Technical University of Denmark

02613 - Python and High-Performance Computing

Group 45

Mini-Project: Wall Heating

April 30th - 2025

This Project is written by:

Kristine Rigmor Agergaard Andersen, s204608
Szymon Borzdyński s250372
Nojan Rezvani, s204426

Table of content

1 Tasks

1.1 1. Load and visualize the input data for floorplan:

In order to visualize the input data i used python's matplotlib library and heatmap. After running simulate_GRID.py on building 9991 chosen randomly as example to visualize i have received the following results. Temperature values are mapped to a color gradient using 'hot' option for cmap argument in Matplotlib. [2]

Script: Visual simulation/simulate_GRID.py

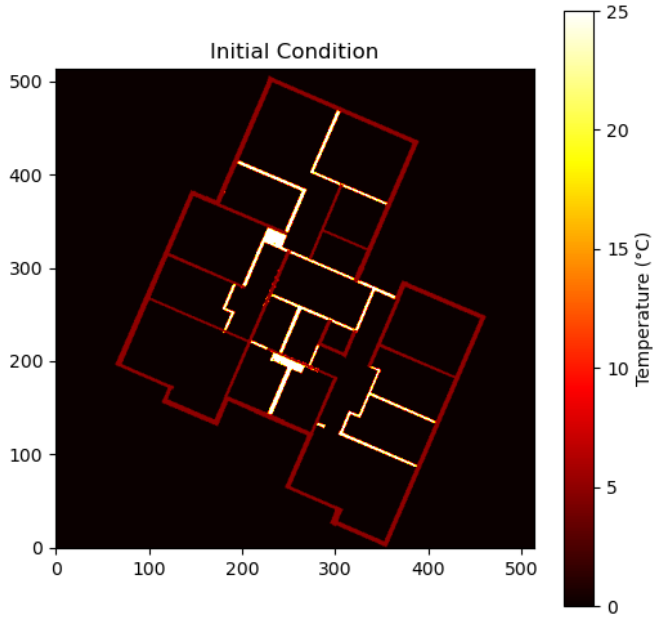


Figure 1: A visualization of building initial condition. The heated walls can be clearly identified

The initial temperature distribution (Figure 1) shows a predominantly uniform temperature across the building, with higher temperatures concentrated along the heated walls.

After running the Jacobi iteration method for steady-state heat diffusion, the final temperature distribution (Figure 2) shows where heat is concentrated.

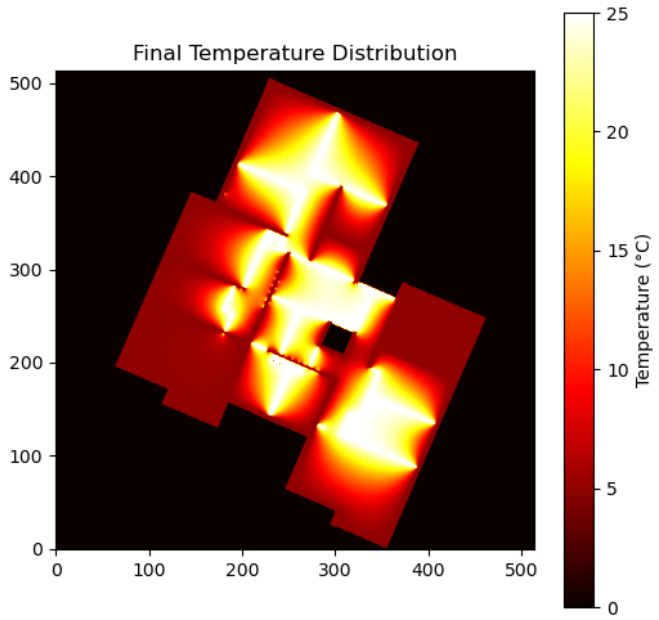


Figure 2: A visualization of building final temperature.

The visualization shows that heat propagates from high-temperature regions outward, with smoother transitions between different zones.

The walls significantly influence the temperature distribution, acting as barriers and causing localized heat accumulations.

