GPU Assignment: Part 2

180127955

May 13, 2019

1 Introduction

In this report, the optimisation of a simple pixellation program on the GPU using CUDA is detailed. Previous results from the optimisation using the CPU and OPENMP are given and are used as a reference point for improvement. The obtained results show that while CUDA is powerful for parallelisation tasks, there are overheads that need to be accounted for.

2 CPU and OPENMP

As documented in the first assignment, the best times achieved for the CPU and OPENMP implementations are given below in Table 1.

Execution time (ms) with respect to c							
c	2	8	16	32	64	128	256
CPU	20.00	11.00	9.00	9.00	13.00	12.00	10.00
OPENMP	5.49	3.54	3.26	3.05	3.62	3.72	3.59

Table 1: CPU and OPENMP execution times

3 CUDA

3.1 CUDAv1

3.1.1 CUDAv1 - Architecture

The architecture of the OPENMP solution was to allocate threads to individual mosaic cells so that a single thread would process all the pixels in the mosaic cell. However this solution was limited by the number of threads available on the CPU. With the GPU we can overcome this limitation by utilising the greater number of threads that are available.

As such, for an image of size N x M, there are $\lceil \frac{M}{c} \rceil$, $\lceil \frac{N}{c} \rceil$ mosaics along the width and height of the image respectively. A grid of dimension ($\lceil \frac{M}{c} \rceil$,$\lceil \frac{N}{c} \rceil$,1) can be launched with each block having a single thread. This specification allocates a single thread to process the rows and columns of a mosaic cell.

Data and memory specifications

1. Data storage: as a first approach, the data is stored in an array of structures where each element is a structure with three `unsigned char`, one for each colour value. Each pixel is therefore stored in exactly one element of the array. For ease, the built in CUDA vector type, `uchar4` is used to create the array. `uchar4` is used instead of `uchar3` to ensure that each element in the array is 4-byte aligned, to ensure that pixel data loaded from global memory belongs to a single cache line (L2 or L1) and is not overlapping.
2. Data storage - memory: since each thread operates on a single mosaic cell, each pixel needs to be loaded only once from global memory and is not reused. Furthermore, since there is only one thread per block, memory types such as texture or shared memory are not useful.
3. Variables - memory: with only one thread per block, variables are loaded from global memory directly to each threads registers.

3.1.2 CUDAv1 - First results

Implementing the above architecture on the GPU we obtain the results given in Table **??** below. The given solution is less performant than the CPU variant that uses OPENMP. Analysis with the NVIDIA Visual Profiler immediately reveals problematic metrics. These are noted below in Table 3.

Execution time (ms) with respect to c							
c	2	8	16	32	64	128	256
CPU	20.00	11.00	9.00	9.00	13.00	12.00	10.00
OPENMP	5.49	3.54	3.26	3.05	3.62	3.72	3.59
CUDAv1	**14.26**	**4.86**	**4.73**	**5.33**	**7.10**	**7.19**	**11.63**

Table 2: CUDAv1 results

Factor	Metric (%)
Compute Utilisation	below 60
Memory Utilisation	below 60
Warp Execution Efficiency	3.1
Not-Predicated-Off Warp Execution Efficiency	3.1
Occupancy Acheived/ Theoretical	49.4 / 50

Table 3: CUDAv1 Visual Profiler results

The observed kernel is bound both by instruction and memory latency with the main performance litmiter being the block size. By using only 1 thread per block, the number of warps that can be executed on a multiprocessor is not optimal. Analysis shows that increasing the number of threads per block may increase the number of warps executed on each multiprocessor. Furthermore, since the number of threads in each block is not a multiple of 32, the warp execution efficiency is low as can be seen in Table **??**.

3.1.3 CUDAv1 - summary

Since the main performance limitation of `CUDAv1` is essentially the number of threads per block, the architecture of the mosaic filter on the GPU has to change to utilise a greater number of threads per block. Even with changes suggested by the Visual Profiler, the performance of `CUDAv1` does not look promising for large values of `c` due to its inherent functionality. As `c` increases, less threads are launched and each thread processes more pixels thus reducing the occupancy obtained on the GPU and increasing the latency arising from computation. Below we consider other implementations of the mosaic filter on the GPU and perhaps optimisation of `CUDAv1` will be made later if needed.

3.2 CUDAv2

3.2.1 CUDAv2 - Architecture

Instead of launching a single thread per mosaic cell, `c` threads can be launched with each thread processing a column of a mosaic cell. That is, the sum of red, green and blue pixel values in a mosaic cell would be accumulated per column and the resulting `c` red, green and blue values would be aggregated to give the total for red, green and blue in the mosaic cell.

For an image of size N x M this is implemented on the GPU by launching a grid of dimension $(\lceil \frac{M}{c} \rceil, \lceil \frac{N}{c} \rceil, 1)$. Each block in the grid has dimension (`c`,1,1), that is, `c` threads per block, one thread per column of a mosaic cell.

On the GPU being used there is a hardware limitation of 1024 threads per block so for `c` larger than 1024, a given mosaic cell is split according to Figure 1 below (the illustration is created for `C` = 2048). Since each block operates on a portion of the mosaic cell, the results from the blocks contributing to a mosaic cell have to be accumulated before the average values in the mosaic cell are calculated. As shown in Figure 1 below, the z axis is used to group blocks contributing to a mosaic cell. This functionality for `c` larger than 1024 is implemented by launching a grid with dimension $(\lceil \frac{M}{c} \rceil, \lceil \frac{N}{c} \rceil, (\frac{c}{1024})^2)$. As an example, if `c` = 2048, the mosaic cell is split into 4 sections, with each section being operated on by a block with dimension (1024,1,1). To ensure we have the correct number of blocks along the z axis for each mosaic cell, the calculation $(\frac{c}{1024})^2 = (\frac{2048}{1024})^2 = 4$ is used.

