CUDA Programming

Gianluigi Zavattaro
Dip. di Informatica—Scienza e Ingegneria (DISI)
Università di Bologna

gianluigi.zavattaro@unibo.it

Acknowledgments

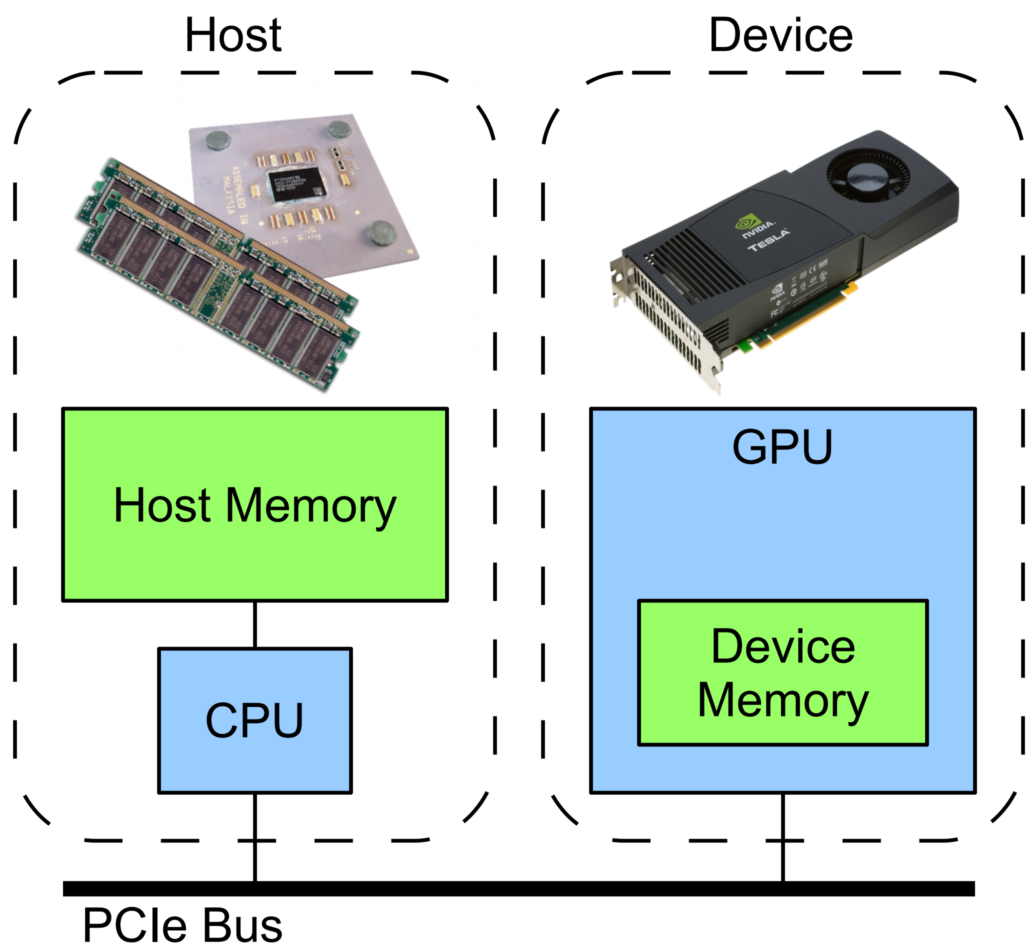
- Most of the content of this presentation is from Mark Harris (Nvidia Corporation), "*CUDA C/C++ BASICS*"
  - http://developer.download.nvidia.com/compute/developertrainingmaterials/presentations/cuda_language/Introduction_to_CUDA_C.pptx
- Salvatore Orlando (Univ. Ca' Foscari di Venezia)
- Tim Mattson (Intel Labs)
  - "*Hands-on Intro to CUDA for OpenCL programmers*"
- CUDA C programming guide
  - http://docs.nvidia.com/cuda/cuda-c-programming-guide/
- Steve Rennich
  - "*CUDA C/C++ Streams and Concurrency*"

Introduction

- Manycore GPUs (Graphics Processing Units) are available in almost all current hardware platforms
- Originally, these processors have been designed for graphics applications
  - Because of the high potential of data parallelism in graphics applications, the design of GPU architectures relied on specialized processor cores
- In addition to graphics processing, GPUs can also be employed for general non-graphics applications
  - If data parallelism is large enough to fully utilize the high number of compute cores in a GPU
- The trend to use GPUs for general numerical applications has inspired GPU manufacturers, such as NVIDIA, to develop the programming environment CUDA and OpenCL

Terminology

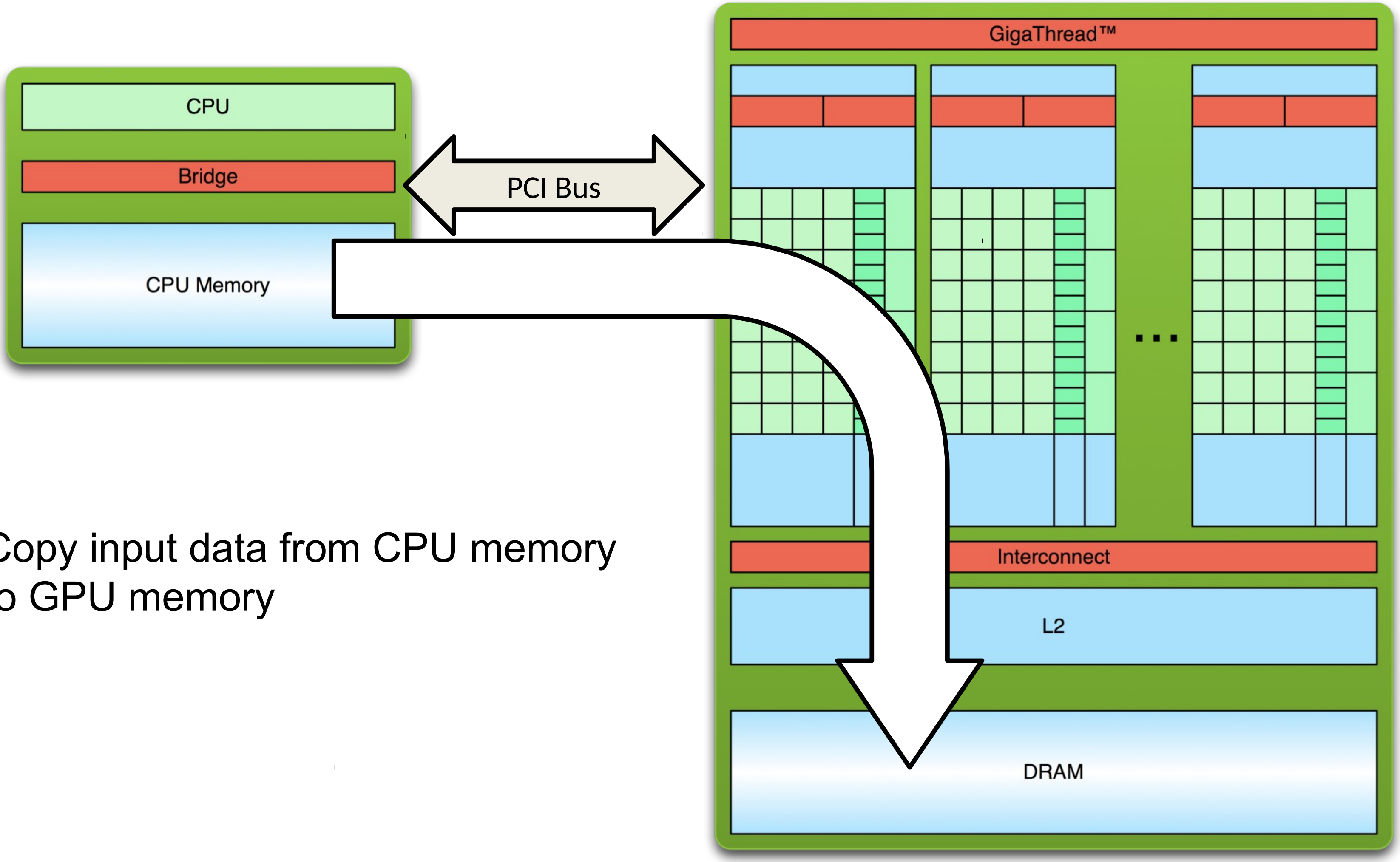
- Host
  - The **CPU** and its memory (host memory)
- Device
  - The **GPU** and its memory (device memory)



Basic concepts

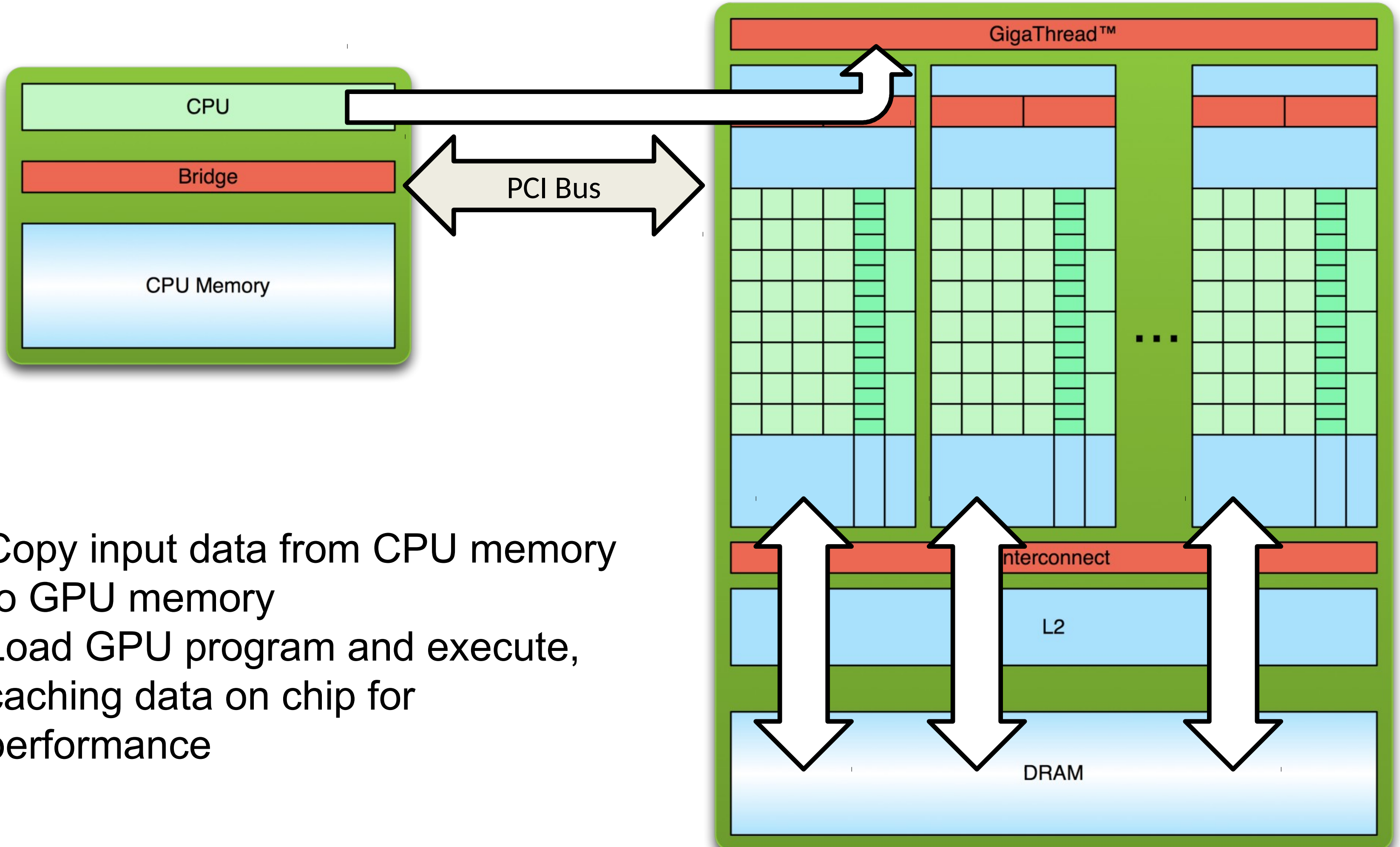
- CUDA separate a program into
  - a CPU program (the host program), which includes all I/O operations or operation for user interaction, and
  - a GPU program (the device program), which contains all computations to be executed on the GPU.
- The simplest case of interaction between host program and device program is implemented by
  - a host program that first copies data into the global memory of the GPU
  - the same host program calls device functions to initiate the processing on the GPU

Simple Processing Flow



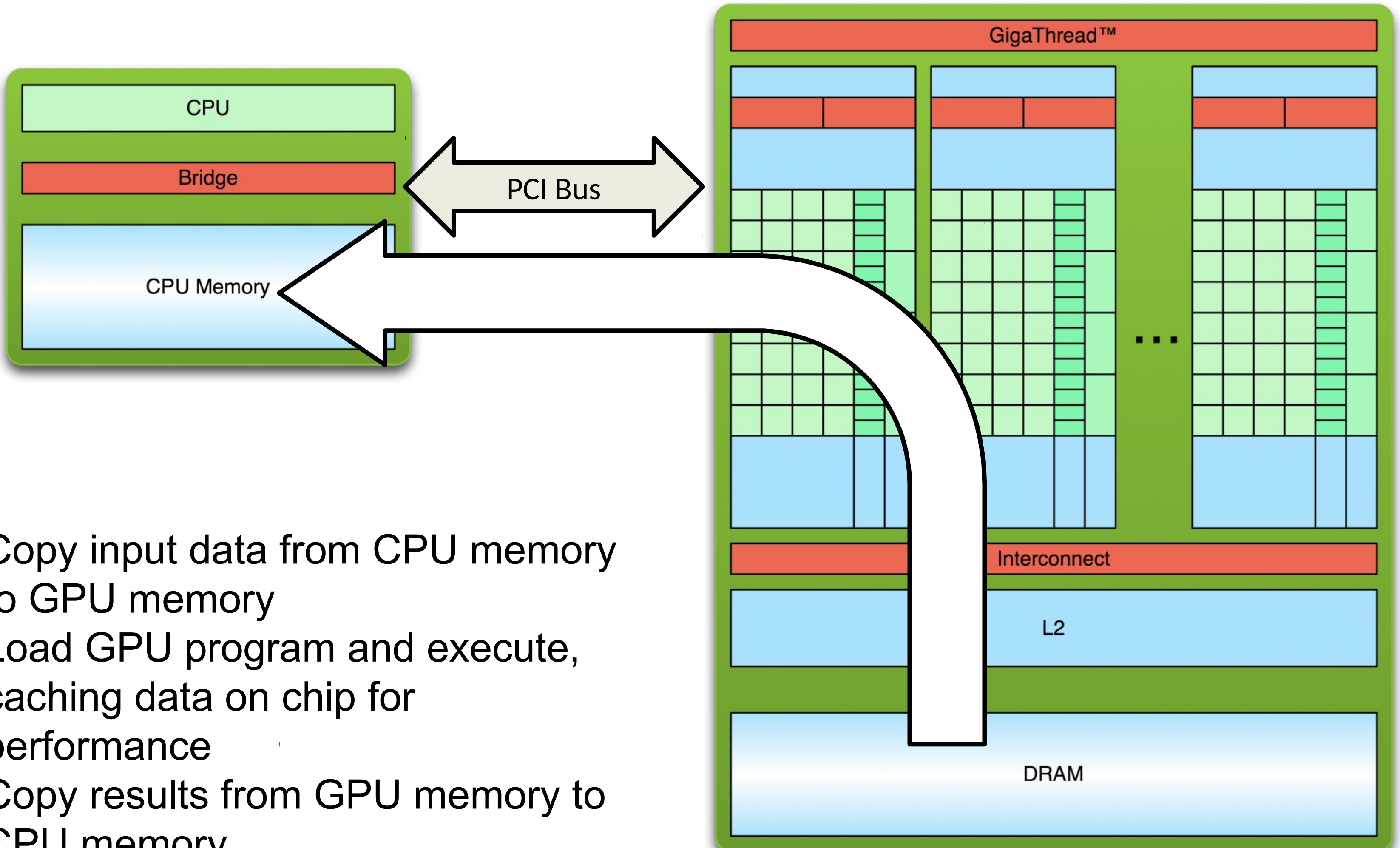
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

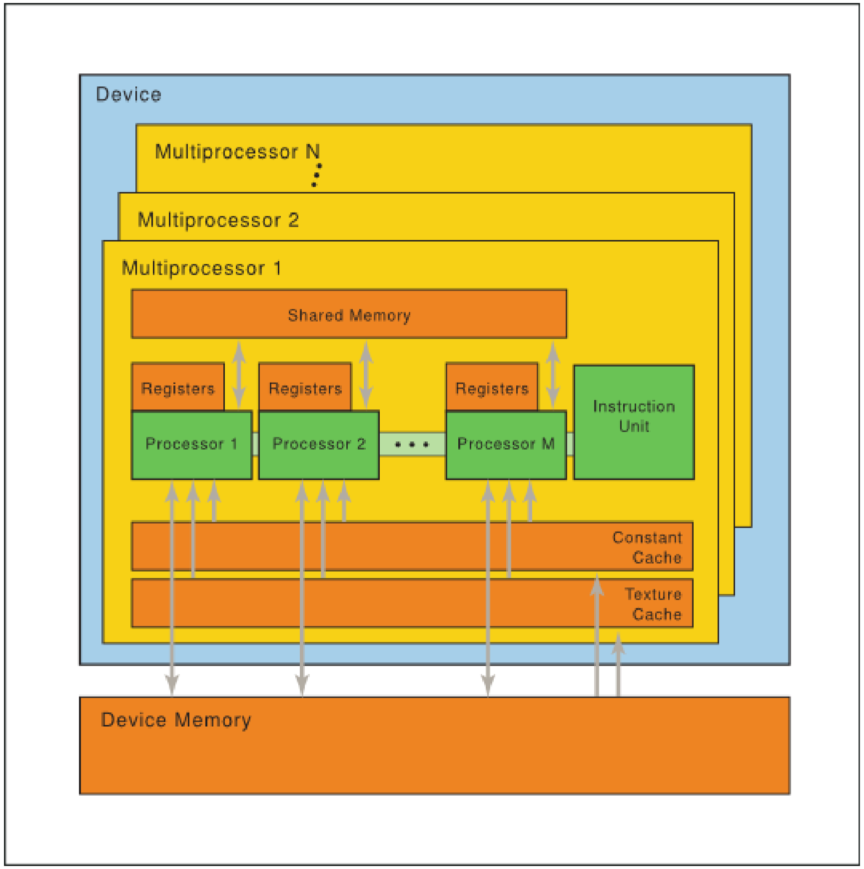
Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Terminology

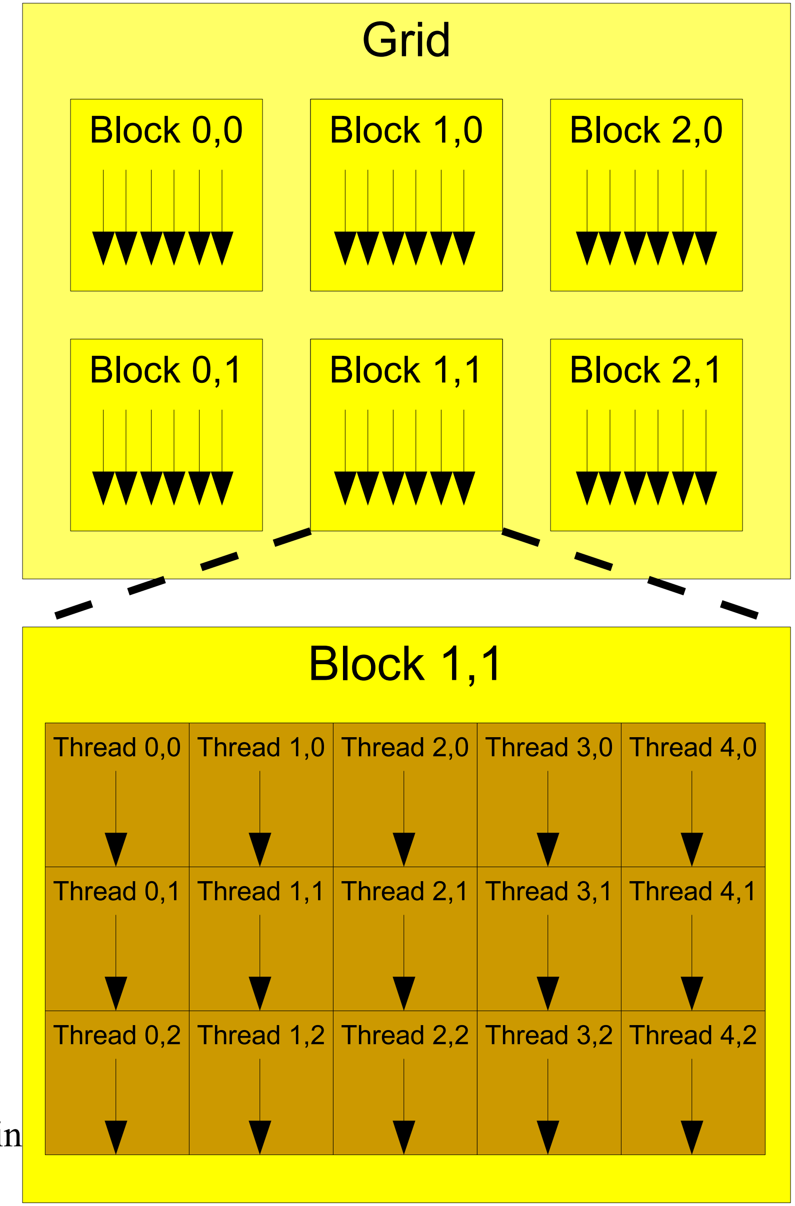
- A GPU comprises several multi-threaded SIMD processors
  - SIMD processor = Streaming Multiprocessors (SMs) composed of many Streaming Processors (SPs)
- Each SIMD processors has several functional units (cores) that can execute the same SIMD instruction on different data
- The actual number of SIMD processors depends on the GPU model
  - For example, the NVIDIA GTX480 GPU has up to 15 SMs



http://docs.nvidia.com/cuda/parallel-thread-execution/

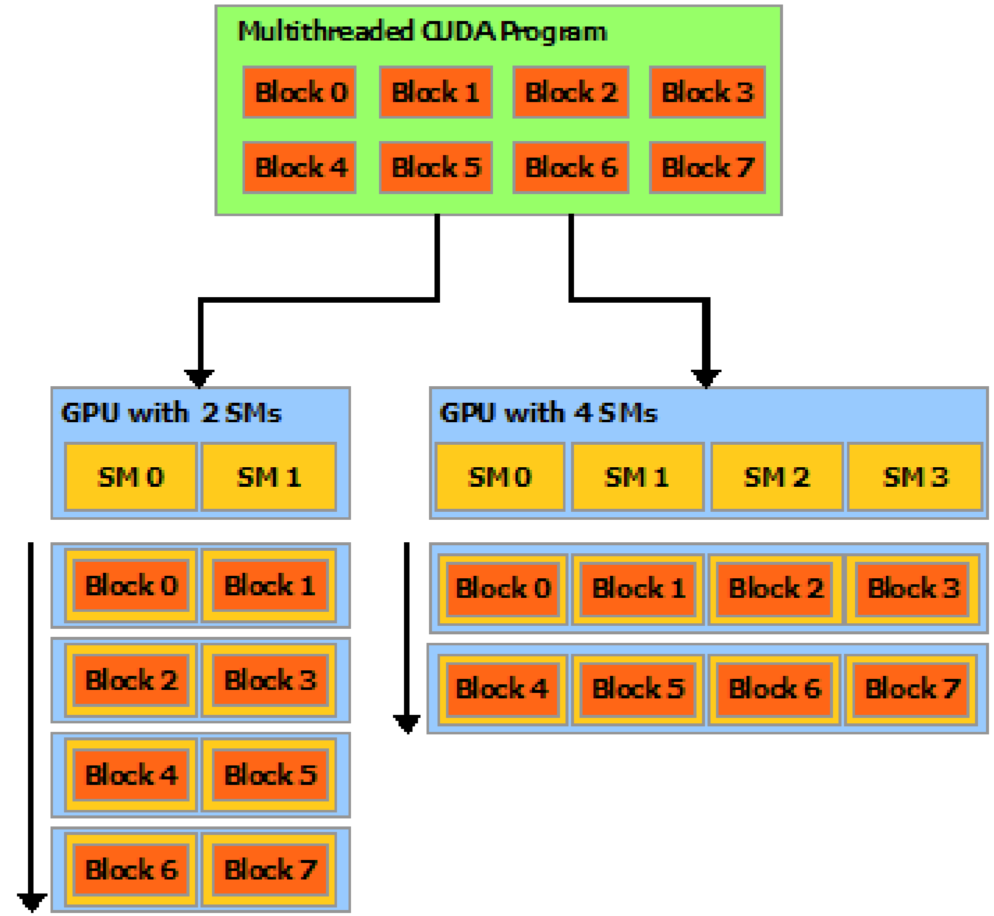
Terminology

- A THREAD is the minimal unit of work.
  - All the threads execute the same KERNEL function
  - Threads are grouped in WARPs of 32 for scheduling, i.e. WARPs are the minimal units of scheduling
- A BLOCK is an independent subpiece of work that can be executed in any order by a SM
  - A 3D array of threads
  - Max no. of threads in a block: 512 threads (up to Version 2) and 1024 threads (since Version 3).
- A GRID is a piece of work that can be executed by the GPU
  - A 2D array of BLOCKs (3D since CUDA Version 3)



