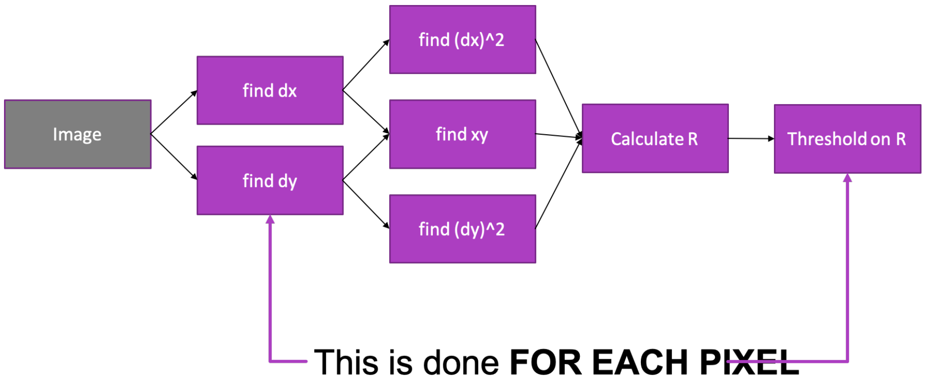
Accelerating the Harris Corner Detector through Multithreading

In this project, I implemented a basic harris corner detector in C++ and then accelerated it using OpenMP and CUDA. Image processing is an incredibly SIMD problem, lending itself incredibly well to acceleration through multithreading.

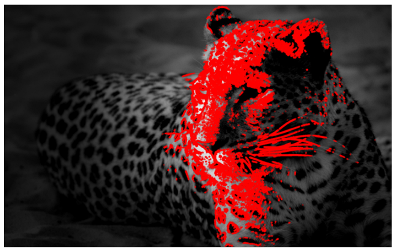


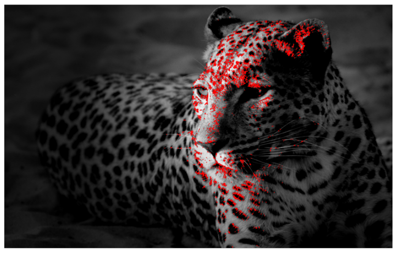
This is because all of the operations in the pipeline are done for each component pixel value, with each pixel value being performed as a stencil operation of the surrouding pixels. This means there isn't a ton of complicated control logic involved and for the most part, processing is embarrisingly parallel.

Verification

In order to verify my algorithm, I compared the output to the Harris Corner Detector implmemented in OpenCV. While I don't expect the results to be exactly the same, the OpenCV method serves as a "dipstick" verfication. Specifically, my implementaiton doesn't do the Gaussian Smoothing and non max supression usuallly found in the Harris Corner Detection.

Original


Accelerated


OpenCV Ground Truth

Speedups

The first method I used for speedup was OpenMP. Most of the loop iterations in this algorithm involved iterating over pixel values. I started with a version which serialized the data from a 2D array to a 1D array to improve memory access speeds. The example below shows the calcualtion of gradients `ix` and `iy` from the input values. Each iteration of this loop is independent of one another and as a result, it can immediately be parallelized through use of OpenMP.

Calculating Gradients

```
void HarrisCorner::calculate_gradients(float* ix, float* iy, float* input, int width, int he

    int x, y;
    int x_minus_one, x_plus_one;
    int y_minus_one, y_plus_one;
    float dx, dy;

    // Iterate over the entire image
    #pragma omp parallel for
    for(int ii = 0; ii < width * height; ++ii) {
        x = ii % width;
        y = ii / width;

        // Break out...
        if(x <= 1 || y <= 1 || x >= width - 2 || y >= height - 2) {
            continue;
        }

        x_minus_one = y*width + (x - 1);
        x_plus_one = y*width + (x + 1);
        y_minus_one = (y - 1)*width + x;
        y_plus_one = (y + 1)*width + x;

        dx = abs(input[x_minus_one] - input[x_plus_one]);
        dy = abs(input[y_minus_one] - input[y_plus_one]);

        ix[ii] = dx;
        iy[ii] = dy;
    }
}
```

This code is lends itself to CUDA. In CUDA, we spawn a single CUDA thread per pixel. From the CUDA thread/block indices and dimensions, we can calculate the x and y pixel position that each CUDA thread operates on. With this information we can also calculate the neighboring "stencil" to find the image gradients.

