Homework 1: Solutions Dominik Gresch

1 Cost analysis

a)

Without knowledge of the cost of each floating point operation, one has to count them separately. Let $c(\mathrm{op})$ be the cost of an operation, $n(\mathrm{op}, N)$ the number of times this operation is called (for problem size N).

$$\Rightarrow C(N) = \sum_{\mathrm{op}} c(\mathrm{op}) \cdot n(\mathrm{op}, N)$$

b)

In this example, $\mathrm{op} \in \{-, *, /, \mathrm{sqrt}\}$:

$$\begin{aligned} n(\mathrm{sqrt}, N) &= \sum_{j=0}^{N-2} 1 = N - 1 \\ n(/, N) &= \sum_{j=0}^{N-2} \sum_{i=j+1}^{N-1} 1 = \frac{1}{2}(N^2 - N) \\ n(-, N) = n(+, N) &= \sum_{j=0}^{N-2} \sum_{k=j+1}^{N-1} \sum_{i=k}^{N-1} 1 = \frac{1}{6}(N^3 - N) \end{aligned}$$

$$\begin{aligned} \Rightarrow C(N) &= \underline{\underline{\frac{1}{6}(N^3 - N)(c(-) + c(*)) + \frac{1}{2}(N^2 - N)c(/) + (N-1)c(\mathrm{sqrt})}} \\ &\in \frac{1}{6}N^3(c(-) + c(*)) + \frac{1}{2}N^2 c(/) + Nc(\mathrm{sqrt}) + \mathcal{O}(N) \end{aligned}$$

c)

assuming $c(*) = c(-) = c(/) = c(\mathrm{sqrt}) =: c$

$$\begin{aligned} \Rightarrow C(N) &= \underline{\underline{c \cdot \left(\frac{1}{3}N^3 + \frac{1}{2}N^2 + \frac{1}{6}N - 1\right)}} \\ &\in \frac{1}{3}cN^3 + \mathcal{O}(N^2) \end{aligned}$$

Exercise 2: Get to know your machine

a)[1]	Intel, Core 2 Duo, T9900
b)[1]	2 Cores
c)[1]	3.06 GHz (peak)
d)[2]	latency: 3, cycles/issue: 1 (single & double precision)
e)[2]	latency: 5, cycles/issue: 1 (double precision) latency: 4, cycles/issue: 1 (single precision)
f)	$\underline{\underline{\text{2 flop/cycle}}}$ (1 add, 1 mult) at 3.06 GHz ⇒ $\underline{\underline{\text{6.12 Gflop/s}}}$

The CPU tested in reference [2] isn't exactly the same, but it is also a model 23 (see p. 6) Intel Core 2 Duo.

[1] from `cat /proc/cpuinfo`

[2] from www.agner.org/optimize/instruction_tables.pdf, p.145 (no info available on intel.com)

Exercise 3: MMM

b)

The compute() function uses:

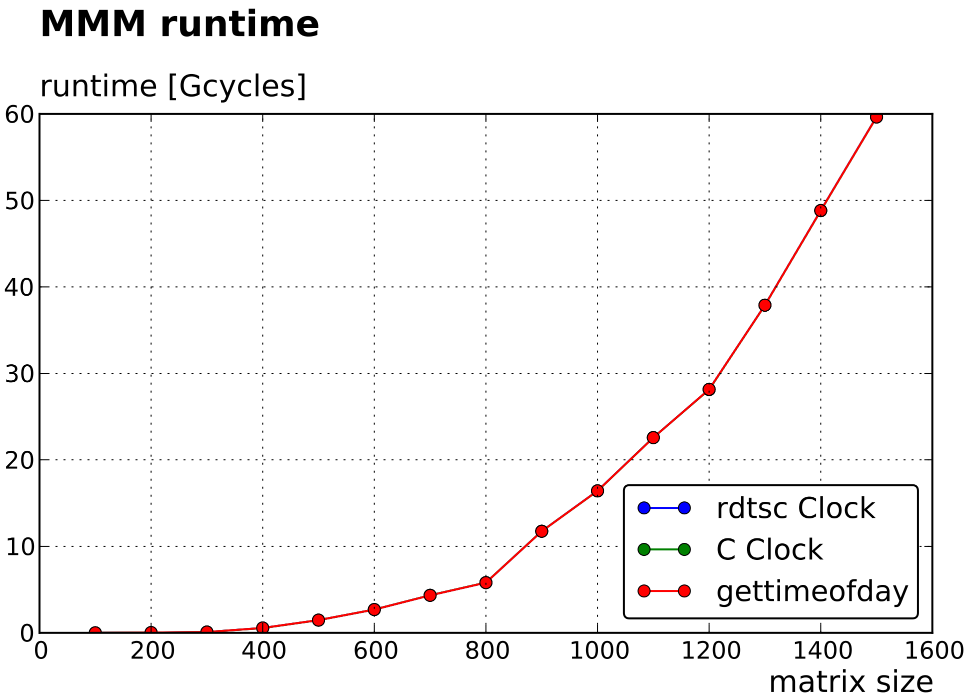
$$m \cdot n \cdot k \,\text{add} + \ m \cdot n \cdot k \,\text{mult} = 2 \cdot m \cdot n \cdot k \,\text{flop}$$

