Homework 2

Julian Sanders
4675045

1 Exercises

In this exercise, we will exploit **shared memory parallelism** to solve a standard partial differential equation, the 3D Poisson equation:

$$-\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}+\frac{\partial^2 u}{\partial z^2}\right)=f(x,y,z) \tag{1}$$

on a unit cube $\Omega = [0\dots1]\times[0\dots1]\times[0\dots1]$, subject to Dirichlet boundary conditions.

- u(x,y,z)=0 if x=0, x=1, y=0, y=1, or z=1,
- u(x,y,z)=g(x,y) if z=0.

The PDE is discretized using second order finite diferences, and solved via the Conjugate Gradient method.

1.1 Exercise 1

Assignment:
If you haven't done so, add a Timer object to `cg_solve` and each of your basic operations from homework 1. Run the CG solver for 100 iterations on a grid on 12 cores and produce a runtime profile, e.g. as a 'pie chart'. What is the approximate size of a vector for this grid size, and how much memory do you need to request on DelftBlue for the CG solver to run? Does it help to use more aggressive compiler optimization, e.g. `-O3 -march=native`? If this run takes more than a few minutes, continue with a more feasible grid size and return to this one after you have optimized your code a bit, see below.
Implementation:
I added timers to each of the operations from `operations.cpp` that are called in `cg.solver`. The results for a 128^3 grid on 1 core can be seen in the following table:

Table 1: The timer results of the CG algorithm on a $128 \times 128 \times 128$ grid on 1 thread.

Timer label	Calls	Total time [s]	Mean time [s]
1. total iteration	599	49.18	0.08211
2. rho = <r, r>	599	1.458	0.002434
3. p = r + alpha * p	598	1.55	0.002592
4. q = op * p	598	41.38	0.0692
5. beta = <p,q>	598	1.604	0.002682
6. x = x + alpha * p	598	1.568	0.002622
7. r = r - alpha * q	598	1.617	0.002704
After cg_solver	1	0.0002955	0.0002955
Before cg_solver	1	49.27	49.27
Start of program	1	49.32	49.32

Because this stencil application still takes a relatively long time, I decided to rewrite the stencil application function so it does not rely on if statements or conditionals expressions to implement the boundary conditions.

The results for this can be found in the following table:

Table 2: The timer results of the CG algorithm with the improved stencil application function on a $128 \times 128 \times 128$ grid on 1 thread. The stencil application function now does not rely on if statements to implement the boundary conditions.

Timer label	**Calls**	**Total time [s]**	**Mean time [s]**
1. total iteration	599	38.32	0.06398
2. rho = <r, r>	599	1.485	0.002479
3. p = r + alpha * p	598	1.594	0.002666
4. q = op * p	598	30.47	0.05096
5. beta = <p,q>	598	1.618	0.002705
6. x = x + alpha * p	598	1.554	0.002598
7. r = r - alpha * q	598	1.591	0.002661
After cg_solver	1	0.0003956	0.0003956
Before cg_solver	1	38.39	38.39
Start of program	1	38.44	38.44

Note that the time to perform the stencil applications has decreased with more than 26%. A significant improvement.

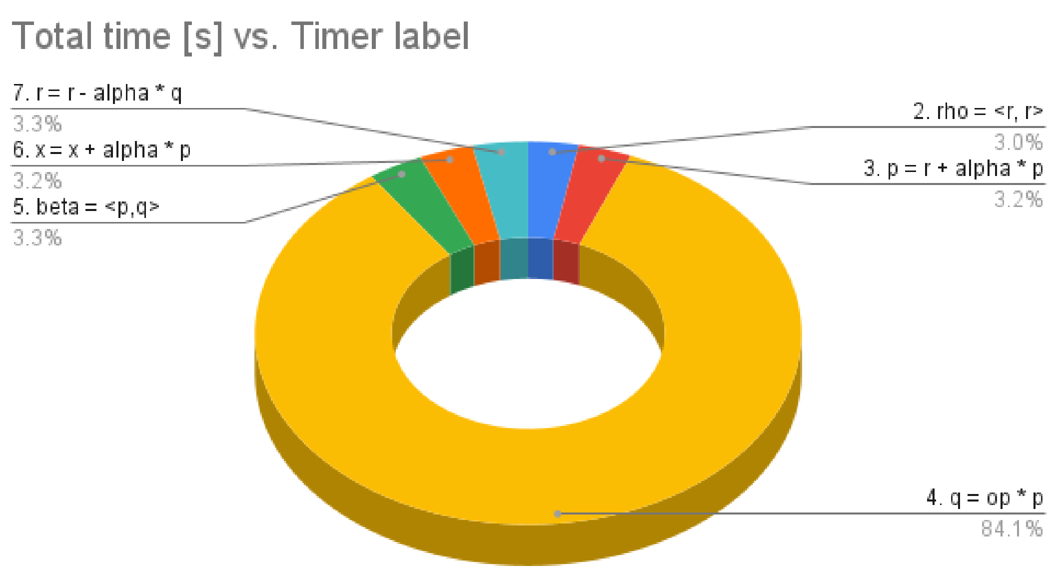

Figure 1: The relative times it takes to perform the operations on a problem with a $128 \times 128 \times 128$ grid on 1 thread. Here the stencil application `4. q = op * p` relies on conditional expressions to implement the boundary conditions.

