

# Parallel Implementation of JPEG Codec

## Project Report GPU Computing

Blake Downey  
12/16/2022

### Description/overview of selected scientific/engineering problem requiring accelerated computing with CUDA/GPU

GPU Parallel Processing allows for the computational speed up of many problems, one problem of interest to me is parallel processing of image encoding and decoding. I think this is an interesting application space as data transfer of images can take a lot of time, and if the images need to be compressed on the way out and decompressed on the way in, having this process be an efficient one is important.

Because of this need for fast data transfer of images, we need a way to parallelize the compression. For this project, I chose JPEG compression and decompression (JPEG codec) for this application. JPEG codec is well suited for GPU computing for a few reasons. In Figure 1 below we see the multiple compression stages of a JPEG Encoder, and in Figure 2 we can note the steps of the image reconstruction via the JPEG Decoder. Because the mathematical process of the DCT, quantization and zig zag algorithms are repeated for all 8x8 blocks in the image, this brings about the opportunity for parallel processing

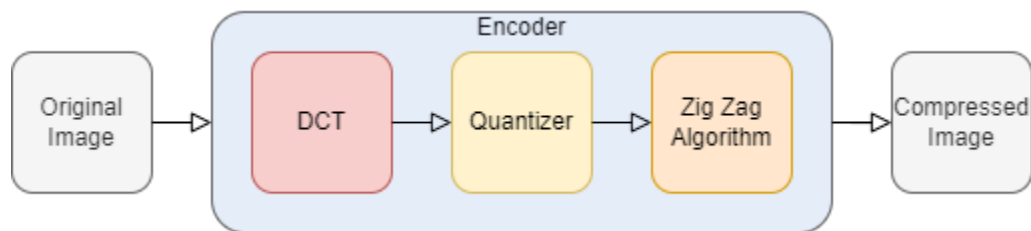


Figure 1: JPEG Encoder

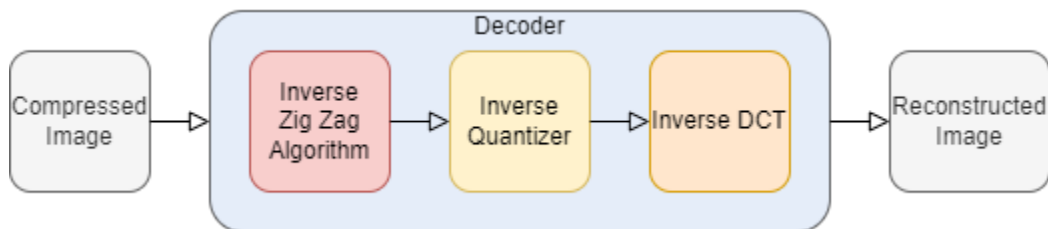


Figure 2: JPEG Decoder

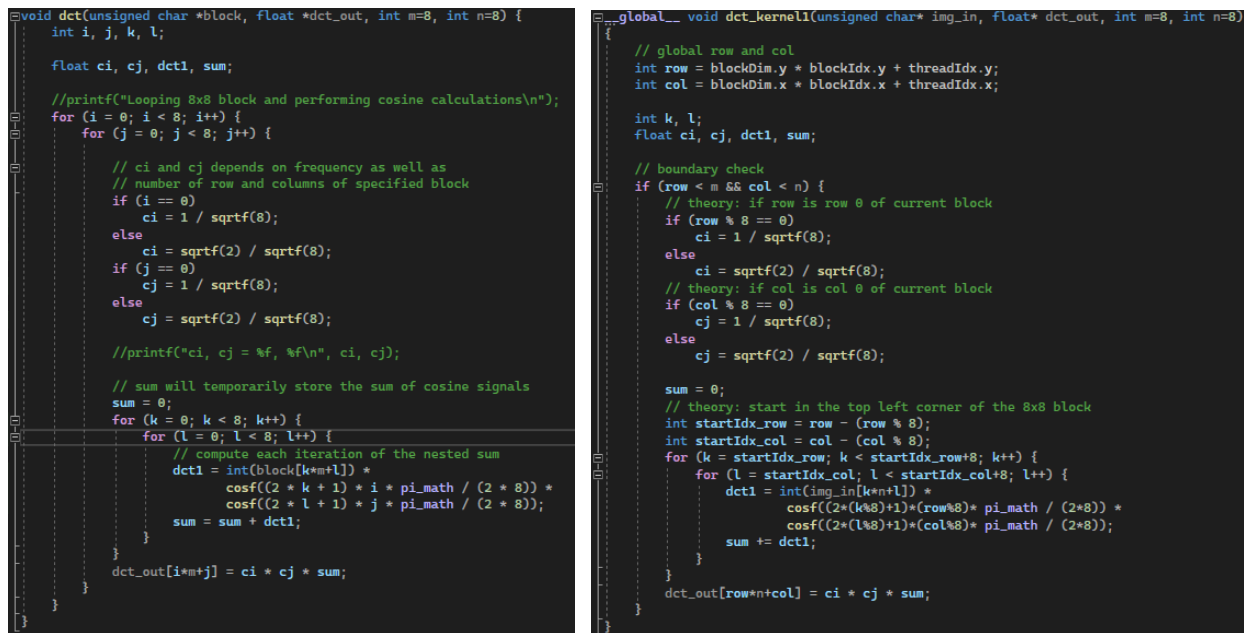
### Summary of Concept/Semantic and Implementation of parallelization

The serial CPU JPEG codec was my first task and from there I utilized the mathematical and algorithmic functions to convert the code to my parallel GPU version. Creating the GPU kernels from the CPU functions was harder than expected for certain kernels as access patterns were tricky. My parallel implementation has two versions (variations to dct and idct kernels), one

where I compute the cosine math in the kernel, and one where I precompute and then read from global memory instead, an analysis of why and how it performed is further on in the report.

