Programming Parallel Computers

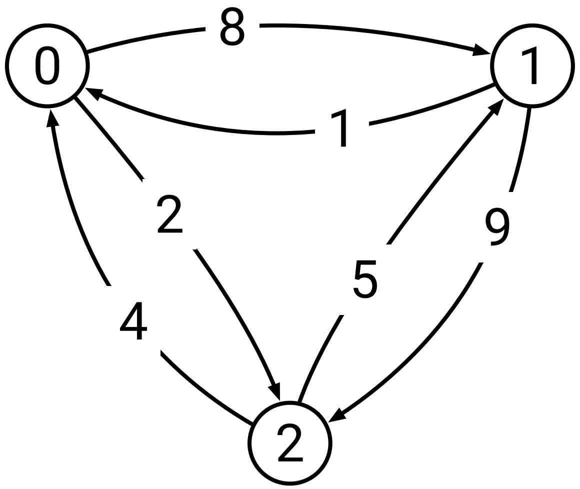
Jukka Suomela · Aalto University · ppc.cs.aalto.fi

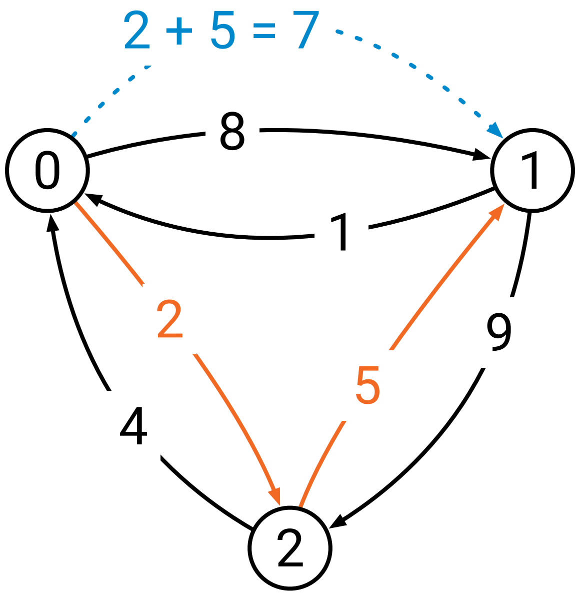
Part 4C: Memory access patterns in CUDA programs

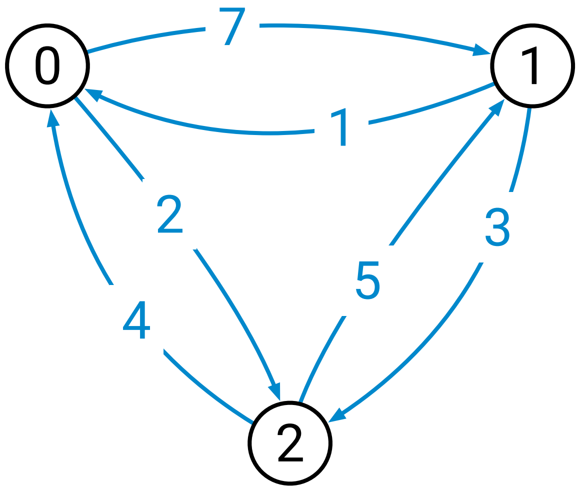
Sample application: cheapest 2-hop path

d (input):

r (output):



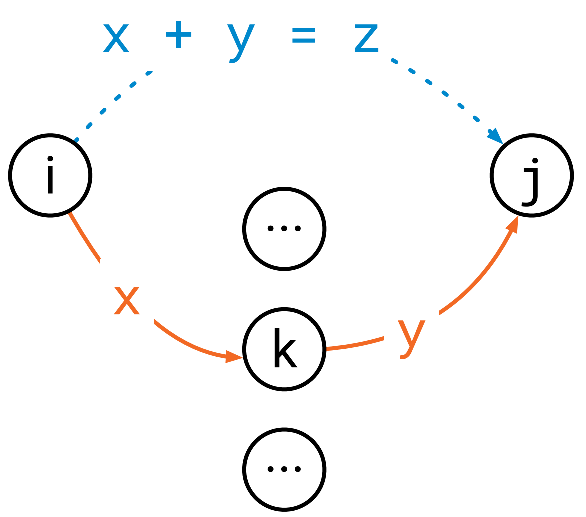





```
d[] = { 0, 8, 2,
        1, 0, 9,
        4, 5, 0 }
```

```
r[] = { 0, 7, 2,
        1, 0, 3,
        4, 5, 0 }
```

```
void step(float* r, const float* d, int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            float v = infinity;
            for (int k = 0; k < n; ++k) {
                float x = d[n*i + k];
                float y = d[n*k + j];
                float z = x + y;
                v = min(v, z);
            }
            r[n*i + j] = v;
        }
    }
}
```



