FDD3258 Assignment 1

Anton Jansen, Michele Pellegrino

September 14, 2020

1 Modeling sparse matrix-vector multiply

1. If we do not consider data movement, we compute performance based on FLOPS and CPU speed i.e. $t = \text{nnz} \times 2c$ for our code. On my local machine (Ryzen 7 3800X @ 3.9 GHz):
   - $\text{nrows} = 10^2 \Rightarrow \text{nnz} = 5 \times 10^2 \Rightarrow t = (5 \times 10^2)(2)(3.9 \times 10^9)^{-1} = 2.56 \times 10^{-7}$ s
   - $\text{nrows} = 10^4 \Rightarrow \text{nnz} = 5 \times 10^4 \Rightarrow t = (5 \times 10^4)(2)(3.9 \times 10^9)^{-1} = 2.56 \times 10^{-5}$ s
   - $\text{nrows} = 10^6 \Rightarrow \text{nnz} = 5 \times 10^6 \Rightarrow t = (5 \times 10^6)(2)(3.9 \times 10^9)^{-1} = 2.56 \times 10^{-3}$ s
   - $\text{nrows} = 10^8 \Rightarrow \text{nnz} = 5 \times 10^8 \Rightarrow t = (5 \times 10^8)(2)(3.9 \times 10^9)^{-1} = 2.56 \times 10^{-1}$ s
2.
   - $\text{nrows} = 10^2 \Rightarrow t_{\text{obs}} = 1 \times 10^{-6}$ s
   - $\text{nrows} = 10^4 \Rightarrow t_{\text{obs}} = 7.3 \times 10^{-5}$ s
   - $\text{nrows} = 10^6 \Rightarrow t_{\text{obs}} = 5.6 \times 10^{-3}$ s
   - $\text{nrows} = 10^8 \Rightarrow t_{\text{obs}} = 5.4 \times 10^{-1}$ s

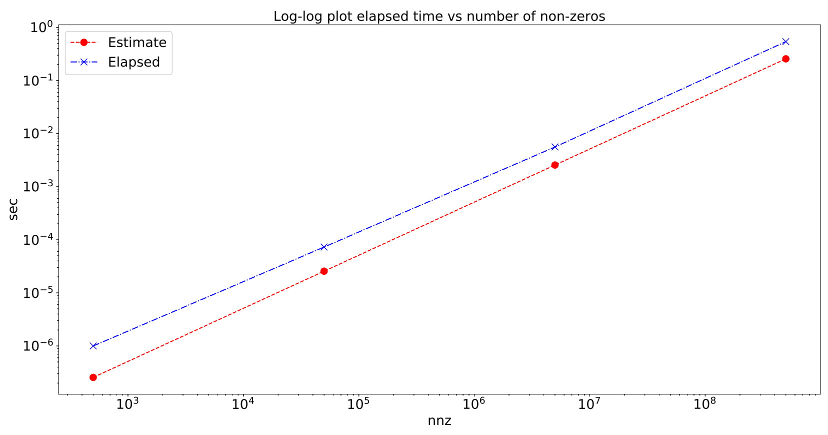

Figure 1: Logarithmic plot of the elapsed time (estimate and actual) versus the number of non-zero elements in the matrix.

3. One important reason is that memory is **not** infinitely large or infinitely fast.
4. Memory bandwidth (bits/seconds) is the hint (total number of bits loaded) divided by t_{obs}:
   - $\text{nrows} = 10^2 \Rightarrow 7046$ Gbit/s
   - $\text{nrows} = 10^4 \Rightarrow 8310$ Gbit/s
   - $\text{nrows} = 10^6 \Rightarrow 12341$ Gbit/s
   - $\text{nrows} = 10^8 \Rightarrow 13320$ Gbit/s
5. The bandwith obtained with the STREAM benchmark is only ~ 171 Gbit/s so there is a difference of a factor 50-100 (the numbers at question 4 are too high because they do not account for cache).

