INF3201: Mandatory Assignment 3

Håkon Atlason

Fall 2021

Abstract

A GTX1070 GPU was shown to be up to 93.9% faster when solving the heat distribution problem, compared to an i5 4690K CPU (simple scheme with 1024x1024 matrix).

1 Introduction

This report will discuss the design, implementation, and experiment results of three different designs of a heat distribution problem-solver. All three designs will be implemented in two version to be run either on a central processing unit (CPU) or graphics processing unit (GPU).

2 Background

The heat distribution problem revolves around calculating the change in temperature in a two-dimensional matrix where three sides are kept at absolute zero (-273.15°C), and one side is hot (40.0°C). The change in temperature of each element in the matrix depends on the temperature of its four orthogonal neighboring elements, and is scaled by a constant factor Ω. The temperature of the elements at the edges of the matrix are always held constant. For each iteration, each pixel is updated once, and the matrix is updated iteratively until the change Δ reaches a certain lower threshold ϵ.

3 Sequential solution analysis

The sequential solution uses a nested loop to loop over each entry in the matrix and updating them. This implementation was profiled (using Kcachegrind) with a 256×256 matrix until convergence $\epsilon = 10^{-3} \cdot 256^2$. This showed that 100.00% of the time was spent in the solver function. The solver function handles calculating new matrix entries, as well as calculating the delta (ie. the sum of absolute differences between the old and new matrix). As this part is parallelizable, this suggests that linear speedup should theoretically be possible. However, this is without considering the overhead that parallelizing involves. The sequential solution was timed, and the theoretical time was estimated with linear speedup with 4 cores (see table 1).

Table 1: Measured execution time in seconds of sequential implementation, and theoretical time assuming linear speedup with 4 cores.

	128px	256px	512px
Measured	0.669	7.38	74.6
4CPU Th.	0.167	1.85	18.7

4 Design

Three different design approaches - simple, red-black, and double-buffer - were used to implement a parallel HDP-solver. In all cases the program runs until the delta-value reaches a certain epsilon.

With the simple implementation, pixels are updated without considering data races. This means that the result of an iteration of calculations is non-deterministic, in contrast to the two other designs. This is because the value of each pixel is needed in order to update its orthogonal neighbors, but the result of the update will differ based on whether or not its neighbors have already been updated for the iteration. For each iteration, the sum of the absolute difference between each new and old entry is calculated.

The red-black design uses an approach where each for each iteration, every other pixel is updated. This is done alternatingly, which means every pixel can be thought of as being in one of two categories ("red" and "black"), hence the name. The "color" of each pixel is the same as its four diagonal neighbors. This scheme means that there are no data races when updating pixels (however, the deltas still need to be updated in a thread-safe manner) and its result is therefore deterministic, meaning that given the same parameters, the result will always be the same. Using this scheme means that for each iteration, only (roughly) half the pixels are updated, so to achieve a "full" iteration where all pixels have been updated once, two of these iterations are needed.

The double-buffer scheme keeps two matrices to solve the HDP. For each iteration, one matrix is used for writing the new entries, and the old entries are kept in the other matrix. For each iteration, the roles of the matrixes are swapped. This scheme also avoids race conditions, as while entries are being updated, the relevant old entries are available as a reference. This means this scheme is also deterministic.

5 Implementation

All three design were implemented both in C with the OpenMP library and in CUDA.

5.1 C/OpenMP implementation

The general idea behind the parallelization in OpenMP is to find a for-loop to parallelize. For all three schemes, the outer of two for-loops used to iterate through the matrix (which would iterate through y and x-values) was made parallel.

Using this approach for the simple scheme, however, lead to segmentation faults, and therefore the algorithm was changed in such a way that the matrix is calculated in two passes, with every other column in the matrix being updated in each pass. This however means that it effectively is quite similar to how the red-black scheme functions. However, this technically does not rid the implementation of race conditions, as each column is updated iteratively, every entry will, when updated, need to be updated, while simultaneously need to be used to update its neighbors. However, since the columns are updated linearly from start to finish, this makes this specific implementation deterministic.

The implementation of the red-black scheme in OpenMP is similar to the simple scheme implementation, with the difference being that every diagonally connected pixel is updated with each of the two passes that constitute an iteration. The decision was made to have two passes in each iteration, rather than only have one pass per iteration, in order to avoid adding complexity to delta-calculations, and to keep the experiment results across the schemes as comparable as possible.

All three OpenMP implementations need a thread-safe way to do add up the total delta. The way this delta sum reduction of each partial delta was implemented, each thread keeps its own private delta when updating a column, and when finished with the column, it acquires a lock for the global total delta, updates it, and releases the lock.

5.2 CUDA implementation

In the CUDA implementations, the x and y indexes for the grid are calculated from each thread's block index, block width, and thread index. The CUDA simple scheme implementation computes all elements at once, not splitting it into two passes like the OpenMP implementation of the same scheme. The CUDA red-black and double-buffer implementations do two passes, with a barrier between, with the result being one full iteration, and two iterations, respectively. This means that the double-buffer implementation only does one delta-calculation for every two passes, half as many as the other two implementations.