The project flow followed this: CPU unit test for each function with matrices of different 8x8 multiples, CPU Codec (240x240 and 512x512 images), GPU unit test for each kernel with matrices of different 8x8 multiples, GPU Codec (240x240 and 512x512 images). The unit test functions can be found in **codec.cu** along with the functions for running the CPU and GPU versions. These can be toggled by setting the bools to the desired run type in the main function. This project ended with roughly 2000+ lines between the three files, which made chasing bugs quite the adventure!

Here in Figures 3 and 4 I have examples of functions I converted into CUDA kernels. Because there ended up being 6 CPU functions and 8 GPU kernels, I didn't feel the need to paste them all here as they can be found in the code (**codec\_serial.h** holds all of the serial functions, **codec\_parallel.h** holds all of the parallel kernels).



The figure displays two side-by-side code snippets. The left snippet is a C function named `dct` that performs a serial DCT on an 8x8 block. It uses nested loops for `i` and `j` (both from 0 to 7) to iterate over the block. Inside, it calculates `ci` and `cj` based on `i` and `j` using `sqrtf` and `cosf` functions. It then calculates a sum `sum` for each `i, j` pair and stores it in `dct_out[i*8+j]`. The right snippet is a CUDA kernel named `dct_kernel1`. It uses global indices `row` and `col` to access the input `img_in` and output `dct_out`. It includes a boundary check to ensure `row` and `col` are within the 8x8 block. It then calculates `ci` and `cj` and performs the DCT calculation for a single output element, updating `dct_out[row*8+col]`.

Figure 3: CPU DCT Function (Left) GPU DCT Kernel (Right)

In the left image of Figure 3, the serial DCT is performed on an 8x8 block. This code loops variables `i` and `j` which are values [0-7] and uses the inner for loop to perform the 64 iterations of dct computations for 1 output pixel of the 8x8 block. Overall this function loops 4096 times at its innermost loop for each 8x8 block in the image.

The right image of Figure 3 is the parallel implementation, which processes one output value per thread. Because this kernel only works on one output element that outer nested for loop in the serial version is not needed, and the cosine computations are now performed 64 times as it takes 64 computations to complete one output element. This kernel makes use of global index via the integers `row` and `col` defined at the top, performs boundary check to ensure no threads

outside the image bounds attempt to access the memory and has a relatively high register count per thread of 108. Because this register count is high, I was only able to allocate ~256 threads per block, which meant that I had to run more blocks on the grid and more overhead for scheduling.

In Figure 4 below I am showing the code for the precomputed cosine implementation of the dct kernel. In order to limit the number of registers used per thread and increase the number of threads I could launch in a block (to increase parallelism), I needed to precompute the cosine values. I did this in the first part of the main function in codec.cu and passed the 4096-d vectors for `cosk` and `cosl` to the GPU image function (`jpeg_codec_parallel_rr`), and then passed them to the DCT kernel.

This allowed me to save registers per thread and run 1024 threads in a block which increased my parallelism overall. This also decreased the time required to run the dct kernel as these values are stored in GMEM and accessed in a mostly coalesced manner in the inner for loop iterating `k` and `l`, meaning really high hit rates on the L1 Cache for reading the data. I also performed the same alterations to the idct kernel with slight change to the access pattern, this proved to reduce the time of the kernel as well.

```
global__ void dct_kernel2(unsigned char* img_in,
float* dct_out,
float* cosk, float* cosl,
int m = 8, int n = 8)
{
    // global row and col
    int row = blockDim.y * blockIdx.y + threadIdx.y;
    int col = blockDim.x * blockIdx.x + threadIdx.x;

    int k, l;
    float ci, cj, dct1, sum;

    // boundary check
    if (row < m && col < n) {
        // theory: if row is row 0 of current block
        if (row % 8 == 0)
            ci = 1 / sqrtf(8);
        else
            ci = sqrtf(2) / sqrtf(8);
        // theory: if col is col 0 of current block
        if (col % 8 == 0)
            cj = 1 / sqrtf(8);
        else
            cj = sqrtf(2) / sqrtf(8);

        sum = 0;

        // theory: start in the top left corner of the 8x8 block
        int startIdx_row = row - (row % 8);
        int startIdx_col = col - (col % 8);
        for (k = startIdx_row; k < startIdx_row + 8; k++) {
            for (l = startIdx_col; l < startIdx_col + 8; l++) {
                dct1 = int(img_in[k * n + l]) *
                    cosk[(row % 8) * 512 + (col % 8) * 64 + (k % 8) * 8 + (l % 8)] *
                    cosl[(row % 8) * 512 + (col % 8) * 64 + (k % 8) * 8 + (l % 8)];
                sum += dct1;
            }
        }
        dct_out[row * n + col] = ci * cj * sum;
    }
}
```

Figure 4: DCT Kernel with Precomputed Cosine Values

In Figure 5 below I have the CPU Quantization function and the GPU Quantization Kernel. This conversion was fairly straightforward. The serial quantization function takes in an 8x8 block and quantizes it with the defined q 64-d vector. This was element for element in a nested for loop iterating 64 times in the inner loop.

