Parallel Computing Assignment 3

Daan Smeets [redacted]
Daniel Schenk
Anaïs Otting
Sofie Vos

We ran all our tests on the slurm22 server with the following specifications:

CPU: 2x AMD EPYC 7313 16 core
Frequency: 3.0Ghz Base, 3.7Ghz Boost
Cache: 2x 128 MB L3, 2x 1 MB L2, 2x 32 KB L1 (204.8 GB/s bandwidth)
Memory: 128 GB @3200Mhz 8-channel
Peak performance: 1536 Gflops/s
GPU: Nvidia A30 Tensor Core GPU with 3804 Streaming processors (CUDA Cores)

Performance comparison

Tabel 1 : Overall findings openCL version

n	workgroup size	total time spend in 300 kernel executions (msec)	total time spend in 2 host 2 device transfers (msec)	total time spend in 2 device 2 host transfers (msec)	CPU time spend (msec)	kernel equivalent on host (msec)	total time (s) min, average, max (5 runs)	Gflops/s min, avg, max (5 runs)
10240	32	2.5	0.1	0.0	189.19	0.0034		
10240	64	2.4	0.1	0.0	274.24	0.0042		
10240	128	2.4	0.1	0.0	234.91	0.0036		
10240	256	2.4	0.1	0.0	244.73	0.0033	0,180 | 0,183 | 0,185	0,055 | 0,073 | 0,081
10240	512	2.9	0.1	0.0	270.30	0.0033		
10240	1024	2.4	0.1	0.0	246.07	0.0038		
102400	32	3.4	0.2	0.2	190.38	0.038		
102400	64	2.9	0.2	0.2	187.24	0.033		
102400	128	2.7	0.2	0.2	195.44	0.036		
102400	256	2.6	0.2	0.2	187.21	0.036	0,178 | 0,178 | 0,179	0,854 | 0,857 | 0,859
102400	512	3.0	0.2	0.2	225.45	0.035		
102400	1024	2.6	0.2	0.2	190.28	0.036		

1024000	32	13.2	1.4	3.2	211.84	0.38		
1024000	64	6.9	1.4	3.1	191.34	0.37		
1024000	128	4.6	1.4	3.1	194.33	0.38		
1024000	256	4.3	1.4	3.1	193.13	0.36	0,181 | 0,185 | 0,187	8,160 | 8,285 | 8,481
1024000	512	5.0	1.4	3.0	229.67	0.40		
1024000	1024	4.5	1.4	3.1	194.96	0.36		
10240000	32	135.2	11.7	28.4	394.53	3.67		
10240000	64	50.0	11.6	27.5	271.44	3.61		
10240000	128	33.8	11.6	27.1	257.37	3.64		
10240000	256	33.9	11.6	27.4	254.56	3.51	0,250 | 0,254 | 0,262	58,625 | 60,502 | 61,440
10240000	512	36.9	11.6	26.2	279.14	3.60		
10240000	1024	36.4	11.9	28.1	264.18	3.71		
102400000	32	1055.8	115.1	259.3	1622.46	36.84		
102400000	64	472.6	113.9	256.4	1022.28	34.59		
102400000	128	315.1	113.9	265.4	880.91	36.08		
102400000	256	315.2	113.7	259.7	872.21	36.99	0,897 | 0,898 | 0,898	171,047 | 171,111 | 171,238
102400000	512	327.7	113.9	257.6	905.93	36.75		
102400000	1024	339.4	114.3	258.4	902.77	36.80		

Tabel 2: Findings sequential version

n	Duration (s), average 5 runs	Gflops/s, average 5 runs
10240	0.00040	39
102400	0.0045	35
1024000	0.052	30
10240000	1.6	9.7
102400000	15	10

Workgroups function by taking a chunk of the total stencil. If *n = 1024* and *workgroup = 32*, then each workgroup computes a 32x32 chunk of the 1024x1024 stencil. For our largest workable *n = 102.400.000*, this means that the optimal workgroup size would be 2^14, since the slurm GPU has 3804 cuda cores. This is the largest workgroup size 2^n which divides our n. However, NVIDIA has a limit on a workgroup size of 1024, meaning that after 1024*3804 = 3.895.296 (as an upper-bound) the speedup should not increase anymore. However, these calculations do not take into account other services running on the GPU, or transfer times. This is only a rough calculation of the optimal workgroup size.

