CUDA Accelerated Gaussian Blur Project

COMP 137 Final Report

Michael Davis

Spring 4/29/2018

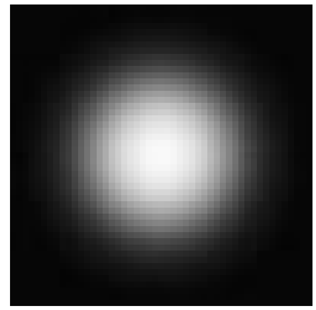
Problem

The project was a CUDA GPU accelerated image Gaussian Blur algorithm. This works by calculating a new blurred colour value for each pixel. This is done by calculating over each colour channel, RGB (Alpha was not used in this project).

The gaussian blur implementation works by using a weighted filter. The filter is used by overlaying the center of the filter on the pixel you would like to calculate. Each pixel the filter covers will be used in the calculation of the target pixels final colour. The values of the filter are used to determine the weight at which the neighbouring pixel is calculated. The filter values are normalized in order to stop energy loss / gain (stops the image getting lighter / darker). Filters generally resemble a normal distribution bell curve, higher weights at the center and lower weights towards the edges.

5	2	5	1	6	8	6
4	6	5	8	4	3	9
5	4	8	3	2	4	8
2	1	2	1	3	2	7
4	6	5	**4**	**6**	**2**	1
5	8	1	**2**	**3**	**9**	8
8	1	5	**6**	**6**	**5**	5

Example Image Pixel Values (**RED** = target pixel, **BLACK** = neighbouring pixels)



0.05	0.1	0.05
0.1	0.4	0.1
0.05	0.1	0.05

Example Filter Values (Normalized to stop energy loss/gain)

$p_{x, y} = \text{SUM}(f_{i, j}\; p_{x-r+i,\, y-r+j})$

$$P_{4,5} = (0.05 * 4) + (0.1 * 6) + (0.05 * 2) + (0.1 * 2) + (0.4 * 3) + (0.1 * 9) + (0.05 * 6) + (0.1 * 6) + (0.05 * 5)$$

$P_{4,\,5} = 4.35$

This process is rather slow on a CPU, especially for filters of larger width as there are $x*y*f^2$ operations for each colour channel. (x, y = dimensions of image, f = filter width).

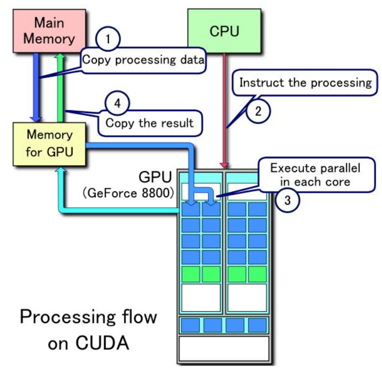
Design

I used CUDA (Compute Unified Device Architecture) to parallelize this project. CUDA is a GPGPU (General Purpose Graphical Processing Unit) API designed to help with the implementation of massively parallel tasks. Since the GPU is designed as a massively parallel set of processors it is perfectly designed for this type of application. I plan to run 1 thread for every pixel in the image. This will mean that the compute time of each thread is mostly dependent on the size of the filter it is calculating.

CUDA is compatible with c, c++ and fortran and compiled with the NVCC compiler. It allows one to write both GPU and CPU code into one file with a .cu extension. GPU code is defined with the __global__ and __device__ function prefix. Since the code is being run on the GPU however data needs to be migrated back and forth between the CPU and GPU. CUDA does have support for a unified memory space that is “shared” between the two, however for this project I used manual memory allocation and migration.

The basic process of a CUDA program follows a flow similar to this:

1. Allocate GPU memory space and copy processing data from CPU to GPU
2. Call and parse compiled GPU kernel to the GPU from CPU
3. Execute parallel instruction set on GPU with the defined number of threads.
4. Copy computed data back from GPU to CPU.



Memory Allocation : cudaMalloc(dev_var, size)

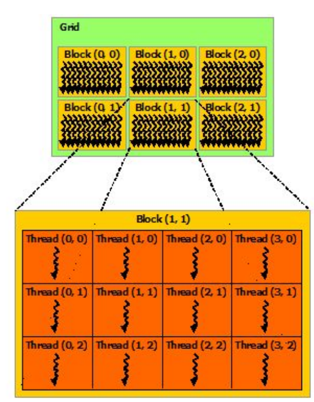
Similar to C memory must be allocated to the GPU’s DRAM, this is done using cudaMalloc(), the pointer is for a space on the GPU and the size is defined as normal.

