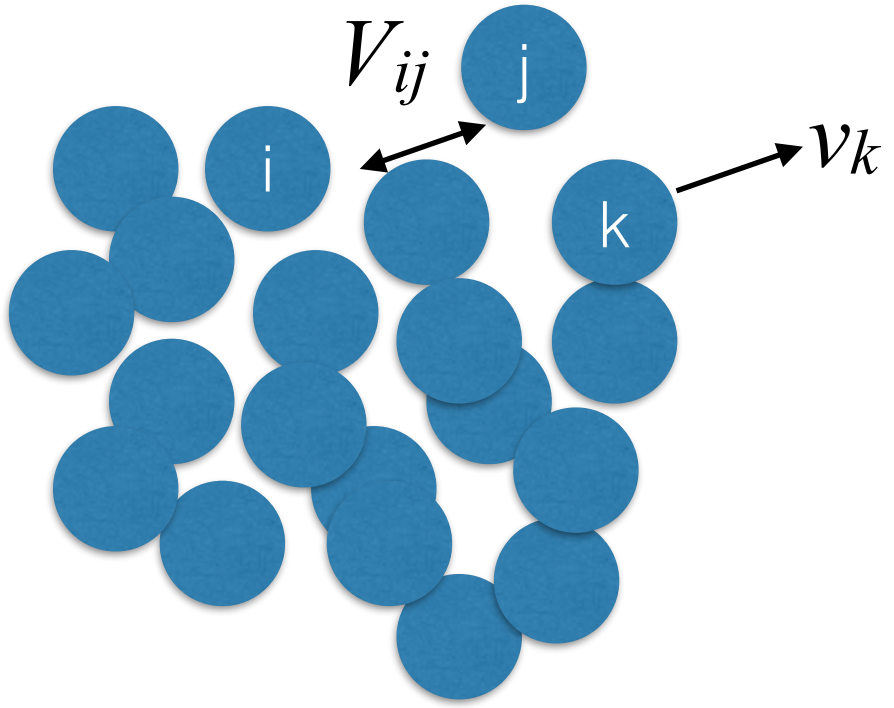
Scientific programming with graphics processing units

Session 2: molecular dynamics

Eugene DePrince
Florida State University

Accelerating molecular dynamics



particles positions are updated in time according to Newton's 2nd law: *f=ma*

for lennard-jones particles, the potential is

$$V_{ij} = \frac{A}{r_{ij}^{12}} - \frac{B}{r_{ij}^{6}}$$

and the corresponding forces are

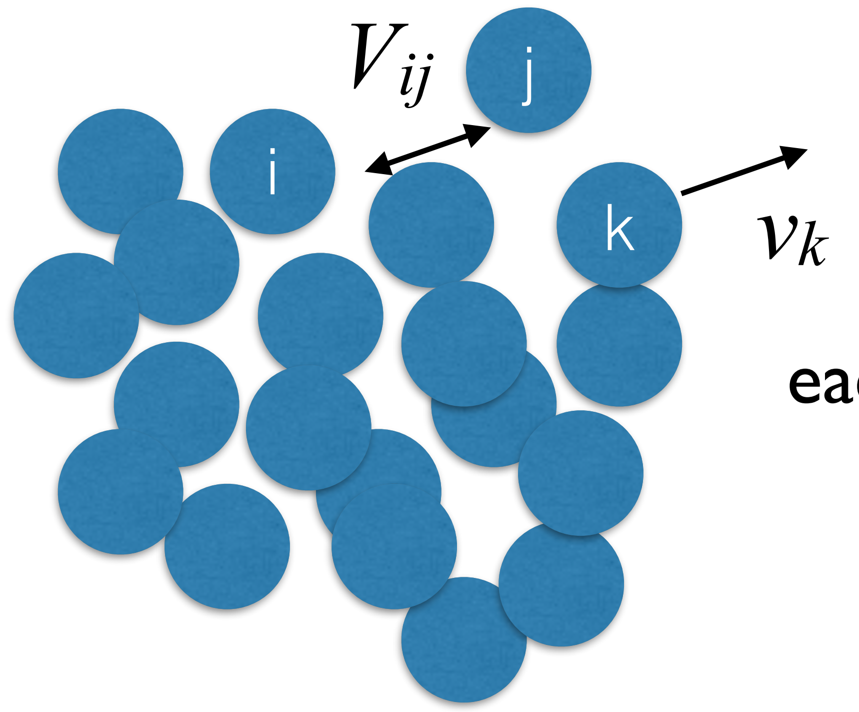
$$f_{x_i} = -\sum_j \frac{\partial V_{ij}}{\partial x_i} \qquad f_{y_i} = -\sum_j \frac{\partial V_{ij}}{\partial y_i}$$

$$f_{x_i} = \sum_j (x_i - x_j) \left(\frac{12A}{r_{ij}^{14}} - \frac{6B}{r_{ij}^{8}} \right)$$

Evaluating forces for molecular dynamics

CPU and GPU code for this section can be found in

S212/lj/main.cu and S212/lj/cpu_only.cc



$$f_{x_i} = \sum_j (x_i - x_j)\left(\frac{12A}{r_{ij}^{14}} - \frac{6B}{r_{ij}^{8}}\right)$$

each particle feels a potential from all other particles

for CPU and GPU code, it makes sense to parallelize over particles and evaluate force contribution from all other particles

technically $f_i = -f_j$, but we will do twice as much work to make parallelizing the code easier

Evaluating forces for molecular dynamics

OMP CPU version

parallelize over particles

```
#pragma omp parallel for schedule(dynamic) num_threads (nthreads)
for (int i = 0; i < n; i++) {

    double xi = x[i];
    double yi = y[i];
    double zi = z[i];

    double fxi = 0.0;
    double fyi = 0.0;
    double fzi = 0.0;

    for (int j = 0; j < n; j++) {
        if ( i == j ) continue;
        double dx  = xi - x[j];
        double dy  = yi - y[j];
        double dz  = zi - z[j];
        …
```

Evaluating forces for molecular dynamics

OMP CPU version

```
                …
                double r2  = dx*dx + dy*dy + dz*dz;
                double r6  = r2*r2*r2;
                double r8  = r6*r2;
                double r14 = r6*r6*r2;
                double f   = A12 / r14 - B6 / r8;
                fxi += dx * f;
                fyi += dy * f;
                fzi += dz * f;

            }

            fx[i] = fxi;
            fy[i] = fyi;
            fz[i] = fzi;
        }
    }
```

pretty straightforward! later, we'll improve this by

(1) only including contributions from nearby particles and
(2) including periodic boundary conditions

Evaluating forces for molecular dynamics

CUDA version: host code

```
// pointers to gpu memory
double * gpu_x;
double * gpu_y;
double * gpu_z;

double * gpu_fx;
double * gpu_fy;
double * gpu_fz;

// allocate GPU memory
cudaMalloc((void**)&gpu_x,n*sizeof(double));
cudaMalloc((void**)&gpu_y,n*sizeof(double));
cudaMalloc((void**)&gpu_z,n*sizeof(double));

cudaMalloc((void**)&gpu_fx,n*sizeof(double));
cudaMalloc((void**)&gpu_fy,n*sizeof(double));
cudaMalloc((void**)&gpu_fz,n*sizeof(double));

// copy particle positions to GPU
cudaMemcpy(gpu_x,x,n*sizeof(double),cudaMemcpyHostToDevice);
cudaMemcpy(gpu_y,y,n*sizeof(double),cudaMemcpyHostToDevice);
cudaMemcpy(gpu_z,z,n*sizeof(double),cudaMemcpyHostToDevice);
```

Evaluating forces for molecular dynamics

CUDA version: host code cont.

```
// threads per block should be multiple of the warp
// size (32) and has max value cudaProp.maxThreadsPerBlock
int threads_per_block = NUM_THREADS;
int maxblocks         = MAX_BLOCKS;

long int nblocks_x = n / threads_per_block;
long int nblocks_y = 1;

if ( n % threads_per_block != 0 ) {
   nblocks_x = (n + threads_per_block - n % threads_per_block)/threads_per_block;
}

if (nblocks_x > maxblocks){
   nblocks_y = nblocks_x / maxblocks + 1;
   nblocks_x = nblocks_x / nblocks_y + 1;
}

// a two-dimensional grid: nblocks_x by nblocks_y
dim3 dimgrid (nblocks_x,nblocks_y);

ForcesOnGPU<<<dimgrid,threads_per_block>>>
    (n,gpu_x,gpu_y,gpu_z,gpu_fx,gpu_fy,gpu_fz,A12,B6);
```

Evaluating forces for molecular dynamics

try writing your own cuda kernel for this function

CPU version: S212/gpu/lj/cpu_only.cc

1. remember, each thread executes the same code
2. each thread should handle one particle (index i from CPU code)
3. don't forget to check your index bounds!

Evaluating forces for molecular dynamics

CUDA version: device code

```
// evaluate forces on GPU
__global__ void ForcesOnGPU(int n, double * x, double * y, double * z,
         double * fx, double * fy, double * fz,double A12, double B6) {

    int blockid = blockIdx.x*gridDim.y + blockIdx.y;
    int i       = blockid*blockDim.x + threadIdx.x;
    if ( i >= n ) return;

    double xi = x[i];
    double yi = y[i];
    double zi = z[i];

    double fxi = 0.0;
    double fyi = 0.0;
    double fzi = 0.0;

    for (int j = 0; j < n; j++) {
        if ( j == i ) continue;

        double dx  = xi - x[j];
        double dy  = yi - y[j];
        double dz  = zi - z[j];
```

Evaluating forces for molecular dynamics

CUDA version: device code

```
        double r2  = dx*dx + dy*dy + dz*dz;
        double r6  = r2*r2*r2;
        double r8  = r6*r2;
        double r14 = r6*r6*r2;
        double f   = A12 / r14 - B6 / r8;

        fxi += dx * f;
        fyi += dy * f;
        fzi += dz * f;

    }

    fx[i] = fxi;
    fy[i] = fyi;
    fz[i] = fzi;

}
```

