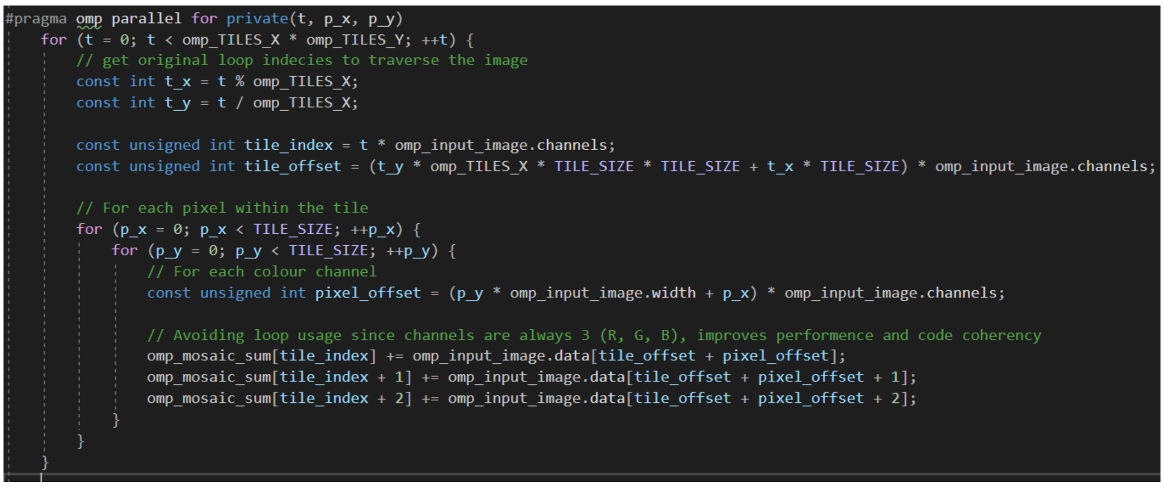
COM6521 21/22 Assignment

ACP21MFM

1 OpenMP

1.1 Stage 1

In stage 1 (Tile Sum), different from the provided algorithm, the double tile loop is fixed into a single loop through index manipulation withing the loop to improve performance and make the code more readable. the original most inner loop is also replaced with assigning values directly to the mosaic sum since the original loop only served changing ch value, which was always 3. OpenMP scoping was kept simple due to the reason that the variables used inside the parallel region were defined in the region, which makes them private to each thread regardless if they were explicitly scoped or not. through experimentation, this had a slight improvement on the performance.



```
#pragma omp parallel for private(t, p_x, p_y)
    for (t = 0; t < omp_TILES_X * omp_TILES_Y; ++t) {
        // get original loop indecies to traverse the image
        const int t_x = t % omp_TILES_X;
        const int t_y = t / omp_TILES_X;

        const unsigned int tile_index = t * omp_input_image.channels;
        const unsigned int tile_offset = (t_y * omp_TILES_X * TILE_SIZE * TILE_SIZE + t_x * TILE_SIZE) * omp_input_image.channels;

        // For each pixel within the tile
        for (p_x = 0; p_x < TILE_SIZE; ++p_x) {
            for (p_y = 0; p_y < TILE_SIZE; ++p_y) {
                // For each colour channel
                const unsigned int pixel_offset = (p_y * omp_input_image.width + p_x) * omp_input_image.channels;

                // Avoiding loop usage since channels are always 3 (R, G, B), improves performence and code coherency
                omp_mosaic_sum[tile_index] += omp_input_image.data[tile_offset + pixel_offset];
                omp_mosaic_sum[tile_index + 1] += omp_input_image.data[tile_offset + pixel_offset + 1];
                omp_mosaic_sum[tile_index + 2] += omp_input_image.data[tile_offset + pixel_offset + 2];
            }
        }
    }
```

Figure 1: OpenMP Stage 1 Code.

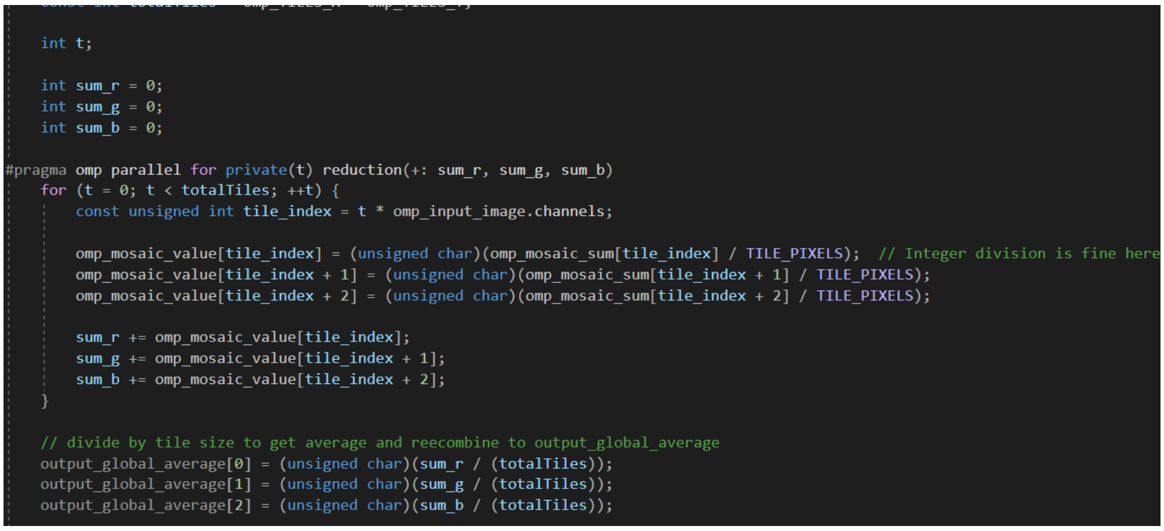
1.2 Stage 2

Stage 2, The average pixel value for each tile is calculated by dividing the tile sums from the previous stage by the square of TILE SIZE (the number of pixels in a tile). Such a task is considered very simple even for a large buffer of pixel values and thus the task of paralleling this stage was perplexing. Many methods were tried in order to improve this stage's time but the original sequential algorithm gave faster times.