Memory Migration : cudaMemcpy(dev_var, host_var, size, direction)

To copy memory from one processor unit to another use cudaMemcpy, defining the device space, host space the size and the direction that is a CUDA defined enum e.g. cudaMemcpyDeviceToHost.

CUDA Kernel Call : kernel_function_name<<<GRID_SIZE, BLOCK_SIZE>>>(args...)

To call a kernel function from the CPU you must use the <<<>>> notation to define the launch size of the kernel. Each of the two can be defined in up to three degrees of freedom based on CUDA version and hardware capabilities. However for this project Only one degree was used for each. GRID_SIZE defines the number of blocks to launch and BLOCK_SIZE defines the number of threads that are launched per block. Arguments for the function can then be passed after the triple chevron notation.



Free CUDA Memory : cudaFree(dev_var)

Similar to C, frees up memory spaces on the GPU memory.

```
// ---------------------------------------------------------------------------
// Function Name : GPUAcceleratedGaussianBlur
//
// Description : Callable GPU acceslerated gaussian blur.
//               Uses CUDA acceleration to create gaussian blurred data.
//               Takes in original RGB channel data and uses a convolutional
//               filter of width fil_w to generate weighted blur values for each
//               pixel. Filter must be an odd valued width.
//
// Input : *f_buff, fil_w - Filter buffer data, should be normalized in order to
//                          maintain image "energy", this stops darkening or
//                          lightening effects of the image upon computation.
//
//         *r_buff, *g_buff, *b_buff - RGB image data channels.
//
//         w, h - Width and heght of the image
// ---------------------------------------------------------------------------
void GPUAcceleratedGaussianBlur(float *f_buff, /* In  - Filter buffer data  */
                                int fil_w,     /* In  - Fileter width       */
                                int *r_buff,   /* Out - Red channel data    */
                                int *g_buff,   /* Out - Green channel data  */
                                int *b_buff,   /* Out - Blue channel data   */
                                int w,         /* In  - Image width         */
                                int h)         /* In  - Image height        */
{

      // ---- Check filter size and generate offset value
      if (fil_w % 2 != 1) return;
      int f_offset = -((fil_w - 1) / 2);

      // ---- Define size variables to allocate space on the GPU ----
      size_t bmp_size = w * h * sizeof(int);
      size_t fil_size = fil_w * fil_w * sizeof(float);

      // ---- Allocate device memory ----
      int *d_r, *d_g, *d_b;
      int *d_r_out, *d_g_out, *d_b_out;
      float *d_f;
      gpuErrchk( cudaMalloc(&d_f, fil_size) );

      gpuErrchk( cudaMalloc(&d_r, bmp_size) );
      gpuErrchk( cudaMalloc(&d_g, bmp_size) );
      gpuErrchk( cudaMalloc(&d_b, bmp_size) );

      gpuErrchk( cudaMalloc(&d_r_out, bmp_size) );
      gpuErrchk( cudaMalloc(&d_g_out, bmp_size) );
      gpuErrchk( cudaMalloc(&d_b_out, bmp_size) );

      // ---- Copy data from host memory to device memory ----
      gpuErrchk( cudaMemcpy(d_r, r_buff,  bmp_size, cudaMemcpyHostToDevice) );
      gpuErrchk( cudaMemcpy(d_g, g_buff,  bmp_size, cudaMemcpyHostToDevice) );
      gpuErrchk( cudaMemcpy(d_b, b_buff,  bmp_size, cudaMemcpyHostToDevice) );
      gpuErrchk( cudaMemcpy(d_f, f_buff,  fil_size, cudaMemcpyHostToDevice) );

      // ---- Determine kernel launch dimensions --- -
      int blockSize = 512;
      int numBlocks = ((w * h) + blockSize - 1) / blockSize;

      // --- Launch Kernel ----
      cudaGaussianBlur<<<numBlocks, blockSize>>>(d_f, fil_w, f_offset, d_r, d_g, d_b, w, h, d_r_out, d_g_out, d_b_out);

      // ---- Read data back from device to host ----
      gpuErrchk( cudaMemcpy(r_buff, d_r_out, bmp_size, cudaMemcpyDeviceToHost) );
      gpuErrchk( cudaMemcpy(g_buff, d_g_out, bmp_size, cudaMemcpyDeviceToHost) );
      gpuErrchk( cudaMemcpy(b_buff, d_b_out, bmp_size, cudaMemcpyDeviceToHost) );


      // ---- Free up allocated memory ----
      gpuErrchk( cudaFree(d_f) );

      gpuErrchk( cudaFree(d_r) );
      gpuErrchk( cudaFree(d_g) );
      gpuErrchk( cudaFree(d_b) );

      gpuErrchk( cudaFree(d_r_out) );
      gpuErrchk( cudaFree(d_g_out) );
      gpuErrchk( cudaFree(d_b_out) );

}
```

