GPU, Many-core and Cluster Computing Assignment - LLLL76

1 STEP ONE

Report demonstrates a mastery of performance analysis tools (5). Bottleneck of codes are identified and characterised (5). Performance model is set up in the report and it is calibrated to the used machine and code (5). Some reasonable predictions are made how much can be improved by performance optimisation (5).

1.1 RESULTS

Figure 1.1:

Function / Call Stack	CPU Time	Instructions Retired	Estimated Call Count	Total Iteration Count	Loop Entry Count
computeP	75.9%	49.7%	0	2,697,714,533	58,782,416
setPressureBoundaryConditions	20.9%	48.2%	185,196	3,515,717,852	182,351,714
getCellIndex	2.3%	1.4%	0	0	0

Figure 1.2:

Function / Call Stack	CPU Time	Memory Bound	FPU Utilization	CPI Rate
[Loop at line 654 in computeP]	47.552s	56.6%	2.8%	2.106
[Loop at line 594 in setPressureBounda	8.169s	4.4%	0.0%	0.621
getCellIndex	1.431s	16.7%	7.8%	3.130
[Loop at line 576 in setPressureBounda	0.621s	2.0%	0.0%	0.289

- These results are taken from... - Talk about compiler optimisation variants - All IO is removed so as to remove superfluous stuff - Used hamilton to run so as to get pure results - would a different machine make a difference - model would probably be amdals - this program will be memory bound - differences that parameters make

1.2 BOTTLENECKS

The bottle neck, as shown above is the loop in the function 'computeP'. This takes up a very large amount of the CPU Time. Another computationally significant part of the code is the function 'setPressureBoundaryConditions', whilst this does not make up as much of the CPU time as 'computeP' it is still a significant percentage.

1.3 PERFORMANCE MODEL

1.4 PREDICTIONS

2 STEP TWO

Report gives a clear description what has been realised in the code (5). Vectorisation report results are summarised in report and discussed (5). Runs with various input data sets have been made (5). Results are presented with state-of-the-art techniques (proper figures, e.g.) (5).

2.1 Description

In step two we work to split the computational domain into smaller cubes and vectorise any parts that can be. A loop can be vectorised if it meets a certain criteria; it must be countable, meaning that the number of iterations must be known before we enter the loop, it must have only a single entry and a single exit, it also must have no branching, ie no if statements. The other criteria is only inner loops, no function calls, and no internal dependencies.

The criteria that this code breaks is the fact that it's main loop in the 'computeP' function contains branching, if this was removed then this loop could be vectorised as it meets all other criteria. We start by splitting the overall box we are working in into smaller cubes, the dimensions of these cubes is hard-coded at a single point in the code, this is easy to change. If the dimensions of the box is such that it cannot be made up of complete cubes then the code will halt. If the input variables are such that the box can be made up of a whole number of cubes then the code will move onto working out if a given cube can be vectorised, this would be the case if, out of all the cells in the given cube, none of them are part of the obstacle. If the given block contains no cells from the obstacle then it can be vectorised as in the standard realisation of the code the if statement is a check if the current cell is part of the obstacle. Thus the pre-processing for checking the status of the cells is required to allow the non-obstacle blocks to be vectorised and the blocks that do contain part of obstacle to be treated with the standard code given, using the if statement.

The stages of the vectorisation are as follows, pre-processing takes place, this will check all cells in each block and set a flag as to what is contained within the block, obstacle or not. This will then be used to choose which set of for loops should be used, the vectorised or non-vectorised. The vectorisation is done using the standard intel SIMD pragma.

The pragma SIMD alters the order of cell propagation, the order is now block by block rather than cell by cell the overall effect is negligible.

2.2 Vectorisation Reports

The vectorisation report for the unvectorised version of the code shows that the function "ComputeP" has no vectorised parts.

```
LOOP BEGIN at gpu1.cpp(659,9)
        remark #15344: loop was not vectorized: vector dependence prevents
        vectorization. First dependence is shown below. Use level 5 report for details
        remark #15346: vector dependence: assumed ANTI dependence
        between _rhs[_ix+_iy*(_numberOfCellsPerAxisX+2)+(_iz*(_numberOfCellsPerA (670:25)
        and _p[$i4] (673:13)
LOOP END
```

There are two loops that are present in step two, one that is vectorised and allowed for blocks that do not contain any parts of the obstacle to avoid the if statement in the standard loop that is unvectorised.

```
LOOP BEGIN at gpu21.cpp(594,17)
                remark #15301: SIMD LOOP WAS VECTORIZED
LOOP END
```

This is the vectorised loops vec-report, the other loop is almost identical to the loop in step one.

2.3 Results

TO BE DONE AT HOME

3 Step Three

Report gives a clear description what has been realised in the code (5). Runs with various input data sets have been made (5). Results are interpreted and future work and shortcomings are identified (5). Results are presented with state-of-the-art techniques (proper figures, e.g.) (5).

3.1 Description

- The arrays that store all values for the cells are padded in such a way that there is a layer of cells surrounding each block.
- at the beginning of each computeP iteration the values of the padded cells are changed to the values of their corresponding non padded cell. (Give a visual example of this)
- This is done via a conversion function that will take in the index of the halo cell and will return the correct value of that halo.
- all other functions that contain loops that used to iterate through the standard array are unchanged however the function call has been changed such that the values passed for the original array are mapped to the corresponding cell in the padded array, this is easier as a mapping function being called instead of changing all for loops in the program is more simple.
- the only place that a padded array can be used is inside the function computeP, in any other function this array cannot be accessed, and thus we write out to the unpadded array after each iteration.
- each padded block is then passed through the OMP parallel pragma such that it is done in parallel using the padded values for any data that would otherwise be from a different block and would thus not allow for parallel as it would introduce data races. After this has been then we will go into a different iteration which would result in the new halo values being written from their corresponding non halo cell..

3.2 Results

3.3 Limitations

- overhead - inaccuracy? - a lot of copying data