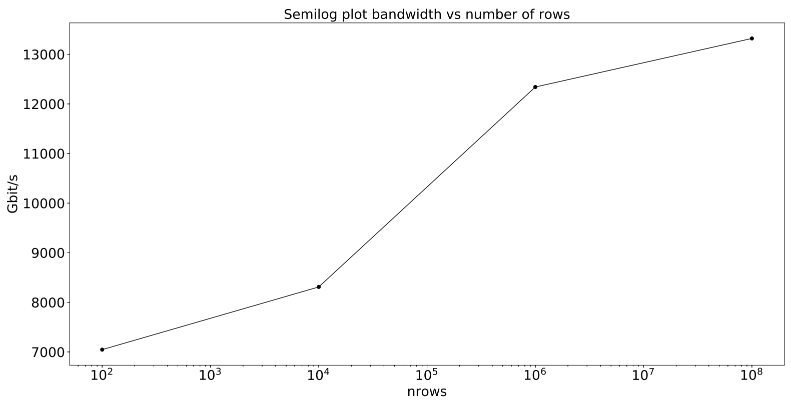

Figure 2: Semi-logarithmic plot of the memory bandwidth versus the number of rows.

2 The memory mountain

1. AMD Ryzen 7 3800x @ 3.9 GHz. L1 is 32 KB, L2 is 512 KB, L3 is 32 MB.
2. See figure 3.
3. An array size of 16 KB with a stride of 1-4 (x8 bytes). Bandwidth around 40 GB/s.
4. An array size of 16-128 MB with a stride of 6-16 (x8 bytes). Bandwidth around 5 GB/s.
5. Even though the larger array sizes indeed do not fit into (even) the L3 cache, a stride=1 still offers better performance because at smaller strides it is "easier" for the prefetcher to guess which parts of the array are going to be needed in the future.
6. Temporal locality: data is accessed (reused) by the CPU multiple times over a *very short* time frame (i.a. local in time). Spatial locality: data that are stored relatively close together (e.g. two ints that are next to each other in an array are next to each other in memory). Locality is a desirable property as it provides a certain predictability that can exploited by CPU catching and prefetching to increase performance.
7. Increased array size will decrease temporal locality because as you increase the array size, at a certain point it won't all fit in (the same level) cache anymore.
8. Increased read stride will decrease spatial locality.
9. We repeat the analysis on Beskow. The overall shape of the memory mountain remains qualitatively similar, however some interesting features can be spotted:
   - while peak performance is still obtained around array sizes of 16KB and strides of 8 bites, now the plateau of lower performance is close to the maximum stride/size combination tested (32×8 bites, 128MB);
   - in general, performance for smaller sizes is higher and seems more consistent across different strides (less spikes due to noisy results).

3 Write a benchmark to measure performance

3.1 Compile the program with optimization flag `-O2` and execute it

1. Average execution time is 9.5×10^{-7} s (for N = 5000).
2. For N = 100,000 the average execution time is still 9.5×10^{-7} s.
3. Execution time is like that because the compiler is smart and sees that although we declare and initiate very large arrays, we don't actually use them for anything: the compiler therefore optimizes these instructions (essentially removes them). This can easily be verified by adding another `for` loop, outside the timed one, just for printing vector `c`: `t2-t1` increases now to $10^{-5} \div 10^{-4}$ seconds. We can prevent the compiler from doing this by removing the `-O` (optimization level) flag, or setting the `-O0` flag (useful for debugging purposes).

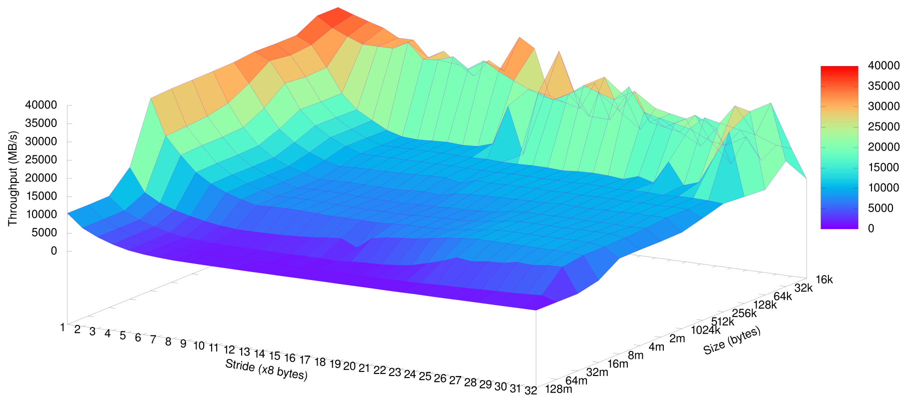

Figure 3: The memory mountain for the AMD Ryzen 7 3800x. L1 is 32 KB, L2 is 512 KB, L3 is 32 MB.

