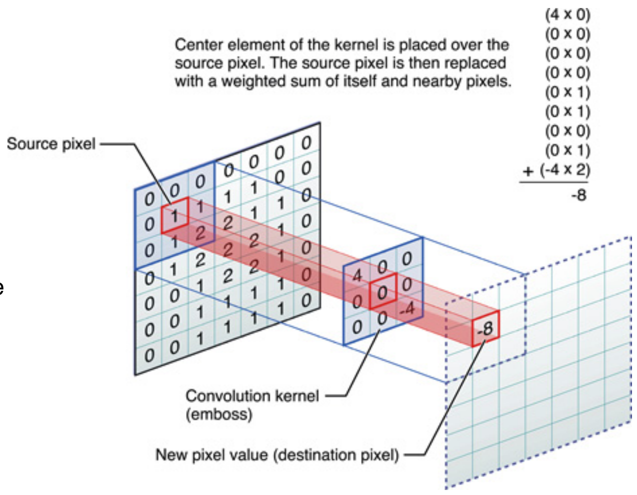
A study of OpenCL image convolution optimization

Khairi Reda
mreda2 -at- uic -dot- edu
http://www.evl.uic.edu/kreda/gpu/image-convolution/
(with minor adaptions by Adrian Clark)

Source code can be downloaded from here: clBenchmark.tar.gz

Overview


Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution kernel (emboss)
New pixel value (destination pixel)

A frequent technique in image processing envolves calculating an output image from the input image by means of a convolution filter. The convolution filter is a square 2D matrix with an odd number of rows and columns (typically 3x3, 5x5, 15x15, etc...). When the input image is processed, an output pixel is caluclated for every input pixel by mixing the neighborhood of the input pixel according to the filter. This mixing usually takes the form of multiplying intensity values of the neighborhood pixels with the terms specified in the filter. Effects such as gaussian blurring and edge detection can be easily described in terms of a filter convolution.

Image convolution comprises, at its lowest level, a large number of independent floating point multiplications and summations. As such, the process conforms to the single instruction, multiple data (SIMD) paradaigm. Therefore, image convolution can be sped up leveraging the todays modern GPUs, which are designed to execute a huge number of floating point operations in parallel.

This document describes a study of optimization techniques for image convolution in OpenCL. I describe a series of OpenCL programs (kernels) with increasing level of optimization to attain a faster convolution performance, and discuss the attained improvement. First, I describe a 'global memory' implementation where every thread directly accesses the global memory without coordination with other threads. This implementation is slowly refined and optimized to get a reasonable performance. After that, I present a 'local memory' implementation that minimizes access to global memory by sharing data between threads. This implementation conforms to standard best-practices when writing OpenCL programs. The performance results of the two implementations along with the refined revisions are illusrated and disucssed.

Terminology and Assumptions

- *W:* width of input image.
- *H:* height of input image.
- *FS (Filter size):* an odd integer representing the size of the convolution filter. For example, a filter size of 3 denotes a 3x3 convolution filter.
- *FA (Filter area):* defined as *FS*FS*
- *HFS (Half Filter Size):* defined as *floor(Filter size / 2)*. For example, a filter size of 3 has a half filter size of 1.
- In this document, we assume that the image is a 4 channel per pixel (red, green, blue, and alpha). Each channel is a 32-bit float. The total is 128-bit per pixel (16 bytes).

Global memory

Note: The performance results shown in the following two sections illustrate image convolution on a 512 x 512 image, unless otherwise noted.

