ETH zürich

High Performance Computing for Science and Engineering I

P. Koumoutsakos
ETH Zentrum, CLT E 13
CH-8092 Zürich

Fall semester 2018

Set 1 - Roofline Model and Performance Measures

Issued: September 28, 2018
Hand in (optional): October 5, 2018 10:00am

Question 1: Setting up your local environment

This exercise will guide you through the setup of your local development environment. The procedure described here is only for Linux machines because the cluster also runs Linux. On the ETH computers, everything is already installed. Follow the following steps:

1. Open a terminal (press windows key and type `terminal`, then click the icon)
2. Clone the repository with the following command:
   ```
   $ git clone https://gitlab.ethz.ch/hpcse18/lecture.git
   ```
   Tip: You can still use copy paste in the terminal: Ctrl+Shift+C to copy from terminal, Ctrl+Shift+V to paste to the terminal.

This creates a directory named `lecture` inside your home folder. Its contents are those of the repository. You're done setting up your machine!

If the repository gets updated, you can get the latest changes with the following command from within the `lecture` directory:
```
git pull
```

Further information about git:
Git Handbook by Github (10min)
Git cheat sheet by Github

Question 2: Getting started on Euler

Euler is the new computing cluster of ETH Zurich, an evolution of the Brutus cluster. The cluster works with a queueing system: you submit your program with its parameters (called job) to a queue and wait for it to be finished. Your first task consists of the following steps:

1. **Creating an account**
   Euler accounts are created automatically when a user logs in for the first time. You will need to enter your nethz username and password. Euler is only accessible within the ETH Network.[1]

[1]You may use VPN https://sslvpn.ethz.ch to connect to the ETH Network from home.

2. **Login**
   Login from within the ETH network on Euler via ssh:
   ```
   $ ssh username@euler.ethz.ch
   ```
   and insert your password when asked to.

   Congratulations! You are now on a login node of the Euler cluster. In this environment you can write code, compile and run small tests. Keep in mind that there are other people working on the same nodes, so be mindful of how you use them!

3. **Modules**
   The Euler environment is organized in modules, which are conceptually software packages that can be loaded and unloaded as needed. The basic commands to use the module system are:
   `$ module load <modulename>`: this command sets the environment variables related to the specified module.
   `$ module unload <modulename>`: this commands unsets the environment variables related to `<modulename>`.
   `$ module list`: lists all the modules currently loaded.
   `$ module avail`: outputs a list of all the modules that can be loaded.

   If, for example, we need to compile a program with the GNU compiler (gcc or g++), we first load its module with
   ```
   $ module load gcc
   ```
   and then proceed with the compilation of our program:
   ```
   $ g++ -O2 main.cpp -o program_name
   ```

4. **Submitting a job**
   Performance measurements and long computations should not be performed on the login nodes but rather they should be submitted to the queue.
   To submit a simple job to the queue, you can use the following command from the folder where your program is stored:
   ```
   $ bsub -n 24 -W 08:00 -o output_file ./program_name program_args
   ```

   This command will submit a job for your executable `program_name` with arguments `program_args` by requesting 24 cores from a single node and a wall-clock time of 8 hours, after which, if the job is not already finished running, it will be terminated. The report of the job, along with the information that would usually appear on the terminal, will be appended in the file `output_file`, in the folder from where the job started.
   While your job is running you can always use the command:
   ```
   $ bjobs
   ```
   to get the status and IDs of your jobs.
   In order to terminate a job you can use the command:
   ```
   $ bkill <jobid>
   ```

Info: I/O performance and `$SCRATCH`:
Since your simulations might involve a lot of I/O (input/output), you must never run your simulations in your `$HOME` directory, but setup the runs in your `$SCRATCH` space. The disks associated with this space are designed for heavy loads[2]. Lastly, your quota in `$HOME` is much

smaller compared to `$SCRATCH`. However, please note that the memory in `$SCRATCH` is temporary and **any files older than 15 days are deleted automatically** (see `$SCRATCH/__USAGE_RULES__`). Follow the links below for more information on the Euler cluster.
Use the following command to change to your scratch directory:

```
$ cd $SCRATCH
```

Additional information on the Euler cluster, its instruments and on how to use it can be found at:

- https://scicomp.ethz.ch/wiki/Euler
- https://scicomp.ethz.ch/wiki/User_documentation

Question 3: Operational Intensity

The purpose of a compute architecture is to perform a certain operation on data. This requires a mechanism to send data along the memory hierarchy[3] to the execution units such that they can perform the operation(s).

