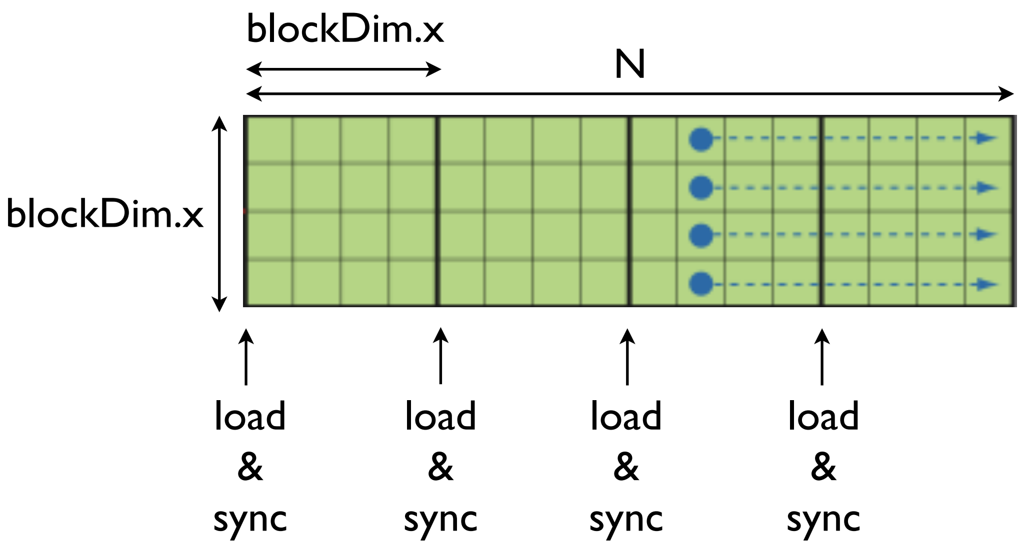
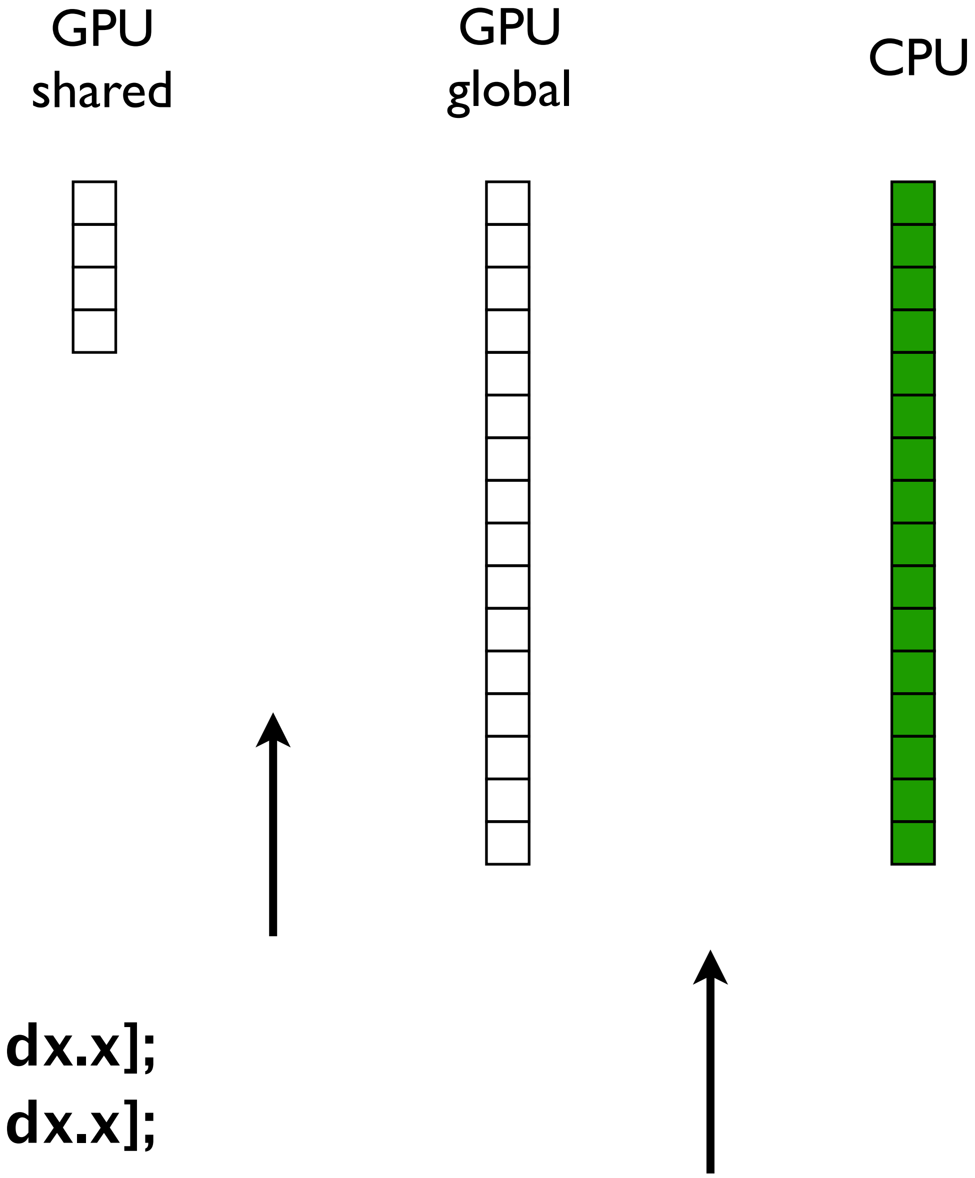
12 Steps to a Fast Multipole Method on GPUs

