Delft University of Technology

Special Topics in Computational Science and Engineering
WI4450

Homework2

Author:
Femke Klein (4904125)

April 11, 2023

Contents

1 Task 1

We have added Timer objects to each of the basic operations in cg_solve. We run the CG solver for 100 iterations on a grid on 12 OMP_NUM_THREADS and produced a runtime profile as a 'pie chart' which can be seen in Figure 1. Then we find that the runtime is mostly due to the basis function apply_stencil3d, which we expected since we use a lot of if-statements in its code.

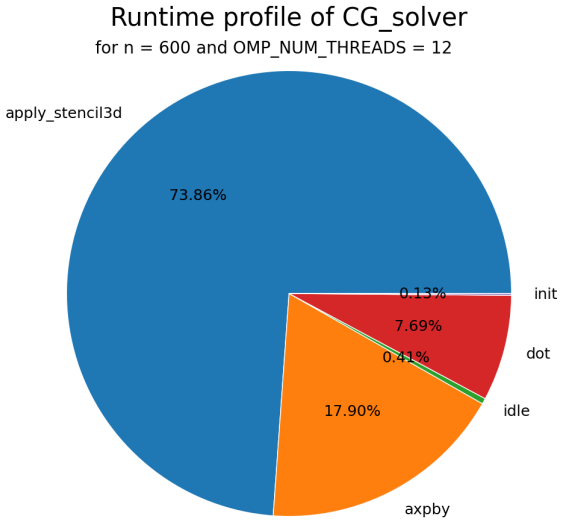

Figure 1: Pie Chart of cg_solve

1.1 Vector size

Recall that an integer takes 4 Bytes and a double 8 Bytes. Using $n = 600^3$, the approximate size of a vector of type double for this grid size is $n \cdot 8$ Bytes, i.e. 1.728 GBytes.

1.2 Amount of Bytes needed to run cg_solver

Inside cg_solver we define three vectors of length n and type double, 4 scalars doubles and 1 integer. Thus we need $4 \cdot 8 + 1 \cdot 4 = 36$ Bytes.

As input for cg_solver, we need 1 stencil3d, 4 integers, 2 doubles and 2 vectors of length n and type double. Type stencil3d exists of 3 integers and 7 doubles.

Then for input of cg_solver we need $(3 \cdot 4 + 7 \cdot 8) + 4 \cdot 4 + 2 \cdot 8 + 2 \cdot n \cdot 8 = 3456000100$ Bytes $=$ 3.456000100 GBytes.

In main_cg_solver we additionally use 7 integers and 6 doubles. There we need $7 \cdot 4 + 6 \cdot 8 = 76$ Bytes.

So in total, we need about $36 + 3456000100 + 76 = 3456000212$ Bytes $= 3.456000212$ GBytes.

1.3 Compiler optimization

Until now we used the following compiler optimization: '-O2 -g'. We now replace it with a more aggressive compiler optimization, '-O3 -march=native', and investigate whether the execution time improves. In Figure 2, a bar plot shows the runtimes of the basis function used in cg_solver and the total runtime itself using '-O2 -g' which is the red bar and '-O3 -march=native' which is the blue bar. The exact execution time of each function and compiler optimization is shown above each bar (in seconds). We find that for all basis function the execution time remains about the same (axpby, dot and init) or improves about 10 seconds (apply_stencil 3d) which is a good result. This means that the total runtime of cg_solver, which are the bars at the most right in Figure 2, is also about 10 seconds smaller for compiler optimization '-O3 -march=native' than for '-O2 -g'.

We have also added a pie chart to see the partition of execution time of cg_solver over the basis functions which can be seen in Figure 3. Since we could improve the runtime of the basis function apply_stencil3d and the others not really, we find that this basis function takes less percentage of total runtime.

Therefore, we use compiler optimization '-O3 -march=native' for the rest of this assignment, unless said otherwise.