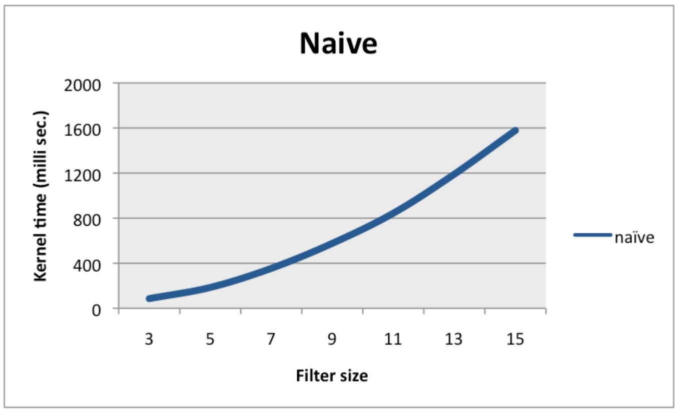
1. Naive implementation

A straightforward way to parallelize image convolution is by launching W x H threads. Each thread has the responsibility to grab 2 x FA pixels from the global memory per thread. The first FA comes from the input image, whereas the second FA is the contents of the convolution filter. Since every pixel requires 4 channels (red, green, blue, and alpha), and assuming the convolution filter specifies different convolution terms for each channel, we need to access 8 components for each pixel. This brings the number of global memory reads to (8 x FA). A naive implementation might also use global memory in the same manner as arrays are used in C by reading and writing directly to it. This adds additional After performing the convolution, we have to write 1 pixel to global memory again to store the result. This is an additional 4 global memory writes per pixel. The grand total is *(8 x FA) + 4* per thread.

Here's the OpenCL code of the kernel. Note that there's a check of whether we are within HFS boundary of image border that we need to perform before doing the computation (since the filter convolution is not valid in those regions). We omit this test from the code listings here for clarity.

```
__kernel void convolute(
        const __global float * input,
        __global float * output,
        __global float * filter)
{
        int rowOffset = get_global_id(1) * IMAGE_W * 4;
        int my = 4 * get_global_id(0) + rowOffset;

        int fIndex = 0;
        float sumR = 0.0;
        float sumG = 0.0;
        float sumB = 0.0;
        float sumA = 0.0;

        for (int r = -HALF_FILTER_SIZE; r <= HALF_FILTER_SIZE; r++)
        {
                int curRow = my + r * (IMAGE_W * 4);
                for (int c = -HALF_FILTER_SIZE; c <= HALF_FILTER_SIZE; c++, fIndex += 4)
                {
                        int offset = c * 4;

                        sumR += input[curRow + offset  ] * filter[fIndex  ];
                        sumG += input[curRow + offset+1] * filter[fIndex+1];
                        sumB += input[curRow + offset+2] * filter[fIndex+2];
                        sumA += input[curRow + offset+3] * filter[fIndex+3];
                }
        }

        output[my    ] = sumR;
        output[my + 1] = sumG;
        output[my + 2] = sumB;
        output[my + 3] = sumA;

}
```

The following chart shows the execution time of this kernel.

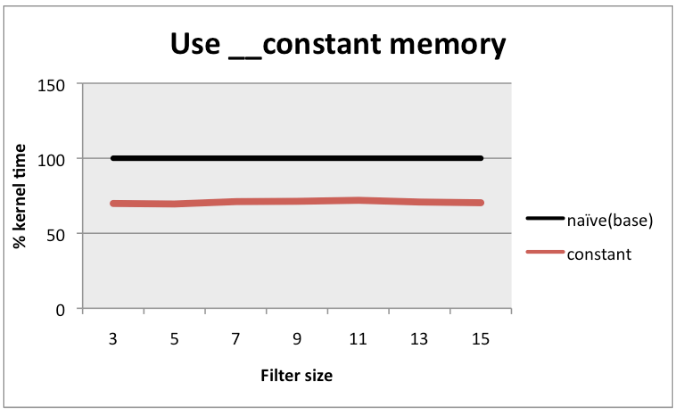


2. Using constant memory

One will notice that the filter does not change during execution. Therefore, instead of putting it in global memory, we can store it in constant memory. The constant memory is faster than global memory as it is pre-cached before kernel launch. It is however limited to 64K, which is enough to fit even large filters. The only change is making the modifier for the `filter` pointer a `__constant` instead `__global` in the kernel header

```
__kernel void convolute(
        const __global float * input,
        __global float * output,
        __constant float * filter)
```

We will still need (8 x FA + 4) memory accesses. However, 4 x FA of those will come from `__constant`. The following chart illustrates the performance of this kernel relative to the naive one. We can see that the new kernel executes at around 70% of the time of the naive kernel, regardless of filter size. This optimization is suggested in [3].



