Scientific programming with graphics processing units

Session 3: working with multiple GPUs

Eugene DePrince
Florida State University

Working with multiple GPUs

many HPC resources boast more than one GPU per node
    (hokiespeed and blueridge each have two per node!)

using both resources requires careful thought
    at least three different memory spaces (CPU, GPU1, GPU2, …)
    different CPU threads can manage different GPUs

the next few slides discuss concepts related to the code found in

S212/gpu/interleaved_dgemm/gpuhelper.cu

The GPUHelper class defined there can manage multiple GPU resources for performing DGEMM, with no real input from the user

Example 1: DGEMM with multiple GPUs

DGEMM is straightforward with one GPU, provided you can fit the three matrices in global memory:

```
// pointers to device memory
double * gpuA, * gpuB, * gpuC;

// allocate device memory
cudaMalloc((void**)gpuA,n*k*sizeof(double));
cudaMalloc((void**)gpuB,m*k*sizeof(double));
cudaMalloc((void**)gpuC,n*m*sizeof(double));

// copy memory to device
cudaMemcpy(gpuA,A,m*k*sizeof(double),cudaMemcpyHostToDevice);
cudaMemcpy(gpuB,B,n*k*sizeof(double),cudaMemcpyHostToDevice);

// dgemm! warning - fortran ordering
cublasDgemm(transa,transb,m,n,k,alpha,gpuA,lda,gpuB,ldb,beta,gpuC,ldc);

// copy result back to host
cudaMemcpy(C,gpuC,m*n*sizeof(double),cudaMemcpyDeviceToHost);
```

if the matrices are too big, you must tile them. This is exactly what we'll do to use multiple GPUs - we tile each matrix and let each GPU handle some subset of the tiles.

Example 1: DGEMM with multiple GPUs

DGEMM is straightforward with one GPU, provided you can fit the three matrices in global memory:

```
// pointers to device memory
double * gpuA, * gpuB, * gpuC;

// allocate device memory
cudaMalloc((void**)gpuA,n*k*sizeof(double));
cudaMalloc((void**)gpuB,m*k*sizeof(double));
cudaMalloc((void**)gpuC,n*m*sizeof(double));

// copy memory to device
cudaMemcpy(gpuA,A,m*k*sizeof(double),cudaMemcpyHostToDevice);
cudaMemcpy(gpuB,B,n*k*sizeof(double),cudaMemcpyHostToDevice);

// dgemm! warning - fortran ordering
cublasDgemm(transa,transb,m,n,k,alpha,gpuA,lda,gpuB,ldb,beta,gpuC,ldc);

// copy result back to host
cudaMemcpy(C,gpuC,m*n*sizeof(double),cudaMemcpyDeviceToHost);
```

if the matrices are too big, you must tile them. This is exactly what we'll do to use multiple GPUs - we tile each matrix and let each GPU handle some subset of the tiles.

Example 1: DGEMM with multiple GPUs

with two GPUs, we need some way to manage different memory spaces

```
// pointers to device memory
double ** gpu_buffer;

// pointers to host memory
double ** cpu_buffer;

// how many GPUs do we have?
int num_gpus;
cudaGetDeviceCount(&num_gpus);
```

```
gpu_buffer = (double**)malloc(num_gpus*sizeof(double*));
cpu_buffer = (double**)malloc(num_gpus*sizeof(double*));

for (int i = 0; i < num_gpus; i++) {
    cudaSetDevice(i);
    cudaMalloc((void**)gpu_buffer[i],gpu_memory*sizeof(double));
    cudaMallocHost((void**)cpu_buffer[i],gpu_memory*sizeof(double));
}
```

to simplify things, we can just allocate memory for a single buffer that is the size of the global memory. We do this for each GPU and use cudaSetDevice to control which GPU we're considering

Example 1: DGEMM with multiple GPUs

with two GPUs, we need some way to manage different memory spaces

```
// pointers to device memory
double ** gpu_buffer;

// pointers to host memory
double ** cpu_buffer;

// how many GPUs do we have?
int num_gpus;
cudaGetDeviceCount(&num_gpus);
```

```
gpu_buffer = (double**)malloc(num_gpus*sizeof(double*));
cpu_buffer = (double**)malloc(num_gpus*sizeof(double*));

for (int i = 0; i < num_gpus; i++) {
    cudaSetDevice(i);
    cudaMalloc((void**)gpu_buffer[i],gpu_memory*sizeof(double));
    cudaMallocHost((void**)cpu_buffer[i],gpu_memory*sizeof(double));
}
```

We're also going to use page-locked memory on the host. This will allow us to overlap communication and computation when we work on each tile

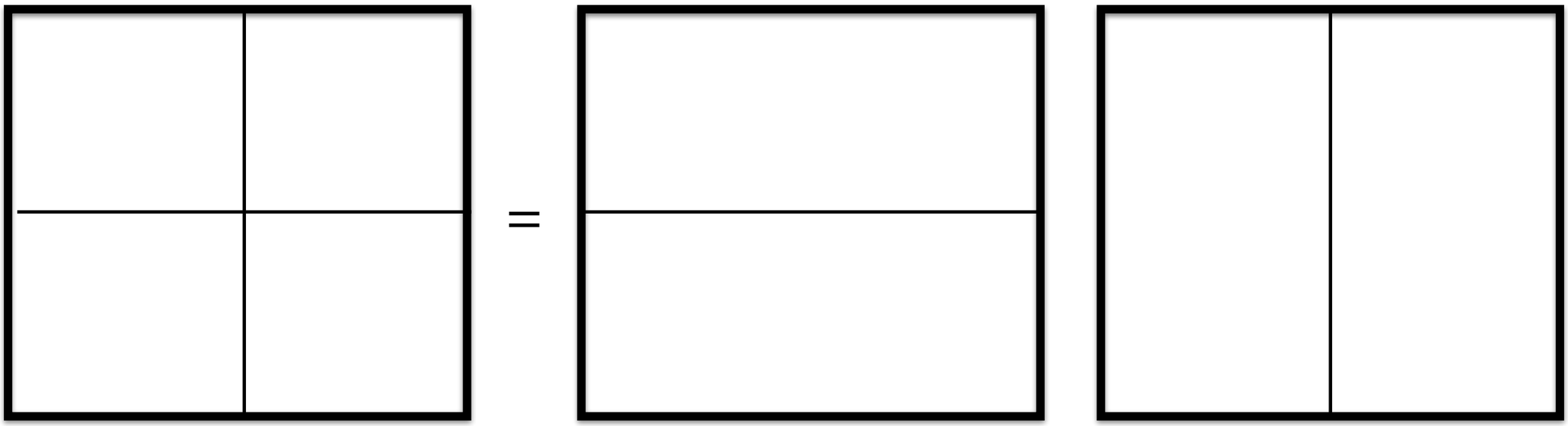
Example 1: DGEMM with multiple GPUs

first, determine an appropriate tiling.

