Our First CUDA Program Step-by-Step

Stefano Markidis and Sergio Rivas-Gomez

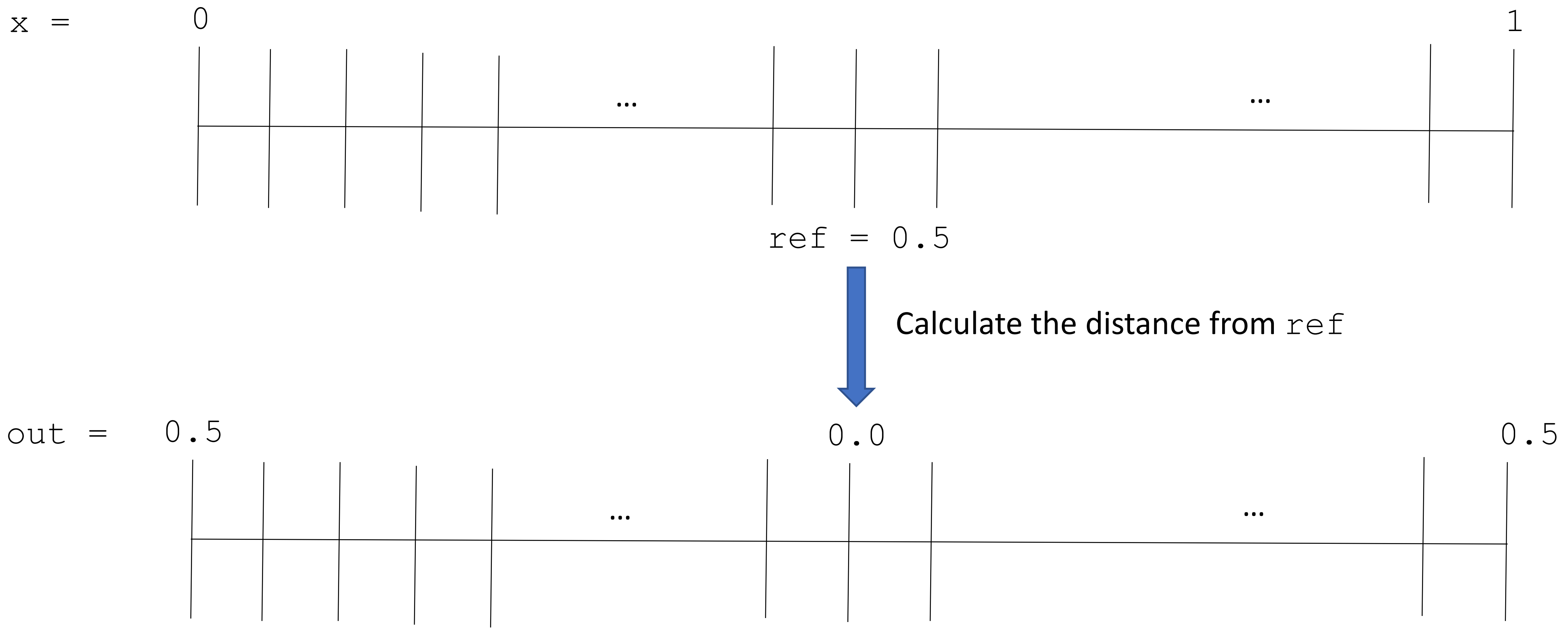
Main point

We create our first program by following the CUDA workflow, implementing few simple steps:

- Create a CUDA code starting from a C code and modifying it
- Allocate memory on GPU launch the kernels providing the execution configuration
- Define kernels to be run on the GPU
- Retrieve the thread ID from index and dimension variables in the kernel.

