

CUDA Exercises

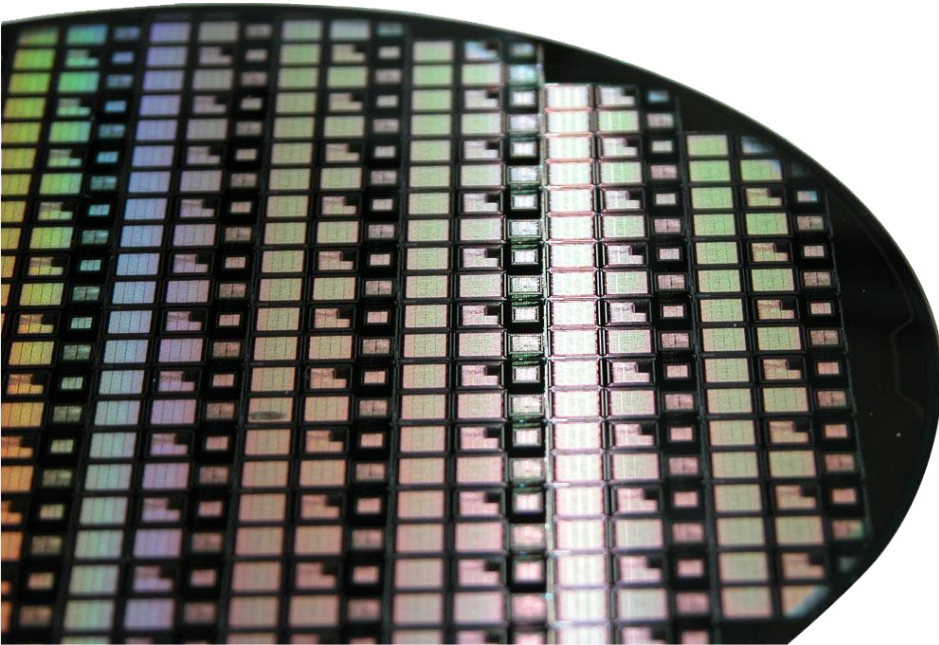
CUDA Programming Model

10.05.2016

Lukas Cavigelli

Daniele Palossi

ETZ E 9 / ETZ D 61.1



Exercises

1. Enumerate GPUs
2. Hello World
 - CUDA kernel
3. Vectors Addition
 - Threads and blocks
4. 1D Stencil
 - Benefits and usage of shared memory in a GPU
5. Matrix Multiplication
 - assigning the right data index to threads
6. Matrix Transpose
 - Coalescence and memory accesses without bank conflicts

Get the code and environment setup

- Copy and extract the source code of the lab exercises

```
$ tar xvf ~soc_master/4_cuda/ex.tar
```

```
$ cd ex
```


```
$ source setup-env.sh
```

- Documentation:
 - online, <http://docs.nvidia.com/cuda> (latest version)
 - or ~soc_master/4_cuda/doc/CUDA_C_Programming_Guide_6.5.pdf
- Now we are ready to start! 😊
- Reminder:
 - Sign up for mini-projects!

Exercise 1: Enumerate GPUs

- This is a very simple application which displays the properties and information about the GPU of your workstation
- We will use this simple exercise to:
 - Get familiar with the framework
 - Learn how to compile and launch applications
 - Review the application output and code
- Follow next simple instructions:

```
$ cd ex  
$ cd 1.Enumerate_GPUs  
$ make  
$ ./enum_gpu.x
```

- 
- Enter in the application folder
 - Compile the source code
 - Launch the application

Reviewing the application output

- The output should be like the one on the side
- Take your time to review the different fields

```
--- General Information for device 0 ---  
Name: xxxx  
Compute capability: xxxx  
Clock rate: xxxx  
Device copy overlap: xxxx  
Kernel execution timeout : xxxx  
  --- Memory Information for device 0 ---  
Total global mem: xxxx  
Total constant Mem: xxxx  
Max mem pitch: xxxx  
Texture Alignment: xxxx  
  --- MP Information for device 0 ---  
Multiprocessor count: xxxx  
Shared mem per mp: xxxx  
Registers per mp: xxxx  
Threads in warp: xxxx  
Max threads per block: xxxx  
Max thread dimensions: (xxxx, xxxx, xxxx)  
Max grid dimensions: (xxxx, xxxx, xxxx)
```

