JOHANNES KEPLER
UNIVERSITY LINZ

INSTITUTE OF
COMPUTER GRAPHICS

Computer Graphics

Lab 7: Introduction to CUDA

cg-lab@jku.at

Setup: Your personal PC

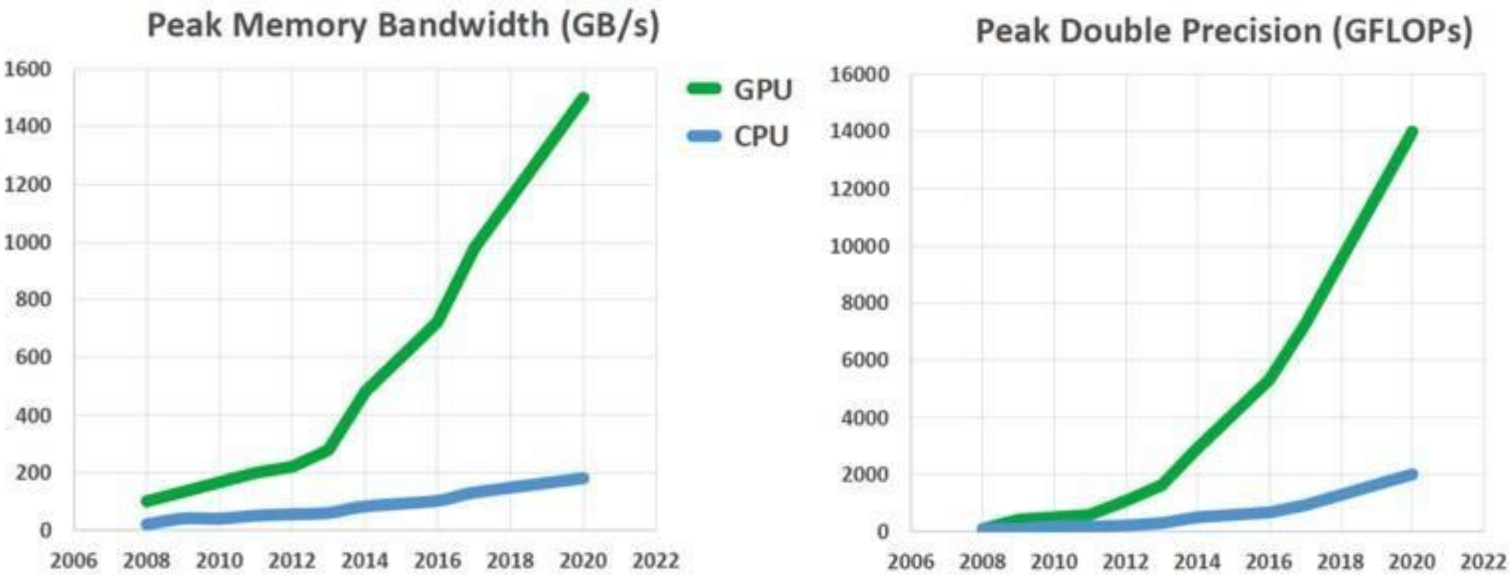
- CUDA requires Nvidia graphics card!
- Install:
    1. Visual Studio (tested with Express 2017)
    2. Cmake (tested with version 3.18.2)
    3. CUDA Toolkit (tested with version 11.0)
- Download and Unpack the provided ZIP file from Moodle
- Adapt "configureAuto.bat" to your installed program versions and paths
- Run bat file and hope for the best :-)
    - In case of errors use cmake. cmake-gui could be quite helpful.

Setup: Project

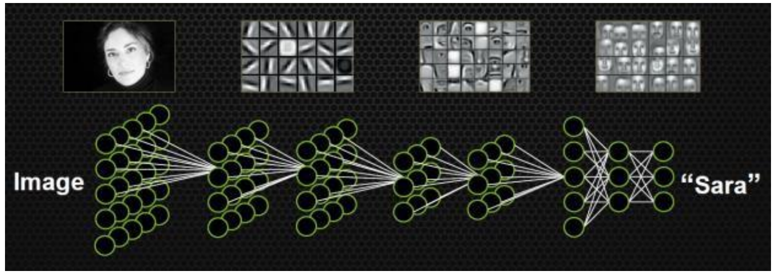
1. Download and Unpack the provided ZIP file from Moodle
2. Execute "configureAuto.bat"
3. Visual Studio should open
4. The project ex7cAdvancedShader should be your startup project
5. Use Release and x64 build
6. Source code can be found in project folder ex7cAdvancedShader
   1. C++ (cpp) files are in sub-folder "Source Files"
   2. CUDA (cu) files are in sub-folder "CU-Files"
7. Start program by pressing green rectangle (Local Windows Debugger)
   1. Answer any dialog with "Yes" or "Reload"
8. (Optional) Enable syntax highlighting for CUDA files:
   1. Tools->Options->Text Editor->File Extension
   2. Add: Extension=cu Editor=Microsoft Visual C++
9. (Optional) If you accidentally close Visual Studio open the project file:
   1. build\ex7cAvancedShader.sln

Motivation

- GPUs are fast



- GPGPU: use it for other tasks, rather than graphics only:
  - Started around 2000 with programmable GPU (GeForce 3)
  - CUDA came 2006
  - Since then:
    Deep Learning,
    Cryptocurrency, …



Structure

- What is CUDA?
- Hardware and Software
- CUDA-C
- Memory Transfer and Types
- CUDA and OpenGL
- Special operations and structures
- Execution and Performance

