

# Fock matrix preparation with GPGPU for fragment molecular orbital (FMO) calculation

Hiroaki UMEDA

Center for Computational Sciences,
University of Tsukuba

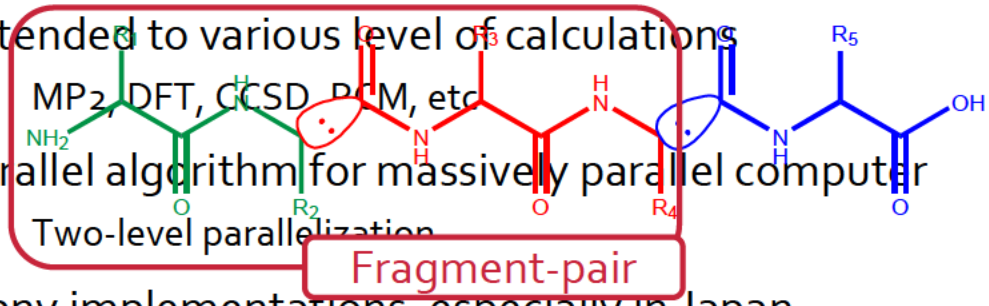
筑波大学
*University of Tsukuba*

# Quantum Chemistry Applications

- Famous Gaussian, GAMESS, … program has a long history, then
  - Massive codes with fortran
  - Parallelization is not efficient
    - long code, many functionalities, and many hotspot
    - difficult to catch up with new architecture
- Need more simple and compact program
  - Enable massively parallel execution in algorithm
  - Fragment Molecular Orbital (FMO) method
  - OpenFMO program
- GPU acceleration of Fock matrix construction, which is one of the hotspot of FMO calculation

# Fragment Molecular Orbital (FMO) Method

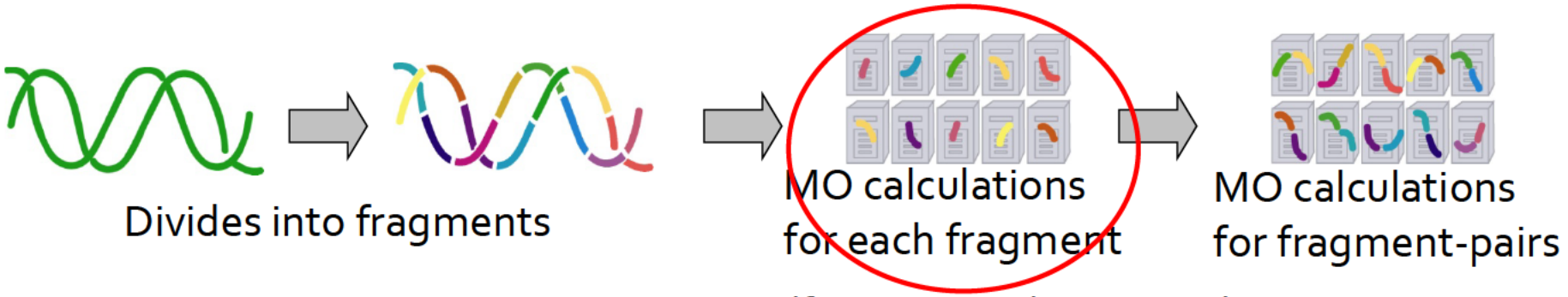
- Computational technique to calculate molecular properties of large molecular system, such as protein
  - K. Kitaura et al., *Chem. Phys. Lett.* **312** (1999) 319
  - Avoid costly Fock matrix calculation O(N⁴) for a entire molecule
    - Divides molecule into many fragments
    - Reconstructs entire properties from fragments and fragment-pairs calculations with environmental electrostatic potential (ESP)
  - Interaction energy analysis between fragments
    - IFIE, PIEDA
  - Extended to various level of calculations
    - MP2, DFT, CCSD, PCM, etc.
  - Parallel algorithm for massively parallel computer
    - Two-level parallelization
  - Many implementations, especially in Japan
    - GAMESS, ABINIT-MP(X), PAICS, OpenFMO, NWChem
    - PC Cluster, BlueGene/P, Earth Simulator, etc.




$$E^{\mathrm{FMO}} = \sum_{I}^{N_f} E_I + \sum_{I>J}^{N_f} \left(E_{IJ} - E_I - E_J\right) = \sum_{I>J}^{N_f} E_{IJ} - \left(N_f - 2\right)\sum_{I}^{N_f} E_I$$

$$\mathbf{D}^{\mathrm{FMO}} = \sum_{I>J}^{N_f} \mathbf{D}_{IJ} - \left(N_f - 2\right)\sum_{I}^{N_f} \mathbf{D}_I$$

# Parallelization of FMO calculation




SCC (self-consistent charge) condition

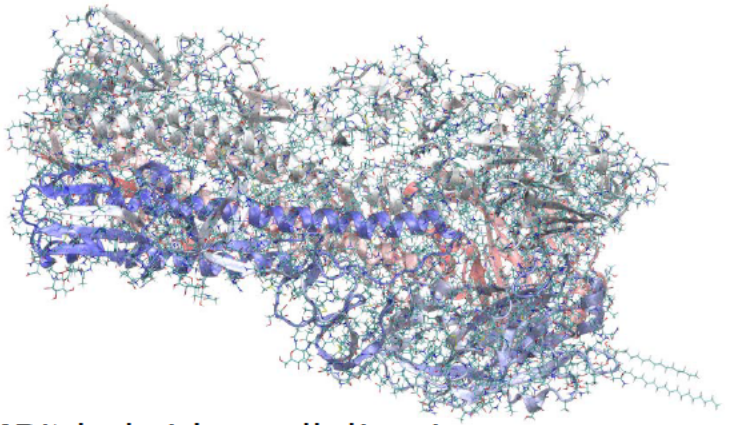
- Initialize fragments
- Until $\{D_i\}$ converged (SCC)
  - Calculate $\{E_i\}$ and $\{D_i\}$ for each fragment
    - Construct ESP
    - SCF Calc. with ESP
- Calc. $\{E_{ij}\}$ and $\{D_{ij}\}$ for each fragment-pair
  - Construct ESP
  - SCF Calc. with ESP
- Calc. $E^{FMO}$ and $D^{FMO}$