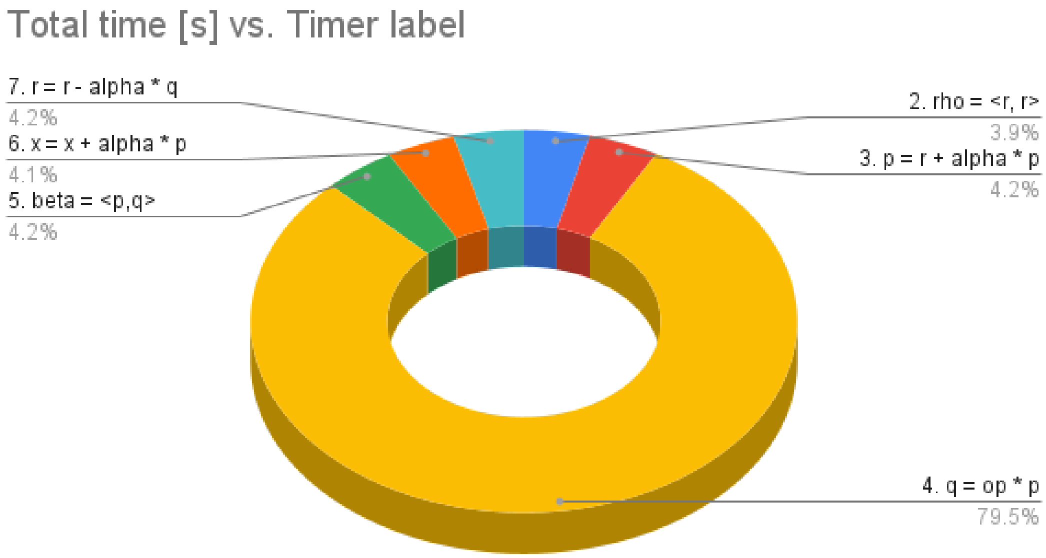

Figure 2: The relative times it takes to perform the operations on a problem with a $128 \times 128 \times 128$ grid on 1 thread. Here the stencil application `4. q = op * p` does not rely on conditional expressions nor if statements to implement the boundary conditions.

With this improved program, I generated the same results for a $600 \times 600 \times 600$ grid:

Table 3: The timer results of the CG algorithm with the improved stencil application function on a $600 \times 600 \times 600$ grid on 12 threads. The stencil application function now does not rely on if statements to implement the boundary conditions.

Timer label	**Calls**	**Total time [s]**	**Mean time [s]**
1. total iteration	3263	3357	1.029
2. rho = <r, r>	3263	76.62	0.02348
3. p = r + alpha * p	3262	183.7	0.0563
4. q = op * p	3262	2637	0.8084
5. beta = <p,q>	3262	112.8	0.03457
6. x = x + alpha * p	3262	160.6	0.04924
7. r = r - alpha * q	3262	186.5	0.05716
After cg_solver	1	0.007823	0.007823
Before cg_solver	1	3358	3358
Start of program	1	3359	3359

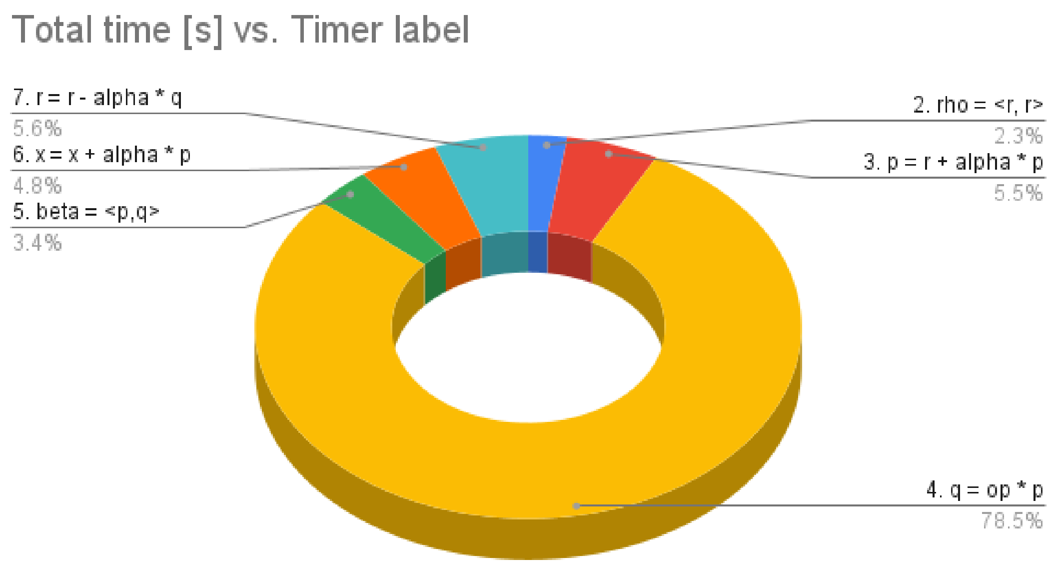

Figure 3: The relative times it takes to perform the operations on a problem with a $600 \times 600 \times 600$ grid on 12 threads. Here the stencil application `4. q = op * p` does not rely on conditional expressions nor if statements to implement the boundary conditions.

The approximate size of a vector for the 600^3 grid is an array of 216 000 000 doubles. That corresponds to 13 824 000 000 bits of data. That would be 1.728 GB.

For the CG solver to run, arrays `x`, `p`, `q` and `r` and `b` are needed. This required 8.64 GB of memory. The memory of this single integer and double values is negligible compared to this.

With a more agressive compiler, using the compiler flags `-O3` and `march=native` the code ran within the following time:

Table 4: The timer results of the CG algorithm with the improved stencil application function on a $600 \times 600 \times 600$ grid on 12 threads. The stencil application function now does not rely on if statements to implement the boundary conditions.

