A report on

IMAGE PROCESSING USING CUDA

Done by

Unmesh N Joshi
VUnet ID uji300

Department of Computer Science
Vrije University
Amsterdam, The Netherlands

Winter Semester

Abstract

CUDA is widely used for image processing. For practical course in parallel programming, an image processing pipeline is implemented using CUDA/C++ 5.0 and is tested on nVIDIA GTX480. Roofline model is used for performance benchmarking. GPUs on DAS4 cluster are used to run CUDA kernels.

Contents

List of Figures

Chapter 1

Objective and Usage of tool

1.1 Objective

Write Image processing application using CUDA C++ (CUDA toolkit 5.0) and test the performance.

1.2 Usage

Accelerated applications has two optional command line parameters: First is special stride value for histogram kernel, which is 8 by default. It is explained in the histogram kernel section. Second value is more of a testing purpose. It can be 0 or 1 or 2. Each indicate which version of smoothing kernel to launch. 0 indicates it is constant memory version, 1 indicates it is shared memory convolution kernel. 2 indicates it is trial version which accesses image using 32-bits at a time. Default value is 0. Some things to note are image09.bmp does not work if second argument is less than 8. This is because it makes number of blocks to be more than 65536 in one dimension, which is not possible. Also keep in mind that if kernel launch argument is to be provided, it must precede second argument like follows:

prun -v -np 1 -native '-l gpu=GTX480' <executable><image><space>8<space>1 means special stride is 8 and launch version 1 of kernel for smoothing

prun -v -np 1 -native '-l gpu=GTX480' <executable><image><space>1 means special stride is 1

Chapter 2

GPU specification study and preparation

2.1 GTX 480

Unlike most programming languages, CUDA is coupled very closely with the hardware implementation. Thus, sufficient time was given to analyse the architecture of GTX 480 GPU and its theoretical performance/s.Some calculations are presented below:

Memory bandwidth = 177.4 GB/s
Peak performance (single precision) = 1350 GFLOPs

2.1.1 Calculation of ideal AI

Arithmetic Intensity is nothing by Floating Point OPerations per bytes. Peak performance when divided by memory bandwidth yields Arithmetic intensity which is required to achieve peak performance for particular GPU:

$$AI = \frac{1.35 \times 10^{12} GFLOP/s}{177.4 \times 10^{9} GB/s} = 7.6 FLOPs/byte \tag{2.1}$$

Following is the part of output shown by Device Query which helped in programming of this assignment.

CUDA Capability Major/Minor version number	2.0
Total amount of global memory	1610153984 bytes
Multiprocessors x Cores/MP = Cores	15 (MP) x 32 (Cores/MP) = 480 (Cores)
Total amount of constant memory	65536 bytes
Total amount of shared memory per block	49152 bytes
Total number of registers available per block	32768
Warp size	32
Maximum number of threads per block	1024
Maximum sizes of each dimension of a block	1024 x 1024 x 64
Maximum sizes of each dimension of a grid	65535 x 65535 x 65535
Maximum memory pitch	2147483647 bytes
Memory Bus Width	384-bit
Memory Clock rate	1848.00 Mhz
GPU Clock rate	1.40 GHz

Table 2.1: GeForce GTX 480 important specifications for programmer

2.1.2 Model used for benchmarking

Roofline model[1] is used for performance analysis. Roofline graph was plotted to know about complexity of kernels and to know whether kernels are memory bound or compute bound.