The provided solution is given as an example of what might make this stage faster for a harder problem, in order to use OpenMP's reduction method in the summation of the pixels, three variables were defined, each representing a different channel (R, G, B). calculated in the parallel section similarly to the original algorithm, then combined and divided after the parallel section in output global average, with respect to each channel's index. While the solution still provided slower performance, through experimentation, it was the fasted.

The initial speculation that this is happening is because the simple calculation requirements, the parallel approach is disadvantageous to the sequential approach for memory allocation reasons.

In real life scenarios, such a piece of code won't be integrated into parallel code since the sequential approach is faster, which is the whole purpose of parallel programming.



```
    int t;

    int sum_r = 0;
    int sum_g = 0;
    int sum_b = 0;

#pragma omp parallel for private(t) reduction(+: sum_r, sum_g, sum_b)
    for (t = 0; t < totalTiles; ++t) {
        const unsigned int tile_index = t * omp_input_image.channels;

        omp_mosaic_value[tile_index] = (unsigned char)(omp_mosaic_sum[tile_index] / TILE_PIXELS);  // Integer division is fine here
        omp_mosaic_value[tile_index + 1] = (unsigned char)(omp_mosaic_sum[tile_index + 1] / TILE_PIXELS);
        omp_mosaic_value[tile_index + 2] = (unsigned char)(omp_mosaic_sum[tile_index + 2] / TILE_PIXELS);

        sum_r += omp_mosaic_value[tile_index];
        sum_g += omp_mosaic_value[tile_index + 1];
        sum_b += omp_mosaic_value[tile_index + 2];
    }

    // divide by tile size to get average and reecombine to output_global_average
    output_global_average[0] = (unsigned char)(sum_r / (totalTiles));
    output_global_average[1] = (unsigned char)(sum_g / (totalTiles));
    output_global_average[2] = (unsigned char)(sum_b / (totalTiles));
```

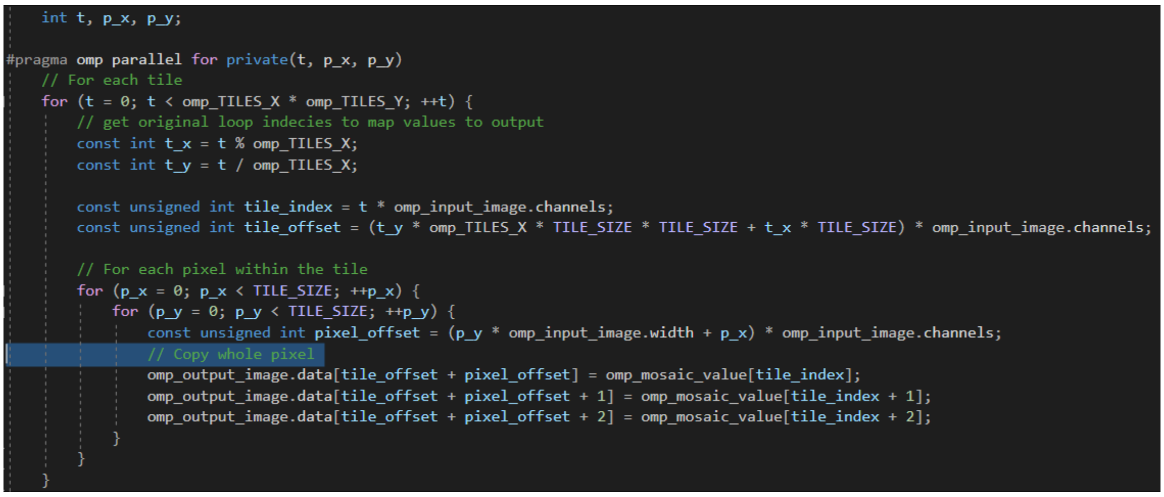
Figure 2: OpenMP Stage 2 Code.

1.3 Stage 3

In stage 3,the final output image is derived by broadcasting the pixel averages from the compact mosaic to a full scale image.

Similar to the approach in Stage 1, the tile loops in this stage are fixed into a single loop through index manipulation, which improves overall performance and makes OpenMP's operations and scoping less complicated.