**Two-level Parallelization**

Parallel calculation with Inter- & Intra-fragment(-pair)s

# FMO Calculation on K-computer

- Influenza HA protein (23,460 atoms)
- FMO-RI-MP2/6-31G(d)+cc-pVDZ
  - 721 fragments
- 24,576 nodes (196,608 cores) of K-computer
  - GAMESS-FMO
  - Large Fortran77 program with OpenMP/DDI(MPI) hybrid parallelization



| # node | Etime /s | Efficiency | DDI group for SCC |
|---|---|---|---|
| 12,288 | 999 | | 64nodes*192 |
| 24,576 | 641 | 0.78 | 64nodes*332 + 128nodes*26 |

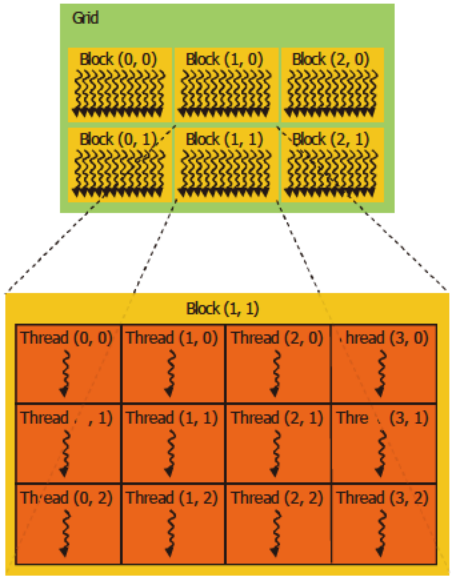
- Parallel efficiency is good, but wanted to be improved more!
  - Load-imbalance among DDI groups
  - Limited parallel efficiency for a fragment calculation

# HA-PACS GPU cluster

- HA-PACS GPU-cluster (Univ. Tsukuba)
  - 268 nodes, 2 CPU (Intel SandyBridge EP, 8 cores) & 4 GPU(NVIDIA M2090)
  - Fat-Tree network with Infiniband QDR x 2 rails
  - CUDA, OpenACC, XcalableMP, etc.

# NVIDIA GPU

- **NVIDIA M2090 GPU (Fermi)**
  - Specification
    - 16 SM/GPU, 32 core/SM (512core/GPU)
    - Global Memory: 6GB/GPU,
    - L2$: 768KB/GPU, L1$/shared memory: 64KB/SM
  - Execution
    - Offload model
      - GpuKernel <<< # thread block, # thread >>> (args)
    - SIMT execution within warp (32 threads)
    - Register is fast, but size is limited.
  - No hardware support to DP atomic operation
- **NVIDIA K20 (Kepler)**
  - Many changes from Fermi
    - SM→SMX (192 core/SMX)
    - Hyper-Q, Warp Shuffle, Read-only cache
    - Increase maximum register size for a thread
    - Improve atomic ops. including DP



# HA-PACS GPU cluster

- HA-PACS GPU-cluster (Univ. Tsukuba)
  - 268 nodes, 2 CPU (Intel SandyBridge EP, 8 cores) & 4 GPU(NVIDIA M2090)
  - Fat-Tree network with Infiniband QDR x 2 rails
  - CUDA, OpenACC, XcalableMP, etc.
- Development of GPU accelerated molecular orbital (MO) program
  - Fock matrix construction with GPGPU (CUDA)
    - Most time-consuming process in Hartree-Fock calculation,
    - Basic and common process in other MO calculations
  - OpenFMO program

# OpenFMO

- OpenFMO is compact (simple) FMO program, targeting for massively parallel computer
  - Y. Inadomi (Kyushu Univ.)
  - Codes
    - C program (~50,000 lines), OpenMP/MPI hybrid parallelization
      - cf.) GAMESS: Fortran77 (~1,300,000 lines), DDI(MPI) parallelization
    - PC cluster, K-computer
  - Limited functionalities
    - Only HF-level FMO energy calculation
      - Should have MP2, DFT, analytical gradient, and faster ESP algorithm
      - Open to anyone for developing other functionalities
    - Providing HF-SCF skeleton-code for a fragment calculation
      - Obara-Saika method for 2e-integral calculation
      - Easily backported to OpenFMO itself
  - Collaborations with computer scientists
    - Toward next-generation supercomputer
    - GPU, MIC, special accelerator

# Hartree-Fock Calculation

$$\mathbf{FC} = \mathbf{SC}\boldsymbol{\varepsilon}$$

Hartree-Fock-Roothaan eq.

$$F_{ij} = H_{ij}^{\text{core}} + \sum_{k,l}^{N} D_{kl}\left\{2(ij\,|\,kl) - (il\,|\,kj)\right\}$$

Fock matrix (G matrix)

$$H_{ij}^{\text{core}} = \int d\mathbf{r}\varphi_i(\mathbf{r})\hat{h}\varphi_j(\mathbf{r})$$

1e-Hamiltonian matrix

$$D_{ij} = 2\sum_{a}^{N_{elec}/2} C_{ia}C_{ja}$$

Density matrix

$$(ij\,|\,kl) = \int d\mathbf{r}_1 \int d\mathbf{r}_2 \varphi_i(\mathbf{r}_1)\varphi_j(\mathbf{r}_1)\frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|}\varphi_k(\mathbf{r}_2)\varphi_l(\mathbf{r}_2)$$

2-e integral

$$S_{ij} = \int d\mathbf{r}\varphi_i(\mathbf{r})\varphi_j(\mathbf{r})$$

Overlap matrix

$$\psi_a(\mathbf{r}) = \sum_{i=1}^{N} C_{ia}\varphi_i(\mathbf{r};\mathbf{n}_i,\mathbf{R}_i)$$

$\psi_a(\mathbf{r})$ Molecular orbital,
$\varphi_i(\mathbf{r})$ Basis functions

- Iterative calculation (Self-consisted Field, SCF)
- Fock(G) matrix calculation is time-consuming, formally O($N^4$)

# G-matrix Construction

$$G_{ij} = \sum_{k,l}^{N} D_{kl} \left\{ 2(ij \mid kl) - (il \mid kj) \right\}$$

$$(ij \mid kl) = \int d\mathbf{r}_1 \int d\mathbf{r}_2 \varphi_i(\mathbf{r}_1) \varphi_j(\mathbf{r}_1) \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|} \varphi_k(\mathbf{r}_2) \varphi_l(\mathbf{r}_2)$$