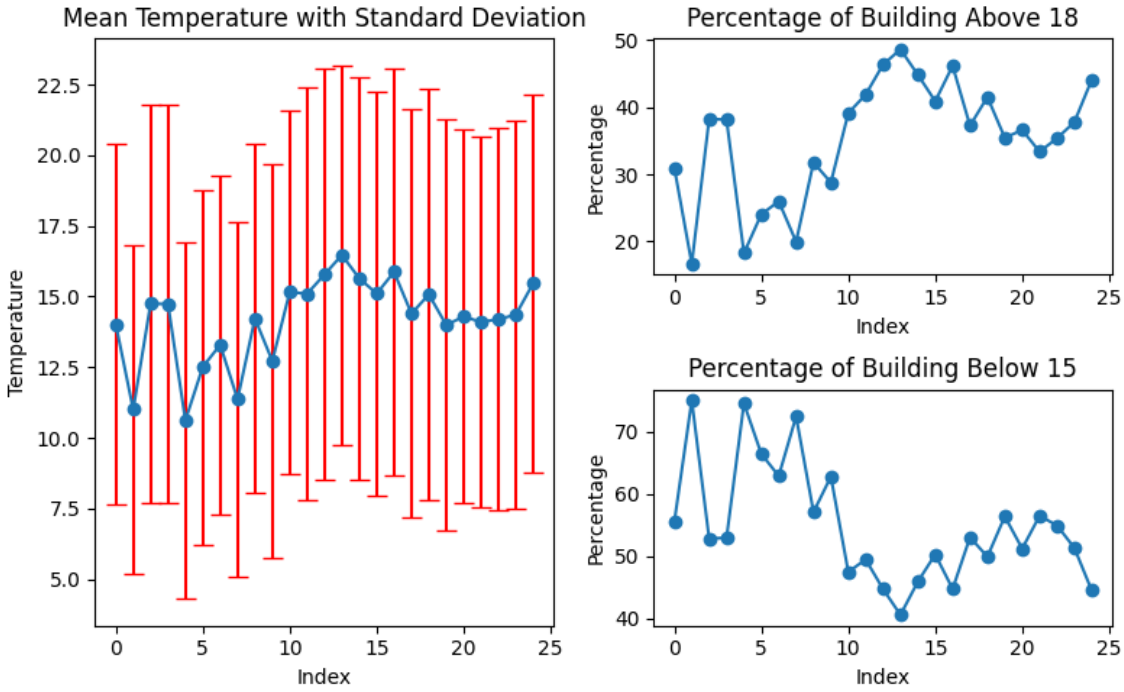
1.2 2. Run and time the reference implementation:

We chose to run and time the reference using the first 10 floorplans. The DTU HPC was utilized for this task; The job script can be seen in Appendix 2.2.

The cumulative time for the script came out to approximately 79 s. Using this as a baseline, the extrapolated time needed to go through the entire dataset, consisting of 4571 floorplans, comes out to 36110.9 s, or approximately 10 h.

1.3 3. Visualize the simulation results for a few floorplans

To visualize the results of a few floor plans, a small script plotting the results (”visualize_stats.py”) is added, giving us the below plots. This was run for 25 floor plans.



1.4 4. Profile and explain the jacobi function

To profile a specific function, we add a @Profile above it in the script, and switch out the python script command in appendix 2.2 with ”kernprof -l simulate.py 10”.

As it turns out, almost all of the computation time is spent inside the jacobi function - Everything else is negligible:

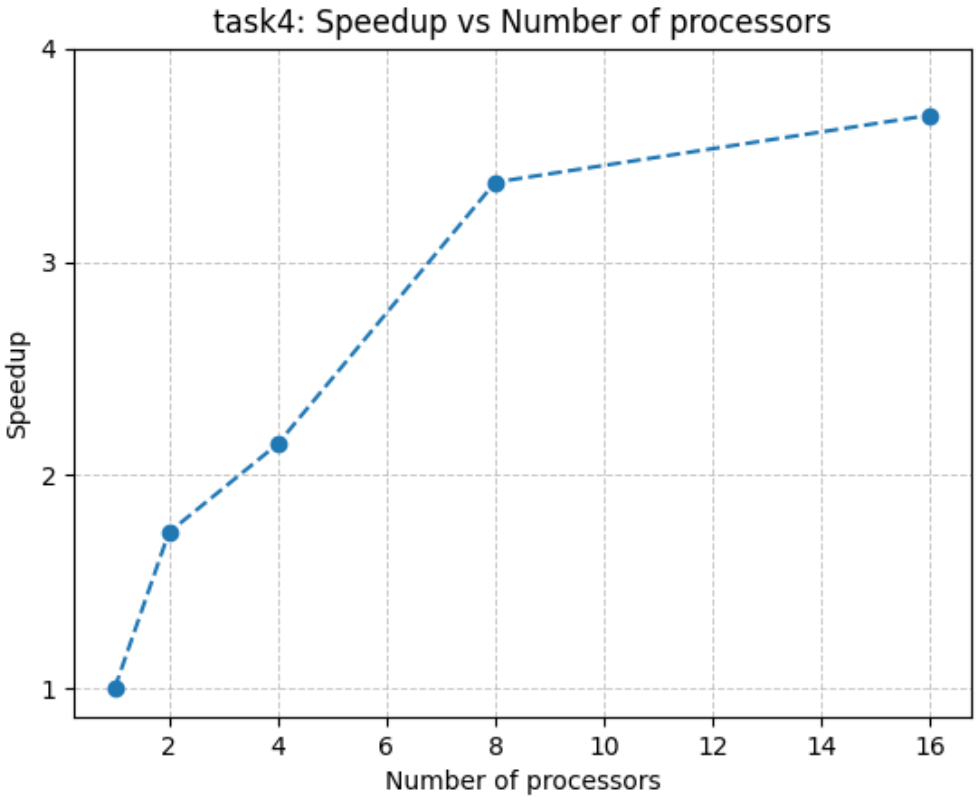
The jacobi function is used to calculate the steady-state heat distribution. For each iteration of the loop, we calculate the temperature of a point to be the average of its neighbours, as seen on line 22. Secondly, we make sure only interior grid points are updated, line 23, by using interior mask as a binary mask. Finally, we update the array that holds the temperatures for the grid points, array u, on line 25.
The loop continues until the solution has converged (The updated grid temperature difference no longer exceeds some threshold value), or until a set max number of iterations has been performed.

```
Total time: 82.098 s
File: simulate.py
Function: jacobi at line 16

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    16                                           #@profile
    17                                           def jacobi(u, interior_mask, max_iter, atol=1e-6):
    18        10       5300.2    530.0      0.0      u = np.copy(u)
    19
    20     47282      22400.4      0.5      0.0      for i in range(max_iter):
    21                                                   # Compute average of left, right, up and down neighbors, see eq. (1)
    22     47282   46718393.7    988.1     56.9          u_new = 0.25 * (u[1:-1, :-2] + u[1:-1, 2:] + u[:-2, 1:-1] + u[2:, 1:-1])
    23     47282    8932626.7    188.9     10.9          u_new_interior = u_new[interior_mask]
    24     47282   16511337.5    349.2     20.1          delta = np.abs(u[1:-1, 1:-1][interior_mask] - u_new_interior).max()
    25     47282    9856405.8    208.5     12.0          u[1:-1, 1:-1][interior_mask] = u_new_interior
    26
    27     47282      51509.5      1.1      0.1          if delta < atol:
    28        10          5.4      0.5      0.0              break
    29        10          2.7      0.3      0.0      return u

 82.10 seconds - simulate.py:16 - jacobi
```