Rio Yokota
Boston
University

GPUs :: Shared memory

blockDim.x :Threads per thread block



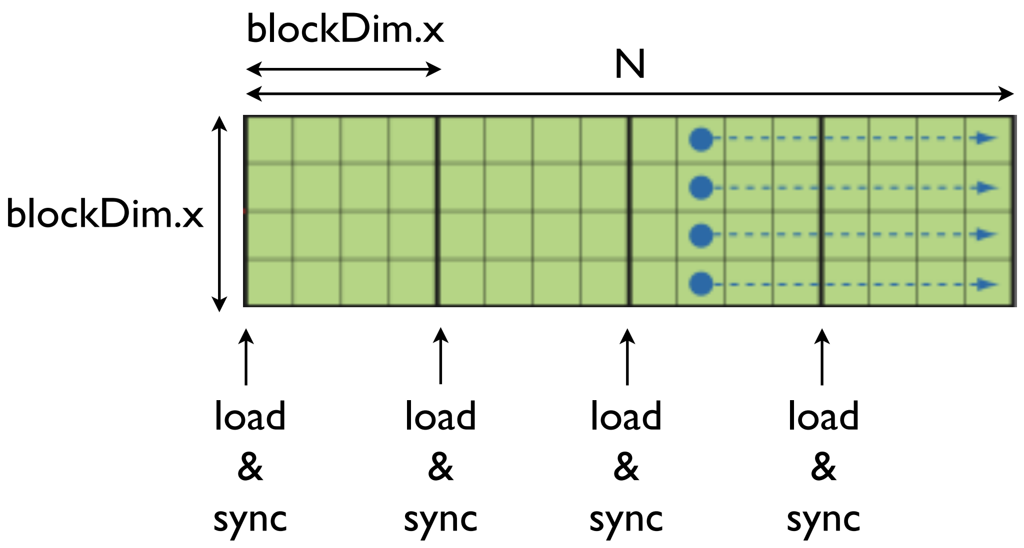


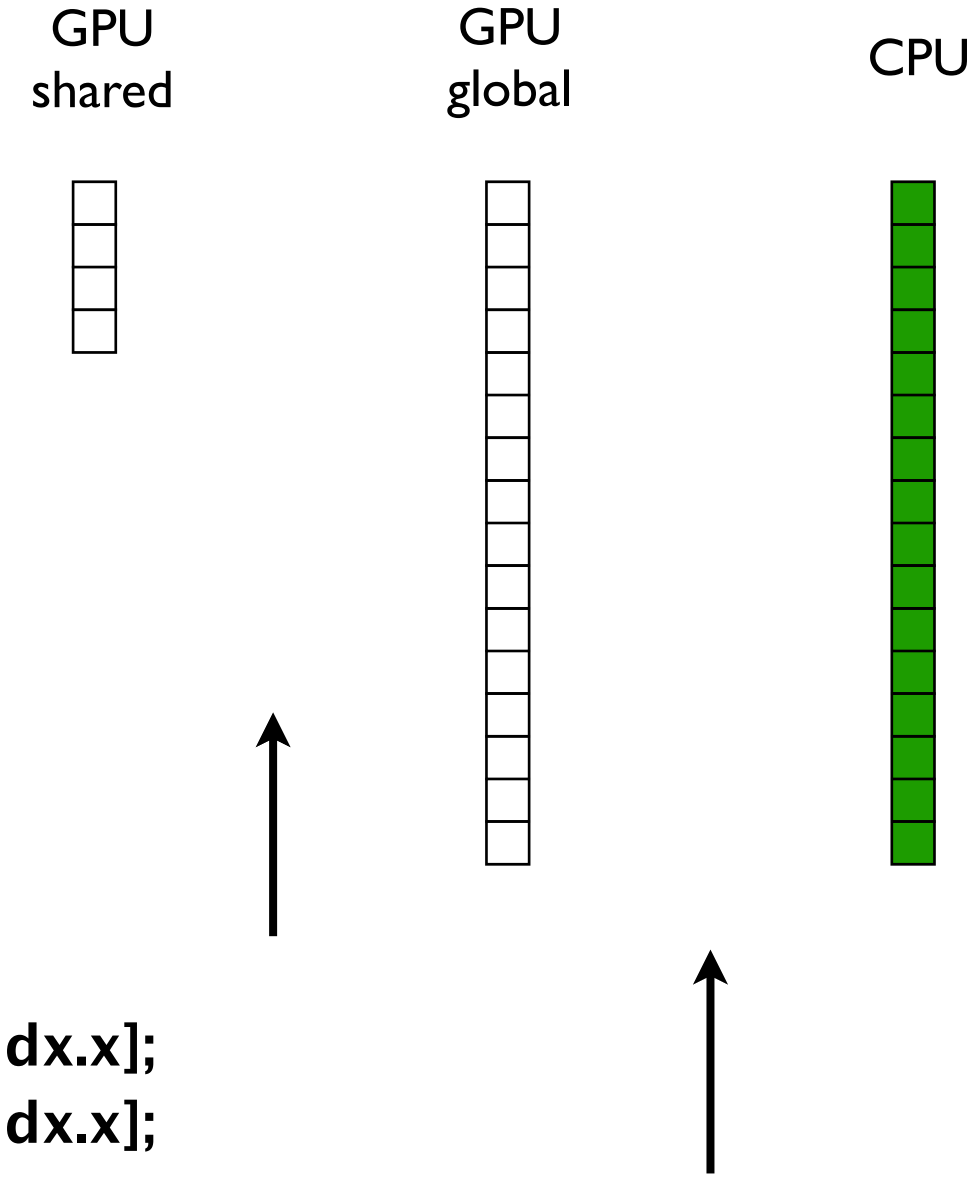
```
shared[threadIdx.x] = global[blockIdx.x * blockDim.x + threadIdx.x];
global[blockIdx.x * blockDim.x + threadIdx.x] = shared[threadIdx.x];
```

```
cudaMemcpy(device,host,size,cudaMemcpyHostToDevice);
cudaMemcpy(host,device,size,cudaMemcpyDeviceToHost);
```