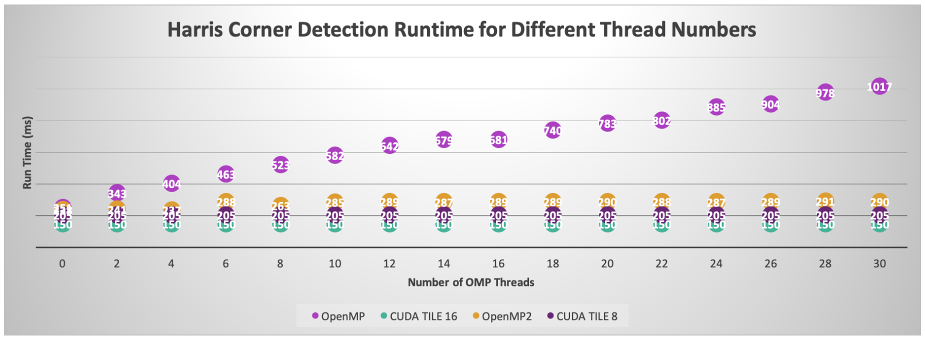
```
_global__ void calculate_response(unsigned char *input,
                       float *output,
                       unsigned int height,
                       unsigned int width){

    int x = blockIdx.x*TILE_SIZE+threadIdx.x;
    int y = blockIdx.y*TILE_SIZE+threadIdx.y;

    float k = .04;

    // This represents the first and last row/column of the image. sobel needs
    // a delta of at least one
    if(x <= 1 || y <= 1 || x >= width - 2 || y >= height - 2) {
        return;
    }

    // calculate the offsets for all regions of interest
    int offset = (y * width) + x;
    int x_minus_one = y*width + (x - 1);
    int x_plus_one = y*width + (x + 1);
    int y_minus_one = (y - 1)*width + x;
    int y_plus_one = (y + 1)*width + x;

    // Calculate dx, dy
    float dx = input[x_minus_one] - input[x_plus_one];
    float dy = input[y_minus_one] - input[y_plus_one];

    // Caclualte dx2, dy2, dxy
    float ix2 = dx*dx;
    float iy2 = dy*dy;
    float ixy = dx*dy;

    // Harris Corner Response Matrix:
    // [ix2, ixy;
    // ixy, iy2]
    float itrace    = ix2 + iy2;
    float idet      = (ix2*iy2) - (ixy*ixy);

    float response = abs(idet - (k * itrace * itrace));

    output[offset] = response;
}
```

One of the most important things to consider when working with the GPUs is to minimizes the number of data transfer back and forth between the GPU and the host machine. The result is a function that computes the entire Harris Corner Response in a single shot per thread as opposed to pipelining by calculating larger blocks at once and combining at a higher level (i.e. computing all `ix`, `iy` values, then computing all `ix^2`, `iy^2`, and `ixy`).

Another important consideration is maximizing the "Compute per Global Memory Access" (CGMA) ratio. This is achievable by reading global memory into a local variable and running from there. However, repeated accesses are still made to global memory as a result of the repeated access incurred by the stencil operations done as part of the gradient computations. This could be further optimized by performing a SINGLE read by each thread into shared memory and then further accessing the memory from there. While some overhead is incurred by the synchronization of threads between the pre-load and execution, memory access as a whole would increase.

Unfortuantely I wasn't able to implement this feature before the 8am deadline due to lack of cluster access as a result of a user using all 8 GPUs on Saturday for several hours.

Timing Results

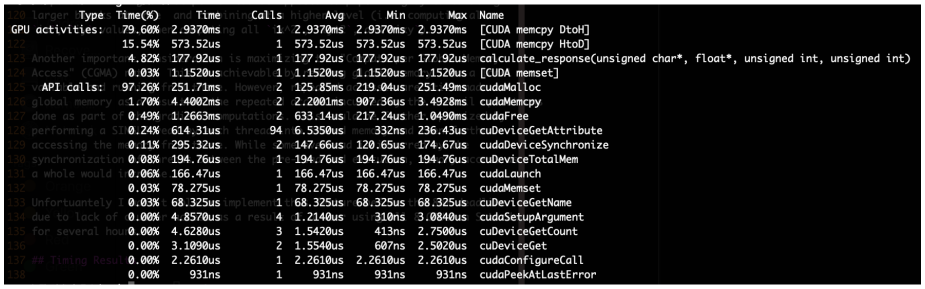


The above graphic shows the timing results achieved by the OpenMP and CUDA implementations. OpenMP had some interesting and incredibly vexing results. On one day, iterations with different OpenMP threads showed timing increasing by 100ms for each additional pair of threads being applied. These results don't quite make sense since the additional threads should work will with the embarissingly parallel problem. At the very least it should keep results consistent, not decrease them. On a different day, the same code yielded fairly consistent timings across the board.

