High Performance Simulations of Quantum Transport using Manycore Computing

Yosang Jeong and Hoon Ryu*

(E: elec1020@kisti.re.kr*)

Division of National Supercomputing
Korea Institute of Science and Technology Information (KISTI)

KiSTi 한국과학기술정보연구원 Korea Institute of Science and Technology Information www.kisti.re.kr

국가슈퍼컴퓨팅연구소 National Institute of Supercomputing and Networking

IPCC KISTI

The Non-Equilibrium Green's Function approach and The Recursive Green's Function algorithm

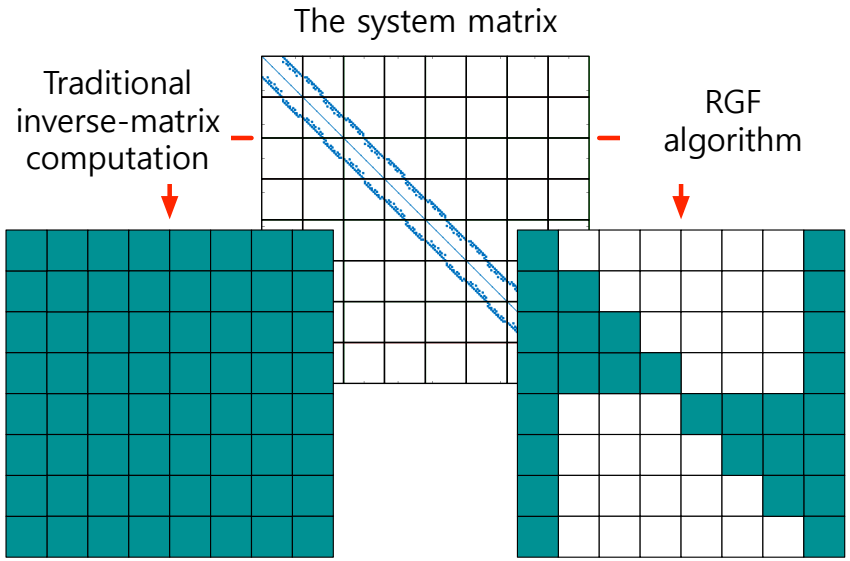
The Non-Equilibrium Green's Function (NEGF) approach

- Essential to predict quantum transport
  - Transmission, Local Density of State, Charge
- Involves the evaluation of an inverse of the large-scale complex number system matrix
  - The most time-consuming part of the NEGF

$$G^R(E) = (E - (H + V) - \sum(E - (H + V)))^{-1}$$

The Recursive Green's Function (RGF) algorithm

- Performs the multiplication of sub-matrices in a recursive manner
- Evaluates parts of the inverse matrix
  - Saves huge computing cost compared to the traditional way