GPUs :: Shared memory

blockDim.x : Threads per thread block



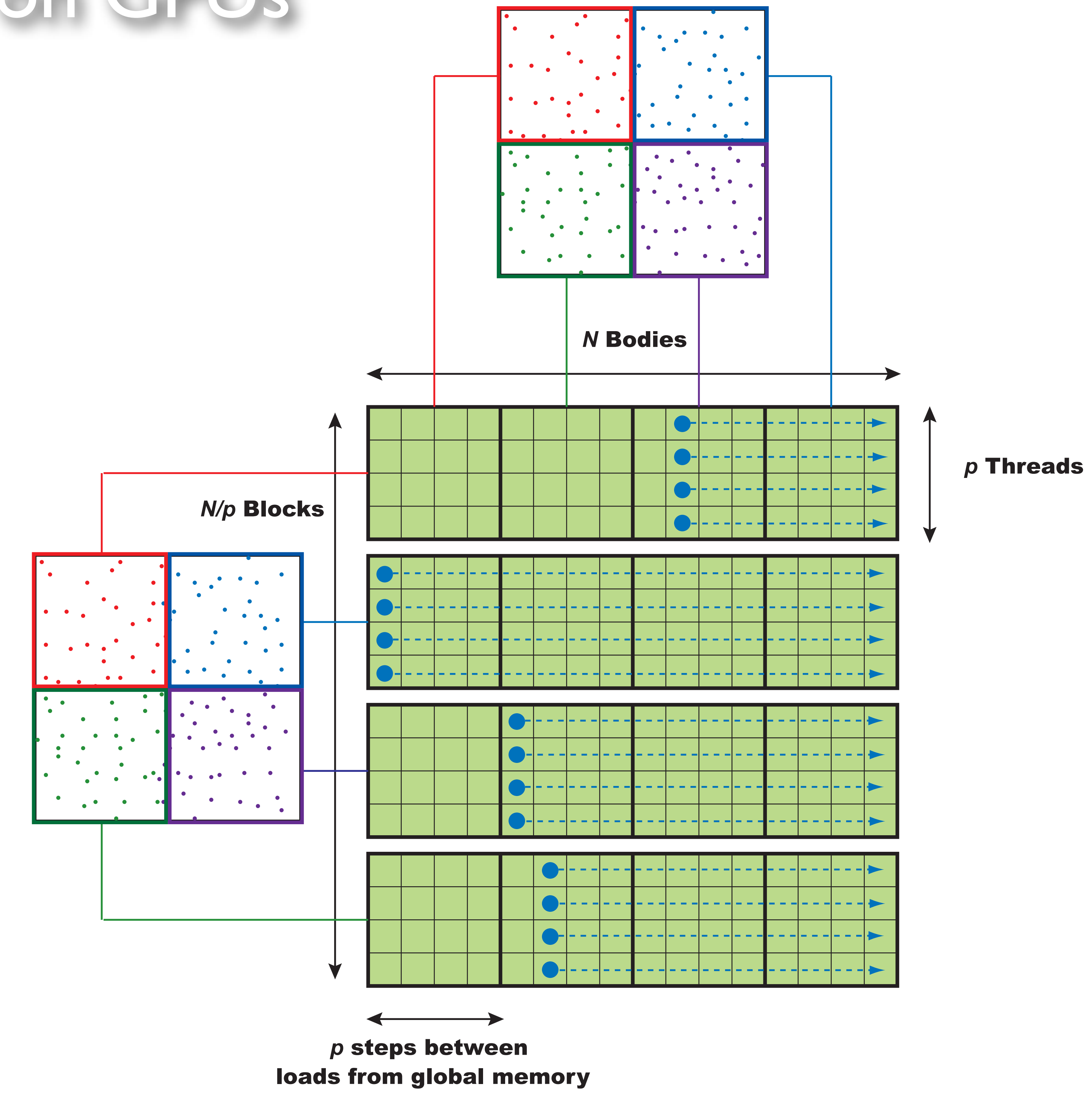


shared[threadIdx.x] = global[blockIdx.x * blockDim.x + threadIdx.x];
global[blockIdx.x * blockDim.x + threadIdx.x] = shared[threadIdx.x];

cudaMemcpy(device,host,size,cudaMemcpyHostToDevice);
cudaMemcpy(host,device,size,cudaMemcpyDeviceToHost);

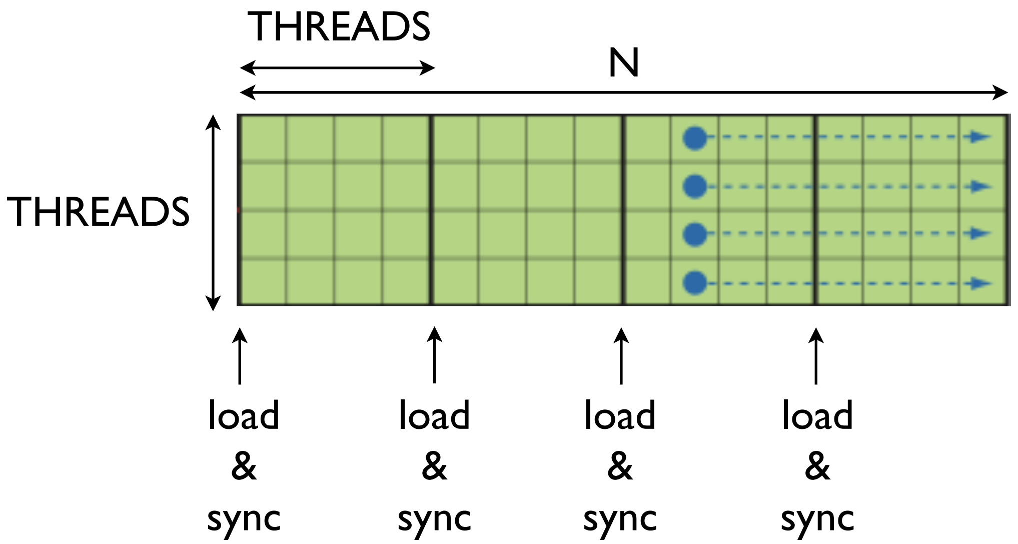
GPUs :: direct summation on GPUs

Direct Summation

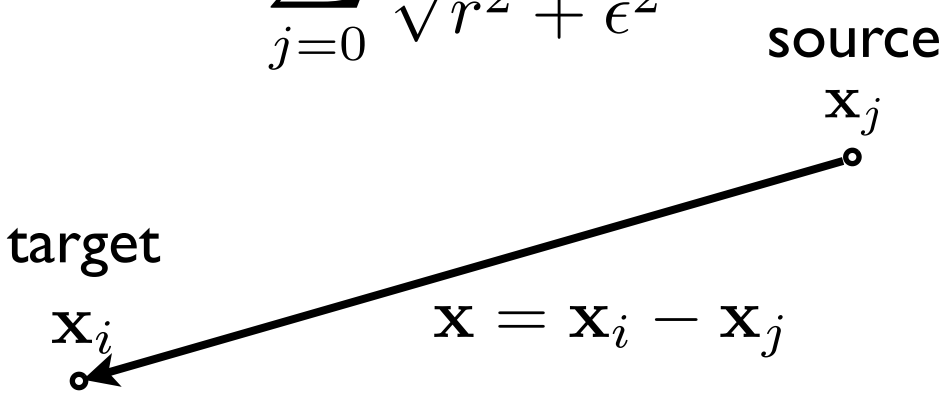


Step08. direct summation on GPUs

```
// Direct summation on host
  float dx,dy,dz,r;
  for( int i=0; i<N; i++ ) {
    float p = -= sourceHost[i].w / sqrtf(EPS2);
    for( int j=0; j<N; j++ ) {
      dx = sourceHost[i].x - sourceHost[j].x;
      dy = sourceHost[i].y - sourceHost[j].y;
      dz = sourceHost[i].z - sourceHost[j].z;
      r = sqrtf(dx * dx + dy * dy + dz * dz + EPS2);
      p += sourceHost[j].w / r;
    }
  }
```



