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

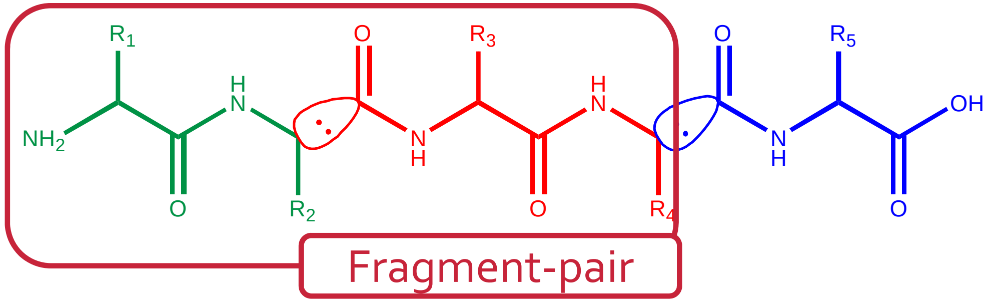
Hiroaki UMEDA

Center for Computational Sciences,
University of Tsukuba

筑波大学
University of Tsukuba

Fragment Molecular Orbital (FMO) Method

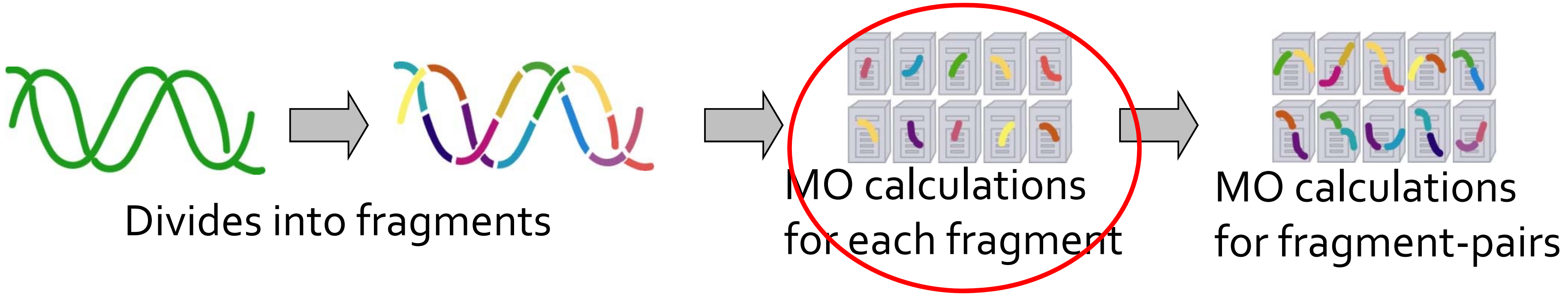
- Computational technique to calculate molecular properties of large molecular system, such as protein, with ab initio level of theory
  - K. Kitaura et al., *Chem. Phys. Lett.* **312** (1999) 319
  - Avoid costly Fock matrix calculation O(N^4) for a entire molecule
    - Divides molecule into many fragments
    - Reconstructs entire properties from fragments and fragment-pairs calculations with environmental electrostatic potential (ESP)
  - Interaction energy analysis between fragments
    - IFIE, PIEDA



