Performance portable single source-code implementation of sparse linear algebra operations on CPUs and GPUs

Leopold Grinberg (IBM/Research)

leopoldgrinberg@us.ibm.com

ocus on:

sparse matrix-vector multiplication (SpMv) and
parallel symmetric Gauss-Siedel solver (SYMGS)

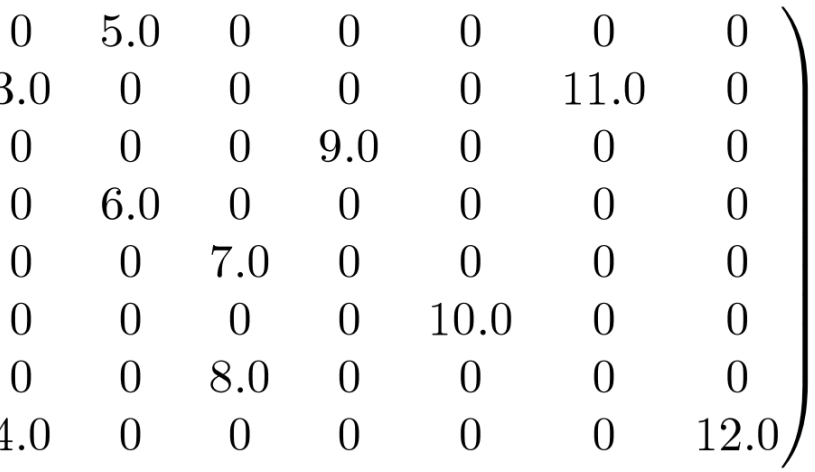
Main objective:

performance portable implementation of SpMv and SYMGS on CPU and GPU using OpenMP4 directives.

isclaimer:

performance characteristics have been obtained on the IBM Firestone node (two Power8 CPUs + two K80 GPUs, connected with the PCIe 3.0) and development version of the IBM LLVM compiler supporting OpenMP 4.0.

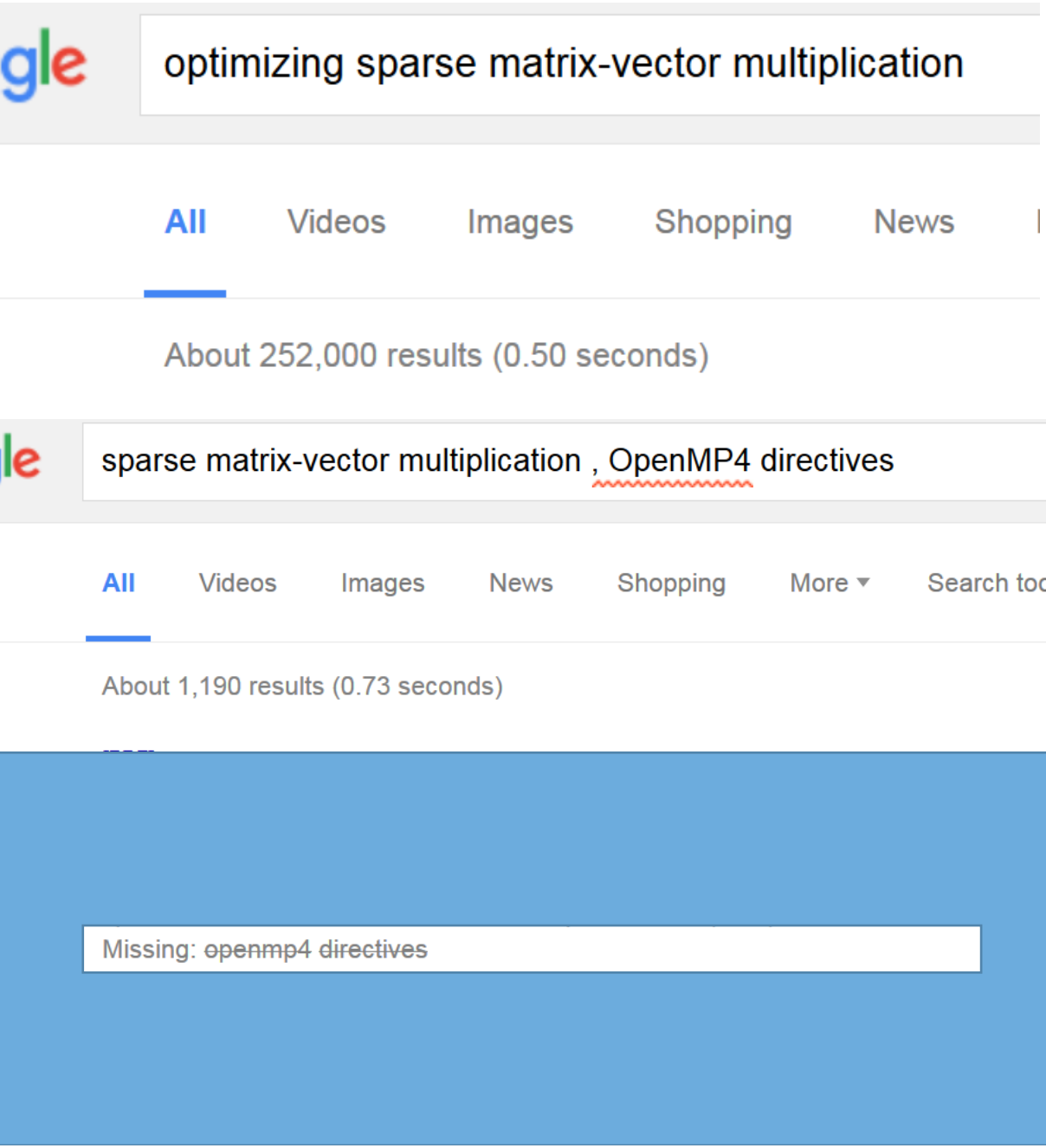
rse matrix - vector multiplication: y=A*x.



A	J	offsets
1.0, 5.0;	0, 2;	0;
3.0, 11.0;	1, 6;	2;
9.0;	4;	4;
6.0;	2;	5;
7.0;	3;	6;
2.0, 10.0;	0, 5;	7;
8.0;	3;	9;
4.0, 12.0;	1, 7;	10;
		12;

Basic algorithm

```
for (row = 0; row < Nrows; ++row){
   sum = 0.0;
   j1 = offsets[row];  j2 = offsets[row+1]
   for (col = 0; col < (j2-j1); ++col)
       sum +=  A[j1+col] * x[ J [j1+col] ];
   y[row] = sum;
}
```




```
for (row = 0; row < Nrows; ++row){
   sum = 0.0;
   j1 = offsets[row];  j2 = offsets[row
   for (col = 0; col < (j2-j1); ++col)
       sum +=  A[j1+col] * x[ J [j1+col
   y[row] = sum;
}
```

Results: SpMv

mpirun -np 4 ./bind.sh ./test_HPCG.exe

CPU CSR SpMv time = 8.48e-02 **GF/s=21.14**

Original (CSR) CPUs only;

np 4 ./bind.sh ./test_HPCG.exe

ACKED_SpMv time = 1.87e-02 **GF/s=95.85**

erformance portable: CPUs + GPUs

mpirun -np 4 ./bind.sh ./test_HPCG.exe

BLOCK_PACKED_SpMv time = 8.71e-02 **GF/s=20**

Performance portable: CPU only

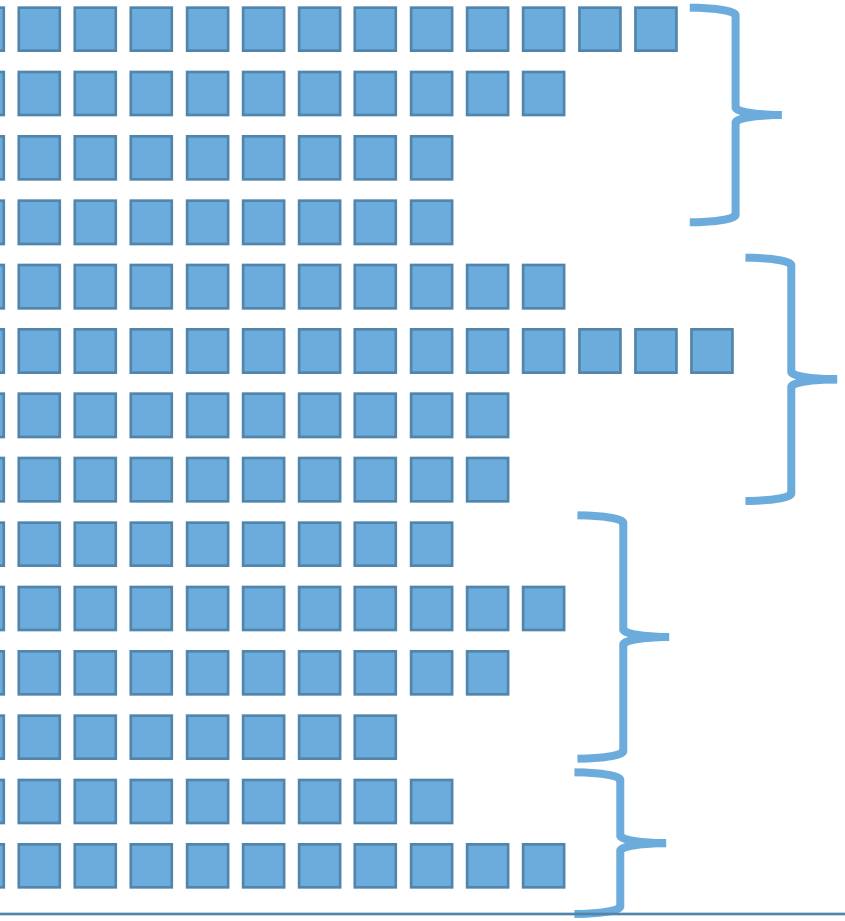
rse: SpMv time: 25.68 [milliseconds] –> 17.46GF/s (on single GPU; 69.84GF/s on 4 GPUs); all data in the GPU
ry

