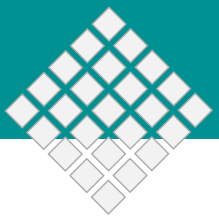
Computing with GPUs

Massively-Parallel Programming with Cuda

M1 Informatique



UNIVERSITÉ TOULOUSE III PAUL SABATIER
Université Fédérale Toulouse Midi-Pyrénées

Christine Rochange - 2022

Motivation

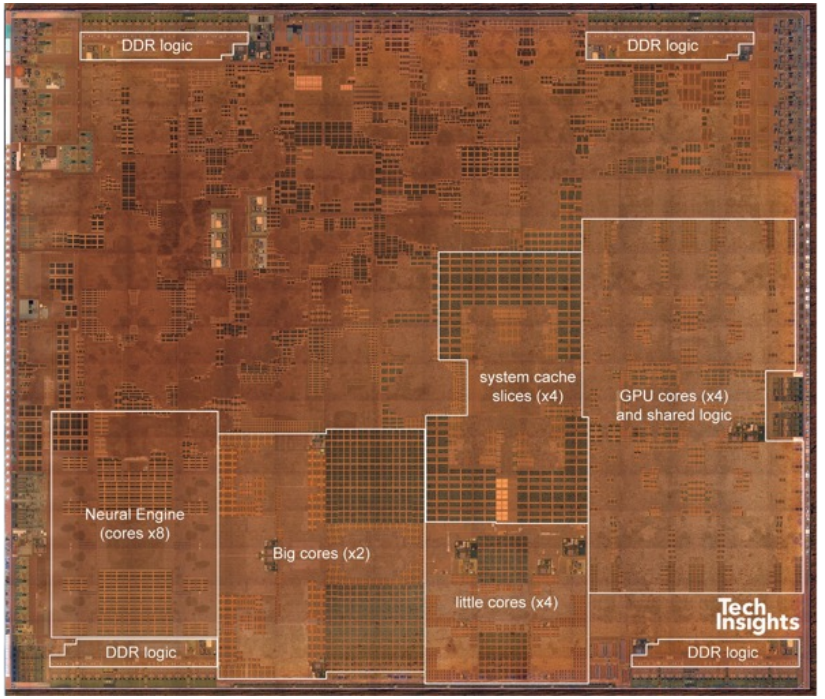
- Graphic Processing Units (GPUs) designed for real-time rendering(video games)
  . . . now available in many computing systems

+

- Growing processing requirements

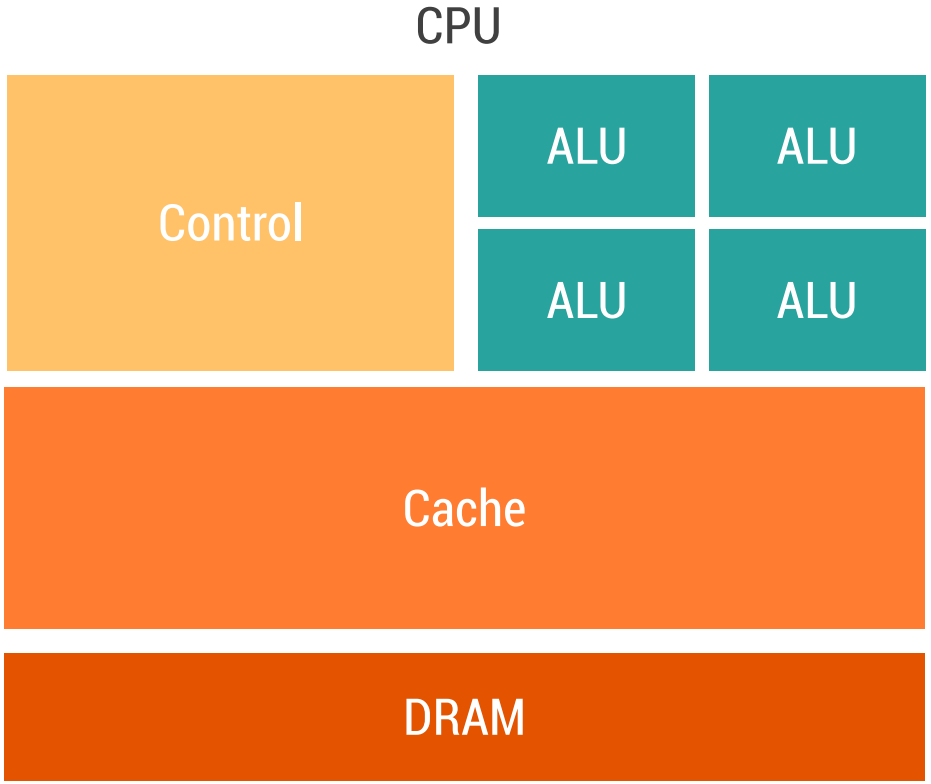
↓

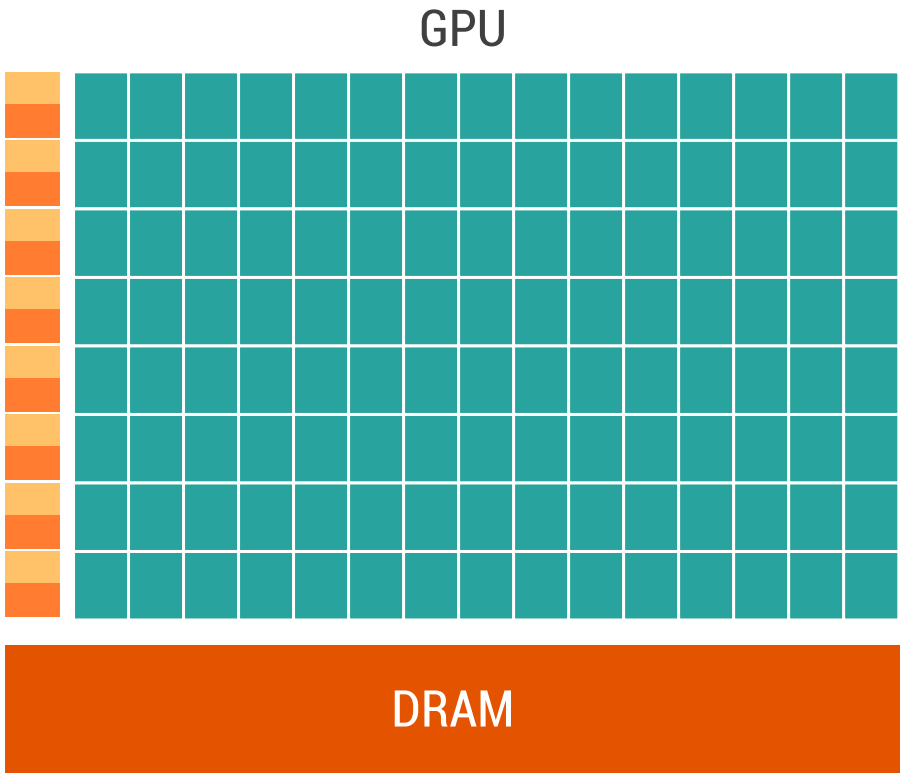
- GPGPU
  (General-Purpose GPU) computing



Apple A12

CPU vs. GPU Architecture





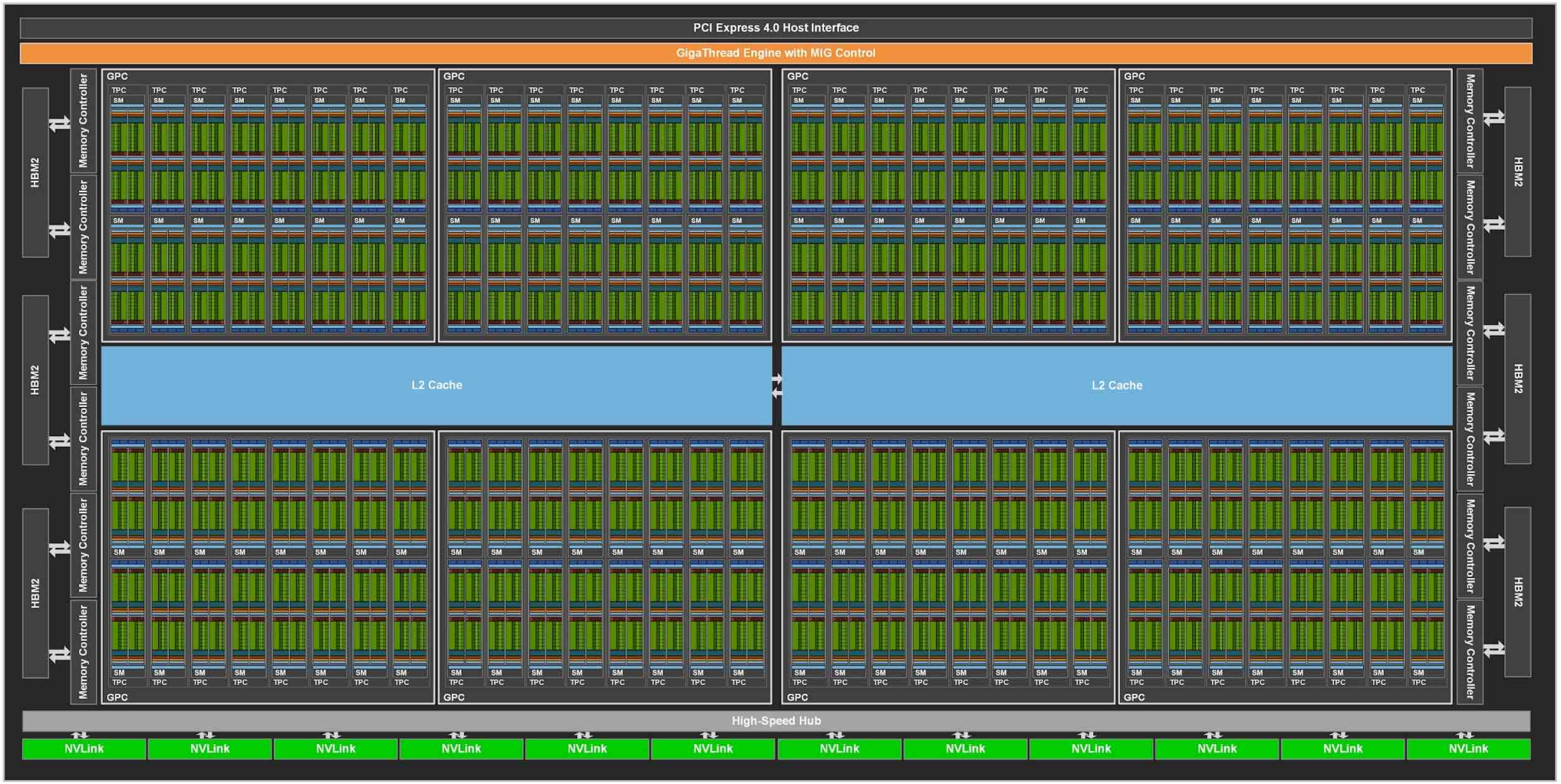
Extracts parallelism in a thread (superscalar, out-of-order execution)

Target: short thread latency

SIMT (Single Instruction-Multiple Threads) execution

Target: high thread throughput

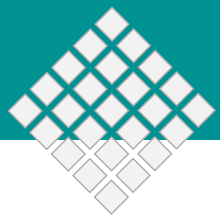
Example: Nvidia Ampere GA100



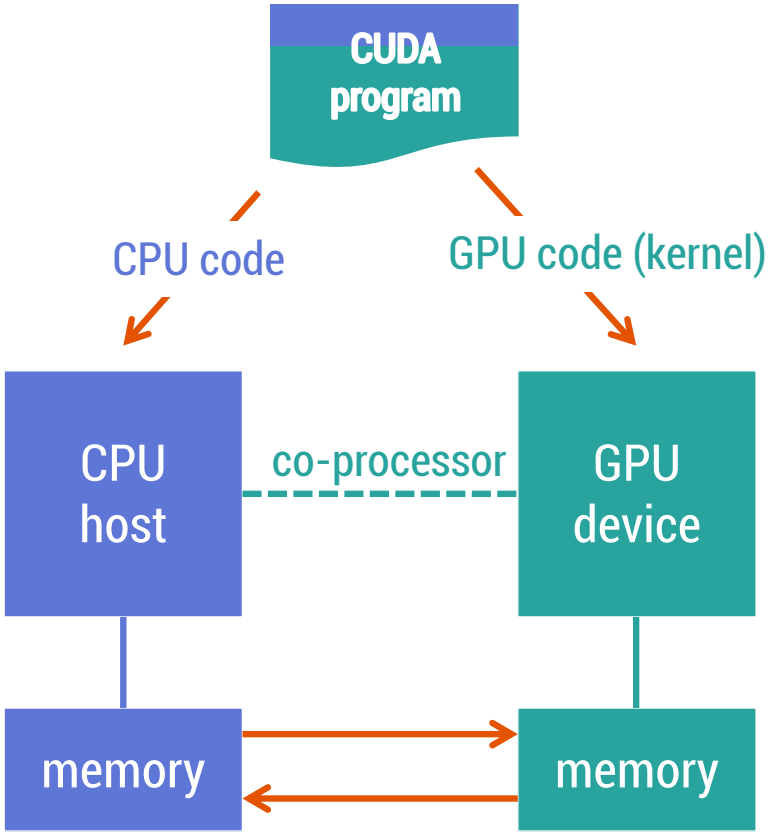
6912 cores

Example : GA100 Streaming Multiprocessor

Data Parallelism and Cuda C



Heterogeneous computation




```
void rgb2grey(uchar4 *hcimg, uchar *hgimg, int isize) {
        int bsize = 256;
        int gsize = ((isize + bsize -1) /bsize);
        int numc = isize*sizeof(uchar4);
        int numg = isize*sizeof(unsigned char);

        uchar4* dcimg;
        unsigned char *dgimg;
        cudaMalloc((void **)&dcimg, numc);
        cudaMemcpy(dcimg, hcimg, numc, cudaMemcpyHostToDevice);
        cudaMalloc((void **)&gdimg, numg) ;

        color2grey<<<gsize, bsize>>>(dcimg , dgimg , isize);

        cudaMemcpy(hgimg, dgimg, numg, cudaMemcpyDeviceToHost);
        cudaFree(dcimg) ; cudaFree(dgimg) ;
}
```

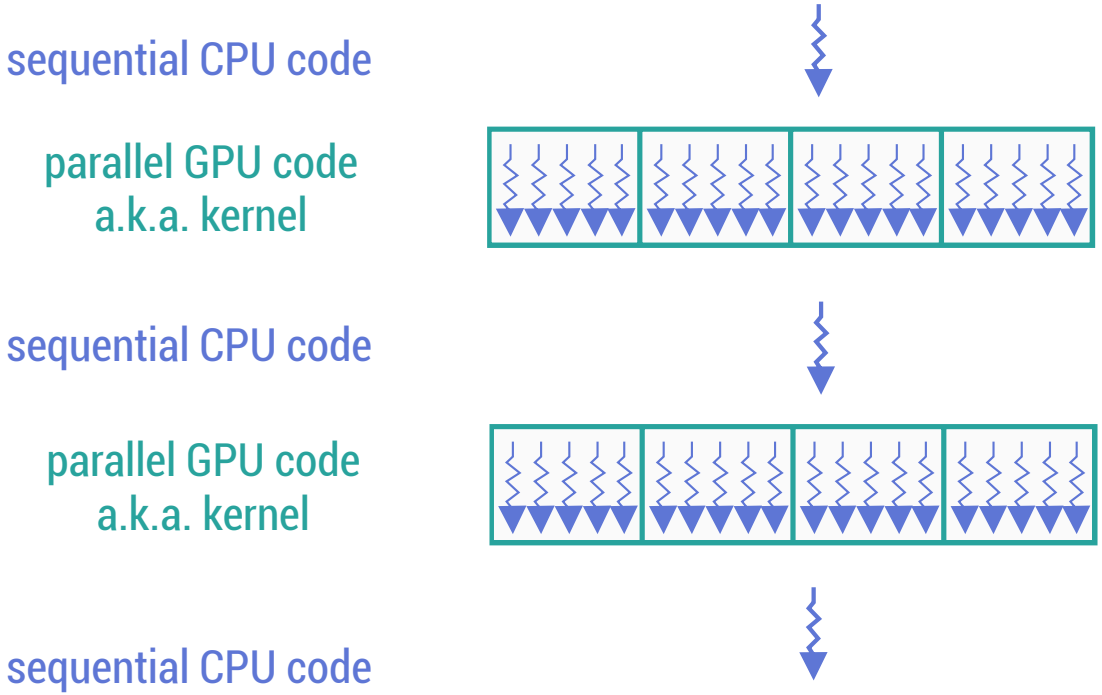
```
__global__
void color2grey(uchar4 *dcimg, uchar *dgimg, int isize){
        int index = blockIdx.x * blockDim.x + threadIdx.x;
        uchar4 p;
        if (index < isize)){
             p = dcimg[index];
             dgimg[index] =
             (299*p.x + 587*p.y + 114*p.z) / 1000;
        }
}
```

kernel = function run on the GPU by thousands of threads

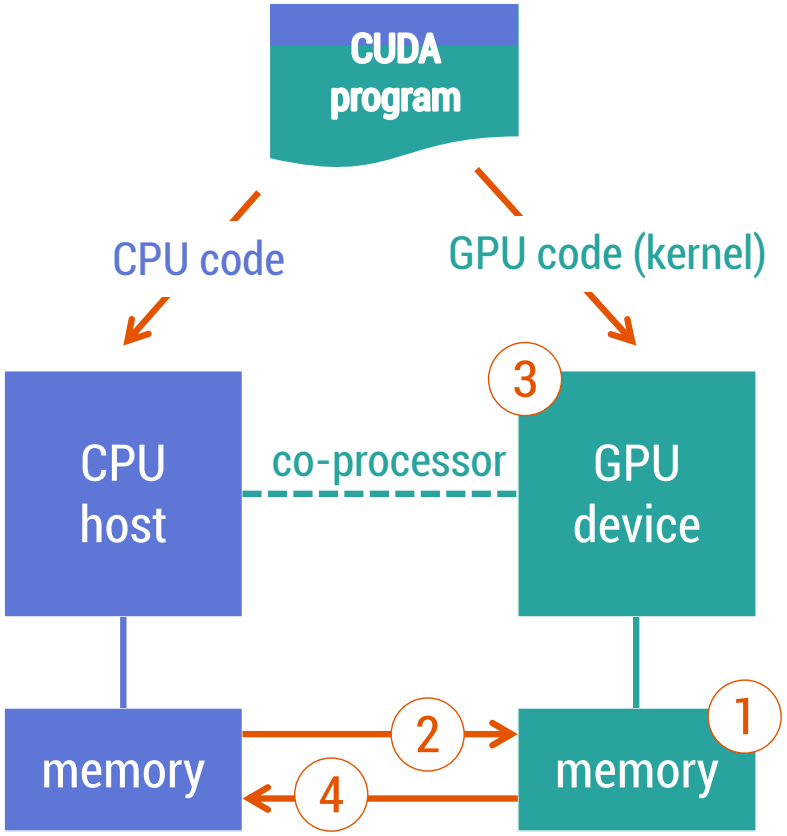
Kernels

Kernels are data-parallel functions

- the CPU initiates the execution of kernels on the GPU
- the GPU creates (*a large number of*) threads that all execute the kernel code

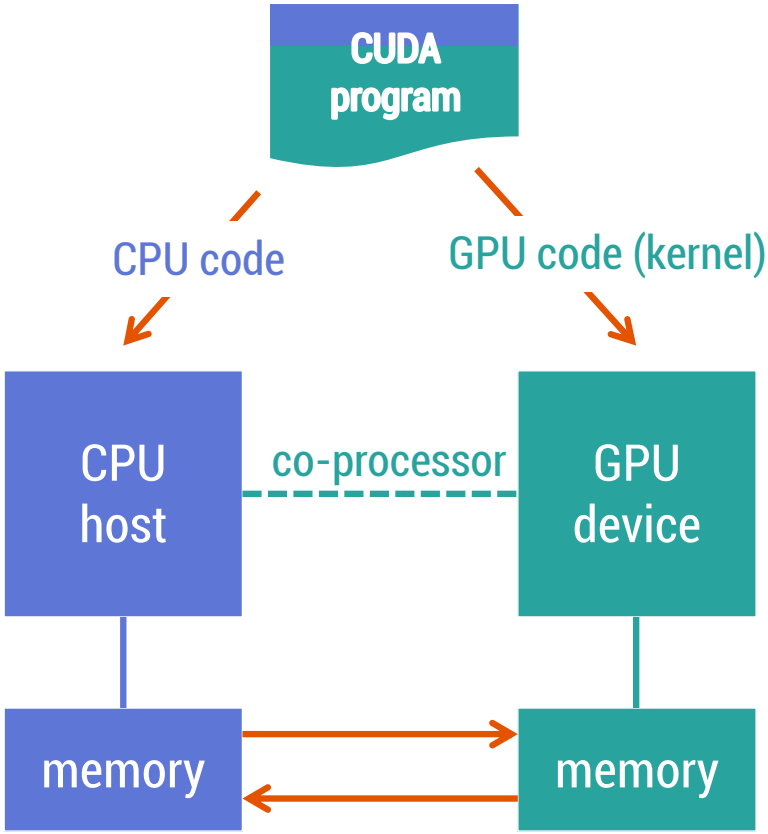


Execution of a Cuda program



1. GPU memory allocation
   cudaMalloc
2. data transfer CPU → GPU
   cudaMemcpy
3. kernel execution
4. data transfer GPU → CPU
   cudaMemcpy

Heterogeneous computation




```
void rgb2grey(uchar4 *hcimg, uchar *hgimg, int isize) {
       int bsize = 256;
       int gsize = ((isize + bsize -1) /bsize);
       int numc = isize*sizeof(uchar4);
       int numg = isize*sizeof(unsigned char);

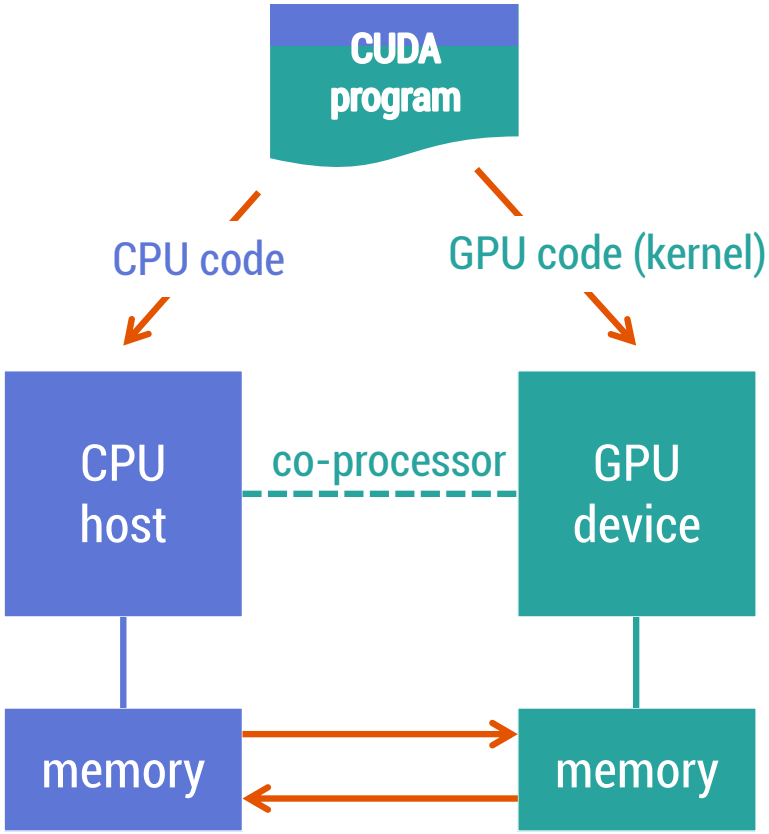
       uchar4* dcimg;
       unsigned char *dgimg;
       cudaMalloc((void **)&dcimg, numc);
       cudaMemcpy(dcimg, hcimg, numc, cudaMemcpyHostToDevice);
       cudaMalloc((void **)&gdimg, numg) ;
       color2grey<<<gsize, bsize>>>(dcimg , dgimg , isize);

       cudaMemcpy(hgimg, dgimg, numg, cudaMemcpyDeviceToHost);
       cudaFree(dcimg) ; cudaFree(dgimg) ;
}

 __global__
 void color2grey(uchar4 *dcimg, uchar *dgimg, int isize){
        int index = blockIdx.x * blockDim.x + threadIdx.x;
        uchar4 p;
        if (index < isize)){
             p = dcimg[index];
             dgimg[index] =
             (299*p.x + 587*p.y + 114*p.z) / 1000;
        }
 }
```

