Comparison of Image Convolution on Heterogeneous Architectures

Patrick Geneva

December 15, 2017

1 Problem Definition

We look to investigate the performance differences on different architectures in the application of image convolutions. Performing convolutions is a common practice in computer vision for apply filters and calculating image gradients. For example, a filter can be used to blur and image, then calculate its image derivatives to extract key features or edges in the image. Thus we are motivated to investigate how this building block of many computer vision algorithms performs on different architectures.

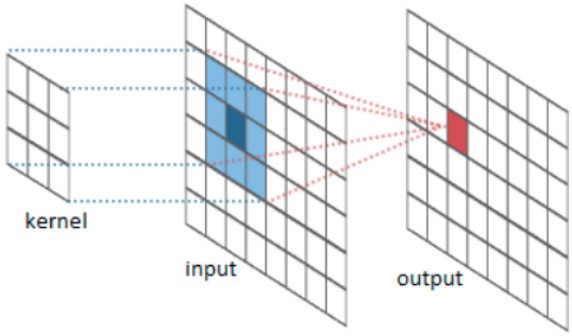


Figure 1: Pictorial explanation of how 2D image convolution is applied.

To provide a fair comparison all algorithms are implements in C++ and are a "stripped down" version of a full algorithm. We look at a 2D image convolution that is multiplying a small kernel matrix to a larger input image. Given a pixel location in the input image, the kernel is applied to the surrounding local pixels using element-wise multiplication. The resulting multiplications are then summed and stored in the final output image. This process is repeated for each pixel in the incoming image as seen above. An example result is shown below:

(a) Input image that will have a kernel applied to it

(b) Outputted image after convolution with a kernel

2 Convolution Implementations

In this section we will present simple pseudo code representations of the implemented convolutions. We will have three different implementations: single threaded, multi-threaded using OpenMP, and GPU accelerated using CUDA. For each we will present the small changes needed to adapt the base code into its parallelizable format. We note here that we are capable of parallelizing out code since the outer for loops are independent.

```
// Loop through in image
for(int i=0; i<rows; ++i) {
    for(int j=0; j<cols; ++j) {
                double sum = 0.0;
                // Sum the image*kernel
                for(int nn=0; nn<krows; ++nn) {
                        for(int mm=0; mm<kcols; ++mm){
                                int ii = i + nn - krows/2;
                                int jj = j + mm - kcols/2;
                                sum += imageIn[ii][jj]*kernel[nn][mm];
                        }
                }
                // Store the result
                imageOut[i][j] = sum;
        }
}
```

The above pseudo code single threaded implementation shows how the based method works. We loop through all pixels in the input image and for each pixel sum the element-wise multiplication with the given kernel. We start in the top left of the kernel and move our way row-wise through it. We then store this sum in the final image out matrix that contains the resulting image (note that we have omitted bound checks and other smaller details for clarity here).

Next we move to multi-thread this code using OpenMP. It is simple to do so since we have independent loops. We use the "collapse" operator to collapse the two outer loops into a single loop and ask OpenMP to parallelize this code. We can see below on line 2 the inserted pragma into the C++ code.

```
1  // Loop through in image
2  #pragma omp parallel for collapse(2)
3  for(int i=0; i<rows; ++i) {
4          for(int j=0; j<cols; ++j) {
5                  double sum = 0.0;
6                  // Sum the image*kernel
7                  for(int nn=0; nn<krows; ++nn) {
8                          for(int mm=0; mm<kcols; ++mm){
9                                  int ii = i + nn - krows/2;
10                                 int jj = j + mm - kcols/2;
11                                 sum += imageIn[ii][jj]*kernel[nn][mm];
12                         }
13                 }
14                 // Store the result
15                 imageOut[i][j] = sum;
16         }
17 }
```