e)

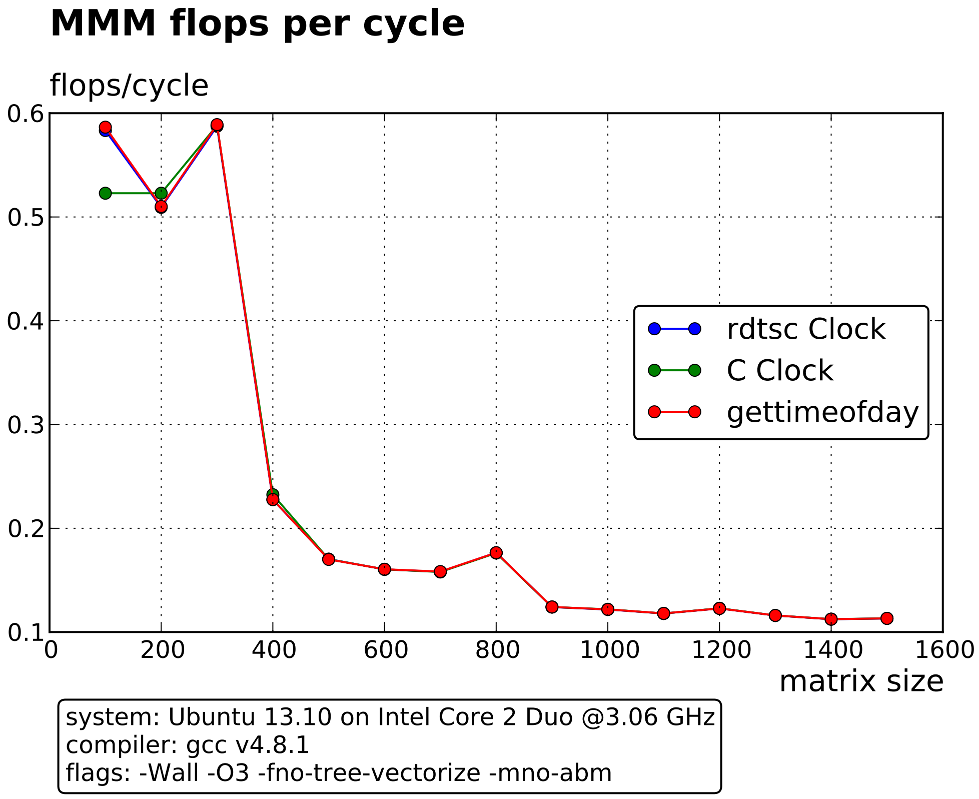
In all the plots, one can see that the different timing methods do not differ significantly (the lines from each method are nearly indistinguishable). Plotting the runtime in cycles, one can see approximately cubic growth in runtime (or rather - one can see some growth in runtime which could well be cubic).

The other plots are more meaningful: they show two significant steps where performance goes down. Those are most likely due to the matrix not fitting into L1 / L2 cache respectively. For small enough matrices, one is still at reasonable $25 - 30\%$ peak performance, but this goes down dramatically for larger matrices ($\sim 6\%$ at $N = 1500$).

