CUDA Programming – Part 2

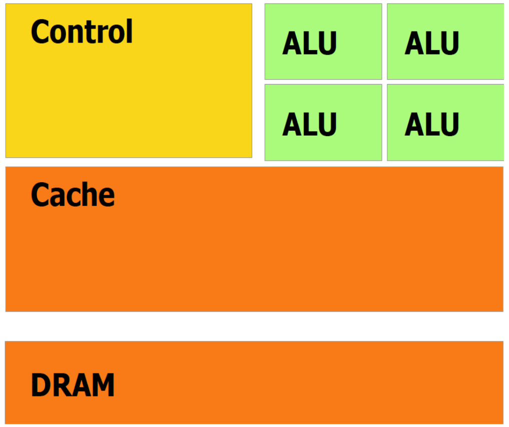
Ivy Peng
Assistant Professor in Computer Science
Scalable Parallel System (ScaLab)
Department of Computer Science, KTH

Recap – GPU Architecture

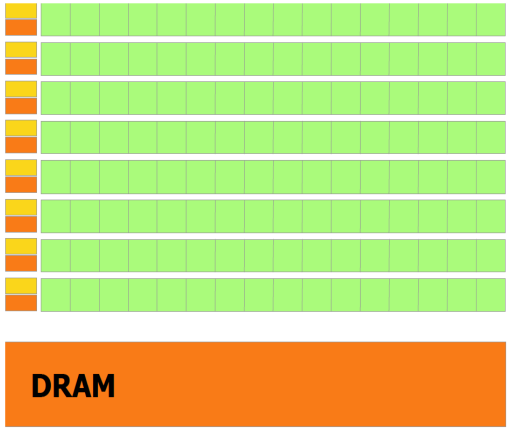
Two Metrics of Processor Performance

- **Task latency** = time elapsed between the initiation and completion of some task
- **Task throughput** = total amount of work completed per unit time

Latency-Oriented Architecture

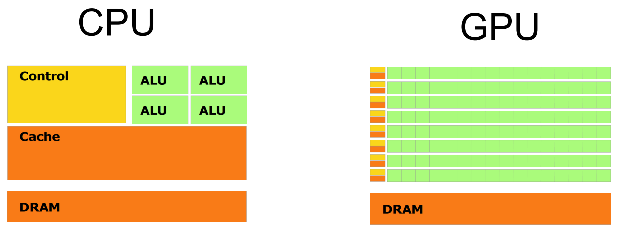


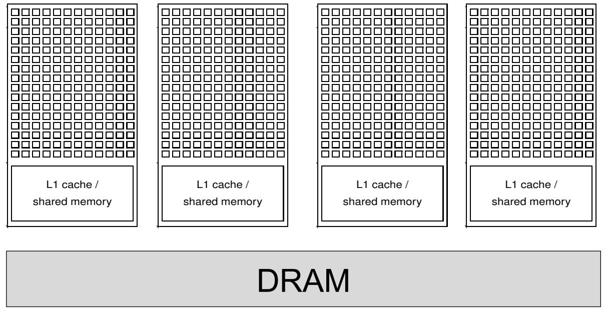
Throughput-Oriented Architecture



Recap – GPU Architecture

- GPU v.s. CPU architecture
  - **Lots of cores**, **fewer control units**: very good in compute-heavy applications with little synchronization
  - **Power efficiency**: lot of parallelism but lower clock frequency
- GPU consists of one or more **SMs**, each one comprising **hundreds of cores**





Recap - CUDA

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

- Create allocate variable on GPU memory
- Move data from CPU to GPU memory and vice-versa
- Define kernel and launch a kernel (which qualifier to use?)
- Synchronize threads

Question: Which CUDA functions you used in the lab?

