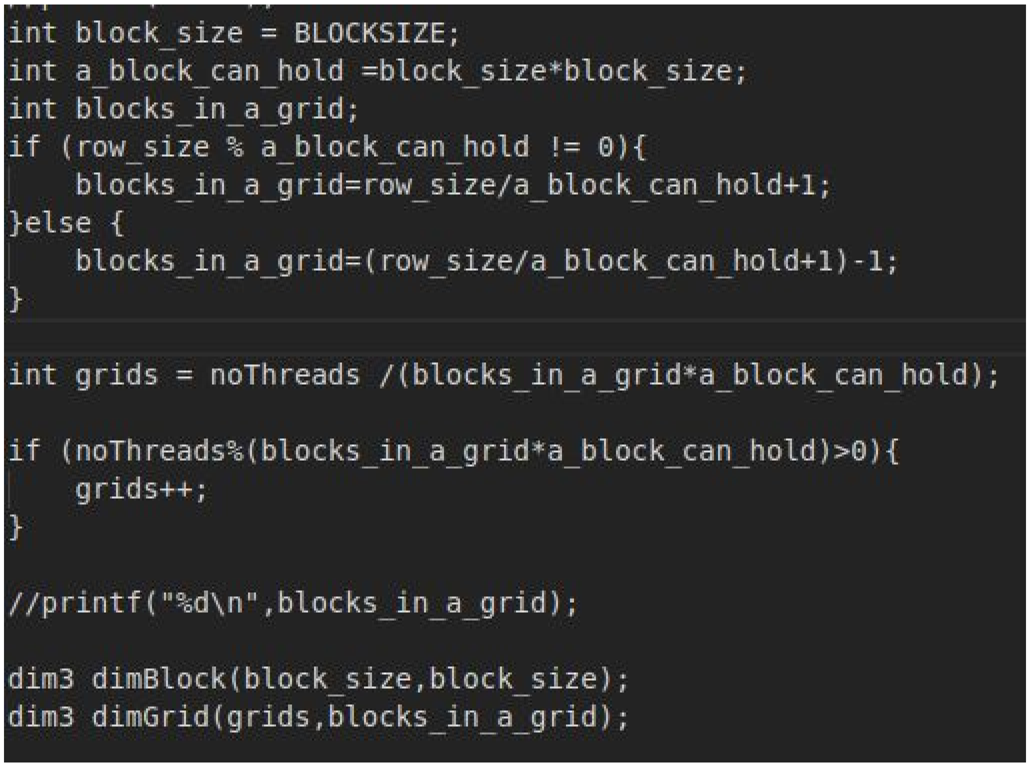
CS426 - Parallel Computing
Project 04 - Report
Aldo Tali
ID: 21500097
Date: 02/06/2019

Part 1) Parallelization strategy used (You can use any parallelization strategy that scales up.)

The parallelization takes two steps in the given implementation. The first step takes into account how many blocks will be put into each grid and the second how many parallel grids we will need. Since i have tested my implementation in google Collab, the maximum block size I was able to achieve was 32 and that is being used as a default constant. In other terms the complete block can hold at most 32 *32 = 1024 threads; This means that we can operate in parallel any number of these as long as we assign enough grids. Of course in the case that the user input is not a power of 2 then the blocks are floored up so that a power of 2 is achieved. The extra threads in this case will not take part on the computation they will be idle. So suppose the user enters 5 then the algorithm will assume as if 8 was entered and 3 were idle. In this architecture each thread will compute its section of elements for the vector result.



```
int block_size = BLOCKSIZE;
int a_block_can_hold =block_size*block_size;
int blocks_in_a_grid;
if (row_size % a_block_can_hold != 0){
    blocks_in_a_grid=row_size/a_block_can_hold+1;
}else {
    blocks_in_a_grid=(row_size/a_block_can_hold+1)-1;
}

int grids = noThreads /(blocks_in_a_grid*a_block_can_hold);

if (noThreads%(blocks_in_a_grid*a_block_can_hold)>0){
    grids++;
}

//printf("%d\n",blocks_in_a_grid);

dim3 dimBlock(block_size,block_size);
dim3 dimGrid(grids,blocks_in_a_grid);
```

Part 2) Figures for each test matrix.

Below all the given figures have been tested in different thread number and in a max iteration number 5. All the given times are in ms. The time measurements depict only the matrix vector multiplication loop. The times were recorded on Google Collab notebook.