Remark: The mmm.c file is to be found in `ex03/src`

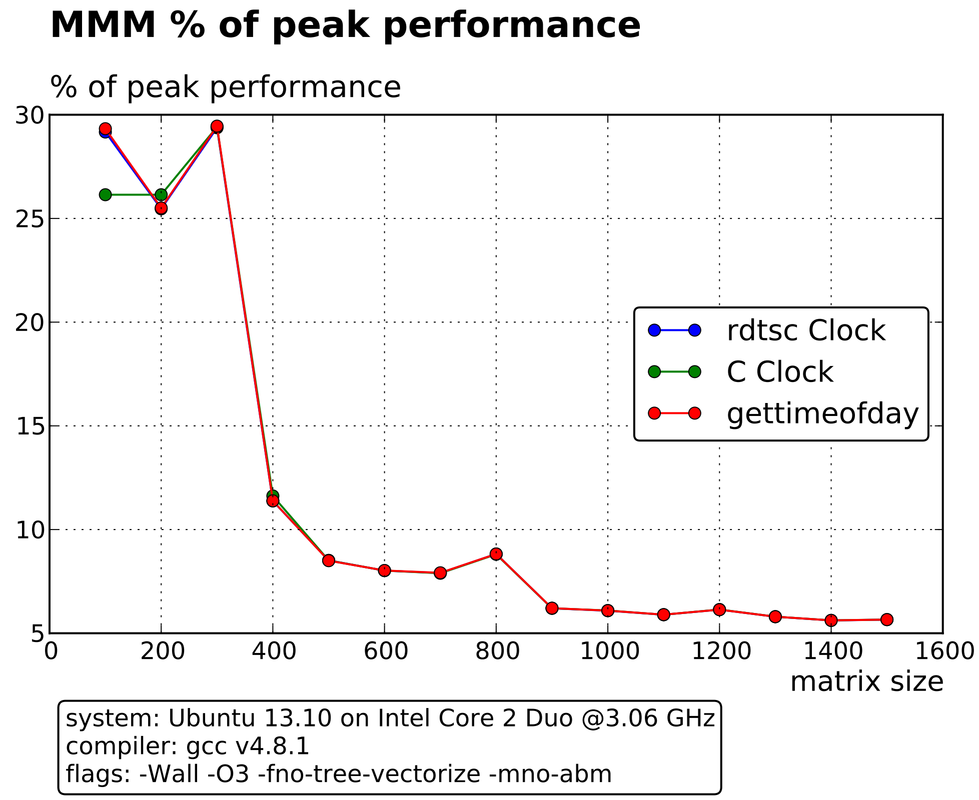


system: Ubuntu 13.10 on Intel Core 2 Duo @3.06 GHz
compiler: gcc v4.8.1
flags: -Wall -O3 -fno-tree-vectorize -mno-abm



MMM flops per cycle

system: Ubuntu 13.10 on Intel Core 2 Duo @3.06 GHz
compiler: gcc v4.8.1
flags: -Wall -O3 -fno-tree-vectorize -mno-abm



MMM % of peak performance

system: Ubuntu 13.10 on Intel Core 2 Duo @3.06 GHz
compiler: gcc v4.8.1
flags: -Wall -O3 -fno-tree-vectorize -mno-abm

Exercise 4: MVM

a)

I have used the exact structure from Ex. 3, but removed the other timing methods and changed the compute() function.