Recap - Execution Configuration

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

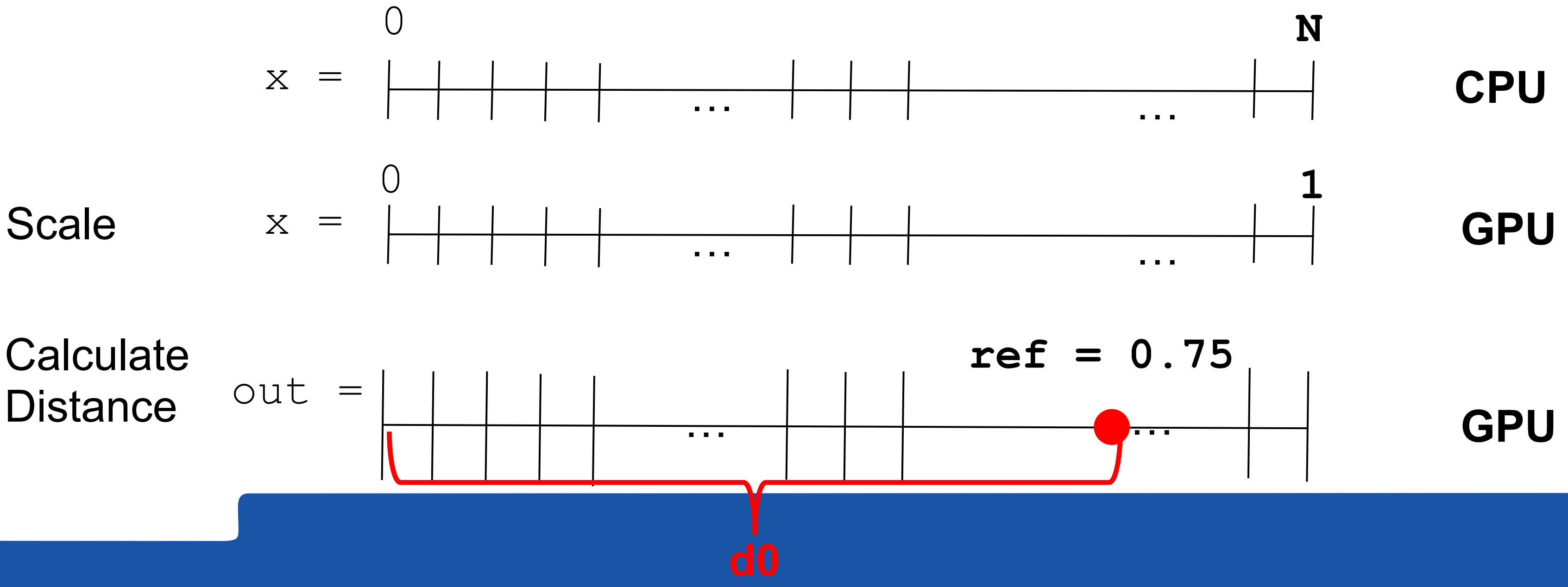
- To choose **some multiple of 32 is reasonable** since it matches up somehow with the number of **CUDA cores in an SM**
- There are limits: a single block **cannot contain more than 1,024 threads**
- For large problems, reasonable to test are 128, 256 and 512

Question: Have you tried different execution configurations in the lab? Which one gave you the best performance?

Questions?

Transform a serial example: dist

Scale an array and compute an array of distances from a reference point to each of N points uniformly spaced along a line segment.



Transform a serial example: dist

```c
#include <math.h> //Include standard math library containing sqrt.
#define N 64 // Specify a constant value for array length.

// A scaling function to convert integers 0,1,...,N-1 to evenly spaced floats
float scale(int i, int n)
{
  return ((float)i) / (n - 1);
}

// Compute the distance between 2 points on a line.
float distance(float x1, float x2)
{
  return sqrt((x2 - x1)*(x2 - x1));
}

int main()
{
    float out[N] = {0.0};
    // Choose a reference value from which distances are measured.
    const float ref = 0.5;
    for (int i = 0; i < N; ++i)
    {
       float x = scale(i, N);
       out[i] = distance(x, ref);
    }
    return 0;
}
```

the **CPU version**, uses a single For Loop that scales the loop index to create an input location and computes the distance from the reference location

