1 Question 1

Nothing to report.

2 Question 2

Nothing to report.

3 Question 3

3.1 Task a)

Assumptions for all subsequent calculations of operational intensity:

- Cache is empty (i.e. cold)
- "Memory transfer" implies transfer from random access memory (DRAM) to cache (typically L3 cache)

3.1.1 DAXPY

$$y = \alpha x + y \quad \alpha \in \mathbb{R}; x, y \in \mathbb{R}^n \tag{1}$$

According to table 1, a DAXPY operation on $x, y \in \mathbb{R}^n$ requires $2n$ FLOPS and $3n$ memory accesses. By definition, **D**AXPY is a double precision operation and thus each involved element has a size of 8 bytes. The resulting total operational intensity is $I(n) = \frac{2n\,\text{FLOPS}}{3n\cdot 8\,\text{bytes}} = \frac{1}{12}$ ops/byte and hence $\mathcal{O}(1)$.

	FLOPS		memory accesses	
$i = 1, 2, \ldots, n$	multiply	add	read	write
n times...	1	1	2	1

Table 1: DAXPY operations per element $i \in \{1, 2, \ldots, n\}$

3.1.2 SGEMV

$$y = Ax + y \quad x, y \in \mathbb{R}^n; A \in \mathbb{R}^{n\times n} \tag{2}$$

According to table 2, a SGEMV operation on $x, y \in \mathbb{R}^n$ requires $2n^2 + n$ FLOPS and $2n^2 + 2n$ memory accesses. To make things easier we focus on the asymptotic bound and only consider the second-order terms. By definition, **S**GEMV is a single precision operation and thus each involved element has a size of 4 bytes. The resulting total asymptotic operational intensity is $I(n) = \frac{2n^2\,\text{FLOPS}}{2n^2\cdot 4\,\text{bytes}} = \frac{1}{4}$ ops/byte and hence $\mathcal{O}(1)$.

	FLOPS		memory accesses	
$i = 1, 2, \ldots, n$	multiply	add	read	write
n times...	n	$n + 1$	$2n + 1$	1

Table 2: SGEMV operations per element $i \in \{1, 2, \ldots, n\}$

3.1.3 DGEMM

$$C = AB + C \quad A, B, C \in \mathbb{R}^{n \times n} \tag{3}$$

Based on the hint given in the task description, we assume that n and thus the matrices A, B, C are small enough to fit in the cache concurrently and do not require multiple reads during a single DGEMM operation. According to table 3, a DGEMM operation on $B, C \in \mathbb{R}^{n \times n}$ requires $2n^3 + n^2$ FLOPS and only $4n^2$ memory accesses because of the above assumption about n versus cache size. As for SGMEV above, we focus on the higher order terms for the following calculations. By definition, **D**GEMM is a double precision operation and thus each involved element has a size of 8 bytes. The resulting total asymptotic operational intensity is $I(n) = \frac{2n^3\,\text{FLOPS}}{4n^2 \cdot 8\,\text{bytes}} = \frac{n}{16}$ ops/byte and hence $\mathcal{O}(n)$.

	FLOPS		memory accesses	
$i, j = 1, 2, \ldots, n$	multiply	add	read	write
$n \cdot n$ times...	n	$n+1$	3	1

Table 3: DGEMM operations per element $i, j \in \{1, 2, \ldots, n\}$

3.2 Task b)

$$u_i^{n+1} = u_i^n + \frac{\Delta t \alpha}{\Delta x^2}(u_{i-1}^n - 2u_i^n + u_{i+1}^n) \tag{4}$$

Based on equation (4) and considering that $\frac{\Delta t \alpha}{\Delta x^2}$ is a fixed value which can be precalculated and stored, there are 3 additions and 2 multiplications for a total of 5 FLOPS per grid point. As for DGEMM in the previous task, we now must make assumptions about the problem size versus the cache size (the cache size being a potential bottleneck): It would be most realistic but also most cumbersome for our calculations to assume that some part of the problem fits in a cache of limited but realistic size and that we have to deal with a mix of compulsory and capacity cache misses. At the extreme ends of the cache size-vs-problem size scale we could take up either the assumption of an infinite cache and thus only one read and one write operation per grid point or the assumption of zero cache with a worst case maximum of four reads and one write per grid point. Assuming we are dealing with double precision data (8 bytes per element), we can thus make the following estimations: $I(N)_{\max} = \frac{5\,\text{FLOPS}}{2 \cdot 8\,\text{bytes}} = \frac{5}{16}$ ops/byte and $I(N)_{\min} = \frac{5\,\text{FLOPS}}{5 \cdot 8\,\text{bytes}} = \frac{1}{8}$ ops/byte with the realistic asymptotic operational intensity bounded as in equation (5).