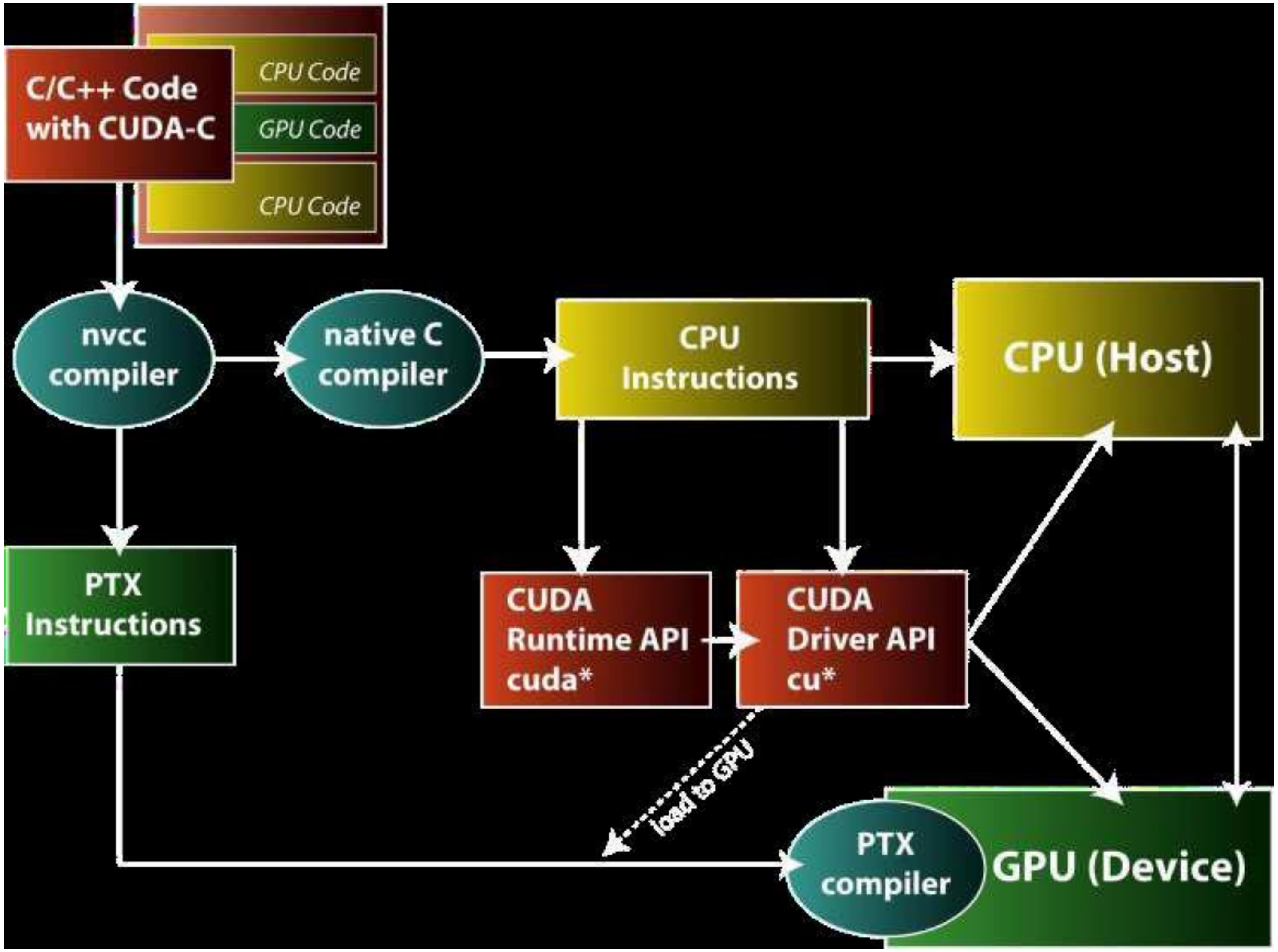
What is CUDA?

- Software platform for parallel computing on NVIDIA GPUs
- CUDA-C: Compatible to C/C++ with few simple extensions
- Simplifies implementation of parallel code which runs in many threads
- Threads and blocks of threads are managed without much additional code
- CUDA is a scalable parallel programming model:
  Usually no recompilation required for different hardware (though, tweaking might improve performance)