say we have two GPUs - split the m/n dimensions

$$C = A \cdot B$$

$$m \times n \qquad m \times k \qquad k \times n$$



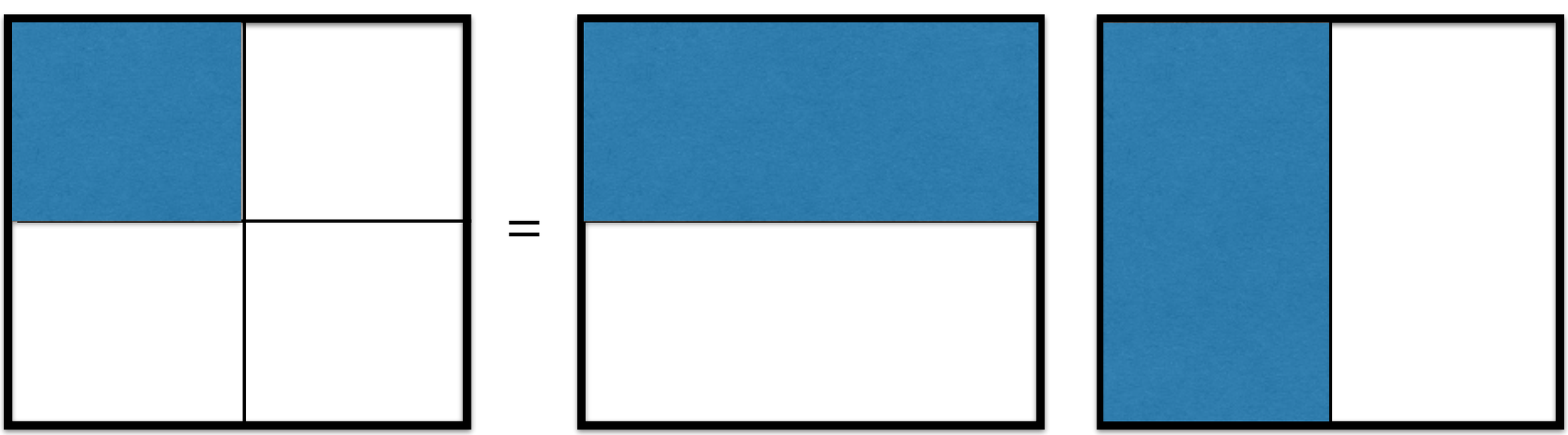
Example 1: DGEMM with multiple GPUs

first, determine an appropriate tiling.

say we have two GPUs - split the m/n dimensions

$$C = A \cdot B$$

$$m \times n \qquad m \times k \qquad k \times n$$



GPU 1, GPU 2 each handle half of the blocks. Easy peasy!

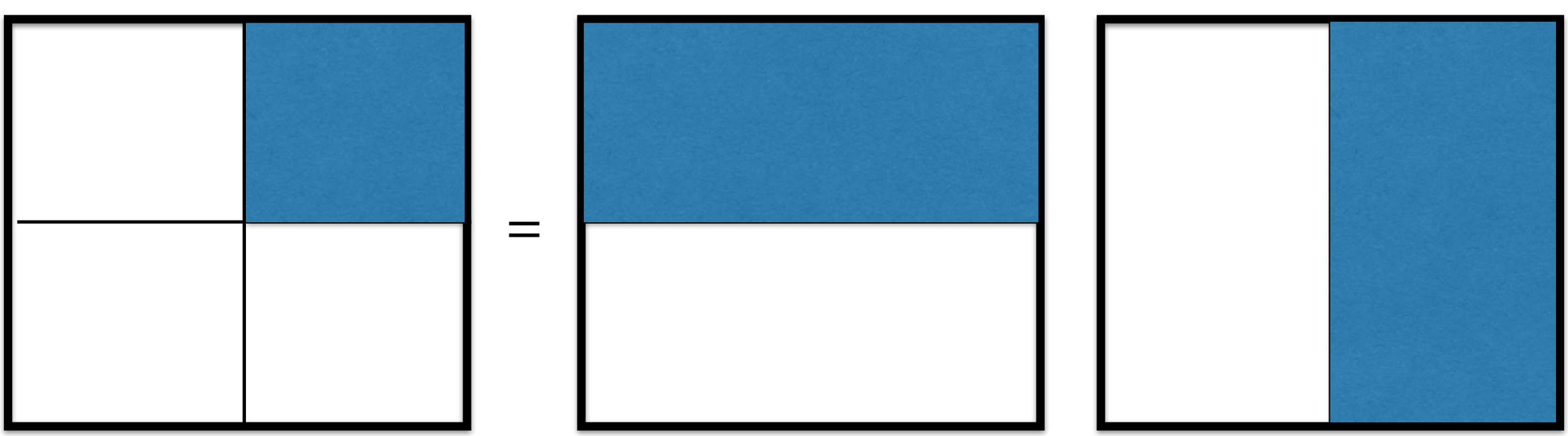
Example 1: DGEMM with multiple GPUs

first, determine an appropriate tiling.

say we have two GPUs - split the m/n dimensions

$$C = A \cdot B$$

$$m \times n \qquad m \times k \qquad k \times n$$



GPU 1, GPU 2 each handle half of the blocks. Easy peasy!

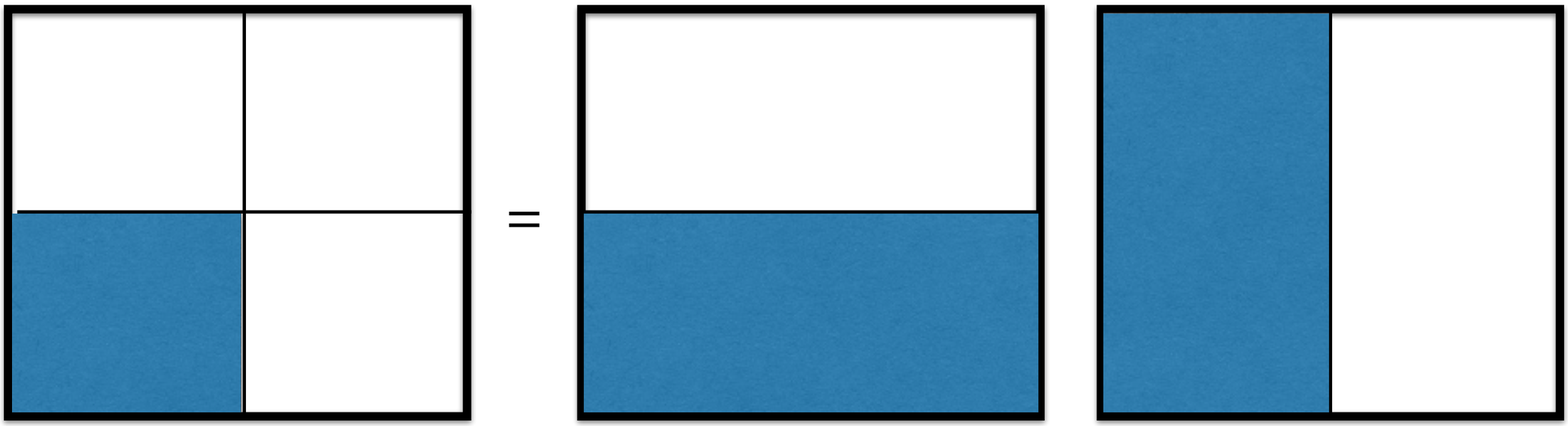
Example 1: DGEMM with multiple GPUs

first, determine an appropriate tiling.

say we have two GPUs - split the m/n dimensions

$$C = A \cdot B$$

$$m \times n \qquad m \times k \qquad k \times n$$



GPU 1, GPU 2 each handle half of the blocks. Easy peasy!

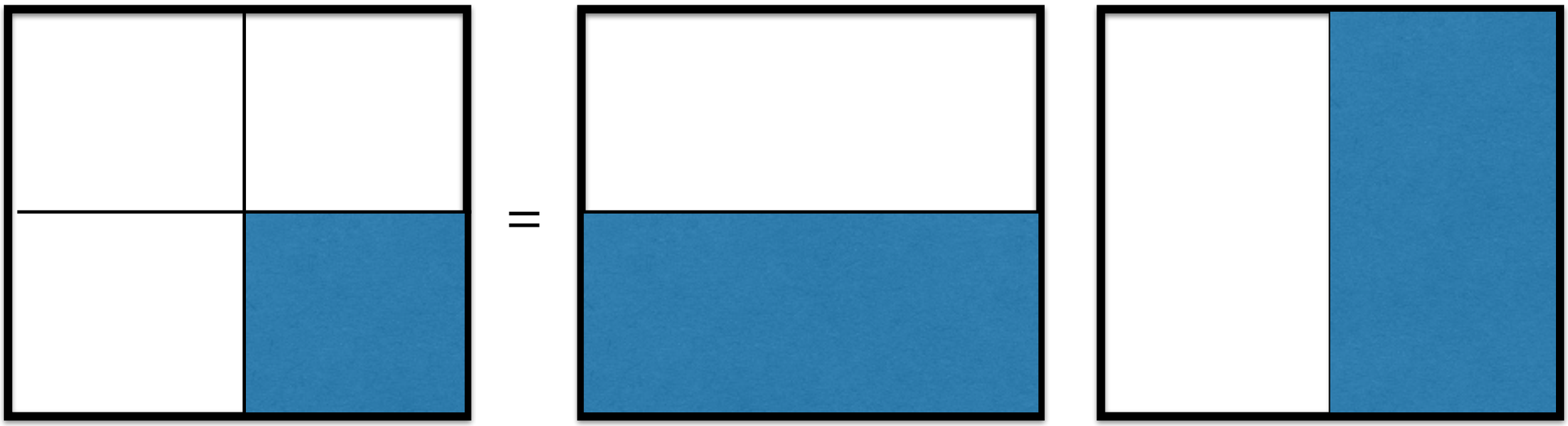
Example 1: DGEMM with multiple GPUs

first, determine an appropriate tiling.

say we have two GPUs - split the m/n dimensions

$$C = A \cdot B$$

$$m \times n \qquad m \times k \qquad k \times n$$



GPU 1, GPU 2 each handle half of the blocks. Easy peasy!