The operational intensity is a measure that relates the amount of work W (operations) to the number of bytes Q (data) that need to be transferred and is defined as

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

The operational intensity I is used to characterize a code and reveals information about the expected utilization of the architecture's execution units and memory bandwidth.
In practice, operations are defined by floating point operations (FLOP) and memory transfer is defined by access from DRAM to the closest processor cache.
Hint: Remember that double precision numbers are typically stored in 8 bytes.

Your tasks

a) Determine the asymptotic bounds on the operational intensity $I(n)$ for the following matrix/vector operations, where n is the dimension of the vector. State your assumptions.

1. DAXPY: $y = \alpha x + y \quad \alpha \in \mathbb{R}; x, y \in \mathbb{R}^n$
2. SGEMV: $y = Ax + y \quad x, y \in \mathbb{R}^n; A \in \mathbb{R}^{n\times n}$
3. DGEMM: $C = AB + C \quad A, B, C \in \mathbb{R}^{n\times n}$
   Hint: Assume that A, B and C fit into the cache at the same time.

We assume that we start with a cold cache and memory transfers are defined from DRAM to the lowest level cache. Furthermore, we define a count F and a count M which correspond to total number of flops and total number of memory accesses (read and write), respectively.

DAXPY (5 points) We write the operation in index notation

$$y_i = \alpha x_i + y_i \quad i \in [1, n], \tag{2}$$

where α is a constant scalar. For each element i we count

[2] However, `$SCRATCH` is not designed for frequent storing. If you are logging temporary results into a file, open the file once at the beginning, and do not flush e.g. more than one per second (or per minute). Related to it, note that `std::endl` not only prints the newline character `'\n'`, but also flushes the stream.

[3] https://en.wikipedia.org/wiki/Memory_hierarchy

- $f_i = 1_{\text{mul}} + 1_{\text{add}} = 2$ flops
- $m_i = 2_{\text{read}} + 1_{\text{write}} = 3$ doubles

For n elements we get $F = nf_i$ and $M = nm_i$. The number of actual bytes transferred over the bus depends on the precision of the operation. The DAXPY is double precision for which each element is of size $p = 8$ bytes/double. Therefore, the asymptotic bound on the operational intensity is

$$I(n) = \frac{F}{Mp} = \frac{1}{12} \text{ flops/byte} \Rightarrow \mathcal{O}(1) \text{ flops/byte.} \tag{3}$$

Points:

- 2 for correct F
- 2 for correct M
- 1 for correct asymptotic behaviour.

SGEMV (5 points) We proceed similar as above. Note that we apply the Einstein summation convention unless noted otherwise. Matrix elements are written in lower-case letters.

$$y_i = a_{ij}x_j + y_i \quad i, j \in [1, n]. \tag{4}$$

For each element i we count

- $f_i = n(1_{\text{mul}} + 1_{\text{add}}) + 1_{\text{add}} = 2n + 1$ flops
- $m_i = 2n_{\text{read}} + 1_{\text{read}} + 1_{\text{write}} = 2n + 2$ floats

For n elements we compute $F = nf_i = 2n^2 + n = 2n^2 + \mathcal{O}(n)$ flops and $M = nm_i = 2n^2 + 2n = 2n^2 + \mathcal{O}(n)$ floats memory accesses. The SGEMV is a single precision operation, thus $p = 4$ byte/float. To determine the asymptotic bound, we only keep the leading order of operations to get

$$I(n) = \frac{F}{Mp} = \frac{1}{4} \text{ flops/byte} \Rightarrow \mathcal{O}(1) \text{ flops/byte.} \tag{5}$$

Note that for recent processors, the machine balance I_B is roughly 5–10. Since $I(n)$ for the SGEMV is *constant* the operation is memory bound independent of the problem size n.

Points:

- 2 for correct F
- 2 for correct M
- 1 for correct asymptotic behaviour.

DGEMM (5 points) For this problem, we further assume that the problem size n is small.

$$c_{ij} = a_{ik}b_{kj} + c_{ij} \quad i, j \in [1, n], \tag{6}$$

For each element i, j we count

- $f_{ij} = n(1_{\text{mul}} + 1_{\text{add}}) + 1_{\text{add}} = 2n + 1$ flops
- $m_{ij} = 3_{\text{read}} + 1_{\text{write}} = 4$