CUDA Compile Pipeline

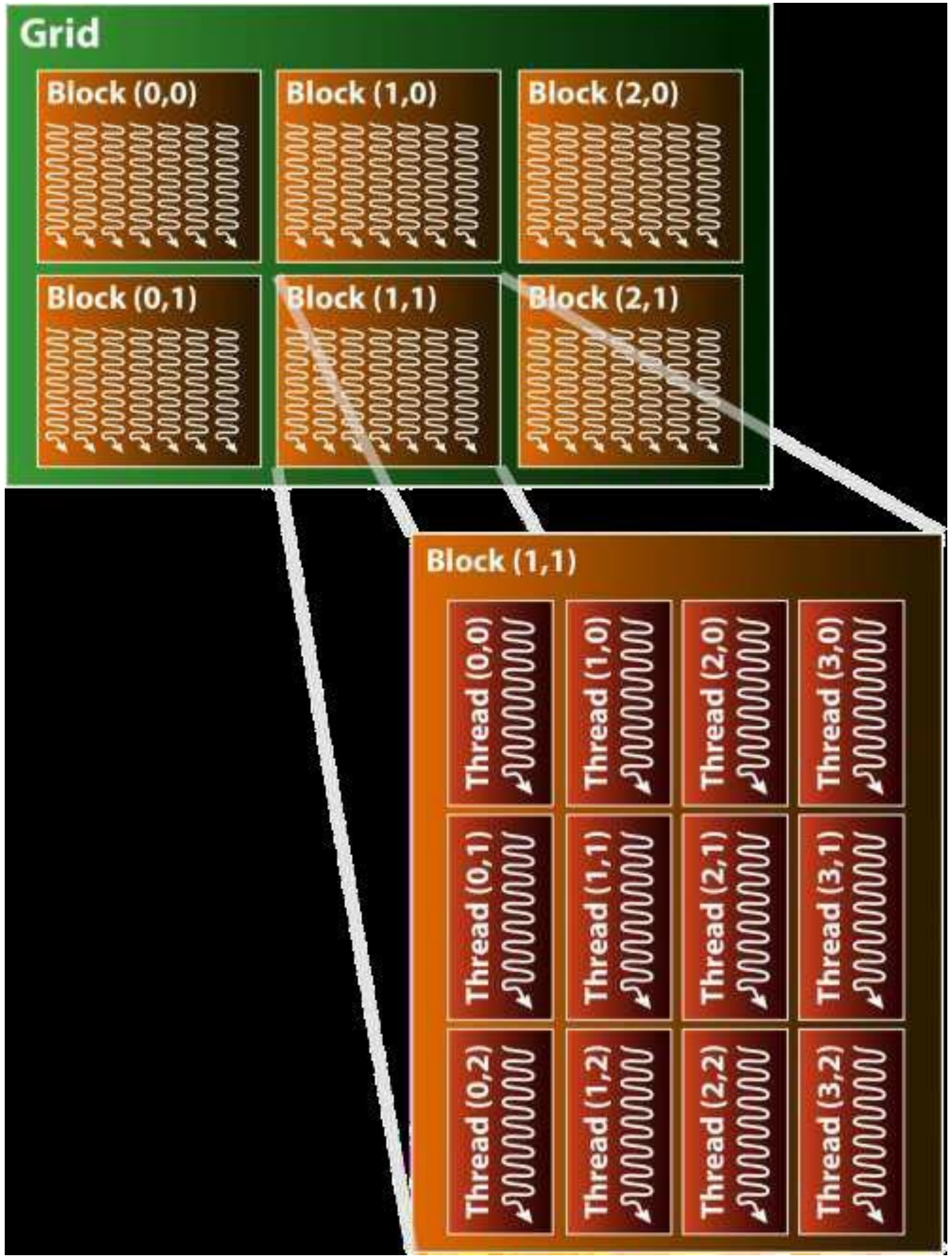
JOHANNES KEPLER UNIVERSITY LINZ

INSTITUTE OF COMPUTER GRAPHICS



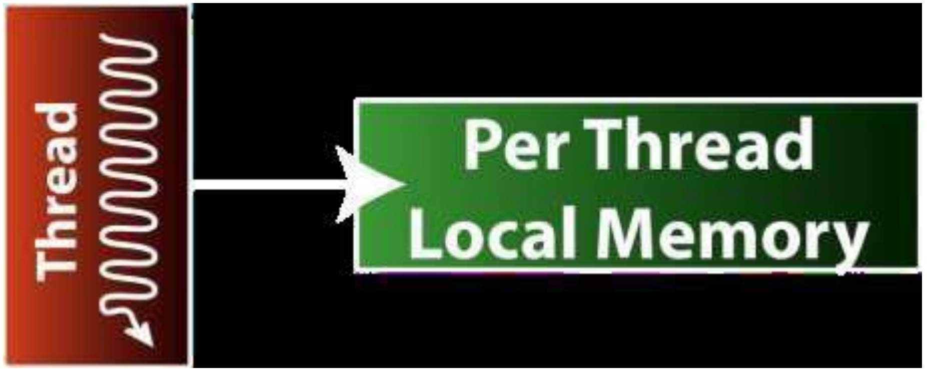
CUDA Thread Hierarchy

- Threads are smallest unit
- Threads are grouped into Blocks
- Blocks together form the Grid



Hierarchy: Threads

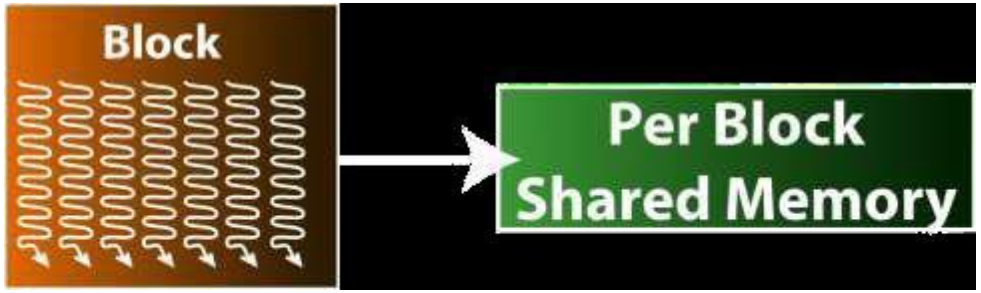
- The same sequential program (kernel function) is executed by all threads
- Each thread has its own per-thread local memory
  (very small but also very fast)
- Thread is identified by its threadIdx



Hierarchy: Blocks

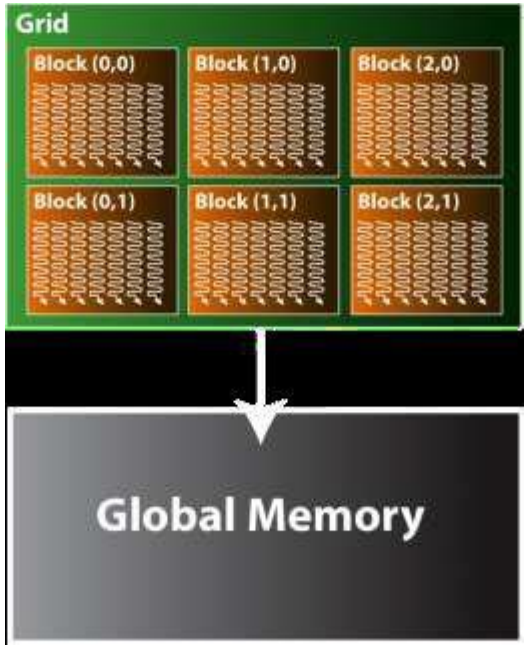
JOHANNES KEPLER UNIVERSITY LINZ
INSTITUTE OF COMPUTER GRAPHICS

- Threads are grouped into blocks
- All threads of a block can synchronize and share memory
- Blocks have per-block shared memory
- Shared memory is also quite small but very fast as well
- Identified by its blockIdx
- Size of a block passed to each block as blockDim



Hierarchy: Grids

- All blocks of a program (kernel) form a Grid
- All grids have access to global memory (huge, but very slow)
- Global memory is actual VRAM of GPU and hence may contain large textures



Identification

- Each thread can access its threadIdx, the blockIdx of its block and the size of the block, blockDim
- Those variables are 3D for easy addressing, however all components default to 1 so they can be also used 1D or 2D
- Each thread can be uniquely identified.
- 1D example:
  int idx = blockIdx.x * blockDim.x + threadIdx.x

CUDA C

- CUDA Code is split into two parts
  - Host code to be executed by CPU (host)
  - Device code to be executed by GPU (device)
- Can be specified in the same file, later on automatically separated and compiled (with nvcc)
- Memory is also split in two parts
  - Host memory can only be accessed by host code
    - Normally in main memory
  - Device memory can only be accessed by device code
    - Normally in graphics card memory
  - Memory can be copied between host and device, as well as mapped
    - CUDA (≥6) + GPU (compute capability ≥3): unified memory (copying is done automatically)

CUDA C Syntax

- Functions (or methods) on host do not need to be specially marked

- Device functions that can be called from host (kernels) are marked with __global __ keyword:
- __global__ void some_func(float* some_param) { ... }

- Pure device helper functions are marked with __device__ keyword (can’t be called from host):
- __device__ float min(float X, float Y) { ... }

Simple Example: Sequential C++ Code

- How does simple element-wise addition of two vectors (=arrays) look like?

- Sequential C++ Code (no CUDA):

```
void vec_add(const float *in_a, const float *in_b, float *result, int size)
{
    for(int idx = 0; idx < size; idx++)
            result[idx] = in_a[idx] + in_b[idx];
}
```

CUDA C Example: Device Code

- How does simple element-wise addition of two vectors (=arrays) look like?
- Define the kernel for the addition:

```
/* Device Code */
__global__
void vec_add(const float *in_a, const float *in_b, float *result, int size)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx >= size) {
        return;
    }
    result[idx] = in_a[idx] + in_b[idx];
}
```

CUDA C Example: Host Code

- Kernel needs to be launched from host code
- Therefore, we need to determine the number of blocks and threads required
- With this information, the kernel can be launched:

```
/* Host Code */
int num_elem = ...;
int num_threads = 128;
int num_blocks = (num_elem+num_threads-1)/num_threads;
/* launch! */
vec_add<<<num_blocks, num_threads>>>(in_a, in_b, result, num_elem);
/* test for errors here! */
```

CUDA C Example: Grid Sizes?

