HowTo Write Fast Numerical Code
Exercise 1

Pascal Spörri
pascal@spoerri.io

March 7, 2013

1 Cost Analysis

We consider the matlab function given in the exercise sheet (listing 1) provided on the course homepage:

```
function z = func (x1, x2, y)
    m = length(x1);
    n = length(y);
    if m == 1
        z = x1(1)+sum(y);
    return
    k = m/2;
    t = func(x1(1:k), x2(1:k), y)*func(x1(k+1:m), x2(k+1:m), y);
    x3 = x1+x2;
    y1 = pi*y;
    z = t*func(x3(1:k), x3(k+1:m), y1);
end
```

Listing 1: Matlab code given in the exercise

1.1 Floating Point Additions

We observe that there are in total $A_1 = n$ floating point additions for the case $length(x1) = m = 1$ and

$$A_m = 3 \cdot A_{m/2} + m \tag{1}$$

floating point additions per recursion step. Using the formula provided in the course we are able to solve the recursion by first substituting m with 2^k,

$$F_k = 3 \cdot F_{k-1} + 2^k. \tag{2}$$

Then solving the recursion:

$$F_k = 3^k n + \sum_{i=0}^{k-1} 3^i \cdot 2^{k-i} = 3^k n - 2(2^k - 3^k). \tag{3}$$

Doing a back substitution gives us a total amount of floating point additions:

$$A_m = 3^{\log_2 m} n + 2(m - 3^{\log_2 m}). \tag{4}$$

1.2 Floating Point Multiplications

We observe that there are in total $M_1 = 0$ floating point additions for the case $length(x1) = m = 1$ and

$$M_m = 3 \cdot M_{m/2} + n + 2 \tag{5}$$

floating point multiplications per recursion step. Using the formula provided in the course we are able to solve the recursion:

$$\begin{aligned}
G_0 &= 0, && (6)\\
G_k &= 3 \cdot G_{k-1} + n + 2 && \text{Substitution of } m \text{ with } 2^k\\
&= 3^k \cdot 0 + \sum_{i=0}^{k-1} 3^i \cdot (2 + n)\\
&= \sum_{i=0}^{k-1} 3^i \cdot (2 + n) = \frac{1}{2}\left(3^k - 1\right)(n + 2). && (7)
\end{aligned}$$

By substituting back we get the total amount of floating point operations:

$$M_m = \frac{1}{2}\left(3^{\log_2 k} - 1\right)(n + 2). \tag{8}$$

1.3 Total Floating Point Operations

Adding the equations (4) and (7) together we get the total amount of floating point operations in listing 1:

$$\begin{aligned}
C_m &= A_m + M_m\\
&= 3^{\log_2 m} n + 2(m - 3^{\log_2 m}) + \frac{1}{2}\left(3^{\log_2 k} - 1\right)(n + 2). && (9)
\end{aligned}$$

2 Machine Information

The computer is running a Mac OSX version 10.8.2 using a Intel Core i7-3720QM CPU using a frequency of 2.60 GHz per core. There are 4 cores (8 threads) available (Datasheet[1]).

Using an Intel Sandy Bridge architecture slideset[2] and the architecture slides[3] we are able to compute the available floating point operations:

$$\begin{aligned}
\text{Floating Point additions/cycle:} &\quad 1\\
\text{Floating Point multiplications/cycle:} &\quad 1\\
\text{GFlops/s:} &\quad 2.6GHz \cdot 2\text{Flops/cycle}\\
&= 5.6 \text{ GFlops/s per Core}
\end{aligned}$$

[1] http://ark.intel.com/products/64891

[2] https://www.cesga.es/gl/paginas/descargaDocumento/id/135

[3] http://www.inf.ethz.ch/personal/markusp/teaching/263-2300-ETH-spring13/slides/arch.pdf

3 Matrix Multiplication

In order to get accurate timings on OSX we installed and activated the DisableTurboBoost kernel module[4] on the computer. The `sysctl` now shows:

```
# sysctl -a hw | grep cpufrequency
hw.cpufrequency: 2600000000
hw.cpufrequency_min: 2600000000
hw.cpufrequency_max: 2600000000
hw.cpufrequency = 2600000000
```

Listing 2: Bash output of the sysctl command

The code shows $n \cdot m \cdot k$ floating point adds and $n \cdot m \cdot k$ floating point multiplications. **Total Floating Point operations: 2nmk**.