For n^2 elements we compute $F = n^2 f_{ij} = 2n^3 + \mathcal{O}(n^2)$ flops and $M = n^2 m_{ij} \geq 4n^2$ memory accesses. The DGEMM is a double precision operation, thus $p = 8$ byte. The asymptotic bound is then given by

$$I(n) = \frac{F}{Mp} \leq \frac{n}{16} \Rightarrow \mathcal{O}(n). \tag{7}$$

The assumption of small n means that the three matrices A, B and C fit in the cache and must be read only once, where C is written once additionally. For example, if $n = 1000$ the memory footprint in double precision is 23 MB which will fit in the L3 cache of the Intel Xeon E5-2680 v3 on the Euler nodes. Therefore, the number of memory accesses M for this assumption gives a lower bound. A naive implementation of DGEMM will likely require more memory accesses for large n for which the operational intensity will become smaller. However, blocking strategies can be applied to increase temporal locality and therefore reduce cache misses for which the DGEMM operation can be optimized towards operational intensities that lie in the compute bound region.
Points:

- 2 for correct F
- 2 for correct M
- 1 for correct asymptotic behaviour.

b) Consider the 1D diffusion equation in a periodic domain with length L

$$u_t(x,t) = \alpha u_{xx}(x,t), \tag{8}$$

$$u_0(x) = u(x,0) = \sin\left(\frac{2\pi}{L}x\right), \tag{9}$$

where $0 \leq x < L$, $t > 0$ and $\alpha > 0$ is the diffusion coefficient. You are asked to solve this problem numerically by using a second order centered finite difference scheme and the explicit Euler method to advance in time. The domain is discretized with N uniformly spaced grid points. Discretization of Equation (8) leads to

$$u_i^{n+1} = u_i^n + \frac{\Delta t \alpha}{\Delta x^2}\left(u_{i-1}^n - 2u_i^n + u_{i+1}^n\right), \tag{10}$$

where $u_i^n \approx u(x_i, t^n)$ is the approximate solution with $t^{n+1} = t^n + \Delta t$ and $x_i = i\Delta x$. The constant time step size and uniform grid spacing are denoted by Δt and Δx, respectively. Determine the operational intensity $I(N)$ for the scheme given in Equation (10). Can you identify a possible hardware bottleneck for this scheme? State your assumptions.

(8 points) As the value at each grid point is computed in the same way, we count the flops and memory accesses per grid point.

In Equation (10) we count $f = 5$ flops per grid point. There are 3 extra operations that we do not count as they are between constant values $\frac{\Delta t \alpha}{\Delta x^2}$ and thus can be precomputed.

To count the memory operations (reads and writes) from and to DRAM we can use different approaches:

1. Assume there is no cache and therefore every read and write operation is done directly in main memory. This assumption leads to a worst case scenario to the number of memory accesses.

2. Assume a cache of infinite size, thus each data point is read once and written once at most. This assumption leads to the best case scenario as it only considers compulsory cache misses.
3. Assume a cache of limited size and estimate the number of memory operations given the memory layout used and the principles of data locality (temporal and spatial).

Following the three approaches listed above we have:

1. Assuming there is no cache, we have 4_{read} and 1_{write} operation per grid point (u_i^n is read twice and the computation could be easily reformulated to reduce the number of read operations). The total number of memory accesses is therefore $m = 5$.
2. With an infinite sized cache, we only need two memory operations per grid point (1_{read} and 1_{write}). Hence, the total number of memory accesses is $m = 2$.
3. In reality we deal with a finite size cache. For the simple 1D stencil of Equation (10) it is safe to assume that the data u_{i-1}^n and u_i^n is already in cache from the previous iteration and only u_{i+1}^n needs to be loaded from DRAM. Thus, the 1D problem computed on a sufficiently large grid will be dominated by compulsory cache misses (rather than capacity misses) similar to the infinite cache above. However, capacity misses become more problematic for stencils in higher dimensions on sufficiently large grids and even more so for high-order stencil schemes. In such cases a blocked memory layout is used to increase temporal locality of the data in the cache.

The operational intensities for the three cases are (assuming double precision $p = 8$ bytes)

1. $I_1 = \frac{5}{40} = 0.125$ flop/byte
2. $I_2 = \frac{5}{16} \approx 0.313$ flop/byte
3. $I_3 = I_2$

Above, we have used that (same notation as in the first part of this question)

$$I = \frac{F}{M} = \frac{nf}{nm} = \frac{f}{m}. \tag{11}$$

For single precision the numbers double.

We note that it is not always trivial to estimate the work of the caches and therefore one approach is to localize the operational intensity within a range instead of finding the exact value. An advantage of estimating the range, is that we are able to also estimate a lower bound on the performance.

Points:

In order to get the full amount of points, it is sufficient to give one of the above solutions.

- **1 for realizing that $\frac{\Delta t \alpha}{\Delta x^2}$ can be precomputed,**
- **2 for giving the correct number of operations per grid point f and/or the total number of operations $F = nf$,**
- **4 for correctly stating the assumptions about cache size and number of memory accesses per grid point m and/or total memory accesses $M = nm$,**

- 1 for correctly calculating the operational intensity I

Question 4: Roofline Model

The roofline model[4] is a simple visual tool that can be used to understand performance on a hardware architecture. It relates the nominal peak performance π and the nominal peak bandwidth β of a given hardware and introduces the concept of operational intensity, studied in the previous question.