- Four-fold loop for $i,j,k,l$: $O(N^4)$
  - Have to reduce massive 2-e integrals
- Symmetric property
  - $(ij|kl)=(ij|lk)=(ji|kl)=\ldots$ → integral-driven algorithm
- Overlap screening: evaluate overlap between basis functions

  → Two-fold loop for basis function pairs $ij$, $kl$ with non-zero overlap
- Schwarz screening: estimate maximum integral value with Cauthy-Schwarz inequality

$$(ij \mid kl) \le \sqrt{(ij \mid ij)} \sqrt{(kl \mid kl)}$$

  → Skip integral calculation if estimated integral value is smaller than threshold

# G-mat constr. pseudo-code: (ps,ss)

```
for (ijcs=0; ijcs<N_ijcs; ijcs++)
   for (klcs=0; klcs<N_klcs; klcs++) {
      if (check_schwarz(ijcs, klcs)) {
         x[] = calc_2e_psss(ijcs, klcs);
         for (i=0,iao=iao0; i<3; i++,iao++) {
            G[iao][jao] += 4*x[i]*D[kao][lao];
            G[kao][lao] += 4*x[i]*D[iao][jao];
            G[iao][kao] -=   x[i]*D[jao][lao];
            G[iao][lao] -=   x[i]*D[jao][kao];
            G[jao][kao] -=   x[i]*D[iao][lao];
            G[jao][lao] -=   x[i]*D[iao][kao];
         }
      }
   }
}
```

ijcs, klcs: index of contracted shell (CS) pairs with non-zero overlap between i CS and j CS

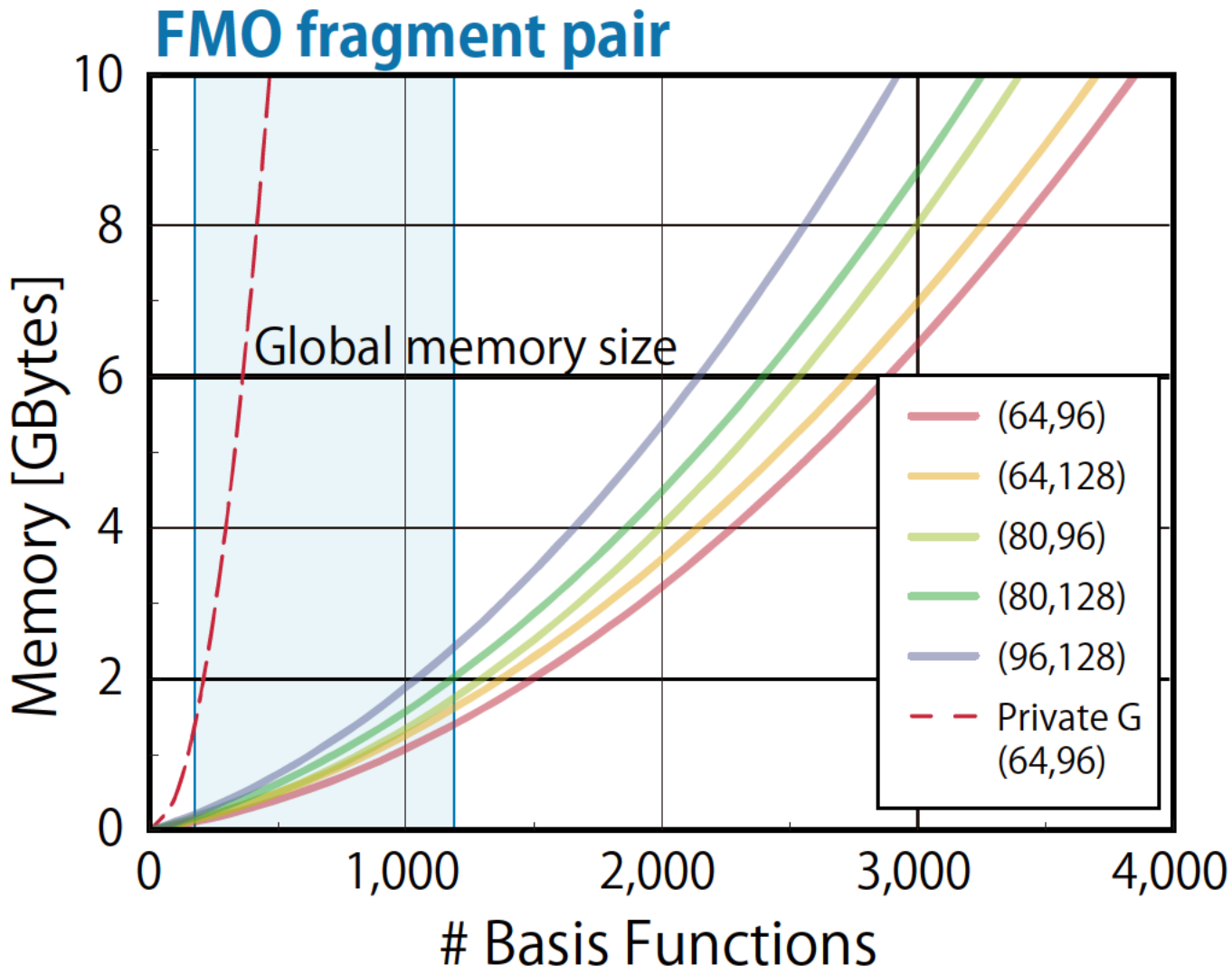
Schwarz inequality screening

2-e Intgral calculation

Accumulattion to G matrix
6 matrix elements

Too many threads to hold thread private G-matrix
→ **DP atomic add op. in loops**

# Memory size for G-mat constr. in FMO



# G-mat constr. pseudo-code: (ps,ss)

```
for (ijcs=0; ijcs<N_ijcs; ijcs++)
   for (klcs=0; klcs<N_klcs; klcs++) {
      if (check_schwarz(ijcs, klcs)) {
         x[] = calc_2e_psss(ijcs, klcs);
         for (i=0,iao=iao0; i<3; i++,iao++) {
                                              ao];
                                              ao];
            G[iao][kao] -=   x[i]*D[jao][lao];
            G[iao][lao] -=   x[i]*D[jao][kao];
            G[jao][kao] -=   x[i]*D[iao][lao];
            G[jao][lao] -=   x[i]*D[iao][kao];
         }
      }
   }
}
```

ijcs, klcs: index of contracted shell (CS) pairs with non-zero overlap between i CS and j CS