Reviewing the application code

- Open and review the file enum_gpu.cu
- It exploits the Device Management API
 - Application can query and select GPUs
 - **cudaGetDeviceCount**(int *count)
 - **cudaSetDevice**(int device)
 - **cudaGetDevice**(int *device)
 - **cudaGetDeviceProperties**(cudaDeviceProp *prop, int device)
 - Multiple host threads can share a device
 - A single host thread can manage multiple devices
 - cudaSetDevice(i) to select current device
 - cudaMemcpy(...) for peer-to-peer copies

Reviewing the application code

- It exploits also the Reporting Errors API
 - 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)
 - 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()));`

Exercise 2: Hello World

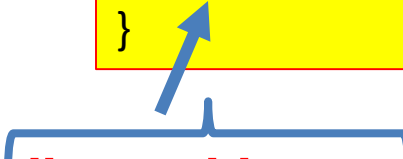
- “Hello World” is the typical basic application
- Move to the “Hello World” application folder and compile

```
$ cd ..  
$ cd 2.Hello_World  
$ make
```

- This application project contains two “Hello World” versions:
 - hello_world.cu: CPU version
 - simple_kernel.cu: GPU version

CPU: Hello World

```
int main(void) {  
    printf("Hello World!\n");  
    return 0;  
}
```



Lets review the code opening the “hello_world.cu” source file:

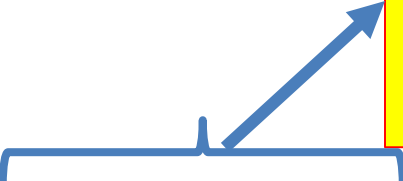
- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code
- To run this version type the following command:

```
$ ./hello_world.x
```

GPU: Hello World with Device Code

- To run this version type the following command:

```
$/simple_kernel.x
```



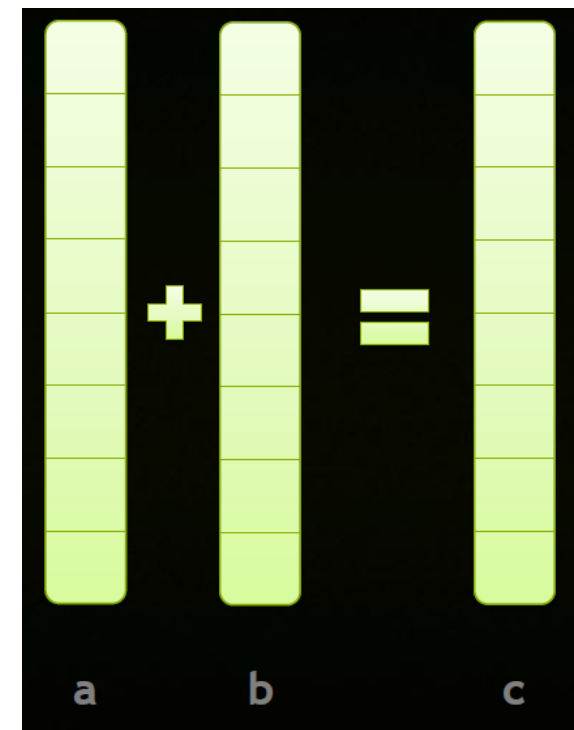
```
__global__ void kernel(void) {  
}  
  
int main(void) {  
    kernel<<<1,1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

Lets review the code opening the “simple_kernel.cu” source file:

- CUDA C/C++ keyword **__global__** indicates a function that:
 - Runs on the device
 - Is called from host code
- nvcc** separates source code into host and device components
 - Device functions (e.g. *kernel()*) processed by NVIDIA compiler
 - Host functions (e.g. *main()*) processed by standard host compiler
 - gcc, cl.exe
- 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!
- kernel()* does nothing!

Parallel Programming in CUDA C/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
- Move to the "3.Vectors_Addition" application folder
- Compile with "make" command
- You will have different application executables:
 - add_simple.x
 - add_simple_blocks.x
 - add_simple_threads.x
 - add_simple_last.x



Exercise 3: Vectors Addition on the Device

- Open and review file “add_simple.cu”
- 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
- Note that we use pointers for the variables
- 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()

```
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  
    int 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;
```

Addition on the Device: main()

```
// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

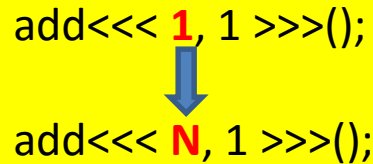
// Launch add() kernel on GPU
add<<<1,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

Moving to Parallel

- Open and review file “add_simple_blocks.cu”
- GPU computing is about massive parallelism
 - So how do we run code in parallel on the device?



