INF3201 - Parallel Programming
Parallelizing the Heat Distribution Problem using NVIDIA CUDA and OpenMP

Narae Park and Jonas Fladset Hoel

October 29, 2020

1 Introduction

The Heat Distribution Problem (or HDP for short) involves calculating the distribution of temperature across a set of theoretical particles in an enclosed space. The workload has a high potential for increased time and power efficiency by parallellizing the workload using OpenMP for CPU or CUDA for GPU compute. This paper describes the design of three different solutions that all aim to calculate the thermal distribution in a sequential fashion, as well as using OpenMP and CUDA. The results produced by implementations following the design are described in the Experiments section.

2 Sequential Solution Analysis

2.1 Profiling

In order to solve the Heat Distribution Problem, each pixel in the space updates its own value by referring to the pixel values around it, just as heat is radiated from the space. In the given problem, since the values of the all boundary area are given, all pixels except this area refer to neighbouring pixels one by one to perform an operation and repeat this operation. When the sum of the differences before and after the update of all pixels is below a certain threshold, it is considered that the heat is completely distributed and the repetition is terminated.

There are three solutions to solve this problem:
First, in the Simple solution, pixels are updated by referring to the surrounding pixels without any certain criteria. In a serial solution, the computational results of one pixel will be applied to the computations of the next pixel. It is simple but not elaborate. In parallel solutions, non-deterministic results are obtained since the order is not set.

In the red-black (rb for short) ordering solution, it focuses on the fact that the pixels referenced when each pixel is updated are its top, bottom, left, and right pixel. So red pixels and black pixels in diagonal position with each other are sequentially calculated for each iteration. It has a fixed order, so we get deterministic results even in parallel solutions. It is also possible to sequentially calculate red pixels and black pixels in one iteration. However, it gets fewer iterations, but the overall execution time is longer.

In the rb solution, considering that the pixel referenced when each pixel is updated is a top, bottom, left, and right pixel, red and black pixels are sequentially calculated for each repetition in the following pattern. It has a fixed order, so you get deterministic results even in parallel solutions. In one iteration, it is possible to sequentially calculate red pixels and black pixels. However, it gets fewer iterations, but the overall execution time is longer.

The double buffered (dbuf) solution uses two identical pixel maps (buffer arrays). At each iteration, one pixel map calculates values of surrounding pixels required for calculation by referring to other pixel maps. Like the rb solution, the sequence is ordered, so we get deterministic results even in parallel solutions.

The results of the execution time of the serial solutions are as following table 1. Results in table 1 were produced on a cluster machine with an Intel Core(TM) i7-4790 @ 3.60GHz, cores 4, processors 8.
The average time and standard deviation were derived five times over.

Table 1: Execution time of the sequential solutions

solution	Problem size	Time (std)	Problem size	Time (std)	Problem size	Time (std)	Num of iters
simple	100 * 100	0.395 (0.057)	200 * 200	3.666 (0.024)	300 * 300	13.526 (0.0234)	30417
rb	100 * 100	0.166 (0.040)	200 * 200	1.162 (0.060)	300 * 300	4.0378 (0.058)	42001
dbuf	100 * 100	0.258 (0.018)	200 * 200	1.985 (0.060)	300 * 300	8.466 (0.063)	39103

2.2 Speedup

From the above results we can calculate the ideal speed-up and efficiency by referring to the following equations 1[3]. And this will be used as a basis for parallel solutions to be implemented in the next sections.

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}, \quad E = \frac{S}{p} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}}\right)}{p} = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}}.$$

Figure 1: Definition of Speedup and Efficiency

2.3 Assumptions & Expectations

The temperature of the space does not change sequentially depending on the space area, but works simultaneously, so parallel programming can be applied. And we can expect to get faster speed and performance for solving HDP problem through parallel programming. In addition, GPU computations will be used for solving this problem. As is well known, the GPU was originally intended for display rendering, and this is exactly the problem (updating each pixel of the display at the same time), so parallel programming using the GPU would be a good approach. The assumptions and expectations will then be summarized as follows, which will be identified in the subsequent sections.