$$\Phi_i = \sum_{j=0}^{N} \frac{m_j}{r} = \sum_{j=0}^{N} \frac{m_j}{\sqrt{r^2+\epsilon^2}}$$



```
// Direct summation on device
__global__ void direct(float4 *sourceGlob, float *targetGlob) {
  float3 d;
  __shared__ float4 sourceShrd[THREADS];
  float4 target = sourceGlob[blockIdx.x * THREADS + threadIdx.x];
  target.w *= -rsqrtf(EPS2);
  for( int iblok=0; iblok<N/THREADS; iblok++) {
    __syncthreads();
    sourceShrd[threadIdx.x] = sourceGlob[iblok * THREADS + threadIdx.x];
    __syncthreads();
    for( int i=0; i<THREADS; i++ ) {
      d.x = target.x - sourceShrd[i].x;
      d.y = target.y - sourceShrd[i].y;
      d.z = target.z - sourceShrd[i].z;
      target.w += sourceShrd[i].w * rsqrtf(d.x * d.x + d.y * d.y + d.z * d.z + EPS2);
    }
  }
  targetGlob[blockIdx.x * THREADS + threadIdx.x] = target.w;
}
```

Step08. direct summation on GPUs

```
int main() {
  float4 *sourceHost,*sourceDevc;
  float  *targetHost,*targetDevc;
// Allocate memory on host and device
  sourceHost = (float4*)     malloc( N*sizeof(float4) );
  targetHost = (float *)     malloc( N*sizeof(float ) );
  cudaMalloc(  (void**) &sourceDevc, N*sizeof(float4) );
  cudaMalloc(  (void**) &targetDevc, N*sizeof(float ) );
// Initialize
  for( int i=0; i<N; i++ ) {
    sourceHost[i].x = rand()/(1.+RAND_MAX);
    sourceHost[i].y = rand()/(1.+RAND_MAX);
    sourceHost[i].z = rand()/(1.+RAND_MAX);
    sourceHost[i].w = 1.0/N;
  }
// Direct summation on device
  cudaMemcpy(sourceDevc,sourceHost,N*sizeof(float4),cudaMemcpyHostToDevice);
  direct<<< N/THREADS, THREADS >>>(sourceDevc,targetDevc);
  cudaMemcpy(targetHost,targetDevc,N*sizeof(float ),cudaMemcpyDeviceToHost);
}
```