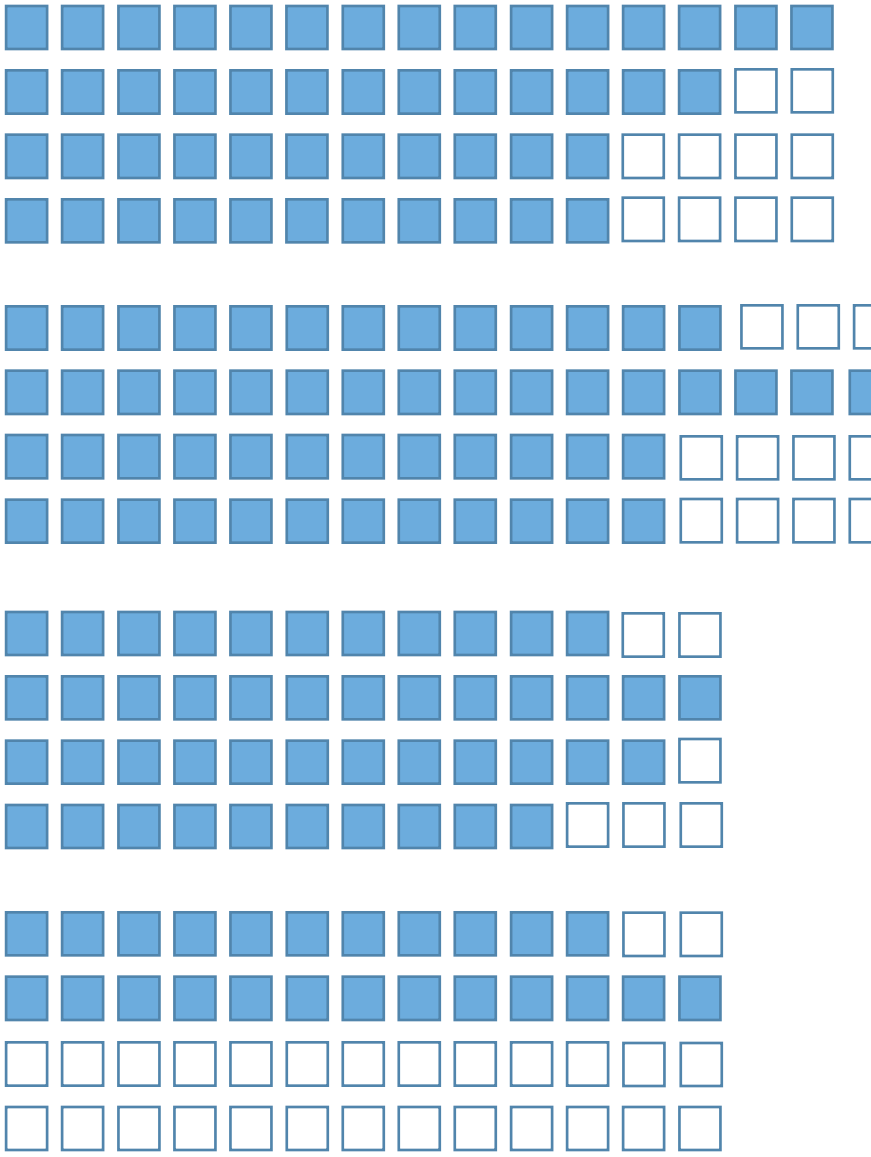
Major considerations to achieve performance portability:

- sufficient level of parallelism (not a problem in SpMv)
- same data layout allowing efficient memory access on different devices
- switching execution from/to CPU to/from GPU at run time
- no need to hardcode kernel parameters
- same source code (no “#ifdef”)

Data layout allowing efficient memory access on different devices*

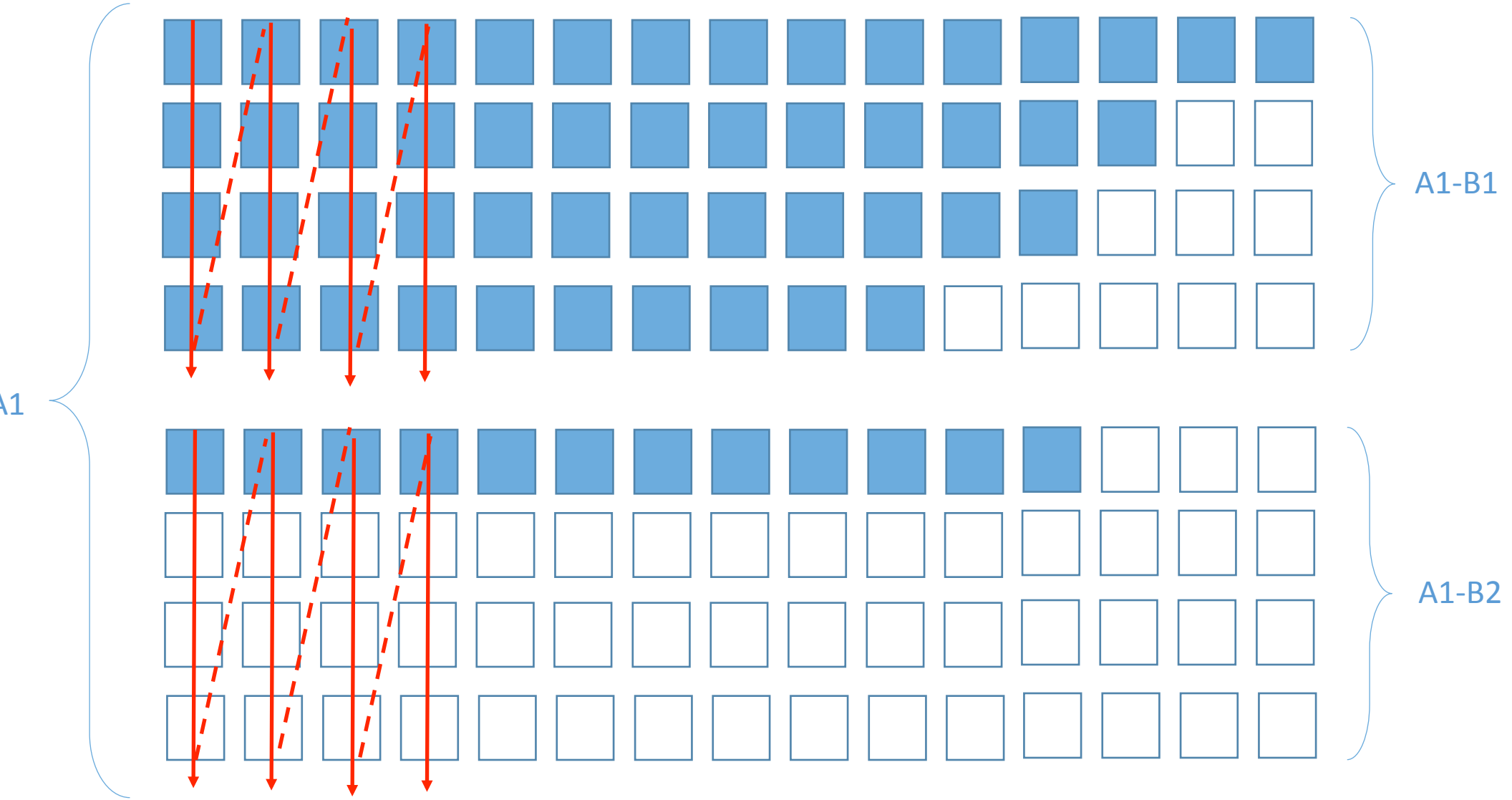
CSR → SELL-C (Sliced ELLPACK)