All CUDA implementations calculate the delta when updating entries, and enter them into a matrix of the same size as the grid, in order to keep the delta values. The delta-array is sum-reduced using an in-built function.

6 Experiments and results

Each of the six implementations were run and timed at different grid sizes (see figure 1). The set-up and tear-down of the program was not included in time. The results can be seen in table 2. Note that each data point is based on only one measurement. The CUDA implementation was run with a constant block size of 32x32. $\Omega = 0.8$ was kept constant for all experiments. The epsilon was kept at $\epsilon = 0.001 \cdot n_{entries} = 0.001 \cdot height \cdot width$.

Table 2: Execution time in seconds as function of problem size.

Size		**128px**	**256px**	**512px**	**1024px**	**2048px**
1CPU	Serial	0.669	7.38	74.6	-	-
Th. 4CPU	Th. Serial	0.167	1.85	18.7	-	-
4CPU	Simple	0.415	2.32	21.2	121	-
„	RB	0.330	2.60	23.7	-	-
„	DBUF	0.531	3.86	27.6	-	-
GPU	Simple	1.79	4.31	12.0	7.35	6.29
„	RB	1.28	3.29	11.8	12.0	14.0
„	DBUF	0.984	2.24	3.15	4.72	4.58

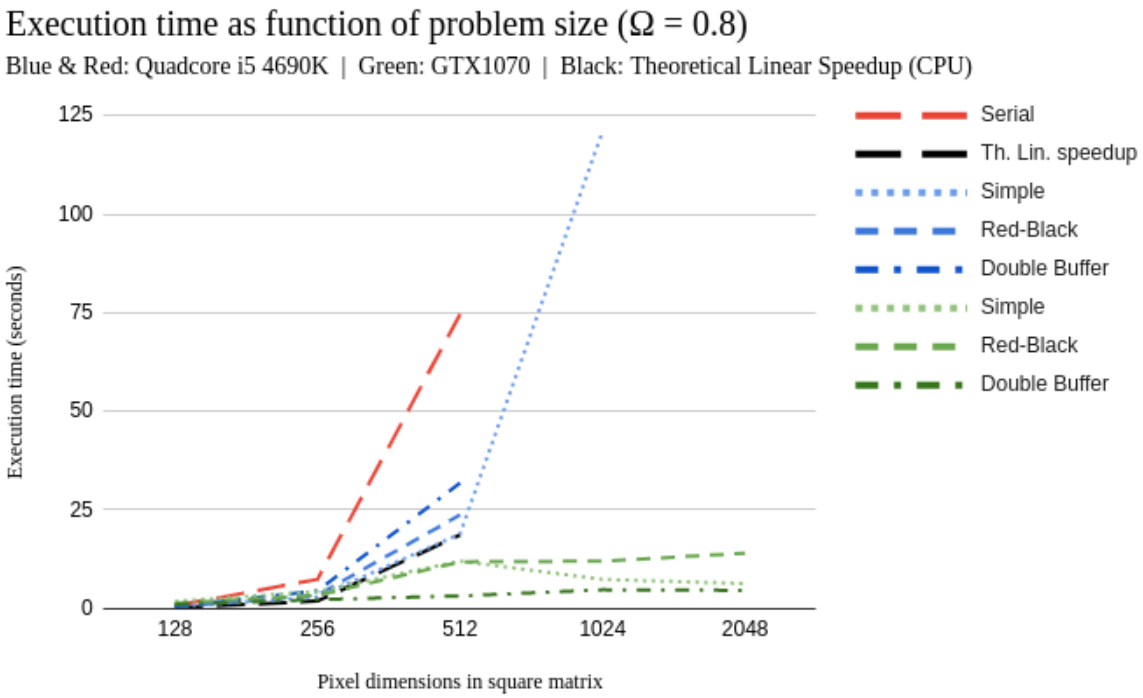

Figure 1: How execution time is affected by matrix size (either $128^2, 256^2, 512^2, 1024^2$, or 2048^2).

The CUDA implementations were profiled using the Nvidia visual profiler, and was used to find the time spent doing matrix entry and delta updates and delta matrix sum reduction. The results are shown in table 3.

The C/OpenMP simple implementations were profiled with KCacheGrind. It showed 29.5% was spent computing pixels, with a further 16.0% in its caller function. For the red-black scheme, the corresponding values were 28.9% and 18.1%. For the double buffer they were 28.8% and 17.8%. For

all implementations, the overhead from the profiling tool made up 42-44% of the execution time. The caller function was used to calculate and reduce deltas (See table 4).

Table 3: Breakdown of CUDA execution times (256x256 matrix).

Scheme	Simple	R-B	DBuf
Pixel and delta calc.	65.5%	78.2%	77.7%
Delta Reduction	34.5%	21.8%	22.3%

Table 4: Breakdown of C/OpenMP exection time (256x256 matrix).