2.3.1 Assumptions

1. HDP is a good problem to solve with GPUs made for parallel programming, especially display rendering purposes.
2. There will be overhead in using OpenMP or GPU.

2.3.2 Expectations

1. Both openmp and cuda will perform better than serial solutions.
2. cuda can use more threads simultaneously, so it will perform better than openmp.
3. If the additional cost of using openmp or GPU does not exceed the benefits of using openmp or GPU, the performance will not be as good as expected.

3 Design

The above three solutions were implemented with openmp and cuda parallel programming as follows.

3.1 OpenMP

Openmp's parallel solution is designed in such a way that the computation of each pixel performed within the double for loop is performed simultaneously with multiple threads.

Therefore basically there is not much difference between sequential solutions in execution ways. The number of iterations is the same (except for simple solutions that obtain a non-deterministic result), but the execution time would be greatly reduced because it is performed with multiple threads.

3.2 CUDA

The parallel programming in GPU is fundamentally different from that in CPU, so much consideration is required in the design of the process. Actually, we had to go through a lot of trial and error in the process of implementing and optimizing the cuda solutions.

The computational structure in gpu consists of blocks, which are bundles of threads, and grids, which are sets of blocks.2[2] In our HDT problem, Since one pixel of the frost map can be mapped to each thread of the gpu, many operations can be performed simultaneously depending on the usage.

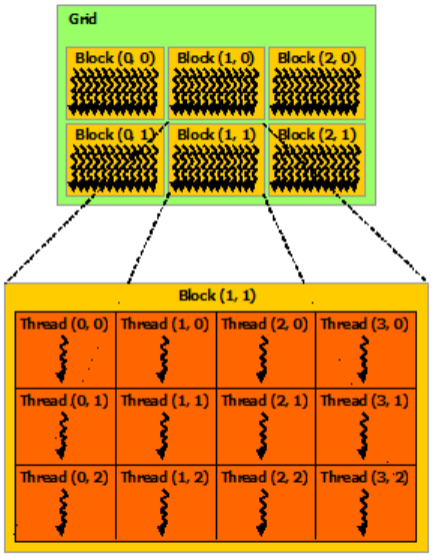

Figure 2: Grid of Thread Blocks

The computation in cuda is done by (1) allocating memory in cpu to gpu, (2) copying the value from cpu to gpu, (3) performing the operation in gpu, and then (4) copying the results from gpu to cpu again. You can refer to the following figure3[1]

Considering the above, there are many factors to consider in the design.

- First, since much time and resources are required to send and receive data between the cpu and gpu, it is necessary to minimize the data transaction.
- Also, the gpu's block can have a maximum size of 1024, but we have to operate on a much larger map. Therefore, the size of blocks and grids must also be considered.
- In addition, in gpu, it is difficult to synchronize between blocks. However, we need to sum the difference before and after the update of the entire pixel map.

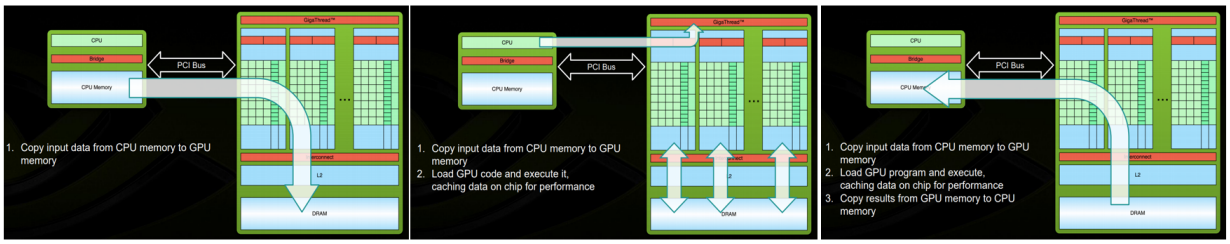

Figure 3: Simple Processing Flow between CPU and GPU

- Finally, the gpu executes the instruction in unit of warp (32 threads), and if there is branches in it, the threads cannot be executed at the same time, so they are executed in serial. We need to use the if branch to detect that pixel in the rb solution, and we need to use the branchs to update red/black, map1/map2 according to the iteration order.

