High Performance Computing for Science and Engineering I.

Exercise Set 1

Florian Mahlknecht

2019-10-11

1 Operational Intensity (Q3)

1.1 Asymptotic bounds for linear algebra operations

Given that the operational itensity I is

$$I = \frac{W}{Q} \quad [\text{flop/byte}] \tag{1}$$

we just need to define how many operations and how much data we need for each of the operations. Note that for each of the following examples the following assumptions are made:

- at the start of the operation the cache is flushed, meaning that we do not use any data which was already loaded into cache.
- the floating point operations (flop) are unaffected from double / single precision meaning that we do not count them twice in case of double precision
- multiplications and additions are assumed to be both floating point operations with the same computational cost
- all other operations, pointer arithmetic and index increments are assumed to be negligible and not to be performed as floating point operations (given that they are integers)

1.1.1 DAXPY

$$\mathbf{y} = \alpha\,\mathbf{x} + \mathbf{y} \qquad \mathbf{x}, \mathbf{y} \in \mathbb{R}^n,\ \alpha \in \mathbb{R} \tag{2}$$

This operation includes n multiplications and n summations, so a total of $2n$ flops. The memory involved is composed of the two vectors **x** and **y** and the real number α, so a total of $(2n+1)$ double precision numbers, or $8\,(2n+1)$ bytes. Note that the resulting **y** needs also to be written back to memory, which makes a total of $8\,(3n+1)$ bytes involved in memory operations

The resulting operational intensity is therefore simply

$$I(n) = \frac{2n}{8(3n+1)}; \qquad \lim_{n\to\infty} I(n) = \frac{1}{12} \tag{3}$$

For this to hold, we need to state the following assumptions:

- α always remains in cache
- Each element of the vectors **x** and **y** is loaded into cache only once. This means that in case the data does not fit in cache from start, equally portions are loaded from both vectors and computations are performed, without having ever to reload portions twice.

1.1.2 SGEMV

In this case, we multiply the vector **x** by a *matrix* A.

$$\mathbf{y} = A\mathbf{x} + \mathbf{y} \qquad \mathbf{x}, \mathbf{y} \in \mathbb{R}^n,\ A \in \mathbb{R}^{n\times n} \tag{4}$$

For the matrix times vector multiplication, we have to calculate for each resulting element y_i, n multiplications (in Einsteins notation $A_{ij}x_j$) and $(n-1)$ summations. Finally there is another summation with the incoming element y_i, which brings for all n rows a total number of $2n^2$ floating point operations.

From a memory point of view, supposing to hold at least **x** and **y** constantly in cache, we have to load $n^2 + n + n$ single precision numbers, which take up 4 bytes each. The resulting vector needs to be written back, so another $4n$ bytes. Under the assumption of, again, not loading anything two times, we get the computational intensity

$$I(n) = \frac{2n^2}{4(n^2 + 3n)}; \qquad \lim_{n\to\infty} I(n) = \frac{1}{2} \tag{5}$$

However, from a practical point of view it surely happens with n getting bigger an bigger, the vector **x** can be hold in cache during the whole operation, but is needed for every row. So the vector **x** might be reloaded q times for every single row. With a limited amount of cache, and a specific maximum partition size p [bytes] which can stay in cache for **x**, it seems natural to predict the reloads

$$q = \frac{4n}{p} = c\,n \tag{6}$$

where c takes the constant term. In this way the limit, replacing the memory loads consideration for **x** with $c\,n^2$, we get

$$I_2(n) = \frac{2n^2}{4(n^2 + c\,n^2 + 2n)}; \qquad \lim_{n\to\infty} I_2(n) = \frac{1}{2(1+c)} \tag{7}$$

resulting in a more memory bound operation as we could expect.

1.1.3 DGEMM

In this example only matrices are involved:

$$C = AB + C \qquad A, B, C \in \mathbb{R}^{n\times n} \tag{8}$$

For the multiplication we need to perform for every resulting element C_{ij} n multiplications and $(n-1)$ summations. Accounting also for the summation with the incoming C_{ij} this becomes n^2 operations for every element. Given that there are n^2 elements, we have a total of n^4 floating point operations.