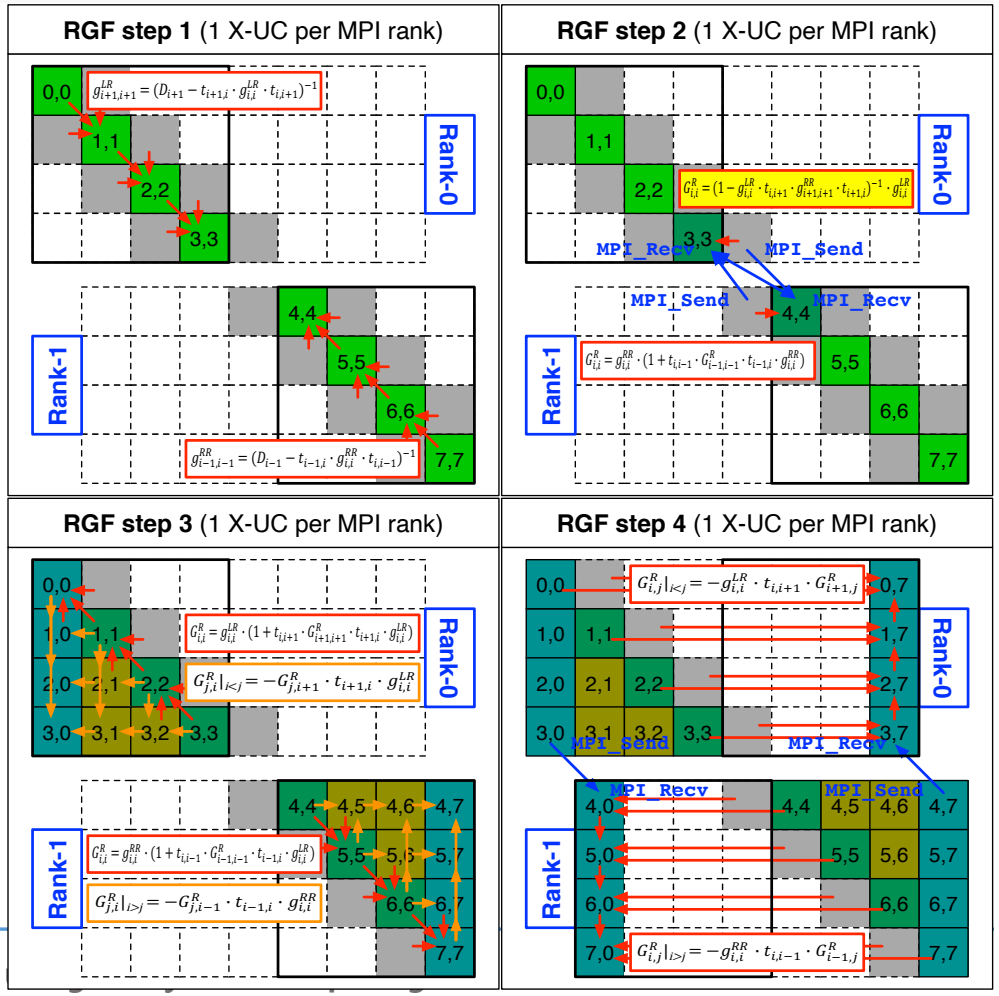
Our Goal: Accelerating our in-house code **QAND** (Quantum simulation tool for Advanced Nanoscale Device designs) **using KNL and GPU**

Processes of RGF computation
The 4-computational steps

- The RGF consists of the 4-computational steps
  - The whole computing process of RGF can be divided into two regions
    (the top and bottom half of the system matrix)
  - Computation in each region is allocated to a single **MPI** process
  - The computing load of a single MPI rank is processed in parallel with **OpenMP** threads
  - Only steps 2 and 4 perform MPI communication

- All steps perform **the sub-matrix multiplication**
  - The number of matrix multiplications in step 3 is much bigger than in other steps
    (Especially off-diagonal computation part)

→ **Need to focus on step 3**



Processes of RGF computation

The cost of the step 3 & the size of the nanowire structure

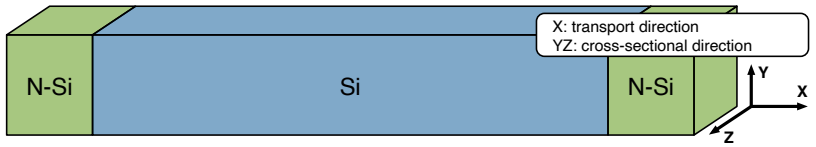
- The nanowire structure consists of multiple atomic unitcells
  - The X-UCs (the number of unitcells along the X-direction) is related to the number of diagonal sub-matrices
  - The YZ-UCs (the number of unitcells on the YZ-plane) is related to the size of each sub-matrix

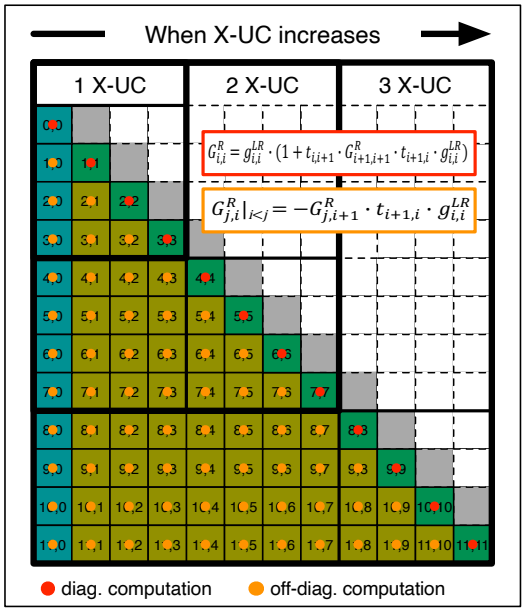
- As the size of nanowires structure grows, the number of the sub-matrices to be computed increases.