Considering the above points, our cuda solution was designed as follows.

1. Checking the condition whether the delta sum, which is the sum of difference between the values before and after the update, is lower than the epsilon value used as the iteration end point, is performed by the CPU.

2. To execute if branches, the CPU calls the kernel repeatedly and the GPU executes the threads corresponding to the branch sequentially.

3. Since checking the condition by sending the sum result to the CPU from GPU every timecauses a great cost, so the GPU repeats the update and sends the result to the cpu at regular intervals to check the condition.

In the case of the design of the solution is like 2, if a certain period is set to, for example, 100 times, the results of the update within 100 times will not be checked, which may exceed the required number of iterations. However, we thought that the cost of transaction data each time was much greater than the cost of performing the over-executions, so we followed 3.. Actually, we were able to get much faster speed by applying 2. A simple illustration of the above is as following figure 4.

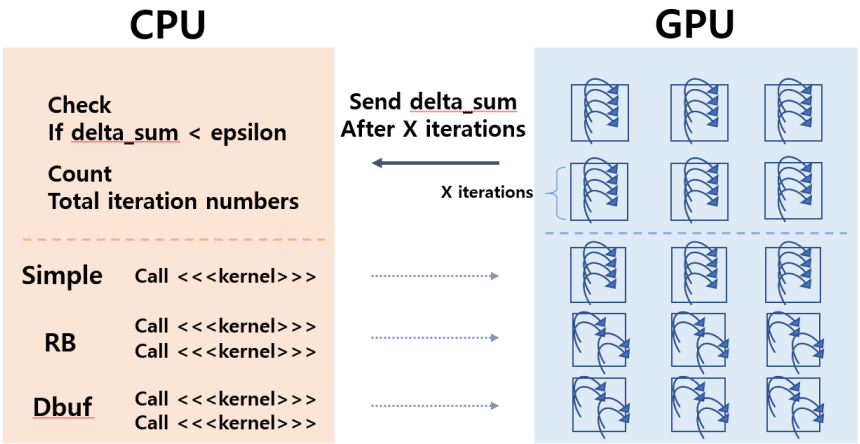

Figure 4: Simple illustration of the CUDA solution design

4 Experiments

4.1 OpenMP

Results in table 2 were produced on a cluster machine with an Intel Core(TM) i7-4790 @ 3.60GHz, cores 4, processors 8.
The computer hardware specifications for this experiment are as follows. Table 2 is the execution result of openmp solutions. We can see that that a large performance improvement has been made, compared to the result of serial solutions in table 1. It can be seen that the advantage of performance improvement through multithreaded execution is much greater than the overhead of using openmp.

Table 2: Execution time of the OpenMP solutions

solution	**Problem size**	**Time (std)**	**Problem size**	**Time (std)**	**Problem size**	**Time (std)**	**Num of iters**
simple	100 * 100	0.121 (0.041)	200 * 200	0.603 (0.032)	300 * 300	2.252 (0.031)	30417
rb	100 * 100	0.141(0.026)	200 * 200	0.397 (0.004)	300 * 300	1.436 (0.021)	42001
dbuf	100 * 100	0.109 (0.034)	200 * 200	0.732 (0.039)	300 * 300	2.436 (0.003)	39103

4.2 CUDA

Results in table 3 were produced on a cluster machine with an Intel Core(TM) i7-4790 @ 3.60GHz, cores 4, processors 8, RAM 32GB (4 x 8GB), GM107 [GeForce GTX 750 Ti].
We tested performance on cluster machines with the same code, and the results were quite different. we have already experienced that the difference in execution time depends on the resource share even on the same machine. We think that it is necessary to consider a way to check more stable performance. We added the two results in this report for reference.