Timer label	**Calls**	**Total time [s]**	**Mean time [s]**
1. total iteration	3263	1757	0.5385
2. rho = <r, r>	3263	70.76	0.02169
3. p = r + alpha * p	3262	119.7	0.03669
4. q = op * p	3262	1242	0.3806
5. beta = <p,q>	3262	89.23	0.02736
6. x = x + alpha * p	3262	115.5	0.03541
7. r = r - alpha * q	3262	120.3	0.03688
Before cg_solver	1	1758	1758
Start of program	1	1758	1758

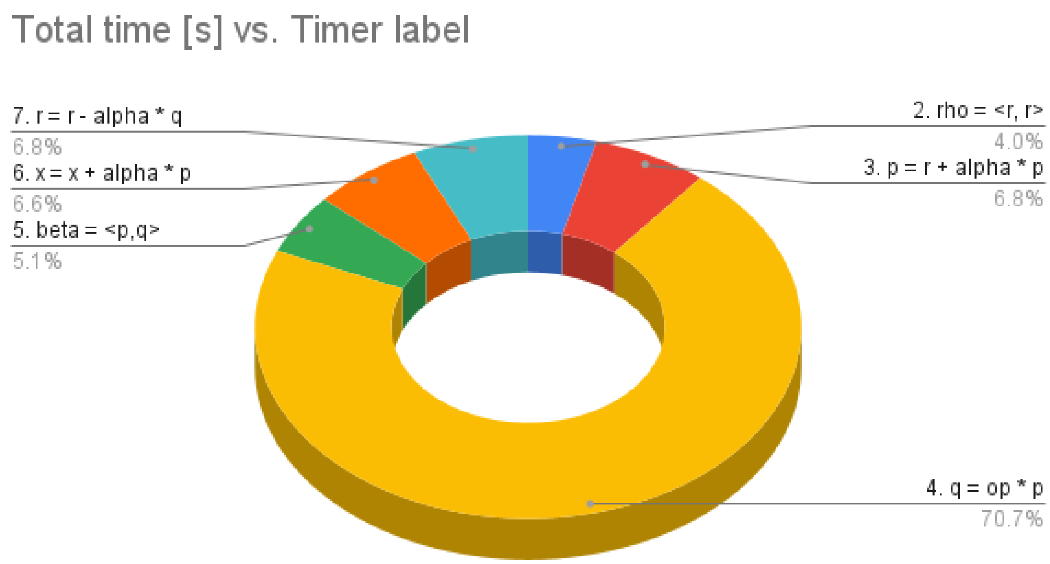

Figure 4: The relative times it takes to perform the operations on a problem with a $600 \times 600 \times 600$ grid on 12 threads. The executable wat made using a more agressive compiler, with compiler level 03 as opposed to 02, and optimized for the architecture of the machine it is compiled on

From the table and the figure it becomes clear that by using a more agressive compiler, by far the most efficiency is gained in the stencil application. This operation becomes more than twice as fast as it was before. The other operations also gain a significant amount of efficiency, albeit not as much as the stencil application.

To complete the rest of the exercises the more agressive compiler settings and the more efficient stencil application function will be used.

1.2 Exercise 2

Assignment

Extend the Timer class to store two additional doubles: the number of floating point operations (flops) performed in the timed section, and the number of bytes loaded and/or stored. The `summarize()` function should print out three additional columns:

- the computational intensity of the timed section
- the average floating point rate achieved (in Gflop/s)
- the average data bandwidth achieved (in GByte/s)

Run your benchmark program for the same problem size and 12 cores (with the values inserted in the Timer calls). What is the limiting hardware factor for each operation based on the roofline diagram from lecture 4? Hint: for `apply_stencil3d` the exact amount of data loaded is unclear due to caching. Here you can start with the most optimistic case (all elements cached after the first time they are accessed).

Implementation

I started by manually determining how many flops each operation from operations.hpp performs, and how much data each operation loads. This can be seen in the following table:

Table 5: The operations, and the data loaded and the computational intensity for each calculation in `cg_solver.cpp`. Here $n = n_x n_y n_z$ is the total grid dimension.

Step	**Operations [flops]**	**Data loaded [64 bits]**	I **[flops/byte]**
1. total iteration	19n	10n	0.2375
2. rho = <r, r>	2n	n	0.2500
3. p = r + alpha * p	3n	2n	0.1875
4. q = op * p	6n	n	0.7500
5. beta = <p,q>	2n	2n	0.1250
6. x = x + alpha * p	3n	2n	0.1875
7. r = r - alpha * q	3n	2n	0.1875

In reality, the stencil without if statement in the for loops has less operations, not $6 \cdot n$ but $6 \cdot (n_x - 2)(n_y - 2)(n_z - 2) + 2 \cdot 5 \cdot (n_y - 2)(n_z - 2) + 2 \cdot 5 \cdot (n_x - 2)(n_z - 2) + 2 \cdot 5 \cdot (n_x - 2)(n_y - 2) + 4 \cdot 4 \cdot (n_x - 2) + 4 \cdot 4 \cdot (n_y - 2) + 4 \cdot 4 \cdot (n_z - 2) + 8$. Since this is of roughly the same order as $6 \cdot n$ for large n, we will use $6 \cdot n$ to invesitigate performance. A similar approximation holds for the $2n - 1$ flops of the inner products, which can be approximated as $2n$ flops.

Timer label	**Calls**	**Total time [s]**	**Mean time [s]**	**Total Gflops**	**Mean Gflops**	**Gflops/s**	**Total GB**	**Mean GB**	**GB/s**
1. total iteration	3263	1913	0.5864	13390	4.104	6.998	7048	2.16	3.683
2. rho = <r, r>	3263	74.43	0.02281	1410	0.432	18.94	5638	1.728	75.75
3. p = r + alpha * p	3262	161.3	0.04945	2114	0.648	13.1	11270	3.456	69.89
4. q = op * p	3262	1270	0.3892	4228	1.296	3.33	704.6	0.216	0.555
5. beta = <p,q>	3262	108	0.03312	1409	0.432	13.04	11270	3.456	104.4
6. x = x + alpha * p	3262	138.4	0.04242	2114	0.648	15.28	11270	3.456	81.47
7. r = r - alpha * q	3262	161.6	0.04954	2114	0.648	13.08	11270	3.456	69.76
Before cg_solver	1	1914	1914	0	0	0	0	0	0
Start of program	1	1914	1914	0	0	0	0	0	0