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

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

Parallelization of FMO calculation

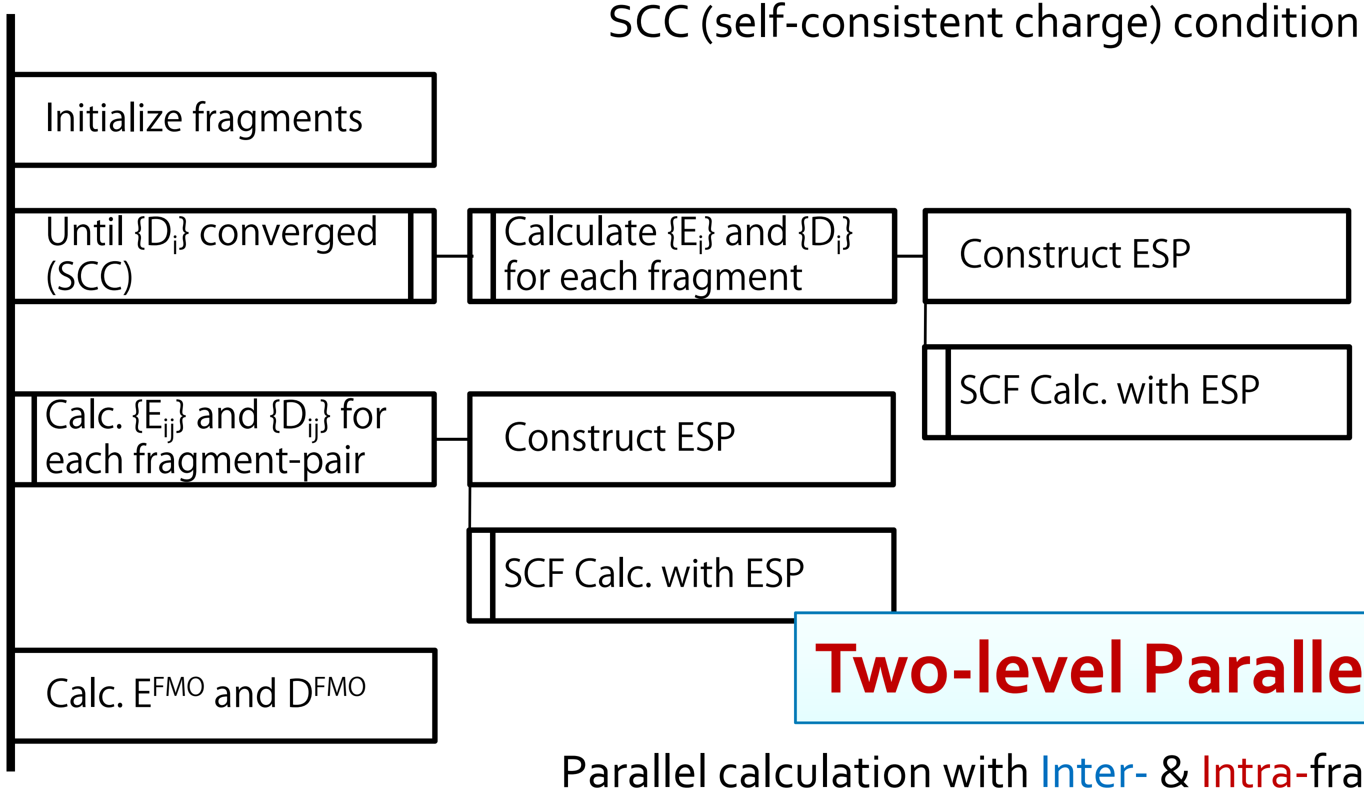


Divides into fragments

MO calculations for each fragment

MO calculations for fragment-pairs

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

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
    - 3 process types: master, worker, memory server
  - 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
      - Easily backport to OpenFMO itself
  - Collaborations with computer scientists
    - Toward next-generation supercomputer
    - GPU, MIC, special accelerator
    - Reconstruction with RPC model for fault tolerance