$$0.125\,\text{ops/byte} \leq I(N) \leq 0.3125\,\text{ops/byte} \tag{5}$$

4 Question 4

4.1 Task a)

Based on the link [2] given in the task description, the CPU used is of the Haswell generation, has 12 cores with a processor base frequency of 2.50 GHz and can use the AVX2 instruction set. As nicely summarized in [1], AVX2 introduced 256-bit (i.e. 4 times 64-bit double-precision elements) floating-point fused multiply-add (FMA) support which allows for 16 double precision floating point operations per cycle (2 FMA execution units each able to execute 2 operations per cycle). In total, this gives $\pi = 12 \times 2.5 \times 10^9 \frac{\text{cycle}}{s} \times 16 \frac{\text{FLOPS}}{\text{cycle}} = 480 \frac{\text{GFLOPS}}{s}$.
According to [2], the fastest memory supported by the CPU is DDR4 memory running at 2133 MHz on four memory channels of size 64 bit each (the 256 bit as described above divided by 4). We thus get a maximum memory bandwidth of $\beta = 2133 \times 10^6 \frac{\text{cycle}}{s} \times 256 \frac{\text{bit}}{\text{cycle}} = 546.048 \frac{\text{Gbit}}{s} = 68.256 \frac{\text{Gbyte}}{s}$ which conveniently also is equal to the *Max Memory Bandwith* of 68GB/s indicated on [2] itself.

4.2 Task b)

4.2.1 I_b when in operating in balance

As described in [4], the ridge point is where both the CPU and the memory bandwidth are fully exploited and is situated at the intersection of π and $I \times \beta$ in a log-log plot of operational intensity I vs GFLOPS$/s$.
In specific, for a full node of Euler II we have $I_b = \frac{\pi}{\beta} = \frac{960 \frac{\text{GFLOPS}}{s}}{136.6 \frac{\text{Gbyte}}{s}} \approx 7\,\text{ops/byte}$.

4.2.2 $P_{\textbf{peak}}$

Also as per the source and calculations cited above, we have that $P_{\text{peak}}(I) = \min(\pi, I \times \beta)$ where π, β are fixed hardware properties as described above.

4.2.3 memory bound vs compute bound

A code is memory bound if its operational intensity I is smaller than the calculated balanced operational intensity I_b: $I < I_b$. Likewise, a code is compute bound if we have that $I > I_b$. The possible optimisations differ according to the bound: For memory bound code, it is recommended [4] to first minimise conflict caches misses by data reordering and then to minimise capacity cache misses by reordering and non-temporal writes. All this is to increase operational intensity I.
For compute bound codes, it is harder to find optimisations as there is no point in further increasing operational intensity. Possible measures include use of SSE/AVX instructions and/or reductions in precision.

4.3 Task c)

According to [3], a full compute node of Euler II has two sockets each according to the above Xeon E5-2680v3 CPU and memory bandwidth specifications π, β with a total of 24 cores per node. Based on the computations This makes for a roofline diagram as in figure 1.

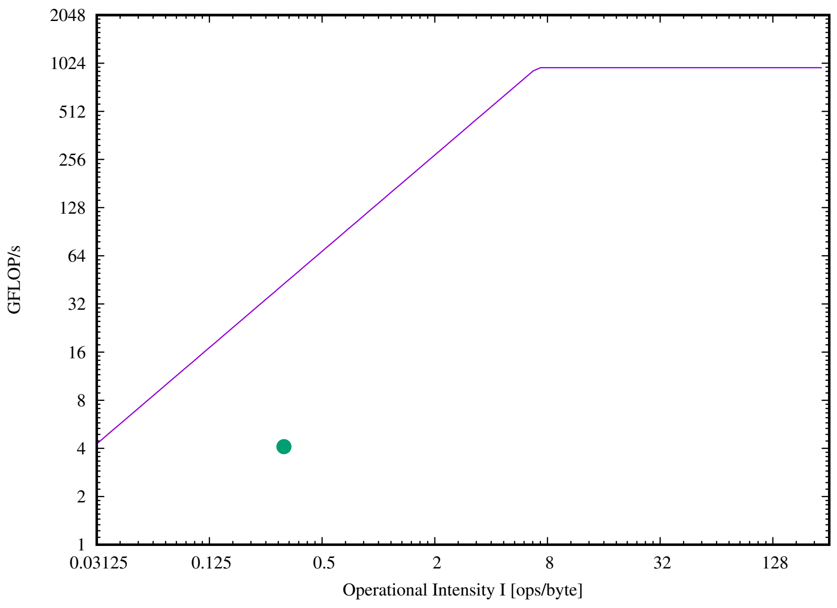

Figure 1: Roofline diagram for full compute node of Euler II; measured value of task d).

