ETH zürich

High Performance Computing for Science and Engineering I

P. Koumoutsakos, M. Troyer
ETH Zentrum, CLT F 12
CH-8092 Zürich

Fall semester 2014

Set 6 - Roofline Model and Performance Measures

Issued: October 31, 2014
Hand in: November 7, 2014, 8:00am

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, finite element analysis programs, 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 (PP) in FLOP/s can be computed as in the following:

$$PP\ [FLOP/s] = f\ [HZ = cycle/s] \times c\ [FLOP/cycle] \times v\ [-] \times n\ [-] \tag{1}$$

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 floats) and n the number of cores.

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 here in 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 (PB) in B/s can be computed as follows:

$$PB\ [B/s] = f_{DDR}\ [Hz = cycle/s] \times c\ [channel] \times w\ [bit/channel/cycle] \times 0.125\ [B/bit] \tag{2}$$

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`).

Memories based on the DDR (Double Data Rate) technology, such as DDR-SDRAM, DDR2-SDRAM, and DDR3-SDRAM[1], transfer two data per clock cycle. As a result, they achieve

[1] `http://en.wikipedia.org/wiki/DDR3_SDRAM`

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 data per clock cycle, and thus are labeled as being a "1,333 MHz" device, even though the clock signal does not really work at 1.866 GHz.

Question 1: Roofline Model

The roofline model[2] is a simple visual tool that can be used to understand performance on a multicore architecture.

The hardware architecture is abstracted to its memory bandwidth and its peak floating point performance.

For this exercise you will learn how to use the Roofline model for the solution of the diffusion equation with Finite Differences.

a) According to the wiki of the Euler cluster, each 24-core Ivy-Bridge compute node is capable of delivering 576 GFLOP/s. In addition, the theoretical memory bandwidth of each of the two NUMA nodes of a single compute node can reach 59.7 GB/s. Try to justify the above numbers.

   $PP = 3\ [GHZ = cycle/s] \times 2\ [FLOP/cycle] \times 4\ [-] \times 24\ [-] = 576\ GFLOP/s$ (in double precision)

   $PB = 1.866\ [GHz = cycle/s] \times 4\ [channel] \times 64\ [bit/channel/cycle] \times 0.125\ [B/bit] = 59.7\ GB/s$

b) Draw the roofline for the 24-core Ivy-Bridge node of Euler. Note that the bandwidth is found as the ratio of GFLOP/s performance to operational intensity.

   Figure 1 shows the roofline for the 24-core Ivy-Bridge node of the Euler cluster. The plot also shows the ceilings for plain C++ codes (3 GFLOP/s), the addition of multithreading (72 GFLOP/s) and vectorization (576 GFLOP/s) as well as the increased bandwidth from 1 (59.7 GB/s) to 2 NUMA nodes (119.4 GB/s). Furthermore, plain C++ codes that manage to balance multiplications and additions can double their achieved performance (6.0 GFLOP/s and 144 GLOP/s on 1 and 24 cores, respectively).

c) Compute the number of FLOP, bytes of data read and written and the operational intensity for a single timestep of diffusion using explicit Euler (use your code from Exercise 3 as a reference):

$$\rho_{i,j}^{n+1} = \rho_{i,j}^{n} + D\delta t \frac{\rho_{i+1,j}^{n} + \rho_{i-1,j}^{n} + \rho_{i,j+1}^{n} + \rho_{i,j-1}^{n} - 4\rho_{i,j}^{n}}{\delta x^2} \tag{3}$$

   As the value at each grid point is computed in the same way, we count the FLOP and memory accesses per grid point.

   In the above formulation we count 7 floating point operations per grid point. There are 3 extra operations that we do not count as they are between constant values $\frac{D\delta t}{\delta x^2}$ and thus can be precomputed.

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

[2]Williams et al, 2009: the original paper can be found on the course website.