Schwarz inequality screening

2-e Intgral calculation

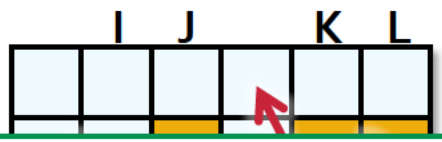
**How to avoid DP atomic op.?**

Accumulattion to G matrix
6 matrix elements

Too many threads to hold thread private G-matrix
→ **DP atomic add op. in loops**

# Proposed Algorithm

```
G[i][j] += 4*x*D[k][l];
G[k][l] += 4*x*D[i][j];
G[i][k] -=   x*D[j][l];
G[i][l] -=   x*D[j][k];
G[j][k] -=   x*D[i][l];
G[j][l] -=   x*D[i][k];
```




Matrix elements updated within inner kl loop are categorize to three-types, as

- G[i][*] → Gi[*]
- G[j][*] → Gj[*]
- G[k][l]

kl loop runs only surviving k,l-pairs from overlap screening, then

- G[k][l] → Gkl[kl]

- Allocate full G-matrix **G**[][], and its kl-part **Gkl**[] for each thread block
- ij-loop is distributed to thread blocks
- kl-loop is distributed to threads of the ij-thread block
- Allocate **Gi**[],**Gj**[] vectors on global memory for each thread
- Accumulate **Gi**[], **Gj**[] into **G**[][] with coalesced fashion, after sync_threads() for ij-thread block at the end of the ij-task
- Accumulate **Gkl**[] into **G**[][] at the last part of the kernel

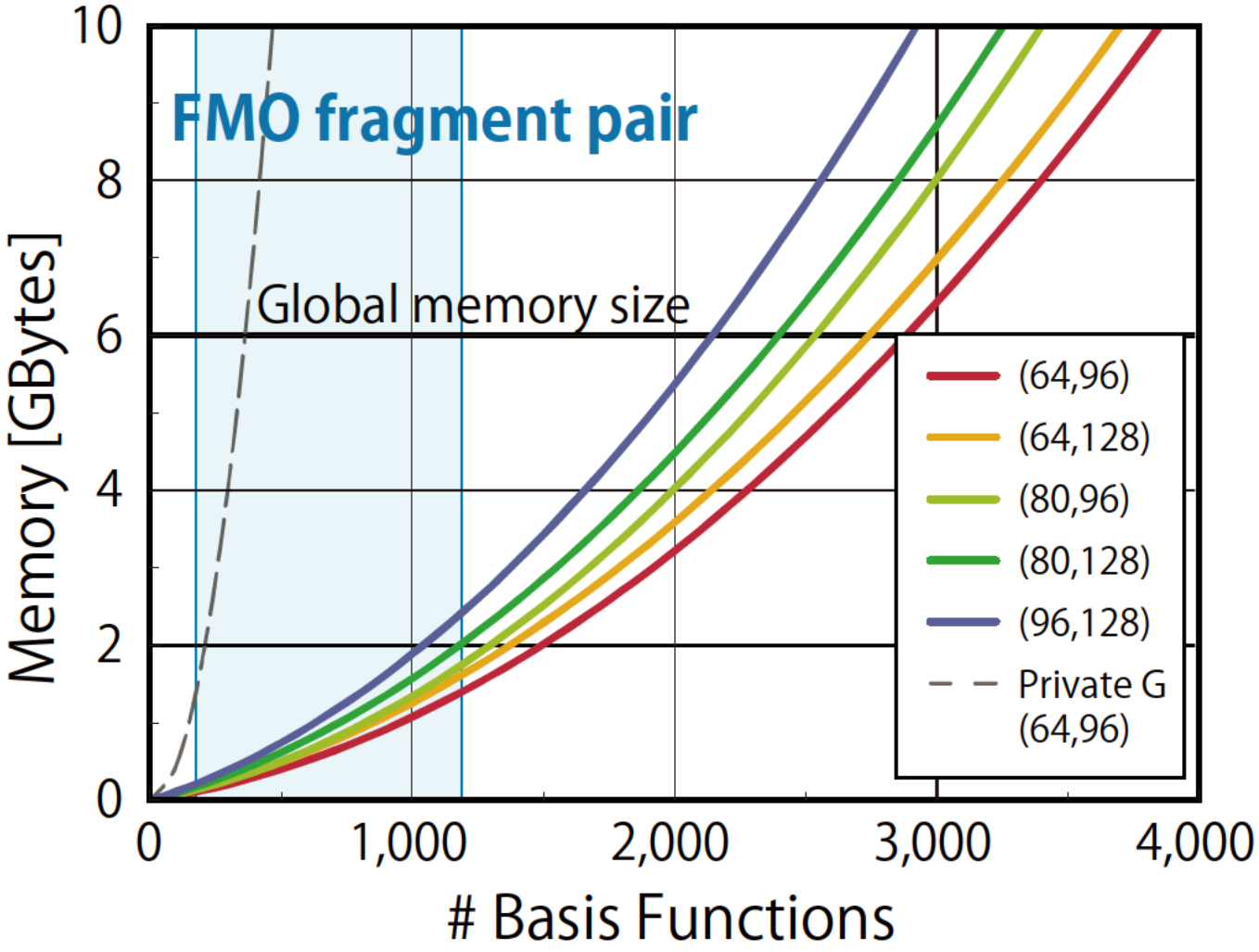
# Pseudo-code of proposed algorithm

```
for (ijcs=bIdx; ijcs<N_ijcs; ijcs+=Nblk) {
  double Gij[3]=ZERO;
  Gi[Nao*3]=ZERO; Gj[Nao]=ZERO;
  for (klcs=tIdx; klcs<N_klcs; klcs+=Nth) {
    if (check_schwarz(ijcs, klcs)) {
      x = calc_2e_psss(ijcs, klcs);
      for (i=0,iao=iao0; i<3; i++,iao++) {
        Gij[i]      += 4*x[i]*D[kao][lao];
        Gkl[klcs]   += 4*x[i]*D[iao][jao];
        Gi[Nao*i+kao] -= x[i]*D[jao][lao];
        Gi[Nao*i+lao] -= x[i]*D[jao][kao];
        Gj[kao] -=   x[i]*D[iao][lao];
        Gj[lao] -=   x[i]*D[iao][kao];
      }
    }
  } // end klcs loop
  __sync_thread();
  Gi[jao+i] += Gij[i];
  for (i=tIdx; i<Nao*3; i+=Nth) {
    double Gtmp=ZERO;
    for (ith=0; ith<Nth; ith++)
        Gtmp += Gi[i](ith);
    G[iao0+i] += Gtmp;
  }
  for (j=tIdx; j<Nao; j+=Nth) {
    double Gtmp=ZERO;
    for (ith=0; ith<Nth; ith++)
        Gtmp += Gj[i](jth);
    G[jao0+j] += Gtmp;
  }
} // end ijcs loop
__sync_thread();

for (klcs=tIdx; klcs<N_klcs; klcs+=Nth) {
  G[kao][lao] += Gkl[klcs];
}
```