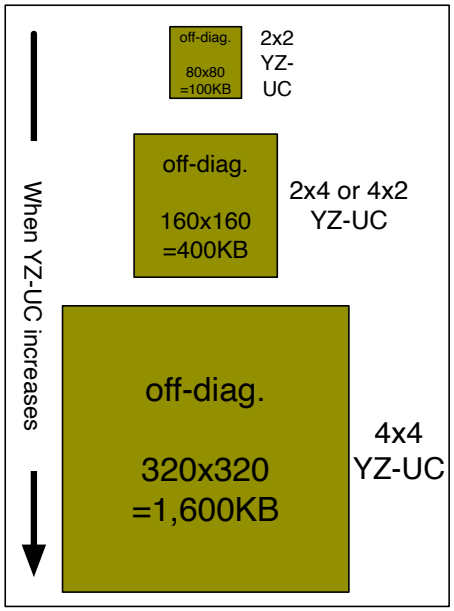
→ The computing cost of step 3 increases **extremely**

  - The number of sub-matrices ∝ (X-UC)^2 in step 3
  - The size of sub-matrix ∝ (YZ-UC)^2

→ **Performance optimization for step 3 is essential**



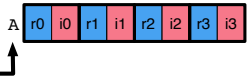




Strategies for performance enhancement
1. Data-restructuring of complex number matrix

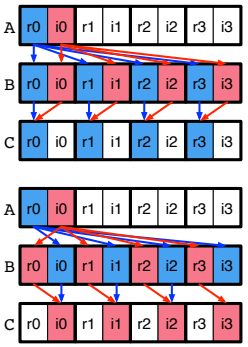
[AoS-type complex number and matrix]

```
typedef struct complexNum {
  double r; // real num
  double i; // imaginary num
} cNum_t; // complex num
cNum_t A[NxN]; // A: complex array(matrix)
```



[AoS-type complex matrix multiplication]

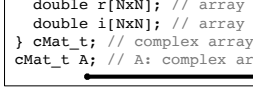
```
cNum_t A[NxN], B[NxN], C[NxN] // C=AxB
for(i=0; i<N; i++) {
  for(k=0; k<N; k++) {
    for(j=0; j<N; j+=4) {
      C[i][j+0].r += A[i][k].r * B[k][j+0].r \
                   - A[i][k].i * B[k][j+0].i;
      C[i][j+1].r += A[i][k].r * B[k][j+1].r \
                   - A[i][k].i * B[k][j+1].i;
      C[i][j+2].r += A[i][k].r * B[k][j+2].r \
                   - A[i][k].i * B[k][j+2].i;
      C[i][j+3].r += A[i][k].r * B[k][j+3].r \
                   - A[i][k].i * B[k][j+3].i;

      C[i][j+0].i += A[i][k].r * B[k][j+0].i \
                   + A[i][k].i * B[k][j+0].r;
      C[i][j+1].i += A[i][k].r * B[k][j+1].i \
                   + A[i][k].i * B[k][j+1].r;
      C[i][j+2].i += A[i][k].r * B[k][j+2].i \
                   + A[i][k].i * B[k][j+2].r;
      C[i][j+3].i += A[i][k].r * B[k][j+3].i \
                   + A[i][k].i * B[k][j+3].r;
} } }
```



Change the data structure AoS to SoA

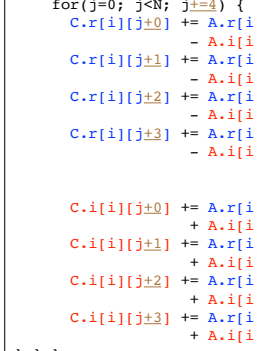
[SoA-type complex number and matrix]

```
typedef struct complexMat {
  double r[NxN]; // array of real num
  double i[NxN]; // array of imaginary num
} cMat_t; // complex array(matrix)
cMat_t A; // A: complex array(matrix)
```



