Clojure Project Report

Gakuo Patrick

June 12, 2019

1 Overview

In this experiment, we ported a sequential implementation of image convolution to OpenCL. We attempted to gain performance while retaining correctness of the sequential implementation. We have four different kernel implementations. First, there is a basic kernel version that fetches all data from global memory. There is a second version of that improves on the first by storing the filter/-convolution kernel in constant memory. A third version builds on the second by vectorizing the input and output convolution operations. The final version improves on the third by storing input portions into local memory.

In the experiments, we tweaked the local and filter sizes and experimented on each of the four kernel versions mentioned above. We performed each experiment on the CPU and the GPU.

2 Implementation

We have a first implementation where each work-item computes the convoluted value for each pixel. Each work-item fetches input pixels and filter values from global memory. On the computation of the result pixel value, it writes back to the output. The global memory fetches are a huge bottleneck.

In the second implementation, we optimize the convolution by storing the filter in constant memory. This improves the computation since constant memory has lower latency than the global memory. Each filter fetches now happens faster than in the first implementation. However, there are still global fetches per work-item computation for the input, which is a further bottleneck.

The third implementation builds on the second by vectorizing the work-item computations. We use float4 computations, which unrolls the sequential addition in both version one and two described above. The input, filter, and output are of type float4. The type change is necessary to avoid having to interchange types between input, output, filter and computation types which will penalize performance. The bottleneck in this version remains in the input which is located in the global memory.

The fourth and final implementation improves the third implementation by making use of the local storage. We task each work-item with loading a pixel into the local storage. However, since we require additional pixels as the filter overhangs outside the local-area during convolution, we task some work-items to fetch the extra pixels outside the local area. The computation then proceeds as before but now fetches input from the local memory. The local memory fetch is faster than the global memory fetch.

The evaluation section shows that each of the optimizations resulted in a performance boost. The fourth implementation was the fastest while the first was the slowest. All the OpenCL implementations were faster than the sequential implementation. Note that we also add a fourth channel to the input image for more straightforward float4 computations.

3 Evaluation

3.1 Platform description

See the appendix section

3.2 Performance

We performed the below experiments: In each experiment we measure the latency in seconds.

1. **kernel size experiments** - each with kernel version 1 (fastest kernel version), 10 iterations (each iteration with 1o iteration), and local size 16. We performed these experiments for both the OpenCL (CPU and GPU) and sequential implementation.

2. **kernel version experiments** - with 30 iterations (each iterations with 10 iterations) with kernel dimension 5 and local size 16. We did these experiments on the GPU and CPU for only the OpenCL experiments

3. **local size experiments** - with with 30 iterations (each iterations with 10 iterations), kernel dimension 5, kernel version 1.We did these experiments on the GPU and CPU for only the OpenCL experiments

kernel version descriptions: Note all kernel versions have output in the global memory

1. **kernel version 1** - local memory for the input, constant memory for the filter, and vectorized computations

2. **kernel version 2** - global memory for input, constant memory for the filter, and vectorized computations

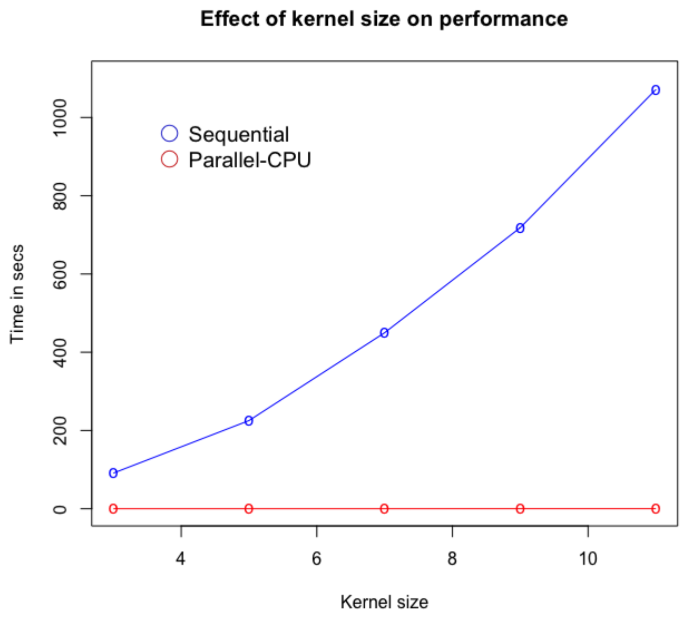
3. **kernel version 3** - global memory for input, constant memory for the filter, and non-vectorized computations