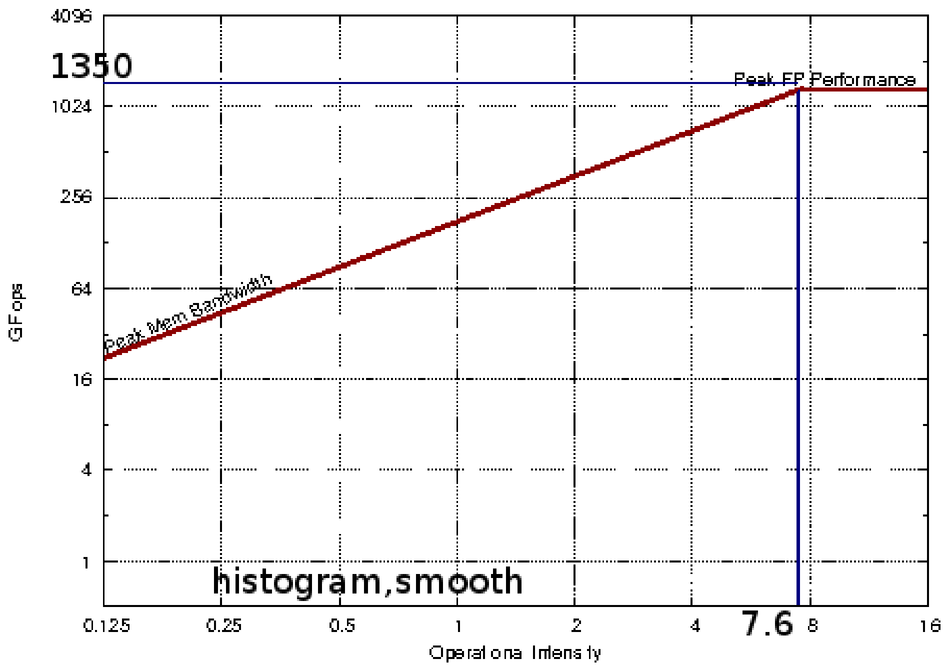

Figure 2.1: Roofline plot for GTX480

Chapter 3

Work Done

3.1 Preliminary steps

As the roofline plot shows, both kernels are far away from the ridge point, indicating that writing those kernels for maximum performance is difficult task. It can be inferred from the location of the ridge point.[1]. Here the fun started!

A note from CUDA occupancy calculator
Higher occupancy does not necessarily mean higher performance. If a kernel is not bandwidth-limited or latency-limited, then increasing occupancy will not necessarily increase performance. If a kernel grid is already running at least one thread block per multiprocessor in the GPU, and it is bottlenecked by computation and not by global memory accesses, then increasing occupancy may have no effect. In fact, making changes just to increase occupancy can have other effects, such as additional instructions, more register spills to local memory (which is off-chip), more divergent branches, etc. As with any optimization, you should experiment to see how changes affect the *wall clock time* of the kernel execution. For bandwidth-bound applications, on the other hand, increasing occupancy can help better hide the latency of memory accesses, and therefore improve performance.

3.1.1 Common practices followed

- There are total of 480 cores for the GTX 480. Thus program needs to be able to create more threads in order to keep this GPU busy.
- Thread block size is always a multiple of 32, because kernels issue instructions in warps (32 threads). For example, if you have a block

size of 50 threads, the GPU will still issue commands to 64 threads and you'd just be wasting them.

- For images which are of odd sizes, it is padded so that width becomes multiple of 16. This is essential for triangular smoothing kernel in particular.
- Scalability is measured in terms of input size. Number of threads and other kernel launch configurations are kept same for all input readings.

3.2 Gray Scale kernel

It is a straight forward CUDA kernel. Having enough threads (as many as pixels) will provide good speed-up and only task is to map the thread index to unique pixel.

Image	Sequential	GPU
image00	7.075	0.253
image01	8.069	0.220
image02	18.398	0.355
image03	12.904	0.265
image04	10.548	0.253
image05	9.758	0.264
image06	54.729	0.764
image07	8.507	0.206
image08	38.048	0.533
image09	436.774	5.013
image10	42.579	0.631
image11	7.178	0.240
image12	18.172	0.340
image13	49.099	0.708
image14	43.758	0.611

Table 3.1: Run times (in milliseconds) of gray scale kernel(average of five observations)

3.2.1 AI and scalability

AI of kernel Two multiplications , three additions = 5 operations Three reads, one write = 4 bytes accessed (of global memory) This ratio can also be called as Compute to Global Memory Access ratio or CGMA.

$$CGMA = \frac{5}{4} = 1.25 \tag{3.1}$$

$$AttainableGFLOPs = AI \times 177.4GB/s = 221.75GFLOPs \tag{3.2}$$

Achieved GFLOPs for image00

$$\#ofoperations = 1024 \times 1024 \times 5 \tag{3.3}$$

$$Timetaken = 0.253milliseconds \tag{3.4}$$

$$AchievedGFLOPs = 20.72GFLOPs \tag{3.5}$$

Achieved GFLOPs for image09

$$\#ofoperations = 8192 \times 8192 \times 5 = 335544320 \tag{3.6}$$

$$Timetaken = 5.013milliseconds \tag{3.7}$$

$$AchievedGFLOPs = 66.97GFLOPs \tag{3.8}$$

Scalability:

The kernel is scalable because we are achieving more FLOPs as input image size increases. It means it makes sense to offload this task to the GPU.