allocate on host

allocate on device

initial condition

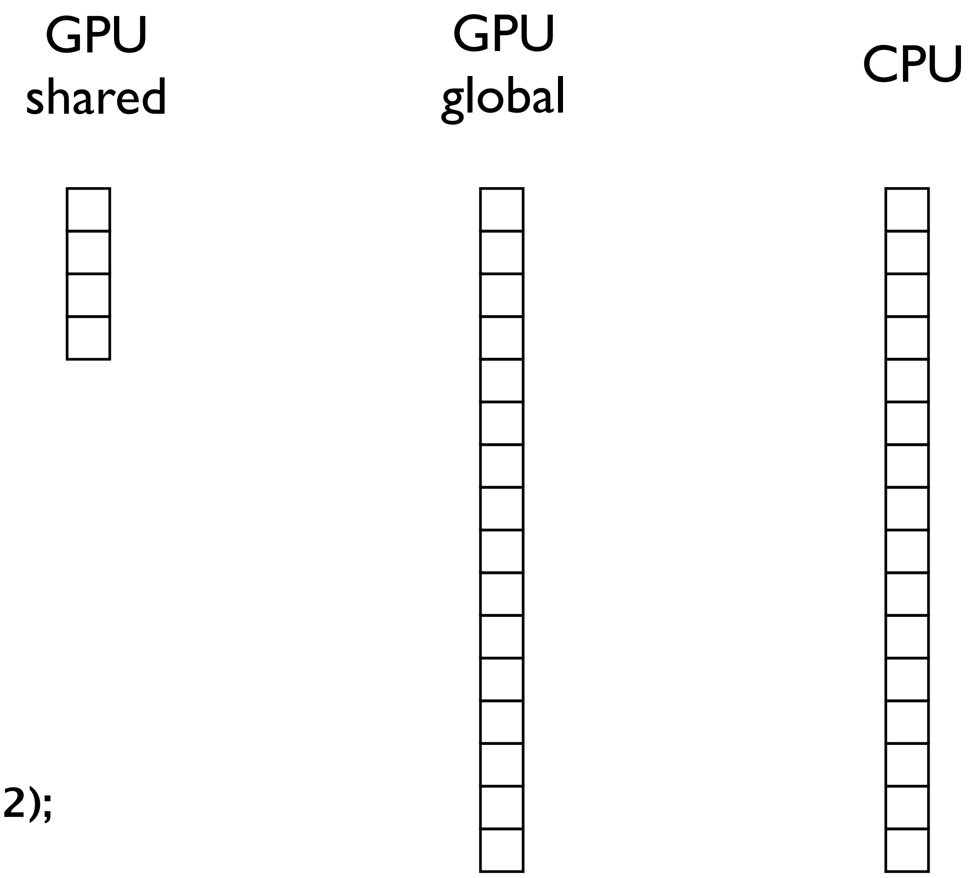
copy to device

CUDA kernel

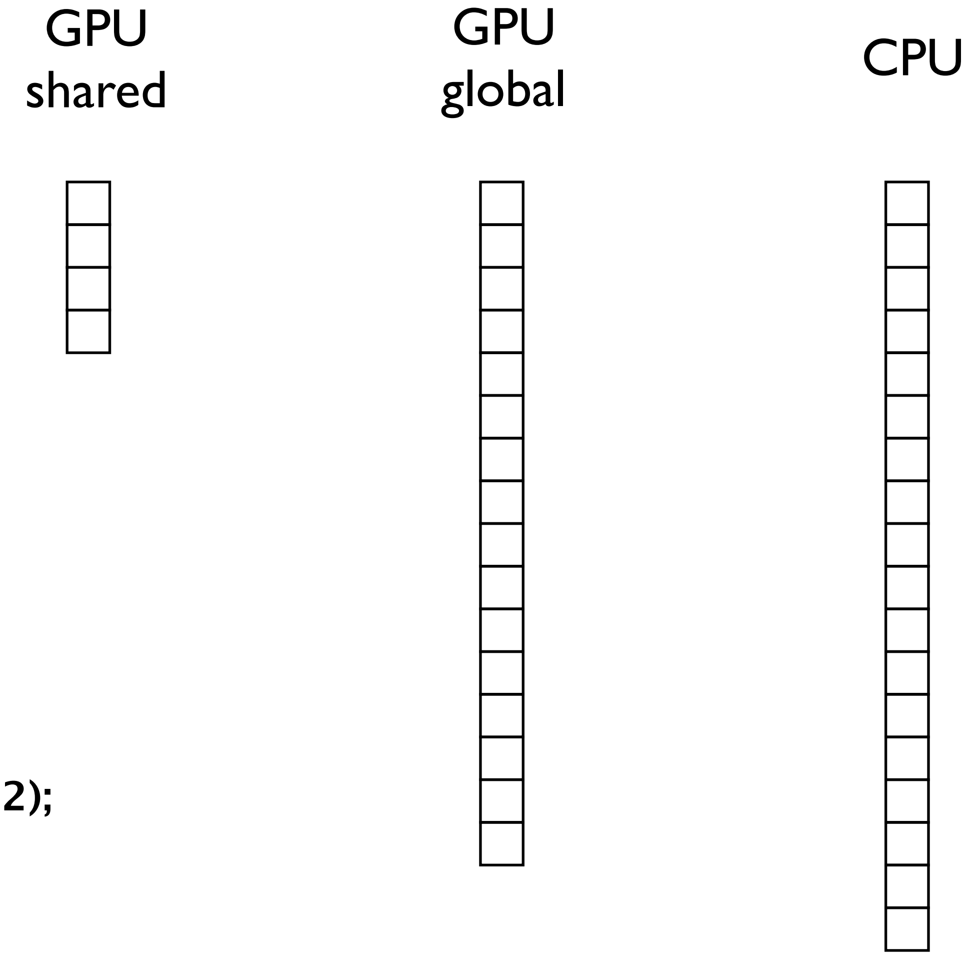
copy to back

Step09. odd number of threads

```
__global__ void direct(float4 *sourceGlob, float *targetGlob) {
  float3 d;
  __shared__ float4 sourceShrd[THREADS];
  float4 target = sourceGlob[blockIdx.x * THREADS + threadIdx.x];
  target.w *= -rsqrtf(EPS2);
  for( int iblok=0; iblok<(N-1)/THREADS; iblok++) {
    __syncthreads();
    sourceShrd[threadIdx.x] = sourceGlob[iblok * THREADS + threadIdx.x];
    __syncthreads();
    for( int i=0; i<THREADS; i++ ) {
      d.x = target.x - sourceShrd[i].x;
      d.y = target.y - sourceShrd[i].y;
      d.z = target.z - sourceShrd[i].z;
      target.w += sourceShrd[i].w * rsqrtf(d.x * d.x + d.y * d.y + d.z * d.z + EPS2);
    }
  }
  int iblok = (N-1)/THREADS;
  __syncthreads();
  sourceShrd[threadIdx.x] = sourceGlob[iblok * THREADS + threadIdx.x];
  __syncthreads();
  for( int i=0; i<N - (iblok * THREADS); i++ ) {
    d.x = target.x - sourceShrd[i].x;
    d.y = target.y - sourceShrd[i].y;
    d.z = target.z - sourceShrd[i].z;
    target.w += sourceShrd[i].w * rsqrtf(d.x * d.x + d.y * d.y + d.z * d.z + EPS2);
  }
  targetGlob[blockIdx.x * THREADS + threadIdx.x] = target.w;
}
```



