

**COM6521 Assignment**

**Parallel Computing with Graphical Processing Units**

**OpenMP: Stage 1**

**Description:**

In the first stage, the cpu\_mosaic\_sum is set to 0 using the function memset().Then all the variables which will be used inside the parallel region has been assigned to 0. #pragma omp parallel has been added above all the for loop and the num\_threads(omp\_get\_max\_threads()) is set. The default has been set to default(none) and added private(t\_x) to the first variable in the for loop. The shared variables are shared(cpu\_TILES\_X, cpu\_TILES\_Y, cpu\_input\_image, cpu\_mosaic\_sum) . Finally, the #pragma omp for is declared with nowait schedule(dynamic,1) which has an chunk size of 1.

**Justification:**

1. The num\_threads(omp\_get\_max\_threads()) is used in the parallel region, because the OpenMP will auto allocate the maximum amount of threads that are available in the CPU. In this way the number of threads which will run the code will be optimal and the CPU will not be forced to create thread which is beyond its capacity.
2. Sharing of each variable in the stage 1 is explicitly specified to control the data being shared inside. The variables are declared outside because the nested for loop has to access them.
3. The first for loop has the variable t\_x which has been set to private as private(t\_x) because the value of first nested loop should not be shared, otherwise the nested loop will have iteration issues in OpenMP. By making t\_x as private will make each thread have its own copy.
4. The shared variables are cpu\_TILES\_X, cpu\_TILES\_Y, cpu\_input\_image, cpu\_mosaic\_sum. The reason of sharing these variables are, the cpu\_TILES\_X, cpu\_TILES\_Y, cpu\_input\_image will have constant value throughout the for loop. So the value of the variables are not going to change, it can be set in shared without any issue. The cpu\_mosaic\_sum is shared because, the data has to be incremented by all the threads, so the value of cpu\_mosaic\_sum has to be shared, otherwise the threads cannot access it and add its own value to it.
5. The nowait is being used once the for loop is declared. The main reason is to avoid the implied barrier that is present at the end of work-sharing (sections, single, workshare) and target constructs. This is to avoid race condition. Once a thread has finished executing its work, it does not wait until the rest of the threads have finished its calculation, it just moves on to the next task, so that the idle time is saved.
6. The schedule(dynamic,1) is used because it works on first come, first serve basis. Once a thread has been assigned an iteration, it works on it, and when it finishes the work, it will work on the next iteration which is not executed yet. Dynamic scheduling is better when the iterations may take very different amounts of time. The chunk size is 1 in our case because I wanted the program to be more dynamic. Increasing the chunk size makes the scheduling more static, and decreasing it makes it more dynamic.

|  |  |  |
| --- | --- | --- |
| Problem size | CPU Reference Timing (ms) | Open MP Stage 1 Timing (ms) |
| 256x256 px | 0.205 | 0.215 |
| 1024x1024 px | 2.714 | 0.922 |
| 2048x2048 px | 11.673 | 3.138 |
| 4096x4096 px | 71.837 | 14.161 |

By looking at the benchmarks, the main problem which will limit the performance of the OpenMP is the allocation of memory to each thread before execution. Basically, the OpenMP will have to do this operation before the logic is executed, so for the 256 image the OpenMP was 10ms slow, because of the memory allocation. But the CPU executed it fast because, it does not do any memory allocation before it goes through the logic. The memory allocation in CPU is in real time. So, the CPU was able to execute it faster than OpenMP. But in the 1024 image, OpenMP picked up speed, because once the memory has been allocated, the next step is to execute the logic using multiple threads. So, the moment multiple threads starts to execute the logic, it was able to do it much faster than CPU which executes the same logic using single thread. The same goes for 2048 and 4096 image, where in 4096 image the difference is very high. So, when the amount of calculation is high, multithreading works much better than single thread.

Profiling has been done and found the memory allocation problem which I have mentioned in the above paragraph. The OpenMP is able to process it much faster than CPU but due to the initial work that has to be done, it is slower.

I have tried all possible combinations in OpenMP and can say that this is the highest optimized value. Which means the full power of OpenMP has been utilized here.

Graphical user interface

Description automatically generated

**OpenMP: Stage 2**

**Description:**

In the second stage, a local variable is created to hold the value of rgb, which will be used in the later part of the execution. #pragma omp parallel has been added above all the for loop and the num\_threads(omp\_get\_max\_threads()) is set. The value of the first iteration is set to 0, and added firstprivate(t) to the first variable in the for loop. The shared variables are shared(cpu\_TILES\_X, cpu\_TILES\_Y) . Finally, the #pragma omp for is declared with nowait schedule(dynamic,2) which has a chunk size of 2.

