Real-time Graphics

9. GPGPU

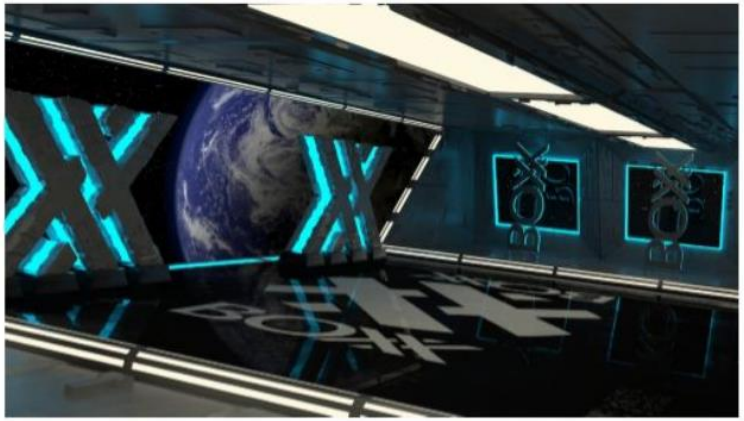
GPGPU

- GPU (Graphics Processing Unit)
  - Flexible and powerful processor
  - Programmability, precision, power
  - Parallel processing
- CPU
  - Increasing number of cores
  - Parallel processing
- GPGPU – general-purpose computation using power of GPU, parallel processor

GPGPU related to rendering

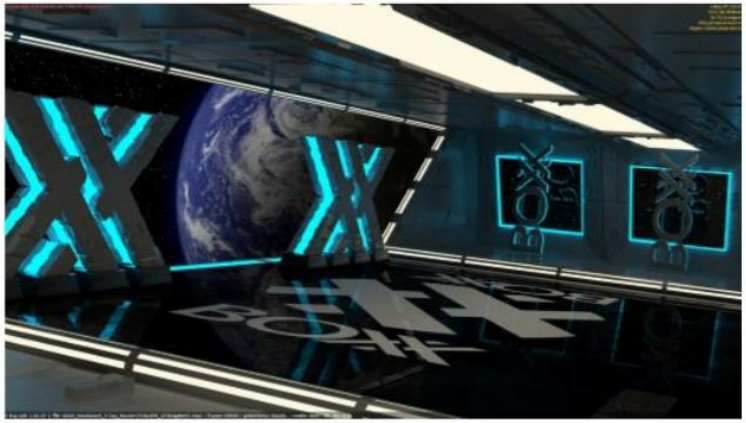
- Deformations
- Sorting - dynamic alpha-blended particles
- Collision detection
- Global illumination
  - Can be done via shaders, but requires hacking

Rendered with V-Ray Advanced CPU



3.4 GHz 8 core Intel® Xeon®
Image Quality = 11.35
Render Time = 19 minutes 11 seconds

Rendered with V-Ray RT GPU



High-end NVIDIA GPU with 2688 CUDA cores
Image Quality = 11.35
Render Time = 3 minutes 4 seconds

Serial vs Parallel processing

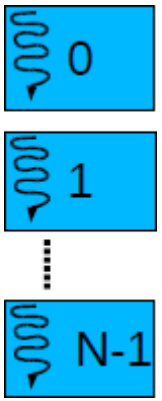
```
void process_in_serial()
{
    for (int i=0;i<N;i++)
        c[i] = a[i]+b[i];
}
```

```
void process_in_parallel()
{
    start_threads(N, thread);
}
void thread()
{
    int i = thread_ID;
    c[i] = a[i]+b[i];
}
```

c[0] = a[0]+b[0]

c[1] = a[1]+b[1]

c[N] = a[N]+b[N]

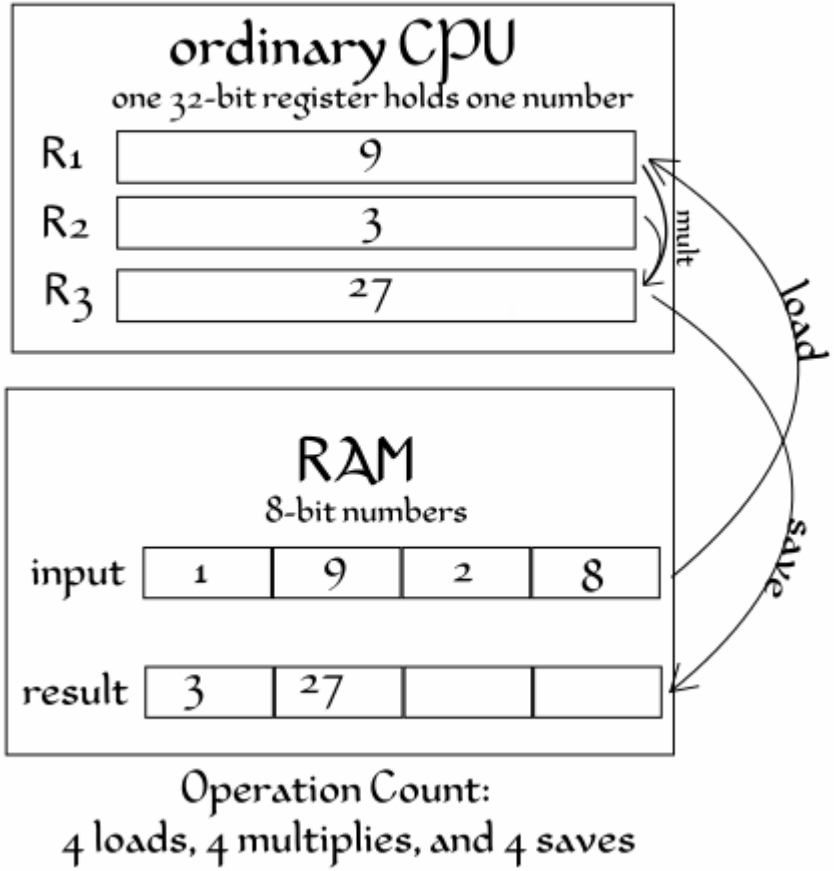


SIMD

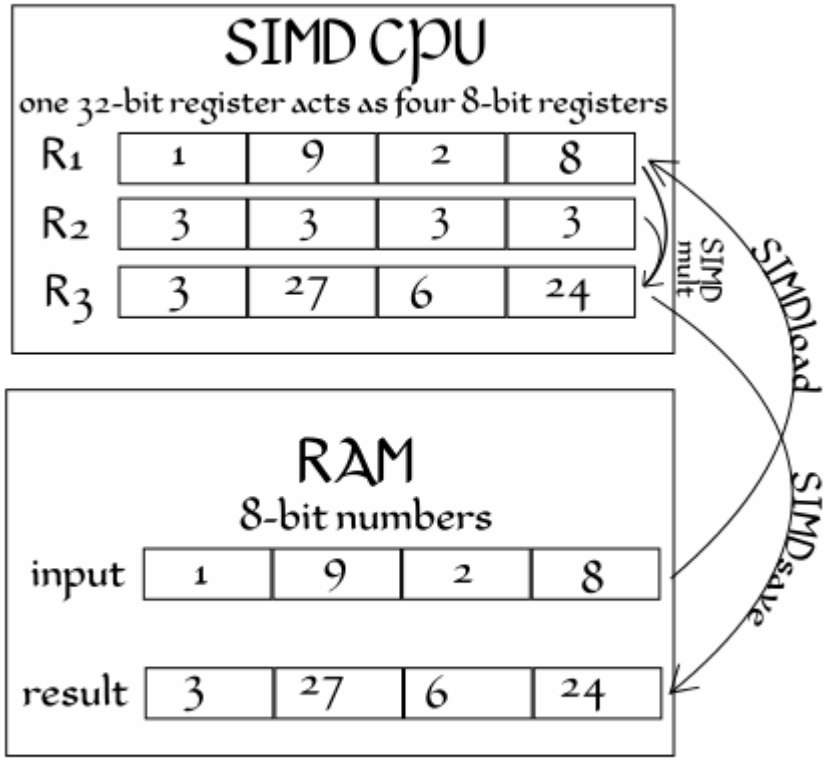
- Single Instruction Multiple Data
- Performing the same operation on multiple data points simultaneously
- Usage - data processing & compression, cryptography, image & video processing, 3D graphics
- CPU - special instructions + registers
- Intel - since 1997, MMX, SSE
- AMD – since 1998, 3DNow!
- IBM – PowerPC, Cell Processor (PS3)