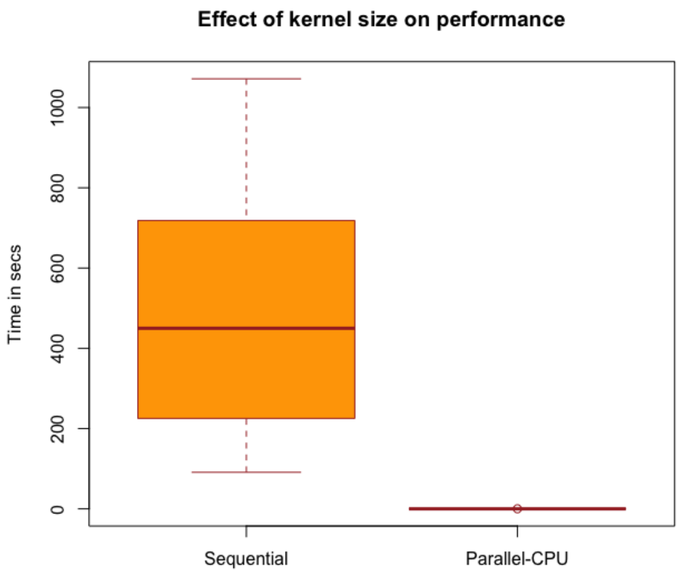
4. **kernel version 4** - global memory for input and filter, and non-vectorized computations

