HPCG Performance Improvement on the K computer ～10min. brief～

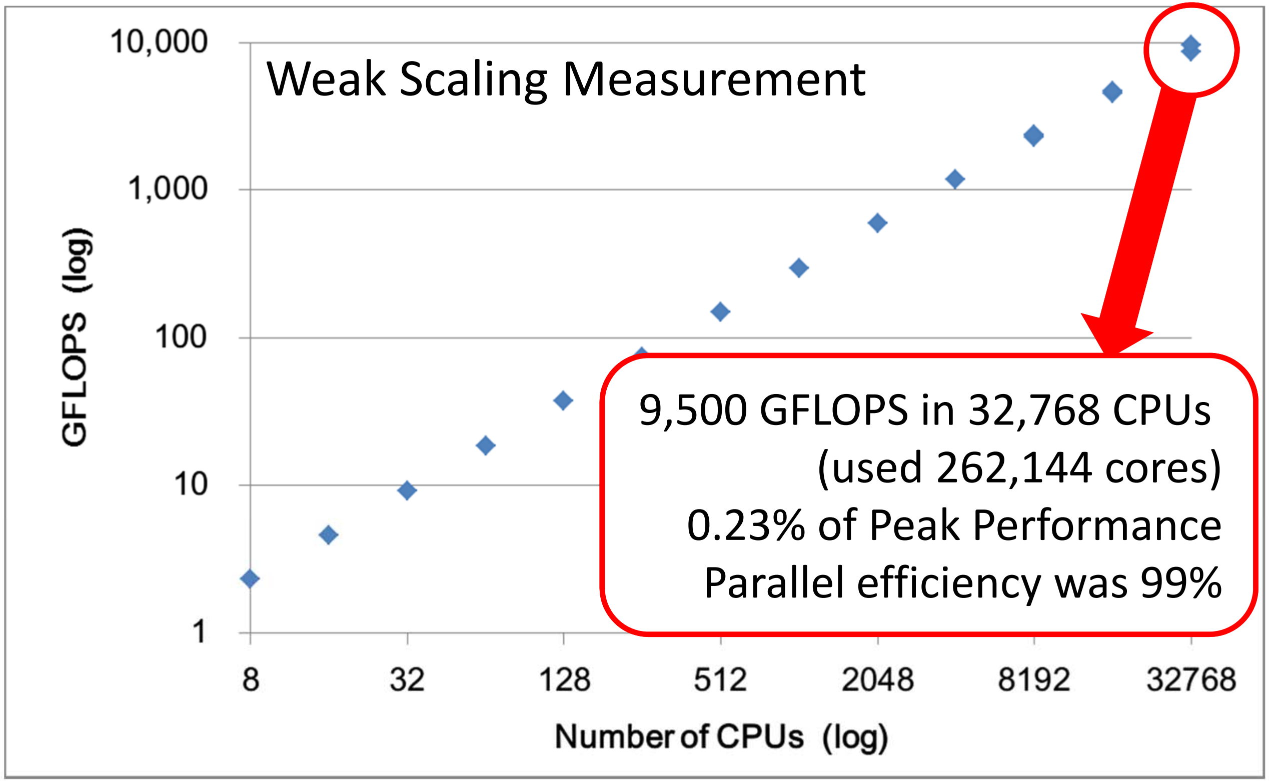
Kiyoshi Kumahata, Kazuo Minami

RIKEN AICS

HPCG BoF #273

SC14 New Orleans

Evaluate Original Code 1/2



9,500 GFLOPS in 32,768 CPUs (used 262,144 cores)
0.23% of Peak Performance
Parallel efficiency was 99%