SIMD

www.wikipedia.org



Operation Count:
4 loads, 4 multiplies, and 4 saves



Operation Count:
1 load, 1 multiply, and 1 save

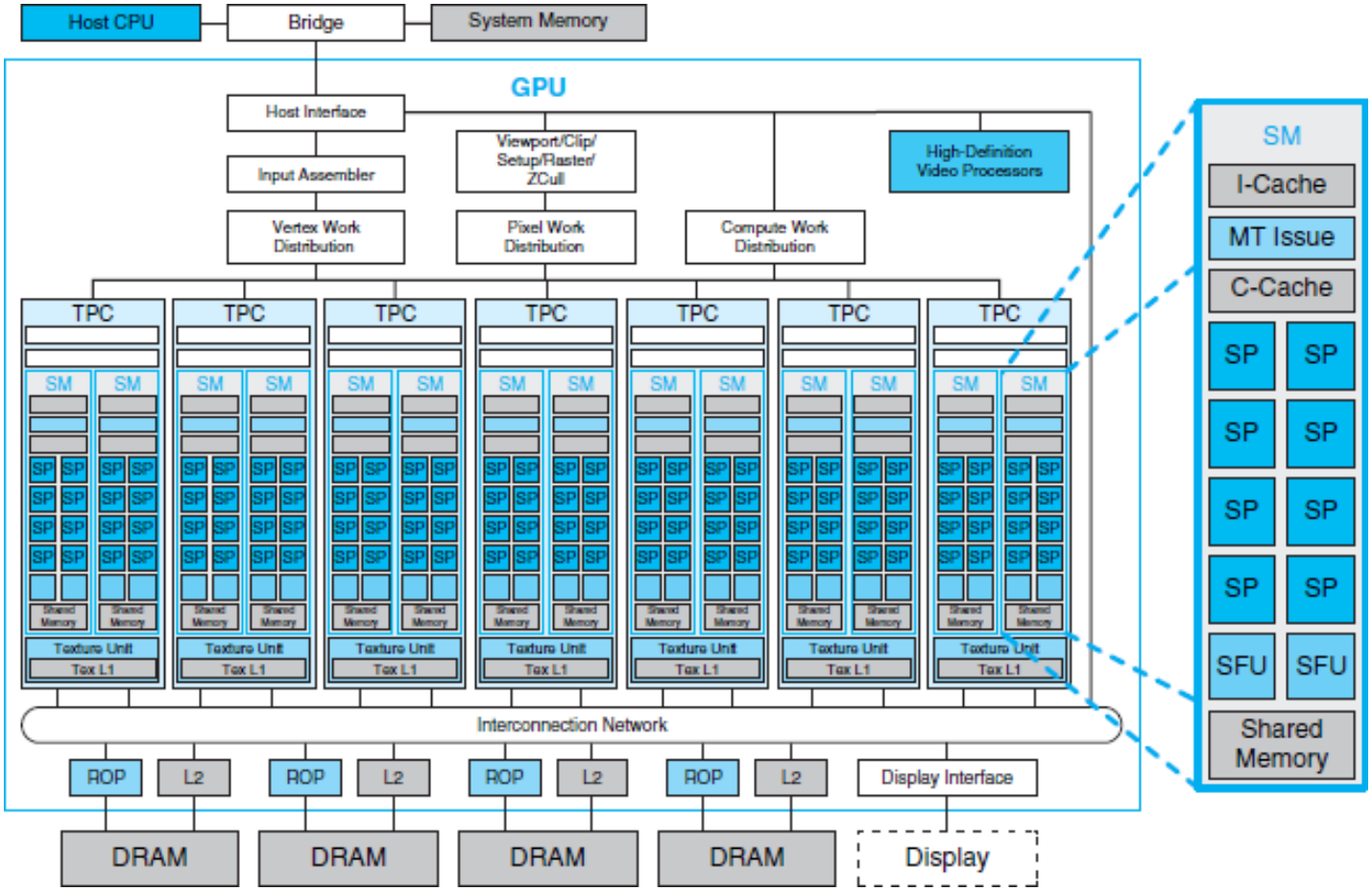
GPU history

- Primarily for transformation, rasterization and texturing
- Data types – SIMD processing
  - vertex – 4 x 32bit FP
  - fragment (pixel) – 4 x 8bit FP/integer
  - texel – 4 x 8bit integer
- Increasing number of processing units (cores, pipelines)
  - Separate pipelines for vertices and fragmets
  - Unified shaders (2006), 60+ cores
- Increasing programmability of pipelines (shader units)
  - Conditional instructions, loops
  - General-purpose programming via new technologies
- Support for 32bit FP texels and fragments

GPU architecture

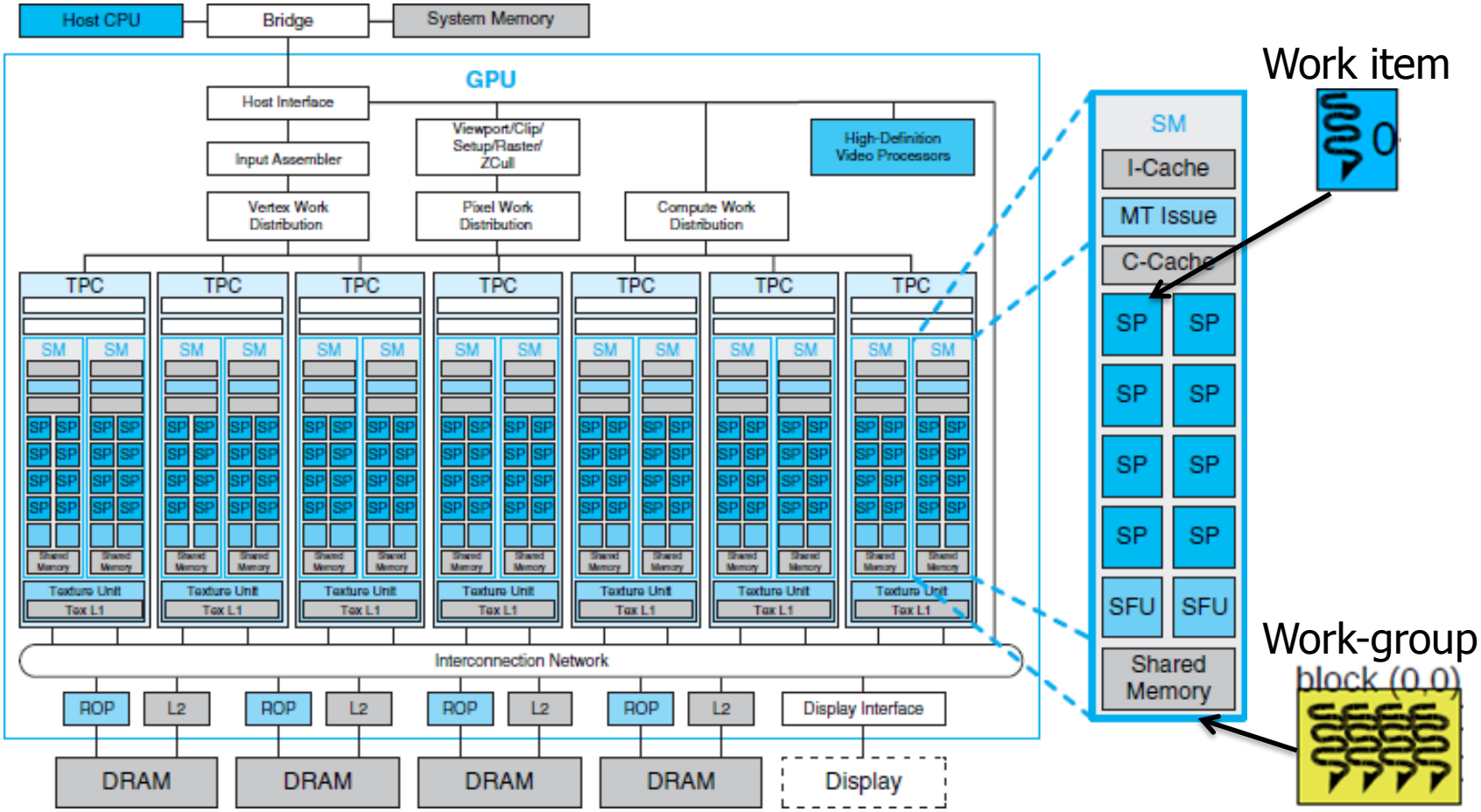
- GPU – multicore processor
- Basic computation unit – Core, Stream Core, Stream (Thread) Processor
- Array of cores - Streaming Multiprocessor, SIMD Engine, Compute Unit
  - Several levels of hierarchy
  - Shared memory for cores in this array
  - Special Function Units
  - Private registers, cahces
  - Scheduler

GPU architecture



NVIDIA GeForce 8800

GPU architecture



NVIDIA GeForce 8800

GPU devices

- Radeon R9 290X 4GB
  - Compute units: 44
  - Stream Processors: 2816

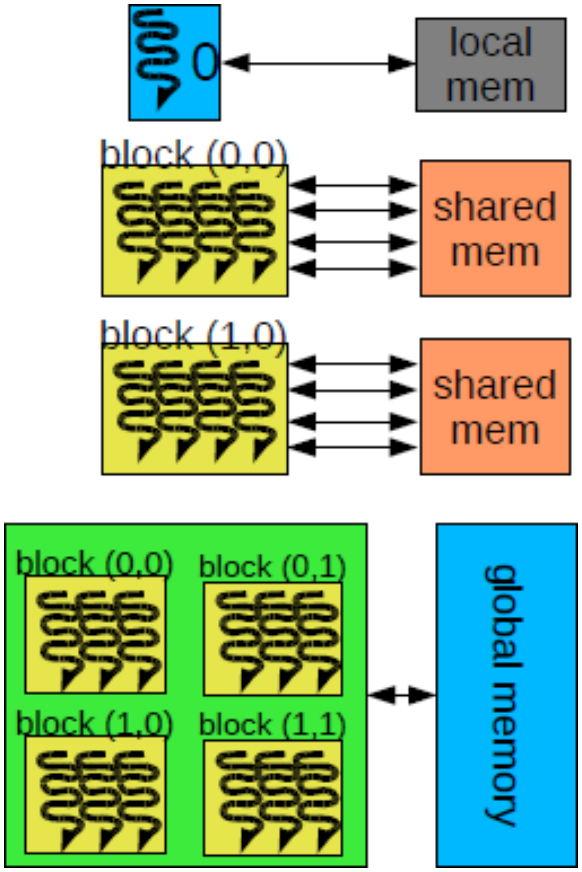
Work-group
block (0,0)