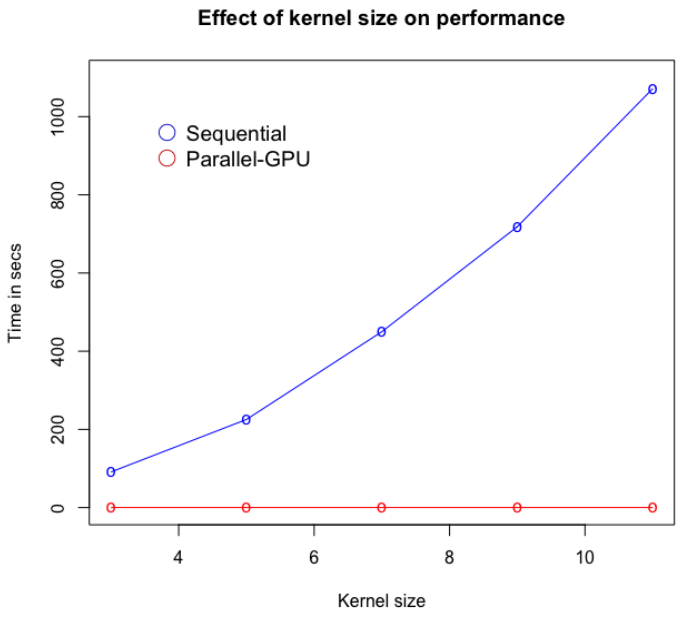
3.2.1 kernel size experiments

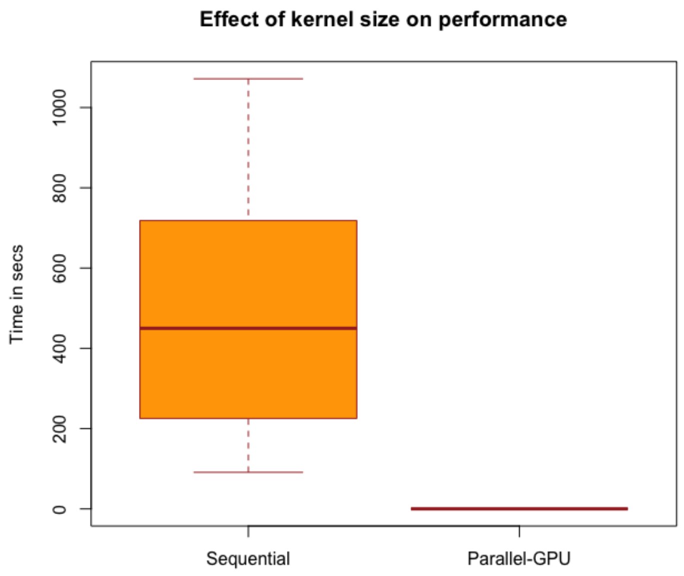
In this experiment, we adjusted the kernel size from 3 to 11 in steps of two.

We observe that the OpenCL implementation outperforms the sequential implementation by a huge difference. That is the sequential implementation registers a much higher latency in seconds. It is also evident that the latency increases with increasing kernel load while that of the OpenCL (kernel version 1) on the CPU and GPU remains relatively constant. OpenCL is massively parallel with each many threads, each per work-item, working on the input.

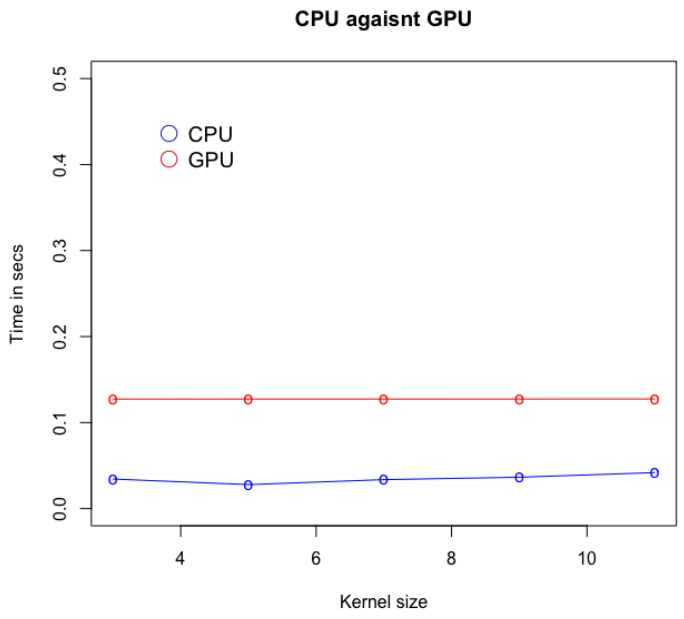


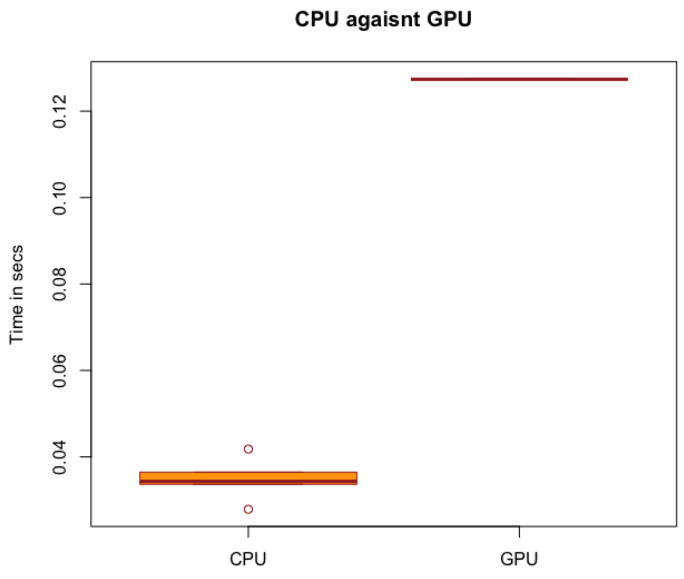

Effect of kernel size on performance
Time in secs
0
200
400
600
800
1000
Sequential
Parallel-CPU


