Parallel Computing Assignment 2

Professor: Ferrandi Fabrizio
Teaching Assistant: Curzel Serena

POLITECNICO
MILANO 1863

Students

Capodanno Mario

Grillo Valerio

Lovino Emanuele

A. Y. 2024-2025

Contents

Chapter 1

Introduction

1.1 Challenge Description

2D convolution is one of the most fundamental operations in image processing and machine learning, with applications in edge detection, convolutional neural networks, and more. However, the computational intensity increases rapidly with an increase in the size of the inputs and masks. The following report is dedicated to the optimization of 2D convolution on GPUs using CUDA. The implemented versions concern both tiled and untiled 2D convolution.

1.2 Experimental Setup

The experimental setup was implemented in Google Colab. This enviroment has permitted us to compile and run CUDA C++ code directly within Colab Notebook utilizing an NVIDIA Tesla T4 GPU whose specs are listed below:

```
Device Number: 0
  Device name: Tesla T4
  max Blocks Per MultiProcessor: 16
  max Threads Per MultiProcessor: 1024
  max Threads Per Block: 1024
  num SM: 40
  num bytes sharedMem Per Block: 49152
  num bytes sharedMem Per Multiprocessor: 65536
  Memory Clock Rate (KHz): 5001000
  Memory Bus Width (bits): 256
  Peak Memory Bandwidth (GB/s): 320.064000
```

Figure 1.1: Output of the cudaGetDeviceProperties() method

1.3 Design choices

1.3.1 Non Tiled Implementation

The first naive implementation doesn't exploit any type of faster memory in the GPU (no constant/shared memory are used) and makes accesses only to the GPU global memory. The crucial point is memory bandwidth, in this implementation in fact, the ratio of floating-point calculation to global memory accesses is about 1.0. In a basic

kernel, every thread in a thread block will perform Mask_Width^2 accesses to the image array, for a total of:

$$\text{Mask\_Width}^2 \times \text{O\_TILE\_WIDTH}^2$$

accesses per block.

1.3.2 Tiled Implementation

For the tiled 2D-convolution, we first decided to use the constant memory of the GPU, which is usually almost as fast as shared memory, as the mask M has three important characteristics: (i) size of M is typically small, (ii) content of M never changes, (iii) all thread access the M elements in the same order.

In our implementation, all the threads in a block first collaboratively load the input tile into the shared memory and then they calculate the elements of the output tile by accessing the input in the shared memory space. In this case the input tile will be larger than the output by $2 \times$ Mask_Radius, meaning that the loading phase will be faster (as each thread loads just one input element), while during computation a small part of the threads (the ones responsible for the load of halo cells) will be disabled. For larger filters, it is expected an increasing in the ratio between arithmetic to global memory accesses as data re-utilization is enhanced.

1.4 Performance measurements

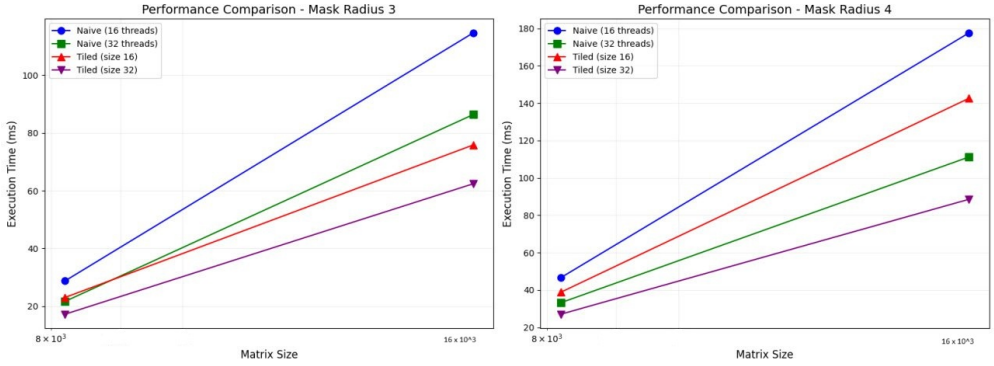

Table 1.1: Performance Comparison for Different Matrix Sizes and Mask Radii

Matrix Size	MASK_RADIUS	Execution Time (ms)		
		Naive (16)	Naive (32)	Tiled (16 TILE / 32 TILE)
8192	3	28.77	21.68	23.04 / 17.25
	4	46.60	33.19	38.80 / 27.04
16384	3	114.552	86.30	75.79 / 62.37
	4	177.60	111.19	142.602 / 88.56