1 GPU memory allocation

Heterogeneous computation




```
void rgb2grey(uchar4 *hcimg, uchar *hgimg, int isize) {
       int bsize = 256;
       int gsize = ((isize + bsize -1) /bsize);
       int numc = isize*sizeof(uchar4);
       int numg = isize*sizeof(unsigned char);

       uchar4* dcimg;
       unsigned char *dgimg;
       cudaMalloc((void **)&dcimg, numc);
       cudaMemcpy(dcimg, hcimg, numc, cudaMemcpyHostToDevice);
       cudaMalloc((void **)&gdimg, numg) ;
       color2grey<<<gsize, bsize>>>(dcimg , dgimg , isize);

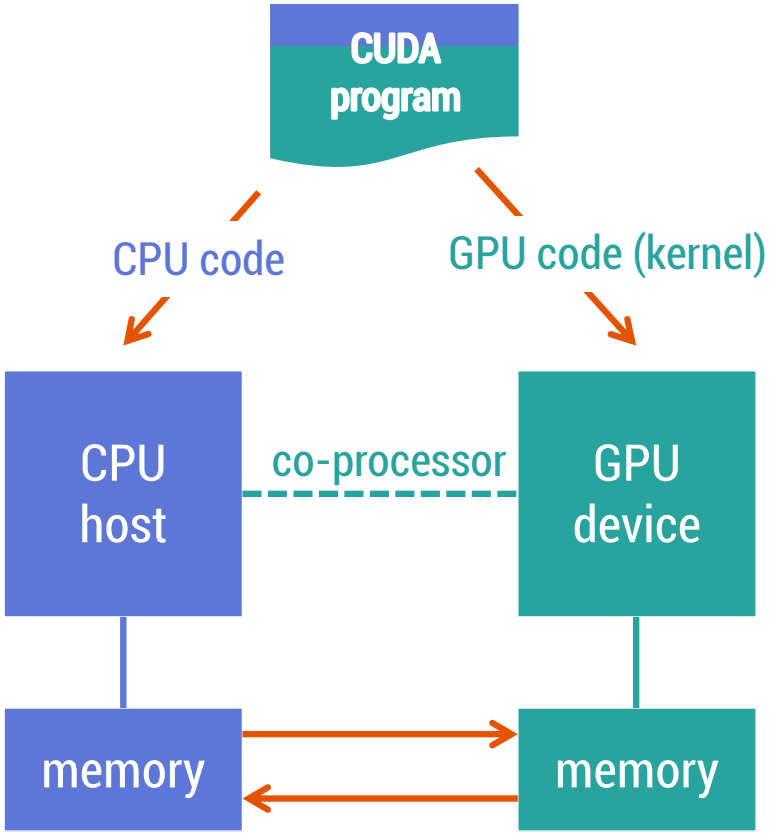
       cudaMemcpy(hgimg, dgimg, numg, cudaMemcpyDeviceToHost);
       cudaFree(dcimg) ; cudaFree(dgimg) ;
}

__global__
void color2grey(uchar4 *dcimg, uchar *dgimg, int isize){
       int index = blockIdx.x * blockDim.x + threadIdx.x;
       uchar4 p;
       if (index < isize)){
            p = dcimg[index];
            dgimg[index] =
            (299*p.x + 587*p.y + 114*p.z) / 1000;
       }
}
```

2

data transfer CPU → GPU

Heterogeneous computation



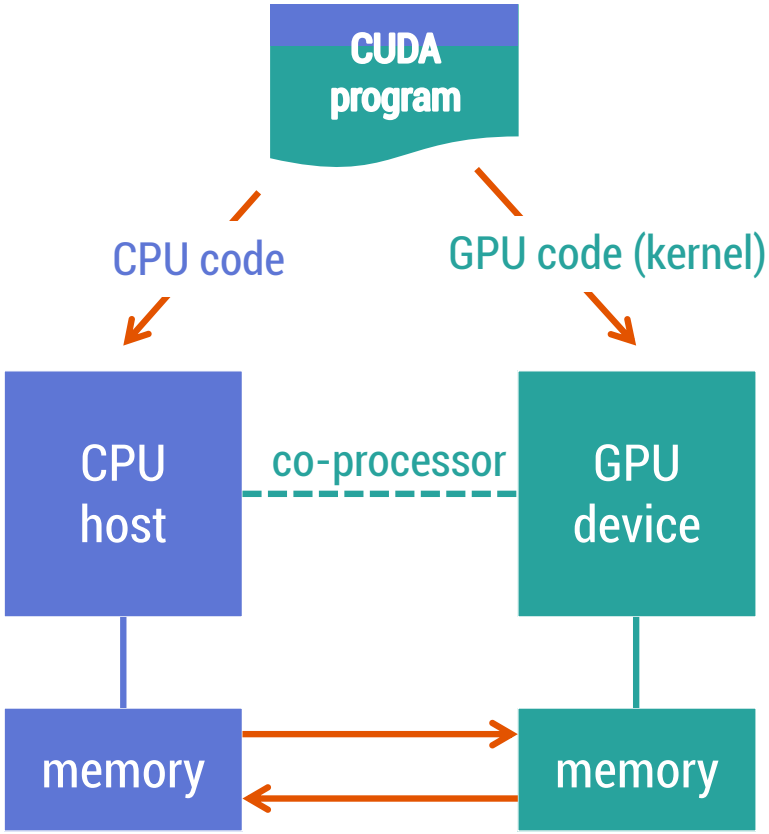

```
void rgb2grey(uchar4 *hcimg, uchar *hgimg, int isize) {
       int bsize = 256;
       int gsize = ((isize + bsize -1) /bsize);
       int numc = isize*sizeof(uchar4);
       int numg = isize*sizeof(unsigned char);

       uchar4* dcimg;
       unsigned char *dgimg;
       cudaMalloc((void **)&dcimg, numc);
       cudaMemcpy(dcimg, hcimg, numc, cudaMemcpyHostToDevice);
       cudaMalloc((void **)&gdimg, numg) ;

3      color2grey<<<gsize, bsize>>>(dcimg , dgimg , isize);
kernel execution
       cudaMemcpy(hgimg, dgimg, numg, cudaMemcpyDeviceToHost);
       cudaFree(dcimg) ; cudaFree(dgimg) ;
}

 __global__
 void color2grey(uchar4 *dcimg, uchar *dgimg, int isize){
        int index = blockIdx.x * blockDim.x + threadIdx.x;
        uchar4 p;
        if (index < isize)){
             p = dcimg[index];
             dgimg[index] =
             (299*p.x + 587*p.y + 114*p.z) / 1000;
        }
 }
```

Heterogeneous computation




```
void rgb2grey(uchar4 *hcimg, uchar *hgimg, int isize) {
       int bsize = 256;
       int gsize = ((isize + bsize -1) /bsize);
       int numc = isize*sizeof(uchar4);
       int numg = isize*sizeof(unsigned char);

       uchar4* dcimg;
       unsigned char *dgimg;
       cudaMalloc((void **)&dcimg, numc);
       cudaMemcpy(dcimg, hcimg, numc, cudaMemcpyHostToDevice);
       cudaMalloc((void **)&gdimg, numg) ;

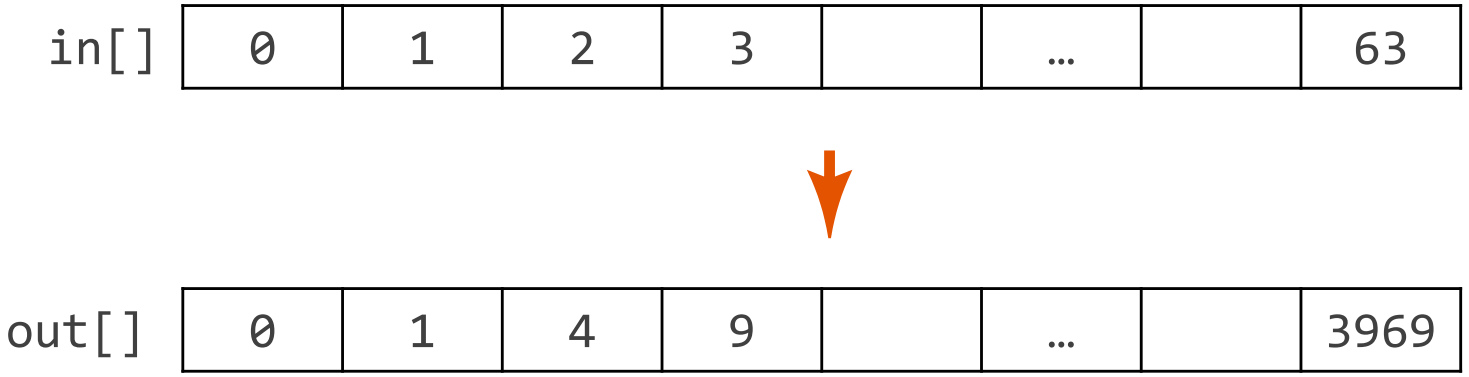
       color2grey<<<gsize, bsize>>>(dcimg , dgimg , isize);

4      cudaMemcpy(hgimg, dgimg, numg, cudaMemcpyDeviceToHost);
       cudaFree(dcimg) ; cudaFree(dgimg) ;
}
```

data transfer GPU → CPU

```
__global__
void color2grey(uchar4 *dcimg, uchar *dgimg, int isize){
       int index = blockIdx.x * blockDim.x + threadIdx.x;
       uchar4 p;
       if (index < isize)){
            p = dcimg[index];
            dgimg[index] =
            (299*p.x + 587*p.y + 114*p.z) / 1000;
       }
}
```

Compute `out[]` with `out[i] = in[i]*in[i]`



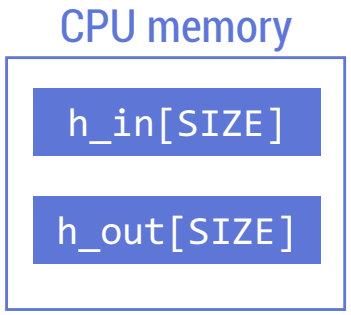
sequential code (to be run on a standard processor)

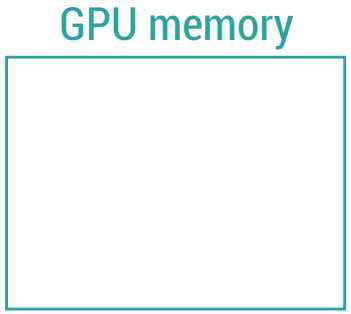
```
for (i=0 ; i<64 ; i++) {
    out[i] = in[i] * in[i];
}
```


```
#define ARRAY_SIZE 64

int main(int argc, char **argv){
    int bytes = ARRAY_SIZE * sizeof(int);

    // generate input data on the CPU
    int h_in[ARRAY_SIZE];
    for (int i=0; i<ARRAY_SIZE ; i++)
        h_in[i] = i;
    int h_out[ARRAY_SIZE];
```





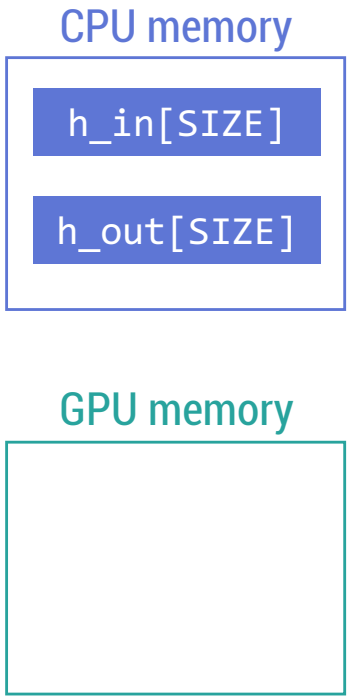

```
#define ARRAY_SIZE 64

int main(int argc, char **argv){
    int bytes = ARRAY_SIZE * sizeof(int);

    // generate input data on the CPU
    int h_in[ARRAY_SIZE];
    for (int i=0; i<ARRAY_SIZE ; i++)
        h_in[i] = i;
    int h_out[ARRAY_SIZE];

    // declare pointers to the GPU memory
    int *d_in, *d_out;
```

1 GPU memory allocation




```
#define ARRAY_SIZE 64

int main(int argc, char **argv){
    int bytes = ARRAY_SIZE * sizeof(int);

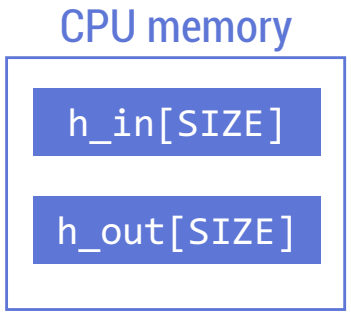
    // generate input data on the CPU
    int h_in[ARRAY_SIZE];
    for (int i=0; i<ARRAY_SIZE ; i++)
        h_in[i] = i;
    int h_out[ARRAY_SIZE];

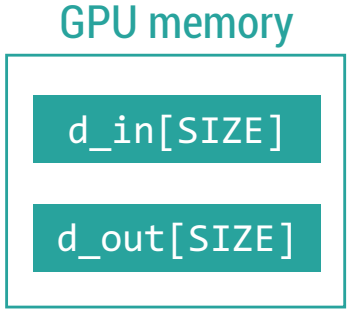
    // declare pointers to the GPU memory
    int *d_in, *d_out;

    // allocate GPU memory
    cudaMalloc((void **) &d_in, bytes);
    cudaMalloc((void **) &d_out, bytes);

    // ... (continued on next slide)
```

1 GPU memory allocation

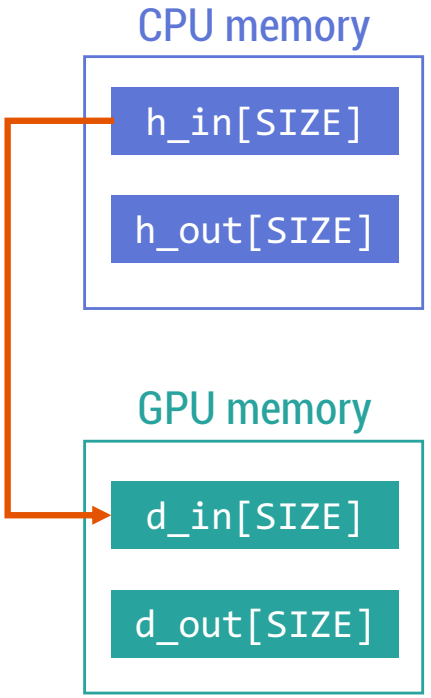





```
// ... continued from previous slide

// transfer the input array from the CPU to the GPU memory
cudaMemcpy(d_in, h_in, bytes, cudaMemcpyHostToDevice);
```

2 data transfer CPU → GPU

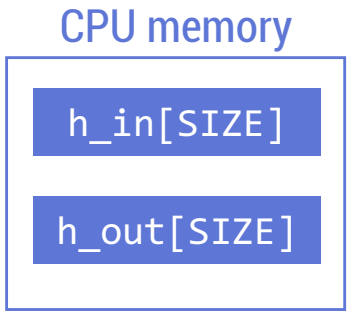


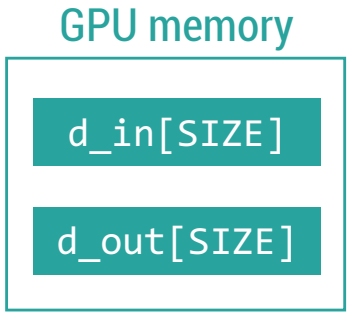

```
// ... continued from previous slide

// transfer the input array from the CPU to the GPU memory
cudaMemcpy(d_in, h_in, bytes, cudaMemcpyHostToDevice);

// start the kernel execution
square<<< 1, ARRAY_SIZE >>>(d_out, d_in);
```

3 kernel execution





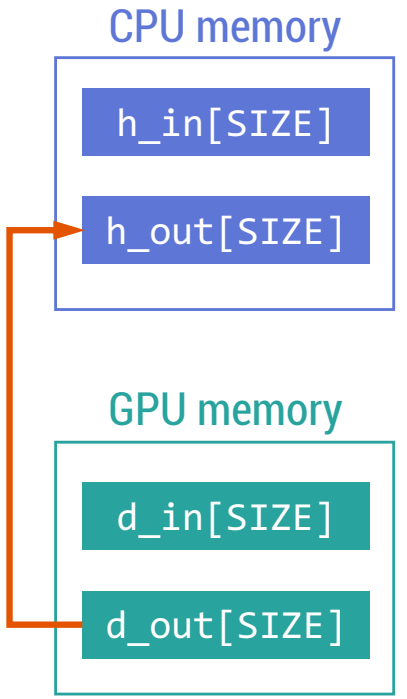

```
// ... continued from previous slide

// transfer the input array from the CPU to the GPU memory
cudaMemcpy(d_in, h_in, bytes, cudaMemcpyHostToDevice);

// start the kernel execution
square<<< 1, ARRAY_SIZE >>>(d_out, d_in);

// transfer the output array from the GPU to the CPU memory
cudaMemcpy(h_out, d_out, bytes, cudaMemcpyDeviceToHost);
```

4 data transfer GPU → CPU




```
// ... continued from previous slide

// transfer the input array from the CPU to the GPU memory
cudaMemcpy(d_in, h_in, bytes, cudaMemcpyHostToDevice);

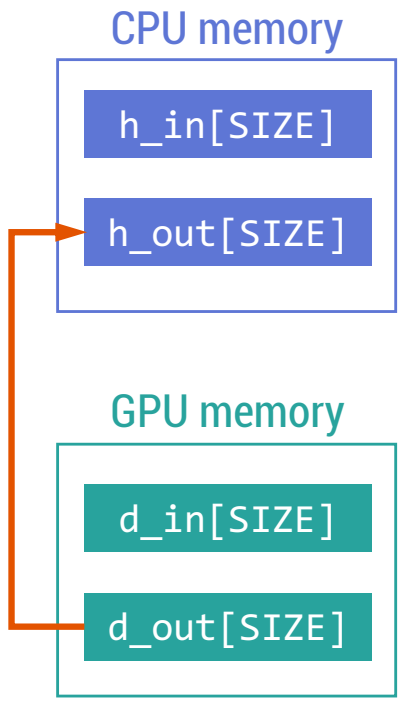
// start the kernel execution
square<<< 1, ARRAY_SIZE >>>(d_out, d_in);

// transfer the output array from the GPU to the CPU memory
cudaMemcpy(h_out, d_out, bytes, cudaMemcpyDeviceToHost);

// display the output array
for (int i=0 ; i<ARRAY_SIZE; i++) {
    printf("%d\n", h_out[i]);
}

// free GPU memory
cudaFree(d_in);
cudaFree(d_out);

return 0;
}
```



Kernel:

```
__global__ void square(int *d_out, int *d_in){
    int idx = threadIdx.x;
    int f = d_in[idx];
    d_out[idx] = f * f;
}
```

`__global__` indicates that the function is a kernel (to be compiled for and run on the GPU)

`idx` and `f` thread private variables

`threadIdx.x` thread identifier (keyword)

Cuda library

```
__host__ __device__ cudaError_t cudaMalloc ( void** devPtr , size_t size )
```

Allocate memory on the device.

docs.nvidia.com

Parameters

`devPtr`
- Pointer to allocated device memory

`size`
- Requested allocation size in bytes

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorMemoryAllocation

Description

Allocates `size` bytes of linear memory on the device and returns in `*devPtr` a pointer to the allocated memory. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. cudaMalloc() returns cudaErrorMemoryAllocation in case of failure.

Cuda library

```
__host__ cudaError_t cudaMemcpy ( void* dst , const void* src , size_t count ,
cudaMemcpyKind kind )
```

Copies data between host and device.

Parameters

`dst`
- Destination memory address

`src`
- Source memory address

`count`
- Size in bytes to copy

`kind`
- Type of transfer

Returns

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidMemcpyDirection

Description

Copies `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `dst`, where `kind` specifies the direction of the copy, and must be one of

Cuda library

__host__ __device__ cudaError_t cudaFree (void* devPtr)

Frees memory on the device.

Parameters

`devPtr`

- Device pointer to memory to free

Returns

cudaSuccess, cudaErrorInvalidValue

Description

Frees the memory space pointed to by `devPtr`, which must have been returned by a previous call to one of the following memory allocation APIs - cudaMalloc(), cudaMallocPitch(),

GPU or CPU code?

`__global__ void f()`

- function run on the GPU (*kernel*)
- must be invoked from the CPU

`__device__ float g()`

- function run on the GPU
- must be invoked from the GPU

`__host__ int h()`

- function run on the CPU
- must be invoked from the CPU

Default: `__host__`

Configuring parallelism

number of threads

```
square<<< 1 , 64 >>>(d_out, d_in)
```

number of blocks

number of threads per block

Threads are organized into blocks.

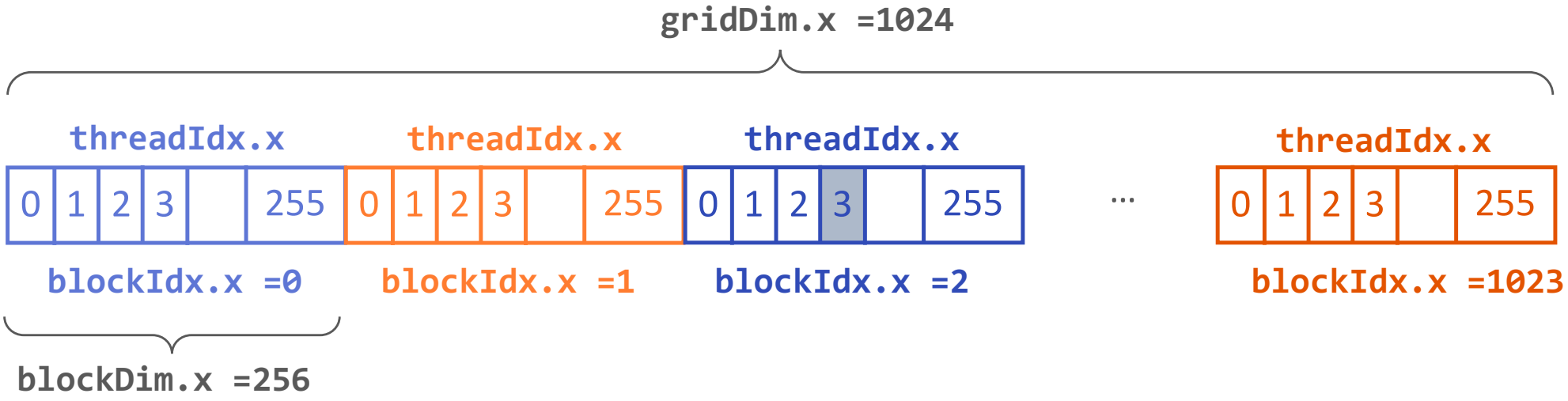
256 threads ?

```
square<<<1, 256>>>(d_out, d_in)
square<<<2, 128>>>(d_out, d_in)
square<<<4, 64>>>(d_out, d_in)
```

Configuring parallelism

```
#define N 262144        // desired number of threads
int block_dim = 256;
int num_blocks = N / block_dim;     // = 1024
kernel<<<num_blocks, block_dim>>>(…)
```

262 144 = 256 * 1024




```
index = blockIdx.x * blockDim.x + threadIdx.x
```

index = (2) * (256) + (3) = 515

Back to the example with arrays of 262144 elements:

```
#define ARRAY_SIZE 262144
int main(int argc, char **argv){
    int bytes = ARRAY_SIZE * sizeof(int);
    int h_in[ARRAY_SIZE];
    for (int i=0; i<ARRAY_SIZE ; i++) h_in[i] = i;
    int h_out[ARRAY_SIZE];
    int *d_in, *d_out;
    cudaMalloc((void **) &d_in, bytes);
    cudaMalloc((void **) &d_out, bytes);
    cudaMemcpy(d_in, h_in, bytes, cudaMemcpyHostToDevice);
    int block_dim = 256;
    int num_blocks = N / block_dim;
    square<<<num_blocks, block_dim>>>(d_out, d_in)
    cudaMemcpy(h_out, d_out, bytes, cudaMemcpyDeviceToHost);
    cudaFree(d_in);cudaFree(d_out);
    return 0;
   }

__global__ void square(int *d_out, int *d_in){
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int f = d_in[idx];
    d_out[idx] = f * f;
}
```

Configuring parallelism

```
#define N 262148
int block_dim = 256;
int num_blocks = N / block_dim;
if (num_blocks * block_dim < N)
        num_blocks++;
square<<<num_blocks, block_dim>>>(d_out, d_in)
```

262 148 = 256 * 1024 + 4

➢ 1025 blocks

```
#define N 262148
int block_dim = 256;
int num_blocks = N/block_dim + ( (N % block_dim) != 0);
square<<<num_blocks, block_dim>>>(d_out, d_in)
```

```
#define N 262148
int block_dim = 256;
int num_blocks = (N + block_dim - 1) / block_dim;
square<<<num_blocks, block_dim>>>(d_out, d_in)
```

➢ 256*1025 = 262 400 threads are created

Back to the example with arrays of 262148 elements:

```
#define ARRAY_SIZE 262148
int main(int argc, char **argv){
    int bytes = ARRAY_SIZE * sizeof(int);
    int h_in[ARRAY_SIZE];
    for (int i=0; i<ARRAY_SIZE ; i++) h_in[i] = i;
    int h_out[ARRAY_SIZE];
    int *d_in, *d_out;
    cudaMalloc((void **) &d_in, bytes);
    cudaMalloc((void **) &d_out, bytes);
    cudaMemcpy(d_in, h_in, bytes, cudaMemcpyHostToDevice);
    int block_dim = 256;
    int num_blocks = (N + block_dim - 1) / block_dim;
    square<<<num_blocks, block_dim>>>(d_out, d_in, ARRAY_SIZE)
    cudaMemcpy(h_out, d_out, bytes, cudaMemcpyDeviceToHost);
    cudaFree(d_in);cudaFree(d_out);
    return 0;
   }

__global__ void square(int *d_out, int *d_in, int N){
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N){
        int f = d_in[idx];
        d_out[idx] = f * f;
    }
}
```

Write the following function

```
void rgb2grey(uchar4 *hcimg, uchar * hgimg, int isize);
```

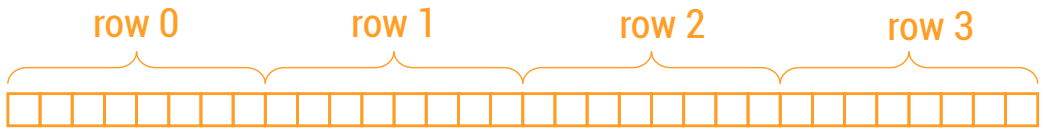
that transforms an `isize`-pixel colour image into a greyscale image.