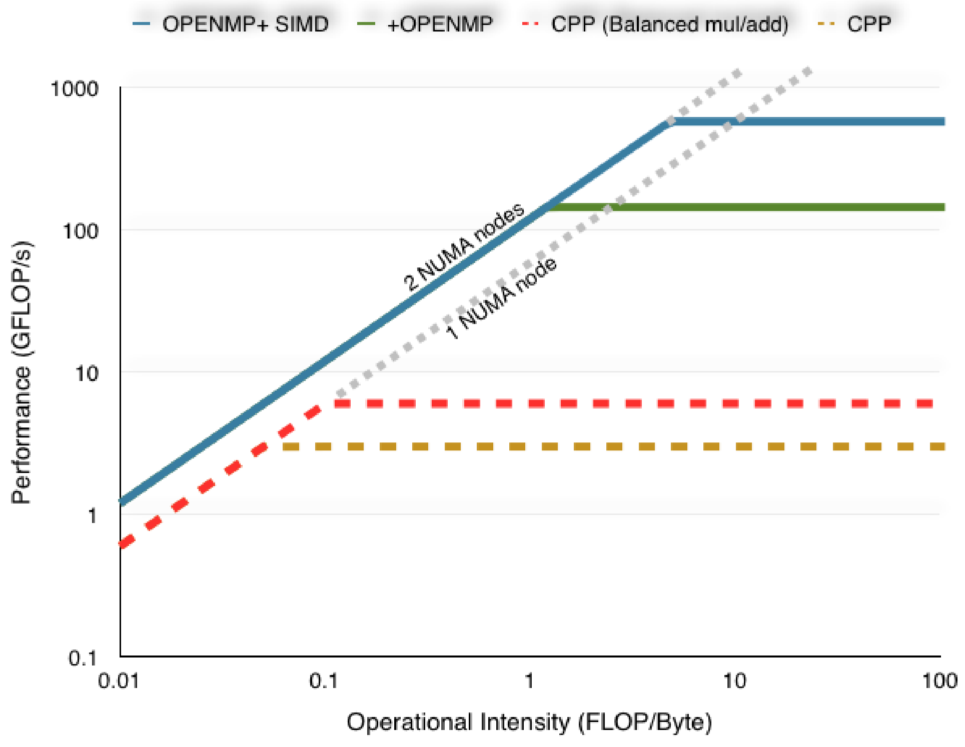

Figure 1: Roofline of the 24-core Ivy-Bridge node from the Euler Cluster with ceilings.

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 6 read and 1 write operation per grid point ($\rho_{i,j}$ is read twice and the computation could be easily reformulated to reduce the number of read operations). The total number of bytes moved for data in double precision is therefore $(6+1)\cdot 8 = 56$ [B].
2. With an infinite sized cache, we only need two memory operations per grid point (1 read and 1 write). Therefore, in double precision, the amount of data moved to and from main memory is $(1+1)\cdot 8 = 16$ [B].
3. If we assume the data is stored in row-major order (or column-major order) and the grid is sufficiently large, we can assume that $\rho^n_{i-1,j}$ and $\rho^n_{i,j}$ (or $\rho^n_{i,j-1}$ and $\rho^n_{i,j}$) are in cache from the previous iterations. Therefore, at each iteration we need to read 3 points and write 1 value. The total amount of data moved in this case is $(3+1)\cdot 8 = 32$ [B]. A reorganization of the memory layout that uses blocking would be able to reduce the number of reads by one value.

The operational intensities for the three cases are

$$\frac{7}{56} = 0.125 \text{ [FLOP/B]}, \tag{4}$$

$$\frac{7}{16} = 0.4375 \text{ [FLOP/B]} \tag{5}$$

and

$$\frac{7}{32} = 0.21875 \text{ [FLOP/B]} \tag{6}$$

respectively.

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.

d) Show the operational intensity on the plot. Is your implementation memory bound or compute bound? What is the maximum performance that your code can reach according to the model?

Figure 2 shows the three operational intensities for the diffusion operator on the roofline plot. From the operational intensity we can estimate the maximum reachable performance with

$$\text{min}(OI \cdot B, P), \tag{7}$$

where B is the bandwidth, P the peak floating point performance and OI the operational intensity.