1. Create the CUDA source file

- Create the file `kernel.cu` where you will have CUDA source code → CUDA codes have extension `.cu`
- Copy and paste the content of `main.cpp` into `kernel.cu`

```
#include <math.h>
#define N 64

float scale(int i, int n)
{
  return ((float)i) / (n - 1);
}

float distance(float x1, float x2)
{
  return sqrt((x2 - x1)*(x2 - x1));
}

int main()
{
    float out[N] = {0.0};
    const float ref = 0.5;
    for (int i = 0; i < N; ++i)
    {
       float x = scale(i, N);
       out[i] = distance(x, ref);
    }
   return 0;
}
```

2.1 Modify `kernel.cu`

- Delete `#include <math.h>` because CUDA internal files already include `math.h`, and insert `<stdio.h>` to enable printing the output

- Add `#define TPB 32`, **to indicate the number of threads per block** that will be used in your kernel launch

```
#include <math.h>
#include <stdio.h>
#define N 64
#define TPB 32

float scale(int i, int n){
  return ((float)i) / (n - 1);
}

float distance(float x1, float x2){
  return sqrt((x2 - x1)*(x2 - x1));
}
…
```

2.2 Modify kernel.cu

- Copy the **loop body** outside the `main()` in a `distanceKernel()` function comprising `scale()` and `distance()`
- Replace the for loop with the **kernel launch**

```
distanceKernel<<<N/TPB,TPB>>>(d_out,ref,N);
```

```
… distanceKernel(…){
  … scale(…);
  … distance(…);
}

int main(){
    float out[N] = {0.0};
    const float ref = 0.5;
    distanceKernel<<<N/TPB, TPB>>>(d_out,ref,N);
    return 0;
}
```

One single function to be run on GPU

No loop… grid instead!

3.1 Create Kernel Definition

```
__xxx__ void distanceKernel(float *d_out,
float ref, int len)
{
    …

}
```

Question: `__global__`, `__device__`, or `__host__` ?

Hint: We call this function from the host and want to run on GPU

3.2 Create Kernel Definition

```
__xxx__ float scale(int i, int n)
{
  return ((float)i)/(n - 1);
}
```

Question: `__global__`, `__device__`, or `__host__` ?

Hint: We call this function from the GPU and want to run on GPU

3.3 Create Kernel Definition

```
__xxx__ float distance(float x1, float x2)
{
  return sqrt((x2 - x1)*(x2 - x1));
}
```

Question: `__global__`, `__device__`, or `__host__` ?

Hint: We call this function from the GPU and want to run on GPU

4. Get the global thread ID using index variables

```
__global__ void distanceKernel(float *d_out, float ref, int len)
{
  const int i = blockIdx.x*blockDim.x + threadIdx.x;
  const float x = scale(i, len);
  d_out[i] = distance(x, ref);
  printf("i = %2d: dist from %f to %f is %f.\n", i, ref, x, d_out[i]);
}
```

Inside the kernel add the formula for computing index `i` (**to replace the loop index of the same name that is now removed**) using built-in index and dimension variables that CUDA provides with every kernel launch:

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

5. Create results array (`d_out`) on the GPU

Question: Which CUDA function do we use?

Did we forget anything?

```
…
int main()
{
  …
  // Declare a pointer for an array of floats
  float *d_out = 0;
  // Allocate device memory for d_out
  cudaMalloc(&d_out, N*sizeof(float));
  // Launch kernel to compute
  distanceKernel<<<N/TPB, TPB>>>(d_out, ref, N);
  return(0);
}
```