The parallel version makes use of shared memory to store the quantization matrix that is stored in GMEM and passed by reference as argument `int* gmem_q`, and also like the dct kernel, calculates one output value per thread. I load the SMEM q vector as 1D, `ds_q`, and call a `syncthreads` so that threads that aren't loading SMEM wait until the loading is done and don't try to access data that is not yet loaded. I then check the bounds so only threads that are in the image bounds are processing data, and then perform the single element quantization using the global index values. For the q matrix I needed to be smart about how I accessed the element and I found that modding the row and col by 8 would give me what I needed to access the correct elements.

```
void quant(float *block, int *quant_out) {
    int i, j;

    int q[64] = {16, 11, 10, 16, 24, 40, 51, 61, 12, 12, 14, 19, 26, 58, 60, 55, 14, 13, 16, 24, 40, 57, 69, 56,
                14, 17, 22, 29, 51, 87, 80, 62, 18, 22, 37, 56, 68, 109, 103, 77, 24, 35, 55, 64, 81, 104, 113, 92,
                49, 64, 78, 87, 103, 121, 120, 101, 72, 92, 95, 98, 112, 100, 103, 99};

    for (i=0; i<8; i++) {
        for (j=0; j<8; j++) {
            quant_out[i*8+j] = block[i*8+j] / q[i*8+j]; //implicit cast to int
        }
    }
}

__global__ void quant_kernel1(float* dct_out, int* quant_out, int* gmem_q, int m, int n)
{
    // q shared mem
    __shared__ int ds_q[BLOCK_SIZE];

    // tx and ty and row/col
    int tx = threadIdx.x; int ty = threadIdx.y;
    int row = blockIdx.y * blockDim.y + ty;
    int col = blockIdx.x * blockDim.x + tx;

    // load shared mem for q
    if (tx < BLOCK_WIDTH && ty < BLOCK_WIDTH) {
        ds_q[tx * BLOCK_WIDTH + ty] = gmem_q[tx * BLOCK_WIDTH + ty];
    }

    // required syncthreads - prevent invalid shared memory accesses
    __syncthreads();

    // check bounds
    if (row < m && col < n) {
        quant_out[row * n + col] = dct_out[row * n + col] / ds_q[(row % 8) * BLOCK_WIDTH + (col % 8)];
    }
}
```

Figure 5: CPU Quant Function (Top) GPU Quant Kernel (Bottom)

For a quick explanation of the zigzag and rle and their counterparts izigzag and irle, the serial to parallel conversion was straightforward, using similar access patterns described above with modding row and col by 8, I created the desired output and tested it for correctness with the

GPU unit tests. There were a lot of GPU unit tests and including them all here does not contribute to the overall analysis of the project. If you are interested in the unit tests check the `codec.cu` file for the function called `'gpu_unit_test'`, it's actually pretty lengthy (over 300 lines of debugging code) and details each step of the kernel output values.

The design I opted for was different in my sequential and parallel implementations. In my sequential version, I process each 8x8 block through the codec and write to the output sequentially. Since I was only interested in the end result for the CPU version I didn't feel it necessary to save the encoded version in the CPU code.

The method in the GPU version differs from this as the GPU computes the image 8x8 blocks simultaneously through each kernel. So the whole image is passed through the codec at once and the encoded format is stored in the parameter called `'encoded'` in the main method which is a vector of lists. I opted for this format as not every 8x8 block gets compressed to be the same number of bytes so statically allocating space for this was not possible to have the least amount of bytes. Because of this, and because the GPU cannot use the `std` library, I computed the encoded vector of lists on the CPU after the image was passed through the GPU compression kernels (dct, quant, zigzag, and rle). This slowed down my GPU version overall but for the 512x512 image it is still faster than the CPU version. As the premise of this project was compressing the data to send somewhere else, I store the encoded GPU version in this variable should the data be transferred, this variable can be used. For demonstration purposes of the codec, the GPU version takes the encoded version and decodes the vector of lists on the CPU into two int arrays and performs the image reconstruction on the GPU (irle, izigzag, iquant, and idct).

Another task in the implementation of the GPU code was the Kernel Launch Configuration. Since I have two image sizes, 240x240 and 512x512, I opted for statically setting the launch configs as doing it dynamically proved difficult and unreliable.

For the 240x240 image, I needed 57,600 threads for the DCT, Quant, iQuant and iDCT kernels as those use one thread per output element. This meant that I would need multiple blocks of multiple threads. In the original GPU version with computing cosine values in the kernel, I was capped at ~256 threads in a block, making the blockDim 16x16, which pushed the gridDim to be 15x15. When I used precomputed values, lowering the registers per thread, I was able to allocate 1024 threads per block and only need a dimGrid of 8x8. The same was done for the 512x512 image where I needed 262,144 threads. In the original version I used 16x16 thread blocks and 32x32 gridDim. This was pretty serialized as there were 1024 blocks that needed to be run sequentially. After implementing precomputing, I was able to run these kernels with 1024 threads per block and only 16x16 blocks in a grid.

For the Zigzag, RLE, iZigzag and iRLE kernels I needed the thread count to equal the number of 8x8 blocks in the image or there were threads overwriting data and producing incorrect outputs. For the 240x240 image, that produces 900 blocks. To allocate this, I used 10x10 thread blocks and a 3x3 grid. I found that this was significantly faster than doing 30x30 thread blocks

and a 1x1 grid. I am still not sure I understand this, but found that it was overall faster by a couple hundred microseconds (which is a large difference when discussing GPU computing). For the 512x512 image there are 4096 8x8 blocks, and with 32x32 threads per block and 2x2 grid, I was able to allocate exactly what I needed.

### **Identify parallel patterns used**

In the process of implementing the GPU kernels, I found that the use of shared memory was critical for this application, as it has been in previous assignments. In the quantization and Inverse quantization kernels I pass the q matrix in as a pass by reference variable which was loaded onto the GMEM. In the kernel I use threads where tx and ty are both less than 8, so the first 64 threads allocated in the block copy the GMEM to the SMEM. Because this creates some control divergence and I don't want the other threads in the block to access the SMEM before it's filled, I utilize a syncthreads call after loading to ensure this memory is loaded and read properly.

As described with Figure 4 above, due to the larger amount of computations needed in the dct and idct kernels with the cosine functions, a large number of registers was needed per thread. Considering the necessary trade off between threads per block and registers per thread, I implemented a precomputed GMEM access for these kernels to reduce the required registers per thread allowing me to run more threads per block. By using this type of memory I was able to decrease the time per dct and idct kernel runs, this can be seen in the timing difference between dct\_kernel1 and dct\_kernel2 (and idct) in Figures 14 & 15 and Figures 16 & 17.