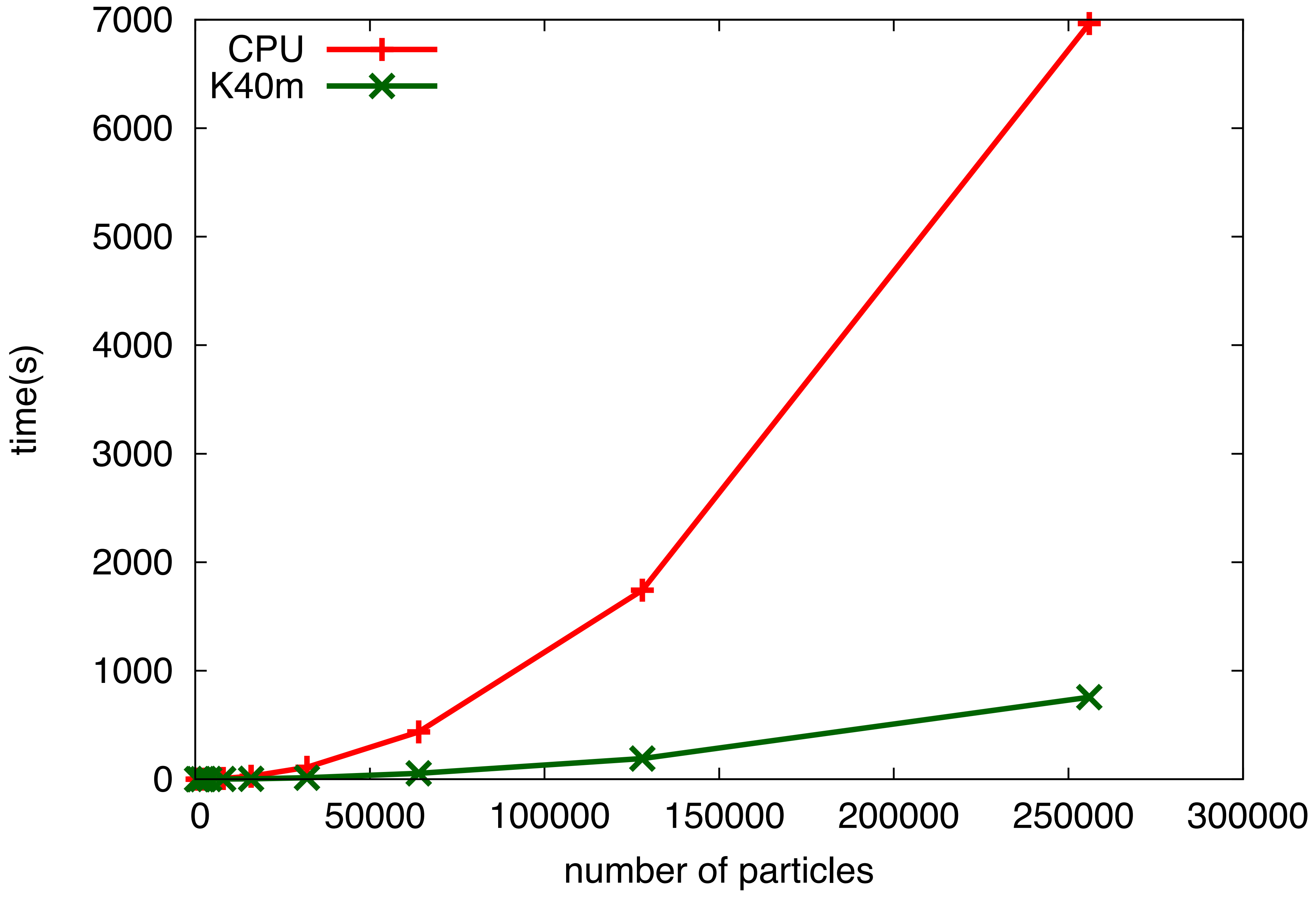
Evaluating forces for molecular dynamics

CUDA version: device code

```
        double r2  = dx*dx + dy*dy + dz*dz;
        double r6  = r2*r2*r2;
        double r8  = r6*r2;
        double r14 = r6*r6*r2;
        double f   = A12 / r14 - B6 / r8;

        fxi += dx * f;
        fyi += dy * f;
        fzi += dz * f;

    }

    fx[i] = fxi;
    fy[i] = fyi;
    fz[i] = fzi;

}
```

Evaluating forces for molecular dynamics

so, how'd we do?

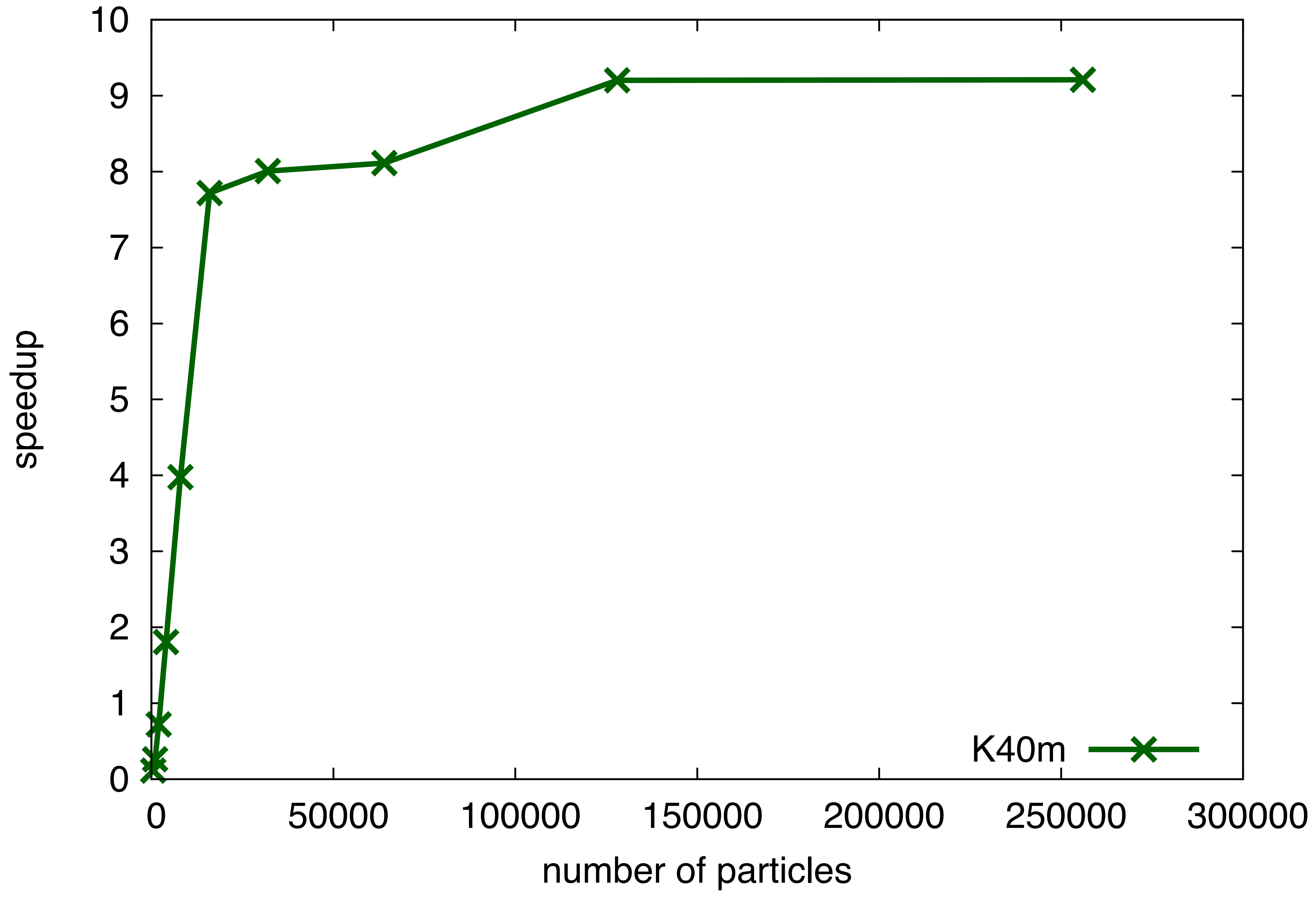


100 force evaluations, no data transfers

CPU = 2 8-core intel sandy bridge CPUs (blueridge)

Evaluating forces for molecular dynamics

so, how'd we do?



GPU slower for small systems, but much more efficient for large systems
can we do even better???

__shared__ memory

- accessing GPU global memory is very costly

shared memory:

- threads within a block can share information with each other

- much faster access than global memory

- like having complete control over cache

- access over warps (32 threads, with consecutive threadIdx.x)
- 32 threads execute simultaneously (SIMD style)
- branching causes threads to execute in serial (ifs are bad)
- threads in a warp always belong to the same block

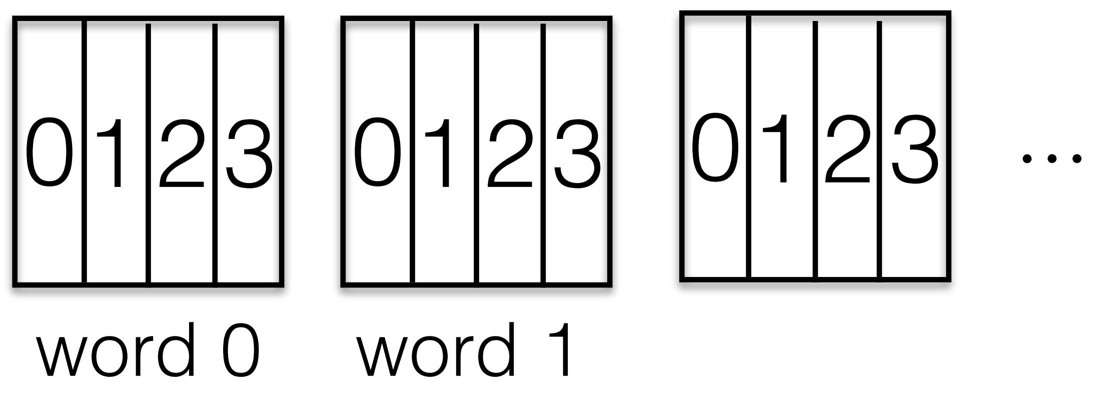
- threads of a warp all access shared memory together
- there are fast accesses and slow accesses (bank conflicts)

- 64k shared/L
called words (

- shared mem
read

__shared__ memory

- 64k shared memory per SM, broken into 4 byte sections, called words (1/2 double).



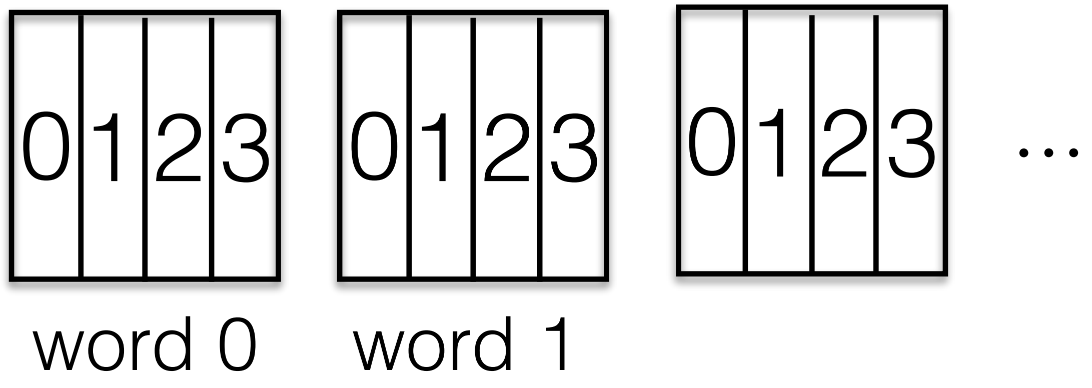
- there are 32 banks, and each successive word belongs to a different bank (word 0 and word 32 belong to the same bank)

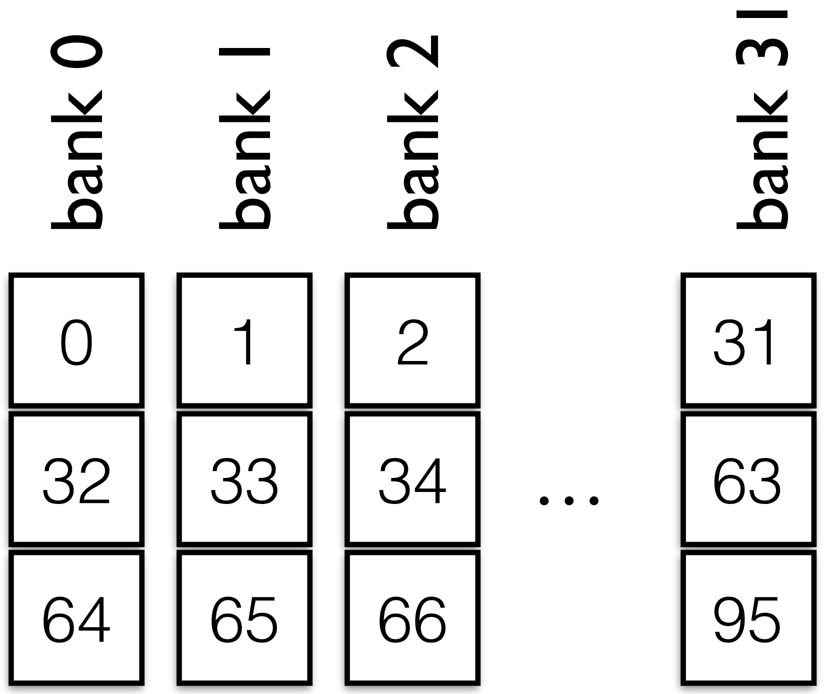
- shared memory is read in entire words. ask for a byte, a word is read

- reading a double requires TWO bank requests (since a double is two words)

__shared__ memory

- 64k shared memory per SM, broken into 4 byte sections, called words (1/2 double).