A colour pixel `c` (`uchar4`) is transformed into a greyscale pixel `g` (`uchar`) with the following formula:

```
g = (299 * c.x + 587 * c.y + 114 * c.z) / 1000;
```

```
struct uchar4 {  unsigned char x ;   // red channel
                 unsigned char y ;   // green channel
                 unsigned char z ;   // blue channel
                 unsigned char w ; } // alpha channel
```

Images are represented by one-dimension arrays in which pixels are stored row by row.




```
void rgb2grey(uchar4 *hcimg, uchar *hgimg, int isize) {
    int bsize = 256;
    int gsize = ((isize + bsize -1) /bsize);
    int numc = isize*sizeof(uchar4);
    int numg = isize*sizeof(unsigned char);

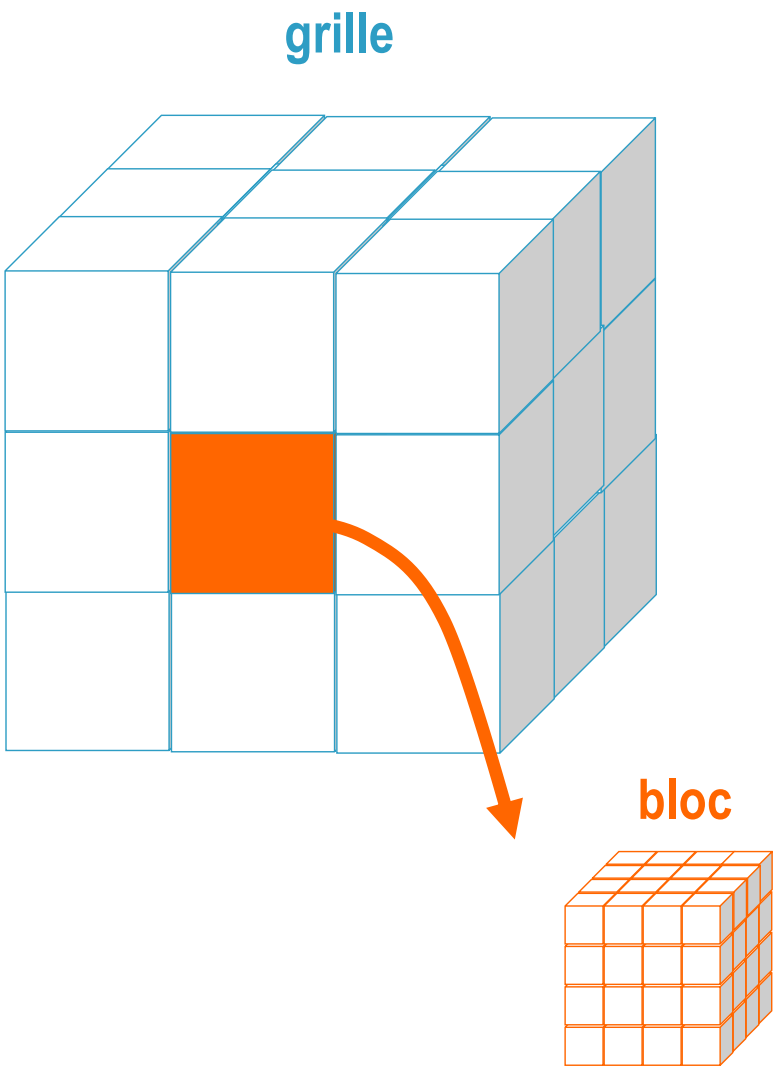
    uchar4* dcimg;
    unsigned char *dgimg;
    cudaMalloc((void **)&dcimg, numc);
    cudaMemcpy(dcimg, hcimg, numc, cudaMemcpyHostToDevice);
    cudaMalloc((void **)&gdimg, numg) ;

    color2grey<<< gsize, bsize >>>(dcimg , dgimg , isize);

    cudaMemcpy(hgimg, dgimg, numg, cudaMemcpyDeviceToHost);
    cudaFree(dcimg) ; cudaFree(dgimg) ;
}
```


```
__global__
void color2grey(uchar4 *dcimg, uchar *dgimg, int isize){
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    uchar4 p;
    if (index < isize){
       p = dcimg[index];
       dgimg[index] = (299*p.x + 587*p.y + 114*p.z) / 1000;
    }
}
```

Configuring parallelism



`threadIdx` : thread identifier within the block it belongs to

`threadIdx.x`
`threadIdx.y`
`threadIdx.z`

`blockDim` : block size (number of threads on each dimension)

`blockIdx` : block identifier

`gridDim` : grid size (number of blocks on each dimension)

Configuring parallelism

```
kernel<<< num_blocks, #num_threads>>> (…)
```

dim3(x,y,z)

Unused dimensions → 1

- `square<<<dim3(1,1,1), dim3(64,1,1)>>>`
- ```
  dim3 dim_grid( (n/256+(n%256!=0)),1,1 );
  dim3 dim_block(256,1,1);
  kernel<<<dim_grid, dim_block>>>(…);
  ```
- short form: `dim3(w,1,1) = w`

Limits

- `gridDim.x (.y, .z)` must range within 1 - 65 536
- number of threads per block limited to 1024
  - `blockDim`=(512,1,1) and (8,16,4) are valid, but not (32,32,2)
  ```

Write the following function

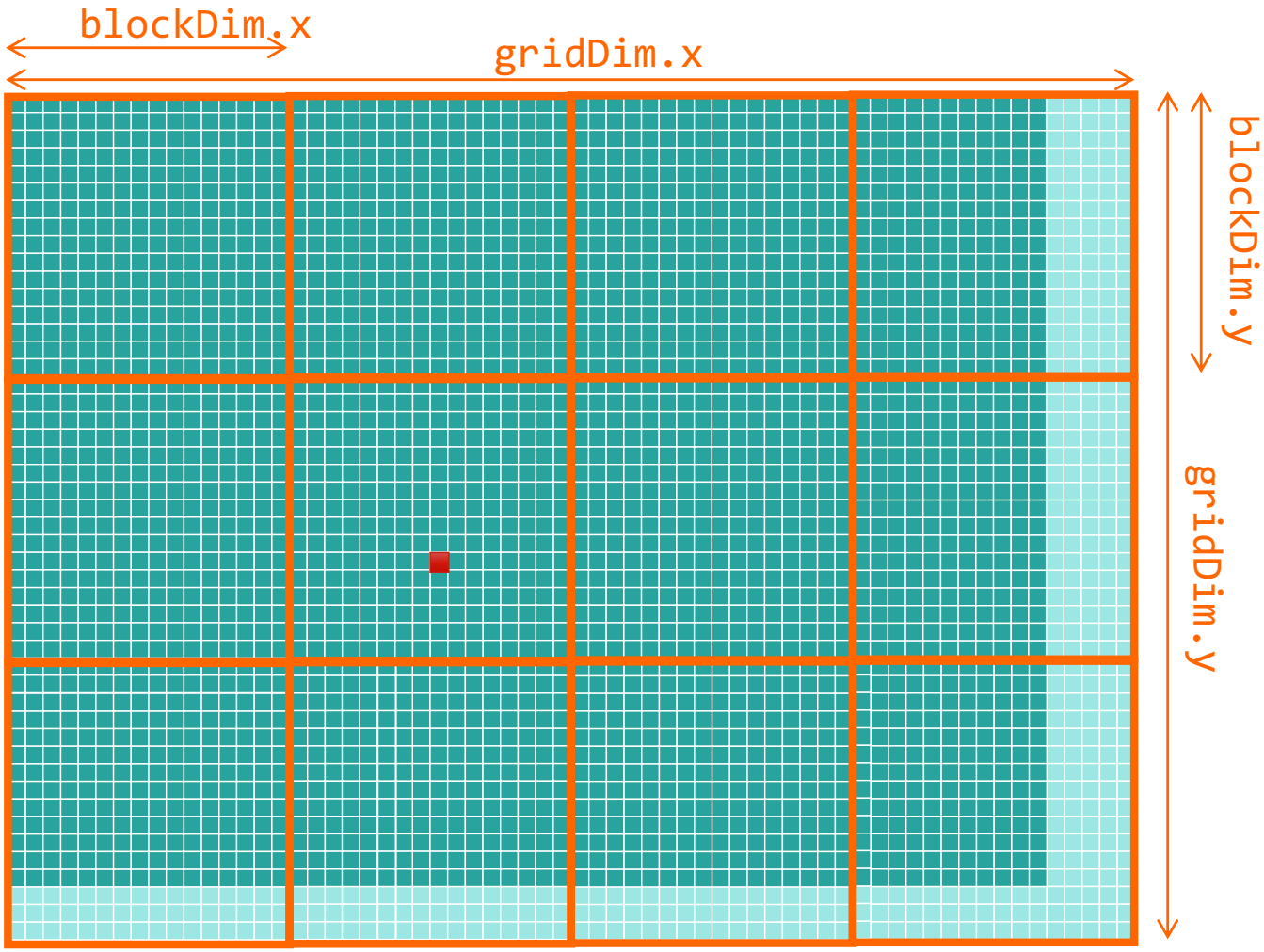
```
void matmul(float *A, float *B, float *C, int N);
```

that computes into C the product of matrices A and B.

We consider N × N matrices that are stored in memory row by row. Each thread must compute one element of C.

sequential code

```
for (i=0 ; i<N ; i++){
    for (j=0 ; j<N ; j++){
        C[i][j] = 0;
        for (k=0 ; k<N ; k++)
            C[i][j] += A[i][k] * B[k][j];
    }
}
```



Which is the element computed by a given thread?

row = `blockIdx.y * blockDim.y + threadIdx.y`

column = `blockIdx.x * blockDim.x + threadIdx.x`


```
#define BSIZE 16
void matmul(float *A, float *B, float *C, int N) {
    int bytes = N*N*sizeof(float);
    int numb = N / BSIZE;
    if (N % BSIZE != 0) numb++;
    dim3 gsize(numb, numb, 1);
    dim3 bsize(BSIZE , BSIZE , 1);

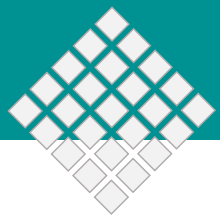
    float *dA, *dB, *dC;
    cudaMalloc((void **)&dA, bytes) ;
    cudaMalloc((void **)&dB, bytes) ;
    cudaMalloc((void **)&dC,bytes) ;
    cudaMemcpy(dA, A, bytes, cudaMemcpyHostToDevice) ;
    cudaMemcpy(dB, B, bytes, cudaMemcpyHostToDevice) ;

    matmulkernel<<<gsize,bsize>>>(dA,dB,dC,N);

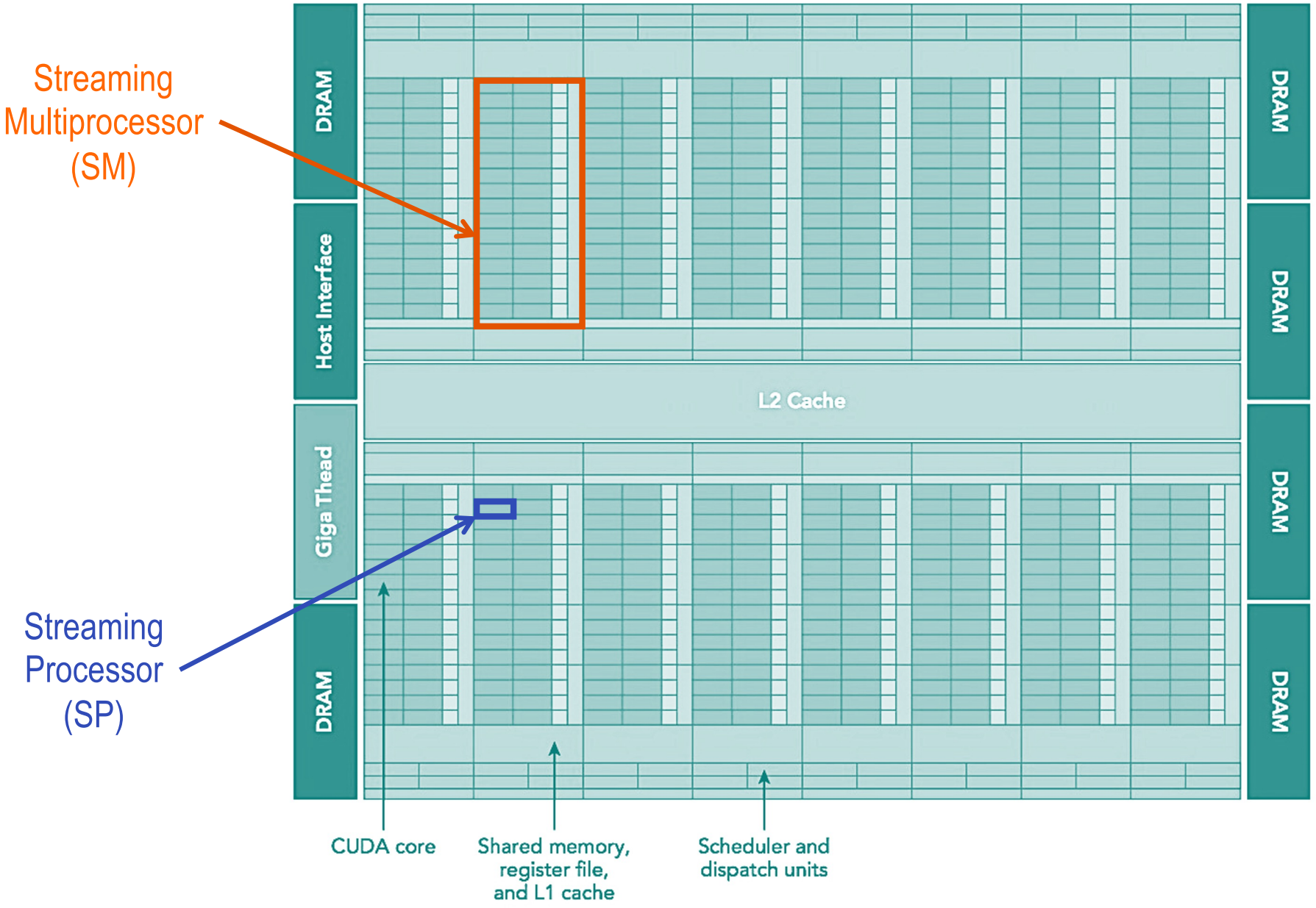
    cudaMemcpy(C, dC, bytes, cudaMemcpyDeviceToHost) ;
    cudaFree(d_) ; cudaFree(dB) ; cudaFree(dC) ;
}
```


```
__global__
void matmulkernel(float *dA, float *dB, float *dC, int N){
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if ((row<N) && (col<N)){
        float res = 0;
        for (int k=0; k<N ; k++)
            res += dA[row*N+k] * dB[k*N+col];
        dC[row*N + col] = res;
    }
}
```

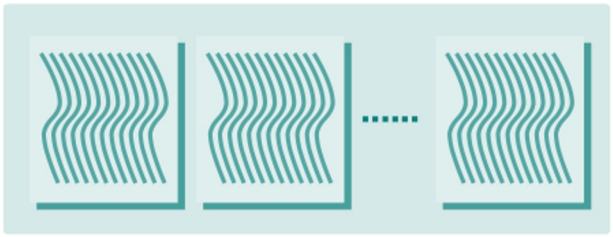
Execution model



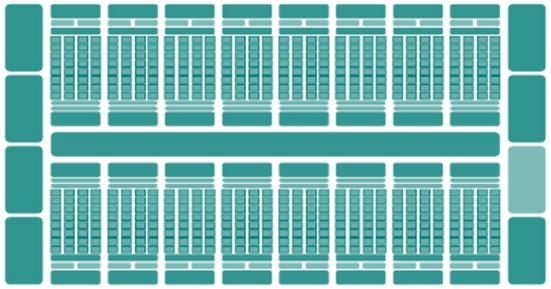
GPU Architecture



Mapping of threads onto cores (SPs)

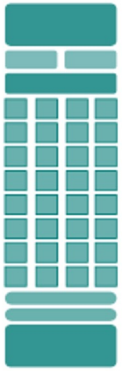


Grid



Device

Thread Block



SM

- a block is mapped to a single SM
- several blocks on the same SM
- less than 1536 threads per SM

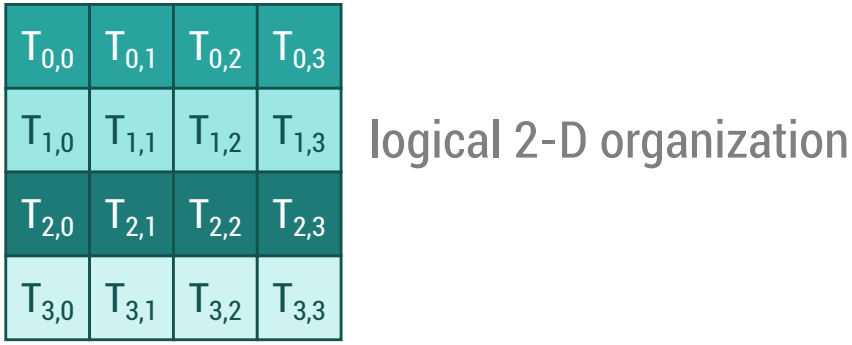
Thread

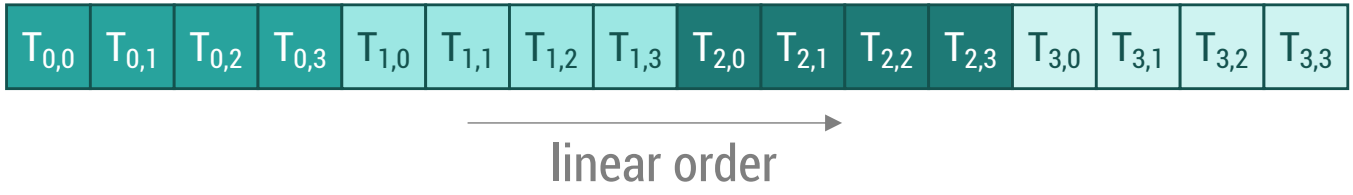
CUDA Core

Warps

A block is split into *warps* of 32 threads

- 32 threads with consecutive numbers (0-31, 32-63, …)
  - 1-dimension blocks: based on `threadIdx.x`
  - 2- or 3-dimension blocks?





Thread scheduling

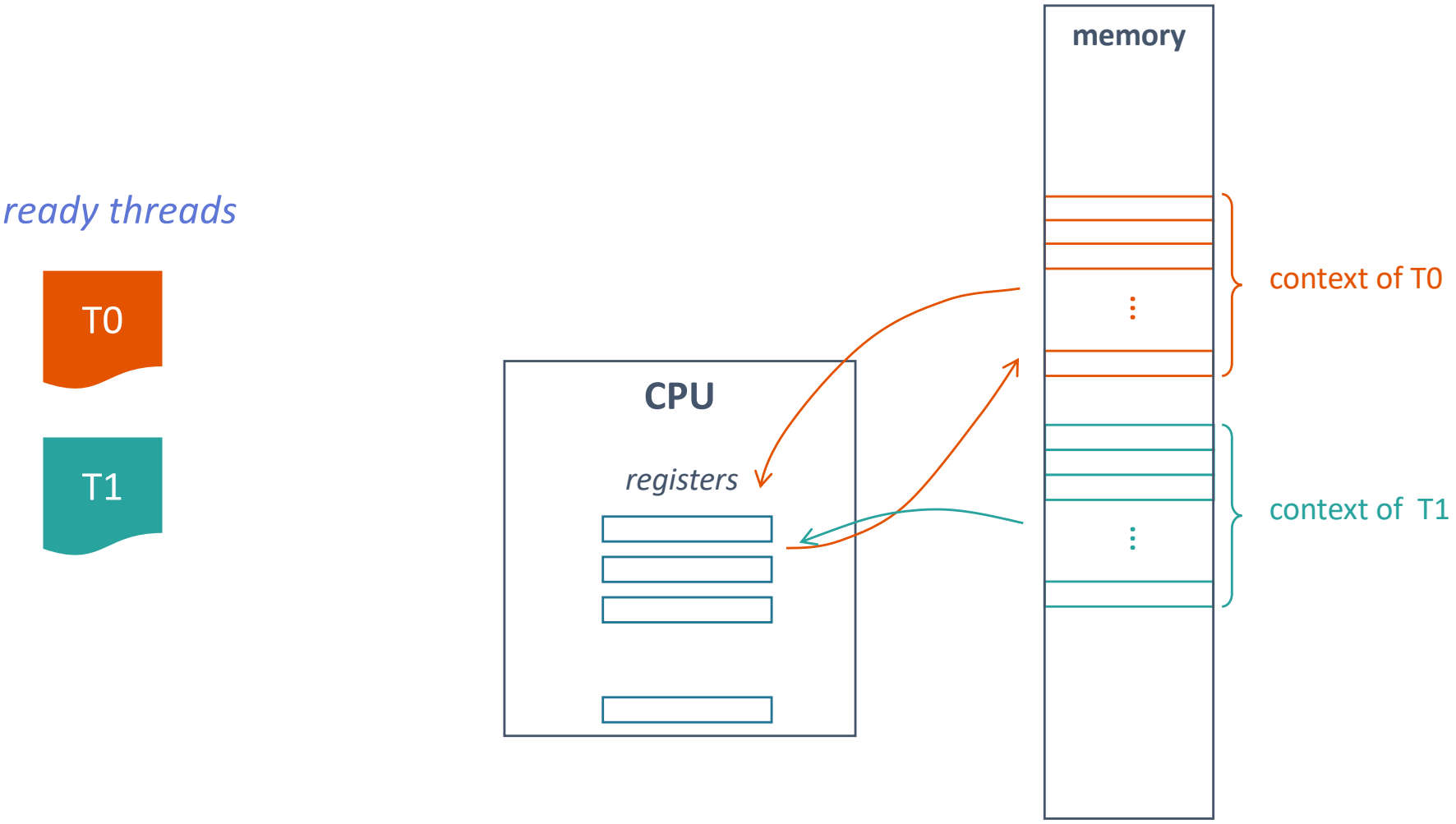
A block is split into *warps* of 32 threads

- 32 threads with consecutive numbers (0-31, 32-63, ...)
- all the threads of a warp are executed in lockstep
  (SIMT: Single Instruction - Multiple Threads)
  - a single instruction at a time is fetched and decoded, then executed by all the threads of the warp

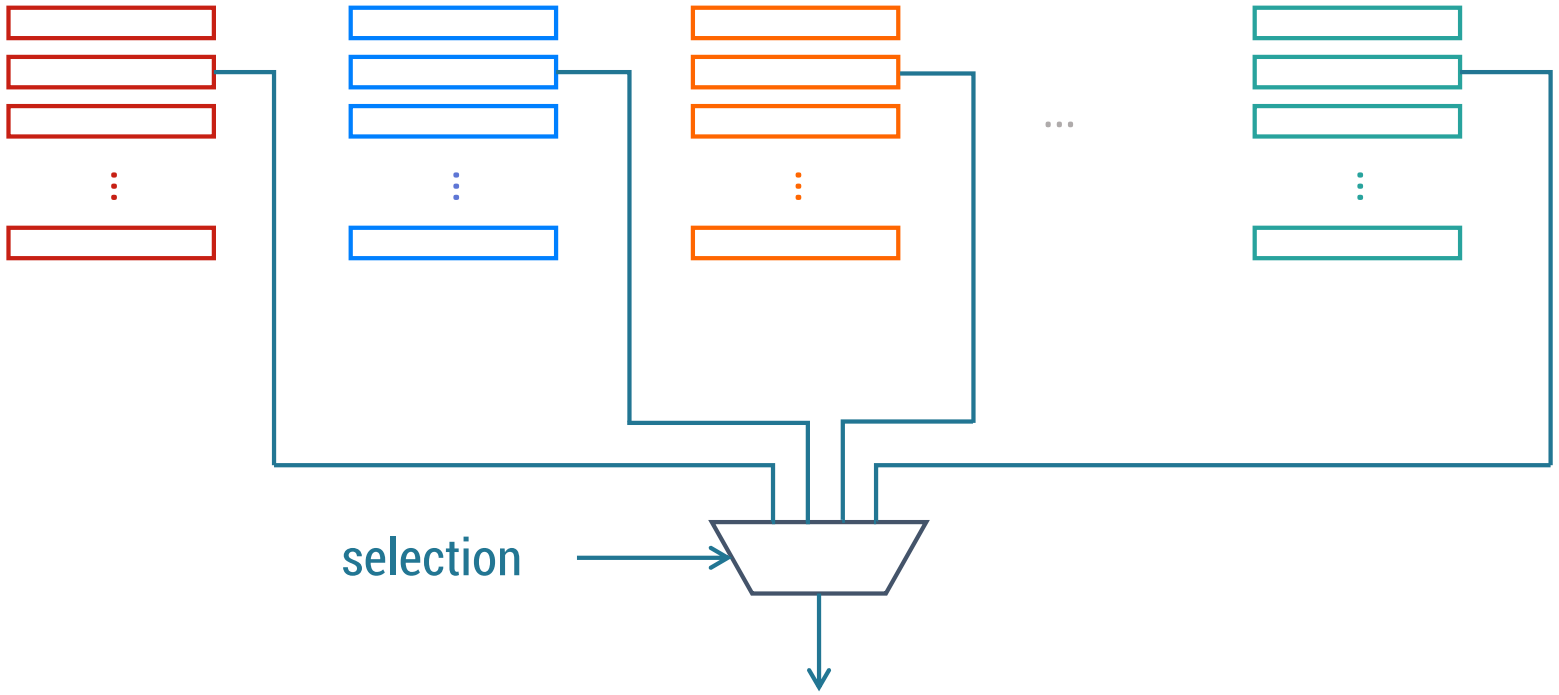
When an instruction is not ready to be executed or has a long latency, another warp is scheduled

- this way, long instruction latencies are hidden
  - memory accesses
  - floating point operations

Context switching on a CPU



Context switching on a GPU



Warp optimisation

Prefer block sizes that make full warps

- multiples of 32
- examples:

```
kernel <<< N, 1 >>>(…)
kernel <<< N/32, 32 >>>(…)
kernel <<< N/128,128 >>>(…)
```

Prefer a large number of threads per block to ensure a large number of warps on the same SM

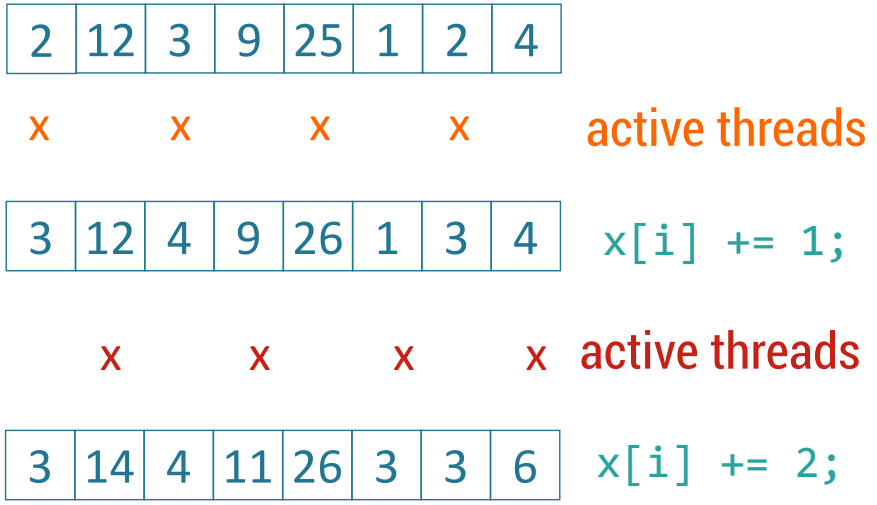
- this is how the GPU can hide long latencies and exhibit a high instruction throughput

Branch divergence

```
__global__ void odd_event(int n, int *x) {
      int i = blockIdx.x * blockDim.x + threadIdx.x;
      if ( (i & 0x01) == 0)
        x[i] = x[i] + 1;
      else
        x[i] = x[i] + 2;
}
```

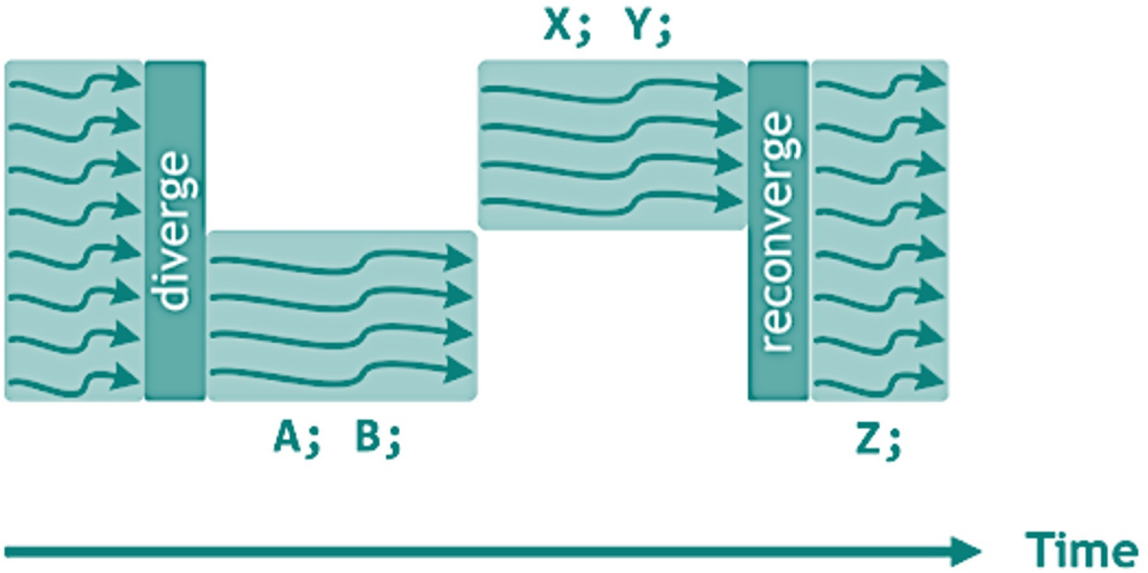
The GPU handles the control flow divergence

- it executes both paths in sequence
- *before that*: threads determine on which path they must be active (vec)



Branch divergence

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



Branch divergence

Nested branches are handled the same way

- with deep nesting, the number of inactive threads grows

The programmer usually does not need to consider branch divergence to reason on the **correctness** of the program

- however, be careful about risk of deadlocks with lock busy waiting

The programmer usually must account for branch divergence to reason on **performance**

Performance of divergent code

The compiler and the hardware can detect when all the threads in a warp follow the same path

- only this path is fetched/executed

The compiler can translate short conditional statements with predicates

- predicated instructions are executed when the predicate is true (for some of the threads) and considered as NOPs when the predicate is false (for other threads)

How many warps?

32

How many divergent warps?

32

```
#define N 1024

void init(char *vect){
    char *v;
    int bytes = N*sizeof(char);
    cudaMalloc((void **)&v, bytes);
    kinit1<<<1,N>>>(v);
    cudaMemcpy(vect, v, bytes, cudaMemcpyDeviceToHost) ;
    return 0;
}

__global__ void kinit1(char *v){
    unsigned int tid = threadIdx.x ;
    if (tid % 2 == 0)
        v[tid] = 0;
    else
        v[tid] = 1;
}
```

How many warps?

32

How many divergent warps?

0

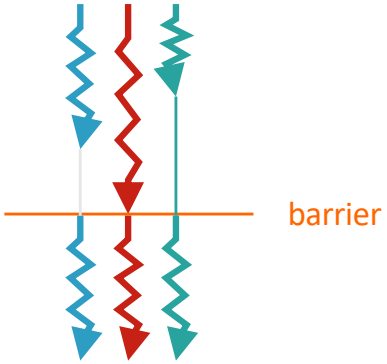
```
#define N 1024

void init(char *vect){
    char *v;
    int bytes = N*sizeof(char);
    cudaMalloc((void **)&v, bytes);
    kinit2<<<1,N>>>(v);
    cudaMemcpy(vect, v, bytes, cudaMemcpyDeviceToHost) ;
    return 0;
}

__global__ void kinit2(char *v){
    unsigned int tid = threadIdx.x ;
    unsigned int half_size = blockDim.x / 2;
    if (tid < half_size)
        v[tid*2] = 0;
    else
        v[(tid-halfsize)*2+1] = 1;
}
```

Synchronisation

Barrier




```
int idx = threadIdx.x;

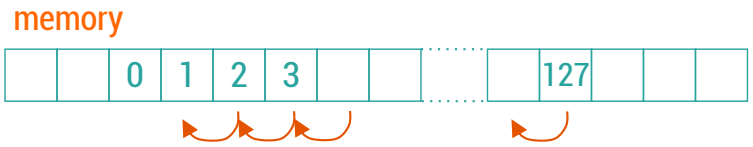
array[idx] = idx;

__syncthreads();

if (idx < 127)
        int temp = array[idx+1];
__syncthreads();

if (idx < 127)
        array[idx] = temp;
__syncthreads();
```

- synchronises threads **of the same block**
- no (easy) inter-block synchronisation



Available resources?

```
int dev_count;
cudaGetDeviceCount(&dev_count);

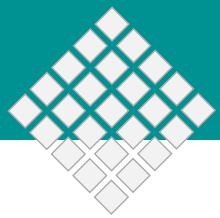
cudaDeviceProp dev_prop;
for (i=0; i<dev_count ; i++)
       cudaGetDeviceProperties(&dev_prop, i);

// dev_prop.multiProcessorCount => number of SMs
// dev_prop.clockRate     => clock frequency

// dev_prop.maxThreadsPerBlock   (1024)
// dev_prop.maxThreadsDim[0] => over dimension x
// dev_prop.maxGridSize[1] => over dimension y

// dev_prop.warpSize => number of threads per warp
// …
```

GPU memories



Memory accesses hinder performance

Example: a kernel for matrix multiplication

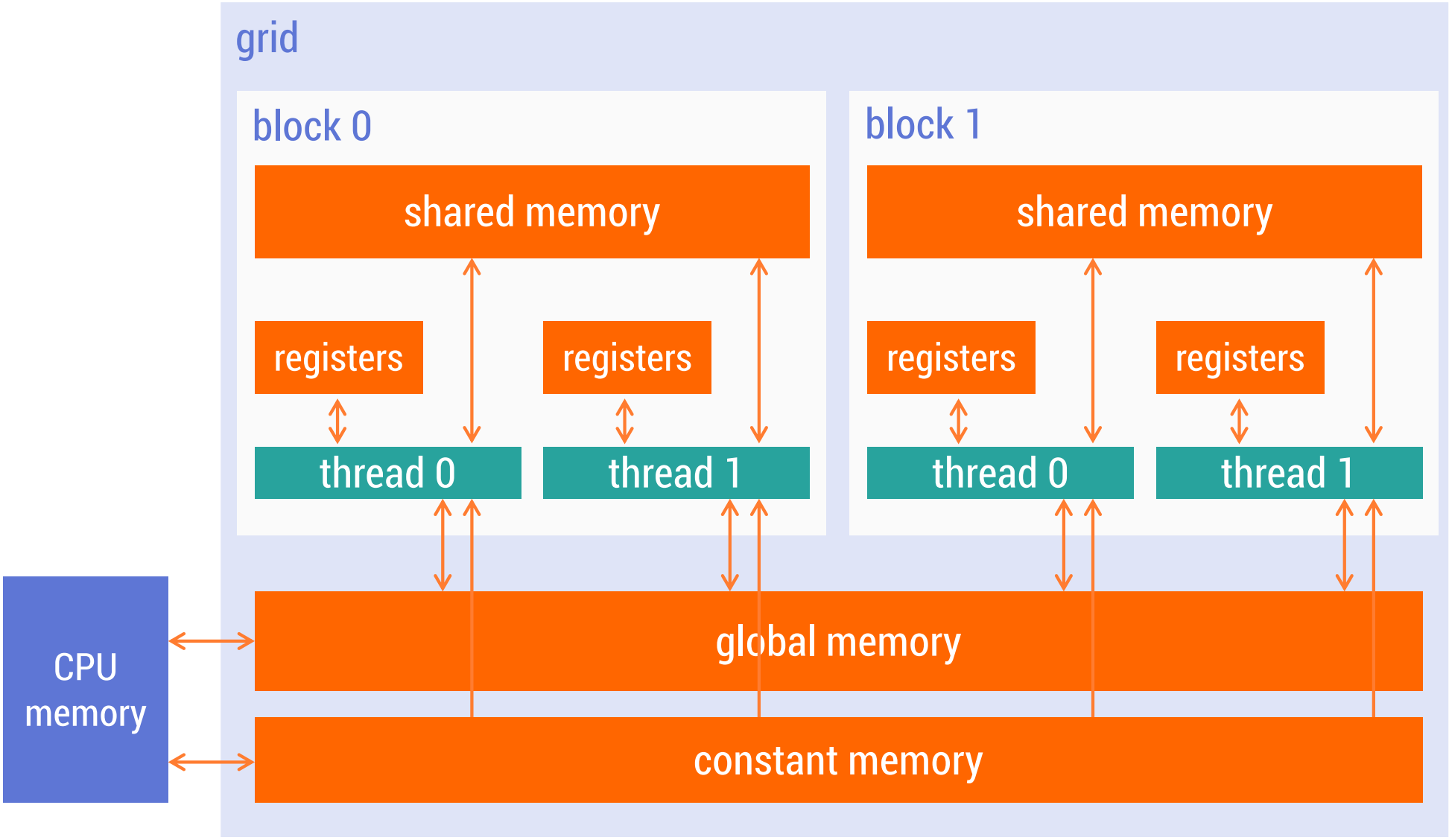
```
for (k=0 ; k<N; k++)
      res += d_A[i*N + k] * d_B[k*N + j];
```

two memory accesses

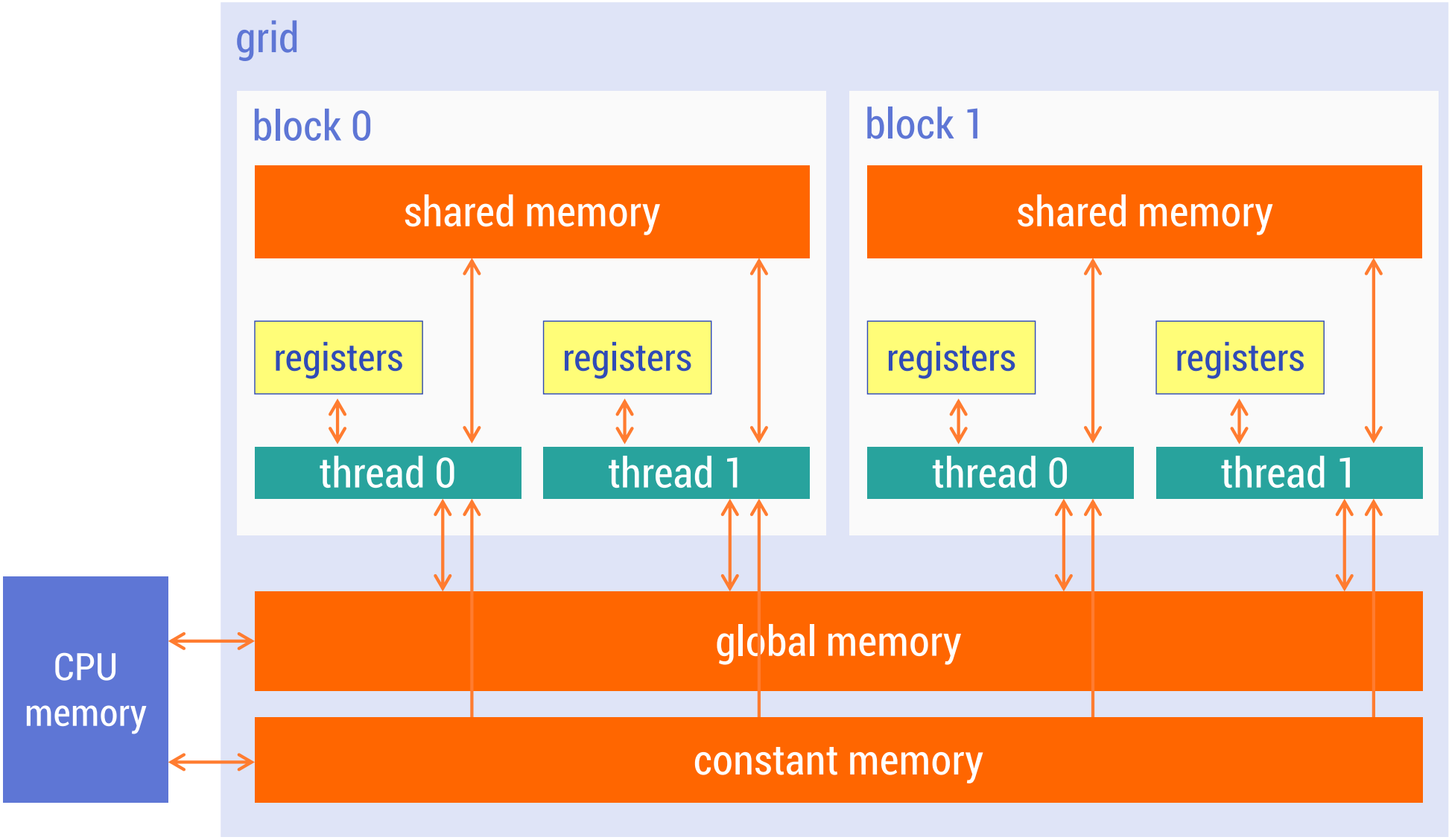
two floating-point operations

- CGMA = Compute to Global Memory Access ratio
  - in the example above = 1:1
- impact on performance
  - memory bandwidth = 200 GO/s ➢ 50 G(float)/s
    ➢ 50 GFLOPS (to be compared to the peak 1500 GFLOPS)
  - conclusion: the CGMA ratio must be increased

GPU memories



GPU memories

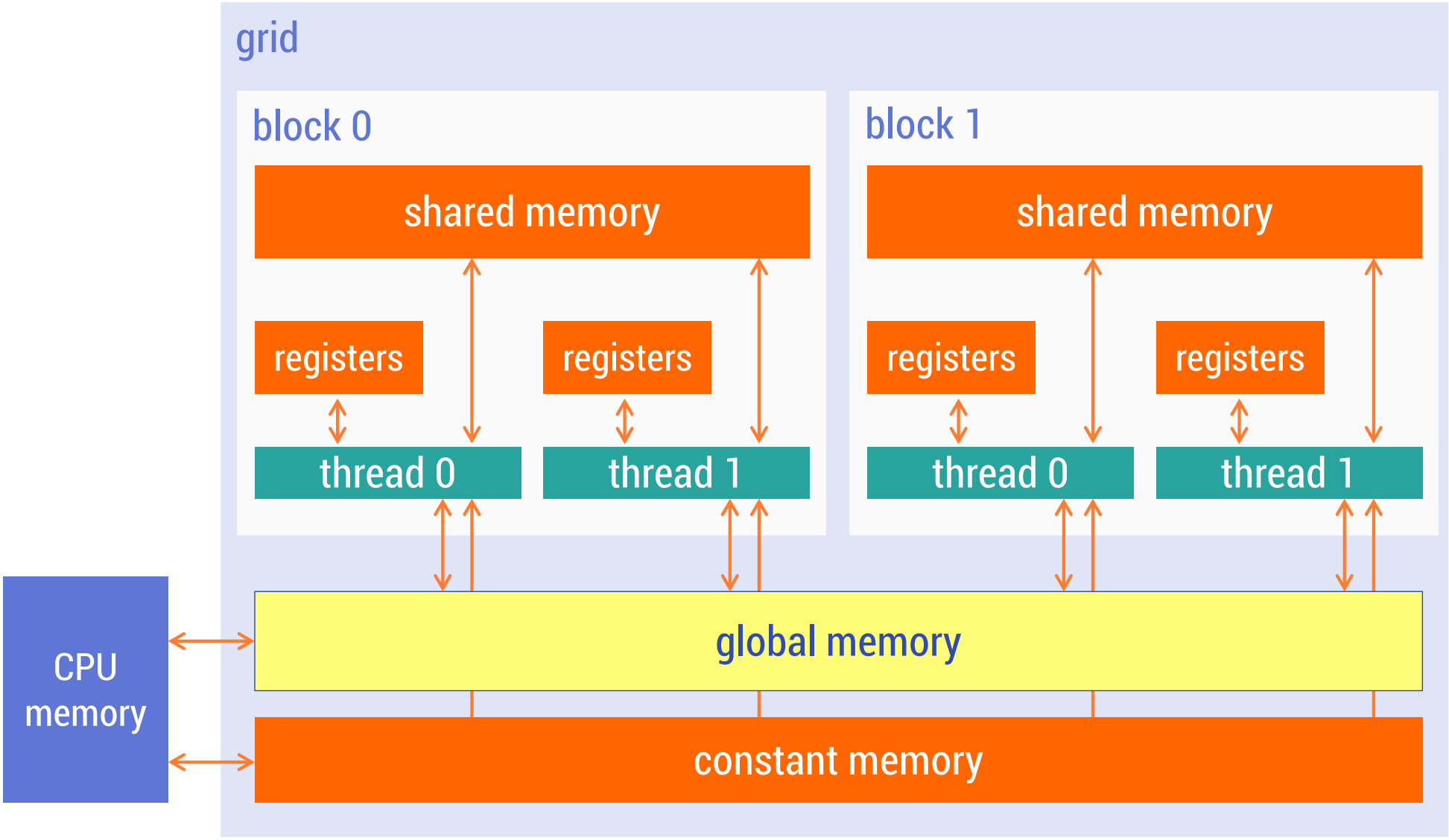


Registers

- used to store thread private scalar variables
- fast access, high bandwidth