CPU called function to run GPU code. Here you can see how the above cuda functions are used in my implementation.

```
for (x = 0; x < w; x++)
{
	for (y = 0; y < h; y++)
	{
		r_val = 0.0f;
		g_val = 0.0f;
		b_val = 0.0f;

		for (f_x = 0; f_x < filter.width; f_x++)
		{
			for (f_y = 0; f_y < filter.width; f_y++)
			{

				x_sam = x + f_x + f_offset;
				y_sam = y + f_y + f_offset;

				if (isInBounds(w, h, x_sam, y_sam) == false)
				{
					x_sam = x;
					y_sam = y;
				}

				r_val += ((float) bmp->red_buff  [_2dto1d(w, x_sam, y_sam)] * (float) filter.weights[_2dto1d(filter.width, f_x, f_y)]);
				g_val += ((float) bmp->green_buff[_2dto1d(w, x_sam, y_sam)] * (float) filter.weights[_2dto1d(filter.width, f_x, f_y)]);
				b_val += ((float) bmp->blue_buff [_2dto1d(w, x_sam, y_sam)] * (float) filter.weights[_2dto1d(filter.width, f_x, f_y)]);
			}
		}

		r[_2dto1d(w, x, y)] = (int)(r_val);
		g[_2dto1d(w, x, y)] = (int)(g_val);
		b[_2dto1d(w, x, y)] = (int)(b_val);
	}
}
```

Serial version of the Gaussian Blur function

```
for (f_x = 0; f_x < fil_w; f_x++)
{
	for (f_y = 0; f_y < fil_w; f_y++)
	{

		// Calculate sample coordinates for original image
		x_sam = x + f_x + fil_off;
		y_sam = y + f_y + fil_off;

		// If ample co-ords out of bounds, sample from current pixel.
		if (cudaIsInBounds(w, h, x_sam, y_sam) == false)
		{
			x_sam = x;
			y_sam = y;
		}

		// Caluclate 1D array index values.
		img_index = cuda2dto1d(w, x_sam, y_sam);
		fil_index = cuda2dto1d(fil_w, f_x, f_y);

		// Increment temporary pixel colour values.
		r_val += ((float) r[img_index] * (float) f_buff[fil_index]);
		g_val += ((float) g[img_index] * (float) f_buff[fil_index]);
		b_val += ((float) b[img_index] * (float) f_buff[fil_index]);

	}
}
```

Parallel Version of the Gaussian Blur function : Note the similarity besides the need for first two x and y for loops.

Experiments

My experiments were based on a changing filter size used on a 845x450 image.

The filters were of widths : 5, 13, 25, 49 and 101.

The data was collected on serial time, GPU time and kernel time.

GPU applications was executed with 380,416 kernel calls.

512 threads per block.

(int) ((845 * 450) + 512 -1) / 512 = 743 Blocks.