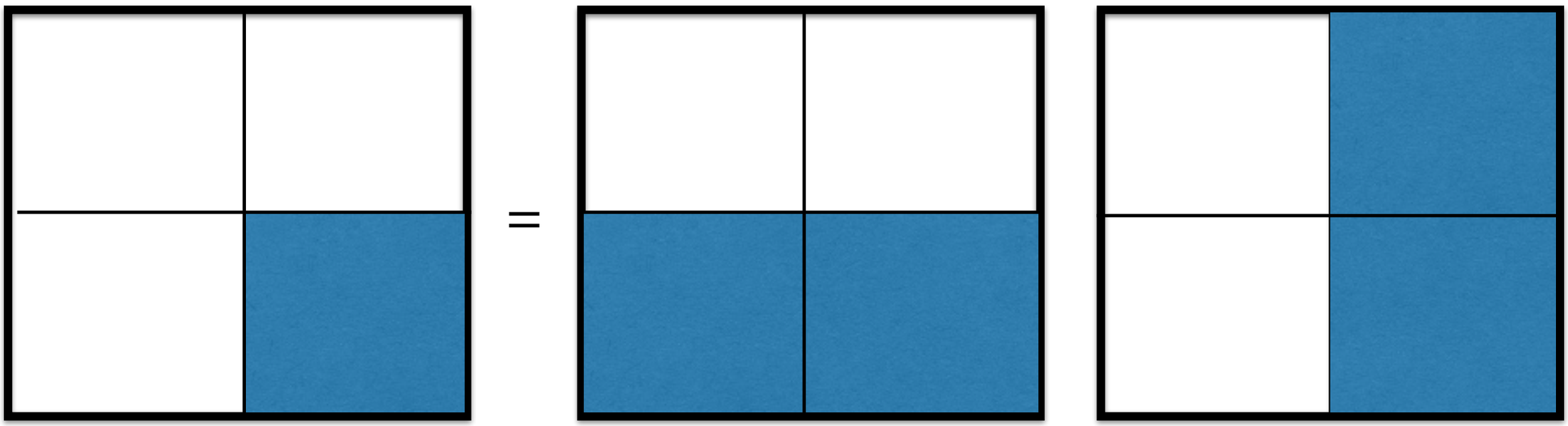
Example 1: DGEMM with multiple GPUs

first, determine an appropriate tiling.

say we have two GPUs - split the m/n dimensions

$$C = A \cdot B$$

$$m \times n \qquad m \times k \qquad k \times n$$



GPU 1, GPU 2 each handle half of the blocks. Easy peasy!

We can also tile the summation index. We'll revisit that later

Example 1: DGEMM with multiple GPUs

OK, we know our tile sizes.

now, loop over each tile. use an omp pragma to parallelize over GPUs

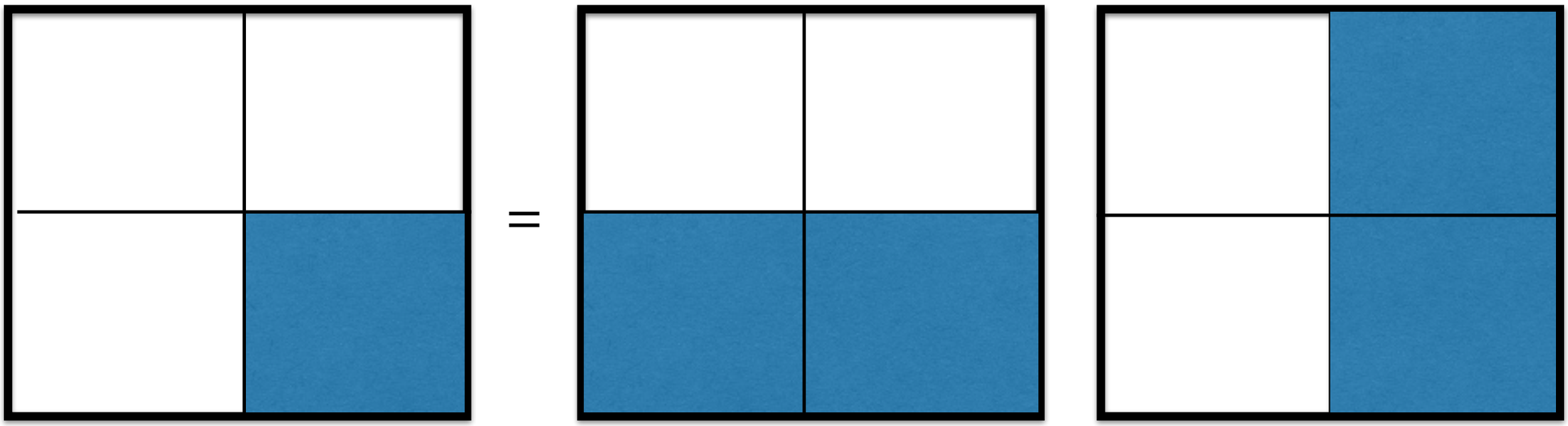
...

```
#pragma omp parallel for schedule (static) num_threads(num_gpus)
for (int mn=0; mn<ntilesM*ntilesN; mn++){
    int thread = omp_get_thread_num();

    cudaSetDevice(thread);

    // pointers to gpu memory
    double*gpuA = gpu_buffer[thread];
    double*gpuB = gpu_buffer[thread]+tilesizeM*tilesizeK;
    double*gpuC = gpu_buffer[thread]+tilesizeM*tilesizeK+tilesizeN*tilesizeK;

    // which tile am i handling?
    long int tn = mn%ntilesN;
    long int tm = (mn-tn)/ntilesN;

    cudaMemset((void*)gpuC,'\0',tilesizesM[tm]*tilesizesN[tn]*sizeof(double));
```

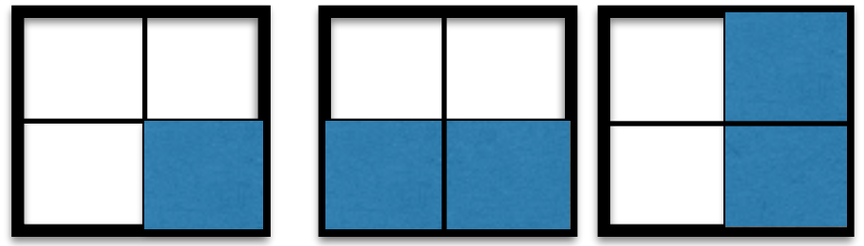
Example 1: DGEMM with multiple GPUs

now, the complicated part.