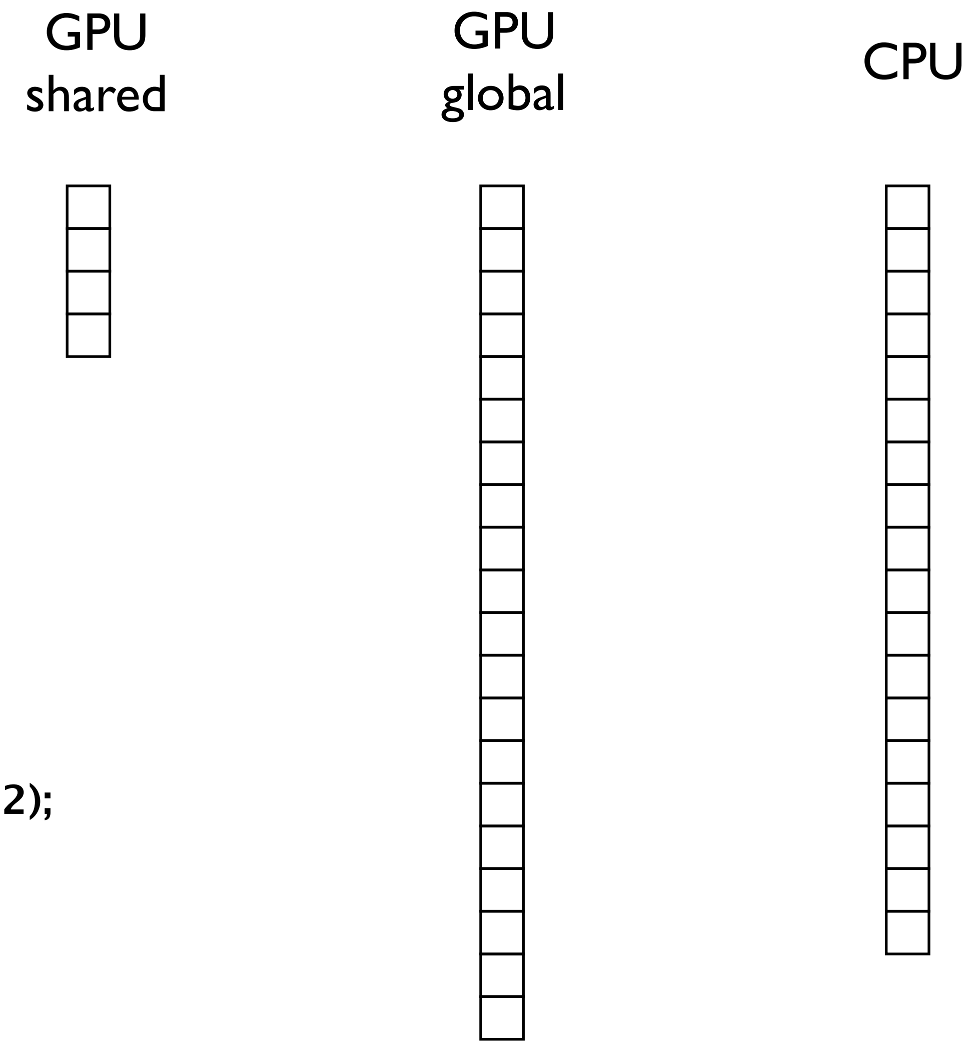
Step09. odd number of threads



```
__global__ void direct(float4 *sourceGlob, float *targetGlob) {
  float3 d;
  __shared__ float4 sourceShrd[THREADS];
  float4 target = sourceGlob[blockIdx.x * THREADS + threadIdx.x];
  target.w *= -rsqrtf(EPS2);
  for( int iblok=0; iblok<(N-1)/THREADS; iblok++) {
    __syncthreads();
    sourceShrd[threadIdx.x] = sourceGlob[iblok * THREADS + threadIdx.x];
    __syncthreads();
    for( int i=0; i<THREADS; i++ ) {
      d.x = target.x - sourceShrd[i].x;
      d.y = target.y - sourceShrd[i].y;
      d.z = target.z - sourceShrd[i].z;
      target.w += sourceShrd[i].w * rsqrtf(d.x * d.x + d.y * d.y + d.z * d.z + EPS2);
    }
  }
  int iblok = (N-1)/THREADS;
  __syncthreads();
  sourceShrd[threadIdx.x] = sourceGlob[iblok * THREADS + threadIdx.x];
  __syncthreads();
  for( int i=0; i<N - (iblok * THREADS); i++ ) {
    d.x = target.x - sourceShrd[i].x;
    d.y = target.y - sourceShrd[i].y;
    d.z = target.z - sourceShrd[i].z;
    target.w += sourceShrd[i].w * rsqrtf(d.x * d.x + d.y * d.y + d.z * d.z + EPS2);
  }
  targetGlob[blockIdx.x * THREADS + threadIdx.x] = target.w;
}
```

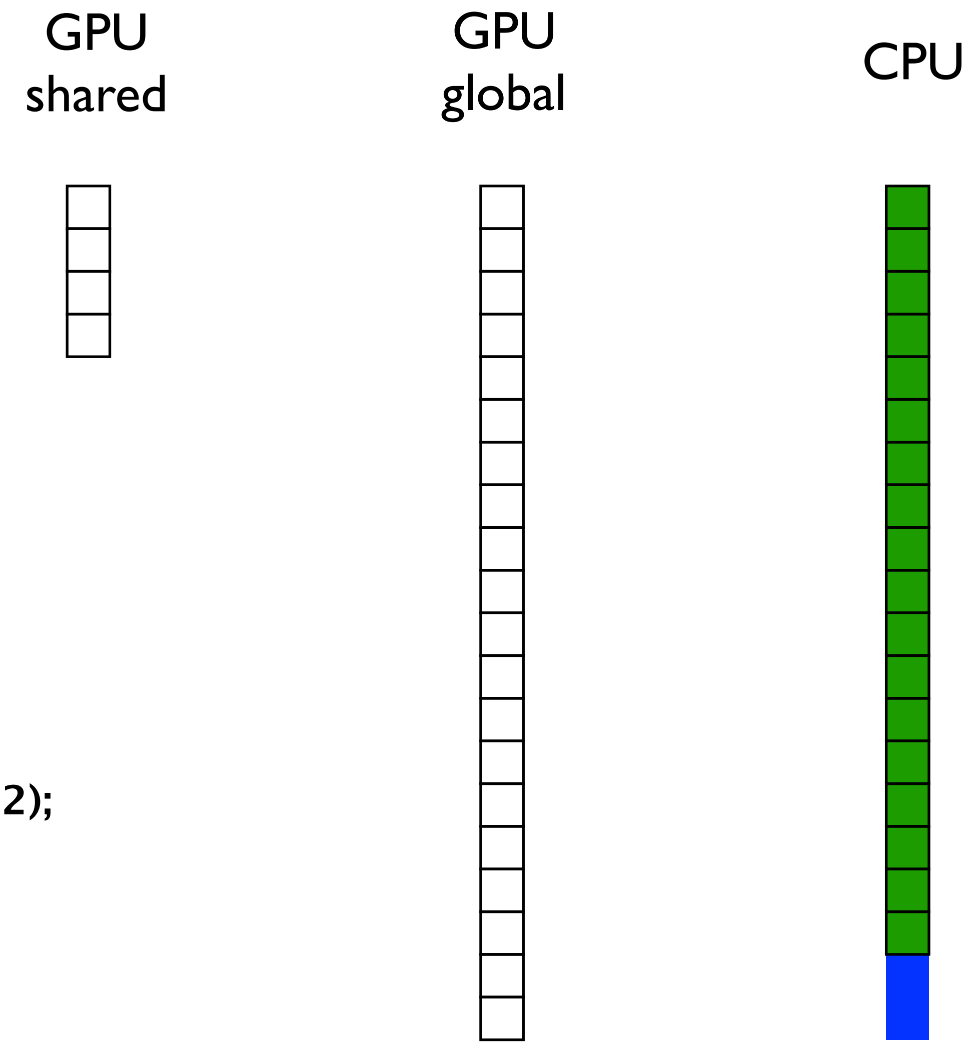
Step09. odd number of threads

```
__global__ void direct(float4 *sourceGlob, float *targetGlob) {
  float3 d;
  __shared__ float4 sourceShrd[THREADS];
  float4 target = sourceGlob[blockIdx.x * THREADS + threadIdx.x];
  target.w *= -rsqrtf(EPS2);
  for( int iblok=0; iblok<(N-1)/THREADS; iblok++) {
    __syncthreads();
    sourceShrd[threadIdx.x] = sourceGlob[iblok * THREADS + threadIdx.x];
    __syncthreads();
    for( int i=0; i<THREADS; i++ ) {
      d.x = target.x - sourceShrd[i].x;
      d.y = target.y - sourceShrd[i].y;
      d.z = target.z - sourceShrd[i].z;
      target.w += sourceShrd[i].w * rsqrtf(d.x * d.x + d.y * d.y + d.z * d.z + EPS2);
    }
  }
  int iblok = (N-1)/THREADS;
  __syncthreads();
  sourceShrd[threadIdx.x] = sourceGlob[iblok * THREADS + threadIdx.x];
  __syncthreads();
  for( int i=0; i<N - (iblok * THREADS); i++ ) {
    d.x = target.x - sourceShrd[i].x;
    d.y = target.y - sourceShrd[i].y;
    d.z = target.z - sourceShrd[i].z;
    target.w += sourceShrd[i].w * rsqrtf(d.x * d.x + d.y * d.y + d.z * d.z + EPS2);
  }
  targetGlob[blockIdx.x * THREADS + threadIdx.x] = target.w;
}
```