743 * 512 = 380,416

```
// ---- Determine kernel launch dimensions --- -
int blockSize = 512;
int numBlocks = ((w * h) + blockSize - 1) / blockSize;

// --- Launch Kernel ----
cudaGaussianBlur<<<numBlocks, blockSize>>>(d_f, fil_w,
```

```
t1 = getProcessTime();

if (GPU_ACCELERATION != 1)
{
     serialGaussianBlur(my_filter, &my_bmp);
} else {
     GPUAcceleratedGaussianBlur(my_filter.weights,
                                my_filter.width,
                                my_bmp.red_buff,
                                my_bmp.green_buff,
                                my_bmp.blue_buff,
                                my_bmp.width,
                                my_bmp.height);
}

t2 = getProcessTime();
```

GPU and CPU time were both calculated with the timer.h code that we have used throughout the semester. Kernel timing was done using Nvidia's nvprof command (A GPU timing profiler). The kernel time was pulled from the cudaGaussianBlur row under the Time column.

```
mike@mike:~/Dropbox/UOP/COMP_137/Project$ nvcc source/main.cpp source/bmp.cpp source/GPU_gaus.cu -o
mike@mike:~/Dropbox/UOP/COMP_137/Project$ nvprof ./main
File Size : 202
Dimensions : 5 x 5
File Size : 1141254
Dimensions : 845 x 450
==29651== NVPROF is profiling process 29651, command: ./main
Initial data input time : 0.017038 seconds
Image processing time   : 0.182095 seconds
Data saving/output time : 0.002655 seconds
==29651== Profiling application: ./main
==29651== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   41.16%  729.26us         4  182.31us     608ns  244.93us  [CUDA memcpy HtoD]
                   39.84%  705.83us         3  235.28us  231.59us  242.31us  [CUDA memcpy DtoH]
                   19.00%  336.71us         1  336.71us  336.71us  336.71us  cudaGaussianBlur(float*
, int*, int, int, int*, int*, int*)
      API calls:   97.45%  144.22ms         7  20.604ms  93.406us  143.55ms  cudaMalloc
                    1.69%  2.5060ms         7  358.00us  188.15us  713.42us  cudaMemcpy
                    0.53%  784.40us         7  112.06us  101.79us  157.62us  cudaFree
                    0.21%  313.22us        94  3.3320us      88ns  137.99us  cuDeviceGetAttribute
```

GPU acceleration was defined at the top of main.cpp, 1 = GPU else CPU

```
#define GPU_ACCELERATION 1
```

Filters were loaded in as images so name is hardcoded into main.cpp

```
loadBMPImage((char *)"res/5_filter.bmp", &bmp_filter);
bmpToFilter(bmp_filter, &my_filter);
```

Timing data was pulled at each filter width for both serial and parallel and each run was executed 3 times.

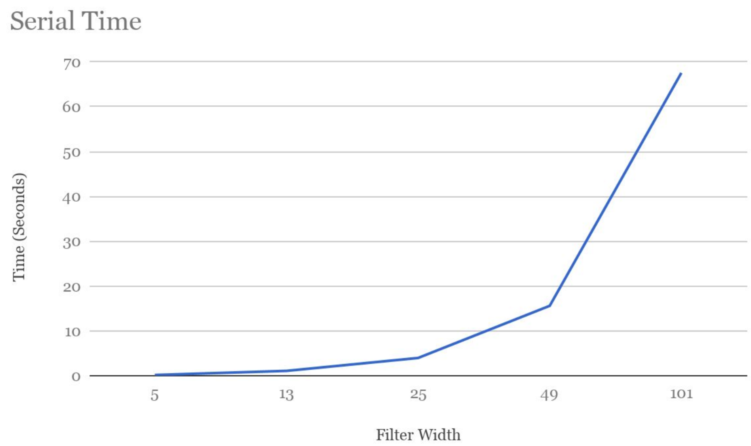
Raw Data :

	Serial (s)			
	1	2	3	Avg
5	0.171391	0.170954	0.16984	0.17072833
13	1.07818	1.07575	1.10153	1.0851533
25	3.93647	3.97088	3.98784	3.9650633
49	15.5266	15.672	15.6024	15.600333
101	67.8704	67.3701	67.4645	67.568333

