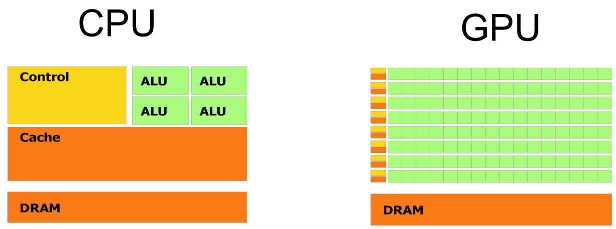
CUDA – Recap and Higher Dimension Grids

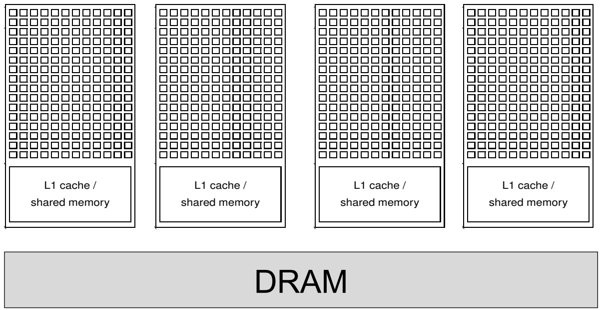
Stefano Markidis

KTH Royal Institute of Technology

Recap - What is a GPU?

- A specialized processor initially designed for graphics-like workload (videogames, video processing and CAD)
    - **Lots of cores**, **fewer control units**: very good in compute-heavy applications with little synchronization
- Now present in several supercomputers
    - **Power efficiency**: lot of parallelism but lower clock frequency
- GPU consists of one or more **SMs**, each one comprising **several cores (K80 almost 5k cores!)**





Recap - What is CUDA?

It is an extension of the C language that provide basic mechanisms to:

- Create allocate variable on GPU memory

Question: Which CUDA function?

- Move data from CPU to GPU memory and vice-versa

Question: Which CUDA function?

- Define kernel and launch a kernel

Question: Which qualifier I have to use? What is the difference between a kernel and a function.

- Synchronize threads

Question: Which CUDA function?

Recap - Lab

- `HelloWorld` in CUDA

Problem: not printing because of the asynchronous nature of the kernel launch

- `saxpy` in CUDA

Problem: `ARRAY_SIZE` was not a multiple of `BLOCK_SIZE`

Problem: create variable on GPU and move data to/GPU.

Easy to get it wrong:

In C, **the size of the data** to be created or moved is **in byte** (Fortran the size is the number of array elements)

How do I choose `TPB` or execution configuration?

To choose the specific execution configuration that will produce the best performance involve both art and science

- To choose **some multiple of 32 is reasonable** since it matches up somehow with the number of **CUDA cores in an SM**
- There are limits: a single block **cannot contain more than 1,024 threads**
- For large problems, reasonable to test are 128, 256 and 512

Careful with Integer Arithmetic!

The kernel execution configuration is specified so that each block has `TPB` threads, and there are `N/TPB` blocks.

1D Computational Grid

Problem: What happens if `N = 65` ?

We get 65/32 = **2 blocks of 32 threads**. In this case, **the last entry in the array would not get computed** because there is no thread with the corresponding index.

The simple trick is to change the number of blocks as `(N+TPB-1)/TPB` to **ensure that the number of blocks is rounded up**.

Back to CUDA – CUDA Vector Types

CUDA extends the standard C data types, like `int` and `float`, to be vector with 2, 3 and 4 components, like `int2`, `int3`, `int4`, `float2`, `float3` and `float4`. Other vector types are also supported.

For example, you can declare an integer vector `d` with three components and initialize with 128, 1 and 1 element in the x, y and z direction:

```
int3 d = int3(128, 1, 1);
```

Question: does this look reminiscent of something you saw in the lab?

CUDA Vector types

Vector types CUDA extends the standard C data types of length up to 4.

```
float4 f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
```

Individual components are accessed with the **suffixes** **`.x`**, **`.y`**, **`.z`**, **and** **`.w`**. Accessing components beyond those declared for the vector type is an error.

```
float3 pos;
pos.z = 1.0f; // is legal
pos.w = 1.0f; // is illegal
```

CUDA dim3 type for Dimension Variables

The `dim3` type is equivalent to `uint3` with unspecified entries set to 1.

CUDA uses the vector type `dim3` for the dimension variables, `gridDim` and `blockDim`.

We use `dim3` variables for specifying execution configuration.

CUDA Type `dim3`

CUDA uses the vector type `dim3` for the dimension variables, `gridDim` and `blockDim`.

The `dim3` type is equivalent to `uint3` **with unspecified entries set to 1.**

As you probably noticed in the Lab1 for the lab, we could use either:

```
dim3 grid(1,1,1); // 1 block in the grid
dim3 block(32,1,1); // 32 threads per block
```

Or set block and thread per block as scalar quantity in the `<<<   >>>` (execution configuration)

Type of `blockIdx` and `threadIdx`

CUDA uses the vector type `uint3` for the index variables, `blockIdx` and `threadIdx`.