3.3 Histogram kernel

This kernel is simple reduction kernel. Block size is 256, as tested to be most convenient and fastest. All threads in a block collaborate in incrementing values at particular bin of histogram. Barrier in form of __ syncthreads() is used so that they all write to global memory synchronously. Atomic addition is used , because there is possibility that two threads might be interested to increment the value of same *bin* of a histogram. Roofline models shows that simple histogram kernel is memory bound. CUDA occupancy calculator and Roofline both suggested that, shared memory could be used for optimization.

Shared memory increases AI (Arithmetic Intensity) of kernel. Thus increases, FLOP count.

For simple histogram kernel, AI is 0.5 (One operation for two bytes of access) For Reduction kernel, AI is 1 (we have one operation per global memory access). Note that when each thread accesses an element from global memory, it increments corresponding *bin* value in the *shared* memory and not in the global memory. The last step in modified histogram example requires that we merge each blocks temporary histogram into the global buffer. Suppose we split the input in half and two threads look at different halves and compute separate histograms. If thread A sees byte 0xFC 20 times in the input and thread B sees byte 0xFC 5 times, the byte 0xFC must have appeared 25 times in the input. Likewise, each bin of the final histogram is just the sum of the corresponding bin in thread As histogram and thread Bs histogram. This logic extends to any number of threads, so merging every blocks histogram into a single final histogram involves adding each entry in the blocks histogram to the corresponding entry in the final histogram. For all the reasons weve seen already, this needs to be done atomically.[3]

After synchronization, all threads write their corresponding value in shared memory to global memory. Thus global memory access is reduced.

Image	Sequential	GPU
image00	1.388	0.133
image01	1.617	0.133
image02	3.690	0.295
image03	2.522	0.235
image04	3.534	1.359
image05	1.969	0.188
image06	15.319	03.178
image07	1.772	0.183
image08	8.050	0.654
image09	123.437	54.040
image10	8.861	1.201
image11	1.572	0.220
image12	3.625	0.332
image13	11.013	1.574
image14	10.119	1.680

Table 3.2: Run times (in milliseconds) of histogram kernel(average of five observations)

3.3.1 Performance of Histogram kernel

The performance of histogram kernel depends on the stride each thread takes to reach next element in the global memory. Coalesced access to global memory ensures the positive results in timing performance. For example, with 100MB of data, we have 104,857,600 bytes of data. We could launch a single block and have each thread examine 409,600 data elements. Likewise, we could launch 409,600 blocks and have each thread examine a single data element. Performance is optimal in between two extremes [3].

Because the number of blocks in not easy input to ask user, command line option (which is optional), is given (argv[2]) which tells about factor by which one should divide the following term : (width * height) / 256. Default value of this option is kept 8.

Experiment was performed on various images and it is concluded that number 8 suits for value of the parameter well. The graphs below are drawn for image00 and image09 from the given dataset.