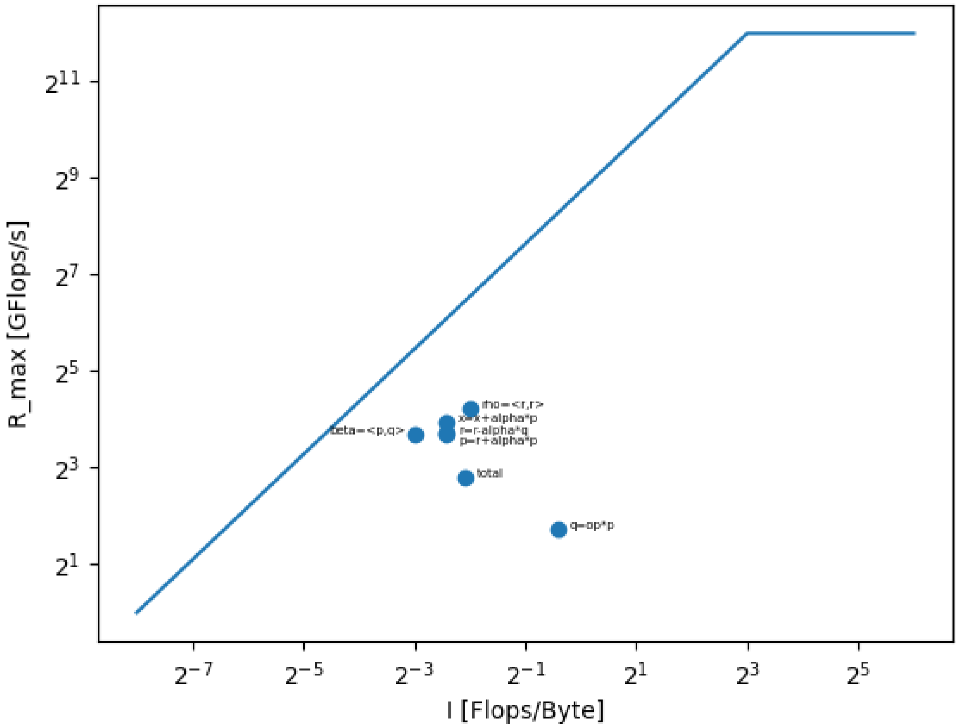

Figure 5: The results of timing the operations for the cg solver working on a 600^3 grid, on 12 threads with agressive compiling.

From the roofline plot, we see that the limiting factor for the operations is the memory, as the scatter plot points are closer to the "memory bound" roofline domain than the "compute bound" roofline domain.

From the dact that `q=op*p` is much further from the other points, we can see that the optimistic estimate that the operation only has to load n doubles per iteration is too optimistic. Most likely the program loads all 9 stencil variables each time it calls them. We will see if we can improve this with looping over blocks in exercise 5.

1.3 Exercise 3

Assignment

For each operation, determine the applicable peak performance R_{max} assuming 12 cores with 2 AVX512 FMA units (see lecture 1). Use the `likwid-bench` tool to measure the bandwidth on 12 cores (flag `-w M0: <size>` where `<size>` is the size of a vector). You can get a list of benchmarks it supports using `-a` and determine a suitable maximum memory bandwidth for each of your operations by selecting one that has a similar load/store ratio (note that you need to `module load 2022r2 likwid` on DelftBlue). Run both the likwid benchmarks and your benchmark program for 1, 2, 4, 6, 8, 10 and 12 threads. Make plots that show the achieved memory bandwidth for the chosen likwid benchmark and the operation in CG you benchmarked, and note down the absolute efficiency of your code compared to the roofline model prediction for the case of 12 threads.

Implementation

Likwid benchmarks for 1, 2, 4, 6, 8, 10, 12 threads:

I decided to compare the following tests to the operations in `cg_solver`. I based these decison on how similar the calculations performed are.

- `rho = <r, r>` $\iff$ `sum_avx512` - Double-precision sum of a vector, optimized for AVX-512
- `p = r + alpha * p` $\iff$ `daxpy_mem_avx512_fma` - Double-precision linear combination of two vectors, optimized for AVX-512
- `q = op * p` $\iff$ `copy_mem_avx512` - Double-precision vector copy, uses AVX-512 and only uses non temporal stores
- `beta = <p,q>` $\iff$ `ddot_avx512` - Double-precision dot product of two vectors, optimized for AVX-512
- `x = x + alpha * p` $\iff$ `daxpy_mem_avx512_fma` - Double-precision linear combination of two vectors, optimized for AVX-512 FMA
- `r = r - alpha * q` $\iff$ `daxpy_mem_avx512_fma` - Double-precision linear combination of two vectors, optimized for AVX-512 FMA

Test	**Threads**	**Performance [Gflops/s]**	**Bandwidth [GB/s]**
sum_avx512	1	2.02584	16.20673
	2	3.55248	28.41984
	4	6.02751	48.22010
	6	7.10272	56.82173
	8	7.51126	60.09004
	10	8.16323	65.30581
	12	8.09488	64.75904
daxpy_mem_avx512_fma	1	1.53606	18.43267
	2	2.69910	32.38923
	4	3.81597	45.79162
	6	4.30230	51.62754
	8	4.44751	53.37010
	10	4.47089	53.65073
	12	4.41214	52.94571
copy_mem_avx512	1	0.00000	10.99429
	2	0.00000	21.19051
	4	0.00000	37.13568
	6	0.00000	46.72065
	8	0.00000	50.99216
	10	0.00000	52.06363
	12	0.00000	51.60917
ddot_avx512	1	2.08782	16.70257
	2	3.60999	28.87990
	4	6.12012	48.96099
	6	7.23293	57.86348
	8	7.53953	60.31623
	10	8.12366	64.98928
	12	8.01506	64.12044