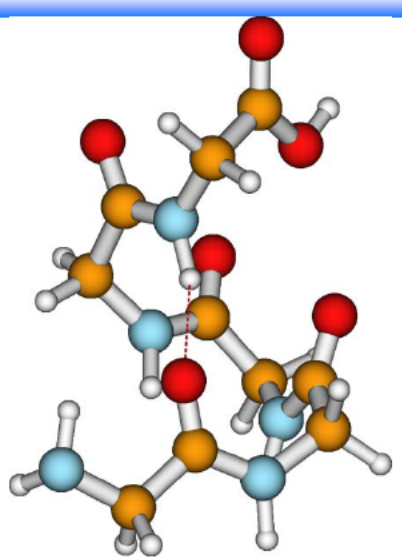
- Distribute ij to thread blocks
- Distribute kl to threads of ij-thread block
- Allocate **Gi**[], **Gj**[] at global memory for each thread
- Sync_thread for ij-thread block
  - Coalesced accumulation of **Gi**[], **Gj**[] to **G**[][]
- No atomic op. for **Gkl**[]

# Memory size for G-mat constr. in FMO

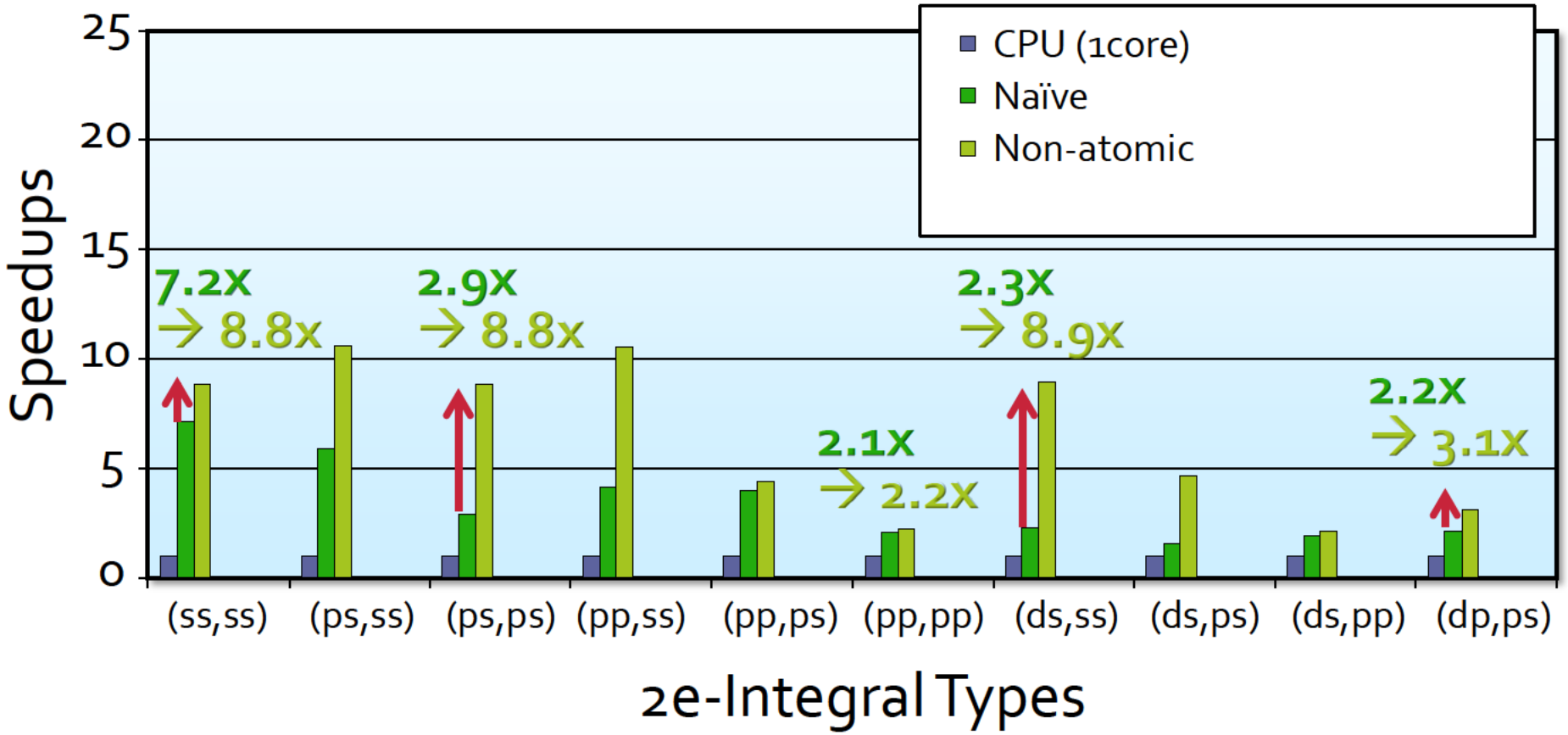


# Performance Evaluation

- $(Gly)_5$ , HF/6-31G(d)
  - 38 atom, 349 atomic orbitals
- HA-PACS
  - 1GPU (NVIDIA M2090; 665GFLOPS)
  - 1CPU core (Intel E5 2.6GHz; 20.8GFLOPS)
  - Software
    - Intel icc 13.0; Nvidia cuda 5.0; Intel MKL 4.0; mvapich2 1.8.1



# Speedups from 1 CPU core




Speedups depend on integral types

- Cost to calculate integrals
  - Obara-Saika integral: large working arrays for higher integral types
- Number of atomic additions

# Perfo

No atomic op.