One more difference from the original algorithm, is removing the usage of memcpy function to assign the pixel values to the output. in C/C++, memcpy is considered to be IO- operation bound is not bound to the CPU so applying parallel programming techniques to it is not applicable or useful, instead, assigning pixel values directly to each distinct channel (indexes [N, N+1, N+2]) is more parallel- friendly and enhanced performance considerably since such an operation is a processing-unit-bound, thus, multi thread operations are able to benefit it.



```
    int t, p_x, p_y;

#pragma omp parallel for private(t, p_x, p_y)
    // For each tile
    for (t = 0; t < omp_TILES_X * omp_TILES_Y; ++t) {
        // get original loop indecies to map values to output
        const int t_x = t % omp_TILES_X;
        const int t_y = t / omp_TILES_X;

        const unsigned int tile_index = t * omp_input_image.channels;
        const unsigned int tile_offset = (t_y * omp_TILES_X * TILE_SIZE * TILE_SIZE + t_x * TILE_SIZE) * omp_input_image.channels;

        // For each pixel within the tile
        for (p_x = 0; p_x < TILE_SIZE; ++p_x) {
            for (p_y = 0; p_y < TILE_SIZE; ++p_y) {
                const unsigned int pixel_offset = (p_y * omp_input_image.width + p_x) * omp_input_image.channels;
                // Copy whole pixel
                omp_output_image.data[tile_offset + pixel_offset] = omp_mosaic_value[tile_index];
                omp_output_image.data[tile_offset + pixel_offset + 1] = omp_mosaic_value[tile_index + 1];
                omp_output_image.data[tile_offset + pixel_offset + 2] = omp_mosaic_value[tile_index + 2];
            }
        }
    }
```

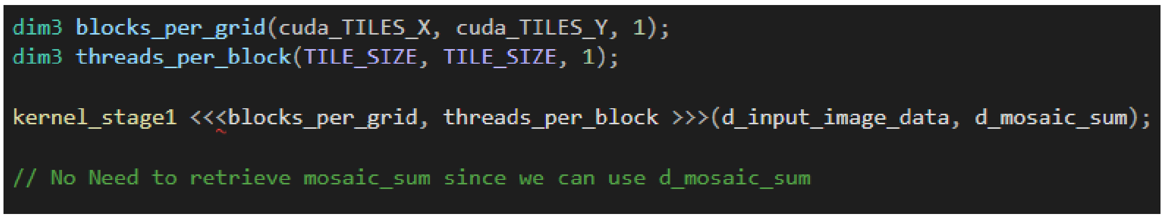
Figure 3: OpenMP Stage 3 Code.

2 CUDA

2.1 Stage 1 Host Code

For CUDA paralleling, the host specifies the kernel parameters, and usually allocates memory for the passed variables. The main required kernel parameters are 1) how many blocks per grid to utilize, blocks_per_grid 2) how many threads per block to utilize, threads_per_block. both parameters' values are chosen, usually, to maximize processor occupancy, thus, increasing performance. in our case block number is chosen according to how many tiles we have in both directions, and thread number is chosen according to tile_size, which makes each pass of an image array occupy a full grid.

After execution of the Kernel is finished, usually, the calculated variable is retrieved to the host via cudaMemCpy, however, since d_mosaic_sum will be used on the device in later stage cudaMemCpy was skipped in this stage, which contributed to performance.

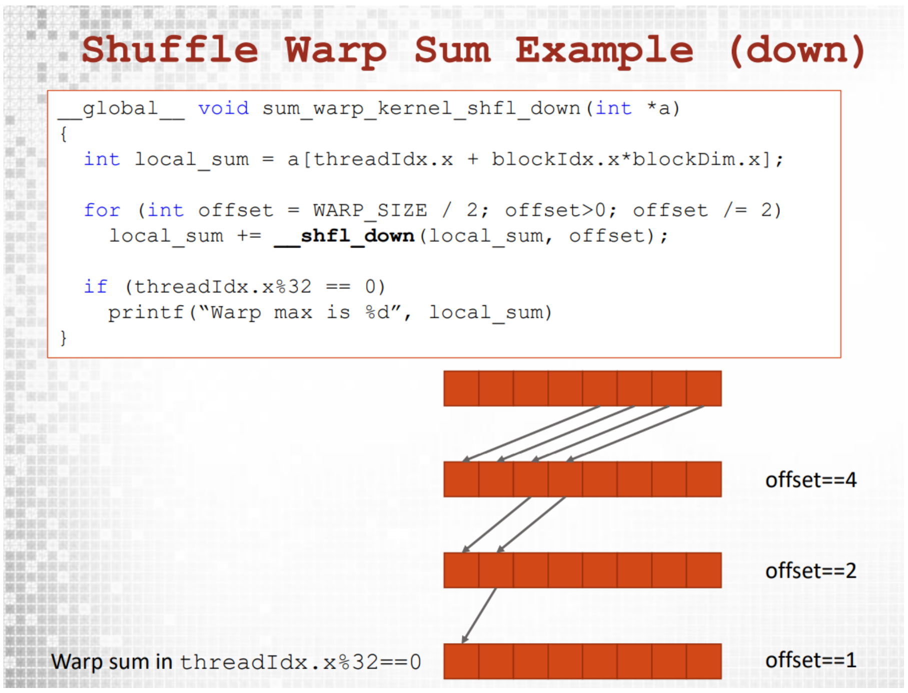


```
dim3 blocks_per_grid(cuda_TILES_X, cuda_TILES_Y, 1);
dim3 threads_per_block(TILE_SIZE, TILE_SIZE, 1);

kernel_stage1 <<<blocks_per_grid, threads_per_block >>>(d_input_image_data, d_mosaic_sum);

// No Need to retrieve mosaic_sum since we can use d_mosaic_sum
```

Figure 4: Stage 1 Host Code.