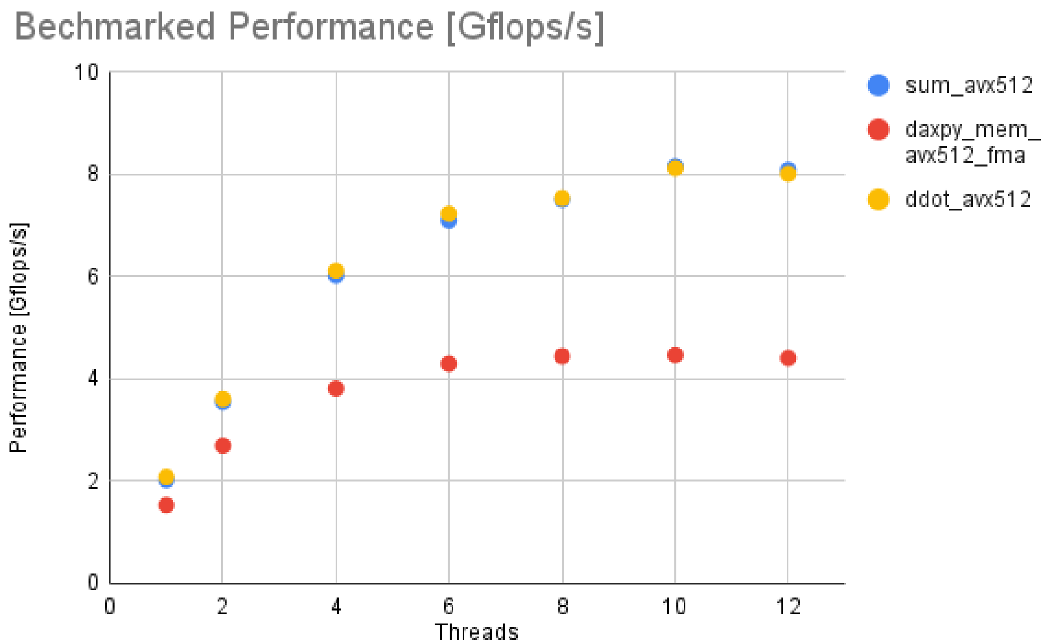

Figure 6: The performance of the different likwid tests.

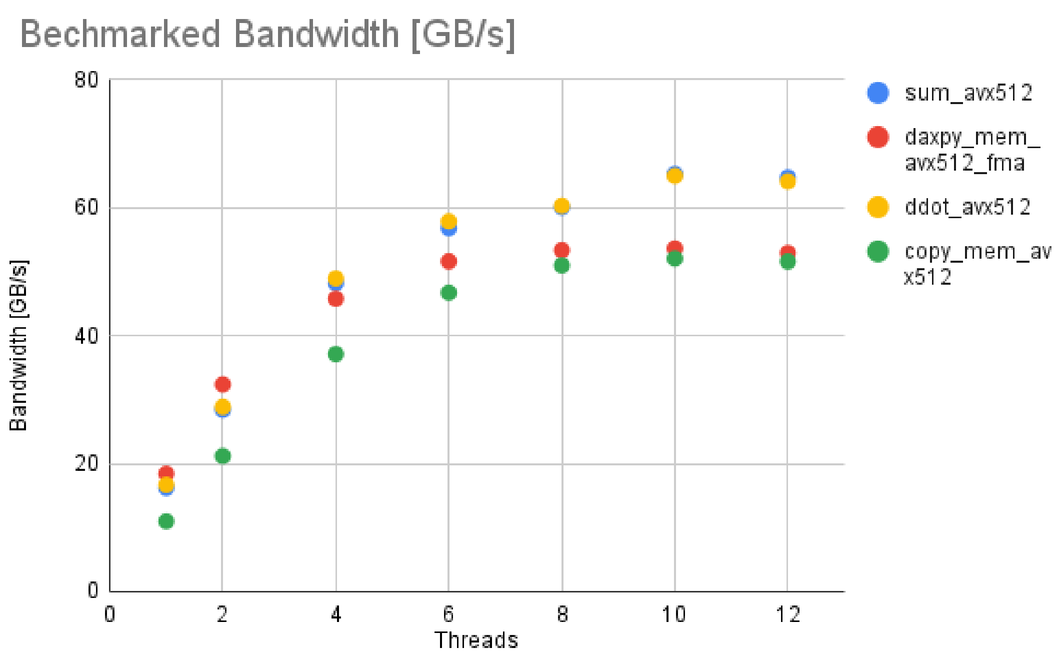

Figure 7: The bandwidth of differend likwid tests.

I ran the tests for vectors of 2GB, this approximates the vector size of 1.728GB of a 600^3 grid CG solver well.

Own benchmarks for 1, 2, 4, 6, 8, 10, 12 threads:

Table 6: The benchmarked results of the CG solver on a 600^3 grid for various numbers of threads.

Timer label	**Threads**	**Performance [Gflops/s]**	**Bandwidth [GB/s]**	**I [Flops/Byte]**
1. total iteration	12	7.31085	30.78248	0.2375
	10	5.62616	23.68912	0.2375
	8	4.89064	20.59216	0.2375
	6	3.86381	16.26864	0.2375
	4	2.62709	11.06144	0.2375
	2	1.43602	6.046392	0.2375
	1	0.747773	3.14852	0.2375
2. rho = <r, r>	12	19.6681	78.6722	0.2500
	10	15.9893	63.9572	0.2500
	8	13.261	53.0439	0.2500
	6	10.2046	40.8182	0.2500
	4	6.97116	27.8846	0.2500
	2	3.67904	14.7162	0.2500
	1	1.85148	7.4059	0.2500
3. p = r + alpha * p	12	15.3361	81.7927	0.1875
	10	9.68243	51.6396	0.1875
	8	10.3098	54.9855	0.1875
	6	9.67337	51.5913	0.1875
	4	6.42717	34.2782	0.1875
	2	4.16917	22.2356	0.1875
	1	2.57594	13.7384	0.1875
4. q = op * p	12	3.35022	4.46696	0.7500
	10	2.78519	3.713592	0.7500
	8	2.23909	2.985448	0.7500
	6	1.68801	2.250672	0.7500
	4	1.15469	1.539592	0.7500
	2	0.608069	0.81076	0.7500
	1	0.305981	0.4079752	0.7500
5. beta = <p,q>	12	14.9701	119.761	0.1250
	10	10.3385	82.7081	0.1250
	8	10.022	80.1761	0.1250
	6	8.46057	67.6846	0.1250
	4	5.73198	45.8558	0.1250
	2	3.04903	24.3922	0.1250
	1	1.61966	12.9573	0.1250
6. x = x + alpha * p	12	16.4673	87.8257	0.1875
	10	10.6421	56.7581	0.1875
	8	11.0219	58.7835	0.1875
	6	9.69228	51.6921	0.1875
	4	6.41786	34.2286	0.1875
	2	4.1572	22.1717	0.1875
	1	2.58048	13.7626	0.1875
7. r = r - alpha * q	12	15.3411	81.8193	0.1875
	10	9.57164	51.0488	0.1875
	8	10.2441	54.6353	0.1875
	6	9.60589	51.2314	0.1875
	4	6.40756	34.1737	0.1875
	2	4.12856	22.019	0.1875
	1	2.57601	13.7387	0.1875