```
__global__ void square(int *d_out, int *d_in){
    int idx = threadIdx.x;
    int f = d_in[idx];
    d_out[idx] = f * f;
}
```

GPU memories



Global memory

- can be read and written by the CPU
- can be read and written by all the threads in the GPU

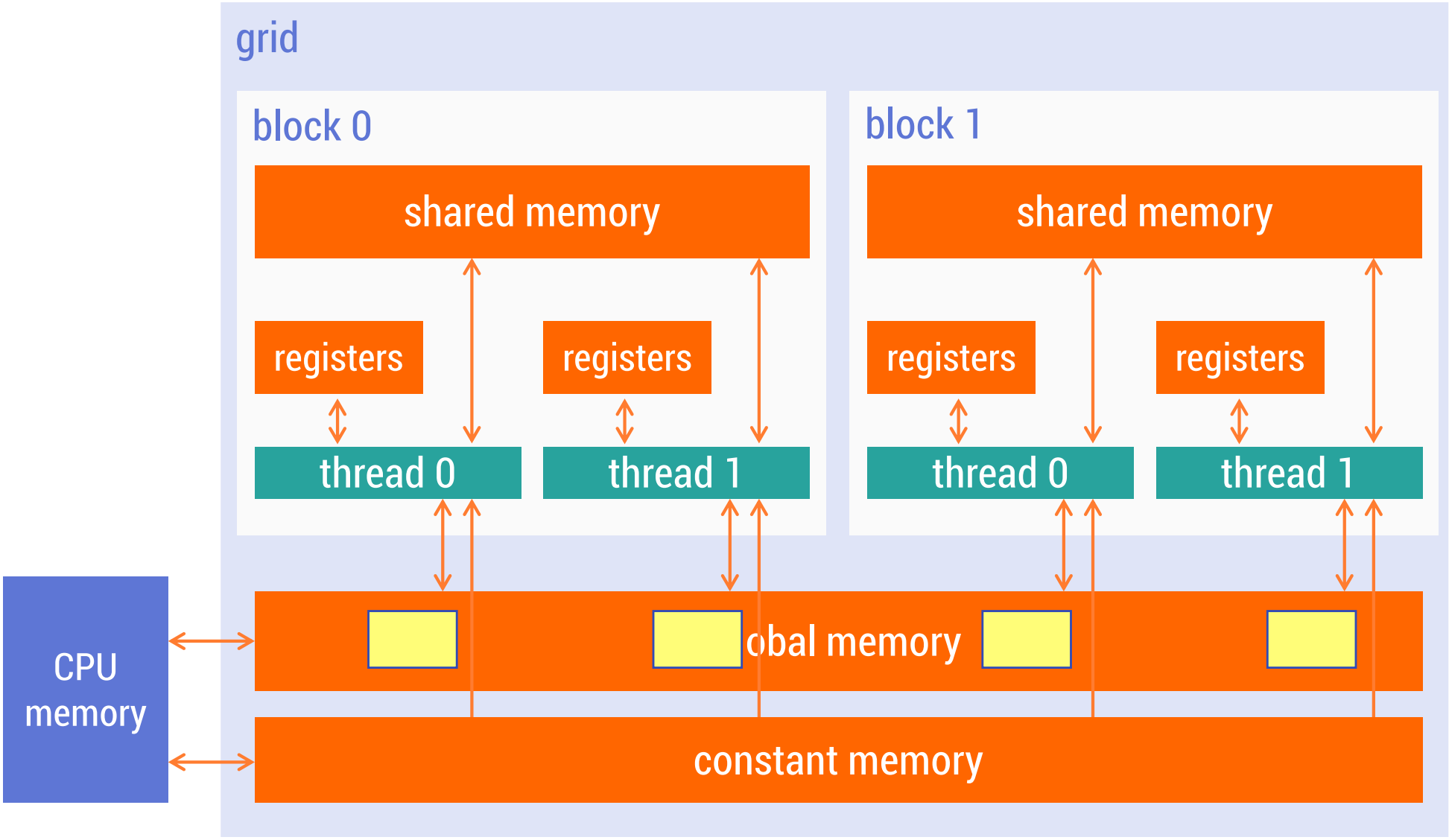
```
#define SIZE 1024
int main(){
    int bytes = SIZE*sizeof(int);
    int h_in[SIZE]={…}; int *d_in, *d_out;
    cudaMalloc((void **) &d_in, bytes);
    cudaMalloc((void **) &d_out, bytes);
    cudaMemcpy(d_in, h_in, bytes, cudaMemcpyHostToDevice);
    square<<<1, SIZE>>>(d_out, d_in);
    cudaMemcpy(h_out, d_out, bytes, cudaMemcpyDeviceToHost);
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

```
__global__ void square(int *d_out, int *d_in){
    int idx = threadIdx.x;
    int f = d_in[idx];
    d_out[idx] = f * f;
}
```

Global memory

- can be read and written by the CPU
- can be read and written by all the threads in the GPU
- analogy with the Von Neumann architecture: external DRAM
  - long latency, limited bandwidth
- optimisations are possible (e.g. coalesced accesses)

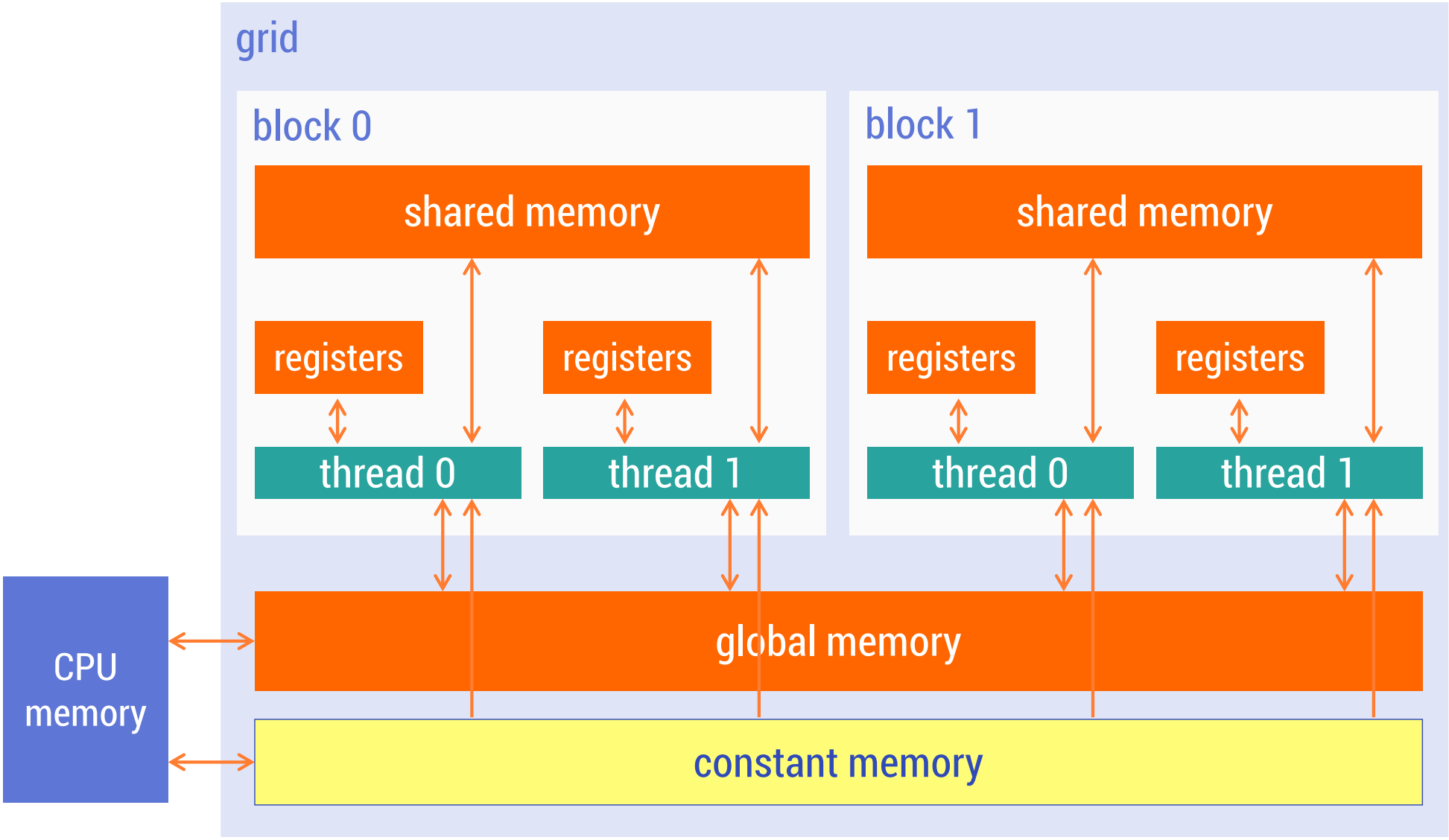
GPU memories



Local memory

- a specific area in the global memory
  - long latency
- private variable (arrays)
  - cannot be stored in registers

GPU memories



Constant memory

- can be read and written by the CPU
- can only be read by all the threads in the GPU
- a read-only cache reduces latency

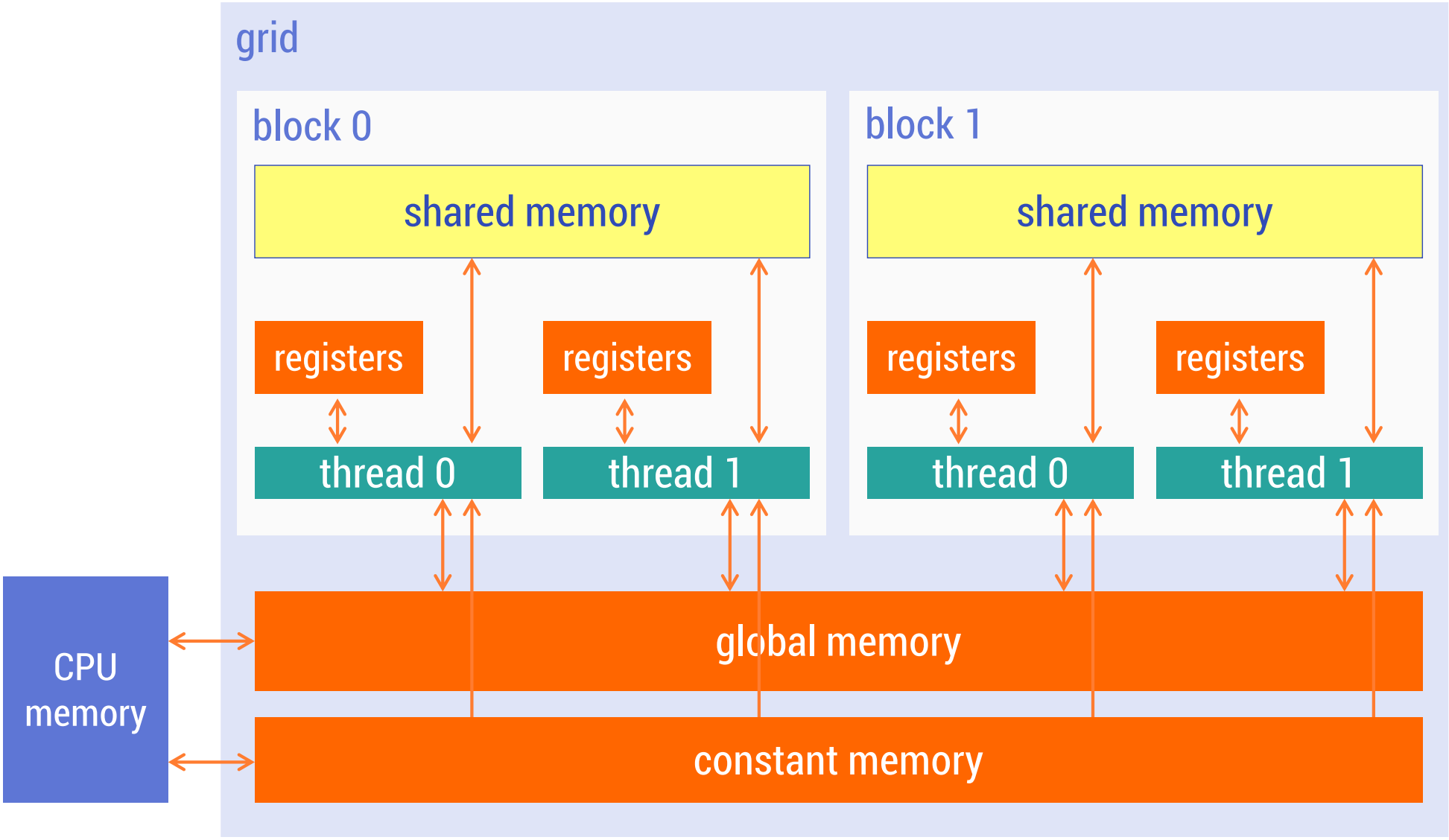
```
#define SIZE 100
float h_val[SIZE] = { … };

__constant__ float d_val[SIZE];
cudaMemcpyToSymbol(d_val, h_val,SIZE*sizeof(float));
kernel<<<…,…>>>(…)

__global__ kernel(…){
    …
    for (int i=0; i<SIZE; i++)
       … = d_val[i] …;
}
```

- all the threads in a warp access the same address simultaneously
  ▻ a single access
- access is faster than to the global memory (cache)

GPU memories



Shared memory

- on-chip scratchpad memory
  - short latency
  - limited size
- shared by all the threads in the same block
- allocation can be static or dynamic
  - static allocation:

```
#define SIZE 1024
__global__ void kernel(…) {
     __shared__ float shared_data[SIZE];
     …
}
```

Shared memory

- on-chip scratchpad memory
  - short latency
  - limited size
- shared by all the threads in the same block
- allocation can be static or dynamic
  - static allocation
  - dynamic allocation:

```
#define SIZE 1024
int main(){
        …
        kernel<<<…, …, SIZE*sizeof(float)>>>(…);
}
__global__ void kernel(…) {
        extern __shared__ float *shared_data;
        …
}
```

Shared memory

- on-chip scratchpad memory
  - short latency
  - limited size
- shared by all the threads in the same block
- allocation can be static or dynamic
- the kernel must copy data from the global memory to the shared memory

```
#define SIZE 1024
__global__ void kernel(float *data) {
      __shared__ float shared_data[SIZE];
      int idx = threadIdx.x;
      shared_data[idx] = data[idx];
      __syncthreads();
      …  // data can be accessed in the fast shared memory
}
```

Variable qualifiers

declaration	memory	scope	lifetime
`int var;`	register	thread	kernel
`int var[100];`	local	thread	kernel
`__device__ __shared__ int var;`	shared	block	kernel
`__device__ int var;`	global	grid	application
`__device__ __constant__ int var;`	constant	grid	application

Pointers can only point to declared/allocated areas in the global memory

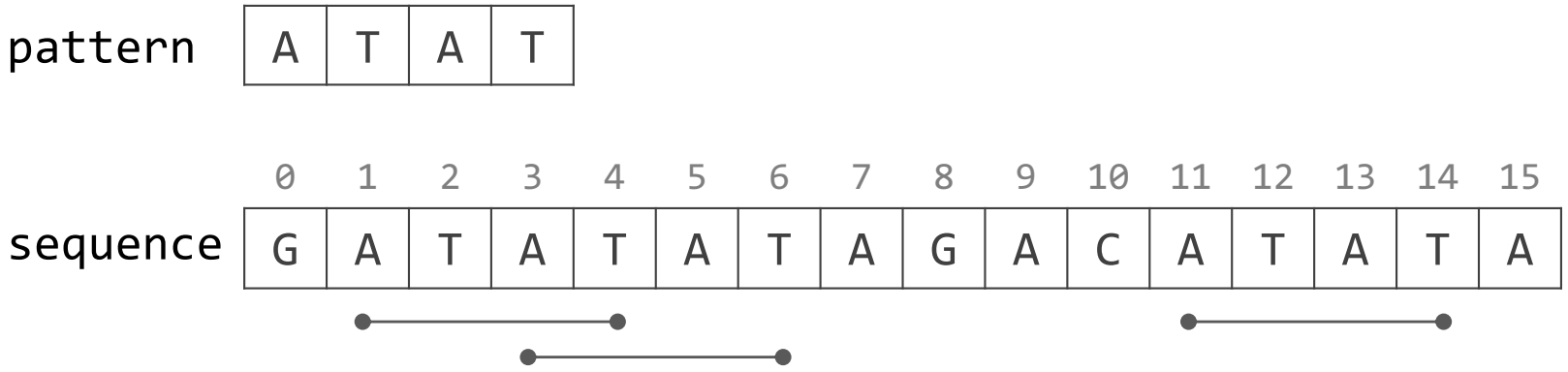
- area allocated by the host, pointer passed as a kernel parameter

```
__global__ void kernel(float *ptr){}
```

- pointer **computed** as the address of a global variable

```
float *ptr = &globalVar;
```

DNA pattern matching




```
bool matches(unsigned char *pattern, unsigned char *seq)
```

determines whether the 8-char pattern appears in the 16384-char sequence

NB: there are 16377 possible positions for the pattern

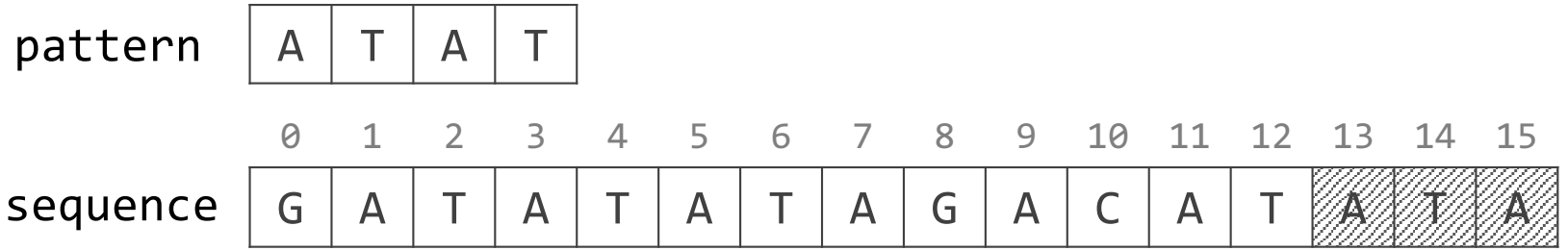

```
#define SLEN 16384
#define PLEN 8
#define BSIZE 512

bool matches(unsigned char *pattern, unsigned char *seq){
    unsigned int bnum = (SLEN - PLEN + 1 + BSIZE-1)/BSIZE) ;
    unsigned int pbytes = PLEN*sizeof(char);
    unsigned int sbytes = SLEN*sizeof(char);
    bool h_match = FALSE; bool *d_match;
    unsigned char *d_seq, *d_pattern;
    cudaMalloc((void **)&d_seq, sbytes);
    cudaMemcpy(d_seq, seq, sbytes, CudaMemcpyHostToDevice);
    cudaMalloc((void **)&d_pattern, pbytes);
    cudaMemcpy(d_pattern, pattern, pbytes, CudaMemcpyHostToDevice);
    cudaMalloc((void **)&d_match, sizeof(bool));
    cudaMemcpy(d_match, &h_match, sizeof(bool), CudaMemcpyHostToDevice);
    search<<<bnum, BSIZE>>>(d_seq, d_pattern, d_match);
    cudaMemcpy(h_match, d_match, sizeof(bool), CudaMemcpyDeviceToHost);
    cudaFree(d_seq); cudaFree(d_pattern); cudaFree(d_match);
    return(h_match);
}
```

1st version of the kernel

```
__global__
void search(  unsigned char *d_seq, unsigned char *d_pattern,
              bool *d_match){
    int gidx = blockIdx.x*blockDim.x+threadIdx.x;
    bool found = TRUE;

    if (idx < SLEN – PLEN + 1){
        for (int i=0; i<PLEN; i++)
            if (d_seq[gidx+i] != d_pattern[i])
                found = FALSE;
        if (found)
            *d_match = TRUE;
    }
}
```

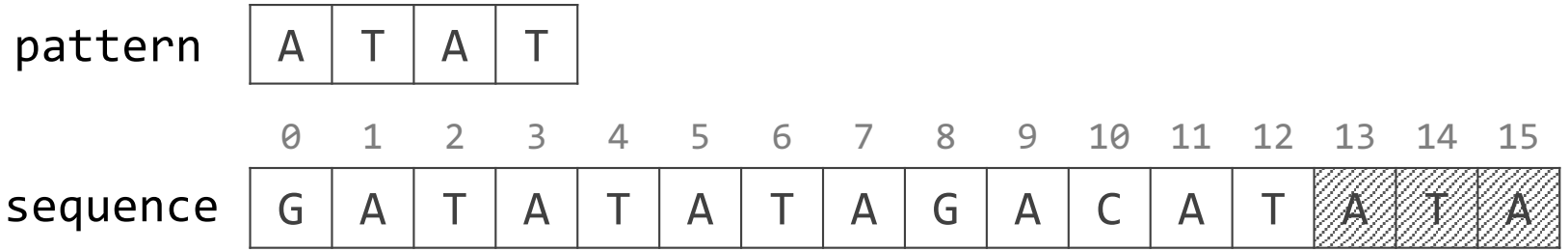


1st version of the kernel

```
__global__
void search(  unsigned char *d_seq, unsigned char *d_pattern,
              bool *d_match){
    int gidx = blockIdx.x*blockDim.x+threadIdx.x;
    bool found = TRUE;

    if (gidx < SLEN - PLEN + 1){
        for (int i=0; i<PLEN; i++)
            if (d_seq[gidx+i] != d_pattern[i])
                found = FALSE;
        if (found)
            *d_match = TRUE;
    }
}
```

accesses to the global memory



2nd version of the kernel: using constant memory

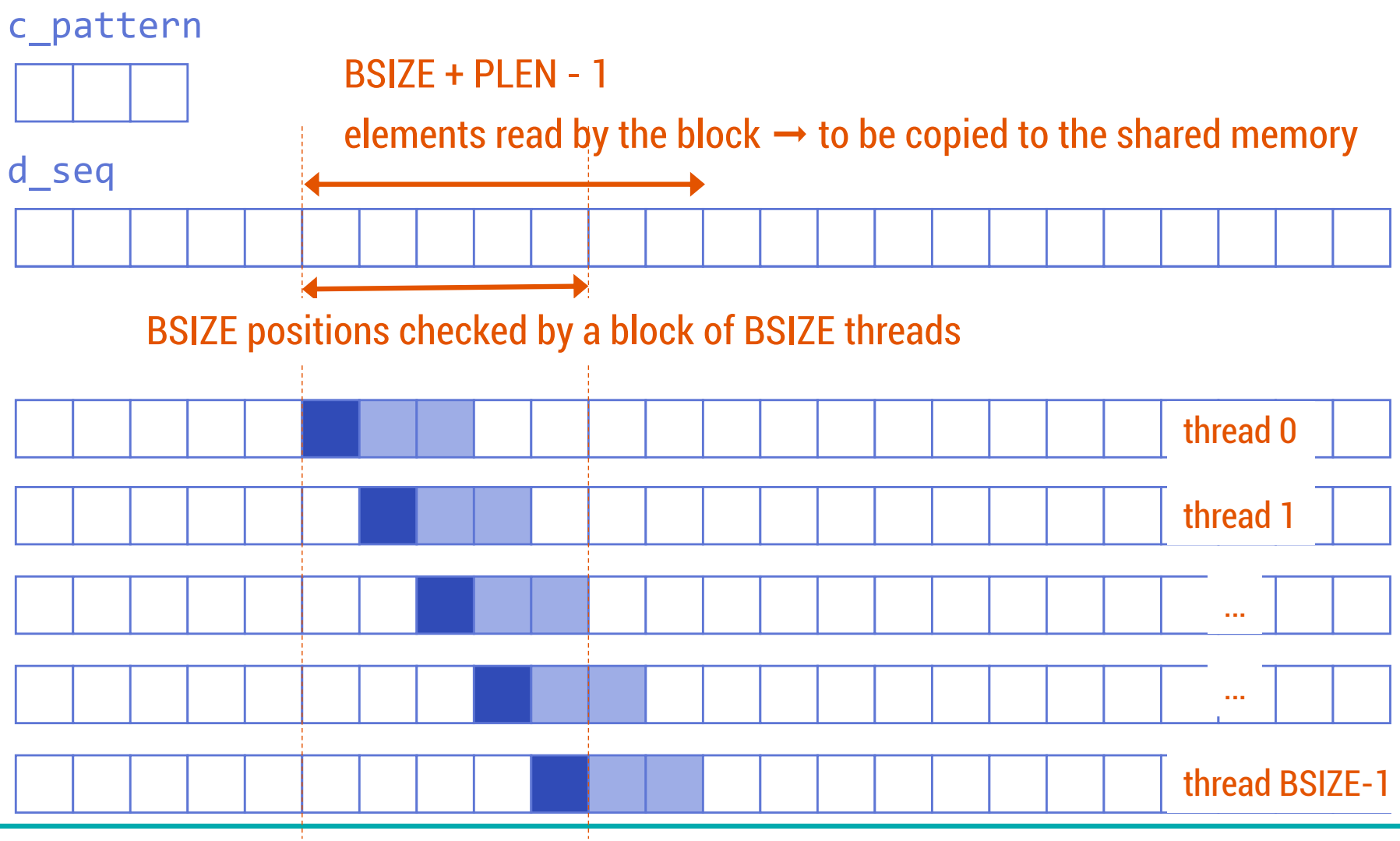
```
#define SLEN 16384
#define PLEN 8
#define BSIZE 512

bool matches(unsigned char *pattern, unsigned char *seq){
    unsigned int bnum = (SLEN – PLEN + 1 + BSIZE-1)/BSIZE) ;
    unsigned int pbytes = PLEN*sizeof(char);
    unsigned int sbytes = SLEN*sizeof(char);
    bool h_match = FALSE; bool *d_match;
    unsigned char *d_seq, *d_pattern;
    cudaMalloc((void **)&d_seq, sbytes);
    cudaMemcpy(d_seq, seq, sbytes, CudaMemcpyHostToDevice);
    __constant__ unsigned char c_pattern[PLEN];
    cudaMemcpyToSymbol(c_pattern, pattern, pbytes);
    cudaMalloc((void **)&d_match, sizeof(bool));
    cudaMemcpy(d_match, &h_match, sizeof(bool), CudaMemcpyHostToDevice)
    search<<<bnum, BSIZE>>>(d_seq, d_match);
    cudaMemcpy(h_match, d_match, sizeof(bool), CudaMemcpyDeviceToHost);
    cudaFree(d_seq); cudaFree(d_pattern); cudaFree(d_match);
    return(h_match);
}
```