**Justification:**

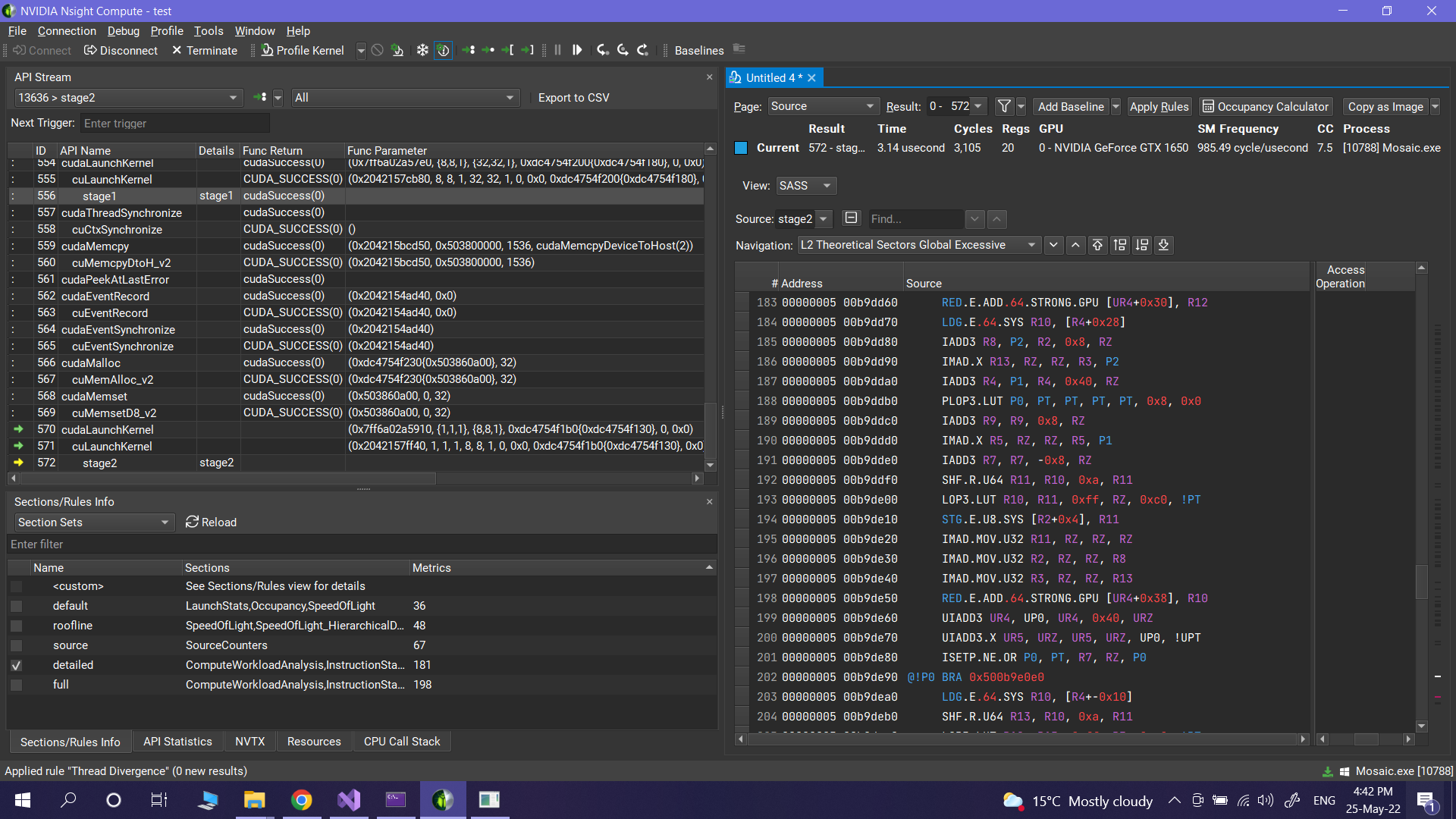
1. The num\_threads(omp\_get\_max\_threads()) is used in the parallel region, because the OpenMP will auto allocate the maximum amount of threads that are available in the CPU. In this way the number of threads which will run the code will be optimal and the CPU will not be forced to create thread which is beyond its capacity.
2. Sharing of each variable in the stage 1 is explicitly specified to control the data being shared inside. The variables are declared outside because the nested for loop has to access them.
3. The nowait is being used once the for loop is declared. The main reason is to avoid the implied barrier that is present at the end of worksharing (sections, single, workshare) and target constructs. This is to avoid race condition. Once a thread has finished executing its work, it does not wait until the rest of the threads have finished its calculation, it just moves on to the next task, so that the idle time is saved.
4. The shared variables are cpu\_TILES\_X, cpu\_TILES\_Y. The reason of sharing these variables are, the cpu\_TILES\_X, cpu\_TILES\_Y will have constant value throughout the for loop. So, the value of the variables are not going to change, it can be set in shared without any issue.
5. The nowait is being used once the for loop is declared. The main reason is to avoid the implied barrier that is present at the end of work-sharing (sections, single, workshare) and target constructs. This is to avoid race condition. Once a thread has finished executing its work, it does not wait until the rest of the threads have finished its calculation, it just moves on to the next task, so that the idle time is saved.
6. The schedule(dynamic,2) is used because it works on first come, first serve basis. Once a thread has been assigned an iteration, it works on it, and when it finishes the work, it will work on the next iteration which is not executed yet. Dynamic scheduling is better when the iterations may take very different amounts of time. The chunk size is 2 in our case because I wanted the program to be decently dynamic. Increasing the chunk size makes the scheduling more static, and decreasing it makes it more dynamic. As this is a very small logic, high dynamic logic is not required, so the chunk size is kept to 2 to make it a little static and decently dynamic.