Understanding the characteristics of the platform on which performance tests are executed is of fundamental importance since it gives a context in which to read performance results. The primary objective of this exercise is to learn how to characterize computing hardware performance.

Peak Performance and System Memory Bandwidth

The number of executed floating point operations (FLOP) is a measure used to characterize the costs of scientific software, e.g. computational fluid dynamics codes, structural mechanics, computational chemistry and computational biology packages.

The peak floating point performance, hereafter simply called peak performance, is a measure of the quantity of FLOP that a machine can execute in a given amount of time. Typically, the peak floating point performance π in FLOP/s can be computed as in the following:

$$\pi\ [FLOP/s] = f\ [HZ = cycle/s] \times c\ [FLOP/cycle/lane] \times v\ [lanes] \times n\ [-], \qquad (12)$$

where f is the core frequency in CPU cycles per second (given in Hz), c the number of FLOP executed in each cycle, v the SIMD width (in number of elements that fit into the register size, also called SIMD *lanes*) and n the number of cores.

The exact values for the above features can be found at the hardware specifications or given by the system administrators. For example, for the Euler cluster we can find the specifications of the Intel Xeon E5-2680v3 here: https://ark.intel.com/products/81908/. Additional information is provided at the Euler wiki: https://scicomp.ethz.ch/wiki/Euler.

Typical floating point performance values are reported in GFLOP/s or TFLOP/s. Floating point performance is also used to assess the performance of a given algorithm implementation (http://en.wikipedia.org/wiki/FLOPS).

The bandwidth of the system memory is a measure of the speed of data movement from the system to caches and vice-versa. As the problems considered herein fit on a single node, we are mostly interested in quantifying the DRAM bandwidth.

Given the specifications of the DRAM memory, the theoretical memory bandwidth β in B/s can be computed as follows:

$$\beta\ [B/s] = f_{DDR}\ [Hz = cycle/s] \times c\ [channel] \times w\ [bit/channel/cycle] \times 0.125\ [B/bit], \quad (13)$$

where f_{DDR} is the DDR clock rate, c the number of memory channels and w the bits moved through a channel per cycle (typically 64 bits). Typical bandwidths are reported in MB/s, GB/s or TB/s (http://en.wikipedia.org/wiki/Memory_bandwidth).

[4] Williams et al, 2009: https://dl.acm.org/citation.cfm?id=1498785

Memories based on the DDR (Double Data Rate) technology, such as DDR-SDRAM, DDR2-SDRAM, and DDR3-SDRAM[5], transfer two units of data per clock cycle. As a result, they achieve double the transfer rate compared to traditional memory technologies (such as the original SDRAM) running at the same clock rate. Because of that, DDR-based memories are usually labeled with double their real clock rate. For example, DDR3-1866 memories actually work at 933 MHz transferring two units of data per clock cycle, and thus are labeled as being a "1866 MHz" device, even though the clock signal does not really work at 1.866 GHz.

Technological advances in the past dictate that the rate π at which operations can be executed doubles roughly every 18 months (Moore's Law), while the rate β at which data can be transported doubles roughly every 36 months.

Your task

a) According to the wiki of the Euler cluster, each of the two sockets on a node hosts an Intel Xeon E5-2680v3 12-core Haswell CPU capable of delivering 480 GFLOP/s each. In addition, the theoretical memory bandwidth for each of the two sockets can reach 68.3 GB/s. Try to justify the above numbers.

(12 points) From https://ark.intel.com/products/81908/ we find the following specifications for the Intel Xeon E5-2680v3:

- 12 cores
- 2.50 GHz clock frequency
- Supports AVX2 extended instruction set (256-bit wide FMA instructions)
- The Intel Haswell architecture can issue 2 FMA instructions per cycle

The Haswell architecture has two execution units for FMA instructions which yields a maximum of four flops per cycle (assuming you have a high instruction density to hide the 5 cycle latency of the FMA instructions). The 256-bit registers can hold either 4 double precision numbers or 8 single precision numbers. The total double precision peak performance is then calculated by (for a base frequency of 2.5 GHz)

$$\pi = 2.50 \times 10^9 \text{ cycle/s} \times 12 \text{ cores} \times 4 \text{ SIMD lanes} \times 4 \text{ flop/cycle/lane} = 480.0 \text{ Gflop/s.} \tag{14}$$

Note that the processor frequency is allowed to vary and can go up to 3.3 GHz.