- Nvidia GeForce GTX Titan 6GB
  - Stream Processors: 2688

Work item

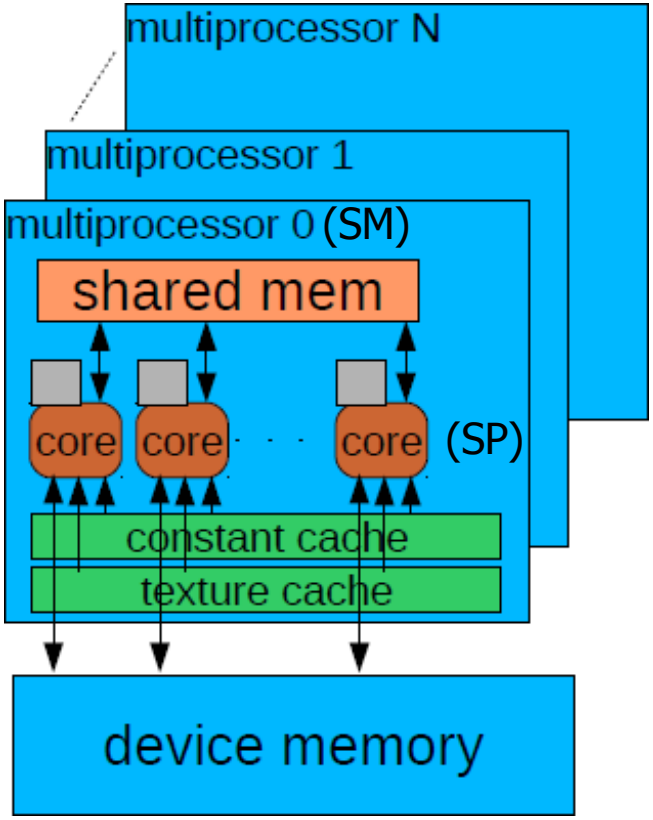
GPU computation model

- Thread, Work item (SP)
  - One running computation
  - Executed on cores
  - Local memory for one thread
  - Kernel – program running on device
- Work-group, Thread block (SM)
  - Group of several threads
  - Shared local per-block memory
  - Organized in 1,2,3 dimensional grids
  - Controlled by streaming multiprocessor
- Grid of blocks, work-groups per device
  - Organized in 1,2,3, N dimensional arrays



GPU memory model

- Global memory
  - Accessible by all threads
  - Slow, used with restrictions, bank conflicts
  - Per device - 512-… MB
- Constant memory
  - Read-only section of global memory
  - Per device = 64 KB - , cache per SM = 8 KB
- Local, Shared memory
  - Sharing data between work items (SP)
  - Per SM = 16 KB -
  - Fast, used with restriction, synchronization needed
- Registers, Private memory
  - per SM = 16384 -
  - Allocated for each thread separately, Fast

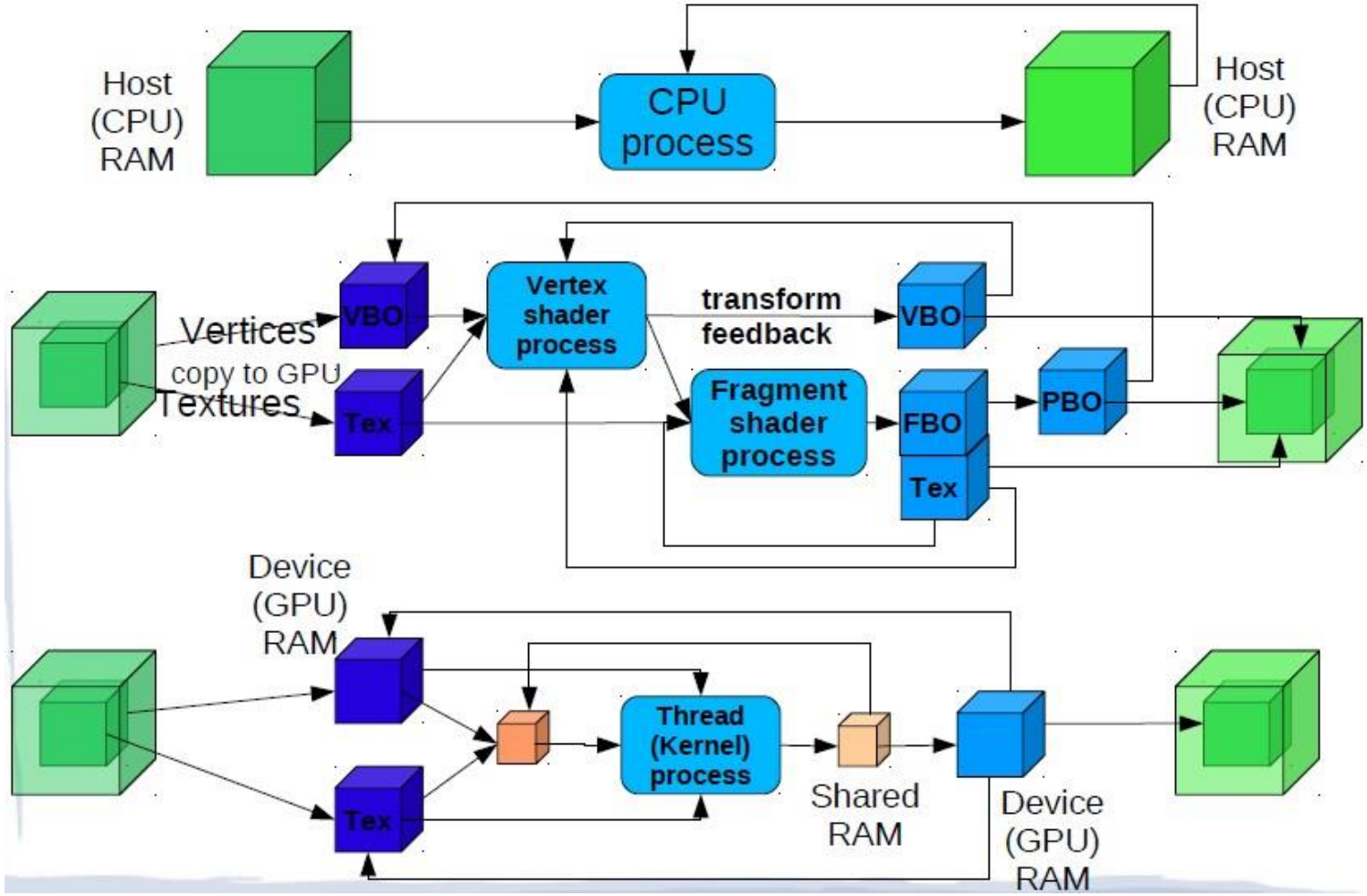


GPU programming

- **Graphics API**
- OpenGL (universal)
  - GLSL
  - Cg
  - ASM
  - Compute shaders
- DirectX (win32)
  - HLSL
  - Cg
  - ASM
  - DirectCompute

- **Computing languages**
- Nvidia CUDA (universal)
  - CUDA
  - PTX ISA
- ATI Stream (universal)
  - Brook+
  - AMD IL
  - ISA
- OpenCL (universal)
  - all devices

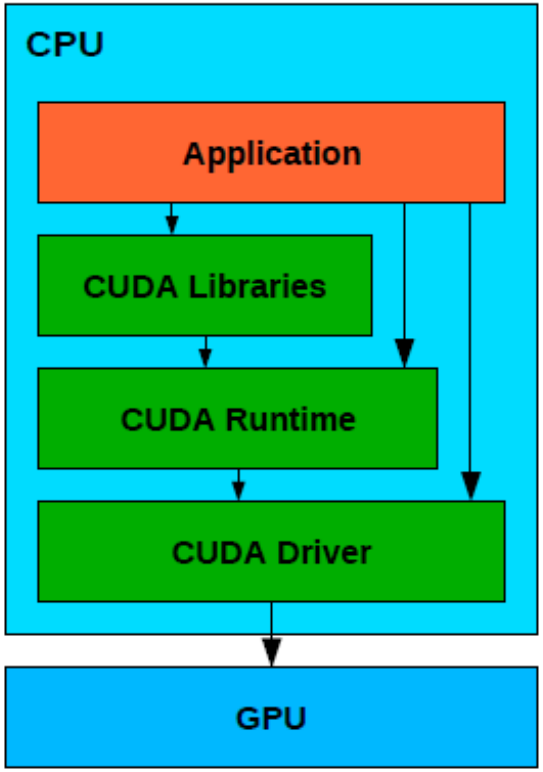
CPU/GPU programming

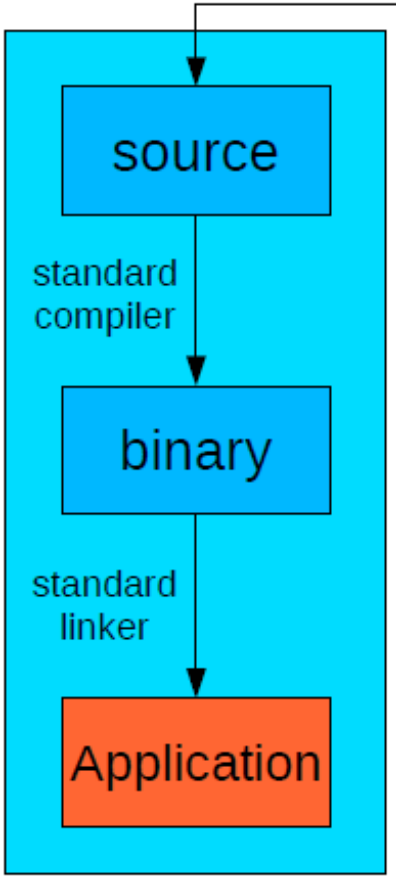


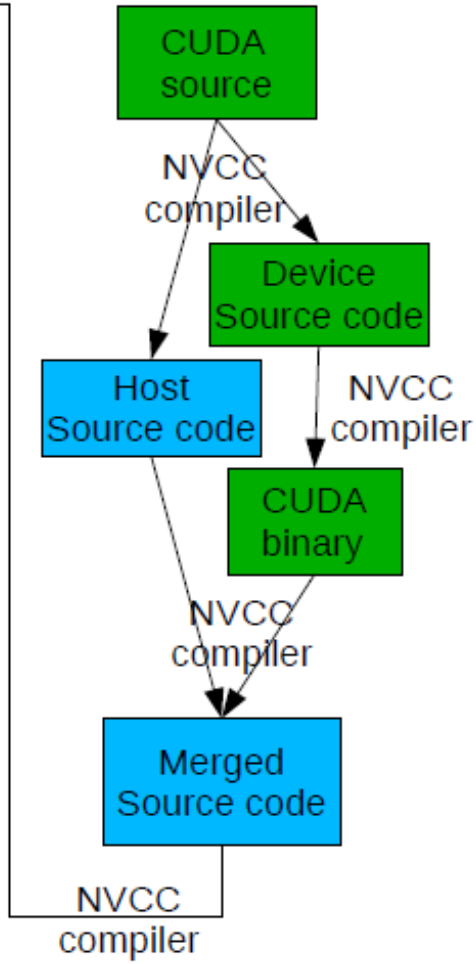
CUDA

- Compute Unified Device Architecture
- parallel computing platform and application programming interface
- Restricted to NVIDIA hardware
- Exploiting GPU capabilities more than graphics API
  - unlimited scatter write
  - direct access to memory
  - utilize shared memory
- Kernel code based on C
- API for loading kernel, work with arrays, managing devices
- http://code.msdn.microsoft.com/windowsdesktop/NVIDIA-GPU-Architecture-45c11e6d

