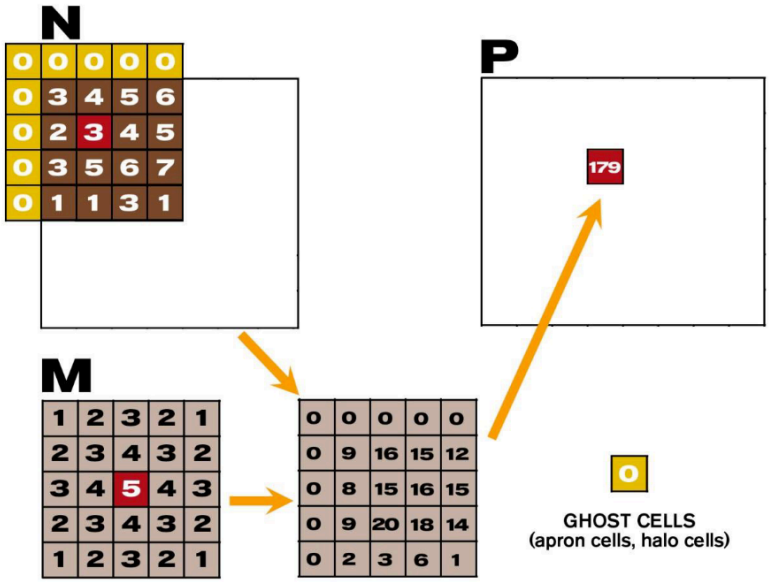
Java & Cuda Kernel
Image processing

Mattia Bacci 7060637
Elaborato Final-Term

13/01/2021

Kernel Image Processing

$$g(x,y) = \omega * f(x,y) = \sum_{dx=-a}^{a} \sum_{dy=-b}^{b} \omega(dx,dy) f(x+dx, y+dy)$$