A `uint3` variable is a vector with three unsigned integer components.

We used `threadIdx.x` and `blockIdx.x` to retrieve indices in 1D grid.

2-Dimensional CUDA Grids

Why do we need higher dimensions CUDA grids?

Several applications points regularly distributed on a **2D plane**. A first example can be a matrix. A second example involves digital image processing.

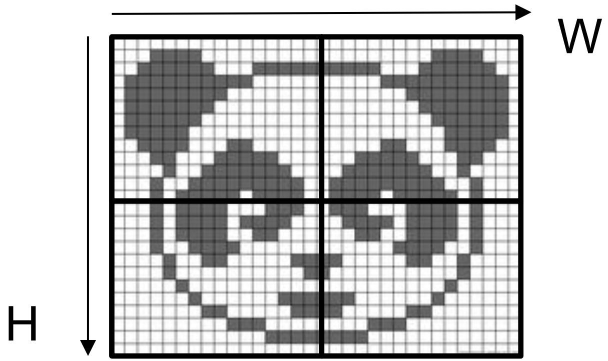
A digital raster imagine consists of a collection of **picture elements** (**pixel**) arranged in a uniform 2D rectangular grid with each pixel having an **intensity value**.

Example of 3x3 .bmp image file (see lab today)

Header								
(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)	(0,8)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)

2D Grid Kernel – Thread per block in x and y

Computing data for an image involves `W` columns and `H` rows, and we can organize the computation into 2D blocks with `TX` threads in the x-direction and `TY` threads in the y-direction.




```
TX = 16
TY = 13
```

```
dim3 blockSize(TX, TY); // Equivalent to dim3 blockSize(TX, TY, 1)
```

2D Grid Kernel – Number of blocks in x and y

Questions: how do we choose the number of blocks in x and y ? If we follow the 1D example, what would be `N` or the `ARRAY_SIZE` equivalent?

We compute the number of blocks (`bx` and `by`) needed in each direction exactly as in the 1D case:

```
int bx = (W + TX - 1)/TX;
int by = (H + TY - 1)/TY;
```

The syntax for specifying the grid size (in blocks) is

```
dim3 gridSize = dim3 (bx, by);
```

2D Grid Kernel Launch

We are ready now to launch (no difference with 1D grid):

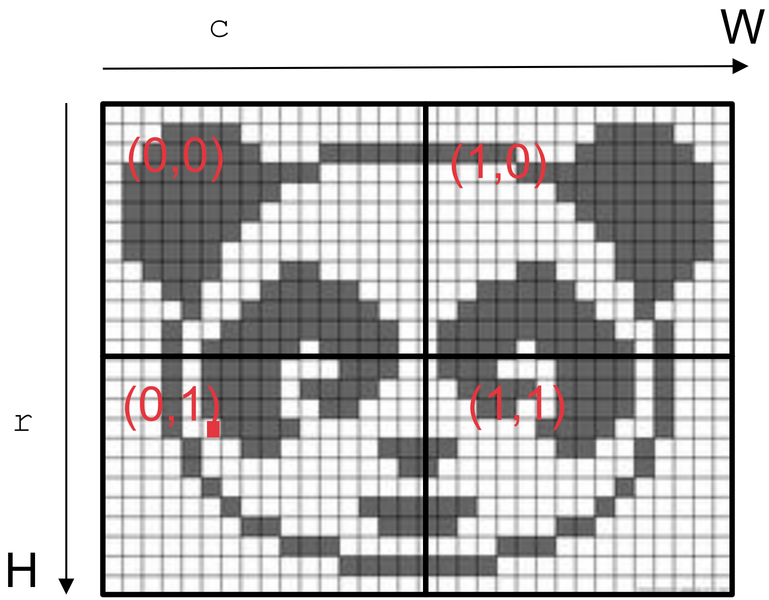
```
kernelName<<<gridSize, blockSize>>>(args)
```

Determine global indices

To identify our pixel in the image we will use to global indices `c` and `r`.

Question: How you calculate `c` and `r` for the red pixel?

```
int c = blockIdx.x*blockDim.x + threadIdx.x;
int r = blockIdx.y*blockDim.y + threadIdx.y;
```

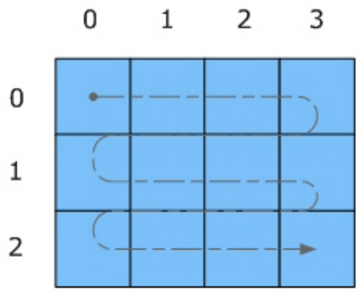


Flattening global indices to 1D global index

In several cases, it is convenient to express our 2D data as 1D data (flattening): use simply a 1D array of length `W*H`

We place values in the 1D array in **row-major order:** we store the data from row 0, followed by data from row 1 and so on.