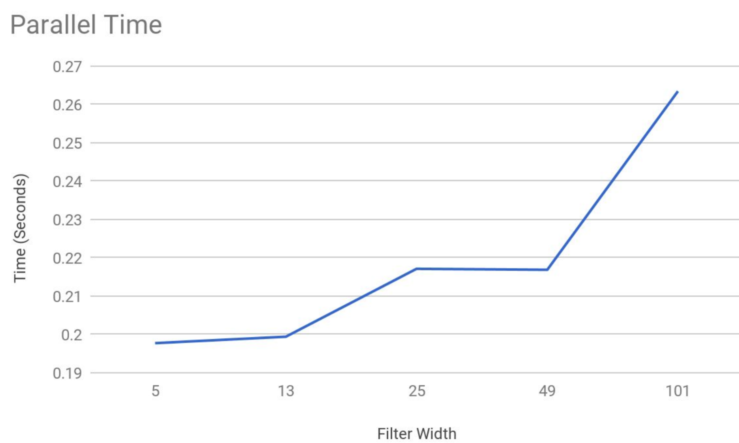
Parallel Timing (s)			
1	2	3	Avg
0.190076	0.205644	0.19748	0.19773333
0.197802	0.198646	0.201778	0.19940866
0.227664	0.209809	0.21391	0.21712766
0.217825	0.216069	0.216712	0.21686866
0.271501	0.257555	0.261284	0.26344666

Parallel GPU Kernel (s)			
1	2	3	Avg
0.00033543	0.00033582	0.00033511	0.00033545333
0.0017378	0.0017521	0.0017443	0.0017447333
0.0057597	0.0050698	0.0045465	0.0051253333
0.015449	0.014903	0.015267	0.015206333
0.065613	0.055849	0.054408	0.058623333

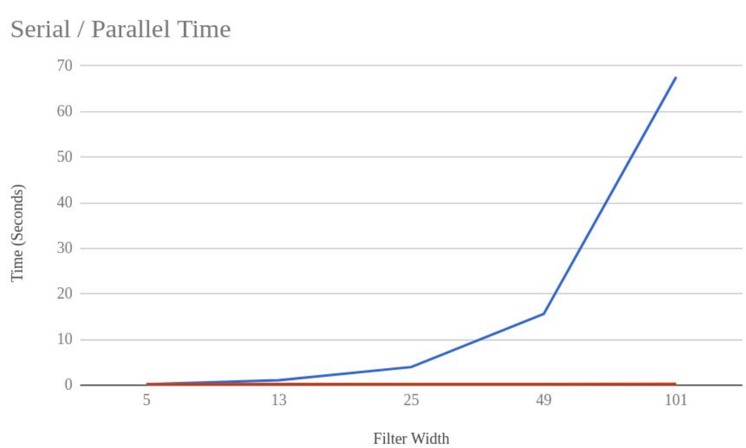
Speedup CPU->GPU	Kernel Speedup CPU->Kernel	Efficiency CPU->GPU	Kernel Efficiency CPU->Kernel
0.8634271746	508.9480901	0.00000226969206	0.001337872461
5.441856422	621.9594207	0.00001430501457	0.001634945483
18.26143759	773.6205775	0.00004800386311	0.002033617349
71.9344734	1025.910257	0.0001890942374	0.002696811534
256.4782246	1152.584295	0.0006742046196	0.003029799733



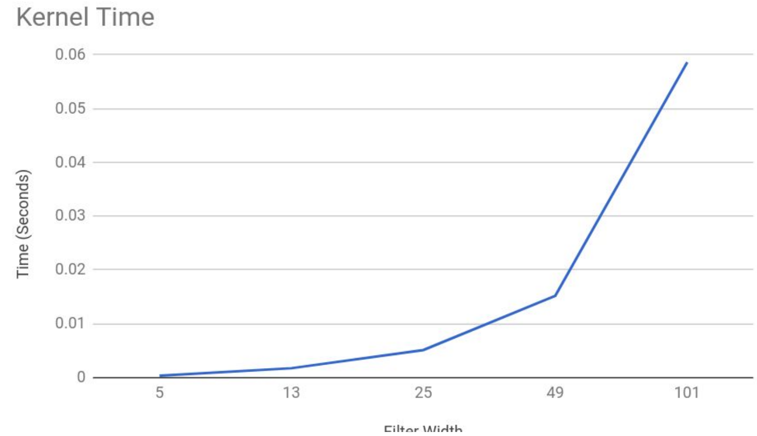
Serial time of program for filter size 5 - 101. Minimum of about ~1 second to a max time of ~68 seconds.