In order to get consistent timings we had to set the frequency define in `mmm.c` to `2.6e9`. Resulting in the following output:

```
m=1000 k=1000 n=1000
RDTSC instruction:
 18399246918.000000 cycles measured => 7.076633 seconds,
    assuming frequency is 2600.000000 MHz. (change in source
    file if different)

C clock() function:
 7117748.000000 cycles measured. On some systems, this number
    seems to be actually computed from a timer in seconds then
    transformed into clock ticks using the variable
    CLOCKS_PER_SEC. Unfortunately, it appears that
    CLOCKS_PER_SEC is sometimes set improperly. (According to
    this variable, your computer should be running at 1.000000
    MHz). In any case, dividing by this value should give a
    correct timing: 7.117748 seconds.

C gettimeofday() function:
 7.117292 seconds measured
```

Listing 3: Execution of the `mmm.c` code compiled with GCC 4.7.2 and the flags `-O3 -m64 -fno-tree-vectorize`.

[4] https://github.com/nanoant/DisableTurboBoost.kext

3.1 Plots

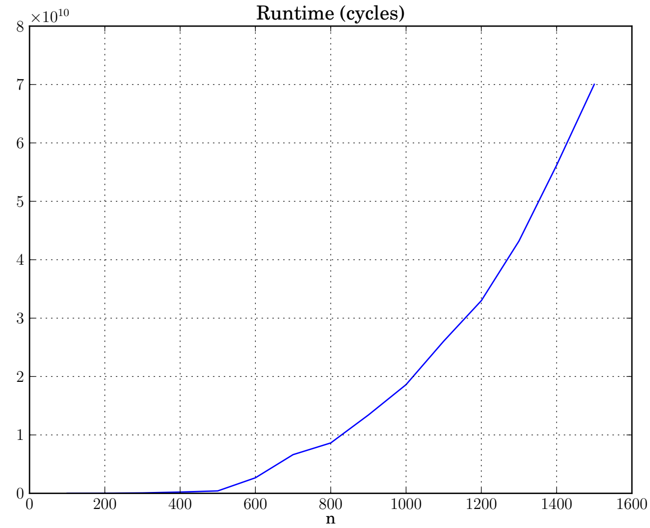

Figure 1: Runtime of `mmm.c` in cycles. Compiler: GCC 4.7.2, Flags: `-O3 -m64 -fno-tree-vectorize`.

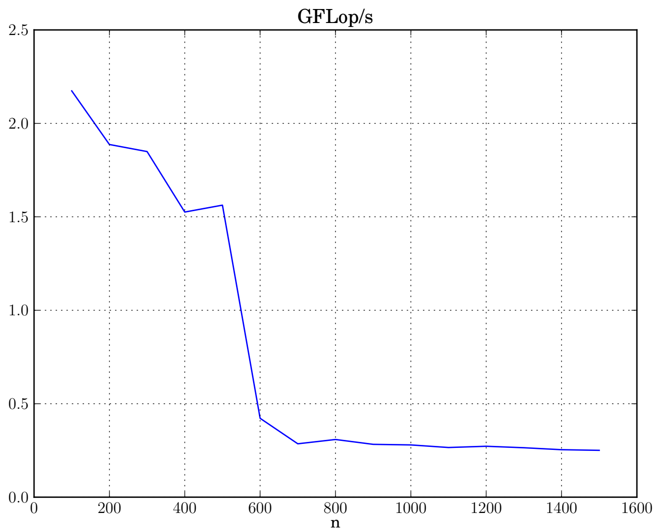

Figure 2: Achieved GFlop/s of `mmm.c` in cycles. Compiler: GCC 4.7.2, Flags: `-O3 -m64 -fno-tree-vectorize`.

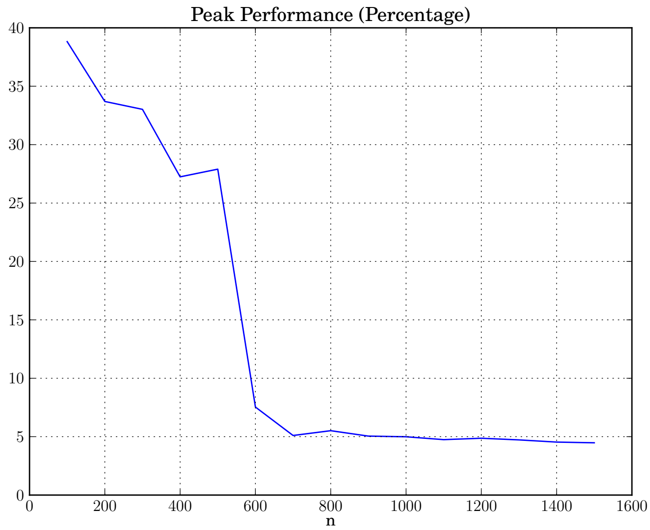

Figure 3: Achieved peak performance (percentage) of `mmm.c` in cycles. Compiler: GCC 4.7.2, Flags: `-O3 -m64 -fno-tree-vectorize`.