Row-major order

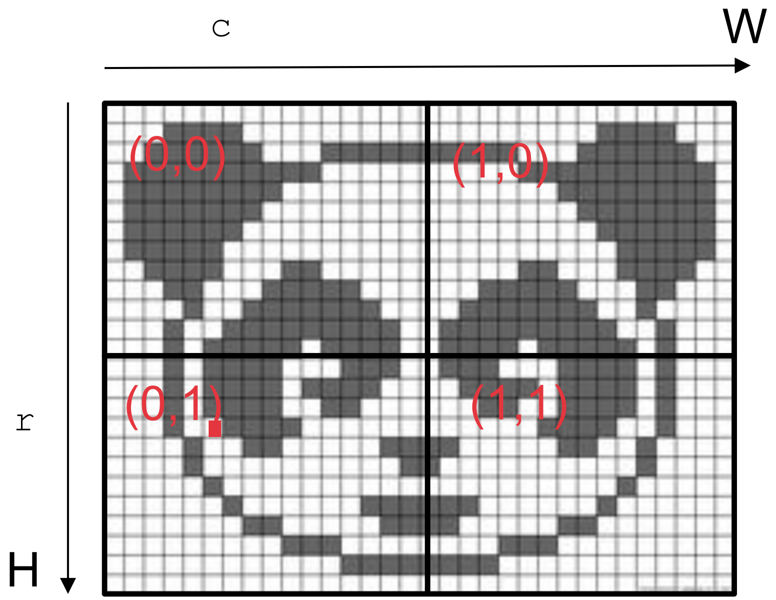


Question: Why row-major order and not column-major order in C?

Question: How do you calculate `i`, 1D index? `int i = r*w + c;`

Question: How do you calculate `i`, 1D index?

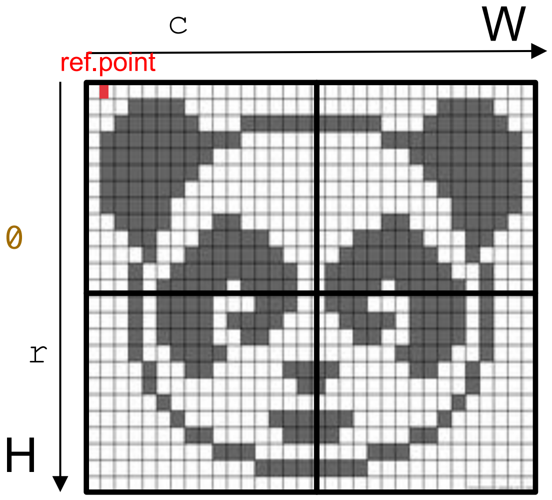
- We calculate r and c
- We flatten `r` and `c` as:
  - `int i = r*W + c;`



CUDA code for distance between points in 2D

```
#define W 32
#define H 32
#define TX 8 // number of threads per block along x-axis
#define TY 8 // number of threads per block along y-axis

int divUp(int a, int b) { return (a + b - 1) / b; }
…
int main() {
  float *out = (float*)calloc(W*H, sizeof(float)); // set all the points to 0
  float *d_out = NULL;
  cudaMalloc(&d_out, W*H*sizeof(float));
  float2 pos = { 1.0, 0.0};    // ref. point
  dim3 blockSize(TX, TY);
  dim3 gridSize(divUp(W, TX), divUp(H, TY));
  distanceKernel<<<gridSize, blockSize>>>(d_out, W, H, pos);
  cudaMemcpy(out, d_out, W*H*sizeof(float), cudaMemcpyDeviceToHost);
  cudaFree(d_out);
  free(out);
  return 0;
}
```



CUDA Kernel and device code

```
__global__ void distanceKernel(float *d_out, int w, int h, float2 pos)
{
  const int c = blockIdx.x * blockDim.x + threadIdx.x; // column
  const int r = blockIdx.y * blockDim.y + threadIdx.y; // row
  const int i = c + r*w;
  if ((c >= w) || (r >= h)) return;
      d_out[i] = distance(c, r, pos); // compute and store result
}


__device__ float distance(int c, int r, float2 pos)
{
  return sqrtf((c - pos.x)*(c - pos.x) + (r - pos.y)*(r - pos.y));
}
```

3D GRIDS

3D data set can be thought as image stack composed of **3D voxels** is a volume `W*H*D` (`D` = Depth)

An execution configuration in 3D will require to define the number of threads in the x, y and z direction, i.e TX, TY and TZ

```
dim3 blockSize(TX, TY, TZ);
```

As usual, the block grid size is then calculate depending on the input size:

```
int bx = (W + blockSize.x - 1)/blockSize.x;
int by = (H + blockSize.y - 1)/blockSize.y;
int bz = (D + blockSize.z - 1)/blockSize.z;
```

Indices 3D

In addition to row (`r`) and column (`c`) global indices, we need a new integer variable to have a global index in the stack (`s` for *stack* or *stratum*):

```
int s = blockIdx.z*blockDim.z + threadIdx.z;
```

The flattened 1D index becomes:

```
int i = c + r*w + s*w*h;
```