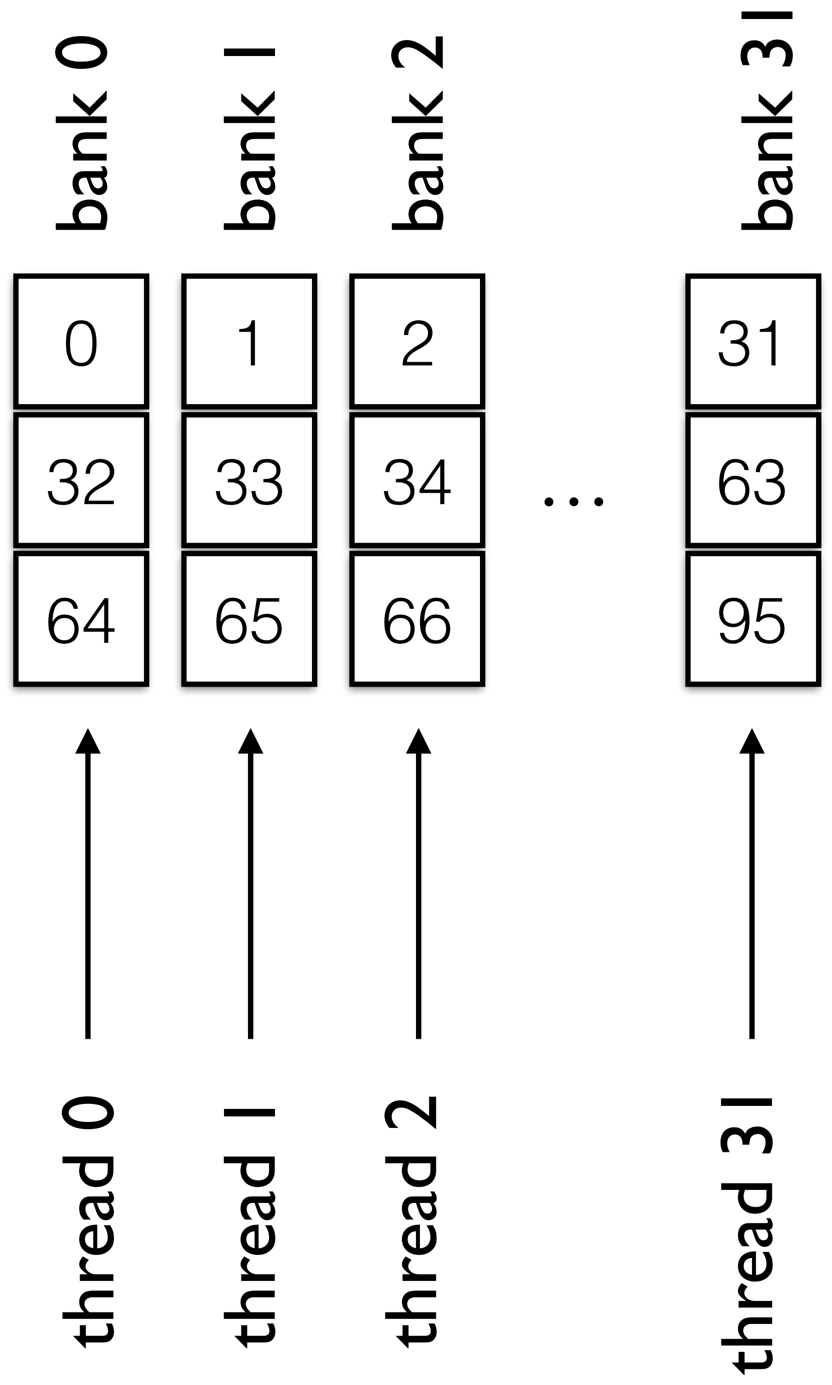
there are 32 banks, and each successive word belongs to a different bank (word 0 and word 32 belong to the same bank)

- shared memory is read in entire words. ask for a byte, a word is read

- reading a double requires TWO bank requests (since a double is two words)

__shared__ memory

threads in a warp access banks in different ways

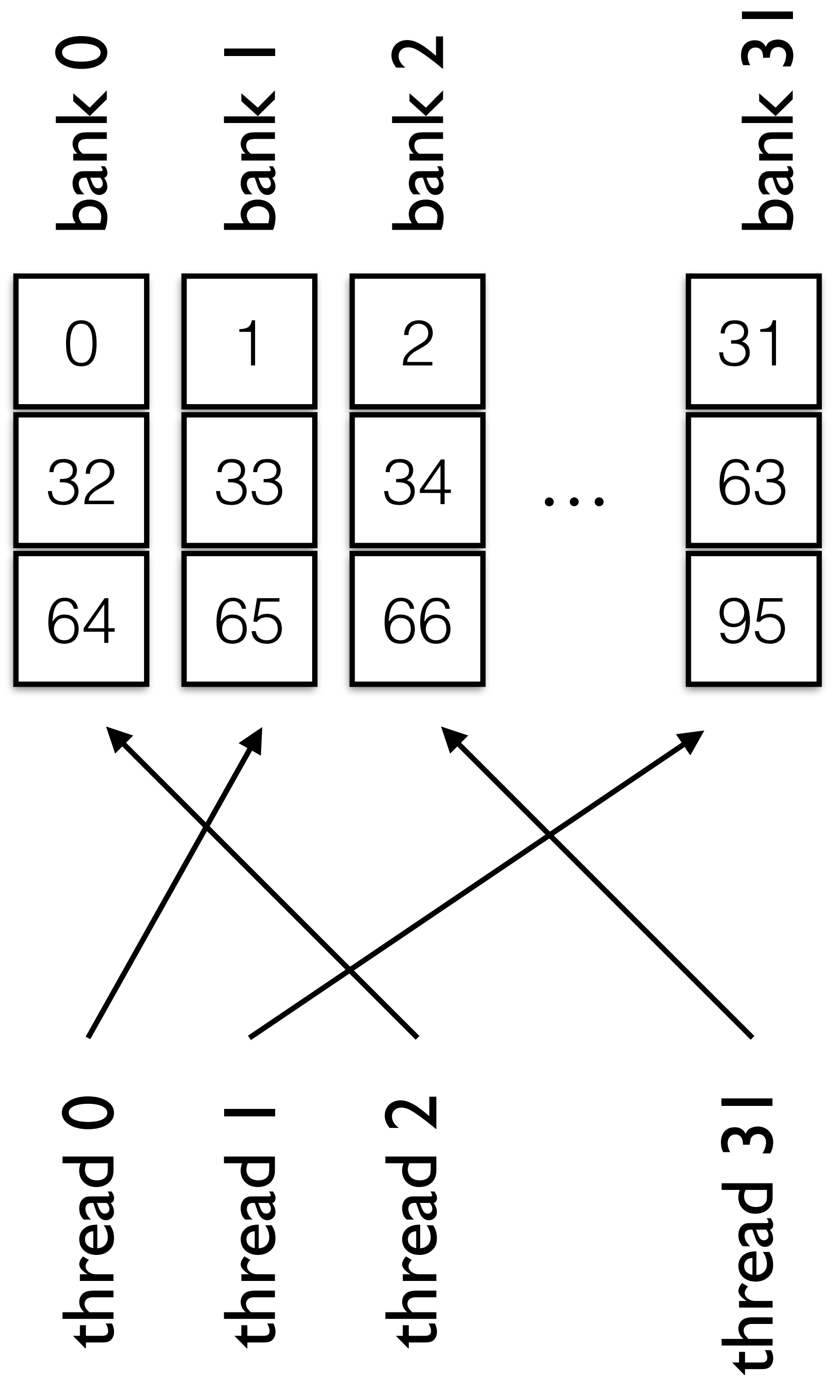


different threads,
different banks:

cool

__shared__ memory

threads in a warp access banks in different ways

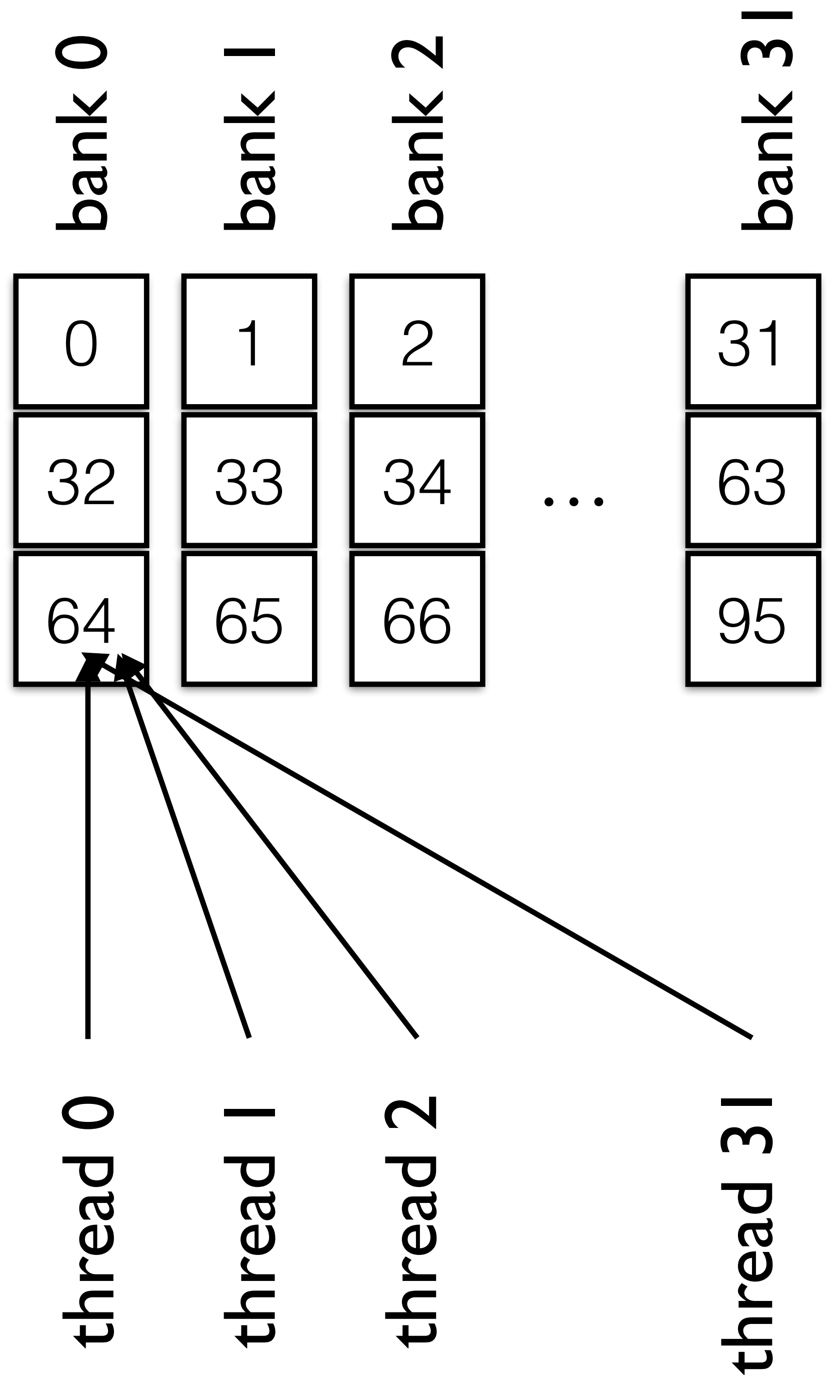


different threads,
different banks:

cool

__shared__ memory

threads in a warp access banks in different ways

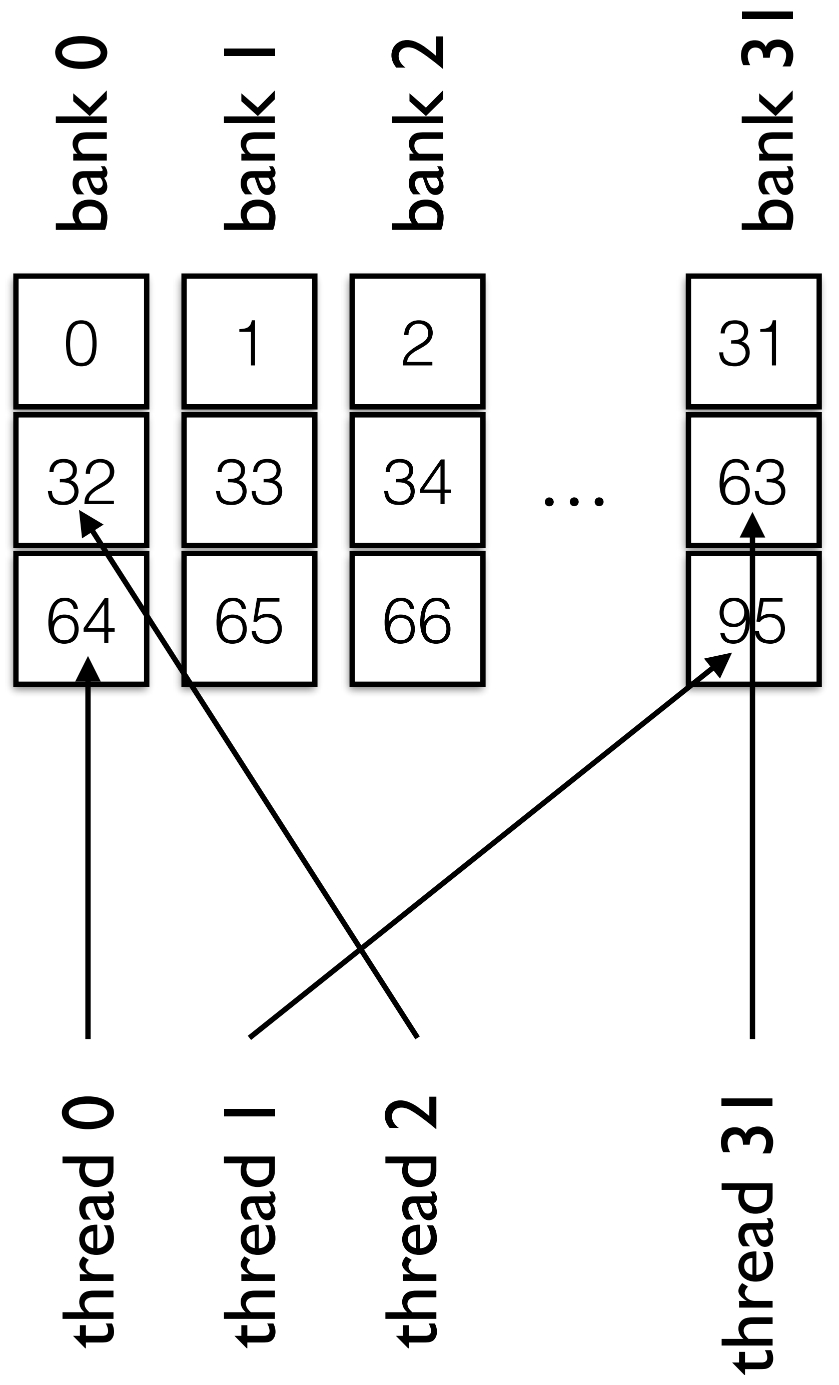


different threads,
same bank, same word:

cool

__shared__ memory

threads in a warp access banks in different ways



different threads,
same bank, *different* word:

not cool

bank conflict! access to
shared memory is serialized

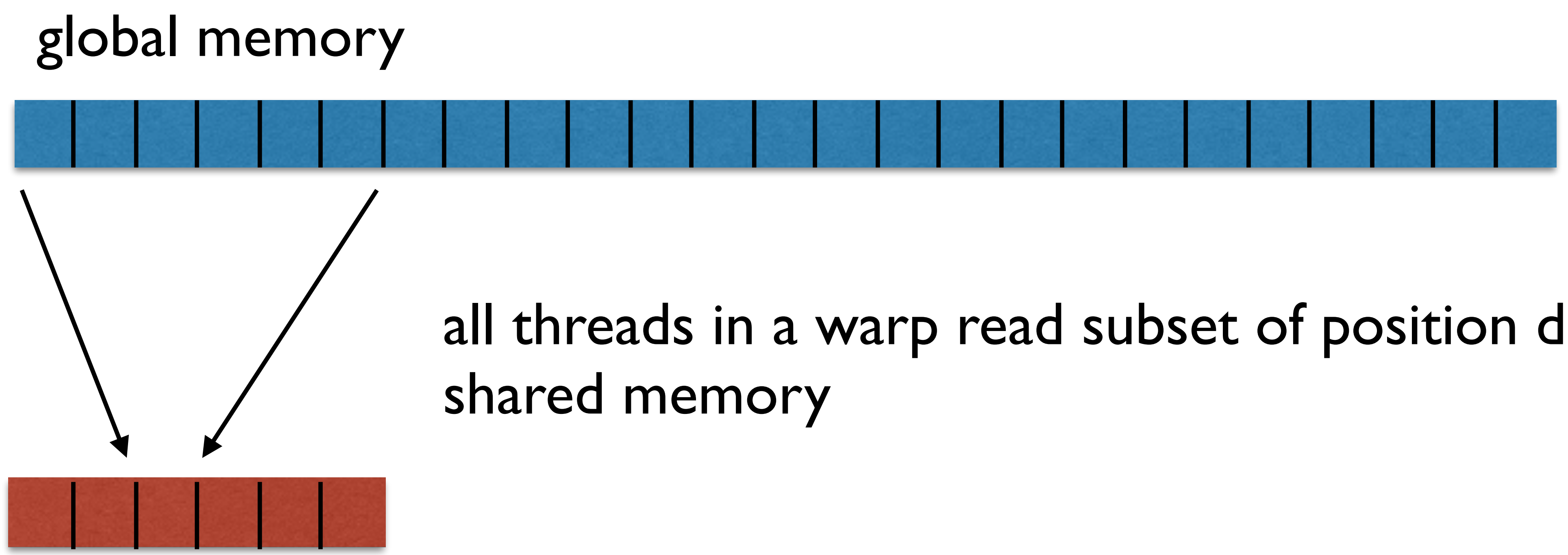
__shared__ memory

how does this relate to our problem?

```
// evaluate forces on GPU
__global__ void ForcesOnGPU(int n, double * x, double * y, double * z,
         double * fx, double * fy, double * fz,double A12, double B6) {

    int blockid = blockIdx.x*gridDim.y + blockIdx.y;
    int i       = blockid*blockDim.x + threadIdx.x;
    if ( i >= n ) return;

    double xi = x[i];
    double yi = y[i];
    double zi = z[i];

    double fxi = 0.0;
    double fyi = 0.0;
    double fzi = 0.0;

    for (int j = 0; j < n; j++) {
        if ( j == i ) continue;

        double dx  = xi - x[j];
        double dy  = yi - y[j];
        double dz  = zi - z[j];
```

we're doing n^2 global memory reads
is this necessary? no!

__shared__ memory

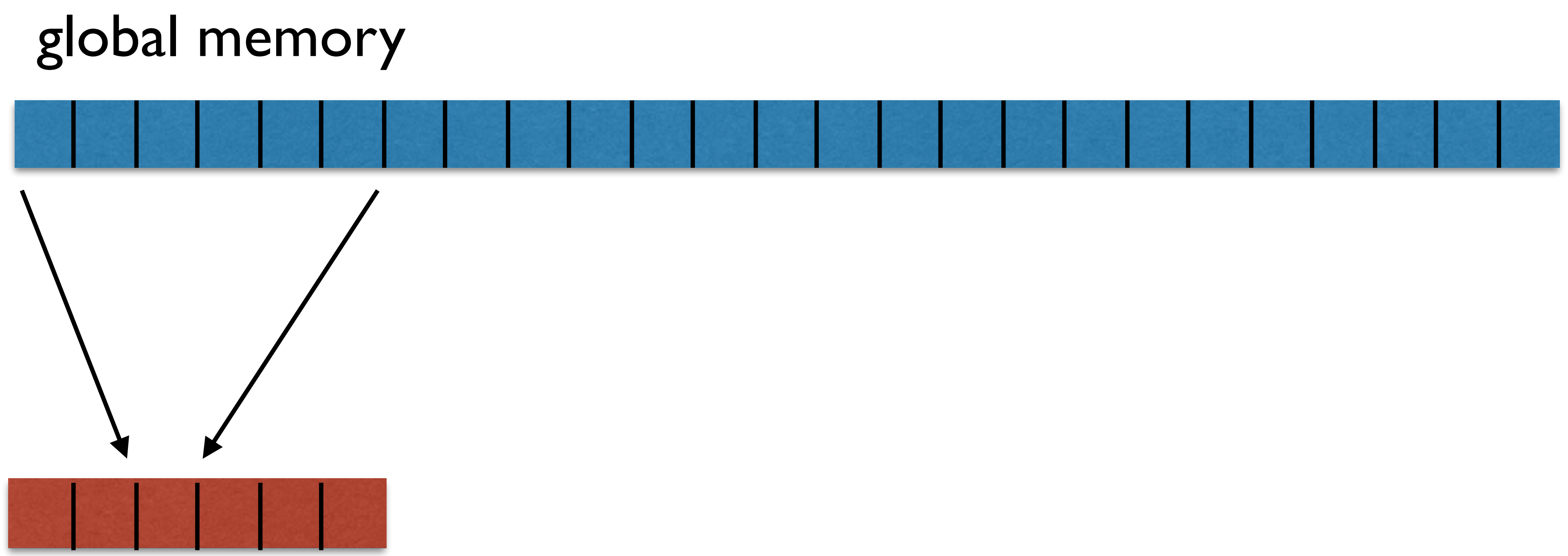


all threads in a warp read subset of position data into shared memory

now, in our inner loop, read from shared memory

result: n reads from global memory, n^2 reads from shared memory

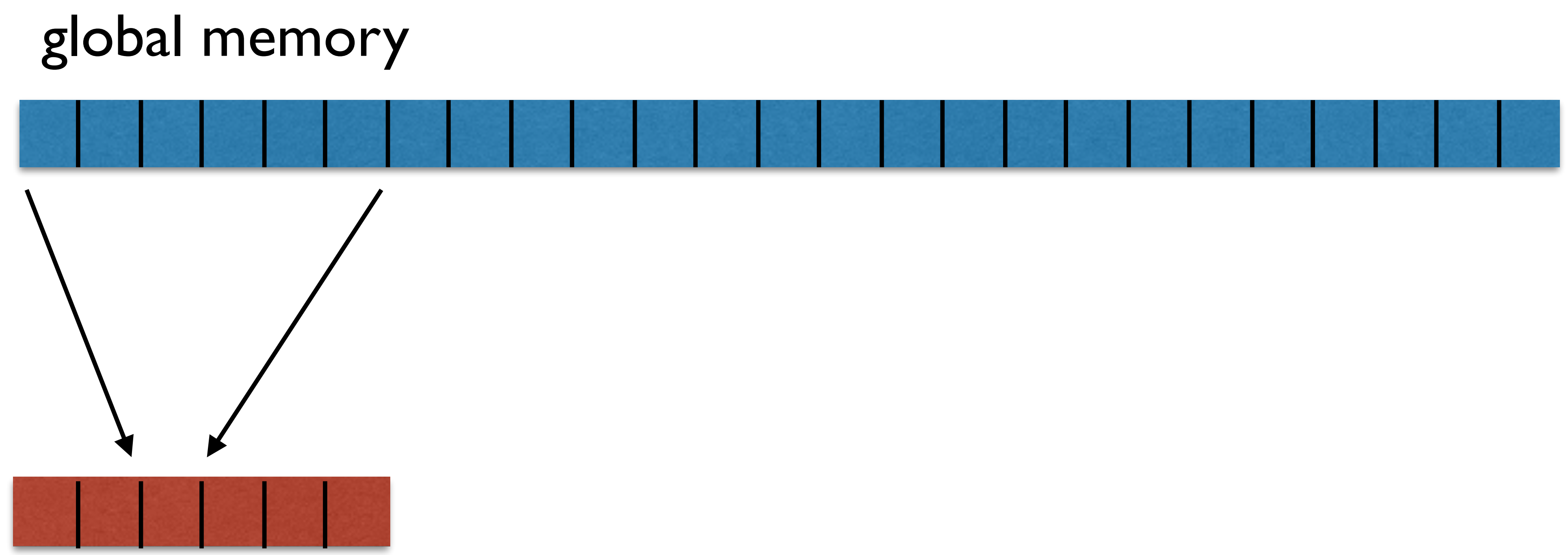
__shared__ memory



looping / indexing becomes more complicated, though, because we can't fit all n particles positions in shared memory