|  |  |  |
| --- | --- | --- |
| Problem size | CPU Reference Timing (ms) | Open MP Stage 2 Timing (ms) |
| 256x256 px | 0.027 | 0.075 |
| 1024x1024 px | 0.073 | 0.079 |
| 2048x2048 px | 0.082 | 0.178 |
| 4096x4096 px | 0.132 | 0.641 |

By looking at the benchmarks, the main problem which will limit the performance of the OpenMP is the allocation of memory to each thread before execution. Basically, the OpenMP will have to do this operation before the logic is executed, so for the 256 image the OpenMP was 48ms slow, because of the memory allocation. But the CPU executed it fast because, it does not do any memory allocation before it goes through the logic. The memory allocation in CPU is in real time. So, the CPU was able to execute it faster than OpenMP. And the most important thing here is, as this is a very small operation the CPU always executes this set of code faster than OpenMP. The OpenMP will never be able to execute this chunk of code faster than CPU as it lags in the memory allocation part. As the image size goes up, for like 2048 image, the amount of memory that has to be allocated by the OpenMP is much larger (which takes more time), so it is always slower than CPU. OpenMP works better only when there are a lot of calculations to be done, and not very suitable for smaller calculations. So technically stage 2 could never be faster than CPU.

Profiling has been done and found the memory allocation problem which I have mentioned in the above paragraph. The OpenMP is able to process it much faster than CPU but due to the initial work that has to be done, it is slower.

I have tried all possible combinations in OpenMP and can say that this is the highest optimized value. Which means the full power of OpenMP has been utilized here.



**OpenMP: Stage 3:**

**Description:**

In the third stage, all the variables which will be used inside the parallel region has been assigned to 0. #pragma omp parallel has been added above all the for loop and the num\_threads(omp\_get\_max\_threads()) is set. The value of the first iteration is set to 0, and added firstprivate(t\_x,t\_y,p\_x,p\_y,pixel\_offset) to the first variable in the for loop. The shared variables are cpu\_TILES\_X and cpu\_TILES\_Y which is declared by shared(cpu\_TILES\_X, cpu\_TILES\_Y) . Finally, the #pragma omp for is declared with nowait schedule(dynamic,1) which has a chunk size of 1. Once all the threads has finished its task, it makes the memcpy() function to copy all the data

Justification:

1. The num\_threads(omp\_get\_max\_threads()) is used in the parallel region, because the OpenMP will auto allocate the maximum amount of threads that are available in the CPU. In this way the number of threads which will run the code will be optimal and the CPU will not be forced to create thread which is beyond its capacity.
2. Sharing of each variable in the stage 3 is explicitly specified to control the data being shared inside. The variables are declared outside because the nested for loop has to access them.
3. The first for loop has the variable t\_x,t\_y,p\_x,p\_y,pixel\_offset which has been set to private as firstprivate(t\_x,t\_y,p\_x,p\_y,pixel\_offset) because the value of first nested loop should not be shared, otherwise the nested loop will have iteration issues in OpenMP. By making t\_x as private will make each thread have its own copy.
4. The shared variables are cpu\_TILES\_X, cpu\_TILES\_Y. The reason of sharing these variables are, the cpu\_TILES\_X, cpu\_TILES\_Y will have constant value throughout the for loop. So, the value of the variables are not going to change, it can be set in shared without any issue.
5. The nowait is being used once the for loop is declared. The main reason is to avoid the implied barrier that is present at the end of work-sharing (sections, single, workshare) and target constructs. This is to avoid race condition. Once a thread has finished executing its work, it does not wait until the rest of the threads have finished its calculation, it just moves on to the next task, so that the idle time is saved.
6. The schedule(dynamic,1) is used because it works on first come, first serve basis. Once a thread has been assigned an iteration, it works on it, and when it finishes the work, it will work on the next iteration which is not executed yet. Dynamic scheduling is better when the iterations may take very different amounts of time. The chunk size is 1 in our case because I wanted the program to be more dynamic. Increasing the chunk size makes the scheduling more static, and decreasing it makes it more dynamic.

|  |  |  |
| --- | --- | --- |
| Problem size | CPU Reference Timing (ms) | Open MP Stage 3 Timing (ms) |
| 256x256 px | 0.229 | 0.153 |
| 1024x1024 px | 3.413 | 1.267 |
| 2048x2048 px | 28.761 | 10.595 |
| 4096x4096 px | 173.686 | 48.665 |

By looking at the benchmarks, unlike other stages, the allocation of memory takes time in OpenMP, but once that’s done, the logic is being run by multiple threads at the same time so, even in the 256 image, the OpenMP is able to perform better than the CPU. The stage 3 has a heavy task, which the CPU struggles to do it using 1 thread. But by the use of multiple threads all the logic is being performed in parallel and for the 1024 image the OpenMP is able to perform 3 times faster than the CPU. As the amount of memory to be transferred to the threads by OpenMP is much smaller than the task to be performed by the CPU, OpenMP is always faster than the CPU in stage 3. The difference is very high for the 4096 image, where the image is processed by harnessing the power of multiple threads running in parallel.

The profiler has been run for this stage, and the place where OpenMP performs at its best is in the processing section, by using multiple threads. As the amount of calculation gets higher, the performance of OpenMP can be seen when comparing it with the CPU.

I have tried all possible combinations in OpenMP and can say that this is the highest optimized value. Which means the full power of OpenMP has been utilized here.

**CUDA: Stage 1**

**Description:**

**Host Description**

In the first stage of CUDA, I have specified the blocks per grid to be the cuda\_TILES\_X in x axis and cuda\_TILES\_Y in y axis which is in this form dim3 blocks\_per\_grid(cuda\_TILES\_X, cuda\_TILES\_Y) and the threads per block to be TILE\_SIZE in both x and y axis which is in the form dim3 threads\_per\_block(TILE\_SIZE, TILE\_SIZE). The size is set to a specific value which is cuda\_TILES\_X \* cuda\_TILES\_Y \* 3 \* sizeof(unsigned long long). The cpu\_input\_image\_data has been allocated in the global address and is accessed in the host area and is allocated a memory of size mentioned above. The kernel call is being performed which holds the blocks\_per\_grid and threads\_per\_block value as the number of threads to be passed into the kernel area. Along with it these values cuda\_TILES\_X, cuda\_TILES\_Y, cuda\_input\_image.width, cuda\_input\_image.channels, d\_input\_image\_data, d\_mosaic\_sum are passed as parameters too.

A cudaThreadSynchronize() after launching the kernel. And the processed data by the GPU is got back to the CPU of the size mentioned above using this cudaMemcpy(cpu\_mosaic\_sum, d\_mosaic\_sum, size, cudaMemcpyDeviceToHost)

**Kernel Description**

In the kernel the parameters are received and sent into execution. The parameters are unsigned int cuda\_TILES\_X, unsigned int cuda\_TILES\_Y, int width, int channels, unsigned char\* data, unsigned long long\* d\_mosaic\_sum. New values are assigned in the kernel, and they are int tile\_index, int tile\_offset, int pixel\_offset, unsigned char pixel. The main cuda logic is being executed which updates each variable by its calculated value. Finally all the calculated values are added to d\_mosaic\_sum using a special function atomicAdd().