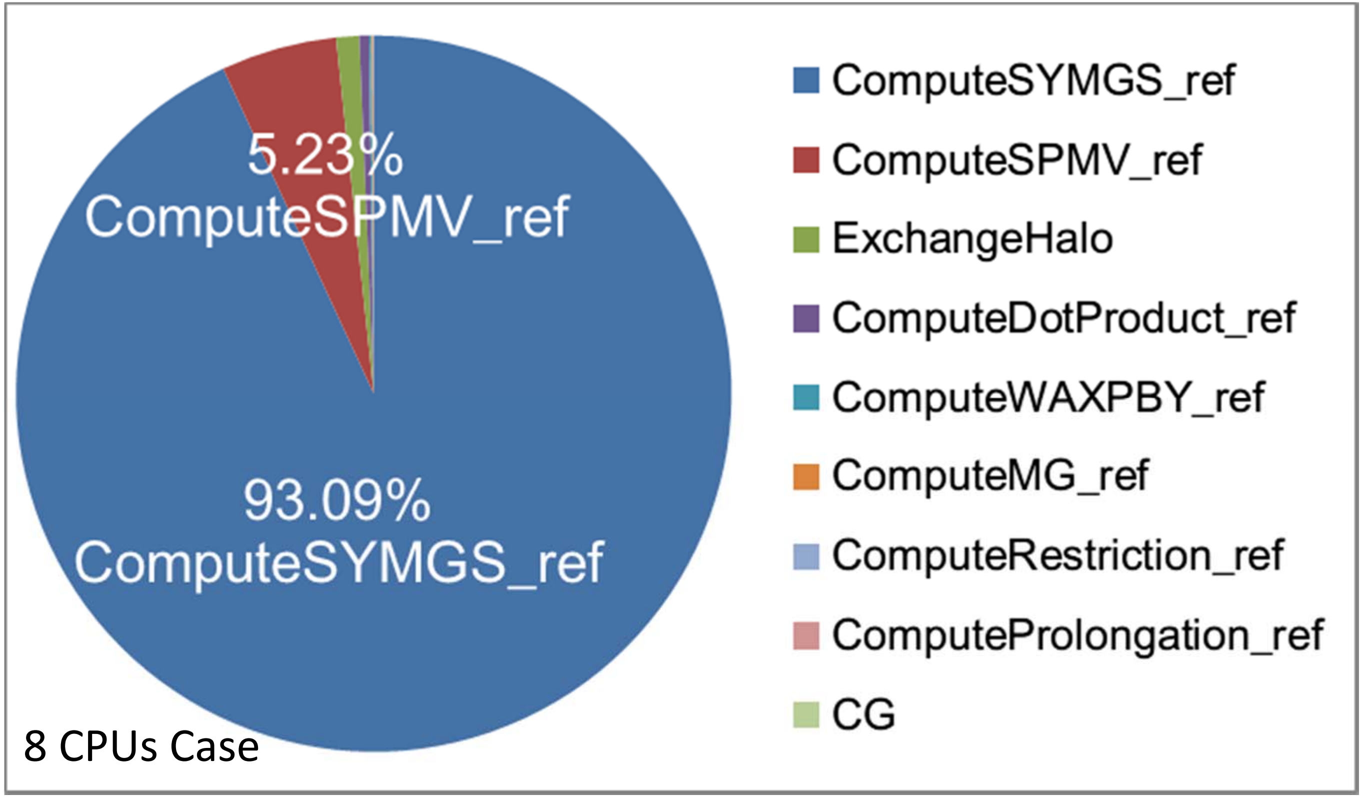
GFLOPS values are from “Total with convergence and optimization phase overhead” in the YAML file

Conditions

- 128^3 DoF/CPU
- 8 threads/CPU
- 10min. for CG
- Typical Compile Option

Good scalability was obtained !!
So tunings for parallel performance is not necessary !!

Evaluate Original Code 2/2



- Procedures time ratio in the total CG running time by profiler
- 98% of total time consists by major 2 procedures (only computation)

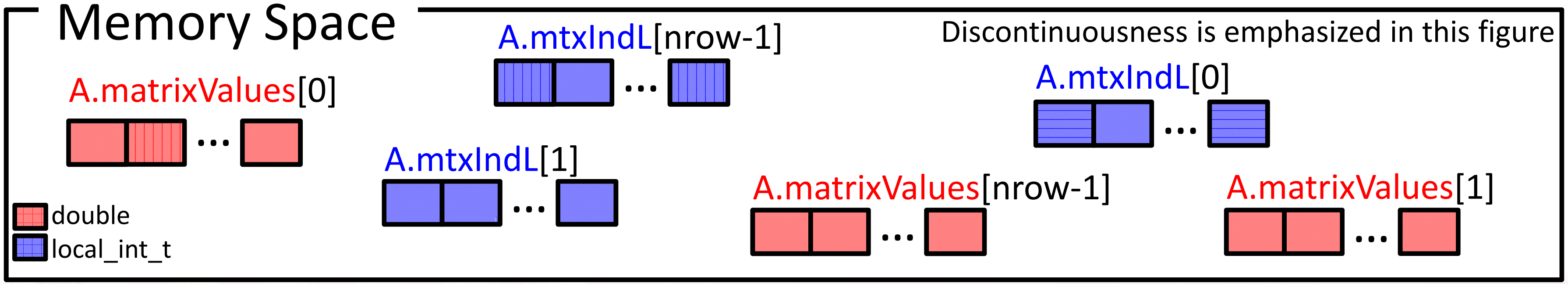
Tuning for Single CPU performance is necessary !!

