CUDA – From Loops to Grids

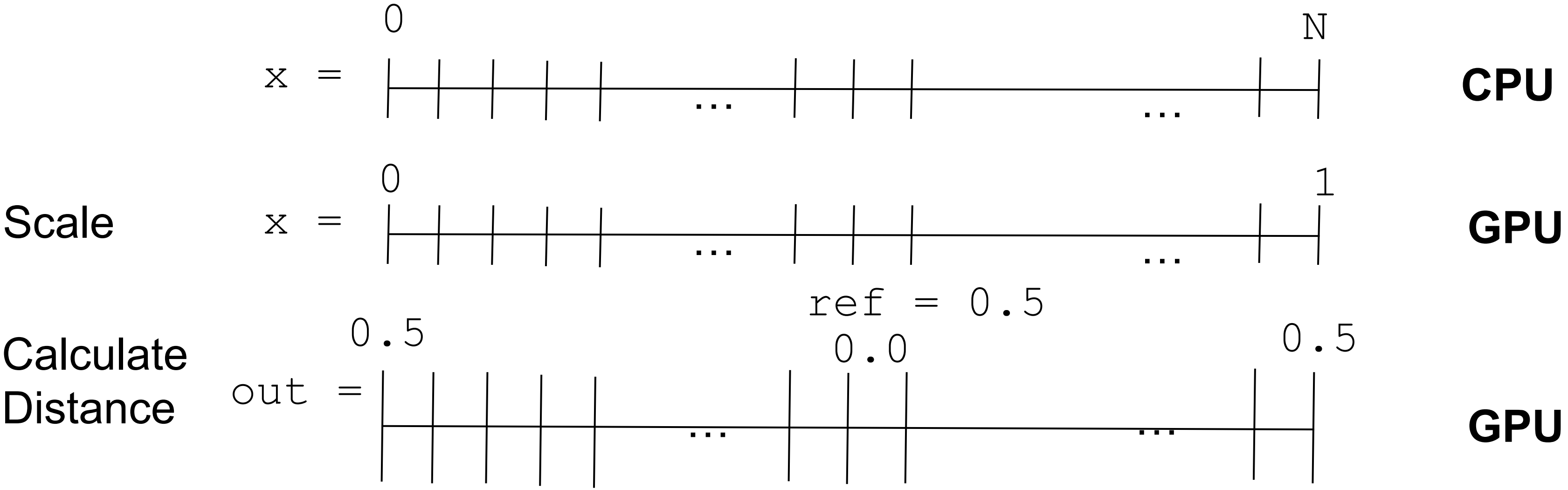
Stefano Markidis

KTH Royal Institute of Technology

Motivational Example: `dist_v1`

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

Take the *scale* and *calculate distance* and port it to GPU!



CPU

Scale

GPU

Calculate Distance

GPU

```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));
}
// 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;
}
```

`dist_v1` has a **single loop that scales** the loop index to create an input location and **the computes/stores 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

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;
}
```

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?

```
…
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);
}
```

Did we forget anything?

Putting everything together: our first CUDA code

```
#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;
}
```

Load the CUDA environment:

```
 module load cuda/7.0
```

On **Tegner** today!
Not Beskow.

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
```

Questions

- Does it work?
- Does it print anything ?

Use `cudaDeviceSynchronize()` !

Careful with Integer Arithmetic!

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

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.

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?

Lab Exercises: CUDA Part 1

- `Hello World` in CUDA.
    - Think about the problem we had in printing from the device
    - CUDA Fortran does not support printing from kernel so no exercise 1 for the Fortran club
- Write a CUDA code `saxpy` (Single-precision alpha***X** + **Y**) to run on GPU
    - Think about how we implemented `kernel.cu`