We calculated the total and Gflops/s for a workgroup size of 256, as this was always (close to) the fastest runtime. The following formula is used to calculate the number of Gflops/s: *Gflops/s = ((iterations * (n - 2) * 5)) / 1e9 / duration (in sec)*, where *(iterations * (n - 2) * 5)* is used because of 3 floating-point multiplications and 2 floating-point additions being done for each element per iteration. For consistency, we used 300 iterations per execution.

Speedup

These are the (average) speedups of workgroup size 256 for different *n*'s with *n* work items.

Tabel 3: Speedup

n	speedup
10240	0,0022
102400	0,025
1024000	0,28
10240000	6,26
102400000	17,0

The reason for no speedup around *n = 1.024.000* is because sending and receiving between the host and the device takes up time, in which case the sequential version quickens.

Efficiency

The device (GPU) that does the computations for the heat diffusion uses 3804 cores

Tabel 4: Efficiency

n	speedup	efficiency
10240	0,0022	0,00000058%
102400	0,025	0,0000066%
1024000	0,28	0,000074%
10240000	6,26	0,0016%
102400000	17,0	0,0044%

Up to and including *n = 1024000*, the efficiency is probably bad due to no speedup.
For the other two values, the efficiency is higher because of a higher speedup, but still quite low because the CPU takes a long time setting up the kernel and transferring the data.

Weak Scaling

For there to be weak scaling, the efficiency needs to have the same linear growth as both n and the work items both grow equally. The work items in our project are always equal to n. Thus, we know that they have the same growth (lower value times 10).

Between *n = 10240* and *n = 102400*, we have weak scaling. Between *n = 102400* and *n = 1024000*, we lose the weak scaling. And we have less weak scaling as n grows.

The reason for there being less weak scaling as n grows larger, is because then the weak scaling depends more on the efficiency of the CPU then the GPU.

Strong scaling

To analyze the strong scaling we examine how the total time spent in 300 kernel executions changes as the workgroup size varies for a constant *n*. We choose *n = 102400000*, as our largest *n* has the best measured times.
Between the workgroup size equal to 32 and 64, we have strong scaling. As the total time spent in 300 kernel executions decreases, the workgroup size increases by an equal amount.
For the other workgroups we do not have strong scaling. As the time spent in 300 kernel executions is not decreased by an equal amount compared to the workgroup size. This is likely because the GPU then has a sufficient number of work items available for the computation at every new iteration.

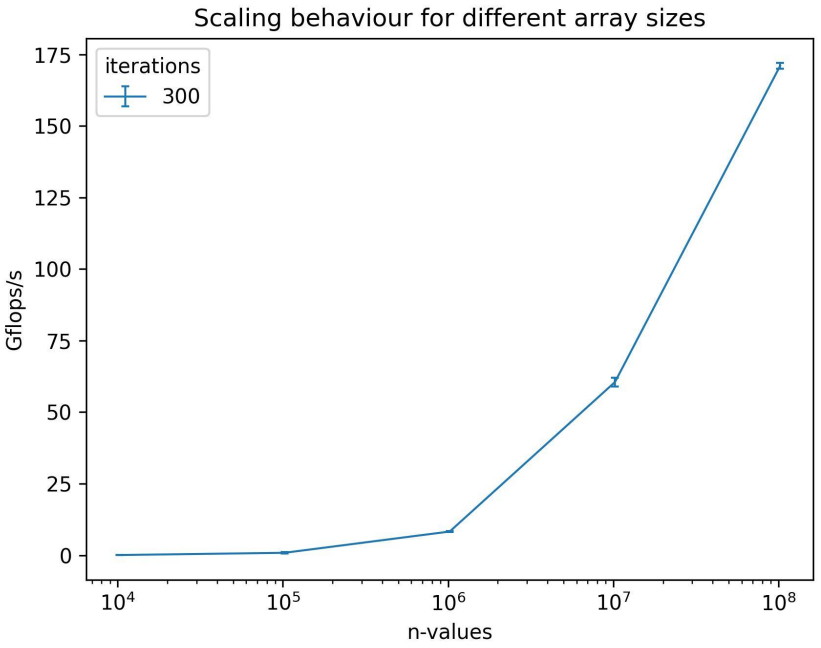
Graphs

Below is a graph to show the total Gflops/s. To create this graph we adjusted a program originally created by student Malte Wiethoff, a friend of ours who follows the data tracks and knows how to use pandas.