Figure 3: task_4_profile Source: Kristine

1.5 5. Static scheduling parallelization

50 floorplans was used to test the scheduling. The script was then tested with n_cores = 1,2,4,8,16 and chunks evenly spread between the cores:



We observe a significant speedup, reaching approximately 3.7x improvement with 16 workers, compared to the fully serial version. To find the parallel fraction, we can use Amdahl's

law:

$$S(p) = \frac{1}{(1-F) + F/p}$$

Plugging our numbers and solving for F gives us $F \approx 0.79$, i.e. 79% of the workload is parallelized.

The theoretical speedup limit can then be found:

$$S(\infty) = \frac{1}{1-F}$$

$$S(\infty) = \frac{1}{1-0.79} = 4.76$$

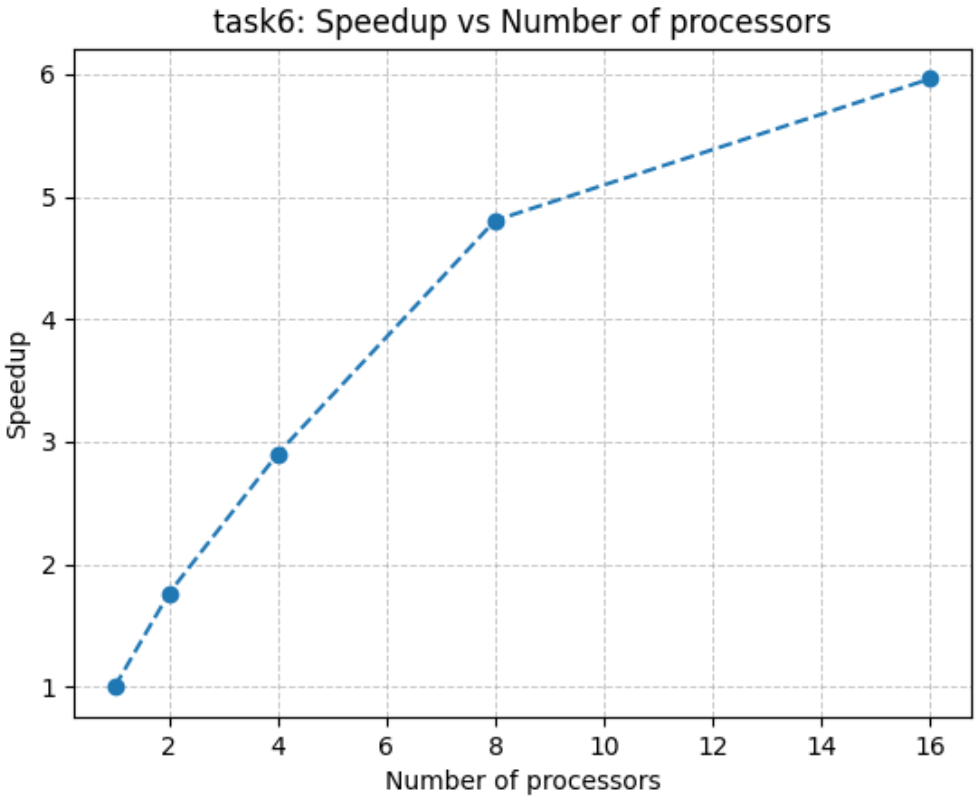
We only achieved 3.7x, with 16 cores. Rerunning the script with higher core counts would likely yield results closer to the limit.

If we use task 2's time serial time as a baseline for the total time needed to process all floorplans, we estimate that using 16 cores can reduce the total time to about 9759.7 s, or $\approx$ 2.71 hours.

simulate_parallel_static.py was used for this exercise.

1.6 6. Dynamic scheduling parallelization

The previous script and setup were reused, with the only difference being that chunksize was set to 1, allowing for dynamic scheduling:



The speedup improved to approximately 6x with 16 workers, compared to the fully serial version. Solving Amdahl's law for the new speedup gives us $F \approx 0.89$.
However, it should be noted that the dynamic scheduled version had a worse 1 core performance in absolute seconds, compared to the static scheduled version, which likely inflated the speedup. In seconds, the runtime, using 16 cores for both static and dynamic, was 151.85 and 101.35, respectively.

simulate_parallel_dynamic.py was used for this exercise.