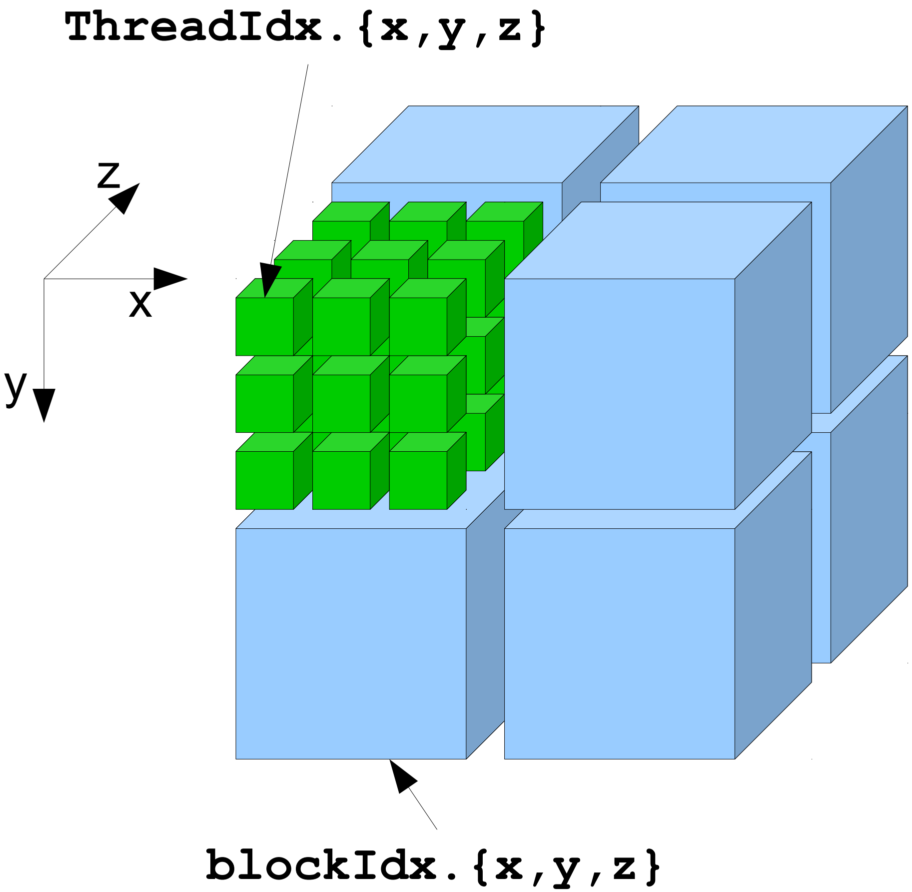
Automatic scalability

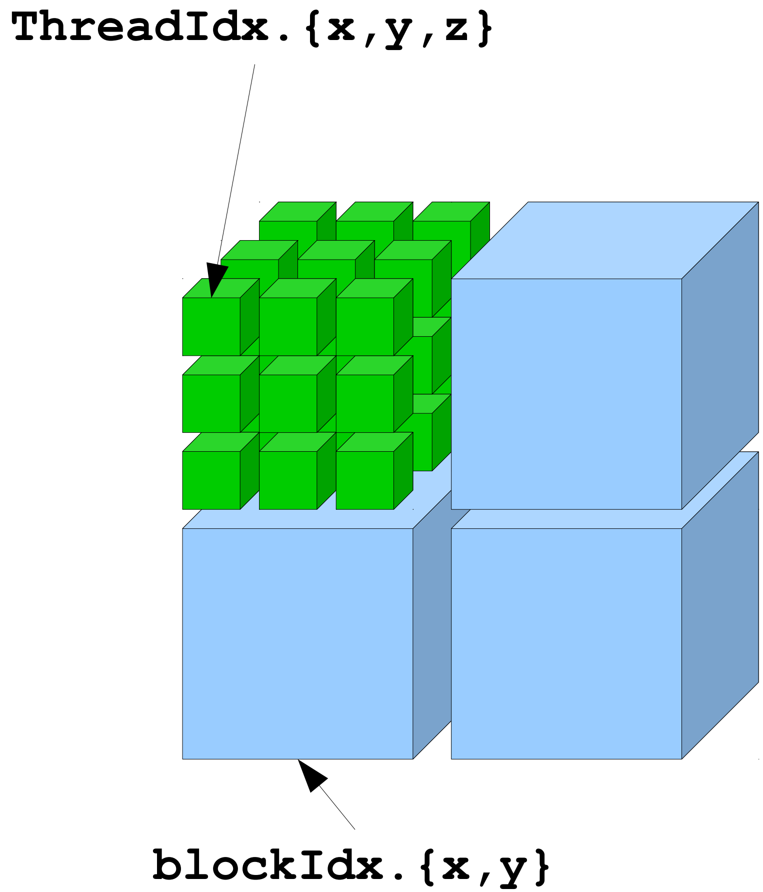
- Do we need to take care of the device computing power (number of SM) ?
  - No, because the block scheduler can re-arrange blocks accordingly
  - A Grid contains a set of independent blocks, which can be executed in any order



http://docs.nvidia.com/cuda/cuda-c-programming-guide/

Summary





Compute Capability ≥ 2.x

Compute Capability < 2.x

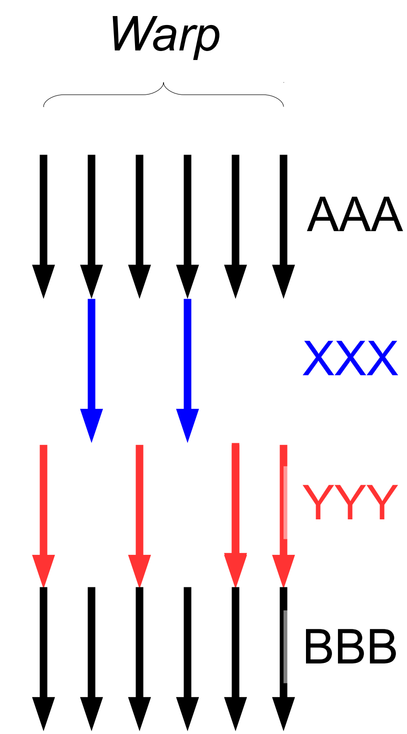
CUDA thread scheduling

- A CUDA warp is a group of 32 CUDA threads that execute simultaneously
  - The hardware is most efficiently utilized when all threads in a warp execute instructions from the same program address
  - If threads in a warp diverge, then some execution pipelines go unused
  - If threads in a warp access aligned, contiguous blocks of DRAM, the accesses are coalesced into a single high-bandwidth access
- A CUDA warp represents the minimum granularity of efficient SIMD execution

Single Instruction Multiple Data

- Individual threads of a warp start together at the same program address
- Each thread has its own program counter and register state
  - Each thread is free to branch and execute independently
  - Provides the MIMD abstraction branch behavior
- Each branch will be executed serially
  - Threads not following the current branch will be disabled

```
int x;
AAA
if (x > 0) {
    XXX
} else {
    YYY
}
BBB
```



Hands-on introduction to CUDA programming

Hello World!

- Standard C that runs on the host
- The NVIDIA compiler (**`nvcc`**) can be used to compile programs, even with no device code

```
/* cuda-hello0.cu */
#include <stdio.h>
int main(void)
{
    printf("Hello World!\n");
    return 0;
}
```

```
$ nvcc hello_world.cu
$ ./a.out
Hello World!
```

Hello World! with Device Code

```
/* cuda-hello1.cu */
#include <stdio.h>
__global__ void mykernel(void) { }

int main(void)
{
    mykernel<<<1,1>>>( );
    printf("Hello World!\n");
    return 0;
}
```

Two new syntactic elements

Hello World! with Device Code

```
__global__ void mykernel(void) {  }
```

- CUDA/C keyword **`__global__`** indicates a function that:
  - Runs on the device
  - Is called from host code
  - **`__global__`** functions must return **`void`**
- **`nvcc`** separates source code into host and device components
  - Device functions (e.g., **`mykernel()`**) are processed by the NVIDIA compiler
  - Host functions (e.g., **`main()`**) are processed by the standard host compiler (e.g., **`gcc`**)

Hello World! with Device Code

```
mykernel<<<1,1>>>( );
```

- Triple angle brackets mark a call from host code to device code
  - Also called a “kernel launch”
  - We’ll return to the parameters (1,1) in a moment
- That’s all that is required to execute a function on the GPU!

Hello World! with Device Code

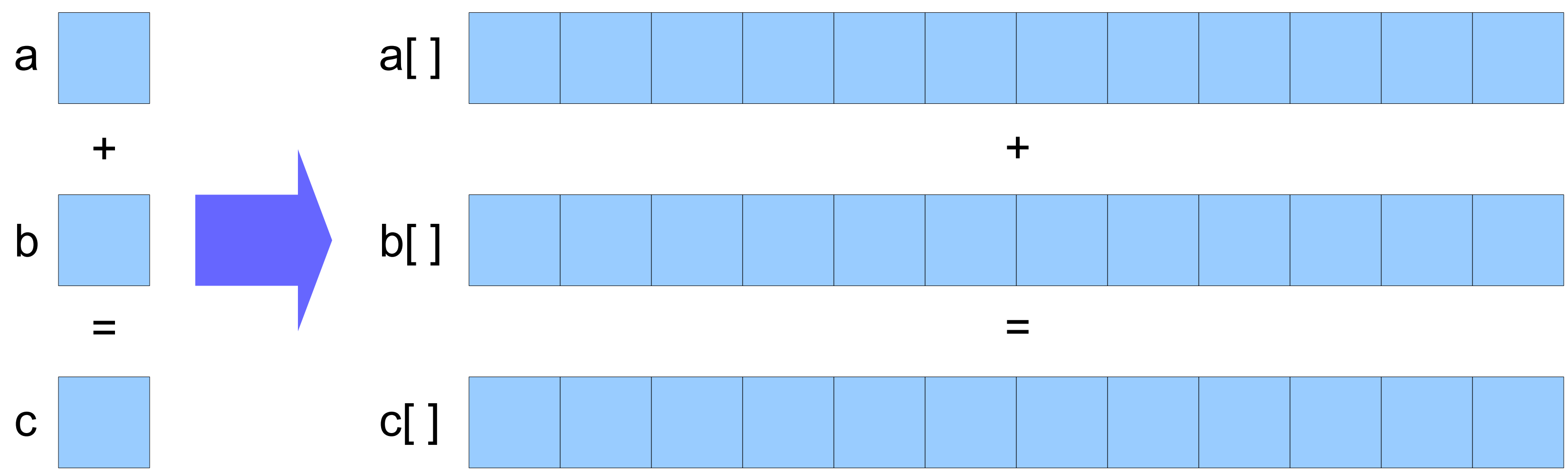
```
__global__ void mykernel(void)
{  }

int main(void) {
    mykernel<<<1,1>>>( );
    printf("Hello World!\n");
    return 0;
}
```

```
$ nvcc cuda-hello1.cu
$ ./a.out
Hello World!
```

- **`mykernel()`** does nothing

Parallel Programming in CUDA/C

- But wait… GPU computing is about massive parallelism!
- We need a more interesting example…
- We'll start by adding two integers and build up to vector addition



Addition on the Device

- A simple kernel to add two integers

```
__global__ void add(int *a, int *b, int *c)
{
    *c = *a + *b;
}
```

- As before **`__global__`** is a CUDA C/C++ keyword meaning
  - **`add()`** will execute on the device
  - **`add()`** will be called from the host

Addition on the Device

- Note the use of pointers for the variables

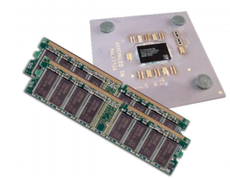
```
__global__ void add(int *a, int *b, int *c)
{
    *c = *a + *b;
}
```

- **`add()`** runs on the device, so *a*, *b* and *c* must point to device memory
- We need to allocate memory on the GPU

Memory Management

- Host and device memory are separate entities
  - Device pointers point to GPU memory
    - May be passed to/from host code
    - May **not** be dereferenced in host code

  - Host pointers point to CPU memory
    - May be passed to/from device code
    - May **not** be dereferenced in device code



- Simple CUDA API for handling device memory
  - **`cudaMalloc()`**, **`cudaFree()`**, **`cudaMemcpy()`**
  - Similar to the C equivalents **`malloc()`**, **`free()`**, **`memcpy()`**

Addition on the Device: `main()`

```
/* cuda-vecadd0.cu */
int main(void) {
    int a, b, c;              /* host copies of a, b, c */
    int *d_a, *d_b, *d_c;     /* device copies of a, b, c */
    const size_t size = sizeof(int);
    /* Allocate space for device copies of a, b, c */
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);
    /* Setup input values */
    a = 2; b = 7;
    /* Copy inputs to device */
    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);
    /* Launch add() kernel on GPU */
    add<<<1,1>>>(d_a, d_b, d_c);
    /* Copy result back to host */
    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
    /* Cleanup */
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Coordinating Host & Device

- Kernel launches are asynchronous
  - Control returns to the CPU immediately
- CPU needs to synchronize before consuming the results

`cudaMemcpy()`	Blocks the CPU until the copy is complete Copy begins when all preceding CUDA calls have completed
`cudaMemcpyAsync()`	Asynchronous, does not block the CPU
`cudaDeviceSynchronize()`	Blocks the CPU until all preceding CUDA calls have completed

Running in parallel

Moving to Parallel

- GPU computing is about massive parallelism
  - So how do we run code in parallel on the device?

```
add<<< 1, 1 >>>();
```

```
add<<< N, 1 >>>();
```

# of blocks *# of threads per block*

- Instead of executing **`add()`** once, execute *N* times in parallel

Vector Addition on the Device

- With **`add()`** running in parallel we can do vector addition
- Terminology: each parallel invocation of **`add()`** is referred to as a block
  - The set of blocks is referred to as a grid
  - Each invocation can refer to its block index using **`blockIdx.x`**

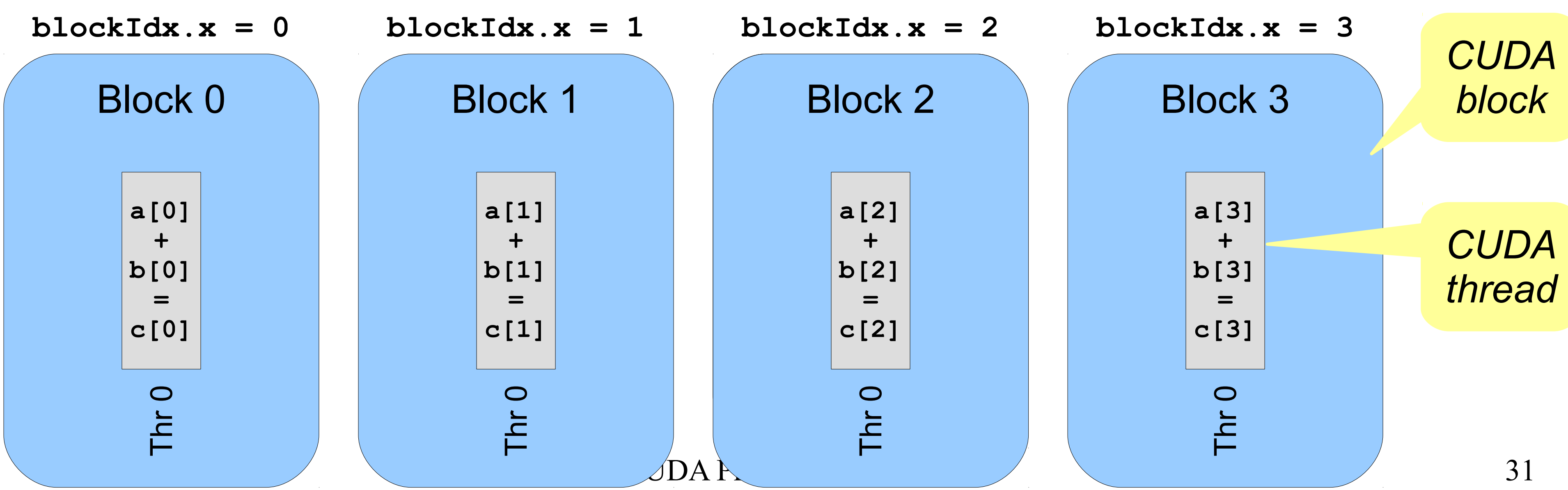
```
__global__ void add(int *a, int *b, int *c)
{
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- By using **`blockIdx.x`** to index into the array, each block handles a different index

Vector Addition on the Device

- On the device, each block can execute in parallel

```
__global__ void add(int *a, int *b, int *c)
{
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```



```
/* cuda-vecadd1.cu */
#define N 1024
int main(void) {
    int *a, *b, *c;          /* host copies of a, b, c */
    int *d_a, *d_b, *d_c;    /* device copies of a, b, c */
    const size_t size = N * sizeof(int);
    /* Alloc space for device copies of a, b, c */
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);
    /* Alloc space for host copies of a,b,c and setup input values */
    a = (int *)malloc(size); vec_init(a, N);
    b = (int *)malloc(size); vec_init(b, N);
    c = (int *)malloc(size);
    /* Copy inputs to device */
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
    /* Launch add() kernel on GPU with N blocks */
    add<<<N,1>>>(d_a, d_b, d_c);
    /* Copy result back to host */
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
    /* Cleanup */
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Review

- Difference between host and device
  - Host ↔ CPU, device ↔ GPU
- Using `__global__` to declare a function as device code
  - Executes on the device
  - Called from the host
- Passing parameters from host code to a device function
- Basic device memory management
  - `cudaMalloc()`
  - `cudaMemcpy()`
  - `cudaFree()`
- Launching parallel kernels
  - Launch *N* copies of `add()` with `add<<<N,1>>>(…);`
  - Use `blockIdx.x` to access block index

Introducing threads

CUDA Threads

- Terminology: a block can be split into parallel threads
- Let's change **`add()`** to use parallel threads instead of parallel blocks