For the three cases considered above, we therefore estimate the reachable performance to be:

1. min$(0.125 \cdot 119.4, 576)$ [FLOP/B] $= 14.9$ [FLOP/B]
2. min$(0.4375 \cdot 119.4, 576)$ [FLOP/B] $= 52.2$ [FLOP/B]
3. min$(0.21875 \cdot 119.4, 576)$ [FLOP/B] $= 26.1$ [FLOP/B]

According to the model, the computation of the diffusion is memory bound for all three cases.

e) Measure the performance (in GFLOP/s) of your code from Exercise 3 or the code provided in the master solution. Put the results as points in your roofline plot. How efficient are the two codes compared to the expected value? What could be done to get closer to the theoretical peak performance?

For the diffusion on a grid of 1024×1024 computational elements and 1000 time steps, we measure a time of 0.177 seconds:

$$\frac{7 \cdot 1024^2 \cdot 1000}{0.177} \cdot 10^{-9} \text{ GFLOP/s} = 41.4 \text{ GFLOP/s} \tag{8}$$

For the diffusion on a grid of 2048×2048 computational elements, i.e. for total memory footprint that exceeds the cache size, and 1000 time steps, we measure a time of 1.027 seconds:

$$\frac{7 \cdot 2048^2 \cdot 1000}{1.027} \cdot 10^{-9} \text{ GFLOP/s} = 28.6 \text{ GFLOP/s} \tag{9}$$

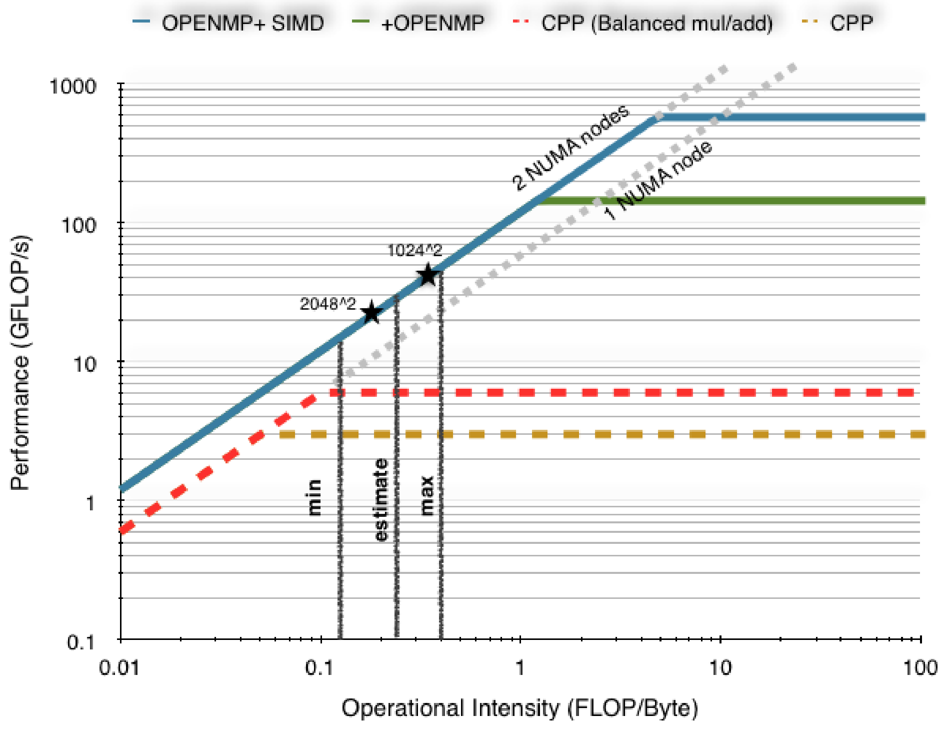

Figure 2: Operational intensities for diffusion on the roofline for double precision. (i) is denoted by "min", (ii) by "max" and (iii) by "estimate".