### **Include results of outputs produced by successful runs**



Figure 6: Original 240x240 Image

In this section I am providing the results of my JPEG Codec, the CPU and GPU versions. In Figure 6 above is the original input image that is 240x240, also Figure 10 is the original 512x512 image.

Turning to the below Figure 7, we can see the images are identical, and after performing both of the cpu and gpu versions of the code, I perform a simple by pixel mean absolute error calculation that shows in Figure 8 that there is an mae of 0 between the CPU and GPU versions for the 240x240 image. Figure 9 shows the output from the run with the reduced register

method. Here it showed a decrease in the timing after implementing the reduced register method for the parallel code from 0.108704 seconds to 0.095308 seconds.

An important thing to note is that the CPU version actually took less time than the GPU version for the 240x240 image. This was surprising to me as the GPU kernels are very parallelized, but with the required use of the CPU in the middle of the GPU codec, this makes sense as there is lots of overhead when transferring the memory to and from the device. For a more accurate sum of the time required for the GPU kernels, the Figures 14-17 provide better timings.



Figure 7: 240x240 Image CPU Output (Left) GPU Output (Right)

```
Running GPU JPEG Compression on sample 240x240 image:
Compression from 57600 to 14880 bytes
Parallel Code took: 0.108704 seconds | 108703.899998 microseconds

Running CPU JPEG Compression on sample 240x240 image:
Sequential Code took: 0.049149 seconds | 49148.799997 microseconds

MAE between CPU and GPU versions of reconstructed images is 0
```

Figure 8: 240x240 Image CPU vs GPU Verification and Timing

```
Running GPU JPEG Compression on sample 240x240 image:
Compression from 57600 to 14880 bytes
Parallel Code took: 0.095308 seconds | 95307.599986 microseconds

Running CPU JPEG Compression on sample 240x240 image:
Sequential Code took: 0.048308 seconds | 48307.600024 microseconds

MAE between CPU and GPU versions of reconstructed images is 0
```

Figure 9: 240x240 Image CPU vs GPU Verification and Timing - Precompute DCT & iDCT

Moving onto the 512x512 image, Figure 11 below shows the comparison of the two output images for the CPU and GPU versions and that plus the MAE difference of 0 in Figure 12 shows the verification that the GPU version worked flawlessly for the bigger image size as well. Also in Figure 12 we can note that the overhead of transferring to and from the device is made up by the parallelism of the GPU kernels as the GPU version took less than half the time of the CPU version.



Figure 10: Original 512x512 Image



Figure 11: 512x512 Image CPU Output (Left) GPU Output (Right)

```
Running GPU JPEG Compression on sample 512x512 image:  
Compression from 262144 to 56614 bytes  
Parallel Code took: 0.108497 seconds | 108496.600005 microseconds  
  
Running CPU JPEG Compression on sample 512x512 image:  
Sequential Code took: 0.235478 seconds | 235478.299990 microseconds  
  
MAE between CPU and GPU versions of reconstructed images is 0
```

Figure 12: 512x512 Image CPU vs GPU Verification and Timing



Like the 240x240 image, I also used the reduced register method on the 512x512 image. Also similarly to the smaller image, the timings are not very accurate here in Figure 13, to see a difference in the timings Figure 16 and 17 hold the timing difference between regular 512x512 and reduced register 512x512. The host verification also succeeds with this method and the larger image shown below with MAE of 0.

```
Running GPU JPEG Compression on sample 512x512 image:
Compression from 262144 to 56614 bytes
Parallel Code took: 0.115689 seconds | 115688.599995 microseconds

Running CPU JPEG Compression on sample 512x512 image:
Sequential Code took: 0.222609 seconds | 222609.199991 microseconds

MAE between CPU and GPU versions of reconstructed images is 0
```

Figure 13: 512x512 Image CPU vs GPU Verification and Timing - Precomputed DCT & iDCT

### Source code for host & kernel(s)

Due to the vast length of the host and kernel code, I opted to not add them to the report as there are over 2000+ lines. Instead please find them in the **deliverables/code** folder. The **codec\_serial.h** is all functions related to the serial codec, **codec\_parallel.h** contains all 8 kernels used in the GPU version of the codec. The file **codec.cu** file contains the main function and functions for unit testing the cpu code, gpu code, and 3 functions for calling the codecs on the images, one for the CPU codec, one for the GPU codec with the original version of the dct and idct kernels and one for the reduced register GPU codec version. For reading and writing images I used the source code files **stb\_image.h** and **stb\_image\_write.h** that were provided to us in a prior assignment. Lastly for my checkCuda call that ensures there are no cuda errors is defined in **CUDA\_error\_check.h**.

In the main function of the codec.cu file, different types of runs can be selected by altering the boolean values at the top of the main function, this allows the user to run unit tests with the change of 1 value.

### Profiling reports and analysis discussion of key performance metrics using Nsight Compute

Since I used profileapi.h to time my code execution and not the nvtx timers, as was discussed in class (either option was viable), using NSight Systems was a bit tricky. I generated some reports (for the codec\_240 and codec240\_rr executables) and tried to dig through the timeline view, but couldn't find the kernel executions, or anything I couldn't already find with NSight Compute. Because of this, and the little instructions given on NSight Systems, I decided to perform all my analysis with NSight Compute reports since we have been using that software for most of the quarters assignments.

For the following parts I used 4 different executable versions of the GPU code: JPEG\_Codec\_240.exe, JPEG\_Codec\_240\_rr.exe, JPEG\_Codec\_512.exe, JPEG\_Codec\_512\_rr.exe. This is two for each image, one with cosine functions performed in the dct and idct kernels and one (\_rr versions) with precomputed values.