**Justification:**

**Host Justification**

In host of stage 1, the reason of fixing the size of blocks\_per\_grid into cuda\_TILES\_X \* cuda\_TILES\_Y is because the tile\_index and tile\_offset should be calculated for the multiple of cuda\_TILES\_X and cuda\_TILES\_Y times. And the reason for fixing the size of threads\_per\_block into TILE\_SIZE \* TILE\_SIZE is the pixel\_offset should be calculated for that number of times. The default value of TILE\_SIZE is 32, so 32\*32=1024. As a result, 1024 threads are assigned to each block. And the block count differs for each image as it is depending on the cuda\_TILES\_X and cuda\_TILES\_Y. The size is allocated to a dynamic value which differs for each image and the formula used is, cuda\_TILES\_X \* cuda\_TILES\_Y \* 3 \* sizeof(unsigned long long). The cpu\_input\_image\_data and cpu\_mosaic\_sum is being allocated some memory in the host of the size mentioned above in correspondence to its data type like unsigned char or unsigned long long. As the kernel call happens, the blocks\_per\_grid and threads\_per\_block are provided to create those number of threads in the kernel. Along with it I am sending cuda\_TILES\_X, cuda\_TILES\_Y, cuda\_input\_image.width, cuda\_input\_image.channels, d\_input\_image\_data, d\_mosaic\_sum as parameters. After the process happens in the kernel, the data has to be taken back to the host. That is through cudaMemcpy(), The processed value of cuda’s mosaic sum value is copied to the cpu’s mosaic sum variable, and it is sent to the validation to verify.

**Kernel Justification**

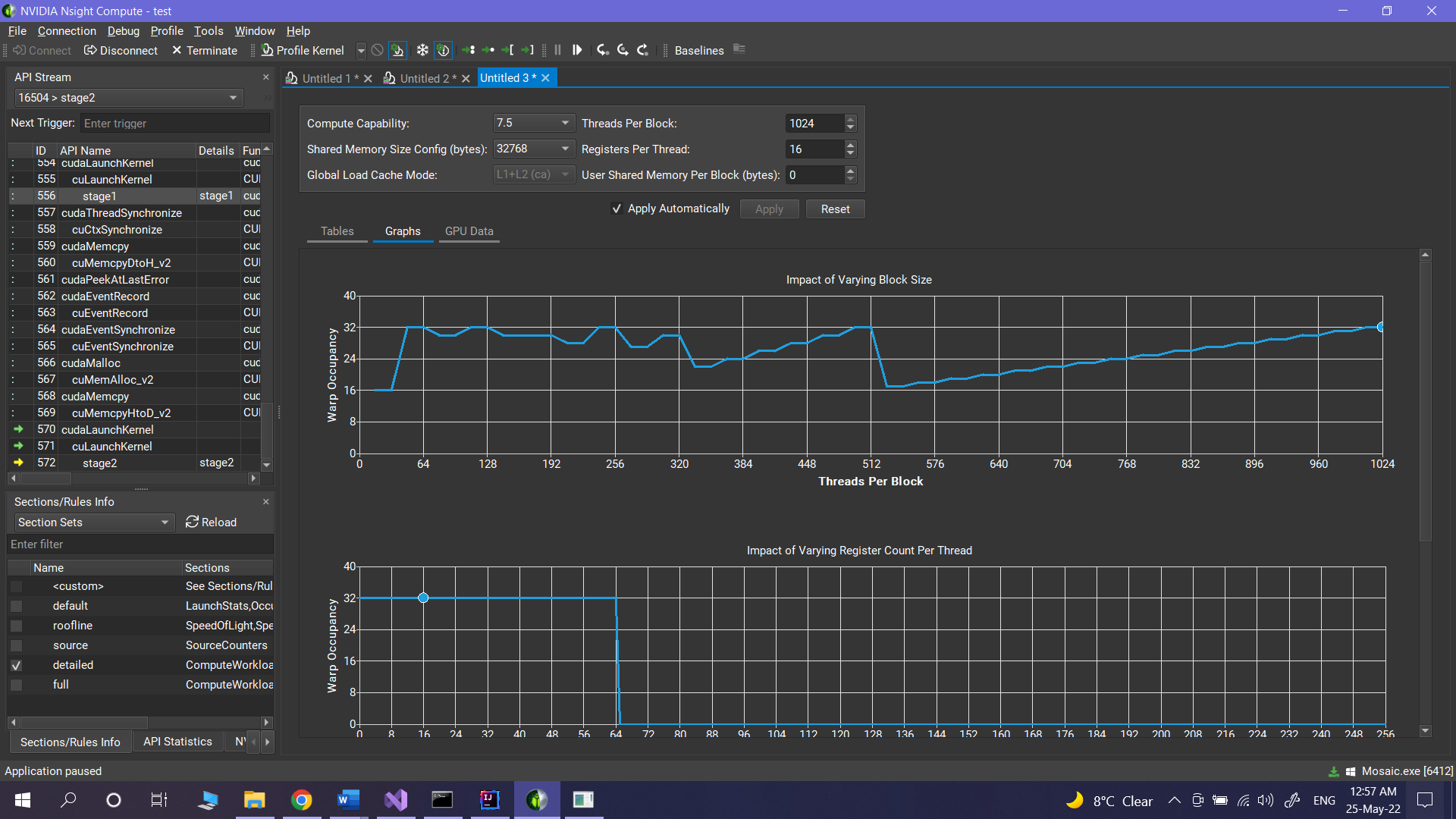
The stage1 kernel gets a set of parameters and passes as arguments to the logic. The parameters that the kernel gets are unsigned int cuda\_TILES\_X, unsigned int cuda\_TILES\_Y, int width, int channels, unsigned char\* data, unsigned long long\* d\_mosaic\_sum. Inside the kernel 4 new variables are defined so that each thread that goes in must have their own copy of int tile\_index, int tile\_offset, int pixel\_offset, unsigned char pixel. There is if condition which will allow the thread to pass in only when the blockIdx.x of the thread and blockIdx.y is less than cuda\_TILES\_X and cuda\_TILES\_Y. This is done to make sure that all the threads in a block have its own copy of tile\_index and tile\_offset. Then the threads has to go through another condition where, if the threadIdx.x and threadIdx.y are less than TILE\_SIZE it has to calculate the pixel\_offset and each thread has to store a copy of all these and calculate the pixel. To avoid a race condition, I have added atomicAdd().