CUDA


CPU
Application
CUDA Libraries
CUDA Runtime
CUDA Driver
GPU


source
standard compiler
binary
standard linker
Application


CUDA source
NVCC compiler
Device Source code
Host Source code
NVCC compiler
CUDA binary
NVCC compiler
Merged Source code
NVCC compiler

CUDA C/C++ extension

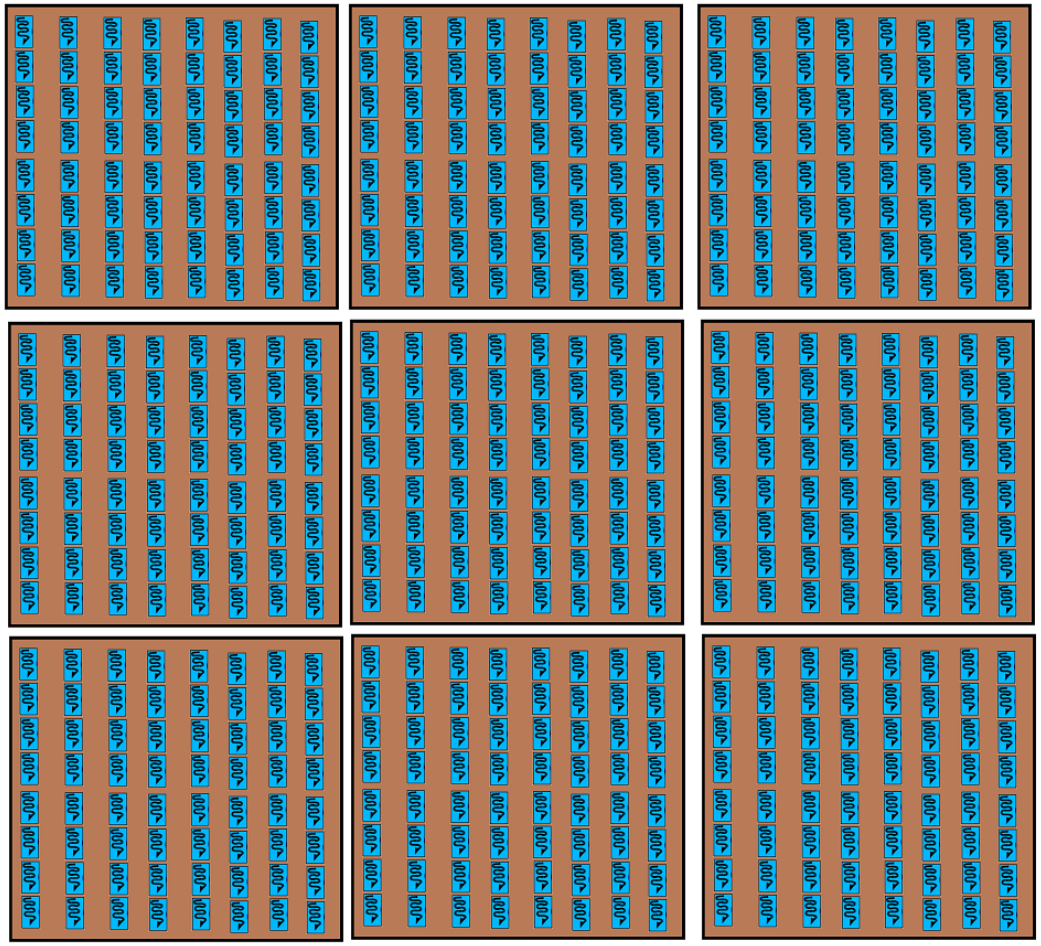
- Device vs Host code
  - Function qualifiers: __device__, __host__, __global__
  - Variable qualifiers: __device__, __constant__, __shared__
- Built-in variables
  - threadIdx – coordinates of thread within a block
  - blockDim – number of threads in a block
  - blockIdx – coordinates of thread's block within grid of blocks
  - gridDim – number of blocks in grid

Example - Host part

- Computing gradients in 2D one component image

```
float* d_idata; // device (GPU) input data
cudaMalloc( (void**) &d_idata, width*height*sizeof(float));

float2* d_odata; // device (GPU) output data
cudaMalloc( (void**) &d_odata, width*height*sizeof(float2));

float* h_idata;
int width, height;
loadImg("input.tga", &width, &height, h_idata); // fill source host data by CPU
cudaMemcpy(d_idata, h_idata, width * height * 4, cudaMemcpyHostToDevice);

dim3 threads( 8, 8, 1); // threads per block
dim3 grid( 1 + (width-1) / 8, 1 + (height-1) / 8, 1); // blocks in grid
compute_gradient<<< grid, threads >>>(d_idata, d_odata, width, height);
cudaThreadSynchronize(); // wait for result

float* h_odata = new float[2 * width * height];
cudaMemcpy(h_odata, d_odata, 2 * width * height * 4, cudaMemcpyDeviceToHost);
saveImg("output.tga", h_odata);
```

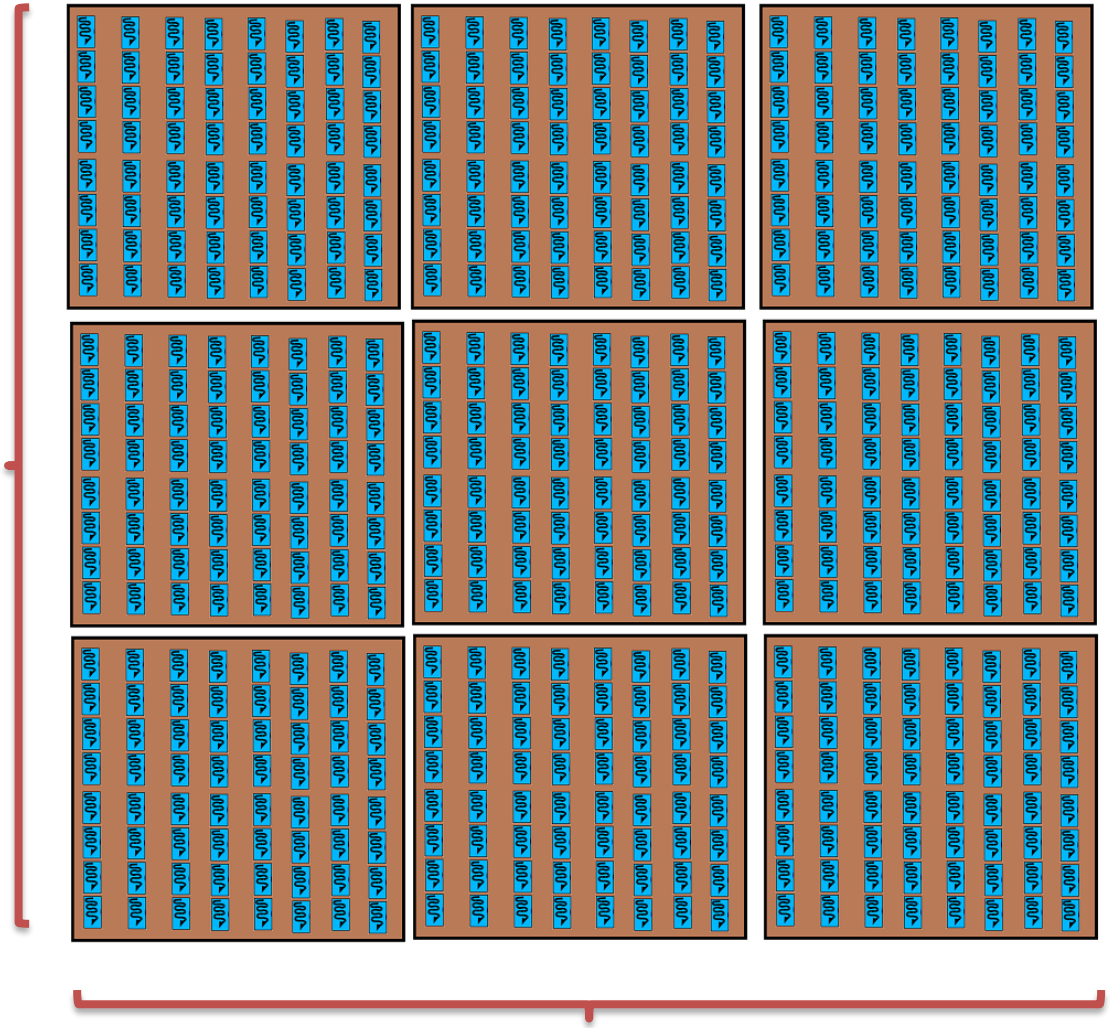
CUDA



width = 24
height = 24

```
// threads per block
dim3 threads( 8, 8, 1);

// blocks in grid
dim3 grid( 1 + (width-1) / 8,
           1 + (height-1) / 8, 1);
```