4.4 Task d)

Based on the code with an attempted CPU data cache flush given in appendix A compiled with GCC 4.8.2 using the flags −fopenmp −std=c++11 −O3 −march=native −funroll−loops (using OpenMP only to get a simple way to measure time), repeated measurements on Euler II result in a performance of about $4.1\,\frac{\text{GFLOPS}}{s}$ short of the theoretically achievable single thread/single channel maximum of $5\,\frac{\text{GFLOPS}}{s}$. This position is marked by a green dot in figure 1. We therefore conclude that the chosen implementation falls into to the memory bound region as described above. Potential reasons for not fully utilising the hardware are the naive implementation (unbalanced adds/multiplies) as well as only using a single thread. A first measure would be to use OpenMP to use all available threads on the CPU's cores.

References

[1] NASA, *Haswell Processors*, online (`https://www.nas.nasa.gov/hecc/support/kb/haswell-processors_492.html`), accessed 30-Sep-2018.

[2] Intel Corporation, *Intel Xeon Processor E5-2680 v3 Product Specifications*, online (`https://ark.intel.com/products/81908/`), accessed 30-Sep-2018.

[3] ETH Zurich, *scientific computing wiki: Euler*, online (`https://scicomp.ethz.ch/wiki/Euler#Euler_II`), accessed 30-Sep-2018.

[4] Petros Koumoutsakos, *HPCSE I Lecture Notes*, online (`http://www.cse-lab.ethz.ch/wp-content/uploads/2018/09/HPCSE_I_1_Intro.pdf`), accessed 30-Sep-2018.

A Question 4, task d)

```cpp
/* Filename: ex01_q4_task_d.cpp
   Course: HPCSE I, HS2018, ETHZ
   Exercise No: 01
   Author: bhubmann@student.ethz.ch
   INPUT: None
   OUTPUT: None
   DESCRIPTION: Runs 1D diffusion FDM scheme as given by exercise,
                1000 iterations for naive FLOP/s performance calculation
*/

#include <iostream>
#include <cmath>
#include <cstring>
#include <omp.h>

void Diffusion1D(const double* const u_old, double* const u_new, const int N, const double factor)
{
        u_new[0]=      u_old[0]   + factor * (u_old[N-1] - 2 * u_old[0]   + u_old[1]);
        for (int i= 0; i < N - 1; i++)
        {
            u_new[i]= u_old[i]   + factor * (u_old[i-1] - 2 * u_old[i]   + u_old[i+1]);
        }
        u_new[N-1]=   u_old[N-1] + factor * (u_old[N-2] - 2 * u_old[N-1] + u_old[0]);
}

int main()
{
        const int kNumIterations= 1000;
        const int kBufferSize = 128 * 1024 * 1024; // 128 M
        const int kGridPointsN= 1024 * 1024; // 1 M
        const double kDomainLengthL= 1000.0;
        const double kAlpha= 1.0e-4;
        const double kCFL= 0.5;
        const double kDeltaX= kDomainLengthL / (kGridPointsN - 1);
        const double kDeltaT= kCFL * kDeltaX * kDeltaX / kAlpha;
        const double kFactor= kDeltaT * kAlpha / (kDeltaX * kDeltaX);

        char* filler= new char[kBufferSize]; // 16 MB

        double* u_0= new double[kGridPointsN]; // 8 MB
        double* u_1= new double[kGridPointsN]; // 8 MB

        auto lInitData = [kDomainLengthL](const double x_value)
         {return std::sin(2. * M_PI / kDomainLengthL * x_value);};

        for (int i= 0; i < kGridPointsN; i++)
        {
                u_0[i]= lInitData(i * kDeltaX);
                u_1[i]= u_0[i];
        }

        double total_time= 0.0;
        for (int i= 0; i < kNumIterations; i++)
        {
                volatile int j= i * i;
                memset(filler, j, kBufferSize);
                const double kStartTime= omp_get_wtime();
                Diffusion1D(u_0, u_1, kGridPointsN, kFactor);
                const double kEndTime= omp_get_wtime();
                total_time += (kEndTime - kStartTime);
                double* u_temp= u_0;
                u_0= u_1;
                u_1= u_temp;
        }
        const long int total_flops= 5 * (long int)kNumIterations * (long int)kGridPointsN;
        const double performance= total_flops / (total_time * 1e9);
        std::cout << kNumIterations << "␣iterations,␣total␣time␣[s]:␣" << total_time << std::endl
                  << "total␣naive␣FLOP␣operations:␣" << total_flops << std::endl
                  << "measured␣performance␣[GFLOP/s]:␣" << performance << std::endl;
        delete[] filler;
        delete[] u_0;
        delete[] u_1;
        return 0;
}
```