[SoA-type complex matrix multiplication]

```
cMat_t A, B, C; // C=AxB
for(i=0; i<N; i++) {
  for(k=0; k<N; k++) {
    for(j=0; j<N; j+=4) {
      C.r[i][j+0] += A.r[i][k] * B.r[k][j+0] \
                   - A.i[i][k] * B.i[k][j+0];
      C.r[i][j+1] += A.r[i][k] * B.r[k][j+1] \
                   - A.i[i][k] * B.i[k][j+1];
      C.r[i][j+2] += A.r[i][k] * B.r[k][j+2] \
                   - A.i[i][k] * B.i[k][j+2];
      C.r[i][j+3] += A.r[i][k] * B.r[k][j+3] \
                   - A.i[i][k] * B.i[k][j+3];

      C.i[i][j+0] += A.r[i][k] * B.i[k][j+0] \
                   + A.i[i][k] * B.r[k][j+0];
      C.i[i][j+1] += A.r[i][k] * B.i[k][j+1] \
                   + A.i[i][k] * B.r[k][j+1];
      C.i[i][j+2] += A.r[i][k] * B.i[k][j+2] \
                   + A.i[i][k] * B.r[k][j+2];
      C.i[i][j+3] += A.r[i][k] * B.i[k][j+3] \
                   + A.i[i][k] * B.r[k][j+3];
} } }
```



- The elements accessed with **a stride of 2**
  - It is not desirable for fetching multiple data due to the poor data locality
  - The multiplication process cannot fully exploit the benefit of SIMD operation in KNL

- The elements accessed **continuously**
  - Excellent data locality for fetching multiple data
  - The benefit of SIMD can be fully exploited and multiplication can be done more efficiently than AoS-type complex matrix multiplication

Strategies for performance enhancement
2. Blocked (tiled) matrix multiplication

- Well known performance optimization techniques for increasing the cache hit ratio
  → How to determine the block size?
  ➔ Depend on the system architecture and the problem
  - In KNL processor, each core has 32KB L1 cache
  - In GV100 GPU, each streaming multiprocessor(SM) has 128KB L1 cache
    - Up to 96KB of 128KB can be used as shared memory for user
    - The remaining capacity is used as L1 cache for system
  - In complex number matrix multiplication using SIMD or SIMT, the continuous data-access occurs in the 2-matrices (A x **B** = **C**)
- We set the block size to 32x32 (16KB = 32x32x8x2)
  → Total 32KB for 2-matrices
  - In KNL, 32KB is perfect size for the L1 cache of KNL
  - In GPU, 32x32 is perfect number to generate 1024(=32x32) thread per thread-block to handle a single element per single thread and map a total of 2 thread-blocks per SM

Strategies for performance enhancement

3. Thread-scheduling for thread-utilization efficiency in step 3

```
for(i=NAL-2; i>-1; i--) { // NAL=8
  cMat_mul(...); // diag. computation
  ...
  for(j=(NAL-1); j>i; j--) {
    cMat_mul(...); // off-diag. computation
    ...
}}
```

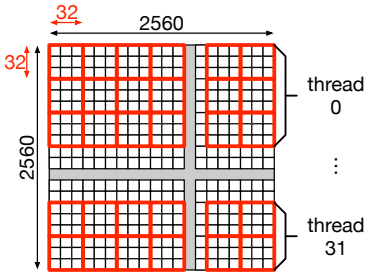
1,0	1,1			
j=2	2,1	cMat_mul w/ 32 threads	80 rows of blocks (32threads x 3)	= 83%
j=3	3,1	cMat_mul w/ 32 threads	80 rows of blocks (32threads x 3)	= 83%
j=4	4,1	cMat_mul w/ 32 threads	80 rows of blocks (32threads x 3)	= 83%
j=5	5,1	cMat_mul w/ 32 threads	80 rows of blocks (32threads x 3)	= 83%
j=6	6,1	cMat_mul w/ 32 threads	80 rows of blocks (32threads x 3)	= 83%
j=7	7,1	cMat_mul w/ 32 threads	80 rows of blocks (32threads x 3)	= 83%

The average efficiency: always **~83%**

Apply the thread-scheduling

- 2560x2560 matrix, 32x32 block, 32 threads
  - 2560x2560 matrix → 80x80 block matrix
  - 80 rows of blocks should be parallelized to 32threads → 80/32 = 2.5