Since the implementation is dependant on block level communication, another kernel `CUDAv2_scatter` with a similar behavior to the above is used to write the mosaic level results to the output image. This is done by creating an array in global memory of size equal to the number of mosaic cells. Each index in the array corresponds to a specific mosaic cell, therefore blocks contributing to the same mosaic cell can increment the resulting sum for the mosaic cell atomically.

Data and memory specifications

1. Data storage: same as `CUDAv1`
2. Data storage - memory: within a warp, threads are reading adjacent pixels. As such, there is no 2d spatial locality of memory accesses that can be taken advantage of - making texture memory not useful. In particular, since accesses to global memory will be coalesced, the performance of texture memory will be the same as that simply using the L2/ L1 cache. With no reuse of pixel data and threads within a warp reading different memory locations, shared memory and constant memory are not useful. For the writing kernel, `CUDAv2_scatter`, shared memory is used to store the pixel that each thread is going to write.
3. Variables - memory: where threads within a block read the same value from global memory, the constant cache is utilised to allow for values to be broadcast within the warp.

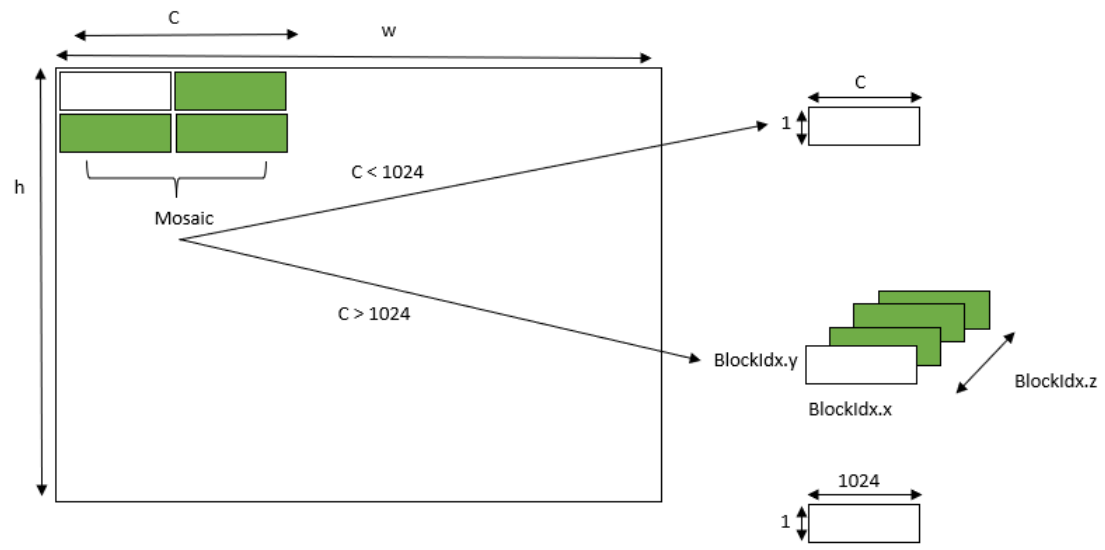

Figure 1: CUDAv2 architecture. w and h represent the width and height respectively. For C > 1024, the above illustration represents the case with C = 2048.

3.2.2 CUDAv2 - First results

Implementing the above architecture on the GPU we obtain the results given in Table **??** below. For `c` larger than 8, the solution is more performant than the CUDAv1 version and OPENMP. An immediate aspect that is to be considered is the behaviour of the implementation for `c` less than 32. As we observed in Section 3.1.2, having block sizes that are not multiples of 32 reduce the warp execution efficiency.

Execution time (ms) with respect to c							
c	2	8	16	32	64	128	256
CPU	20.00	11.00	9.00	9.00	13.00	12.00	10.00
OPENMP	5.49	3.54	3.26	3.05	3.62	3.72	3.59
CUDAv1	14.26	4.86	4.73	5.33	7.10	7.19	11.63
CUDAv2	**22.97**	**4.28**	**2.12**	**1.12**	**0.79**	**0.74**	**0.60**

Table 4: CUDAv2 results

Using the NVIDIA Visual Profiler for the kernel `CUDAv2`, we note some of the problematic metrics for `c` less than 32 (`c` = 16) in Table **??**. Again the observed kernel is bound both by instruction and memory latency with the main performance limiter being the block size. For `c` = 16, only half of the threads (instead of 32) can be used. This is resulting in the warp execution efficiency being 50%. Another factor contributing to the low performance of the kernel is the instruction latency, over 70% of the causes of warp stalls are memory dependant, which could be caused by memory alignment and access patterns

Similarly, Using the NVIDIA Visual Profiler for the kernel `CUDAv2_scatter`, we note some of the problematic metrics:

Factor	Metric (%)
Compute Utilisation	below 60
Memory Utilisation	below 60
Warp Execution Efficiency	50
Not-Predicated-Off Warp Execution Efficiency	48.2
Occupancy Acheived/ Theoretical	49.4 / 50
Instruction latency; memory dependency	above 70

Table 5: CUDAv2 Visual Profiler results

Factor	Metric
Compute Utilisation	below 60%
Memory Utilisation	below 60%
Instruction latency; memory dependency	above 80%
Global Store L2 Transactions/Access	4

Table 6: CUDAv2_scatter Visual Profiler results

This kernel is also bound by both instruction and memory latency with the primary stall issue being the memory dependency as shown in the table above. The memory bandwidth is also not being utilised efficiently due to access patterns. The issues identifed above will be addressed in the sections that follow.

3.2.3 CUDAv2.1 warp divergence and warp execution efficiency - c less than 32

Firstly, it should be noted that for images where the number of pixels is less than 32, intra-warp divergence is unavoidable. However for larger images, typically having a width of atleast 64, an architectural change can be made for `c` less than 32 to ensure that the block size is always a multiple of 32.