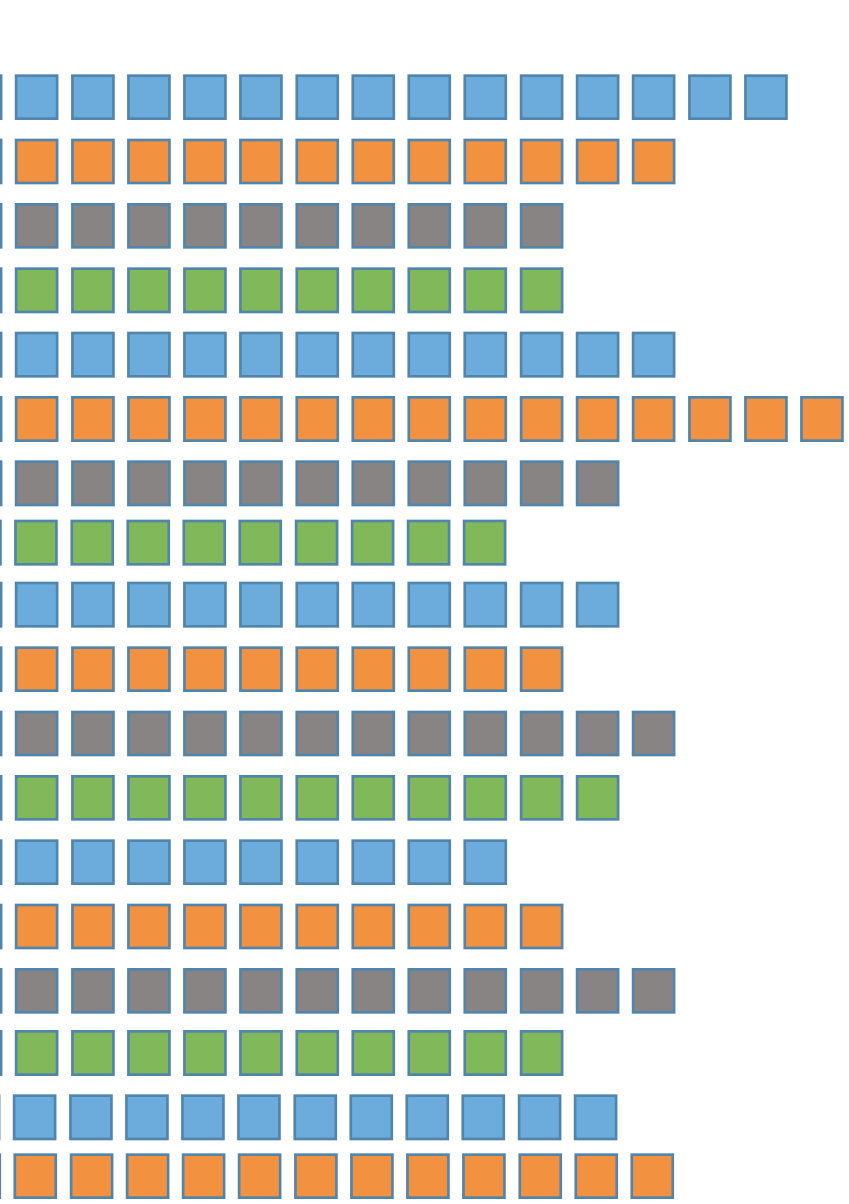


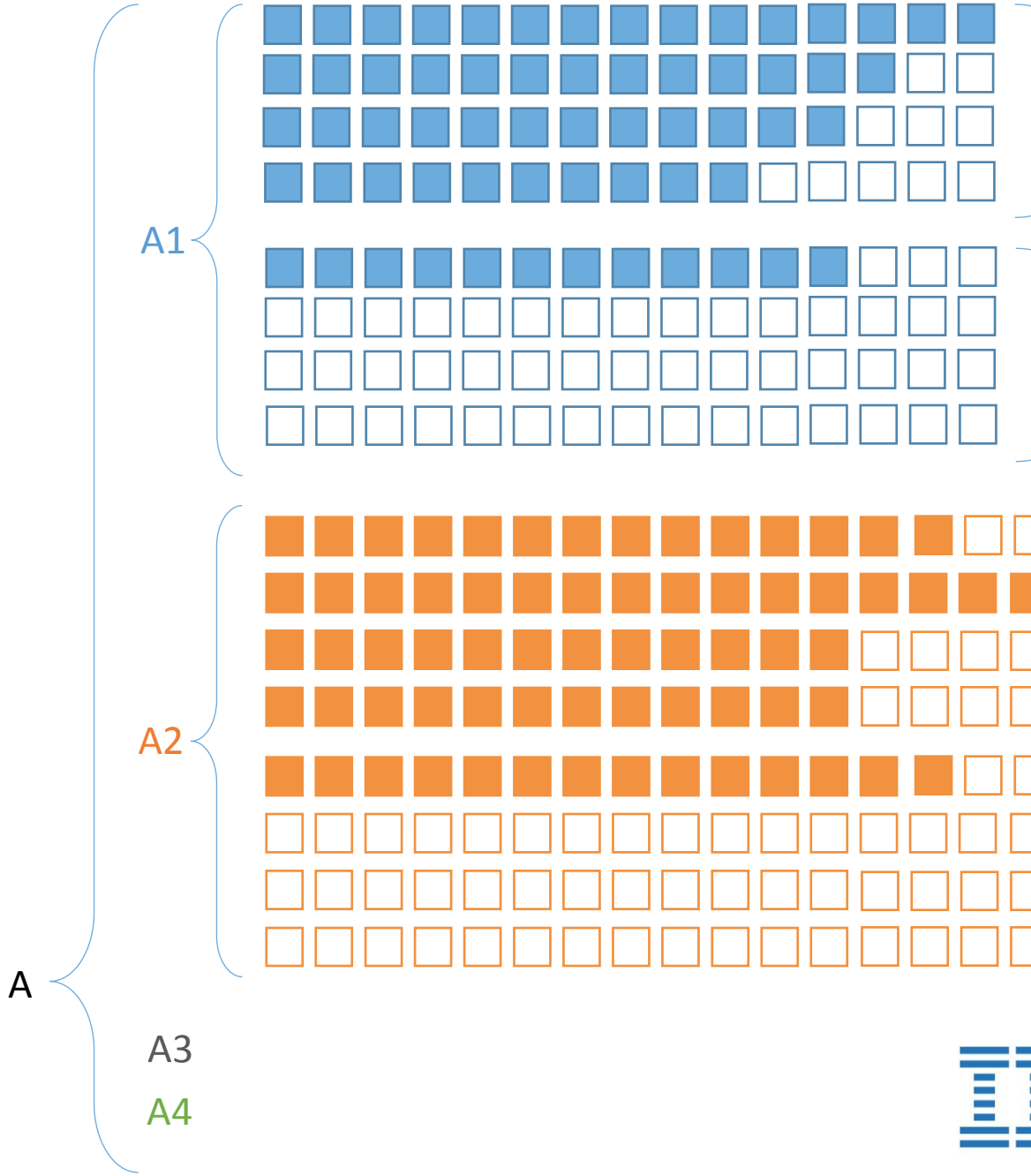
er *at el. A unified sparse matrix data format for efficient general sparse matrix-*
ply on modern processors with wide SIMD units, SIAM Journal on Scientific
2014 36:5.

ata layout allowing efficient memory access on different devices

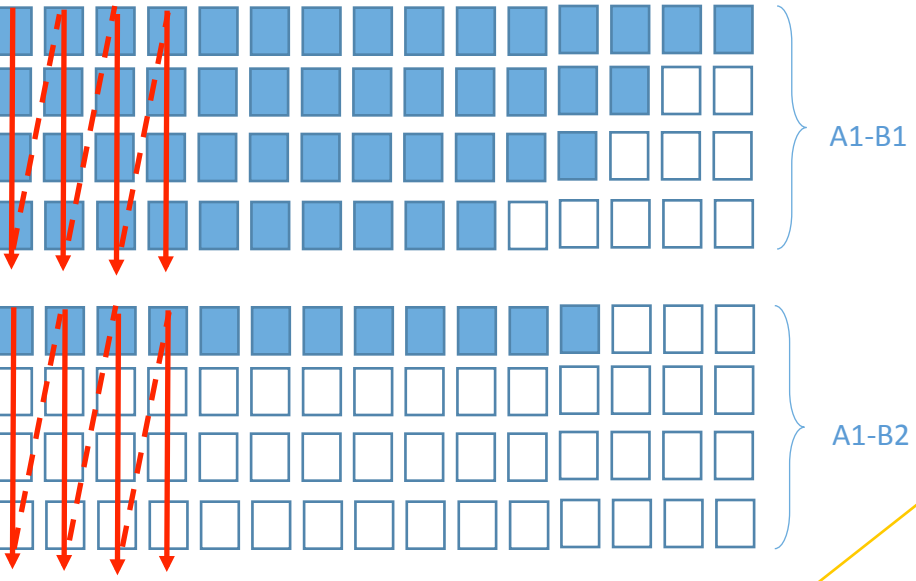


ayout allowing efficient memory access on different devices: +multicol





*y=A*x:*
multiple levels of parallelism




```
lor = 0; color < Ncolors; ++colors){

blk = blk_start; blk < blk_end; ++blk){

r (blk_row = 0; blk_row < Nrows[blk]; ++blk_row){

for (col = 0; col < nnz; ++col){
    sum += A[color][blk][f(blk_row,col)] * x[k];
}
```