Table 3: Execution time of the CUDA solutions

Simple	**Problem size**	**(100 * 100)**					
Block size	(16, 1, 1)	(32, 1, 1)	(64, 1, 1)	(128, 1, 1)	(256, 1, 1)	(512, 1, 1)	(1024, 1, 1)
Grid size	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)
Time	6.350157	3.742593	2.337364	1.777527	1.630727	1.548133	1.614152
Num. iter	17400	17600	17600	18500	20300	19600	20600
Block size	(16, 1, 1)	(16, 1, 1)	(16, 1, 1)	(16, 1, 1)	(16, 1, 1)	(16, 1, 1)	(16, 1, 1)
Grid size	(1, 1, 1)	(2, 1, 1)	(4, 1, 1)	(8, 1, 1)	(16, 1, 1)	(32, 1, 1)	(64, 1, 1)
Time	6.350157	1.266989	0.192021	0.057971	0.022637	0.014025	0.011916
Num. iter	17400	6000	1400	600	300	200	200

Simple	**200 * 200**						
Block size	(16, 1, 1)	(32, 1, 1)	(64, 1, 1)	(128, 1, 1)	(256, 1, 1)	(512, 1, 1)	(1024, 1, 1)
Grid size	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)
Time	77.55053	41.49465	22.29676	13.00474	10.29132	9.678045	9.976875
Num. iter	59200	59600	59800	59900	67800	63300	66400
Block size	(16, 1, 1)	(16, 1, 1)	(16, 1, 1)	(16, 1, 1)	(16, 1, 1)	(16, 1, 1)	(16, 1, 1)
Grid size	(1, 1, 1)	(2, 1, 1)	(4, 1, 1)	(8, 1, 1)	(16, 1, 1)	(32, 1, 1)	(64, 1, 1)
Time	77.55053	13.06457	1.586316	0.348557	0.082128	0.019855	0.015369
Num. iter	59200	19200	4300	1600	600	200	200

Results in tables 4, 5 and 6 were produced using an NVIDIA GeForce RTX 2060 which has 1920 CUDA cores and a core frequency of between 1365 and 1680 MHz. There is uncertainty to the frequency due to automatic frequency overclocking built into the GPU that cannot be controlled. The resolution of the particle grid was static at 100 by 100, varying block- and grid sizes for CUDA core allocation. If the number of cores

Simple	**300 * 300**						
Block size	(16, 1, 1)	(32, 1, 1)	(64, 1, 1)	(128, 1, 1)	(256, 1, 1)	(512, 1, 1)	(1024, 1, 1)
Grid size	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)
Time	344.3695	180.2848	93.94521	50.59393	35.47549	35.03191	35.74949
Num. iter	119200	119800	120300	120400	127600	127200	129900
Block size	(16, 1, 1)	(16, 1, 1)	(16, 1, 1)	(16, 1, 1)	(16, 1, 1)	(16, 1, 1)	(16, 1, 1)
Grid size	(1, 1, 1)	(2, 1, 1)	(4, 1, 1)	(8, 1, 1)	(16, 1, 1)	(32, 1, 1)	(64, 1, 1)
Time	344.3695	54.72475	6.473553	1.267502	0.242857	0.058	0.020003
Num. iter	119200	37200	8500	3100	1000	400	200

requested is larger than what the GPU has available, the GPU will autonomously re-allocate previously used cores until the number of requested cores have been fullfilled. It should also be noted that all testing were done on a shared cluster, and as such the results produced are subject to shared usage by other users. For this reason and to take into account run-to-run variance, all results shown are average of three runs in the given configuration.

Table 4: CUDA Results, Simple (RTX 2060)

Time Elapsed (avg)	Time Elapsed (std)	Block Size	Grid Size
31,5177	0,165252	1x1x1	1x1x1
9,6223	0,096582	8x8x1	1x1x1
9,6268	0,059923	16x16x1	1x1x1
17,3847	0,150323	1x1x1	2x2x1
4,0021	0,014663	8x8x1	2x2x1
6,5184	0,046318	16x16x1	2x2x1
9,8216	0,027729	1x1x1	4x4x1
11,5583	0,034045	8x8x1	4x4x1
20,6385	0,083983	16x16x1	4x4x1

Table 5: CUDA Results, Double-Buffered (RTX 2060)