For `c` less than 32, the block size proposed in Section 3.2.1 can be changed to (64, 1, 1), that is, requiring that the minimal number of threads in a block is 64. Since $c < 32$, previous blocks with dimension (`c`, 1, 1) can be combined together in one block. For example, if `c` = 16, a single block size of 64 threads can be composed of $\frac{64}{16} = 4$ potential mosaic cells depending on the size of the image. Since blocks are combined together (along the x axis), the grid will change from the previous $(\lceil\frac{M}{c}\rceil,\lceil\frac{N}{c}\rceil,1)$ (see Section 3.2.1) to $(\lceil\frac{\lceil\frac{M}{c}\rceil}{\frac{64}{c}}\rceil,\lceil\frac{N}{c}\rceil,1)$ where $\frac{64}{c}$ gives the necessary "squashing" factor.

Making this change to CUDAv2, we obtain the results given in Table 7 below (**CUDAv2.1**):

	Execution time (ms) with respect to c						
c	2	8	16	32	64	128	256
CPU	20.00	11.00	9.00	9.00	13.00	12.00	10.00
OPENMP	5.49	3.54	3.26	3.05	3.62	3.72	3.59
CUDAv1	14.26	4.86	4.73	5.33	7.10	7.19	11.63
CUDAv2	22.97	4.28	2.12	1.12	0.79	0.74	0.60
CUDAv2.1	**12.12**	**3.24**	**1.78**	**1.08**	**0.79**	**0.74**	**0.60**

Table 7: CUDAv2.1 results

These results are a consequence of utilising all 32 threads that would be available in a warp. Previously for `c`<32, only `c` threads would be utilized. This effect is seen most significantly for `c` = 2.

3.2.4 CUDAv2.3 Global memory access patterns

As identified in 3.2.2, the kernel `CUDAv2_scatter` appears to be reduced in efficiency because of memory access patterns. The kernel is requiring 4 transactions per access from global memory when the ideal is 1 transaction per access. The problematic code section is given below:

```
1 // if thread is within bounds, write the average mosaic data to the output
2 if (x < width) {
3   for (int i = y; i < upper_lim; i++) {
4     location = i * width + x;  // pixel location
5     d_output[location].x = (unsigned char)mosaic_reduce[location2].x;
6     d_output[location].y = (unsigned char)mosaic_reduce[location2].y;
7     d_output[location].z = (unsigned char)mosaic_reduce[location2].z;
8   }
9 }
```

Listing 1: CUDAv2_scatter 4 transactions per access

Here `d_output` and `mosaic_reduce` are arrays of type `uchar4` and `float4` respectively stored in global memory. Each time the value `location` is accessed from the variable `d_output`, threads in a warp have strided access (4 bytes) to obtain their required data from global memory. Line 5 corresponds to obtaining the `r` pixel value, line 6 the `g` pixel value and finally line 7 giving the `b` pixel value. Rather than loading a single 32 byte cache line from `L2` cache, 4 32 byte cache lines are loaded. This can be overcome by changing `d_output` from an array of structures to a structure of arrays as shown in Listings 2 below:

```
// if thread is within bounds, write the average mosaic data to the output
if (x < width) {
    // read average values once from global memory
    float4 pixel2 = mosaic_reduce[location2];
    uchar4 pixel1;
    pixel1.x = (unsigned char) pixel2.x;
    pixel1.y = (unsigned char) pixel2.y;
    pixel1.z = (unsigned char) pixel2.z;

    for (int i = y; i < upper_lim; i++) {
      location = i * width + x;
      // write output to structure of arrays
      d_output.r[location] = pixel1.x;
      d_output.g[location] = pixel1.y;
      d_output.b[location] = pixel1.z;
    }
  }
```

Listing 2: CUDAv2_scatter coalesced writing

Making this change gives the result in Table 8 below. Since memory bandwidth has been improved, the performance of the kernel has also improved.

3.2.5 CUDAv2.4 Block level reduction

One optimisation technique not identified by the Visual Profiler is the potential of the program to use block level reduction. Due to the architecture presented in 3.2.1, the resulting thread level

Execution time (ms) with respect to c							
c	2	8	16	32	64	128	256
CPU	20.00	11.00	9.00	9.00	13.00	12.00	10.00
OPENMP	5.49	3.54	3.26	3.05	3.62	3.72	3.59
CUDAv1	14.26	4.86	4.73	5.33	7.10	7.19	11.63
CUDAv2	22.97	4.28	2.12	1.12	0.79	0.74	0.60
CUDAv2.1	12.12	3.24	1.78	1.08	0.79	0.74	0.60
CUDAv2.2	**11.83**	**3.06**	**1.62**	**0.92**	**0.61**	**0.54**	**0.46**

Table 8: CUDAv2.2 results

results need to be aggregated to obtain the block level results for a given mosaic cell. Furthermore, for `c` > 1024, multiple blocks would be launched in order to process mosaic sizes greater than 1024. All block level results contributing to a specific mosaic cell therefore need to be aggregated to give the average pixel values. In order to do this, atomic operations are used as shown in Listing 9 below.

```
1         // location2 : index for a specific mosaic cell, eg 4 blocks
2         // contributing to the same mosaic will write to the same index
3 
4     location2 = ceil((width / (float)c)) * blockIdx.y + blockIdx.x;
5     atomicAdd(&(mosaic_reduce[location2].x), avg_pixel.x / (float)size);
6     atomicAdd(&(mosaic_reduce[location2].y), avg_pixel.y / (float)size);
7     atomicAdd(&(mosaic_reduce[location2].z), avg_pixel.z / (float)size);
8     atomicAdd(&(red_average), (avg_pixel.x / img_size));
9     atomicAdd(&(green_average), (avg_pixel.y / img_size));
10    atomicAdd(&(blue_average), (avg_pixel.z / img_size));
```

