\text{new}}$$

$$\delta_{\text{new}} = r^T r$$

$$\beta = \delta_{\text{new}} / \delta_{\text{old}}$$

$$d = r + \beta d }$$

} Update process

## Examples

```
•  $\delta_{\text{new}} = r^T r$   
Float DotProduct(...)  
{ return cublasDdot(...); }
```

```
•  $x = x + \alpha d$   
void VectorAddVector(...)  
{ cublasDaxpy(...); }  
...
```

- All of vector operations are processed using **CUBLAS** library except "Dirac operation"

# Dirac operation

- Dirac equation  $h = A\chi$        $A \equiv -D^2 + m^2$

- $D_{x,y} = U_\mu(x)\delta_{y,x+\mu} - U_\mu^\dagger(x-\mu)\delta_{y,x-\mu}$

- Part of Dirac operation (**staggered fermion**)

$$D\chi(x) = \sum_\mu U_\mu(x)\chi(x+\mu) - U_\mu^\dagger(x-\mu)\chi(x-\mu)$$

$$\begin{pmatrix} x''1 \\ x''2 \\ x''3 \end{pmatrix} = \begin{pmatrix} a1 & a2 & a3 \\ b1 & b2 & b3 \\ c1 & c2 & c3 \end{pmatrix} \begin{pmatrix} x1 \\ x2 \\ x3 \end{pmatrix} - \begin{pmatrix} a'1 & a'2 & a'3 \\ b'1 & b'2 & b'3 \\ c'1 & c'2 & c'3 \end{pmatrix} \begin{pmatrix} x'1 \\ x'2 \\ x'3 \end{pmatrix}$$

- Each vector & matrix are complex value  
6 output, 8 x 6 input vector, 18 x 8 matrix data → 1584 bytes
- $U_\mu(x)\chi(x+\mu)$  part has 72 floating point calculations  
→ 8 x 72 = 576 floating point calculations per site
- For  $28^3 \times 96$  lattice, there are  $10^6$  of lattice even(or odd) sites  
→ **0.61 Giga floating point calculations**  
→ **1.55 Giga bytes of data transfer : Memory IO is major bottle neck !**

# Machine Environment

## David Cluster

### · specification

- MPI Lib : MVAPICH v1.1(MPICH for infiniband)
- HW : Qlogic 4x DDR 20Gbps infiniband SW, HCA
- Tested bandwidth

Over packet size 64Kbytes,

it shows 95% of its maximum bandwidth.

Tested maximum Bandwidth is about 1.4 GB/sec.