dgemm wants contiguous memory, so we need to pack our tiles accordingly



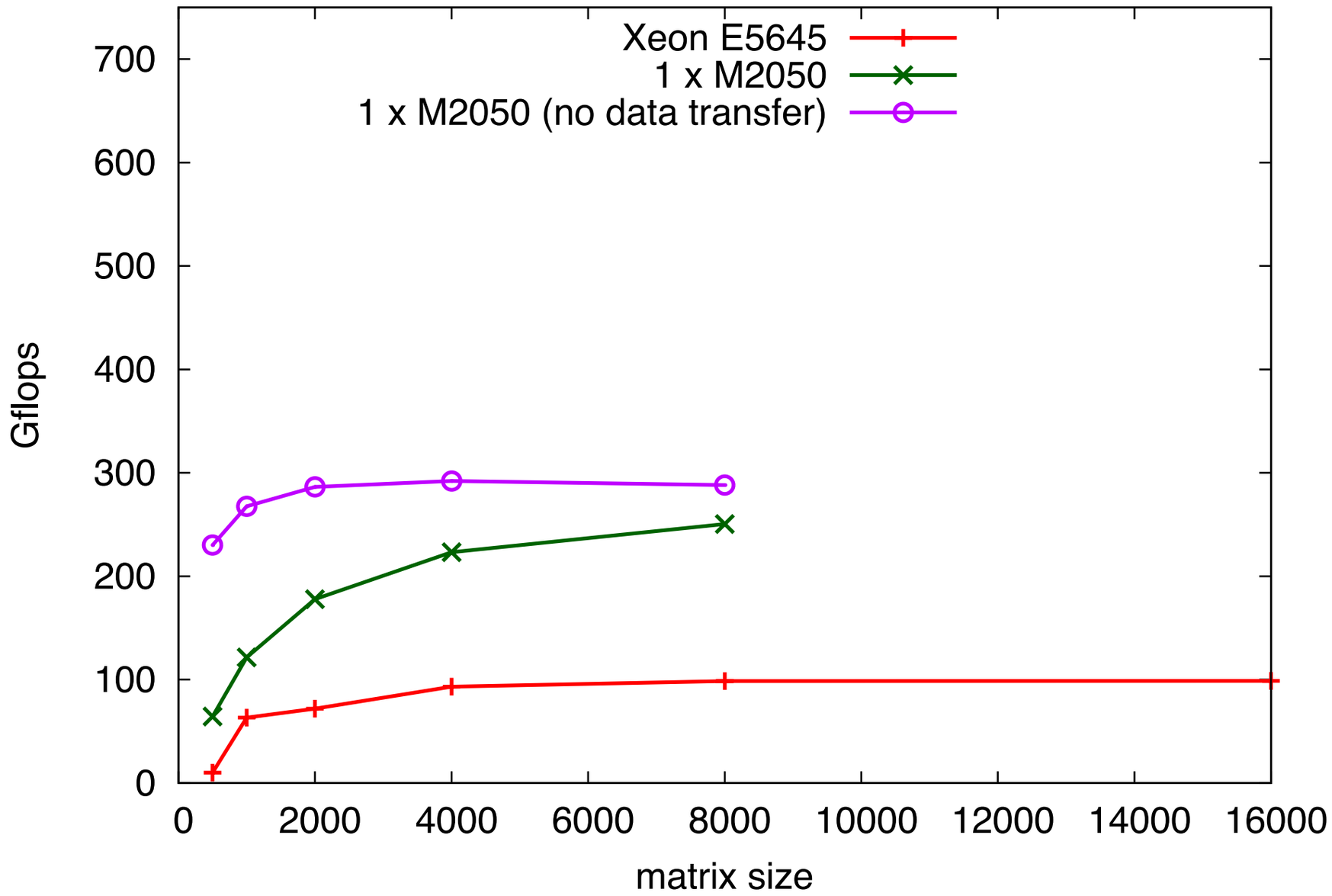
Example 1: DGEMM with multiple GPUs



```
...
    for (long int tk=0; tk<ntilesK; tk++){
        for (long int i=0; i<tilesizesK[tk]; i++){
            // pack up a tile of A
            // A(tileM,tileK)-> page-locked CPU memory

            cudaMemcpy(gpuA,cpu_buffer[thread],
                tilesizesM[tm]*tilesizesK[tk]*sizeof(double),
                cudaMemcpyHostToDevice);
            // pack up a tile of B
            // B(tileK,tileN)-> page-locked CPU memory

            cudaMemcpy(gpuB,cpu_buffer[thread],
                tilesizesN[tn]*tilesizesK[tk]*sizeof(double),
                cudaMemcpyHostToDevice);

            cublasDgemm(transa,transb,
               tilesizesM[tm],tilesizesN[tn],tilesizesK[tk],alpha
               gpuA,tilesizesM[tm],gpuB,tilesizesN[tn],1.0,gpuC,tilesizesM[tm]);

        }
    }
    cudaMemcpy(cpu_buffer[thread],gpuC,
    tilesizesN[tn]*tilesizesM[tm]*sizeof(double),cudaMemcpyDeviceToHost);
    // add result to appropriate tile in CPU memory, C(tileM,tileN)
```

Example 1: DGEMM with multiple GPUs

So, how'd we do??

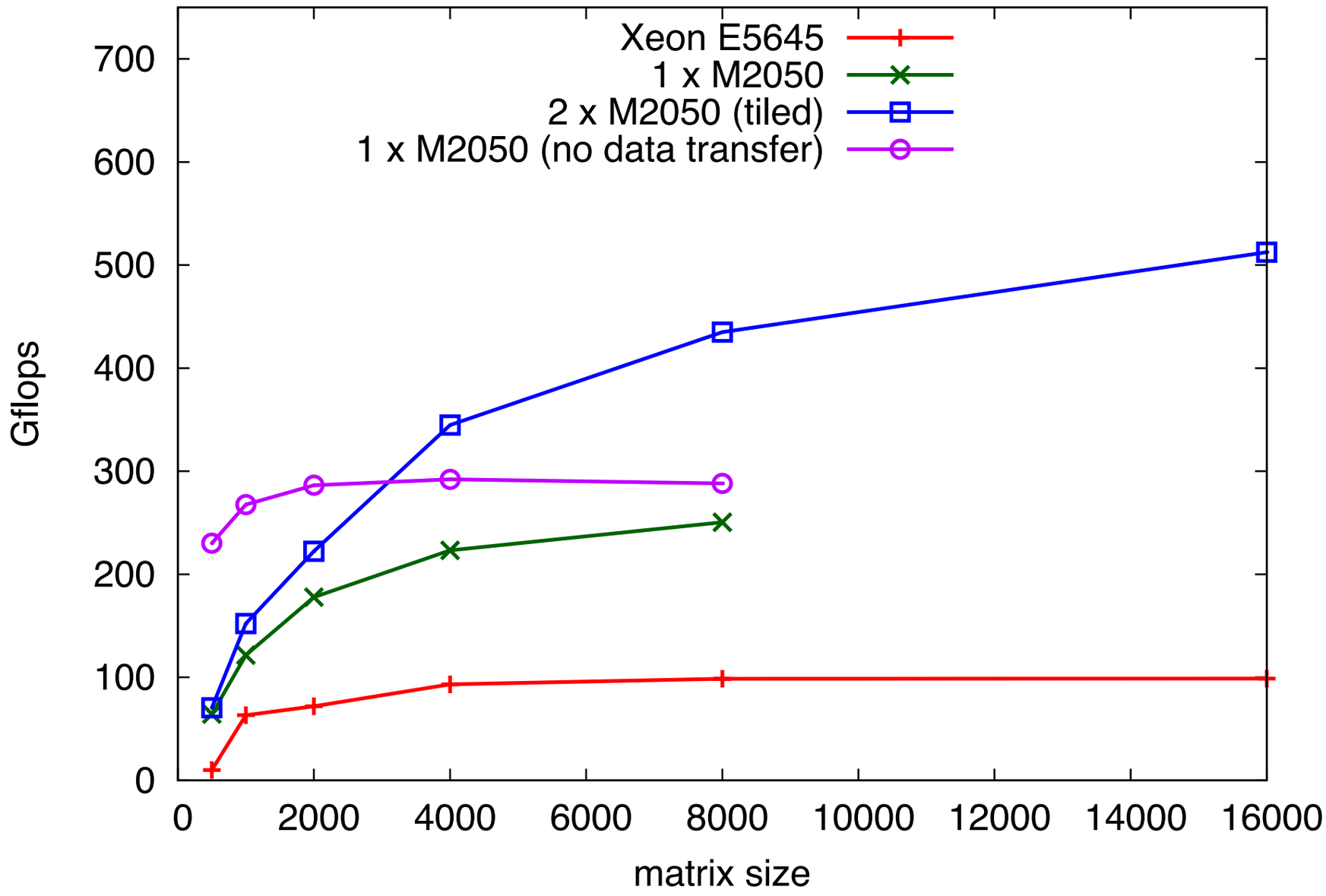


data motion ruins performance for small matrices

if your application allows you to reuse data on the device, do it!

