CUDA Exercises

Karl Ljungkvist

25 February 2016

Example: PDE solver

Heat equation:

$$\begin{aligned} u_t &= u_{xx} + u_{yy} && \text{on } \Omega = [0,1] \times [0,1] \\ u &= 0 && \text{on } \partial\Omega \\ u(x,y,0) &= u_0(x,y) \end{aligned}$$

Discretization:

$$\frac{u_{i,j}^{n+1} - u_{i,j}^n}{k} = \frac{u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n - 4u_{i,j}^n}{h^2} \text{ for } i,j = 1,\ldots,N-1$$

Time stepping:

$$u_{i,j}^{n+1} = (1-4\lambda)\, u_{i,j}^n + \lambda \left(u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n \right), \; \lambda = \frac{k}{h^2}$$

Stop criterion:

```
if (max(u)-min(u)) < limit : stop
```

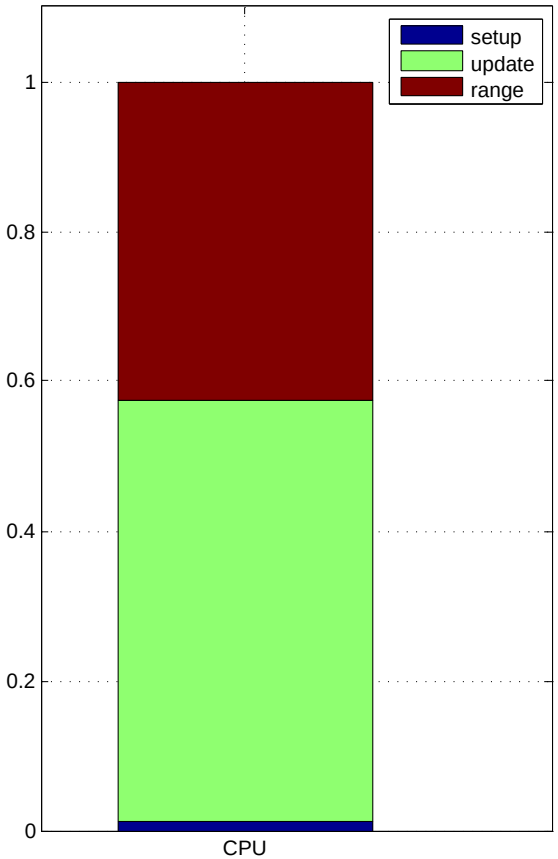
Example: PDE solver

Code:

```
void main() {
  float range = VERY_LARGE_NUMBER;
  float *u = malloc(N*N*sizeof(float));
  float *u_old = malloc(N*N*sizeof(float));

  /* setup initial condition */
  init_data(u,N);
  while(range > LIMIT) {
    /* swap pointers */
    swap(&u_old,&u);
    /* compute next time step */
    update(u,u_old,N);
    /* find range of solution */
    range = compute_range(u,N);
  }
}
```

```
void update(float *u, const float *u_old,
            int N)
{
  for(int i=1; i<N-1; i++) {
    for(int j=1; j<N-1; j++) {
      const float a = u_old[i*N+j];
      const float b = u_old[(i+1)*N+j];
      const float c = u_old[(i-1)*N+j];
      const float d = u_old[i*N+j+1];
      const float e = u_old[i*N+j-1];
      u[i*N+j] = (1-4*LAMBDA)*a +
                    LAMBDA*(b+c+d+e);
    }
  }
}

float compute_range(const float *u,
                    int N) {
  float umin = VERY_LARGE_NUMBER;
  float umax = -VERY_LARGE_NUMBER;
  for(int i=0; i<N*N; i++) {
    if (u[i] < umin)
      umin = u[i];
    if (u[i] > umax)
      umax = u[i];
  }
  return umax-umin;
}
```

Example: PDE solver



Profiling the code:

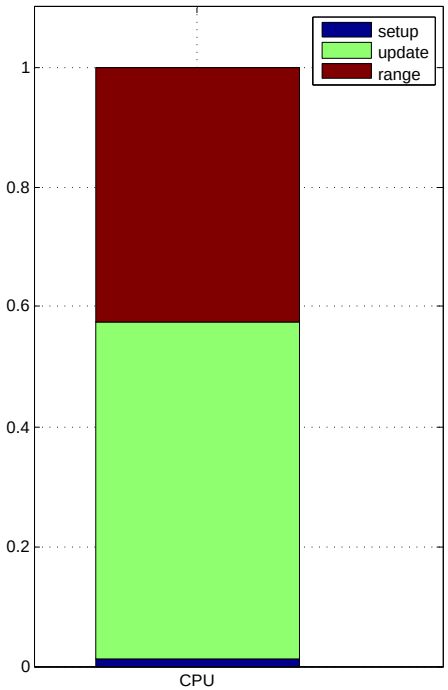
- First try putting `update` on GPU

```
__global__ void update_kernel(float *u, float *u_old) {
  int i = threadIdx.x + blockIdx.x*blockDim.x;
  int j = threadIdx.y + blockIdx.y*blockDim.y;
  if(i>0 && i<N-1 && j>0 && j<N-1) {
     const float a = u_old[i*N+j];
     ...
     u[i*N+j] = (1-4*LAMBDA)*a + LAMBDA*(b+c+d+e);
  }
}

void update_cuda(float *u_host, const float *u_old_host) {
  float *u_dev, *u_old_dev;
  cudaMalloc(&u_dev,N*N*sizeof(float));
  cudaMalloc(&u_old_dev,N*N*sizeof(float));
  cudaMemset(u_dev,0,N*N*sizeof(float));
  cudaMemcpy(u_old_dev,u_old_host,N*N*sizeof(float),
             cudaMemcpyHostToDevice);
  int num_blocks = N/BSIZE + (N%BSIZE != 0);
  dim3 grid_dim(num_blocks,num_blocks);
  dim3 block_dim(BSIZE,BSIZE);
  update_kernel<<<grid_dim,block_dim>>>(u_dev,u_old_dev);
  cudaMemcpy(u_host,u_dev,N*N*sizeof(float),
             cudaMemcpyDeviceToHost);
  cudaFree(u_dev); cudaFree(u_old_dev);
}
```