20Gbps Infiniband SW

# Machine Environment

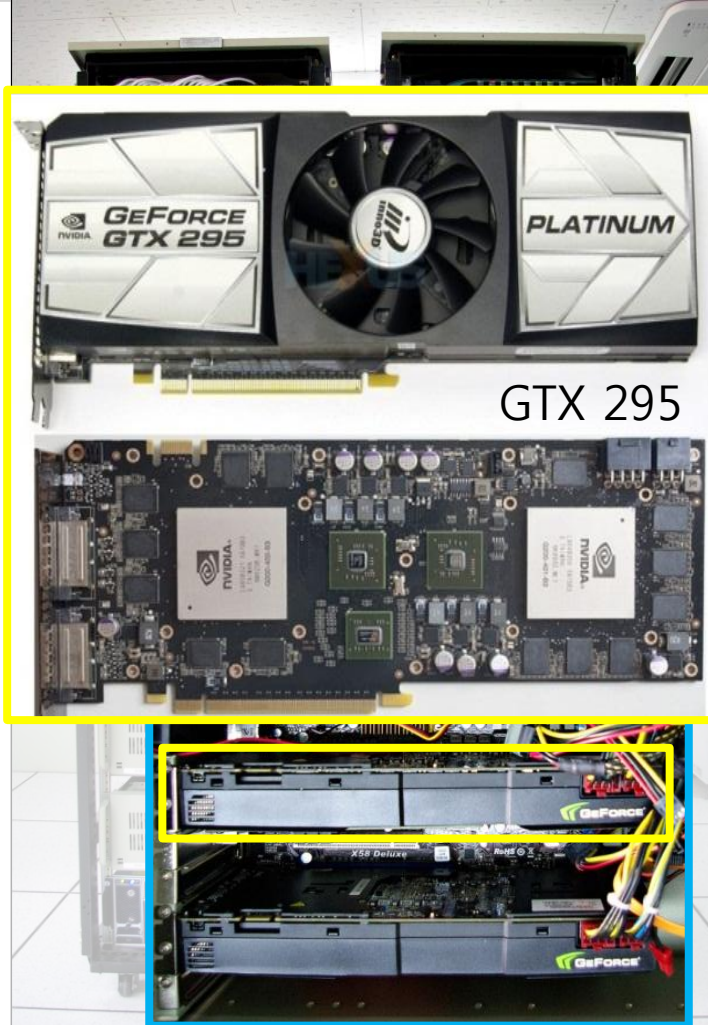
## David Cluster

### • specification

- 4 core x 16 = 64 node GPU cluster
- intel i7 920 2.66Ghz
- 1.3Ghz 12GB triple channel
- 2 x nVIDIA GTX 295 = 4 GPUs per node



# Machine Environment



GTX 295

## David Cluster

### • specification

- 2 x GT200 GPU
- 2 x 895 GFLOPS (Single precision)
- 2 x 74.4 GFLOPS (Double precision)
- 2 x 111.5 GB/sec memory bandwidth
- Sustained bandwidth(Max. value)

Host to Device(paged) : 2850MB/sec

Device to Host(paged) : 2950MB/sec

Device To Device : 94GB/sec

} PCI-E Band.



# 1st CUDA CG code

## CPU code(CPS Lib.)

```
...
for(x = 0;x < Nx ;x++)
for(y = 0;y < Ny ;y++)
for(z = 0;z < Nz ;z++)
for(t = 0;t < Nt ;t++)
{
    for(  $\mu$  = 0;  $\mu$  < 4 ;  $\mu$ ++)
    {
        ...
        if( cur_l[mu] == nx[mu]-1 )
            ...
        else
            ...
        uDotXPlus(sol, U, src);
        ...
        uDagDotXMinus(sol, U, src);
        ...
    }
}
```




## CUDA code

```
...
position = blockIdx.x*blockDim.x+
           threadIdx.x;
Get_location(cur_l, position);
...
for(  $\mu$  = 0;  $\mu$  < 4;  $\mu$ ++)
{
    ...
    if( cur_l[mu] == nx[mu]-1 )
        ...
    else
        ...
    uDotXPlus(sol, U, src);
    ...
    uDagDotXMinus(sol, U, src);
    ...
}
```

- Initial performance is 0.97 GFLOPS per 1 GPU(DP).
- GPU is only twice faster than CPU code(0.46GFLOPS,DP).