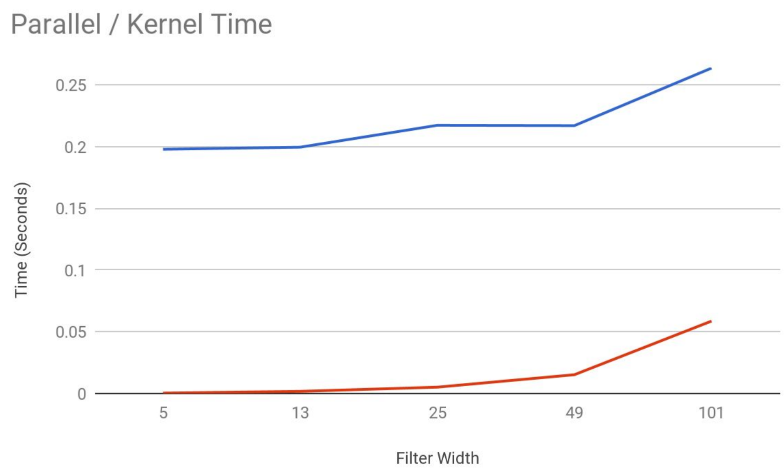
Parallel time of program for filter size 5 - 101. Minimum of about ~0.20 second to a max time of ~0.26 seconds.



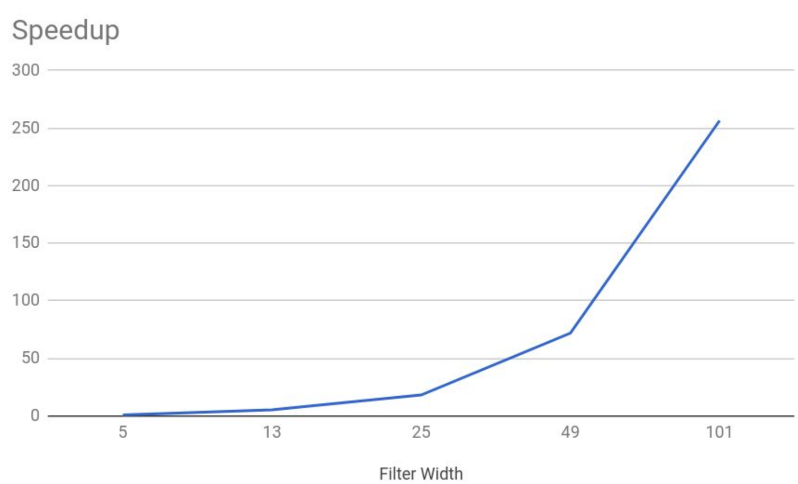
Serial vs Parallel Time plot. Demonstrating the difference between times on the same scale.



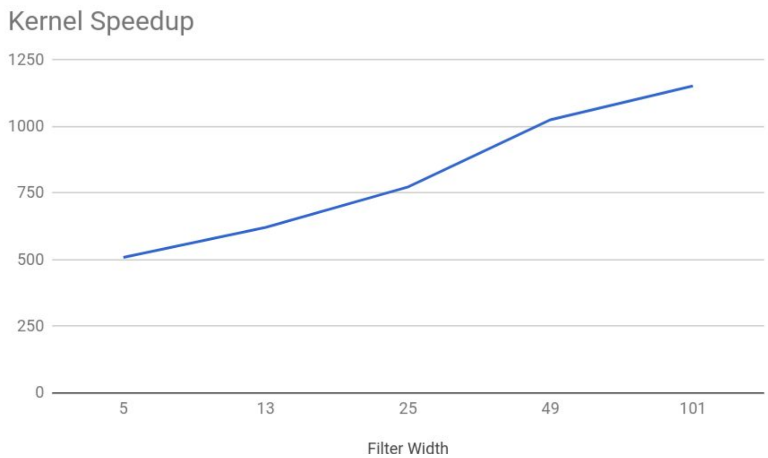
Parallel kernel time of program for filter size 5 - 101. Minimum of about ~0.0003 second to a max time of ~0.06 seconds.