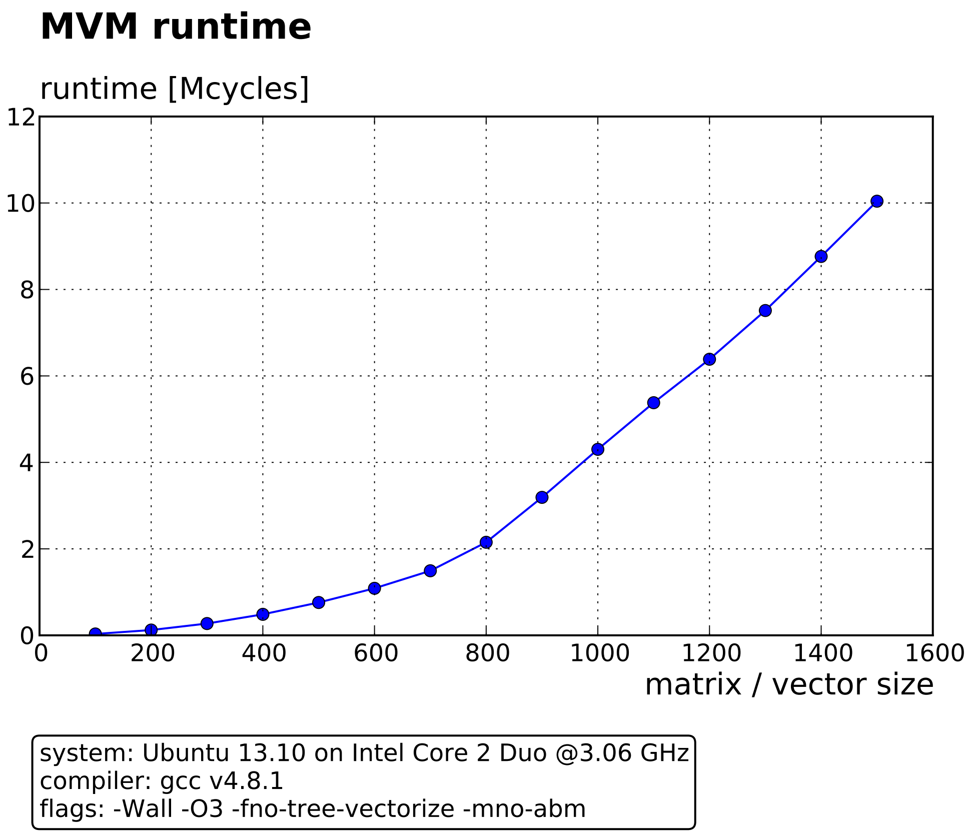
Remark: mvm.c can be found in `ex04/src/`

b)

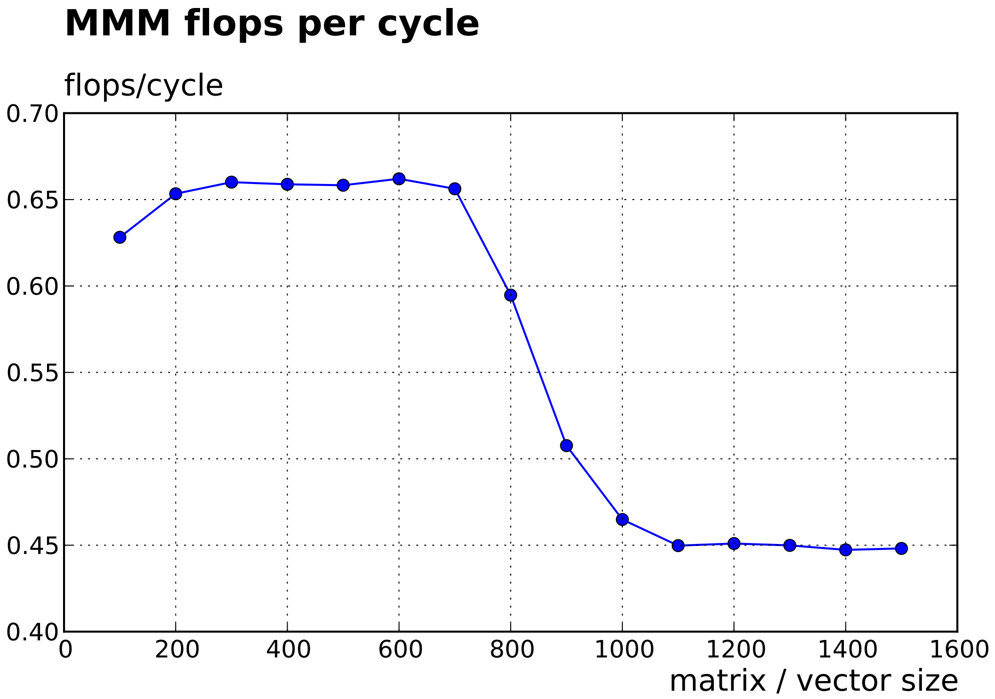
The compute() function uses:

$$m \cdot n \,\text{add} + m \cdot n \,\text{mult} = 2 \cdot m \cdot n \,\text{flop}$$

