

Parallel Image Convolution

A Comparison of OpenCL, CUDA, and Python

Submitted by Alexander Stein

EECSE4750

Dr. Zoran Kostic

Assignment 3

Columbia University

November 9th, 2016

Problem Statement:

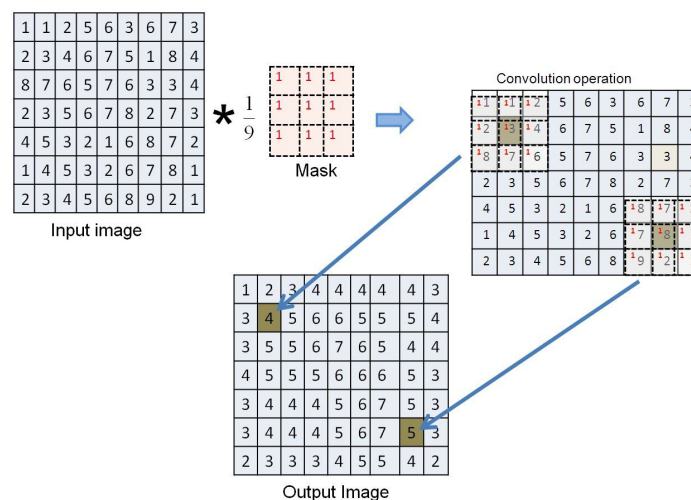
This assignment focuses on the problem of parallelizing discrete convolution on the GPU, and comparing the outputs and performance with a serial implementation.

Discrete convolution (circular convolution), is formally defined as follows:

$$(f * g)[n] = \sum_{m=0}^{N-1} \left(\sum_{k=-\infty}^{\infty} f(m + kN) \right) g_N(n - m).$$

Where the summation on k is a periodic summation of f , and the summation on m is a periodic summation of g . The functions f and g are assumed to be periodic.

In the context of our problem, we will simply be moving a window over a matrix as in the diagram below:



We are to implement this operation serially, and in parallel, and to compare the results. Finally, we will do a “real-world” application via image convolution, and compare outputs.

Section A: Theory Questions

Question 1

Identify the errors in the code below, intended to implement averaging. Rewrite the code with the requisite barriers included. (Hint: think of Read-Modify-Write racing conditions and the need for synchronizing threads)

Original Code

```
int tx = (blockDim.X * blockIdx.x) + threadIdx.x;

__shared__ int arr(1024);

arr [tx] = tx;

if (tx > 0 && tx < 1023)
{
    arr[tx] = (arr[tx - 1] + arr[tx] + arr[tx + 1])/3;
}
```

Corrected Code

```
1.  int tx = (blockDim.X * blockIdx.x) + threadIdx.x;
2.
3.  __shared__ int arr(1024);
4.
5.  arr [tx] = tx;
6.  __syncthreads();
7.
8.  int atm = arr[tx - 1];
9.  int atp = arr[tx + 1];
10. __syncthreads();
11.
12. if (tx > 0 && tx < 1023)
13. {
14.     arr[tx] = (atm + tx + atp)/3;
15. }
16. __syncthreads();
```

Comments:

It should be noted that the specifications for exactly what the expected output of this “average” functionality are ambiguous. It is possible that the designer intends for the array to be “dynamically averaged,” where the updated value $\text{arr}[tx] = (\text{arr}[tx - 1] + \text{arr}[tx] + \text{arr}[tx + 1])/3$ assumes “ $\text{arr}[tx - 1]$ ” to be the average of its two surrounding values. This seems, though, impossible in a parallel context; so it is assumed in answering this question that the intention is for $\text{arr}[tx]$ to be updated with the original values of its surrounding elements.

Synchronization is added on Line 6 to ensure that the entire array in shared memory

contains valid values before those values are used for calculations. In keeping with the assumption mentioned above, the initial values of `arr[tx - 1]` and `arr[tx + 1]` are stored in each thread's private memory, and this task is synchronized so that no other thread receives any but the original values. The final correction, or optimization, is that instead of calling `arr[tx]` in the averaging equation, I eliminate a memory access by just using "tx" instead of "`arr[tx]`" in the averaging equation. A final "`__syncthreads()`" is added at the end to ensure that the calculation is complete before returning any information to the host.