- Why so complicated?
- if(idx >= size) { return; }
- int num_blocks = (num_elem+num_threads-1)/num_threads;

- Because num_elem could possibly be not a multiple of num_threads!

  num_elem/num_threads: Some values might be missing from computation!

  (num_elem+num_threads-1)/num_threads: Actual grid is too large now, so each thread needs to check if it is accessing valid data

- This is not bad: the if() branch will hardly be taken, so there won't be any divergence but for the very last few threads

Global Memory

- Global memory (on the graphics card) has to be allocated by host
- Can be done using runtime API or driver API, however I will only describe runtime API
- Done using cudaMalloc call
- Example:

```
int num_elem = ...;
float *d_data = NULL;
cudaMalloc( (void**)&d_data, sizeof(float)*num_elem );
```

- Remember, d_data cannot be accessed from host code! If you try, this will result in undefined behavior!
- You should always check if allocation was successful!

Memory Transfer

- To actually do something meaningful, we need to get data to and from device!
- This is typically done using cudaMemcpy
- Synchronized example:

```
float *d_data = ...; // pointer to device data
float *h_data = ...; // pointer to host data
cudaMemcpy(d_data, h_data, size, cudaMemcpyHostToDevice);
cudaMemcpy(h_data, d_data, size, cudaMemcpyDeviceToHost);
cudaMemcpy(d_data2, d_data, size,cudaMemcpyDeviceToDevice);
```

- Also asynchronous cudaMemcpyAsync available

- Good news: unified memory does copying automatically (we will not use it today)

Simple CUDA Example – Task 1

- CUDA has to be initialized before usage:

  Look into our cudaul::initCUDA() helper function for details…

- simple.cu:

  Write CUDA kernel for (element-wise) multiplication of two arrays (step 1.1)

  Add device memory allocation and copy input to device memory (step 1.2)

  Define block and grid size and call cuda kernel (step 1.3)

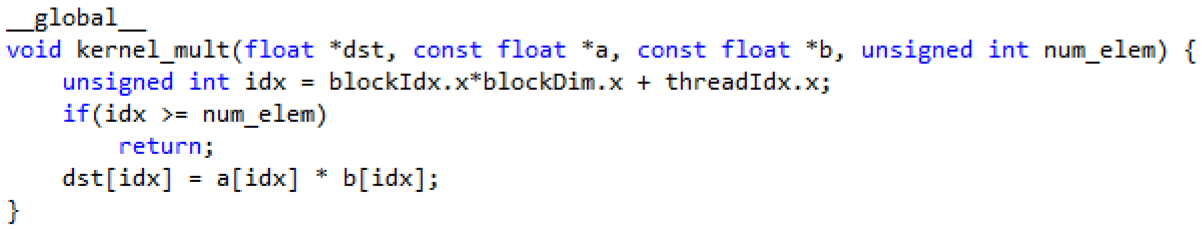
  Copy the result back to host memory (step 1.4)

- When your code is right the total squared error should be 0.

- (Not a very useful task for CUDA, memory transfer too expensive, CPU implementation is faster!)

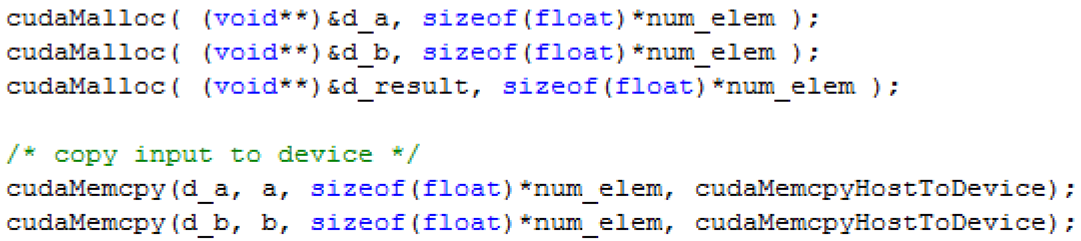
Simple CUDA Solution

- Step 1.1



```
__global__
void kernel_mult(float *dst, const float *a, const float *b, unsigned int num_elem) {
    unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;
    if(idx >= num_elem)
        return;
    dst[idx] = a[idx] * b[idx];
}
```

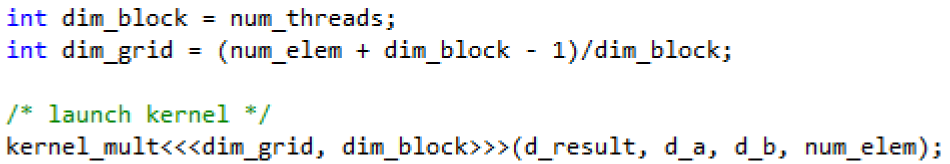
- Step 1.2



```
cudaMalloc( (void**)&d_a, sizeof(float)*num_elem );
cudaMalloc( (void**)&d_b, sizeof(float)*num_elem );
cudaMalloc( (void**)&d_result, sizeof(float)*num_elem );

/* copy input to device */
cudaMemcpy(d_a, a, sizeof(float)*num_elem, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, sizeof(float)*num_elem, cudaMemcpyHostToDevice);
```

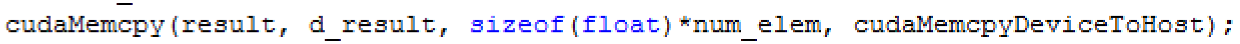
- Step 1.3



```
int dim_block = num_threads;
int dim_grid = (num_elem + dim_block - 1)/dim_block;

/* launch kernel */
kernel_mult<<<dim_grid, dim_block>>>(d_result, d_a, d_b, num_elem);
```

- Step 1.4



```
cudaMemcpy(result, d_result, sizeof(float)*num_elem, cudaMemcpyDeviceToHost);
```

More on indices and launch parameters

- They do not need to be 1D and int
- CUDA C comes with dim3 data type
- Example: Launch of 2D kernel over an image:

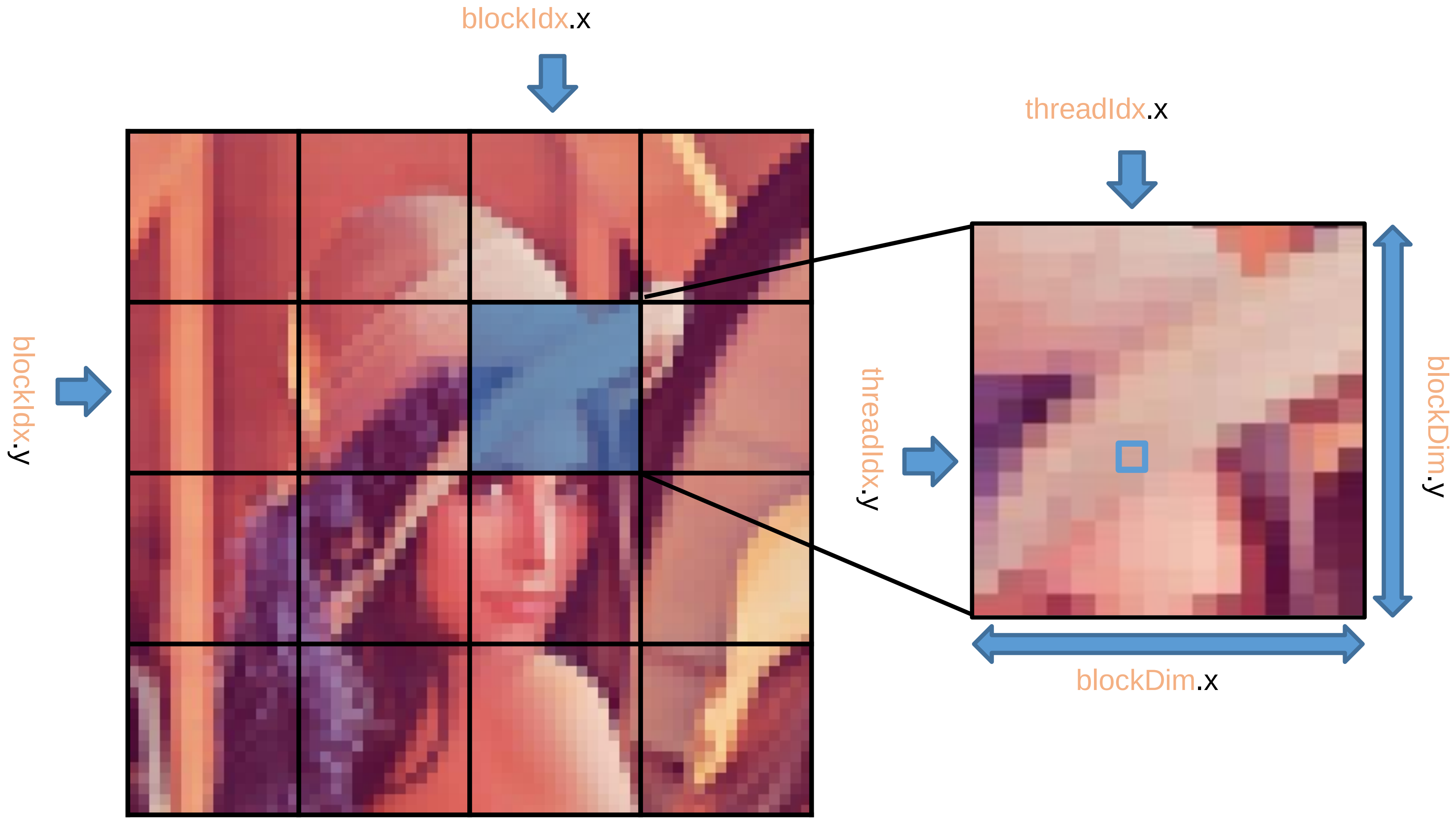
```
int width = ...;
int height = ...;
/* 16x16 = 256 threads */
dim3 block_size(16, 16);
/* adjust to process each pixel! */
dim3 grid_size;
grid_size.x = (width+block_size.x-1)/block_size.x;
grid_size.y = (height+block_size.y-1)/block_size.y;
/* launch: */
process_image<<<grid_size, block_size>>>(image, width, height);
```

More on indices and launch parameters II

- Device code is very similar.
- Image is accessed as a 1D array of pixels:

```
__global___
void process_image(uchar3 *image, int width, int height) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if(x<width && y<height) {
            image[y*width+x] = …;
    }
}
```

Indexing for Images



OpenGL Interoperability

- CUDA offers interoperability with OpenGL
- Transfer of data from or to OpenGL can be performed directly on device (theoretically high performance, sometimes bugs)
- OpenGL buffer objects can be directly mapped into CUDA's address space, e.g.:

  Create PBO (pixel buffer object) or VBO (vertex …) in OpenGL

  Register as CUDA mappable buffer

  Map buffer to CUDA → e.g. uchar3* on device

  OpenGL must not access buffer object any longer!

  Read/Write buffer from inside CUDA kernels

  Unmap buffer → CUDA must not access pointer any longer!

  OpenGL can use buffer to draw vertices or pixels

Pixel Buffer Objects

- Create PBO

```
GLuint id= 0;
glGenBuffers(1, & id);
glBindBuffer(GL_ARRAY_BUFFER, id);
glBufferData(GL_ARRAY_BUFFER, size, data, GL_DYNAMIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

- Register PBO