3. Using float4

One can also see that the per channel-mulitplication can be parallelized. Fortunately, there's an easy way to do that. The GPU has special hardware for performing vector operations in parallel, including vector multiplication. To take advantage of this, we have to change our data from `float` to `float4`. Again, this is very easy to do, and does not need any change to the main program. This has three benefits:

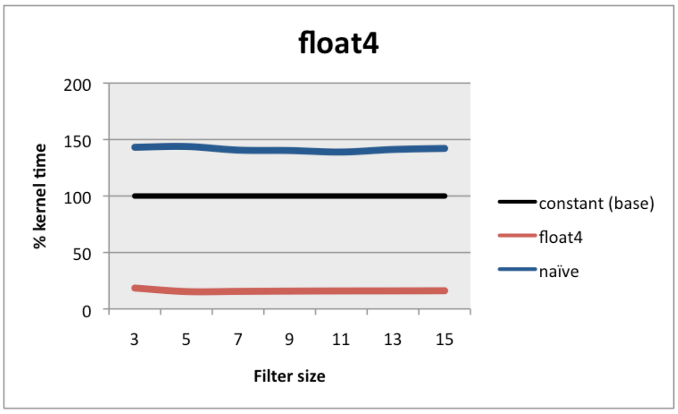
1. Instead of 4 accesses to global memory (1 per channel), we only need 1 now, since a float4 can be accessed with a single read.
2. The per-channel multiplication will happen in parallel, taking advantage of the GPU's vector arithmetic optimization

3. This saves time needed by the GPU to offset to individual channels and extracting 32-bit from a 128-bit memory block.

The new kernel looks like:

```
__kernel void convolute(
        const __global float4 * input,
        __global float4 * output,
        __constant float4 * filter
)

int fIndex = 0;
float4 sum = (float4) 0.0;

for (int r = -HALF_FILTER_SIZE; r <= HALF_FILTER_SIZE; r++)
{
        int curRow = my + r * IMAGE_W;
        for (int c = -HALF_FILTER_SIZE; c <= HALF_FILTER_SIZE; c++)
        {
                sum += input[curRow + c] * filter[fIndex];
                fIndex++;

        }
}
output[my] = sum;
```

The following chart shws the performance of the kernel relative to the two previous versions. This causes about 5x improvement in speed relative to the __constant version, and about 7x improvement over the naive version.



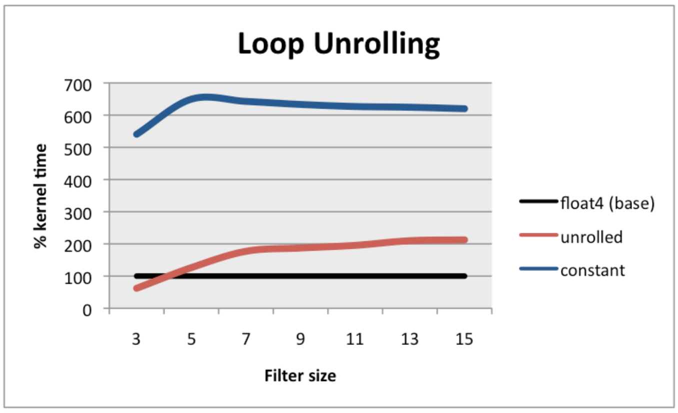
4. Unrolling the loop

Loops have an overhead. The loop condition needs to be tested and the counters need to be incremented at every iteration. While this overhead is negligible in the CPU world, it makes a difference in the GPU. The reason being is that the speed of executing a single instruction is slower in the GPU, which could make the loop control overhead a significant portion of the kernel execution. In our filter convolution code, the two loops can be easily unrolled. However, since the loops are responsible for covering the filter, we have to generate a separate kernel and compile it individually for every filter size that we need to operate on. Luckily, this could be easily done using a stub kernel code with a place holder for the unrolled loop. The CPU generates an unroleld multiplication code, replacing the stub before passing the code to the OpenCL compiler.