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];
                                          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][la
         }
      }
   }
}
```

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

Schwarz inequality screening

2-e Intgral calculation

Accumulattion to G matrix
6 matrix elements

How to avoid DP atomic op.?

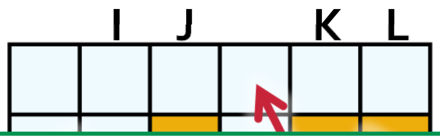
G-matrix $G_{ij} = \sum_{k,l}^{N} D_{kl} \left\{ 2(ij|kl) - (il|kj) \right\}$

Two-electron repulsion integral

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

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

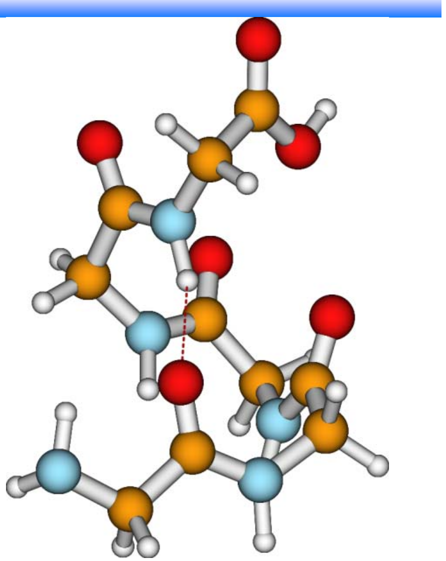
- **G**[i][*] → **Gi**[*]
- **G**[j][*] → **Gj**[*]
- **G**[k][l]

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

- **G**[k][l] → **Gkl**[kl]

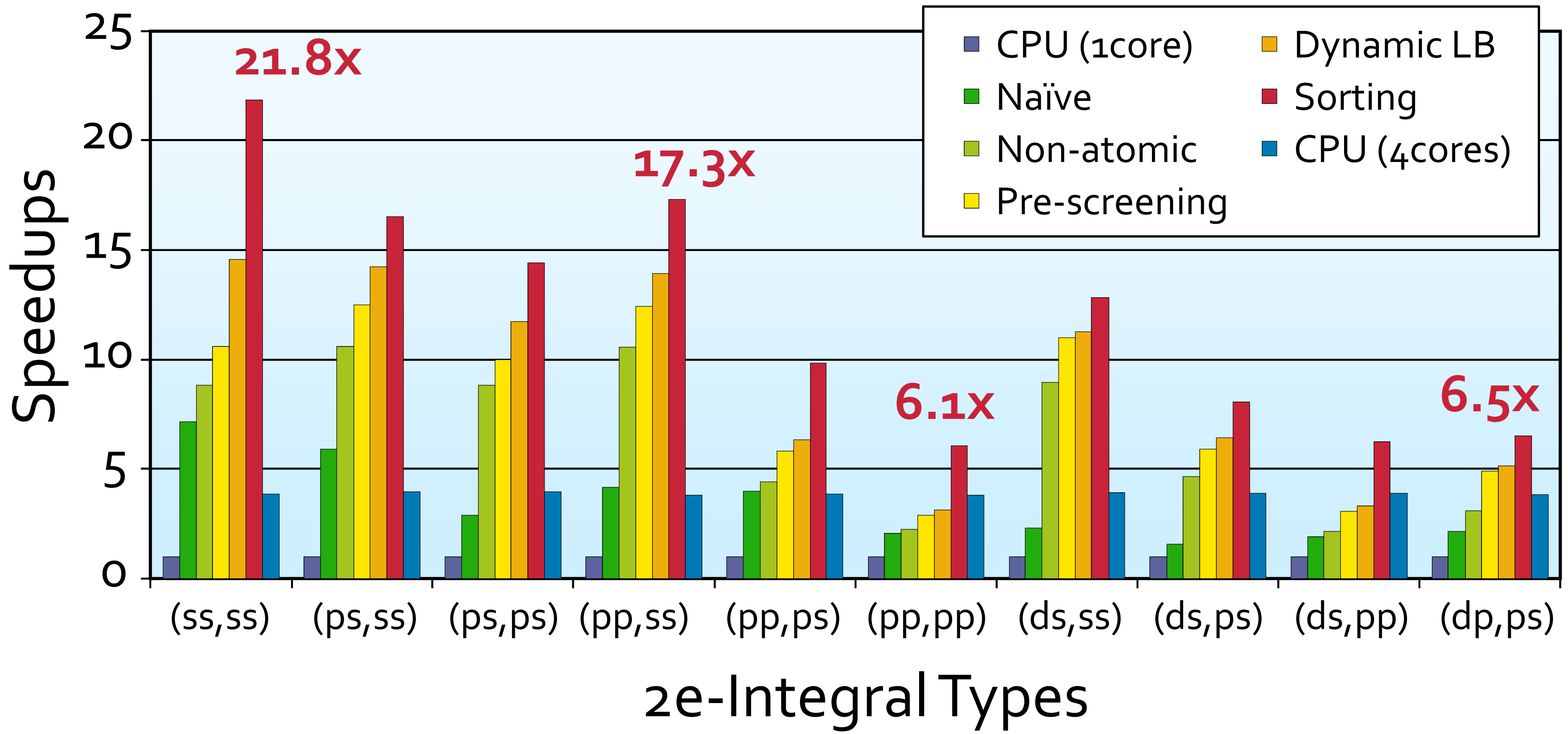