Example 1: DGEMM with multiple GPUs

So, how'd we do??



Can we mask data motion?

Example 1: DGEMM with multiple GPUs

Interleave communication and computation:

Use 2 CPU threads and 2 cuda streams

stream: sequence of operations executed on the device in order. operations in a different stream can occur simultaneously

thread 1, stream 1:
    cublasDgemm using current tiles of A and B

thread 2, stream 2:
    copy next tiles of A and B to device

Example 1: DGEMM with multiple GPUs

for actual code, see gpuhelper.cu, line 357

step 1. create streams / events

```
// create streams:
cudaStream_t stream1 = NULL;
cudaStreamCreate(&stream1);
cudaEvent_t estart1,estop1;
cudaEventCreate(&estart1);
cudaEventCreate(&estop1);
```

```
cudaStream_t stream2 = NULL;
cudaStreamCreate(&stream2);
cudaEvent_t estart2,estop2;
cudaEventCreate(&estart2);
cudaEventCreate(&estop2);
```

```
// cublas will use stream1
cublasSetKernelStream(stream1);
```

step 2. copy initial tile to device

just like before, pack tile into page-locked CPU memory and then transfer

Example 1: DGEMM with multiple GPUs

for actual code, see gpuhelper.cu, line 357

step 3. DGEMM in stream1, copy in stream2

```
#pragma omp parallel num_threads(2)
{

    long int thread2 = omp_get_thread_num();
    if (thread2 == 0) {

        // determine where current tile resides
        double * A_curr = ( tk % 2 == 0 ) ? gpuA : gpuA + offsetA;
        double * B_curr = ( tk % 2 == 0 ) ? gpuB : gpuB + offsetB;

        cudaEventRecord(estart1,stream1);
            cublasDgemm(…)
            cudaStreamSynchronize(stream1);
        cudaEventRecord(estop1,stream1);

}else {
    // copy next tiles if we need them
…
```

Example 1: DGEMM with multiple GPUs

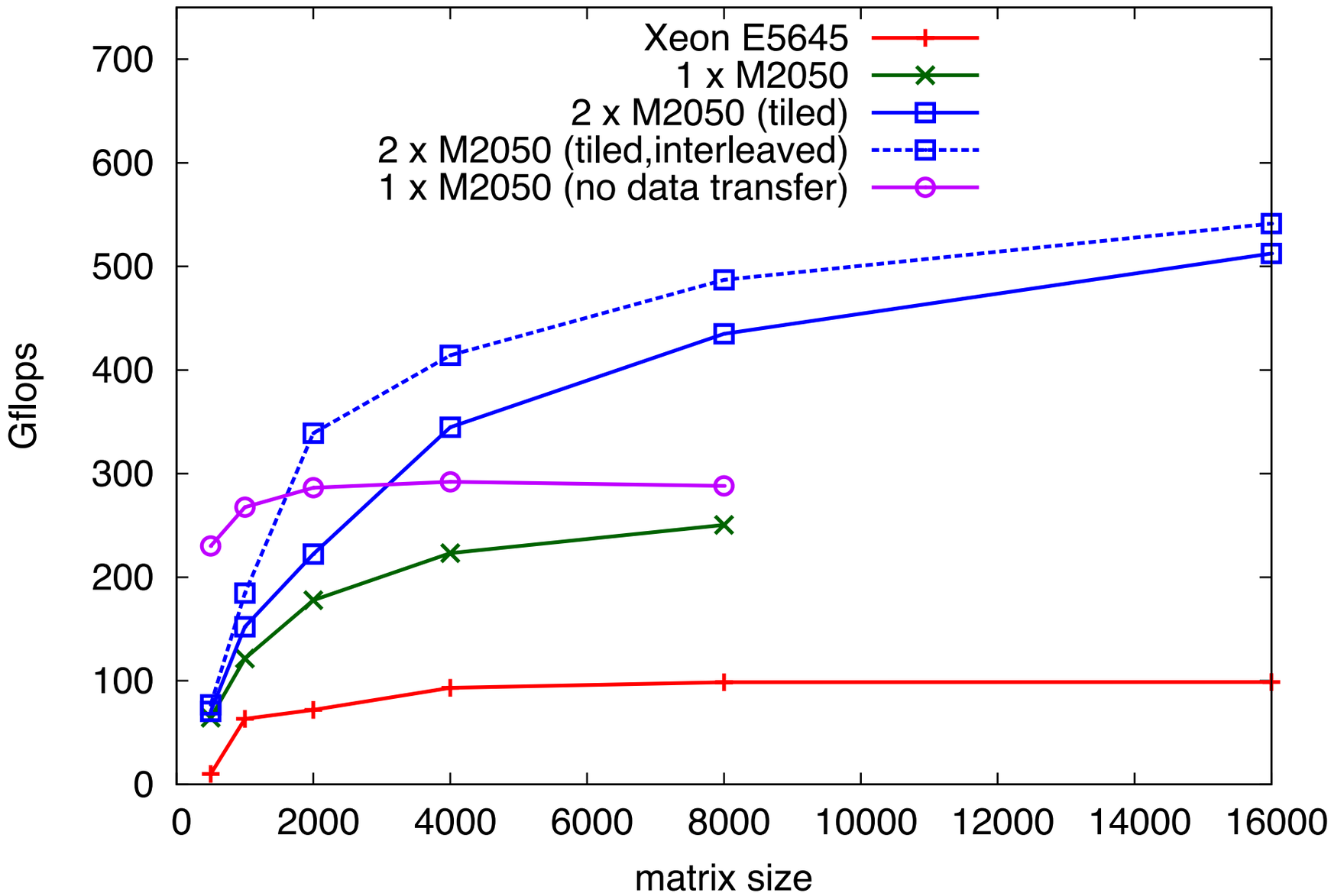
```
…
    }else {
        // copy next tiles if we need them:
        if ( tk < ntilesK - 1 ) {
            double * A_next = ( tk % 2 == 0 ) ? gpuA + offsetA : gpuA;
            double * B_next = ( tk % 2 == 0 ) ? gpuB + offsetB : gpuB;
            cudaEventRecord(estart2,stream2);

                UnpackTile(A,A_next);
                cudaMemcpyAsync(A_next,cpu_buffer[thread],size,
                    cudaMemcpyHostToDevice,stream2);
                cudaStreamSynchronize(stream2);

                UnpackTile(B,B_next);
                cudaMemcpyAsync(B_next,tmp[thread],size,
                    cudaMemcpyHostToDevice,stream2);
                cudaStreamSynchronize(stream2);

            cudaEventRecord(estop2,stream2);
    }
```

note new function call: cudaMemcpyAsync

- requires page-locked cpu memory (allocated with cudaMallocHost)