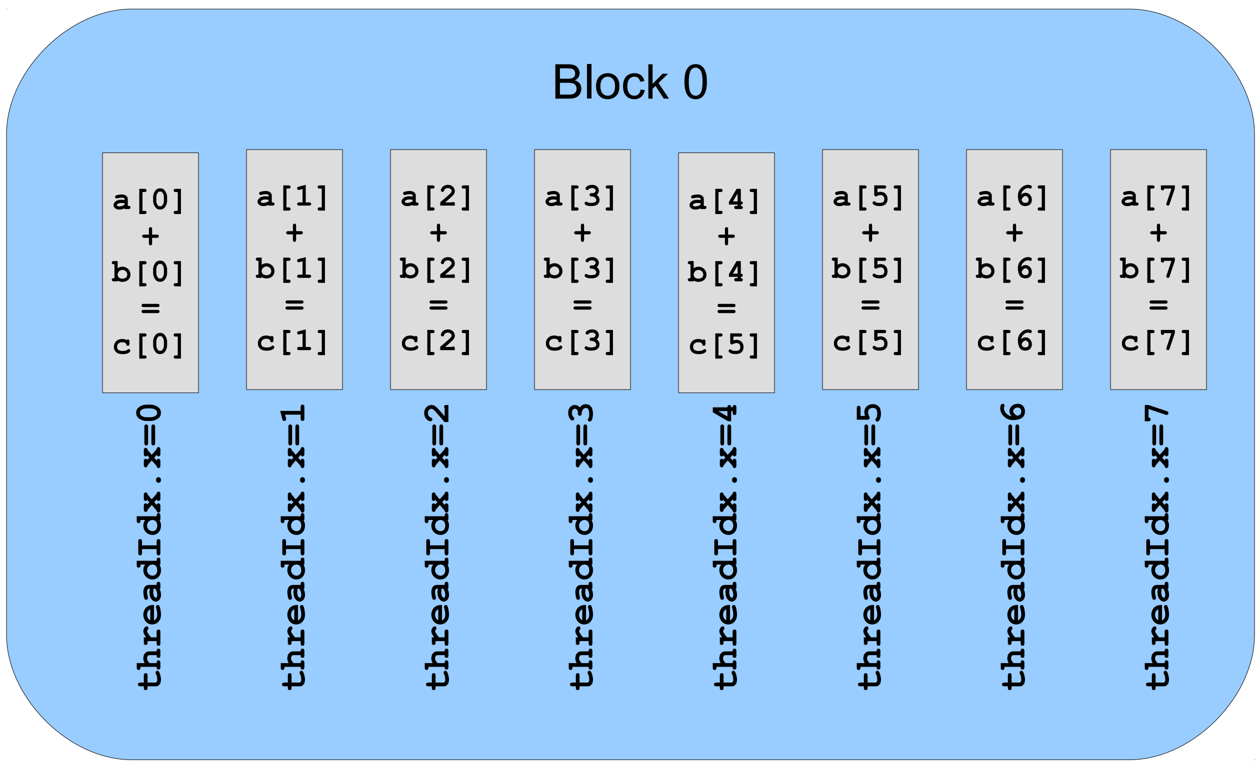
```
__global__ void add(int *a, int *b, int *c)
{
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

- We use **`threadIdx.x`** instead of **`blockIdx.x`**
- Need to make one change in **`main()`**…

CUDA Threads

```
__global__ void add(int *a, int *b, int *c)
{
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

blockIdx.x = 0



```
/* cuda-vecadd2.cu */
#define N 1024
int main(void) {
    int *a, *b, *c;          /* host copies of a, b, c */
    int *d_a, *d_b, *d_c;    /* device copies of a, b, c */
    const size_t size = N * sizeof(int);
    /* Alloc space for device copies of a, b, c */
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);
    /* Alloc space for host copies of a,b,c and setup input values */
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
    /* Copy inputs to device */
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
    /* Launch add() kernel on GPU with N threads */
    add<<<1,N>>>(d_a, d_b, d_c);
    /* Copy result back to host */
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
    /* Cleanup */
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

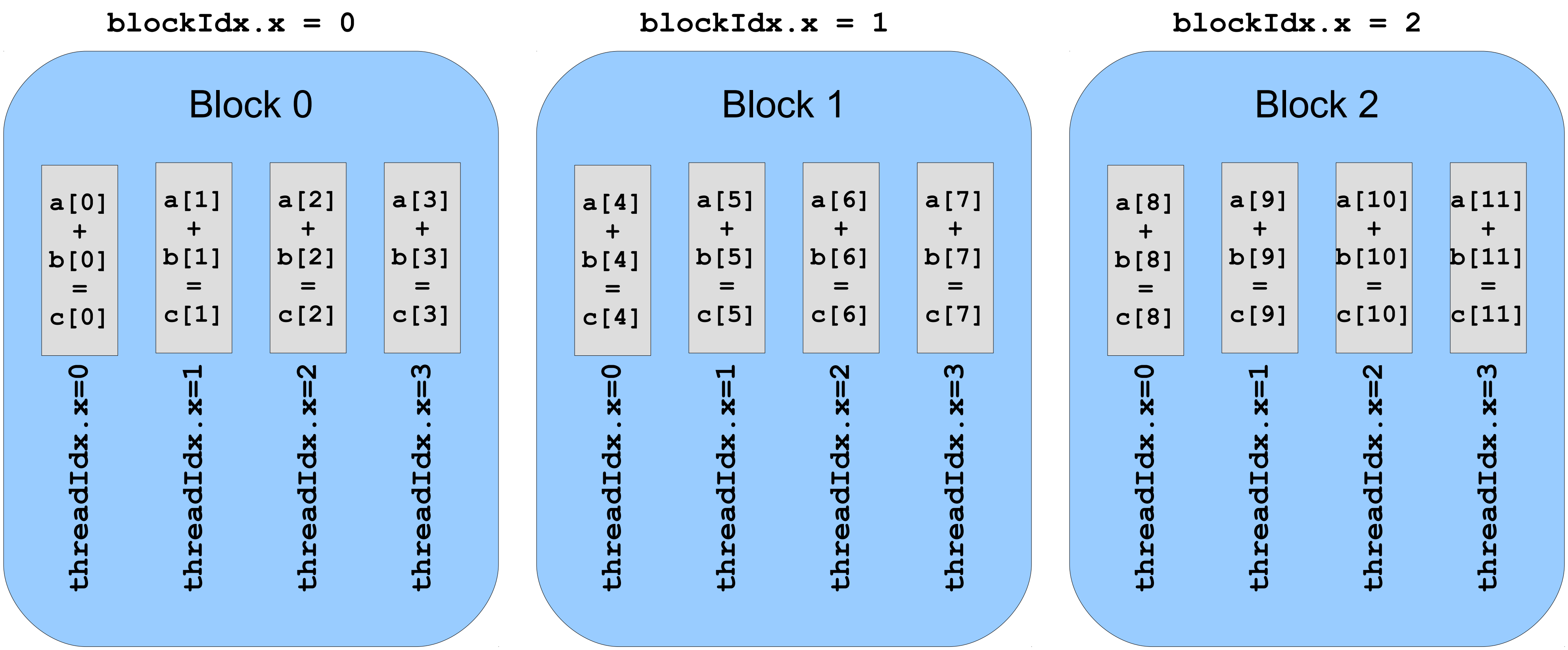
Combining threads and blocks

Combining Blocks and Threads

- We have seen parallel vector addition using:
  - Many blocks with one thread each
  - One block with many threads
- Let's adapt vector addition to use both blocks and threads
  - Why? We'll come to that…
- First let's discuss data indexing…

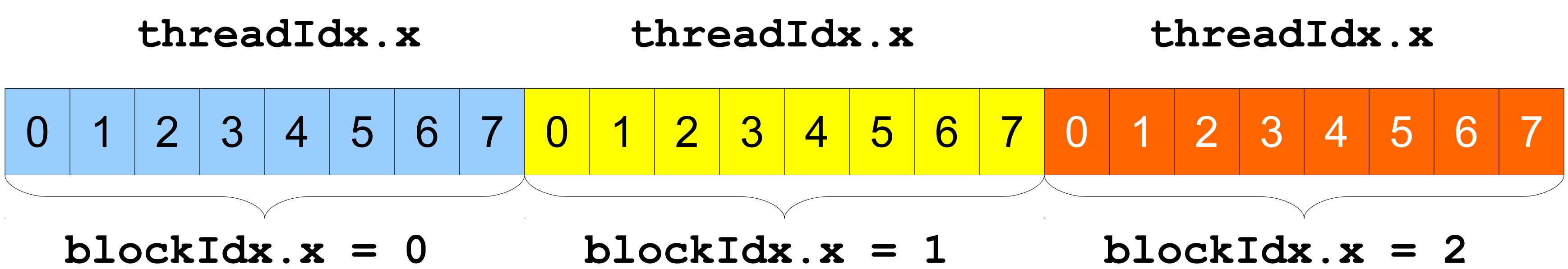
Combining Blocks and Threads

- We must somehow assign to each thread a different array element to operate on



Indexing Arrays with Blocks and Threads

- No longer as simple as using **`blockIdx.x`** or **`threadIdx.x`** alone
- Consider indexing an array with one element per thread, 8 threads per block

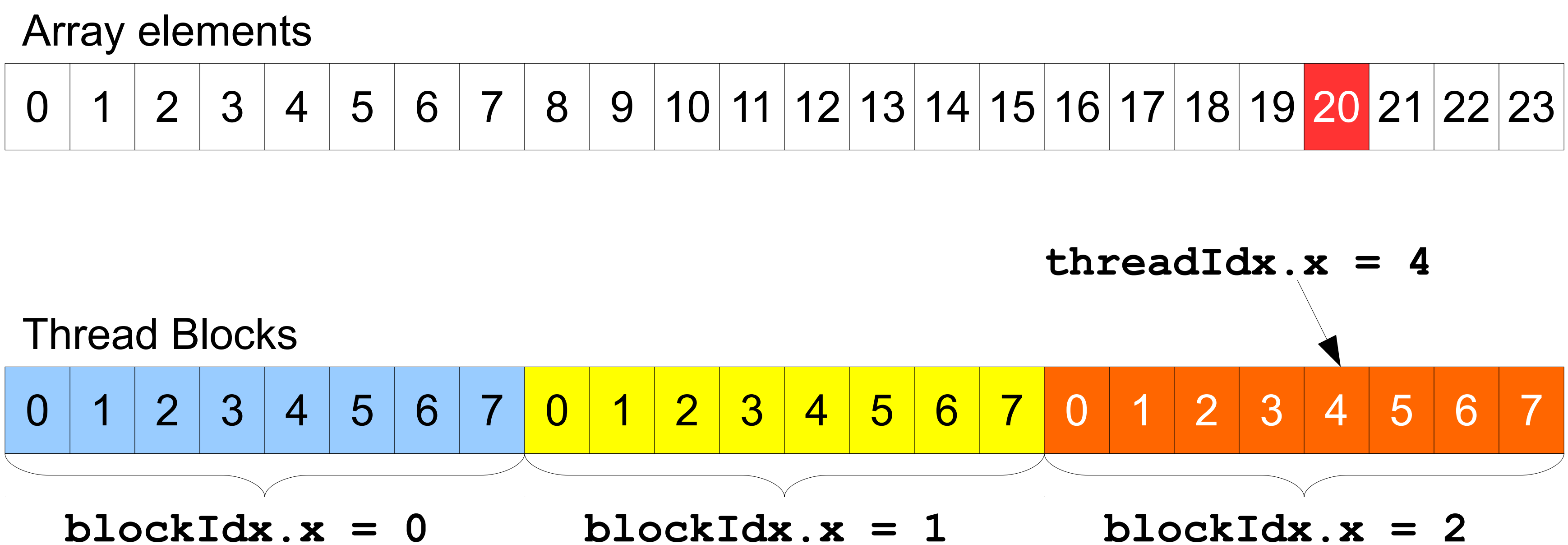


- With *M* threads per block a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```

Indexing arrays: Example

- Which thread will operate on the red element?




```
int index = threadIdx.x + blockIdx.x * M;
          =       4     +      2      * 8;
          = 20;
```

Vector Addition with Blocks and Threads

- Use the built-in variable **`blockDim.x`** to get the number of threads per block

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

- Combined version of **`add()`** to use parallel threads and parallel blocks:

```
__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```

- What changes need to be made in **`main()`**?

Addition with blocks and threads

```
#define N (2048*2048)
#define BLKDIM 1024

int main(void) {
    …
    /* Launch add() kernel on GPU */
    add<<<N/BLKDIM, BLKDIM>>>(d_a, d_b, d_c);
    …
}
```

Number of blocks

Number of threads per block

- However, the problem size might not be multiple of the block size...

Handling arbitrary vector sizes

- Avoid accessing beyond the end of the array

```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n) {
        c[index] = a[index] + b[index];
    }
}
```

- Update kernel launch

```
add<<<(N + BLKDIM-1)/BLKDIM, BLKDIM>>>(d_a, d_b, d_c, N);
```

- See cuda-vecadd3.cu

Review

- Launching parallel kernels
  - Launch ~*N* copies of **`add()`** with
    `add<<<(N + BLKDIM-1)/BLKDIM, BLKDIM>>>(…);`
  - Use **`blockIdx.x`** to access block index
  - Use **`threadIdx.x`** to access thread index within block
- Assign array elements to threads:

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

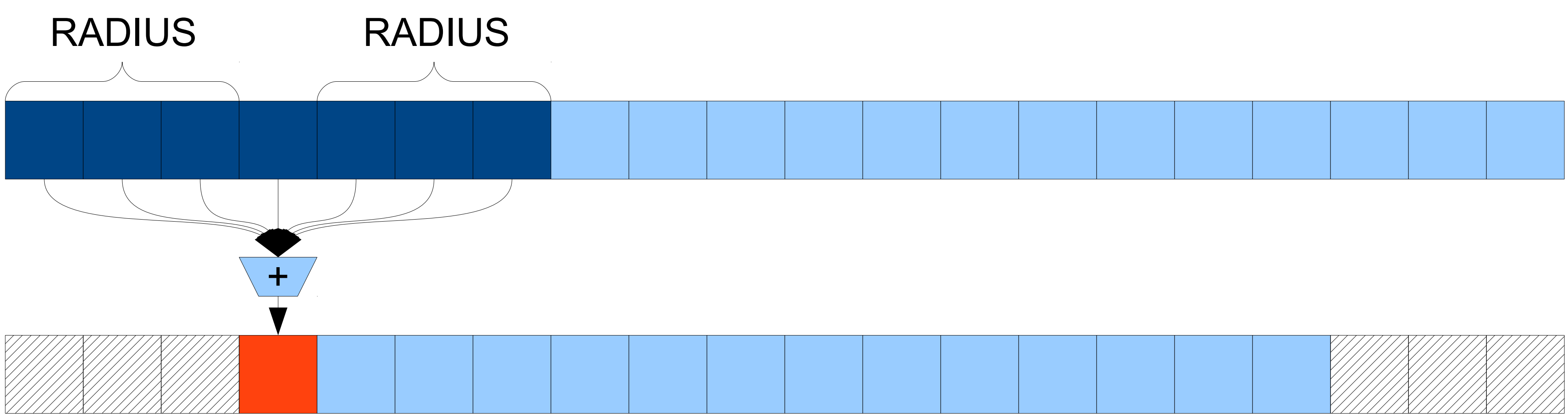
Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?
- Unlike parallel blocks, threads have mechanisms to:
  - Communicate
  - Synchronize
- To look closer, we need a new example…

Cooperating threads

1D Stencil

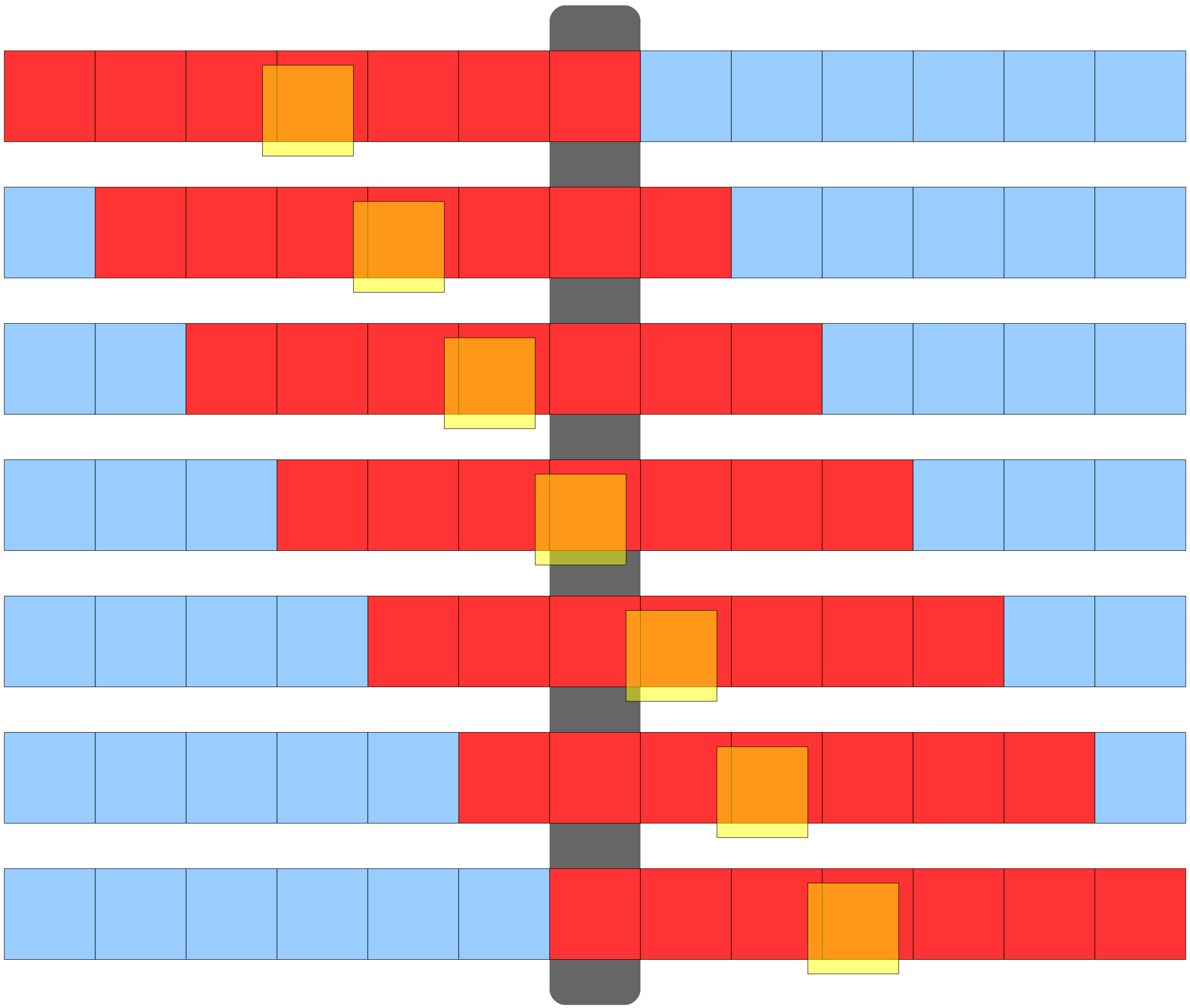
- Consider applying a stencil to a 1D array of elements
  - Each output element is the sum of input elements within a given radius
- If RADIUS is 3, then each output element is the sum of 7 input elements
  - The first and last RADIUS elements of the output array are not computed



See cuda-stencil1d.c

Implementing Within a Block

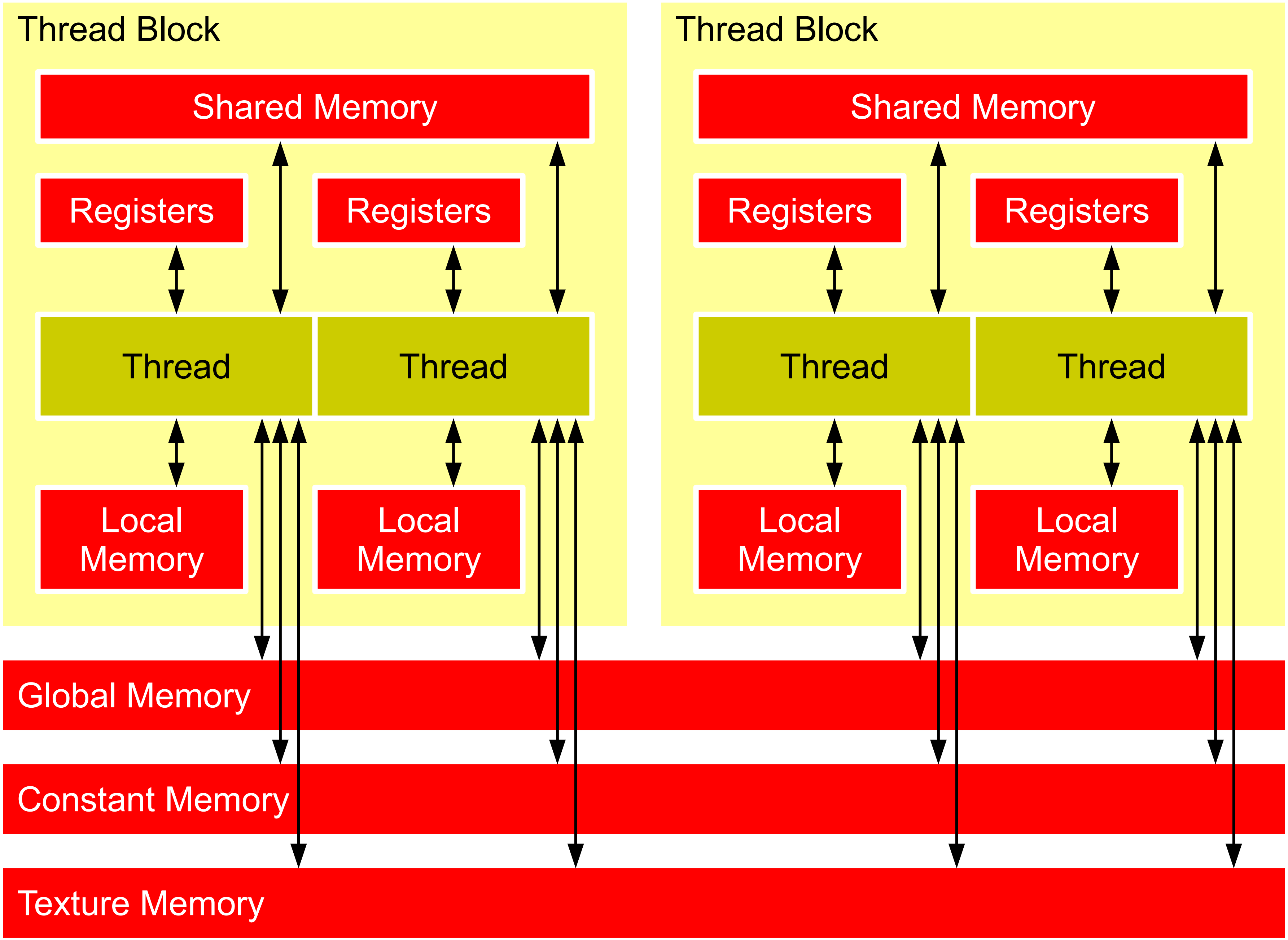
- Each thread processes one output element
  - **`blockDim.x`** elements per block
- Input elements are read several times
  - With radius 3, each input element is read seven times
  - With radius *R*, each input element is read (2*R*+1) times