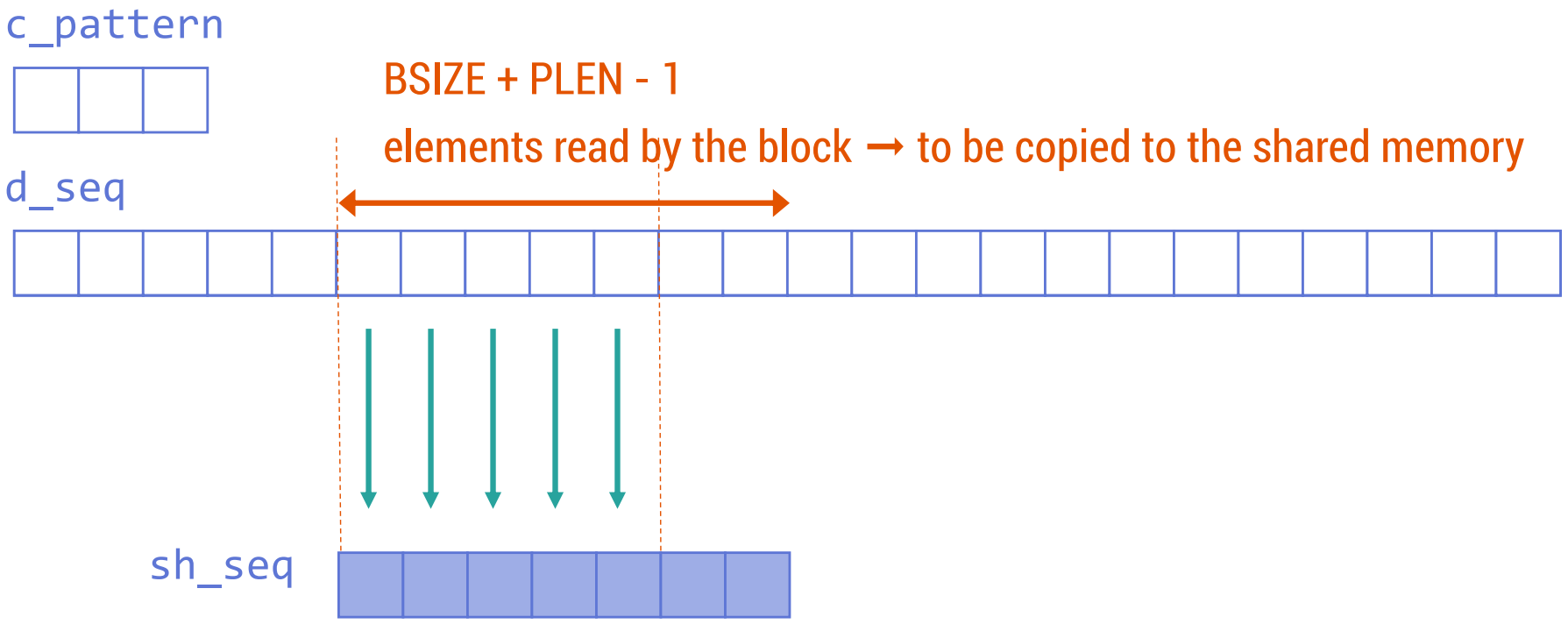

```
__global__
void search(  unsigned char *d_seq, unsigned char *d_pattern,
              bool *d_match){
    int gidx = blockIdx.x*blockDim.x+threadIdx.x;
    bool found = TRUE;

    if (gidx < SLEN - PLEN + 1){
        for (int i=0; i<PLEN; i++)
            if (d_seq[gidx+i] != c_pattern[i])
                found = FALSE;
        if (found)
            *d_match = TRUE;
    }
}
```

3rd version of the kernel: using shared memory

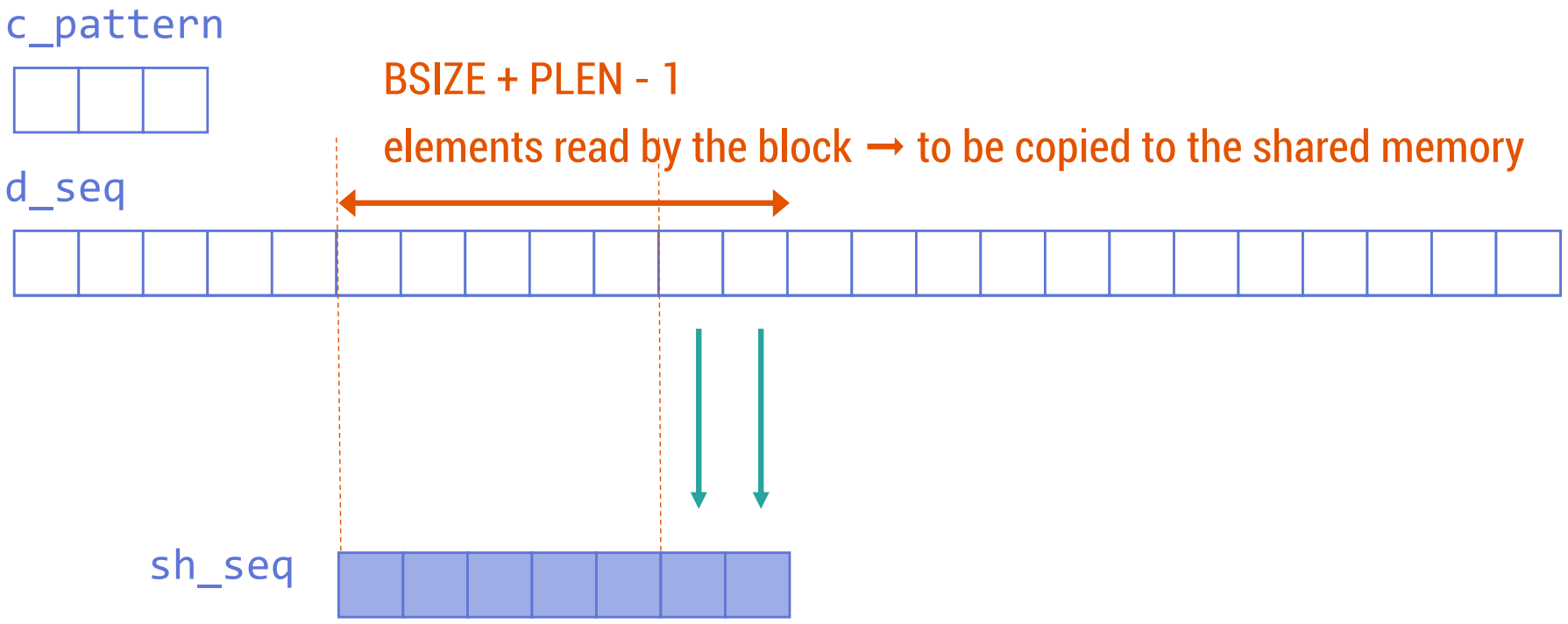


3rd version of the kernel: using shared memory




```
int gidx = blockIdx.x*blockDim.x + threadIdx.x;
int lidx = threadIdx.x;
sh_seq[lidx] = seq[gidx];
```

3rd version of the kernel: using shared memory




```
if ((lidx < PLEN-1) && (gidx + BSIZE < SLEN))
    sh_seq[lidx+BSIZE] = seq[gidx+BSIZE];
__syncthreads();
```

3rd version of the kernel: using shared memory

```
__global__
void search(unsigned char *d_seq, bool *d_match){
    int gidx = blockIdx.x*blockDim.x+threadIdx.x;
    bool found = TRUE;
    __shared__ unsigned char sh_seq[BSIZE+PLEN-1];
    int lidx = threadIdx.x;
    sh_seq[lidx] = d_seq[gidx];
    if ( (lidx < PLEN- 1) && (gidx + BSIZE < SLEN) )
        sh_seq[lidx+BSIZE] = seq[gidx+BSIZE];
    __synchthreads();
    if (gidx < SLEN - PLEN + 1){
        for (int i=0; i<PLEN; i++)
            if (sh_seq[lidx+i] != c_pattern[i])
                found = FALSE;
        if (found)
            *d_match = TRUE;
    }
}
```

Tiling:
a strategy to reduce trafic to the global memory

Example: matrix multiplication

access order →

$thread_{0,0}$	$A_{0,0} * B_{0,0}$	$A_{0,1} * B_{1,0}$	$A_{0,2} * B_{2,0}$	$A_{0,3} * B_{3,0}$
$thread_{0,1}$	$A_{0,0} * B_{0,1}$	$A_{0,1} * B_{1,1}$	$A_{0,2} * B_{2,1}$	$A_{0,3} * B_{3,1}$
$thread_{1,0}$	$A_{1,0} * B_{0,0}$	$A_{1,1} * B_{1,0}$	$A_{1,2} * B_{2,0}$	$A_{1,3} * B_{3,0}$
$thread_{1,1}$	$A_{1,0} * B_{0,1}$	$A_{1,1} * B_{1,1}$	$A_{1,2} * B_{2,1}$	$A_{1,3} * B_{3,1}$

B

$B_{0,0}$	$B_{0,1}$		
$B_{1,0}$	$B_{1,1}$		
$B_{2,0}$	$B_{2,1}$		
$B_{3,0}$	$B_{3,1}$		

A

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$

C

$C_{0,0}$	$C_{0,1}$		
$C_{1,0}$	$C_{1,1}$		

assumption: the shared memory is too small to store A and B

Tiling:
a strategy to reduce trafic to the global memory

	Phase 0			Phase 1		
$thread_{0,0}$	$A_{0,0}$ ↓ $AS_{0,0}$	$B_{0,0}$ ↓ $BS_{0,0}$				
$thread_{0,1}$	$A_{0,1}$ ↓ $AS_{0,1}$	$B_{0,1}$ ↓ $BS_{0,1}$				
$thread_{1,0}$	$A_{1,0}$ ↓ $AS_{1,0}$	$B_{1,0}$ ↓ $BS_{1,0}$				
$thread_{1,1}$	$A_{1,1}$ ↓ $AS_{1,1}$	$B_{1,1}$ ↓ $BS_{1,1}$				

BS

$B_{0,0}$	$B_{0,1}$
$B_{1,0}$	$B_{1,1}$

B

$B_{0,0}$	$B_{0,1}$		
$B_{1,0}$	$B_{1,1}$		
$B_{2,0}$	$B_{2,1}$		
$B_{3,0}$	$B_{3,1}$		

AS

$A_{0,0}$	$A_{0,1}$
$A_{1,0}$	$A_{1,1}$

A

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$

C

$C_{0,0}$	$C_{0,1}$		
$C_{1,0}$	$C_{1,1}$		

Tiling:
a strategy to reduce trafic to the global memory

	Phase 0			Phase 1		
$thread_{0,0}$	$A_{0,0}$ ↓ $AS_{0,0}$	$B_{0,0}$ ↓ $BS_{0,0}$	$C_{0,0}$ += $AS_{0,0} * BS_{0,0}$ + $AS_{0,1} * BS_{1,0}$			
$thread_{0,1}$	$A_{0,1}$ ↓ $AS_{0,1}$	$B_{0,1}$ ↓ $BS_{0,1}$	$C_{0,1}$ += $AS_{0,0} * BS_{0,1}$ + $AS_{0,1} * BS_{1,1}$			
$thread_{1,0}$	$A_{1,0}$ ↓ $AS_{1,0}$	$B_{1,0}$ ↓ $BS_{1,0}$	$C_{1,0}$ += $AS_{1,0} * BS_{0,0}$ + $AS_{1,1} * BS_{1,0}$			
$thread_{1,1}$	$A_{1,1}$ ↓ $AS_{1,1}$	$B_{1,1}$ ↓ $BS_{1,1}$	$C_{1,1}$ += $AS_{1,0} * BS_{0,1}$ + $AS_{1,1} * BS_{1,1}$			

BS

$B_{0,0}$	$B_{0,1}$
$B_{1,0}$	$B_{1,1}$

B

$B_{0,0}$	$B_{0,1}$		
$B_{1,0}$	$B_{1,1}$		
$B_{2,0}$	$B_{2,1}$		
$B_{3,0}$	$B_{3,1}$		

AS

$A_{0,0}$	$A_{0,1}$
$A_{1,0}$	$A_{1,1}$

A

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$

C

$C_{0,0}$	$C_{0,1}$		
$C_{1,0}$	$C_{1,1}$		

Tiling:
a strategy to reduce trafic to the global memory

	Phase 0			Phase 1		
$thread_{0,0}$	$A_{0,0}$ ↓ $AS_{0,0}$	$B_{0,0}$ ↓ $BS_{0,0}$	$C_{0,0}$ += $AS_{0,0} * BS_{0,0}$ + $AS_{0,1} * BS_{1,0}$	$A_{0,2}$ ↓ $AS_{0,0}$	$B_{2,0}$ ↓ $BS_{0,0}$	
$thread_{0,1}$	$A_{0,1}$ ↓ $AS_{0,1}$	$B_{0,1}$ ↓ $BS_{0,1}$	$C_{0,1}$ += $AS_{0,0} * BS_{0,1}$ + $AS_{0,1} * BS_{1,1}$	$A_{0,3}$ ↓ $AS_{0,1}$	$B_{2,1}$ ↓ $BS_{0,1}$	
$thread_{1,0}$	$A_{1,0}$ ↓ $AS_{1,0}$	$B_{1,0}$ ↓ $BS_{1,0}$	$C_{1,0}$ += $AS_{1,0} * BS_{0,0}$ + $AS_{1,1} * BS_{1,0}$	$A_{1,2}$ ↓ $AS_{1,0}$	$B_{3,0}$ ↓ $BS_{1,0}$	
$thread_{1,1}$	$A_{1,1}$ ↓ $AS_{1,1}$	$B_{1,1}$ ↓ $BS_{1,1}$	$C_{1,1}$ += $AS_{1,0} * BS_{0,1}$ + $AS_{1,1} * BS_{1,1}$	$A_{1,3}$ ↓ $AS_{1,1}$	$B_{3,1}$ ↓ $BS_{1,1}$	

BS

$B_{2,0}$	$B_{2,1}$
$B_{3,0}$	$B_{3,1}$

B

$B_{0,0}$	$B_{0,1}$		
$B_{1,0}$	$B_{1,1}$		
$B_{2,0}$	$B_{2,1}$		
$B_{3,0}$	$B_{3,1}$		

AS

$A_{0,2}$	$A_{0,3}$
$A_{1,2}$	$A_{1,3}$

A

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$

C

$C_{0,0}$	$C_{0,1}$		
$C_{1,0}$	$C_{1,1}$		

Tiling:
a strategy to reduce trafic to the global memory

	Phase 0			Phase 1		
$thread_{0,0}$	$A_{0,0}$ ↓ $AS_{0,0}$	$B_{0,0}$ ↓ $BS_{0,0}$	$C_{0,0}$ += $AS_{0,0} * BS_{0,0}$ + $AS_{0,1} * BS_{1,0}$	$A_{0,2}$ ↓ $AS_{0,0}$	$B_{2,0}$ ↓ $BS_{0,0}$	$C_{0,0}$ += $AS_{0,0} * BS_{0,0}$ + $AS_{0,1} * BS_{1,0}$
$thread_{0,1}$	$A_{0,1}$ ↓ $AS_{0,1}$	$B_{0,1}$ ↓ $BS_{0,1}$	$C_{0,1}$ += $AS_{0,0} * BS_{0,1}$ + $AS_{0,1} * BS_{1,1}$	$A_{0,3}$ ↓ $AS_{0,1}$	$B_{2,1}$ ↓ $BS_{0,1}$	$C_{0,1}$ += $AS_{0,0} * BS_{0,1}$ + $AS_{0,1} * BS_{1,}$
$thread_{1,0}$	$A_{1,0}$ ↓ $AS_{1,0}$	$B_{1,0}$ ↓ $BS_{1,0}$	$C_{1,0}$ += $AS_{1,0} * BS_{0,0}$ + $AS_{1,1} * BS_{1,0}$	$A_{1,2}$ ↓ $AS_{1,0}$	$B_{3,0}$ ↓ $BS_{1,0}$	$C_{1,0}$ += $AS_{1,0} * BS_{0,0}$ + $AS_{1,1} * BS_{1,0}$
$thread_{1,1}$	$A_{1,1}$ ↓ $AS_{1,1}$	$B_{1,1}$ ↓ $BS_{1,1}$	$C_{1,1}$ += $AS_{1,0} * BS_{0,1}$ + $AS_{1,1} * BS_{1,1}$	$A_{1,3}$ ↓ $AS_{1,1}$	$B_{3,1}$ ↓ $BS_{1,1}$	$C_{1,1}$ += $AS_{1,0} * BS_{0,1}$ + $AS_{1,1} * BS_{1,1}$

BS

$B_{2,0}$	$B_{2,1}$
$B_{3,0}$	$B_{3,1}$

B

$B_{0,0}$	$B_{0,1}$		
$B_{1,0}$	$B_{1,1}$		
$B_{2,0}$	$B_{2,1}$		
$B_{3,0}$	$B_{3,1}$		

AS

$A_{0,2}$	$A_{0,3}$
$A_{1,2}$	$A_{1,3}$

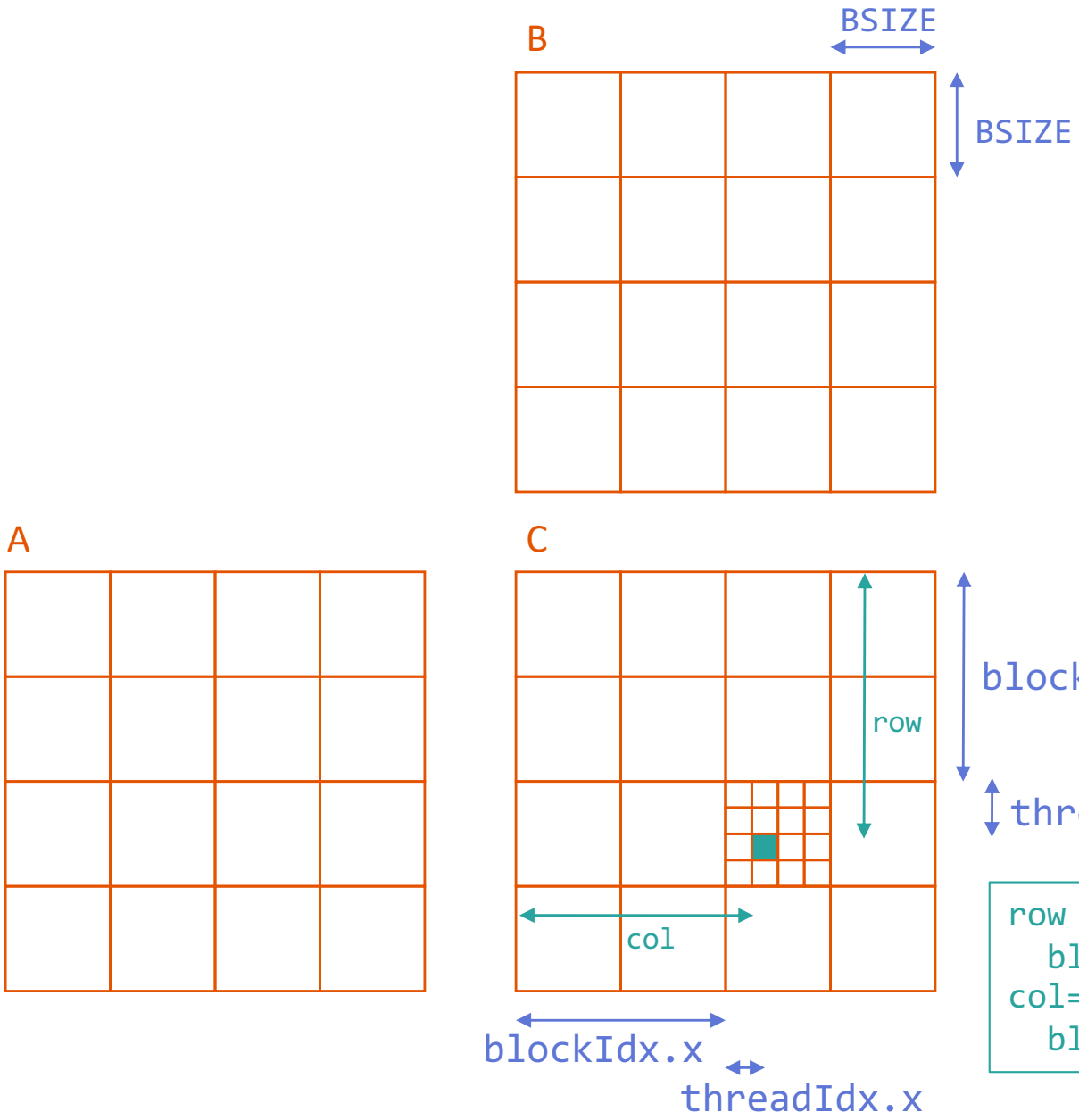
A

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$

C

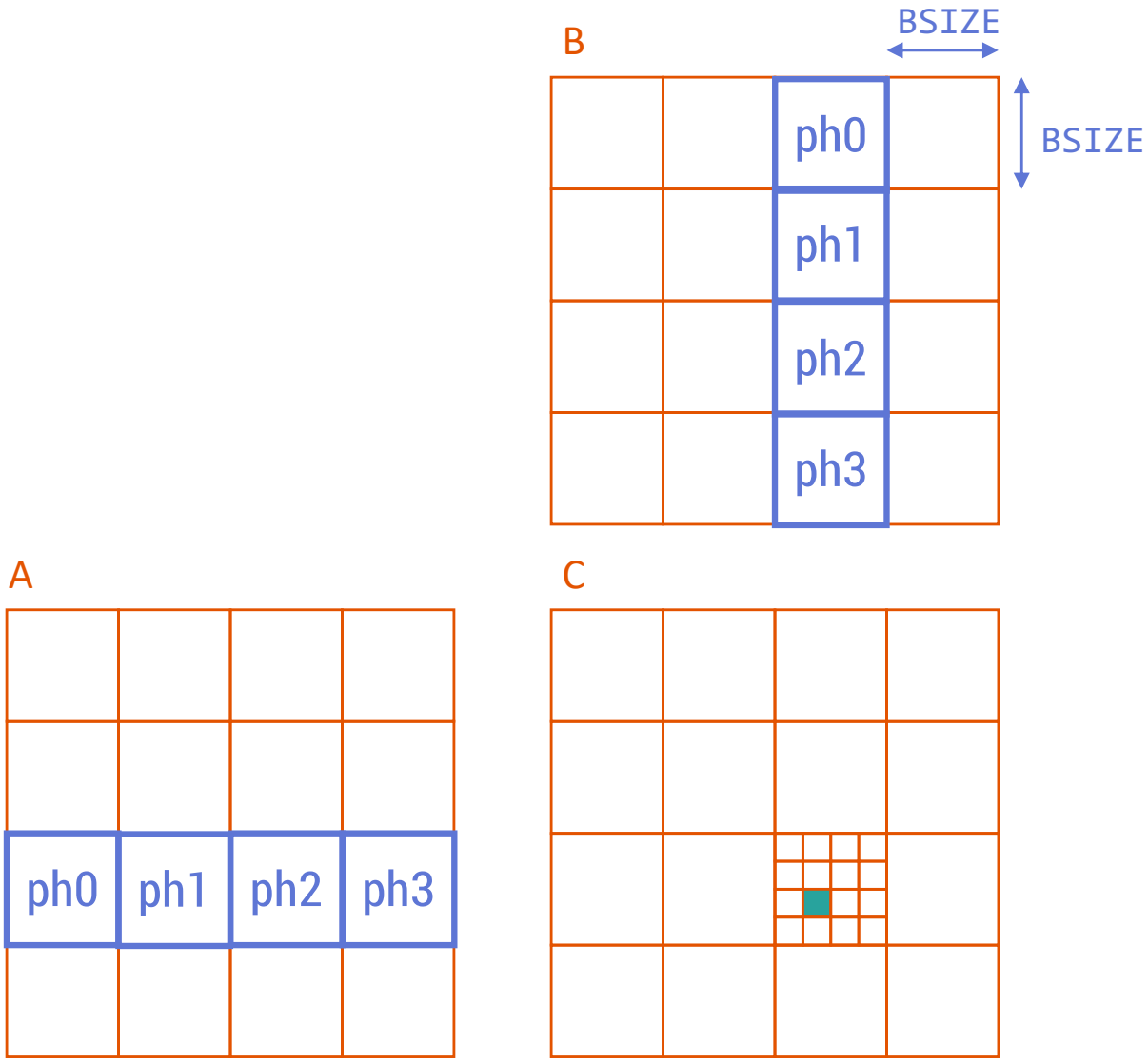
$C_{0,0}$	$C_{0,1}$		
$C_{1,0}$	$C_{1,1}$		