- Allocate full G-matrix **G**[][], and its kl-part **Gkl**[] for each thread block
- ij-loop is distributed to thread blocks
- kl-loop is distributed to threads of the ij-thread block
- Allocate **Gi**[],**Gj**[] vectors on global memory for each thread
- Accumulate **Gi**[], **Gj**[] into **G**[][] 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

Performance Evaluation



- $(\mathrm{Gly})_5$, HF/6-31G(d)
  - 38 atom, 349 atomic orbitals
- HA-PACS base cluster
  - 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

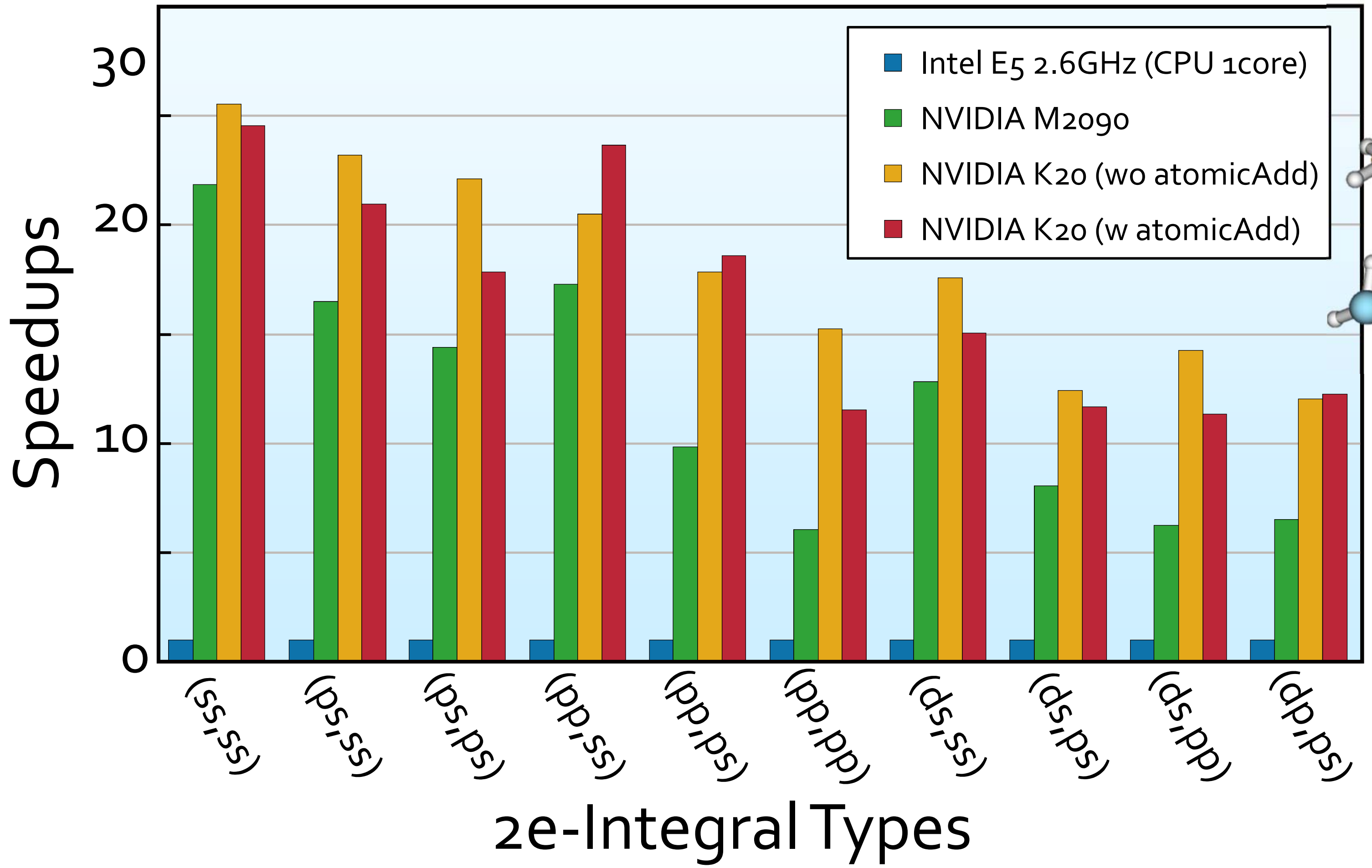


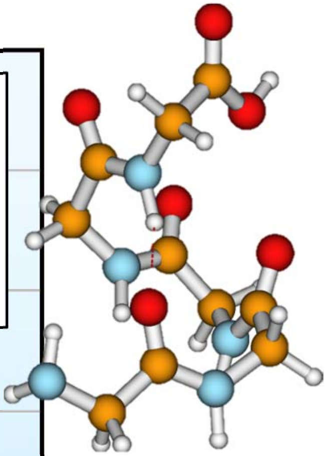
6x~22x Speedups by GPGPU (M2090)

Speedups depend on integral types

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

K20 Performance Evaluation



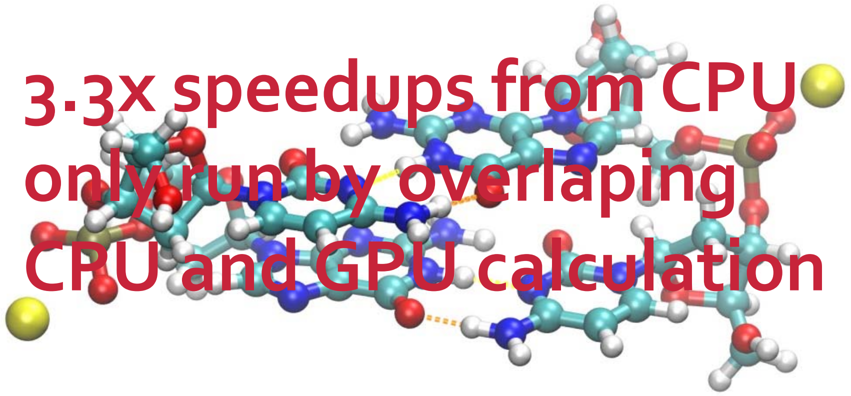