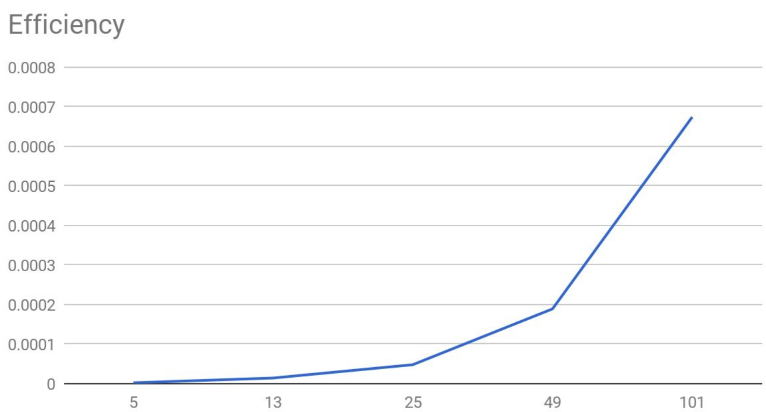
Difference between Parallel and Kernel time, demonstrating the overhead time taken for cuda processes such as memory allocation, memory migration etc. is ~0.2 seconds.



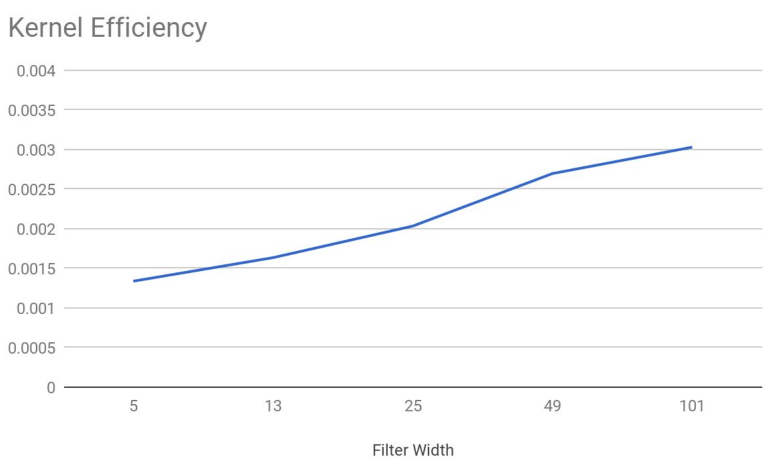
The speedup of CPU to GPU time. With a maximum speedup of over 250x.



The kernel speed up is significantly higher as this is just the algorithm computation time. This reaches a speedup of ~1100x.



Efficiency of CPU to GPU, this is calculated with 380,416 processes. The values are very small as GPU processors are much slower than CPU.



Efficiency of CPU to Kernell, this is calculated with 380,416 processes. The values are very small as GPU processors are much slower than CPU.

Analysis

Although the efficiency results are not very high, it is only because of the extremely high number of processes that are being launched by the GPU. Comparatively GPU cores are also much slower than CPU cores therefore a drop in efficiency is to be expected. Also the GPU does not necessarily launch all 380,416 processes at once. There is a considerable amount of scheduling that happens by the GPU on the SM (simultaneous multiprocessor) chips. This would also lower the efficiency values.

However looking at the raw speedup of the computation is astounding. Even with all of the overhead of GPU memory allocation and memory data migration we are still reaching over 250x speedup results. If you look at the pure computation time of the kernels we start to see up to 1100x results.

Although these tests did not account for a changing processor size I am confident that based off of the serial and GPU data that a solution like this is weakly scalable as the number of processors to the number of pixels will scale linearly and the time would stay constant.

For this basic implementation I believe that this is a good solution. However higher performance could be achieved. More work could be done to the memory management of the program. CUDA has different tiers of memory and specific data can be allocated to specific blocks or threads for quicker reading and writing by those threads or blocks. Higher performance could also be achieved by launching a thread to compute chunks of filter values rather than a single thread computing the entire filter. These chunks of whatever size could then be gathered by a second kernel call to atomically add them together to compute the final values for each pixel. However I felt that level of memory management and process management was beyond the scope of this project.

Implementation

Hardware : Intel 7700k, Geforce GTX 1080 Ti

OS : Ubuntu 16.04.4

Software : nvcc CUDA compiler drivers V9.1.85

Project compiled and run with:

nvcc source/main.cpp source/bmp.cpp source/GPU_gaus.cu -o main

nvprof ./main

Filter image file and GPU acceleration can be defined in main.cpp as explained in experiments section.

Source code with resources attached in zip file.