➔ **load imbalance occurs**



- Our main idea:
  - Perform multiple MatMul simultaneously
  - Adjust # of thr. used in a single MatMul
- The average efficiency increases from ~83% to ~100% as the size of X-UCs increases
  - The efficiency gain is up to **~17%**

```
for(i=NAL-2; i>-1; i--) { // NAL=8
  cMat_mul(...); // diag. computation
  ...
  // parallelized w/ thread-scheduling
  for(j=(NAL-1); j>i; j--) {
    cMat_mul(...); // off-diag. computation
    ...
}}
```

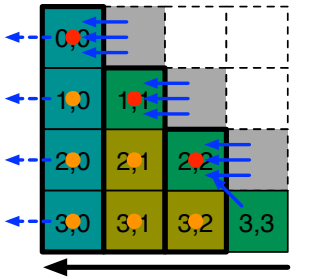
1,0	1,1			
j=2~3	2,1	cMat_mul w/ 16 threads	80 rows of blocks (16threads x 5)	= 100%
	3,1	cMat_mul w/ 16 threads	80 rows of blocks (16threads x 5)	= 100%
j=4~7	4,1	cMat_mul w/ 8 threads	80 rows of blocks (8threads x 10)	= 100%
	5,1	cMat_mul w/ 8 threads	80 rows of blocks (8threads x 10)	= 100%
	6,1	cMat_mul w/ 8 threads	80 rows of blocks (8threads x 10)	= 100%
	7,1	cMat_mul w/ 8threads	80 rows of blocks (8threads x 10)	= 100%

The average efficiency: **~96%** in 2 X-UC, **~99.7%** in 100 X-UC

Strategies for performance enhancement
4. Offload computing with GPU accelerators in step 3

- **More computation, Less data-transfer**
  is one of the keys to efficient offload computing
  - So, we designed a scheme of offload computing that can exploit the strength of GPU devices
- In our scheme, step 3 is processed in the unit of sub-matrix columns
  - The number of computations ∝ (X-UCs)[2]
    * same as the number of sub-matrices
  - The number of data-transfers ∝ (X-UCs)
    H to G: 3 per column (4 in the first column)
    G to H: # of sub-matrices in the last column

  → More beneficial as the nanostructure Becomes longer along the X-direction



- diag. computation

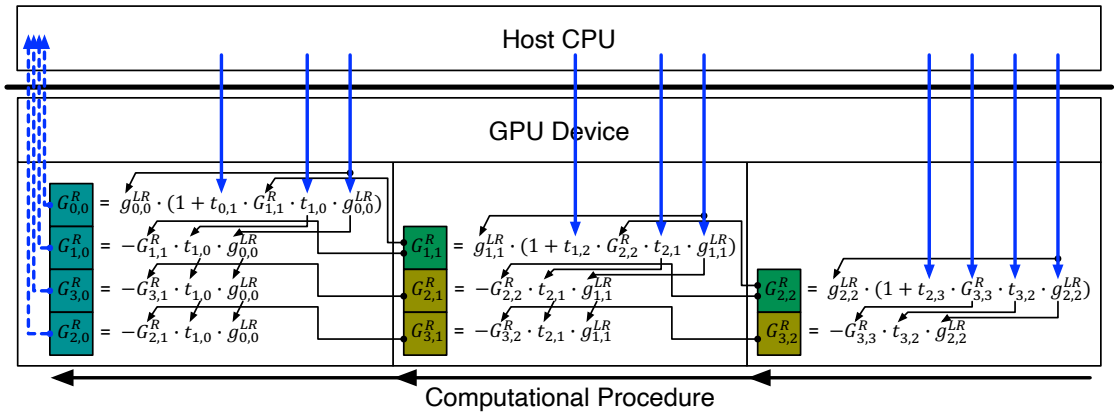
$$G^R_{i,i} = g^{LR}_{i,i} \cdot (1 + t_{i,i+1} \cdot G^R_{i+1,i+1} \cdot t_{i+1,i} \cdot g^{LR}_{i,i})$$