# CG optimization 1: coalesced memory access

1 	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16

**Usual program(serial) memory access pattern**



1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16

**parallel program memory access pattern**

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16

**Coalesced memory access pattern**

# CG optimization 1: coalesced memory access

1		2		3	4	5	6	7	8					
9		10		11		12		13		14		15		16

**Usual program(serial) memory access pattern**




1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16

**parallel program memory access pattern**

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16

**Coalesced memory access pattern**

# CG optimization 1: coalesced memory access

1		2		3		4	5	6	7	8
9	10	11	12	13	14	15	16			

**Usual program(serial) memory access pattern**





1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16

**parallel program memory access pattern**

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16

**Coalesced memory access pattern**

# CG optimization 1: coalesced memory access

1		2		3		4		5	6	7	8
9		10		11		12		13	14	15	16

**Usual program(serial) memory access pattern**






1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16

**parallel program memory access pattern**

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16

**Coalesced memory access pattern**

# CG optimization 1: coalesced memory access

1		2		3		4		5		6		7		8
9		10		11		12		13		14		15		16

**Usual program(serial) memory access pattern**







1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16

**parallel program memory access pattern**

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16

**Coalesced memory access pattern**

# CG optimization 1: coalesced memory access

1		2		3		4		5		6		7		8
9		10		11		12		13		14		15		16

**Usual program(serial) memory access pattern**








1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16

**parallel program memory access pattern**

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16

**Coalesced memory access pattern**

# CG optimization 1: coalesced memory access

1		2		3		4		5		6		7		8
9		10		11		12		13		14		15		16

**Usual program(serial) memory access pattern**

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16









**parallel program memory access pattern**

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16

**Coalesced memory access pattern**



# CG optimization 1: coalesced memory access

1		2		3		4		5		6		7		8	
9		10		11		12		13		14		15		16	

**Usual program(serial) memory access pattern**









1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16

**parallel program memory access pattern**

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16

**Coalesced memory access pattern**

# CG optimization 1: coalesced memory access

1		2		3		4		5		6		7		8	
9		10		11		12		13		14		15		16	

**Usual program(serial) memory access pattern**









1		2		3		4		5		6		7		8
9		10		11		12		13		14		15		16

**parallel program memory access pattern**




1		2		3		4		5		6		7		8
9		10		11		12		13		14		15		16

**Coalesced memory access pattern**

# CG optimization 1: coalesced memory access

1		2		3		4		5		6		7		8	
9		10		11		12		13		14		15		16	

**Usual program(serial) memory access pattern**









1		2		3		4		5		6		7		8	
9		10		11		12		13		14		15		16	

**parallel program memory access pattern**





1		2		3		4		5		6		7		8	
9		10		11		12		13		14		15		16	

**Coalesced memory access pattern**

# CG optimization 1: coalesced memory access

1		2		3		4		5		6		7		8	
9		10		11		12		13		14		15		16	

**Usual program(serial) memory access pattern**









1		2		3		4		5		6		7		8	
9		10		11		12		13		14		15		16	

**parallel program memory access pattern**

















1		2		3		4		5		6		7		8	
9		10		11		12		13		14		15		16	

**Coalesced memory access pattern**

# CG optimization 1: coalesced memory access

1		2		3		4		5		6		7		8	
9		10		11		12		13		14		15		16	

**Usual program(serial) memory access pattern**









1		2		3		4		5		6		7		8	
9		10		11		12		13		14		15		16	

**parallel program memory access pattern**

















1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16

**Coalesced memory access pattern**





# CG optimization 1: coalesced memory access

1		2		3		4		5		6		7		8	
9		10		11		12		13		14		15		16	