It is important to note that this is a toy example, because if `arr[tx] = tx`, then `arr[tx-1] + arr[tx] + arr[tx + 1] = 3 * tx`, and so the average is just tx.

Question 2

What are memory fences, and how are they different from barriers?

A Memory fence, as defined in the CUDA developers manual, is a method for enforcing a stronger memory ordering than initially specified by the language (which is weak). Specifically, the CUDA `__threadfence()` function ensures for all threads in the block that read/write to shared and global memory above the function call will occur before reads/writes below the function call. An interesting addendum: *"Note that for this ordering guarantee to be true, the observing threads must truly observe global memory and not cached versions of it; this is ensured by using the volatile keyword."* The "volatile" keyword in C is an indicator to the compiler not to optimize the assignment of a particular variable.

This is different from `__syncthreads()` because with that function, no other threads in the block can continue until all reads/writes before `syncthreads` have completed and all threads have reached the function call. With the `threadfence`, all threads can continue operation, but they will be forced to view memory in a stronger order. Take the example below:

Without `__threadfence()` -

```
__device__ volatile int X = 1, Y = 2;
__device__ void writeXY()
{
    X = 10;
    Y = 20;
}

__device__ void readXY()
{
    int A = X;
    int B = Y;
}
```

It is possible for A to contain 1 and B to contain 20.

With `__threadfence()` -

```
__device__ volatile int X = 1, Y = 2;
__device__ void writeXY()
{
    X = 10;
    __threadfence();
    Y = 20;
}

__device__ void readXY()
{
    int A = X;
    int B = Y;
}
```

There are only three possible outcomes for A, B:

- A = 1, B = 2
- A = 10, B = 20
- A = 10, B = 2

Question 3

- For an element in matrix size 512X1024, is situated at position (5,63) globally. If the matrix is tiled into largest possible square blocks supported by tesseraact server GPUs, what will be the `blockIdx` and `threadIdx` in both x and y directions of the element?
 - Total matrix size is $2^9 + 2^{10} = 2^{19}$, but the allowable amount of threads in a block is 2^{10} . So our maximum square block size is $2^5 * 2^5$, or 32 threads per x and y dimension of each block. This means we will have 2^9 or 512 blocks (tiles).
 - Since we are treating the blocks as tiles, we want the grid organized in the same way as in the input matrix, so the grid should have dimensions $2^9 / 2^5 = 2^4$, $2^{10} / 2^5 = 2^5$. In the host code we would dispatch the kernel as `block=(32, 32, 1)`, `grid = (16, 32, 1)`.
 - Finally, for element (5,63), we will have **`blockIdx.x = 1, blockIdx.y = 0, threadIdx.x = 31, threadIdx.y = 5`**.
- If our input matrix is less than the specified block size in OpenCL, what manipulation can we do to the input to ensure the program runs smoothly?
 - If our input matrix is less than the specified block size in OpenCL, we can pad the input with zeros (or empty strings, etc.), or we can set boundaries inside of the kernel to ensure the output doesn't include data from indexes outside of the input's range.

Question 4

Assuming $M > N$:

Code 1:

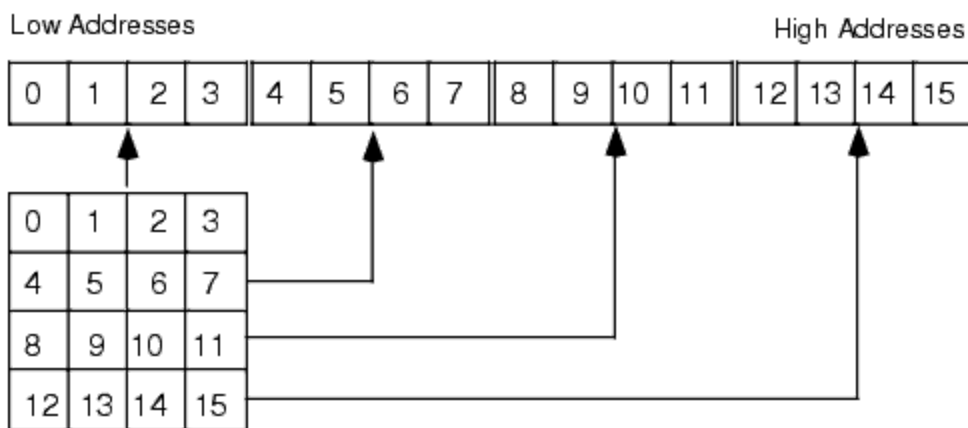
```
for(int i=0; i<M; i++) for(int j=0; j<N; j++) val = A[i][j]
```

Code 2:

```
for(int j=0; j<N; j++) for(int i=0; i<M; i++) val = A[i][j]
```

Will the above two codes give the same performance? Why/Why not?

The two codes will not give the same performance. OpenCL and CUDA store matrices in **row-major order**:



So **code 1 will be more efficient** because it will pull in contiguous memory locations, whereas code 2 will pull in very far-away memory locations.

Section B: Programming

Platform Information:

GPU: Nvidia GTX 660M

CPU: Intel Core i7 3630QM @ 2.4GHz

OS: Ubuntu 14.04 LTS

RAM: 24 GB SODIMM DDR3 Synchronous 1333 MHz (0.8 ns)

Overall Structure Pseudocode:

Since we have discussed the initial setup of GPU devices in detail through previous assignments, only the kernel pseudo-code will be summarized here.

```
kernel( int* INPUT, int* MASK, int M, int N, int F, int* OUTPUT){
    // M - row dimension of input
    // N - column dimension of input
    // F - square dimension of mask

    // global index
    int i = global_id(rows);
    int j = global_id(cols);

    // pad input with zeros, store in local/shared memory
    __shared d_INPUT[] = PAD(INPUT);

    barrier_synchronize();

    // fill local padded input array with necessary dimensions from INPUT
    for k in group_row_boundary:
        for l in group_col_boundary:
            d_INPUT[k+offset][l+offset] = INTERNAL(INPUT[i][j]);

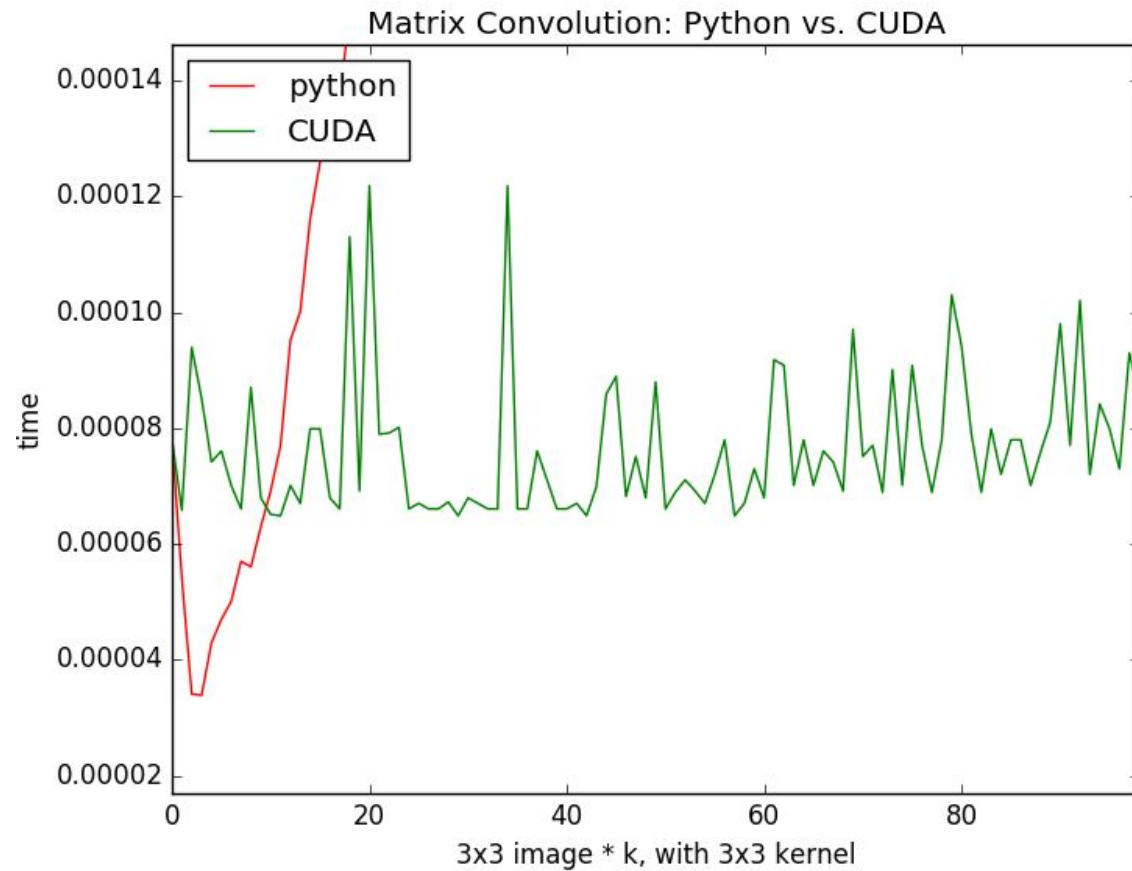
    barrier_synchronize();

    // Convolution Calculation
    OUTPUT[i][j] = 0;
    for p in mask_row_boundary:
        for q in mask_col_boundary:
            OUTPUT[i][j] +=
                MASK[p][q] * d_INPUT[local_id(col)+offset - p][local_id(row)+offset - q];

    barrier_synchronize();
}
```