Tiled matrix multiplication

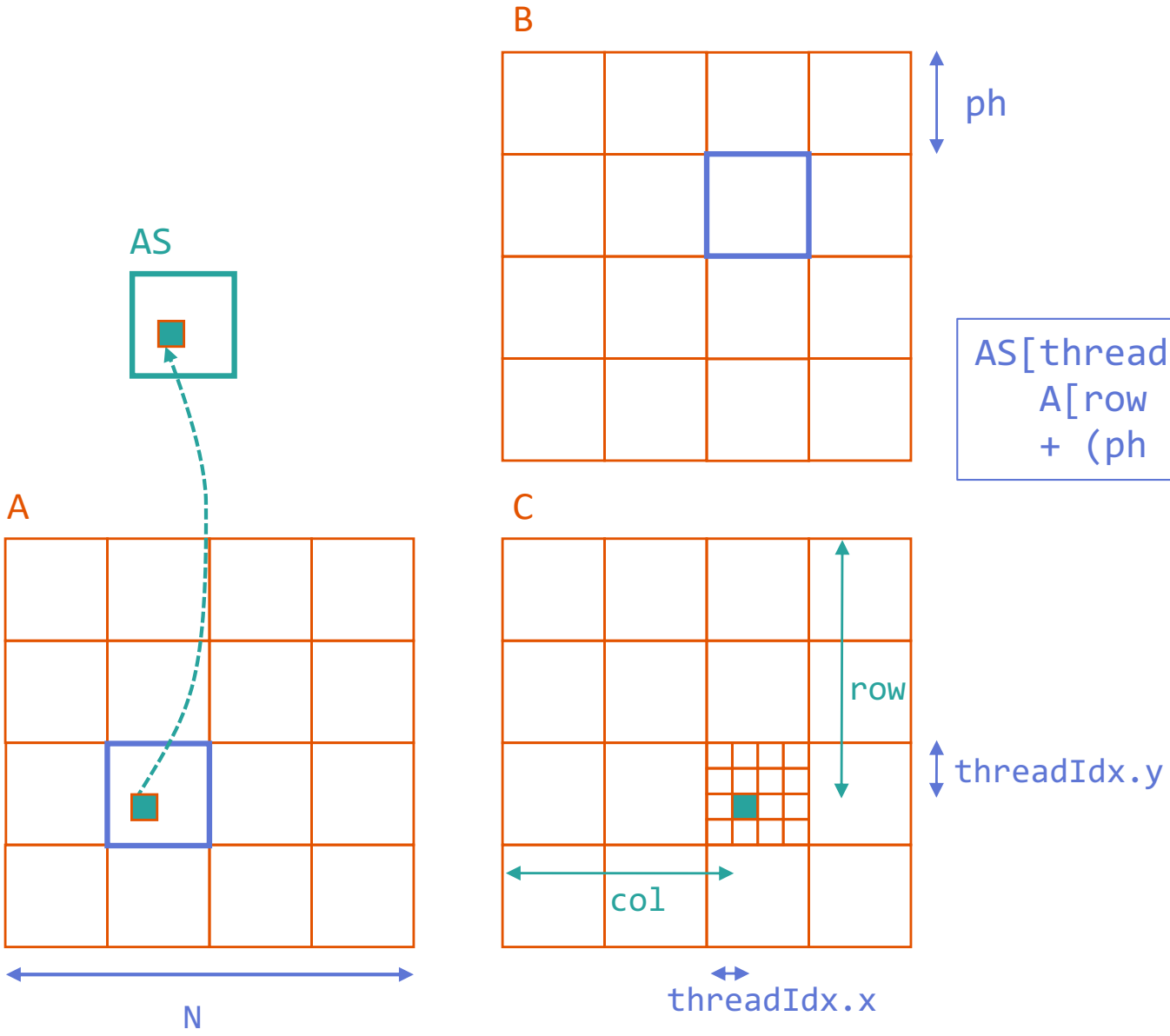



```
row =
  blockIdx.y * BSIZE + threadIdx.y;
col=
  blockIdx.x * BSIZE + threadIdx.x;
```

Tiled matrix multiplication

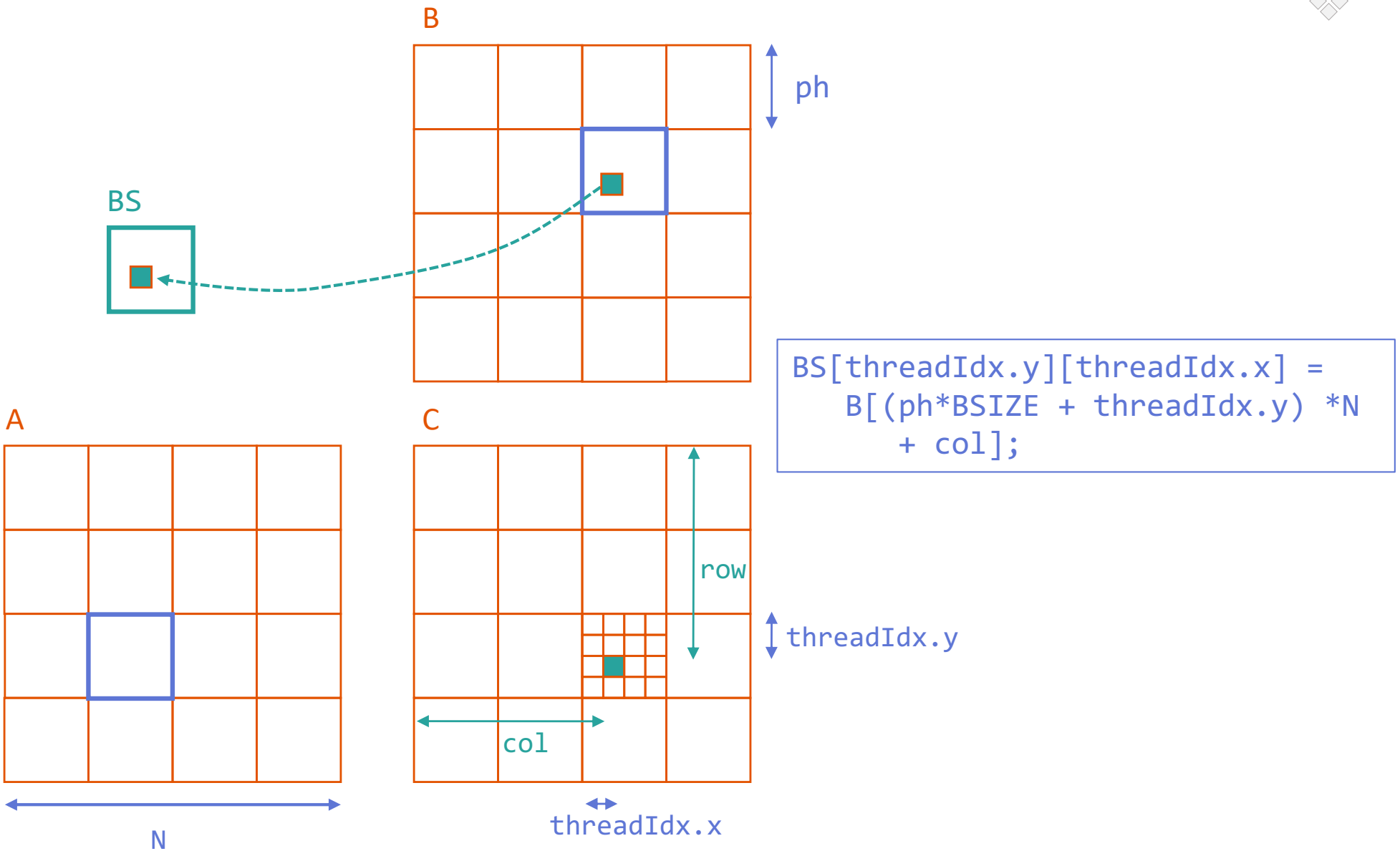


Tiled matrix multiplication




```
AS[threadIdx.y][threadIdx.x] =
   A[row * N
   + (ph * BSIZE + threadIdx.x)];
```

Tiled matrix multiplication




```
BS[threadIdx.y][threadIdx.x] =
   B[(ph*BSIZE + threadIdx.y) *N
      + col];
```

Tiled matrix multiplication

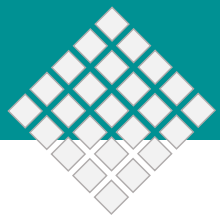
```
__global__ void MatmulKernel(float *A, float *B, float *C, int N){

    __shared__ float AS[BSIZE][BSIZE];
    __shared__ float BS[BSIZE][BSIZE];
    int row = blockIdx.y * BSIZE + threadIdx.y;
    int col= blockIdx.x * BSIZE + threadIdx.x;
    float res = 0;
```

assumption: TSIZE divides N

```
    for (int ph=0 ; ph<(N/BSIZE); ph++){
        AS[threadIdx.y][threadIdx.x] = A[row * N + ph * BSIZE + threadIdx.x];
        BS[threadIdx.y][threadIdx.x] =  B[[(ph* BSIZE + threadIdx.y) * N + col];
        _syncthreads();
        for (int k=0; k<BSIZE; k++){
            res += AS[threadIdx.y][k] * BS[k][threadIdx.x];
        }
        __syncthreads();
    }
    C[row*N + col] = res;
}
```

GPU-specific algorithms



Scan operation

Definition

- an input array
- a binary associative operator
- an identity element [`I op a = a`]

Inclusive ou exclusive *scan*

```
int acc = id_element;

for (i=0 ; i<n; i++) {
    acc = acc op in[i];
    out[i] = acc;
}
```

inclusive

```
int acc = id_element;

for (i=0 ; i<n; i++) {
    out[i] = acc;
    acc = acc op in[i];
}
```

exclusive

Scan operation

Example

- input: [1 2 3 4 5 6 7 8]
- operator: +
- identity element: 0

- output of **inclusive scan**:
  [1 3 6 10 15 21 28 36]

- output of **exclusive scan**:
  [0 1 3 6 10 15 21 28]

Remark: each output element is a reduction of previous input elements

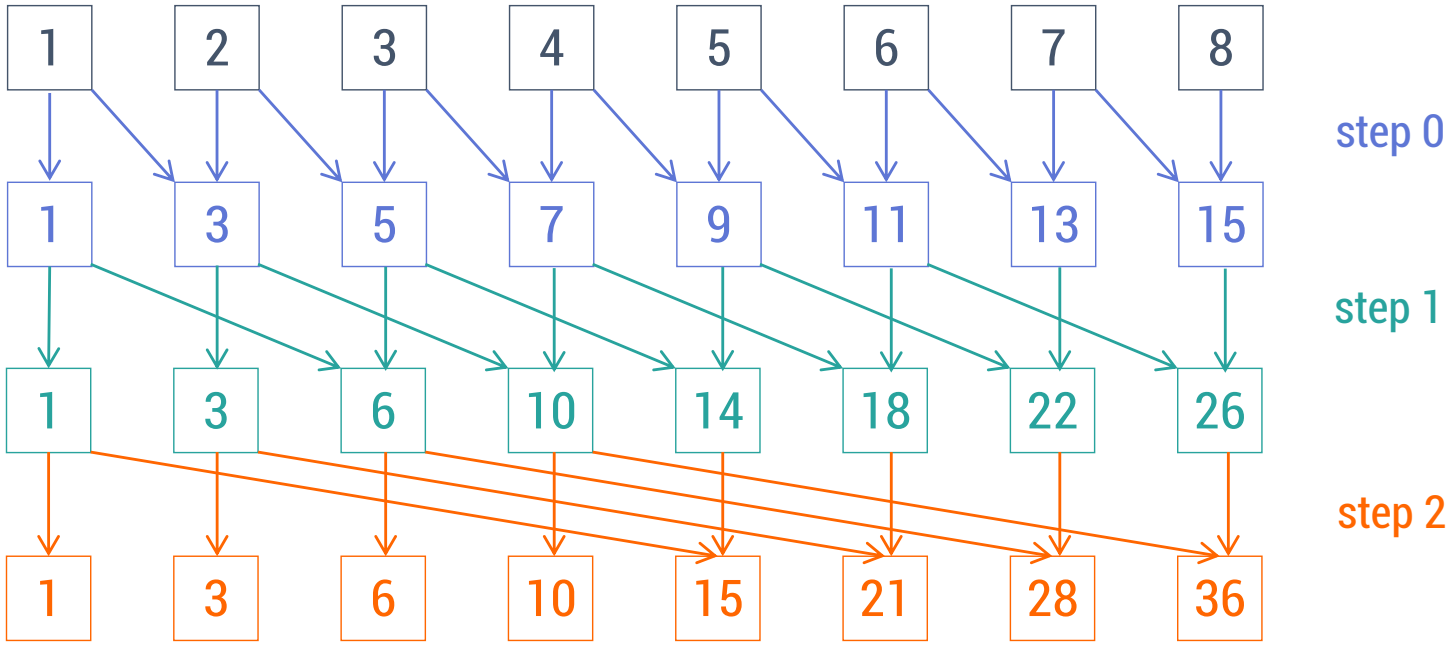
```
int acc = id_element;

for (i=0 ; i<n; i++) {
     acc = acc op in[i];
     out[i] = acc;
}
```

```
int acc = id_element;

for (i=0 ; i<n; i++) {
     out[i] = acc;
     acc = acc op in[i];
}
```

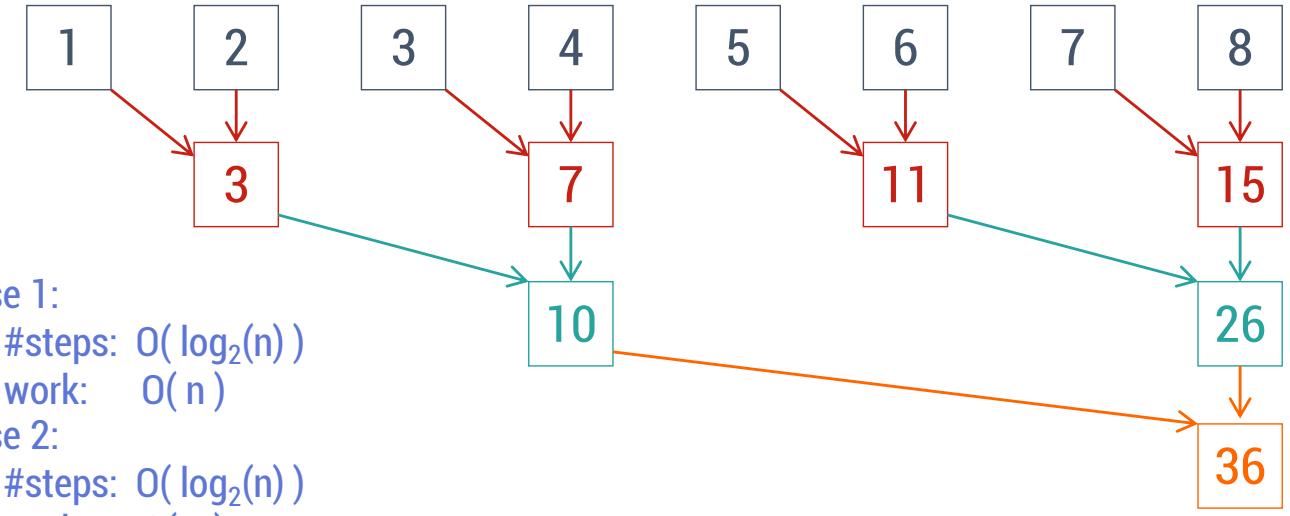
Hillis/Steele algorithm

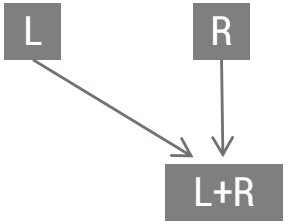


Step i :
each thread with id $\geq 2^i$ adds to its local value the value of its 2^i-positionleft neighbour

Complexity ? number of steps: $O(\log_2(n))$
work: $O(n.\log_2(n))$

Blelloch algorithm





Phase 1:

#steps: $O(\log_2(n))$

work: $O(n)$

Phase 2:

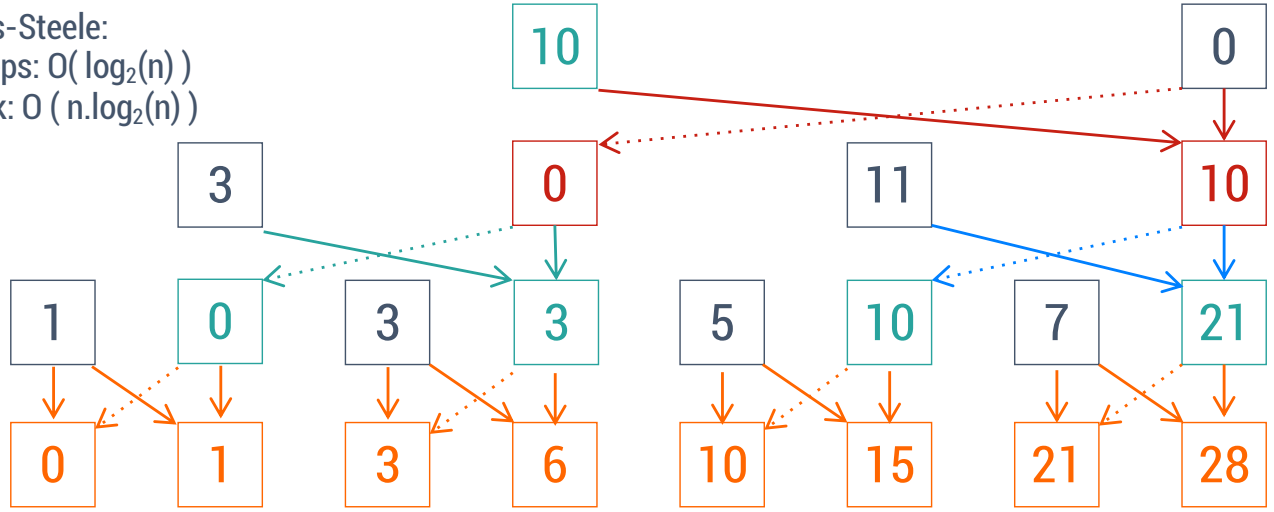
#steps: $O(\log_2(n))$

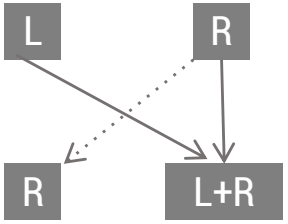
work: $O(n)$

Higgins-Steele:

#steps: $O(\log_2(n))$

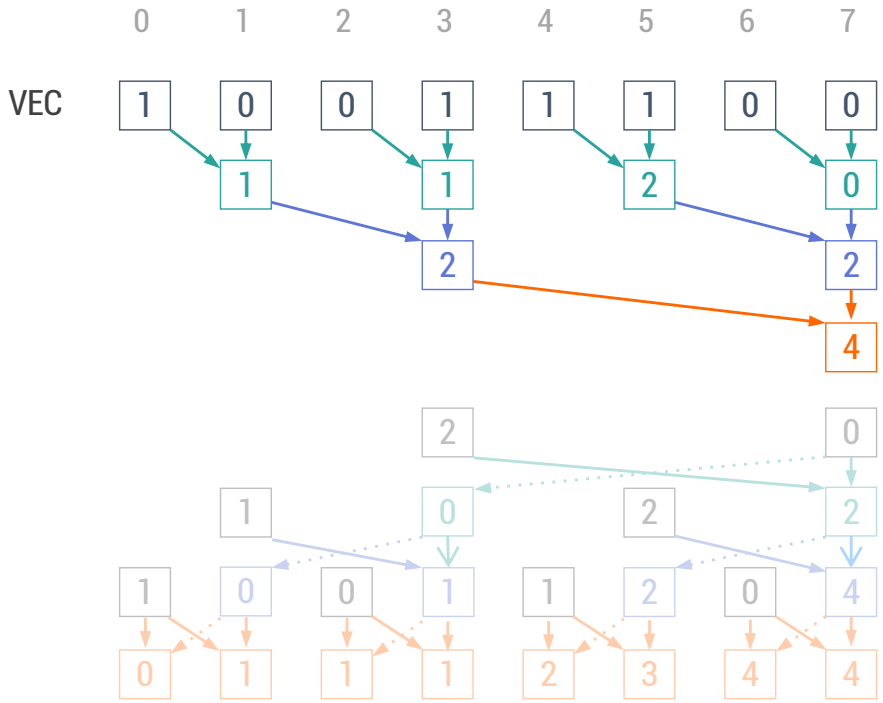
work: $O(n.\log_2(n))$





Blelloch algorithm

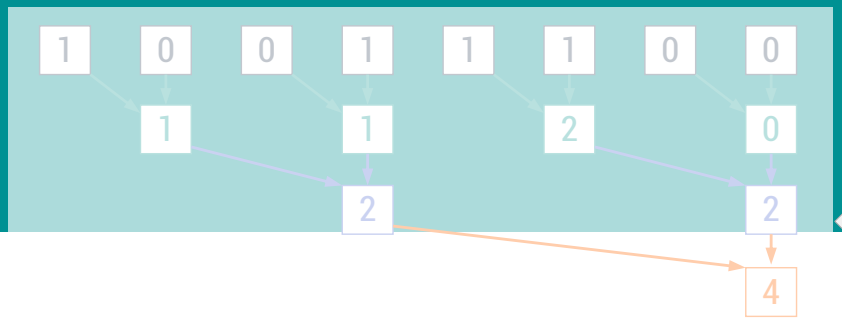
Phase 1

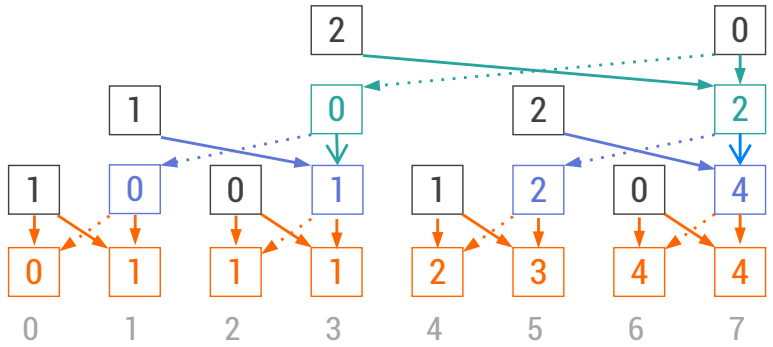


	offset = 1	offset = 2	offset = 4
th0	`vec[1]+=vec[0]`	`vec[3]+=vec[1]`	`vec[7] += vec[3]`
th1	`vec[3]+=vec[2]`	`vec[7]+=vec[5]`	
th2	`vec[5]+=vec[4]`		
th3	`vec[7]+=vec[6]`		

```
vec[ offset * (2*tid+2) -1 ] +=
        vec[ offset * (2*tid+1) - 1 ];
```

Blelloch algorithm





Phase 2

	offset = 4	offset = 2	offset = 1

th0	`tmp = vec[7]`
	`vec[7] += vec[3]`
	`vec[3] = tmp`
		`tmp = vec[3]`
		`vec[3] += vec[1]`
		`vec[1] = tmp`
			`tmp = vec[1]`
			`vec[1] += vec[0]`
			`vec[0] = tmp`

th1		`tmp = vec[7]`
		`vec[7] += vec[5]`
		`vec[5] = tmp`
			`tmp = vec[3]`
			`vec[3] += vec[2]`
			`vec[2] = tmp`

th2	`tmp = vec[ offset * (2*tid +2) - 1 ];`
	`vec[ offset * (2*tid +2) - 1] +=`
	`  vec[ offset * (2*tid +1) - 1 ];`
	`vec[ offset * (2*tid +1) - 1 ] = tmp;`
			`tmp = vec[5]`
			`vec[5] += vec[4]`
			`vec[4] = tmp`

th3			`tmp = vec[7]`
			`vec[7] += vec[6]`
			`vec[6] = tmp`

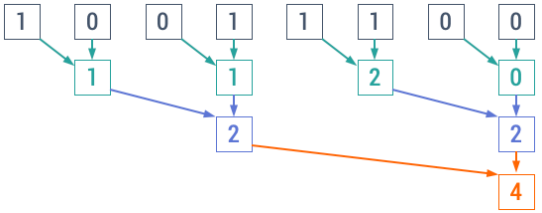
Scan function

This **function**, that can be called from a kernel, computes the contribution of **one** thread to the computation of the ***exclusive or inclusive scan*** (+) on an array the size of which is a power of 2 (and ≤ 1024).