- off-diag. computation

$$G^R_{j,i}|_{i<j} = -G^R_{j,i+1} \cdot t_{i+1,i} \cdot g^{LR}_{i,i}$$

Host to Device data transfer

Device to Host data transfer



- As the X-UCs increases from 2 to 100,
  - The number of data-transfers increases from **14**(4+3x2 + 4) to **798**(4+3x198 + 200)
  - The number of computations increases from **9**(2+3+4) to **20,999**(2+3+…+200)

  → The ratio of **data-transfer** to **computation** is about **1** : **26** in X-UCs = 100

Maximize the GPU resource occupancy
The CUDA Occupancy Calculator

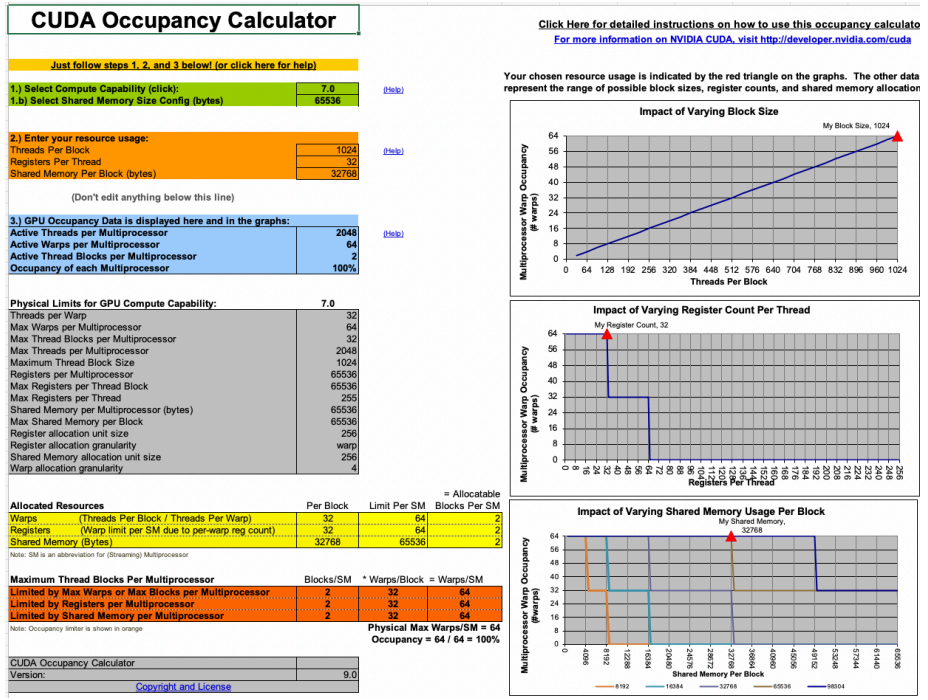
- NVIDIA provides the CUDA Occupancy Calculator
  - Compile with compute capability and –Xptxas –v
  - Enter the information into the CUDA Occupancy Calculator

  → The occupancy of GPU resource is reported as a percentage
- In this study:
  - The compute capability of NVIDIA Quadro GV100: 7.0
    - Max registers per SM: 65,536
    - Max threads per SM: 2,048
    - Max shared memory per SM: 96KB
  - Set up the computing resources as below:
    - The number of registers per thread: 32
    - The number of threads per thread-block: 1,024
    - The size of shared memory per thread-block: 32KB
      * the remaining 64KB per SM is used as L1 cache

  → A total of 2 thread-blocks can be mapped to each SM
  All computing resources of the SM are fully utilized.

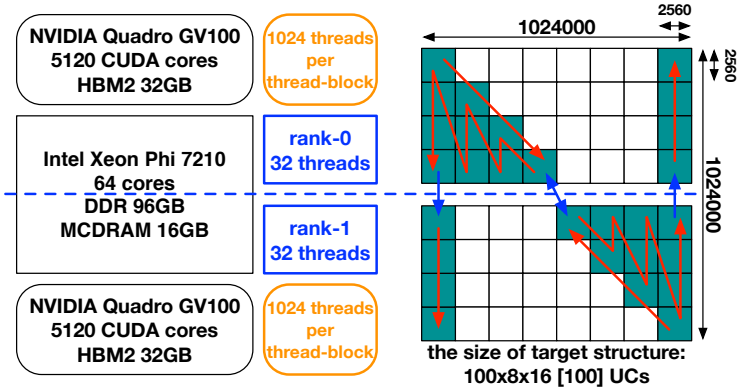
  ➔ We achieved a **100%** occupancy of GPU resources

```
ptxas info    : Compiling entry function '_Z27GINJA_multiplyMatrix_KernelPdS_iS_S_iS_S_' for 'sm_70'
ptxas info    : Function properties for _Z27GINJA_multiplyMatrix_KernelPdS_iS_S_iS_S_
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 32 registers, 32768 bytes smem, 416 bytes cmem[0]
```