**Usual program(serial) memory access pattern**









1		2		3		4		5		6		7		8	
9		10		11		12		13		14		15		16	

**parallel program memory access pattern**

















1		2		3		4		5		6		7		8
9		10		11		12		13		14		15		16

**Coalesced memory access pattern**









# CG optimization 1: coalesced memory access

1		2		3		4		5		6		7		8	
9		10		11		12		13		14		15		16	

**Usual program(serial) memory access pattern**









1		2		3		4		5		6		7		8	
9		10		11		12		13		14		15		16	

**parallel program memory access pattern**

















1		2		3		4		5		6		7		8	
9		10		11		12		13		14		15		16	

**Coalesced memory access pattern**













# CG optimization 1: coalesced memory access

1		2		3		4		5		6		7		8	
9		10		11		12		13		14		15		16	

**Usual program(serial) memory access pattern**

1		2		3		4		5		6		7		8	
9		10		11		12		13		14		15		16	









**parallel program memory access pattern**

1		2		3		4		5		6		7		8	
9		10		11		12		13		14		15		16	

















**Coalesced memory access pattern**



















# CG optimization 1: coalesced memory access

1		2		3		4		5		6		7		8	
9		10		11		12		13		14		15		16	

**Usual program(serial) memory access pattern**









1		2		3		4		5		6		7		8	
9		10		11		12		13		14		15		16	

**parallel program memory access pattern**

















1		2		3		4		5		6		7		8	
9		10		11		12		13		14		15		16	

**Coalesced memory access pattern**

















# CG optimization 1: coalesced memory access

1		2		3		4		5		6		7		8	
9		10		11		12		13		14		15		16	

**Usual program(serial) memory access pattern**

1		2		3		4		5		6		7		8	
9		10		11		12		13		14		15		16	

**parallel program memory access pattern**

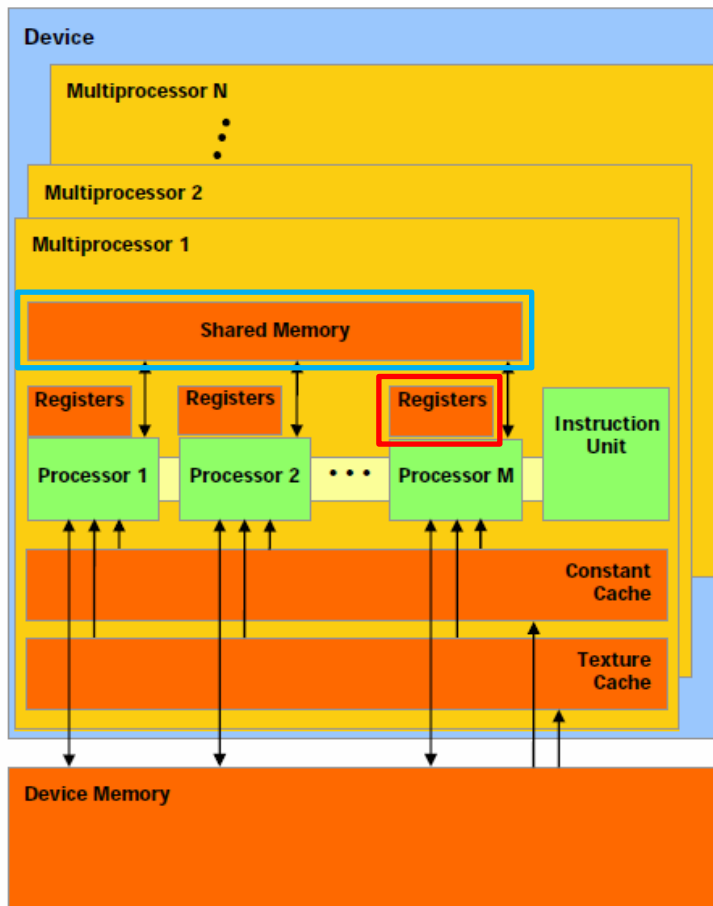
1		2		3		4		5		6		7		8	
9		10		11		12		13		14		15		16	

**Coalesced memory access pattern**

Coalesced memory access → **maximum bandwidth** is possible between GPU and device memory

Performance is **enhanced by 8.5 times**, it shows **8.2 GFLOPS**

# CG optimization 2: Register & Shared Memory



- Register & shared Mem. are very fast on-chip memory .
- compute capability 1.3(CUDA compute mode), 16K of shared Mem. & 64K of registers are usable. But **in double precision, shared memory has intrinsic bank-conflict problem.**
- Register also has memory bank-conflict, but if the number of thread block is multiples of 64, bank-conflict could be avoidable
- By using these fast buffer, we can **accelerate about 3 times more.** GPU FLOPS is **25 GFLOPS**
- Register used code is **~15% faster** than shared memory used program.

# CG optimization 3: Occupancy

$$\begin{pmatrix} x'1 \\ x'2 \\ x'3 \end{pmatrix} = \begin{pmatrix} a1 & a2 & a3 \\ b1 & b2 & b3 \\ c1 & c2 & c3 \end{pmatrix} \begin{pmatrix} x1 \\ x2 \\ x3 \end{pmatrix}$$

## Two ways of Matrix-vector Multiplication

### - More Register -

$x'1 += a1*x1 + a2*x2 + a3*x3$   
Total 77 registers are needed

### - Less Register -

$x'1 += a1*x1$   
 $x'1 += a2*x2$   
 $x'1 += a3*x3$   
Total 61 registers are needed

$$\text{Occupancy} = \frac{\text{active threads number}}{\text{Max number of threads}} * 100$$

- Occupancy is very complicated issue, there is no "always correct answer".
- In our case, we reduced the number of used registers under 64, so 2 thread blocks can be launched per multiprocessors(Occupancy = 25%). This enables us 7% of performance benefit.  
※ total 16348 registers → there are 128 threads per block, so 128 registers(=16384/128) are assigned for each multiprocessors.

# CG optimization 4 : Etc.

- **Reduce the "branch code"(if, case, switch ...)**
  - GPU is not good in "branch prediction", so unnecessary branch code should be removed.
  - By this work, we can get additional 30% of performance enhancement
- **SU(3) matrix reconstruction(12 parameter)**
  - Matrix 3<sup>rd</sup> row :  $c = (a \times b)^*$   
this reduces data transfer by 1/3.
  - The amount of calculation is increased by 67%
  - More registers are used( occupancy ↓ )
  - In actual calculation,  
**overall performance is decreased about 7%**
- **bit operator, loop unrolling, etc ...**
  - Performance is improved by 5~8 %

$$\text{SU3 reconstruct} \begin{pmatrix} a1 & a2 & a3 \\ b1 & b2 & b3 \\ (a \times b)^* \end{pmatrix}$$

**SU3 reconstruction result**  
Total 5.1ms (measured)

Data loading(2.75ms,ideal)

GPU calculation(3.35ms, ideal)

**Most Optimized result**  
Total 4.7ms (measured)

Data loading(4.1ms, ideal)