```
cudaGLRegisterBufferObject(id);
```

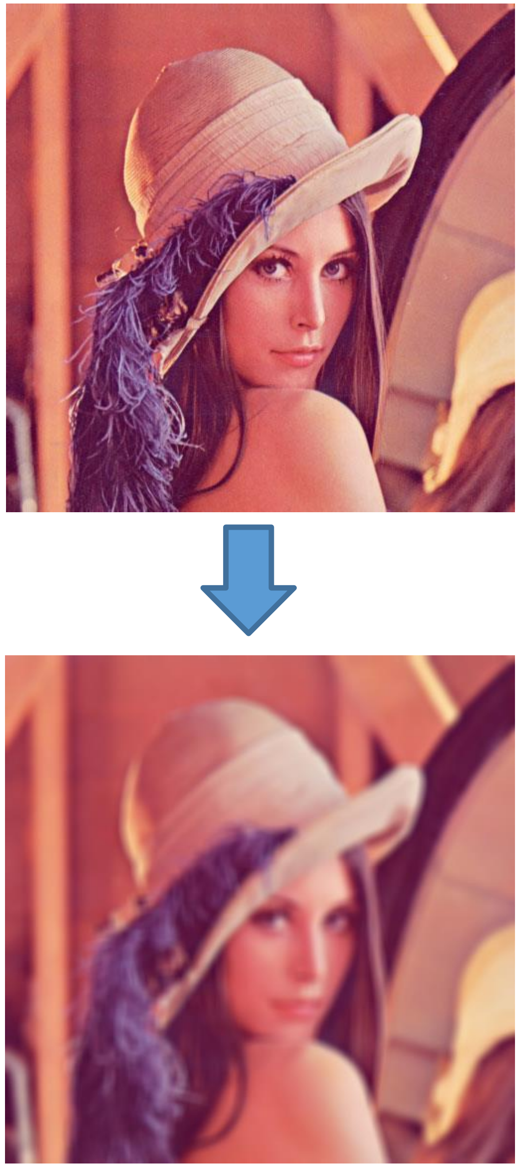
- Map/Unmap PBO

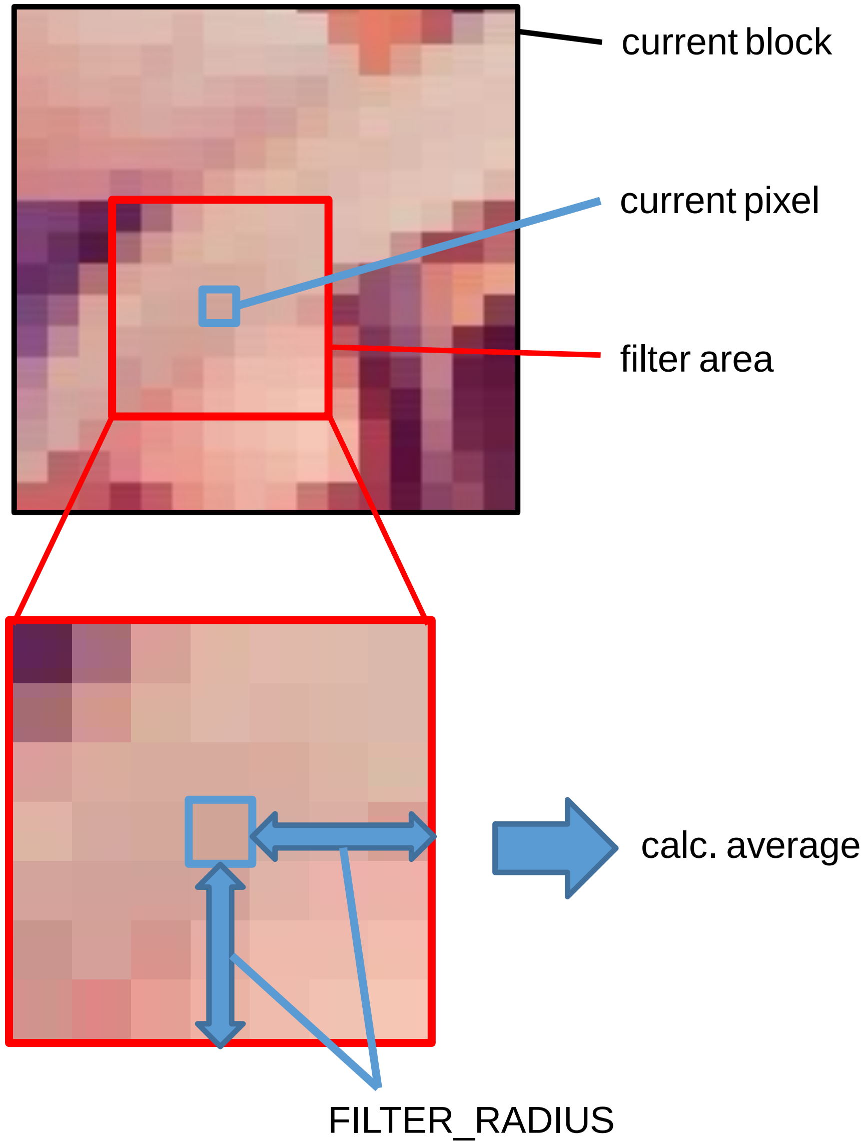
```
unsigned char *array_d = NULL;
cudaGLMapBufferObject((void**)&array_d, id);
…
cudaGLUnmapBufferObject(id);
```

- Copy PBO to Texture

```
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, id);
glTexImage2D(GL_TEXTURE_2D, 0, 3, width, height,0, GL_RGB, GL_UNSIGNED_BYTE, 0);
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);
```

Mean Filter





current block

current pixel

filter area

calc. average

FILTER_RADIUS

OpenGL Example – Task 2

- Uncomment oglexample(…) in main.cpp and comment simplecudaexample()
- Add code for mapping and unmapping the used PBOs (pbo_input and pbo_output) in openglmain.cpp, (steps 2.1-2)

  (You should now see a greyscale image.)
- Change kernel function to apply a mean filter instead of converting the image to greyscale (oglex.cu, step 2.3)
- Hints:

  use 2 for-loops to iterate over the filter area (box)

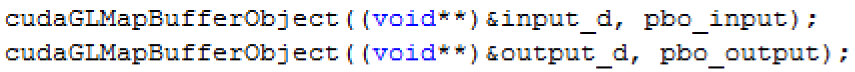
  FILTER_RADIUS constant defines size for mean filter

  use clamp(…) to ensure new pixel coordinates are valid

  Use built in vector types: uchar3, uint3
- Info: Look at initCUDAWithOpenGL to see how to enable OpenGL interoperability for CUDA.

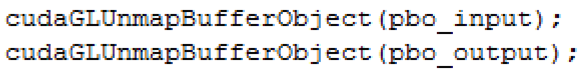
OpenGL Example Solution

- Step 2.1 (openglmain.cpp)



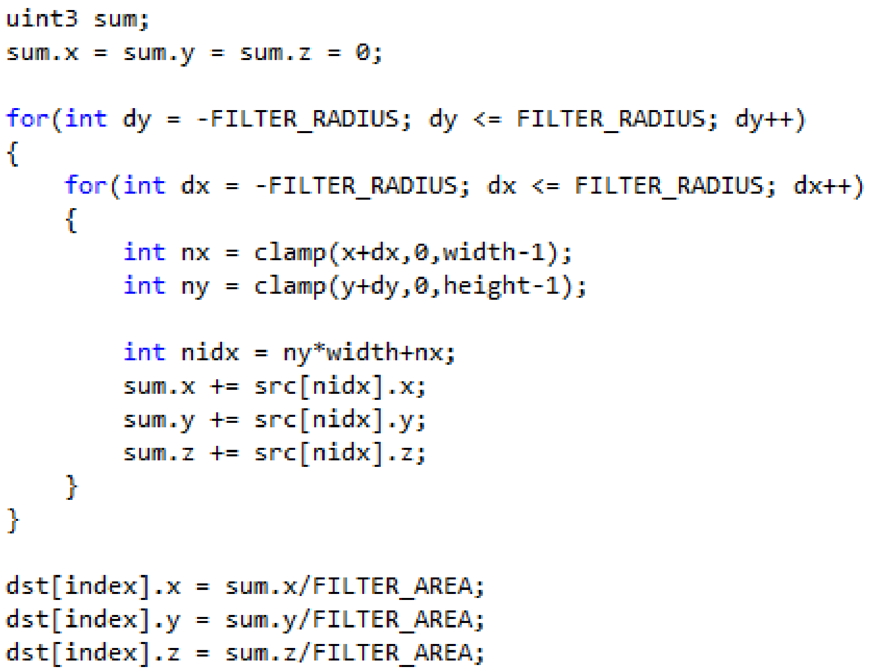
```
cudaGLMapBufferObject((void**)&input_d, pbo_input);
cudaGLMapBufferObject((void**)&output_d, pbo_output);
```

- Step 2.2 (openglmain.cpp)