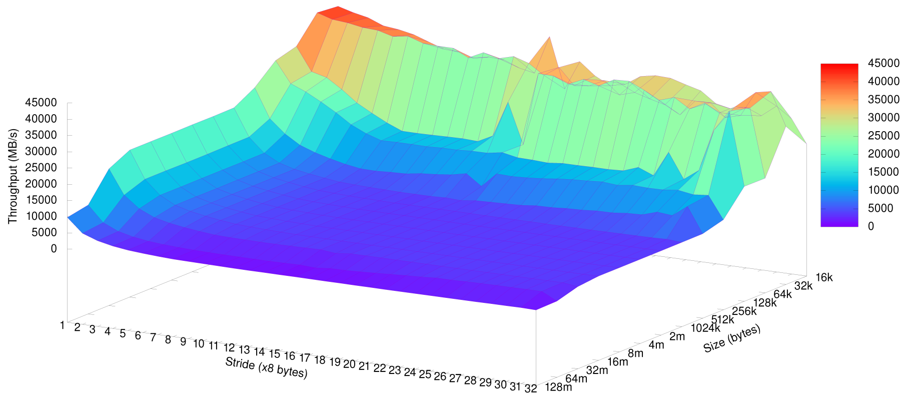

Figure 4: The memory mountain obtained by running on Beskow (1 Haswell CPU, cache: L1 64KB, L2 256KB, L3 45MB).

4. Execution time is 2×10^{-5} s for N = 5000, and 2×10^{-4} s for N = 100,000.

3.2 Check the tick (clock granularity) on Beskow or your local computer

1. Clock granularity (local machine) is 9.54×10^{-7} s (using clockgranularity.c).
2. Theoretically, since we are directly measuring the amount of clock cycles for a certain operation, the granularity would be $1/\text{CPUspeed} = (3.9 \times 10^9 \text{ Hz})^{-1} \approx 2.6 \times 10^{-10}$ s. However, measuring "nothing at all" in the code already gives an average of 122.8 cycles, so we might need to subtract this when we bench an actual operation?

3.3 Modify the program in 1.1

1. Average run time is now 2.4×10^{-6} s for $N = 5000$, and 4.8×10^{-5} s for $N = 100000$. Averaged over 100,000 runs.

Benchmark	naive N=64	naive N=1000	optimized N=64	optimized N=1000
Elapsed [sec]	0.002556	20.402589	0.001918	6.724911
IPC	2.223877	1.088776	2.975347	2.936540
L1 cache miss ratio	0.114576	0.609410	0.019447	0.062875
L1 c. m. r. PTI	32.401968	174.017899	5.521993	17.959024
LLC cache miss ratio	0.070411	0.249709	0.088393	0.231928
LLC c. m. r. PTI	0.011512	4.830218	0.013629	0.160214

Table 1: Dell laptop, Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz, averaged over `-r 10` tests.

Benchmark	naive N=64	naive N=1000	optimized N=64	optimized N=1000
Elapsed [sec]	0.009906	18.106535	0.007400	5.845140
IPC	1.241768	1.051635	1.951368	4.107198
L1 cache miss ratio	0.077857	0.602505	0.024028	0.062764
L1 c. m. r. PTI	30.049127	200.688763	8.736777	17.916551
LLC cache miss ratio	N/D	0.000013	N/D	0.000103
LLC c. m. r. PTI	N/D	0.000279	N/D	0.000059

Table 2: Beskow, 1 Haswell CPU, average over `-r 5` tests.

4 Measure the cache usage in matrix-matrix multiply

We utilize PERF tool to monitor the performance of dense matrix-matrix multiplication code both on our local machine and on Beskow; in the latter case, the bash comand to run `perf` and store results is:

```
srun -n 1 perf stat -e instructions,cycles,L1-dcache-load-misses,L1-dcache-loads,
  LLC-load-misses,LLC-loads ./matrix_multiply_naive_64.out > perf.out 2> perf.err
```

Results for elapsed time, instructions per cycle and cache misses is reported on table 1, 2.
The main factors hindering the performance of the matrix multiply could be summarized as follow.