The diagram illustrates the concept of parallelism in GPU computing. It shows two lines of CUDA code. The first line is `add<<< 1, 1 >>>();` where the number `1` is red. A blue arrow points down from this `1` to the `N` in the second line, `add<<< N, 1 >>>();`, where `N` is also red. Both lines of code are enclosed in a yellow rectangular box with a red border.

- 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 element of the array

Vector Addition on the Device

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

- On the device, each block can execute in parallel:



Vector Addition on the Device: main()

```
#define N 512
int main(void) {
    int *a, *b, *c;                // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;    // device copies of a, b, c
    int size = N * sizeof(int);
    ....

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&dev_a, size);
    cudaMalloc((void **)&dev_b, size);
    cudaMalloc((void **)&dev_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);
```

Vector Addition on the Device: main()

```
// Copy inputs to device
cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N blocks
add<<<N,1>>>(dev_a, dev_b, dev_c);

// Copy result back to host
cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);
return 0;
}
```

What we have learnt so far

- 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

CUDA Threads

- Open and review file “add_simple_threads.cu”
- 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()...

Vector Addition Using Threads: main()

```
#define N 512
int main(void) {
    int *a, *b, *c;                // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;    // device copies of a, b, c
    int size = N * sizeof(int);
    ...

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&dev_a, size);
    cudaMalloc((void **)&dev_b, size);
    cudaMalloc((void **)&dev_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);
```

Vector Addition on the Device: main()

```
// Copy inputs to device
cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N threads
add<<<1,N>>>(dev_a, dev_b, dev_c);

// Copy result back to host
cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);
return 0;
}
```


Combining Blocks and Threads

- Open and review file “add_simple_last.cu”
- We have seen parallel vector addition using:
 - Several blocks with one thread each
 - One block with several threads
- Let’s adapt vector addition to use both blocks and threads

Indexing Arrays with Blocks and Threads

- No longer as simple as using `blockIdx.x` and `threadIdx.x`
 - Consider indexing an array with one element per thread (8 threads/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;  
          = 5 + 2 * 8;  
          = 21;
```

Vector Addition with Blocks and Threads

- Use the built-in variable blockDim.x for 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: main()

```
#define N 512
int main(void) {
    int *a, *b, *c;                // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;    // device copies of a, b, c
    int size = N * sizeof(int);
    ...

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&dev_a, size);
    cudaMalloc((void **)&dev_b, size);
    cudaMalloc((void **)&dev_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);
```

Addition with Blocks and Threads: main()

```
// Copy inputs to device
cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(dev_a, dev_b, dev_c);

// Copy result back to host
cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);
return 0;
}
```

Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of `blockDim.x`
- Avoid accessing beyond the end of the arrays:

```
__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 the kernel launch:

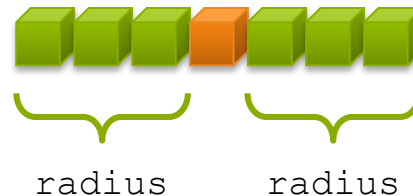
```
add<<<(N + M - 1) / M, M>>>(d_a, d_b, d_c, N);
```

Why Bother with Threads?

- Threads seem unnecessary
 - They add a level of complexity
 - What do we gain?
- Unlike parallel blocks, threads have mechanisms to efficiently:
 - Communicate
 - Synchronize

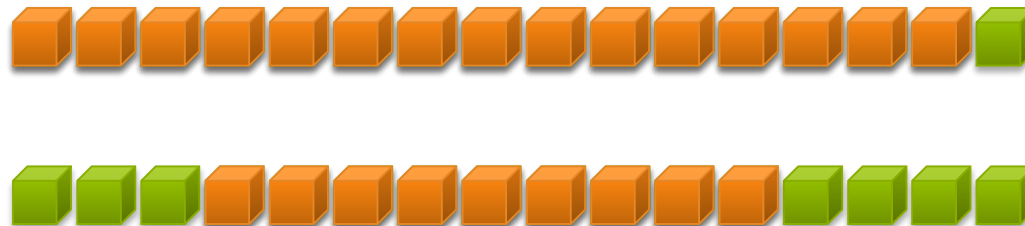
Exercise 4: 1D Stencil

- Consider applying a 1D stencil to a 1D array of elements
 - Each output element is the sum of input elements within a radius
- If radius is 3, then each output element is the sum of 7 input elements:



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

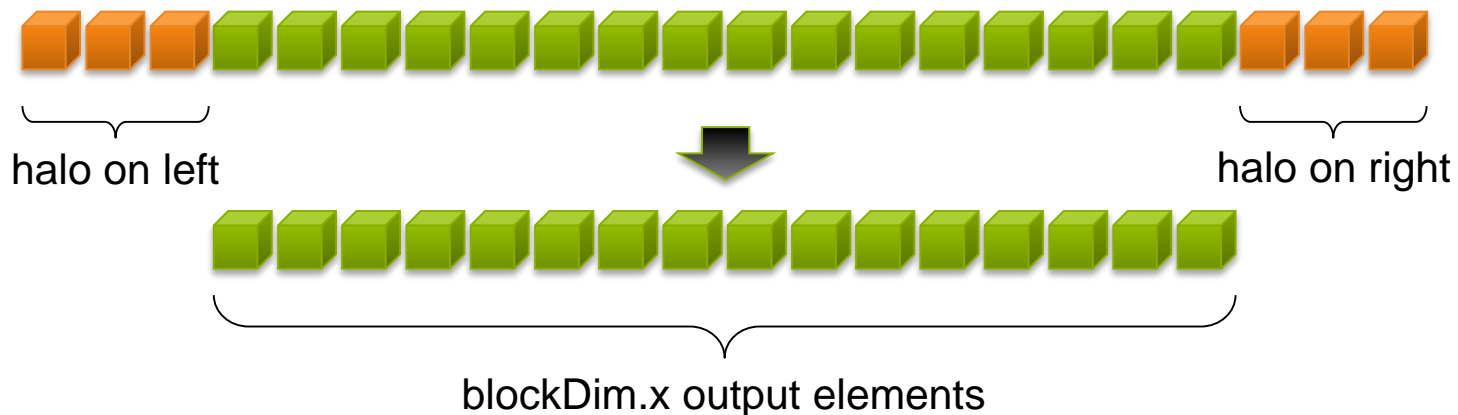


Sharing Data Between Threads

- Terminology: within a block, threads share data via **shared memory**
- Extremely fast on-chip memory, user-managed
- Declare using `__shared__`, allocated per block
- Data is not visible to threads in other blocks

Implementing With Shared Memory

- Cache data in shared memory
 - Read $(\text{blockDim.x} + 2 * \text{radius})$ 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



Question?

Find the right vector size and complete the code

```
__global__ void stencil_1d(int *in, int *out) {  
    __shared__ int temp[/* WHAT SIZE? */];  
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;  
    int lindex = threadIdx.x + RADIUS;
```



// Read input elements into shared memory

```
temp[lindex] = in[gindex];  
if (threadIdx.x < RADIUS) {  
    temp[lindex - RADIUS] = in[gindex - RADIUS];  
    temp[lindex + BLOCK_SIZE] =  
        in[gindex + BLOCK_SIZE];  
}
```



Stencil Kernel

```
// Apply the stencil
```

```
int result = 0;
```

```
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
```

```
    result += temp[lindex + offset];
```



```
// Store the result
```

```
out[gindex-RADIUS] = result;
```

```
}
```

Data Race!

- The stencil example will not work...
- Suppose thread 15 reads the halo before thread 0 has fetched it...

```
temp[lindex] = in[gindex];           Store at temp[18]   
if (threadIdx.x < RADIUS) {  
    temp[lindex - RADIUS] = in[gindex - RADIUS];    Skipped, threadIdx > RADIUS  
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];  
}  
  
int result = 0;  
result += temp[lindex + 1];          Load from temp[19] 
```