ID	Issues	Detects	Function Name	Demangled Name	Process	Device N Grid Size	Block Size	Cycles [cycle]	Duration [usecond]	Compute Thrc	Memory Throughput	# Registers [register/thread]
0	11	dct_kernel1	dct_kernel1(unsigne...	[25772] J...	NVIDL...	15, 15, 1	16, 16, 1	84,357	60.19	57.64	7.77	108
1	14	quant_kernel1	quant_kernel1(float *, int...	[25772] JP...	NVIDL...	15, 15, 1	16, 16, 1	4,622	4.38	14.37	16.01	16
2	17	zigzag_kernel1	zigzag_kernel1(int *, int ...	[25772] JP...	NVIDL...	3, 3, 1	10, 10, 1	21,498	16.58	1.54	24.03	38
3	16	rie_kernel1	rie_kernel1(int *, int *, int ...	[25772] JP...	NVIDL...	3, 3, 1	10, 10, 1	167,424	119.52	1.36	3.45	16
4	16	irie_kernel1	irie_kernel1(int *, int *, int ...	[25772] JP...	NVIDL...	3, 3, 1	10, 10, 1	36,383	26.88	3.66	7.99	22
5	17	izigzag_kernel1	izigzag_kernel1(int *, int ...	[25772] JP...	NVIDL...	3, 3, 1	10, 10, 1	19,172	14.62	1.78	18.90	24
6	14	iquant_kernel1	iquant_kernel1(int *, int ...	[25772] JP...	NVIDL...	15, 15, 1	16, 16, 1	4,652	4.42	14.28	30.55	16
7	15	idct_kernel1	idct_kernel1(int *, unsign...	[25772] JP...	NVIDL...	15, 15, 1	16, 16, 1	94,950	68.29	73.59	6.70	94

Figure 14: JPEG\_Codec\_240 NSight Compute Summary

As referenced a few times prior, let's look at the speed up of using the reduced registers precomputing method timing difference. In the above Figure 14, the dct\_kernel1 achieves 60.19 useconds timing, and in the below Figure 15 the timing is reduced to 54.27 useconds. This is a small decrease, but exemplified more with larger images. Also the same thing happens with the idct\_kernel1 vs idct\_kernel2, the timing goes from 68.29 useconds to 23.87 useconds. This was a larger speedup and contributed to the speed up we saw in the reduced register method with the 240x240 image.

ID	Issues	Detects	Function Name	Demangled Name	Process	Device N Grid Size	Block Size	Cycles [cycle]	Duration [usecond]	Compute Thrc	Memory Throughput	# Registers [register/thread]
0	16	dct_kernel2	dct_kernel2(unsigne...	[21124] J...	NVIDL...	8, 8, 1	32, 32, 1	75,234	54.27	26.27	73.52	40
1	14	quant_kernel1	quant_kernel1(float *, int...	[21124] JP...	NVIDL...	8, 8, 1	32, 32, 1	5,759	5.12	12.78	12.98	16
2	17	zigzag_kernel1	zigzag_kernel1(int *, int ...	[21124] JP...	NVIDL...	3, 3, 1	10, 10, 1	21,684	15.94	1.55	23.83	38
3	16	rie_kernel1	rie_kernel1(int *, int *, int ...	[21124] JP...	NVIDL...	3, 3, 1	10, 10, 1	166,442	119.14	1.37	1.73	16
4	16	irie_kernel1	irie_kernel1(int *, int *, int ...	[21124] JP...	NVIDL...	3, 3, 1	10, 10, 1	36,361	26.85	3.66	8.00	22
5	17	izigzag_kernel1	izigzag_kernel1(int *, int ...	[21124] JP...	NVIDL...	3, 3, 1	10, 10, 1	19,333	14.91	1.76	18.75	24
6	14	iquant_kernel1	iquant_kernel1(int *, int ...	[21124] JP...	NVIDL...	8, 8, 1	32, 32, 1	5,631	5.02	13.08	21.88	16
7	14	idct_kernel2	idct_kernel2(int *, unsign...	[21124] JP...	NVIDL...	8, 8, 1	32, 32, 1	31,534	23.87	62.79	62.79	40

Figure 15: JPEG\_Codec\_240\_rr NSight Compute Summary

ID	Issues	Detects	Function Name	Demangled Name	Process	Device N Grid Size	Block Size	Cycles [cycle]	Duration [usecond]	Compute Thrc	Memory Throughput	# Registers [register/thread]
0	11	dct_kernel1	dct_kernel1(unsigne...	[12976] J...	NVIDL...	32, 32, 1	16, 16, 1	362,828	258.24	60.90	8.21	108
1	15	quant_kernel1	quant_kernel1(float *, int...	[12976] JP...	NVIDL...	32, 32, 1	16, 16, 1	11,827	9.15	25.61	59.13	16
2	15	zigzag_kernel1	zigzag_kernel1(int *, int ...	[12976] JP...	NVIDL...	2, 2, 1	32, 32, 1	152,488	109.34	1.20	16.20	38
3	16	rie_kernel1	rie_kernel1(int *, int *, int ...	[12976] JP...	NVIDL...	2, 2, 1	32, 32, 1	212,548	151.49	4.23	6.10	16
4	16	irie_kernel1	irie_kernel1(int *, int *, int ...	[12976] JP...	NVIDL...	2, 2, 1	32, 32, 1	210,664	148.06	2.45	11.86	22
5	15	izigzag_kernel1	izigzag_kernel1(int *, int ...	[12976] JP...	NVIDL...	2, 2, 1	32, 32, 1	141,010	100.38	0.86	14.58	24
6	14	iquant_kernel1	iquant_kernel1(int *, int ...	[12976] JP...	NVIDL...	32, 32, 1	16, 16, 1	9,009	9.25	33.62	37.12	16
7	15	idct_kernel1	idct_kernel1(int *, unsign...	[12976] JP...	NVIDL...	32, 32, 1	16, 16, 1	419,696	296.93	75.64	6.89	94

Figure 16: JPEG\_Codec\_512 NSight Compute Summary

Comparing these two Figures 16 & 17, we see a similar performance improvement with dct\_kernel1 to dct\_kernel2 taking 258.24 useconds to 183.33 useconds. As well as the improvement from idct\_kernel1 to idct\_kernel2 taking 296.93 useconds to 72.51 useconds. This is a huge improvement in timing, but because most of the time required for the GPU version is taken up with the overhead of transferring back and forth between device and host and performing the the encoding on the CPU side with the vector of lists, we didn't see an overall improvement in timing for the 512 image. In order for the GPU version to reflect the actual time the kernels took, the encoding CPU steps can be skipped if the code just needs to be used to compress the image and decompress to jpeg.

