Cairo University
Faculty of Engineering

Department of Computer Engineering

Parallel Computing

Lab 4

Team Member 1	
Full Name	عبدالرحمن حمدي احمد مخيمر
Section	1
BN	38
Code	9202833

Team Member 2	
Full Name	عبدالرحمن نعمان لقمان ابراهيم
Section	2
BN	4
Code	9202851

Submitted To: Eng. Mohamed Abdullah

Date: 4/18/2024

Complete Analysis on all kernels & Python Code

*For all experiments, we have fixed the block and tile size to be **16**.*

The values recorded represent the **execution time**.

Mask Size: 3x3

Batch Size = 3

Image Size	Kernel 1	Kernel 2	Kernel 3	Python
500x500	158.20 us	118.37 us	102.29 us	2574.50 us
3840x2160	5.1829 ms	3.9267 ms	3.7146 ms	94482.25 us

Using Constant Mask

Implementation:

```
__constant__ float d_mask[9];
cudaMemcpyToSymbol(d_mask, mask, maskSize * maskSize * sizeof(float));
```

Image Size	Mask	Kernel 1	Kernel 2	Kernel 3
1200x800	GLOBAL	595.86 us	468.49 us	434.04 us
1200x800	CONSTANT	561.78 us	422.20 us	383.83 us

Image Size: 500x500

Batch Size = 3

Mask Size	Kernel 1	Kernel 2	Kernel 3	Python
7x7	760.46 us	378.09 us	350.86 us	3443.75 us
11x11	1.8285 ms	1.1075 ms	0.995 ms	15035.25 us

Image Size: 500x500

Mask Size: 3x3

Batch Size	Kernel 1	Kernel 2	Kernel 3	Python
1	57.182 us	41.567 us	38.703 us	1096.00 us
5	268.95 us	200.32 us	180.90 us	6797.00 us

Explanation

Kernel 1:

- Since this is the basic implementation, each thread will be responsible for a single output element but will load the whole input image at the same time. Therefore, it makes sense that it is the slowest in all cases no matter what parameter changed because it takes more time for all threads to load the entire input data iteratively from global memory to registers.

Kernel 2:

- Each block is responsible for loading a tile of input images, where the tile size matches the specified input tile size.
- Now what's that input tile? It's basically the normal TILE_SIZE plus the extra image pixels that will be accounted for when applying convolution with the mask…
- So due to the block size being the same as the input tile size, each thread will load that corresponding cell ensuring that we'll cover the whole area including the extra pixels coming from the convolution mask.
- Since each thread within a block is responsible for loading a single cell of the input tile. This approach optimizes memory access patterns by utilizing shared memory and reducing redundant memory loads.
- It reduces the total amount of data transferred from global memory to registers compared to Kernel 1. This resulted in improved performance due to better memory locality and reduced memory bandwidth usage.

Kernel 3:

- Now here's the tricky situation… The block size doesn't match the input tile size but the OUTPUT tile size instead, which is supposedly equal to TILE_SIZE meaning that it holds the convolution results without the extra dimensions coming from the mask.
- For each thread to operate, it won't be enough to only load the corresponding cell from the image since it needs the few upper and side pixels to apply convolution using the mask. This is why when implementing Kernel 3, we loop from the thread index and increment by the block dimension to have all threads collectively scan the needed cells, making it almost similar to Kernel 2 except for these extra pixels to be loaded by each thread.
- Each thread within a block is responsible for loading the corresponding cell of the output tile plus additional cells for overlap.
- This approach ensures that all necessary input data for computing the output tile is loaded efficiently into shared memory. The additional cells loaded by each thread account for the overlap required for applying the convolution mask.
- By loading input data into shared memory and accounting for the required overlap, Kernel 3 minimizes memory access latency and maximizes memory throughput. How? It may seem like that it should be slower since each thread loads more pixels compared to Kernel 2… BUT when observing the overall throughput, it actually turned out to be faster since data is available more quickly.

Placing the mask in the constant memory speeded up the execution time as well since reading from the constant memory in the SM is faster than the global memory… again makes sense!

Python:

Launching CUDA kernels from Python incurs overhead. This overhead arises from various tasks involved in kernel launch, such as data transfer between CPU and GPU, GPU memory management, and synchronization between CPU and GPU. These tasks introduce latency and diminish the efficiency of GPU computations compared to invoking CUDA kernels directly from C code. Additionally, in Python, all necessary GPU declarations must be made upfront, leading to increased initialization time, especially noticeable in the first batch processed.