2.2 Stage 1 Kernel

In consistency with block and threads size chosen by the host, the original algorithm's loop parameters are chosen as each tile starting with the block ID and each pixel chosen according to thread ID. and just like the approach in OpenMP, array usage is avoided through breaking down into three different variables (R, G, B) which eliminates channel offset usage as well. Finally, in order to avoid branch divergence conditions __syncthreads() was used which improved performence by 30% for stage 1.

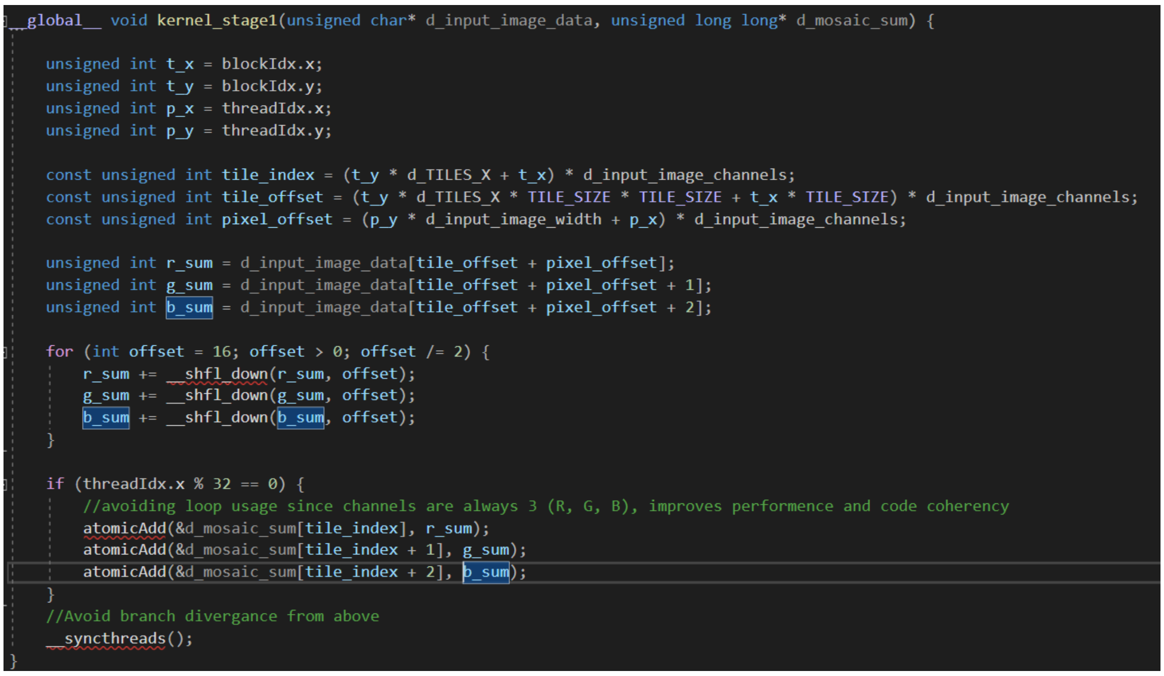


Shuffle Warp Sum Example (down)

```
__global__ void sum_warp_kernel_shfl_down(int *a)
{
  int local_sum = a[threadIdx.x + blockIdx.x*blockDim.x];

  for (int offset = WARP_SIZE / 2; offset>0; offset /= 2)
    local_sum += __shfl_down(local_sum, offset);

  if (threadIdx.x%32 == 0)
    printf("Warp max is %d", local_sum)
}
```

Figure 5: __shfl_down() logic, From Lecture12 COM6521 By Dr Paul Richmond.

For the summation of the pixel values, wrap operation shuffling down was used, which was the best one for the provided algorithm as its application purpose directly goes with the task as hand, moreover, to recombine the values into the mosaic sum array, atomicAdd was used to avoid race conditions.



```
__global__ void kernel_stage1(unsigned char* d_input_image_data, unsigned long long* d_mosaic_sum) {

    unsigned int t_x = blockIdx.x;
    unsigned int t_y = blockIdx.y;
    unsigned int p_x = threadIdx.x;
    unsigned int p_y = threadIdx.y;

    const unsigned int tile_index = (t_y * d_TILES_X + t_x) * d_input_image_channels;
    const unsigned int tile_offset = (t_y * d_TILES_X * TILE_SIZE * TILE_SIZE + t_x * TILE_SIZE) * d_input_image_channels;
    const unsigned int pixel_offset = (p_y * d_input_image_width + p_x) * d_input_image_channels;

    unsigned int r_sum = d_input_image_data[tile_offset + pixel_offset];
    unsigned int g_sum = d_input_image_data[tile_offset + pixel_offset + 1];
    unsigned int b_sum = d_input_image_data[tile_offset + pixel_offset + 2];

    for (int offset = 16; offset > 0; offset /= 2) {
        r_sum += __shfl_down(r_sum, offset);
        g_sum += __shfl_down(g_sum, offset);
        b_sum += __shfl_down(b_sum, offset);
    }

    if (threadIdx.x % 32 == 0) {
        //avoiding loop usage since channels are always 3 (R, G, B), improves performence and code coherency
        atomicAdd(&d_mosaic_sum[tile_index], r_sum);
        atomicAdd(&d_mosaic_sum[tile_index + 1], g_sum);
        atomicAdd(&d_mosaic_sum[tile_index + 2], b_sum);
    }
    //Avoid branch divergance from above
    __syncthreads();
}
```