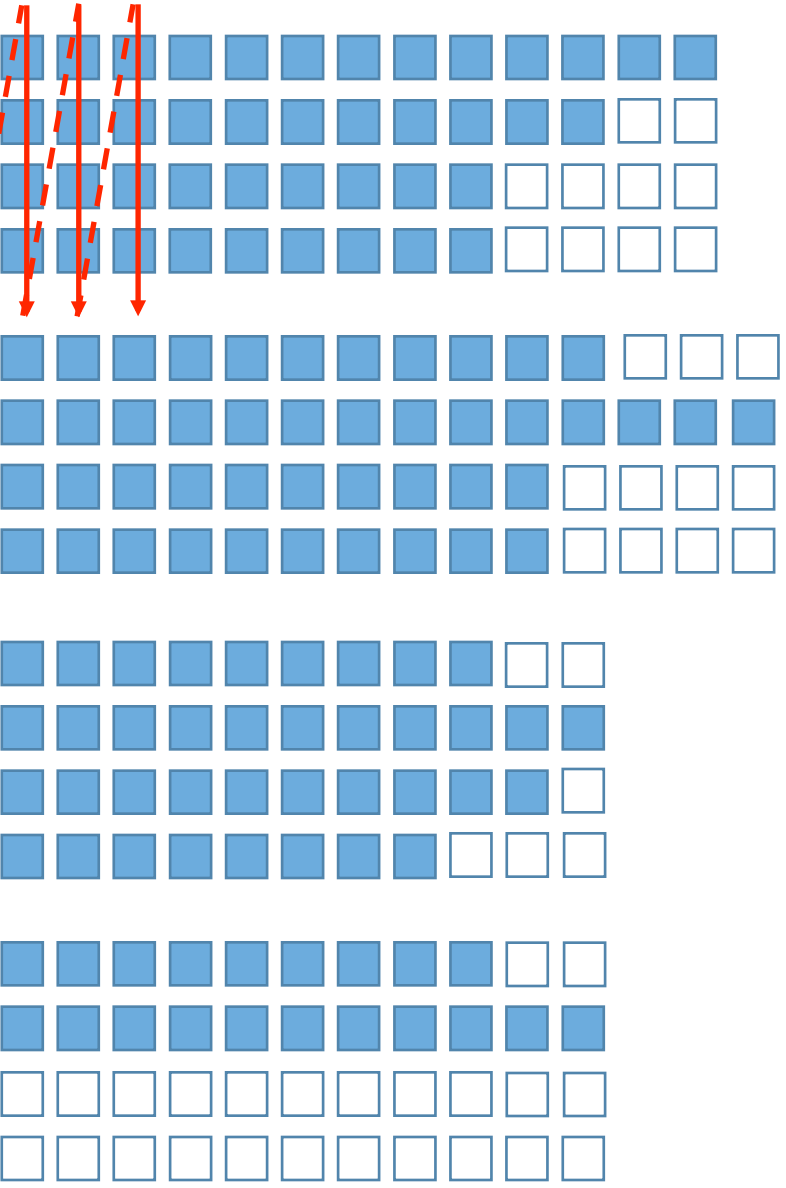
Can be done in parallel for SpMv (y=A*x), sequential for SYMGS (x=[r-A*x]*D^{-1})

Can be done in parallel (CPU threads, OpenMP4 teams, C blocks ,OpenMP4 threads, CUDA threads)

Can be done in parallel, using unroll (i.e., SIMD lanes

Can be done in parallel, but due to reduction it is mor suitable for sequential execution

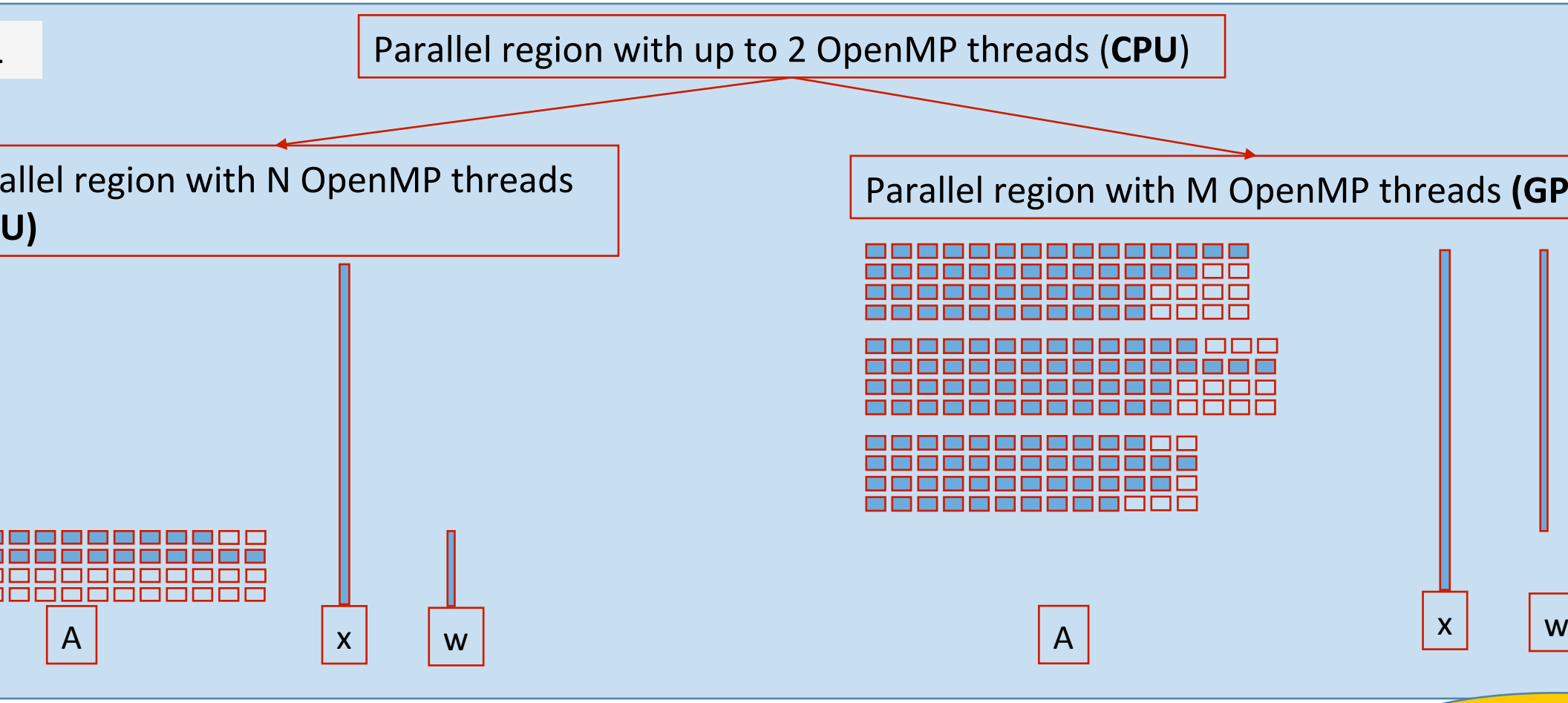
Levels of parallelism: one more thing …



Execute on device (GPU)