Under the assumption that there is enough cache to hold the three matrices, we can state that there are 3 loading and 1 writing operation, which sums up to $4n^2$ DRAM accesses for a double precision floating point number, resulting in $32n^2$ bytes.

$$I(n) = \frac{n^4}{32\,n^2}; \qquad \lim_{n\to\infty} I(n) = \lim_{n\to\infty} \mathcal{O}(n^2) \tag{9}$$

So in this case we we have an infinite operational intensity for infinite large matrices, growing in the order of n^2. So this problem is clearly computational bound. However, we need to remind that this is only under the assumption of fitting all matrices in to cache. If we considered also the necessary reloads, the order would decrease.

1.2 Operational intensity for a code snippet

```
for (int i = 0; i < N; ++i)
{
  double val = C[i] - 5.0;
  for (int j = 0; j < P ; ++j)
    val = 0.5 * val + C[i];
  A[i] = val;
}
```

Listing 1: Code Snippet

Assuming no compiler optimization are applied, we can simply count the floating point operations. In line 3, 1 floating point operation is executed, in line 5, 2 operations are executed. Considering the loops, we get a total count of $N(1 + 2\,P)$ operations.

From a memory point of view, we notice that we only access `P[i]` and `A[i]` consecutively. So even if the cache was small, consecutive reads and writes of the vectors **P** and **A** would allow to load / write them only once. The temporary value `val` can safely be assumed to stay on the stack in some register during each loop. So we simply get N reads and N writes, both of double precision variables[1].

Putting all together we have:

$$I(N, P) = \frac{N(1 + 2\,P)}{16\,N} = \frac{2\,P + 1}{16} \tag{10}$$

This means that on the platform with $I_{\text{ripple}} = 1$, for $P \leq 7$ the code will be memory bound, meanwhile $P \geq 8$ the procedure results compute bound.

1.3 Operation Intensity of 1D diffusion equation

Starting from the formula we first determine the amount of operations. For each element u_i we have 2 summations, one multiplication and another summation, so a total of 4 floating point operations, neglecting the fact that we have to compute once the multiplication factor $\frac{\delta t \alpha}{\delta x^2}$. For one iteration we have therefore 4 N floating point operations. In M iterations, this scales to $4\,N\,M$ flops.

[1]assumed double precision, given the type choice of the temporary variable `val`

For counting the memory operations, we assume that the discretized vector **u** along x fits into the cache. This makes then 8 N bytes of reading operations for double precision numbers. If for now, we assume to save just the result in the final time frame back into RAM, we get:

$$I(N,M) = \frac{4\,N\,M}{16\,N} = \frac{M}{4} \tag{11}$$

Which of course results in a operational demanding algorithm, scaling just with the numbers of iterations M. If we wanted to save instead all intermediate steps back to the memory, we get:

$$I(N,M) = \frac{4\,N\,M}{8\,M\,N} = \frac{1}{2} \tag{12}$$

which most likely will be memory bounded again.

The hardware bottleneck would therefore lie more in the memory accesses, if we want to save intermediate results. However, to optimally use the hardware, this algorithm can be optimized, if some of the intermediate results can be neglected.

2 Roofline Model (Q4)

2.1 Euler II Memory Bandwidth

Euler II uses DDR4 RAM at 2133Mhz. Since it is a double data rate RAM, the actual frequency is 1066Mhz, but delivering 2 units of data per clock cycle. The processor supports 4 channels of memory and the amount of bits in one channel transferred in one cycle is 64. If we multiply those numbers together and convert to byte we get:

$$\beta = 1066\text{Mhz} \times 2 \text{ data units} \times 4 \text{ channels } \times 64\text{bits} \times 0.125 = 68.3 \text{ GB/s} \tag{13}$$

$\square$ *q.e.d.*

2.2 Hardware Operational Intensity

1. The hardware is operated in balance for

$$I_b = \frac{\pi}{\beta} = \left[\frac{\text{Gflop/s}}{\text{Gbyte/s}}\right] \tag{14}$$

for an Euler node this translates to:

$$I_b = \frac{2 \times 480\text{Gflop/s}}{2 \times 68.3\text{Gbyte/s}} = 7.03\,\frac{\text{flop}}{\text{byte}}$$

2. The peak performance for a software with operational intensity I is given by:

$$P_{peak}(I) = min\begin{cases}\pi \\ \beta \times I\end{cases} \tag{15}$$

where π is the already introduced peak floating point performance in [flop/s] and β again the memory bandwidth in [byte/s].