__syncthreads()

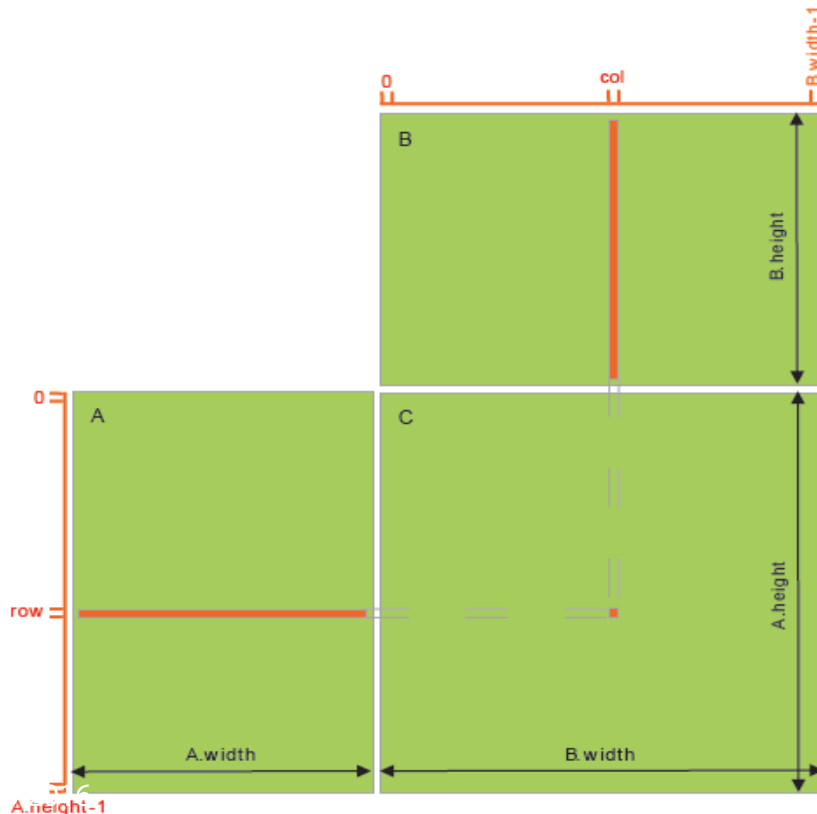
- `void __syncthreads();`
- Synchronizes all threads within a block
 - Used to prevent RAW / WAR / WAW hazards
- All threads must reach the barrier
 - In conditional code, the condition must be uniform across the block

QUESTION

- **Add the `__syncthreads()` call in the right code line.**

Exercise 5: Matrix Multiplication

- The application performs a Matrix-Matrix Multiplication on GPU
- Complete the kernel code in order to assign the right data index to threads
 - Assign one thread per output



```
__global__ void matmatmulgpu( double *a, double *b, double *c, int lda){
    unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;

    // assign one thread per output - row major over whole matrix
    int row = /* WHAT INDEX? */ ;
    int col = /* WHAT INDEX? */ ;
    double sum = 0.0;

    for ( int k=0; k<lda; k++ ) {
        sum += a[row*lda+k] * b[k*lda+col];
    }
    c[tid] = sum; return;
}
```

Exercise 6: Matrix Transpose

- This example transposes arbitrary-size matrices
- It compares a naive transpose kernel that suffers from non-coalesced writes, to an optimized transpose with fully **coalesced memory access** and **no bank conflicts**
- Complete the GPU kernel
- Gain additional insight using the profiler ('nvvp')
- Configure what is being measured (e.g. DRAM write BW)
- Compare effect of shared mem in the stencil example

```
// This kernel is optimized to ensure all global reads and writes are coalesced,  
// and to avoid bank conflicts in shared memory. This kernel is up to 11x faster  
// than the naive kernel below. Note that the shared memory array is sized to  
// (BLOCK_DIM+1)*BLOCK_DIM. This pads each row of the 2D block in shared memory  
// so that bank conflicts do not occur when threads address the array column-wise.  
__global__ void transpose(float *odata, float *idata, int width, int height){  
  
    __shared__ float block[BLOCK_DIM][BLOCK_DIM+1];  
  
    // read the matrix tile into shared memory  
    // load one element per thread from device memory (idata) and store it  
    // in transposed order in block[][]  
    unsigned int xIndex = /* COMPLETE */ ;  
    unsigned int yIndex = /* COMPLETE */ ;  
  
    if((xIndex < width) && (yIndex < height))  
    {  
        unsigned int index_in = /* COMPLETE */ ;  
        block[threadIdx.y][threadIdx.x] = /* COMPLETE */ ;  
    }  
  
    // synchronise to ensure all writes to block[][] have completed  
    /* COMPLETE */ ;  
  
    // write the transposed matrix tile to global memory (odata) in linear order  
  
    xIndex = /* COMPLETE */ ;  
    yIndex = /* COMPLETE */ ;  
    if((xIndex < height) && (yIndex < width))  
    {  
        unsigned int index_out = /* COMPLETE */ ;  
        odata[index_out] = /* COMPLETE */ ;  
    }  
}
```

NVVP (Nvidia Visual Profiler)

- Launch with 'nvvp'
- Create session
- Explore the timeline
- Select kernel to see more info
- Use 'Examine individual kernels' to explore further
 - Stall reasons
 - Mem BW analysis
 - Load on different units
 - Occupancy