Tune1: Continuous Memory 1/4

Essence of Matrix Memory Allocation Part (Original)

```
for(local_int_t i=0; i< localNumberOfRows; ++i) {
   mtxIndL[i]      = new local_int_t [numberOfNonzerosPerRow];
   matrixValues[i] = new double      [numberOfNonzerosPerRow];
   mtxIndG[i]      = new global_int_t[numberOfNonzerosPerRow];
}
```

- Memory for storing a matrix row is allocated separately
- Each row information are arranged discontinuous in memory space. It disturbs efficient cache memory utilization when computation



Discontinuousness is emphasized in this figure

Tune1: Continuous Memory 2/4

Essence of Matrix Memory Allocation Part (Modified)

```
int total_size;
local_int_t*  templ = new local_int_t [total_size];
double*       tempd = new double      [total_size];
global_int_t* tempg = new global_int_t[total_size];
int ofset = 0;
for (local_int_t i=0; i<localNumberOfRows; ++i){
    mtxIndL[i]       = tmpl + ofset
    matrixValues[i] = tmpd + ofset
    mtxIndG[i]       = tmpg + ofset
    ofset += max_nnz;
}
```

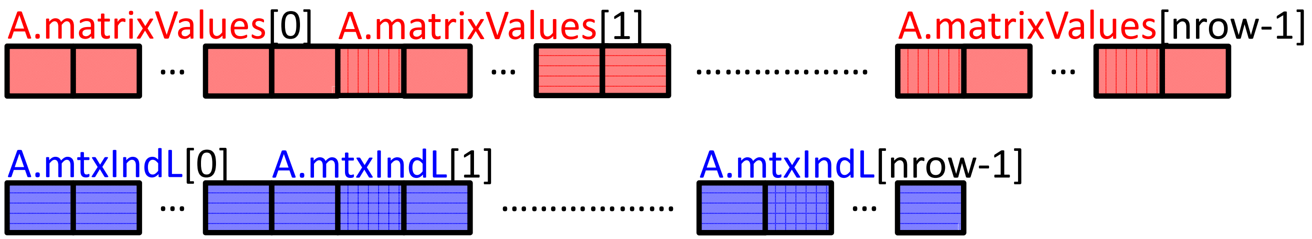
Allocate all once

Assign pointer for each row

max number of nonzeros for a row in matrix

- Every row information are arranged continuously

Memory Space



Tune1: Continuous Memory 3/4

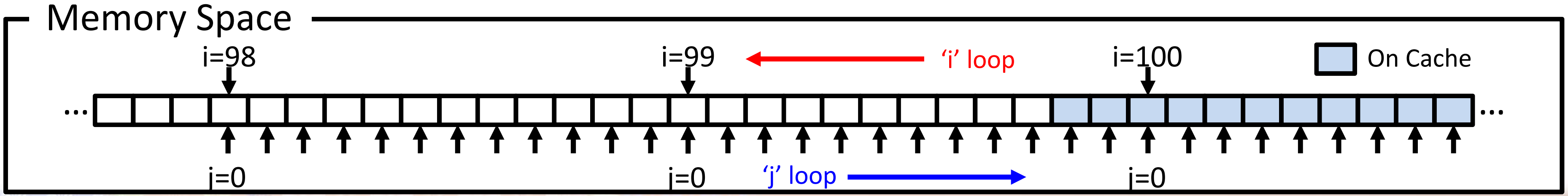
Essence of of Backward Loop of SYMGS (Original)

```
for(int i=nrow-1; i>=0; i--){
   double* curValues  = A.matrixValues[i];
   int*    curIndices = A.mtxIndL[i];
   int     curNZ      = A.nonzerosInRow[i];
   double  curDiag    = matrixDiagonal[i][0];

   double sum = rv[i];
   for(int j=0; j<curNZ; j++){
      int curCol = curIndices[j];
      sum -= curValues[j]* xv[curCol];
   }
   sum   += xv[i] * curDiag;
   xv[i] = sum / curDiag;
}
```