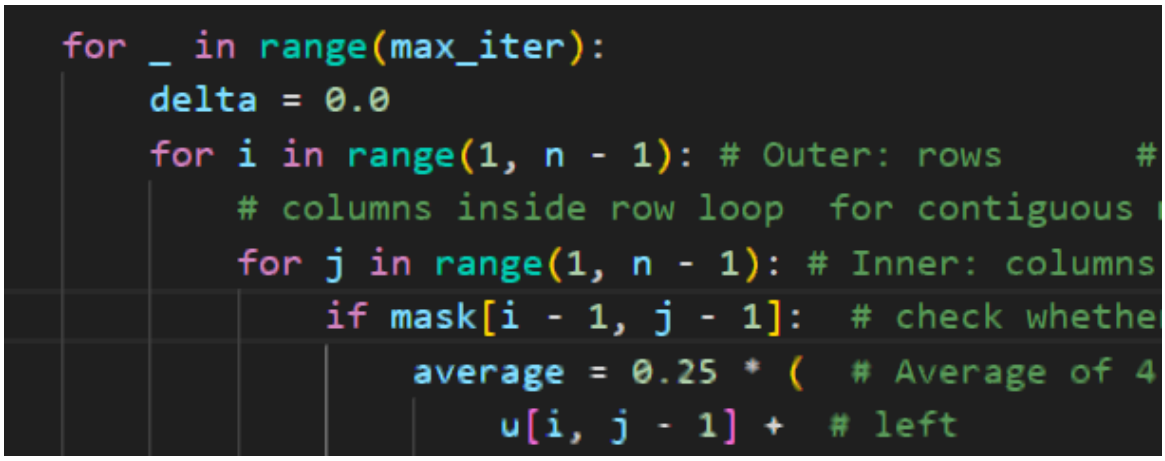
1.7 7. Numba JIT implementation

Numba implementation upon compilation turned out to be few orders of magnitude faster than reference script. The test was run for 10 floorplans as tests were run on local machine.

- Reference script - 79 seconds
- Numba (after compilation) - 18.2 seconds

Numba implementation was approx. 4.3 times faster.

1.7.1 Access pattern



The above part of Numba function uses 2 loops to iterate over elements in 2D array. The outer loop iterates over rows and inner over columns. By iterating this way, memory is accessed in the same sequence it's laid out, so the CPU cache prefetches and reuses loaded data which gives better performance.

To process all 4571 floor plans it would take approx. $4571 \div 10 = 457.1 457.1 \times 18.2 sec = 8226 sec = 137 minutes = 2h17 minutes$

Scripts: *JIT/jitversion.py, JIT/numpycompare.py*

1.8 8. CUDA kernel implementation

As per instruction, CUDA implementation was written some changes (no convergence check) and tested on GPU from HPC.

I rewrote the Jacobi iteration using a CUDA kernel in Numba.

The kernel (jacobi_kernel) updates u_new[x, y] using neighboring values (left, right, up, down), but calculates them explicitly based on x and y indices, since i cannot used vectorized operations like in NumPy in cuda kernel. Only interior points are computed and updated. A bounds check was added so threads don't go out of bounds of block.

The helper function (jacobi_helper) sets up the block sizes and threads, copies data to the GPU, and loops over max_iter iterations, swapping u and u_new at each step so new data is processed at each loop.

In main function all data is copied because exterior values not carry over later and device_array_like does not make copy of values.

The test was run for 10 floor-plans.

- Reference script - 79 seconds
- CUDA implementation - 11.8 seconds

The CUDA version is about 6.7 times faster.

The results of CUDA operations differ slightly from those on numpy: The same inputs will give the same results for individual IEEE 754 operations to a given precision on the CPU and GPU. As we have explained, there are many reasons why the same sequence of operations may not be performed on the CPU and GPU. The GPU has fused multiply-add while the CPU does not. Parallelizing algorithms may rearrange operations, yielding different numeric results. [3] [4]

To process all 4571 floor plans it would take approx. $4571 \div 10 = 457.1457.1 \times 11.8sec = 5393sec = 89minutes = 1h29minutes$

Script: *CUDA/cudaversion.py*

1.9 9. GPU using CuPy

To rewrite the reference script, the CuPy package was imported instead of the numpy package. Hereafter all the numpy functions were replaced with CuPy functions (np -¿ cp). The time it took to run on DTU's GPU was 24.834 s.

a - In task 2, we found that the run time for the reference was approx. 79 s. For the CuPy solution, we found the run time was 24.834 s. This means the CuPy solution is about 3.2 times faster.

b - Using the above time as baseline, and going through the whole dataset, consisting of 4571 floor plans, it would take about 11351.6 s or approx. 31.5 h.

c - Yes — although the CuPy implementation ran 3.2 times faster than the reference, this speedup is lower than expected. For GPU-accelerated array computations, speedups of 10× to 100× are commonly reported, depending on the problem size and structure. The low gain here may be due to limited parallelism in the task, overhead from data transfer between CPU and GPU, or a problem size too small to fully utilize GPU resources [1].