Listing 3: c threads write to global memory

Since each thread processes a column of a mosaic cell, there will be `c` writes to global memory for lines 5-10. Instead of having `c` writes to global memory, a single write per line can be accomplished by reducing all the values held by `c` threads. The required change to the program is given below in Listings 10:

```
    // each thread stores its value in shared memory
    block_data[threadIdx.x] = avg_pixel;
  __syncthreads();

  int stride_val = blockDim.x/2;
  float4 pixel_temp;
  float4 pixel_receive;
  for (int stride = stride_val; stride > 0; stride >>= 1) {
    if (threadIdx.x < stride) {
        // load data at offset index
      pixel_temp = block_data[threadIdx.x + stride];
      // load data at threadIdx.x
      pixel_receive = block_data[threadIdx.x];
      // aggregate values
      pixel_receive.x += pixel_temp.x;
      pixel_receive.y += pixel_temp.y;
      pixel_receive.z += pixel_temp.z;
      // store new values at threadIdx.x
      block_data[threadIdx.x] = pixel_receive;
    }
    __syncthreads();
  }
```

Listing 4: block level reduction of c threads

By reducing all `c` values to a single value, only one thread needs to write to global memory. Making this change gives the result in Table 9 below. Note that the performance for `c` = 2 has reduced. This is because block level reduction cannot be applied when `c` < 32 for the architecture given in Section 3.2.3. For 2 < `c` < 32, though we have an improved performance, warp execution efficiency is still low and this impacts the result for $c = 2$. The performance can be made equivalent by setting the algorithm to use the method in Section 3.2.3 for `c` = 2.

Execution time (ms) with respect to c							
c	2	8	16	32	64	128	256
CPU	20.00	11.00	9.00	9.00	13.00	12.00	10.00
OPENMP	5.49	3.54	3.26	3.05	3.62	3.72	3.59
CUDAv1	14.26	4.86	4.73	5.33	7.10	7.19	11.63
CUDAv2	22.97	4.28	2.12	1.12	0.79	0.74	0.60
CUDAv2.1	12.12	3.24	1.78	1.08	0.79	0.74	0.60
CUDAv2.2	11.83	3.06	1.62	0.92	0.61	0.54	0.46
CUDAv2.3	**16.21**	**1.38**	**0.57**	**0.36**	**0.36**	**0.36**	**0.36**

Table 9: CUDAv2.3 results

3.2.6 CUDAv2.4 warp level reduction for c less than 32

As noted in Section 3.2.5, for `c` < 32, due to the improved architecture that deals with having block sizes of less than 32 (see Section 3.2.3), block level reduction is no longer possible. By sacrificing occupancy for the potential of reduction, execution times are improved for `c` > 2. One way of utilising the architecture in 3.2.3 as well as a reduction mechanism is by using warp level reduction. Since mosaic cells are grouped together in block sizes of 64, warp level reduction can be applied by setting the width of the shuffle to be `c` and the offset within such a width as $\frac{c}{2}$. Instead of the code shown in Listing 4 (Section 3.2.5), the required code for `c` < 32 is given below in Listing 5.

```
/* warp level reduction : since block is arranged as e.g c c c c,
can reduce within a mosaic by setting the offset to be c/2
*/
  for (int stride = c / 2; stride > 0; stride >>= 1) {
    avg_pixel.x += __shfl_down(avg_pixel.x, stride, c);
    avg_pixel.y += __shfl_down(avg_pixel.y, stride, c);
    avg_pixel.z += __shfl_down(avg_pixel.z, stride, c);
  }
```

Listing 5: warp level reduction of c threads

Implementing the above change, the corresponding results are shown in Table 11 below. There is an observed improvement in the execution time for `c` < 32 due to a reduced number of writes to global memory. Previously, each thread in a block of size 64 would write to global memory, however due to the architecture in Section 3.2.3, and the warp level reduction implemented, only $\frac{64}{c}$ writes to global memory are required.

3.2.7 CUDAv2.3 Array of Structures vs Structure of arrays

Though section 3.2.4 suggested that using an array of structures may result in poor access patterns when accessing global memory or shared memory data, `CUDAv2.3` has no erroneous access patterns to global memory (See Listing 6 below).

Execution time (ms) with respect to c							
c	2	8	16	32	64	128	256
CPU	20.00	11.00	9.00	9.00	13.00	12.00	10.00
OPENMP	5.49	3.54	3.26	3.05	3.62	3.72	3.59
CUDAv1	14.26	4.86	4.73	5.33	7.10	7.19	11.63
CUDAv2	22.97	4.28	2.12	1.12	0.79	0.74	0.60
CUDAv2.1	12.12	3.24	1.78	1.08	0.79	0.74	0.60
CUDAv2.2	11.83	3.06	1.62	0.92	0.61	0.54	0.46
CUDAv2.3	16.21	1.38	0.57	0.36	0.36	0.36	0.36
CUDAv2.4	**6.12**	**0.57**	**0.38**	**0.36**	**0.36**	**0.36**	**0.36**

Table 10: CUDAv2.4 results

This is because CUDA vector types are optimised for ”vector loads”. In Listing 6 below, `d_image` is an array of type `uchar4`. For a normal array of structures, each thread requesting the pixel at position `location` would require 4 requests (for the adjacent values of `r`, `g`, `b` and an unused padding byte). However CUDA ”vector loads” allow a single 4 byte load per thread, and because threads are reading adjacent pixels, access is still coalesced. This behavior does not depend on the memory type (e.g. global memory and shared memory) (See https://devblogs.nvidia.com/cuda-pro-tip-increase-performance-with-vectorized-memory-access/ for more information).

```
if (x < width) {
  for (int i = y; i < upper_lim; i++) {
    location = i * width + x;
    pixel = d_image[location];
    avg_pixel.x += pixel.x;
    avg_pixel.y += pixel.y;
    avg_pixel.z += pixel.z;

  }
}
```