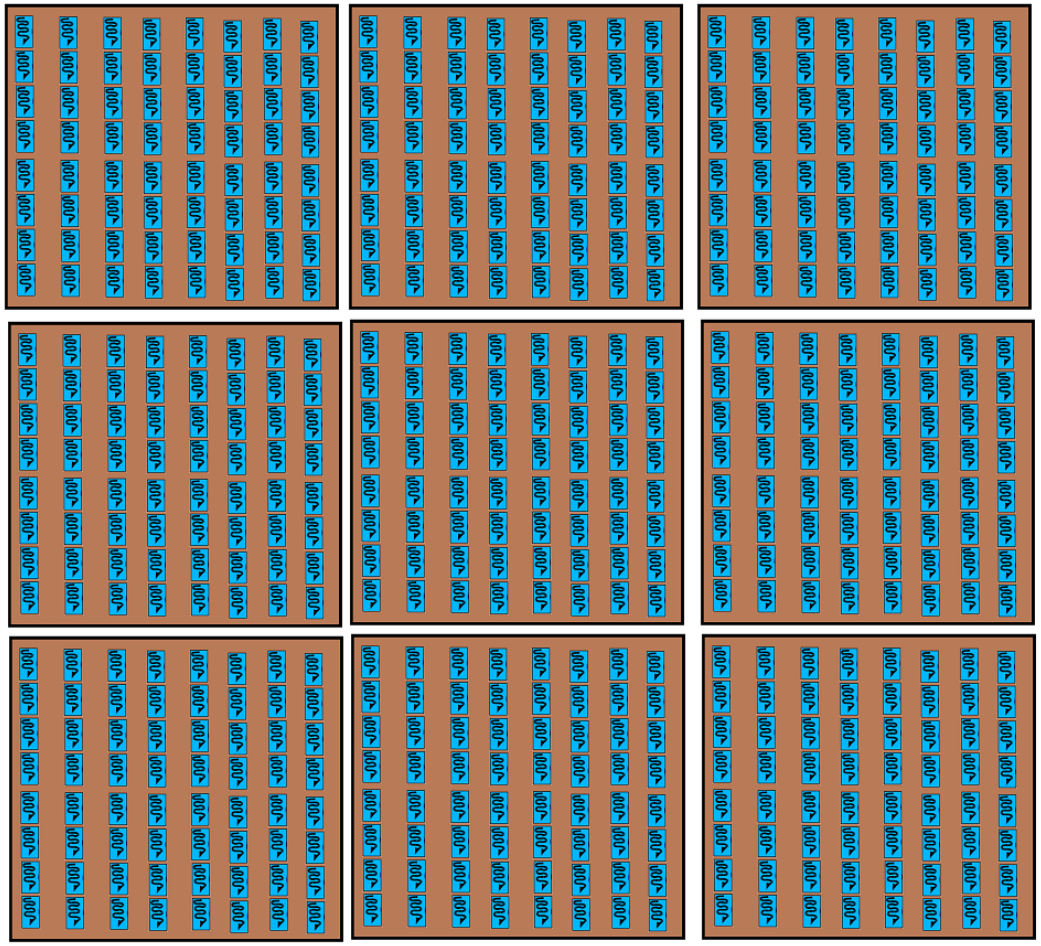
CUDA



width = 24
height = 24

```
// threads per block
dim3 threads( 8, 8, 1);

// blocks in grid
dim3 grid( 1 + (24-1) / 8,
           1 + (24-1) / 8, 1);
```

CUDA

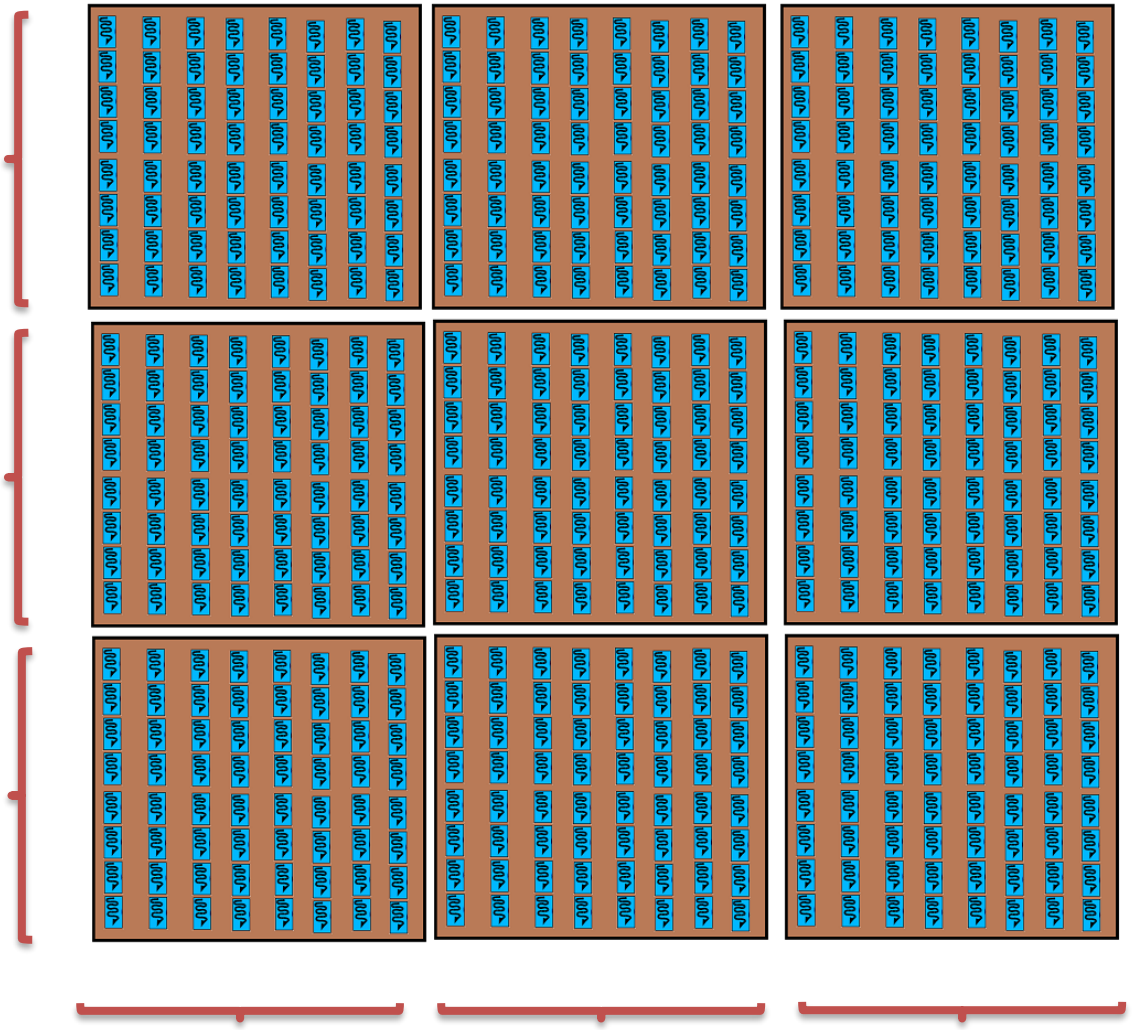


width = 24
height = 24

```
// threads per block
dim3 threads( 8, 8, 1);

// blocks in grid
dim3 grid( 1 + (24-1) / 8,
           1 + (24-1) / 8, 1);
```

$1 + \lfloor 23 / 8 \rfloor = 1 + \lfloor 2.875 \rfloor = 3$

CUDA



width = 24
height = 24

// threads per block
dim3 threads(8, 8, 1);

// blocks in grid
dim3 grid(3, 3, 1);

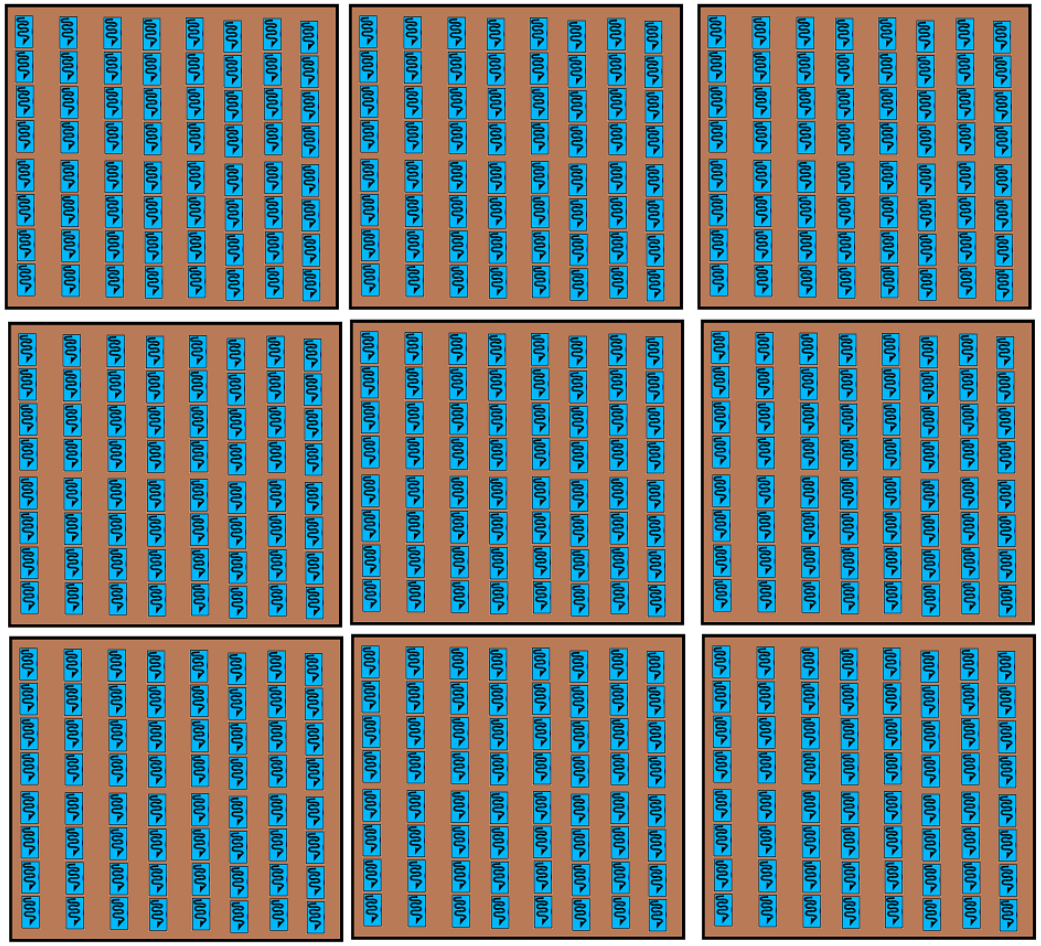
$1 + \lfloor 23 / 8 \rfloor = 1 + \lfloor 2.875 \rfloor = 3$