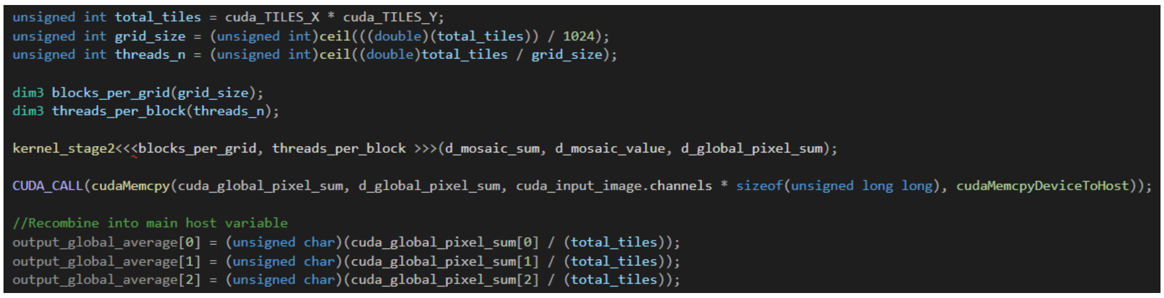
Figure 6: Stage 1 Kernel.

2.3 Stage 2 Host Code

Just like stage 1, in order to maximize hardware occupancy, thread and block size were calculated. in most architectures the maximum thread per block number is 1024, which in our case directly goes with the tile size (ends up with grid size of 1 and thread number of 1024), in such a way, if a bigger tile size is provided the parameters would utilize the kernel in the most efficient way too.However, like OpenMP, CUDA failed to deliver better performance as the task is relatively simple.

The provided solution was found to be the fastest through experimentation when forcing CUDA implementation. As mentioned before this contradicts the purpose of parallel programming and in real life scenarios parallelism would not be applied here.

Afterwards, the global average is retrieved to the host through cudaMemCpy and the global sums are recombined and divided by total tiles to gain the average.

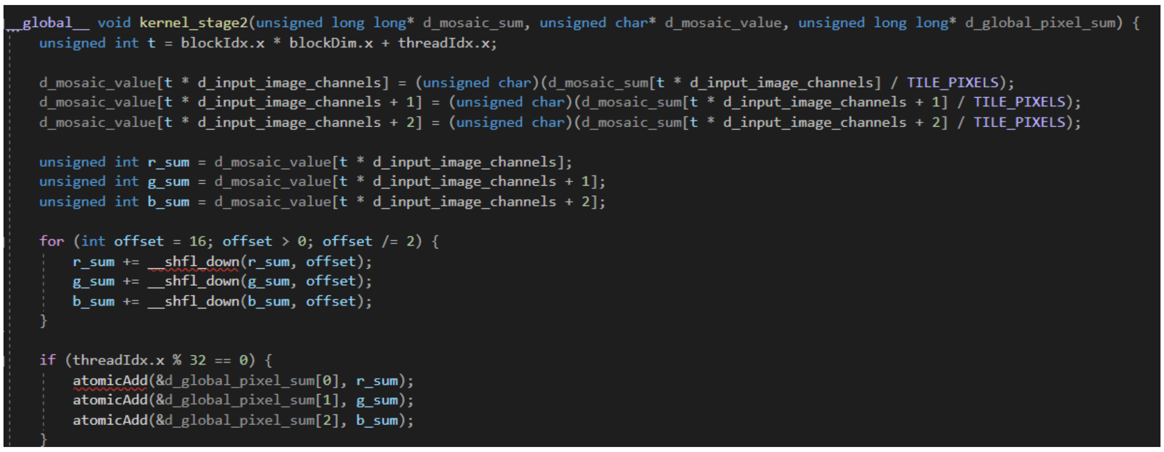


```
unsigned int total_tiles = cuda_TILES_X * cuda_TILES_Y;
unsigned int grid_size = (unsigned int)ceil(((double)(total_tiles)) / 1024);
unsigned int threads_n = (unsigned int)ceil((double)total_tiles / grid_size);

dim3 blocks_per_grid(grid_size);
dim3 threads_per_block(threads_n);

kernel_stage2<<<blocks_per_grid, threads_per_block >>>(d_mosaic_sum, d_mosaic_value, d_global_pixel_sum);

CUDA_CALL(cudaMemcpy(cuda_global_pixel_sum, d_global_pixel_sum, cuda_input_image.channels * sizeof(unsigned long long), cudaMemcpyDeviceToHost));

//Recombine into main host variable
output_global_average[0] = (unsigned char)(cuda_global_pixel_sum[0] / (total_tiles));
output_global_average[1] = (unsigned char)(cuda_global_pixel_sum[1] / (total_tiles));
output_global_average[2] = (unsigned char)(cuda_global_pixel_sum[2] / (total_tiles));
```

Figure 7: Stage 2 Host Code.

2.4 Stage 2 Kernel

Similarly to stage 1, loop usage is avoided through directly indexing the specific channel, (in our application this would always work, but it's assumed that RGB images are used and the indexing is always set up in the same approach).