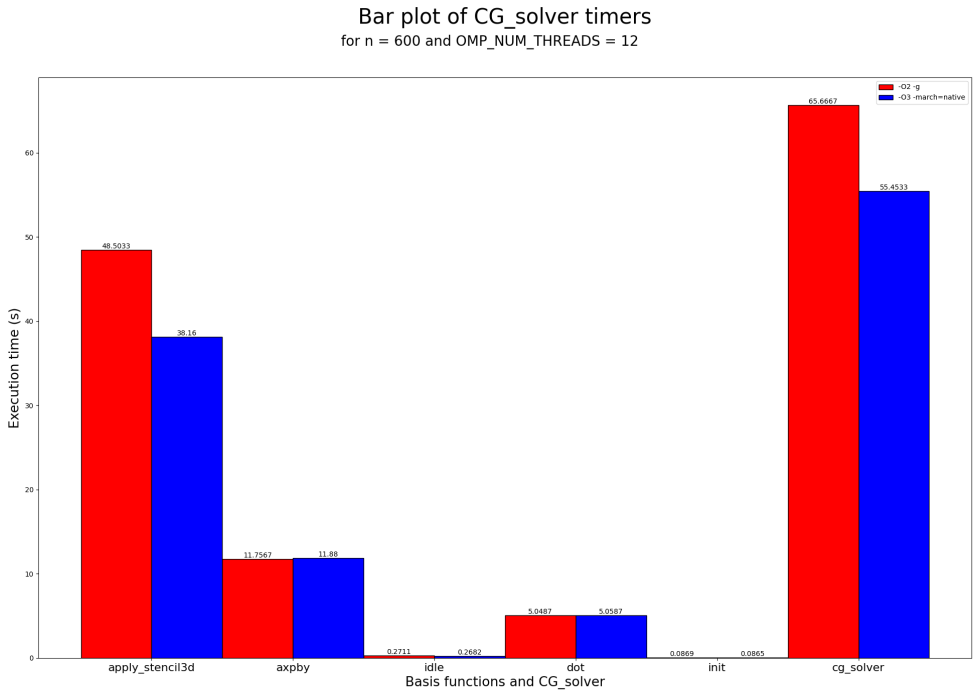

Figure 2: Bar plot of cg_solve using '-O2 -g' (red) or '-O3 -march=native' (blue) as compiler optimization.

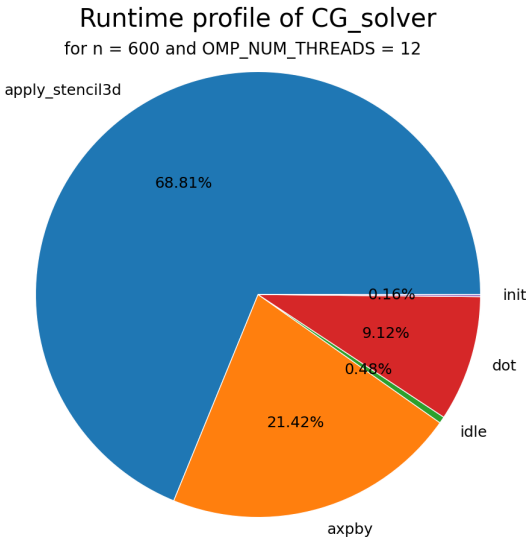

Figure 3: Pie Chart of cg_solve

2 Task 2

We have added the following three additional columns to the summarize() function to print:

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

The number of calls, execution time per basis function, mean execution time per basis function, CI, average Gflops/s and average GBytes/s for $n = 600^3$ with 12 OMP_NUM_THREADS are denoted in Table 1.