Putting everything together

```
#include <stdio.h>
#define N 64
#define TPB 32


__device__ float scale(int i, int n)
{
  return ((float)i)/(n - 1);
}


__device__ float distance(float x1, float x2)
{
  return sqrt((x2 - x1)*(x2 - x1));
}


__global__ void distanceKernel(float *d_out, float ref, int len)
{
  const int i = blockIdx.x*blockDim.x + threadIdx.x;
  const float x = scale(i, len);
  d_out[i] = distance(x, ref);
}
```

```
int main()
{
  const float ref = 0.5f;

  // Declare a pointer for an array of floats
  float *d_out = 0;

  // Allocate device memory to store the output array
  cudaMalloc(&d_out, N*sizeof(float));

  // Launch kernel to compute and store distance values
  distanceKernel<<<N/TPB, TPB>>>(d_out, ref, N);

  cudaFree(d_out); // Free the memory
  return 0;
}
```

Putting everything together

Compile it:

```
nvcc kernel.cu -o dist_v1
```

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);
```

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.

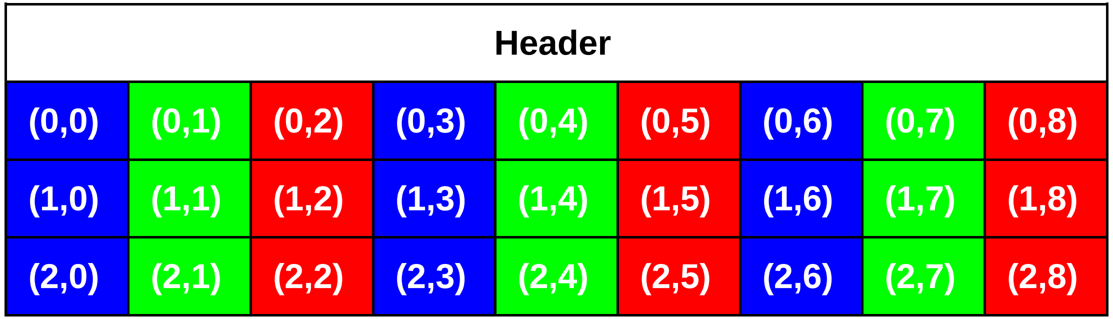
2-Dimensional Grids

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 [TX,TY]

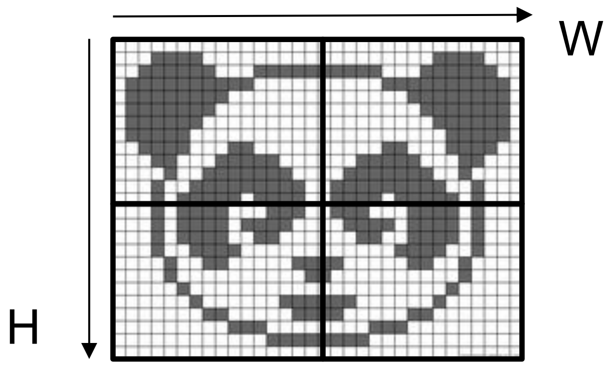
Computing data for an image of `W` columns and `H` rows

We can organize the computation into 2D blocks with `TX` threads in the x-direction and `TY` threads in the y-direction.

```
dim3 DimBlock(TX, TY);
dim3 DimBlock(TX, TY, 1);

dim3 DimGrid((W-1)/TX + 1, (H-1)/TY+1);
dim3 DimGrid((W-1)/TX + 1, (H-1)/TY+1, 1);