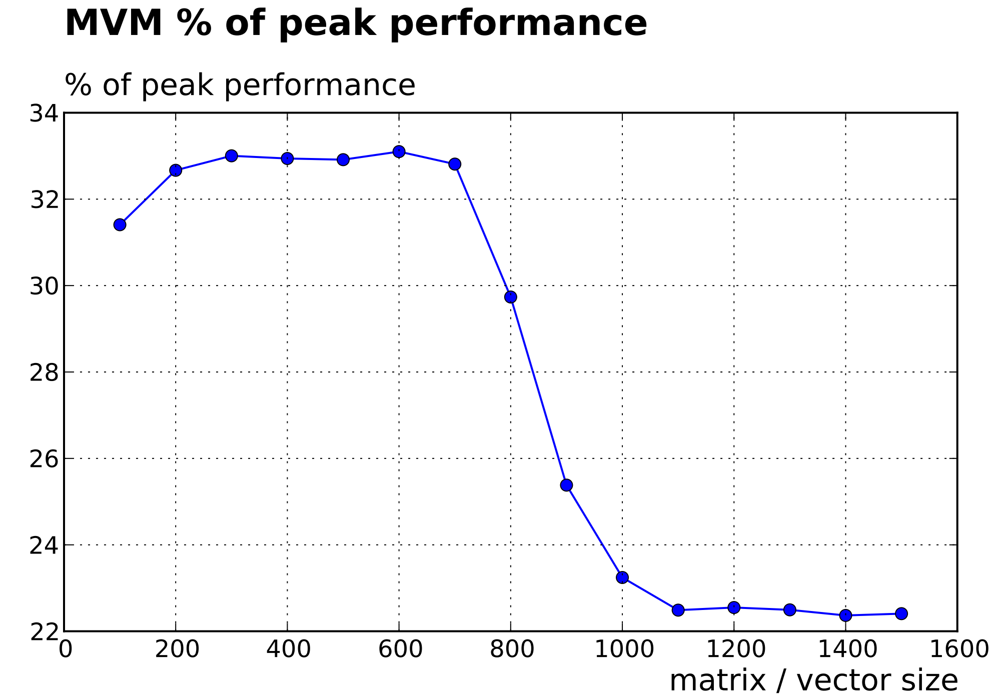
c)



system: Ubuntu 13.10 on Intel Core 2 Duo @3.06 GHz
compiler: gcc v4.8.1
flags: -Wall -O3 -fno-tree-vectorize -mno-abm



system: Ubuntu 13.10 on Intel Core 2 Duo @3.06 GHz
compiler: gcc v4.8.1
flags: -Wall -O3 -fno-tree-vectorize -mno-abm



system: Ubuntu 13.10 on Intel Core 2 Duo @3.06 GHz
compiler: gcc v4.8.1
flags: -Wall -O3 -fno-tree-vectorize -mno-abm

d)

At smaller N, both MMM and MVM start out at about 34% peak performance (both compiled at $-$O3). The MVM, too, shows a dip in performance, but it is at much larger N and the drop is less significant. This might be due to the fact that in the MVM compute() - function, the matrix (the part which - in memory size - is N^3) is accessed only once, and consecutively.

Exercise 6: Bounds

a)

$9\,\text{mult and } 8\,\text{add per innermost loop} \Rightarrow 17 \cdot (N-1)^2\,\text{flop} = W(N)$

b)

reads: $\geq N^2$ (size of G) $+9$ (size of h) doubles $= 8N^2 + 72\,\text{bytes}$
writes: $(N-1)^2$ (one for each innermost loop) $= 8(N-1)^2\,\text{bytes}$

$$\begin{aligned} Q(N) =& 16N^2 - 16N + 80\,\text{bytes} \\ I(N) =& \frac{W(N)}{Q(N)} = \frac{17(N-1)^2}{16N^2-16N+80} \approx \frac{17}{16} \end{aligned}$$

the approximation is exact for large enough N.

c)

i) Hard lower bound (not taking into account dependencies):

   for adds (latency: 3, throughput: 1): $8 \cdot (N-1)^2 + 2\,\text{cycles}$
   for mults (latency: 5, throughput: 1): $9 \cdot (N-1)^2 + 4\,\text{cycles}$

   together: $\underline{9 \cdot (N-1)^2 + 4\,\text{cycles}}$ (limited by mults)

ii) Numer of cycles used to load everything into memory for latency l and throughput tp (round up because you can only count full cycles): $\lceil \frac{Q(N)-1}{tp} + l \rceil$

   in L1 (latency: 4, throughput: 4): $4N^2 - 4N + 24\,\text{cycles}$
   in L2 (latency: 12, throughput: 4): $4N^2 - 4N + 32\,\text{cycles}$
   in RAM (assuming latency: 100, throughput: 1/4): $64N^2 - 64N + 116\,\text{cycles}$