ID	Issues	Detects	Function Name	Demangled Name	Process	Device N Grid Size	Block Size	Cycles [cycle]	Duration [usecond]	Compute Thrc	Memory Throughput	# Registers [register/thread]
0	14	dct_kernel2	dct_kernel2(unsigne...	[12920] J...	NVIDL...	16, 16, 1	32, 32, 1	257,218	183.33	32.73	91.61	40
1	13	quant_kernel1	quant_kernel1(float *, int...	[12920] JP...	NVIDL...	16, 16, 1	32, 32, 1	13,352	11.23	22.68	25.41	16
2	15	zigzag_kernel1	zigzag_kernel1(int *, int ...	[12920] JP...	NVIDL...	2, 2, 1	32, 32, 1	152,853	109.22	1.20	16.14	38
3	16	rie_kernel1	rie_kernel1(int *, int *, int ...	[12920] JP...	NVIDL...	2, 2, 1	32, 32, 1	214,312	152.03	4.20	6.05	16
4	16	irie_kernel1	irie_kernel1(int *, int *, int ...	[12920] JP...	NVIDL...	2, 2, 1	32, 32, 1	207,664	148.99	2.49	12.02	22
5	15	izigzag_kernel1	izigzag_kernel1(int *, int ...	[12920] JP...	NVIDL...	2, 2, 1	32, 32, 1	140,504	102.43	0.86	14.66	24
6	13	iquant_kernel1	iquant_kernel1(int *, int ...	[12920] JP...	NVIDL...	16, 16, 1	32, 32, 1	12,752	11.20	23.76	26.59	16
7	11	idct_kernel2	idct_kernel2(int *, unsign...	[12920] JP...	NVIDL...	16, 16, 1	32, 32, 1	101,280	72.51	83.42	83.42	40

Figure 17: JPEG\_Codec\_512\_rr NSight Compute Summary

## 1. roofline, speed of light, memory workload, etc

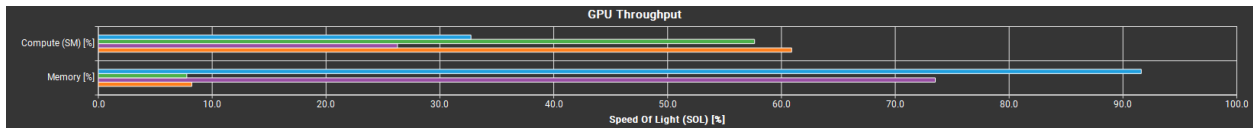


Figure 18: GPU Speed of Light Throughput (Compute and Memory) Comparison

I added baselines for each of the runs and the results are pretty interesting. In the above Figure 18, I am showing the GPU Speed of Light Analysis. For reference, the green bars are the 240x240, the purple bars are 240x240\_rr, the orange bars are 512x512, and lastly the blue bars are the 512x512\_rr. A brief explanation of what we should see between the two types of runs for each image. When I implemented the GMEM based precomputing to reduce the registers, I exchanged compute time for memory time, so what we should see is the compute bar go down and the memory bar go up, and this is reflected in the above bar graph.

Taking the 512x512 example for reference because it's a little easier to pick those colors apart on the graph. The orange bar which should reflect high compute does, and when we compare that to the blue bar showing the reduced register method, it shows a large decrease in the compute throughput. This makes sense and is what I was expecting as I am now waiting on the GMEM to be read. Looking at the same for the memory throughput section in Figure 18, we see the expected results: the orange bar has low memory throughput and the blue bar has significantly higher memory throughput as we are loading from GMEM.

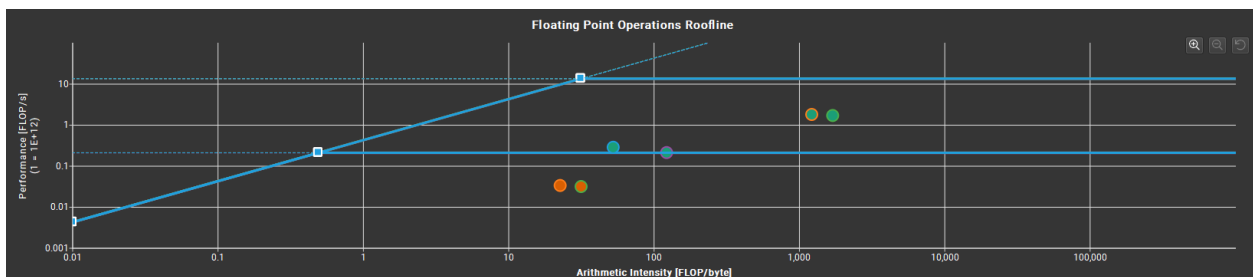


Figure 19: Roofline Analysis Comparison

Turning the analysis to the roofline analysis in Figure 19, we can expect the regular versions to be more compute bound and the reduced register versions to be more memory bound. It might be a little hard to see but hopefully you can zoom in and see the graph. I also included the reports in the deliverables! The two points in the upper right are the regular versions and their positions represent more compute bound regions as they have higher Ai (Arithmetic Intensity in FLOP/byte) and higher Performance in FLOP/s. This makes sense as these kernels are doing more computations per second and per byte of data. When I swapped the reduced register method the two points on the horizontal line are reflecting these runs. These are more memory bound with lower Ai and Performance as they are performing less computations per second and byte of data. There are also two more points below the lower roofline, these are double precision and showed up due to the use of the cosf cosine functions performing double

precision calculations in the original dct and idct kernels. These values are supported in the below Figure 20 which shows the Compute Workload Analysis comparison of 240x240 and 240x240\_rr. In the figure the green line reflects the original version and the blue line reflects the reduced register method. This shows the use of FP84 (double precision) in the original version where cosf is used, when those computations are removed the LSU line shoots up as those reduced register kernels are simply reading more data from GMEM. A similar result shows up for the 512x512 image comparison.