For the memory we find

- Euler II supports DDR4 memory with frequency 2133 MHz
- Maximum number of memory channels is 4 (see CPU fact sheet)
- 64-bit architecture with word size 64-bit

Given this data, we can compute the nominal peak bandwidth by (for the fastest possible memory)

$$\beta = 2.133 \times 10^9 \text{ cycle/s} \times 4 \text{ channel} \times 64 \text{ bit/channel/cycle} \times \frac{1}{8} \text{ byte/bit} = 68.3 \text{ Gbyte/s} \tag{15}$$

Points:

[5] http://en.wikipedia.org/wiki/DDR3_SDRAM

- 1 for finding correct number of cores and frequency of the CPU
- 2 for finding out that the CPU supports AVX2
- 2 for finding out that Intel Haswell can perform 2 FMA instructions per cycle
- 1 for correctly calculating π
- 1 for finding the correct frequency of the memory of Euler II
- 1 for finding out that $c = 4$
- 1 for gettig the correct word size
- 3 for correctly calculating β

b) 1. What is the value of the operational intensity I_b for which the hardware is operated in balance, i. e. both the CPU and the memory bandwidth are optimally utilised? What is the value in numbers for a full node on Euler?

2. Give an expression for the maximum performance $P_{\text{peak}}(I)$(in FLOP/s) a piece of software with operational intensity I can reach. This expression should be a function of I that depends only on properties of the hardware.

3. The operational intensity I allows to classify codes into two groups: *memory bound* codes and *compute bound* codes. The classification depends on the used hardware! How can one determine from the roofline plot if a code is memory or compute bound? What implications does this have on the optimizations we will choose?

(10 points)

1. A machine is operated in balance if the CPU needs as much time to work on data as long as the memory system needs to deliver fresh data from RAM. If both the CPU and the memory system should utilize their full potential, this means that the CPU performs π FLOPper second, while the memory bandwidth also reaches its peak value and delivers β byte per second.
   A code that achieves this must therefore have an operational intensity

$$I_b = \frac{\text{operations per s}}{\text{memory accesses per s}} = \frac{\pi}{\beta}.$$

   In the case of a full Euler II node, we have two sockets. Each of them provides $\pi = 480$GFlop/s and $\beta = 68.3$Gbyte/s. Therefore, a piece of code needs an operational intensity

$$I_{\text{b, Euler node}} = \frac{480 \text{ GFlop/s}}{68.3 \text{ Gbyte/s}} \approx 7.03 \text{ flops/byte}$$

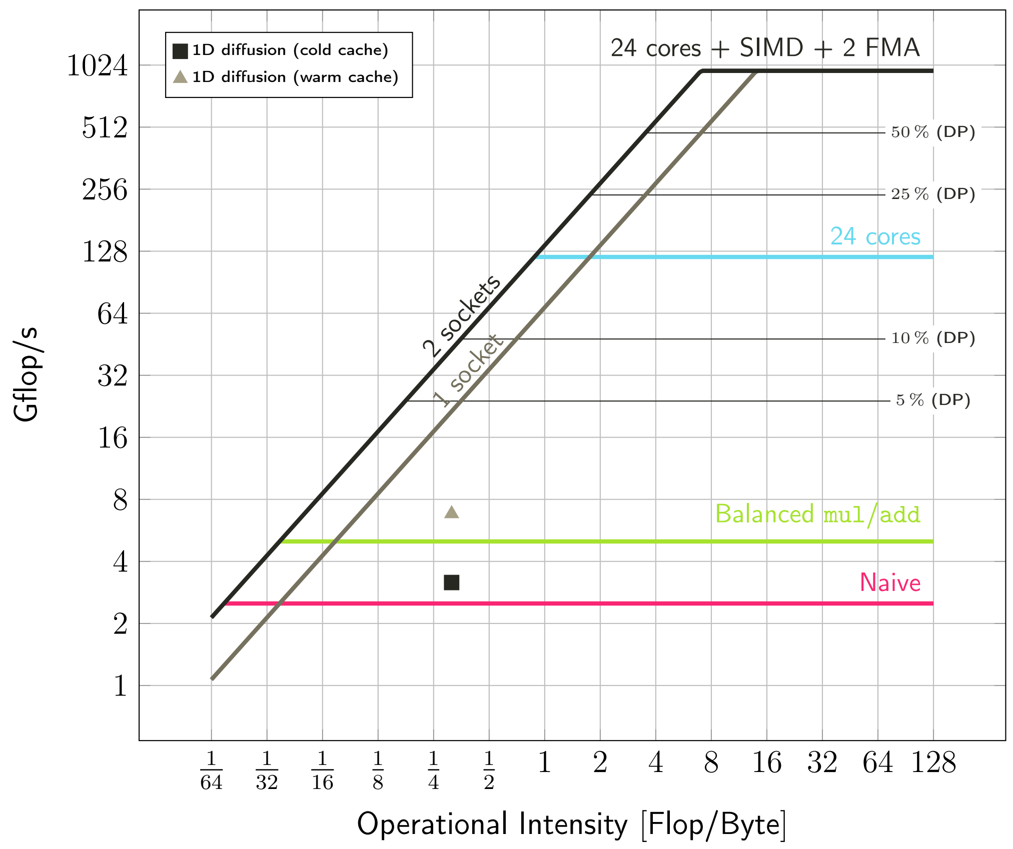
   to operate a full node in balance.
   Points:
   - 1 for the correct formula
   - 2 for the correct explanation
   - 1 for the correct value for Euler