Listing 6: CUDA vector loading

3.2.8 CUDAv2.5 Atomicadd(float) to Atomicadd(int)

In order to be able to calculate the global pixel average, `CUDAv2` utilises atomic operations in global memory as block level communication is required for the image average to be calculated. One observation is that using Atomicadd(int) is more efficient than using AtomicAdd(float) for low values of `c`. As `c` increases, the use of reduction reduces the number of writes to global memory, however this is not the case for low values of `c`. One possible reason for this increase in efficiency when calculating the global average using Atomicadd(int) is that the kernel is already utilising the floating point pipe. Changing to the integer pipe may be reducing the amount of stress on the floating point pipe. The code with the corresponding problem is given below in Listings 7.

```
if (threadIdx.x == 0) {
/*
location2 : index of mosaic cell
mosaic_reduce : array of mosaic cell average values
red_average/blue_average/green_average : image average variables
*/
  location2 = ceil((width / (float)c)) * blockIdx.y + blockIdx.x;
  atomicAdd(&(mosaic_reduce[location2].x), pixel_receive.x / (float)size);
  atomicAdd(&(mosaic_reduce[location2].y), pixel_receive.y / (float)size);
```

```
    atomicAdd(&(mosaic_reduce[location2].z), pixel_receive.z / (float)size);
    atomicAdd(&(red_average), (pixel_receive.x / img_size));
    atomicAdd(&(green_average), (pixel_receive.y / img_size));
    atomicAdd(&(blue_average), (pixel_receive.z / img_size));
  }
```

Listing 7: Global average atomics with variables of type float4

Along with changing the type of the variable being written to, we also avoid dividing each time the average needs to be updated (See Listings 8).

```
  if (threadIdx.x == 0) {
  /*
  location2 : index of mosaic cell
  mosaic_reduce : array of mosaic cell average values
  red_average/blue_average/green_average : image average variables
  */
    location2 = ceil((width / (float)c)) * blockIdx.y + blockIdx.x;
    atomicAdd(&(mosaic_reduce[location2].x), pixel_receive.x / (float)size);
    atomicAdd(&(mosaic_reduce[location2].y), pixel_receive.y / (float)size);
    atomicAdd(&(mosaic_reduce[location2].z), pixel_receive.z / (float)size);
    atomicAdd(&(red_average), (pixel_receive.x));
    atomicAdd(&(green_average), (pixel_receive.y));
    atomicAdd(&(blue_average), (pixel_receive.z));
  }
}
```

Listing 8: Global average atomics with variables of type uin4

Making the above changes shows an improved performance for low values of `c` as can be seen in Table 11 below.

Execution time (ms) with respect to c							
c	2	8	16	32	64	128	256
CPU	20.00	11.00	9.00	9.00	13.00	12.00	10.00
OPENMP	5.49	3.54	3.26	3.05	3.62	3.72	3.59
CUDAv1	14.26	4.86	4.73	5.33	7.10	7.19	11.63
CUDAv2	22.97	4.28	2.12	1.12	0.79	0.74	0.60
CUDAv2.1	12.12	3.24	1.78	1.08	0.79	0.74	0.60
CUDAv2.2	11.83	3.06	1.62	0.92	0.61	0.54	0.46
CUDAv2.3	16.21	1.38	0.57	0.36	0.36	0.36	0.36
CUDAv2.4	6.12	0.57	0.38	0.36	0.36	0.36	0.36
CUDAv2.5	**3.58**	**0.47**	**0.37**	**0.36**	**0.36**	**0.36**	**0.36**

Table 11: CUDAv2.4 results

3.2.9 CUDAv2 - summary

This implementation appears to be the most promising in terms of performance achieved using the GPU for the pixellation program. For `c` > 2 a speedup of x8-10 relative to the OPENMP solution is achieved. Analsyis with the Visual Profiler reveals that the main kernel, `CUDAv2.5` is still memory bound (utilisation greater than 70%), with compute utilisation taking a value below 30%. Given the nature of the problem, it is expected for the compute utilisation and memory utilisation to be in this state. Aggregating pixel values over a mosaic cell only uses simple addition, which the kernel can perform with ease. However to do so requires that data is available to the required CUDA cores, making this factor the main bottleneck.

3.3 CUDAv3

3.3.1 CUDAv3 - Architecture

In Section 3.2.1 the architecture of the pixellation program was based on using a single thread per column of a mosaic cell. This implementation was motivated by the low occupancy arising from using a single thread per mosaic cell as shown in Section 3.1.2. Instead of using `c` threads per mosaic cell, another approach is to use `c` x `c` threads per mosaic cell with the hope that using a higher number of threads per mosaic will increase performance.

Each thread in a block will load a single value from device memory to shared memory and the resulting values in shared memory will be reduced to single values for `r`, `g` and `b`.

As documented in Section 3.2.1, for an image of size N x M, a grid of dimension ($\lceil\frac{M}{c}\rceil$,$\lceil\frac{N}{c}\rceil$,1) is launched. However in this case each block in the grid has dimension (`c`,`c`,1), that is, `c` x `c` threads per block. For `c` < 8, each block will contain at most 16 threads, causing the warp execution efficiency to be at most 50%. In a similar way to section 3.2.3, blocks for `c` < 8 can be combined along the x axis so that the minimal block size is 64. Given that the height of a mosaic cell is `c`, and the corresponding number of pixels in the cell is `c` x `c`, the required factor to increase the number of threads in the x axis by is $\frac{64}{c*c}$ given that the minimal number of threads in a block is 64. Similarly, the number of blocks along the x axis are "squashed by this factor". As a result, the grid size changes from ($\lceil\frac{M}{c}\rceil$,$\lceil\frac{N}{c}\rceil$,1) to ($\lceil\frac{\lceil\frac{M}{c}\rceil}{\frac{64}{c}}\rceil$,$\lceil\frac{N}{c}\rceil$,1) after blocks are combined.