need to loop over chunks of data

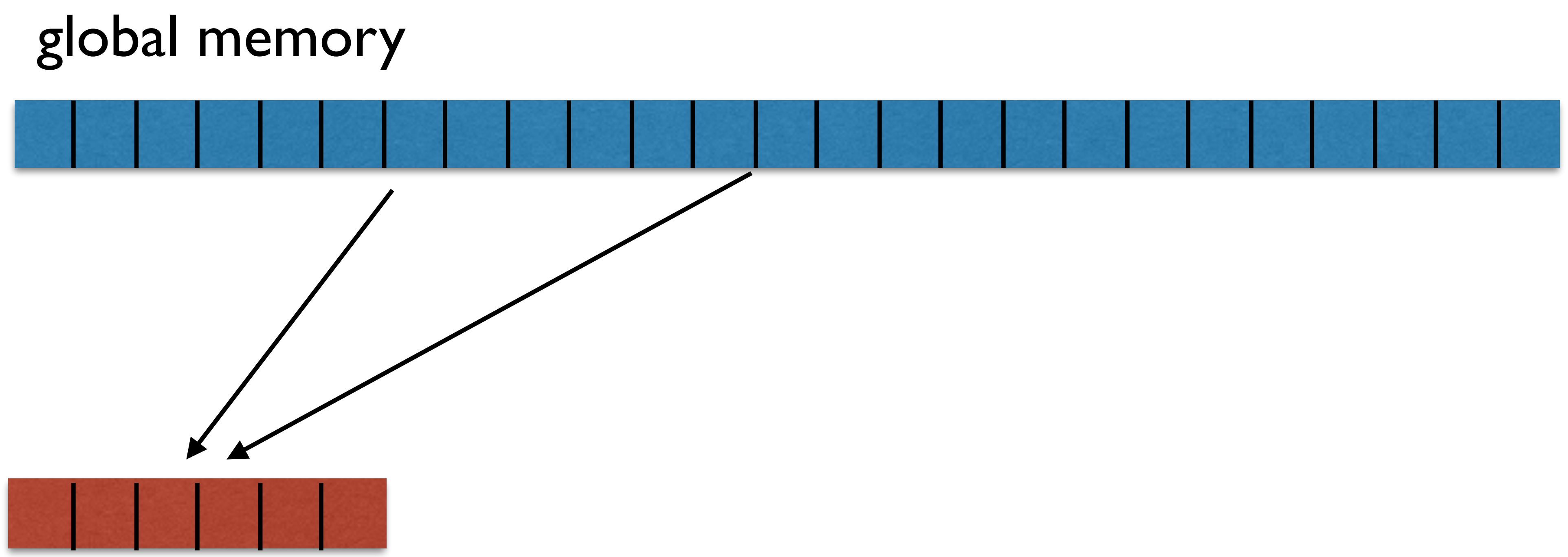
__shared__ memory



block 1:

particle i sees particles 0-31 (if we have 32 threads per block)

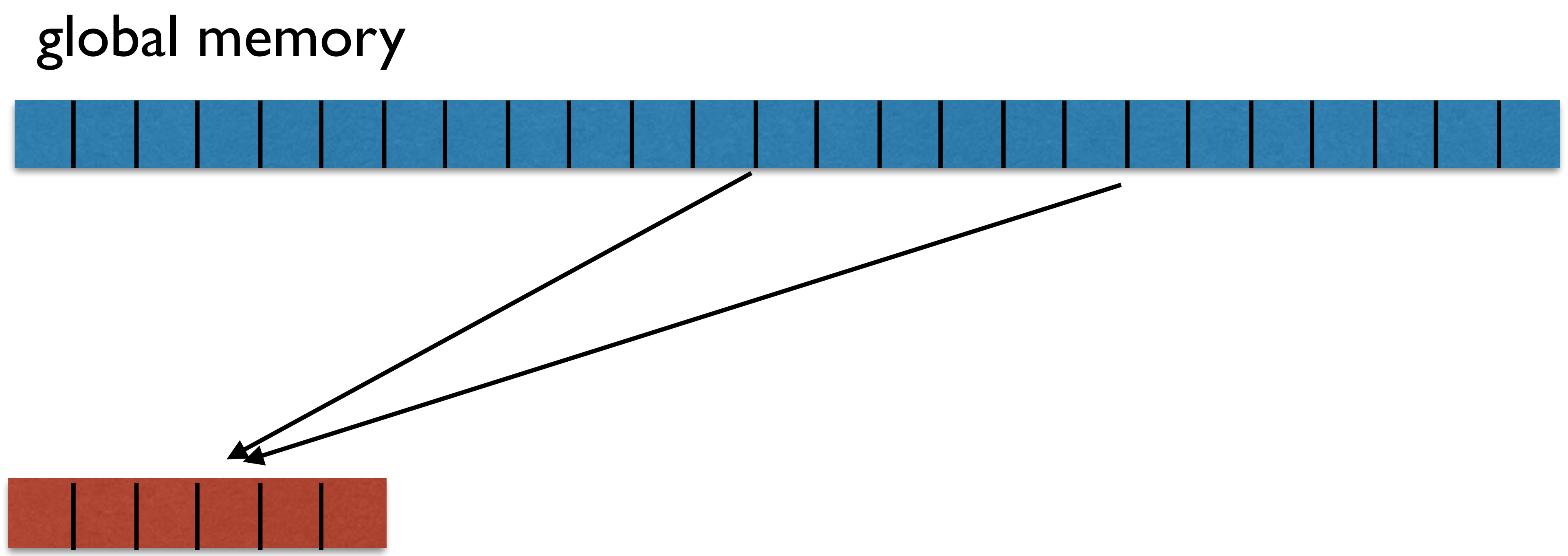
__shared__ memory



block 2:

particle i sees particles 32-63 (if we have 32 threads per block)

__shared__ memory



block 3:

particle i sees particles 64-95 (if we have 32 threads per block)

and so on!

__shared__ memory

code:

```
// evaluate forces on GPU, use shared memory
__global__ void ForcesSharedMemory(int n, double * x, double * y, double * z,
              double * fx, double * fy, double * fz,double A12, double B6) {

    __shared__ double xj[NUM_THREADS];
    __shared__ double yj[NUM_THREADS];
    __shared__ double zj[NUM_THREADS];

    int blockid = blockIdx.x*gridDim.y + blockIdx.y;
    int i       = blockid*blockDim.x + threadIdx.x;

    double xi = 0.0;
    double yi = 0.0;
    double zi = 0.0;
    if ( i < n ) {
        xi = x[i];
        yi = y[i];
        zi = z[i];
    }

    double fxi = 0.0;
    double fyi = 0.0;
    double fzi = 0.0;
```

shared memory

check index, but DO NOT RETURN
why? we need the extra threads to manage shared memory

__shared__ memory

code:

```
int j = 0;
while( j + blockDim.x <= n ) {

    // load xj, yj, zj into shared memory
    xj[threadIdx.x] = x[j + threadIdx.x];
    yj[threadIdx.x] = y[j + threadIdx.x];
    zj[threadIdx.x] = z[j + threadIdx.x];

    // synchronize threads
    __syncthreads();

    for (int myj = 0; myj < blockDim.x; myj++) {

        double dx  = xi - xj[myj];
        double dy  = yi - yj[myj];
        double dz  = zi - zj[myj];
```

make sure the next blockDim.x particles have indices <= n

put a barrier to make sure all of the memory is loaded

loop over blockDim.x particles

read from shared memory

__shared__ memory

code:

trying to prevent branching (avoiding if i==j continue) … not sure if this really helps

```
            double r2  = dx*dx + dy*dy + dz*dz + 10000000.0 * ((j+myj)==i);
            double r6  = r2*r2*r2;
            double r8  = r6*r2;
            double r14 = r6*r6*r2;
            double f   = A12 / r14 - B6 / r8;

            // slowest step
            fxi += dx * f;
            fyi += dy * f;
            fzi += dz * f;

        }

        // synchronize threads
        __syncthreads();

        j += blockDim.x;
    }
```