Unfortunately, the graph does not show a roofline. The reason for this appeared to be in the limited memory size of slurm. When we ran the program with values larger than *n = 102.400.000*, the program could not finish the computation. In contrast to the previous assignments, where the limit was the runtime, the larger computations should not have taken more than a few seconds.

Fortunately however, a clear increasing efficiency is visible in the graph. The scaling appears to be exponentially increasing, at least up to *n = 102.400.000*. As explained at the beginning of this report, there is an upper bound for speedup because of the number of cuda cores. With a workgroup size of 256, this limit is *256 * 3804 = 973.824*. As is clear from the higher *n*, we went over this number and there is still an efficiency increase. This can be explained through our (relatively) high CPU time. The values for *n* need to be placed into memory by the CPU. This takes a large majority of the time, regardless of the efficiency. With large numbers for *n*, this process becomes more efficient by itself.

To truly test the speedup, tests with much higher *n*'s need to be performed. In theory, this should then start to form a roofline that varies for different workgroup sizes, depending on the percentage of cuda cores used.



Newbie advice

OpenCL

OpenCL has a steeper learning curve compared to other parallel computing technologies. It requires an already reasonable understanding of parallel programming concepts, since openCL is a low-level programming framework. Developers have more control but also more responsibility for managing memory, synchronization, and optimizing code. Therefore, it is important to understand OpenCL optimization techniques like work-group organization, memory access patterns, and data collecting to maximize performance. As a result, developing efficient OpenCL code can be challenging for beginners(/newbies).

OpenMP vs MPI vs openCL

The choice of which technology to use depends on the specific requirements of your application and the hardware architecture available. Firstly, OpenMP is suitable for shared memory parallelism on multi-core CPUs. Secondly, MPI is designed for distributed memory parallelism across multiple computing nodes. Lastly, OpenCL enables parallel computing across different types of devices. Thus, if your program requires parallel processing across different devices such as GPUs and CPUs, OpenCL is the best choice.

What hardware to buy and what technologies to use

OpenCL is specifically designed for parallel computing using GPUs. Therefore, when using this, it is important to have a GPU with good compute performance and memory capacity. The number of cores, memory bandwidth, and compatibility with OpenCL should also be considered. While GPUs are mostly designed for parallel computations, CPUs also play a crucial role, especially for tasks requiring complex control flow or memory management. In this case the best CPUs to use have multiple cores, high clock speeds, and good cache performance. Development environments that support OpenCL are necessary as well. There are a few available, such as the Intel OpenCL SDK, AMD APP SDK, or NVIDIA CUDA Toolkit.

Our progress: First Attempt

The first attempt started by fully utilizing the functions provided in the *simple.c* and *simple.h* files. This resulted in a main loop, in which we created (and closed) a new kernel in every iteration. Additionally, we did have to clear the memory of the kernel after every iteration, which did include a small change in the *simple* files. These files are included in the *First_Attempt.zip*. Initially we were glad that we got a working version, however the speed drastically underperformed from what we were expecting. At this point, we knew we had to make major improvements, starting with creating a single kernel, in which all would be run.

Our progress: Second Attempt

For our second attempt, we changed the program to create only a single kernel. However, we did notice that it would not work by only using the provided functions from the simple files. Fortunately, using these functions as a temple, we directly placed them in our main function. This code enabled us to alter the commands of the kernel every iteration, resulting in an updated stencil after each loop. As seen in the results, this was significantly faster than the sequential stencil.

We encountered that making a new kernel for every iteration massively degrades performance. Instead, you should change the arguments of the kernel (which can be done after every iteration). We are not sure what causes this discrepancy, however we speculate that creating a kernel takes a lot of time in comparison to only changing arguments, since the bandwidth between the GPU and CPU is quite limited (and probably a limiting factor). Therefore, less data over this constrained data bus means a faster overall program.

Team

Task 1: All
Task 2: All
Task 3: All

As we are friends with similar schedules, it was easy for us to find time to work together on this project. We were all present during the workgroup, multiple afternoons at university, and online during the weekend before the deadline. In the end, the workload was well-balanced and each person did around 20-25% of the work.