More interesting is the resulting timing from the CUDA implementation. I ran the CUDA code with two different tile sizes: one with a 16x16 tile and another with an 8x8 tile size. The tiles here referred to the number of threads per thread block in each dimension. With a 2D problem, I created 2D tiles.

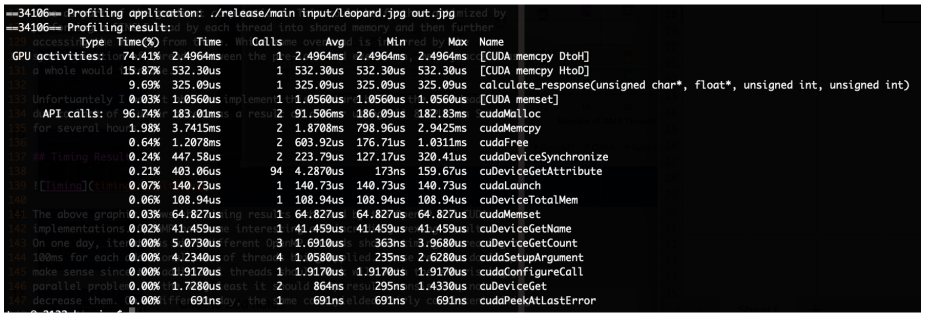
Here we see a timing of 150ms for the 16x16 block and ~200 ms for the 8x8 block. At first glance these results don't seem too much more impressive than the original CPU implementation until you consider the breakdown of the timing.

16x16 Tiling



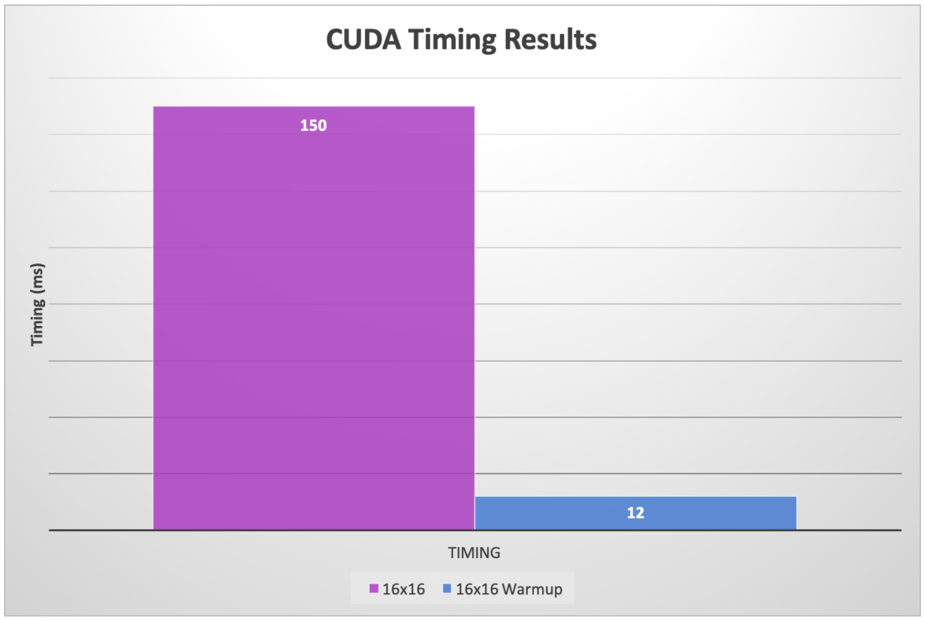
```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   79.60%  2.9370ms         1  2.9370ms  2.9370ms  2.9370ms  [CUDA memcpy DtoH]
                   15.54%  573.52us         1  573.52us  573.52us  573.52us  [CUDA memcpy HtoD]
                    4.82%  177.92us         1  177.92us  177.92us  177.92us  calculate_response(unsigned char*, float*, unsigned int, unsigned int)
                    0.03%  1.1520us         1  1.1520us  1.1520us  1.1520us  [CUDA memset]
      API calls:   97.26%  251.71ms         2  125.85ms  219.04us  251.49ms  cudaMalloc
                    1.70%  4.4002ms         2  2.2001ms  907.36us  3.4928ms  cudaMemcpy
                    0.49%  1.2663ms         2  633.14us  217.24us  1.0490ms  cudaFree
                    0.24%  614.31us        94  6.5350us     332ns  236.43us  cuDeviceGetAttribute
                    0.11%  295.32us         2  147.66us  120.65us  174.67us  cudaDeviceSynchronize
                    0.08%  194.76us         1  194.76us  194.76us  194.76us  cuDeviceTotalMem
                    0.06%  166.47us         1  166.47us  166.47us  166.47us  cudaLaunch
                    0.03%  78.275us         1  78.275us  78.275us  78.275us  cudaMemset
                    0.03%  68.325us         1  68.325us  68.325us  68.325us  cuDeviceGetName
                    0.00%  4.8570us         4  1.2140us     310ns  3.0840us  cudaSetupArgument
                    0.00%  4.6280us         3  1.5420us     413ns  2.7500us  cuDeviceGetCount
                    0.00%  3.1090us         2  1.5540us     607ns  2.5020us  cuDeviceGet
                    0.00%  2.2610us         1  2.2610us  2.2610us  2.2610us  cudaConfigureCall
                    0.00%     931ns         1     931ns     931ns     931ns  cudaPeekAtLastError
```