put another barrier before reading next chunk of positions

__shared__ memory

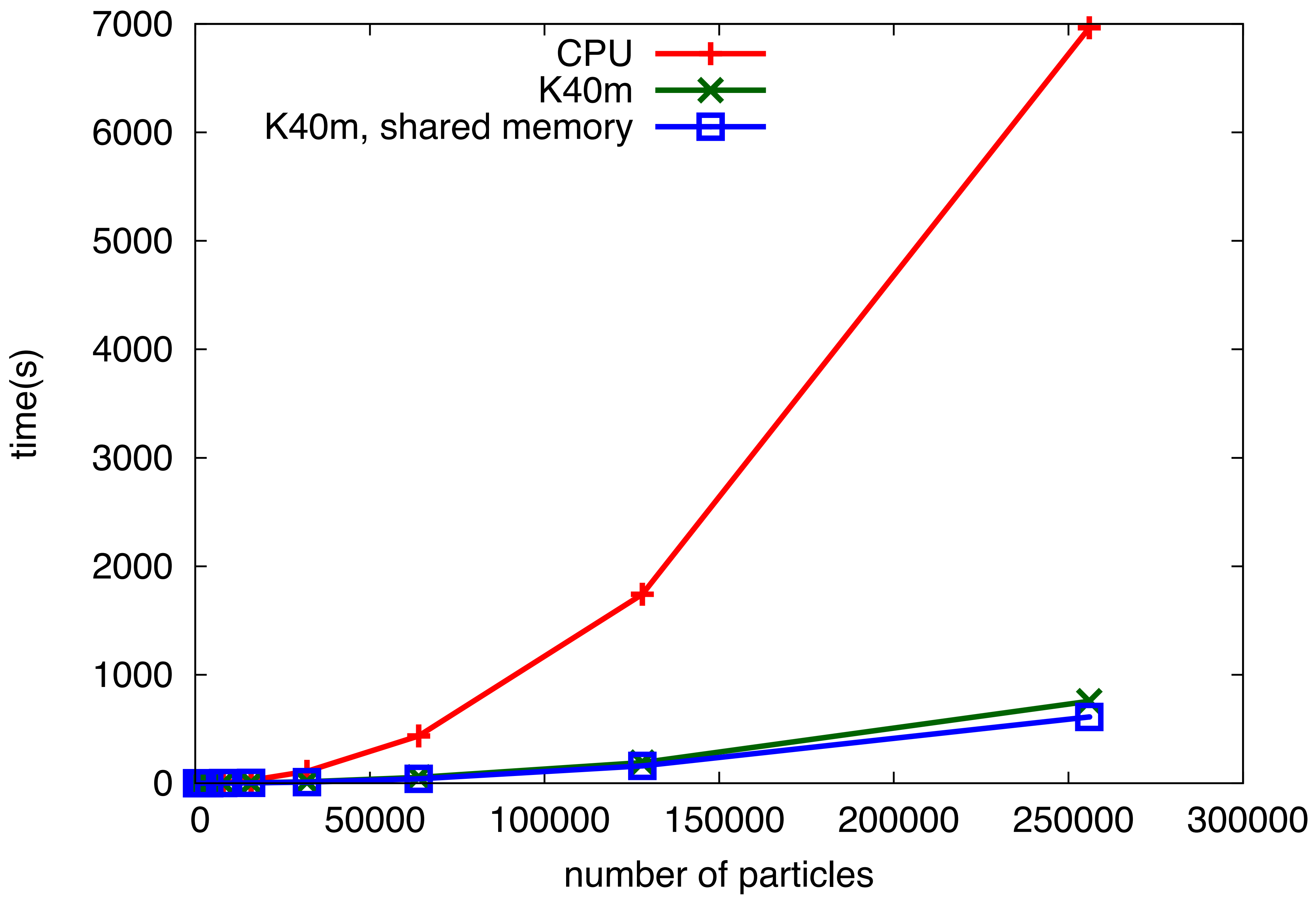
since n might not be an even multiple of blockDim.x, we might have some leftover work to do

```
int leftover = n - (n / blockDim.x) * blockDim.x;

// synchronize threads
__syncthreads();

// last bit
if ( threadIdx.x < leftover ) {
    // load rj into shared memory
    xj[threadIdx.x] = x[j + threadIdx.x];
    yj[threadIdx.x] = y[j + threadIdx.x];
    zj[threadIdx.x] = z[j + threadIdx.x];
}

// synchronize threads
__syncthreads();

for (int myj = 0; myj < leftover; myj++) {

    double dx  = xi - xj[myj];
    double dy  = yi - yj[myj];
    double dz  = zi - zj[myj];
```

code looks the same, just different chunk size

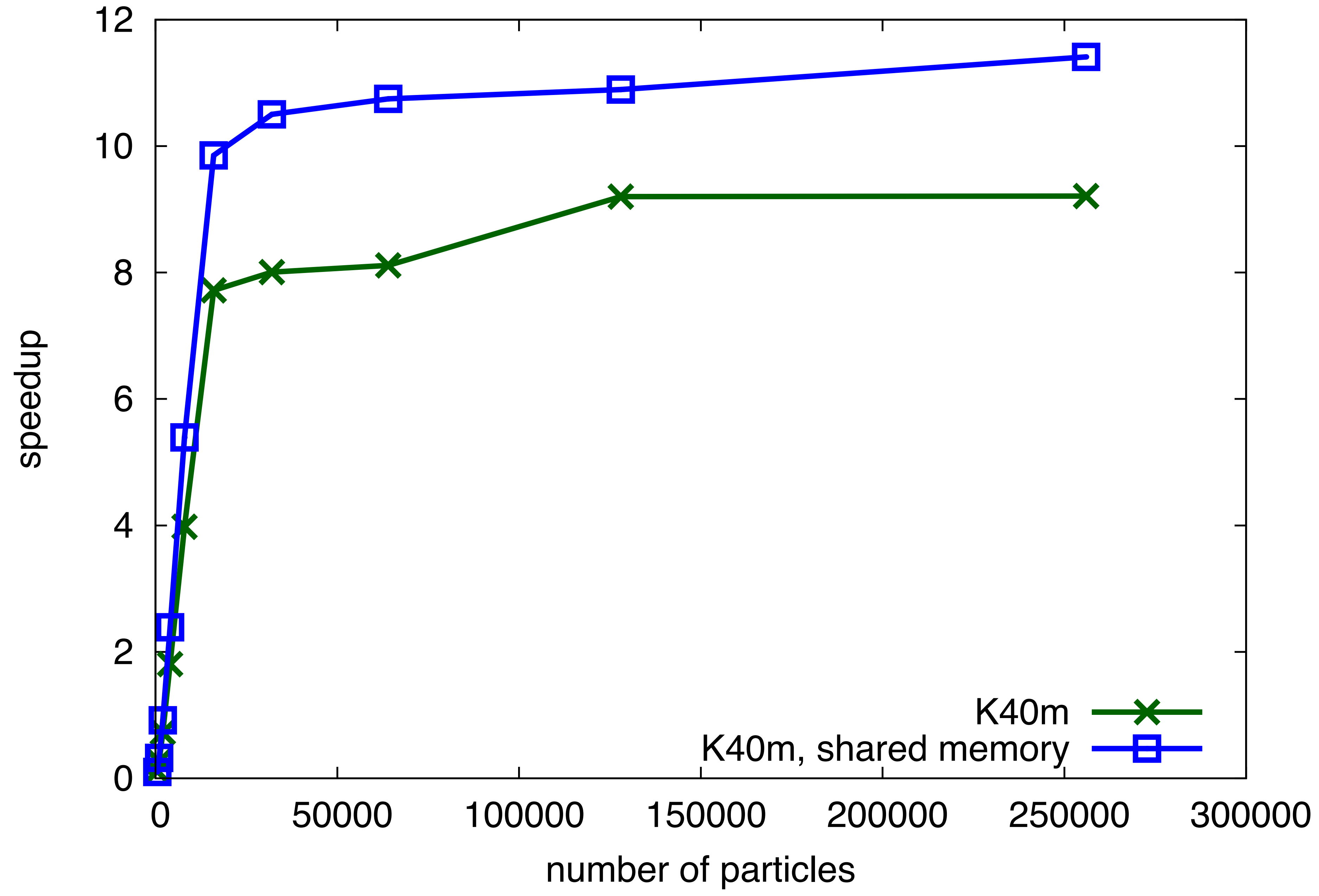
shared memory

so, how'd we do?



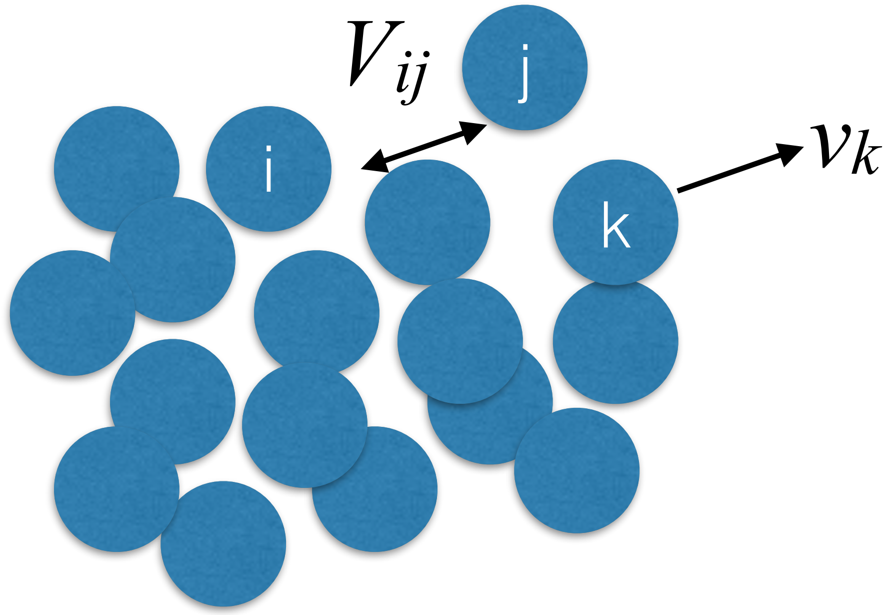
shared memory

so, how'd we do?



for this problem, on the K40m, using shared memory in this way gets us a 30% boost in performance. On m2050, boost is much more modest.

molecular dynamics



n lennard-jones particles moving in
3-d with periodic boundary conditions

velocity verlet (https://en.wikipedia.org/wiki/Verlet_integration#Velocity_Verlet)

start with some initial positions, velocities

$$x(t+dt) = x(t) + v_x(t)dt + \frac{1}{2}a_x(t)dt^2$$ ← just vector addition!

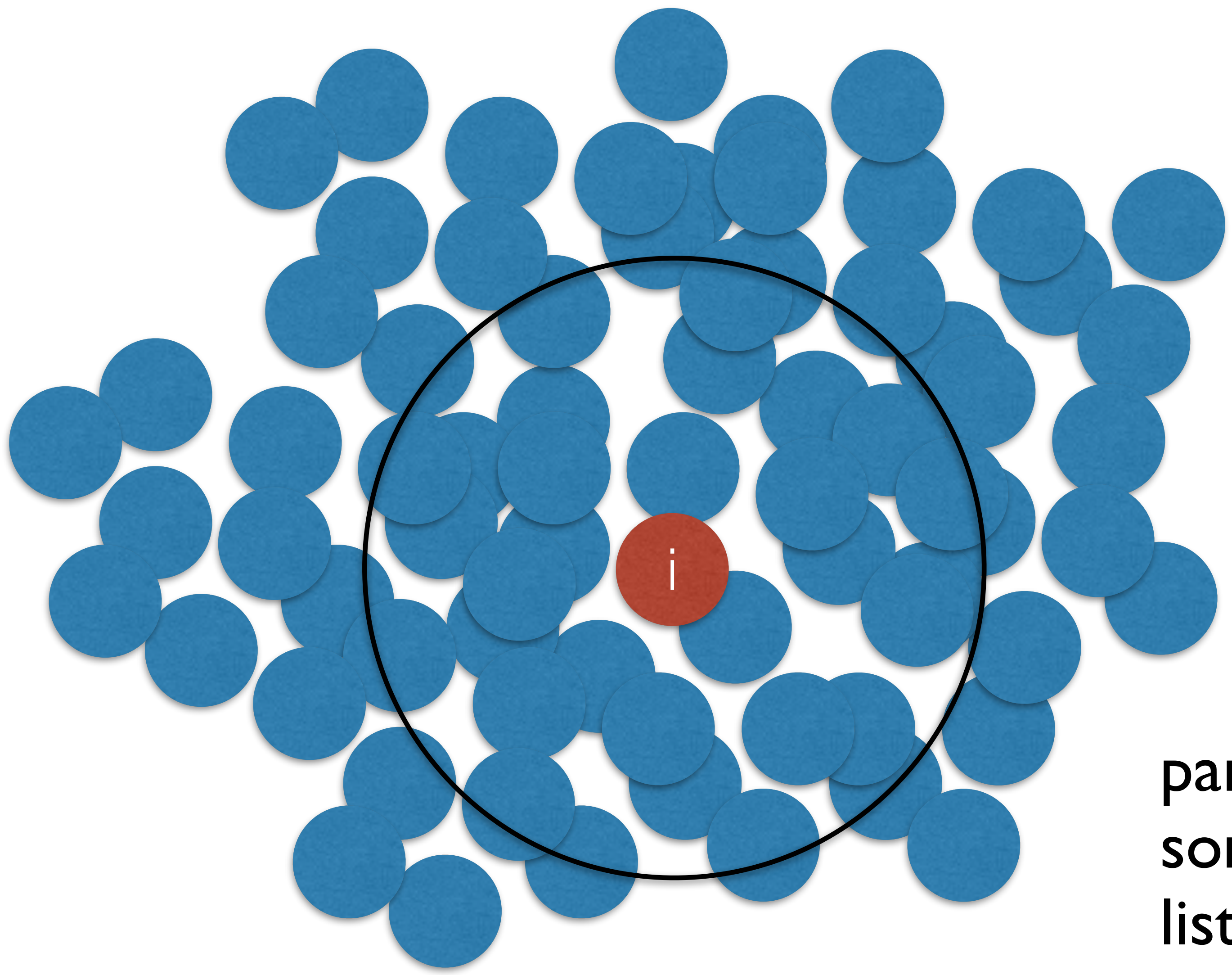
$a(t+dt)$ determined from lennard-jones forces
(we just did this!)