Since `c` x `c` threads are launched for a mosaic cell of size `c` x `c`, a hardware limitation arises when `c` is larger than 32. A solution to this problem is similar to that introduced in Section 3.2.1 to compensate for `c` larger than 32. A given mosaic for `c` > 32 is split according to Figure 2 below. Each block operates on a portion of the mosaic cell and the results from the blocks contributing to a mosaic cell are accumulated before the average values in the mosaic cell are calculated. The z axis as can be seen below is used to group blocks contributing to a mosaic cell. For `c` larger than 32, a grid with dimension ($\lceil\frac{M}{c}\rceil$,$\lceil\frac{N}{c}\rceil$, $(\frac{c}{32})^2$) is launched. For example, if `c` $= 64$, the mosaic cell is split into 4 sections, each with dimension (32,32,1). To ensure there is a correct number of blocks along the z axis for each mosaic cell, the calculation $(\frac{c}{32})^2 = \frac{64*64}{32*32} = 4$ is used.

Again, since the implementation is dependant on block level communication, another kernel `CUDAv3_scatter` with a similar behavior to the above is used to write the mosaic level results to the output image.

Data and memory specifications

1. Data storage: same as `CUDAv1`
2. Data storage - memory: Same as `CUDAv2`. However, to accomplish the block level reduction mentioned earlier, shared memory is used to store block level pixel values so that pixel data is accessible to all threads within a block. For the writing kernel, `CUDAv3_scatter`, shared memory is used to store the pixel(s) that each thread is going to write.
3. Variables - memory: where threads within a block read the same value from global memory, the constant cache is utilised to allow for values to be broadcast within the warp.

3.3.2 CUDAv3 - First results

Implementing the above solution we obtain the results shown in Table **??** below. We see a decrease in performance in comparison to the `CUDAv2.3` implementation. An initial suspicion of why this implementation is less performant is that all threads in a block (`c` x `c`) are required to have loaded their values from global memory before any computation can be performed. However, the `CUDAv2`

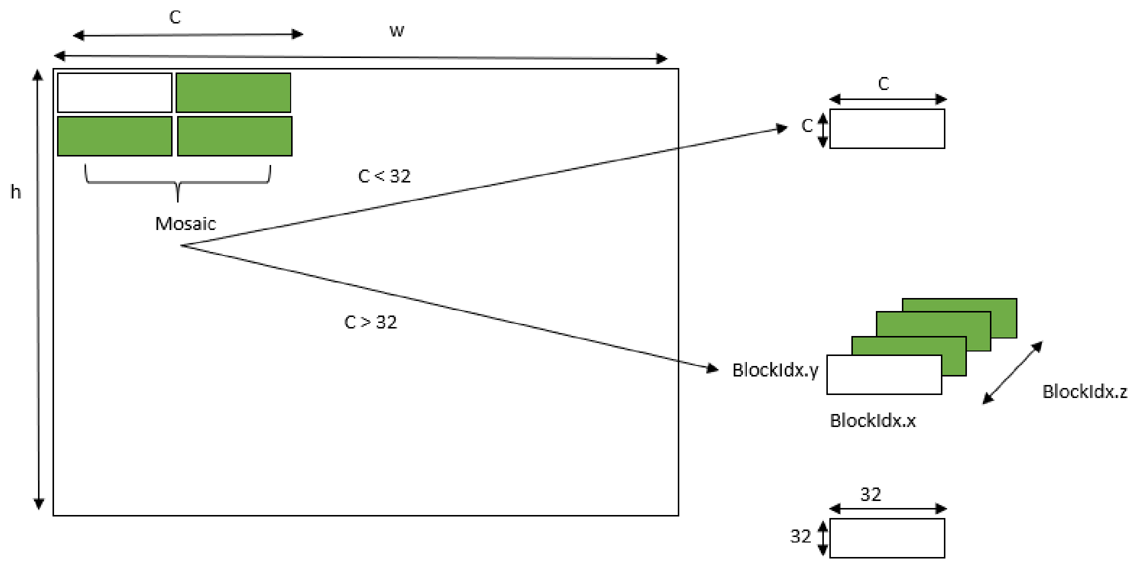

Figure 2: CUDAv3 architecture. w and h represent the width and height respectively. For C > 32, the above illustration represents the case with C = 64.

variant loads values and computes per warp, so the latency caused by memory transactions can be masked by computation.

Execution time (ms) with respect to c							
c	2	8	16	32	64	128	256
CPU	20.00	11.00	9.00	9.00	13.00	12.00	10.00
OPENMP	5.49	3.54	3.26	3.05	3.62	3.72	3.59
CUDAv1	14.26	4.86	4.73	5.33	7.10	7.19	11.63
CUDAv2	22.97	4.28	2.12	1.12	0.79	0.74	0.60
CUDAv2.1	12.12	3.24	1.78	1.08	0.79	0.74	0.60
CUDAv2.2	11.83	3.06	1.62	0.92	0.61	0.54	0.46
CUDAv2.3	16.21	1.38	0.57	0.36	0.36	0.36	0.36
CUDAv2.4	6.12	0.57	0.38	0.36	0.36	0.36	0.36
CUDAv2.5	3.58	0.47	0.37	0.36	0.36	0.36	0.36
CUDAv3	**12.98**	**1.62**	**1.31**	**1.84**	**1.86**	**1.89**	**1.91**

Table 12: CUDAv3 results

Analysis with the Visual Profiler reveals the problematic metrics (see Table 13 below). The analysis shows that the kernel `CUDAv3` is bound by both memory and instruction latency. The key performance limiters are the two main stall issues: Memory dependency and Synchronisation. As suspected, the kernel is spending a proportion of time inactive (see Instruction execution count) due to waiting for the required memory from global memory to be loaded to shared memory. The use of synchronisation (for block level reduction) exacerbates the latency as threads within a warp are waiting for all threads in the block to be at the same step in the reduction process. Solutions to these issues are investigated in the following sections.