Example: PDE solver

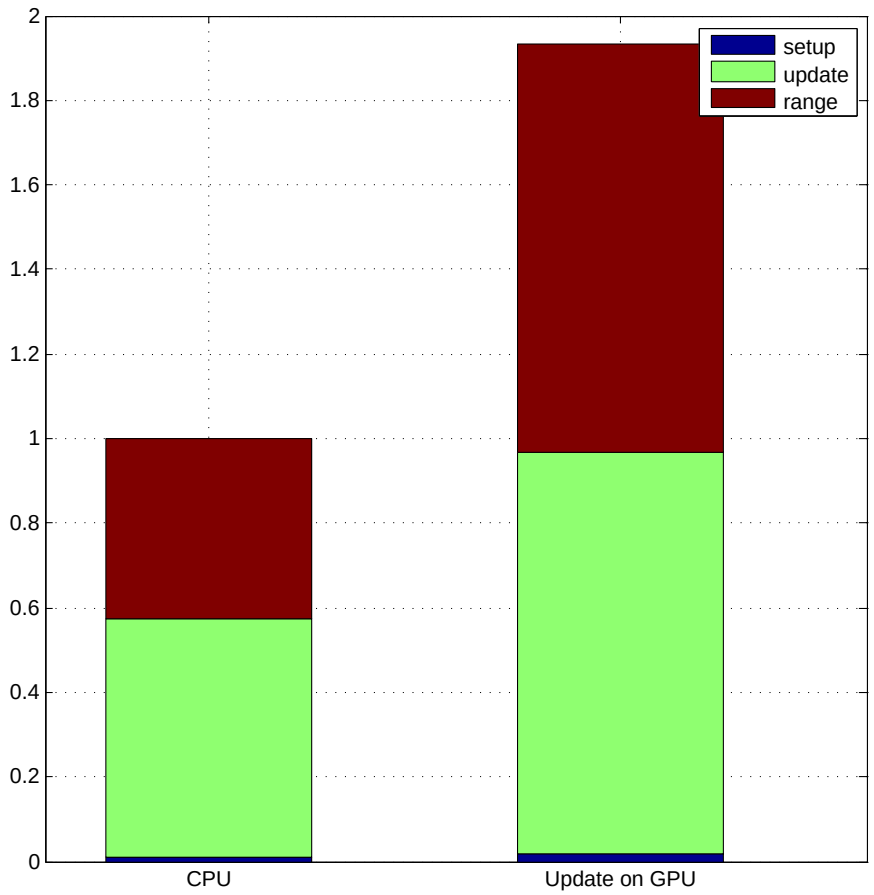
Discuss:

- What performance can be expected?
- Any suggestions for improvement?




```
__global__ void update_kernel(float *u, float *u_old) {
  int i = threadIdx.x + blockIdx.x*blockDim.x;
  int j = threadIdx.y + blockIdx.y*blockDim.y;
  if(i>0 && i<N-1 && j>0 && j<N-1) {
     const float a = u_old[i*N+j];
     ...
     u[i*N+j] = (1-4*LAMBDA)*a + LAMBDA*(b+c+d+e);
  }
}

void update_cuda(float *u_host, const float *u_old_host) {
  float *u_dev, *u_old_dev;
  cudaMalloc(&u_dev,N*N*sizeof(float));
  cudaMalloc(&u_old_dev,N*N*sizeof(float));
  cudaMemset(u_dev,0,N*N*sizeof(float));
  cudaMemcpy(u_old_dev,u_old_host,N*N*sizeof(float),
             cudaMemcpyHostToDevice);
  int num_blocks = N/BSIZE + (N%BSIZE != 0);
  dim3 grid_dim(num_blocks,num_blocks);
  dim3 block_dim(BSIZE,BSIZE);
  update_kernel<<<grid_dim,block_dim>>>(u_dev,u_old_dev);
  cudaMemcpy(u_host,u_dev,N*N*sizeof(float),
             cudaMemcpyDeviceToHost);
  cudaFree(u_dev); cudaFree(u_old_dev);
}
```

Example: PDE solver



Results: update on GPU

- Almost 2x slower
- Also range is slower
- What is going on?