- C arrays are arranged in row-major order; the naive implementation fetches a value from a different row of `matrix_b` at each inner-most loop iteration and therefore fails to keep spacial locality, making inefficient use of the L1 data cache; indeed, the L1 cache misses in the naive case are one order of magnitude larger than the ones in the optimized code.
- Temporal locality is not ensured in the case of multuplication of *dense* matrices; in general we observe an higher miss ratio (both for L1 and LLC) for N=1000, regardless on optimization. Given the large amount of L3 cache on Haswell CPUs, we no not observe any LLC-miss in the N=64 case for the tests run on Beskow.
- Neither the naive code nor the presented optimized code utilize *loop unrolling* tecniques; only one value of `matrix_r` ir reused in each inner-loop iteration, leading to relatively inefficient cache usage.

5 Vectorization

- To enable vectorization for `gcc`, one has to set the `-O2` flag and the `-ftree-vectorize` flag. Optionally, one can just set the `-O3` flag as this includes the `-ftree-vectorize` flag (among others).
- To obtain a compiler report on vectorization, we set the `-fopt-info-vec` and `-fopt-info-vec-missed` flags. These will gives us reports on succesful and failed vectorization attempts, respectively.

-
  ```
  gcc -O3 -fopt-info-vec -fopt-info-vec-missed matrix_multiply.c
      -o matrix_multiply.out
  ```
- Vectorization report for the command above:

  ```
  matrix_multiply.c:18:19: missed: statement clobbers memory: gettimeofday (&tp, &tzp);
  matrix_multiply.c:26:18: missed: couldn't vectorize loop
  matrix_multiply.c:28:37: missed: statement clobbers memory: _1 = rand ();
  matrix_multiply.c:27:20: missed: couldn't vectorize loop
  matrix_multiply.c:28:37: missed: statement clobbers memory: _1 = rand ();
  matrix_multiply.c:28:37: missed: statement clobbers memory: _1 = rand ();
  matrix_multiply.c:29:37: missed: statement clobbers memory: _4 = rand ();
  matrix_multiply.c:39:18: missed: couldn't vectorize loop
  matrix_multiply.c:39:18: missed: not vectorized: multiple nested loops.
  matrix_multiply.c:40:20: optimized: loop vectorized using 16 byte vectors
  matrix_multiply.c:41:22: missed: couldn't vectorize loop
  matrix_multiply.c:41:22: missed: outer-loop already vectorized.
  matrix_multiply.c:52:18: missed: couldn't vectorize loop
  matrix_multiply.c:54:27: missed: not vectorized: complicated access pattern.
  matrix_multiply.c:53:20: optimized: loop vectorized using 16 byte vectors
  matrix_multiply.c:52:18: missed: couldn't vectorize loop
  matrix_multiply.c:54:27: missed: not vectorized: complicated access pattern.
  matrix_multiply.c:53:20: optimized: loop vectorized using 16 byte vectors
  matrix_multiply.c:71:17: missed: couldn't vectorize loop
  matrix_multiply.c:71:17: missed: not vectorized: multiple nested loops.
  matrix_multiply.c:39:18: missed: couldn't vectorize loop
  matrix_multiply.c:39:18: missed: not vectorized: multiple nested loops.
  matrix_multiply.c:40:20: optimized: loop vectorized using 16 byte vectors
  matrix_multiply.c:41:22: missed: couldn't vectorize loop
  matrix_multiply.c:41:22: missed: outer-loop already vectorized.
  matrix_multiply.c:39:18: missed: couldn't vectorize loop
  matrix_multiply.c:39:18: missed: not vectorized: multiple nested loops.
  matrix_multiply.c:40:20: optimized: loop vectorized using 16 byte vectors
  matrix_multiply.c:41:22: missed: couldn't vectorize loop
  matrix_multiply.c:41:22: missed: outer-loop already vectorized.
  ```

  If we look at the report, there are two reasons why vectorization fails, either:

  – there are multiple (more than two) nested loops,
  – the access pattern is too complicated.

  Both boil down to the fact that for complicated situations (e.g. vectorization of non-primitive types) the compiler simply does not know how to do it and/or the architecture does not allow it.