For this task, the script is called *CuPy/CuPy.py*

1.10 10. Profiling CuPy solution

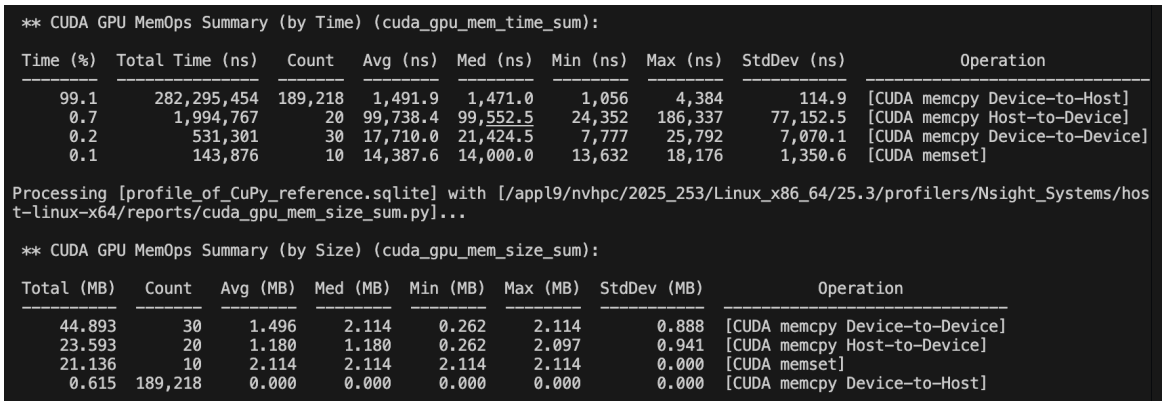
For this task, nsys profiler was used: *"nsys profile -o profile_of_CuPy_reference python CuPy_changed.py 10"*. The relevant output from the nsys profiler can be seen below.

** CUDA API Summary (cuda_api_sum):

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
55.3	4,965,688,477	898,568	5,526.2	5,457.0	3,789	1,305,664	3,628.9	cuLaunchKernel
32.4	2,914,461,569	189,268	15,398.6	15,047.0	6,790	61,776	1,817.7	cudaMemcpyAsync
5.8	518,232,631	94,604	5,477.9	5,674.0	3,830	1,172,253	4,088.8	cudaLaunchKernel
3.1	280,840,783	189,218	1,484.2	1,423.0	825	22,559	444.8	cudaStreamSynchronize
1.7	153,284,066	189,238	810.0	785.0	379	24,291	318.8	cudaStreamIsCapturing_v10000
1.5	137,821,887	10	13,782,188.7	156,604.5	64,932	136,285,793	43,043,473.4	cudaMalloc
0.1	7,265,320	24	302,721.7	248,820.0	99,721	1,686,449	373,458.5	cuModuleLoadData
0.0	2,125,375	2	1,062,687.5	1,062,687.5	1,030,499	1,094,876	45,521.4	cudaHostAlloc
0.0	1,700,261	10	170,026.1	62,698.0	50,208	612,204	219,485.8	cuModuleUnload
0.0	155,434	39	3,985.5	3,292.0	905	22,403	4,497.0	cudaEventQuery
0.0	148,061	10	14,806.1	10,627.5	8,905	40,972	10,003.1	cudaMemsetAsync
0.0	95,706	384	249.2	188.5	124	1,180	151.2	cuGetProcAddress
0.0	87,469	20	4,373.4	3,157.0	2,883	8,126	1,987.8	cudaEventRecord
0.0	63,000	20	3,150.0	2,213.5	1,774	9,380	1,984.4	cudaEventCreateWithFlags
0.0	52,982	1	52,982.0	52,982.0	52,982	52,982	0.0	cudaMemGetInfo
0.0	47,310	19	2,490.0	1,820.0	1,646	4,626	1,104.2	cudaEventDestroy
0.0	1,966	1	1,966.0	1,966.0	1,966	1,966	0.0	cuInit
0.0	1,027	2	513.5	513.5	168	859	488.6	cuModuleGetLoadingMode