$$v(t+dt) = v(t) + \frac{1}{2}[a(t) + a(t+dt))dt$$

molecular dynamics - extra details

catch - acceleration naively scales as n^2

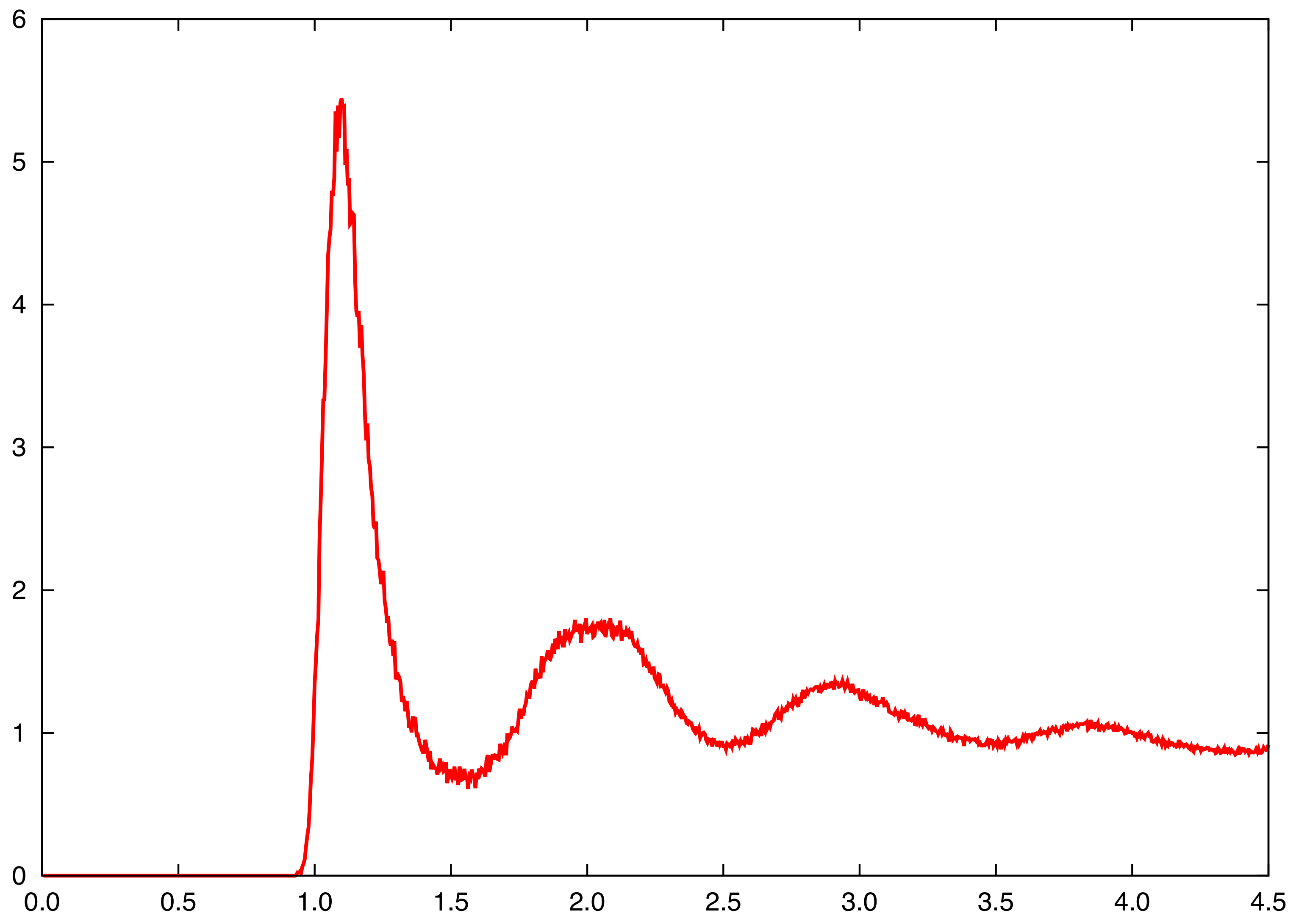
solution - use neighbor lists:



particle i only sees particles j within some radius. computing the neighbor list is expensive, so we only update it every once in a while

molecular dynamics - extra details

analysis: pair correlation function

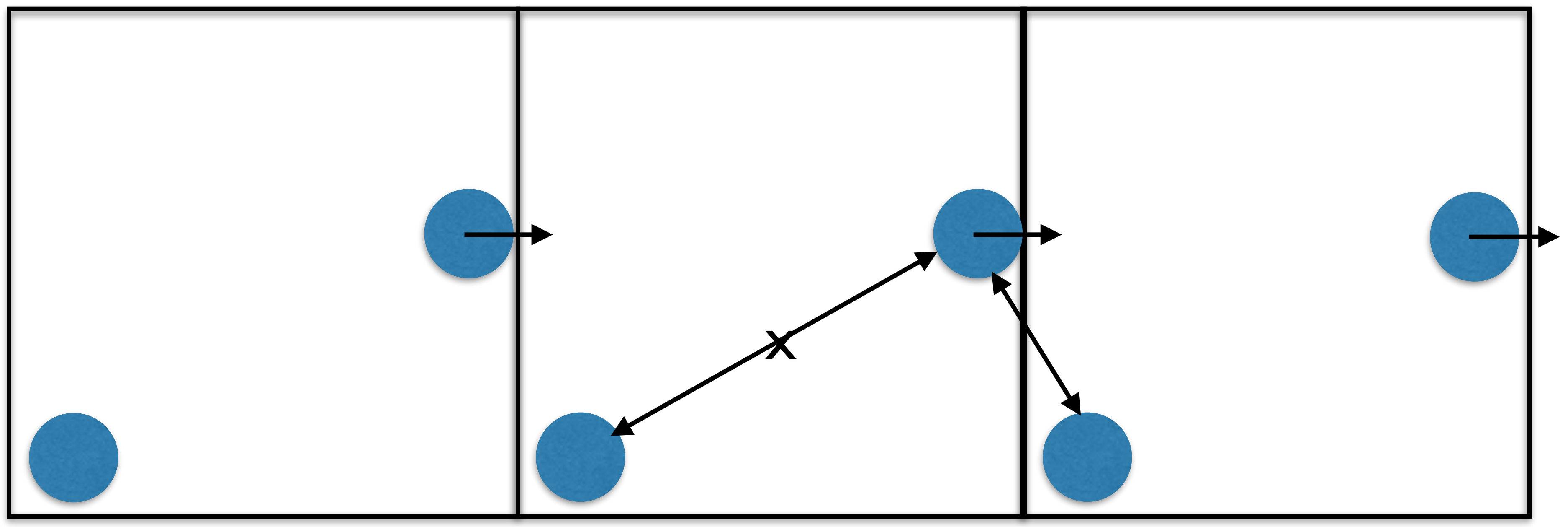


basically, check distances between all pairs of particles and bin them

molecular dynamics - extra details

periodic boundary conditions and minimum image convention

if particle leaves box, put it back in on other side



particles interact with only one "image" of other particles
the one that is closest to them

molecular dynamics

cpu code

S212/gpu/md_simulation/cpu/md_cpu.cc

gpu code

S212/gpu/md_simulation/gpu/md_gpu.cc

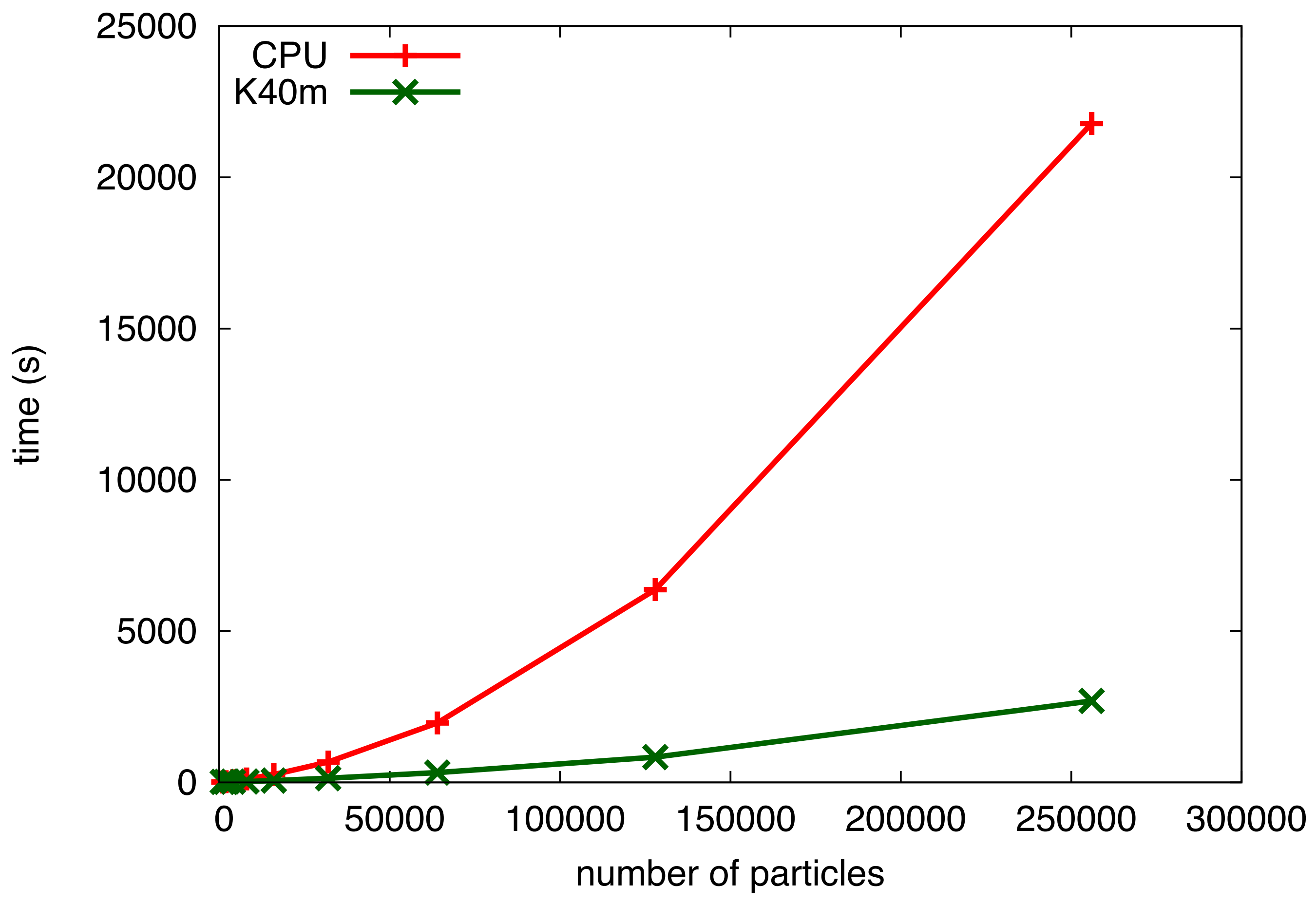
start modifying the CPU code to run on GPUs … and … go!