The following code shows an unrolled version of the loop for a 5x5 filter. Notice that we no longer need the `fIndex` variable, and that all the offsets that can be calculated in advance are already put in place:

```
        float4 sum = (float4) 0.0;

        // unrolled loop

        sum += input[my - 2050] * filter[0];
        sum += input[my - 2049] * filter[1];
        sum += input[my - 2048] * filter[2];
        sum += input[my - 2047] * filter[3];
        sum += input[my - 2046] * filter[4];
        sum += input[my - 1026] * filter[5];
        sum += input[my - 1025] * filter[6];
        sum += input[my - 1024] * filter[7];
        sum += input[my - 1023] * filter[8];
        sum += input[my - 1022] * filter[9];
        sum += input[my -    2] * filter[10];
        sum += input[my -    1] * filter[11];
        sum += input[my       ] * filter[12];
        sum += input[my +    1] * filter[13];
        sum += input[my +    2] * filter[14];
        sum += input[my + 1022] * filter[15];
        sum += input[my + 1023] * filter[16];
        sum += input[my + 1024] * filter[17];
        sum += input[my + 1025] * filter[18];
        sum += input[my + 1026] * filter[19];
        sum += input[my + 2046] * filter[20];
        sum += input[my + 2047] * filter[21];
        sum += input[my + 2048] * filter[22];
        sum += input[my + 2049] * filter[23];
        sum += input[my + 2050] * filter[24];
```

Here's the performance of the unrolled kernel relative to the two previous ones:



The surprising result here is that the unrolled loop performs worse most of the time; it is only better for a tiny 3x3 filter. As the filter size grows, it performs about twice as slow as the float4 version. This suggest that the OpenCL compiler is doing some optimization on the loop, perhaps unrolling it in a way that produces a more-efficient memory access pattern than our unrolled-version. The lesson here is don't bother to unroll loops, unless they are really tiny, in which case their overhead is big relative to their body.

Interestingly, when the multiplication order in the above unrolled code listing is randomized, the performance of the kernel improves slightly. Nevertheless, the unrolled version remains slower than the original looped version for larger filter sizes.

5. Using #define macros

Notice that we don't declare variables such as `HALF_FILTER_SIZE` and `IMAGE_W` in the kernel code. The reason we did not declare them is that we 'passed' them in as defined macros when compiling the OpenCL kernel. This is done by passing them in the compiler option line from the main program. Here's an example:

```
"-D IMAGE_W=512 -D IMAGE_H=512 -D FILTER_SIZE=5 -D HALF_FILTER_SIZE=2 -D TWICE_HALF_FILTER_SIZE=4 -D HALF_FILTER_SIZE_IMAGE_W=1024"
```