GPU calculation (2ms, ideal)

# CG performance

Initial CG performance : 0.97 GFLOPS



optimized performance : 35 GFLOPS

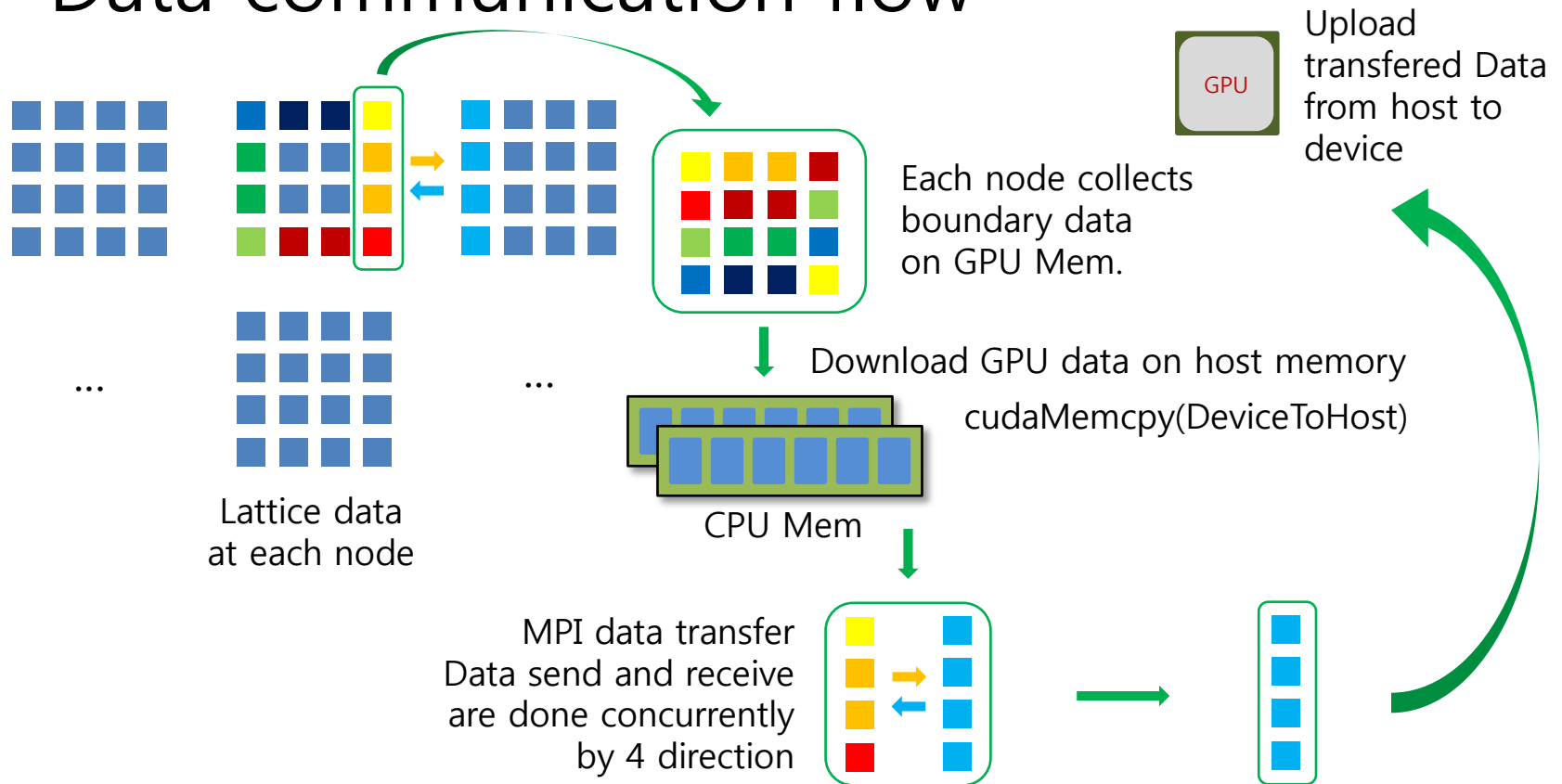
**This is 47% of peak performance**

in GTX 295 GPU (double precision)

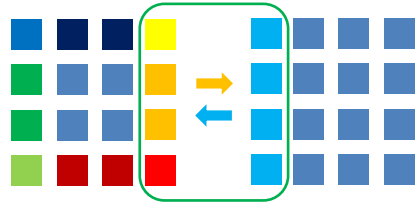
Performance is enhanced by 76 times  
than CPU code!