8x8 Tiling



```
==34106== Profiling application: ./release/main input/leopard.jpg out.jpg
==34106== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   74.41%  2.4964ms         1  2.4964ms  2.4964ms  2.4964ms  [CUDA memcpy DtoH]
                   15.87%  532.30us         1  532.30us  532.30us  532.30us  [CUDA memcpy HtoD]
                    9.69%  325.09us         1  325.09us  325.09us  325.09us  calculate_response(unsigned char*, float*, unsigned int, unsigned int)
                    0.03%  1.0560us         1  1.0560us  1.0560us  1.0560us  [CUDA memset]
      API calls:   96.74%  183.01ms         2  91.506ms  186.09us  182.83ms  cudaMalloc
                    1.98%  3.7415ms         2  1.8708ms  798.96us  2.9425ms  cudaMemcpy
                    0.64%  1.2078ms         2  603.92us  176.71us  1.0311ms  cudaFree
                    0.24%  447.58us         2  223.79us  127.17us  320.41us  cudaDeviceSynchronize
                    0.21%  403.06us        94  4.2870us     173ns  159.67us  cuDeviceGetAttribute
                    0.07%  140.73us         1  140.73us  140.73us  140.73us  cudaLaunch
                    0.06%  108.94us         1  108.94us  108.94us  108.94us  cuDeviceTotalMem
                    0.03%  64.827us         1  64.827us  64.827us  64.827us  cudaMemset
                    0.02%  41.459us         1  41.459us  41.459us  41.459us  cuDeviceGetName
                    0.00%  5.0730us         3  1.6910us     363ns  3.9680us  cuDeviceGetCount
                    0.00%  4.2340us         4  1.0580us     235ns  2.6280us  cudaSetupArgument
                    0.00%  1.9170us         1  1.9170us  1.9170us  1.9170us  cudaConfigureCall
                    0.00%  1.7280us         2     864ns     295ns  1.4330us  cuDeviceGet
                    0.00%     691ns         1     691ns     691ns     691ns  cudaPeekAtLastError
```

The function that computes the actual response (`calculate_response`) takes on the order of ~177-300 us to run, an ordr of magnitude less than the CPU counterpart. This shows that while the computation itself is significantly faster, a steep overhead is paid for setting up the problem. Namely, the `cudaMalloc` takes close to 97% of the time (~250ms). Looking more closely at the timing, it appears `cudaMalloc` is called twice, with the min and max valus ranging WILDLY, from ~250ms as the max to a min of ~220us. At the suggestion of one of my classmates I tried "warming up" the GPU by calling a malloc outside the first timing test to see what this would do to the results. The results were drastic.