Splitting work

- Work to do:
  - need to compute **$n \times n$ results**
  - computing one result takes n steps
- How do we split this in blocks and threads?
- Natural idea:
  - one thread computes **one result**
  - one block computes **$b \times b$ results**, for some suitable b
  - if we choose e.g. $b = 16$, then a block consists of 8 warps

Splitting work

- Example: input dimensions are 1600 x 1600:
  - we want to create **100 x 100 blocks**
  - each block consists of **16 x 16 threads**
- Create 10 000 blocks with 256 threads:
  - blocks numbered **0 … 9999**
  - threads numbered **0 … 255**
- Convert block & thread index to (i, j) pair:
  - thread number **123** in block number **4567** computes the result for i = **???** and j = **???**

Splitting work

- Example: input dimensions are 1600 x 1600:
  - we want to create **100 x 100 blocks**
  - each block consists of **16 x 16 threads**
- Create 10 000 blocks with 256 threads:
  - blocks numbered **0 … 9999**
  - threads numbered **0 … 255**
- Convert block & thread index to (i, j) pair:
  - thread number **123 = 7** · 16 **+ 11** in block number **4567 = 45** · 100 **+ 67** computes the result for i = **67** · 16 + **11** and j = **45** · 16 + **7**

Splitting work: using 2D indexes

- Example: input dimensions are 1600 x 1600:
  - we want to create **100 x 100 blocks**
  - each block consists of **16 x 16 threads**
- Create 10 000 blocks with 256 threads using 2D indexes:
  - blocks numbered **(0, 0) … (99, 99)**
  - threads numbered **(0, 0) … (15, 15)**
- Convert block & thread coordinates to (i, j) pair:
  - thread number **(11 , 7)** in block number **(67 , 45)** computes the result for $i = \mathbf{67} \cdot 16 + \mathbf{11}$ and $j = \mathbf{45} \cdot 16 + \mathbf{7}$

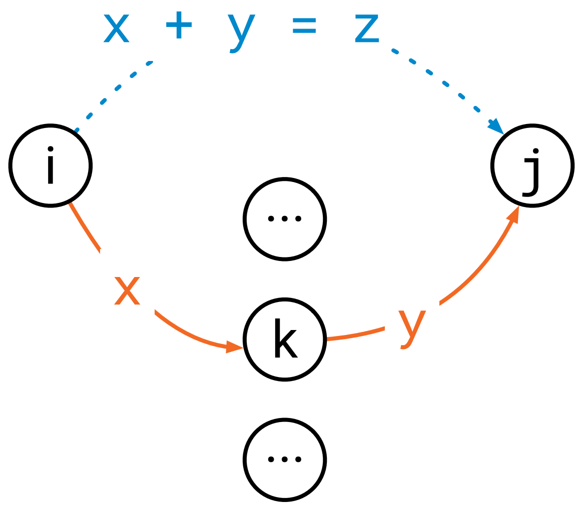
Splitting work: rounding

- Example: input dimensions are 1601 x 1601:
  - we want to create **101 x 101 blocks**
  - each block consists of **16 x 16 threads**
- Create 10 000 blocks with 256 threads using 2D indexes:
  - blocks numbered **(0, 0) … (100, 100)**
  - threads numbered **(0, 0) … (15, 15)**
- There will be some threads with $i \geq$ **1601** and/or $j \geq$ **1601**, they will do nothing

```
__global__ void mykernel(float* r, const float* d, int n) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;
    if (i >= n || j >= n)
        return;
    float v = HUGE_VALF;
    for (int k = 0; k < n; ++k) {
        float x = d[n*i + k];
        float y = d[n*k + j];
        float z = x + y;
        v = min(v, z);
    }
    r[n*i + j] = v;
}
```

What if n is not a multiple of 16

blockDim.x = 16
blockDim.y = 16



```
float* dGPU = NULL;
cudaMalloc((void**)&dGPU, n * n * sizeof(float));
float* rGPU = NULL;
cudaMalloc((void**)&rGPU, n * n * sizeof(float));
cudaMemcpy(dGPU, d, n * n * sizeof(float),
           cudaMemcpyHostToDevice);

dim3 dimBlock(16, 16);
dim3 dimGrid(divup(n, dimBlock.x), divup(n, dimBlock.y));
mykernel<<<dimGrid, dimBlock>>>(rGPU, dGPU, n);

cudaMemcpy(r, rGPU, n * n * sizeof(float),
           cudaMemcpyDeviceToHost);
cudaFree(dGPU); cudaFree(rGPU);
```