Florian Mahlknecht
fmahlknecht@student.ethz.ch
19-945-351

3. A code will be memory bound if the operational intensity hits the roof on the left of the ridge point, i.e. $I < I_b$. Compute bound code will hit the flat part of the roof, i.e. $I > I_b$. If this peak floating point performance is not reached, there is potential of improving the algorithm by gaining instruction-level parallelism, e.g. by using single instruction multiple data (SIMD) instructions, or improving the the instruction mix by ensuring parallel multiplication and addition operations.

2.3 Euler II Roofline

Figure 1 shows a sketch of the roofline of a full Euler II Numa Node, as considered before.

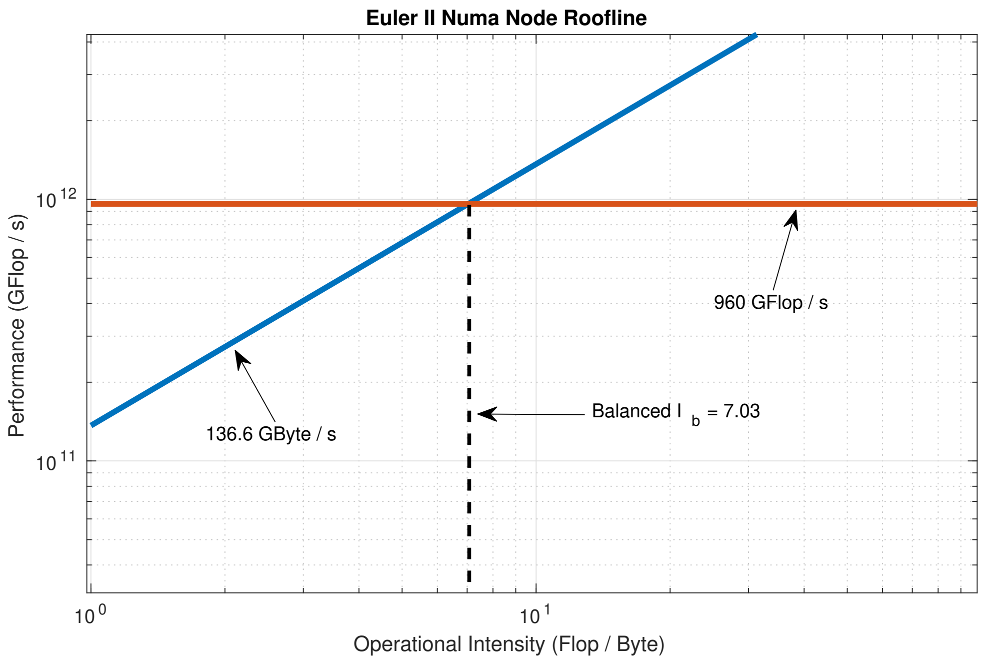

Figure 1: Euler Roofline Draft

2.4 Diffusion Kernel Benchmark

In the following listings the implemented code is presented, see listings 2 to 4.

```cpp
class DiffusionSolver
{
public:
    DiffusionSolver();

    static const int N = 20000;
    constexpr static const double L = 1000.0;
    constexpr static const double DX = L/N;
    constexpr static const double ALPHA = 10e-4;
    constexpr static const double DT = DX*DX/(4*ALPHA);

    /**
     * @brief init initializes values to U0
     */
    void init();

    void solve(double t_f);

    void printCSVLine();

private:
    std::vector<double> m_u;

    void doStep();
};
```

Listing 2: Diffusion Solver Class Declaration

```cpp
//#define _USE_MATH_DEFINES
#include "diffusionsolver.h"

#include <cmath>
#include <iostream>

DiffusionSolver::DiffusionSolver()
    : m_u(N)
{ }

void DiffusionSolver::init()
{
    for (int i = 0; i < N; ++i) {
        // do not accumulate summation errors
        auto x = i*DX;
        m_u[i] = std::sin(2*M_PI/L*x);
    }
}

void DiffusionSolver::solve(double t_f)
{
    int nit = std::ceil(t_f/DT);

    for (int n = 0; n < nit; ++n) {
        doStep();
    }
}

void DiffusionSolver::doStep()
{
    const double coeff = DT * ALPHA / (DX * DX);
    double u_old_0 = m_u[0];
    double u_old_i_prev = m_u[N-1];
    for (int i = 0; i < N-1; ++i) {
        double u_new = m_u[i] + coeff * (u_old_i_prev - 2*m_u[i] + m_u[i+1]);
        u_old_i_prev = m_u[i];
        m_u[i] = u_new;
    }

    m_u[N-1] = m_u[N-1] + coeff * (u_old_i_prev - 2*m_u[N-1] + u_old_0);
}

void DiffusionSolver::printCSVLine()
{
    bool first = true;
    for (const auto& ui : m_u) {
        if (!first) std::cout << ";";
        std::cout << ui;
        first = false;
    }
    std::cout << std::endl;
}
```