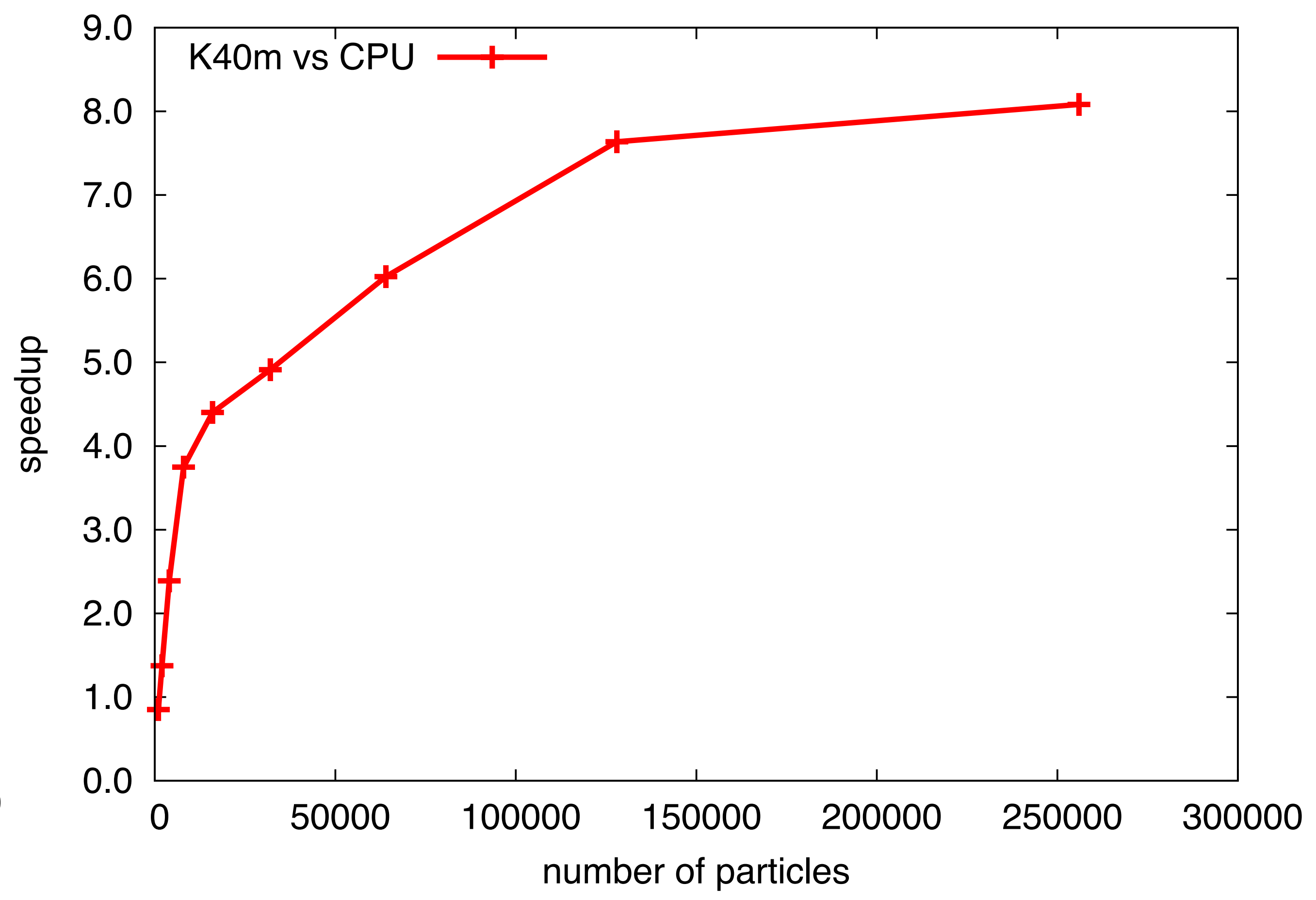
molecular dynamics

how did we do?

blue ridge: 2 x 8-core Intel Sandy bridge CPU vs NVIDIA K40m GPU

timings are for 10000 time steps



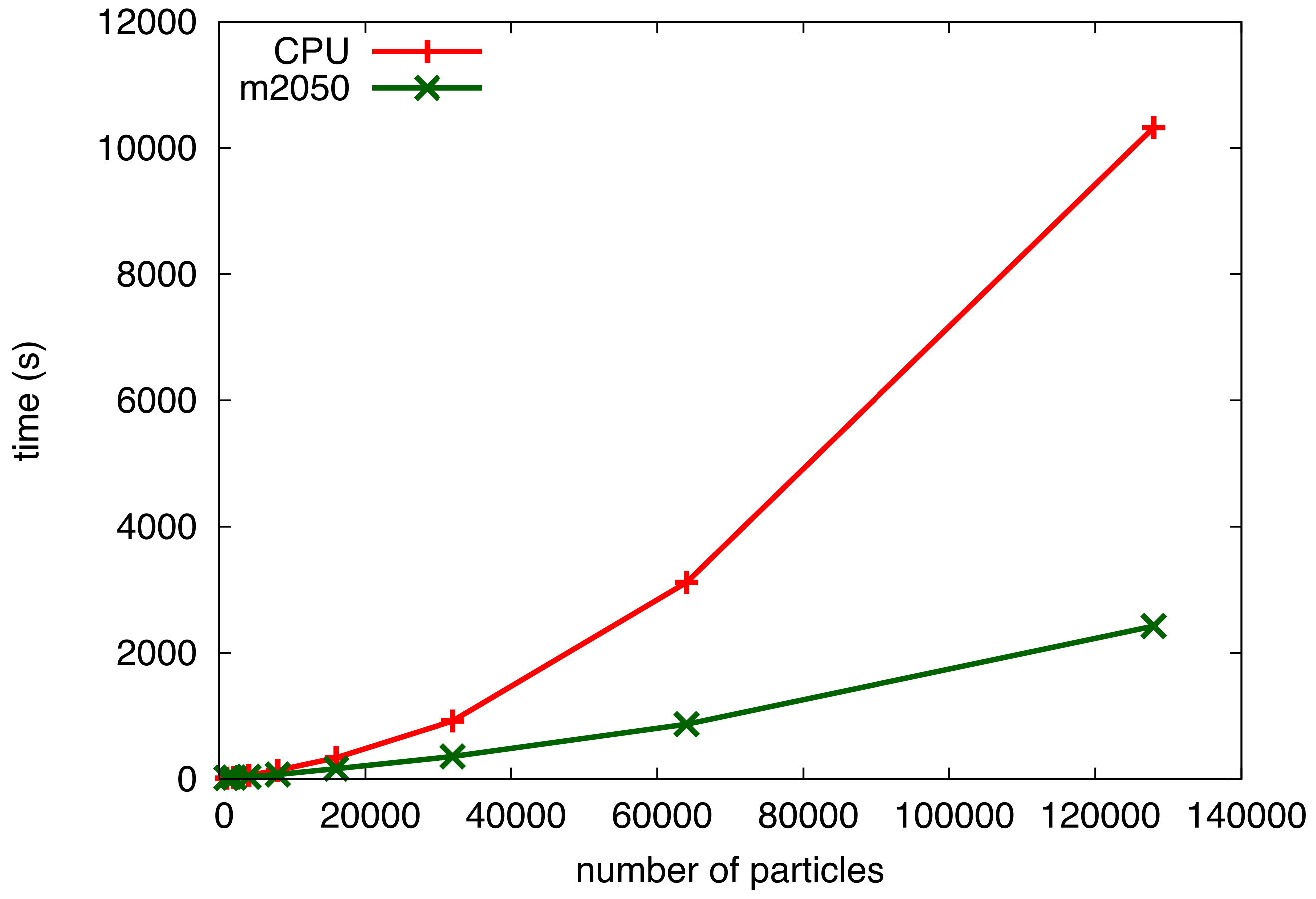


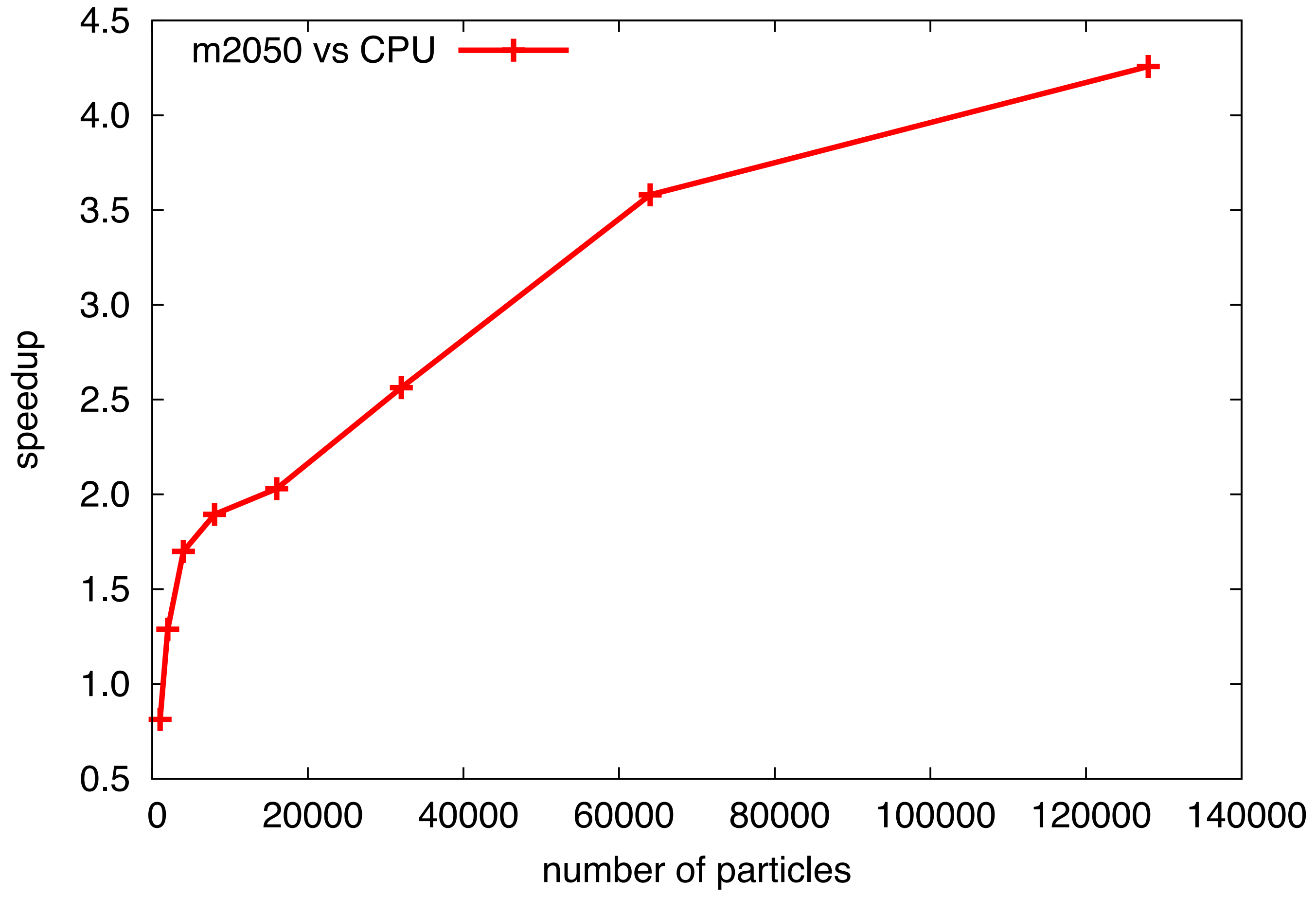
molecular dynamics

how did we do?

hokie speed: 2 x 6-core Intel Xeon CPU vs NVIDIA m2050 GPU

timings are for 10000 time steps


CPU
m2050
time (s)
number of particles


m2050 vs CPU
speedup
number of particles