# MINI-PROJECT NOTES

## Exercise 5 & 6:

Using starmap (locks in order)

A graph with colored lines

AI-generated content may be incorrect.

To explain this image: image generated with starmap, the chunksize changes from 100 images down to 50, 33, 25, 20. When increasing the number of processors, this chunk number is split by the number of processors/workers. So eventually, the last run of all of them, spawns 20 workers with a chunksize of 20, each taking a load of a single image to process. This methodology is static for the case where chunksize = number\_images/n\_proc and dynamic where chunksize < number\_images/n\_proc as this way you ensure to have more chunks than workers, letting workers take work as soon as they are done. However with starmap, this did not make sense as seen in the picture above as it is a locking process, waiting to process chunks in order. As image processing have difference convergence times, cycling through the load does not see a positive effect. When having bigger pools to work through, the short convergence images, even out the timing with the ones that take a longer time to converge. However, as you shrink the chunksize, the timings of the pool of data worked through have a higher standard deviation therefore creating batches where the images that take long time to process are not offset by the ones that converge super fast and vice-versa.

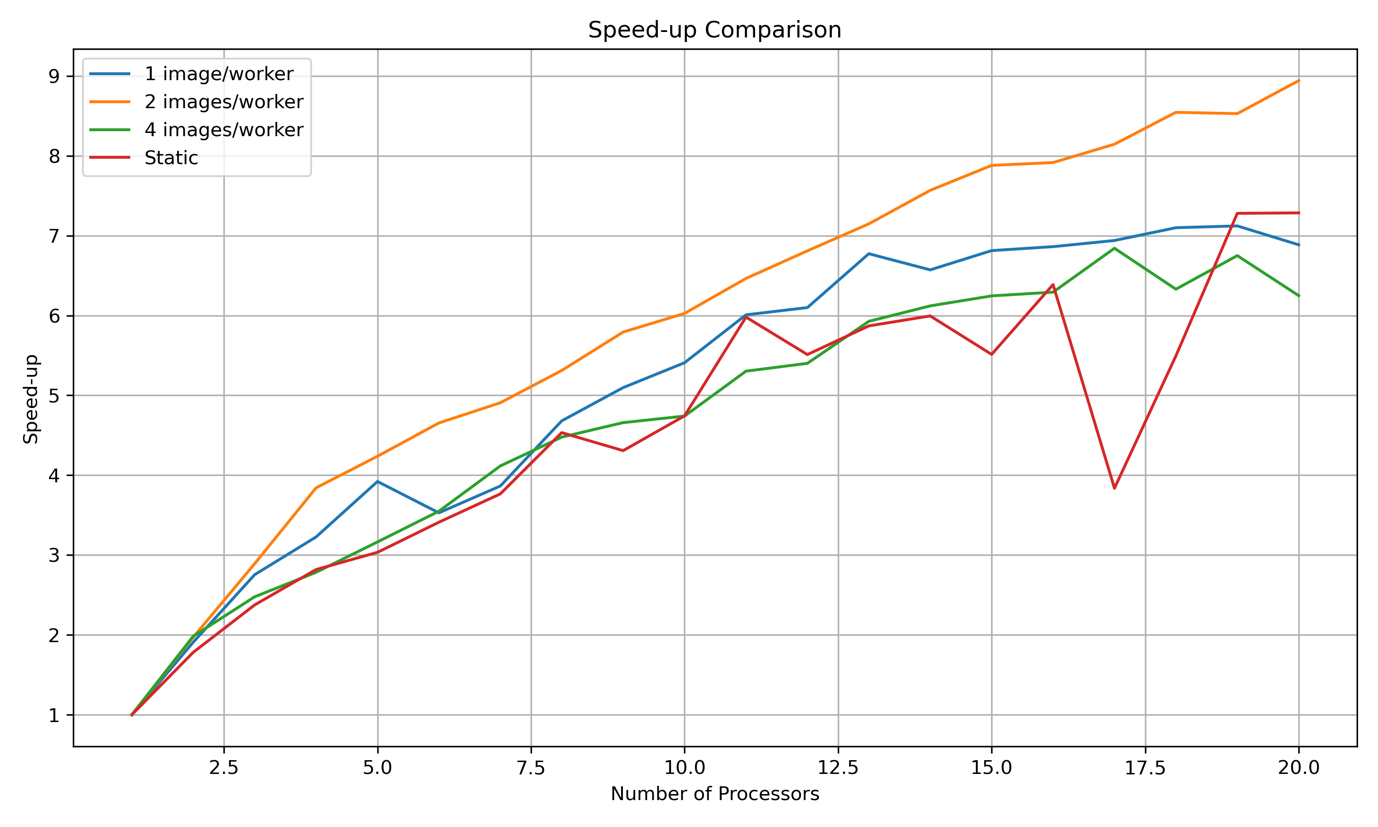
|  |  |  |
| --- | --- | --- |
| A graph with a red line  AI-generated content may be incorrect. | A graph with a red line  AI-generated content may be incorrect. | A graph with a red line  AI-generated content may be incorrect. |
| A graph with a red line  AI-generated content may be incorrect. |  | A graph with a red line and a graph  AI-generated content may be incorrect. |