Pre-screening

Dynamic LB

Sorting

## SLB (Static Load-Balance)

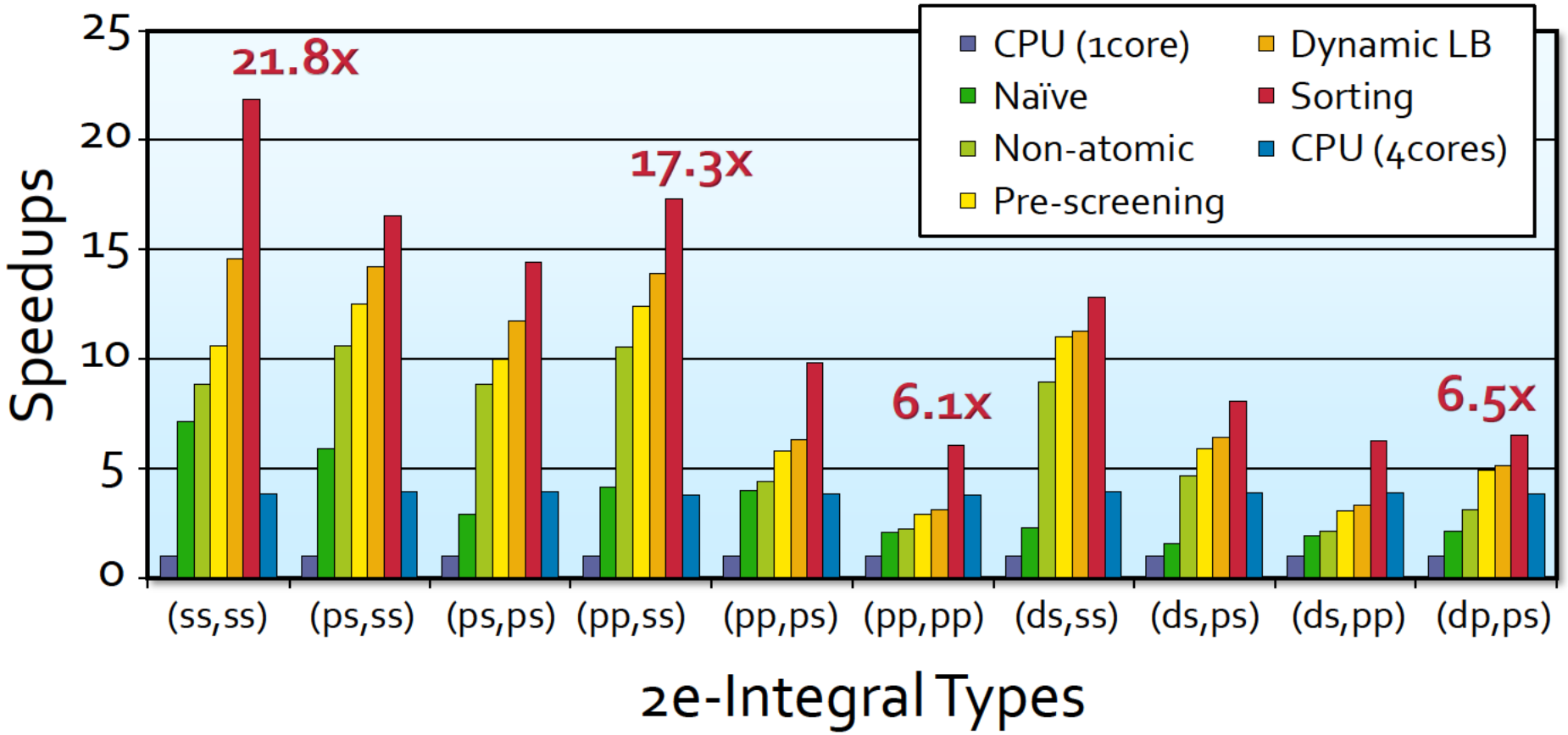
```
for (ijcs=bIdx; ijcs<N_ijcs; ijcs+=Nblk) {
   Process for ijcs
} // end ijcs loop
```

## DLB (Dynamic Load-Balance)

```
int *counter; // Counter on global memory,
              // which initialized by host.

ijcs = bIdx;
while (ijcs < N_ijcs) {
  Process for ijcs
  ijcs = atomicAdd(counter, 1);
} // end ijcs loop
```