In original code, loop direction of inner loop in backward loop of SYMGS is reverse

- Outer ‘i’ loop goes backward direction
- Inner ‘j’ loop goes forward direction

After outer ‘i’ loop switched to next iteration, memory address referred by inner 'j' loop first iteration will not on cache because inner ‘j’ loop goes to reverse direction to outer ‘i’ loop



Tune1: Continuous Memory 4/4

Essence of of Backward Loop of SYMGS (Modified)

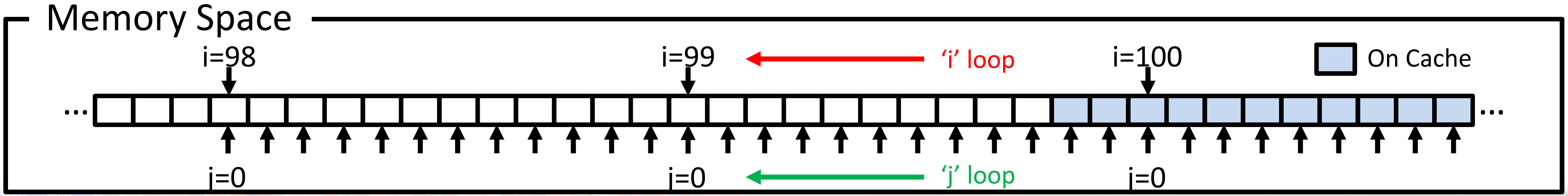
```
for(int i=nrow-1; i>=0; i--){
   double* curValues  = A.matrixValues[i];
   int*    curIndices = A.mtxIndL[i];
   int     curNZ      = A.nonzerosInRow[i];
   double  curDiag    = matrixDiagonal[i][0];

   double sum = rv[i];
   for(int j=curNZ-1; j>=0; j--){
      int curCol = curIndices[j];
      sum -= curValues[j]* xv[curCol];
   }
   sum   += xv[i] * curDiag;
   xv[i] = sum / curDiag;
}
```

Inner 'j' loop direction is not constraint to be forward direction. So it can be reversed

- Outer 'i' loop goes backward direction
- Inner 'j' loop goes backward direction

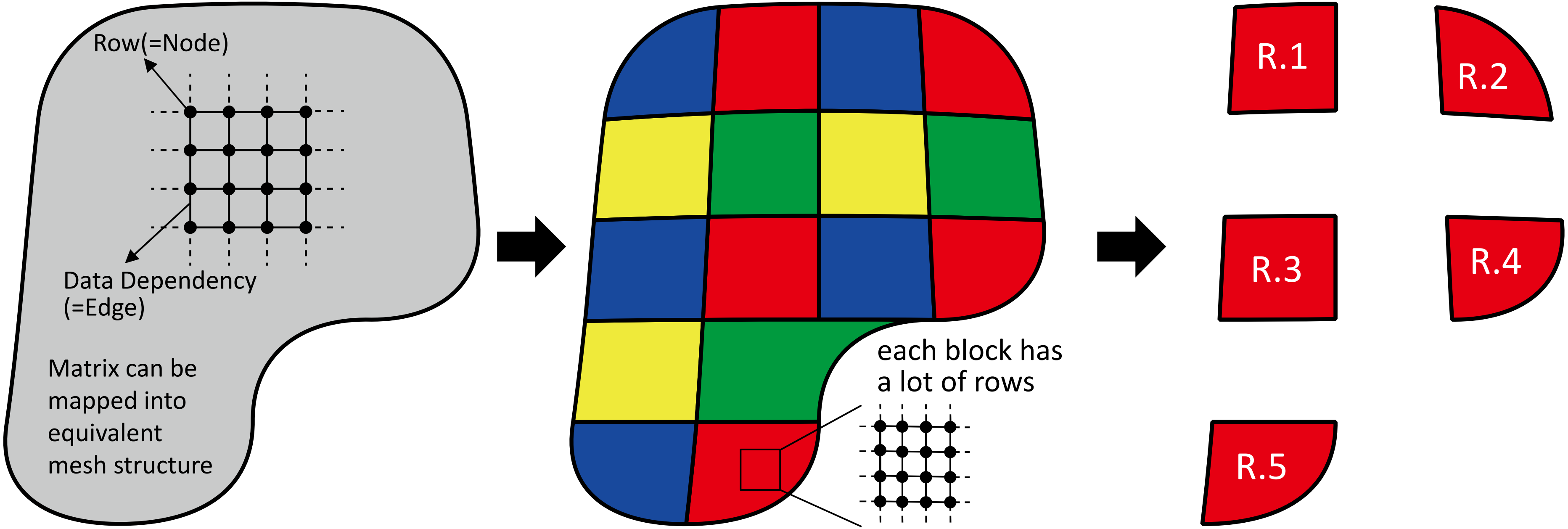
By reversing inner 'j' loop direction, memory address referred by inner 'j' loop first iteration will be on cache!