Execute on host (CPU)

lementation: OpenMP4, nested parallelism



For reor
opera

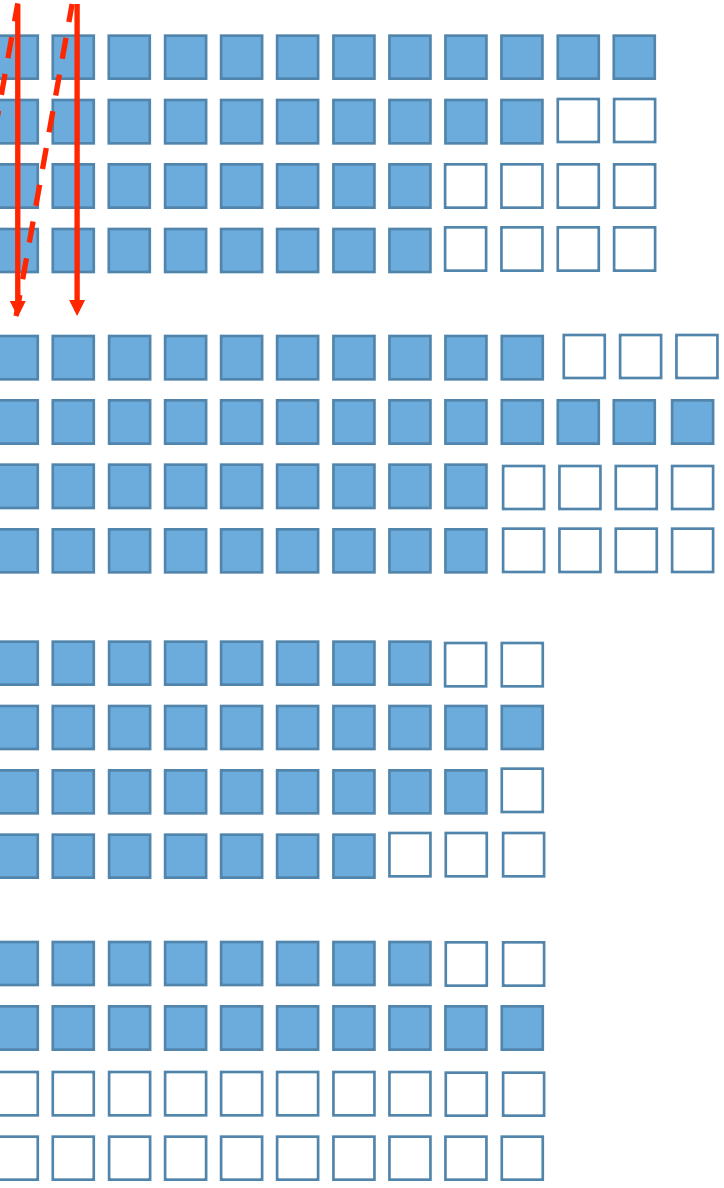
Executed on CPU (in parallel) : for (i=0; i < Nrows; i++) y[i] = w[reorderAtoB [i]];

x are in the CPU memory

A and x are mapped to the GPU memo

w is allocated in the pinned CPU memory, *y* in the CPU memory

Implementation: OpenMP4, *outer* parallel region

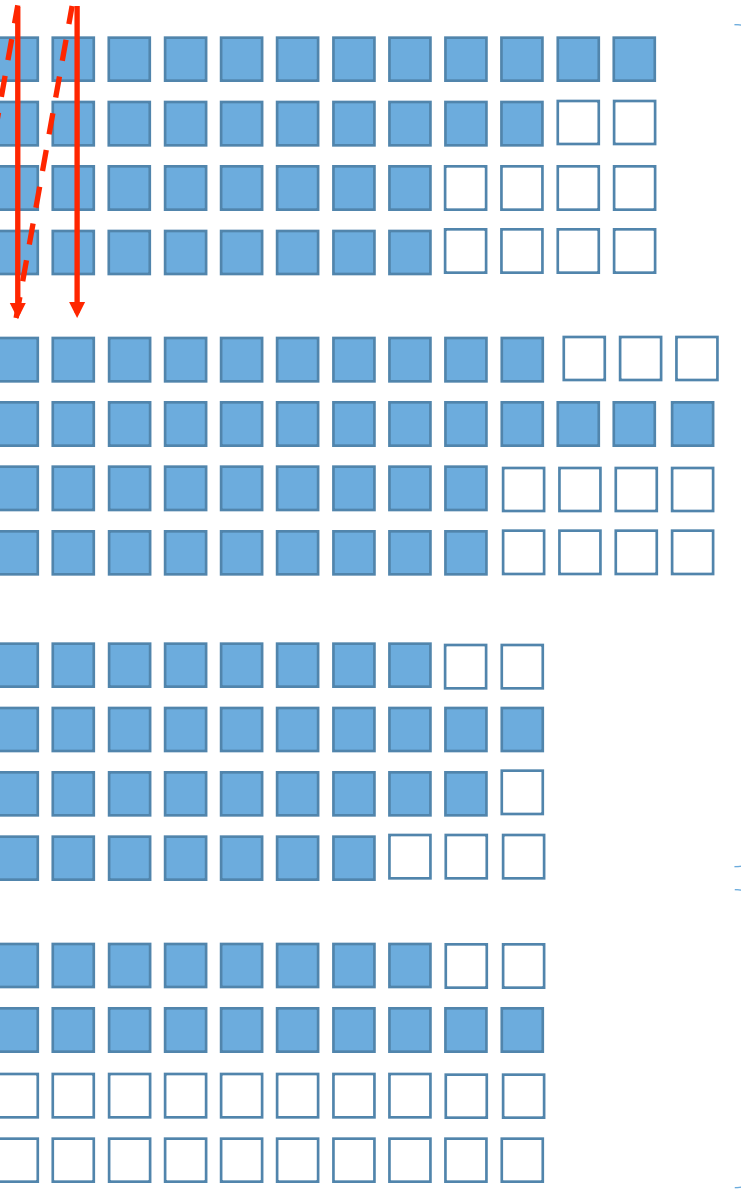


Execute on device (GPU)

Execute on host (CPU)

```
omp_set_nested(1);

#pragma omp parallel num_threads(2) if(U
 {
  int tid = omp_get_thread_num();
  int nteams,nthreads;

  int blk_start, blk_end;
  bool USE_GPU_local;
  double GPU_fraction = 0.75; //75% on GPU; range [0

  if (tid == 0){
     USE_GPU_local = USE_GPU;
     blk_start = 0;
     blk_end  = Nblocks* GPU_fraction;
  }

  else{
       USE_GPU_local = false;
       blk_start = Nblocks* GPU_fraction;
       blk_end  =  NBlocks ;
  }
....
```

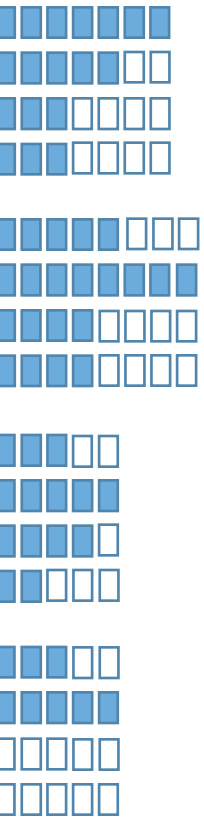
Implementation: OpenMP4, *outer* parallel region




```
#pragma omp parallel num_threads(2) if(USE_GP
  {
   int tid = omp_get_thread_num();
   int nteams,nthreads;

   int blk_start, blk_end;
   bool USE_GPU_local;
   double GPU_fraction = 0.75; //75% on GPU; range [0 1]

   if (tid == 0)  {USE_GPU_local = USE_GPU; ….}
   else           {USE_GPU_local = false; ….}

   if (USE_GPU_local) {
    nteams = 512;  nthreads = 8;
   }
   else{
    // if USE_GPU == false - use all CPU threads
   nteams = omp_get_max_threads() - ( 1 - omp_get_num_th
   nthreads = 1;
   }

   //inner parallel region

}
```

Implementation: OpenMP4, *inner* parallel region

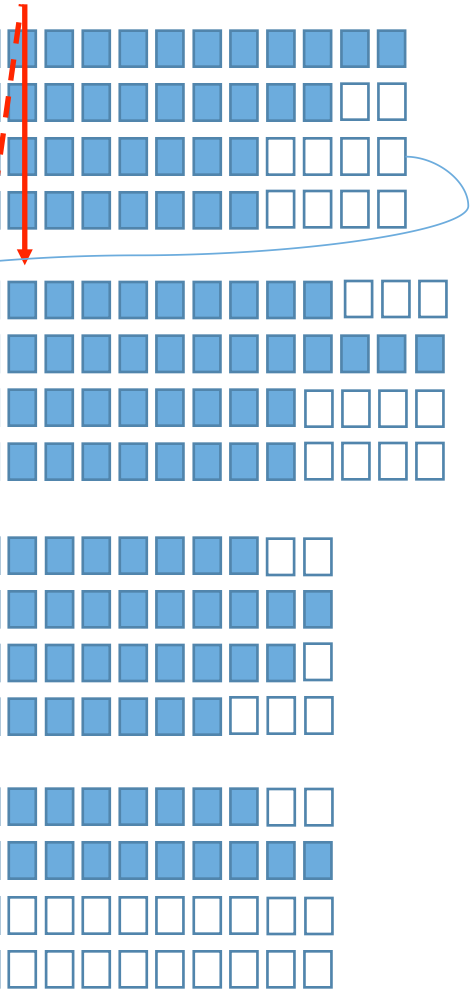
STEP 1



```
#pragma omp target map(alloc:PACKED_MAT[0:1], x[0:Ncolumns]) map(to:params[0:4])  if(USE_C
 {
   #pragma omp teams  distribute parallel for num_teams(nteams)  thread_limit(params[3]) \
                                                                  num_threads(params[3])
   for (int blk = params[0]; blk < params[1]; ++blk){

    const int BLK_DIM = PACKED_MAT->BLK_DIM;
    const int row_start = blk*BLK_DIM;

    const double * __restrict__  AA = PACKED_MAT->AA + PACKED_MAT->Block_offset[blk];
    const int    * __restrict__  AJ = PACKED_MAT->AJ + PACKED_MAT->Block_offset[blk];
    const int nnz = (PACKED_MAT->Block_offset[blk+1] - PACKED_MAT->Block_offset[blk])/BLK_DIM;

     #pragma omp simd
     for (int blk_row = 0; blk_row < BLK_DIM; ++blk_row){
      double sum = 0.0;
      for (int col = 0; col < nnz; ++col){
        const int index = AJ [blk_row + col * BLK_DIM];
        const double AAval = AA[blk_row + col * BLK_DIM];
        const double xval = x [index];
        sum += AAval*xval;
      }
      work[row_start + blk_row] = sum;      // USING ZERO COPY (results are written with stride 1 into the
                                            // pinned CPU memory)
   }
  }//end of for (int blk = 0; blk
 }
```