We observe that we don't manage to use the utilise all the available floating point units.

4 Daxpy

The implementation of the "daxpy" function was straightforward. We copied the `mmm.c` code and replaced the matrix multiplication with the vector addition.

4.1 Plots

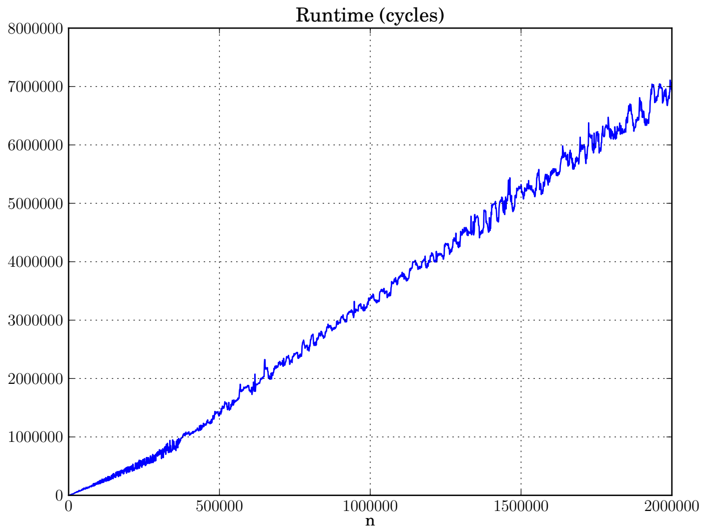

Figure 4: Runtime of `daxpy.c` in cycles. Compiler: GCC 4.7.2, Flags: `-O3 -m64 -fno-tree-vectorize`.

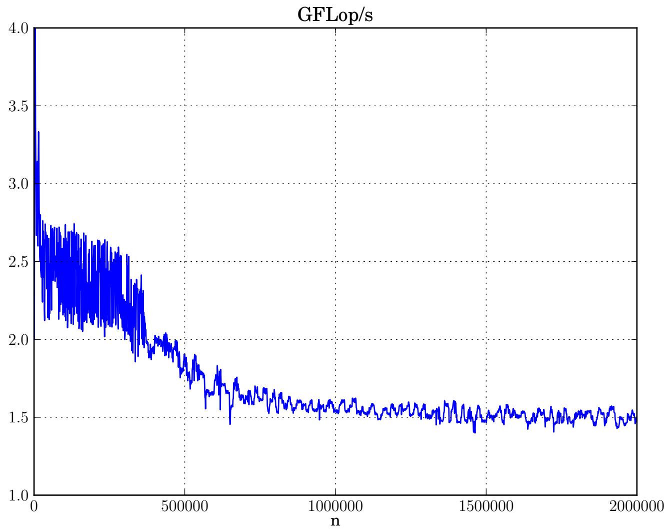

Figure 5: Achieved GFlop/s of `daxpy.c` in cycles. Compiler: GCC 4.7.2, Flags: `-O3 -m64 -fno-tree-vectorize`.

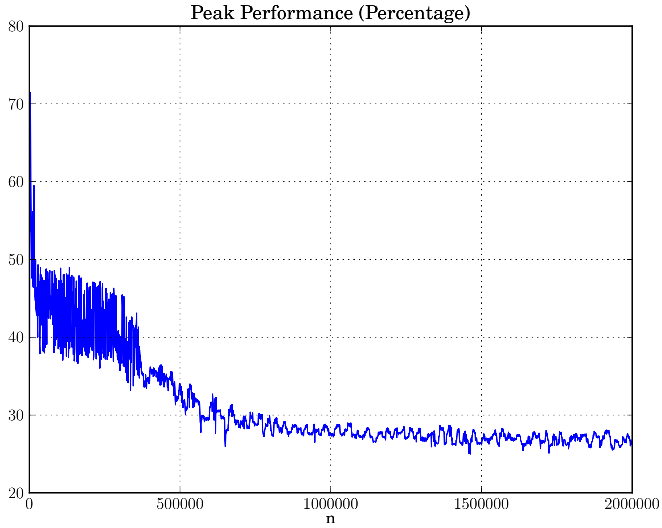

Figure 6: Achieved peak performance (percentage) of `daxpy.c` in cycles. Compiler: GCC 4.7.2, Flags: `-O3 -m64 -fno-tree-vectorize`.

We notice that we were able to utilize 72% of the available floating point peak performance using a vector length of $n = 1000$.

5 Bounds