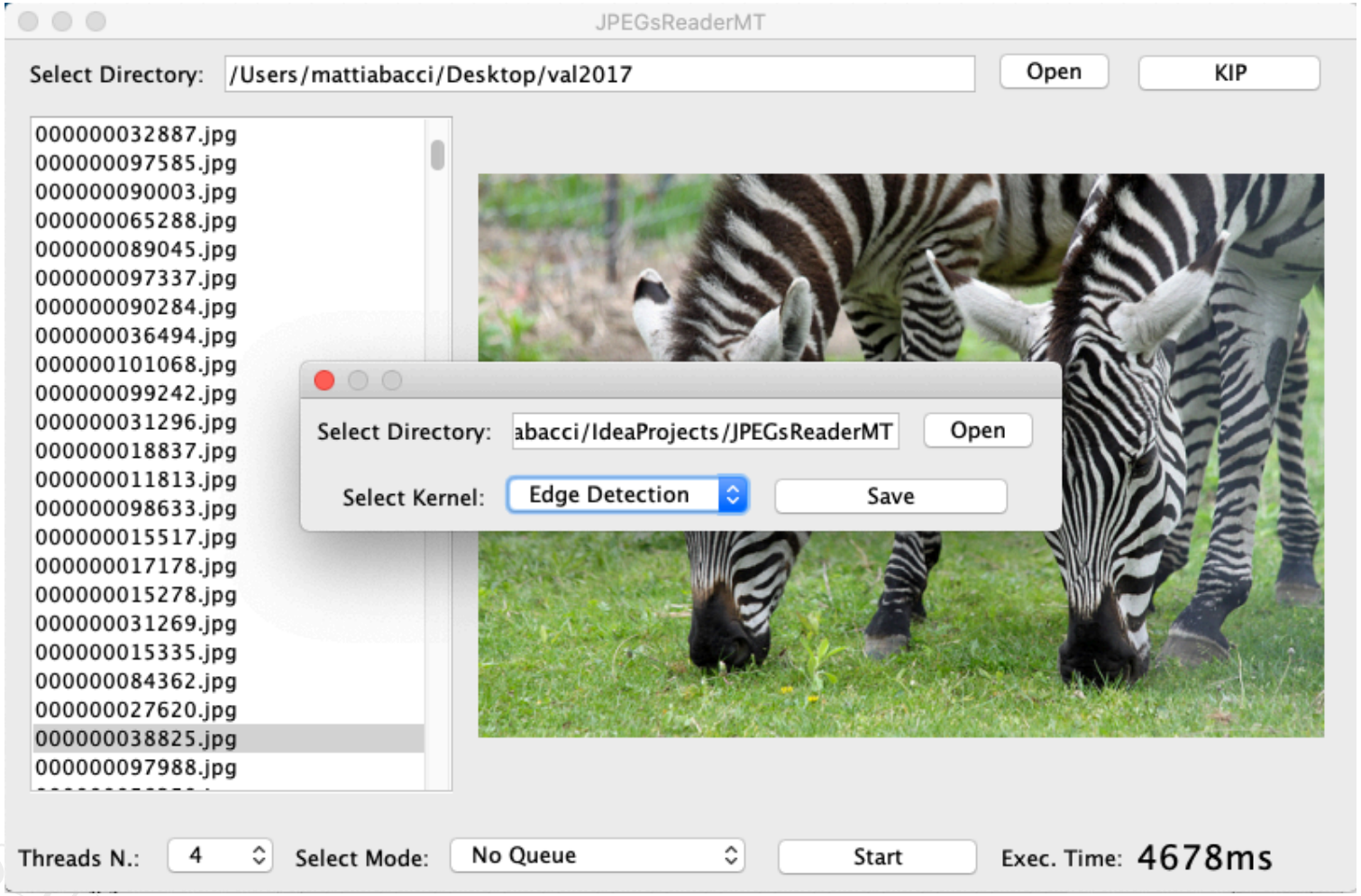


Kernel Image Processing

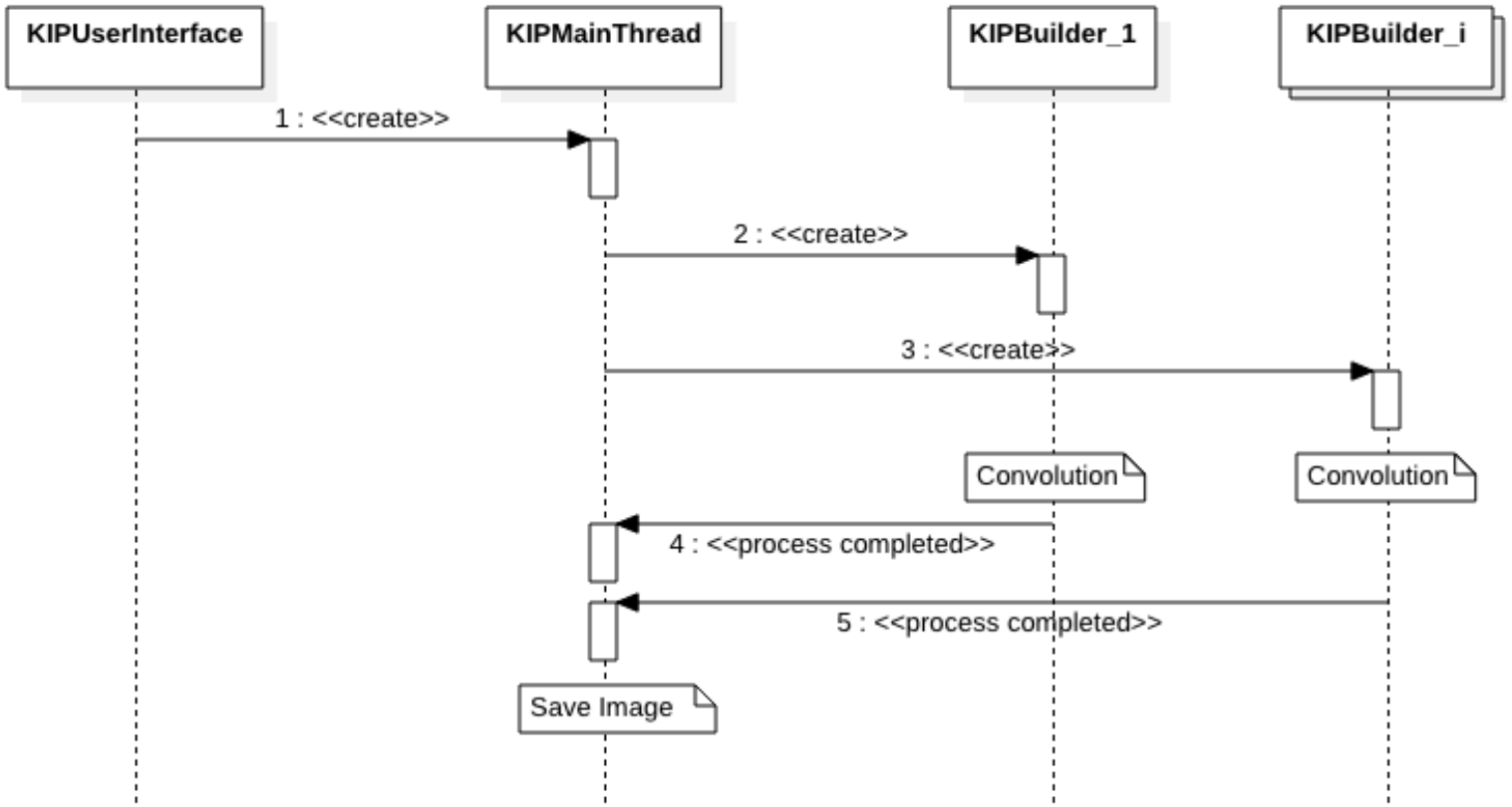
All tests are performed using the Edge Detection Kernel:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Java


JPEGsReaderMT
Select Directory:
/Users/mattiabacci/Desktop/val2017
Open
KIP
000000032887.jpg
000000097585.jpg
000000090003.jpg
000000065288.jpg
000000089045.jpg
000000097337.jpg
000000090284.jpg
000000036494.jpg
000000101068.jpg
000000099242.jpg
000000031296.jpg
000000018837.jpg
000000011813.jpg
000000098633.jpg
000000015517.jpg
000000017178.jpg
000000015278.jpg
000000031269.jpg
000000015335.jpg
000000084362.jpg
000000027620.jpg
000000038825.jpg
000000097988.jpg
Select Directory:
abacci/IdeaProjects/JPEGsReaderMT
Open
Select Kernel:
Edge Detection
Save
Threads N.:
4
Select Mode:
No Queue
Start
Exec. Time:
4678ms