Example - Device part

```
__global__void compute_gradient(float* g_idata, float2* g_odata, const int width, const int height)
{
   const unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
   const unsigned int j = blockIdx.y * blockDim.y + threadIdx.y;
   if (i >= width || j >= height)
      return;
   float value = g_idata[i + j * width];
   float prev_x = value;
   if (i > 0)
      prev_x = g_idata[i - 1 + j * width];
   float next_x = value;
   if (i < width-1)
      next_x = g_idata[i + 1 + j * width];
   float prev_y = value;
   if (j > 0)
      prev_y = g_idata[i + (j – 1) * width];
   float next_y = value;
   if (j < height -1)
      next_y = g_idata[i + (j + 1) * width];
   float z_dx = next_x – prev_x;
   float z_dy = next_y – prev_y;
   g_odata [i + j * width] = make_float2(z_dx, z_dy );
}
```

$$\left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right)$$

CUDA

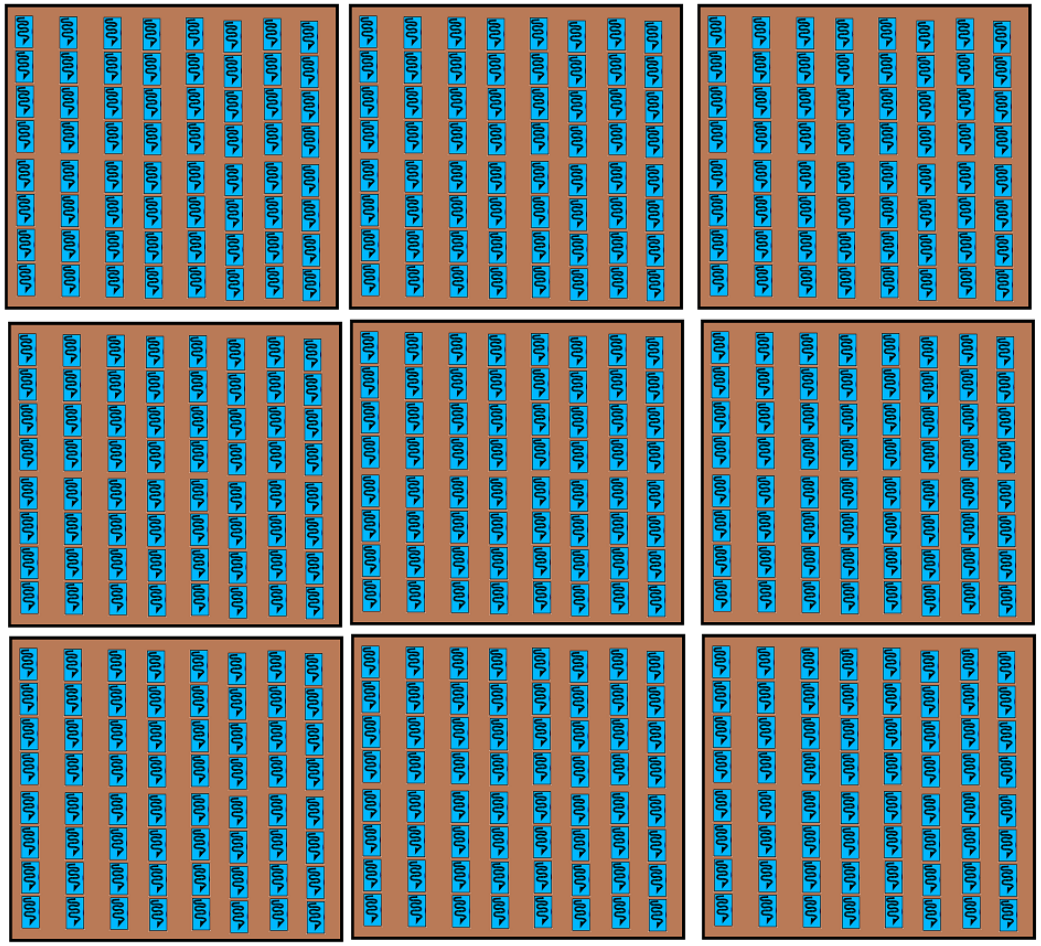


?

width = 20
height = 20

```
// threads per block
dim3 threads( 8, 8, 1);

// blocks in grid
dim3 grid( ?, ?, 1);
```

CUDA



?

width = 20
height = 20

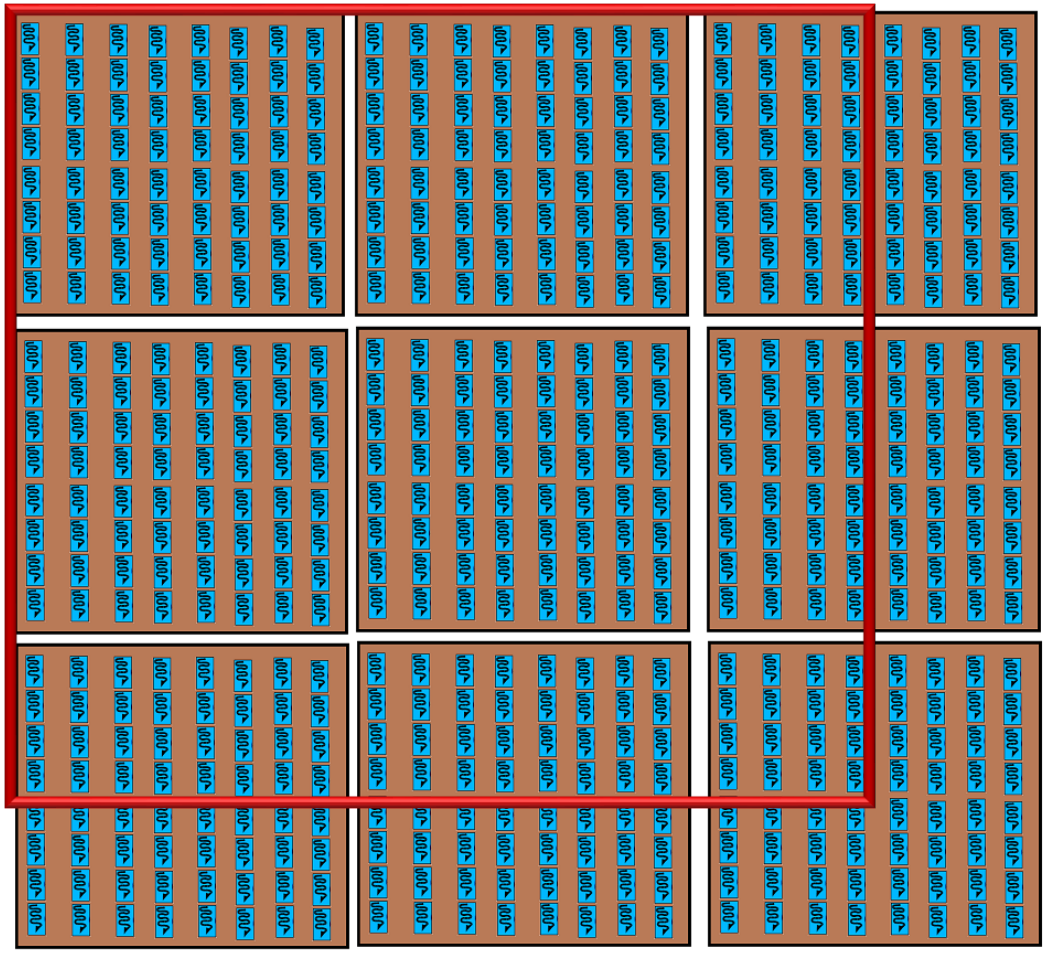
```
// threads per block
dim3 threads( 8, 8, 1);

// blocks in grid
dim3 grid( 1 + (width-1) / 8,
           1 + (height-1) / 8, 1);
```

$1 + \lfloor 20 / 8 \rfloor = 1 + \lfloor 2.5 \rfloor = 3$

Example - Device part

```
__global__void compute_gradient(float* g_idata, float2* g_odata, const int width, const int height)
{
   const unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
   const unsigned int j = blockIdx.y * blockDim.y + threadIdx.y;
   if (i >= width || j >= height)
      return;
   float value = g_idata[i + j * width];
   float prev_x = value;
   if (i > 0)
      prev_x = g_idata[i - 1 + j * width];
   float next_x = value;
   if (i < width-1)
      next_x = g_idata[i + 1 + j * width];
   float prev_y = value;
   if (j > 0)
      prev_y = g_idata[i + (j – 1) * width];
   float next_y = value;
   if (j < height -1)
      next_y = g_idata[i + (j + 1) * width];
   float z_dx = next_x – prev_x;
   float z_dy = next_y – prev_y;
   g_odata [i + j * width] = make_float2(z_dx, z_dy );
}
```

CUDA



width = 20
height = 20

// threads per block
dim3 threads(8, 8, 1);

// blocks in grid
dim3 grid(3, 3, 1);

$1 + \lfloor 20 / 8 \rfloor = 1 + \lfloor 2.5 \rfloor = 3$

Example - Device part

```
__global__void compute_gradient(float* g_idata, float2* g_odata, const int width, const int height)
{
   const unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
   const unsigned int j = blockIdx.y * blockDim.y + threadIdx.y;
   if (i >= width || j >= height)
      return;
   float value = g_idata[i + j * width];
   float prev_x = value;
   if (i > 0)
      prev_x = g_idata[i - 1 + j * width];
   float next_x = value;
   if (i < width-1)
      next_x = g_idata[i + 1 + j * width];
   float prev_y = value;
   if (j > 0)
      prev_y = g_idata[i + (j – 1) * width];
   float next_y = value;
   if (j < height -1)
      next_y = g_idata[i + (j + 1) * width];
   float z_dx = next_x – prev_x;
   float z_dy = next_y – prev_y;
   g_odata [i + j * width] = make_float2(z_dx, z_dy );
}
```

width = 20
i = 3
j = 2

	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
j																					
0		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1		20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
2		40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
3		60	61	62	63	64	...														
4		...																			
.																					
.																					
.																					