And prefetch mechanism can predict easily required memory address.



Tune2: Coloring for SYMGS 1/2

To avoid side effect of coloring (cache thrashing), we employed new way using block

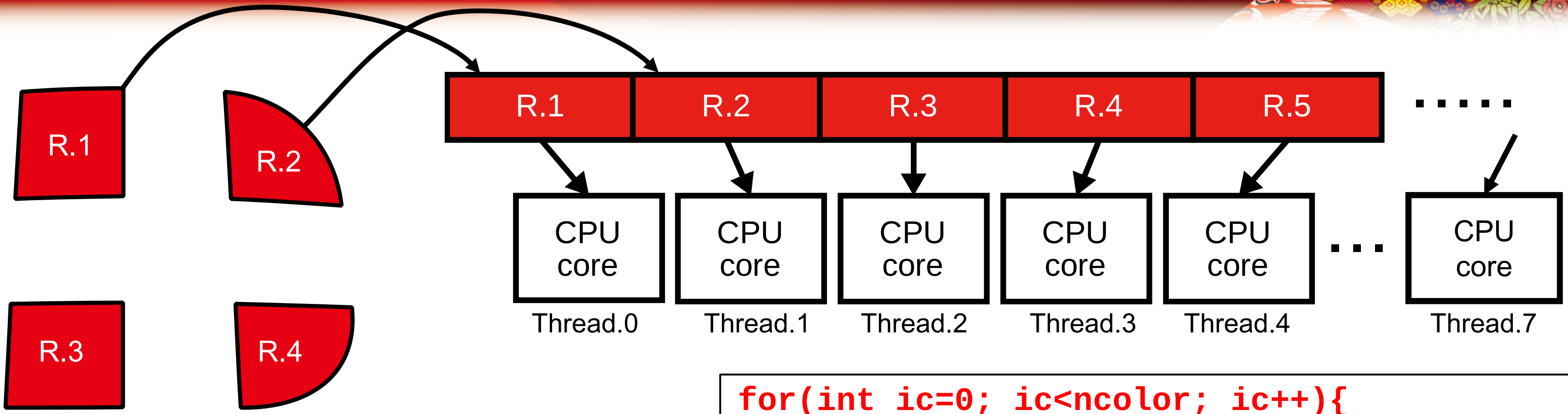


Matrix can be mapped into equivalent mesh structure

each block has a lot of rows

1. Mesh structure is divided into lots of blocks. Each block has a lot of rows. And number of rows in block is as same as possible to avoid work imbalance
2. In here, color is assigned to a block instead of a node. And different color is assigned into neighboring blocks.
3. The, there are no data dependencies between blocks in same color.

Tune2: Coloring for SYMGS 2/2



There are no data dependencies between blocks in same color.

But dependencies occur among the rows in same block.

Therefore, thread parallelism is applied to block, code is modified into right figure.