# Speedups from 1 CPU core



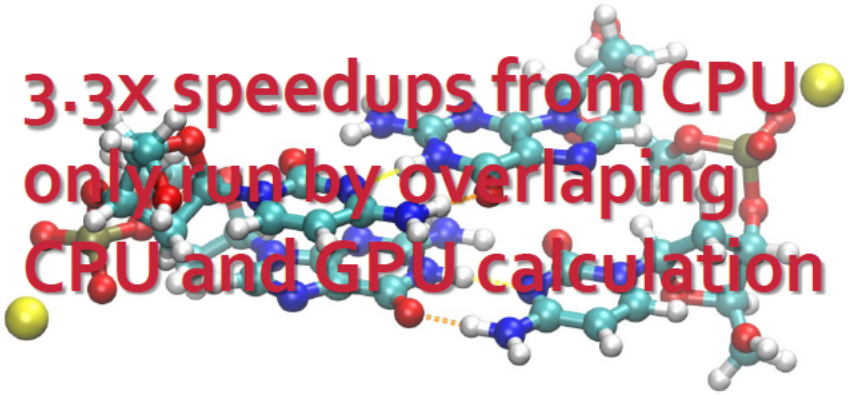

**6x~22x Speedups by GPGPU (M2090)**

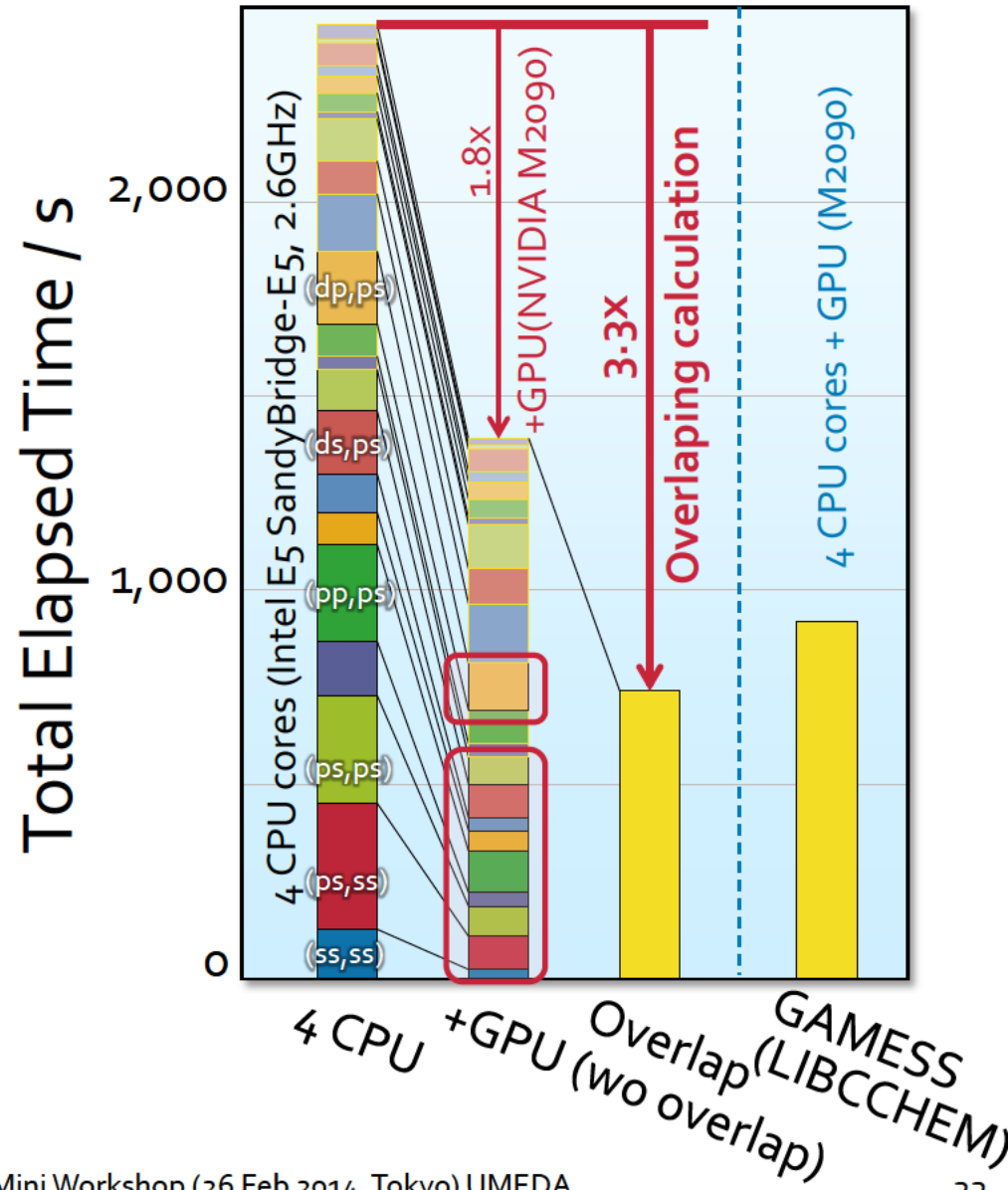
Speedups depend on integral types

- Cost to calculate integrals
  - Obara-Saika integral: large working arrays for higher integral types

# CPU and GPU Overlap Calculation

- Model DNA $(CG)_2$ , HF/6-31G(d)
  - 126 atom, 1,208 AO
- HA-PACS (16CPUcores+4GPUs/nodes)
  - 4CPU cores (OpenMP) + 1GPU
- Software
  - OpenFMO
  - GAMESS (LIBCCHEM)
    - Version: 1 MAY 2013 (R1)



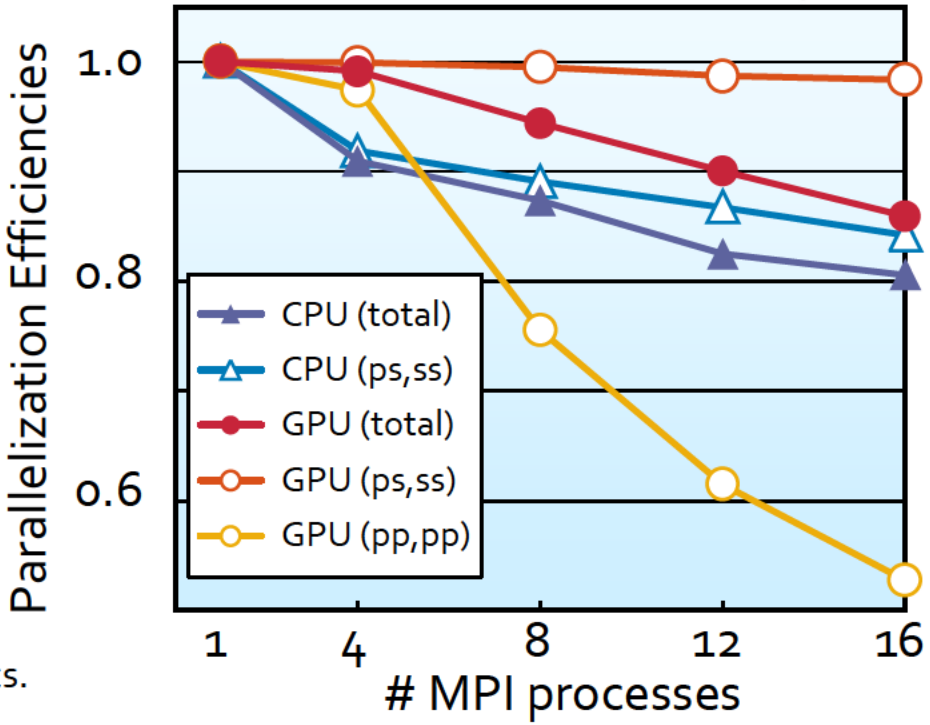



# Parallelization with Multi-GPUs

- Policy
  - Use original parallelization algorithm in OpenFMO
    - Assign ij cs-pair to MPI processes as round robin fashion
  - 1 MPI process handles 1 GPU

- HA-PACS
  - 16 CPU cores + 4 GPUs / node
  - 4 MPI processes / node
  - 4 OMP threads / MPI process