Sharing Data Between Threads

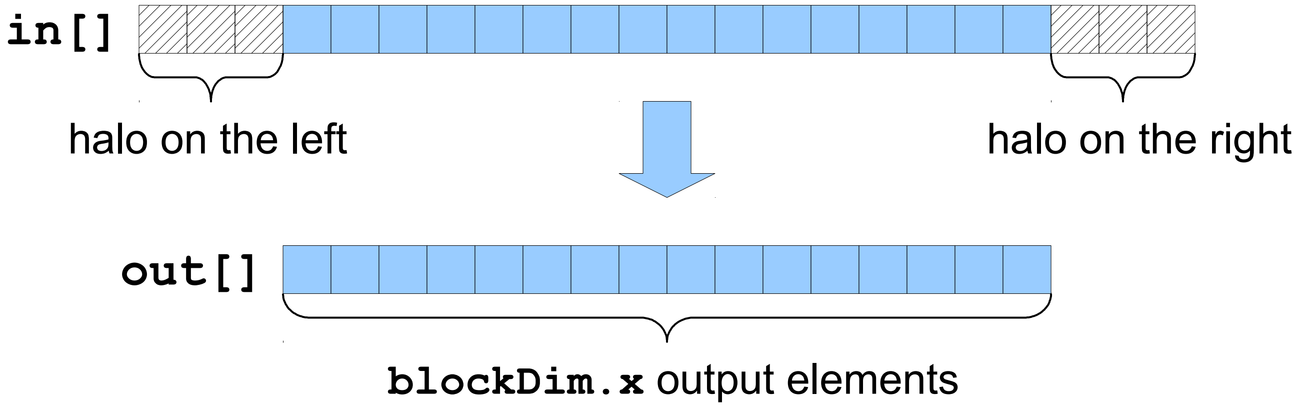
- Global memory accesses are likely to cause a bottleneck due to the limited memory bandwidth
- Within a block, threads can share data via shared memory
  - Extremely fast on-chip memory, user-managed
  - Think of it as a user-managed local cache
- Declare using **`__shared__`**, allocated per thread-block
- Data is not visible to threads in other blocks

CUDA memory model



Implementing with shared memory

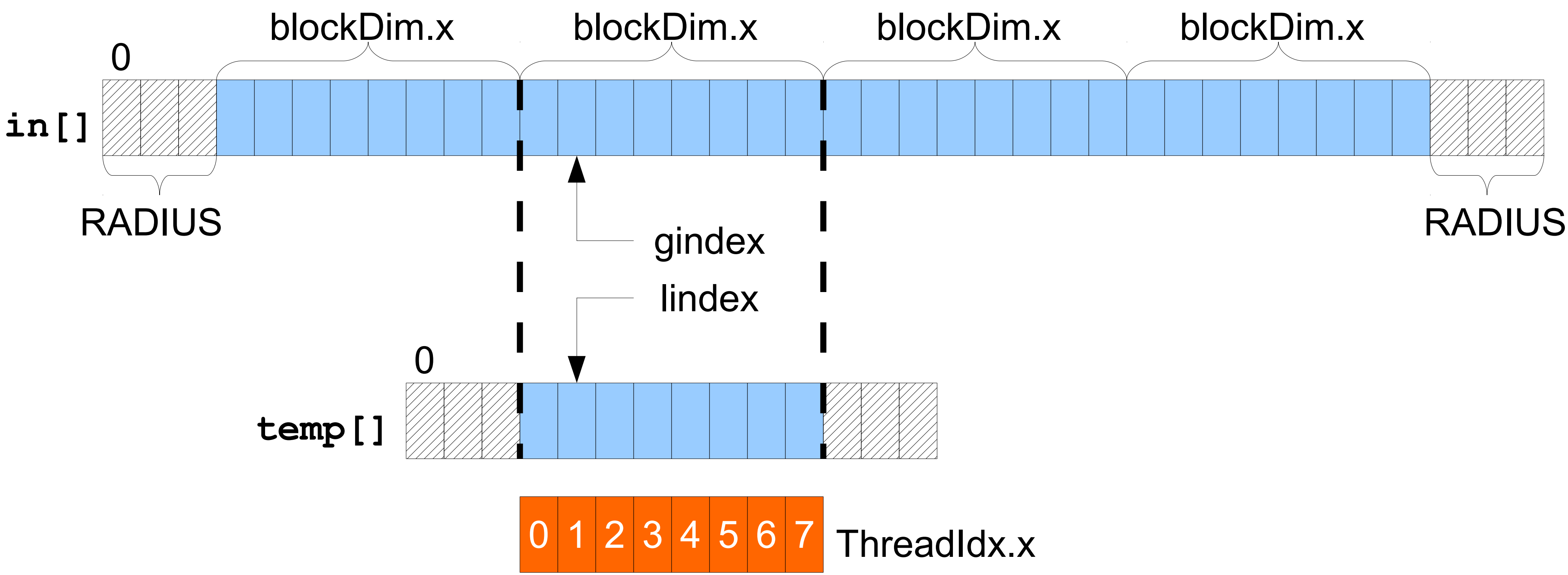
- Cache data in shared memory
  - Read (**`blockDim.x`** + 2 $\times$ radius) input elements from global memory to shared memory
  - Compute **`blockDim.x`** output elements
  - Write **`blockDim.x`** output elements to global memory
  - Each block needs a halo of *radius* elements at each boundary



Implementing with shared memory

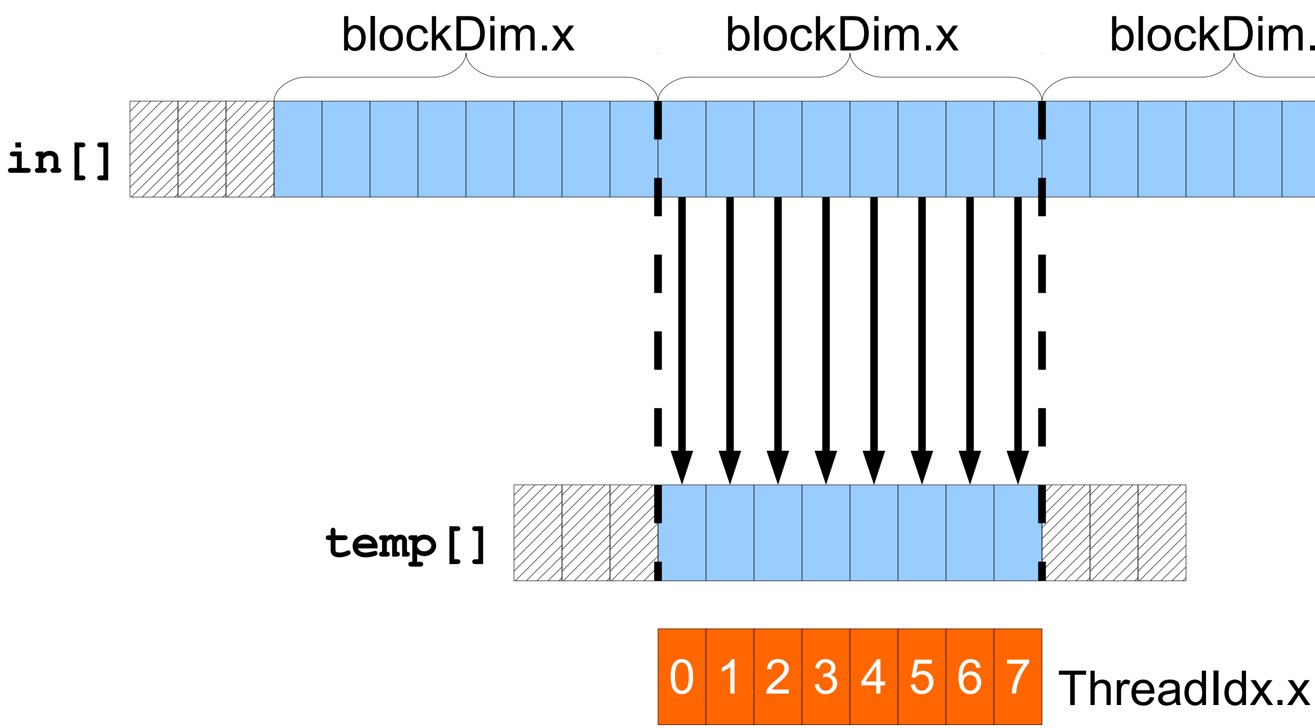
- Let us make a few simplifying assumptions
  - The array length is a multiple of the thread block size
  - The input (and output) array already includes an halo of **`2*RADIUS`** elements
    - The halo is ignored for the output array
- Idea
  - Each thread block keeps a local cache of **`blockDim.x + 2*RADIUS`** elements
  - Each thread copies one element from the global array to the local cache
  - The first **`RADIUS`** threads also take care of of filling the halo

Implementing with shared memory



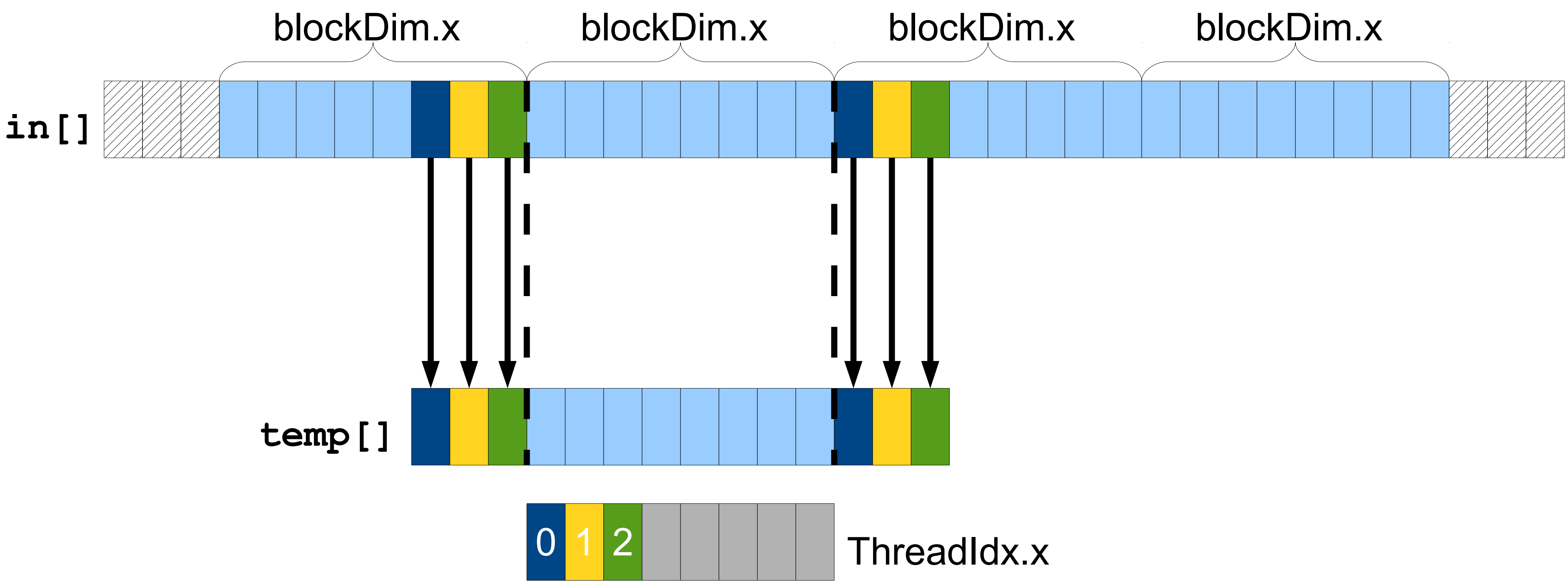

```
__shared__ int temp[BLKDIM + 2 * RADIUS];
const int gindex = threadIdx.x + blockIdx.x * blockDim.x + RADIUS;
const int lindex = threadIdx.x + RADIUS;
/* … */
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
     temp[lindex - RADIUS] = in[gindex - RADIUS];
     temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
```

Implementing with shared memory




```
__shared__ int temp[BLKDIM + 2 * RADIUS];
const int gindex = threadIdx.x + blockIdx.x * blockDim.x + RADIUS;
const int lindex = threadIdx.x + RADIUS;
/* … */
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLKDIM] = in[gindex + BLKDIM];
}
```

Implementing with shared memory




```
__shared__ int temp[BLKDIM + 2 * RADIUS];
const int gindex = threadIdx.x + blockIdx.x * blockDim.x + RADIUS;
const int lindex = threadIdx.x + RADIUS;
/* … */
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLKDIM] = in[gindex + BLKDIM];
}
```

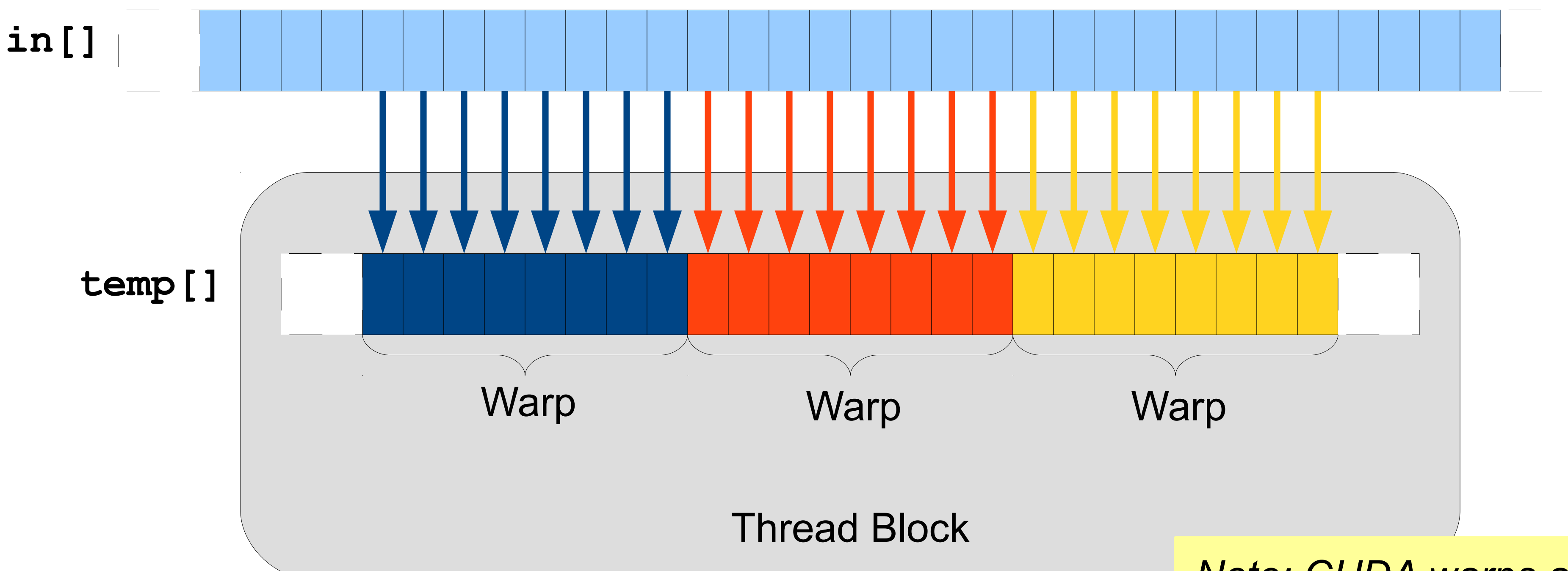
Stencil kernel (does not work!)

Wrong!

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLKDIM + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x + RADIUS;
    int lindex = threadIdx.x + RADIUS;
    int result = 0, offset;
    /* Read input elements into shared memory */
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + blockDim.x] = in[gindex + blockDim.x];
    }
    /* Apply the stencil */
    for (offset = -RADIUS ; offset <= RADIUS ; offset++) {
        result += temp[lindex + offset];
    }
    /* Store the result */
    out[gindex] = result;
}
```

The problem

- All threads are not necessarily fully synchronized
- Suppose that thread (**`blockDim.x`** - 1) reads the halo before thread 0 has fetched it
  - Data race!



Note: CUDA warps actually contain 32 threads

The solution: `__syncthreads()`

- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards
- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

Stencil kernel that works

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLKDIM + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x + RADIUS;
    int lindex = threadIdx.x + RADIUS;
    int result = 0, offset;
    /* Read input elements into shared memory */
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + blockDim.x] = in[gindex + blockDim.x];
    }
    __syncthreads();
    /* Apply the stencil */
    for (offset = -RADIUS ; offset <= RADIUS ; offset++) {
        result += temp[lindex + offset];
    }
    /* Store the result */
    out[gindex] = result;
}
```

See cuda-stencil1d-shared.c

Review

- Use `__shared__` to declare a variable/array in shared memory
  - Data is shared between threads in a block
  - Not visible to threads in other blocks
- Use `__syncthreads()` as a barrier
  - Use to prevent data hazards

Managing the device

Timing CUDA kernels

- You can use the timing routines provided in **`hpc.h`**
- However, since kernel invocations are asynchronous, you must call **`cudaDeviceSynchronize()`** to wait the kernel to complete execution

```
#include "hpc.h"
…
double tstart, tend;

tstart = hpc_gettime();
mykernel<<<X, Y>>>( );    /* kernel invocation        */
cudaDeviceSynchronize(); /* wait for kernel to finish */
tend = hpc_gettime();

printf("Elapsed time %f\n", tend - tstart);
```

The `__device__` qualifier

- The **`__device__`** qualifier defines functions that
  - execute on the device
  - can be called from device code only
- **`__device__`** functions are inlined, so they can return a value

```
__device__ float cuda_fmaxf(float a, float b)
{
    return (a>b ? a : b);
}

__global__ void my_kernel( float *v, int n )
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i<n) {
        v[i] = cuda_fmaxf(1.0, v[i]);
    }
}
```

The `__host__` qualifier

- (Optional) denotes a function that
  - is executed on the host
  - can be called from host code only
  - default behavior when neither `__global__` nor `__device__` are specified
- You can use both the `__host__` and `__device__` qualifiers for the same function
  - The compiler produces two versions of the function: one for the GPU and one for the CPU

```
__host__ __device__ float my_fmaxf(float a, float b)
{
    return (a>b ? a : b);
}
```

Recap function declaration

	Executed on:	Only callable from:
`__device__ float deviceFunc()`	Device	Device
`__host__ float hostFunc()`	Host	Host
`__global__ void kernelFunc()`	Device	Host

Reporting Errors

- All CUDA API calls return an error code (**`cudaError_t`**)
  - Error in the API call itself, OR
  - Error in an earlier asynchronous operation (e.g., kernel)
  - **`cudaSuccess`** means no error
- Get the error code for the last error:

```
cudaError_t cudaGetLastError(void)
```

- Get a string to describe the error:

```
char *cudaGetErrorString(cudaError_t)
printf("%s\n", cudaGetErrorString(cudaGetLastError()));
```

Reporting Errors

- Some useful macros defined in **`hpc.h`**
  - **`cudaSafeCall(Exp)`** execute Exp and checks for return status
    - Exp can be any CUDA function returning an error code, e.g., cudaMalloc, cudaMemcpy, …
  - **`cudaCheckError()`** checks error code from last CUDA operation
    - Usually, a kernel call
    - **`cudaCheckError()`** calls **`cudaDeviceSynchronize()`**

```
cudaSafeCall( cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice) );
my_kernel<<< 1, 1 >>>(d_a); cudaCheckError();
cudaSafeCall( cudaMemcpy(h_a, d_a, size, cudaMemcpyDeviceToHost) );
```

Device management

- On a multi-GPU installation, you can select the device to use with the **`CUDA_VISIBLE_DEVICES`** environment variable
- **`CUDA_VISIBLE_DEVICES=0 ./cuda-stencil1d`**
  - select the first device
- **`CUDA_VISIBLE_DEVICES=1 ./cuda-stencil1d`**
  - select the second device

Limits on the lab machine

- Use the **`deviceQuery`** command
- The lab server has the following features

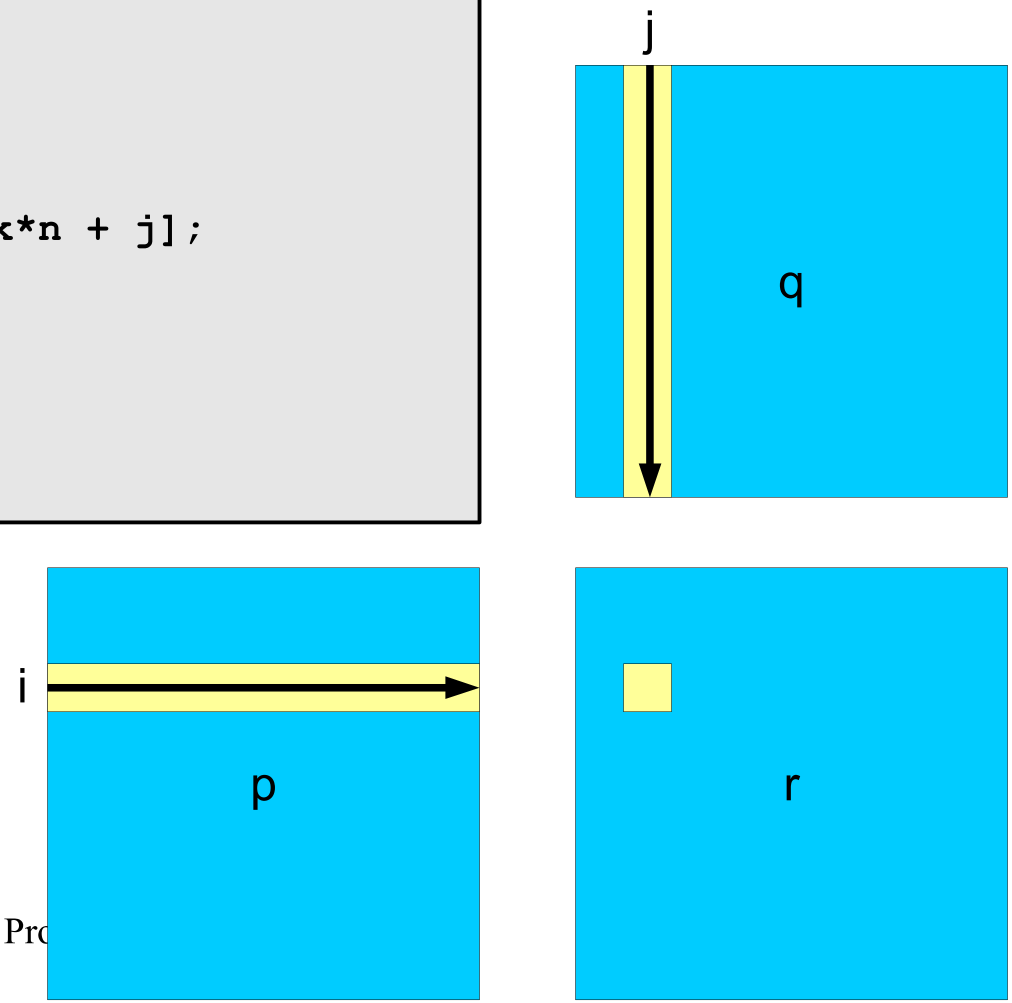
CUDA capability	2.0
Global memory	3005 MB
CUDA cores	512
Warp size	32
Shared memory per block	49152 B
Constant memory	65536 B
Max number of threads per block	1024
Max size of a thread block	(1024, 1024, 64)
Max grid size	(65535, 65535, 65535)