Example 1: DGEMM with multiple GPUs

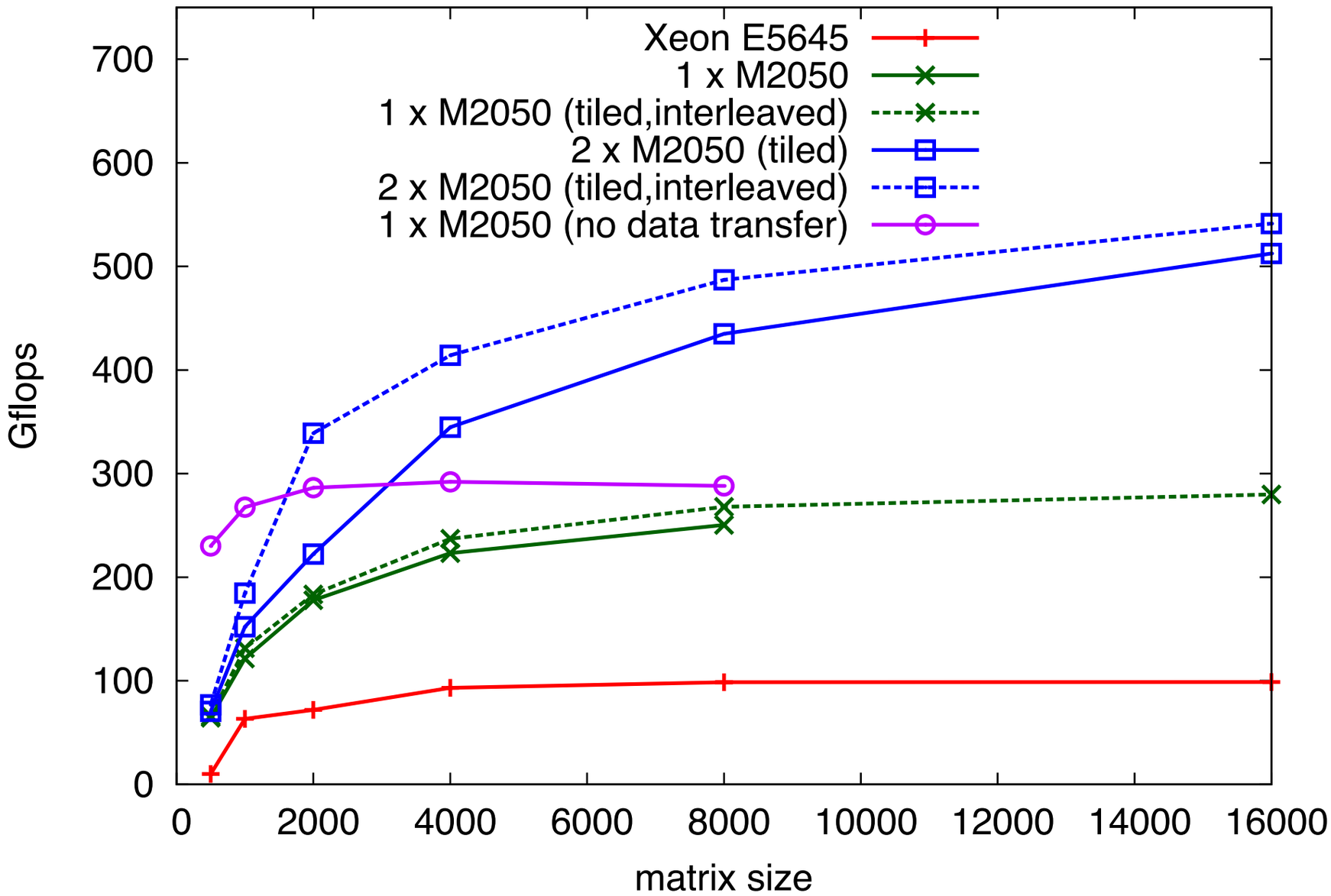
So, how'd we do??



definitely an improvement, but nothing spectacular

Example 1: DGEMM with multiple GPUs

So, how'd we do??



we can play these interleaving tricks even when we have only a single GPU

Example 1: DGEMM with multiple GPUs

now, you try S212/gpu/interleaved_gpu

```
dgemm.x -- cublas dgemm performance

usage: ./dgemm.x n nrepeats kernel transpose

n:          dimension of matrices
nrepeats:   number of times to run the kernel
kernel:     kernel type, allowed values:
            cpu = cpu blas dgemm (mkl)
            naive = cublasDgemm with cudaMemcpy
            nocopy = cublasDgemm with no cudaMemcpy
            interleaved = cublasDgemm in using tiles,
                          and interleaved cudaMemcpy
 transpose: matrix transposes, allowed values:
            nn = 'n','n'
            nt = 'n','t'
            tn = 't','n'
            tt = 't','t'
```

note, only the interleaved kernel makes use of >1 GPU

gpuhelper.cu, line 32: hard code 1 GPU
gpuhelper.cu, lines 221, 355, 487, 624: use “naive” kernel with multiple GPUs

Example 1: DGEMM with multiple GPUs

now, you try S212/gpu/interleaved_gpu

You can use cublasDgemm with any of the other codes you've written during this workshop using the GPUHelper class.

```
#include"gpuhelper.h"
int main() {

    … your awesome code
    GPUHelper * gpu (new GPUHelper());
    gpu->GPUTiledDGEMM('n','t',alpha,A,lda,B,ldb,beta,C,ldc);
}
```

Note:

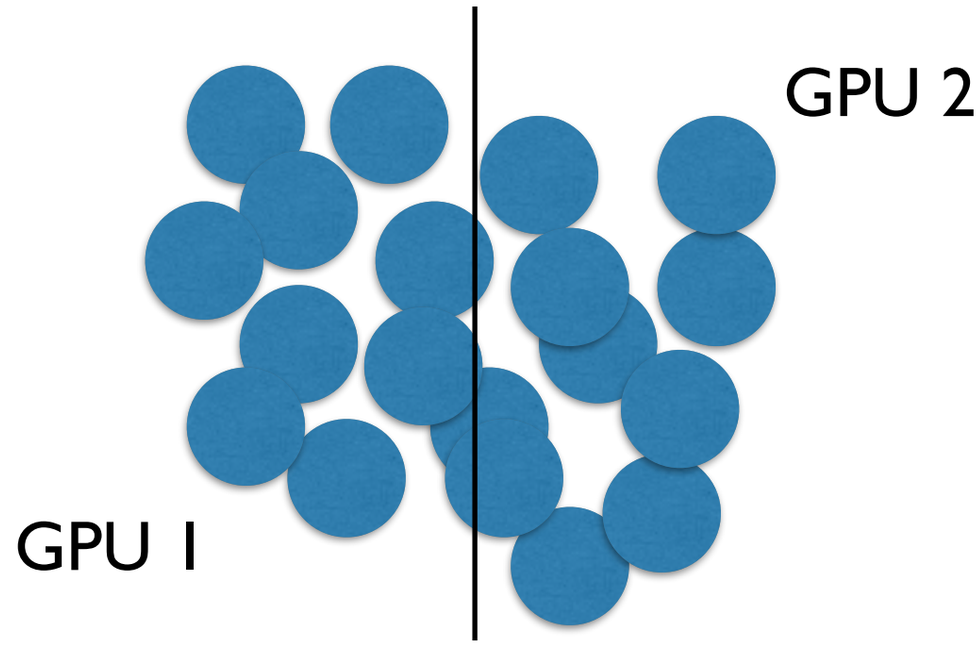
- A, B, C are normal buffers in CPU memory (no need for cudaMallocHost)

- the main code can be compiled without nvcc. You'll need to compile gpuhelper.cu with nvcc (see S212/gpu/interleaved_gpu/Makefile)

- this function assumes Fortran ordering … you may need to adjust your code

Example 2: MD with multiple GPUs

this one is much simpler. if we have N particles and n GPUs, each GPU will update the acceleration, neighbor list, and pair correlation function for N/n of the particles.



We can replicate all data, and both GPUs can perform easy tasks (position update) to avoid data transfers.