Listing 3: Diffusion Solver Class Definition

```
#include "cacheflusher.h"
#include "diffusionsolver.h"

using hrc = std::chrono::high_resolution_clock;

int main()
{
    CacheFlusher cf;
    DiffusionSolver solver;

    const int N = 10;
    std::vector<double> msrmt;
    msrmt.reserve(N);

    for (int i = 0; i < N; ++i) {
        // initialize to u_0
        solver.init();

        cf.flush();
        auto start = hrc::now();
        solver.solve(5000.0);
        auto end = hrc::now();

        std::chrono::duration<double> duration = end-start;
        msrmt.push_back(duration.count());

        if (i > 0)
            std::cout << "; ";
        std::cout << duration.count();
    }

    std::cout << std::endl;

    auto mean = std::accumulate(msrmt.begin(), msrmt.end(), 0.0)/msrmt.size();

    std::cout << "Mean computing time; " << mean << std::endl;

    return 0;
}
```

Listing 4: Main

To validate the implementation the possible CSV output function was used to perform a qualitative analysis. For practical reasons faster converging solution parameters (higher α) have been used and plotted, see fig. 2. As expected, the initial sin wave smooths out after some time.

On the Euler cluster, the following parameters have been used:

N = 20000	**DX** = 0.05	**DT** = 0.625
L = 1000	**ALPHA** = 0.001	

It has been iterated up to 5000 s, resulting in 8000 iterations. The job was executed on `eu-c7-077-05`. The following output has been obtained:

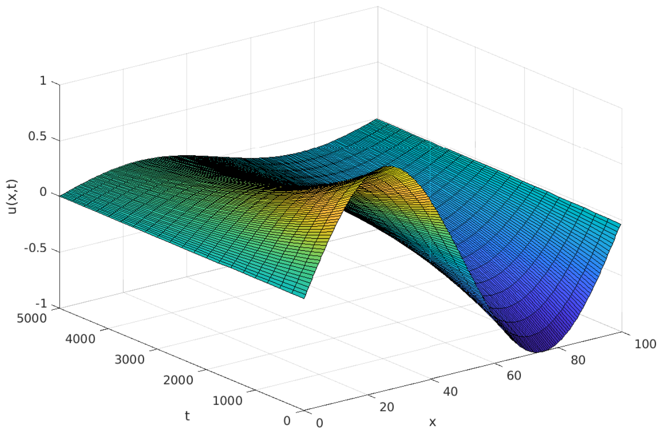

Figure 2: Diffusion

2.28624	2.17239	2.17716	2.1751	2.17003	2.17252	2.17296	2.18244	2.17323	2.17836
Mean computing time:	2.186043								

Table 1: Result from job execution on Euler

It is noticeable from table 1, that the first value is 100 ms higher than all the others. Most likely the flush cache routine has been optimized out by the compiler. The other values are all around 2.175 s.

If we consider now our predictions we estimated $4\,N\,M$ floating point operations, where M is the number of iterations. Putting in the numbers in, they summed up they should be 640 MFlop, which divided by the average time (neglecting the first one) results in effectively 294 MFlop/s. Given that our implementation was just single-threaded and did just use one of the 24 (2×12) cores, experienced performance was much worse.

3 Amdahl's law (Q5)

For number of processors $N \geq 2$ the speedup can be expressed as:

$$\text{speedup}(N) = \frac{t}{0.01t + 0.04\frac{t}{2} + 0.95\frac{t}{N}} = \frac{1}{0.03 + \frac{0.95}{N}} \tag{16}$$

The maximum speedup without limitations on the numbers of processors is:

$$\lim_{N\to\infty} \text{speedup}(N) = \frac{1}{0.03} \approx 33 \tag{17}$$

To obtain a speedup of at least 8, we need:

$$8 = \frac{1}{0.03 + \frac{0.95}{N}} \qquad \Rightarrow N = \frac{0.95}{0.76} 8 = 10 \tag{18}$$

at least 10 processors.