Scheme	Simple	R-B	DBuf
Caller func.	29.5%	28.9%	28.8%
Pixel calc.	16.0%	18.1%	17.8%
Prof. overhead	44.0%	43.2%	42.4%

The number of iterations during these experiments are shown in figure 2. It shows the number of iterations increasing until size is 512x512, and decreasing after.

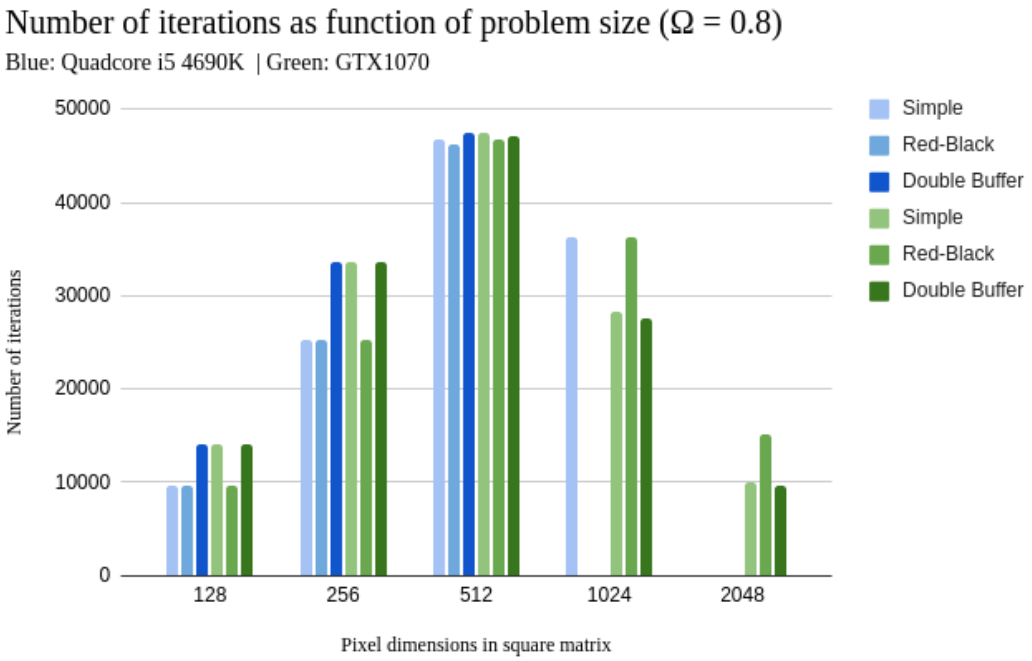

Figure 2: How number of iteratiosn is affected by matrix size (either $128^2, 256^2, 512^2, 1024^2$, or 2048^2).

7 Discussion

The results of the experiments show that a GTX1070 GPU always spends less time than a i5-4690K when the matrix size is 512x512, or higher. The highest GPU speed-up was achieved with a 1024x1024 matrix using a simple scheme, for which the GPU spent 7.35s, compared to the 121s spent by the CPU (93.9% speedup). However, in the case of the simple scheme on a 128x128 matrix, the GPU spent 3.8 times as long as the CPU. This suggests there is some overhead when doing computations on a GPU, and it might not always be worth it to transfer data to it to do computations when the task is small.

Interestingly, the simple scheme achieved the best results on the CPU, while the double-buffer scheme achieved the best results on the GPU. It is believed that this is to some degree helped by the fact that the double-buffer scheme on the GPU does only half as many delta-reductions as the other schemes on the GPU.

The results show that the execution time for the GPU implementations stay relatively stable (compared to CPU execution times), and this might be due to the fact that the GPU implementations have a constant block size, and more blocks are added as the problem size grows. It might also be due to the fact that number of iterations seem to decrease when the problem size is 1024x1024 and larger.

When doing the delta sum reduction in the OpenMP implementation, two approaches were tested. The first was to have each thread keep a partial delta, which was then used to update a global delta, which was kept thread-safe with a lock. The second was to have an array of partial deltas, and have each thread keep its partial delta at its own index within this array, and at the end of each implementation, after leaving the parallel for-loop, sum up the array of partial deltas. When testing the run-time with these two approaches, the lock-scheme was marginally ($<5\%$) faster, and was therefore used.

When doing the delta sum reduction in the CUDA implementation, two approaches were tested. The first was to use make a copy of the matrix before updating it, and using pyCUDA and numpy built-in functions to get the reduced sum of the absolute value of the difference between two arrays. The second was to keep a delta matrix to which delta values are written (explained in Implementation). Running an experiment showed that the second approach was around 30% faster, and it was therefore chosen.

In the previous version of this report, a bug with the CUDA implementation made the results invalid. The results for both CUDA and C/OpenMP in this report have been updated from new experiments. The bug likely stemmed from having threads return from the CUDA kernel before having a synchronization barrier.

8 Conclusion

This report has shown that when solving the heat distribution problem with a simple scheme on a 1024x1024 array, a GTX1070 graphics processing unit running a CUDA program can be 93.9% faster compared to a quadcore i5 4690K central processing unit running a C/OpenMP equivalent program.