|  |  |  |
| --- | --- | --- |
| Problem size | CPU Reference Timing (ms) | CUDA Stage 1 Timing (ms) |
| 256x256 px | 0.205 | 0.184 |
| 1024x1024 px | 2.714 | 1.602 |
| 2048x2048 px | 11.673 | 5.641 |
| 4096x4096 px | 71.837 | 17.811 |

By the benchmarks, it is very clear that the CUDA’s performance is very high right from the 256 image. The CUDA code was able to perform better as it was processed my huge number of threads in the kernel. The only place where it will perform slow is where is transfers the memory from host to kernel and kernel to host. The GPU is always slow in this operation. In the 1024 image the time taken for the kernel is almost half the time taken than that of the CPU. And when it comes to huge complex calculations, especially in the 4096 image, CUDA has performed much better than CPU, this is purely because of the massive amount of multithreading that happens in the CUDA. CUDA will perform better if large amounts of calculation has to be done at once.

I have profiled it, and I could clearly see that the memory transfer between host and the kernel takes the highest amount of time, and the process of calculating is finished in an instance.

If I had more time, I would have tried to implement the same with the Reductions, Warp operations, Shared Memory, etc. And if they worked, my code may be a little faster.



**CUDA: Stage 2**

**Description:**

**Host description**

In the second stage of CUDA, I have specified the blocks per grid to be the cuda\_TILES\_X in x axis and cuda\_TILES\_Y in y axis which is in this form dim3 blocks\_per\_dimension(TILE\_SIZE/4,TILE\_SIZE/4) and the threads per block to be global\_width/256 in both x and y axis which is in the form dim3 threads\_per\_block((global\_width/256) \* (global\_height/256)). Host copies of cpu\_whole\_image\_sum and cuda\_whole\_image\_sum is created of equivalent size. sum\_size and value\_size is assigned. Memory allocation of cpu\_mosaic\_value is done, Kernel is called, and processed data is taken out from kernel back to the host, and one small for loop is happening.

**Kernel description**

In the kernel the parameters are received and sent into execution. The parameters are unsigned int cuda\_TILES\_X, unsigned int cuda\_TILES\_Y, int channels,unsigned long long\* d\_mosaic\_sum, unsigned char\* mosaic\_value, unsigned long long\* whole\_image\_sum. The formula for calculating individual threadID is calculated. The main cuda logic is being executed which updates each variable by its calculated value. Finally all the calculated values are added to whole\_image\_sum using a special function atomicAdd().