n/16, rounded up

Performance

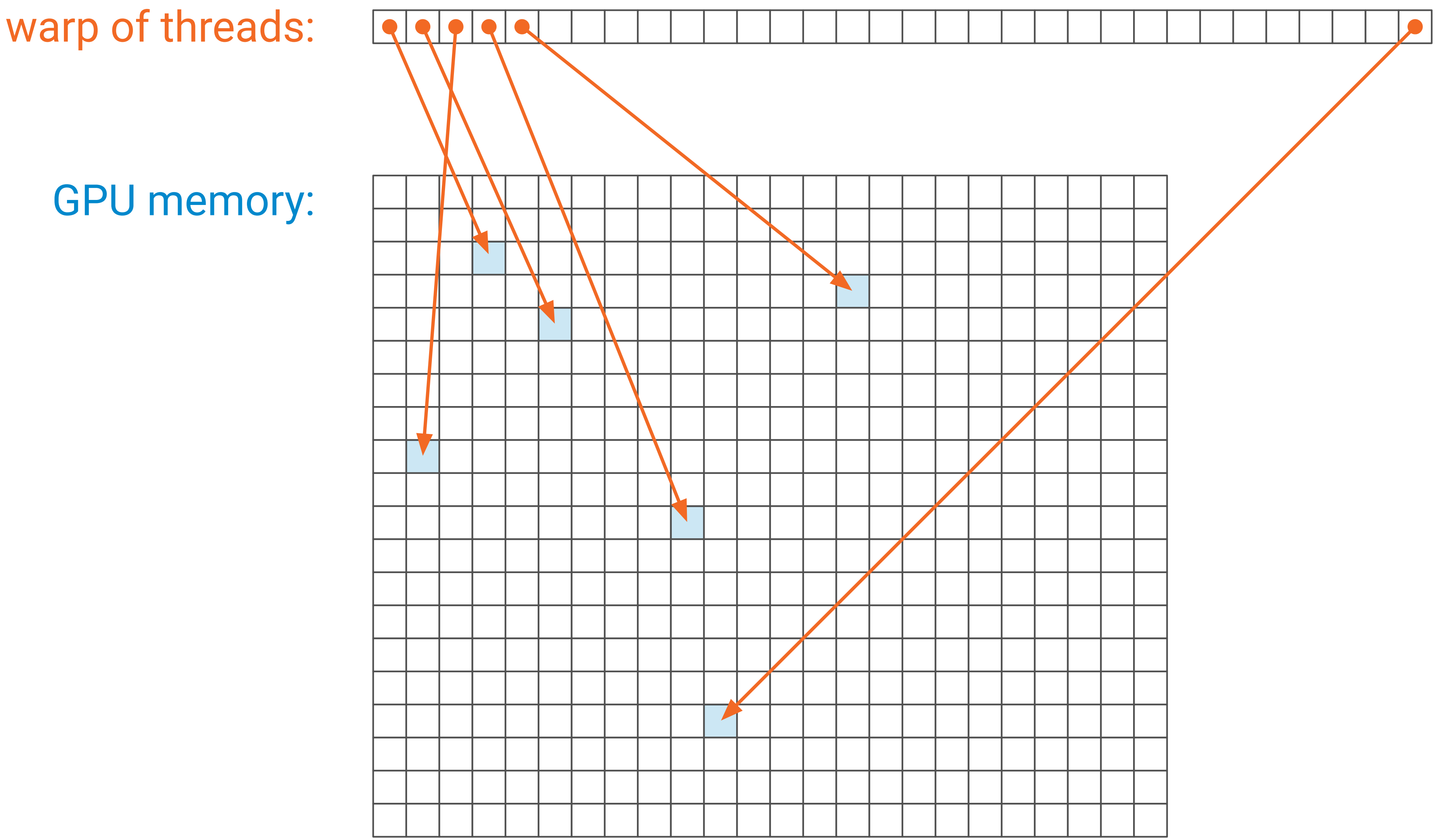
- Test input: ***n* = 6300**
- Maari computers:
  - baseline CPU solution: **397 s**
  - best CPU solution: **2.3 s**
  - current GPU solution: **42 s**
- What is the bottleneck?