Explanation

- Data allocated each time
- `update`: Host-device transfer 50% of time

Idea:

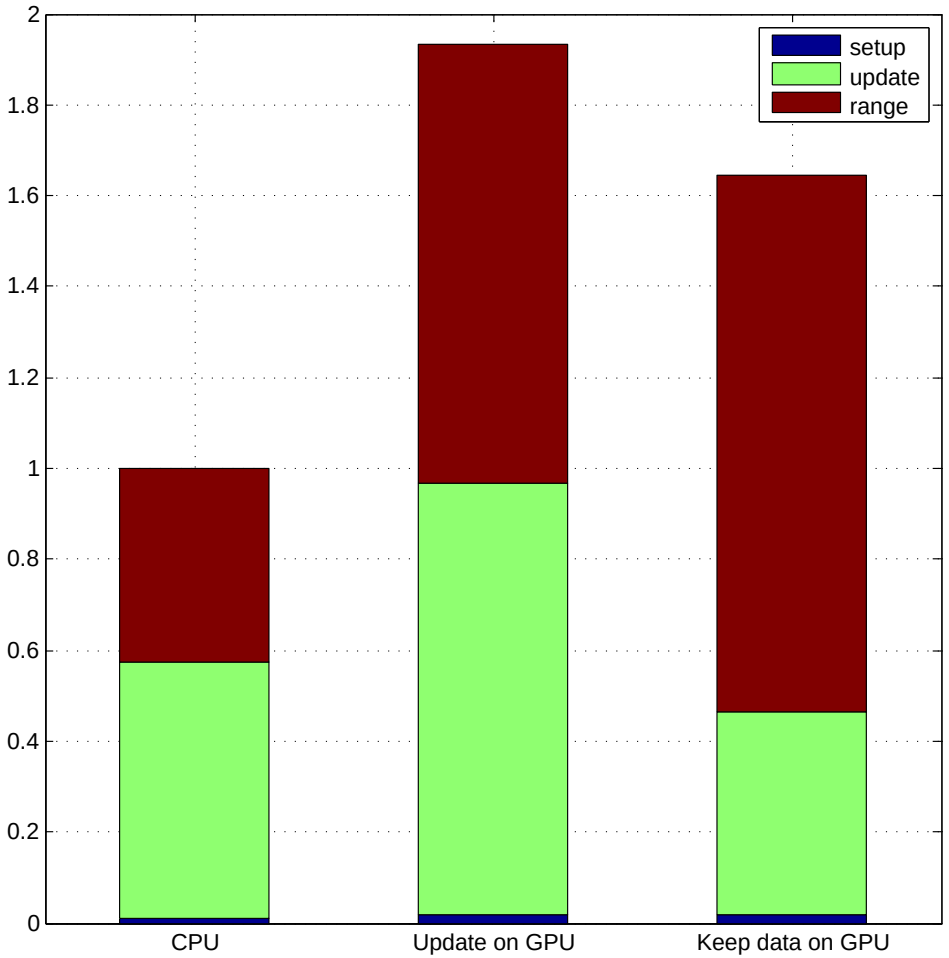
- Only need to get data for `range`
- Keep data on GPU

Example: PDE solver

Keep data on GPU:

```
void update_cuda(float *u, const float *u_old) {
  int num_blocks = N/BSIZE + (N%BSIZE != 0);
  dim3 grid_dim(num_blocks,num_blocks);
  dim3 block_dim(BSIZE,BSIZE);
  update_kernel<<<grid_dim,block_dim>>>(u,u_old);
}
void main() {
  float range = VERY_LARGE_NUMBER;
  float *u_host = malloc(N*N*sizeof(float));
  float *u, *u_old;
  cudaMalloc(&u,N*N*sizeof(float));
  cudaMalloc(&u_old,N*N*sizeof(float));
  /* setup initial condition */
  init_data(u_host,N);
  cudaMemset(u_old_dev,0,N*N*sizeof(float));
  cudaMemcpy(u,u_host,N*N*sizeof(float),
             cudaMemcpyHostToDevice);

  while(range > LIMIT) {
    /* swap pointers */
    swap(&u_old,&u);
    /* compute next time step */
    update_cuda(u,u_old,N);
    /* copy data to host and compute range */
    cudaMemcpy(u_host,u,N*N*sizeof(float),
               cudaMemcpyDeviceToHost);
    range = compute_range(u_host,N);
  }
}
```

Example: PDE solver



Results: keep data on GPU

- Data transfer reduced
- Still, `range` slow
- Reason: cache invalidated at memcpy

Idea:

- Perform `range` on GPU
- Compute max and min in columns in parallel

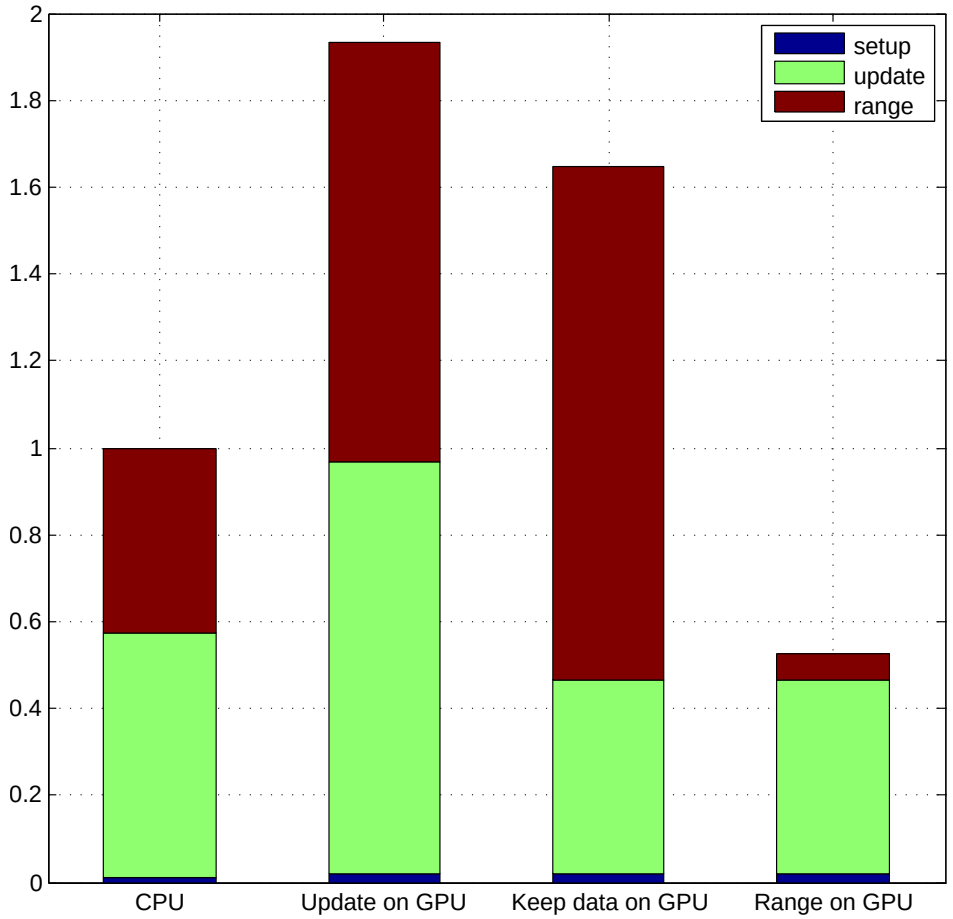
Example: PDE solver

`range` on GPU:

```
__global__ void maxmin_kernel(float *umin_col,
                              float *umax_col,
                              float *u) {
  /* column to find max/min in */
  int j = threadIdx.x + blockIdx.x*blockDim.x;
  if(j<N) {
    float umin = VERY_LARGE_NUMBER;
    float umax = -VERY_LARGE_NUMBER;
    for(int i=0; i<N; i++) {
      if (u[i*N+j] < umin)
        umin = u[i*N+j];
      if (u[i*N+j] > umax)
        umax = u[i*N+j];
    }
    umin_col[j] = umin;
    umax_col[j] = umax;
  }
}
```

```
float compute_range (const float *u_dev)
{
  float *umax_dev, *umin_dev;
  cudaMalloc(&umax_dev,N*sizeof(float));
  cudaMalloc(&umin_dev,N*sizeof(float));

  int num_blocks = N/BSIZE+ (N%BSIZE!= 0);
  const dim3 gdim(num_blocks);
  const dim3 bdim(BSIZE);
  maxmin_kernel<<<gdim,bdim>>>(umin_dev,
                               umax_dev,
                               u_dev);
  float umax[N], umin[N];
  cudaMemcpy(umax,umax_dev,N*sizeof(float),
             cudaMemcpyDeviceToHost);
  cudaMemcpy(umin,umin_dev,N*sizeof(float),
             cudaMemcpyDeviceToHost);
  cudaFree(umax_dev); cudaFree(umin_dev);

  for(int i=1; i<N; i++){
    if(umax[i] > umax[0]) umax[0] = umax[i];
    if(umin[i] < umin[0]) umin[0] = umin[i];
  }
  return umax[0]-umin[0];
}
```

Example: PDE solver



Results: `range` on GPU

- `range` now 20x faster
- 7x faster than original code

Question:

- Can we improve `update`?

Idea:

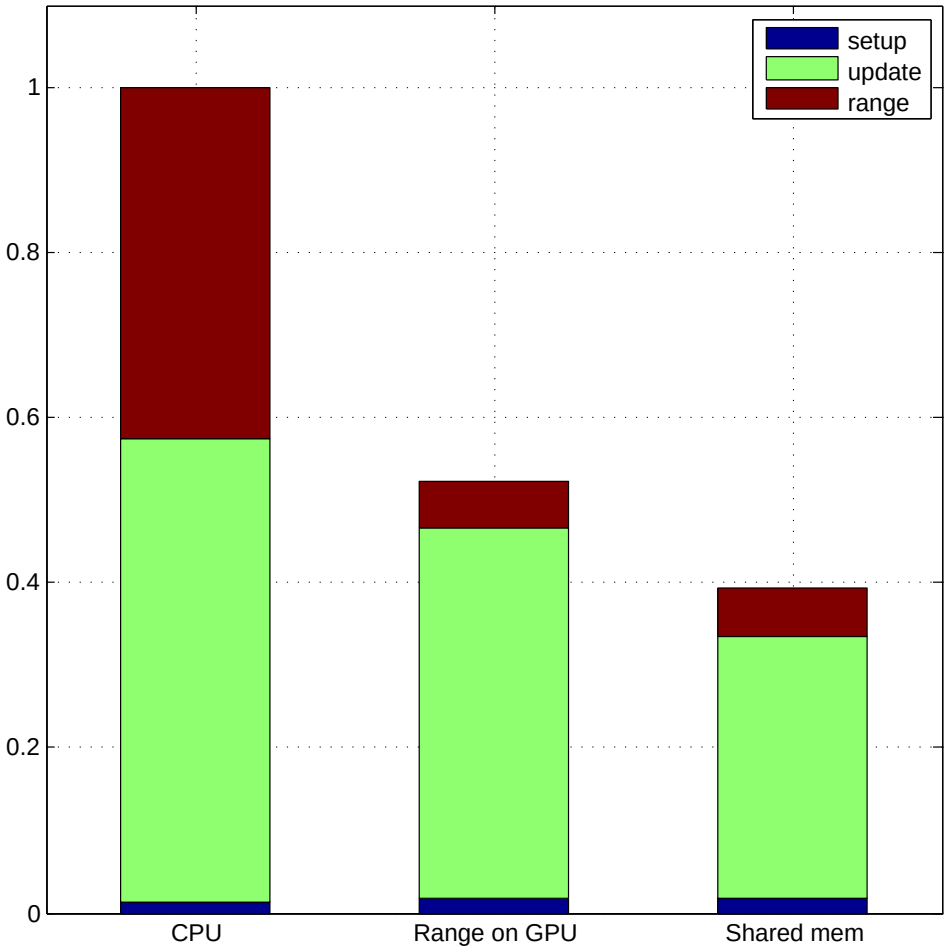
- Reusage of data within block
- Use `__shared__` memory!

Example: PDE solver

Use `__shared__` memory:

```
__global__ void update_kernel(float *u, const float *u_old)
{
  const int i = threadIdx.x + blockIdx.x*blockDim.x;
  const int j = threadIdx.y + blockIdx.y*blockDim.y;
  const int iloc = threadIdx.x+1;
  const int jloc = threadIdx.y+1;
  __shared__ float us[BLOCK_SIZE+2][BLOCK_SIZE+2];

  if(i>0 && i<N-1 && j>0 && j<N-1) {
    //fetch my item
    us[iloc][jloc] = u_old[i*N+j];
    // fetch top ghost cells (outer if clause makes sure we're not at the boundary)
    if(iloc==1) us[iloc-1][jloc] = u_old[(i-1)*N+j];
    // fetch bottom ghost cells
    if(iloc==BLOCK_SIZE) us[iloc+1][jloc] = u_old[(i+1)*N+j];
    // fetch left ghost cells
    if(jloc==1) us[iloc][jloc-1] = u_old[(i)*N+j-1];
    // fetch right ghost cells
    if(jloc==BLOCK_SIZE) us[iloc][jloc+1] = u_old[(i)*N+j+1];
    __syncthreads();

    const float a = us[iloc][jloc];
    const float b = us[iloc+1][jloc];
    const float c = us[iloc-1][jloc];
    const float d = us[iloc][jloc+1];
    const float e = us[iloc][jloc-1];

    u[i*N+j] = (1-4*LAMBDA)*a + LAMBDA*(b+c+d+e);
  }
}
```

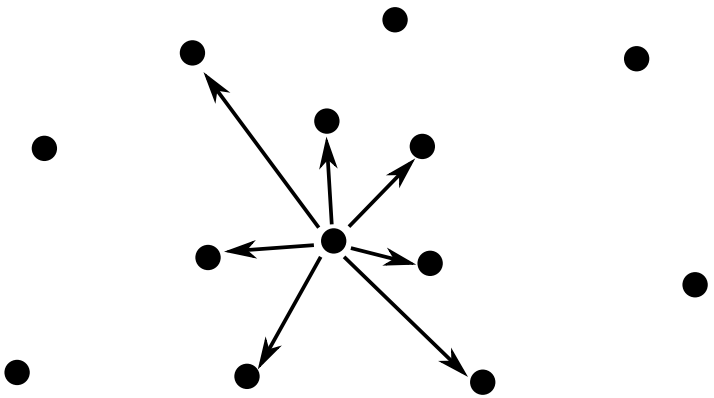
Example: PDE solver



Results: using shared memory

- `update` now 40% faster
- Overall 2.5x faster than CPU

Example: N-body simulation



- System of N pair-wise interacting particles:

$$m_i \ddot{\mathbf{x}}_i = \sum_{j=1}^{N} \mathbf{F}(\mathbf{x}_i, \mathbf{x}_j)\,, \; i = 1, \ldots, N$$

- Astrophysics, Molecular dynamics, etc.
- E.g. molecular dynamics, Lennard-Jones potential

$$\mathbf{F}(\mathbf{x}_i, \mathbf{x}_j) = \left(\frac{A}{|\mathbf{r}_{ij}|^8} - \frac{B}{|\mathbf{r}_{ij}|^{14}} \right) \mathbf{r}_{ij} \quad , \quad \mathbf{r}_{ij} = \mathbf{x}_i - \mathbf{x}_j$$