Step09. odd number of threads

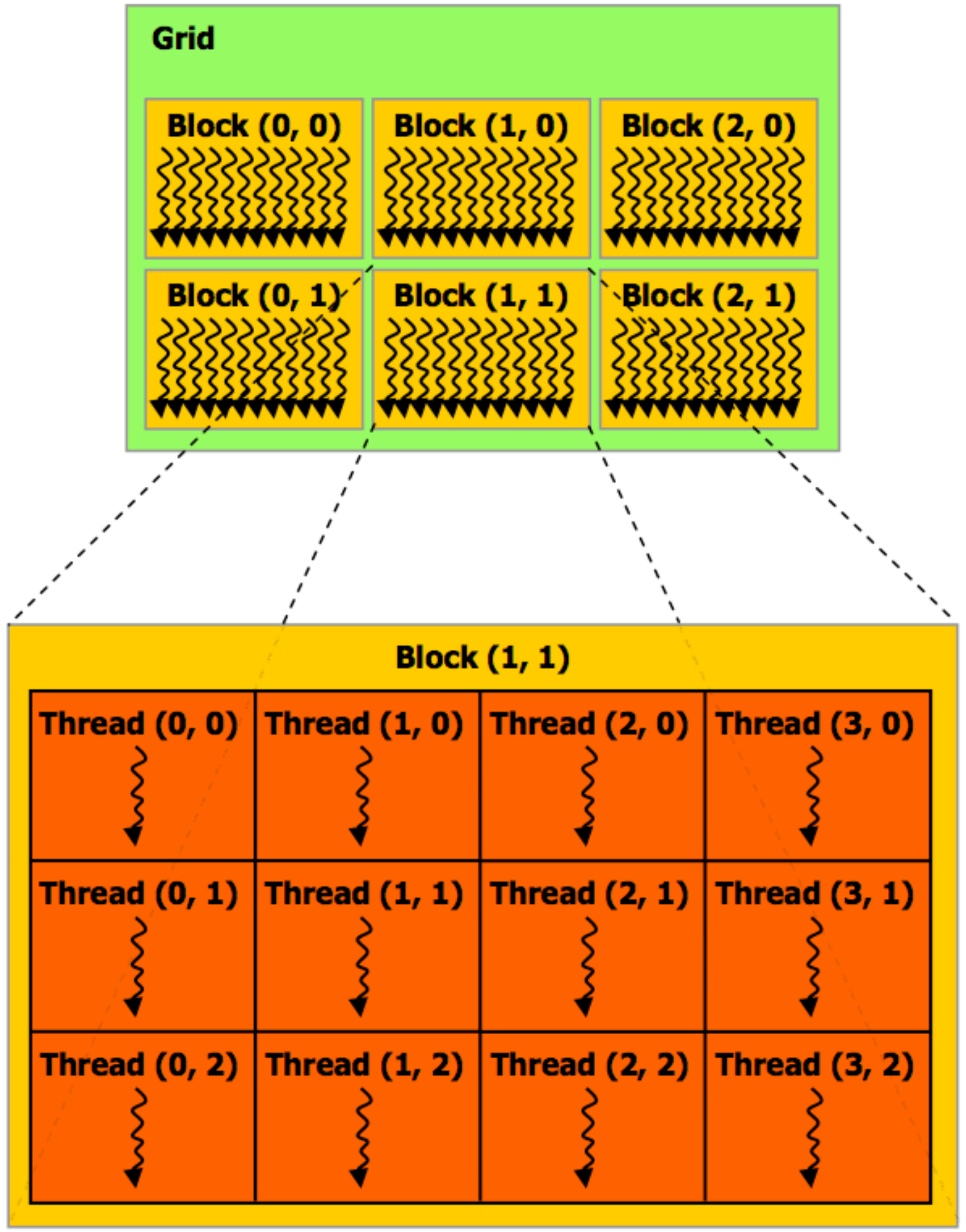
```
__global__ void direct(float4 *sourceGlob, float *targetGlob) {
  float3 d;
  __shared__ float4 sourceShrd[THREADS];
  float4 target = sourceGlob[blockIdx.x * THREADS + threadIdx.x];
  target.w *= -rsqrtf(EPS2);
  for( int iblok=0; iblok<(N-1)/THREADS; iblok++) {
    __syncthreads();
    sourceShrd[threadIdx.x] = sourceGlob[iblok * THREADS + threadIdx.x];
    __syncthreads();
    for( int i=0; i<THREADS; i++ ) {
      d.x = target.x - sourceShrd[i].x;
      d.y = target.y - sourceShrd[i].y;
      d.z = target.z - sourceShrd[i].z;
      target.w += sourceShrd[i].w * rsqrtf(d.x * d.x + d.y * d.y + d.z * d.z + EPS2);
    }
  }
  int iblok = (N-1)/THREADS;
  __syncthreads();
  sourceShrd[threadIdx.x] = sourceGlob[iblok * THREADS + threadIdx.x];
  __syncthreads();
  for( int i=0; i<N - (iblok * THREADS); i++ ) {
    d.x = target.x - sourceShrd[i].x;
    d.y = target.y - sourceShrd[i].y;
    d.z = target.z - sourceShrd[i].z;
    target.w += sourceShrd[i].w * rsqrtf(d.x * d.x + d.y * d.y + d.z * d.z + EPS2);
  }
  targetGlob[blockIdx.x * THREADS + threadIdx.x] = target.w;
}
```