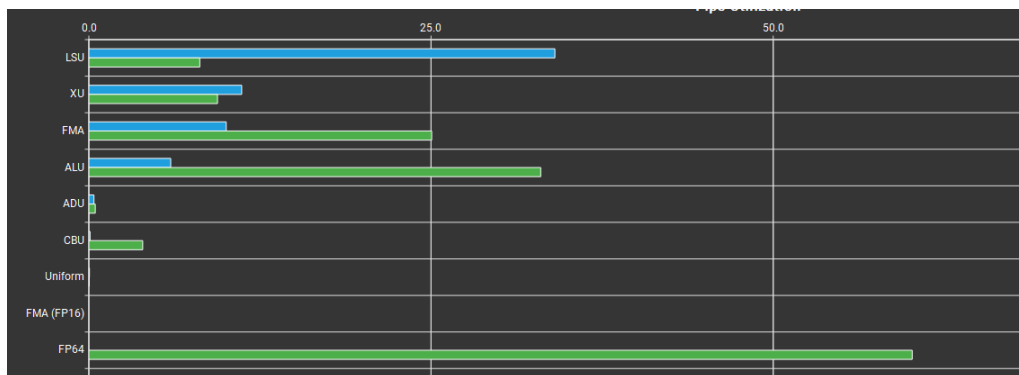


Figure 20: Compute Workload Analysis comparison of 240x240 and 240x240\_rr

In terms of memory workload analysis I'll use the 240x240 and 240x240\_rr example shown in Figure 21 below. In the original version that has high compute throughput (top image), we see that the sectors / request is 1. This is a 1:1 ratio of requests to sectors, which indicates that the requests are not coalesced. The Global requests that are happening in this kernel as the global reads from the image data. These are performing the 8x8 block accesses so it makes sense that these are not coalesced. However the Hit Rate is quite high at 97.12% which is indicative of the L1 cache storing the 8 bytes of data per row of the 8x8 block correctly.

	Instructions	Requests	Wavefronts	% Peak	Sectors	Sectors/Req	Hit Rate	Bytes
Local Load	0	0	0	0	0	0	0	0
Global Load	115,200	115,200			115,200	1	97.12	3,686,400
Global Load To Shared Store (access)	0	0	115,200	3.60	0	0	-	0
Global Load To Shared Store (bypass)	0	0	0	0	0	0	-	0
Surface Load	0	0	0	0	0	0	0	0
Texture Load	0	0	0	0	0	0	0	0
Global Store	1,800	1,800	2,043	0.06	7,200	4	7.78	230,400
Local Store	0	0	0	0	0	0	0	0

	Instructions	Requests	Wavefronts	% Peak	Sectors	Sectors/Req	Hit Rate	Bytes
Local Load	0	0	0	0	0	0	0	0
Global Load	368,640	368,640			2,142,720	5.81	98.06	68,567,040
Global Load To Shared Store (access)	0	0	615,168	21.57	0	0	-	0
Global Load To Shared Store (bypass)	0	0	0	0	0	0	-	0
Surface Load	0	0	0	0	0	0	0	0
Texture Load	0	0	0	0	0	0	0	0
Global Store	1,920	1,920	1,980	0.07	7,200	3.75	0	230,400
Local Store	0	0	0	0	0	0	0	0

Figure 21: Memory Workload Analysis L1 Cache Original (Top) Reduced Register (Bottom)

Looking at the effect of using the GMEM in the reduced register method (bottom image) we have two more reads for memory in this kernel per inner loop iteration, reading cosk and cosl values. These are arranged in a 1D memory pattern in an effort to increase sectors per request to be more coalesced. The value is 5.81 sectors per request. Essentially what I believe

happened here is the inner loop iterating k and l accessed the image in the same way (uncoalesced 8x8 blocks) but the GMEM reads to get the cosk and cosl values, the memory is coalesced as the cosk and cosl variables are written the same way that they are accessed. I was expecting the sectors per request to be ~4 as the code is written in such a way that the memory accesses should be fully coalesced for reading the cosk and cosl values, but given the higher than 4 sectors per request I think something went wrong with the theory -> actual. Looking at the Hit Rate for L1 Cache we still have a very high number 98.06% which is good and indicates the adjacent threads are reading memory that is close together when read from GMEM.

In both the versions (top and bottom images) we can expect to see the global store close or equal to 4, which it is. This is because each thread computes one output element and stores them one after the other.

In terms of control divergence, there is room for control divergence in a few different areas throughout the kernels. Firstly we can look at the kernels using 1 thread per output element and then look at the ones using 1 thread per 8x8 block as these use different launch configs.

The DCT, Quant, iQuant and iDCT kernels all use 1 thread per output element. Because there is an overallocation of threads for the image, there is some control divergence between the threads that perform computations on the data and some that do not get utilized. This could be optimized with better launch configurations and more time to consider better configurations. Besides this control divergence, there is also some present in the dct kernel where ci and cj are set, but because these are small code sections and if else block I don't believe this has much effect on the performance. However, in the quantization kernel the control divergence is more present as some threads load to SMEM and other threads wait at the syncthreads call, this hurts performance as some threads are waiting but the use of SMEM for the 64-d quant matrix makes up for this gap. The same control divergence scenarios are seen in the iquant and idct kernels.