2. $P_{\text{peak}} = \min(I\beta, \pi)$
   Points: 1 point for the correct formula

3. A code is *memory bound* if its operational intensity I is smaller than the optimal operational intensity I_b, $I < I_b$.
   A code is *compute bound* if its operational intensity I is bigger than the optimal operational intensity I_b, $I > I_b$.
   A code is *optimal* for a given hardware platform if $I = I_b$.
   If a code appears in the roofline plot on the left of the ridge, it is memory bound. In order to optimize such a code, the number of cache misses must be reduced. This can be achieved by increasing spatial and temporal locality of memory accesses, or by choosing an algorithm that needs less memory accesses (often not possible).
   If a code appears in the roofline plot to the right of the ridge, it is compute bound. In order to optimize this kind of code, one could for example try to better cache values that were already calculated, or try to inline repeated calculations (or even compute them at compile time).
   Points:
   - **1 for correct definition of memory/compute bound**
   - **2 for correct explanation of how to read it off the roofline plot**
   - **2 for giving at least one direction of optimization for each region.**

c) Draw the roofline for a full NUMA node of Euler.
   (5 points) For a full node on Euler we have two sockets, each delivers $\pi = 480$ Gflop/s (double precision) and a nominal peak bandwidth of $\beta = 68.3$ Gbyte/s. To reach the peak performance, both NUMA nodes must be fully utilized. The roofline for a compute node on Euler is shown below.

Euler compute node (2 sockets with Intel Xeon E5-2680v3, double-precision)



In the above plot, "DB" means double precision.
The "naive" performance plateau is suitable for codes that are not optimized at all.
The plateau for "balanced mulitplication/addition" describes the peak performance for codes that have about the same number of additions and multiplications. This uses the fact that modern CPUs have separate multiplication and addition units: They can execute one addition and one multiplication in the same cycle. This plateau does not consider SIMD units, i. e. the SIMD width is assumed to be 1.
"24 cores" means the same as "balanced mult/add", but now 24 cores are occupied with balanced mult/add. The SIMD width is still 1.
Finally, the peak performance considers that the SIMD width is 4 (4 mult/add per cycle), and that the CPU can execute 2 FMA instructions in one cycle.
The marks for the diffusion code refers to the next subquestion.

Points:

- **3 for the correct axis: correct quantities P and I, log-log scale, and units**
- **2 points for the correct slope (slope of 1 in the log-log plot) and correct offset in y-direction (the memory bandwidth β)**

d) Write a small C/C++ code that implements Equation (10). Choose a suitable number of grid points N, where $L = 1000$ and $\alpha = 10^{-4}$. For stability reasons, you must choose your time step such that

$$\Delta t < \frac{\Delta x^2}{2\alpha}. \qquad (16)$$

Benchmark the diffusion kernel on one Euler node. Measure multiple kernel executions (each is on full iteration over the grid) to obtain an average time for your measurement. Use the operational intensity calculated in Question 3b and indicate the measured performance in the roofline plot. Is your implementation memory bound or compute bound? What is the maximum performance that your code can reach according to the model? Explain issues in case you are not able to fully utilize the hardware.

Hint: Assume periodic boundary conditions and take care of the first and the last temporal iteration (indices...)!

Hint: Flush the cache between time measurements to ensure that you always read the data from DRAM! (We should do this because our benchmark size is small and will benefit from the cache otherwise.)