Therefore, the formula for amount of flops and bytes becomes

$$\text{GFlops/second} = \frac{\text{number of GFlops of operation}}{\text{mean run time}}$$
$$\text{GBytes/second} = \frac{\text{number of GBytes of operation}}{\text{mean run time}}.$$

In the most optimistic case, when all elements of type double cached after the first time they are accessed, it loads/stores $2 \cdot n \cdot 8 + \text{sizeof}(op)) = 3456000076$ Bytes $= 3.456000076$ GBytes. Note that $\text{sizeof}(op) = 7 \cdot 8 + 4 \cdot 4 = 72$. In rest of report, we assume the most optimistic case for apply_stencil3d. Since the size of op is so small compared to the rest, we assume $2 \cdot n$ loads/stores for likwid later on. Furthermore, apply_stencil3d takes $1 \cdot n + 4 \cdot ((nx-1) \cdot ny \cdot nz + nx \cdot (ny-1) \cdot nz + nx \cdot ny \cdot (nz-1))$ flops, which equals 2803680000 flops $= 2.80368$ Gflops.

In function axpby, we store/load $3 \cdot n \cdot 8 = 5184000000$ Bytes $= 5.184$ GBytes and have $3 \cdot n = 648000000$ flops $= 0.648$ Gflops.

In function dot, we store/load $2 \cdot n \cdot 8 = 3456000000$ Bytes $= 3.456$ GBytes and have $2 \cdot n = 432000000$ flops $= 0.432$ Gflops.

In function init, we store $1 \cdot n \cdot 8 = 1728000000$ Bytes $= 1.728$ GBytes have 0 Gflops.

Note that the above number of flops and bytes are per iteration.

Furthermore, the computational intensity (I_c) is given by

$$I_c = \frac{\text{Gflops}}{\text{GBytes}}$$

Then we find that I_c equals 0.125, 0.125, 0.125 and 0 for operations apply_stencil3d, axpby, dot and init, respectively. This is also stated in Table 1 . Then we find that for $I_c = 0.125$, we are on the left side of the kink, so we have that all operations are memory-bound which is the bottle neck here.

Table 1: (Averaged) execution times per function and their floating point rate and data bandwidth, number of threads = 12, $n = 600^3$.

Basis function	Calls	Time (seconds)	Mean time (seconds)	I_c	Floating point rate Gflops / second	Data bandwith GBytes / second
apply_stencil3d	100	38.4966	0.384966	0.81124998	7.28292888	8.97741637
axpby	100	5.41445	0.0541445	0.125	11.96797459	95.74379669
dot	100	3.56683	0.0356683	0.125	12.11159489	96.89275912
init	200	6.52955	0.0326477	0	0	52.92869023

3 Task 3

In the Roofline model stated in powerpoint of lecture 4 and on https://doc.dhpc.tudelft.nl/delftblue/perf-Xeon_6248R/, we find that the applicable peak performance $R_{\max}$ (lecture 1) is

$$R_{\max} = I_c \cdot R_{\text{data}},$$

where R_{data} denotes the bandwidth that Likwid measures and $R_{\max}$ is its corresponding applicable peak performance.

Note that in likwid, they calculate bandwidth per CPU socket. Since we have two CPU units, the total number of bytes has to be multiplied with two.

Then for number of threads equal to 12, we find that the applicable bandwidths according to likwid are 119.14002, 106.33818, 119.14002 and 50.79416 GBytes/s for operations apply_stencil3d, axpby, dot and init, respectively. In Figure 4 the applicable bandwidths are shown against the number of cores, where the red line represents the measurements by likwid-bench and the green line represents our own benchmarks. The peak perfomances are $12{\cdot}2{\cdot}(8{+}8){\cdot}2.5 = 960, 12{\cdot}(2{\cdot}8{+}1{\cdot}8){\cdot}2.5 = 720, 12{\cdot}2{\cdot}(8{+}8){\cdot}2.5 = 960, 12 \cdot 2.5 = 30$ GFlops/s and the applicable peak performances $R_{\max}$ are $0.81124998 \cdot 119.14002 = 96.6523388, 0.125 \cdot 106.33818 = 13.2922725, 0.125 \cdot 119.14002 = 14.8925025, 0 \cdot 50.79416 = 0$ GFlops/s for operations apply_stencil3d, axpby, dot and init, respectively. This shown in Figure 5, where the peak performance is denoted in black, the applicable peak performance $R_{\max}$ in red and the applicable bandwidth in blue.