The Zigzag, RLE, iRLE and iZigZag kernels are launched with the exact number of threads I need to calculate the output for the image, and these are simply memory bound kernels as they are loading and storing the data in different formats for compression. The zig zag and izig zag kernels have shockingly poor performance with sectors / request reaching 25. This is due to the type of format the GPU needs to change the data to be in. These kernels are huge bottlenecks for the codec and take up a significant amount of the processing time. This is mainly because of the large amounts of uncoalesced memory reads and writes to GMEM. I think a better way to perform these would require a rewrite of the kernels entirely as this algorithm needs to access data in such a way that provides highly uncoalesced accesses. Supporting this bottleneck we can look at the Avg. Active Threads Per Warp count for each of the kernels: 25, 3.08, 10.50, and 25 respectively. These stats plus the incredibly low theoretical vs actual occupancy goes to show that these kernels are damping performance by a lot. I believe this is because the access patterns on the memory and the fact that its an 8x8 block being processed which is nearly completely uncoalesced access. A possible solution for this could be to unravel the 8x8 blocks

into memory with a type of unraveling kernel. This kernel would perform really poorly but would allow the matrix to be read in faster for the zigzag kernel. However, this does not change the overall performance because the data would only be used once in that format: for the zigzag kernel.

The levels of parallelism can be summarized with the information in Table 1 below. The numbers represent the achieved occupancy / theoretical occupancy in percentage. These are grabbed from the 240x240 and 240x240\_rr profiles of NSight Compute.

%	dct	quant	zigzag	rle	irle	izigzag	iquant	idct	avg
Original Method	93.43	78.08	7.89	5.91	6.97	7.57	75.46	92.41	45.97
Reduced Register Method	94.27	89.17	7.88	5.90	7.00	7.70	93.70	94.03	49.96

Table 1: 240x240 vs 240x240\_rr Occupancy Comparison

Highlighted in the table is the maximum and minimum levels of parallelism achieved, the dct2 kernel was able to achieve the best performance on occupancy and this was expected as the reduced register method allowed for more threads per block to run in parallel. The theoretical occupancy of the original dct kernel was 33.33% and the theoretical occupancy of the dct2 kernel was 66.67%, doubling the theoretical occupancy and shrinking the register count from 108 to 40 registers per thread. The worst performing kernel was the rle kernel, which makes sense as this kernel is quite inefficient. After completing this project I think implementing a histogram type format would have been better performance wise and better for occupancy for this kernel. With a histogram based binning I would be able to efficiently sort each of the 8x8 blocks laid out linearly in memory.

The average level of parallelism is stated on the right hand side of the table as 45.97% for the original method and 49.96% for the reduced register method. These are not very good and in order to improve them the kernels zigzag, rle, irle, and izigzag need to be reworked.

### Comparison to sequential reference algorithm (performance, computational cost/complexity, work efficiency)

In terms of performance, there were two different outcomes, this depended on the image being processed. When running the Release version on my pc with an rtx 3060 ti I got significantly faster speeds than the T4 on the AWS server, likely some latency with using cloud computing. When processing the 240x240 image I was achieving faster timing on the CPU Codec than on the GPU Codec, but this flipped when performing the codec on the 512x512 image. The CPU codec is block by block compression and then decompression, this is a relatively fast method and there is no data transfer to and from a different device. However in the GPU version, in order to reduce the memory footprint I used a vector of lists to store the compressed version of the image. Since std vectors and lists are not able to be used on the GPU I had to transfer the

RLE result off the device and back to the host to perform the CPU code to complete the compression. This transfer latency to and then back to the device plus the use of the CPU caused the GPU codec to run slower overall for smaller images. However the actual kernel runtimes were significantly faster overall. As the image scales up in size it is my theory that the GPU codec will remain the faster algorithm as the biggest bottleneck is data transfer and the use of vector lists.

The computational complexity of the CPU functions is the same as the computational complexity of the GPU kernels as they are performing the same algorithms only one happens serially and the other in parallel. However in the reduced register method the computations required for the dct and idct kernels are less than the computations for the CPU version. Because the values were precomputed and I used the CPU for this and not a GPU kernel, the calculations were not a part of the GPU Codec timing or computational complexity.

The big difference is the work efficiency. The CPU version is indeed serial, but the resources on the CPU are fully utilized, allowing for quick speeds despite the lack of parallelism. Because the GPU does not fully utilize the hardware, the work efficiency of the GPU is much lower than that of the CPU, but due to the evident parallelism achieved, the GPU version is still quicker at performing each step of the codec.

#### **Cool Extensions of this Project:**

- Support for Non-square images: this is a harder problem, but most of the kernel functions are set up to handle non square images with some modifications to access patterns that would need to be made in order for correct functionality.
- Use of Shared Memory in the DCT and iDCT 2 kernels: The GMEM is a throttle point for the DCT and iDCT 2 kernels as they are memory bound. In theory this would speed up the computations of the two kernels and allow for the roofline to tend back towards compute bound. Because of the complex access pattern needed for copying this particular part of GMEM to SMEM for each warp, I opted for the simple GMEM implementation to show a difference in register count and thread count.
- Launching Child Kernels from a main kernel: I think this principle could be used here as there is extra overhead that is introduced with waiting for the kernels to synchronize in the main function. Maybe implementing the parent child kernel method would decrease some of the required overhead.
- Re-write the RLE kernel to perform histogram binning in an effort to improve performance of the kernel. I believe this would perform better as the kernel would be able to run quicker and more efficiently read from GMEM. The method that is implemented right now utilizes GMEM reads pretty inefficiently.

For purposes of running the project please find the whole VS Project here:

<https://drive.google.com/drive/folders/1R0VBrNvRSe7oeYhqJ1tweypbU5aRiZPw?usp=sharing>

You will need to change the location of reading and writing the filename but that is it! Enjoy