```
cudaGLUnmapBufferObject(pbo_input);
cudaGLUnmapBufferObject(pbo_output);
```

- Step 2.3 (oglex.cu)



```
uint3 sum;
sum.x = sum.y = sum.z = 0;

for(int dy = -FILTER_RADIUS; dy <= FILTER_RADIUS; dy++)
{
    for(int dx = -FILTER_RADIUS; dx <= FILTER_RADIUS; dx++)
    {
        int nx = clamp(x+dx,0,width-1);
        int ny = clamp(y+dy,0,height-1);

        int nidx = ny*width+nx;
        sum.x += src[nidx].x;
        sum.y += src[nidx].y;
        sum.z += src[nidx].z;
    }
}

dst[index].x = sum.x/FILTER_AREA;
dst[index].y = sum.y/FILTER_AREA;
dst[index].z = sum.z/FILTER_AREA;
```

Shared Memory

- Kernels can make use of shared memory variables to increase performance and/or as temporal storage, they
- Reside in shared memory of each block
- Have lifetime of that block
- Are only accessible from within the same block

- Shared memory is declared using__shared__keyword, e.g.
- __shared__float array[16];

- Size can be determined at run-time as well (only one array):
- extern__shared__float array[];
- The size is given as a third parameter between <<<…>>> in the kernel call.

Thread Synchronization

- Threads of a block can be synchronized with the __syncthreads() statement

- This barrier will only be passed if all threads of the block have finished

- Example pattern:
  - First, each thread reads data from global memory to shared memory
  - Wait until all threads are done using __syncthreads()
  - Work on shared data
  - Wait until all threads finished work using __syncthreads() again
  - Each threads writes back to global memory

Shared memory / sync Example

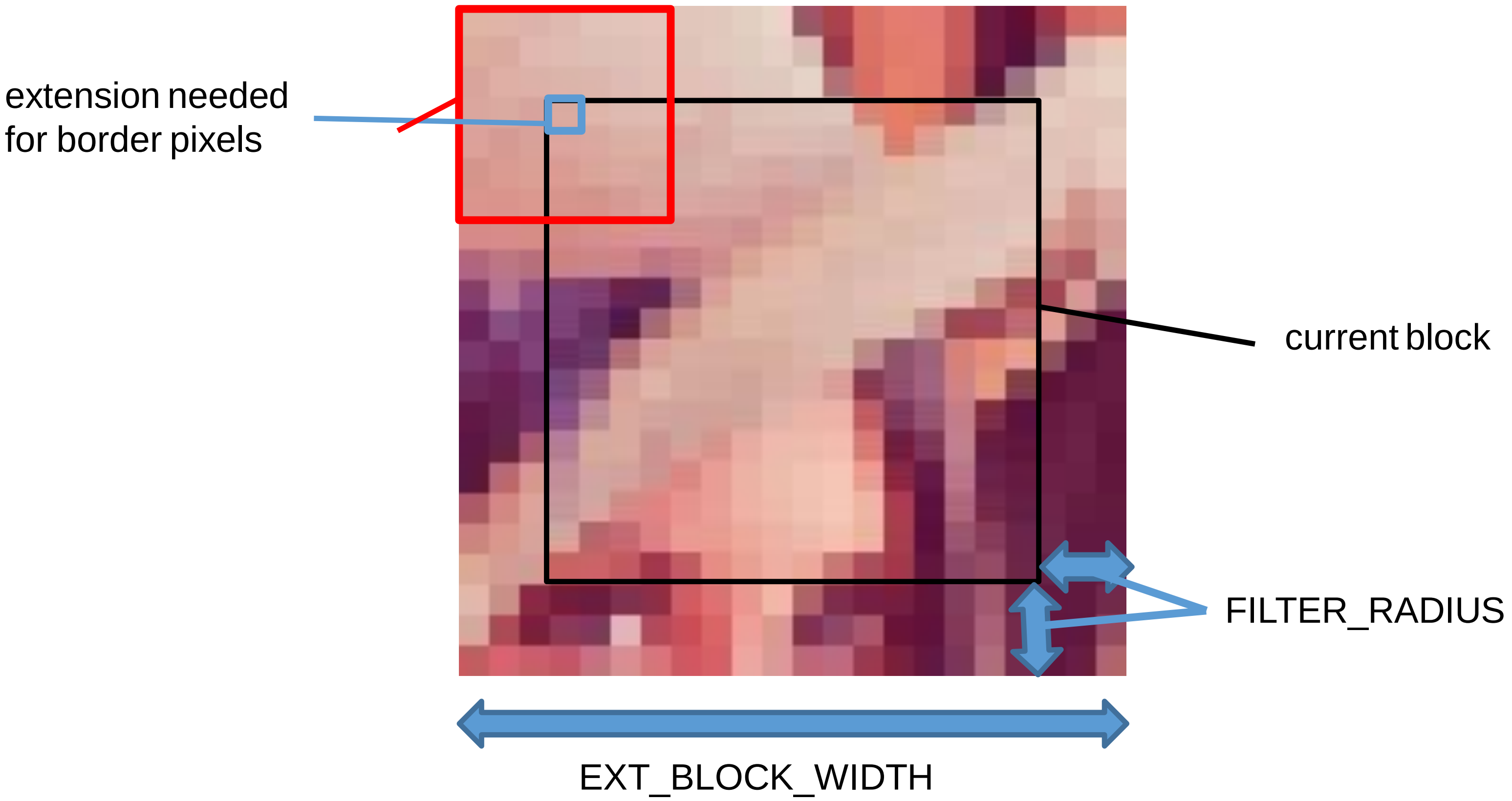
- Simple example on shared memory and synchronization:

```
__global__
void worker(float *data, float *result) {
extern__shared__float array[];
int idx = blockIdx.x * blockDim.x + threadIdx.x;
/* read to shared */
array[threadIdx.x] = data[idx];
/* sync! */
__syncthreads();
/* process data */
…
/* wait */
__syncthreads();
/* write back */
result[idx] = array[threadIdx.x];
}
```

- Many Possibilities for Optimization! (not discussed in this lab)

Shared Memory Example

- Use shared memory to speed up mean filter.
- Copy current pixel block extended by filter radius to shared memory before processing:



Shared Memory Example – Task 3

- New kernel function: kernel_process_sharedmem

- Copying of extended block to shared memory is already done.

- Add mean filter part again (oglex.cu, step 3.1):

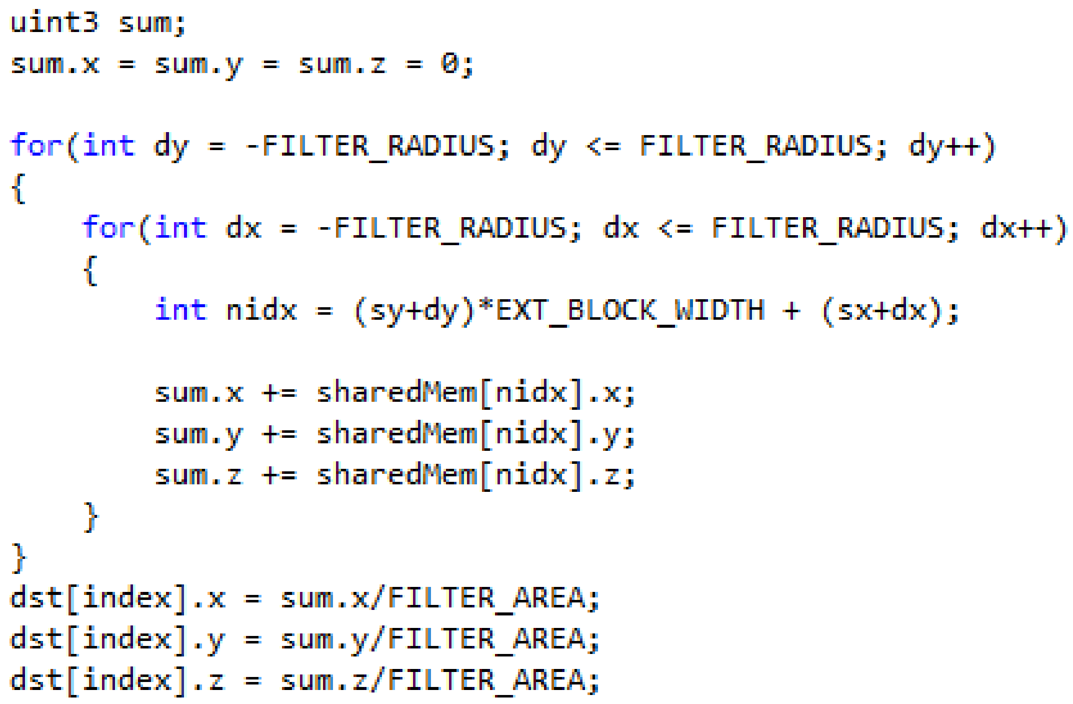
    very similar to first example
    use shared memory (sharedMem) instead of source image (src)
    index within shared memory (extended block) has to be calculated
    width of extended block is EXT_BLOCK_WIDTH

- Call new kernel with shared memory (oglex.cu, step 3.2).

- Compare the processing time of the two kernels.

Shared Memory Example Solution

- Step 3.1:

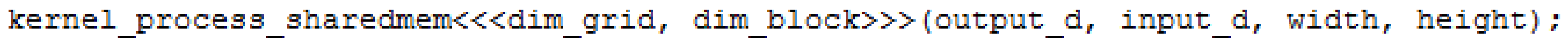


```
uint3 sum;
sum.x = sum.y = sum.z = 0;

for(int dy = -FILTER_RADIUS; dy <= FILTER_RADIUS; dy++)
{
    for(int dx = -FILTER_RADIUS; dx <= FILTER_RADIUS; dx++)
    {
        int nidx = (sy+dy)*EXT_BLOCK_WIDTH + (sx+dx);

        sum.x += sharedMem[nidx].x;
        sum.y += sharedMem[nidx].y;
        sum.z += sharedMem[nidx].z;
    }
}
dst[index].x = sum.x/FILTER_AREA;
dst[index].y = sum.y/FILTER_AREA;
dst[index].z = sum.z/FILTER_AREA;
```

- Step 3.2:



```
kernel_process_sharedmem<<<dim_grid, dim_block>>>(output_d, input_d, width, height);
```

Atomic Operations

- If synchronized access is necessary, CUDA provides atomic operations since compute capability 1.1

- atomicAdd(), atomicSub(), atomicMin(), atomicMax(), atomicInc(), atomicDec(), atomicXor(), …

- Useable variable types (int, float, …) and memory types (shared, global) dependent on compute capability of the graphics card!

Careful with Atomic Operations

- Prevents possible race-conditions

- However, threads put on hold!

- For example:
  - You should NOT compute a histogram using atomic operations
  - This will result in really bad performance, as many threads wait for exclusive access!

- Use atomic operation sparsely and only if really necessary

Textures

- Graphics Hardware is usually optimized for textures
- Data in global memory can be declared as texture
- Access in CUDA with tex2D function like in shaders
- This has several advantages:
  - Caching: 2D caching for increased speed
  - Potentially faster access: addresses computed by dedicated units
  - Normalized access if desired (access by coordinates ranging from 0 to 1)
  - Automatic float conversion of integer data, if desired
  - Linear filtering if desired
  - Out-of-range addressing like clamp or repeat
- But also disadvantages
  - Writing to textures not possible! (use surfaces instead)
- (No further details about textures in this lab…)

Execution

- Obviously, threads are executed in parallel

- Technically, threads are launched in groups of warps

    32 parallel threads form a single warp

    Threads are partitioned into warps

    All threads in a single warp execute the same operation

    → Branch divergence only happens within warps

    Warp scheduler selects a new warp to launch whenever ready

- Order of execution of blocks is not defined and can be arbitrary!

Performance Hints

- Remember memory hierarchy of speed:
- Registers > Local > Shared > Global

- Memory transfer host → device subject to
    - Bandwidth limitation
    - Latency

- Therefore, avoid unnecessary transfers!
    - Sometime better to perform something slower on host or device if transfer can be saved

Alternatives to CUDA

- CUDA
  - NVIDIA only
- OpenCL
  - all OSs and hardware (theoretically …); very similar to CUDA
- OpenGL Compute Shaders
  - GLSL like language (OpenGL >4.3); easy setup
- Direct3D Compute Shaders
  - Windows only; similar to OpenGL Compute Shaders
- DirectCompute
  - Windows only
- Metal
  - Apple only
- Vulkan
  - for 3D rendering applications

References

- Some Slides and Examples from Visual Computing Lecture
  - A. Grundhöfer and M. Grosse
- CUDA Developer Zone
  - http://developer.nvidia.com/category/zone/cuda-zone
- Official CUDA Dokumentation
  - http://docs.nvidia.com/cuda/index.html
  - Many guides, tutorials and samples
  - CUDA C Programming Guide
    - http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
  - NVIDIA CUDA Runtime API
    - http://docs.nvidia.com/cuda/cuda-runtime-api/index.html

Thanks!
Have fun with your CG-Projects.

Questions / Feedback: cg-lab@jku.at

Final Submission Deadline:
22.06.2021