Example 2: MD with multiple GPUs

Data transfers between GPUs

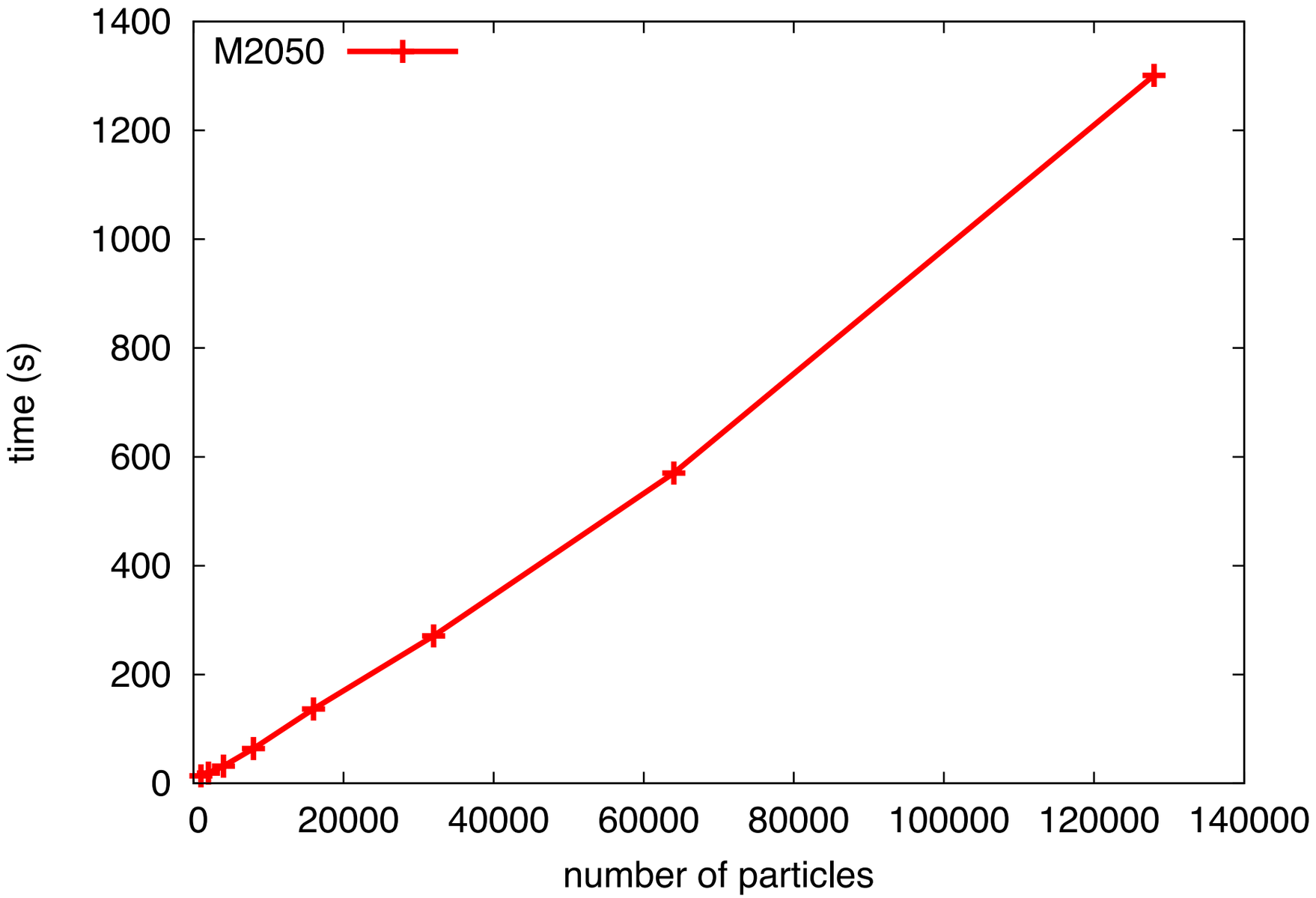
```
cudaMemcpyPeer(buf_target,dev_target,buf_src,dev_src,size);
```

Like other transfers, cudaMemcpyPeer involves data motion across the PCI bus and should be used sparingly.

This cost is the reason that it is better to replicate some work (updating the particle positions, etc.) to avoid data transfers.

Example 2: MD with multiple GPUs

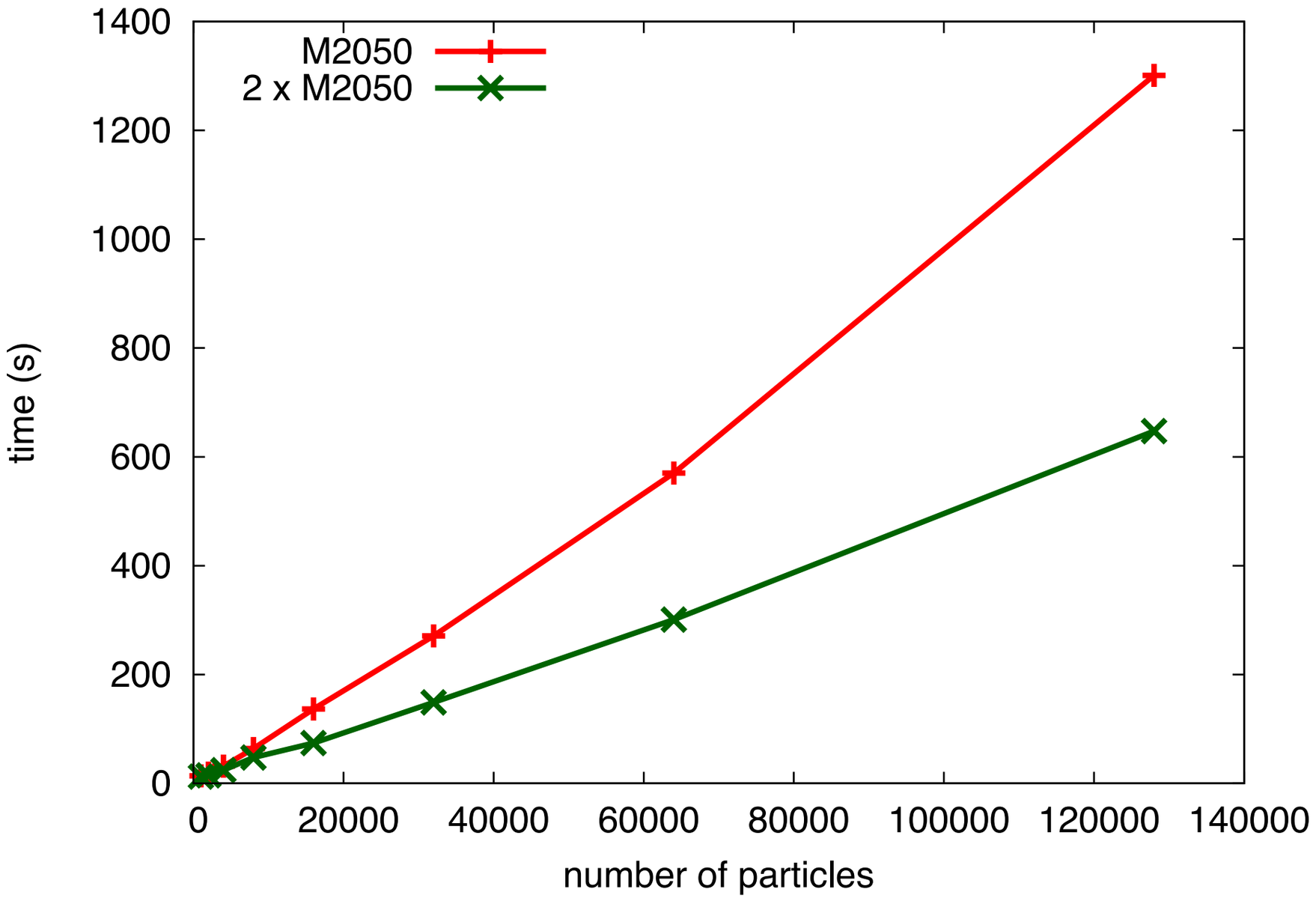
time for acceleration update kernel for system of lennard-jones particles (10000 time steps)