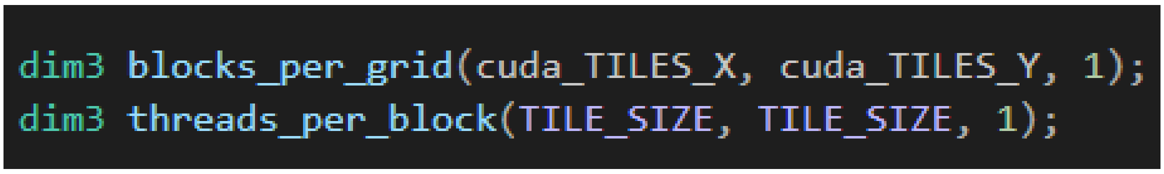


```
__global__ void kernel_stage2(unsigned long long* d_mosaic_sum, unsigned char* d_mosaic_value, unsigned long long* d_global_pixel_sum) {
    unsigned int t = blockIdx.x * blockDim.x + threadIdx.x;

    d_mosaic_value[t * d_input_image_channels] = (unsigned char)(d_mosaic_sum[t * d_input_image_channels] / TILE_PIXELS);
    d_mosaic_value[t * d_input_image_channels + 1] = (unsigned char)(d_mosaic_sum[t * d_input_image_channels + 1] / TILE_PIXELS);
    d_mosaic_value[t * d_input_image_channels + 2] = (unsigned char)(d_mosaic_sum[t * d_input_image_channels + 2] / TILE_PIXELS);

    unsigned int r_sum = d_mosaic_value[t * d_input_image_channels];
    unsigned int g_sum = d_mosaic_value[t * d_input_image_channels + 1];
    unsigned int b_sum = d_mosaic_value[t * d_input_image_channels + 2];

    for (int offset = 16; offset > 0; offset /= 2) {
        r_sum += __shfl_down(r_sum, offset);
        g_sum += __shfl_down(g_sum, offset);
        b_sum += __shfl_down(b_sum, offset);
    }

    if (threadIdx.x % 32 == 0) {
        atomicAdd(&d_global_pixel_sum[0], r_sum);
        atomicAdd(&d_global_pixel_sum[1], g_sum);
        atomicAdd(&d_global_pixel_sum[2], b_sum);
    }
```

Figure 8: Stage 2 Kernel.

2.5 Stage 3 Host Code

Stage 3 host code can be approached like stage 1, as the same number of threads and blocks is the most suitable for the task.

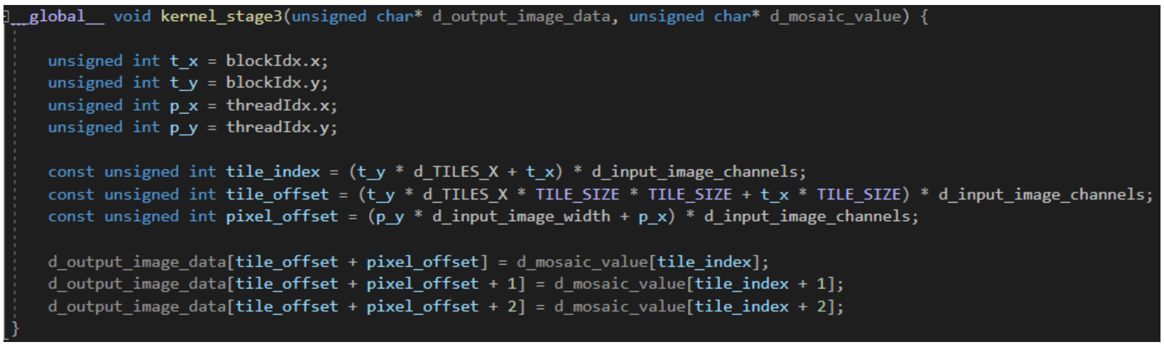


```
dim3 blocks_per_grid(cuda_TILES_X, cuda_TILES_Y, 1);
dim3 threads_per_block(TILE_SIZE, TILE_SIZE, 1);
```

Figure 9: Stage 3 Host Code.

2.6 Stage 3 Kernel

The kernel code is relatively simple, thread and block locations' IDs are utilized to eliminate loop usage and broadcasting is done with the provided tile and pixel offset variables.



```
__global__ void kernel_stage3(unsigned char* d_output_image_data, unsigned char* d_mosaic_value) {

    unsigned int t_x = blockIdx.x;
    unsigned int t_y = blockIdx.y;
    unsigned int p_x = threadIdx.x;
    unsigned int p_y = threadIdx.y;

    const unsigned int tile_index = (t_y * d_TILES_X + t_x) * d_input_image_channels;
    const unsigned int tile_offset = (t_y * d_TILES_X * TILE_SIZE * TILE_SIZE + t_x * TILE_SIZE) * d_input_image_channels;
    const unsigned int pixel_offset = (p_y * d_input_image_width + p_x) * d_input_image_channels;

    d_output_image_data[tile_offset + pixel_offset] = d_mosaic_value[tile_index];
    d_output_image_data[tile_offset + pixel_offset + 1] = d_mosaic_value[tile_index + 1];
    d_output_image_data[tile_offset + pixel_offset + 2] = d_mosaic_value[tile_index + 2];
}
```

Figure 10: Stage 3 Kernel.

3 Results

From the results below, it is apparent that parallel programming techniques are greatly beneficial to increasing speeds of such algorithms, in some cases CUDA, for example, improved performance by more than 100x. Also, it was noted that parallel methods work best with computationally expensive operations, or cpu-bound operations. Using all of the techniques that were learned during the course of COM6521, parallel methods were successfully applied in the assignment, while also, discovering the best case uses for parallel programming.

if time was not a constraint, more can be discovered using profiling and benchmarking methods (Nsight compute, Visual Profiler) to improve the speeds of CUDA and aslo better algorithms could be developed to use within OpenMP boundaries as the provided algorithm was developed thinking in sequential manner.