# MPI for Multi GPU

- Data communication flow



# MPI Communication



	4 node time (ms)	8 node time (ms)
GPU calculation time	4.7	2.45
boundary data collect	0.9	0.5
cudaMemcpy DtoH	2.9	2.1
MPI communication	2.3	1.8
cudaMemcpy HtoD	3.4	1.7
Total Comm. time	~9.5	~6.1
Total time	~14.5	~8.6

- **MPI Communication function**(used CPS library)

GetPlusData(double \*send, double \*rcv,...)

GetMinusData(double \*send, double \*rcv,...)

They communicate in "Asynchronous way"

→ Do MPI "Send" & "Receive" simultaneously

- **Network optimization idea**

If we can overlap "cudaMemcpy time" with

"MPI communication time",

then total communication time could be reduced !

Memcpy DtoH

MPI Comm.

Memcpy HtoD

Synchronous  
Comm.



Asynchronous  
Comm.

Memcpy DtoH

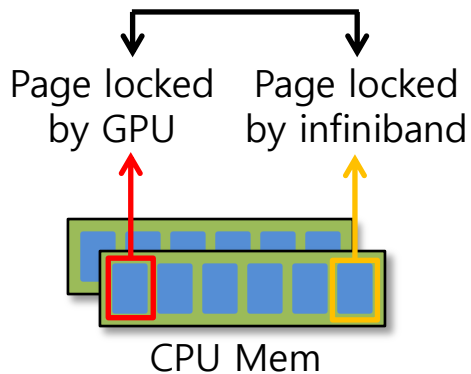
MPI Comm.

Memcpy HtoD



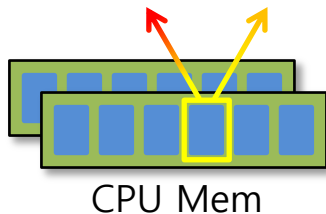
# Problems & Future work

**Not compatible!**  
"memcpy" is needed



**GPU-Direct**(Mellanox-nVIDIA)

Page locked Mem. sharing  
on GPU & infiniband

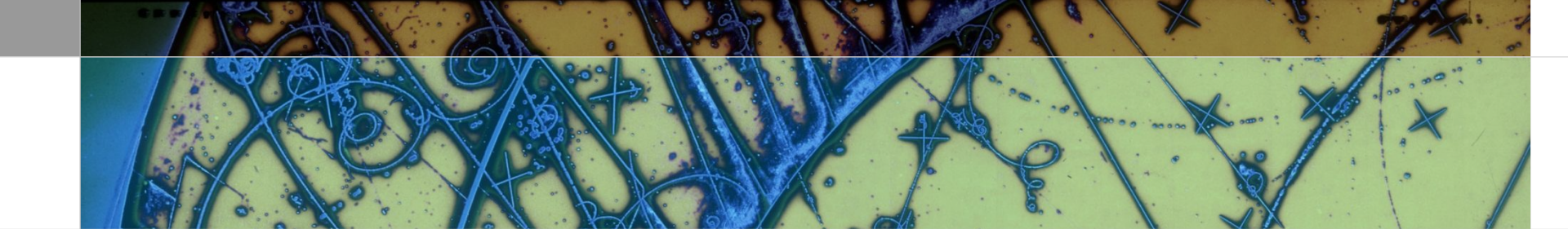


- **Bi-directional cudamemcpy is not supported in our GPU.**
  - GPU download & upload sequence could not be overlapped
- **Non compatible page locked memory**
  - Asynchronous MPI communication
  - Asynchronous cudamemcpy

In both case, they need **their own "page-locked memory"**
- **These Problems should be resolved to improve the network Comm. on the cluster system !**
  - From Fermi version of Tesla, it supports bi-directional memcpy between CPU and GPU
  - By using "Mellanox-nVIDIA GPU-Direct Technology", Infiniband and GPU can share the page locked memory
- **Implement mixed precision**

# Summary

- We can get a good result from optimized CUDA CG program in staggered fermion.
  - It is **35 GFLOPS (47% of peak)**, and this is 75.6 times faster result than CPU code
  - Including network communication time, FLOPS is reduced to **12.3 GFLOPS**
- Current bottle-neck in GPU programming is in the network communication & GPU memory bandwidth



Thank you  
Any Question?