Time Elapsed (avg)	Time Elapsed (std)	Block Size	Grid Size
54,1100	0,016383	1x1x1	1x1x1
12,9783	0,137783	8x8x1	1x1x1
12,6021	0,361775	16x16x1	1x1x1
29,9187	0,275439	1x1x1	2x2x1
9,92593	0,025934	8x8x1	2x2x1
16,2379	0,094376	16x16x1	2x2x1
17,1173	0,017348	1x1x1	4x4x1
16,6002	0,057702	8x8x1	4x4x1
29,5567	0,025665	16x16x1	4x4x1

Table 6: CUDA Results, Red-black (RTX 2060)

Time Elapsed (avg)	Time Elapsed (std)	Block Size	Grid Size
52,3855	0,009832	1x1x1	1x1x1
17,3353	0,015816	8x8x1	1x1x1
13,7420	0,035561	16x16x1	1x1x1
29,3443	0,164334	1x1x1	2x2x1
9,7906	0,024872	8x8x1	2x2x1
8,6305	0,031485	16x16x1	2x2x1
16,8751	0,034781	1x1x1	4x4x1
11,6958	0,041159	8x8x1	4x4x1
18,4325	0,007812	16x16x1	4x4x1

5 Discussion

It should be noted that especially in the case of the double-buffered solution, memory transactions between the device and host are prominent and negatively affect performance.

(It is assumed that it has been confirmed that data transactions and reception takes a lot of resources through profiling.)
At first, delta sum was obtained at each iteration to compare the conditions in cpu, but it took a long time. So by changing to compare the conditions in the CPU taking the delta sum from the GPU every certain iterations, which means that reducing the expensive operation of transferring data(host to device, device to host) and repeating the operation of the threads in GPU, the time was greatly saved. Instead, the condition comparisons are performed every iteration intervals, so the exact number of repetitions cannot be obtained, and the number of repetitions can be checked in units of iteration intervals. If we schedule tasks in a way that makes the most of use the hiding latency and zero switching context, which are the characteristics of cuda, it is thought that we can use the fast performance of the GPU.
In GPU, understanding and utilizing the memory hierarchy also greatly affects performance. This time, it was not fully utilized, but in order to further improve performance, a work design taking this into account is necessary.
According to profiling data provided by NVIDIA Nsight, the threads spend the vast majority of their runtime waiting for other thread. Spesifically, for every second that has passed, approximately 900 milliseconds are spent busy-waiting or otherwise performing synchronisation operations. See figure 5 for a screenshot of the profiler showing the time spent syncronizing in light orange. Light blue is kernel calls, and is barely visible at this zoom level. This shows that there is room for improvement in the CUDA implementation. Interestingly, the time spent doing synchronisation operations according to the profiler seems to actually decrease when the number of CUDA cores increases. The reason behind this is unknown and at a glance defies logic. Overhead in any form of parallelizing is expected due to, among other things, having to send communication between threads and dealing with synchronisation. If the profiler is to be believed and its results are correctly interpreted, this is exactly not the fact in this case. It's therefore more likely that the way the profiling program visualises functions makes it seem that it's taking more time to synchronize in a single-core execution compared to an 8-core execution. Regardless, the results were interesing and worth discussing. As shown in experimental results in the Experiments section, for reasons not entirely clear through profiling, the simple solution in CUDA is overall the fastest solution of the three, with the double-buffered solution being the 2nd best and the red-black one being the 3rd best.
The variance between different hardware is also considerable. Preliminary tests were done on a variety of hardware, but results were held back due to the space it would occupy on paper. Results showed a semi-consistent increase in time efficiency with the number of CUDA cores available on the device, the core clock of the device and the architecture. However, this was not always the case, as a NVIDIA GeForce GTX 750