Take these images above with caution, as these are visual representations of random samples that were chunked together for processing measuring the number of iterations needed for convergence. As seen in the first graph, all the serration and inconsistency in the processing can be explained by the variation of the data and how smaller chunks can fall into having a higher mean processing time. The full case (100 image) scenario has a mean number of iterations between 6 and 8 thousand. That can be seen in the first 4 pictures meaning the chunked solution average time is very likely to remain constant across the workers, so locking would not affect performance by a lot. However, in the last 2 cases for smaller chunksizes or the equivalent envelope between 10 and 20 workers (for the blue line scenario) we start seeing serration, as the mean processing time is more likely to be higher than 8000 iterations creating locking ineficciencies.

Note: there will be cases where chunks despite being big or small that the mean iteration count will be lower than the total mean (i.e below 6000) as these images are based on random sampling. These are just used to represent the true locking sccenarios that happen with smaller chunksizes, which are the ones MORE LIKELY to cause issues.

Using asynchronous processing (apply\_async)

Now, the solver was slightly modified to capture a range of asynchronous cases. There are 2 key components for optimization here, overhead and “chunking”. Usually, the smaller the chunks processed per worker the “quicker” the parallelized computation but memory overhead takes a toll in order to synchronize the results. Starting from static, where each worker runs its workset once as it processes the number of images divided by the number of processors. A similar case to the one found in the locking scenario is posed. When running every subworker once, the asynchronous effect loses its purpose. Every worker has now been assigned a slice of work that will very likely overtress some workers and leave others idle as the average amount of iterations per chunk changes. So that is why static loading performs the worst. The trade-off comes when finding the sweetspot between the maximum amount of splits where you assign 1 image to every worker or find some optimized blend assigning a few images to each worker to optimize for intercommunication, making the overall computation even faster.



Exercise 7

I used both jit and njit to see if there was a difference as njit caches the compiled version to disk. No big differences were seen so apparently jit works in the HPC saving the compiled version to memory. These are the timings seen when processing for 100 images:

|  |  |  |
| --- | --- | --- |
| Standard script (original without changes) | Jit compiled script (optimized for jit) | Standard script but optimized for jit (no compiling) |
| 25 minutes 25 seconds | 6 minutes 37 seconds | 3 hours + |

Exercise 8

Processes 100 images at 7K iter in 17 secs, there are 4571 images. All of them can be processed in 13 minutes. Simply amazing ☺