Problem: dist_v1

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



Serial C Code

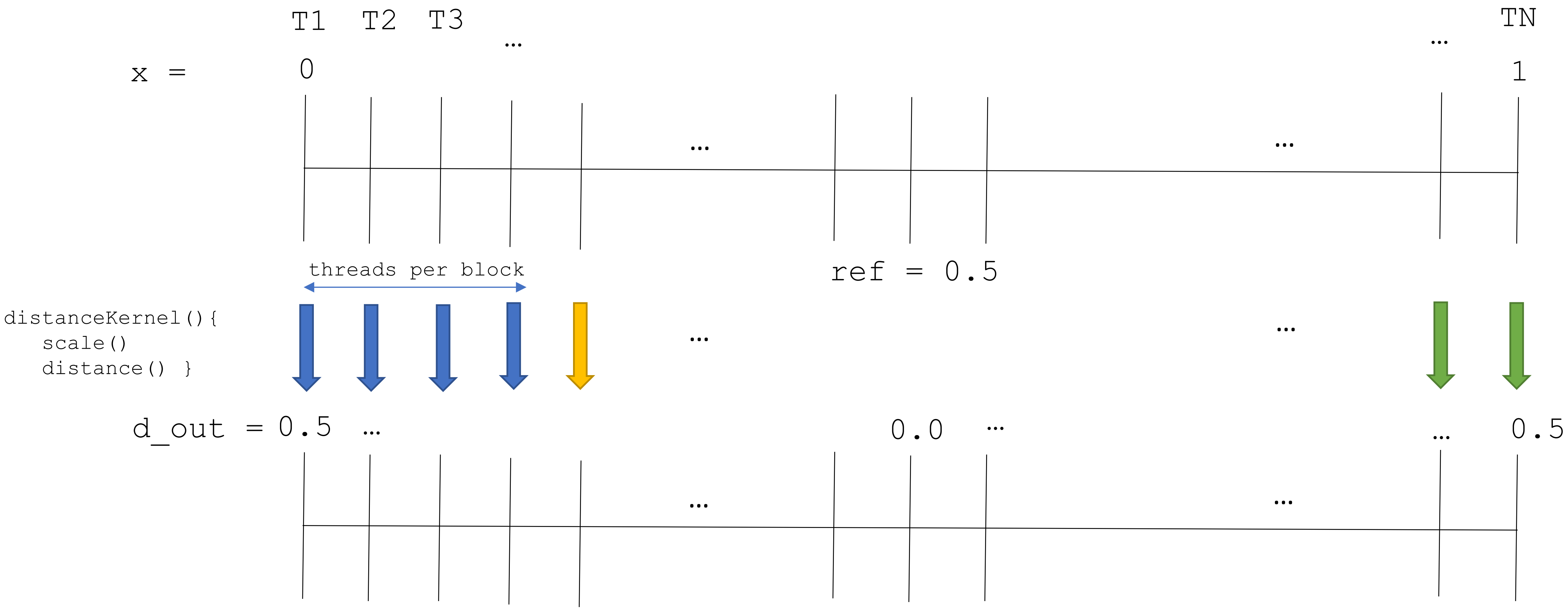
```
#include <math.h> //Include standard math library containing sqrt.
#define N 64 // Specify a constant value for array length.

// 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));
}
// main function
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;
}
```

Overall Strategy for Porting to GPUs

Launch N kernels, each one processing one point



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`
- **Question:** Is this a CUDA code?

```
#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;
}
```

Instead of loop from 0 to N, we launch N Threads!

3. Create Results Array (`d_out`) on the GPU

```
…
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);
}
```

4.1 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.

4.2 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

4.3 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

5. Get the my Thread ID in the Kernel

```
__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
```

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);
  printf("i = %2d: dist from %f to %f is %f.\n", i, ref, x,
d_out[i]);
}
```

```
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;
}
```

Did we forget anything here?

Running on Tegner

Load the CUDA environment:

```
module load cuda/7.0
```

Compile it:

```
nvcc -arch=sm_30 kernel.cu –o dist_v1
```

Ask for allocation:

```
salloc --nodes=1 --gres=gpu:K420:1 -t 00:05:
00 –A … - -reservation=…
```

Run it:

```
srun –n 1 ./dist_v1
```

Where is my Data: Host or Device Memory?

- Remember that the kernel (`distanceKernel()`) executes on the device, so it cannot return a value to the host.

- The kernel generally has access to device memory, not to the host memory, so we allocate device memory for the output array using `cudaMalloc()`

Question: In `kernel.cu`, how would you move `d_out` from the device to host memory?

Careful with Integer Arithmetic!

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

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.

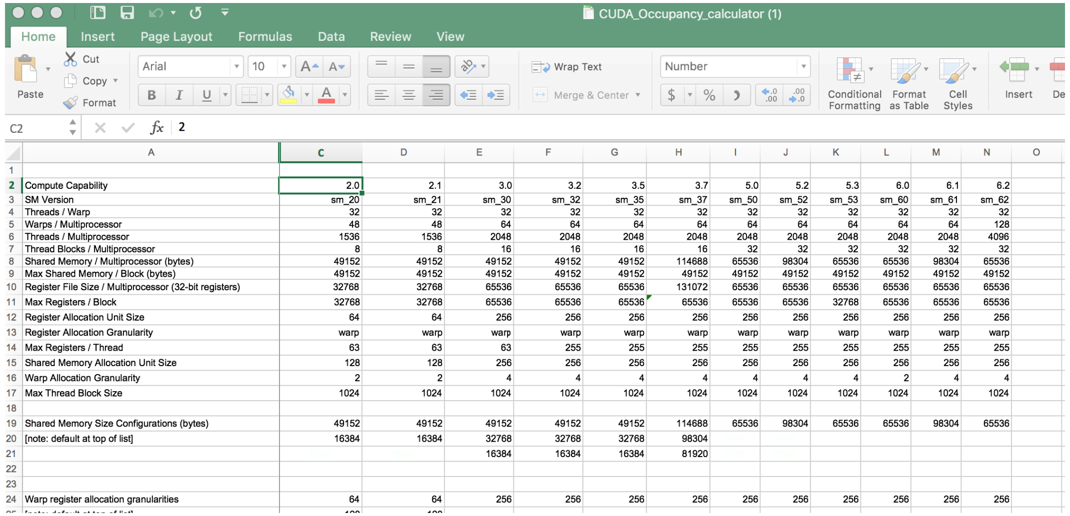
How do I choose `TPB` or Execution Configuration?

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.

GPU Occupancy – choose `TPB`

- In general you want to size your blocks/grid to match your data and simultaneously maximize occupancy, that is, how many threads are active at one time.
- You might want to use the CUDA Occupancy Calculator to compute the multiprocessor *occupancy* of a GPU by a given CUDA kernel.
  - This tool is an MS excel spreadsheet that helps you choose thread block size for your kernel.



	A	C	D	E	F	G	H	I	J	K	L	M	N
1													
2	Compute Capability	2.0	2.1	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2
3	SM Version	sm_20	sm_21	sm_30	sm_32	sm_35	sm_37	sm_50	sm_52	sm_53	sm_60	sm_61	sm_62
4	Threads / Warp	32	32	32	32	32	32	32	32	32	32	32	32
5	Warps / Multiprocessor	48	48	64	64	64	64	64	64	64	64	64	128
6	Threads / Multiprocessor	1536	1536	2048	2048	2048	2048	2048	2048	2048	2048	2048	4096
7	Thread Blocks / Multiprocessor	8	8	16	16	16	16	32	32	32	32	32	32
8	Shared Memory / Multiprocessor (bytes)	49152	49152	49152	49152	49152	114688	65536	98304	65536	65536	98304	65536
9	Max Shared Memory / Block (bytes)	49152	49152	49152	49152	49152	49152	49152	49152	49152	49152	49152	49152
10	Register File Size / Multiprocessor (32-bit registers)	32768	32768	65536	65536	65536	131072	65536	65536	65536	65536	65536	65536
11	Max Registers / Block	32768	32768	65536	65536	65536	65536	65536	65536	32768	65536	65536	65536
12	Register Allocation Unit Size	64	64	256	256	256	256	256	256	256	256	256	256
13	Register Allocation Granularity	warp	warp	warp	warp	warp	warp	warp	warp	warp	warp	warp	warp
14	Max Registers / Thread	63	63	63	255	255	255	255	255	255	255	255	255
15	Shared Memory Allocation Unit Size	128	128	256	256	256	256	256	256	256	256	256	256
16	Warp Allocation Granularity	2	2	4	4	4	4	4	4	4	2	4	4
17	Max Thread Block Size	1024	1024	1024	1024	1024	1024	1024	1024	1024	1024	1024	1024
18													
19	Shared Memory Size Configurations (bytes)	49152	49152	49152	49152	49152	114688	65536	98304	65536	65536	98304	65536
20	[note: default at top of list]	16384	16384	32768	32768	32768	98304						
21				16384	16384	16384	81920						
22													
23													
24	Warp register allocation granularities	64	64	256	256	256	256	256	256	256	256	256	256

http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

To Summarize

We create our first program by implementing few simple steps:

- Allocate memory on GPU
- Define one kernel to be launched from the CPU (`__global__` qualifier)
- Define two kernels to be launched from the GPU (`__device__` qualifier)
- Launch kernel providing the execution configuration, basically how many threads = input size
- When we print from the GPU, we need to use `cudaDeviceSynchronize()` to allow flushing results to screen before terminate