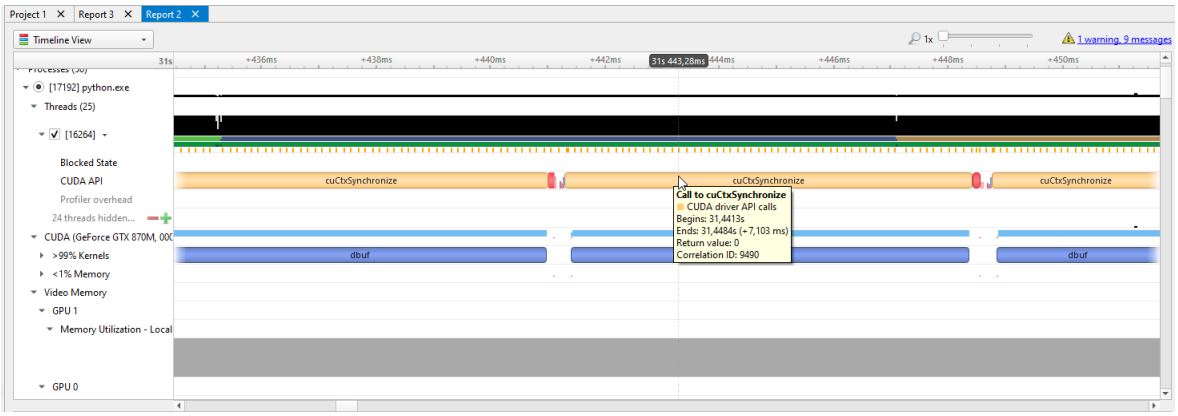

Figure 5: Nsight profiling of the CUDA double-buffered solution, single CUDA core

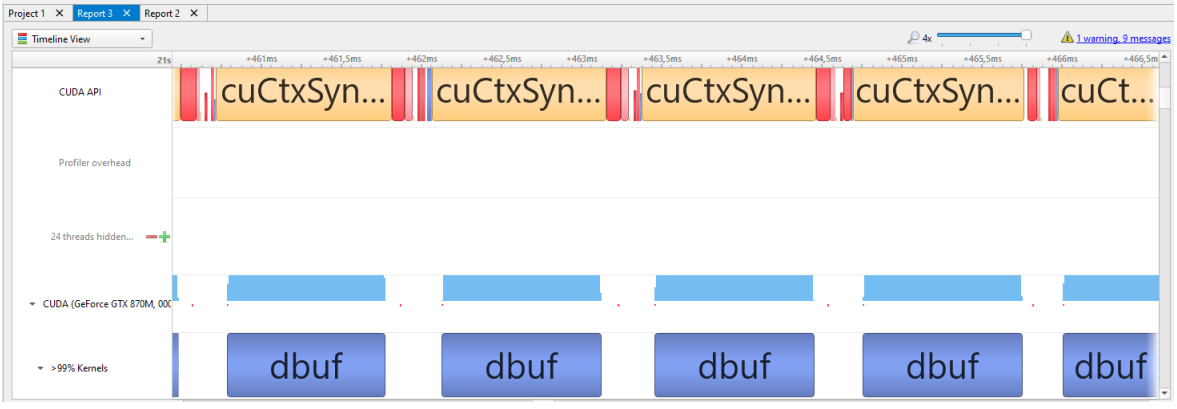

Figure 6: Nsight profiling of the CUDA double-buffered solution, multiple CUDA cores

Ti outperformed the reference NVIDIA GeForce RTX 2060 used in the results shown earlier, in the simple solution when set at 64x1x1 block size and 16x1x1 grid size. The 750 Ti got the solution solved in 0.02 seconds, while the RTX 2060 solved it in 2 seconds. The difference is a magnitude of 100x. The reason behind this is not clear either through interpreting profiling results or time results directly. One possible cause could be that the resources used to produce these results are shared between many users, and that many other users could also have chosen to use the one single RTX 2060 in the cluster environment, which admittedly stands out quite a bit when most of the cluster nodes are equipped with Quadro K620 GPUs.

6 Summary

The thermal distribution problem is a very good candidate for experimenting with parallelizing workloads, particularly highly parallel toolkits such as those provided by CUDA and OpenMP.

References

[1] N. C. Cyril Zeller. Cuda c/c++ basics supercomputing 2011 tutorial, 2011.

[2] NVIDIA. Grid of thread blocks, 2020.

[3] P. Pacheco. *An introduction to parallel programming.* Elsevier, 2011.