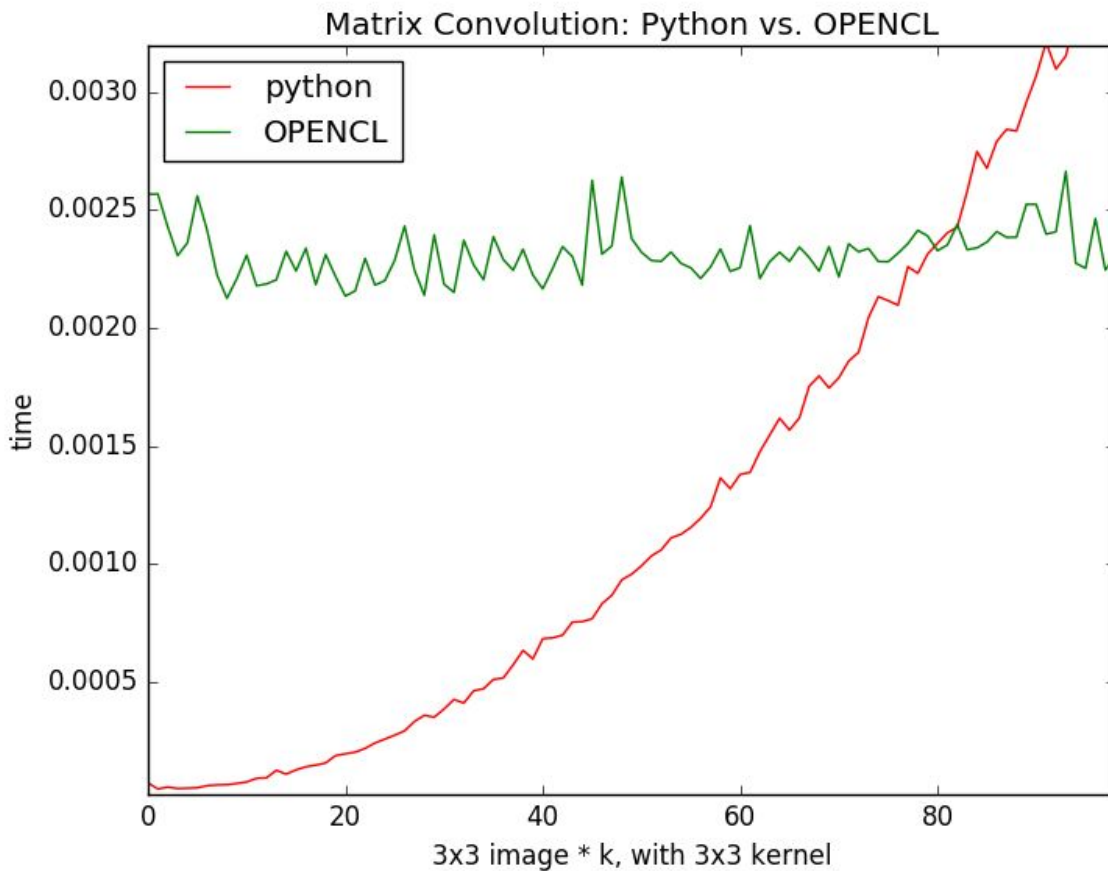
Results: Graphs with Explanation

Part 1-b, Fixed Kernel Size - CUDA vs. Python