CPU and G
write into t
same arra

Implementation: OpenMP4, *inner* parallel region



STEP 1

```
#pragma omp target map(alloc:PACKED_MAT[0:1], x[0:Ncolumns]) map(to:params[0:4])   if(USE_GPU_local)
 {
  #pragma omp teams  distribute parallel for num_teams(nteams)  thread_limit(params[3]) num_threads(params
  for (int blk = params[0]; blk < params[1]; ++blk){

    const int BLK_DIM = PACKED_MAT->BLK_DIM;
    const int row_start = blk*BLK_DIM;

    const double * __restrict__  AA = PACKED_MAT->AA + PACKED_MAT->Block_offset[blk];
    const int    * __restrict__  AJ = PACKED_MAT->AJ + PACKED_MAT->Block_offset[blk];
    const int nnz = (PACKED_MAT->Block_offset[blk+1] - PACKED_MAT->Block_offset[blk])/BLK_DIM;

    #pragma omp simd
    for (int blk_row = 0; blk_row < BLK_DIM; ++blk_row){
      double sum = 0.0;
     for (int col = 0; col < nnz; ++col){
       const int index = AJ [blk_row + col * BLK_DIM];
       const double AAval = AA[blk_row + col * BLK_DIM];
       const double xval = x [index];
       sum += AAval*xval;
     }
     work[row_start + blk_row] = sum;     // USING ZERO COPY (results are written contiguously into the CPU
                                          // memory)
   }
  }//end of for (int blk = 0; blk
 }
```

omp_set_num_threads(max_threads);

STEP 2

```
//reorder data on CPU
#pragma omp parallel for
for (int i = 0; i < PACKED_MAT->Nrows; ++i)
  y[i] = work[PACKED_MAT->map_Rorg2Rnew[i]];
```

For reordered operator

sults: test setup
d timing

1 Firestone node.
two Power8 CPUs +
two K80 cards
(4 GPUs/node)

4 MPI ranks, 20 threads/rank,
5 CPU cores/rank,
1 GPU/rank,
each rank executes the same task.

```
t iter = 0; iter < MAX_ITER; ++iter){
I_Barrier(MPI_COMM_WORLD);
uble t_start = omp_get_wtime();
MAT_BLOCK_PACKED_SpMv(PACKED_MAT , v, rhs, USE_GPU);
I_Barrier(MPI_COMM_WORLD);
uble t_end = omp_get_wtime();
uble FLOPRATE = 2.0 * ((double) nnz*nranks) / (t_end-t_start) /  (1000.0*1000.0*1000.0);
ntf("SPMAT_BLOCK_PACKED_SpMv time = %e GF/s=%g \n",iter,time_end-time_start,FLOPRATE);
```

esults: SpMv – no row reordering

irun -np 4 ./bind.sh ./test_HPCG.exe
k=0: cpu_id = 0; GPU_ID = 0
k=1: cpu_id = 40; GPU_ID = 1
k=2: cpu_id = 80; GPU_ID = 2
k=3: cpu_id = 120; GPU_ID = 3
z=224140792 Nrow=8388608 Ncol=8388608

CPUs + GPUs; SELL-32

E_GPU = FALSE: SPMV time = 0.0179233
E_GPU = FALSE: SPMV time = 0.0178725
E_GPU = FALSE: SPMV time = 0.018004
E_GPU = TRUE: SPMV time = 0.018304
E_GPU = TRUE: SPMV time = 0.0183329
E_GPU = FALSE: SPMV time = 0.0183392
E_GPU = TRUE: SPMV time = 0.0185356
E_GPU = TRUE: SPMV time = 0.0186651

MAT_BLOCK_PACKED_SpMv time = 1.870673e-02 GF/s=95.85
MAT_BLOCK_PACKED_SpMv time = 1.870785e-02 GF/s=95.85
MAT_BLOCK_PACKED_SpMv time = 1.870927e-02 GF/s=95.84
MAT_BLOCK_PACKED_SpMv time = 1.870852e-02 GF/s=95.85

CPU CSR SpMv time = 8.483625e-02 GF/s=21.14

CPUs only; CSR

USE_GPU = FALSE: SPMV time = 0.0867225
USE_GPU = FALSE: SPMV time = 0.0867581
USE_GPU = FALSE: SPMV time = 0.0866689
USE_GPU = FALSE: SPMV time = 0.0870566

CPUs only; SELL-32

SPMAT_BLOCK_PACKED_SpMv time = 8.712003e-02 GF/s=20.58
SPMAT_BLOCK_PACKED_SpMv time = 8.711867e-02 GF/s=20.58
SPMAT_BLOCK_PACKED_SpMv time = 8.712031e-02 GF/s=20.58
SPMAT_BLOCK_PACKED_SpMv time = 8.711782e-02 GF/s=20.58

rse: SpMv time: 25.68 [milliseconds] –> 17.46GF/s (on single GPU; 69.84GF/s on 4 GPUs); all data in the GPU
ry

sults: SpMv – with row ordering

40,792 Nrow=8,388,608

: USE_GPU = TRUE: SPMV time = 0.0205524
: USE_GPU = TRUE: SPMV time = 0.0205923
: USE_GPU = TRUE: SPMV time = 0.0206904
: USE_GPU = TRUE: SPMV time = 0.0209094
: USE_GPU = FALSE: SPMV time = 0.0217777
: USE_GPU = FALSE: SPMV time = 0.021805
: USE_GPU = FALSE: SPMV time = 0.021969
: USE_GPU = FALSE: SPMV time = 0.0232831
: reordering data time: 0.00398043
: reordering data time: 0.00468169
: reordering data time: 0.00349937
: reordering data time: 0.00530664
SPMAT_BLOCK_PACKED_SpMv time = 2.717577e-02 GF/s=65.98
SPMAT_BLOCK_PACKED_SpMv time = 2.716966e-02 GF/s=66.0
SPMAT_BLOCK_PACKED_SpMv time = 2.716869e-02 GF/s=66.0
SPMAT_BLOCK_PACKED_SpMv time = 2.717585e-02 GF/s=65.98

CPUs + GPUs; SELL-32

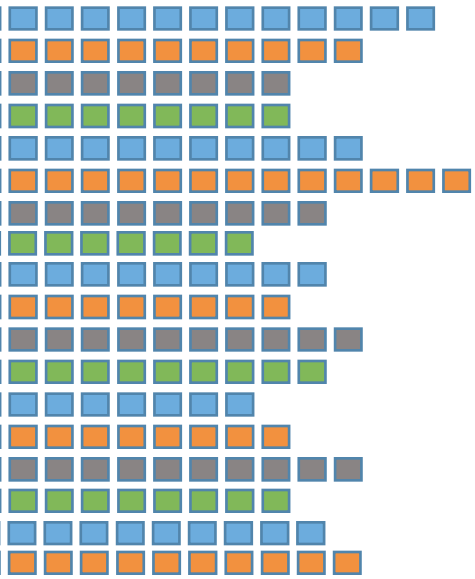
MULTICOLOR: USE_GPU = FALSE: SPMV time = 0.0937613
MULTICOLOR: USE_GPU = FALSE: SPMV time = 0.0940417
MULTICOLOR: USE_GPU = FALSE: SPMV time = 0.0943051
MULTICOLOR: USE_GPU = FALSE: SPMV time = 0.0944391
MULTICOLOR: reordering data time: 0.0160486
MULTICOLOR: reordering data time: 0.0166181
MULTICOLOR: reordering data time: 0.0167874
MULTICOLOR: reordering data time: 0.0166573
MULTICOLOR SPMAT_BLOCK_PACKED_SpMv time = 1.112842e-01
MULTICOLOR SPMAT_BLOCK_PACKED_SpMv time = 1.112823e-01
MULTICOLOR SPMAT_BLOCK_PACKED_SpMv time = 1.112831e-01
MULTICOLOR SPMAT_BLOCK_PACKED_SpMv time = 1.112825e-01