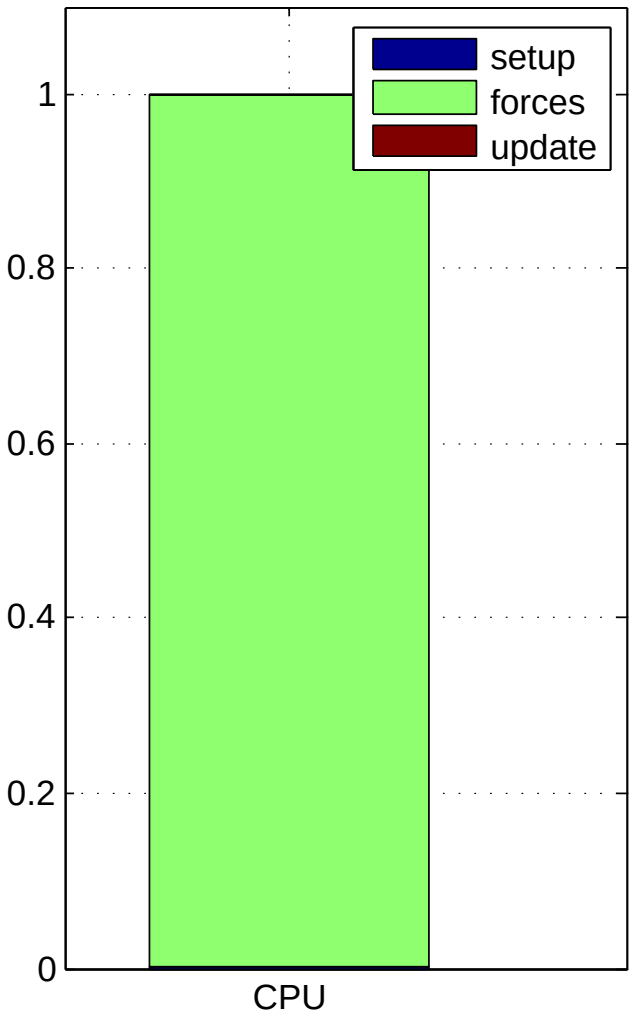
- System of ODEs – Time-stepping

Example: N-body simulation

```
void force(particle_t *i, particle_t *j
           float *fx,float *fy) {
  const float dx = i->x - j->x;
  const float dy = i->y - j->y;
  const float r2 = dx*dx + dy*dy;
  const float r4 = r2 * r2;
  const float r8 = r4 * r4;
  const float r14 = r8 * r4 * r2;
  const float coeff = LJ_A / r8 - LJ_B / r14;
  *fx = coeff*dx;
  *fy = coeff*dy;
}

void calcForces(particle_t *pts) {
  float tmpx,tmpy,fx,fy;
  for(int i = 0; i < N; ++i) {
    tmpx=0;tmpy=0;
    for(int j = i+1; j < N; ++j) {
      force(&pts[i],&pts[j],&fx,&fy);
      tmpx += fx;
      tmpy += fy;
      pts[j].fx -= fx;
      pts[j].fy -= fy;
    }
    pts[i].fx = tmpx;
    pts[i].fy = tmpy;
  }
}
```

```
void update(particle_t *pts)
  {
  for (int i = 0; i < N; ++i) {
    float accx = pts[i].fx / pts[i].mass;
    float accy = pts[i].fy / pts[i].mass;
    pts[i].vx += 0.5*DT*accx;
    pts[i].vy += 0.5*DT*accy;
    pts[i].x += DT*pts[i].vx + 0.5*DT*DT*accx;
    pts[i].y += DT*pts[i].vy + 0.5*DT*DT*accy;
    pts[i].vx += 0.5*DT*accx;
    pts[i].vy += 0.5*DT*accy;
    pts[i].fx = 0.0;
    pts[i].fy = 0.0;
  }
}

int main(int argc, char *argv[])
{
  particle_t *pts = malloc(N*sizeof(particle_t))
  // initialize
  init(pts);
  // iterate in time
  for (int k = 0; k < NUMSTEPS; k++) {
    // Calculate forces
    calcForces(pts);
    // Update positions and velocities
    update(pts);
  }
  free(pts);
}
```

Example: N-body simulation



Profiling the code:

- Computing forces takes all time $\sim \mathcal{O}(N^2)$
- Update $\sim \mathcal{O}(N)$
- Do force calculation on GPU

Recall from pthreads lecture:

- Dependencies
- Many forces (threads) contribute to one particle

Idea:

- Avoid concurrent updates by doing extra work
- Gflop/s abundant on GPU

Example: N-body simulation

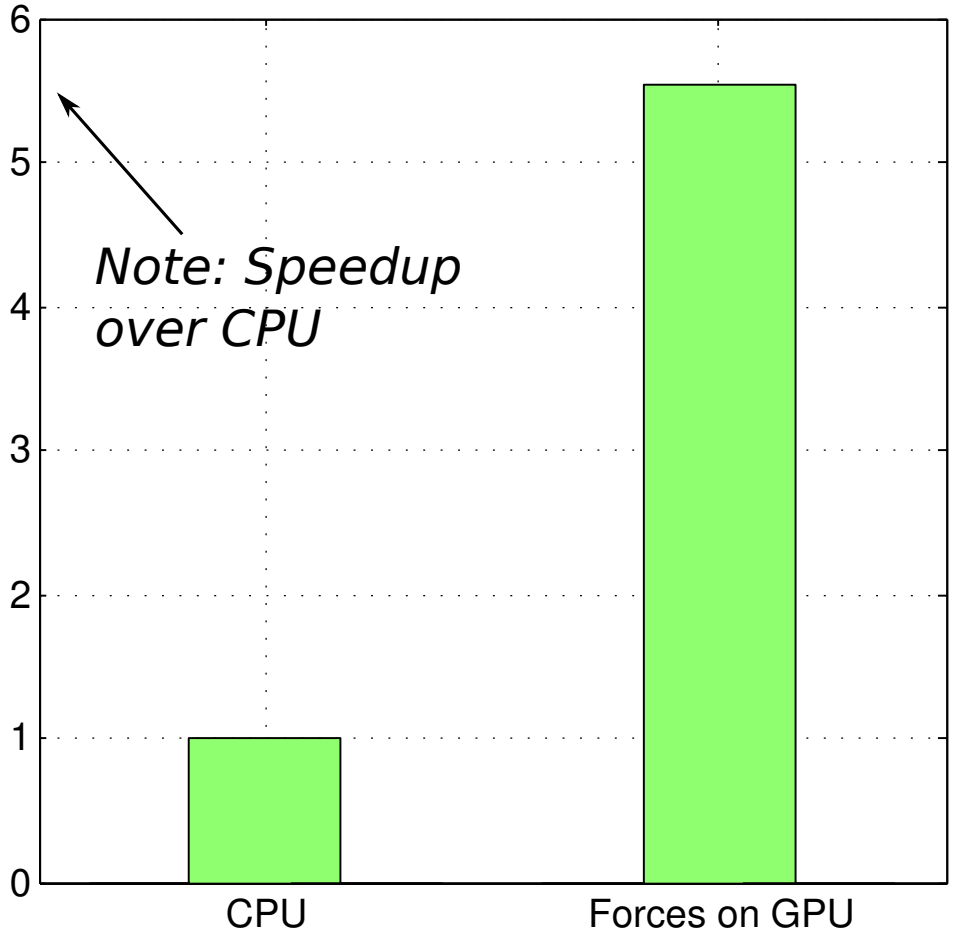
```
/* now __device__ function */
__device__ void force(particle_t *i,
                      particle_t *j,
                      float *fx, float *fy)
{
  ...
}

__global__ void force_kernel(particle_t *pts)
{
  int i = threadIdx.x + blockIdx.x*blockDim.x;

  float tmpx,tmpy;
  float fx,fy;
  if(i < N) {
    tmpx=0;tmpy=0;
    for(int j = 0; j < N; ++j)
      if(i != j) {
        force(&pts[i],&pts[j],&fx,&fy);
        tmpx += fx;
        tmpy += fy;
      }
    // only update my particle
    pts[i].fx = tmpx;
    pts[i].fy = tmpy;
  }
}
```

```
void calcForces(particle_t *pts)
{
  particle_t *particles_dev;
  cudaMalloc(&particles_dev,
             N*sizeof(particle_t));
  cudaMemcpy(pts_dev,pts,N*sizeof(particle_t),
             cudaMemcpyHostToDevice);

  int num_blocks = N/BSIZE + (N%BSIZE != 0);
  const dim3 gdim(num_blocks);
  const dim3 bdim(BSIZE);
  force_kernel<<<gdim,bdim>>>(pts_dev);

  cudaMemcpy(pts,pts_dev,N*sizeof(particle_t),
             cudaMemcpyDeviceToHost);
  cudaFree(particles_dev);
}
```

Discussion:

- What performance can be expected?
- Difference from PDE example?

Example: N-body simulation



Results: `calcForce` on GPU

- 5.5x faster
- Data transfer negligible
- Again $\mathcal{O}(N)$ data and $\mathcal{O}(N^2)$ computations

Can we improve this?

Example: N-body simulation

```
/* now __device__ function */
__device__ void force(particle_t *i,
                      particle_t *j,
                      float *fx, float *fy)
{
  ...
}

__global__ void force_kernel(particle_t *pts)
{
  int i = threadIdx.x + blockIdx.x*blockDim.x;

  float tmpx,tmpy;
  float fx,fy;
  if(i < N) {
    tmpx=0;tmpy=0;
    for(int j = 0; j < N; ++j)
      if(i != j) {
        force(&pts[i],&pts[j],&fx,&fy);
        tmpx += fx;
        tmpy += fy;
      }
    // only update my particle
    pts[i].fx = tmpx;
    pts[i].fy = tmpy;
  }
}
```

Problem:

- Divergent execution due to if statement

Question:

- What can be done?
- *Hint:* $\mathbf{F}_{ij} = \left(\frac{A}{r_{ij}^8} - \frac{B}{r_{ij}^{14}} \right) \mathbf{r}_{ij}$
- *Hint 2:* Flops are "free" on GPUs

