Harun Sasmaz

59900

Comp 429 – Assignment 2

REPORT

1. Introduction

In this assignment there are three parts which includes different optimization steps and built top on each other. This submission includes three implementations named as part1.cu, part2.cu, and part3.cu respectively. All three parts are working properly and giving the correct average sum of the pixels as the result. However, there is negligible deviation from the serial version’s average sum because of the data type that is used in computations, float. Floats are four-byte floating type and it can differ as a result of the pattern of summation. Serial version adds number one by one, consecutively. On the other hand, my implementation uses a reduction kernel which is discussed in the class. Reduction kernel sums the elements pairwise and hence some floating points might differ than the serial version. Amount of deviation is between 0.001 and 0.002, which can be considered as negligible. Note that all three parts using the same reduction kernel, so performance increase due to the reduction kernel applies for each part equally.

This assignment uses GFLOP/s as the performance metric. In comparison, where coffee.pgm with 100 iteration and 16x16 thread blocks are used, the third part gives the best performance and the first part gives the worst performance. The statistical analysis of performances of each part will be discussed with plots and charts in the next section. However, a file named denoise.out is provided which is the result of KUACC batch job. In the next section, performance discussion will be based on thread blocks 8x8, 16x16 and 32x32.

1. Implementation

There are four kernels;

1. Reduction Kernel

This kernel uses the serial addressing with no bank conflicts approach discussed in the class. It involves two shared float arrays to store summation results of each block. Since there are two summation in the serial version, we need two shared arrays. Note that this kernel uses a one-dimensional grid, because it is more appropriate choice for reduction. Since shared arrays require a constant size, a global variable BLOCK\_SIZE is defined and it is tuned as 64, 256, 1024. To run this kernel, two device arrays is defined, sums and sums2 with size the number of blocks included in grid.

1. Standard Deviation Kernel

This kernel is to avoid ping-pong cudaMemcpy usage. It is run by only one thread on the device side. It takes sums and sums2 as inputs and do the computation same as the serial version by using a for-loop. To update standard deviation, I allocate a one-element pointer as std\_dev, this kernel updates std\_dev device variable to be used in the next kernel. However, the loop inside the kernel must know the number of elements in sums and sums2 arrays, so the number of blocks is also passed as input.

1. First Compute Kernel

This kernel is run by 2D thread blocks, threads in each block calculates their index, read related values from image array and update the directive arrays according to their index. Since halos are not included in computation, indexes related to halos are excluded with an if statement. This kernel also includes the update of diff\_coef array, so it locally calculates needed variables and updates the related index of diff\_coef array. The first part doing these updates by reaching global memory each time. However, the same index of image array and directive arrays are reached multiple times. So, the second part handles this by declaring local variables and keep those elements in their register files. As a result, each thread declares six new variables and do the computation by using them.

1. Second Compute Kernel

This kernel is also run by 2D thread blocks, threads in each block calculates their index, read related values from diff\_coef array, computes divergence and write to the related index of image. However, reading pattern includes redundant reads because each thread reads an element from the next row and an element from the next column. To avoid this, the third part uses a shared memory. A block first brings related elements of diff\_coef into shared memory, and then read from the shared memory to do computations. Again, to avoid updates in halos, there is an if statement which prevents halos to be updated.

By using these kernels, the program copies the image array to the device and there is no need to directional derivative arrays or other computation variables on the host side. All of the arrays are allocated in device and we avoid redundant cudaMemcpy calls. In the main for loop, we just call the kernels in order and when the computation is done, we copy image array back to the host side just once.

1. Performance Results

The table below shows the GFLOP/s rates for each part and for each block size. All measurements are for “coffee.pgm” and 100 iteration.

|  |  |  |  |
| --- | --- | --- | --- |
| Parts \ Block Size | 8x8 | 16x16 | 32x32 |
| Serial | 1.045865 | 1.045865 | 1.045865 |
| Part I | 19.828997 | 48.929657 | 62.013393 |
| Part II | 20.772631 | 51.312778 | 65.273964 |
| Part III | 20.842783 | 51.937889 | 61.773754 |

A close up of a map

Description automatically generatedA picture containing text, water, white, table

Description automatically generatedPart I: Part II:

Part III:

A picture containing text, water, white, group

Description automatically generated

1. Discussion

If we compare the results by thread block size, it seems that when we increase the block size, the performance increases. However, in the third part, when the block size is 32x32, we observe that the performance is lower than the second part. The reason behind this is the second compute kernel. It uses 33x33 shared memory, which is clearly larger than 32x32, this amount of request results in worse performance because we use more than we have.

If we compare results within a constant block size, let it be 16x16, the performance slightly increases part by part. The reason why the performance does not increase in a considerable amount is because each part uses the same reduction kernel. It plays an integral on the performance improvement. Other optimizations such as register usage or tiling have a slight effect on the performance.

If we compare the overall results, the second part with the 32x32 thread blocks must be chosen to apply for this algorithm. In comparison to the serial version, we can achieve up to 64 times performance improvement, which can be higher. Some performance improvements still can be done, for example, thread shuffling in reduction kernel or a warm-up kernel to set some environment variables on the device side and also tiling approach can be applied to the other kernels