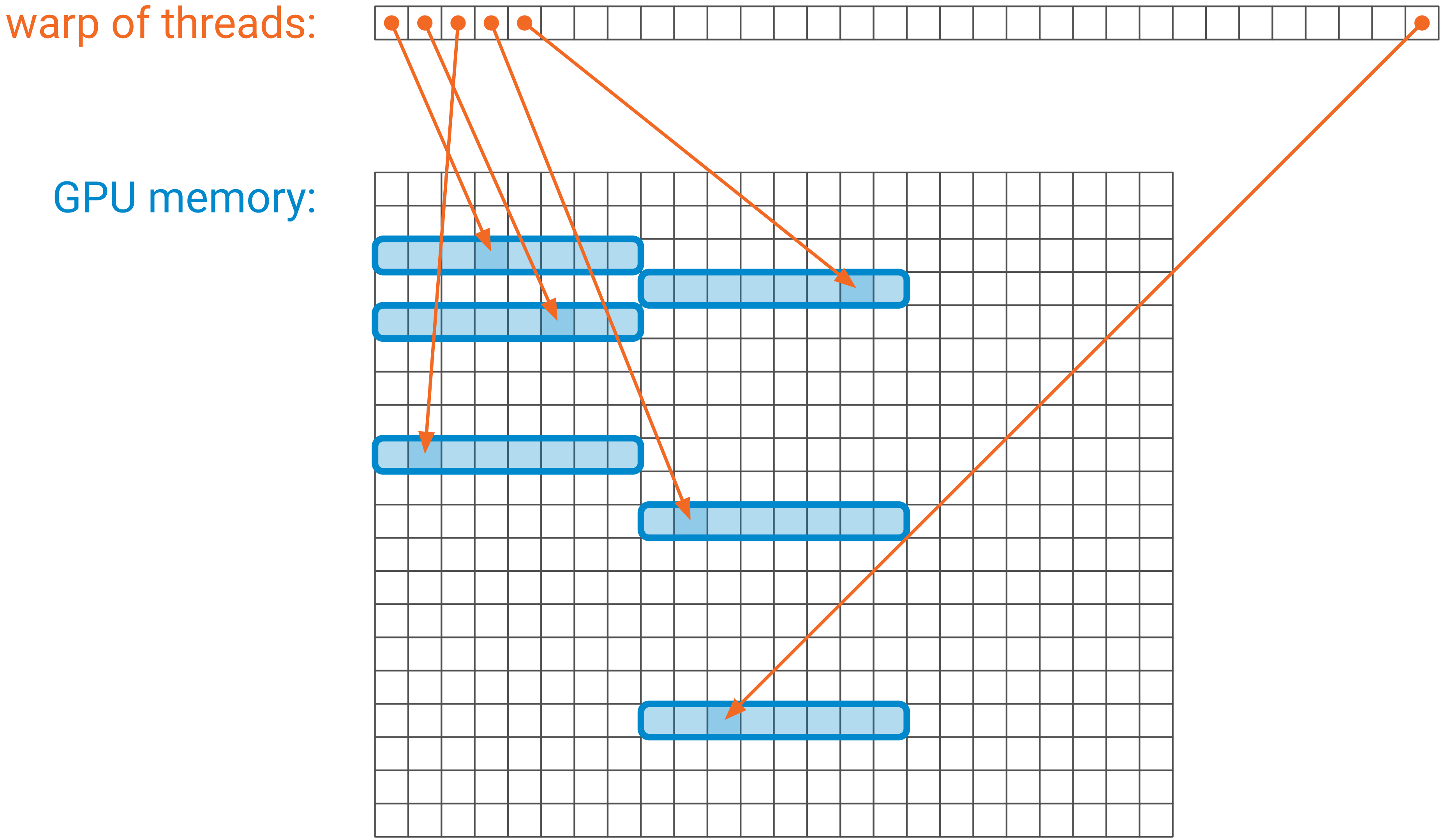
Memory

- A key challenge in CPU code:
  getting data fast enough from the CPU memory
- A key challenge in GPU code:
  getting data fast enough from the GPU memory