To parallelize the code using CUDA we create a kernel that will be launched on the GPU device. We also note that one could use OpenMP to target the GPU device instead of creating a CUDA kernel. We launch this kernel in sets of 32x32 blocks and then calculate the needed grid size from the number of pixels needing the operation divided by this 32x32 (1024) threads inside of a single block. The input image and kernel is first allocated and the copied onto the device. From there the kernel is launch and the final output matrix is copied back to the host. Seen below it can be seen that we calculate the i and j pixel locations using the standard thread and block id variables.

```
1  // Calculate loc from block/thread id
2  int i = blockIdx.x*blockDim.x + threadIdx.x;
3  int j = blockIdx.y*blockDim.y + threadIdx.y;
4
5  double sum = 0.0;
6  // Sum the image*kernel
7  for(int nn=0; nn<krows; ++nn) {
```

```
        for(int mm=0; mm<kcols; ++mm){
                int ii = i + nn - krows/2;
                int jj = j + mm - kcols/2;
                sum += imageIn[ii][jj]*kernel[nn][mm];
        }
}

// Store the result
imageOut[i][j] = sum;
```

It should be notes that we can accelerate the above code by leveraging shared memory. All threads use the kernel thus this information can be copied into the local shared memory for all threads to use inside of a block. We perform that as follows:

```
// Copy to shared memory
__shared__ double skernel[KERNEL_SIZE*KERNEL_SIZE];
// Only have the subset of threads copy to the local memory
if (threadIdx.x < KERNEL_SIZE && threadIdx.y < KERNEL_SIZE) {
      skernel[threadIdx.x*kRows+threadIdx.y] = kernel[threadIdx.x*kRows+threadIdx.y];
}
// Sync so we know we have copied everything
__syncthreads();
```

3 Experiential Results

To evaluate these different methods we run the code on three different systems. The hardware specifications can be summarized in the below table. We run each convolution 25 times and for each calculate the average and deviation each set of trails had. Additionally for the GPU implementation we record sub-sections of the program to provide an interesting discussion on the distribution of time.

	Thinkpad p51	**Desktop #1**	**Desktop #2**
CPU Model	Intel Xeon E3-1505M v6 (kaby lake)	AMD FX-8120 Eight-Core Processor (fx-series)	Intel Core i7-7800X (skylake)
CPU Cores	4 real, 4 virt	8 real	6 real, 6 virt
CPU Clock Speed (GHz)	3.0, 4.0 boost	3.1, 4.0 boost	3.5, 4.0 boost
GPU Model	Quadro M2200 (maxwell)	GeForce GTX 970 (maxwell)	GeForce GTX 1080 Ti (pascal)
GPU Clock Speed (GHz)	1.04	1.33	1.67
GPU Memory (GB)	4	4	11
GPU CUDA Cores	1024 (8 MP)	1664 (13 MP)	3584 (28 MP)

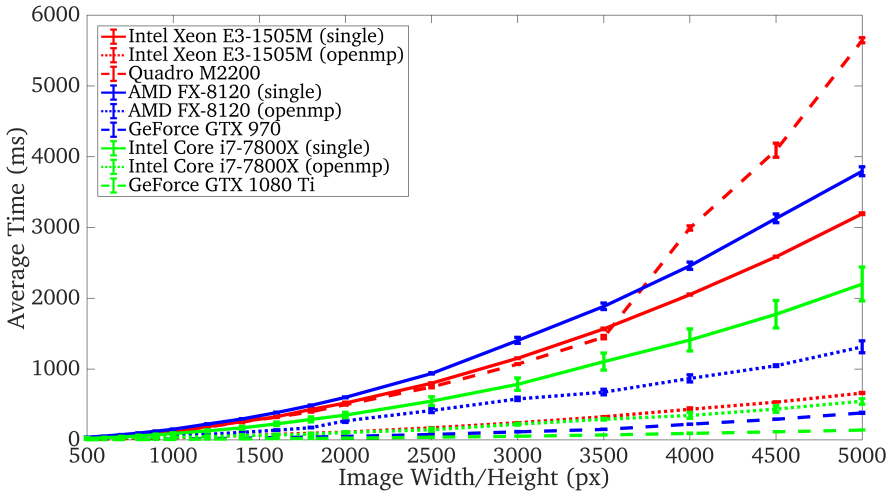


Figure 2: Execution times in milliseconds with deviations for each system show in different colors.