Example - Device part

```
__global__void compute_gradient(float* g_idata, float2* g_odata, const int width, const int height)
{
   const unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
   const unsigned int j = blockIdx.y * blockDim.y + threadIdx.y;
   if (i >= width || j >= height)
      return;
   float value = g_idata[i + j * width];
   float prev_x = value;
   if (i > 0)
      prev_x = g_idata[i - 1 + j * width];
   float next_x = value;
   if (i < width-1)
      next_x = g_idata[i + 1 + j * width];
   float prev_y = value;
   if (j > 0)
      prev_y = g_idata[i + (j – 1) * width];
   float next_y = value;
   if (j < height -1)
      next_y = g_idata[i + (j + 1) * width];
   float z_dx = next_x – prev_x;
   float z_dy = next_y – prev_y;
   g_odata [i + j * width] = make_float2(z_dx, z_dy );
}
```

width = 20
i = 3
j = 2

	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
j	0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	1	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	2	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
	3	60	61	62	63	64	…														
	4	…																			
	.																				
	.																				
	.																				

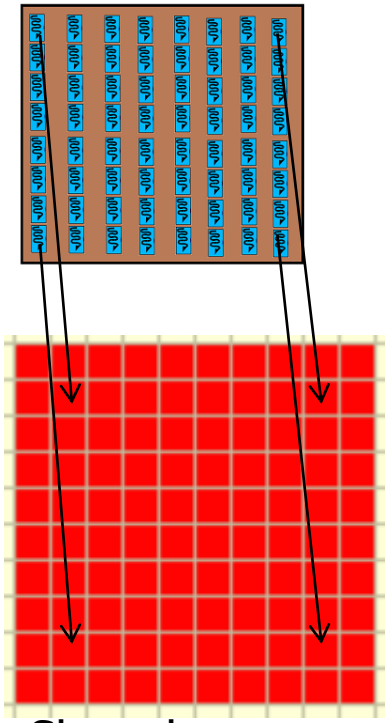
$$2(20) + 3 = 43$$

Example – Shared memory

```
__global__ void compute_gradient(float* g_idata, float2* g_odata, const int
width, const int height){
   __shared__ float s_data[10][10];

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

   if (i >= width || j >= height)
      return;

   float value = g_idata[i + j * width];
   s_data[threadIdx.x+1] [threadIdx.y+1] = value;
   __syncthreads();

   float prev_x = s_idata[threadIdx.x+1 - 1] [threadIdx.y+1];
   float next_x = s_idata[threadIdx.x+1 + 1] [threadIdx.y+1];
   float prev_y = s_idata[threadIdx.x+1] [threadIdx.y+1 - 1];
   float next_y = s_idata[threadIdx.x+1] [threadIdx.y+1 + 1];
   float z_dx = next_x – prev_x;
   float z_dy = next_y – prev_y;
   g_odata [i + j * width] = float2(z_dx, z_dy);

   }
```

Example – Shared memory

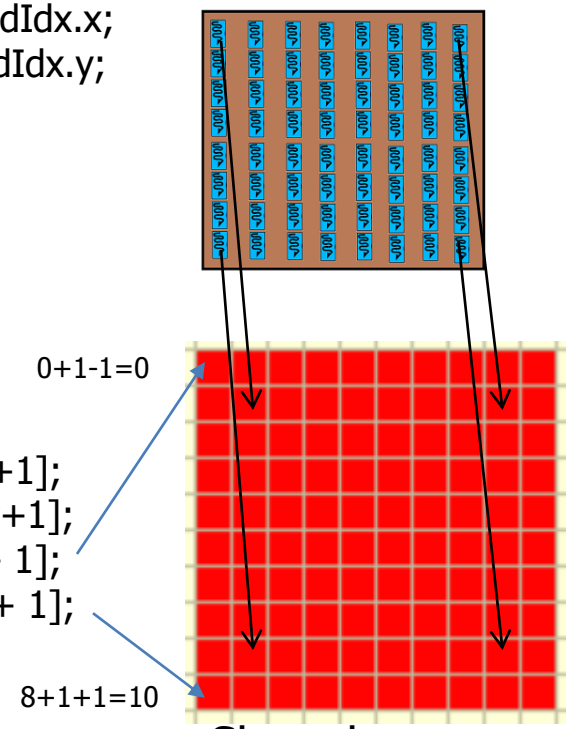
```
__global__ void compute_gradient(float* g_idata, float2* g_odata, const int
width, const int height){
    __shared__ float s_data[10][10];

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

    if (i >= width || j >= height)
        return;

    float value = g_idata[i + j * width];
    s_data[threadIdx.x+1] [threadIdx.y+1] = value;
    __syncthreads();

    float prev_x = s_idata[threadIdx.x+1 - 1] [threadIdx.y+1];
    float next_x = s_idata[threadIdx.x+1 + 1] [threadIdx.y+1];
    float prev_y = s_idata[threadIdx.x+1] [threadIdx.y+1 - 1];
    float next_y = s_idata[threadIdx.x+1] [threadIdx.y+1 + 1];
    float z_dx = next_x – prev_x;
    float z_dy = next_y – prev_y;
    g_odata [i + j * width] = float2(z_dx, z_dy);
}
```



Shared memory

Example – Shared memory

```
__global__ void compute_gradient(float* g_idata, float2* g_odata, const int
width, const int height){
    __shared__ float s_data[10][10];

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

    if (i >= width || j >= height)
        return;

    float value = g_idata[i + j * width];
    s_data[threadIdx.x+1] [threadIdx.y+1] = value;
    __syncthreads();

    float prev_x = s_idata[threadIdx.x+1 - 1] [threadIdx.y+1];
    float next_x = s_idata[threadIdx.x+1 + 1] [threadIdx.y+1];
    float prev_y = s_idata[threadIdx.x+1] [threadIdx.y+1 - 1];
    float next_y = s_idata[threadIdx.x+1] [threadIdx.y+1 + 1];
    float z_dx = next_x – prev_x;
    float z_dy = next_y – prev_y;
    g_odata [i + j * width] = float2(z_dx, z_dy);
}
```

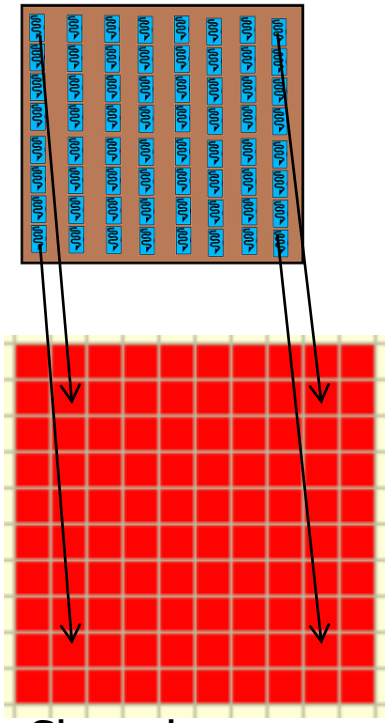


Shared memory

10×10 to avoid index out of bounds

Example – Shared memory

```
__global__ void compute_gradient(float* g_idata, float2* g_odata, const int
width, const int height){
   __shared__ float s_data[10][10];

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

   if (i >= width || j >= height)
      return;

   float value = g_idata[i + j * width];
   s_data[threadIdx.x+1] [threadIdx.y+1] = value;
   __syncthreads();

   float prev_x = s_idata[threadIdx.x+1 - 1] [threadIdx.y+1];
   float next_x = s_idata[threadIdx.x+1 + 1] [threadIdx.y+1];
   float prev_y = s_idata[threadIdx.x+1] [threadIdx.y+1 - 1];
   float next_y = s_idata[threadIdx.x+1] [threadIdx.y+1 + 1];
   float z_dx = next_x – prev_x;
   float z_dy = next_y – prev_y;
   g_odata [i + j * width] = float2(z_dx, z_dy);
}
```