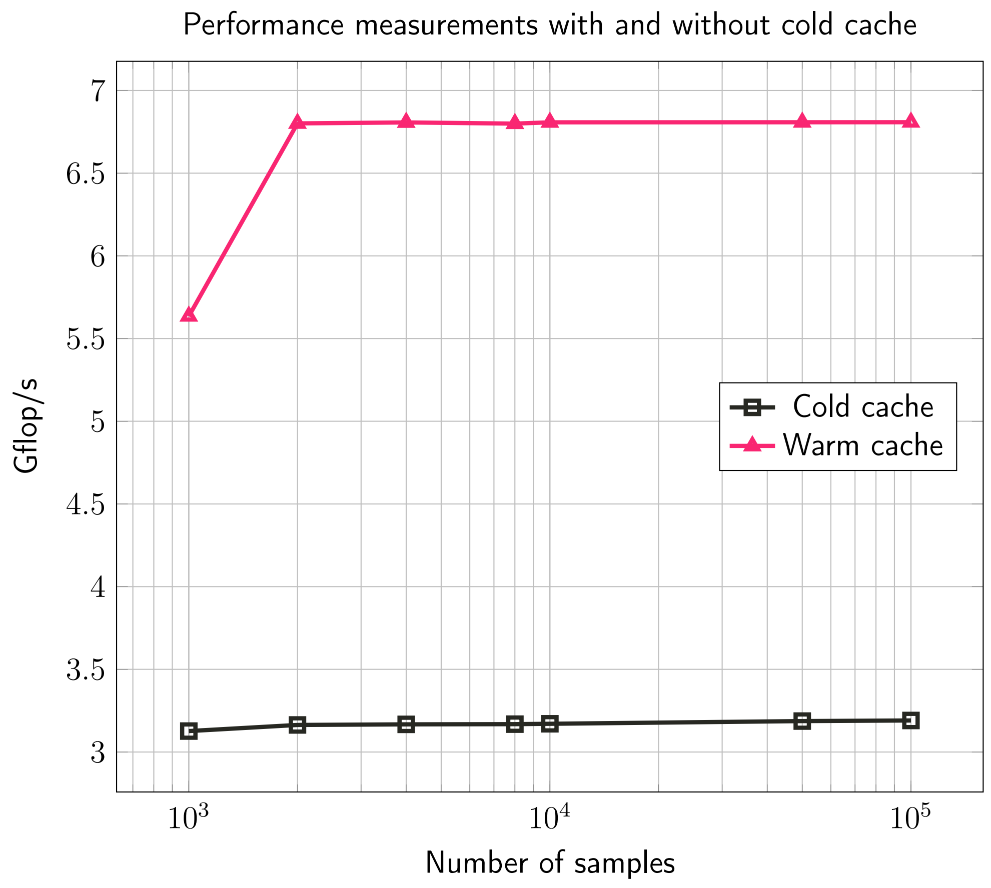
(45 points) The source code for this exercise is in the `benchmark_1Ddiffusion` directory. The number of grid points is set to $N = 2^{20}$, for which the memory footprint of the grid is 8 MB. To prevent overwriting values at time level t^n, a second grid is required to store the updated values at time level t^{n+1}. Both grids are small and will fit into the last level cache.

Because the update for one single grid point is too small to measure, we measure the time it takes for a full sweep over the grid. To reduce further measurement error, like fluctuations in the clock frequency, we measure over a number of sweeps and average the measurements. If we decide to do this, we must ensure that for every sweep the read and write access to the grid data is from DRAM, since the operational intensity is defined by accesses to DRAM (cold cache). This can be achieved by polluting the cache with junk data before every sweep measurement. Notice that the cache flushing should be declared as `volatile`. The reason for this is the following: If a function has no effect, modern compilers consider it as unnecessary and will optimize it away. In other words: No cache flushing will take place! Declaring a variable as volatile tells the compiler that it is not allowed to optimize memory accesses (reads and writes) to/from that variable.

For this naive implementation we expect a peak performance of 2.5 Gflop/s on the Euler nodes (single core, no vectorization and not perfectly balanced `mul` and `add` instructions). Measurements over 1000 sweeps on Euler yields 3.16 Gflop/s using GCC with `-O3 -march=core-avx2 -ffast-math -funroll-loops` optimizations. We measure slightly higher performance than expected since the scheme in Equation (10) contains a mix of additions and multiplications. The Haswell architecture has two execution units for multiplication and one for addition which can not be optimally utilized with the given mix of operations. Furthermore, variation in the processor frequency may also influence the expected result.

Running the same experiment without resetting the cache to a cold state results in a performance of 6.81 Gflop/s. This is due to the reuse of data that is still contained in the cache from the previous iteration. These memory accesses can be performed faster than access to DRAM and therefore result in higher performance. The two measurements are indicated in the roofline plot above (see question 4.c) for the operational intensity $I_2 = 0.313$ (see question 3.b)).

Given that the problem is memory-bound, the next optimization that should be taken into account is parallelization with OpenMP for example. Using all threads will saturate the memory bus before the code would benefit from further low-level optimizations like explicit vectorization.