Effect of kernel size on performance
Time in secs
0
200
400
600
800
1000
Sequential
Parallel-GPU
4
6
8
10
Kernel size

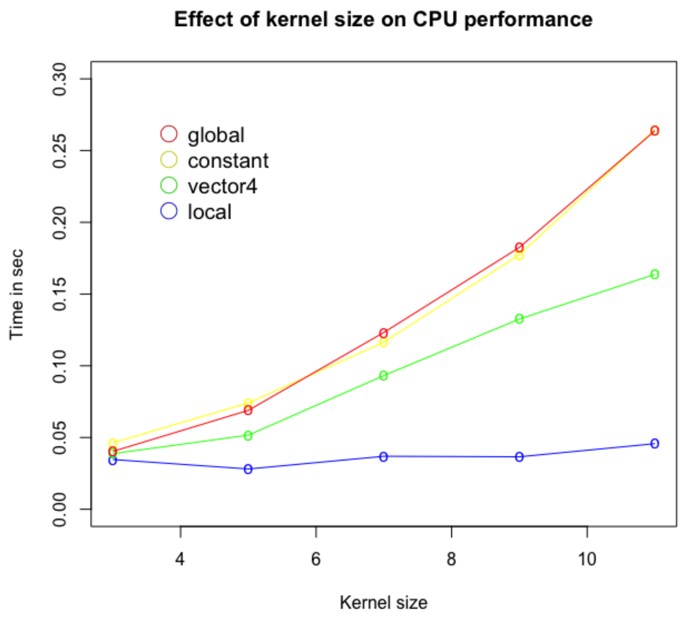


We then compare the performance of the CPU and GPU with increasing threads. We see that the CPU significantly outperforms the GPU. This is the opposite of what we expected as the GPU has more compute units and more local memory than our CPU.





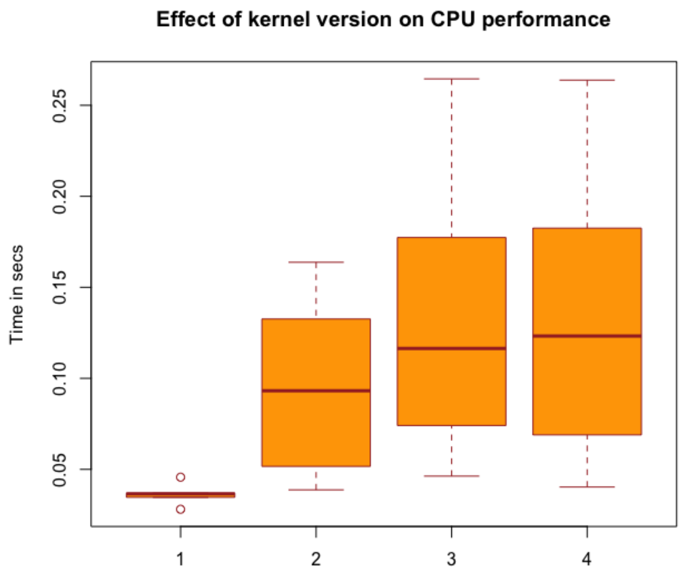
Further, we trace the performance of the CPU with different kernel versions and changing kernel sizes. We noticed that the kernel version 1 (local) latency remains relatively low with increasing kernels. This is because it stores the input in a low local latency memory. The bigger the kernel, the higher the global fetches per pixel convolution. The rest of the implementations suffer the consequences of increased input fetches.