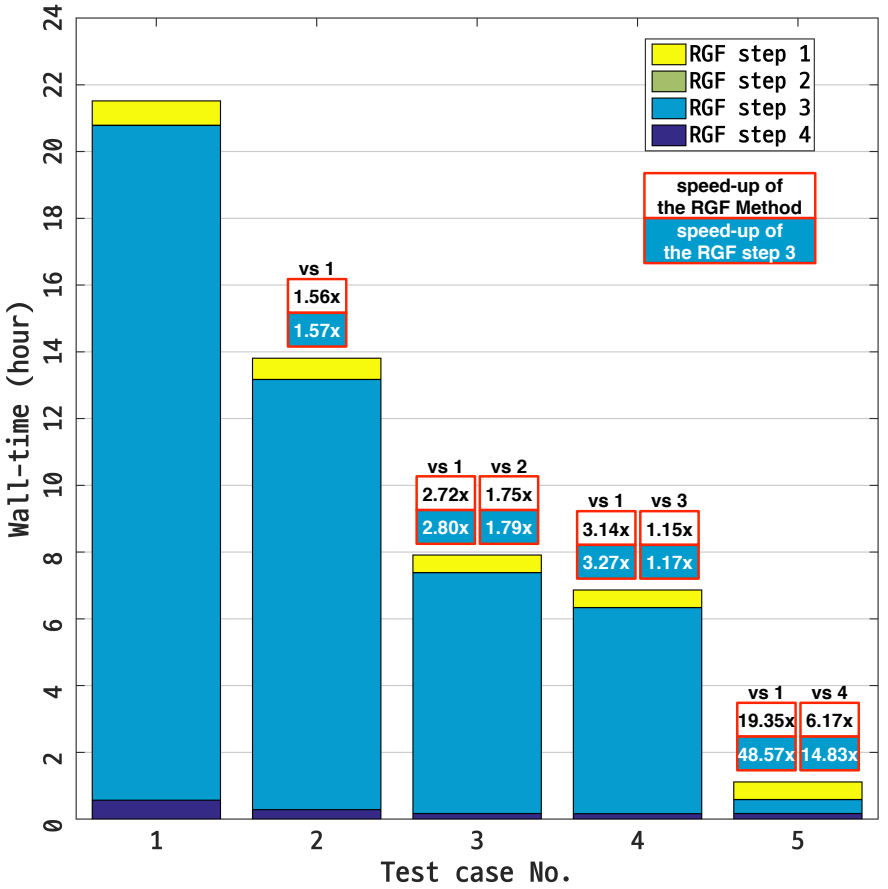


Benchmark tests

Test environments and Results of benchmark tests



Strategy \ Test case	1	2	3	4	5
MPI & OpenMP	O	O	O	O	O
Data-restructuring	X	O	O	O	O
Blocked MatMul	X	X	O	O	O
Thread-scheduling	X	X	X	O	O
Offload computing	X	X	X	X	O



- All the 4 techniques contribute significantly to enhancing the speed of RGF, particularly step 3
- When all the four techniques are applied, the entire computation can be completed in a about 1 hour
- The percentage of the total wall-time taken by the step 3 decreased from ~95% (case 1) to ~37% (case 5)

→ The most-time consuming part becomes the step 1 (~47% of the total wall-time)

Conclusions
Summary

- In this study, we proposed technical strategies to accelerate the Recursive Green's Function algorithm
- The effective of proposed technical strategies is verified by performing benchmark tests in manycore computing resources
- We observe the wall-time of the entire RGF process can be reduced by a factor of ~19.3x
- The details of the techniques are quite universal since the multiplication of dense complex matrices is one of the most basic operations

Thanks for your attention!