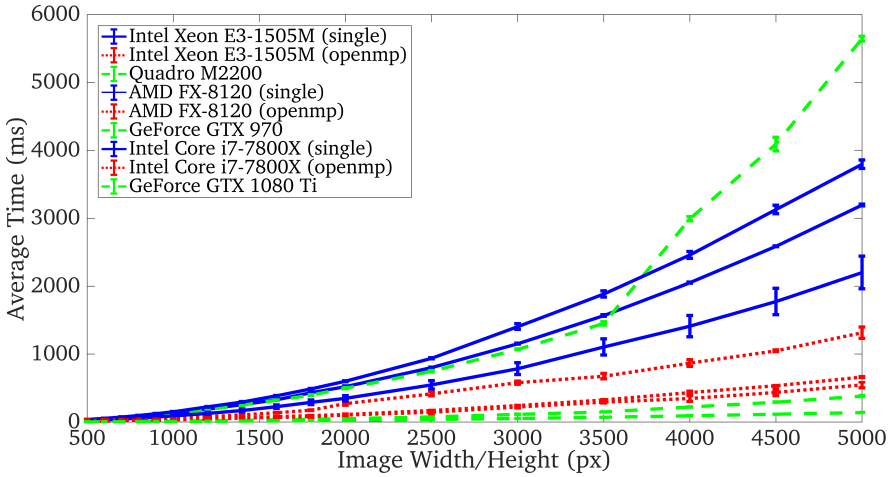
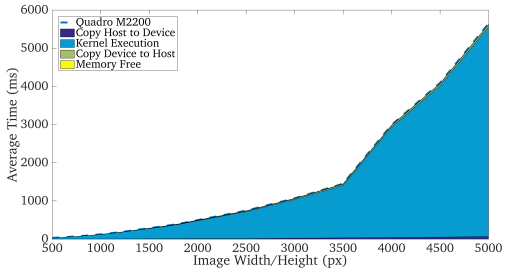
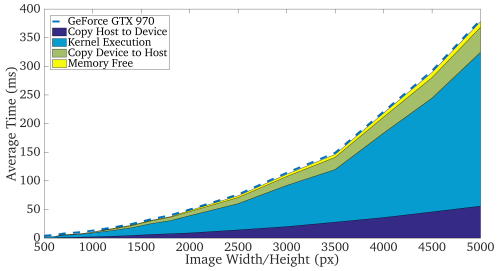


Figure 3: Execution times in milliseconds with deviations for each type of optimization. CPUs are shown in blue, OpenMP multi-threading shown in red and GPU CUDA implementations shown in green.



(a) Nvidia Quadro M2200 GPU execution time splits.



(b) Nvidia GTX 970 GPU execution time splits.

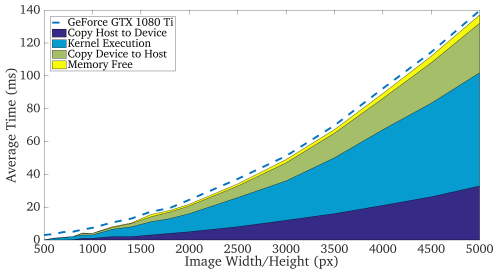


Figure 4: Nvidia GTX 1080ti GPU execution time splits.

4 Discussion

Seen above we can clearly see the speedup due to both multi-threading and parallelization onto a GPU. It is interesting to note that the Quadro M2200 did not scale that well with the increased size of the image. This is likely due to the smaller number of CUDA cores and the clock speed on the mobile laptop platform. We can also see in the above time split breakdowns for the Quadro M2200 that the kernel execution is what takes the majority of the execution time. We can also see that the GTX 1080ti has a more even split between the components that contribute to its execution time, suggesting that it can both process faster but also more data.