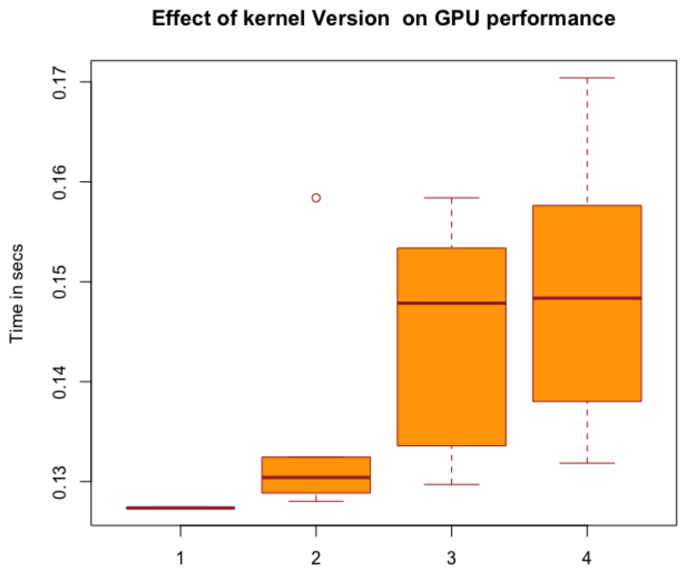


3.2.2 Kernel version experiments

In this experiment we compare the performance of our four kernel versions.

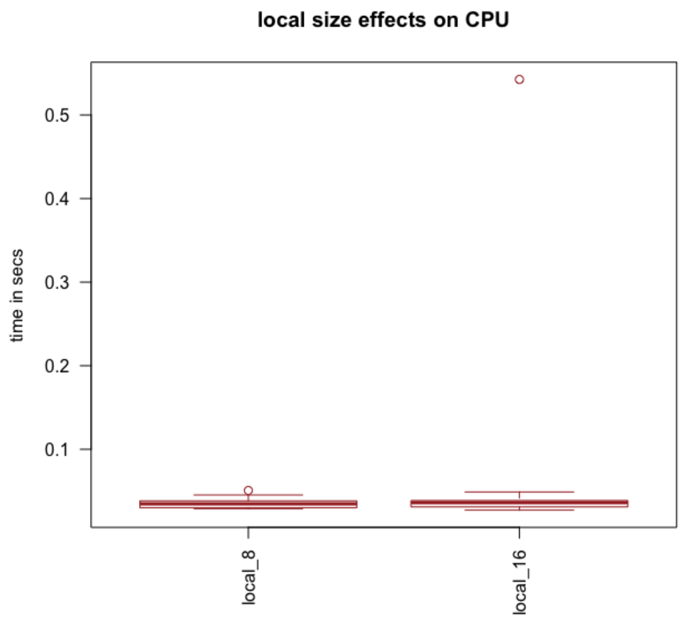
We notice that the kernel version one is significantly faster than all the rest of the versions owing to is the use of all optimization's disused earlier(local memory, constant memory and vectorization). However, out t-test could not significantly differentiate the performance of kernel one from kernel 2 and neither would it differentiate kernel two from either of kernel 3 and kernel 4. This was the case for both CPU and GPU. We note that the application of all the optimization techniques mentioned earlier stated helps set kernel version 1 from all the other kernels. It is also possible that a suitable load would have differentiated the other kernels.