Higher Block Dimensions

Matrix-Matrix multiply

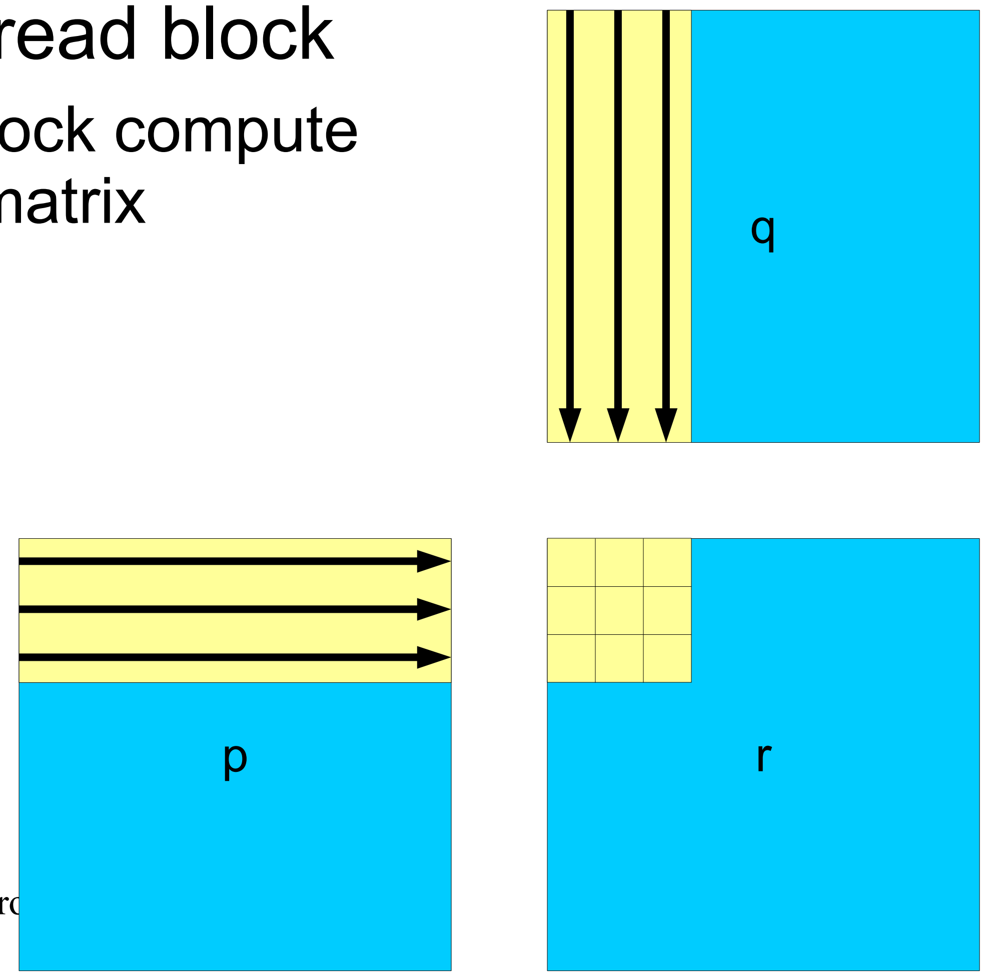
- Multiply two $n \times n$ matrices p and q

```
void matmul( float *p, float* q, float *r, int n)
{   int i, j, k; float v;
    for (i=0; i<n; i++) {
        for (j=0; j<n; j++) {
            v = 0.0;
            for (k=0; k<n; k++) {
                v += p[i*n + k] * q[k*n + j];
            }
            r[i*n + j] = v;
        }
    }
}
```



Matrix-Matrix multiply with thread blocks

- Decompose the result matrix *r* into square blocks
- Assign each block to a thread block
  - All threads of the thread block compute one element of the result matrix



Setting up the thread blocks

- The dim3 data type can be used to define a one-, two-, or three-dimensional “size” for a thread or grid block
  - **`dim3 blk(3)`**
    - defines a variable “dim” representing a 3x1x1 block
  - **`dim3 blk(3, 4)`**
    - defines a variable “dim” representing a 3x4x1 block
  - **`dim3 blk(3, 4, 7)`**
    - defines a variable “dim” representing a 3x4x7 block

Examples

- Launch 3 blocks, 16 threads per block (1D)

```
mykernel<<<3, 16>>>( );
```

- The same

```
dim3 grid(3);
dim3 block(16);
mykernel<<<grid, block>>>( );
```

- Launch (16 × 4) blocks, (8 × 8 × 8) threads per block

```
dim3 grid(16, 4);     /* 2D */
dim3 block(8, 8, 8); /* 3D */
mykernel<<<grid, block>>>( );
```

Kernel invocation

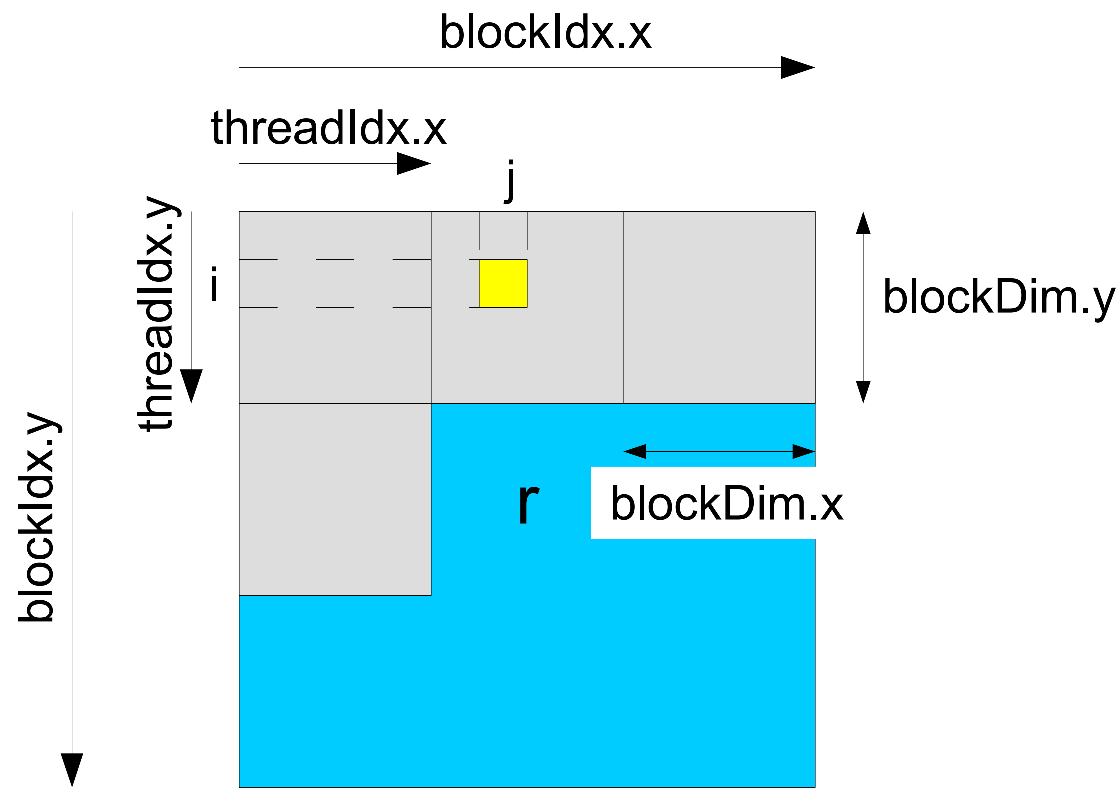
- Setup and kernel invocation

```
/* cuda-matmul.cu */
#define BLKDIM 32

int main( void )
{
    …
    dim3 block(BLKDIM, BLKDIM);
    dim3 grid((N+BLKDIM-1)/BLKDIM, (N+BLKDIM-1)/BLKDIM);
    …
    /* Launch matmul() kernel on GPU */
    matmul<<<grid, block>>>(d_p, d_q, d_r, N);
    …
}
```

The matmul kernel

- Each thread computes a single element *r*[*i*][*j*] of the result matrix *r*

```
const int i = blockIdx.y * blockDim.y + threadIdx.y;
const int j = blockIdx.x * blockDim.x + threadIdx.x;
```



The matmul kernel

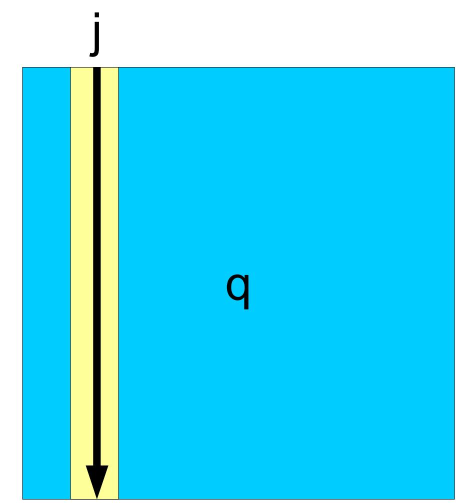
- The kernel function

```
__global__ void matmul( float *p, float *q, float *r, int n )
{
    const int i = blockIdx.y * blockDim.y + threadIdx.y;
    const int j = blockIdx.x * blockDim.x + threadIdx.x;
    int k;
    float val = 0.0;
    if ( i < n && j < n ) {
        for (k=0; k<n; k++) {
            val += p[i*n + k] * q[k*n + j];
        }
        r[i*n + j] = val;
    }
}
```

Matrix-Matrix multiply

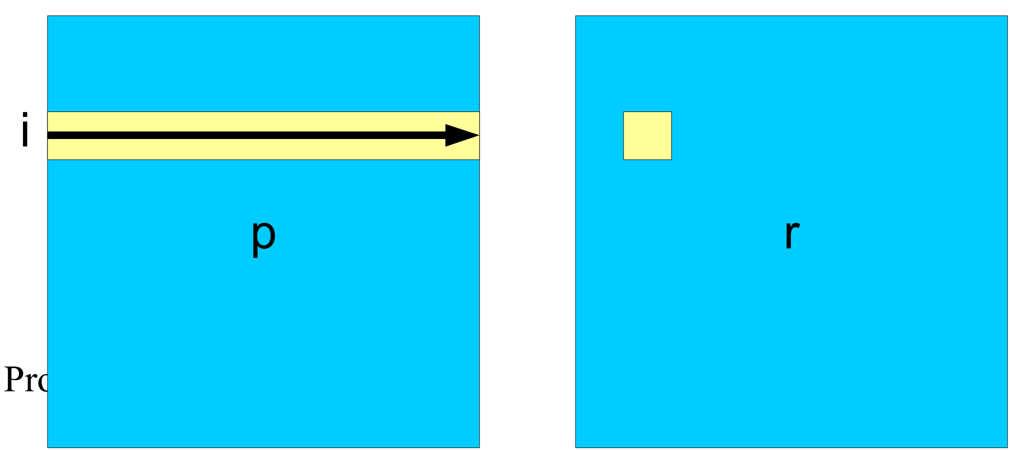
- Multiply two $n \times n$ matrices p and q

```
void matmul( float *p, float* q, float *r, int n)
{   int i, j, k; float v;
    for (i=0; i<n; i++) {
        for (j=0; j<n; j++) {
            v = 0.0;
            for (k=0; k<n; k++) {
                v += p[i*n + k] * q[k*n + j];
            }
            r[i*n + j] = v;
        }
    }
}
```



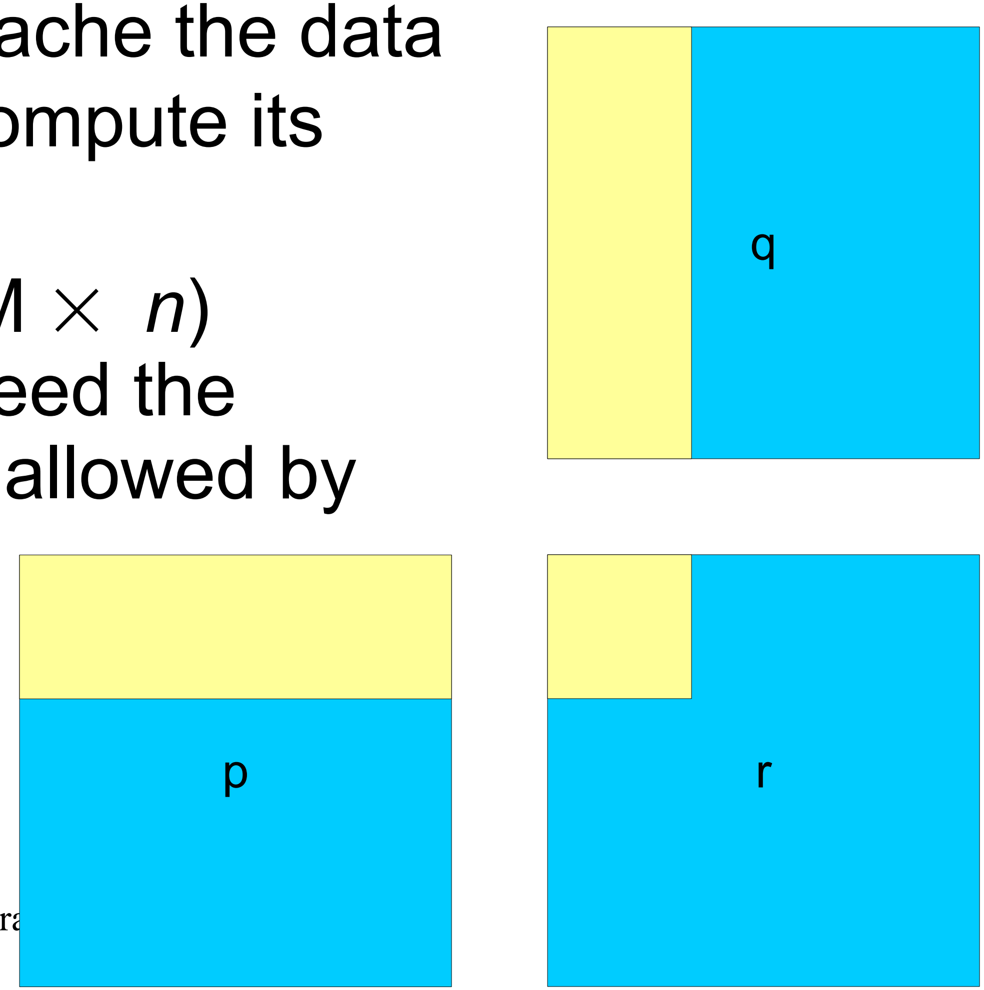
How many times are the elements of matrix *p* accessed?

How many times are the elements of matrix *q* accessed?



Reducing the memory pressure

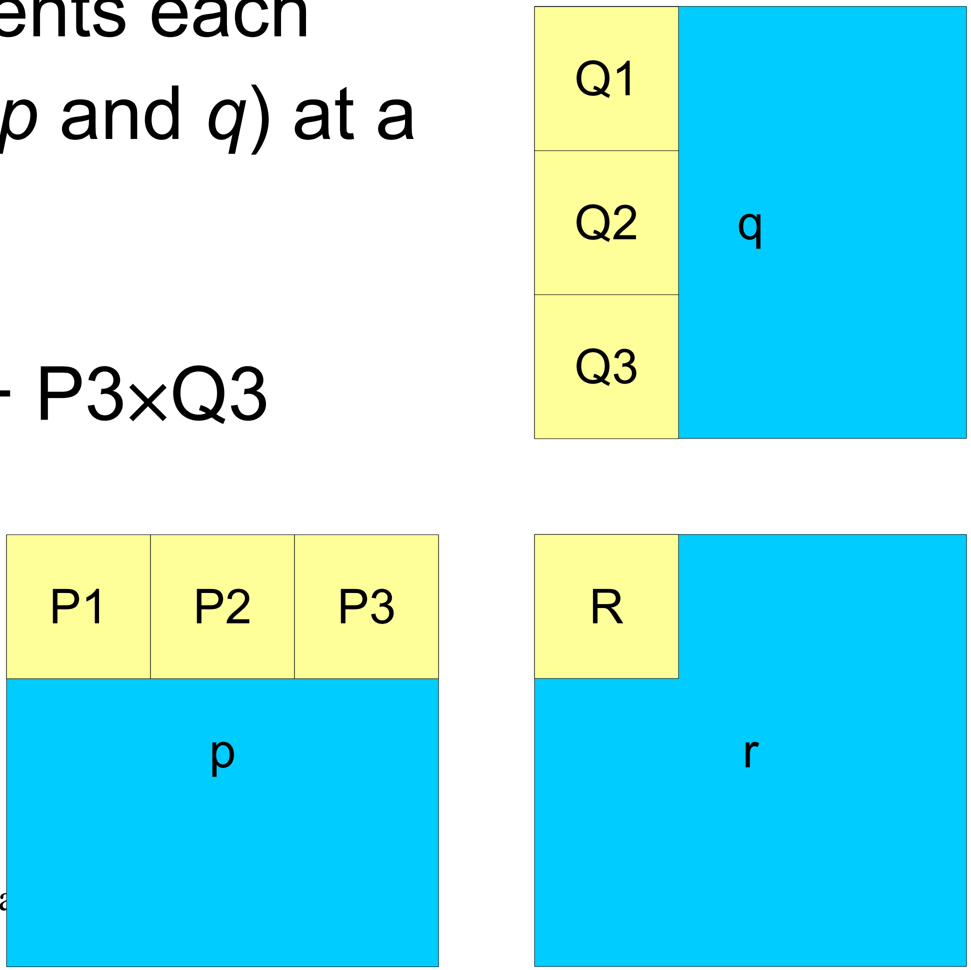
- To reduce the number of read operations to global memory, we use **`__shared__`** memory to cache the data needed by each block to compute its portion of the result
- This requires $(2 \times \text{BLKDIM} \times n)$ elements, which might exceed the amount of shared memory allowed by the device



Reducing the memory pressure

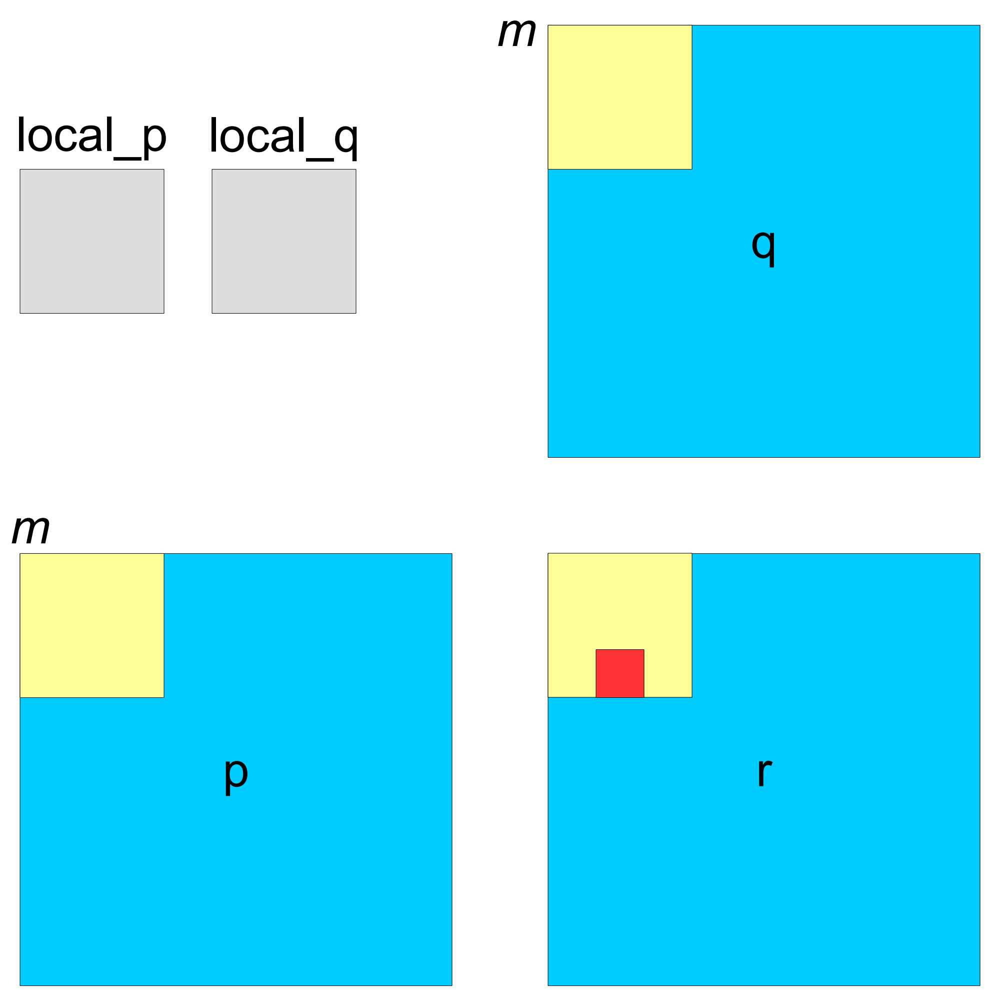
- The solution: divide the (BLKDIM $\times$ *n*) stripes of *p* and *q* into square blocks of (BLKDIM $\times$ *BLKDIM*) elements each
- Operate on two blocks (for *p* and *q*) at a time

$$R = P1 \times Q1 + P2 \times Q2 + P3 \times Q3$$



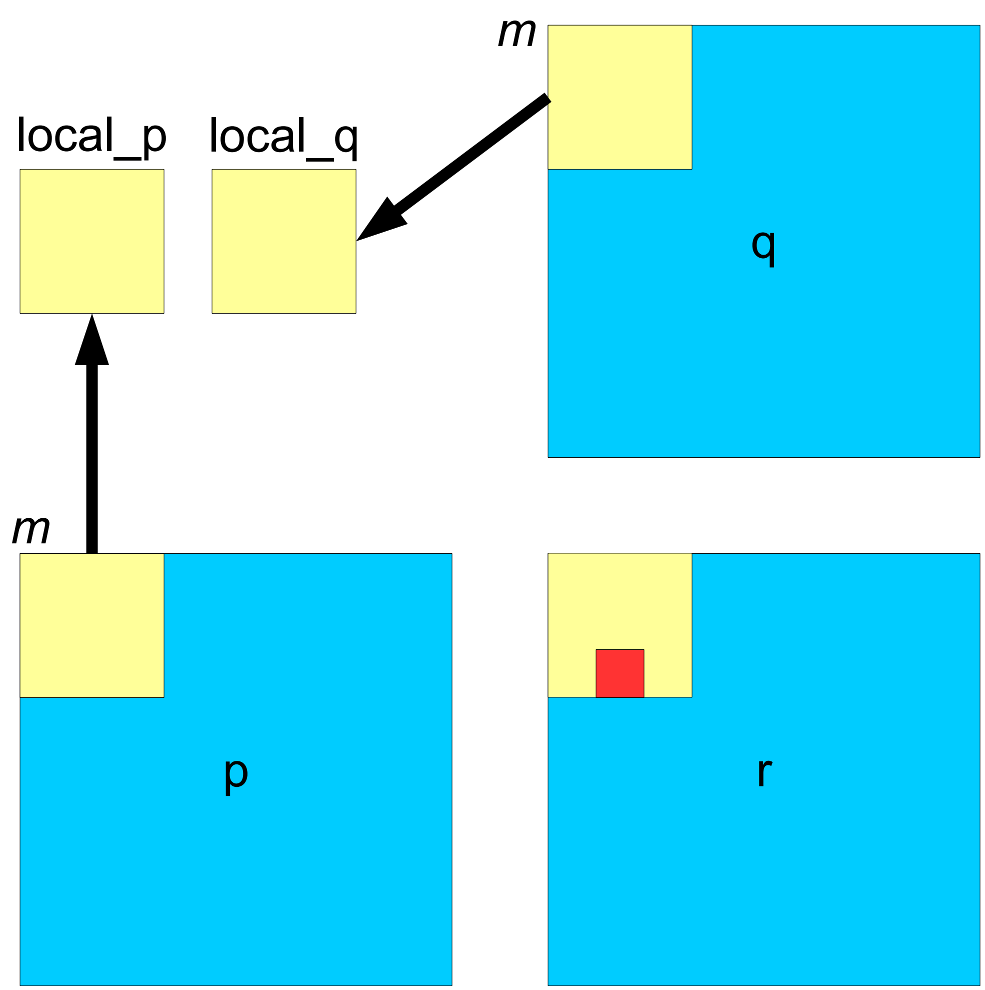
How the new kernel works

- For each *m*:



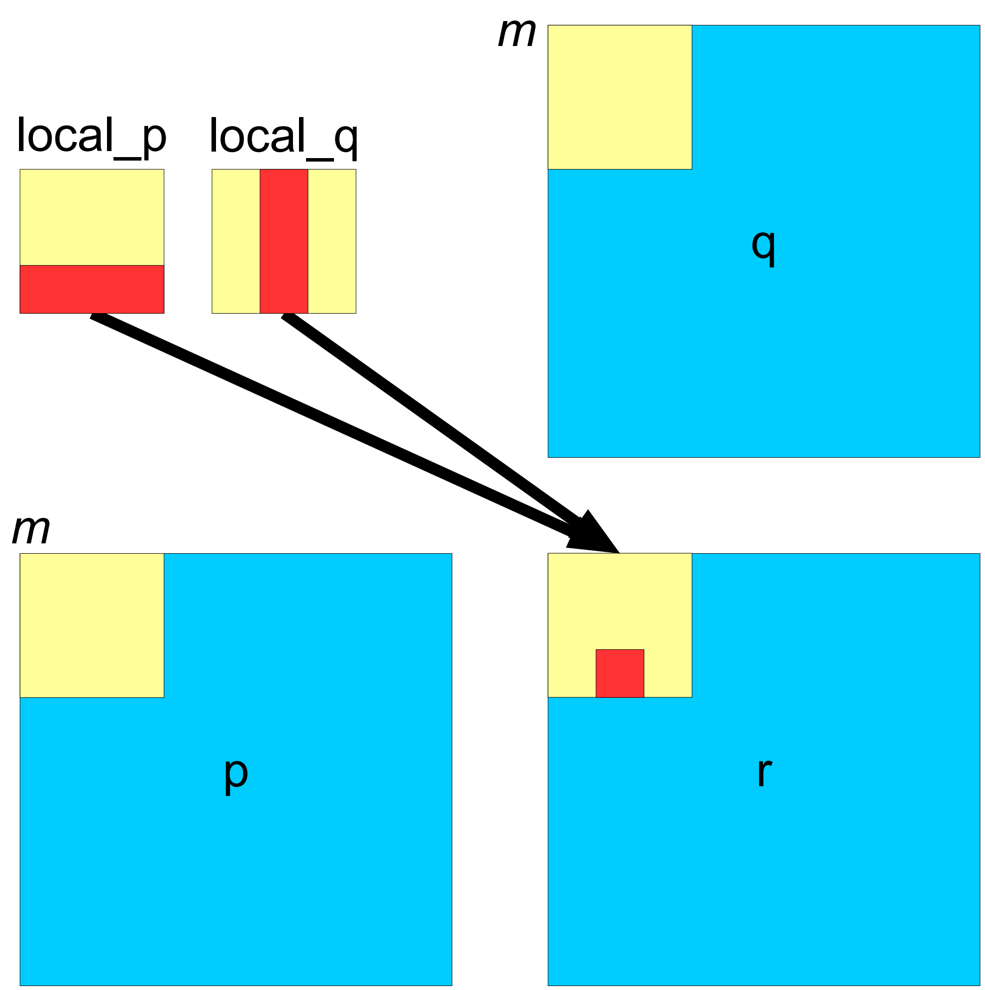
How the new kernel works

- For each *m*:
  - Copy blocks from *p* and *q* into shared memory *local_p* and *local_q*



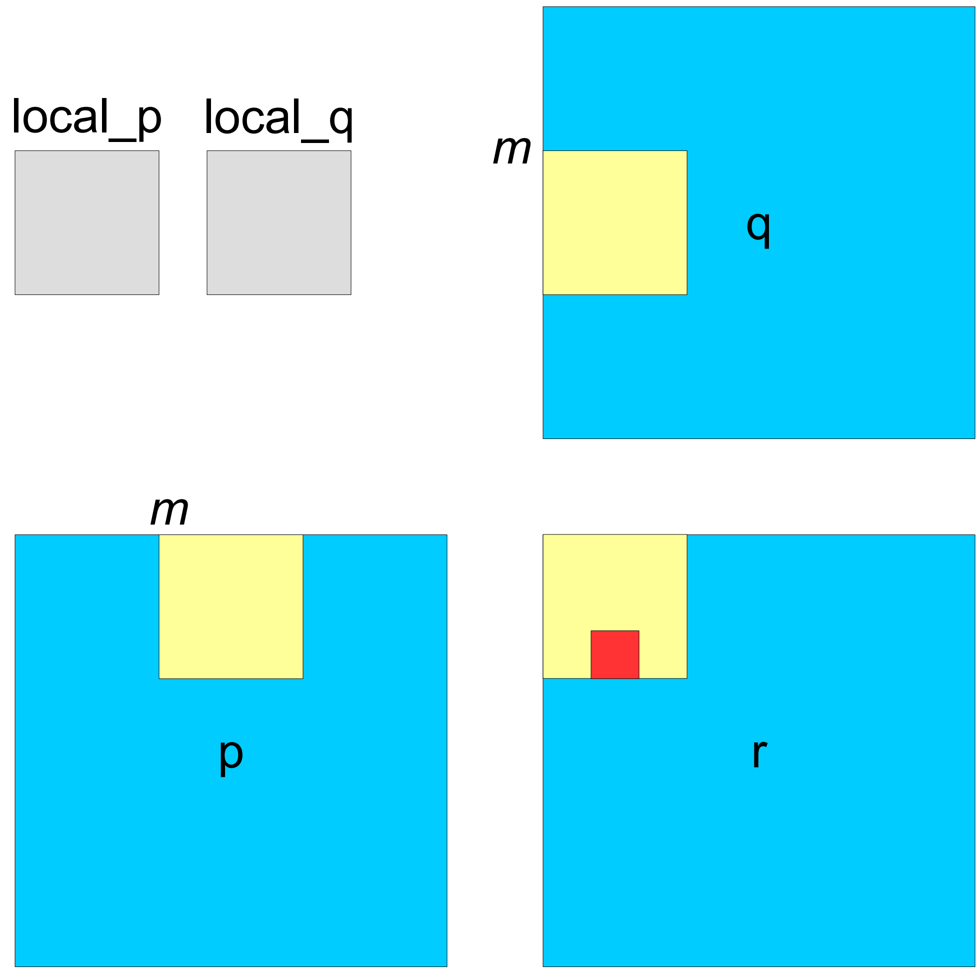
How the new kernel works

- For each *m*:
  - Copy blocks from *p* and *q* into shared memory *local_p* and *local_q* in parallel
  - compute the matrix product *local_p* × *local_q* in parallel



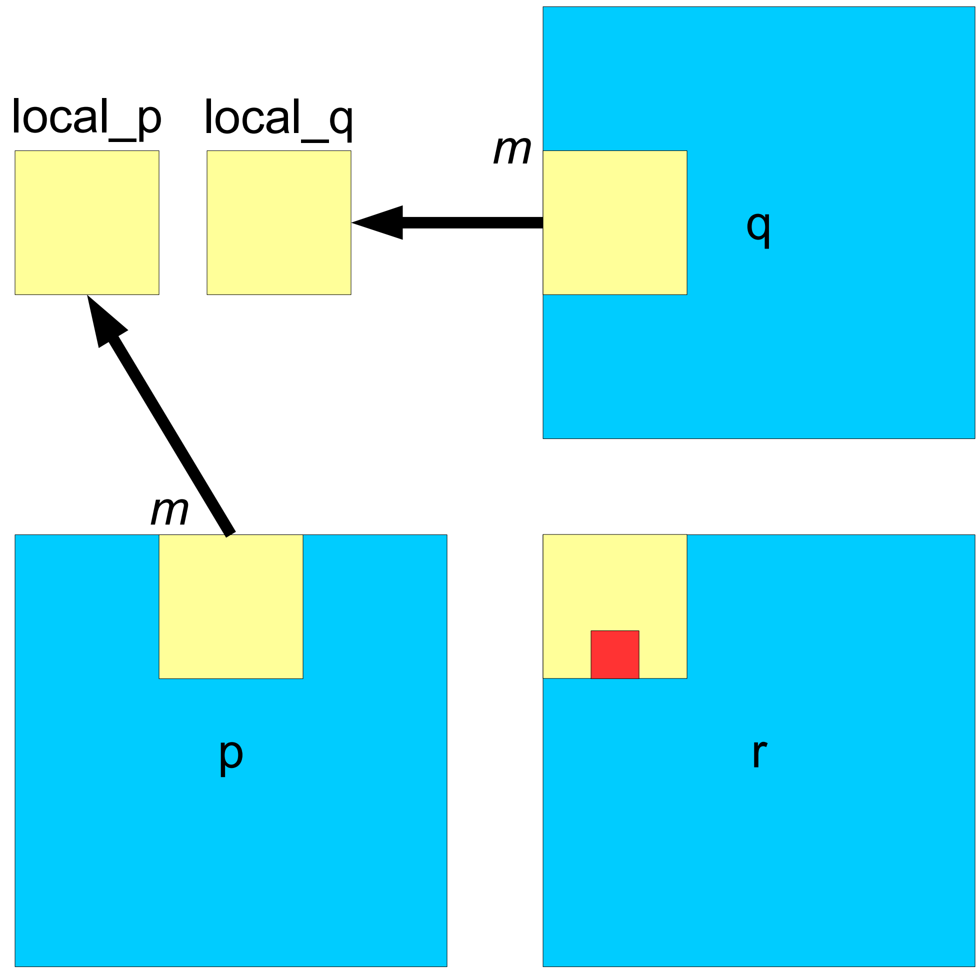
How the new kernel works

- For each *m*:
  - Copy blocks from *p* and *q* into shared memory *local_p* and *local_q* in parallel
  - compute the matrix product *local_p* × *local_q* in parallel
  - $m \leftarrow m$ + BLKDIM



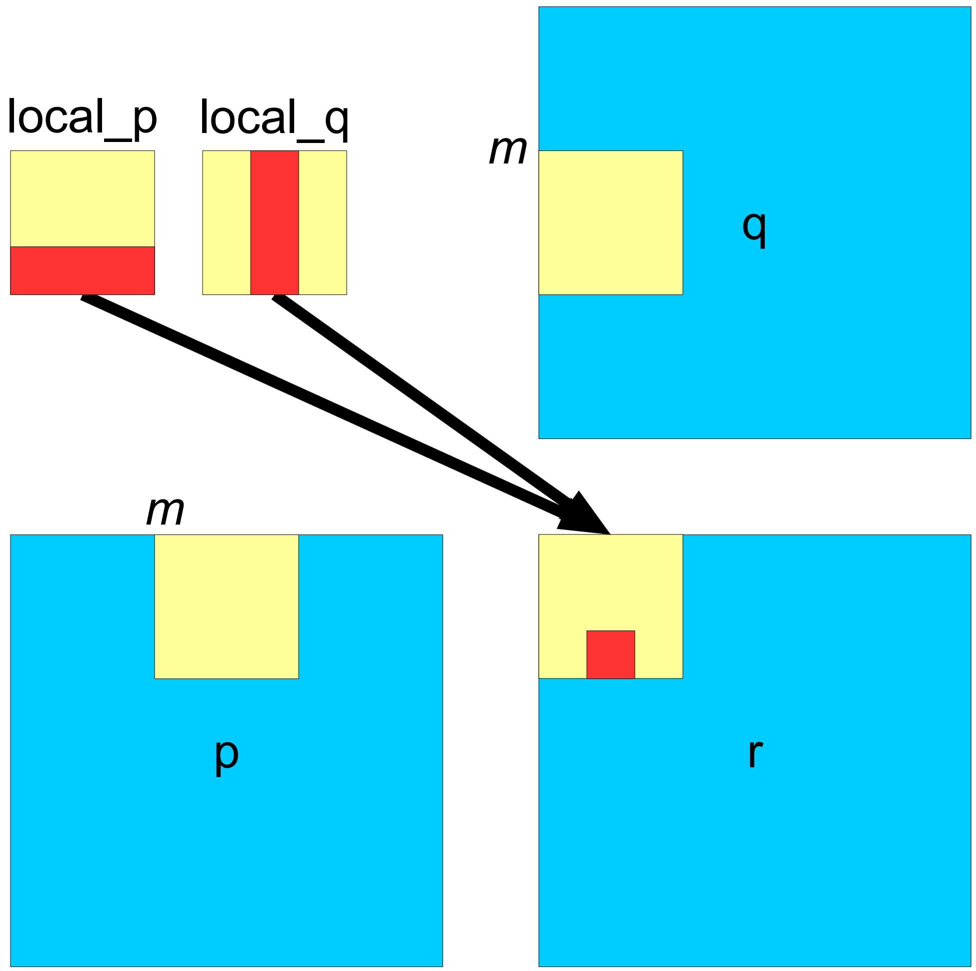
How the new kernel works

- For each *m*:
  - Copy blocks from *p* and *q* into shared memory *local_p* and *local_q* in parallel
  - compute the matrix product *local_p* × *local_q* in parallel
  - $m \leftarrow m$ + BLKDIM



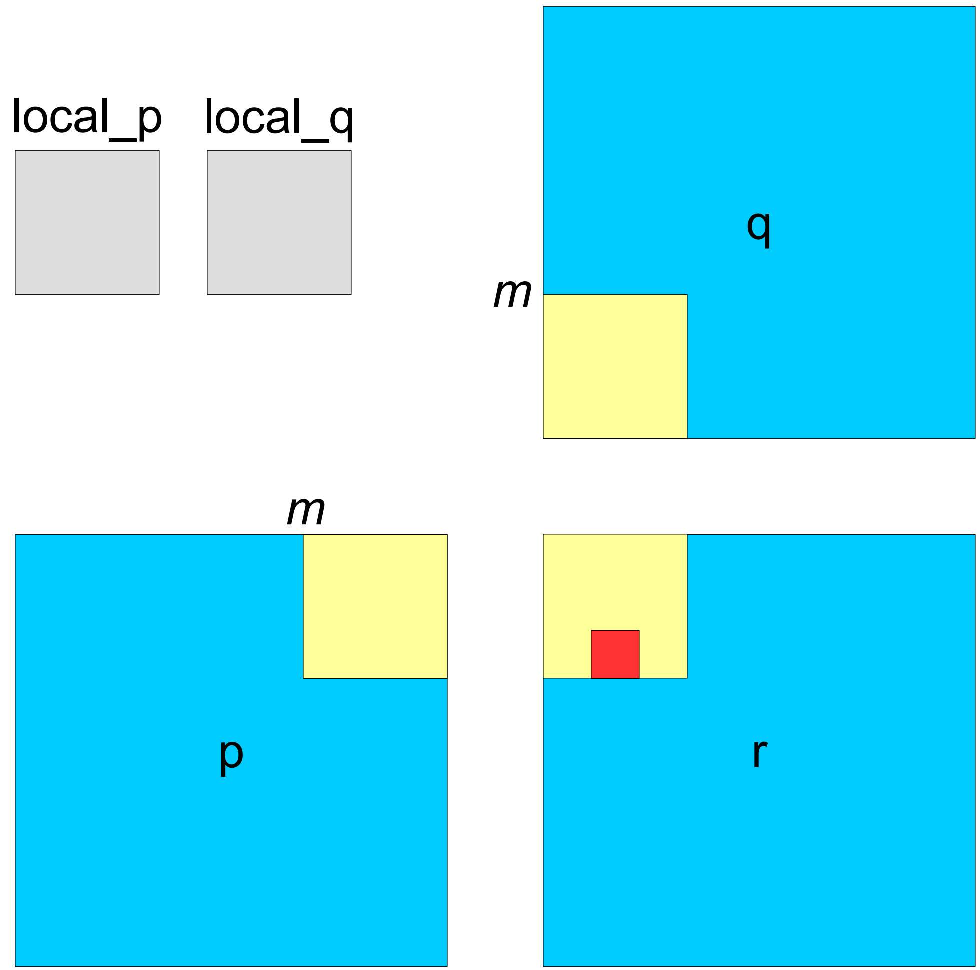
How the new kernel works

- For each *m*:
  - Copy blocks from *p* and *q* into shared memory *local_p* and *local_q* in parallel
  - compute the matrix product *local_p* × *local_q* in parallel
  - $m \leftarrow m$ + BLKDIM



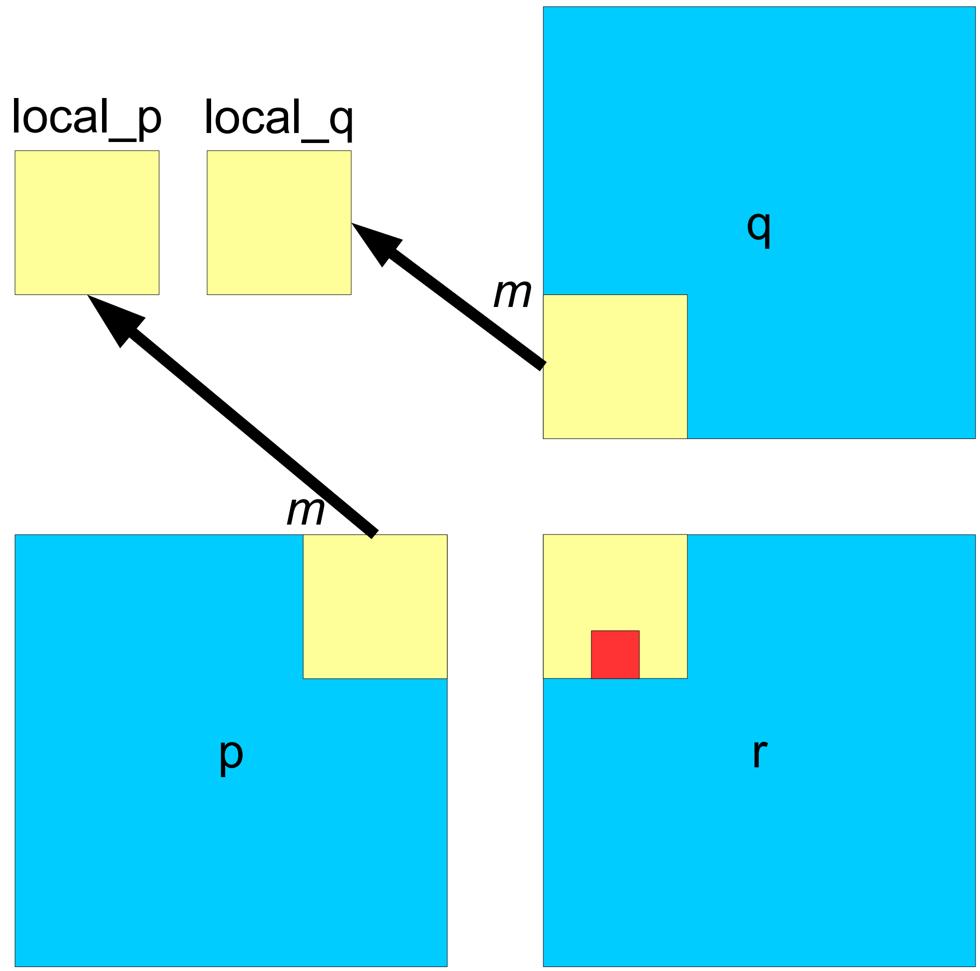
How the new kernel works

- For each *m*:
  - Copy blocks from *p* and *q* into shared memory *local_p* and *local_q* in parallel
  - compute the matrix product *local_p* $\times$ *local_q* in parallel
  - $m \leftarrow m$ + BLKDIM