** CUDA GPU Kernel Summary (cuda_gpu_kern_sum):

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
29.6	3,386,928,125	141,846	23,877.5	20,799.0	11,712	46,304	5,926.5	cupy_add__float64_float64_float64
28.7	3,285,328,372	283,712	11,579.8	16,720.5	3,104	29,792	8,025.5	cupy_scan_naive
11.1	1,264,539,847	94,574	13,370.9	12,000.0	10,432	19,616	2,239.7	cupy_getitem_mask
10.7	1,220,360,036	141,856	8,602.8	7,936.0	7,455	14,336	957.8	cupy_bsum_shfl
5.2	597,209,478	47,282	12,630.8	12,545.0	11,776	18,976	566.7	cupy_multiply__float_float64_float64
4.7	535,364,757	47,282	11,322.8	11,296.0	10,528	17,152	507.5	cupy_scatter_update_mask
2.5	288,862,000	47,282	6,109.3	6,112.0	4,736	19,745	704.6	void cub::CUB_200200_350_370_500_520_600_610_700_750_800_860_890_900_NS::DeviceReduceKernel<cub::CU…
2.5	281,656,703	47,282	5,957.0	6,304.0	3,488	13,536	1,061.9	cupy_subtract__float64_float64_float64
2.2	256,729,017	47,282	5,429.7	5,600.0	3,360	12,864	801.5	cupy_absolute__float64_float64
1.5	167,929,899	47,282	3,551.7	3,520.0	3,136	13,216	203.4	void cub::CUB_200200_350_370_500_520_600_610_700_750_800_860_890_900_NS::DeviceReduceSingleTileKern…
1.3	144,826,562	47,292	3,062.4	3,103.0	2,528	13,280	221.3	cupy_less__float64_float_bool
0.0	509,159	30	16,972.0	20,496.0	2,912	28,320	10,501.3	cupy_copy__float64_float64
0.0	260,483	10	26,048.3	23,968.5	12,736	36,928	8,145.9	cupy_var_core_float64
0.0	90,913	20	4,545.6	4,416.0	4,032	5,440	453.4	void cub::CUB_200200_350_370_500_520_600_610_700_750_800_860_890_900_NS::DeviceReduceKernel<cub::CU…
0.0	83,904	20	4,195.2	4,192.0	3,487	4,864	424.1	cupy_cub_sum_pass1
0.0	68,963	20	3,448.2	3,456.0	3,296	4,032	168.0	cupy_cub_sum_pass2
0.0	64,992	20	3,249.6	3,263.0	3,136	3,456	82.8	void cub::CUB_200200_350_370_500_520_600_610_700_750_800_860_890_900_NS::DeviceReduceSingleTileKern…
0.0	59,744	20	2,987.2	2,976.0	2,880	3,264	89.5	cupy_true_divide__float64_float_float64
0.0	59,424	20	2,971.2	2,960.0	2,912	3,201	58.3	cupy_true_divide__int64_int_float64
0.0	58,306	20	2,915.3	2,912.0	2,880	2,976	25.2	cupy_multiply__float64_float_float64
0.0	49,473	10	4,947.3	4,768.0	3,712	5,952	733.9	cupy_greater__float64_float_bool
0.0	30,144	10	3,014.4	2,976.0	2,944	3,392	133.1	cupy_sqrt__float64_float64

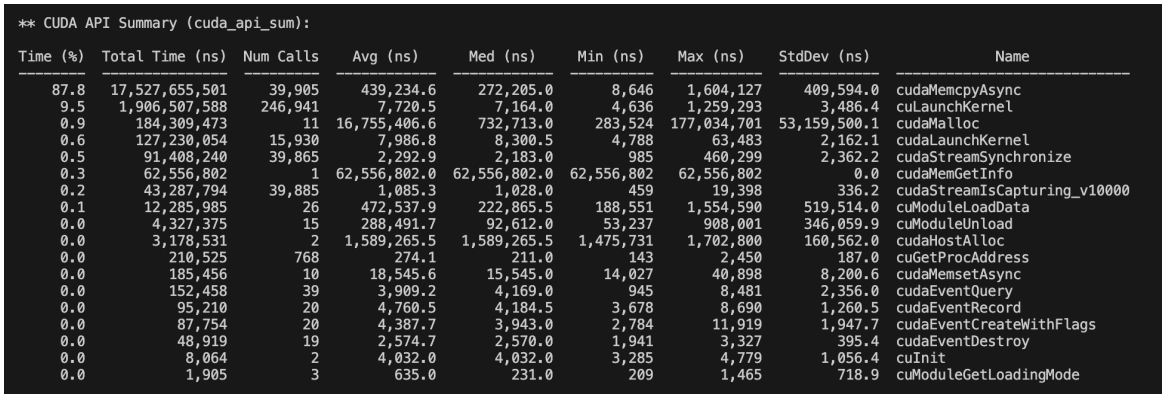


```
** CUDA GPU MemOps Summary (by Time) (cuda_gpu_mem_time_sum):

 Time (%)  Total Time (ns)   Count   Avg (ns)  Med (ns)  Min (ns)  Max (ns)  StdDev (ns)           Operation
 --------  ---------------  -------  --------  --------  --------  --------  -----------  ----------------------------
     99.1      282,295,454  189,218   1,491.9   1,471.0     1,056     4,384        114.9  [CUDA memcpy Device-to-Host]
      0.7        1,994,767       20  99,738.4  99,552.5    24,352   186,337     77,152.5  [CUDA memcpy Host-to-Device]
      0.2          531,301       30  17,710.0  21,424.5     7,777    25,792      7,070.1  [CUDA memcpy Device-to-Device]
      0.1          143,876       10  14,387.6  14,000.0    13,632    18,176      1,350.6  [CUDA memset]

Processing [profile_of_CuPy_reference.sqlite] with [/appl9/nvhpc/2025_253/Linux_x86_64/25.3/profilers/Nsight_Systems/hos
t-linux-x64/reports/cuda_gpu_mem_size_sum.py]...

 ** CUDA GPU MemOps Summary (by Size) (cuda_gpu_mem_size_sum):

 Total (MB)   Count   Avg (MB)  Med (MB)  Min (MB)  Max (MB)  StdDev (MB)           Operation
 ----------  -------  --------  --------  --------  --------  -----------  ----------------------------
     44.893       30     1.496     2.114     0.262     2.114        0.888  [CUDA memcpy Device-to-Device]
     23.593       20     1.180     1.180     0.262     2.097        0.941  [CUDA memcpy Host-to-Device]
     21.136       10     2.114     2.114     2.114     2.114        0.000  [CUDA memset]
      0.615  189,218     0.000     0.000     0.000     0.000        0.000  [CUDA memcpy Device-to-Host]
```

We can see from the first picture that it takes more than half of the time to launch kernels. This is because it launches a kernel for every calculation.