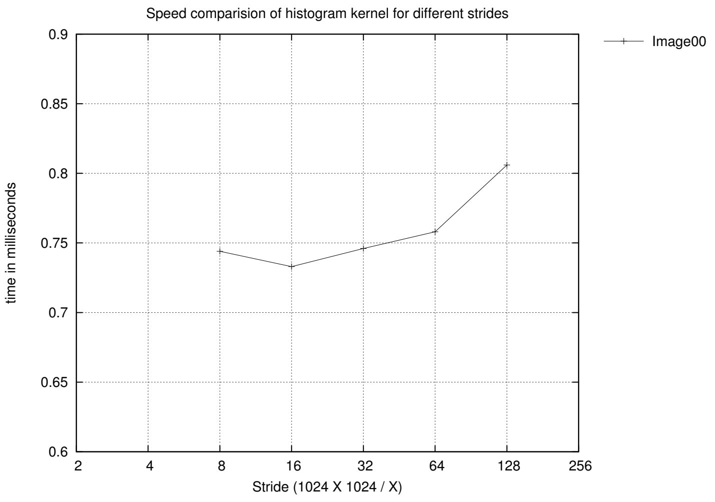

Figure 3.1: Stride and speed graph for image00

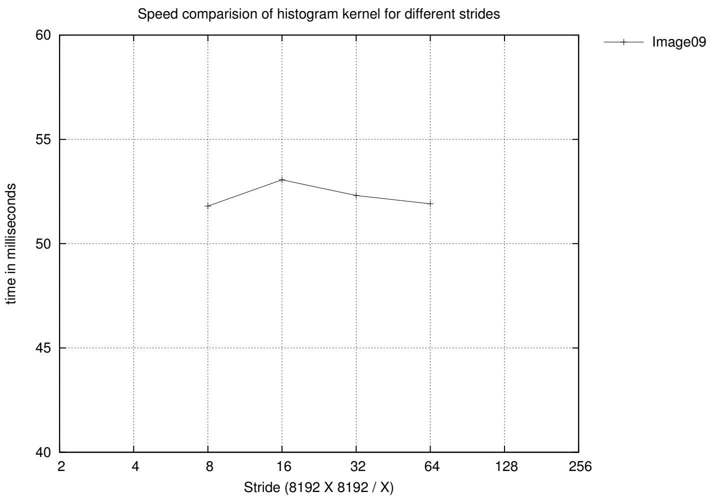

Figure 3.2: Stride and speed graph for image09

3.3.2 AI and scalability

AI of kernel (shared memory version) Two additions = 2 operations (atomic adds) One read, one write = 2 bytes accessed (of global memory) This ratio can also be called as Compute to Global Memory Access ratio or CGMA.

$$CGMA = \frac{2}{2} = 1 \tag{3.9}$$

$$AttainableGFLOPs = AI \times 177.4GB/s = 177.4GFLOPs \tag{3.10}$$

Achieved GFLOPs for image00

$$\#ofoperations = 1024 \times 1024 \times 2 \tag{3.11}$$

$$Timetaken = 0.133milliseconds \tag{3.12}$$

$$AchievedGFLOPs = 15.76GFLOPs \tag{3.13}$$

Achieved GFLOPs for image09

$$\#ofoperations = 8192 \times 8192 \times 2 \tag{3.14}$$

$$Timetaken = 54.040milliseconds \tag{3.15}$$

$$AchievedGFLOPs = 2.53GFLOPs \tag{3.16}$$

Scalability:

The timings show that kernel does not achieve same speed-up over sequential version. But for the largest image, it still does the work in half the time as compared to sequential version. One factor affecting this is *synchronization barriers*. Other possible factors like stalling of warps and their relation to the number of blocks we are launching can also be studied to analyse this further. However we can conclude that kernel can work better than sequential version and also has potential to work for any size of image.

3.4 Contrast Kernel

It is, like gray scale kernel, a simple kernel where achieving maximum performance in parallel versions merely needs to launch enough number of threads.

image	Sequential	GPU
image00	8.342	0.286
image01	9.767	0.296
image02	22.201	0.399
image03	15.259	0.341
image04	12.568	0.300
image05	11.919	0.306
image06	70.968	1.011
image07	10.420	0.292
image08	48.665	0.741
image09	455.822	8.615
image10	53.615	0.812
image11	08.988	0.291
image12	21.959	0.414
image13	63.657	0.932
image14	57.004	0.820

Table 3.3: Run times (in milliseconds) of contrast kernel(average of five observations)

3.4.1 AI and scalability

AI of kernel 1 multiplication, 1 subtraction, 1 division = 3 operations One read, one write = 2 bytes accessed (of global memory) This ratio can also be called as Compute to Global Memory Access ratio or CGMA.

$$CGMA = \frac{3}{2} = 1.5 \tag{3.17}$$

$$AttainableGFLOPs = AI \times 177.4GB/s = 266.1GFLOPs \tag{3.18}$$

Achieved GFLOPs for image00

$$\#ofoperations = 1024 \times 1024 \times 3 \tag{3.19}$$

$$Timetaken = 0.286milliseconds \tag{3.20}$$

$$AchievedGFLOPs = 10.99GFLOPs \tag{3.21}$$

Achieved GFLOPs for image09

$$\#ofoperations = 8192 \times 8192 \times 3 \tag{3.22}$$

$$Timetaken = 8.615milliseconds \tag{3.23}$$

$$AchievedGFLOPs = 25.16GFLOPs \tag{3.24}$$

Scalability:

Although we are achieving more GFLOPs for image having bigger size, some simple optimizations are possible. We can launch the same kernel without *diff* parameter. Launching kernel with less number of parameters can help in speed improvement as it reduces number of local variables per thread. This kernel simply scales for image of any size.