Add middle loop to iterate block.
And parallelize block loop by inserting a directive

```
for(int ic=0; ic<ncolor; ic++){
   #pragma omp parallel for
   for(int ib=0; ib<nblock[ic]; ib++){
      for(int i=st[ic][ib]; i<=ed[ic][ib]++){
         …Innermost loop…
      }
   }
}
```

Tune3: Loop Optimization 1/3

Sample for SPMV (Original)

```
for(local_int_t i=0; i<nrow; i++){
  double sum = 0.0;
  const double* const cur_vals = A.matrixValues[i];
  const local_int_t* const cur_inds = A.mtxIndL[i];
  const int cur_nnz = A.nonzerosInRow[i];

  for(int j=0; j<cur_nnz; j++)
    sum += cur_vals[j]*xv[cur_inds[j]];
  yv[i] = sum;
}
```

Complicated access path for matrix nonzero information via pointer

Software pipeline don’t work well for short loop

SYMGS also has same problem

Tune3: Loop Optimization 2/3

Modify Sample for SPMV

```
double* val = A.matrixValues[0];
local_int_t* index = A.mtxIndL[0];
for(local_int_t i=0; i<nrow-1; i=i+2){
  id1 = (i  )*max_nnz;
  id2 = (i+1)*max_nnz;
  sum1 = 0.0;
  sum2 = 0.0;
  for(int j=0; j<max_nnz; j++){
    sum1 += val[id1+j] * xv[index[id1+j]];
    sum2 += val[id2+j] * xv[index[id2+j]];
  }
  yv[i  ] = sum1;
  yv[i+1] = sum2;
}
```

matrix is continuous in memory,
so simple path is able

Tune3: Loop Optimization 2/3

Modify Sample for SPMV

```
double* val = A.matrixValues[0];
local_int_t* index = A.mtxIndL[0];
for(local_int_t i=0; i<nrow-1; i=i+2){
  id1 = (i  )*max_nnz;
  id2 = (i+1)*max_nnz;
  sum1 = 0.0;
  sum2 = 0.0;
  for(int j=0; j<max_nnz; j++){
    sum1 += val[id1+j] * xv[index[id1+j]];
    sum2 += val[id2+j] * xv[index[id2+j]];
  }
  yv[i  ] = sum1;
  yv[i+1] = sum2;
}
```

2. Software Pipelined

3. Unroll 2

1. Unroll Full

Avoiding short loop is necessary

SPMV for max_nnz=7

SPMV for max_nnz=13

SPMV for max_nnz=27

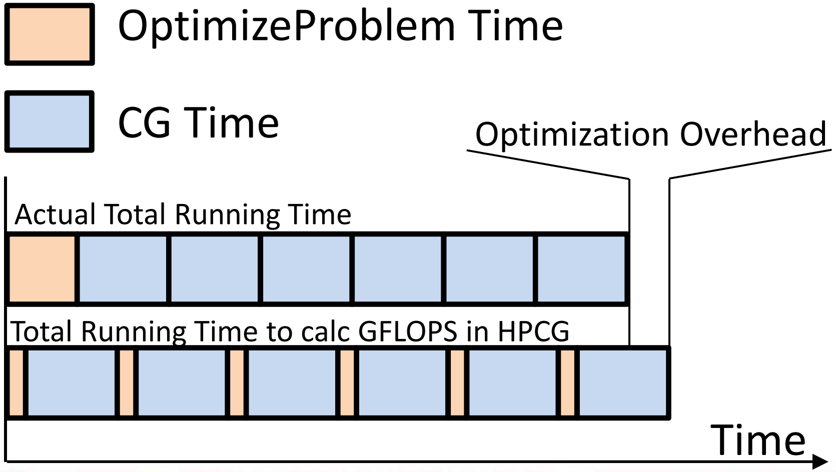
- make several SPMV for various max_nnz
- Full unrolling innermost loop j