After batching, we get the following, where we can see the kernel launch went from 4,965,688,477 ns (4.965688 s) to 1,906,507,588 ns (1.906508 s).

For this task, the script is called *CuPy/CuPy_changed.py*



```
** CUDA API Summary (cuda_api_sum):

Time (%)  Total Time (ns)  Num Calls    Avg (ns)      Med (ns)     Min (ns)    Max (ns)    StdDev (ns)             Name
--------  ---------------  ---------  ------------  ------------  ----------  ----------  -----------  ----------------------------
    87.8   17,527,655,501     39,905     439,234.6     272,205.0       8,646   1,604,127     409,594.0  cudaMemcpyAsync
     9.5    1,906,507,588    246,941       7,720.5       7,164.0       4,636   1,259,293       3,486.4  cuLaunchKernel
     0.9      184,309,473         11  16,755,406.6     732,713.0     283,524 177,034,701  53,159,500.1  cudaMalloc
     0.6      127,230,054     15,930       7,986.8       8,300.5       4,788      63,483       2,162.1  cudaLaunchKernel
     0.5       91,408,240     39,865       2,292.9       2,183.0         985     460,299       2,362.2  cudaStreamSynchronize
     0.3       62,556,802          1  62,556,802.0  62,556,802.0  62,556,802  62,556,802           0.0  cudaMemGetInfo
     0.2       43,287,794     39,885       1,085.3       1,028.0         459      19,398         336.2  cudaStreamIsCapturing_v10000
     0.1       12,285,985         26     472,537.9     222,865.5     188,551   1,554,590     519,514.0  cuModuleLoadData
     0.0        4,327,375         15     288,491.7      92,612.0      53,237     908,001     346,059.9  cuModuleUnload
     0.0        3,178,531          2   1,589,265.5   1,589,265.5   1,475,731   1,702,800     160,562.0  cudaHostAlloc
     0.0          210,525        768         274.1         211.0         143       2,450         187.0  cuGetProcAddress
     0.0          185,456         10      18,545.6      15,545.0      14,027      40,898       8,200.6  cudaMemsetAsync
     0.0          152,458         39       3,909.2       4,169.0         945       8,481       2,356.0  cudaEventQuery
     0.0           95,210         20       4,760.5       4,184.5       3,678       8,690       1,260.5  cudaEventRecord
     0.0           87,754         20       4,387.7       3,943.0       2,784      11,919       1,947.7  cudaEventCreateWithFlags
     0.0           48,919         19       2,574.7       2,570.0       1,941       3,327         395.4  cudaEventDestroy
     0.0            8,064          2       4,032.0       4,032.0       3,285       4,779       1,056.4  cuInit
     0.0            1,905          3         635.0         231.0         209       1,465         718.9  cuModuleGetLoadingMode
```

1.11 11. Improve the performance (Optional task)

I have chosen to speed up my JIT solution by using multiprocessing module. I have created a process pool of size 16 (To match core count of XeonGold6226R). Each process in pool would be given building to process in parallel unlike previous JIT implantation which would process buildings one by one. After finishing its building, process would take another one to compute. Tests were done for 10, 32, 64, 128, 256 buildings . Times were as follows:

Implementation	**Buildings**	**Total Time [sec]**
Reference Script	N/A	79
Normal JIT (after compilation)	10	18.2
Improved JIT	10	4.01
Improved JIT	32	10.5
Improved JIT	64	16.77
Improved JIT	128	41.55
Improved JIT	256	94.63

Table 1: Execution time comparison of different JIT implementations and building counts.

As we can see Improved JIT implementation is 4.5 times faster then previous JIT implementation from task 7. Moreover for higher amount of buildings times per building are improved.

Implementation	**Buildings**	**Time per Building [sec]**
Normal JIT (after compilation)	10	1.82
Improved JIT	10	0.40
Improved JIT	32	0.328
Improved JIT	64	0.26
Improved JIT	128	0.324
Improved JIT	256	0.36

Table 2: As above but, average time per building

With more buildings however, we can see performance gains become lower due to overhead of process pool and many queue delays. It would be possible to further improve performance by processing a batch of buildings per process (instead of one at a time per process) at once to reduce overhead.

Script: JIT/jitversion_parallel.py

1.12 12. Processing of all floor plans

In order to process all building CUDA implementation was chosen as it was the fastest. (Improved JIT was created after this task was completed)

Information collected according to questions in instruction:

- Average mean temperature: 14.71 C
- Standard deviation of mean temperatures: 2.1454
- Buildings with at least 50% of area with mean temperature above 18°C: 813
- Buildings with at least 50% of area with mean temperature below 15°C: 2460