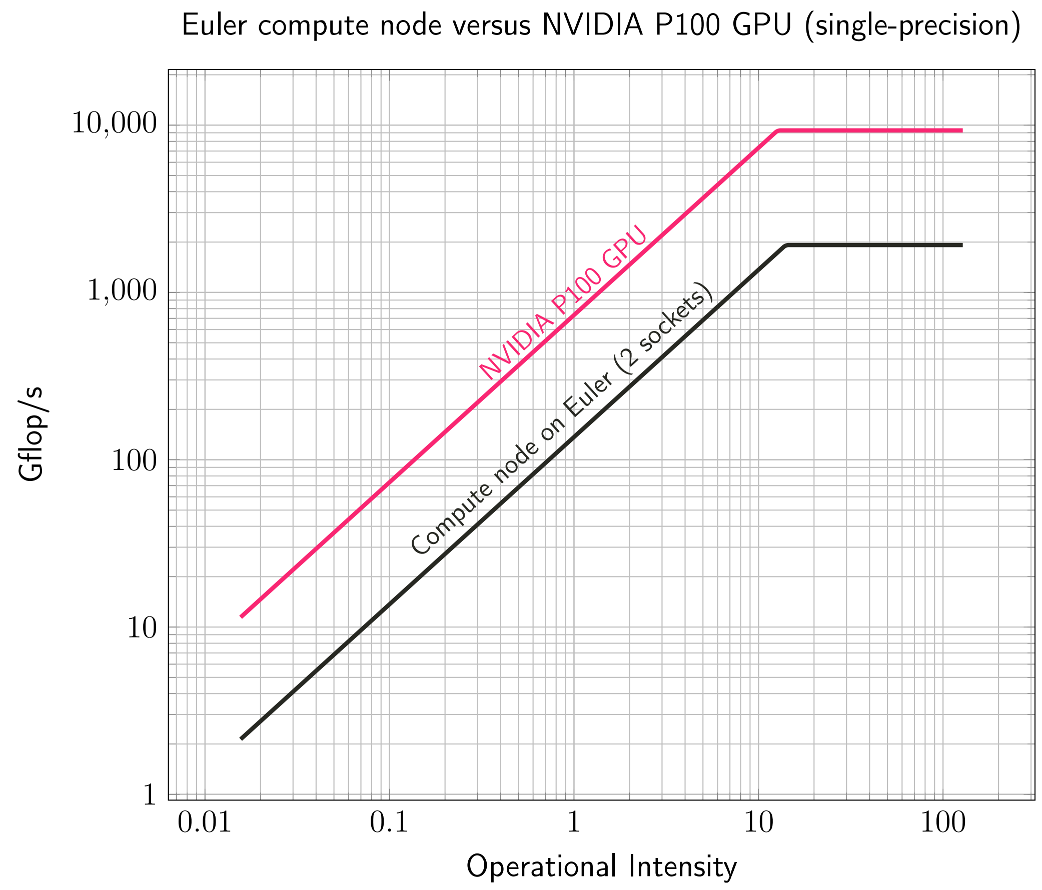
The position of the code in a roofline plot is depicted in the solution plot of the last subquestion.

Points:

- 20 for a working code
- 15 points for performing the measurement properly, i. e. measure many iterations and correctly flushing the cache
- 5 points for running the code on Euler
- 5 points for giving a short description of the measurement outcome and a brief discussion/interpretation thereof

e) (Optional). The Swiss National Supercomputing Centre (CSCS) in Lugano is home to the Piz Daint supercomputer. The machine reaches a performance of 19.6 PFLOP/s based on the Linpack benchmark and is currently the sixth fastest computer in the world (https://www.top500.org/lists/2018/06/, as of June 2018). Piz Daint is composed of Cray XC50 compute nodes which are hybrid nodes with GPU accelerators. Go to http://www.cscs.ch/computers/piz_daint/index.html and find out the specifications of the GPUs. Based on the numbers you have researched, draw the roofline for the GPU. Compare this roofline with the one you have drawn in part 4c. Describe the differences you observe in terms of how it would impact you as application developer.

GPU accelerators offer roughly $10\times$ higher peak performance. This means that the code must exploit a very high data-parallelism to reach the compute bound region. Unless your application does not exploit a very high density of fused-multiply-add instructions, it will be very difficult for a general application to reach peak performance of a GPU. If the application is memory bound, the maximum speedup that can be gained on a GPU is limited to the faster bandwidth and corresponds to roughly $5\times$ compared to current bandwidths available for CPUs.


Euler compute node versus NVIDIA P100 GPU (single-precision)
Gflop/s
Operational Intensity
NVIDIA P100 GPU
Compute node on Euler (2 sockets)