Exercise 9

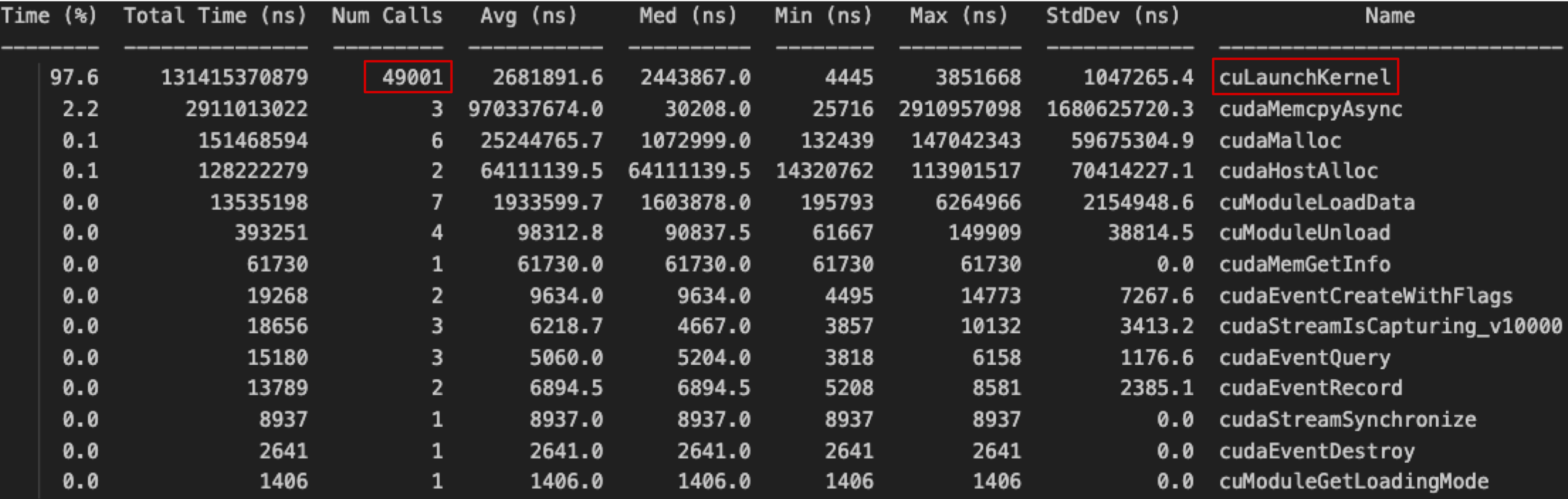
Processes 100 images at 7K iter in 2 minutes and 20 seconds, there are 4571 images. All of them would be processed in 1 hour and 47 minutes.

It was quite surprising to see that Cupy that apparently is a GPU optimized mathematical indexing tool performs worse than the version that uses numpy.

Exercise 10

This is the standard cupy implementation the one presented in exercise 9, and as we can see a lot of kernels are being launched for the loop operation in the jacobi\_gpu function. This function slices multiplies and most importantly reads from memory a few times to apply the mask and do the mathematical operation, increasing the overhead. There is a cupy decorator called fuse, whereby we tag a function that collects the jacobi operation in one line of code for which fuse forces the use of a single kernel, and pass this “fused” function in the iteration loop.

This out file is: source\_MMB/ex9/profiling/cupy7000\_100im\_24781931.out



And here the improved version, with a reduction by 3 times the amount of kernels, each kernel does more work, optimizing its use and reducing overhead.

The out file is: source\_MMB/ex9/profiling/cupy7000\_100im\_24782311.out

A screenshot of a computer screen

AI-generated content may be incorrect.

And here is the total timing for both:

Standard 🡪 2 minutes 20 seconds

Fused 🡪 38 seconds (more than 3.5 times faster!)

Exercise 12

The image processor process all 4571 images in 3 minutes and 2 seconds, timing was even better for this higher workload than it was for the smaller workload to scale with 100 images. The out file was “cuda7000\_all\_im\_24773825”

Wow!! ☺

RESULTS:

Average Mean Temperature: 14.59 °C

Average Temperature Standard Deviation: 6.82

Number of buildings with ≥50% area above 18ºC: 742

Number of buildings with ≥50% area below 15ºC: 2552

A graph with a red line

AI-generated content may be incorrect.