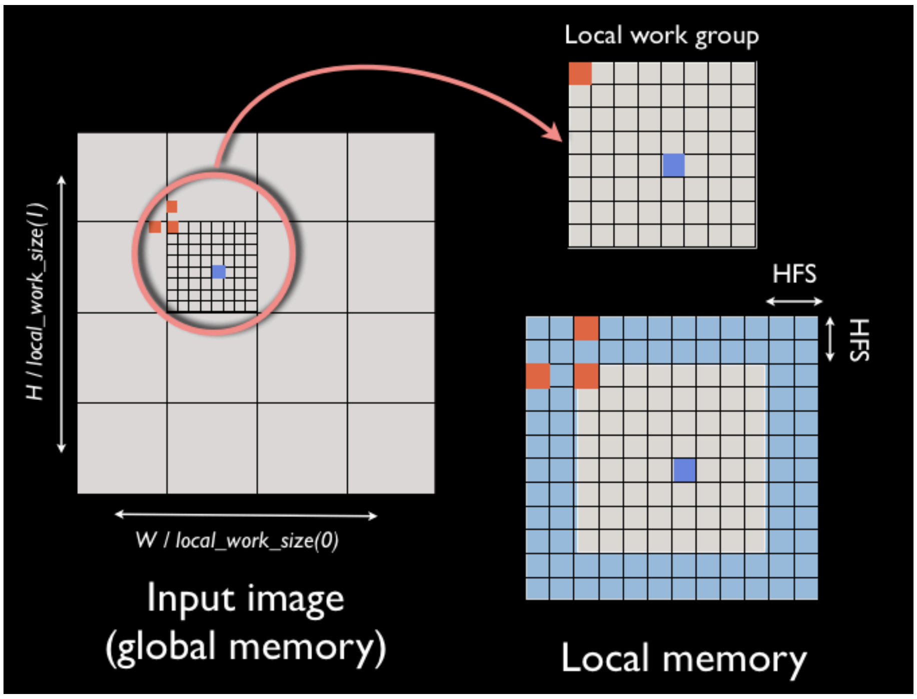
Notice that we also calculate some offsets such as `HALF_FILTER_SIZE_IMAGE_W` which is W * HFS. Passing those values as defined macros saves the time needed to calculate them in the kernel. Additionally, because they are declared as literal values, using them in the OpenCL kernel is more efficient than loading them from a variable. The downside of this is that a separate kernel has to be compiled for every combination of those values. All the kernels illustrated in this document employ this optimization.

Local memory

Having optimized the computation, it is now time to turn attention to global memory accesses. Global memory is the slowest memory in the GPU. Therefore, access to it need to be minimized. By studying the code above and taking into account that it is executing W x H times, one can deduce that there must be a big amount of redundancy in global memory accesses. This is because every thread needs to read an FA neighborhood of pixels around it. However, threads that are responsible for adjacent pixels in the image will have overlapping FA neighborhood. However, the current global memory implementation does not take advantage of this overlap.

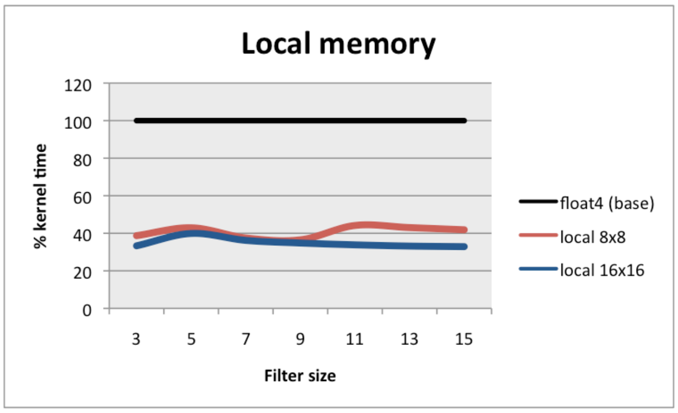
To alleviate this redundancy, we can make the threads 'cooperate' and together grab the pieces of data they need, splitting the work among them, before the computation begin. The data grabbed from global memory will be put in local memory before the computation. Once all threads within a local work group are done copying data, the fun can begin. The computation will be identical (each thread performing a convolution over FA), only this time it will happen on data in the local memory, which is much faster than global memory.

Each thread in the local work group will grab one pixel at least from global memory and copy it to local memory. If a thread is located within a HFS of the border of the local work group, it needs to grab additional data. The following figure illustrates this idea. For example, the red thread needs to grab two additional pixels beside its own (a total of three) in order to cover its filter. The blue thread on the other hand needs to grab just one pixel from global memory. The idea for this scheme came from [1].