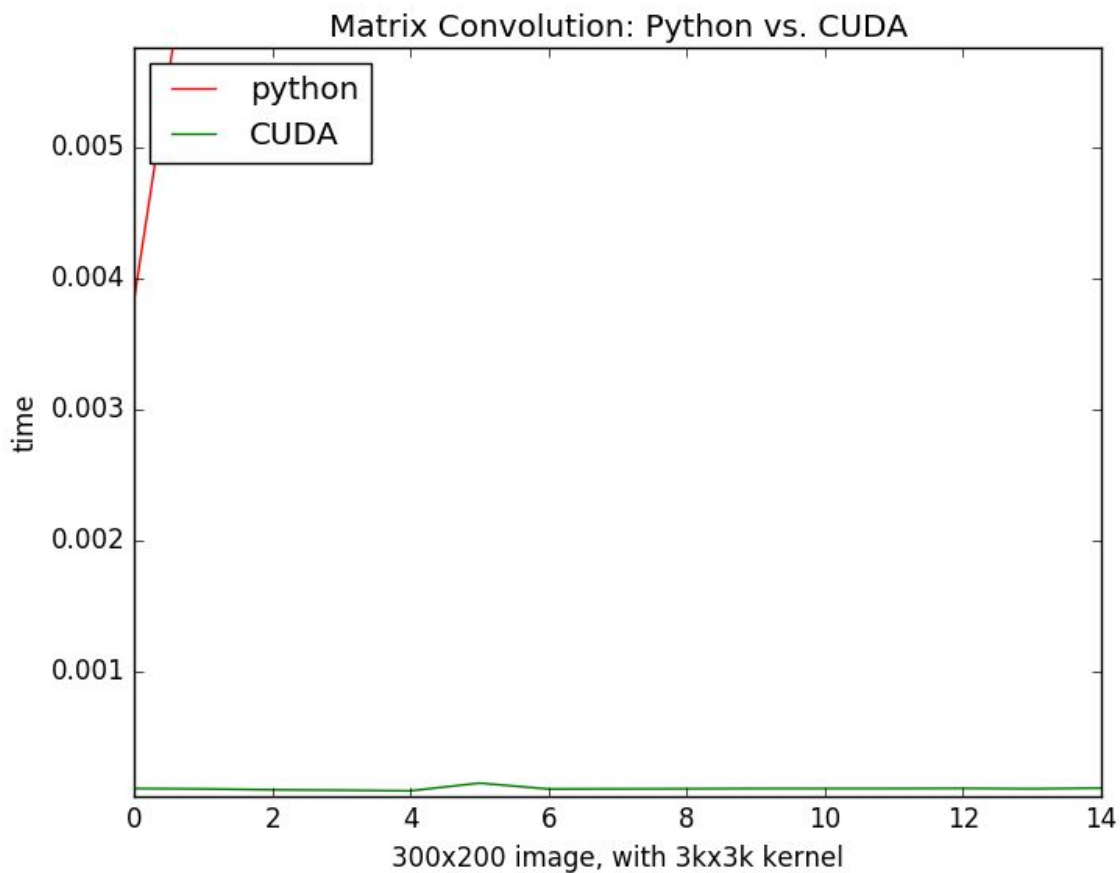
The scaling of image size shows that the GPU quickly overtakes the CPU, here after the image size is $> (30,30)$ for CUDA.