kernel<<<DimGrid, DimBlock>>>(……);
```



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):

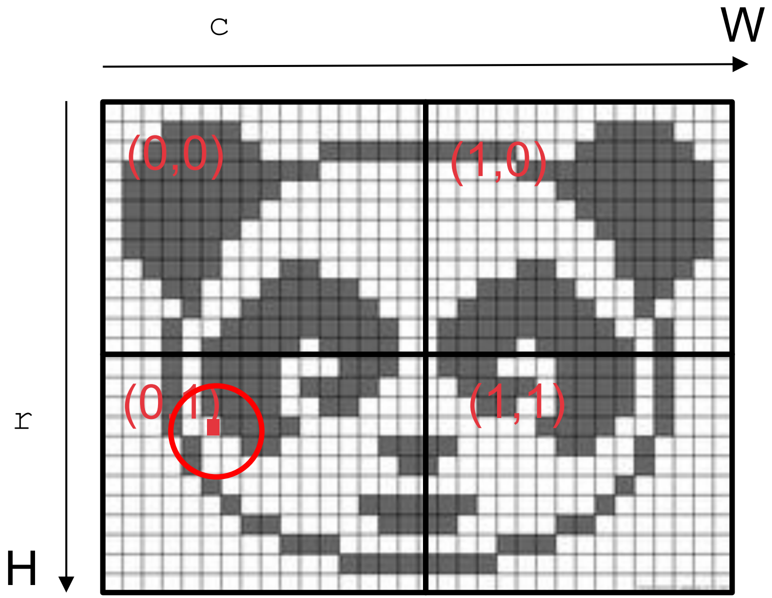
```
kernelName<<<gridSize, blockSize>>>(args)
```

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? (same as 1D grid, with .y direction)

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



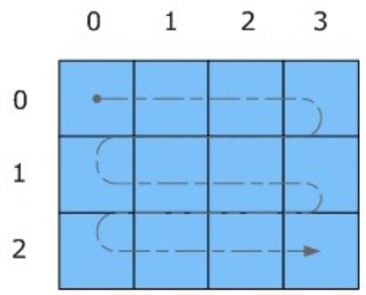
Flattening global indices to 1D global index

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.

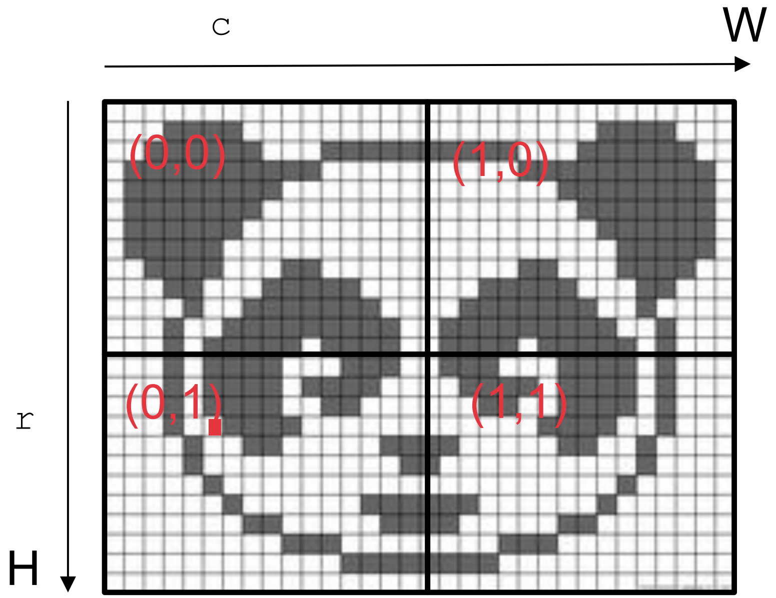
Question: Why do flattening in row-major order instead of column-major order?

Row-major order



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

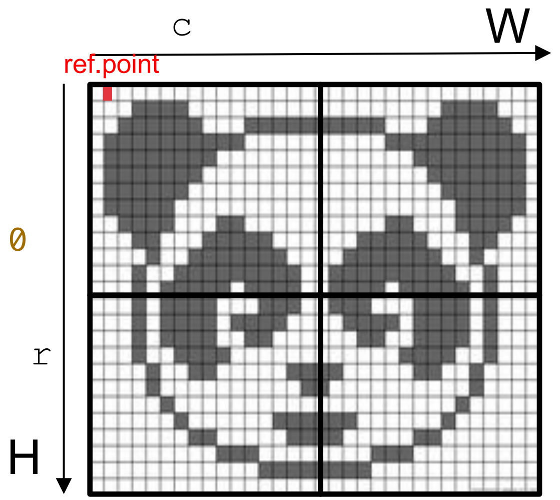
- We calculate position at [`r`, `c`]
- We flatten `r` and `c` as:
  - `int i = r*W + c;`



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

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

Q&A