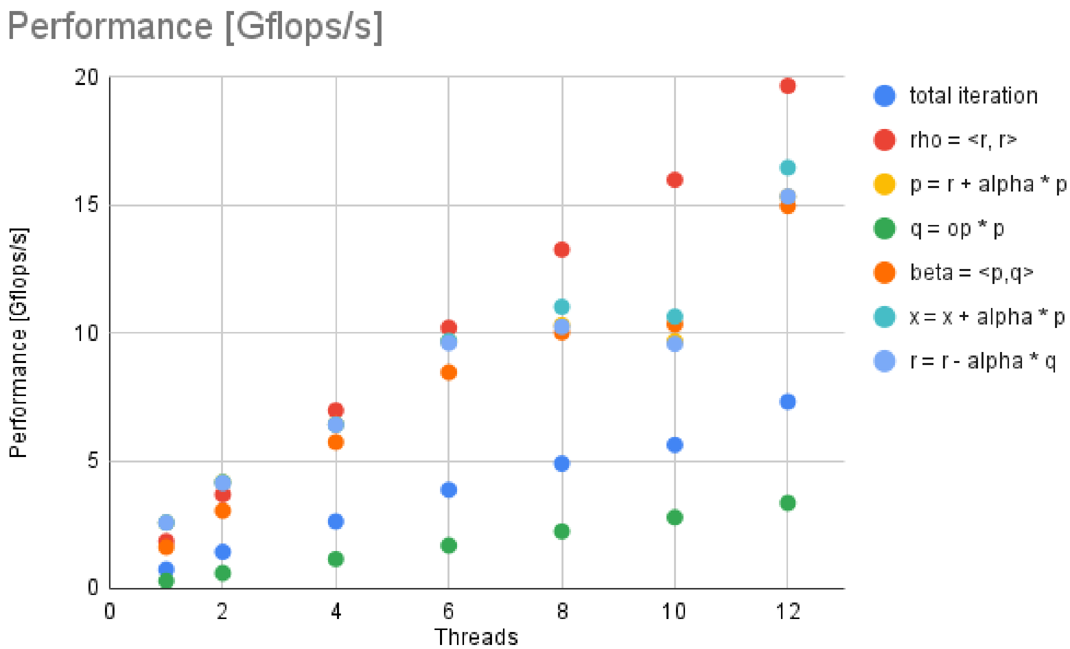

Figure 8: The performance of the different operations in `cg_solver` for a 600^3 grid.

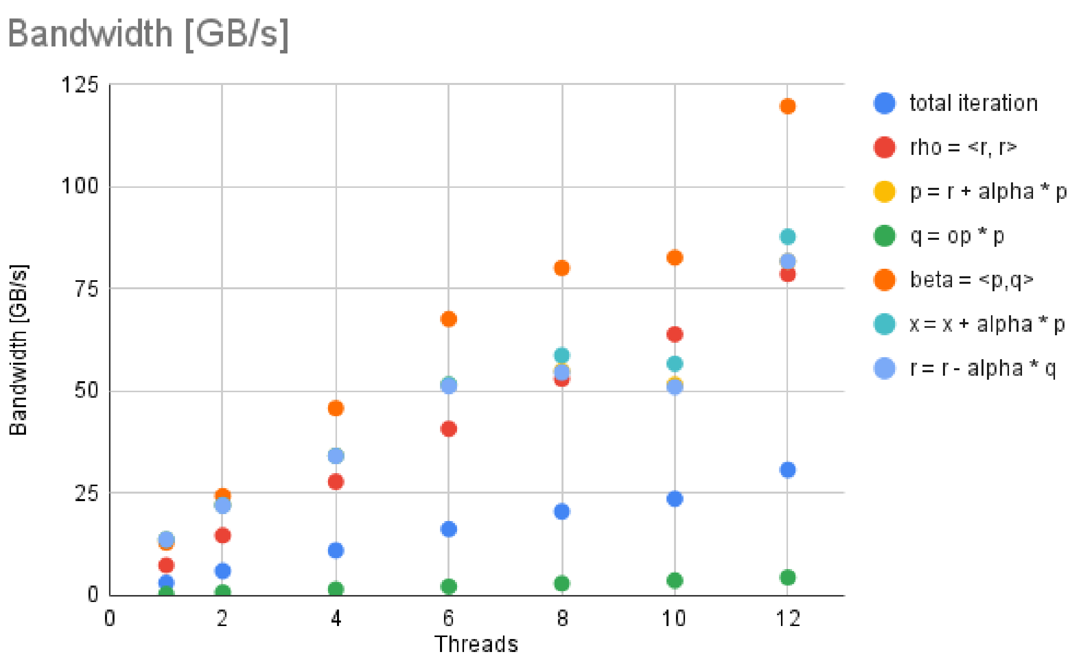

Figure 9: The bandwidth of the different operations in `cg_solver` for a 600^3 grid.

From these two tables we see that most of the operations actually outperform the likwid bencmarks. This could be due to the fact that the 1.728 GB vector is smaller than the 2GB vector. Only the stencil application is significantly slower. We will try to improve this further in exercise 5.