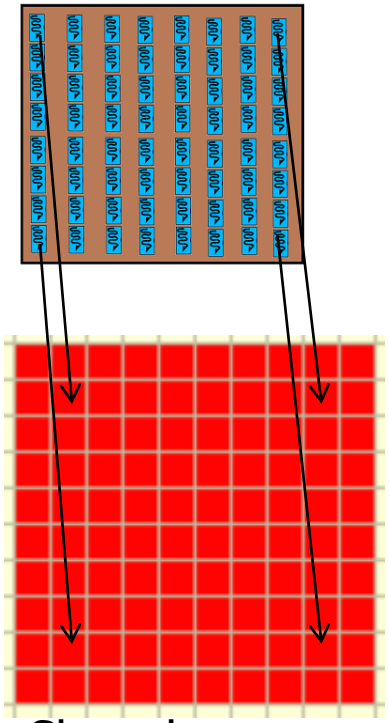
Shared memory

Example – Shared memory

```
__global__ void compute_gradient(float* g_idata, float2* g_odata, const int
width, const int height){
   __shared__ float s_data[10][10];

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

   if (i >= width || j >= height)
      return;

   float value = g_idata[i + j * width];
   s_data[threadIdx.x+1] [threadIdx.y+1] = value;
   __syncthreads();
```

some threads will not reach

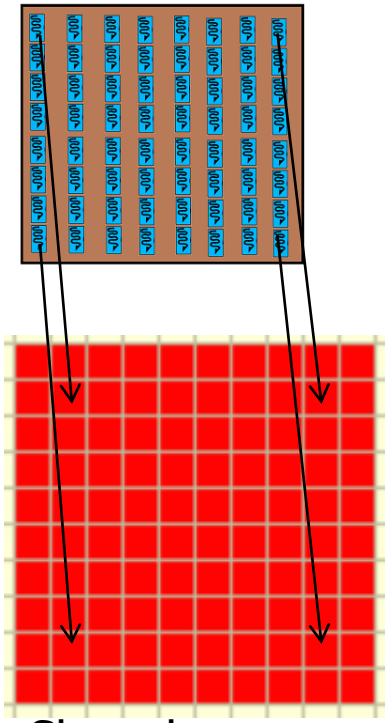
```
   float prev_x = s_idata[threadIdx.x+1 - 1] [threadIdx.y+1];
   float next_x = s_idata[threadIdx.x+1 + 1] [threadIdx.y+1];
   float prev_y = s_idata[threadIdx.x+1] [threadIdx.y+1 - 1];
   float next_y = s_idata[threadIdx.x+1] [threadIdx.y+1 + 1];
   float z_dx = next_x – prev_x;
   float z_dy = next_y – prev_y;
   g_odata [i + j * width] = float2(z_dx, z_dy);
}
```



Shared memory

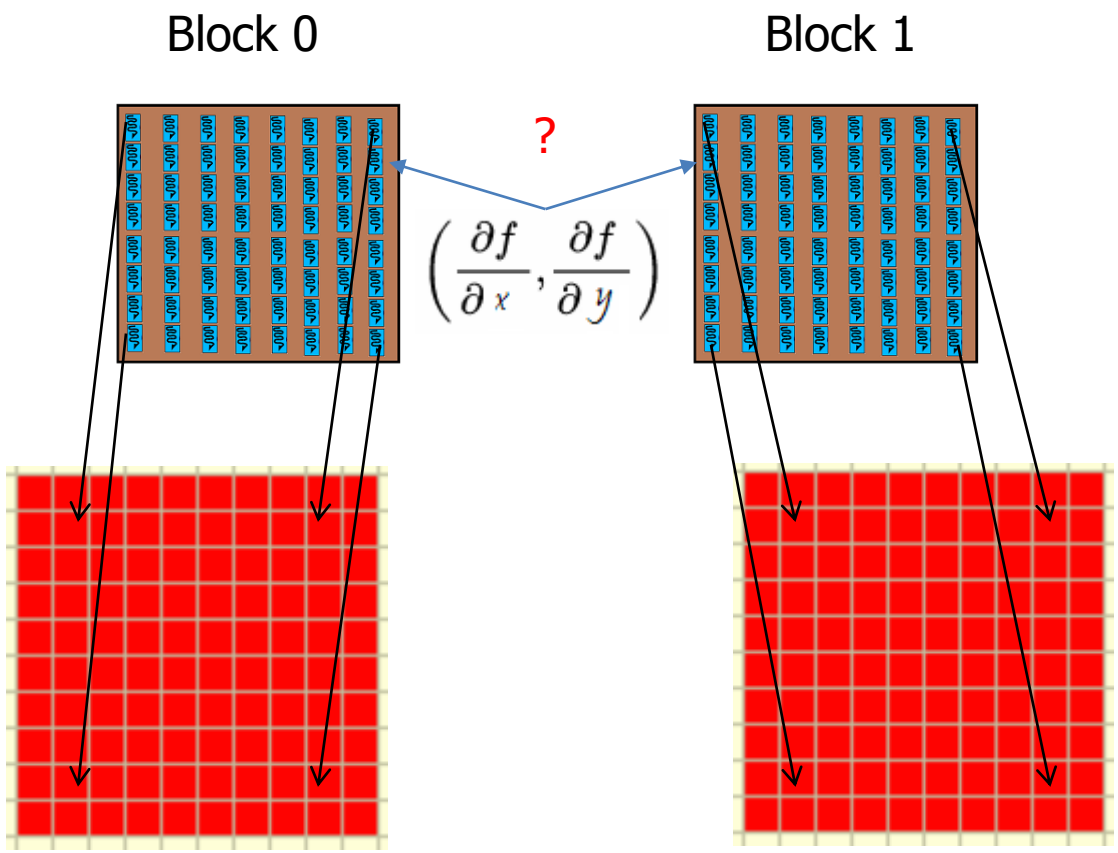
Example – Shared memory

```
__global__ void compute_gradient(float* g_idata, float2* g_odata, const int
width, const int height){
   __shared__ float s_data[10][10];

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

   if (i >= width || j >= height){
      return;
      __syncthreads();
      }

   float value = g_idata[i + j * width];
   s_data[threadIdx.x+1] [threadIdx.y+1] = value;
   __syncthreads();

   float prev_x = s_idata[threadIdx.x+1 - 1] [threadIdx.y+1];
   float next_x = s_idata[threadIdx.x+1 + 1] [threadIdx.y+1];
   float prev_y = s_idata[threadIdx.x+1] [threadIdx.y+1 - 1];
   float next_y = s_idata[threadIdx.x+1] [threadIdx.y+1 + 1];
   float z_dx = next_x – prev_x;
   float z_dy = next_y – prev_y;
   g_odata [i + j * width] = float2(z_dx, z_dy);
}
```



Shared memory

Example – Shared memory



OpenCL

- Open standardized library for parallel computation
- https://www.khronos.org/opencl/
- For use, drivers and SDKs must be installed
- http://developer.amd.com/tools-and-sdks/opencl-zone/opencl-tools-sdks/
- https://developer.nvidia.com/opencl
- Language based on C,C++
- Many similarities with CUDA

OpenCL

- Host part - opencl.codeplex.com/wikipage?title=OpenCL Tutorials - 1
- Getting platforms = host+collection of devices
- Getting devices for given platform
- Create context for devices, defines entire OpenCL environment
- Fill command queue with kernels that will be executed on device
- Create memory buffers and fill it with data
- Prepare, compile and build programs
- Create kernel from program, set its arguments and launch it
- Wait for completition and read data

OpenCL kernel

```
__kernel  void compute_gradient(__global const float* g_idata,___global float2* g_odata, const int width,
      const int height)
{
   __local float s_data[10][10];
   const unsigned int i = get_global_id(0); //get_group_id(0) * get_local_size(0) + get_local_id(0);
   const unsigned int j = get_global_id(1); // get_group_id(1) * get_local_size(1) + get_local_id(1);
   if (i >= width || j >= height)
      return;

   float value = g_idata[i + j * width];
   s_data[get_local_id(0)+1] [get_local_id(1)+1] = value;
   barrier(CLK_LOCAL_MEM_FENCE);

   float prev_x = s_idata[get_local_id(0)+1 - 1] [get_local_id(1)+1];
   float next_x = s_idata[get_local_id(0)+1 + 1] [get_local_id(1)+1];
   float prev_y = s_idata[get_local_id(0)+1] [get_local_id(1)+1 - 1];
   float next_y = s_idata[get_local_id(0)+1] [get_local_id(1)+1 + 1];
   float z_dx = next_x – prev_x;
   float z_dy = next_y – prev_y;
   g_odata [i + j * width] = float2(z_dx, z_dy);
}
```

OpenCL & OpenGL

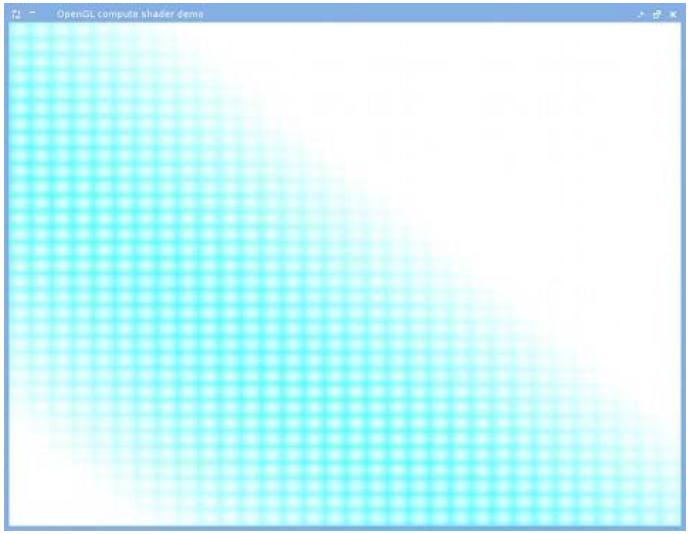
- Possibility to share resources, buffers and textures between OpenCL and OpenGL contexts
- Ping-pong between contexts – problem!
- Based on optimalization in drivers
- Or use compute shaders if possible

Compute shaders

- General purpose computation inside OpenGL context
- Direct access to OpenGL resources
- Not part of rendering pipeline – one function to execute shader
- Access to work-items and groups indices
- Access to shared, local memory and synchronization
- Using well-known API for loading shaders, managing buffers and textures
- Using GLSL

Compute shader example

- http://wili.cc/blog/opengl-cs.html




```
#version 430
uniform float roll;
uniform image2D destTex;
layout (local_size_x = 16, local_size_y = 16) in;
void main()
{
   ivec2 storePos = ivec2(gl_GlobalInvocationID.xy);
   float localCoef = length(vec2(ivec2(gl_LocalInvocationID.xy)-8)/8.0);
   float globalCoef = sin(float(gl_WorkGroupID.x+gl_WorkGroupID.y)*0.1 + roll)*0.5;
   imageStore(destTex, storePos, vec4(1.0-globalCoef*localCoef, 0.0, 0.0, 0.0));
}
```

Questions?