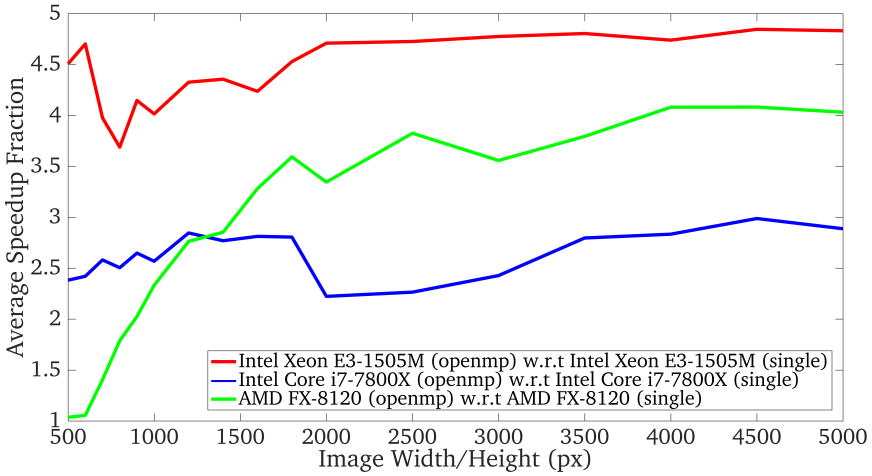


Figure 5: Speedup going from a single threaded program to a multi-threaded program using OpenMP.

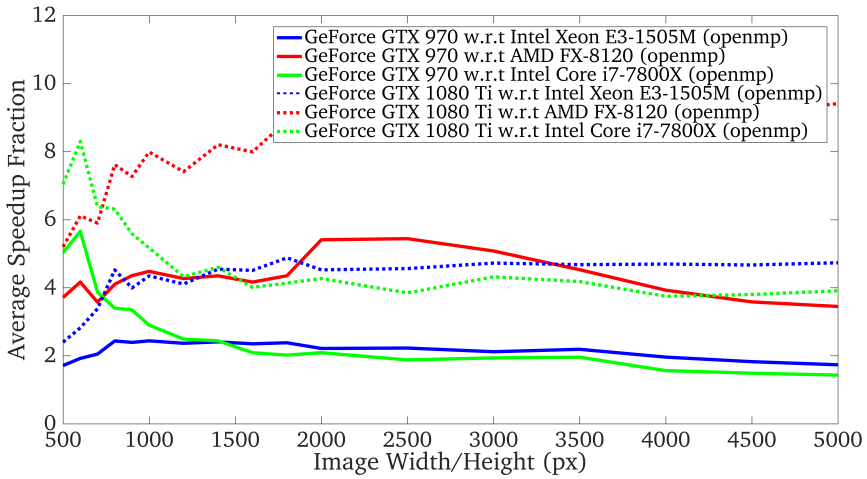


Figure 6: Speedup going from a multi-threaded program using OpenMP to a GPU accelerated program using a CUDA kernel implementation.

Looking at the speedup from taking a single threaded program and using OpenMP to multithread it, we can see that the Intel Xeon had an massive speedup of x4.5. It is interesting to see that the AMD FX-8120 does not have a constant speedup and instead only exhibits speedup when the image is greater then 1000x1000 pixels. This is likely due to the thread overhead being greater then the computations done by each thread (a thread calculates a convolution for a single pixel). When looking at the speedup when going from a multithreaded version to a GPU CUDA accelerated version we can see that there is a large speedup between 2-4x. We can see that the newer 1080ti care has a 2x speedup relative to GTX 970. While this is an expected result, we want to note that if one has a Xeon or i7 one can simply multithread using OpenMP and will only miss out on a 2x speedup if it was parallelize on the GPU.

In conclusion, we see that the parallelization of image convolution allows for a speedup of the over all execution time. We also show that in newer GPU units the kernel execution time on smaller problems is a smaller fraction relative to the transfer times. Finally we note that enabling multithreading on a CPU will give you the highest "value proposition" if you can sacrifice a 2x-3x speedup out of a total of 6x speedup. This is especially true if the processor being used is a Intel Xeon.