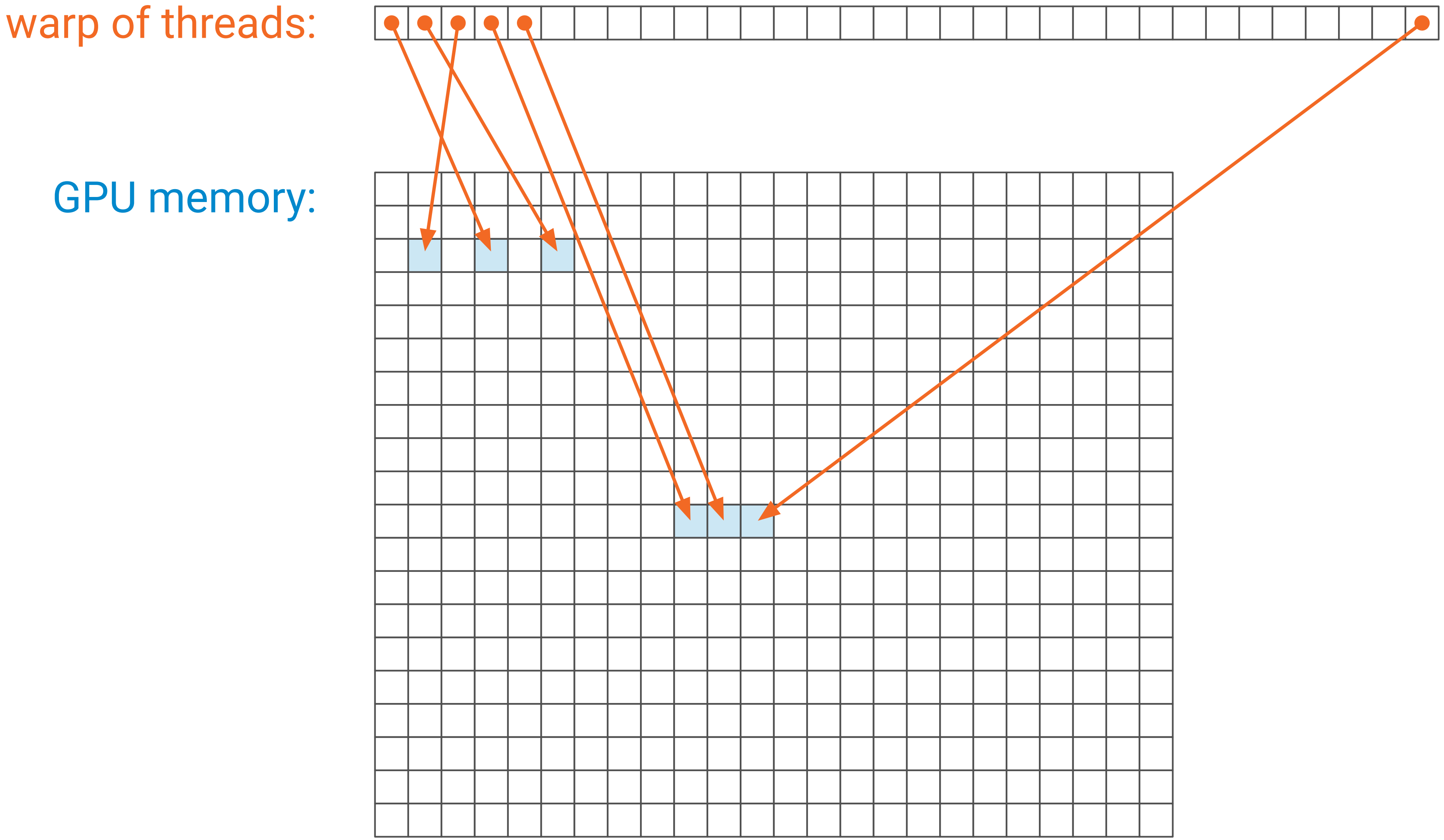
Memory access pattern

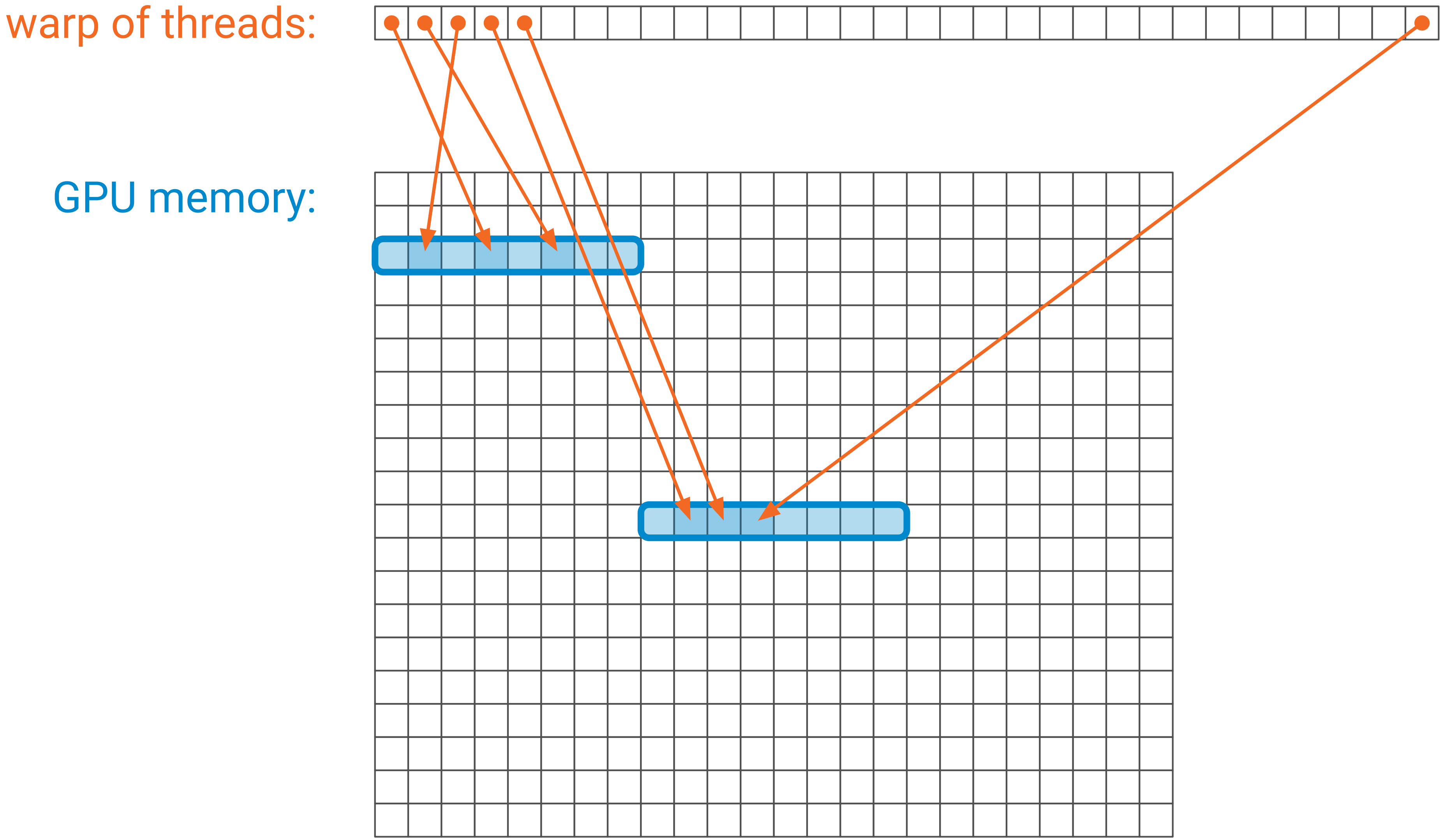
- Blocks are divided in warps
  - warp = 32 threads
- ***Entire warp executes synchronously***
- If one thread reads some memory,
  all threads of the warp read some memory