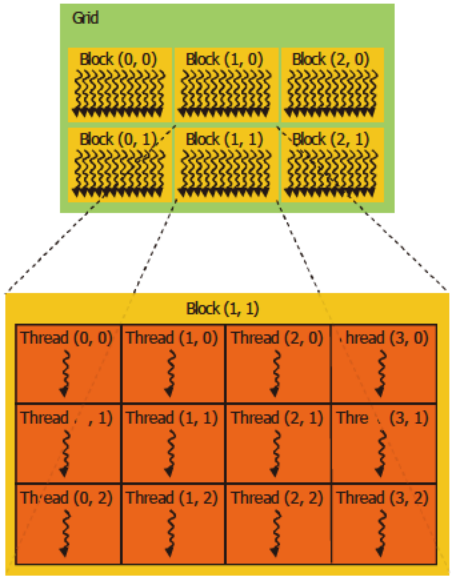
**Basically, better parallel efficiency than CPU**



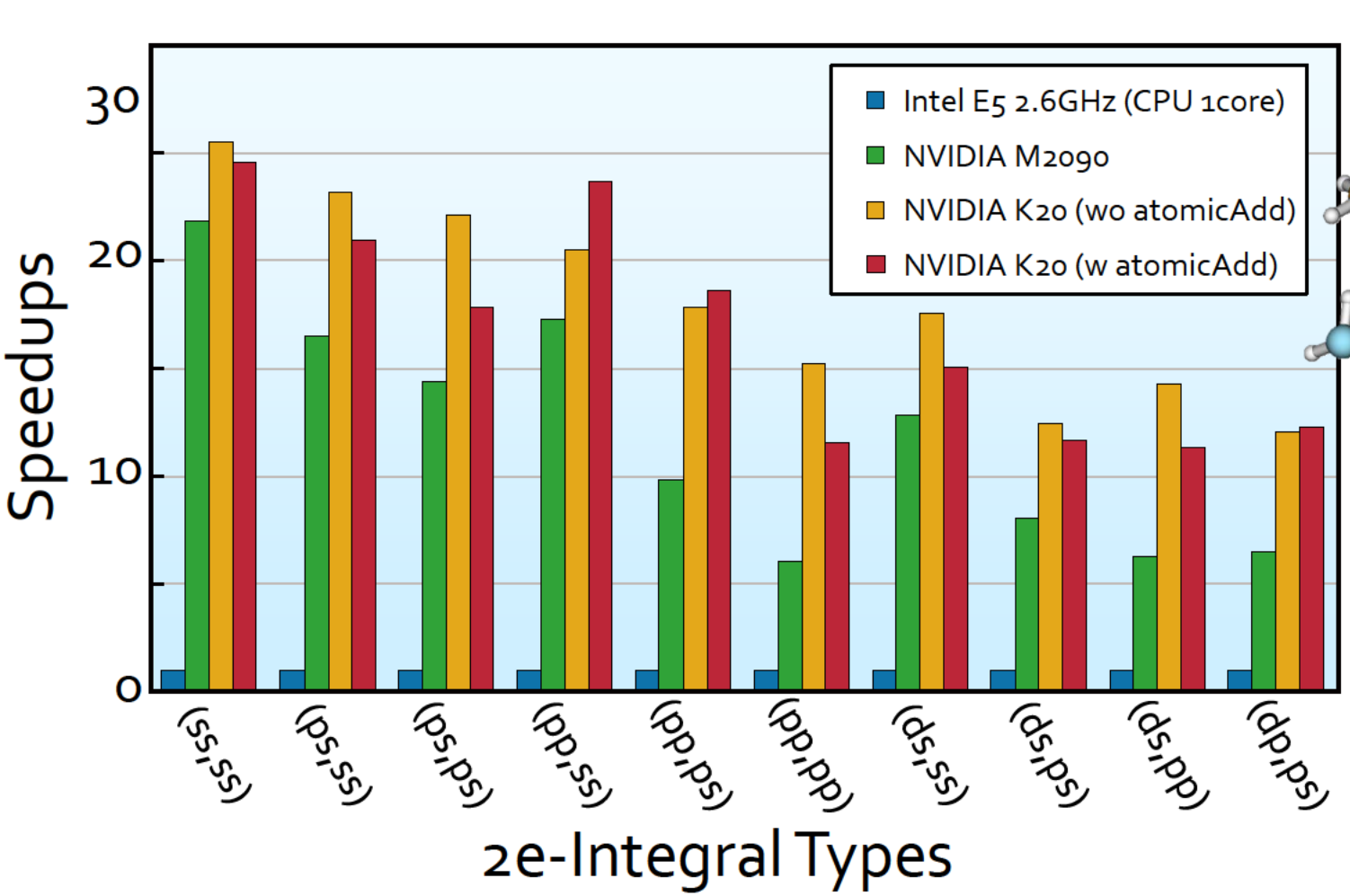

Bad efficiency for (pp,pp)-type integrals
- causes from load-imbalance among MPI procs.

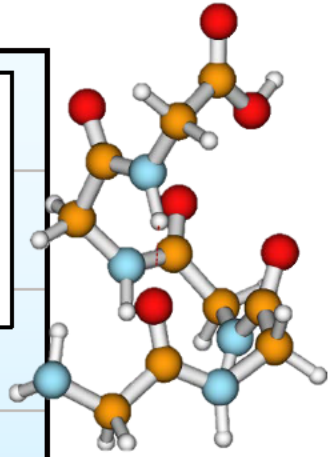
# NVIDIA GPU

- **NVIDIA M2090 GPU (Fermi)**
  - Specification
    - 16 SM/GPU, 32 core/SM (512core/GPU)
    - Global Memory: 6GB/GPU,
    - L2$: 768KB/GPU, L1$/shared memory: 64KB/SM
  - Execution
    - Offload model
      - GpuKernel <<< # thread block, # thread >>> (args)
    - SIMT execution within warp (32 threads)
    - Register is fast, but size is limited.
  - No hardware support to DP atomic operation
- **NVIDIA K20 (Kepler)**
  - Many changes from Fermi
    - SM→SMX (192 core/SMX)
    - Hyper-Q, Warp Shuffle, Read-only cache
    - Increase maximum register size for a thread
    - Improve atomic ops. including DP



# K20 Performance Evaluation



**AtomicAdd-less algorithm is still efficient**

Depend on balance of overheads: DP atomic operation and **Gi**[], **Gj**[] accumulation

# Summary

- GPU accelerated Fock matrix preparation routine with our proposed algorithm runs 3.3 times faster than CPU only execution
  - No DP atomic operation
  - Optimizations
    - Pre-screening
    - Dynamic load-balancing
    - Sorting
    - CPU-GPU Overlap calculation
  - Parallelization
  - K20 architecture
- Will be backported to OpenFMO program