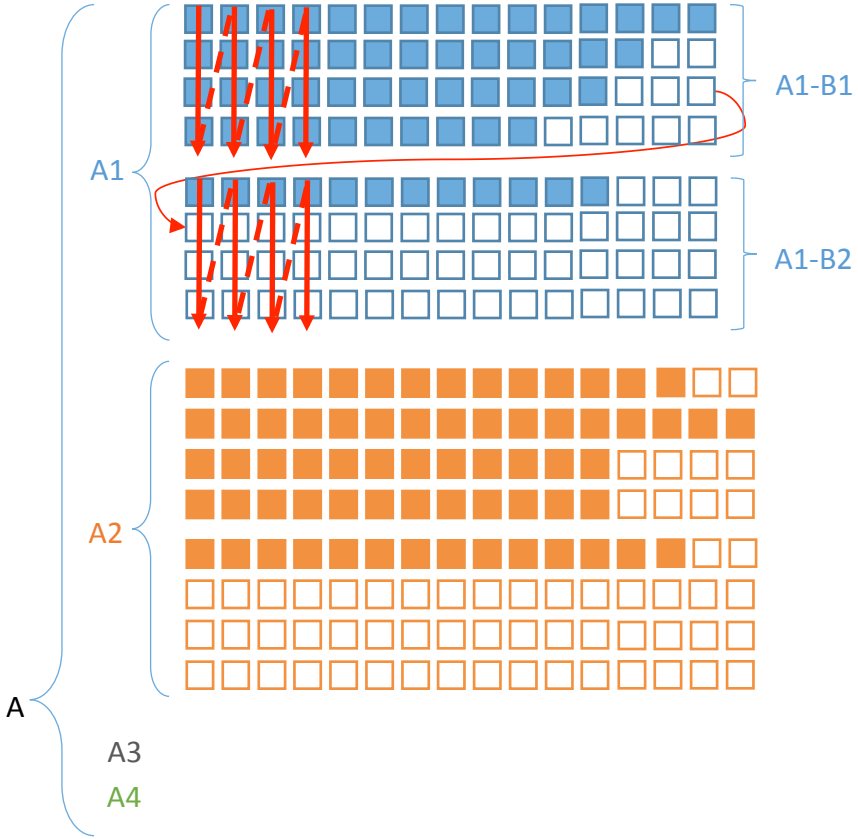
C
SE

Results: SpMv , nvprof data (single GPU)

	no row reordering	with row reordering
equested Global Load Throughput	192.60GB/s	170.90GB/s
evice Memory Read Throughput	193.77GB/s	193.46GB/s
lobal Load Throughput	204.57GB/s	246.68GB/s
lobal Memory Load Efficiency	94.15%	69.28%
evice Memory Utilization	High (9)	High (9)
ystem Memory Write Throughput	2.86GB/s (over PCIe 3.0)	2.54GB/s (over PCIe 3.
CC Throughput	43.362GB/s	41.066GB/s
LOP Efficiency(Peak Double)	1.41% (2.91 TF / 2 *1.41% = 21GF/s)	1.25% (18.6GF/s)
Executed IPC	0.57	0.50

Parallel SYMGS [x=(r-A*x)*D^{-1}]: considerations

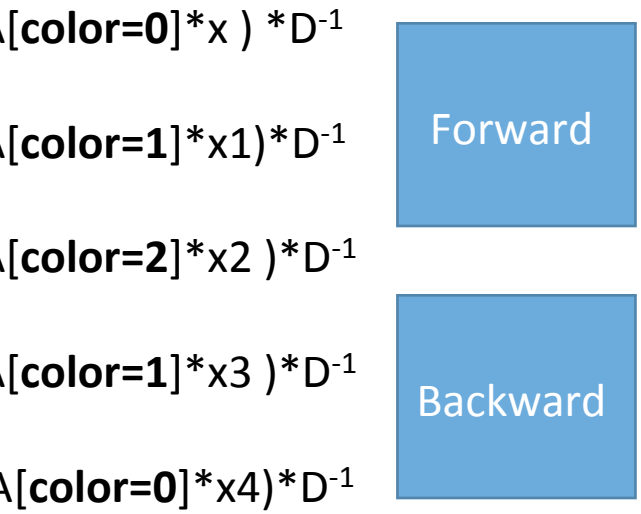


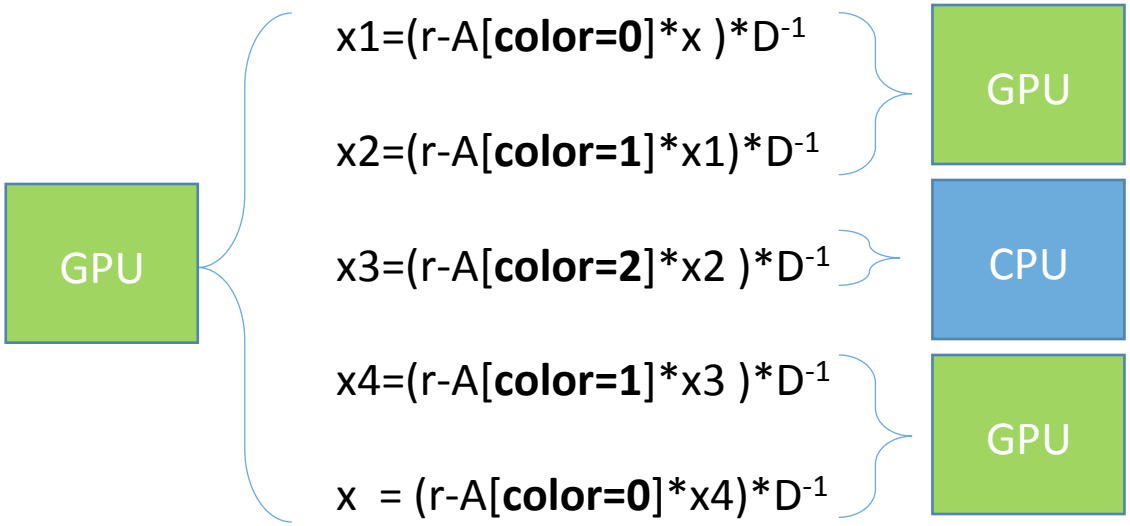


- Need to process one color at a
- Each color can be processed in
- Work associated with each colo
  divided between the CPU and G
  however, this will require to up
  the vector "*x*" on both devices
  to additional communication.
- Can perform SYMGS on GPU or
  map the end result to the CPU.

- If GPU memory is not sufficient
  all the data (coefficients for all
  can process one or more colors
  CPU and the rest on the GPU.

Parallel SYMGS [x=(r-A*x)*D^{-1}]: implementation





Unlike in SpMv, to avoid reordering of "*x*" on the CPUs with subsequent mapping to the GPU memory, we process each color on one device at a time.

Parallel SYMGS: setup

```
1; // 1, or more
 = 0; trip  < 2*ntrips; ++trip){

_start, color_increment;
 == 0){
tart = 0;
ncrement = 1;


art = PACKED_MAT->Ncolors - 2;
crement = -1;

_counter = 0;
= color_start;

lor_counter < (PACKED_MAT->Ncolors - (trip%2))  ){
ounter++;

;
ACKED_MAT->NFullBlocks_per_color[color];
_color = blk_start + PACKED_MAT->color_block_offset[color];
color   = blk_end   + PACKED_MAT->color_block_offset[color];

r < Ngpu_colors) && (USE_GPU)) {
PU_local = true;  nteams = 512;  nthreads = 8;


PU_local = false;   nteams = max_threads;   nthreads = 1;
```

1

Trip forward

Trip backward (+ keep symmetry)

Process N colors on GPU, the rest on the CPU; set number of threads and teams accordingly

$x1=(r-A[\textbf{color=0}]*x\ )\ *D^{-1}$

$x2=(r-A[\textbf{color=1}]*x1)*D^{-1}$

$x3=(r-A[\textbf{color=2}]*x2\ )*D^{-1}$

$x4=(r-A[\textbf{color=1}]*x3\ )*D^{-1}$

$x\ =(r-A[\textbf{color=0}]*x4)*D^{-1}$

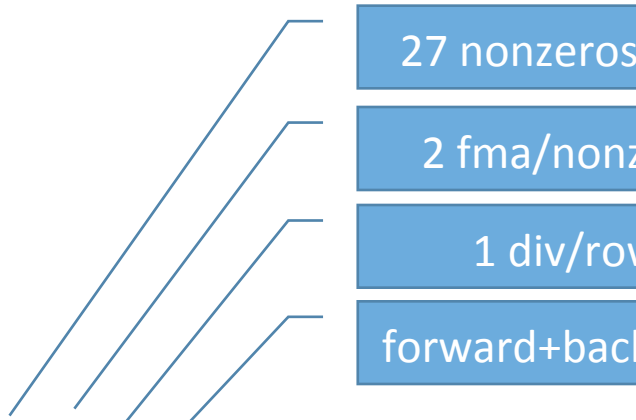
```
//update data on CPU if needed
if ((trip%2 == 1) && ((color == Ngpu_colors-1)) ) {
      #pragma omp target update to(x[0:localNumberOfRows]) if (USE
 }

//update data on GPU if needed
if ((trip%2 == 0) && (color == Ngpu_colors) ){
      #pragma omp target update from(x[0:localNumberOfRows]) if
(USE_GPU)
}
```