Factor	Metric (%)
Compute Utilisation	below 60
Memory Utilisation	below 60
Not-Predicated-Off Warp Execution Efficiency	65.7
Instruction latency; Memory dependency	30
Instruction latency; Synchronization	40
Instruction execution count; Inactive	40

Table 13: CUDAv3 Visual Profiler results

3.3.3 CUDAv3.1 - c x c to 16 x 16

Firstly, the inactivity measured by the Visual Profiler can be attributed to the fact that for `c` > 32, block sizes of 32 x 32 are used. For the reduction algorithm to work, all 1024 threads are required to have access to their pixel values from global memory, and each iteration of the reduction will cause all threads within a warp to wait for the block. One alternative is to partition mosaics of sizes larger than 32 into partitions of 16 x 16 rather than 32 x 32, so that a reduced number of threads are required to be synchronised. Making this change to the program, the results obtained are given in 14 below. An improvement is observed for `c` > 32. Partitions of size less than 16 x 16 were tried however this resulted in a less performant program.

	Execution time (ms) with respect to c						
c	2	8	16	32	64	128	256
CPU	20.00	11.00	9.00	9.00	13.00	12.00	10.00
OPENMP	5.49	3.54	3.26	3.05	3.62	3.72	3.59
CUDAv1	14.26	4.86	4.73	5.33	7.10	7.19	11.63
CUDAv2	22.97	4.28	2.12	1.12	0.79	0.74	0.60
CUDAv2.1	12.12	3.24	1.78	1.08	0.79	0.74	0.60
CUDAv2.2	11.83	3.06	1.62	0.92	0.61	0.54	0.46
CUDAv2.3	16.21	1.38	0.57	0.36	0.36	0.36	0.36
CUDAv2.4	6.12	0.57	0.38	0.36	0.36	0.36	0.36
CUDAv2.5	3.58	0.47	0.37	0.36	0.36	0.36	0.36
CUDAv3	12.98	1.62	1.31	1.84	1.86	1.89	1.91
CUDAv3.1	**12.98**	**1.62**	**1.32**	**1.41**	**1.44**	**1.42**	**1.43**

Table 14: CUDAv31 results

3.3.4 CUDAv3.2 - Synchronization

As suggested in Sections 3.3.3 and 3.3.2, the cost of performing reduction on large block sizes is an increased latency. As can be seen in Listing 9 below, a `__syncthreads()` is required after each iteration. An alternative reduction approach that does not incur these penalties is warp level reduction. The corresponding changes to the kernel are given in Listing 10 below.

```
// wait for all threads in the block to finish
__syncthreads();

for (stride = stride_val; stride > 0; stride >>= 1)
{
    // thread aggregates values only if its index is less than stride
    if (location2 < stride)
    {
      pixel1 = block_data[location2];
      pixel2 = block_data[location2 + stride];
      pixel1.x += pixel2.x;
      pixel1.y += pixel2.y;
      pixel1.z += pixel2.z;
      block_data[location2] = pixel1;
    }
    // wait for all threads in the block to finish

    __syncthreads();

  }
```

Listing 9: CUDAv3.1 block level reduction

```
// threads in a warp perform reduction
  for (int stride = 16; stride > 0; stride >>= 1)
  {
    pixel.x += __shfl_down(pixel.x, stride);
    pixel.y += __shfl_down(pixel.y, stride);
    pixel.z += __shfl_down(pixel.z, stride);
  }
```

Listing 10: CUDAv3.2 warp level reduction

Making these changes gives no improvement on the result obtained with implementation `CUDAv3.1`. A reason for this is that while there is some stalling introduced by line 18 in Listing 5, warp level reduction also requires values in a block to be reduced. In particular, as can be seen in Listing 11, rather than having 1 write to global memory, a single thread from each warp will write to global memory, counteracting the reduced latency from the removal of `__syncthreads()`.

```
// one thread in each warp writes to global memory
  if (threadIdx.x % 32 == 0) {
    atomicAdd(&(mosaic_reduce[cells_width * blockIdx.y + blockIdx.x].x), (pixel.x / (float)size));
    atomicAdd(&(mosaic_reduce[cells_width * blockIdx.y + blockIdx.x].y), (pixel.y / (float)size));
    atomicAdd(&(mosaic_reduce[cells_width * blockIdx.y + blockIdx.x].z), (pixel.z / (float)size));
    atomicAdd(&(red_average), (pixel.x / (float)img_size));
    atomicAdd(&(green_average), (pixel.y / (float)img_size));
    atomicAdd(&(blue_average), (pixel.z / (float)img_size));
  }
```

Listing 11: CUDAv3.2 warp level reduction - increased writes to global memory

3.3.5 CUDAv3 - summary

Given the performance achieved by the `CUDAv2` implementation, this implementation was expected to be the best way to utilise the resources available on the GPU. However the requirement of having each thread load a single thread only to perform a single computation results in an observable latency. The analysis with the Visual Profiler in section 3.3.2 showed an early warning of high inactivity of warps on the GPU. Perhaps with an increased computational demand per thread in a block the incurred latency of loading data from global memory can be overlooked.

4 CUDA - Summary

As can be seen from Table 15 below, `CUDAv2.5` is the most performant implementation of the pixellation program on the GPU. We have a noted x8-10 speedup relative to the OPENMP solution for `c` > 2, however the implementation does not perform as well for `c` $= 2$. Since `CUDAv1` is the extension of the OPENMP solution to the GPU, it was expected to perform atleast as well. Notably, the utilisation of the GPU as shown in Section 3.1.2 is not good, with the main problem being block sizes not being multiples of 32. Below, techniques applied to `CUDAv2` are applied to `CUDAv1` to attempt to improve performance.