1.4 Exercise 4

Assignment

Repeat this to create similar graphs for up to 48 threads and report the overall efficiency on a full node. If this is significantly worse than on 12 cores, you may be struggling with the Non-Uniform Memory Architecture (NUMA): 12 cores can access memory which they initialized at the maximum speed (one NUMA domain). If you go beyond that, you may need to make sure that threads mostly access the memory portions they initialized, by adding the `schedule(static )` clause to your `#pragma omp parallel for` statements. Also, make sure to set the environment variables `OMP_PROC_BIND=close` and `OMP_PLACES=cores`.

Implementation

I am skipping this exercise due to not having time. I have a resit next week. I would rather test if my block stencil implementation is more efficient than to repeat exercise 3 for more cores.

I did add the `OMP_PROC_BIND=close`, `OMP_PLACES=cores` and `schedule(static)` lines just to be safe.

1.5 Exercise 5

Assignment

For any operation that performs significantly worse than the roofline prediction on 12 cores (say, less than 50%), try to optimize that operation by

- experimenting with compiler flags
- checking the model assumptions and hardware parameters
- actually changing the code. For example, if you used if-statements in the `apply_stencil3d` innermost loop, try to get rid of them. If you have more than one read of the `u` vector and one write of the `v` vector because of multiple passes over them, reduce the data traffic. If your code is much faster for certain grid sizes and then suddenly the performance drops as you increase it, try implementing a variant that loops over blocks or try parallelizing the innermost loop instead of the outermost one. Use the 'layer condition' introduced in lecture 6 by Prof. Wellein to guide these optimizations. Document the changes that have a positive effect along with the achieved percentage of the roofline model. And obviously run your tests after each step to make sure that your code is producing correct results

Implementation

I already implemented a a more efficient stencil operator without if statements or conditional expressions. See exercise 1.

Here I decided to introduce blocking in the stencil application function.

In this 3d case, we will split the total domain into block shaped columns in the z direction. These have size blockx and blocky in the x and y direction respecively.

In order to make sure that the stencil only has to load one new element instead of 8, we need to make sure that three layers of doubles of dimension `blockx·blocky` are loaded in the cache. To this end, we need to satisfy the condition:

$$3 \cdot \texttt{blockx} \cdot \texttt{blocky} \cdot 8\texttt{B} < \texttt{CacheSize}/2 \quad (2)$$

We can find the cache sizes by running `likwid-topology` on one of the compute nodes of DelftBlue. If we choose `blockx` and `blocky` to be equal, we get the following table:

Table 7: The cache size and blocking sizes for the different cache levels. The block sizes will be rounded down to the nearest integer.

Cache	**Size**	`blockx=blocky`
L1	1 KB	4.56
L2	1 MB	144.34
L3	17.88 MB	610.33

I decided to test this for a 290^3 grid. This will allow blocking over blocksize of 144 nicely, as the non boundary loop of the stencil is a loop of 288 elements in both the x and y direction.

Table 8: The timing, performance and bandwidth resultsy result of the stencil application function for a 290^3 grid performed on one thread. The stencil was ran without blocking, and with blocking for various block sizes `blockx` = `blocky`

Block size	**Total time**	**Average time**	**Performance [Gflops/s]**	**Bandwidth [GB/s]**
no blocking	728.674	0.485459	0.301434	0.401912
290	718.242	0.478509	0.305812	0.40775
144	738.158	0.491778	0.297561	0.396748
72	768.998	0.512324	0.285628	0.380837
36	946.505	0.630583	0.232061	0.309415
7	956.056	0.636946	0.229743	0.306324
6	937.882	0.624838	0.234195	0.31226
5	931.926	0.62087	0.235692	0.314256
4	864.051	0.57565	0.254206	0.338942

Note that from the table, it becomes clear that blocking is not advantageous in when running the program on one thread. Stragely, the stencil applicaton time is faster for a block size that is equal to the stencil size than than running it without blocking.

Table 9: The timing, performance and bandwidth results of the stencil application function for a 290^3 grid performed on 4 threads. The stencil was ran without blocking, and with blocking for various block sizes `blockx` = `blocky`

Block size	**Total time**	**Average time**	**Performance [Gflops/s]**	**Bandwidth [GB/s]**
no blocking	182.837	0.129948	1.1261	1.50146
290	659.551	0.469766	0.311504	0.415339
144	184.475	0.130926	1.11768	1.49025
72	194.328	0.138115	1.05951	1.41267
36	251.111	0.1786	0.819341	1.09245
7	258.699	0.184259	0.794176	1.0589
6	252.678	0.179332	0.815997	1.088
5	247.208	0.175699	0.832869	1.11049
4	221.655	0.157537	0.928886	1.23851

In this case, we see that the block size of 144 is the most eddicient, albeit less efficient than the no blocking stencil application.

In order to see if blocking will add a speedup for larger grids, I also timed the stencil application for a 600^3 grid. The results can be seen in the table below:

Table 10: The timing, performance and bandwidth results result of the stencil application function for a 600^3 grid performed on 16 threads. The stencil was ran without blocking, and with blocking for various block sizes `blockx` = `blocky`

Block size	**Total time**	**Average time**	**Performance [Gflops/s]**	**Bandwidth [GB/s]**
no blocking	956.764	0.293396	4.41724	5.88965
600	13633.9	4.18089	0.309982	0.413309
150	998.47	0.306185	4.23273	5.64364
144	1829.36	0.560982	2.31024	3.08031
72	1855.29	0.568932	2.27795	3.03727
7	1400.19	0.429374	3.01835	4.02446
6	1400.46	0.429457	3.01776	4.02368
5	1378.42	0.422699	3.06601	4.08801
4	1428.87	0.43817	2.95775	3.94367

From the last table, we see that blocking still does not provide a speedup for a 600^3 grid compared to the normal grid function. The reason that the 150 block time is so much faster is that this divides the 600^2 x-y grid into exactly 16 blocks, which can be parallelized on the 16 cores. For the 144 block size there will be 25 blocks, which have to be distributed in 16 treads. This is significatly less efficient.