256x256	CPU	OpenMP	CUDA
Stage 1	0.148ms	0.100ms	0.021ms
Stage 2	0.007ms	0.028ms	0.056ms
Stage 3	0.193ms	0.029ms	0.015ms

Figure 11: Results on 256x256 picture.

1024x1024	CPU	OpenMP	CUDA
Stage 1	2.241ms	0.367ms	0.105ms
Stage 2	0.011ms	0.050ms	0.059ms
Stage 3	3.029ms	0.512ms	0.057ms

Figure 12: Results on 1024x1024 picture.

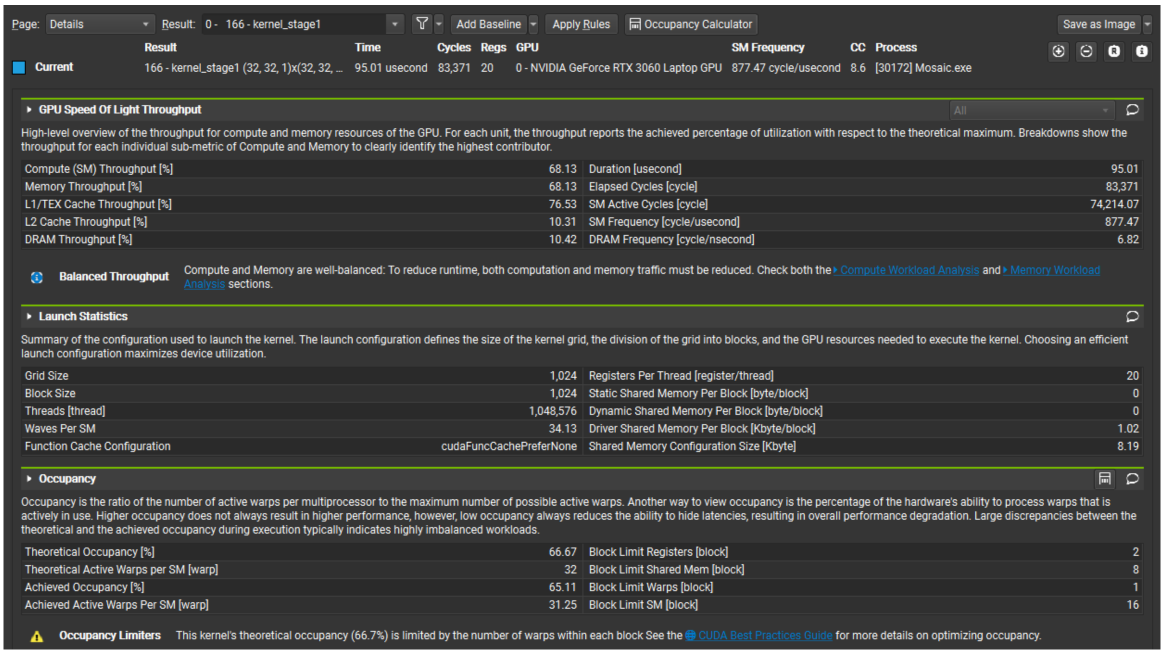
2048x2048	CPU	OpenMP	CUDA
Stage 1	10.212ms	1.238ms	0.393ms
Stage 2	0.021ms	0.036ms	0.052ms
Stage 3	19.458ms	1.870ms	0.203ms

Figure 13: Results on 2048x2048 picture.

4096x4096	CPU	OpenMP	CUDA
Stage 1	43.833ms	4.591ms	1.491ms
Stage 2	0.071ms	0.069ms	0.059ms
Stage 3	101.8ms	9.402ms	0.775ms

Figure 14: Results on 4096x4096 picture.

4 Nsight Compute



	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	166 - kernel_stage1 (32, 32, 1)x(32, 32, ...	95.01 usecond	83,371	20	0 - NVIDIA GeForce RTX 3060 Laptop GPU	877.47 cycle/usecond	8.6	[30172] Mosaic.exe

GPU Speed Of Light Throughput

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor.

Compute (SM) Throughput [%]	68.13	Duration [usecond]	95.01
Memory Throughput [%]	68.13	Elapsed Cycles [cycle]	83,371
L1/TEX Cache Throughput [%]	76.53	SM Active Cycles [cycle]	74,214.07
L2 Cache Throughput [%]	10.31	SM Frequency [cycle/usecond]	877.47
DRAM Throughput [%]	10.42	DRAM Frequency [cycle/nsecond]	6.82

Balanced Throughput Compute and Memory are well-balanced: To reduce runtime, both computation and memory traffic must be reduced. Check both the Compute Workload Analysis and Memory Workload Analysis sections.

Launch Statistics

Summary of the configuration used to launch the kernel. The launch configuration defines the size of the kernel grid, the division of the grid into blocks, and the GPU resources needed to execute the kernel. Choosing an efficient launch configuration maximizes device utilization.