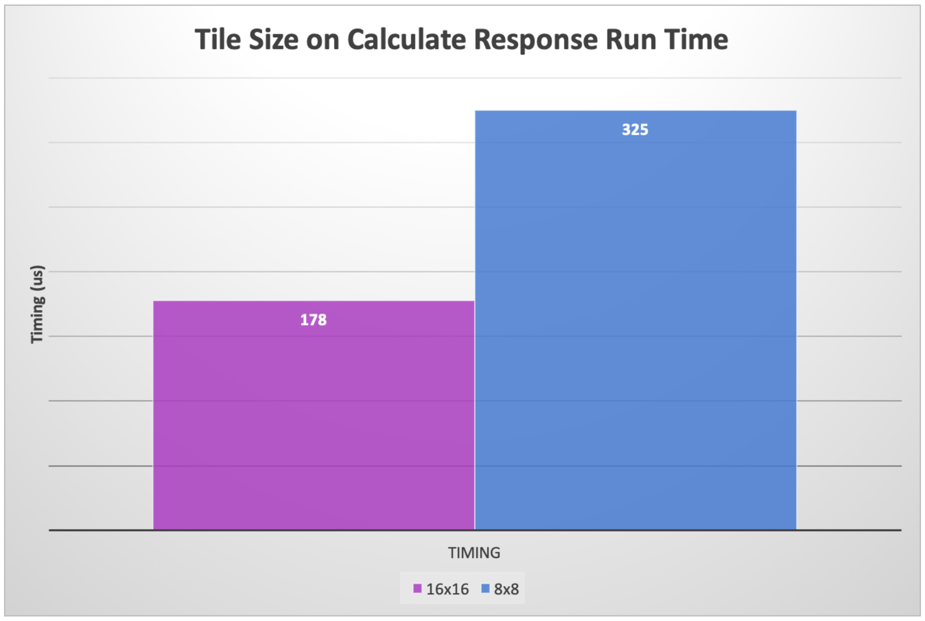


By adding in the warm up routine to prior to timing the run of the GPU, the `runtime` of the GPU code dropped 12ms. Its important to note here that the runtime itself did not *actually* change, just the timing of the run that did. This is because the first call to the GPU takes a bit longer to "initialize" the GPU and bring all systems online. When we were running the benchmark, this meant that the timing data for the first malloc was causing a higher run time. This is fixed when we "warmup" the program by allocating some memory (unused later on) prior to beginning the actual computations. In essence, this experiment showed the importance of the warmup procedure when evaluating benchmarks.

This is also extended to the CPUs which are run multiple times before the first computation actually takes place.

Tiling

As part of my experiments, I explored how different sized tiles played an impact on the timing of results. The previous two nvprof screen shots show the difference in timing between different sized tiles.



The size of the tile ultimately refers to how many threads are being launched per block running on a stream multiprocessor (SM). With larger tiles, fewer blocks of more threads are launched while with smaller tiles, more blocks with fewer threads are launch. This results in differences in caching since oversubscribing the number of blocks results in more context switches between blocks which can introduce an overhead.

A tradeoff to consider (in future implementations) is the size of shared memory associated to each block for different block sizes. Since multiple blocks can run on a single SM but each SM only has a finite amount of block size, the amount of shared memory each block can get is directly related to the number of blocks being launched.

Future Work

There were a number of components that I wanted to explore for this project that I unfortunatley was not able to explore.

Shared memory:

For improving the timing on a single image, the use of shared memory would have been one of the most interesting investigaitons to exploit existing parallelism. While shared memory is significantly samller than global memory, it is also significantly faster to access.

Asynchronous Access:

Asynchronous access could enable the simulataneous access and processing of data resulting in up to 2x speedups over the current implementation. This speedup comes from the ability of the GPU to begin processing data as soon as it arrives rather than waiting for all data to arrive before beginning the process it. However, this is dont by taking advantage of the rastorized nature of data transfers - data is sent row by row. For certain computations which require the blendeing of data in different columns AND rows, temporal constraints dictate that a row must have completed transfer before processing can begin. As a result, some algorithms may need to be revised in order to accomodate this optimization.

Multiple GPUs:

Multiple GPUS can become useul in splitting up problems even further to obtain a higher degree of parallelism. My intuition says for a single image the benefits of using mutliple GPUs may not outweight the overhead in coordination and that this sort of solution is likely better suited for the larger problem sizes.

3D data (Multiple pictures)

While note an additional optimization, increasing the problem size would provide a better sense o how his algorithms scales across a number of nodes. Temporal data (such as sucessive frames in a video) can be incredibly compute heavy depending on the rate/size of the problem and will be able to further take advantage of the parallel hardware to solve the problems.

Code

All code is publically available on GitHub at the following link. https://github.com/drewtu2/eece5640/tree/master/final_project/harris