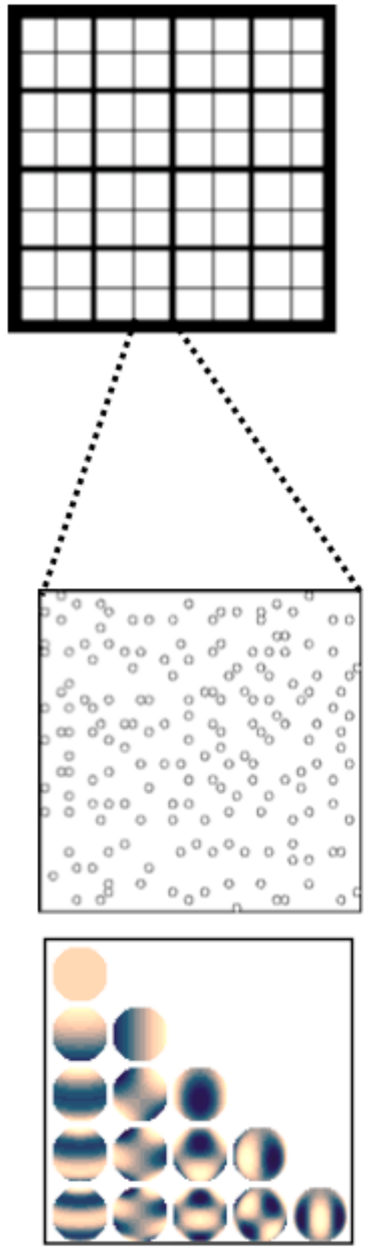


Step10. multipole expansion on GPUs

```
__global__ void multipole(float4 *targetGlob, float *multipGlob) {
  float R,R3,R5;
  float3 d;
  float4 target = targetGlob[blockIdx.x * THREADS + threadIdx.x];
  target.w = 0;
  d.x = target.x – multipGlob[0];
  d.y = target.y – multipGlob[1];
  d.z = target.z – multipGlob[2];
  R = sqrtf(d.x * d.x + d.y * d.y + d.z * d.z);
  R3 = R * R * R;
  R5 = R3 * R * R;
  target.w += multipGlob[ 3] / R;
  target.w += multipGlob[ 4] * (–d.x / R3);
  target.w += multipGlob[ 5] * (–d.y / R3);
  target.w += multipGlob[ 6] * (–d.z / R3);
  target.w += multipGlob[ 7] * (3 * d.x * d.x / R5 – 1 / R3);
  target.w += multipGlob[ 8] * (3 * d.y * d.y / R5 – 1 / R3);
  target.w += multipGlob[ 9] * (3 * d.z * d.z / R5 – 1 / R3);
  target.w += multipGlob[10] * (3 * d.x * d.y / R5);
  target.w += multipGlob[11] * (3 * d.y * d.z / R5);
  target.w += multipGlob[12] * (3 * d.z * d.x / R5);
  targetGlob[blockIdx.x * THREADS + threadIdx.x] = target;
}
```

GPUs :: mapping data





Grid = FMM Domain

Thread Block = FMM SubBox

Thread = Particle

Thread = Multipole

Step11. treecode on GPUs part 1

```
__global__ void direct(int *offsetGlob, float4 *sourceGlob, float4 *targetGlob) {
  int N = offsetGlob[blockIdx.x+1] - offsetGlob[blockIdx.x];
  int offset = offsetGlob[blockIdx.x];
  float3 d;
  __shared__ float4 sourceShrd[THREADS];
  float4 target = targetGlob[blockIdx.x * THREADS + threadIdx.x];
  target.w *= -rsqrtf(EPS2);
  for( int iblok=0; iblok<(N-1)/THREADS; iblok++) {
    __syncthreads();
    sourceShrd[threadIdx.x] = sourceGlob[offset + iblok * THREADS + threadIdx.x];
    __syncthreads();
    for( int i=0; i<THREADS; i++ ) {
      d.x = target.x - sourceShrd[i].x;
      d.y = target.y - sourceShrd[i].y;
      d.z = target.z - sourceShrd[i].z;
      target.w += sourceShrd[i].w * rsqrtf(d.x * d.x + d.y * d.y + d.z * d.z + EPS2);
    }
  }
  int iblok = (N-1)/THREADS;
  __syncthreads();
  sourceShrd[threadIdx.x] = sourceGlob[offset + iblok * THREADS + threadIdx.x];
  __syncthreads();
  for( int i=0; i<N - (iblok * THREADS); i++ ) {
    d.x = target.x - sourceShrd[i].x;
    d.y = target.y - sourceShrd[i].y;
    d.z = target.z - sourceShrd[i].z;
    target.w += sourceShrd[i].w * rsqrtf(d.x * d.x + d.y * d.y + d.z * d.z + EPS2);
  }
  targetGlob[blockIdx.x * THREADS + threadIdx.x] = target;
}
```

Step12. treecode on GPUs part 2

```
__global__ void kernel(int *offSrcGlob, float4 *sourceGlob, int *offMtpGlob, float *multipGlob, float4
*targetGlob) {
  int N = offMtpGlob[blockIdx.x+1]-offMtpGlob[blockIdx.x];
  int offset = offMtpGlob[blockIdx.x];
  float3 d;
  __shared__ float  multipShrd[13*THREADS];
  for( int iblok=0; iblok<(N-1)/THREADS; iblok++) {
    int index = offset + iblok * THREADS + threadIdx.x;
    __syncthreads();
    for( int i=0; i<13; i++ )
      multipShrd[threadIdx.x*13+i] = multipGlob[index*13+i];
    __syncthreads();
    for( int i=0; i<THREADS; i++ ) {
      multipole(i,target,multipShrd);
    }
  }
  iblok = (N-1)/THREADS;
  int index = offset + iblok * THREADS + threadIdx.x;
  __syncthreads();
  for( int i=0; i<13; i++ )
    multipShrd[threadIdx.x*13+i] = multipGlob[index*13+i];
  __syncthreads();
  for( int i=0; i<N - (iblok * THREADS); i++ ) {
    multipole(i,target,multipShrd);
  }
  targetGlob[blockIdx.x * THREADS + threadIdx.x] = target;
}
```