Grid Size	1,024	Registers Per Thread [register/thread]	20
Block Size	1,024	Static Shared Memory Per Block [byte/block]	0
Threads [thread]	1,048,576	Dynamic Shared Memory Per Block [byte/block]	0
Waves Per SM	34.13	Driver Shared Memory Per Block [Kbyte/block]	1.02
Function Cache Configuration	cudaFuncCachePreferNone	Shared Memory Configuration Size [Kbyte]	8.19

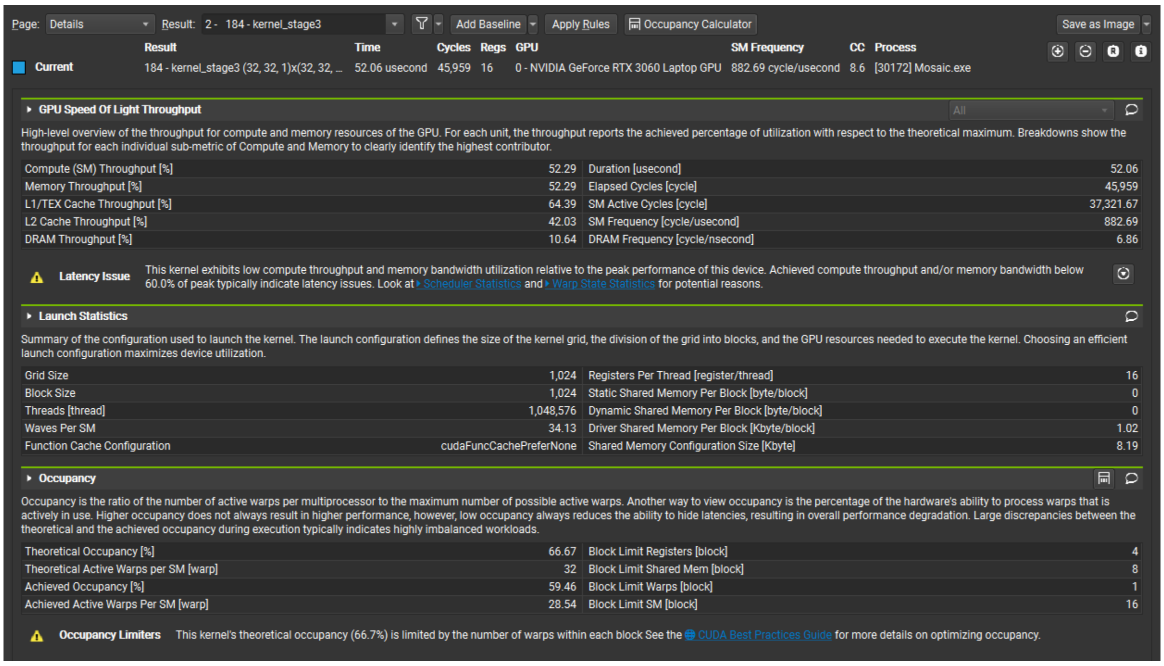
Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	66.67	Block Limit Registers [block]	2
Theoretical Active Warps per SM [warp]	32	Block Limit Shared Mem [block]	8
Achieved Occupancy [%]	65.11	Block Limit Warps [block]	1
Achieved Active Warps Per SM [warp]	31.25	Block Limit SM [block]	16

Occupancy Limiters This kernel's theoretical occupancy (66.7%) is limited by the number of warps within each block See the CUDA Best Practices Guide for more details on optimizing occupancy.

Figure 15: Stage 1 Nsight Analysis.



	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	184 - kernel_stage3 (32, 32, 1)x(32, 32, ...	52.06 usecond	45,959	16	0 - NVIDIA GeForce RTX 3060 Laptop GPU	882.69 cycle/usecond	8.6	[30172] Mosaic.exe

GPU Speed Of Light Throughput

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor.

Compute (SM) Throughput [%]	52.29	Duration [usecond]	52.06
Memory Throughput [%]	52.29	Elapsed Cycles [cycle]	45,959
L1/TEX Cache Throughput [%]	64.39	SM Active Cycles [cycle]	37,321.67
L2 Cache Throughput [%]	42.03	SM Frequency [cycle/usecond]	882.69
DRAM Throughput [%]	10.64	DRAM Frequency [cycle/nsecond]	6.86

Latency Issue This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of this device. Achieved compute throughput and/or memory bandwidth below 60.0% of peak typically indicate latency issues. Look at Scheduler Statistics and Warp State Statistics for potential reasons.

Launch Statistics

Summary of the configuration used to launch the kernel. The launch configuration defines the size of the kernel grid, the division of the grid into blocks, and the GPU resources needed to execute the kernel. Choosing an efficient launch configuration maximizes device utilization.

Grid Size	1,024	Registers Per Thread [register/thread]	16
Block Size	1,024	Static Shared Memory Per Block [byte/block]	0
Threads [thread]	1,048,576	Dynamic Shared Memory Per Block [byte/block]	0
Waves Per SM	34.13	Driver Shared Memory Per Block [Kbyte/block]	1.02
Function Cache Configuration	cudaFuncCachePreferNone	Shared Memory Configuration Size [Kbyte]	8.19

Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	66.67	Block Limit Registers [block]	4
Theoretical Active Warps per SM [warp]	32	Block Limit Shared Mem [block]	8
Achieved Occupancy [%]	59.46	Block Limit Warps [block]	1
Achieved Active Warps Per SM [warp]	28.54	Block Limit SM [block]	16

Occupancy Limiters This kernel's theoretical occupancy (66.7%) is limited by the number of warps within each block See the CUDA Best Practices Guide for more details on optimizing occupancy.

Figure 16: Stage 3 Nsight Analysis.