**Justification:**

**Host Justification**

In host of stage 2, the reason of fixing the size of blocks\_per\_dimension into TILE\_SIZE/4,TILE\_SIZE/4 is because there has to be 8\*8=64 blocks in for all the images. And the reason for fixing the size of threads\_per\_block into (global\_width/256) \* (global\_height/256) is the mosaic\_value should be calculated for that number of times. This is when the blocks and threads will be able to parallelize and work evenly. The reason of keeping the block size as 8 is because I have allocated a dynamic thread allocation formula for each image in threads\_per\_block(). A new cpu\_whole\_image\_sum[4] is created and its corresponding cuda\_whole\_image\_sum has also been created. cudaMalloc() operation is happening where cuda\_whole\_image\_sum is getting its value allocated on the device memory. cudaMemset() is used to Initialize device memory to a value. Then size for the d\_mosaic\_sum and d\_mosaic\_value is separately created and the cpu\_mosaic\_value is allocated in the host and then sent to the kernel for execution. Once the processing is done the processed value is taken out by the cudaMemcpy() which copies data between the host and device. I have received three processed values back from the cuda, they are d\_mosaic\_sum, d\_mosaic\_value, cuda\_whole\_image\_sum. cudaThreadSynchronize() will wait for compute device to finish and runs a small calculation which can be easily handled by the CPU. The reason why the last for loop runs on the host is because very small calculation can be easily handled by the CPU than the GPU. Then they all are being sent into the validation to validate.

**Kernel justification**

The stage1 kernel gets a set of parameters and passes as arguments to the logic. The parameters that the kernel gets are unsigned int cuda\_TILES\_X, unsigned int cuda\_TILES\_Y, int channels,unsigned long long\* d\_mosaic\_sum, unsigned char\* mosaic\_value, unsigned long long\* whole\_image\_sum. Inside the kernel a new variables is defined, which calculates the unique ThreadID that goes into the kernel. The ThreadID is the index for the iteration of the threads. There is a for loop which iterates each thread for 3 times. So each thread will hold a calculate value of mosaic\_value with them. The mosaic\_value is added in the final stage of kernel. To avoid a race condition, I have added atomicAdd() to add the values in the thread into the whole\_image\_sum.

|  |  |  |
| --- | --- | --- |
| Problem size | CPU Reference Timing (ms) | CUDA Stage 2 Timing (ms) |
| 256x256 px | 0.205 | 0.153 |
| 1024x1024 px | 2.714 | 0.202 |
| 2048x2048 px | 11.673 | 0.310 |
| 4096x4096 px | 71.837 | 0.417 |

By the benchmarks, it is very clear that the CUDA’s performance is very high right from the 256 image. The CUDA code was able to perform better as it was processed my huge number of threads in the kernel. The only place where it will perform slow is where is transfers the memory from host to kernel and kernel to host. The GPU is always slow in this operation. In the 256 image the time taken for the kernel is almost half the time taken than that of the CPU. And when it comes to huge complex calculations, especially in the 4096 image, CUDA has performed much better than CPU, this is purely because of the massive amount of multithreading that happens in the CUDA. CUDA will perform better if large amounts of calculation has to be done at once. In this case, even though the amount of calculation that happens is very small, a part of the calculation is handled by the GPU and another part is handled by the CPU. In this stage, both GPU and CPU combinedly makes some calculation, which helped to quickly finish this stage . When the CPU alone works on this task, it is getting much slower.

I have profiled it, and I could clearly see that the memory transfer between host and the kernel takes the highest amount of time, and the process of calculating is finished in an instance.

If I had more time, I would have tried to implement the same with the Reductions, Warp operations, Shared Memory, etc. And if they worked, my code may be a little faster.

A screenshot of a computer

Description automatically generated with medium confidence

**CUDA: Stage 3**

**Description:**

**Host description**

In the third stage of CUDA, I have specified the blocks per grid to be the cuda\_TILES\_X in x axis and cuda\_TILES\_Y in y axis which is in this form dim3 blocks\_per\_grid(cuda\_TILES\_X, cuda\_TILES\_Y) and the threads per block to be TILE\_SIZE in both x and y axis which is in the form dim3 threads\_per\_block(TILE\_SIZE, TILE\_SIZE). The size is set to a specific value which is cuda\_TILES\_X \* cuda\_TILES\_Y \* 3 \* sizeof(unsigned char). The kernel call is being performed which holds the blocks\_per\_grid and threads\_per\_block value as the number of threads to be passed into the kernel area. Along with it these values cuda\_TILES\_X, cuda\_TILES\_Y, cuda\_input\_image.width, cuda\_input\_image.channels, d\_input\_image\_data, d\_mosaic\_value are passed as parameters too.