CUDAv1.1 - warp divergence and warp execution efficiency

For `CUDAv1` a single mosaic is processed by a block with a single thread. To ensure block sizes are multiples of 32, threads within blocks can be combined to ensure the block size is a multiple of 32. Given an image of size N x M a grid of dimension $(\lceil \frac{M}{c} \rceil, \lceil \frac{N}{c} \rceil, 1)$ can be launched. However, by requiring that the smallest block size is 64, the number of blocks can be reduced along the x axis by a factor of 64. That is, a grid of dimension $(\lceil \frac{\lceil \frac{M}{c} \rceil}{64} \rceil, \lceil \frac{N}{c} \rceil, 1)$ can be launched instead, with each block having 64 threads.

Since threads within a block are now processing adjacent mosaic cells texture memory 1D fetches can be used to take advantage of the 2d spatial locality of adjacent mosaic cells. The optimisation of using Atomicadd(int) as documented in 3.2.8 is also applied.

Implementing the above with `CUDAv1`, the results obtained are showin in Table 15. There is a significant improvent for `c` $= 2$ however the performance starts to decline after `c` > 16 (`CUDAv1.1` is less performant than `CUDAv1`). Fortunately, the implementations `CUDAv2.5` and `CUDAv1.1` can be conditioned upon to ensure the performance is always optimal. For `c` $= 2$, `CUDAv1.1` can be used, and for `c` > 2, `CUDAv2.5` can be used.

	Execution time (ms) with respect to c						
c	2	8	16	32	64	128	256
CPU	20.00	11.00	9.00	9.00	13.00	12.00	10.00
OPENMP	5.49	3.54	3.26	3.05	3.62	3.72	3.59
CUDAv1	14.26	4.86	4.73	5.33	7.10	7.19	11.63
CUDAv1.1	**0.42**	**2.13**	**4.20**	**8.18**	**8.55**	**11.31**	**15.84**
CUDAv2	22.97	4.28	2.12	1.12	0.79	0.74	0.60
CUDAv2.1	12.12	3.24	1.78	1.08	0.79	0.74	0.60
CUDAv2.2	11.83	3.06	1.62	0.92	0.61	0.54	0.46
CUDAv2.3	16.21	1.38	0.57	0.36	0.36	0.36	0.36
CUDAv2.4	6.12	0.57	0.38	0.36	0.36	0.36	0.36
CUDAv2.5	3.58	0.47	0.37	0.36	0.36	0.36	0.36
CUDAv3	12.98	1.62	1.31	1.84	1.86	1.89	1.91
CUDAv3.1	12.98	1.62	1.32	1.41	1.44	1.42	1.43

Table 15: CUDAv11 results

4.1 CUDA - Large image sizes

The most performant implementation (using `CUDAv2.5` and `CUDAv1.1`) was tested on an image of dimension 10000x15000. The corresponding results for this test are given below in Table 16.

Execution time (ms) with respect to c							
c	2	8	16	32	64	128	256
CPU	897.00	561.00	560.00	587.00	634.00	639.00	578.00
OPENMP	226.82	109.35	107.28	125.13	137.00	168.67	136.64
CUDA	**14.38**	**14.99**	**14.66**	**19.57**	**22.21**	**22.07**	**22.33**

Table 16: CUDAv11 results

As can be seen from the table, the CUDA implementation outperforms both the CPU and OPENMP implementations for large images by atleast a factor of 5-10. However, one identified problem with the CUDA implementation for large images is an observed integer overflow problem when calculating the average of the entire image.

This problem was fixed by altering the program to compute the sum of the averaged mosaic pixel values using thrust (thrust reduce) and returning the image average by dividing by its size. The corresponding code required to do this is given below in Listing 12

```
// start timing
  cudaEventRecord(start, 0);
  r = thrust::reduce(h_output->r, h_output->r + width * height, (unsigned long long)
    0);
  g = thrust::reduce(h_output->g, h_output->g + width * height, (unsigned long long)
    0);
  b = thrust::reduce(h_output->b, h_output->b + width * height, (unsigned long long)
    0);
  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);
// end timing
  cudaEventElapsedTime(&ms2, start, stop);
```

Listing 12: CUDAv3.2 warp level reduction - increased writes to global memory

4.2 CUDA - movement of data

Though the performance of the pixellation program on the GPU is now x10 faster than the OPENMP program (using the Dog2048x2048.ppm image), the amount of time taken to port the data to and from the GPU should also be noted to evaluate the entire performance of the GPU solution. Using the Visual Profiler, the most efficient CUDA implementation (on the Dog2048x2048.ppm image) requires 4.72ms to move data to the GPU and 1.90ms to move data back to the host. This gives a total time of 6.62ms, greater than the time taken for the OPENMP solution to execute, for the values of `c` chosen without the addition of the execution of the program on the GPU.

Given that the most efficient implementation on the GPU is memory bound, this result is expected. The major bottleneck in optimising the program to be more performant than both the CPU and OPENMP versions is the latency of having to move data from global memory. On both the CPU and OPENMP versions, data is already available in RAM, so reading is very quick. The lack of complexity in computation means that the CPU can process an image quickly. For a problem with a larger demand of compute, it would be expected that the overhead of porting data to and from the GPU would be masked by the amount of time taken to compute, making the GPU an obvious option when constructing a program that takes advantage of parallelism.

5 Result

Many different implementations have been considered in order to arrive at efficient solutions for the CPU, OpenMP and CUDA problems. For the CUDA version, we have arrived at a solution that efficiently processes an image for pixellation but this efficiency is masked by the amount of time required to move the data to and from the GPU. The realisation is that while the GPU is powerful for tasks that can be parallelised, other, less powerful but easier to implement tools can achieve a similar result as the GPU depending on the scale of the problem.