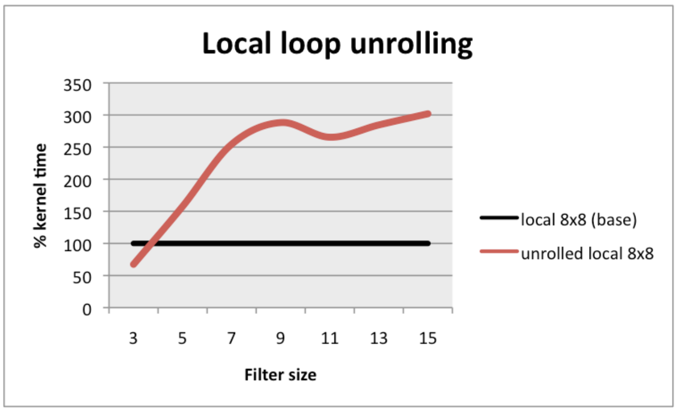
The above scheme reduces redundancy in global memory reads. It does not completely eliminate it, since for every workgroup we need to cache an extra HFS around the border of workgroup in each direction (the blue shaded region of the local memory in the above diagram). This region will overlap for adjacent workgroups, causing redundant reads from global memory. However, because there's no way of inter-workgroup communication, this redundancy is unavoidable. The redundancy is proportional to the filter size, and disproportional to workgroup size, with larger workgroups incuring less redundancy.

The following chart shows the performance of the local memory implementation using a 8x8 and a 16x16 local workgroup sizes. When compared to the optimized versino of the global memory implementation, the local memory version finishes with about 35% of the time for the 16x16 local work group. The 8x8 local work group performs a bit slower with larger filter sizes. With local memory, we get about 3x improvement over the `float4` global memory version.



Unrolling the local loop

One might be tempted to unroll the loop in the local memory implementation.

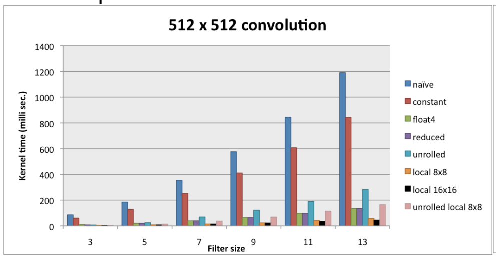


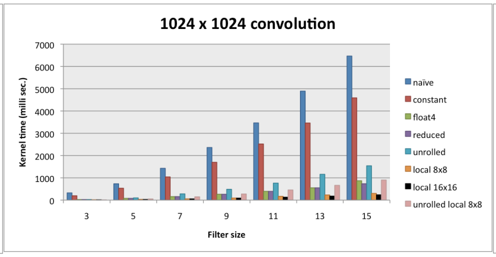
Again, the result are worse, with up to 3x slower for larger filter sizes!

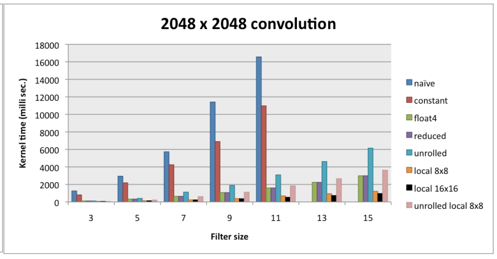
Performance results

The following shows performance for the above OpenCL kernels for a (512 x 512), (1024 x 1024), and (2048 x 2048) image sizes with filter sizes of 3 ... 15. The GPU used in the tests is a *NVidia GeForce 9600M G* with 512 MB of dedicated graphic memory. The host system is a three years old 15-inch MacBook Pro with 2.53 GHz dual-core processor and 4 GB of RAM.

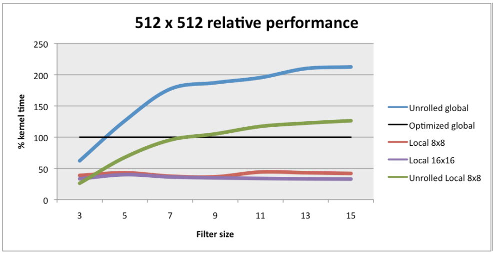
Absolute performance of all kernels in milliseconds