3.5 Triangular smoothing kernel

This kernel is toughest of four. Filtering of image using convolution can be a topic of a case study. This exercise was instructive and a good learning experience. Steps taken to optimize this kernel are presented in following sub-sections:

3.5.1 Constant memory

First and intuitive thought was to access filters from constant memory as opposed to from global memory. Reading from constant memory can conserve memory bandwidth when compared to reading the same data from global memory. There are two reasons [3] why reading from the 64KB of constant memory can save bandwidth over standard reads of global memory:

- A single read from constant memory can be broadcast to other " nearby " threads, effectively saving up to 15 reads.
- Constant memory is cached, so consecutive reads of the same address will not incur any additional memory traffic.

This approach achieved required performance for image09.bmp!

3.5.2 Convolution memory using shared memory

Sequential algorithm for triangular smoothing has limitation that every pixel is read 25 times (except corner pixels). When running on GPU, this means the expensive access of global memory. Thus, we can restrict ourselves to load certain pixels twice. The idea is a block will be responsible for N pixels (say 16 X 16 threads). Threads belonging to the same block can use shared memory which is available per block. Accessing shared memory is extremely fast and highly parallel [2]

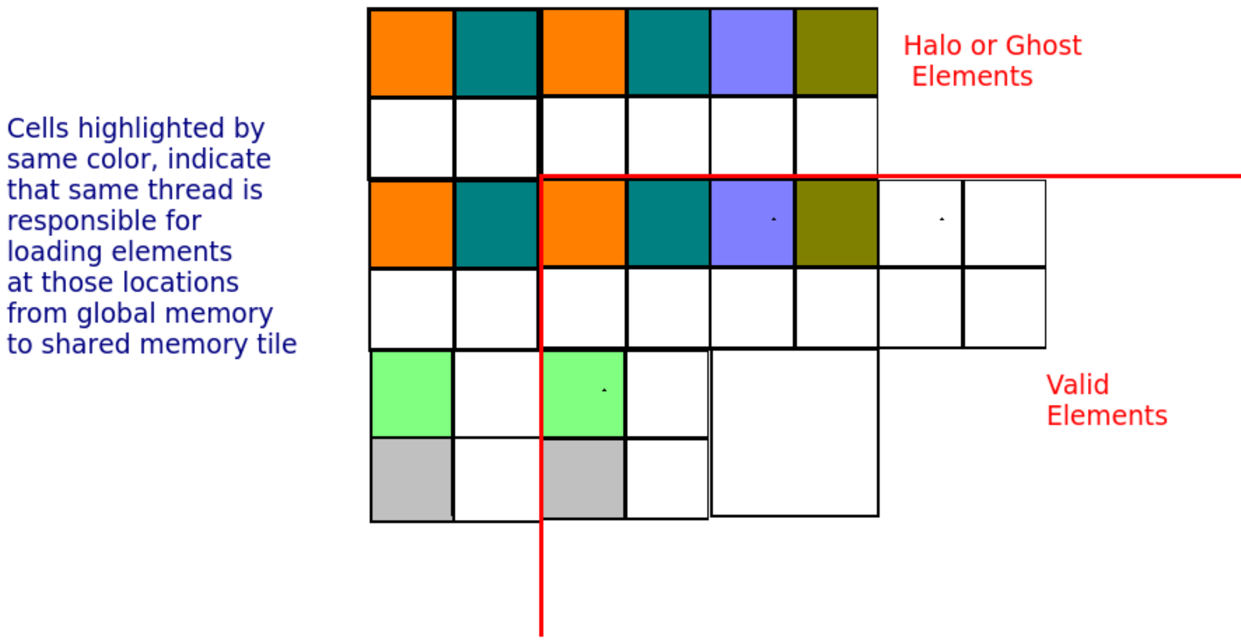
Cells highlighted by same color, indicate that same thread is responsible for loading elements at those locations from global memory to shared memory tile
Halo or Ghost Elements
Valid Elements

Figure 3.3: Logic of convolution kernel

Efforts were required to correctly load tile with ghost and halo elements. Achieving *correct* output using this approach was challenge. Following things are notable in this kernel:

- Giving tile size as template parameter (faster than dynamic shared memory array).
- Use of *short* variables in the kernel code wherever possible.
- Keep track of *valid* pixels of input image (some of them could be 0)

3.5.3 Occupancy calculator

This tool was used to analyse occupancy of kernel. ptxinfo flag of nvcc compiler shows enough information about the kernel:

```
Compiling entry function '_ Z25kernel_ smooth_ convolutionILs400EEvPhS0_
iii' for 'sm_ 20'
Function properties for _ Z25kernel_ smooth_ convolutionILs400EEvPhS0_ iii
0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
Used 19 registers, 800 bytes smem, 60 bytes cmem[0], 8 bytes cmem[16]
```

On the first line, name of our kernel can be seen in mangled form. On the last line, it tells the register and shared memory, constant memory usage. These parameters were given to CUDA occupancy calculator and occupancy of kernel was found to be 100%.