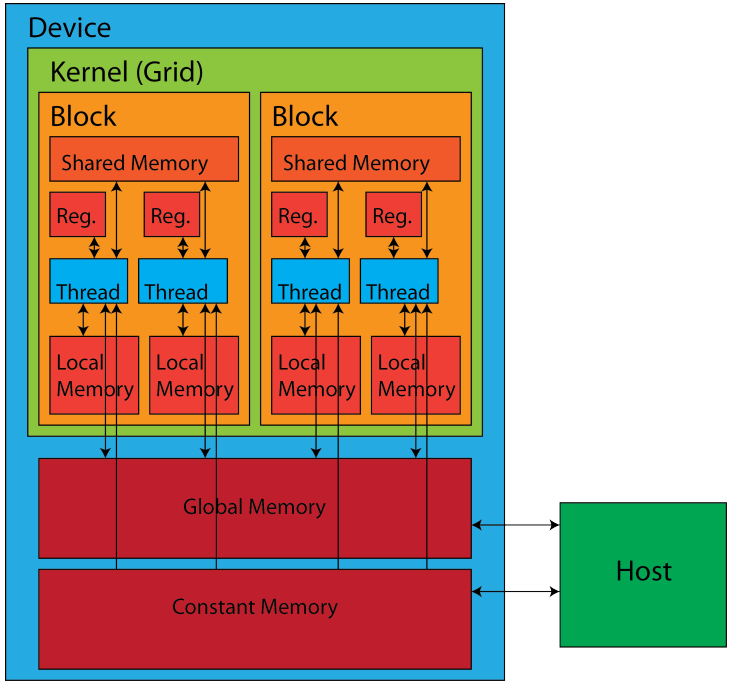
Sequence Diagram



CUDA

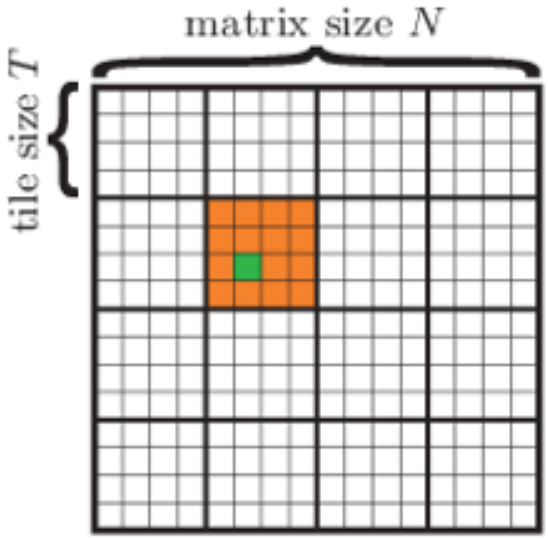
In CUDA fast memory access is required to perform fast computation, analyzing the problem we discover that a lot of elements are used more tha once. We have to copy these elements in the Shared Memory of the GPU.

The higher bandwidth of the Shared Memory respect to the bandwith of the Global Memory allows us to have faster computation.



CUDA - Tiling

Partition the data into subsets called tiles so that each tile fits into the shared memory. We can do that because the kernel computation on these tiles can be done independently of each other.



In this implementation TILE's WIDTH is set to 16. Counting the border elements, the number of the actually copied elements in the SM is given by:

$$(TILE\_WIDTH + KERNEL\_WIDTH - 1)^2$$

From now on: $NS\_WIDTH^2 = (TILE\_WIDTH + KERNEL\_WIDTH - 1)^2$

CUDA - Tiling

Two ways can be chosen to load the elements in the SM:

1. Create a Block with the number of threads equal to the TILE's dimension(256), the first $NS\_WIDTH^2 - TILE\_WIDTH^2$ Threads have to load a second element. In this case **all Threads are used for the convolution**.

2. Create a Block with the number of threads equal to

$$NS\_WIDTH^2$$

In this case we can write simpler code but for each Block we will have $NS\_WIDTH^2 - TILE\_WIDTH^2$ Thread that will stop.

If we consider TILE's WIDTH equal to 16 and KERNEL WIDTH equal to 3, for each Block **96 out of 352 Thread will NOT perform the convolution**:
$(16 + 3 – 1)^2 = 324$ ------> 352 (to maintain the warp size)(28 don't do anything)
352 – 256 = 96

More On Memory

It is possible to calculate the bandwidth reduction thanks to tiling through the the following formula:

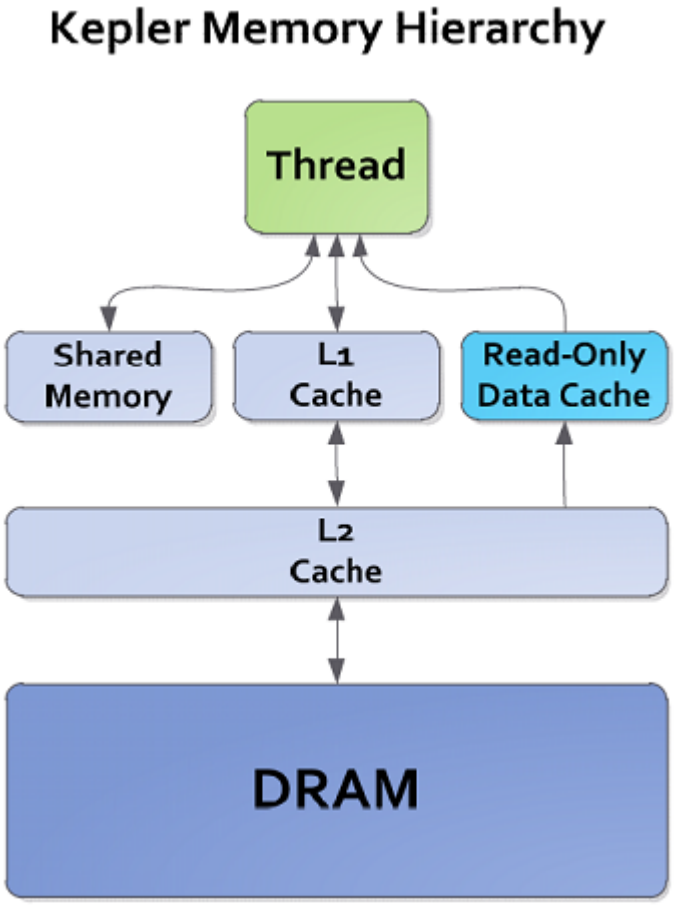
$$\frac{TILE\_WIDTH^2 * KERNEL\_WIDTH^2}{(TILE\_WIDTH + KERNEL\_WIDTH - 1)^2}$$

TILE_WIDTH	16
KERNEL_WIDTH = 3	7
KERNEL_WIDTH = 5	16

More On Memory

KERNEL Matrix doesn't change through the computation so we can load it in the constant memory. Instead of __costant__ we use const modifier together with the __restric__ keyword.

The use of this two modifier to decorate a pointer allows the compiler to provide hight optimization and , when generating code for device with Compute Capabilities >= 3.5, to allow the flow of data to the Read-Only cache.



Results

Java/CUDA	Risoluzione	Tempo Sequenziale	Tempo MultiThread	SpeedUp	Efficienza
Java	256x256	64 ms	21 ms	3	75%
CUDA	256x256	21 ms	690 * 10^-3 ms	~30	--%
Java	1920x1080	720 ms	210 ms	3.4	85%
CUDA	1920x1080	730 ms	17 ms	~43	--%
Java	7680×4320	10800 ms	3000 ms	3.6	90%
CUDA	7680×4320	12800 ms	260 ms	~50	--%

Results

Java/CUDA	Risoluzione	Tempo Sequenziale	Tempo MultiThread	SpeedUp	Efficienza
Java	256x256	64 ms	21 ms	3	75%
CUDA	256x256	21 ms	690 * 10^-3 ms	~30	--%
Java	1920x1080	720 ms	210 ms	3.4	85%
CUDA	1920x1080	730 ms	17 ms	~43	--%
Java	7680×4320	10800 ms	3000 ms	3.6	90%
CUDA	7680×4320	12800 ms	260 ms	~50	--%

The **C++ sequential version is 2 second slower than the same Java version**, Java version should be much slower. Other than C++ nature to be much faster than Java, **our Java code start 2 Threads to implement the sequential version!!**

The evil of Floating-Point Computation

Different structures are used to store the Kernel Matrix:

- Java – Integer Bidimensional Array with float multiplication factor
- C++ - Float Bidimensional Array

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

In 7680x4320 image there are ~100 million value to be updated.

For each value are performed 9 floating-point products and 9 floating point sums.

1.8 billion floating-point operations

~12s

1.8 billion integer operations

~8s