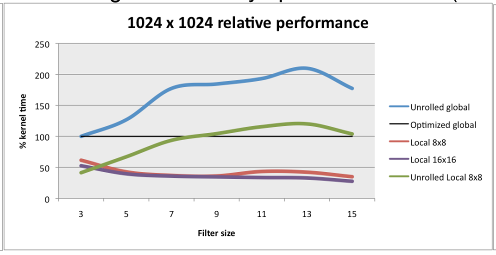


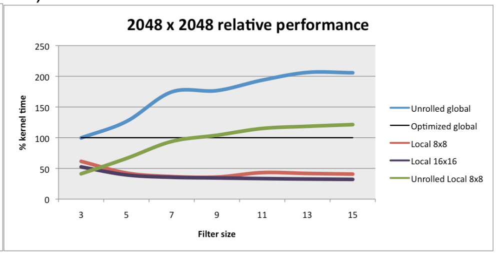




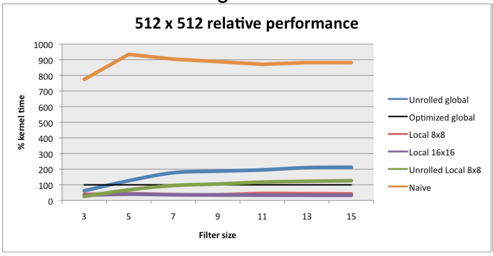
Performance of a selected subset of kernels relative to the global memory optimized version (`float4`) shown in a black line at 100%.

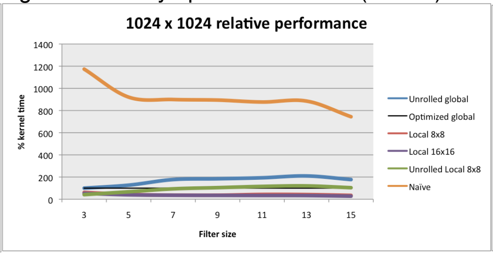


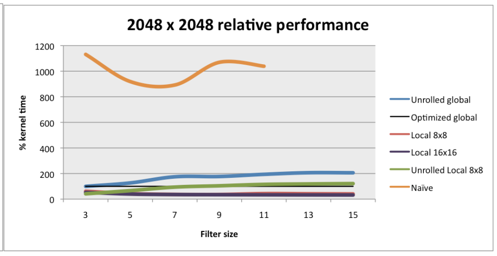




Performance including naive kernel relative to the global memory optimized version (`float4`) shown in a black line at 100%.







Discussion

The fastest performing code used a local memory implementation with a 16x16 local workgroup size. The only exception is the local memory implementation with an unrolled loop and a filter size of 3x3. The worst performing version was naive global memory implementation.

Overall, with the series of optimization discussed in this document, we have managed to improve by performance of the OpenCL image convolution by a factor of **25x** for a 512x512 image (from the naive version to the local memory with 16x16 workgroup version).

For an image size of 1024x1024, the improvement is about **60x** for a filter size of 3x3. This speedup factor is about **44x** for a 15x15 filter size.

Finally, for an image size of 2048x2048, there was an improvement of **55x** for a filter size of 3x3, and a **50x** improvement for a filter size of 11x11. I could not get data on the performance of the naive and `__constant` kernel for a 13x13 and a 15x15 filter sizes. For a reason I could not figure out, the two kernels kept crashing under these two conditions.

Local memory consideration

The 16x16 local workgroup performs slightly better than the 8x8 local workgroup. This is mainly because a 16x16 local workgroup reduces the redundancy in global memory access as described above. However, a 16x16 workgroup needs more local memory. Precisely, the local memory requirement is *(2 * HFS + local_work_size(0)) * (2 * HFS + local_work_size(1))* pixels per workgroup. For a 16x16 workgroup with a filter size of 15x15 and a 4 32-bit channels per pixel (a total of 16 bytes per pixel), this translates to 14,400 bytes of local memory. Since the local memory is limited to 16K, this is pretty much the maximum filter size-workgroup combination that we can handle with the current generation of hardware. A 8x8 workgroup can handle a maximum filter size of 49x49, albiet at a higher global memory access overhead than a 16x16 workgroup.

References

1. Case study: High performance convolution using OpenCL __local memory
2. Image Convolution using OpenCL: a step-by-step tutorial
3. Optimization Techniques: Image convolution

Last update: April 30, 2012