3.5.4 Performance comparison

image	Sequential	GPU(Version 1)	GPU(Version 2)
image00	84.784	01.316	01.536
image01	99.391	01.542	01.797
image02	225.750	03.474	04.060
image03	155.408	02.391	02.780
image04	135.735	02.141	02.492
image05	121.364	01.877	02.201
image06	710.024	10.869	12.619
image07	103.829	01.616	01.903
image08	495.451	07.561	08.785
image09	6331.249	84.231	97.801
image10	545.194	08.363	09.726
image11	91.485	01.427	01.666
image12	223.259	03.428	03.985
image13	646.824	09.981	11.585
image14	568.473	08.687	10.127

Table 3.4: Run times (in milliseconds) of triangular smooth kernel(average of five observations)

Why shared memory version is slower? Barrier synchronization: A block of size 16 X 16 is responsible for 16 X 16 pixels only. Each thread loads one or two elements and waits for all threads to do the same. Though we are increasing locality for each block, we have to use barrier twice. Once for loading the tile by each thread in the block and once for initialization of shared memory tile which tells about validity of data. The need for this is as follows: In sequential version or simple global memory kernel, bounds of image can be used to decide whether a pixel is valid or not. This decides whether to increment *filterItem* variable or not. In case of shared memory implementation, we are assigning 0 for ghost elements and 0 for *padded* elements too. However, it was observed that some of the valid image pixels also contain value 0. For this purpose, use of some memory structure to tell about validity of a pixel is important. This introduces branching in a kernel, which also accounts for slower performance than expected. Improvements are suggested in next section.

3.6 84 hours for 84 milliseconds

Triangular smoothing kernel got *smoothed* due to following access pattern and much better results are achieved! It now takes less than 84 milliseconds for largest image in the set. Kernel version is 3, meaning that it can be launched by giving optional argument 3, as indicated in Usage. This must have taken more than 84 hours to achieve the expected speed-up!

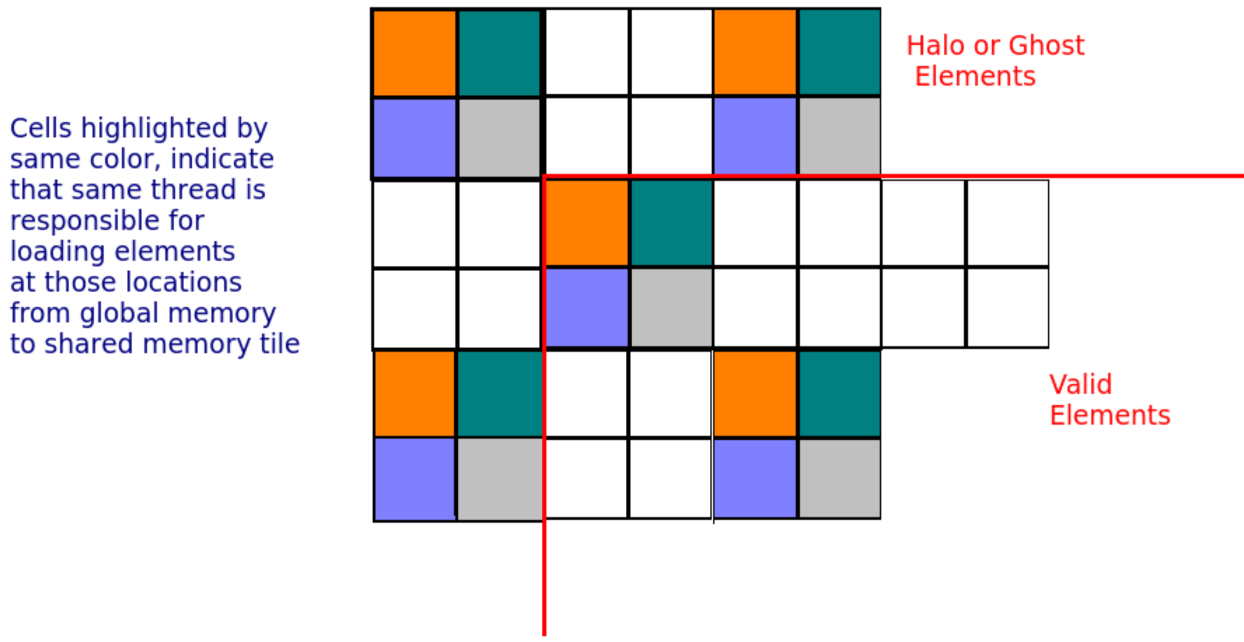
Cells highlighted by same color, indicate that same thread is responsible for loading elements at those locations from global memory to shared memory tile
Halo or Ghost Elements
Valid Elements