Parallel SYMGS: kernel

```
#pragma omp target map(alloc:PACKED_MAT[0:1],x[0:1],params[0:4],rv[0:1]) if(USE_GPU_local)

  #pragma omp teams  distribute parallel for num_teams(params[2]) thread_limit(params[3]) num_threads(params[3])
  for (int blk = params[0]; blk < params[1]; ++blk){

    const int BLK_DIM = PACKED_MAT->BLK_DIM;
    const int row_start = blk*BLK_DIM;
    const double * __restrict__  AA = PACKED_MAT->AA + PACKED_MAT->Block_offset[blk];
    const int    * __restrict__  AJ = PACKED_MAT->AJ + PACKED_MAT->Block_offset[blk];
    const int nnz = (PACKED_MAT->Block_offset[blk+1] - PACKED_MAT->Block_offset[blk])/BLK_DIM;

   #pragma omp simd
   for (int blk_row = 0; blk_row < BLK_DIM; ++blk_row){
      double sum = rv[ PACKED_MAT->map_Rnew2Rorg[row_start+blk_row] ];
      double currentDiagonal = 1.0;
      int index_not_set = 1;
     for (int col = 0; col < nnz; ++col){
        const int index = AJ [blk_row + col * BLK_DIM];
        const double AAval = AA[blk_row + col * BLK_DIM];
        const double xval = x [index];
        if ((index == PACKED_MAT->map_Rnew2Rorg[row_start+blk_row]) && (index_not_set))  {
           currentDiagonal = AAval; index_not_set = 0;
        }
        else
          sum -= AAval*xval;
     }
     int  index = PACKED_MAT->map_Rnew2Rorg[row_start + blk_row];
     if (index != -1) x[index] = sum/currentDiagonal;
   }
  }//end of for (int blk = 0; blk
// end of #pragma omp target
```

MGS: Results

ngle GPU data, monitoring
PCG-OpenMP4's parallel SYMGS



.localNumberOfRows= 16777216, SYMGS time = 0.135477 → (16777216*(27*2+1)*2)/0.135/1000^3 = 13.67
.localNumberOfRows= 2097152, SYMGS time = 0.023057 → (2097152*(27*2+1)*2)/0.023/1000^3 = 10.03

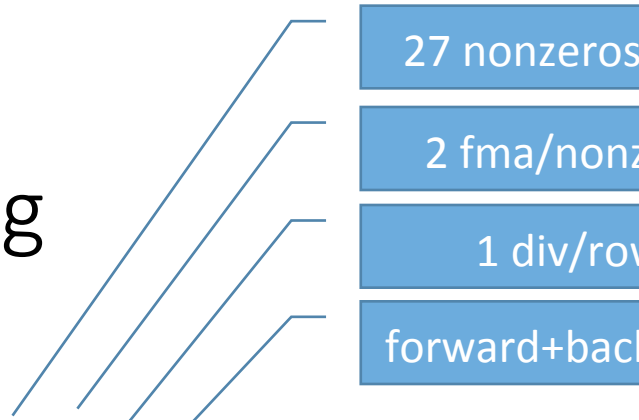
ence: multicolored SpMv kernel executed on GPUs + CPUs achieved about 66.04 GF/s per node, that is about
mpi rank. If CPU was not involved in SpMv, the expected flop rate would be about 10-20% lower, i.e. about 1

e that here we compute on GPUs only, and instead of executing one kernel with high work load as in the SpM
15 kernels each having 1/8 of the SpMv workload.

ect measurements : since we process 7 colors two times and one color 1 time we need to correct the obtained flop rate by a factor of (7*2
s will lead to 12.81 GF/s and 9.4 GF/s respectively to the problem size, well , considering that the *division* operation takes more cycles than *f*
her correction … but this is not the point of this exercise …

MGS: Results

MGS on CPU, monitoring
PCG-OpenMP4's parallel SYMGS running
5 cores of the Power 8



.localNumberOfRows = 16777216, SYMGS time = 0.38793 → (16777216*(27*2+1)*2)/0.38793/1000^3 = 4.7
.localNumberOfRows = 2097152, SYMGS time = 0.06146 → (2097152 * (27*2+1)*2)/0.06146/1000^3 = 3.7

ence: multicolored SpMv kernel executed on CPUs achieved about 16.1 GF/s per node (including reordering),
02 GF/s per 5 cores.

rrect measurements : since we process 7 colors two times and one color 1 time we need to correct the obtained flop rate
(7*2+1)/(8*2) = 0.9375, this will lead to 4.46 GF/s and 3.51 GF/s respectively to the problem size, well , considering that th
takes more cycles than *fma* we can make another correction … but this is not the point of this exercise …

Summary

SpMv and parallel SYMGS kernels have been implemented using OpenMP4

Switching execution between the CPU and GPU is achieved by dynamically by setting the value of the variable to the OpenMP4's "*if*" clause (i.e. *#pragma omp target map… if(USE_GPU==true)*)

Performance of both kernels scales with respect to the memory BW of Power8 CPU and NVIDIA's K80 GPU

SpMv – OpenMP4.0 kernel executed on Firestone node achieves over 95 GFLOP/s compared to ~21 GF/s achieved in executing the same kernel on the CPUs only

Parallel multicolored SYMGS – OpenMP4.0 runs at about 55 GF/s on two K80 cards (4 GPUs) and ~19 GF/s on two Power8 CPUs

It is expected that performance of the two kernels will be improved with evolution of the LLVM compiler which will support the OpenMP 4.5 standard and enhanced optimizations for simulations on the CPUs and the GPUs

Mapping to GPU (deep copy)

```
PMAT_BLOCK_PACKED *&PACKED_MAT_ptr = PACKED_MAT;
ouble * __restrict__  &AA = PACKED_MAT->AA;
nt    * __restrict__  &AJ = PACKED_MAT->AJ;
nt * __restrict__     &Block_offset = PACKED_MAT->Block_offset;
nt size = PACKED_MAT->sizeAA;
nt nblk =  PACKED_MAT->NFullBlocks + PACKED_MAT->NPartialBlocks + 1;
pragma omp target enter data map(to:PACKED_MAT_ptr[0:1],AA[0:size], AJ[0:size],Block_offset[0:nblk]) if(USE_GPU)
pragma omp target map(alloc:PACKED_MAT_ptr[0:1],AA[0:size], AJ[0:size],Block_offset[0:nblk])  if(USE_GPU)

PACKED_MAT_ptr->AA = AA;
PACKED_MAT_ptr->AJ = AJ;
PACKED_MAT_ptr->Block_offset = Block_offset;
```

Results: SpMv – no row reordering

-np 4 ./bind.sh ./test_HPCG.exe
cpu_id = 0; GPU_ID = 0
cpu_id = 40; GPU_ID = 1
cpu_id = 80; GPU_ID = 2
cpu_id = 120; GPU_ID = 3

CPUs + GPUs;
SELL-32

278,488 Nrow=26,634,240
U = TRUE: SPMV time = 0.0570154
U = TRUE: SPMV time = 0.0570554
U = TRUE: SPMV time = 0.0572517
U = TRUE: SPMV time = 0.0572598
U = FALSE: SPMV time = 0.0585307
U = FALSE: SPMV time = 0.0588508
U = FALSE: SPMV time = 0.0590637
U = FALSE: SPMV time = 0.0613168
BLOCK_PACKED_SpMv time = 6.135144e-02 GF/s=93.14
BLOCK_PACKED_SpMv time = 6.134996e-02 GF/s=93.14
BLOCK_PACKED_SpMv time = 6.135244e-02 GF/s=93.14
BLOCK_PACKED_SpMv time = 6.135162e-02 GF/s=93.14

SpMv time: 81.72 [milliseconds]: 17.48 GF/s on 1 GPU (i.e. 69.92GF/s on 4 GPUs)

Performance portability

"Broadly, this [performance portability] means that an application should have qualitatively 'good' performance on different architectures and that a relatively small amount of effort should be required to tune application performance from one architecture to another."

Source: http://www.nersc.gov/research-and-development/application-readiness-across-doe-labs/

One way to achieve performance portability is by using libraries, however, libraries are also applications that someone needs to code, tune, maintain, and port to different architectures ….