AtomicAdd-less algorithm is still efficient

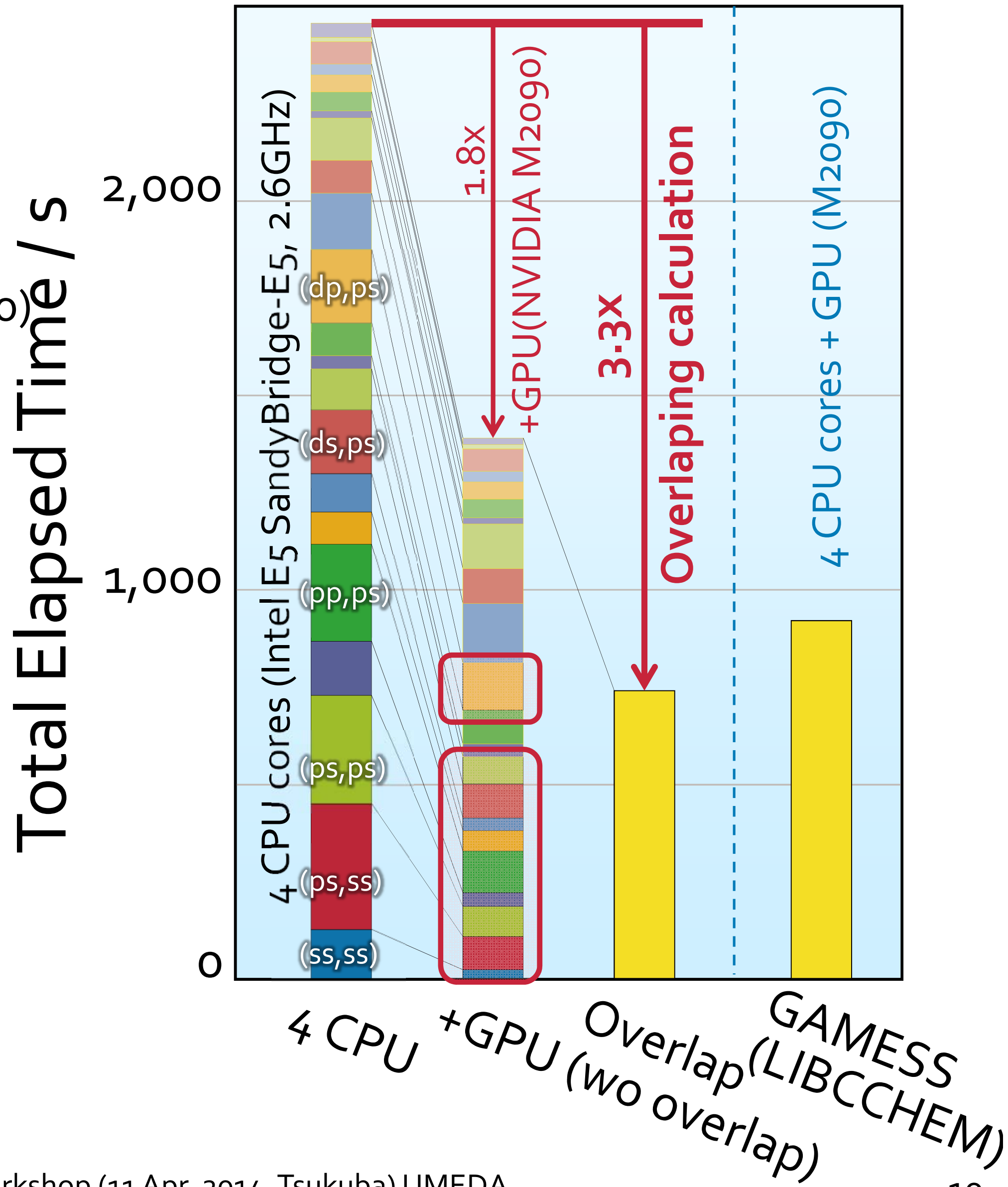
Depend on balance of overheads: DP atomic operation and **Gi**[], **Gj**[] accumulation

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(M2090)
- Software
  - OpenFMO
  - GAMESS (LIBCCHEM)
    - Version: 1 MAY 2013 (R1)



3.3x speedups from CPU only run by overlaping CPU and GPU calculation



Benchmark for FMO calculation

- Backport our code for the HF skeleton program into original OpenFMO program
- icc 14.0 / CUDA 5.0.35 / mvapich2 1.8.1 / 1MPI rank = 4OMP threads + 1GPU
- $(Gly)_{10}$, FMO-HF/6-31G(d)
  - 112 atoms, 10 residues, 5 fragments
  - HA-PACS 2nodes: 3 MPI ranks / fragment(-pair) SCF
- Crambin, FMO-HF/6-31G(d)
  - 642 atom, 46 residues, 20 fragments
  - HA-PACS 8nodes: 5 MPI ranks / fragment(-pair) SCF

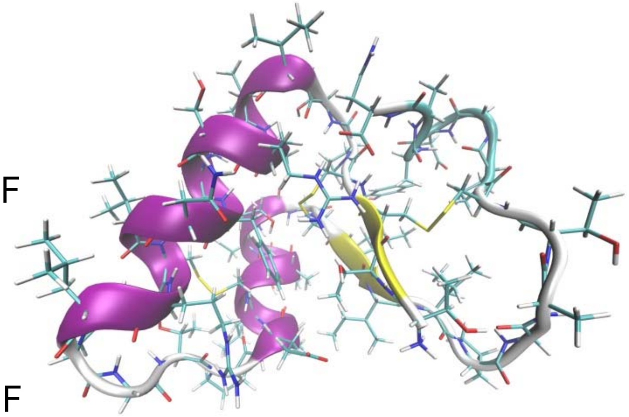


Table. Elapsed time [sec.] for benchmark FMO calculations

	$(Gly)_{10}$			Crambin		
Algorithm	SCC	DimerSCF	Total	SCC	DimerSCF	Total
Direct	240	166	414			
In-core	214	113	335	1,513	1,382	3,007
Direct(GPU+CPU)	196	88	302	1,355	1,185	2,661

GPU accelerated direct method is faster than in-core method, where all 2-e integrals are stored in memory

Summary

- GPU accelerated Fock matrix preparation routine with our proposed algorithm runs 3.3 times faster than CPU only execution
  - No DP atomic operation
  - CPU and GPU overlap calculation
  - Parallelization
  - Further optimization
    - SP calculation when possible
- GPU accelerated direct FMO calculation is faster than in-core FMO calculation
  - Further optimization
    - GPU direct & in-core CPU overlap calculation
    - GPU acceleration of preparing 2e-integrals for in-core method (?)