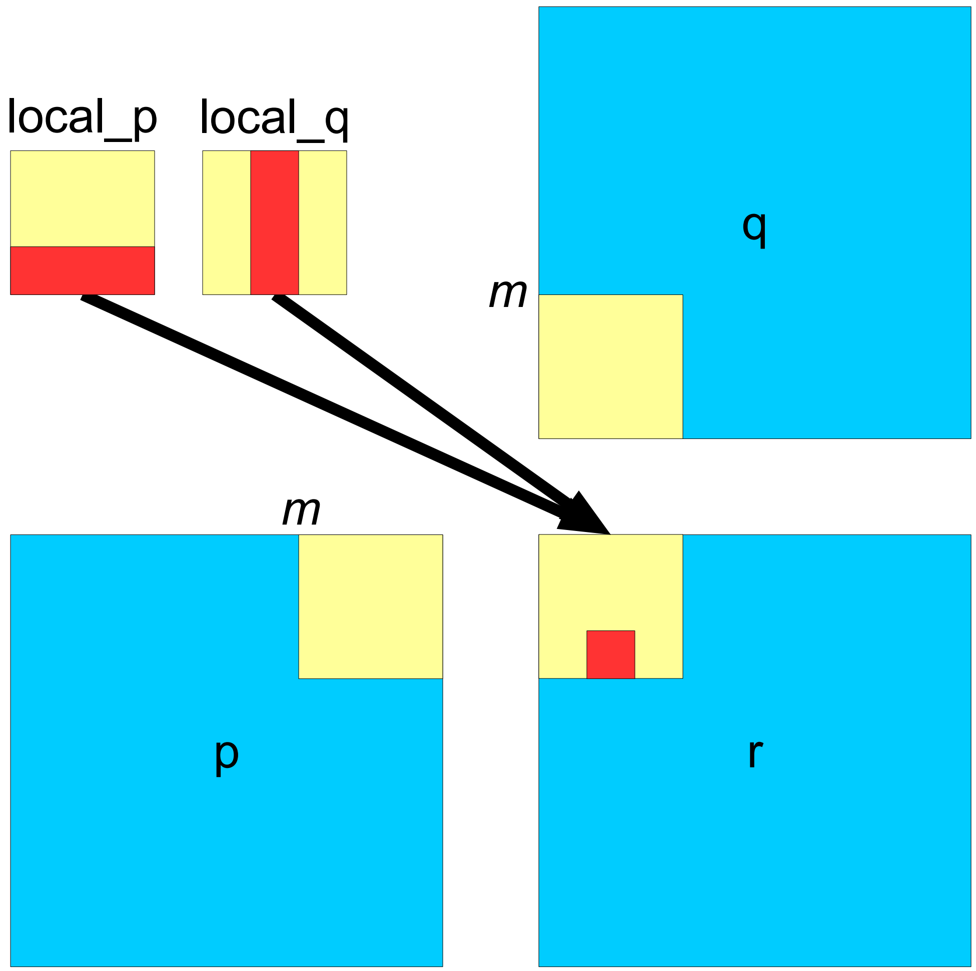
How the new kernel works

- For each *m*:
  - Copy blocks from *p* and *q* into shared memory *local_p* and *local_q* in parallel
  - compute the matrix product *local_p* × *local_q* in parallel
  - $m \leftarrow m$ + BLKDIM



How the new kernel works

- For each *m*:
  - Copy blocks from *p* and *q* into shared memory *local_p* and *local_q* in parallel
  - compute the matrix product *local_p* × *local_q* in parallel
  - $m \leftarrow m$ + BLKDIM



The new matmul kernel

```
__global__ void matmulb( float *p, float *q, float *r, int n )
{
    __shared__ float local_p[BLKDIM][BLKDIM];
    __shared__ float local_q[BLKDIM][BLKDIM];
    const int bx = blockIdx.x; const int by = blockIdx.y;
    const int tx = threadIdx.x; const int ty = threadIdx.y;
    const int i = by * BLKDIM + ty;
    const int j = bx * BLKDIM + tx;
    float v = 0.0; int m, k;
    for (m = 0; m < n; m += BLKDIM) {
        local_p[ty][tx] = p[i*n + (m + tx)];
        local_q[ty][tx] = q[(m + ty)*n + j];
        __syncthreads();
        for (k = 0; k < BLKDIM; k++) {
            v += local_p[ty][k] * local_q[k][tx];
        }
        __syncthreads();
    }
    r[i*n + j] = v;
}
```

Wait for all threads in the block to complete before starting the next iteration (overwriting local_p and local_q)

Credits: Salvatore Orlando

The new matmul kernel

- The setup and kernel invocation remain the same
- See cuda-matmul.cu

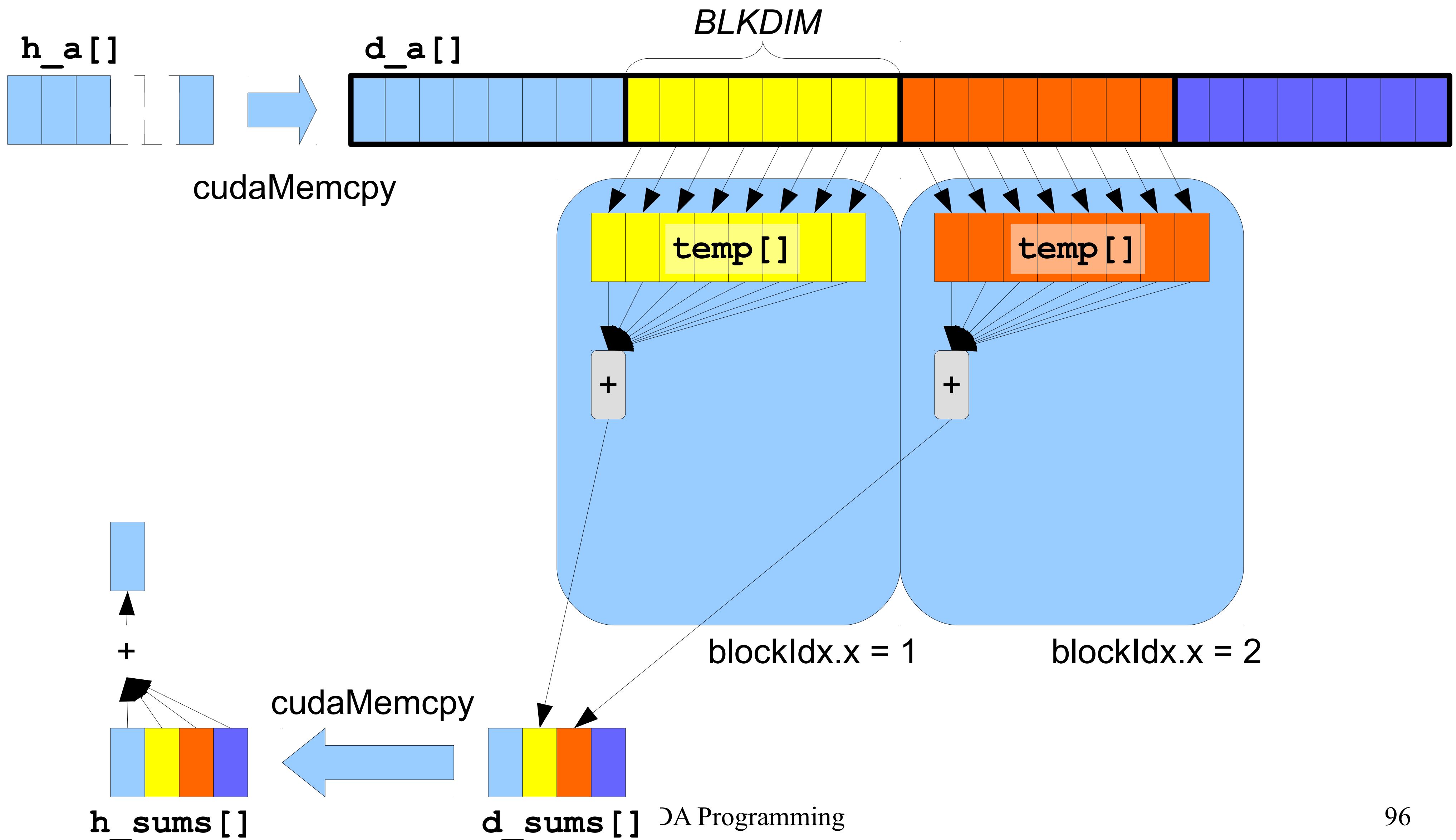
Reduction on the GPU

Problem statement

- Given a nonempty array of floats or ints, compute the sum of all elements of the array
- Basic idea
  - Decompose the problem across thread blocks
  - Each thread block computes a partial reduction
  - The CPU completes the reduction

Solution #0 (naive)

- Thread 0 of each block computes the local sum



Solution #0 kernel

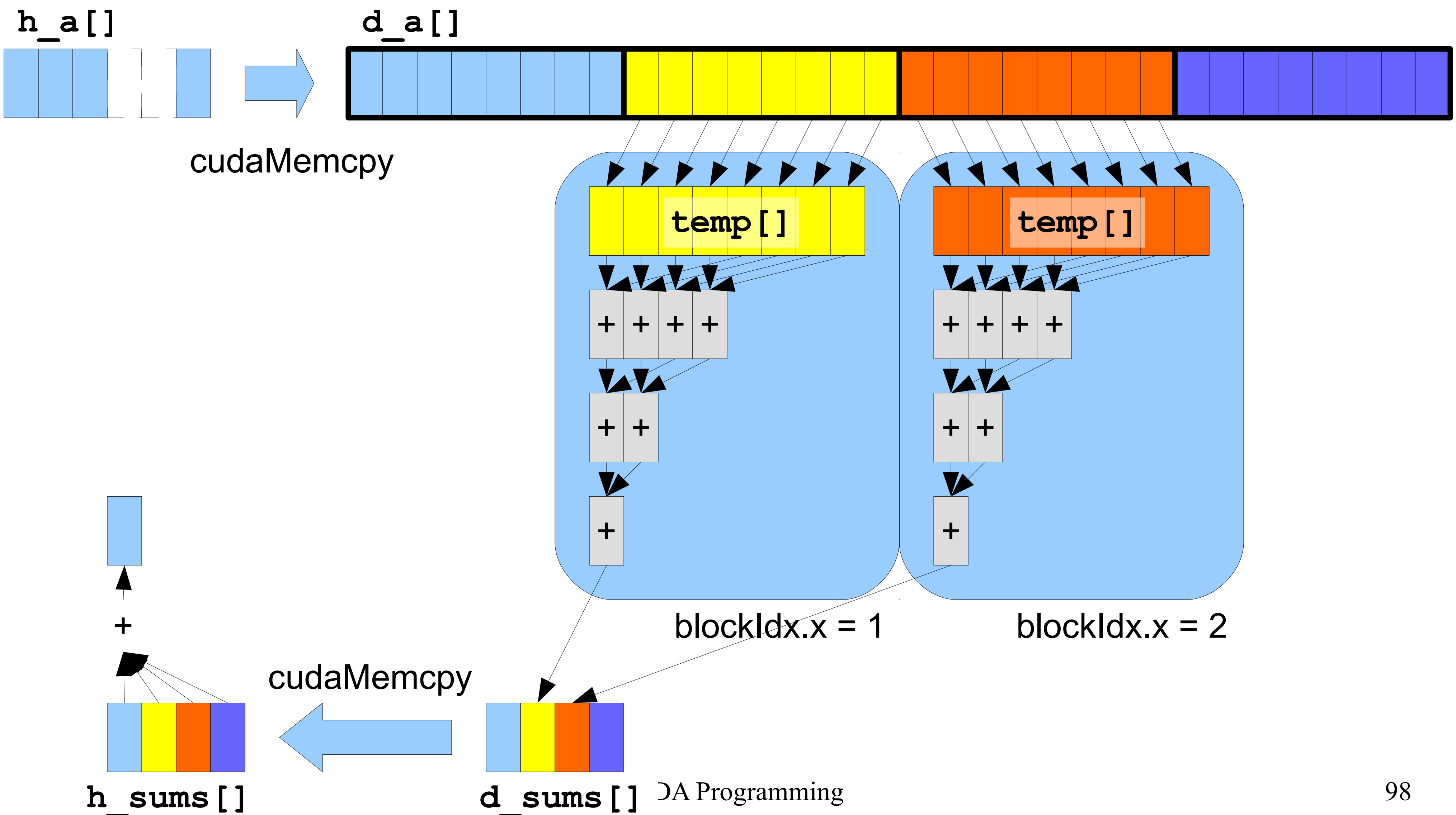
```
#define BLKDIM 512
#define N_OF_BLOCKS 1024
#define N ((N_OF_BLOCKS)*(BLKDIM))

__device__ int d_sums[N_OF_BLOCKS];
int h_sums[N_OF_BLOCKS];

__global__ void sum( int *a, int n )
{
    __shared__ int temp[BLKDIM];
    int lindex = threadIdx.x;
    int bindex = blockIdx.x;
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    temp[lindex] = a[gindex];
    __syncthreads();
    if ( 0 == lindex ) {
        int i, my_sum = 0;
        for (i=0; i<blockDim.x; i++) {
            my_sum += temp[i];
        }
        d_sums[bindex] = my_sum;
    }
}
```

Shared memory is not useful here; it will be useful in the other versions of this kernel

See cuda-reduction0.cu

Solution #1 (better)

- All threads within each block cooperate to compute the local sum



Solution #1 kernel

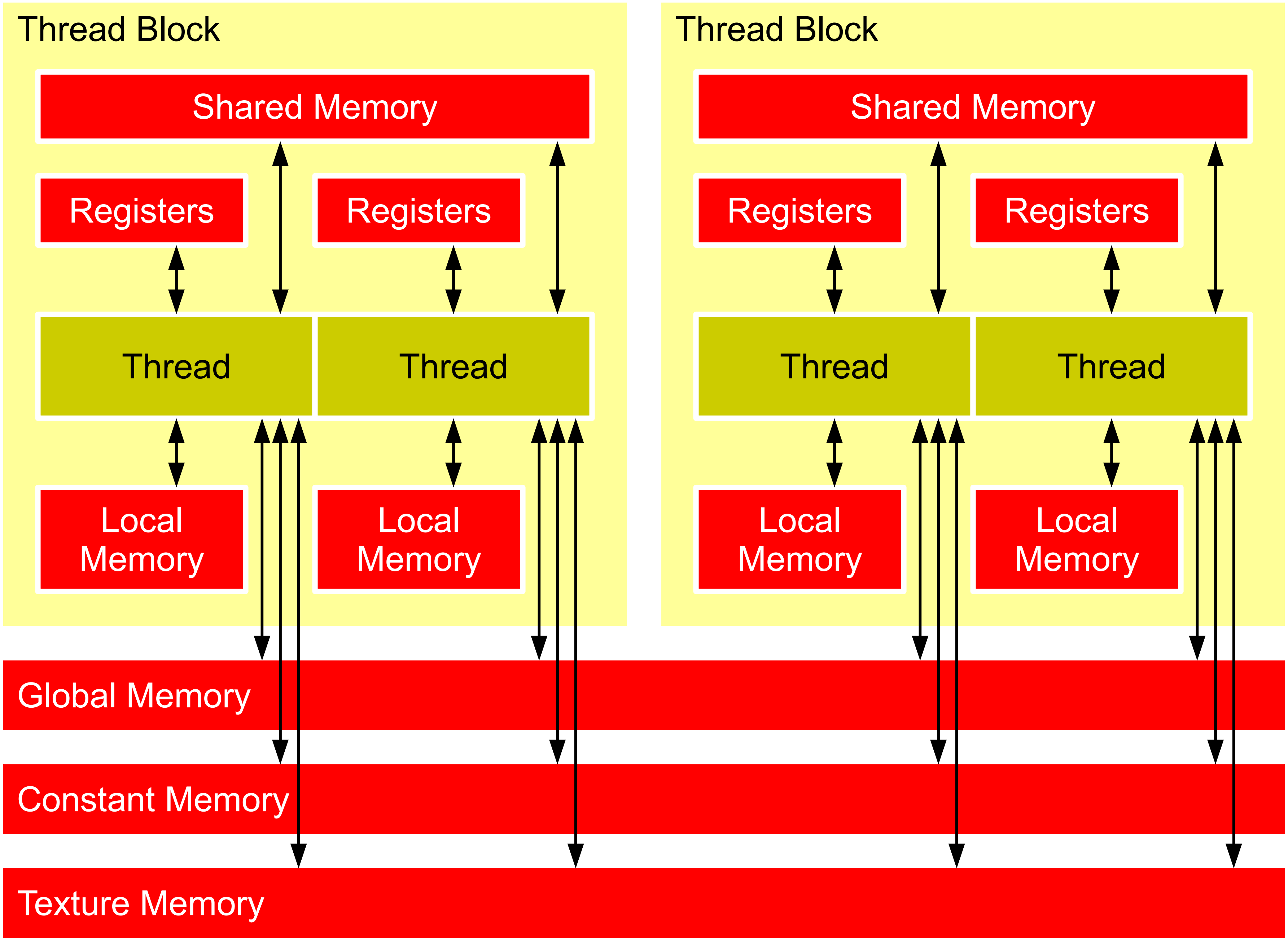
This kernel only works if
- BLKDIM is a power of two;
- *n* is a multiple of BLKDIM

```
__global__ void sum( int *a, int n )
{
    __shared__ int temp[BLKDIM];
    const int lindex = threadIdx.x;
    const int bindex = blockIdx.x;
    const int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int bsize = blockDim.x / 2;
    temp[lindex] = a[gindex];
    __syncthreads();
    while ( bsize > 0 ) {
        if ( lindex < bsize ) {
            temp[lindex] += temp[lindex + bsize];
        }
        bsize = bsize / 2;
        __syncthreads();
    }
    if ( 0 == lindex ) {
        d_sums[bindex] = temp[0];
    }
}
```

See cuda-reduction1.cu

Memory Access Optimization Techniques for GPUs

CUDA memory model

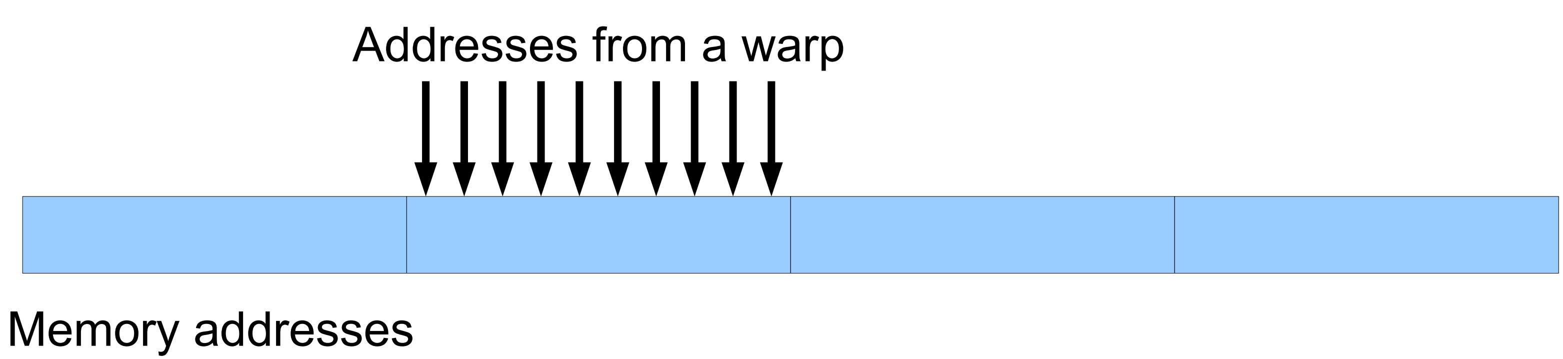


Memory access patterns

- Each memory access moves 32 or 128 consecutive bytes
  - So, if a thread just needs a single float (4B), this results in 32B or 128B being moved
- The GPU can pack together (coalesce) memory accesses when consecutive threads access consecutive memory addresses
  - Examples follow

Caching load

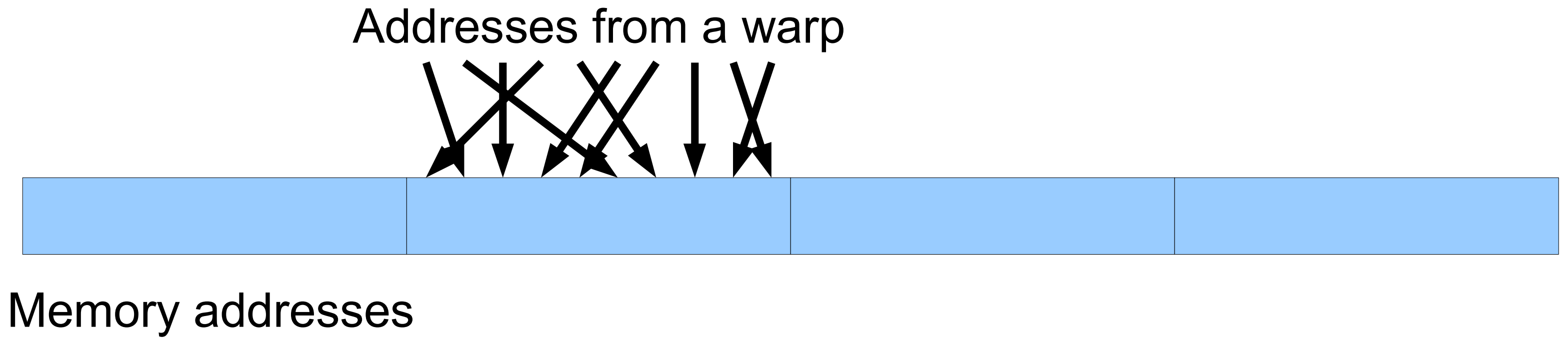
- Warp requests 32 aligned, consecutive 4-byte words
- Addresses fall within 1 cache-line
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%
  - Transactions: 1