Part 1-b, Fixed Kernel Size - OPENCL vs. Python



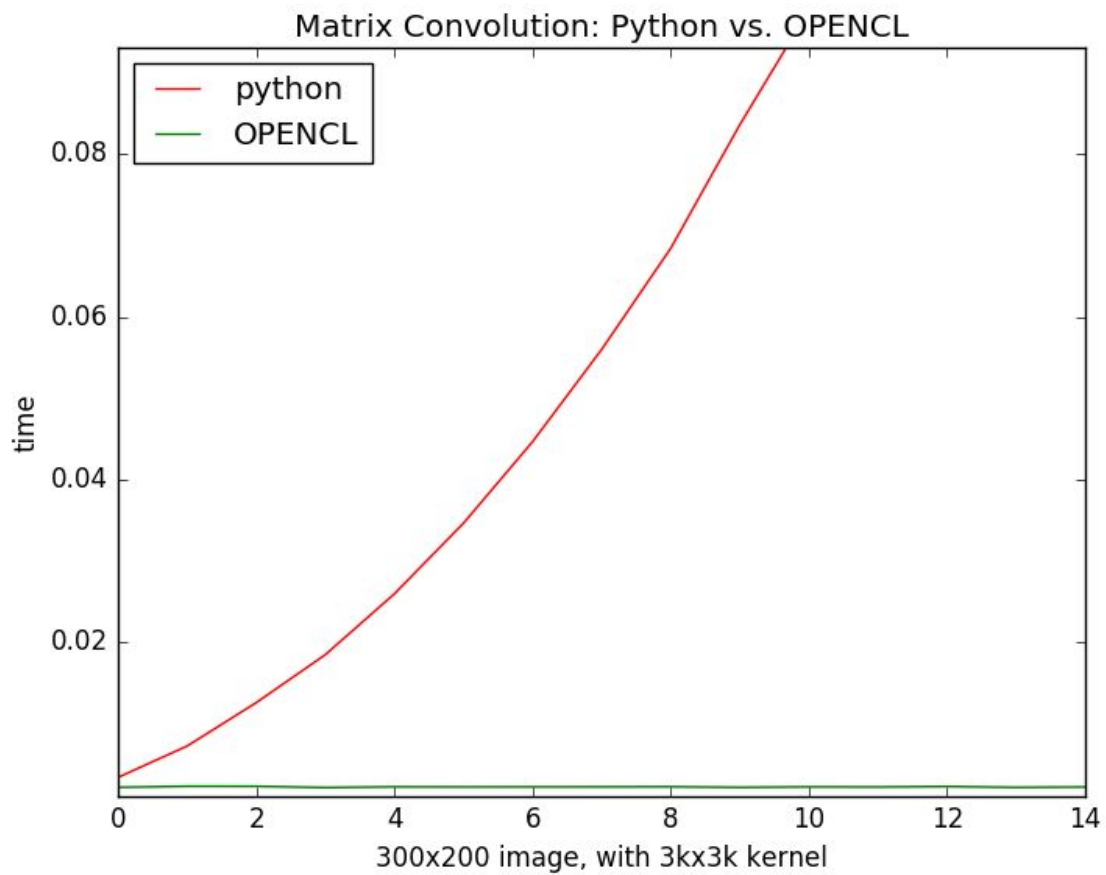
The scaling of image size shows that the GPU quickly overtakes the CPU, here after the image size is $> (240,240)$ for OpenCL. This is much slower than CUDA, perhaps because the GPU is an NVIDIA device optimized to their proprietary language. Furthermore, NVIDIA devices do not support advances in OpenCL past version 1.1, which is now relatively old.