Idea:

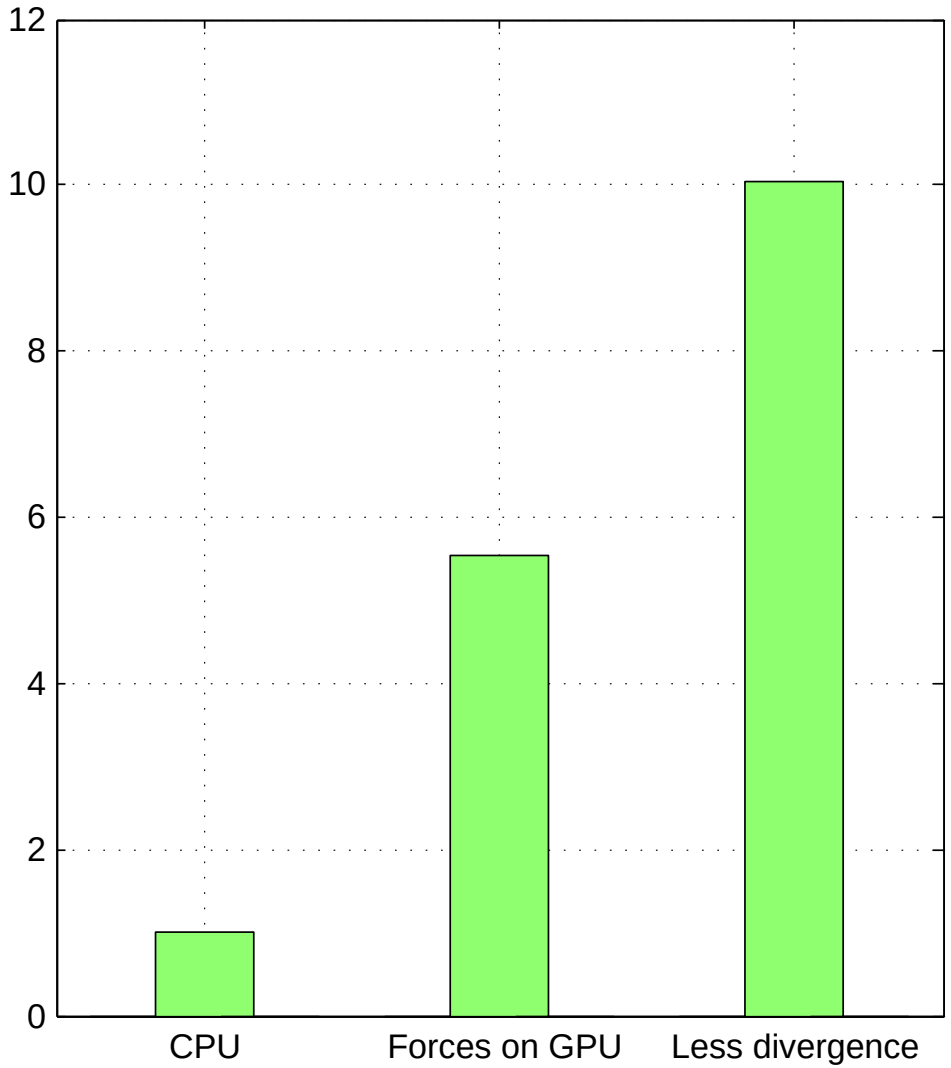
- Again, compute abundant
- Introduce damping $r^2 := |\mathbf{r}_{ij}|^2 + \epsilon^2$
- Then compute $\mathbf{F}_{ji}$ aswell

Example: N-body simulation

Idea: Less branch divergence

```
__device__ void force(particle_t *i, particle_t *j
                      float *fx, float *fy) {
  const float dx = i->x - j->x;
  const float dy = i->y - j->y;
  // add damping to prevent division by 0
  const float r2 = dx*dx + dy*dy + EPS2;
  const float r4 = r2 * r2;
  const float r8 = r4 * r4;
  const float r14 = r8 * r4 * r2;
  const float coeff = LJ_A / r8 - LJ_B / r14;
  *fx = coeff*dx;
  *fy = coeff*dy;
}
__global__ void force_kernel(particle_t *pts)
{
  int i = threadIdx.x + blockIdx.x*blockDim.x;
  float tmpx,tmpy,fx,fy;
  if(i < N) {
    tmpx=0;tmpy=0;
    for(int j = 0; j < N; ++j) {
      // include case i==j
      force(&pts[i],&pts[j],&fx,&fy);
      tmpx += fx;
      tmpy += fy;
    }
    pts[i].fx = tmpx;
    pts[i].fy = tmpy;
  }
}
```

Example: N-body simulation



Results: less branch divergence

- Improvement: 80% faster
- 10x faster than baseline

System:

- CPU: 37 Gflop/s (single core)
- GPU: 422 Gflop/s
- $\Rightarrow$ 87 % efficiency

Optimization in CUDA

Summary:

- Use many small threads
- Keep data on GPU
- Use shared memory
- Avoid branch divergence

Also:

- Coalesced memory access (not in examples)
- Less of an issue on modern GPUs