1.6 Exercise 6

Assignment

Finally, what is the total CG runtime on 12 and 48 cores you achieve for the 600^3 problem, and what is the total runtime predicted by the Roofline model

Implementation

The total runtime results for the 600^3 grid for 12, 16 and 48 threads with and without blocking can be seen in the tables below.

Table 11: The riming results for a 600^3 grid on 12 threads, with the regular stencil function

Operation	**Total Time**	**Mean Time**	**Performance [Gflops/s]**	**Bandwidth [GB/s]**	**I [Flops/Byte]**
1. total iteration	2038.75	0.624809	6.56841	27.6564	0.2375
2. rho = <r, r>	70.204	0.0215152	20.0789	80.3155	0.25
3. p = r + alpha * p	196.339	0.0601898	10.7659	57.4184	0.1875
4. q = op * p	1270.08	0.389357	3.32857	4.43809	0.75
5. beta = <p,q>	105.877	0.0324578	13.3096	106.477	0.125
6. x = x + alpha * p	196.572	0.0602611	10.7532	57.3504	0.1875
7. r = r - alpha * q	199.651	0.0612051	10.5874	56.4659	0.1875

Table 12: The riming results for a 600^3 grid on 12 threads, with the block stencil function with `blockx` = 200 and `blocky` = 150.

Operation	**Total Time**	**Mean Time**	**Performance [Gflops/s]**	**Bandwidth [GB/s]**	**I [Flops/Byte]**
1. total iteration	2088.36	0.640014	6.41236	26.9994	0.2375
2. rho = <r, r>	70.1038	0.0214844	20.1076	80.4303	0.25
3. p = r + alpha * p	201.03	0.0616277	10.5147	56.0786	0.1875
4. q = op * p	1310.65	0.401793	3.22554	4.30072	0.75
5. beta = <p,q>	108.427	0.0332396	12.9966	103.972	0.125
6. x = x + alpha * p	198.365	0.0608107	10.656	56.8321	0.1875
7. r = r - alpha * q	199.757	0.0612374	10.5818	56.4361	0.1875

Table 13: The riming results for a 600^3 grid on 16 threads, with the regular stencil function

Operation	**Total Time**	**Mean Time**	**Performance [Gflops/s]**	**Bandwidth [GB/s]**	**I [Flops/Byte]**
1. total iteration	1764.03	0.540781	7.58903	31.9538	0.2375
2. rho = <r, r>	56.8569	0.0174301	24.7848	99.139	0.25
3. p = r + alpha * p	215.305	0.0660243	9.81458	52.3444	0.1875
4. q = op * p	956.528	0.293324	4.41833	5.8911	0.75
5. beta = <p,q>	102.528	0.0314406	13.7402	109.922	0.125
6. x = x + alpha * p	214.805	0.065871	9.8374	52.4661	0.1875
7. r = r - alpha * q	217.976	0.0668432	9.69433	51.7031	0.1875

Table 14: The riming results for a 600^3 grid on 16 threads, with the block stencil function with `blockx` = `blocky` = 150.

Operation	**Total Time**	**Mean Time**	**Performance [Gflops/s]**	**Bandwidth [GB/s]**	**I [Flops/Byte]**
1. total iteration	1813.32	0.555892	7.38273	31.0852	0.2375
2. rho = <r, r>	56.7327	0.017392	24.839	99.356	0.25
3. p = r + alpha * p	216.999	0.0665437	9.73795	51.9358	0.1875
4. q = op * p	999.918	0.306629	4.2266	5.63547	0.75
5. beta = <p,q>	104.216	0.0319581	13.5177	108.141	0.125
6. x = x + alpha * p	215.512	0.0660876	9.80517	52.2942	0.1875
7. r = r - alpha * q	219.911	0.0674366	9.60902	51.2481	0.1875

Table 15: The riming results for a 600^3 grid on 48 threads, with the regular stencil function

Operation	**Total Time**	**Mean Time**	**Performance [Gflops/s]**	**Bandwidth [GB/s]**	**I [Flops/Byte]**
1. total iteration	664.347	0.221523	18.5263	78.0055	0.2375
2. rho = <r, r>	22.882	0.00762989	56.6194	226.478	0.25
3. p = r + alpha * p	90.7246	0.0302617	21.4132	114.204	0.1875
4. q = op * p	328.381	0.109533	11.832	15.776	0.75
5. beta = <p,q>	44.6179	0.0148825	29.0273	232.218	0.125
6. x = x + alpha * p	88.9745	0.0296779	21.8344	116.45	0.1875
7. r = r - alpha * q	88.7323	0.0295972	21.894	116.768	0.1875

Table 16: The riming results for a 600^3 grid on 48 threads, with the block stencil function with `blockx` = 100 and `blocky` = 75.

Operation	**Total Time**	**Mean Time**	**Performance [Gflops/s]**	**Bandwidth [GB/s]**	**I [Flops/Byte]**
1. total iteration	848.717	0.283094	14.4969	61.0397	0.2375
2. rho = <r, r>	23.2692	0.00776157	55.6588	222.635	0.25
3. p = r + alpha * p	119.721	0.0399469	16.2215	86.5149	0.1875
4. q = op * p	432.66	0.144364	8.97728	11.9697	0.75
5. beta = <p,q>	77.4686	0.0258487	16.7126	133.701	0.125
6. x = x + alpha * p	93.3838	0.0311591	20.7965	110.915	0.1875
7. r = r - alpha * q	102.174	0.0340922	19.0073	101.372	0.1875

The most efficient programs are those without blocking, both for 12, 16 as 48 cores. However, the stencil application is still much slower than predicted by the roofline model. This is due to the fact that the stencil has a much slower bandwidth than the hardware allows. A reason for this the fact that the stencil loads all 9 points each time it applies the stencil to a lattice point. In the ideal situation it would just load each point from memory once. I attempted to achieve this with data blocking, but I did not achieve a higher efficiency.