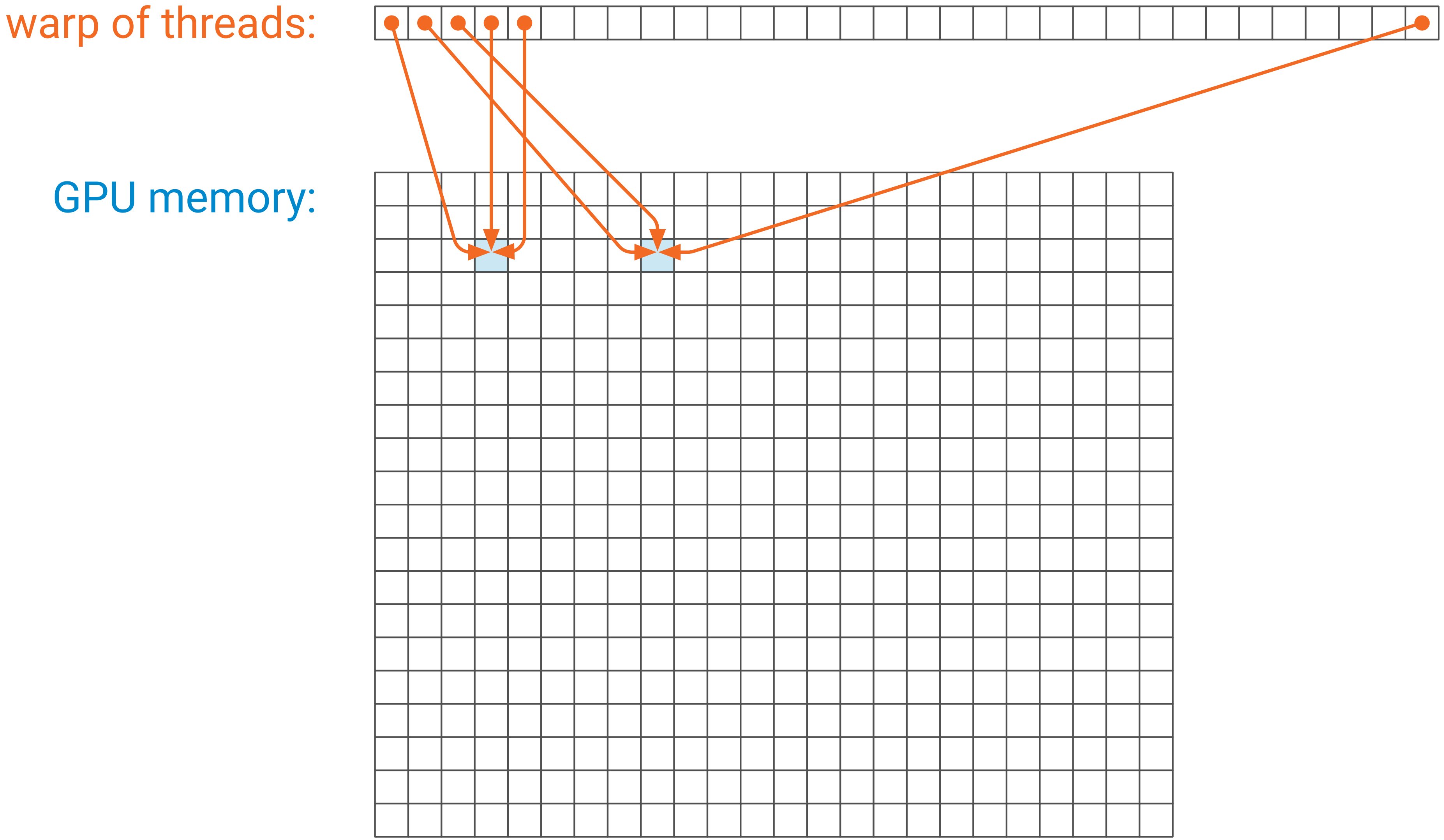

warp of threads:
GPU memory:

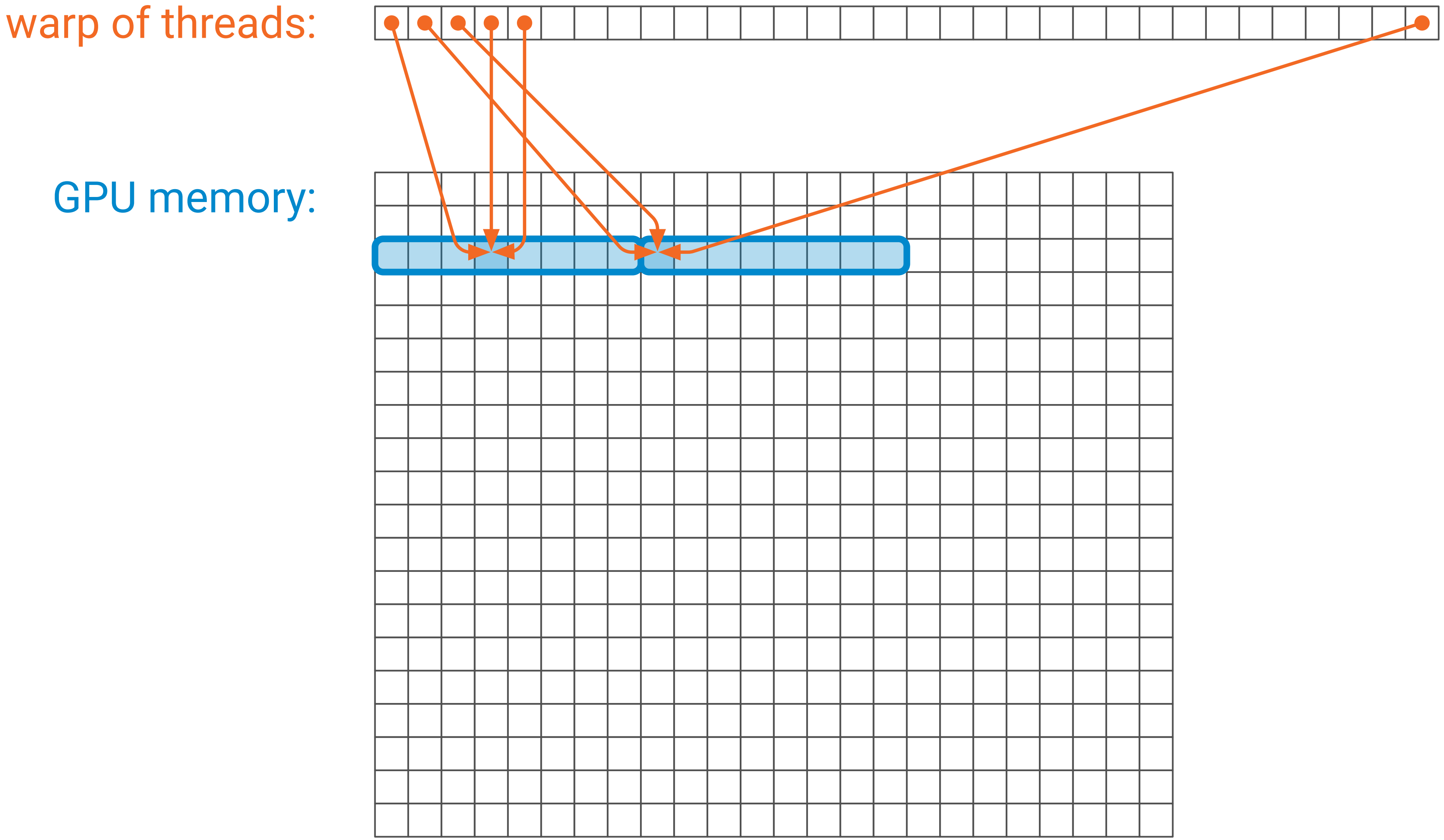


warp of threads:
GPU memory:


warp of threads:
GPU memory:


warp of threads:
GPU memory:


warp of threads:
GPU memory:


warp of threads:
GPU memory:

Memory access pattern

- One memory read in kernel:
  entire warp of threads reads memory simultaneously
- Threads access small continuous parts of memory:
  need to load few cache lines → **good**
- Threads access 32 different locations far from each other:
  need to load many cache lines → **bad**

First warp:

thread 0: i = 0, j = 0

thread 1: i = 1, j = 0

thread 2: i = 2, j = 0

thread 3: i = 3, j = 0

…

thread 31: i = 15, j = 1

Pay attention to this index!

```
int i = threadIdx.x + ...
int j = threadIdx.y + ...
for (... ++k) {
    float x = d[n*i + k];
    float y = d[n*k + j];
    ...
}
```

Bad

First warp, first iteration:

threads 0 & 16: read **d[0]**

threads 1 & 17: read **d[1000]**

threads 2 & 18: read **d[2000]**

threads 3 & 19: read **d[3000]**

...

```
int i = threadIdx.x + ...
int j = threadIdx.y + ...
for (... ++k) {
    float x = d[n*i + k];
    float y = d[n*k + j];
    ...
}
```

Bad

First warp, first iteration:

threads 0 & 16: read d[0]

threads 1 & 17: read d[1000]

threads 2 & 18: read d[2000]

threads 3 & 19: read d[3000]

...

Good

threads 0–15: read d[0]

threads 16–31: read d[1]

```
int i = threadIdx.x + ...
int j = threadIdx.y + ...
for (... ++k) {
    float x = d[n*i + k];
    float y = d[n*k + j];
    ...
}
```

Bad

First warp, second iteration:

threads 0 & 16: read d[1]

threads 1 & 17: read d[1001]

threads 2 & 18: read d[2001]

threads 3 & 19: read d[3001]

...

Good

threads 0–15: read d[1000]

threads 16–31: read d[1001]

```
int i = threadIdx.x + ...
int j = threadIdx.y + ...
for (... ++k) {
    float x = d[n*i + k];
    float y = d[n*k + j];
    ...
}
```

First warp, third iteration:

Bad

threads 0 & 16: read d[2]

threads 1 & 17: read d[1002]

threads 2 & 18: read d[2002]

threads 3 & 19: read d[3002]

...

Good

threads 0–15: read d[2000]

threads 16–31: read d[2001]

```
int i = threadIdx.x + ...
int j = threadIdx.y + ...
for (... ++k) {
    float x = d[n*i + k];
    float y = d[n*k + j];
    ...
}
```

Exchange the roles of i and j

```
int i = threadIdx.x + ...
int j = threadIdx.y + ...
for (... ++k) {
    float x = d[n*i + k];
    float y = d[n*k + j];
    float x = d[n*j + k];
    float y = d[n*k + i];
    ...
}

r[n*i + j] = v;
r[n*j + i] = v;
```

Good

First warp, first iteration:

threads 0–15: read d[0]

threads 16–31: read d[1000]

```
int i = threadIdx.x + ...
int j = threadIdx.y + ...
for (... ++k) {
    float x = d[n*j + k];
    float y = d[n*k + i];
    ...
}
```

First warp, first iteration:

```
int i = threadIdx.x + ...
int j = threadIdx.y + ...
for (... ++k) {
    float x = d[n*j + k];
    float y = d[n*k + i];
    ...
}
```

Good

threads 0–15: read d[0]

threads 16–31: read d[1000]

Good

threads 0 & 16: read d[0]

threads 1 & 17: read d[1]

threads 2 & 18: read d[2]

...

Good

First warp, second iteration:

threads 0–15: read d[1]

threads 16–31: read d[1001]

Good

threads 0 & 16: read d[1000]

threads 1 & 17: read d[1001]

threads 2 & 18: read d[1002]

...

```
int i = threadIdx.x + ...
int j = threadIdx.y + ...
for (... ++k) {
    float x = d[n*j + k];
    float y = d[n*k + i];
    ...
}
```

Good

First warp, third iteration:

threads 0–15: read d[2]

threads 16–31: read d[1002]

Good

threads 0 & 16: read d[2000]

threads 1 & 17: read d[2001]

threads 2 & 18: read d[2002]

...

```
int i = threadIdx.x + ...
int j = threadIdx.y + ...
for (... ++k) {
    float x = d[n*j + k];
    float y = d[n*k + i];
    ...
}
```

Performance

- **V0:** baseline — **42 s**
- **V1:** better memory access pattern — **8 s**
- But we can do much better by applying familiar ideas:
  - **reuse data in registers**
  - **reuse data in “cache”** (here: shared memory)

Performance

V0: baseline

V1: memory access

V2: registers

V3: shared memory