Slide credits: Justin Luitjens, NVIDIA Corporation

Caching load

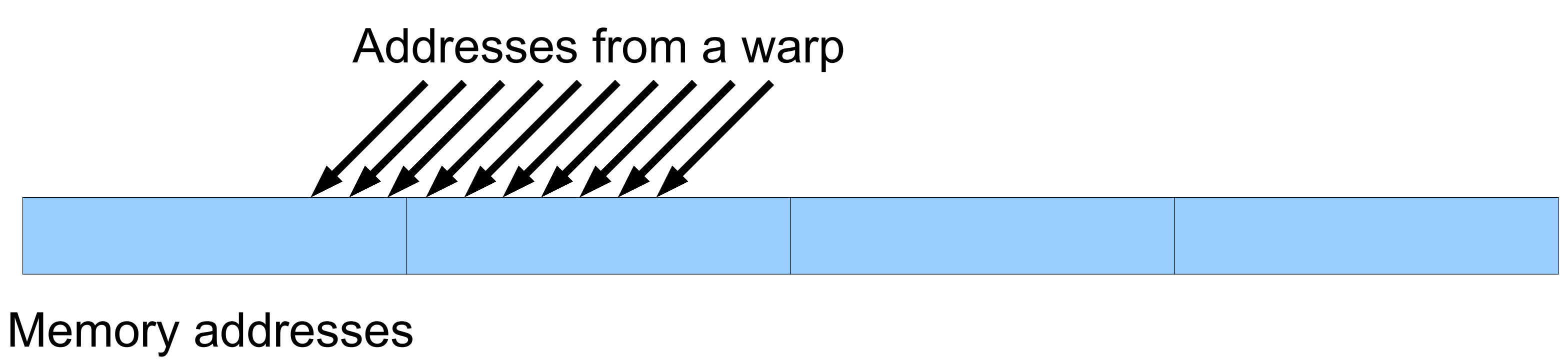
- Warp requests 32 aligned, permuted 4-byte words
- Addresses fall within 1 cache-line
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%
  - Transactions: 1



Slide credits: Justin Luitjens, NVIDIA Corporation

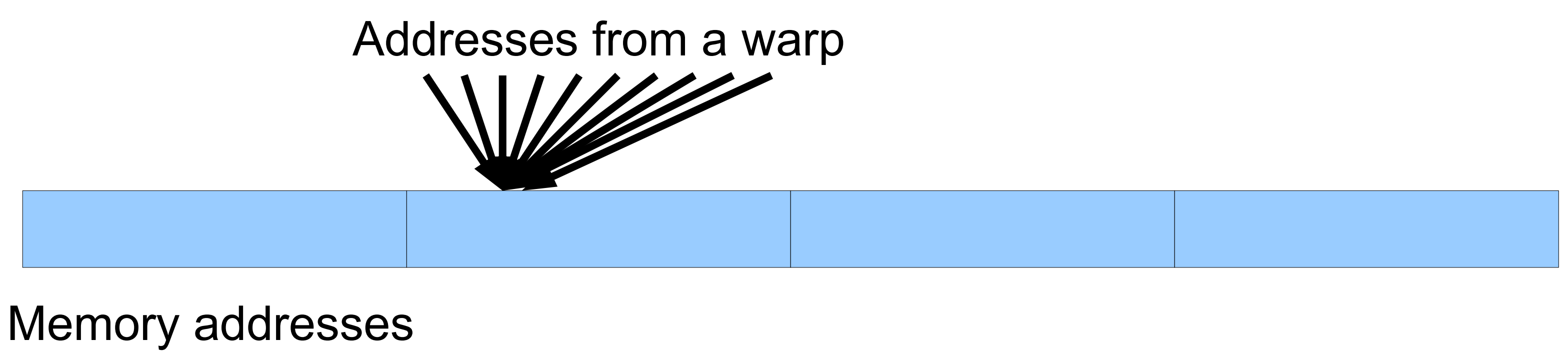
Caching load

- Warp requests 32 misaligned, consecutive 4-byte words
- Addresses fall within 2 cache-lines
  - Warp needs 128 bytes
  - 256 bytes move across the bus on misses
  - Bus utilization: 50%
  - Transactions: 2



Caching load

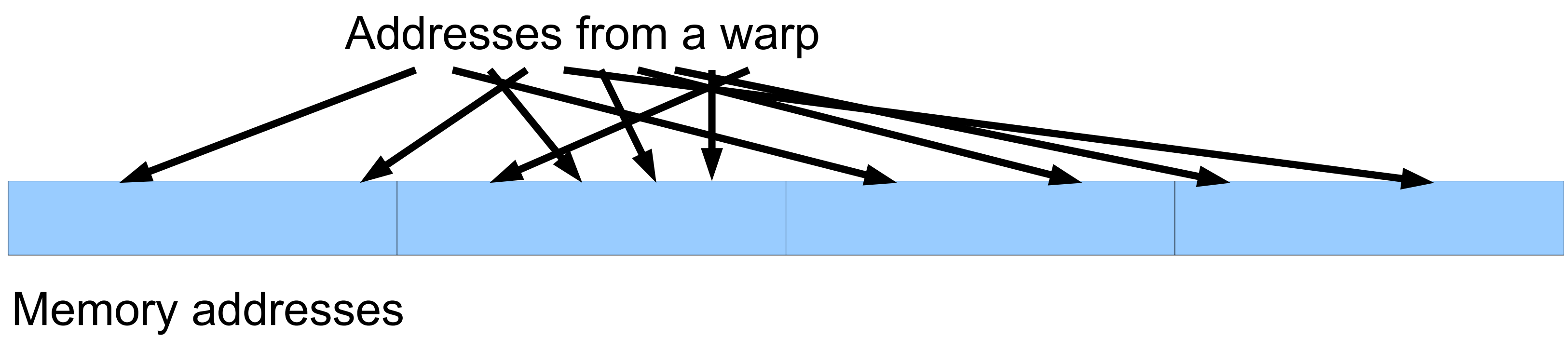
- All threads in a warp request the same 4-byte word
- Addresses fall within a single cache-line
  - Warp needs 4 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 3.125%



Slide credits: Justin Luitjens, NVIDIA Corporation

Caching load

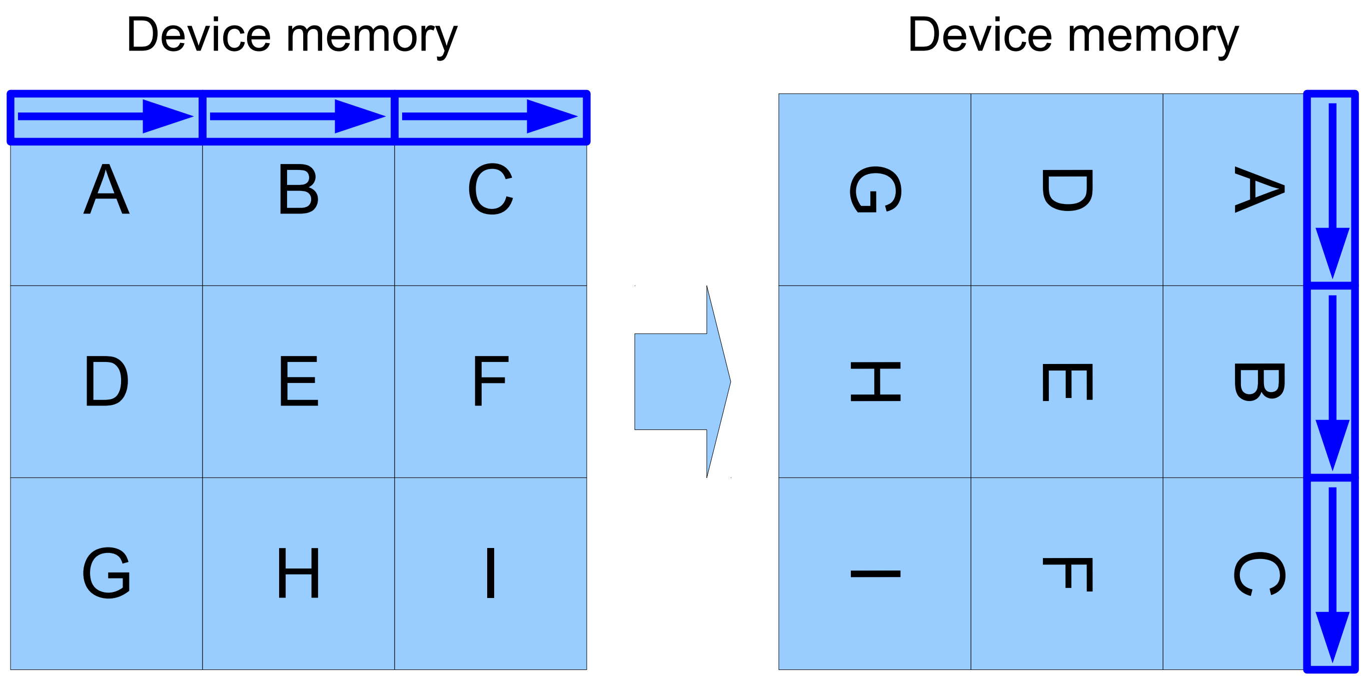
- Warp requests 32 scattered 4-byte words
- Addresses fall within *N* cache-lines
  - Warp needs 128 bytes
  - $N \times 128$ bytes move across the bus on a miss
  - Bus utilization: $128 / (N \times 128)$



Slide credits: Justin Luitjens, NVIDIA Corporation

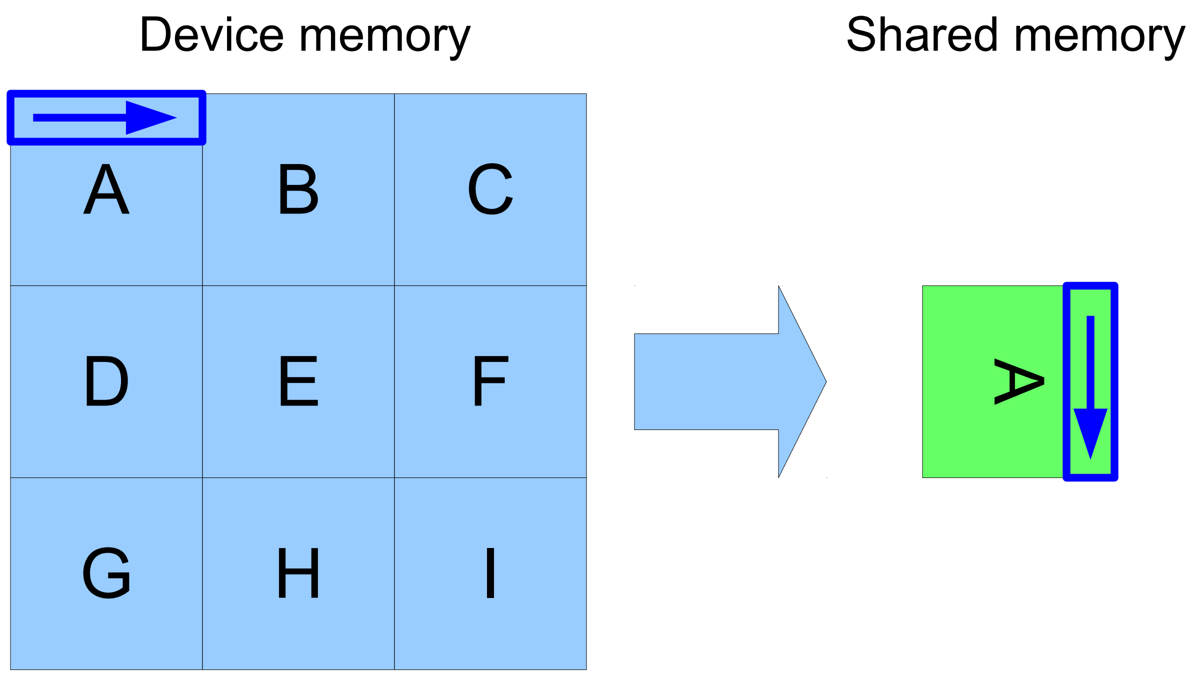
Example

- See cuda-image-rotation.cu



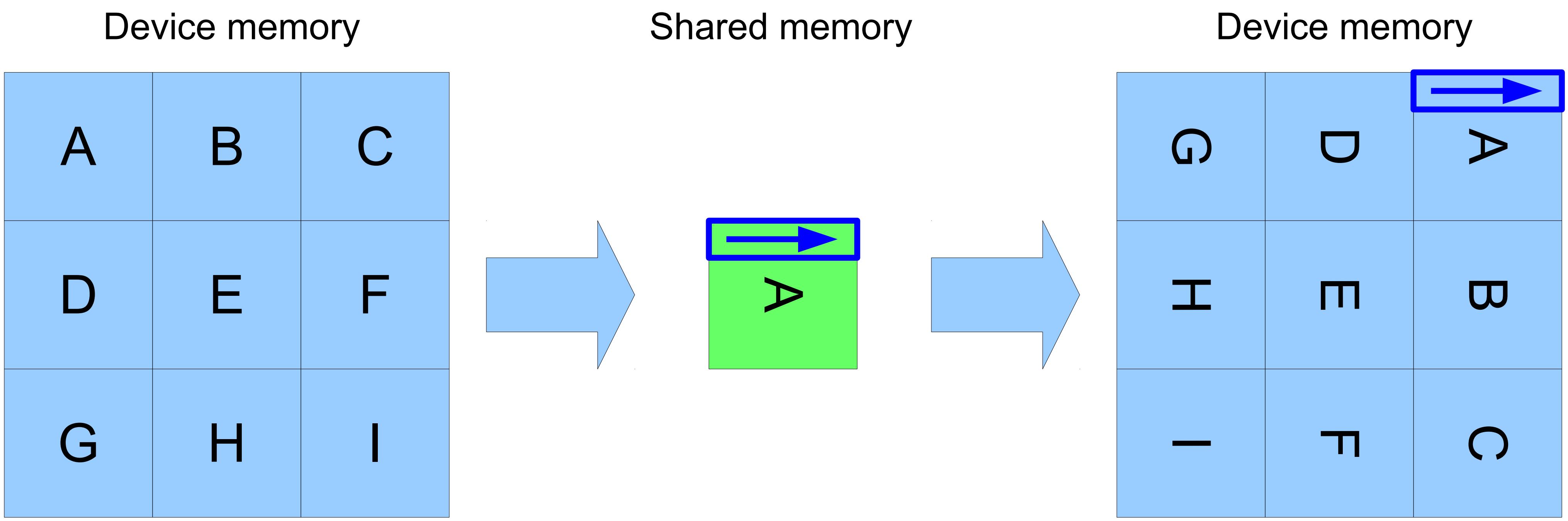
Example

- See cuda-image-rotation.cu



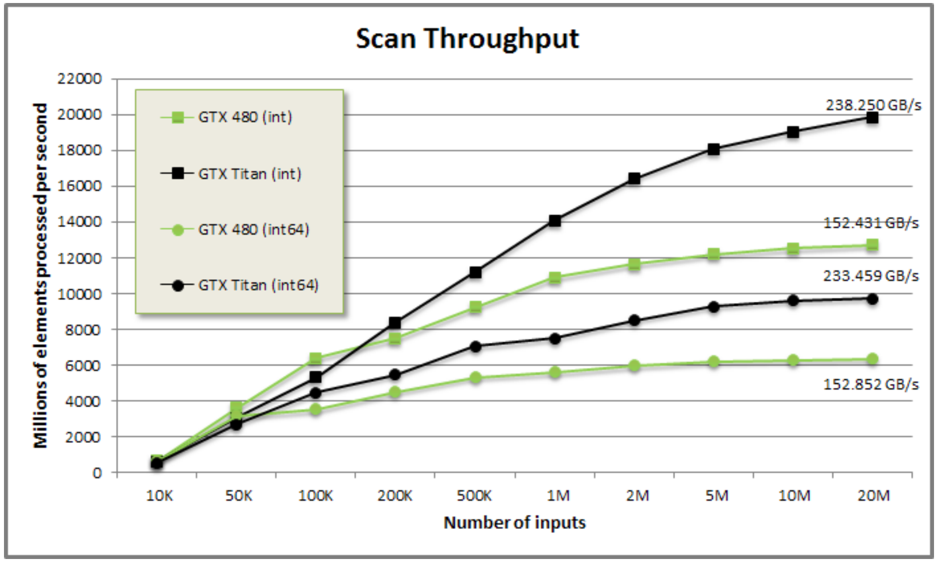
Example

- See cuda-image-rotation.cu



Performance Evaluation

- We conventional concept of speedup can not be used
  - The program has little or no control over the number of CUDA cores used
  - The hardware multiplexes CUDA threads to CUDA cores
- We need different metrics, e.g.
  - Throughput: number of processed data items/seconds as a function of the input size
  - Speedup vs CPU implementation



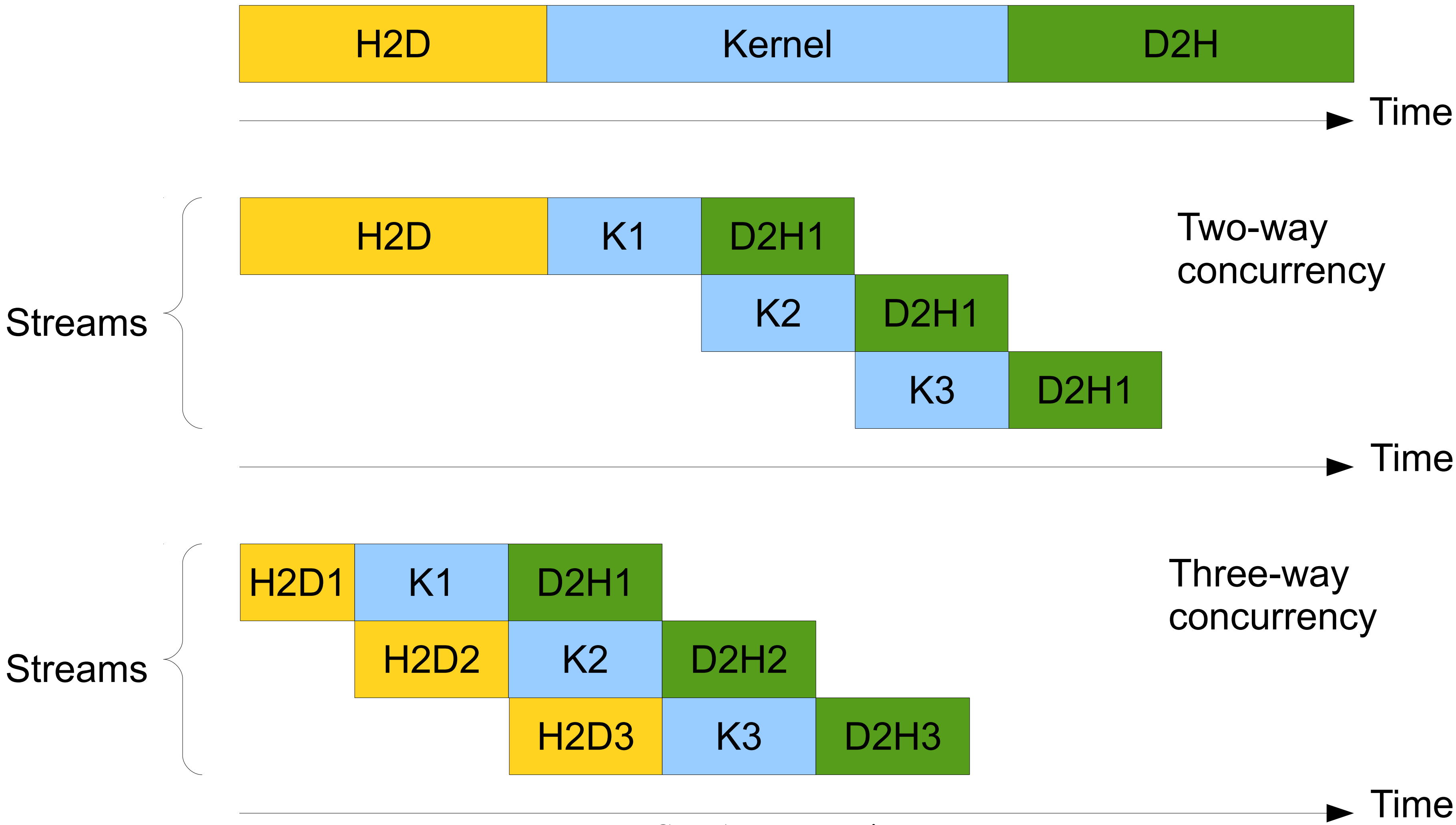
Source: https://moderngpu.github.io/scan.html

CUDA Streams

- CUDA Streams are work queues to express concurrency between different tasks, e.g.
  - host to device memory copies
  - device to host memory copies
  - kernel execution
- To overlap different tasks just launch them in different streams
  - All tasks launched into the same stream are executed in order
  - Tasks launched into different streams might execute concurrently (depending on available resources)

Slide credits:Jochen Kreutz, FZJ

Concurrency Example



Slide credits:Steve Rennich, NVIDIA corporation

Default Stream (stream #0)

- Stream used when no stream is specified
- Completely synchronous w.r.t. host and device
  - As if cudaDeviceSynchronize() inserted before and after every CUDA operation
- Exceptions – asynchronous w.r.t. host
  - Kernel launches in the default stream
  - cudaMemcpy*Async
  - cudaMemset*Async
  - cudaMemcpy within the same device
  - H2D cudaMemcpy of 64kB or less

Slide credits:Steve Rennich, NVIDIA corporation

Requirements for concurrency

- CUDA operations must be in different, non-0, streams
- cudaMemcpyAsync with host from 'pinned' memory
  - Page-locked memory
  - Allocated using cudaMallocHost() or cudaHostAlloc()
- Sufficient resources must be available
  - cudaMemcpyAsyncs in different directions
  - Device resources (SMEM, registers, blocks, etc.)

Slide credits:Steve Rennich, NVIDIA corporation

Example

```
cudaStream_t stream1, stream2, stream3, stream4 ;
cudaStreamCreate ( &stream1) ;
...
cudaMalloc ( &dev1, size ) ;
cudaMallocHost ( &host1, size ) ;
...
cudaMemcpyAsync ( dev1, host1, size, H2D, stream1 ) ;
kernel2 <<< grid, block, 0, stream2 >>> ( ..., dev2, ... ) ;
kernel3 <<< grid, block, 0, stream3 >>> ( ..., dev3, ... ) ;
cudaMemcpyAsync ( host4, dev4, size, D2H, stream4 ) ;
some_CPU_method ();
...
cudaStreamDestroy( &stream1 );
...
```

Allocate host pinned memory

Potentially overlapped

Slide credits:Steve Rennich, NVIDIA corporation

Synchronization

- Synchronize everything
  - **`cudaDeviceSynchronize()`**
  - Blocks host until all issued CUDA calls are complete
- Synchronize w.r.t. a specific stream
  - **`cudaStreamSynchronize( streamid )`**
  - Blocks host until all CUDA calls in streamid are complete

Slide credits:Steve Rennich, NVIDIA corporation