Figure 3.4: Logic of convolution kernel with coalesced access pattern

This ensure coalesced access to global memory. *Load of loading* elements from global memory into shared memory tile, is equally distributed among all threads in a block. It also alleviates branching in the kernel code. Results: good timing performance!

3.6.1 AI and scalability

AI of kernel (Smoothing kernel with constant memory optimization) : In nested for loops, number of times global memory accessed, all operations (addition, division) are repeated. Thus they are counted as just 4 operations and 1 access. 3 additions, 1 multiplications, 1 division = 5 operations One read, one write = 2 bytes accessed (of global memory) This ratio can also be called as Compute to Global Memory Access ratio or CGMA.

$$CGMA = \frac{5}{2} = 2.5 \tag{3.25}$$

$$AttainableGFLOPs = AI \times 177.4GB/s = 443.5GFLOPs \tag{3.26}$$

Achieved GFLOPs for image00

$$\#ofoperations = 1024 \times 1024 \times 5 \tag{3.27}$$

$$Timetaken = 1.31milliseconds \tag{3.28}$$

$$AchievedGFLOPs = 4.03GFLOPs \tag{3.29}$$

Achieved GFLOPs for image09

$$\#ofoperations = 8192 \times 8192 \times 5 \tag{3.30}$$

$$Timetaken = 84.23milliseconds \tag{3.31}$$

$$AchievedGFLOPs = 3.99GFLOPs \tag{3.32}$$

Scalability:

Important observation here is that kernel is performing almost similar number of FLOPs per second for images of various input sizes. Thus this kernel can scale very well. Second implementation of smoothing kernel has more potential to scale if following optimizations are also applied:

- Allow rectangular tiles and let one thread populates more than one elements with coalesced access to global memory
- Number of threads per block can be increased from current 256 to 512 or even more to experiment the performance achieved.
- Accessing 32 bits at a time as opposed to 8-bit access for images. Such a kernel is attempted and can be found out in source code of assignment. This kernel is not tested.

3.7 Conclusions

Performance of overall accelerated application is presented for image09.bmp here.

Kernel/memory	Time(for kernel V0)	time (for kernel V2)
rgb2gray (kernel)	5.013	5.014
rgb2gray (memory)	77.267	110.81
histogram1D (kernel)	54.040	53.93
histogram1D (memory)	16.472	24.30
contrast1D (kernel)	8.615	8.60
contrast1D (memory)	35.534	49.7
triangularSmooth (kernel)	84.231	77.13
triangularSmooth (memory)	43.605	66.67
Execution time	643.878	609.89

Table 3.5: Running time (in milliseconds) of GPU version for image09.bmp

Chapter 4

Future Work

Triangular smoothing presented interesting problems to study. Separation of vertical and horizontal smoothing can be analysed and done. Different access patterns and different sizes of tiles can also affect performance. Coding for different access patterns and achieving correct results even in case of padded elements (considering the corner pixels too) can be challenging and interesting task.

References

[1] Samuel Williams, Andrew Waterman, and David Patterson, R̈oofline: an insightful visual performance model for multicore architectures;̇Communications of the ACM 52 (4), pp. 65-76 (April 2009)

[2] Wen-mei W. Hwu, David B. Kirk, P̈rogramming massively parallel processors A Hands-on approach ;̇ nVIDIA and Morgan Kaufmann

[3] Jason Sanders, Edward Kandrot, C̈UDA by example - An introduction to general purpose GPU programming;̇ nVIDIA