nice near-linear scaling for the single GPU kernel

Example 2: MD with multiple GPUs

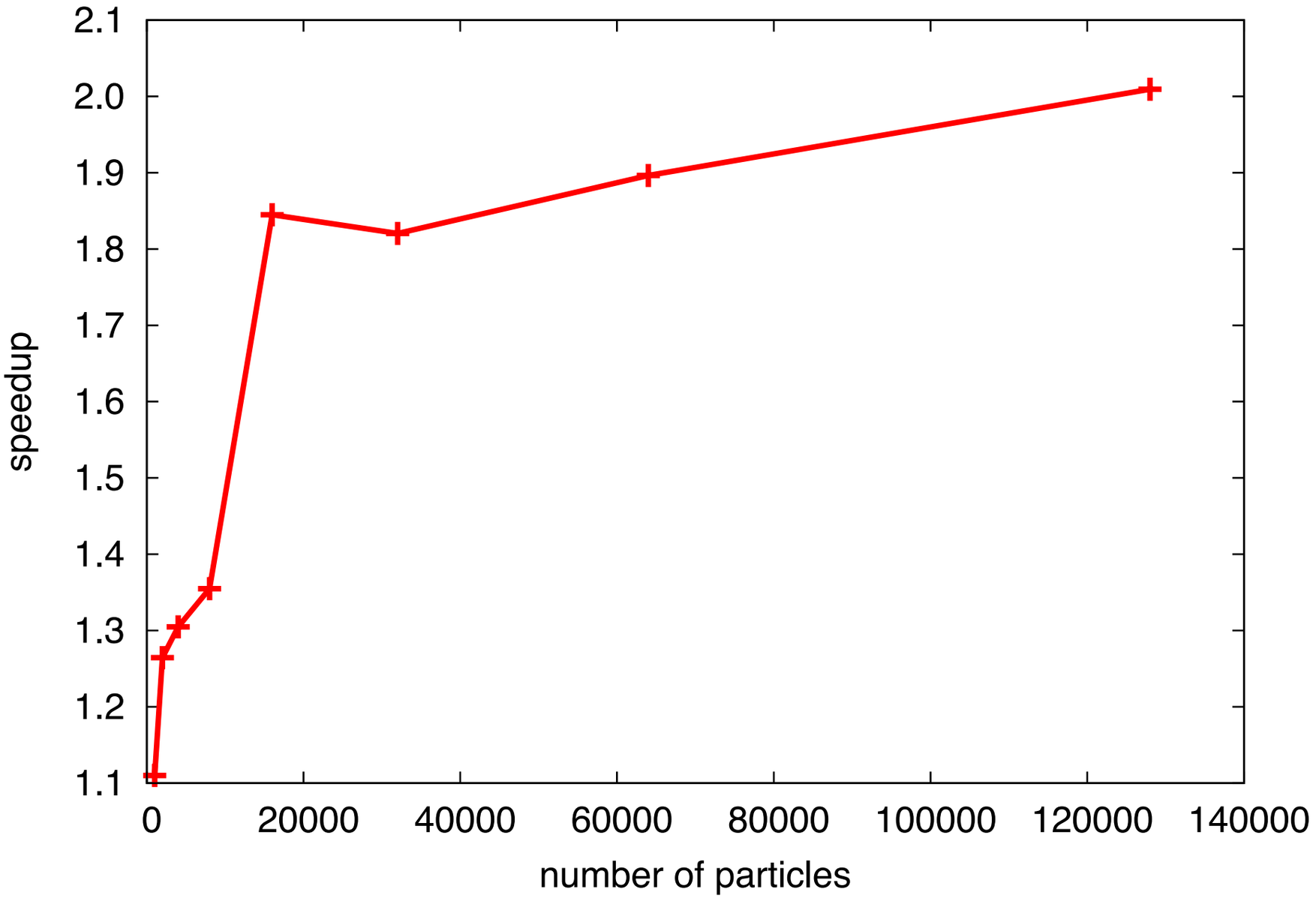
time for acceleration update kernel for system of lennard-jones particles (10000 time steps)



nice near-linear scaling for both algorithms

Example 2: MD with multiple GPUs

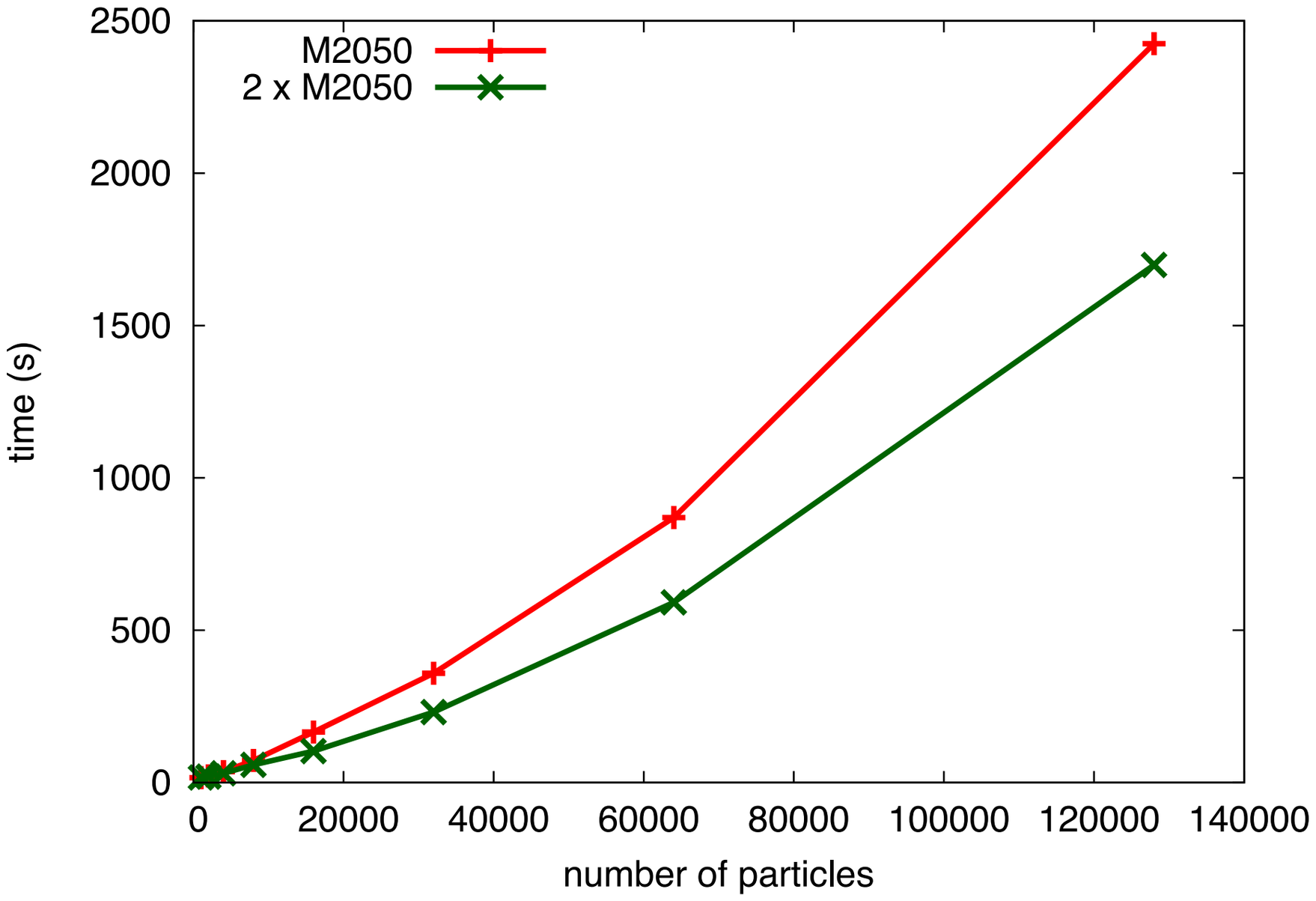
time for acceleration update kernel for system of lennard-jones particles (10000 time steps)



great speedup for 16000+ particles

Example 2: MD with multiple GPUs

performance increase isn't as impressive for the algorithm as a whole …
need multi-GPU algorithm to update the neighbor list!



great speedup for 16000+ particles

Example 2: MD with multiple GPUs

CPU code:
S2I2/gpu/md_simulation/cpu/md_cpu.cc

single GPU code:
S2I2/gpu/md_simulation/gpu/md_gpu.cu

partially parallelized GPU code:
S2I2/gpu/md_simulation/gpu/md_gpu_mult.cu

Now, the fun part:

(1) parallelize the acceleration kernel in md_gpu.cu. Can your code beat mine?
(2) parallelize the neighbor list update. Do you see a factor of 2 increase?