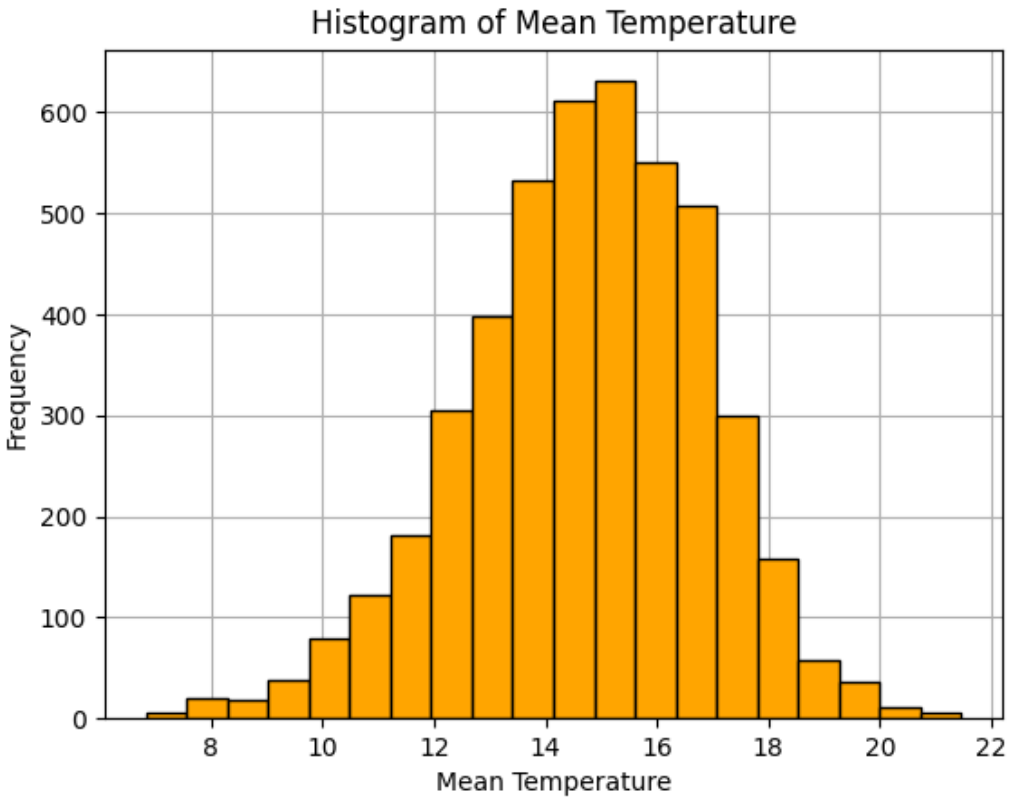


Figure 4: Histogram of mean temperature. Source: Szymon Borzdyński

Script: *CUDA/analysis.py*

2 Appendix

2.1 Table of contribution

Section	S204608	s250372	S204426
Task 1		X	
Task 2			X
Task 3	X		
Task 4			X
Task 5	X		X
Task 6			X
Task 7		X	
Task 8		X	
Task 9	X		
Task 10	X		
Task 11		X	
Task 12		X	

2.2 Task 2 job script

```
#!/bin/bash
#BSUB -J project
#BSUB -q hpc
#BSUB -W 10
#BSUB -R "rusage[mem=4GB]"
#BSUB -R "select[model == XeonGold6226R]"
#BSUB -n 1
#BSUB -R "span[hosts=1]"
#BSUB -o project_%J.out
#BSUB -e project_%J.err

#InitializePythonenvironment
source /dtu/projects/02613_2025/conda/conda_init.sh
conda activate 02613

# Run Python script
#python -m cProfile -o script.prof program.py input.csv
#kernprof -l simulate.py 10
python -m cProfile -s cumulative simulate.py 10
```

2.3 Task 4 jacobi profile

```
Total time: 82.098 s
File: simulate.py
Function: jacobi at line 16

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    16                                           #@profile
    17                                           def jacobi(u, interior_mask, max_iter, atol=1e-6):
    18        10       5300.2    530.0      0.0      u = np.copy(u)
    19
    20     47282      22400.4      0.5      0.0      for i in range(max_iter):
    21                                                   # Compute average of left, right, up and down neighbors, see eq. (1)
    22     47282   46718393.7    988.1     56.9          u_new = 0.25 * (u[1:-1, :-2] + u[1:-1, 2:] + u[:-2, 1:-1] + u[2:, 1:-1])
    23     47282    8932626.7    188.9     10.9          u_new_interior = u_new[interior_mask]
    24     47282   16511337.5    349.2     20.1          delta = np.abs(u[1:-1, 1:-1][interior_mask] - u_new_interior).max()
    25     47282    9856405.8    208.5     12.0          u[1:-1, 1:-1][interior_mask] = u_new_interior
    26
    27     47282      51509.5      1.1      0.1          if delta < atol:
    28        10          5.4      0.5      0.0              break
    29        10          2.7      0.3      0.0      return u

 82.10 seconds - simulate.py:16 - jacobi
```

References

[1] CuPy. “CuPy documentation”. In: (2025). URL: https://docs.cupy.dev/en/stable/user_guide/performance.html.

[2] Matplotlib. *Choosing Colormaps in Matplotlib*. 2025. URL: https://matplotlib.org/stable/users/explain/colors/colormaps.html.

[3] NVIDA. “CUDA documentation”. In: (2024). URL: https://docs.nvidia.com/cuda/floating-point/index.html.

[4] Nicholas Vilt. “CUDA GPU vs CPU precision”. In: (2013). URL: https://www.cudahandbook.com/2013/08/floating-point-cpu-and-gpu-differences/.