Then we have that

$$P = \min(R_{\max}, R_{\text{flops,code}}),$$

where $R_{\text{flops,code}} = I_c \times R_{\text{data,code}}$. Then P equals $\min(96.6523388, 7.28292888) = 7.28292888$, $\min(13.2922725, 11.96797459) = 11.96797459$, $\min(14.8925025, 12.11159489) = 12.11159489$, $\min(0, 0) = 0$ Gflops/s for operations apply_stencil3d, axpby, dot and init, respectively. Thus the bottleneck for each operation for 12 threads is memory-bound except for operation init.

The absolute efficiency is given by

$$\text{absolute efficiency} = \frac{P}{R_{\max}}.$$

Thus we have $\frac{7.28292888}{96.6523388} = 0.0753518$, $\frac{11.96797459}{13.2922725} = 0.9003708$, $\frac{12.11159489}{14.8925025} = 0.8132679$ for operations apply_stencil3d, axpby, dot, respectively. Then we find that we can improve the code most for operation apply_stencil3d, since the absolute efficiency is lower than the others.

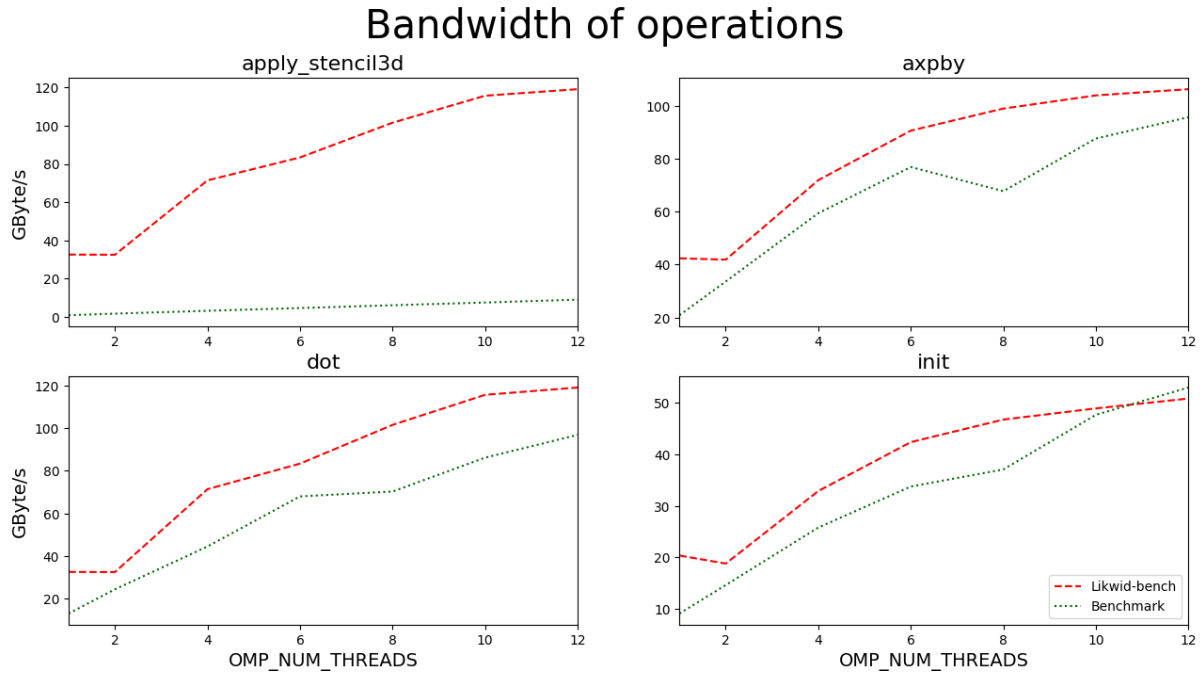

Figure 4: Bandwidth plots of the basis operations apply_stencil3d, axpby, dot and init, and number of threads equal to 1,2,4,6,8,10,12.