Part 1-c, Fixed Image Size - CUDA vs. Python



For this reasonably large image size, the GPU is faster than the CPU even for the smallest kernel size, 3x3. I had to scale this graph to 40x the max GPU time just to see the sliver of data from the CPU in the upper left-hand corner.










Part 1-c, Fixed Image Size - OPENCL vs. Python










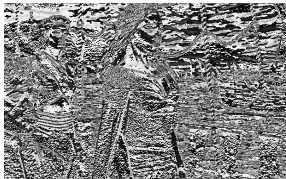
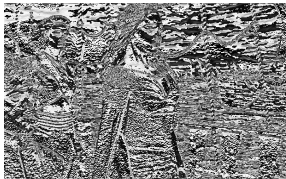
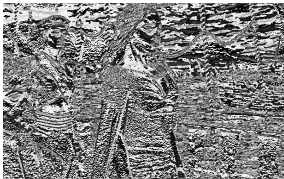
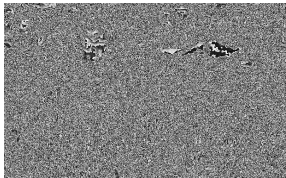
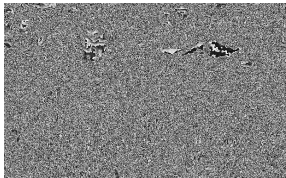
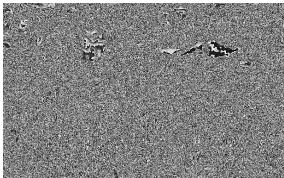
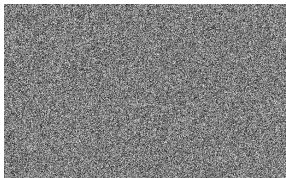
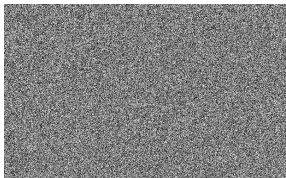
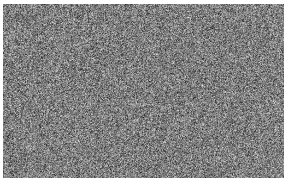


OpenCL is also faster than the CPU for even the smallest kernel size; though it is slower than CUDA likely for the same reasons stated above.

Part 2, Image Convolution Outputs



Filter	CPU Image	CUDA Image	OPENCL Image
Identity			
Sharpen			
Blur			

Edge Detect			
Emboss			
Sob_x			
Sob_y			
Smooth 5x5			
Smooth 11x11			

Note that the images between CPU, CUDA, and OPENCL are exactly identical under the filter, verified in the delivered code via `numpy.allclose()`.

Results: Analysis

Optimal Image & Kernel Sizes:

- As is apparent from my results above, the GPU performance is almost unchanged for both scaled image sizes and kernel sizes
- With this in mind, I kept the image size the same dimensions as the input for part 2 so as not to distort it.
- I did not have a chance to increase the kernel for part 2 much past 11x11, but I would choose 32x32 in the optimal situation because this fits well with the warp size of the NVIDIA streaming processors, and because this is the maximum size of each block/work-group. Larger kernel's are better mathematically speaking because they increase the precision of the filtering operation: in a Gaussian filter for example, the larger the kernel, the more standard-deviations of error we reduce the result by.

Note on the code:

- I spent a great deal of time trying to do the zero-padding in an efficient way -- though I did not succeed and had to stick with the brute-force method -- which I feel would have increased both CUDA and OPENCL performance. I think there is an easy way to do this - by just checking for global "out-of-bounds" in the actual convolution calculation instead of pre-padding the device local memory with zeros.

References

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#memory-fence-functions>