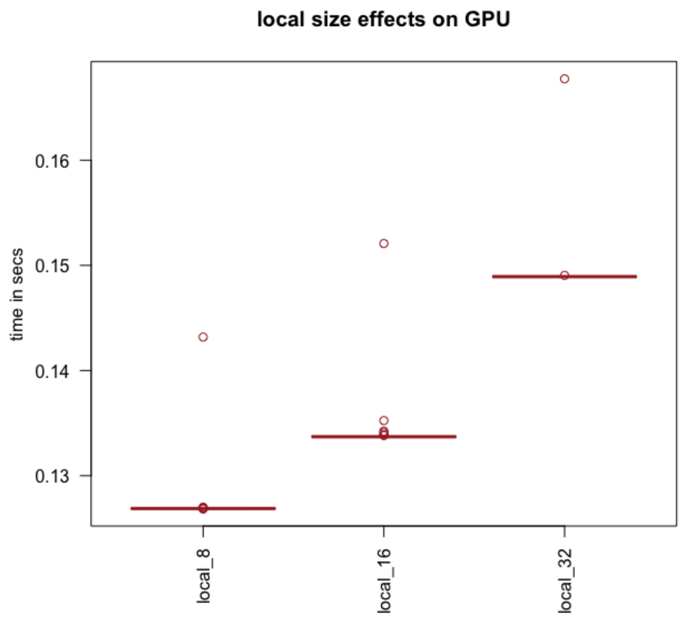

Effect of kernel version on CPU performance
Time in secs
0.05
0.10
0.15
0.20
0.25
1
2
3
4


Effect of kernel Version on GPU performance
Time in secs
0.13
0.14
0.15
0.16
0.17
1
2
3
4

3.2.3 Local size experiments

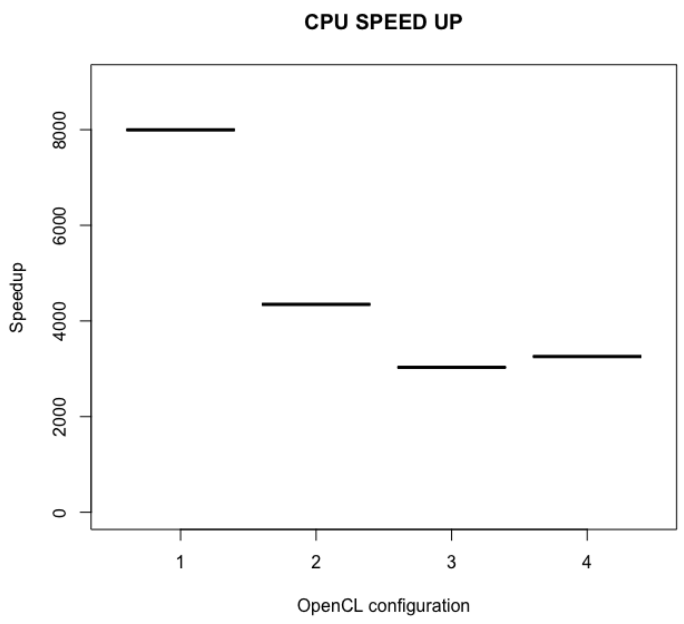
In this experiment, we adjusted the group sizes and observed the effect on the performance of kernel version 1 on the GPU and CPU. We noted that the kernel size had a significant effect on the GPU's performance, but it did not significantly affect the CPU. The CPU work item dimensions explain this result (see the appendix section). The larger the local-size, the less the pixel data overlap between the work-groups and the higher the performance we registered in the GPU(Reda, 2012).