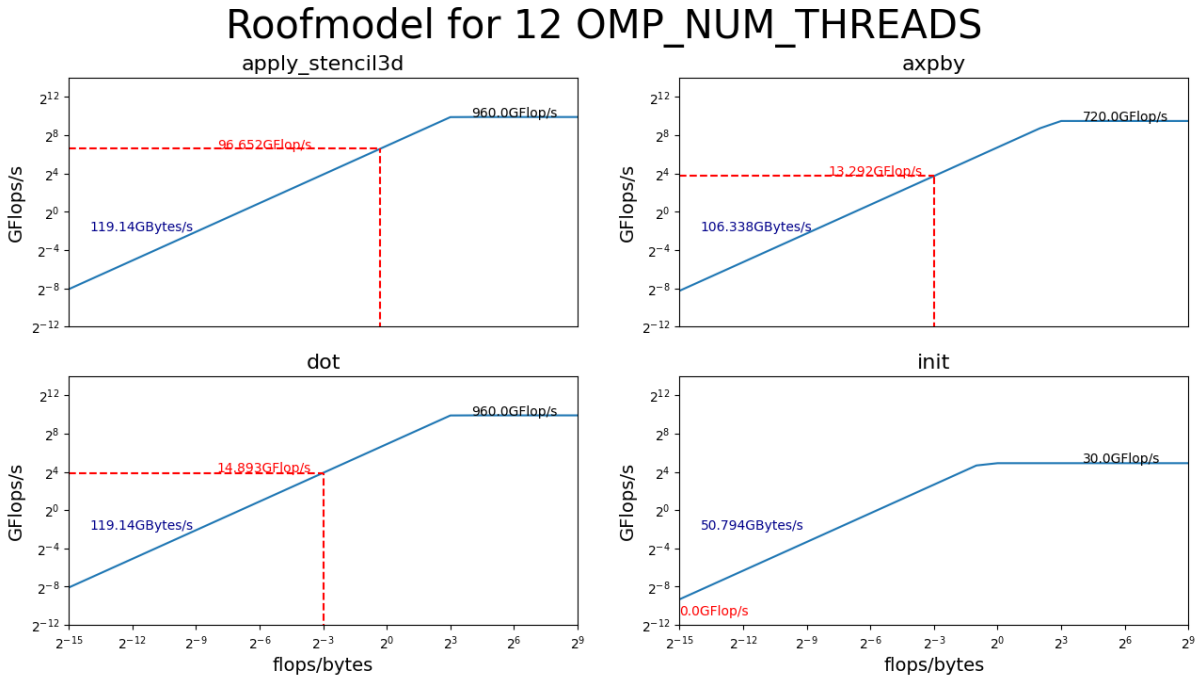

Figure 5: Roofmodel of the basis operations apply_stencil3d, axpby, dot and init, and number of threads equal to 12.

4 Task 4

We repeat task 3, but now for number of threads equal to 16, 20, 24, 28, 32, 36, 40, 44 and 48. For this, we added before hand the schedule(static) clause to our #pragma omp parallel for statements and set the environment variables OMP_PROC_BIND=close and OMP_PLACES=cores to avoid struggling with NUMA.