To increase software pipelined operations

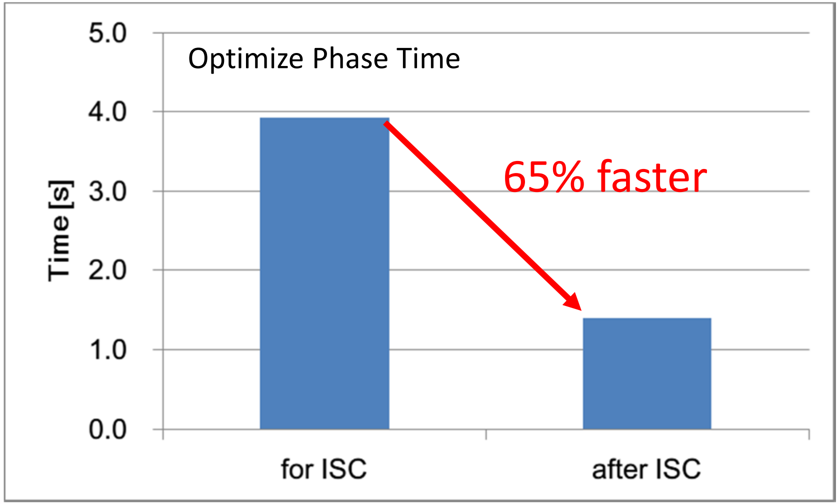
- 2 unrolling loop i

Tuning after ISC14

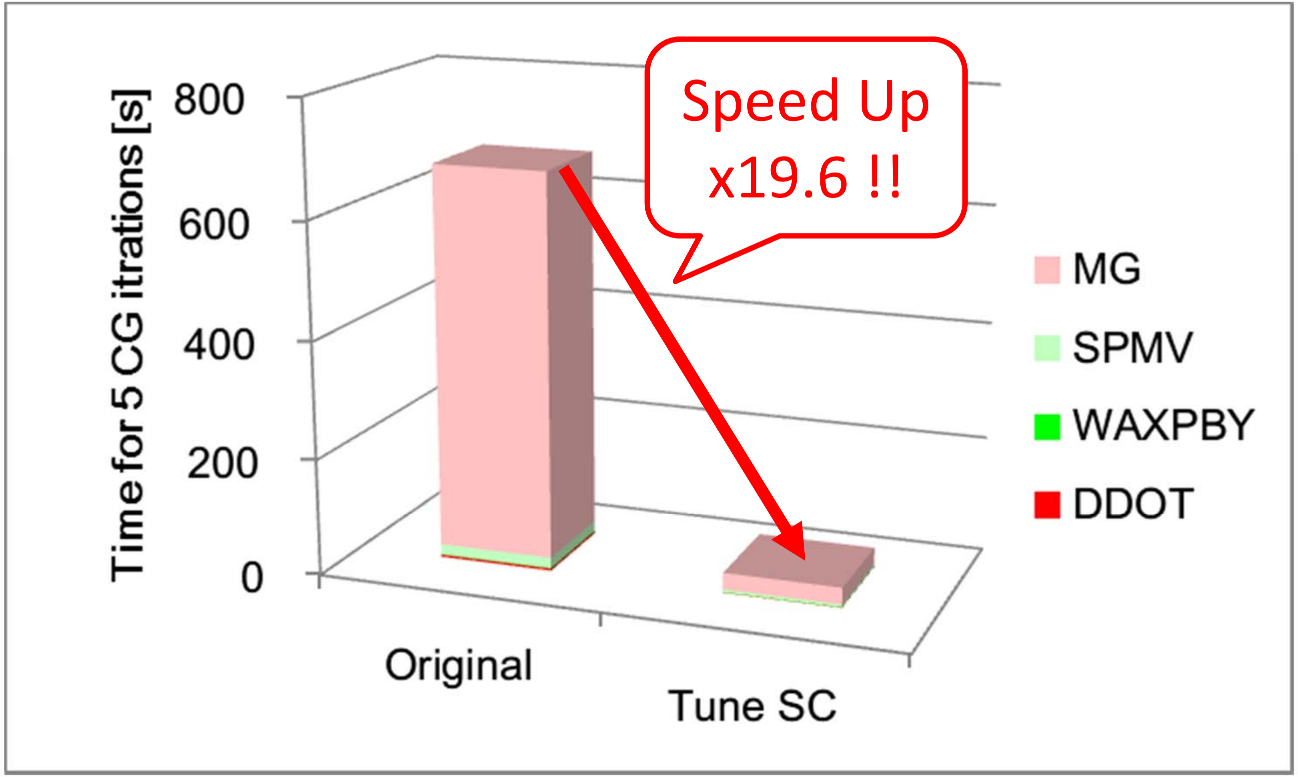
1. Parameter adjustment
   - Running environment parameter on the K
   - Block size
2. Code Refinement for OptimizeProblem.cpp to decrease overhead



OptimizeProblem Time
CG Time
Optimization Overhead
Actual Total Running Time
Total Running Time to calc GFLOPS in HPCG
Time



Summary: Latest Tuning Effect



Speed Up x19.6 !!

Employ tuning ways

- Continuous Memory
- Coloring for SYMGS multithreading with blocking
- Loop optimization
- Parameter adjustment
- Code refinement for OptimizeProblem

Good improve obtained