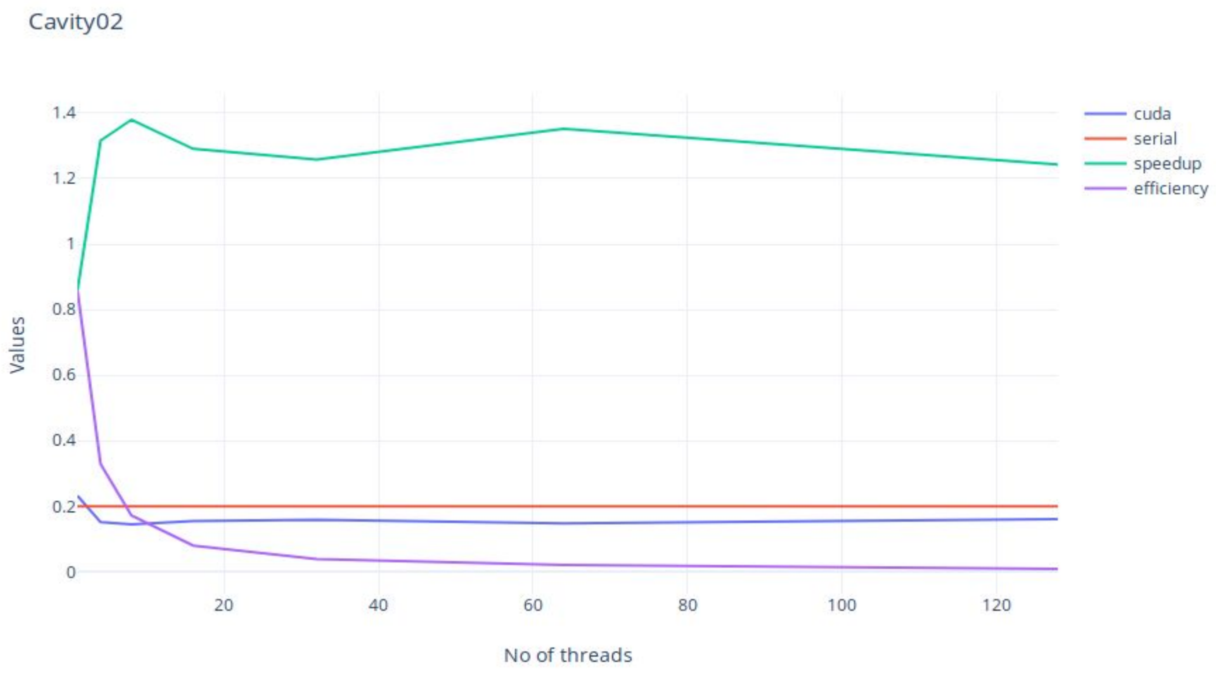


Figure 1.0

Figure 2.0

Threads	Parallel Time	Serial	Speedup	Efficiency
1	0.233	0.2	0.85836909871	0.85836909871
4	0.152	0.2	1.31578947368	0.32894736842
8	0.145	0.2	1.37931034483	0.1724137931
16	0.155	0.2	1.29032258065	0.08064516129
32	0.159	0.2	1.25786163522	0.0393081761
64	0.148	0.2	1.35135135135	0.02111486486
128	0.161	0.2	1.24223602484	0.00970496894

Table 1.0 The values for the Cavity 02 recorded for the figures 1.0 and 2.0 recorded.

Fidapm08 Data figures:

Figure 3.0

Figure 4.0

Threads	Parallel Time	Serial	Speedup	Efficiency
1	0.804	1.5	1.86567164179104	1.86567164179104
4	0.501	1.5	2.9940119760479	0.748502994011976
8	0.516	1.5	2.90697674418605	0.363372093023256
16	0.533	1.5	2.81425891181989	0.175891181988743
32	0.522	1.5	2.8735632183908	0.089798850574713
64	0.51	1.5	2.94117647058823	0.045955882352941
128	0.537	1.5	2.79329608938548	0.021822625698324

Table 2.0 The values for the Fidapm08 recorded for the figures 3.0 and 4.0 recorded.

Fidapm11 Data figures:

Figure 5.0

Figure 6.0

Threads	Parallel Time	Serial	Speedup	Efficiency
1	2.483	12.5	5.03423278292388	5.03423278292388
4	2.473	12.5	5.05458956732713	1.26364739183178
8	2.466	12.5	5.06893755068938	0.633617193836172
16	2.552	12.5	4.89811912225705	0.306132445141066
32	2.469	12.5	5.06277845281491	0.158211826650466
64	2.467	12.5	5.06688285366842	0.079170044588569
128	2.502	12.5	4.99600319744205	0.039031274980016

Table 3.0 The values for the Fidapm11 recorded for the figures 5.0 and 6.0 recorded.

Part 3) Short discussion about the results.

It can be clearly seen from the figures that as the size of the matrix increases the time required for the matrix vector multiplication to happen is proportionally larger if we were to run the serial script. With the increase of the number of threads the overall execution time follows a trend of maintaining similar times of computation. In all examples the outliers are in the beginning when the number of threads is too small and in the end when the number of threads gets ridiculously large, meaning that no useful job will be done regardless of the threads. Judging from these expectations and results then the speedup being S = Ts/Tp should start at a good rate when the number of threads is low, increase as the number of threads gets larger and eventually somehow balance its execution around some convergence point. Now these results should be logical since initially we start with a larger computation time for the parallel computation and as we increase the threads the amount of benefit starts decreasing and therefore the convergence point. Completely the opposite holds for efficiency. Since its formula is E = S/p where p is the number of processes/threads then basically as the threads increase at each point in our graph where the speedup increases the efficiency should drop and inn the reverse case the opposite should happen. Of course this is observable in all our graphs. Perhaps the most reliable dataset is fidapm11.mtx since the matrix itself is larger which in turn means that we can judge the performance on a macro mode. As we can see the serial implementation gives an overall 12.5 ms execution time for all 5 iterations whereas CUDA processing falls to about 6 times less. Of course in making the correct judgements we should also consider the initial overheads, however due to the computation of S and E those were left out in this report. In that case however we would expect cavity02.mtx to have a better serial execution and fidapm11.mtx a better parallel execution.