Then for number of threads equal to 48, we find that the applicable bandwidths according to likwid are 150.37572, 146.34202, 150.37572 and 62.77732 GBytes/s for operations apply_stencil3d, axpby, dot and init, respectively. In Figure 4 the applicable bandwidths are shown against the number of cores, where the red line represents the measurements by likwid-bench and the green line represents our own benchmarks. The peak perfomances are $48 \cdot 2 \cdot (8+8) \cdot 2.5 = 3840, 48 \cdot (2 \cdot 8 + 1 \cdot 8) \cdot 2.5 = 2880, 48 \cdot 2 \cdot (8+8) \cdot 2.5 = 3840, 12 \cdot 2.5 = 120$ GFlops/s and the applicable peak performances $R_{\max}$ are $0.81124998 \cdot 150.37572 = 121.9922998, 0.125 \cdot 146.34202 = 18.2927525, 0.125 \cdot 150.37572 = 18.796965, 0 \cdot 62.77732 = 0$ GFlops/s for operations apply_stencil3d, axpby, dot and init, respectively.

P equals $\min(121.9922998, 25.86468385) = 25.86468385$, $\min(18.2927525, 22.03991674) = 18.2927525$, $\min(18.796965, 2$ $18.796965, \min(0,0) = 0$ Gflops/s for operations apply_stencil3d, axpby, dot and init, respectively. Thus the bottleneck for operation apply_stencil3d for 48 threads is memory-bound and for operations axpby and dot the bottleneck is compute-bound. The abosulute efficiency is given by $\frac{P}{R_{\max}}$: $\frac{25.86468385}{121.9922998} = 0.2120190$, $\frac{18.2927525}{18.2927525} = 1$, $\frac{18.796965}{18.796965} = 1$ for operations apply_stencil3d, axpby and dot, respectively.

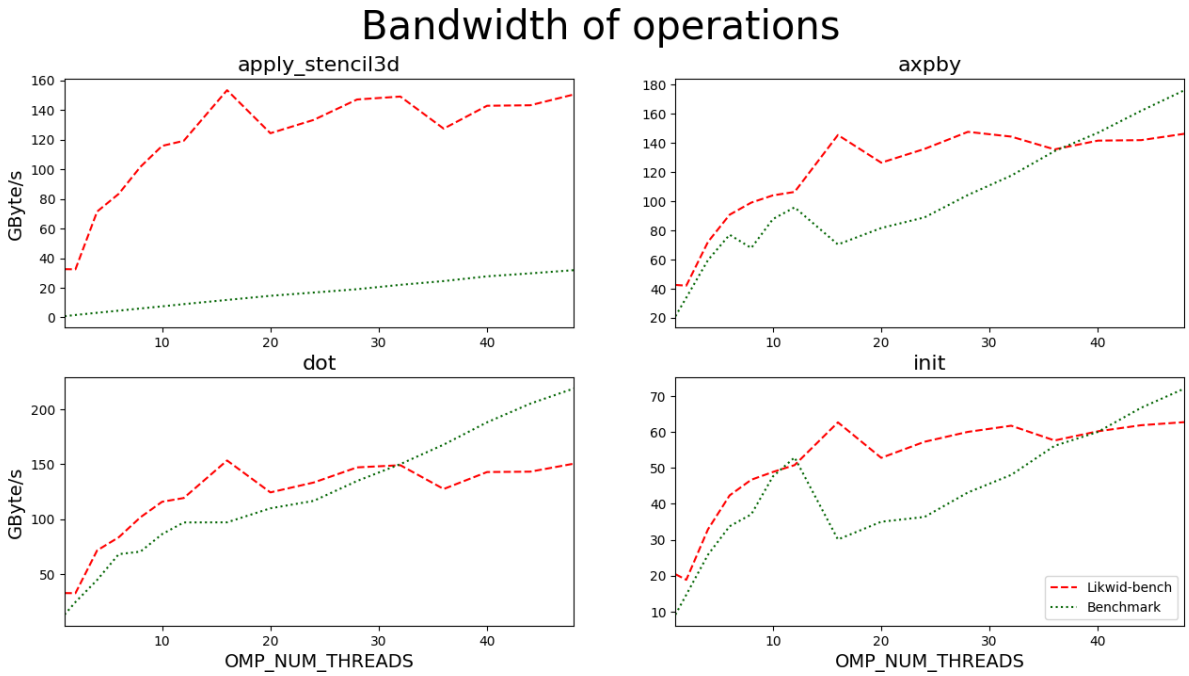

Figure 6: Bandwidth plots of the basis operations apply_stencil3d, axpby, dot and init, and number of threads equal to 1,2,4,6,8,10,12,16,20,24,28,32,36,40,44,48.