Scan function

```
__device__ void scanBlock(int *d_out, int *d_in, int n, int exc){
    // scan over one block (n <= BLOCK_SIZE) [Blelloch's algorithm]
    int tid = threadIdx.x ;
    int offset, a, b, tmp;
    offset = 1 ;
    d_out[tid] = d_in[tid];
    for (int d=n/2 ; d>0 ; d/=2) {
        __syncthreads() ;
        if (tid < d) {
            a = offset * (2*tid+1) - 1 ;
            b = offset * (2*tid+2) - 1 ;
            if (b<n) d_out[b] += d_out[a] ;
        }
        offset *= 2 ;
    }
    __syncthreads();
    // to be continued on next slide
    …
```

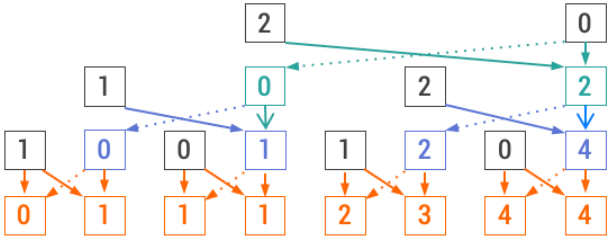
Phase 1



Scan function

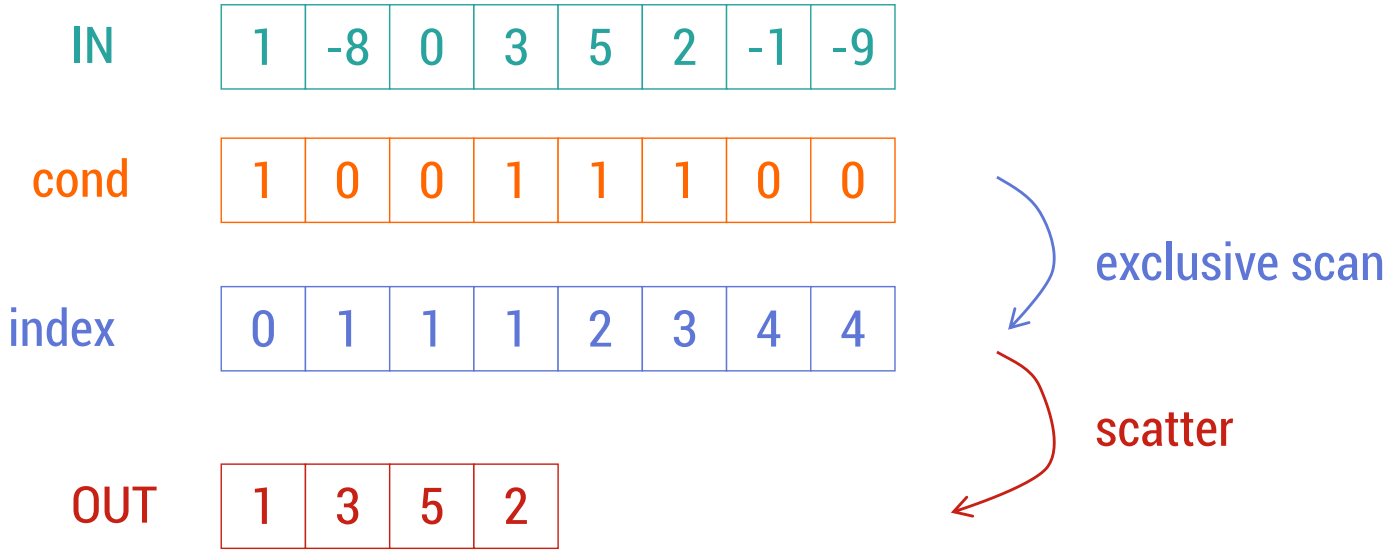
```
    //continued from previous slide
    …
    if (tid == 0)
        d_out[n-1] = 0 ;
    for (int d=1 ; d<n ; d *= 2){
        offset /= 2 ;
        __syncthreads() ;
        if (tid < d)
            a = offset * (2*tid+1) - 1 ;
            b = offset * (2*tid+2) - 1 ;
            tmp = d_out[b] ;
            d_out[b] += d_out[a] ;
            d_out[a] = tmp ;
        }
    }
    if (!exc){ // inclusive scan
        __syncthreads();
        if (tid<n) d_out[tid] += d_in[tid];
    }
}
```

Phase 2



Flow compaction

From input array IN of n elements, generate output array OUT that contains all the elements of IN that are strictly greater than 0, in the same order (i.e. filter out elements that are lower than or equal to 0).




```
kcompact<<<1, N, N*sizeof(int)>>>(d_out, d_in, N);
```

```
__global__ void kcompact(int *out, int *in, int n){
    extern __shared__ int vec[] ;
    int tid = threadIdx.x ;

    int my_data = in[tid];

    int my_cond;
    if (tid < n){
        if (my_data> 0)
           my_cond = 1 ;
        else
           my_cond = 0 ;
        vec[tid] = my_cond ;
    }

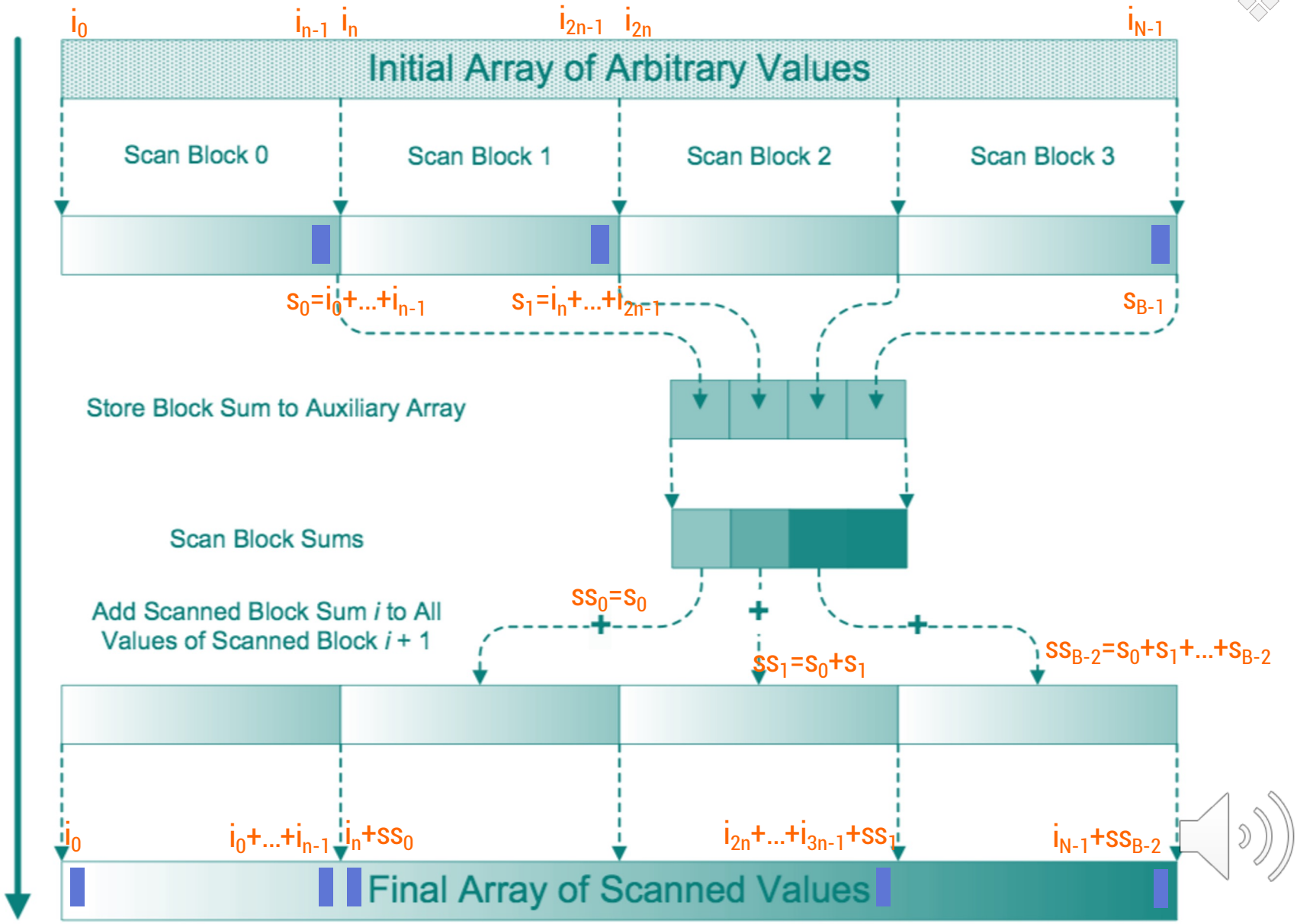
     scanBlock(vec, vec, n, 1 /*exclusive*/);

    if (my_cond == 1)
         out[vec[tid]] = my_data ;
}
```

IN	1	-8	0	3	5	2	-1	-9
cond	1	0	0	1	1	1	0	0
index	0	1	1	1	2	3	4	4

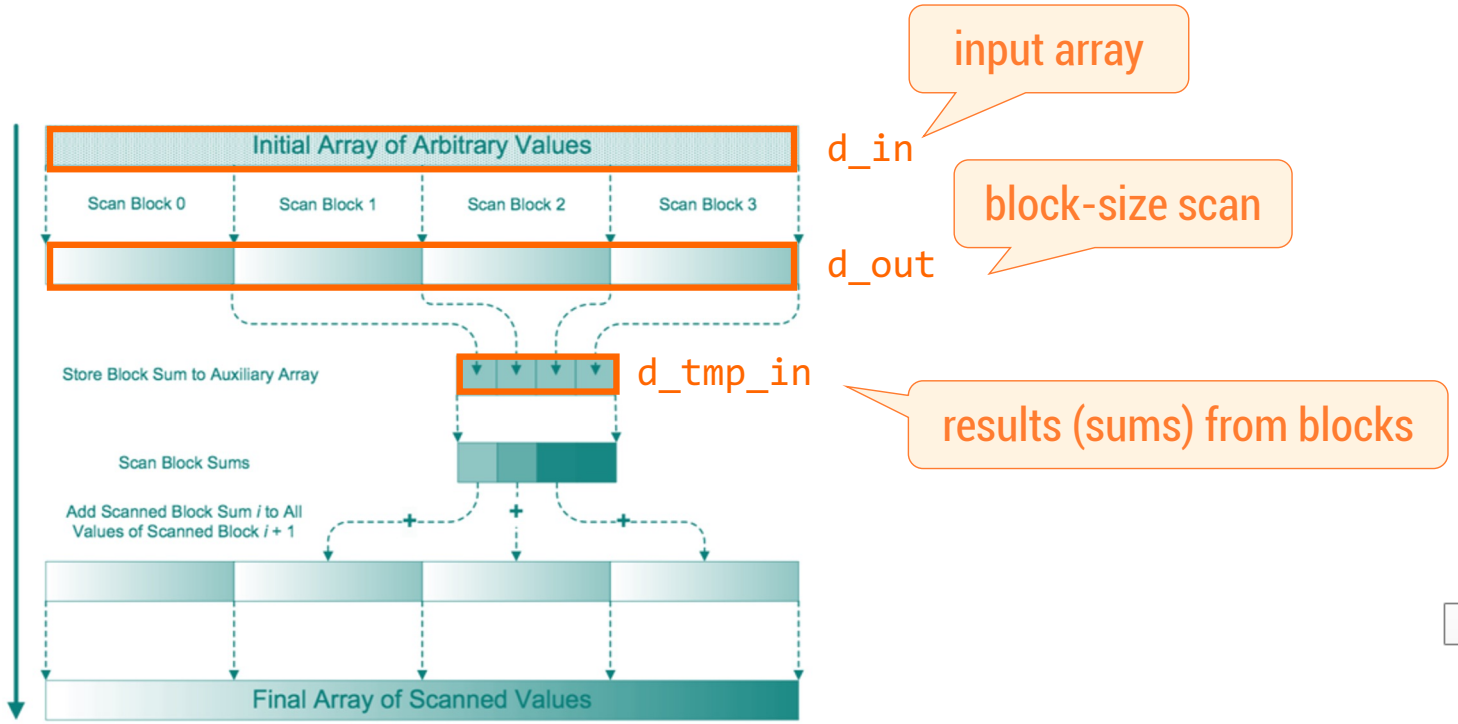
OUT	1	3	5	2

Scan of arrays larger than 1024



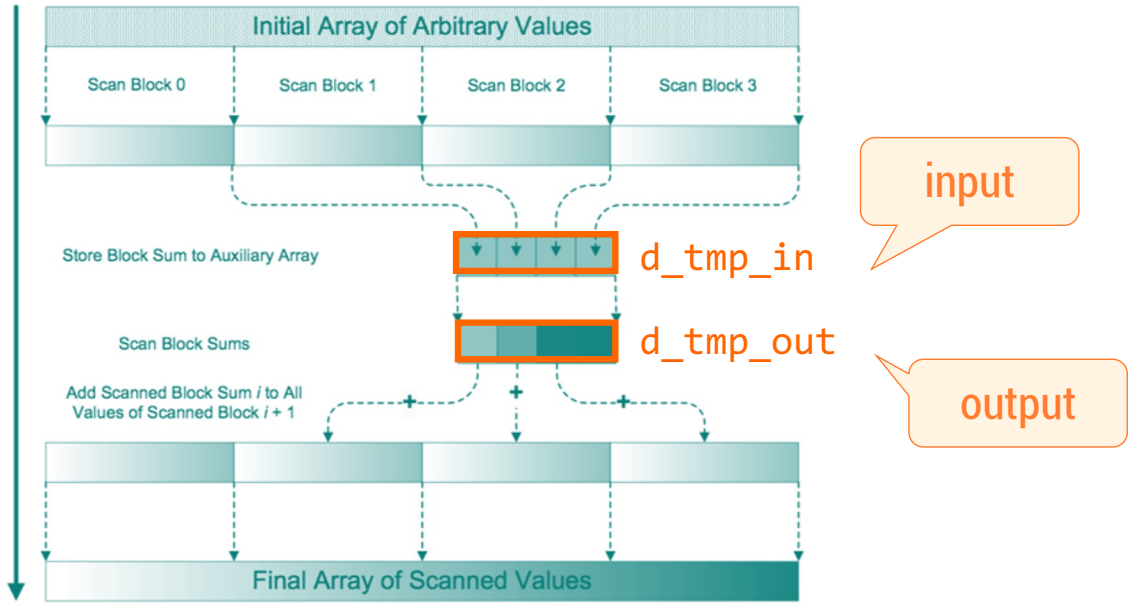
Scan of arrays larger than 1024

```
__device__ void scanBlock(int *out, int *in, int n, int exc){
        … // scan over a singe block (n <= BLOCK_SIZE)
}
__global__ void scanKernel_1(int *out, int *in, int *tmp, int size){
}
```



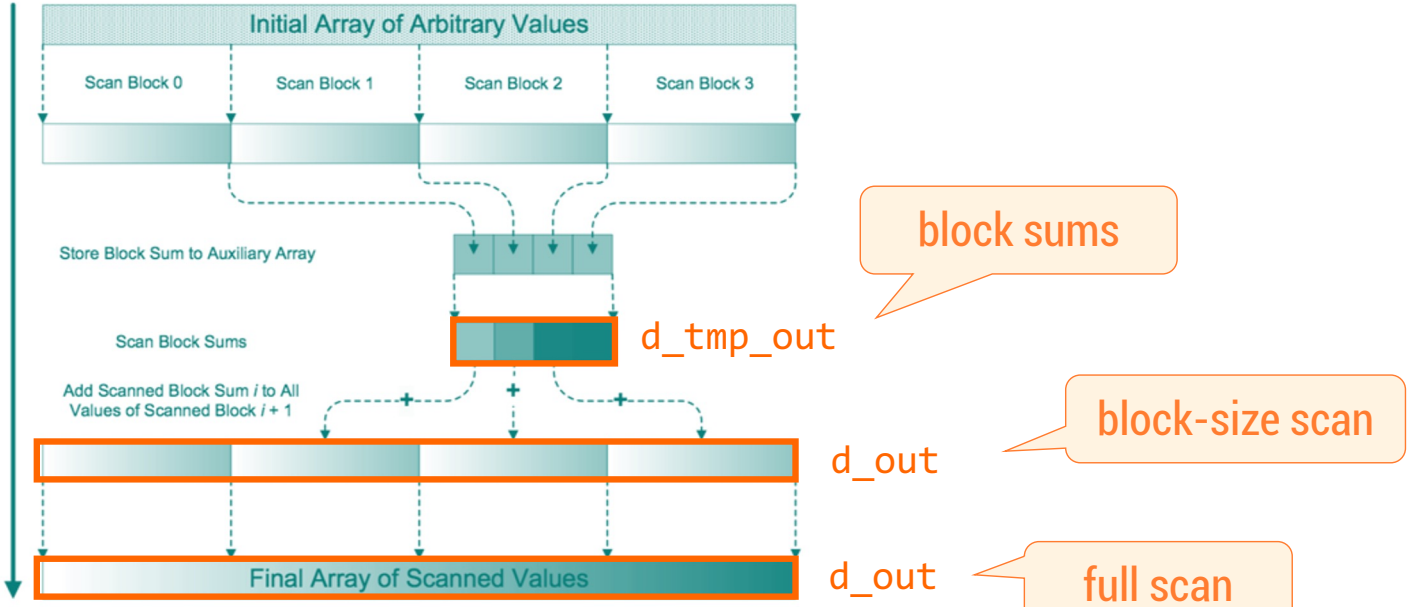
Scan of arrays larger than 1024

```
__device__ void scanBlock(int *out, int *in, int n, int exc){
        … // scan over a singe block (n <= BLOCK_SIZE)
}
__global__ void scanKernel_1(int *out, int *in, int *tmp, int size){
}
__global__ void scanKernel_2(int *out, int *in, int size){
}
```



Scan of arrays larger than 1024

```
__device__ void scanBlock(int *out, int *in, int n, int exc){
        … // scan over a singe block (n <= BLOCK_SIZE)
}
__global__ void scanKernel_1(int *out, int *in, int *tmp, int size){
}
__global__ void scanKernel_2(int *out, int *in, int size){
}
__global__ void scanKernel_3(int *d_out, int *d_tmp, int size){
}
```



```
#define BLOCK_SIZE 1024

void scan(int *out, int *in, int size){
    int *d_in, *d_out, *d_tmp_in, *d_tmp_out;
    int *tmp;
    int num_blocks = size / BLOCK_SIZE;
    if (size % BLOCK_SIZE)
        num_blocks++;

    cudaMalloc((void **)&d_in, size*sizeof(int));
    cudaMalloc((void **)&d_out, size*sizeof(int));
    cudaMalloc((void **)&d_tmp_in, num_blocks*sizeof(int));
    cudaMalloc((void **)&d_tmp_out, num_blocks*sizeof(int));

    cudaMemcpy(d_in, in, size*sizeof(int), cudaMemcpyHostToDevice);

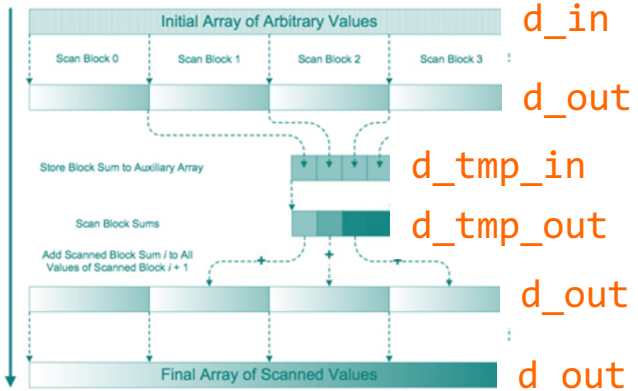
    scanKernel_1<<<num_blocks, BLOCK_SIZE>>>(d_out, d_in, d_tmp_in, size);

    scanKernel_2<<<1, num_blocks>>>(d_tmp_out, d_tmp_in, num_blocks);

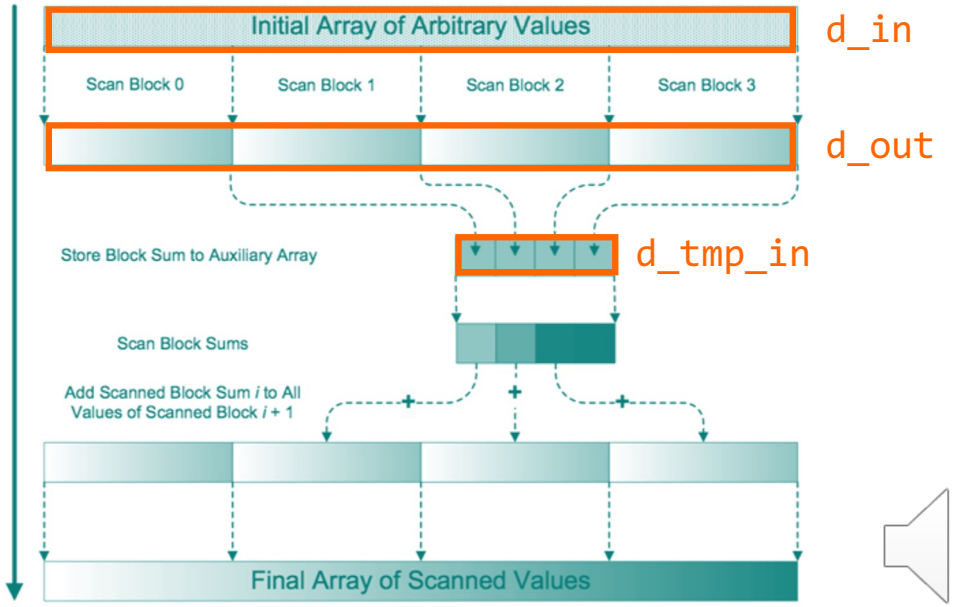
    scanKernel_3<<<num_blocks-1, BLOCK_SIZE>>>(d_out, d_tmp_out, size);

    cudaMemcpy(out, d_out, size*sizeof(int), cudaMemcpyDeviceToHost);

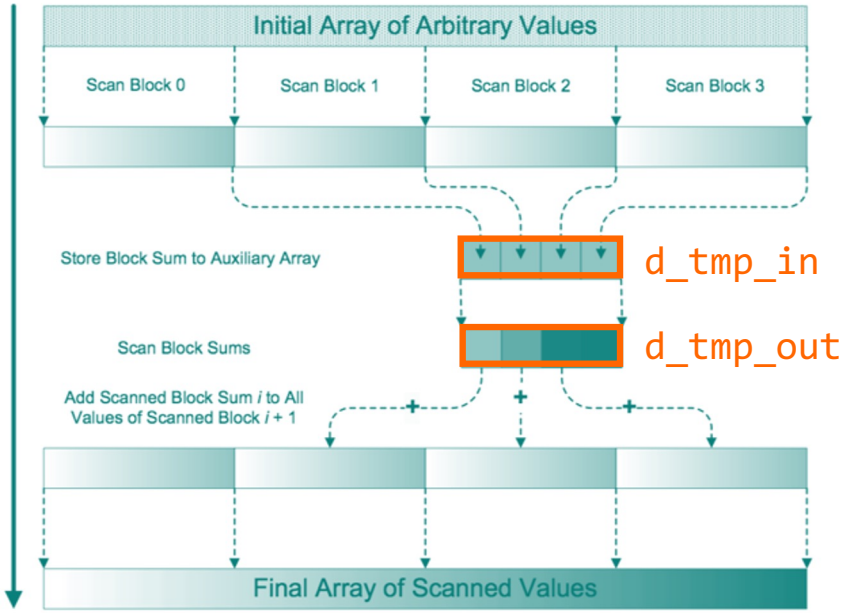
    cudaFree(d_in); cudaFree(d_out); cudaFree(d_tmp_in); cudaFree(d_tmp_out);
}
```



```
__device__ void scanBlock(int *d_out, int *d_in, int n, int exc){
        … // scan over a singe block (n <= BLOCK_SIZE)
}
__global__ void scanKernel_1(int *d_out, int *d_in, int *d_tmp, int size){
    int block_id = blockIdx.x;
    int offset = block_id * BLOCK_SIZE;
    int n = BLOCK_SIZE;
    if (block_id == gridDim.x - 1) // dernier bloc
         n = size – block_id * BLOCK_SIZE;
    scanBlock(&(d_out[offset]), &(d_in[offset]), n, 0);
    if (threadIdx.x == 0)
         d_tmp[block_id] = d_out[offset + n-1] ;
}
```



```
__device__ void scanBlock(int *d_out, int *d_in, int n, int exc){
    …
}
__global__ void scanKernel_2(int *d_out, int *d_in, int size){
    scanBlock(d_out, d_in, size, 0);
}
```



```
__global__ void scanKernel_3(int *d_out, int *d_tmp, int size){
    int block_id = blockIdx.x;
    int offset = (block_id+1) * BLOCK_SIZE;
    int index = offset + threadIdx.x;
    if (index < size)
         d_out[index] += d_tmp[block_id];
}
```