Report

OpenMP: Stage1

CPU: Ryzen 5 3600   6/12thread

Description

This section basically keeps the loop structure of the original code, with the #pragma omp parallel for flag added at three places: tile x, tile y, Pixel x, etc., for parallel computation. The sum is calculated in a double layer loop for Pixel x and Pixel y, and uses reduction(+: sum). The loop level has been adjusted by moving the channel loop outside the double layer loop that computes sum.

Justification

Using #pragma omp parallel for turns the loop into a multi-threaded simultaneous execution, but the final calculation of sum creates a write conflict, so use reduction(+: sum) to prevent conflicts. The reason for moving the channel loop is that the variables needed to determine the subscript of sum are pixel x and pixel y, and no other variables are included, so it makes no sense to traverse channels in reduction(+: sum). So the choice is to iterate outside of reduction(+: sum).

In addition, #pragma omp parallel for was not chosen for traversing pixel y because it was tested that adding it would reduce performance. The use of #pragma omp parallel for for channel loops does not have much impact on performance.

Performance

Problem size	CPU Reference Timing (ms)	OpenMP Stage 1 Timing (ms)
256 x 256 px	0.211	0.126
1024x 1024 px	2.972	0.617
2048 x 2048 px	10.494	1.691
4096 x 4096 px	45.926	10.770

In several tests, results that using #pragma omp parallel for   for pixel y reduces performance. This is due to the fact that too many fork levels can reduce performance. As a result of the above, I believe that the current use of #pragma omp parallel for is not optimal and that performance can probably be optimised by increasing or decreasing the use of #pragma omp parallel for over many more attempts.

OpenMP: Stage2

Description

This stage starts with a double-level parallel for loop (channel, tile index) to calculate the sum. Use reduction(+: sum) to avoid thread conflicts.
The next step is to average each channel using parallel for.

Justification

The use of parallel for and reduction(+: sum) is intended to improve performance while avoiding incorrect results.

The final result is correct and there is some performance improvement when working with large images

Performance

Problem size	CPU Reference Timing (ms)	OpenMP Stage 2 Timing (ms)
256 x 256 px	0.059	0.062
1024x 1024 px	0.078	0.069
2048 x 2048 px	0.071	0.074
4096 x 4096 px	0.131	0.108

This is because this part is about summing individual values and there is not much room for parallel operations. The next attempt to use dichotomous summation may improve the performance of the summation part somewhat.

OpenMP: Stage3

Description

This step does not contain a summation operation, so this section uses a similar mechanism to the original code, and adds parallel for to the tile x, tile y, Pixel x loop.

Justification

This part of the code does not add parallel for to all levels of the loop, because in testing, adding parallel for to channel or Pixel x would reduce performance, and the current three levels of parallelism are a better combination after many attempts.

Performance

Problem size	CPU Reference Timing (ms)	OpenMP Stage 3 Timing (ms)
256 x 256 px	0.208	0.131
1024x 1024 px	3.333	1.473
2048 x 2048 px	14.303	6.012
4096 x 4096 px	63.606	25.202

In terms of time consumed, the performance is about 2.5 times higher, while the cpu of the environment the code is running on has 6 physical cores, and 12 logical cores. Clearly the cpu is not being used to its full potential. This is probably due to the fact that the threads are not divided up properly. The next step could be to improve performance by dividing the threads more rationally, for example by performing assignment operations in parallel in groups of 12 mosaic blocks. Or consider using a double layer of parallel for to improve the performance of the code.

CUDA: Stage1

GPU: RTX2060 6G

Description

This part do a sum operation on each mosaic block separately by running the sum function on the GPU. The number of blocks running is the number of mosaics, and the number of threads in the block is one quarter of the mosaic area. In the sum function some initial data is first calculated or obtained, such as the x and y subscripts of the current mosaic block. Next, all the data in the mosaic block is added up in groups of four and stored in a piece of shared memory, the length of which is one quarter of the mosaic block.

After waiting for synchronisation, the length of the valid data in the shared memory is continuously accumulated to half of the original length. As TILE_SIZE has already been determined, this part of the accumulation code is determined at compile time using a template, and eventually calculated using the __device__ function last64 when 64 numbers are left, with the final mosaic colours summing to the first of the shared memory array.

Eventually, the data is written to the corresponding location in the sum array.

Justification

In this function the data is first summed in groups of four in order to change the number of

threads that need to be opened to a quarter of the original number, so that more threads can be concurrent at the same time and thus increase the speed.
The first calculated sum is stored in a shared array because shared memory is very fast to access, so the summation process can be moved to faster memory, which speeds up the program.
Next, using a template to create the code reduces unnecessary if judgements. This saves the consumption of multiple judgements.

Performance

Problem size	CPU Reference Timing (ms)	CUDA Stage 1 Timing (ms)
256 x 256 px	0.211	0.059
1024x 1024 px	2.972	0.107
2048 x 2048 px	10.494	0.246
4096 x 4096 px	45.926	0.776

The algorithm in this section does not parallelise the channels and the data in the channels are not determined by #define.
The code at the beginning of this section merges the data in groups of four. This is to reduce the number of idle threads, so that you can try to find the fastest algorithm by summing in groups of two or by copying the data directly to shared memory, or even more.

CUDA: Stage2

Description

This section first uses average to find the final colour of each mosaic in the gpu, and then uses the sun_4_v2 function to find the sum.

The average function opens a thread directly for each sum data to find the average.

sum_4_v2 uses two blocks of memory alternately for the original and the resultant data. And the final result is calculated in groups of four.

Justification

Opening a thread for each pixel maximises the use of the gpu's multi-threading capabilities.

Summing in groups of four reduces the number of summing cycles.

Performance

Problem size	CPU Reference Timing (ms)	CUDA Stage 2 Timing (ms)
256 x 256 px	0.059	0.162
1024x 1024 px	0.078	0.157
2048 x 2048 px	0.071	0.165
4096 x 4096 px	0.131	0.155

In the end this part performs poorly relative to the cpu, and the second summation function does not perform well, as this part takes only 0.05ms to process a 4096 x 4096 px image after commenting out the parts other than averga. Next, consider optimising the performance of the sun_4_v2() function and optimising the cpu's calls to the sun_4_v2() part of the loop code.

Another part of the reason is that the cudaMemcpy() function is needed at the end of the step to output the final result in order to sum up the pixels. Once the result was known to be correct and this part was commented out, the performance was significantly improved.

CUDA: Stage3

Description

This section creates a block for each mosaic block, with the number of threads in each block being one quarter of the number of pixels in the mosaic block.
As the function runs, each thread assigns values to the four output pixels. Each pixel has three channels, which means that each thread will assign values to 12 memory locations.

Justification

Each thread needs to calculate the corresponding original mosaic coordinates or calculate the image coordinates that need to be output. When using a group of four it is possible to reduce the coordinates calculation by 75% and reduce the number of threads per block, allowing the program to create more blocks to parallelise more mosaic blocks.

Performance

Problem size	CPU Reference Timing (ms)	CUDA Stage 3 Timing (ms)
256 x 256 px	0.208	0.053
1024x 1024 px	3.333	0.095
2048 x 2048 px	14.303	0.219

4096 x 4096 px	63.606	0.742

Next, I can allow threads to process other numbers of pixels, e.g. 2, 8 or 16 per thread, in order to obtain higher performance.