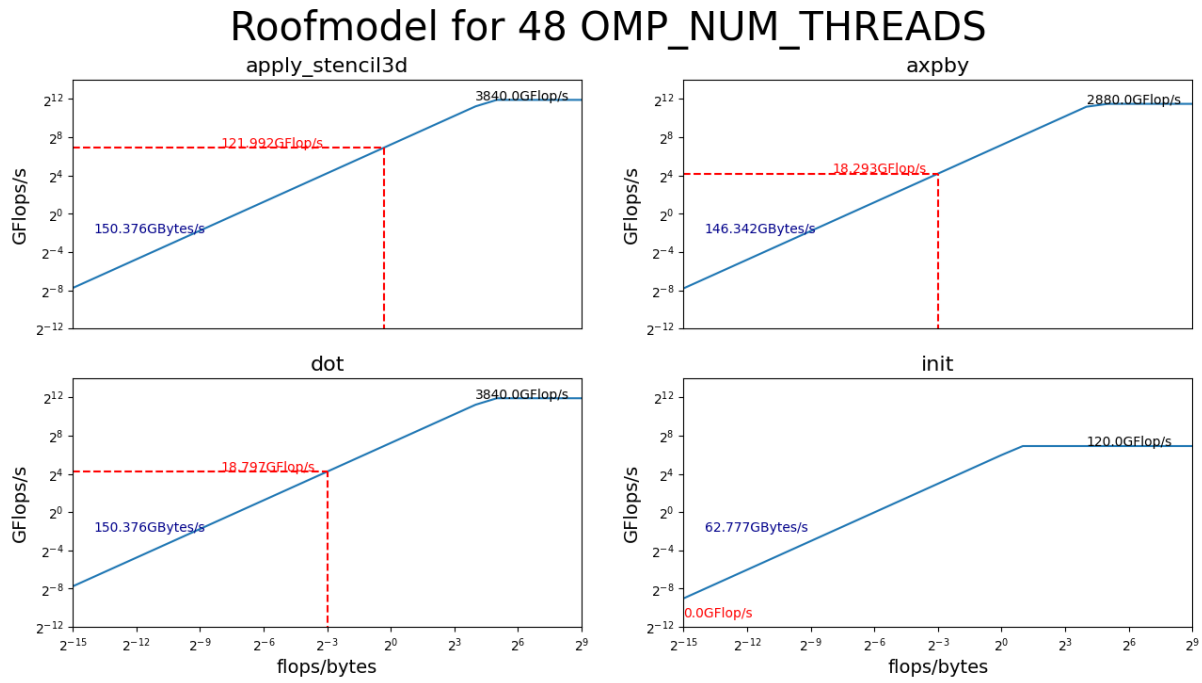

Figure 7: Roofmodel of the basis operations apply_stencil3d, axpby, dot and init, and number of threads equal to 48.