A cudaThreadSynchronize() after launching the kernel. And the processed data by the GPU is got back to the CPU of the size mentioned above using this cudaMemcpy(cuda\_output\_image.data, d\_output\_image\_data, image\_data\_size, cudaMemcpyDeviceToHost)

**Kernel description**

In the kernel the parameters are received and sent into execution. The parameters are unsigned int cuda\_TILES\_X, unsigned int cuda\_TILES\_Y, int width, int channels,unsigned char \*output\_data, unsigned char \*mosaic\_value. New values are assigned in the kernel, and they are int tile\_index, int tile\_offset, int pixel\_offset. The main cuda logic is being executed which updates each variable by its calculated value. Finally, all the calculated values are memory copied back to the output\_data which holds all the data to the entire output image.

**Justification:**

**Host Justification**

In host of stage 3, the reason of fixing the size of blocks\_per\_grid into cuda\_TILES\_X \* cuda\_TILES\_Y is because the tile\_index and tile\_offset should be calculated for the multiple of cuda\_TILES\_X and cuda\_TILES\_Y times. And the reason for fixing the size of threads\_per\_block into TILE\_SIZE \* TILE\_SIZE is the pixel\_offset should be calculated for that number of times. The default value of TILE\_SIZE is 32, so 32\*32=1024. As a result, 1024 threads are assigned to each block. And the block count differs for each image as it is depending on the cuda\_TILES\_X and cuda\_TILES\_Y. The size is allocated to a dynamic value which differs for each image and the formula used is, cuda\_TILES\_X \* cuda\_TILES\_Y \* 3 \* sizeof(unsigned char). As the kernel call happens, the blocks\_per\_grid and threads\_per\_block are provided to create those number of threads in the kernel. Along with it I am sending cuda\_TILES\_X, cuda\_TILES\_Y, cuda\_input\_image.width, cuda\_input\_image.channels, d\_output\_image\_data, d\_mosaic\_value as parameters. After the process happens in the kernel, the data has to be taken back to the host. That is through cudaMemcpy(), The processed value of cuda’s mosaic sum value is copied to the cpu’s mosaic sum variable, and it is sent to the validation to verify.

**Kernel Justification**

The stage3 kernel gets a set of parameters and passes as arguments to the logic. The parameters that the kernel gets are unsigned int cuda\_TILES\_X, unsigned int cuda\_TILES\_Y, int width, int channels, unsigned char \*output\_data, unsigned char \*mosaic\_value. Inside the kernel, 3 new variables are defined so that each thread that goes in must have their own copy of int tile\_index, int tile\_offset, int pixel\_offset. There is if condition which will allow the thread to pass in only when the blockIdx.x of the thread and blockIdx.y is less than cuda\_TILES\_X and cuda\_TILES\_Y. This is done to make sure that all the threads in a block have its own copy of tile\_index and tile\_offset. Then the threads has to go through another condition where, if the threadIdx.x and threadIdx.y are less than TILE\_SIZE it has to calculate the pixel\_offset and each thread has to store a copy of all these and calculate the pixel. All the calculated data is then being transferred back using memcpy() and is stored in the output\_data, which is an empty image where all the processed data is being stored. Once that’s done the created Image is taken back to the host memory.

|  |  |  |
| --- | --- | --- |
| Problem size | CPU Reference Timing (ms) | CUDA Stage 3 Timing (ms) |
| 256x256 px | 0.205 | 0.165 |
| 1024x1024 px | 2.714 | 0.996 |
| 2048x2048 px | 11.673 | 3.873 |
| 4096x4096 px | 71.837 | 14.277 |

By the benchmarks, it is very clear that the CUDA’s performance is very high right from the 256 image. The CUDA code was able to perform better as it was processed my huge number of threads in the kernel. The only place where it will perform slow is where is transfers the memory from host to kernel and kernel to host. The GPU is always slow in this operation. In the 256 image the time taken for the kernel is almost half the time taken than that of the CPU. And when it comes to huge complex calculations, especially in the 4096 image, CUDA has performed much better than CPU, this is purely because of the massive amount of multithreading that happens in the CUDA. CUDA will perform better if large amounts of calculation has to be done at once.

I have profiled it, and I could clearly see that the memory transfer between host and the kernel takes the highest amount of time, and the process of calculating is finished in an instance.

If I had more time, I would have tried to implement the same with the Reductions, Warp operations, Shared Memory, etc. And if they worked, my code may be a little faster.