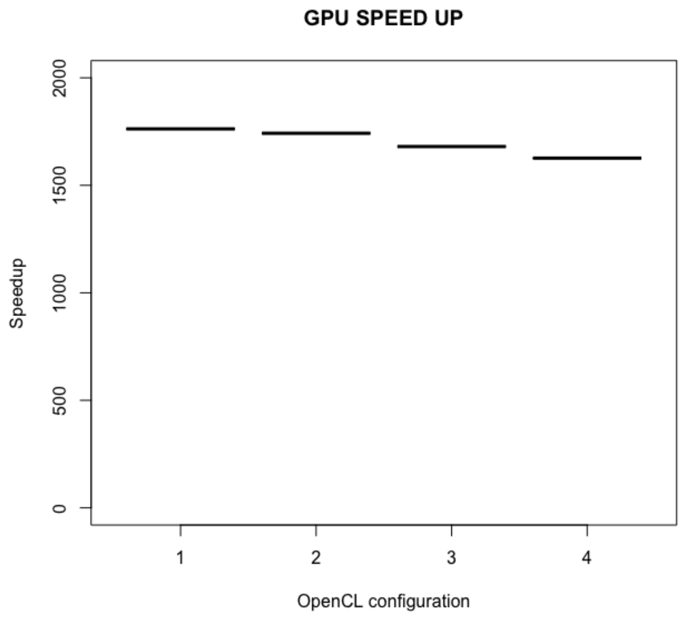




3.2.4 speed-up comparison

Finally, we computed the speed-up obtained from both the CPU and GPU relative to the relative implementation. As aforementioned, the CPU registered a surprisingly higher speed. The GPU might have been busy, or the lack of work-item task tuning might have led to higher latency as compared to the CPU. Each of the speed-ups shown below for the CPU and GPU represents each kernel version's speed up.


CPU SPEED UP
Speedup
0
2000
4000
6000
8000
1
2
3
4
OpenCL configuration


GPU SPEED UP
Speedup
0
500
1000
1500
2000
1
2
3
4
OpenCL configuration

3.3 Correctness

To check the correctness, we saved back our resulting image and did a visual inspection. The image exhibited some discrepancies that are related to indexing. We then called the sequential implementation and compared our results against its results. We found out that our implementation was not exactly correct, but its results fell within a tolerance.

4 Insight questions

4.1 If you had more time, which optimizations would you apply to your OpenCL implementation, and why do you think they are a good idea?

In this experiment, we used local memory, constant memory, and vectorization. We also suspect that higher vectorization beyond float4 might unroll the filter loop, which can make our implementations way faster. We also added padding in a way that retained memory alignment(multiples of 16 and 32). We also ensured memory coalescing by ensuring adjacent work-items accessed adjacent work-items in the global and local memory. We, however, could do better tuning by tweaking work-items loads for better resources utilization. Additionally, we could have made use of the image data types supported by OpenCL. These types offer further access to optimized access of the texture memory.

4.2 Consider the portability of your OpenCL algorithms. For example, the Iris Pro GPU in an Intel Core i7 has 64 KB of local memory and a maximum of 512 work-items in a work-group. On the other hand, an AMD Radeon R9 M370X has 32KB of local memory, and only supports up to 256 work-items in a work-group. Explain how these different possible specifications could harm or improve the performance of your implementation

Iris Pro GPU has higher local memory than our CPU but a lower work-group size. While we can store more data in the local memory, we will have to do with lower parallelism. However, we could make more utilization of each work-item. On the other hand, AMD Radeon has lower local memory and lower work-group size. The is a double bottleneck might make performance impossible to replicate with high loads.

References

Reda, K. (2012). A study of opencl image convolution optimization. *mreda@uic.edu*.

Table 1: Platform description

Spec name	**value**
plat 0	
Platform: Apple	
Vendor: Apple	
Version: OpenCL 1.2 (May 24 2018 22:33:53)	
Number of devices: 3	
device 0	
Name	Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz
Version	OpenCL C 1.2
Max. Compute Units	12
Local Memory Size	32.0 KB
Global Memory Size	16384.0 MB
Max Alloc Size	4096.0 MB
Max Work-group Total Size	1024
Cache Size	9437184
Max Work-group Dims	(1024 1 1)
device 1	
Name	Intel(R) UHD Graphics 630
Version	OpenCL C 1.2
Max. Compute Units	24
Local Memory Size	64.0 KB
Global Memory Size	1536.0 MB
Max Alloc Size	384.0 MB
Max Work-group Total Size	256
Cache Size	0
Max Work-group Dims	(256 256 256)
device 2	
Name	AMD Radeon Pro 555X Compute Engine
Version	OpenCL C 1.2
Max. Compute Units	12
Local Memory Size	32.0 KB
Global Memory Size	4096.0 MB
Max Alloc Size	1024.0 MB
Max Work-group Total Size	256
Cache Size	0
Max Work-group Dims	(256 256 256)