5 Task 5

We will now improve operations apply_stencil3d. Note that we use a lot of if-statements which slow down the code significantly. Therefore, we will rewrite this part and use less if-statements. One possibility is to break up the three nested loops over $(i = 1, \ldots, nx,\ j = 1, \ldots, ny,\ k = 1, \ldots, nz)$ with if-statements to multiple (smaller) nested loops which run in such a way in x, y, z-direction that we do not need any if-statements. Then we get one for-loop over $(i = 2, \ldots, nx-1,\ j = 2, \ldots, ny-1,\ k = 2, \ldots, nz-1)$, one over $(i = 1,\ j = 2, \ldots, ny-1,\ k = 2, \ldots, nz-1)$, one over $(i = nx,\ j = 2, \ldots, ny-1,\ k = 2, \ldots, nz-1)$, one over $(i = 2, \ldots, nx-1,\ j = 1,\ k = 2, \ldots, nz-1)$, one over $(i = 2, \ldots, nx-1,\ j = ny,\ k = 2, \ldots, nz-1)$, one over $(i = 2, \ldots, nx-1,\ j = 2, \ldots, ny-1,\ k = 1)$ and one over $(i = 2, \ldots, nx-1,\ j = 2, \ldots, ny-1,\ k = nz)$. However, we got similar results as before. Therefore, we also try adding compiler optimization flags. We try -funroll-loops. This results in a small performance improvement.

Then, having 12 threads, P equals $\min(96.6523388, 7.45816275) = 7.45816275$, $\min(13.2922725, 9.46073794) = 9.46073794$, $\min(14.8925025, 12.64859167) = 12.64859167$, $\min(0, 0) = 0$ Gflops/s for operations apply_stencil3d, axpby, dot and init, respectively. The absolute efficiencies are 0.0771648, 0.7117472 and 0.8493261 for operations apply_stencil3d, axpby, and dot, respectively. Then we find that the absolute efficiency for 12 threads is slightly increased for operations apply_stencil3d and dot, and decreased a bit for axpby.

And, having 48 threads, P equals $\min(121.9922998, 27.21147593) = 27.21147593$, $\min(18.2927525, 21.71501721) = 18.2927525$, $\min(18.796965, 27.12034654) = 18.796965$, $\min(0, 0) = 0$ Gflops/s for operations apply_stencil3d, axpby, dot and init, respectively. The absolute efficiencies are 0.2230590, 1 and 1 for operations apply_stencil3d, axpby, and dot, respectively. Then we find that the absolute efficiency for 48 threads for operation apply_stencil3d is increased with 1.1%. The absolute efficiency for operations axpby and dot remained the same, 100%.

6 Task 6

The total CG run time for the 600^3 problem on 12 threads is 64.91 seconds (0.75 seconds faster than in task 1) and on 48 threads is 22.65 seconds. The total runtime predicted by the Roofline model (likwid measurements) on 12 threads is $\frac{102 \cdot 3.45600007}{119.14002} + \frac{304 \cdot 5.184}{106.33818} + \frac{203 \cdot 3.456}{119.14002} + \frac{2 \cdot 1.728}{50.79416} = 23.7354835$ seconds and on 48 threads is $\frac{102 \cdot 3.45600007}{150.37572} + \frac{304 \cdot 5.184}{146.34202} + \frac{203 \cdot 3.456}{150.37572} + \frac{2 \cdot 1.728}{62.77732} = 17.833549$ seconds. These results are also denoted in Tables 2 and 3. Then we find that the best (predicted) time by likwid is on 12 threads about 2.73 times as fast than our code and on 48 threads about 1.27 times as fast than our code.

Table 2: Total CG runtime (seconds) of code and predicted by Likwid on 12 and 48 threads.

Number of OMP_NUM_THREADS	Total CG runtime own code (seconds)	Total CG runtime Likwid prediction (seconds)
12	64.91	23.7354835
48	22.65	17.833549

Table 3: Number of calls of operations apply_stencil3d, axpby, dot and init in cg_solver, with max_iter=100.

Basis function	Calls	GBytes	GBytes/s on 12 cores	GBytes/s on 48 cores
apply_stencil3d	102	3.45600007	119.14002	150.37572
axpby	304	5.184	106.33818	146.34202
dot	203	3.456	119.14002	150.37572
init	2	1.728	50.79416	62.77732