The red dot in Figure 2 shows the performance measured for the code provided in the solution of Exercise 3. We observe that our measured performance is within the lower and the upper bounds computed, which is expected.
The first optimization to apply is to increase locality, for example by blocking and tiling the computational domain, in order to move the operational intensity away from the memory bound region and improve the potential contributions given by multithreading and vectorization. Even with increased locality however, the operational intensity will never be larger than our estimated upper bound. To increase even more the operational intensity we would need to change the algorithm used to solve the diffusion equation.

Question 2: Performance Measures

Sometimes the computation of the peaks of a platform is not sufficient as you might want a more realistic upper bound on what performance you can effectively reach with a program. The objective of this exercise is to get as high as possible with your current knowledge.

When designing a benchmark for measuring the peak bandwidth and peak floating point performance of a platform, we have to consider that the two resources are competing. This means that when we try to stress one, the usage of the other should be kept to a minimum to reduce its influence. In addition, we show the results obtained with a code written purely in C++ and parallelized with OpenMP.

a) Write a C/C++ benchmark to measure the peak single precision performance of a given platform. Report the method you used to measure the peak performance. Try your code on a compute node of the Euler cluster. Can you reach the theoretical peak performance reported before? Please explain your observations.

Solution code in `src/computerpower.cpp`

We try to measure the peak performance by using implementing the following computation:

$$y = (...(((x * a_0 + b_0) * a_1 + b_1) * a_2 + b_2)...)a_n + b_n \tag{10}$$

where a_i and b_i, $i = 0, .., n$ are constant values.

The interesting characteristic about this scheme is that it only requires a single memory access for x and one to write the result y back and the degree of the polynomial can be increased at will so that the actual number of multiplications and additions can be freely controlled. Because of this, we can be sure that the bandwidth is not going to be a bottleneck for the benchmark.

With `./computepower` and `OMP_PROC_BIND` equal to $TRUE$, we measured 5.92 GFLOP/s (the 90th percentile) for 1 OpenMP thread and 141.91 GLFLOP/s for 24 threads, which are around 98.5% of the nominal peak performance for a computation with balanced multiplications and additions.

b) Write a C/C++ benchmark to measure the peak memory bandwidth of a given platform. Report a short description of the method you use in the benchmark. Plot the measured bandwidth against the number of threads.

Disable any OpenMP pragmas and run your program on a single core (i.e one thread) for different memory footprints (total size of your data set). Starting your measurements for 10KB of memory footprint. Plot using a log-log scale the bandwidth against the memory footprint. What do you observe in the plot?

Solution code in `src/bandwidth.cpp`

The simplest way to measure the bandwidth is to copy an array into another. This technique cuts out completely the floating point computations and depending on the problem size, is able to measure the bandwidth of the various memories in the memory hierarchy.

Figure 3 shows the memory bandwidth as measured with `./bandwidth` and having allocated our arrays either with malloc(), provided by the standard C library, or with an allocation routine provided by the libnuma library. The latter distributes the memory allocate across the two NUMA nodes and allows us to achieve a maximum memory bandwidth of approximately 60GB/s, which is around 50% of the nominal peak memory bandwidth. If we replace the copy operation with an addition of three arrays, the measured memory bandwidth increases to 67.6GB/s (57% of the theoretical peak).

Figure ?? shows the bandwidth measured with the code provided, executed on a single core 24 cores Brutus node with `./bandwidth -range -size 1024`. The memory is allocated with malloc() and the OpenMP pragmas has been disabled. From the plot we can observe different regimes that can be identified with the cache hierarchy of the platform.

The Ivy-Bridge has 32 kB of L1 memory per core (marked in light blue), 256 kB of L2 cache per core (marked in blue is the sum of L1 and L2 caches) and 30 MB of L3 cache shared among the 12 cores (the vertical dark blue line delimits the sum of the three levels of cache). If we only use one core, that core can make use of 30 MB of L3. Below that, in the memory hierarchy, we find the DRAM memory.

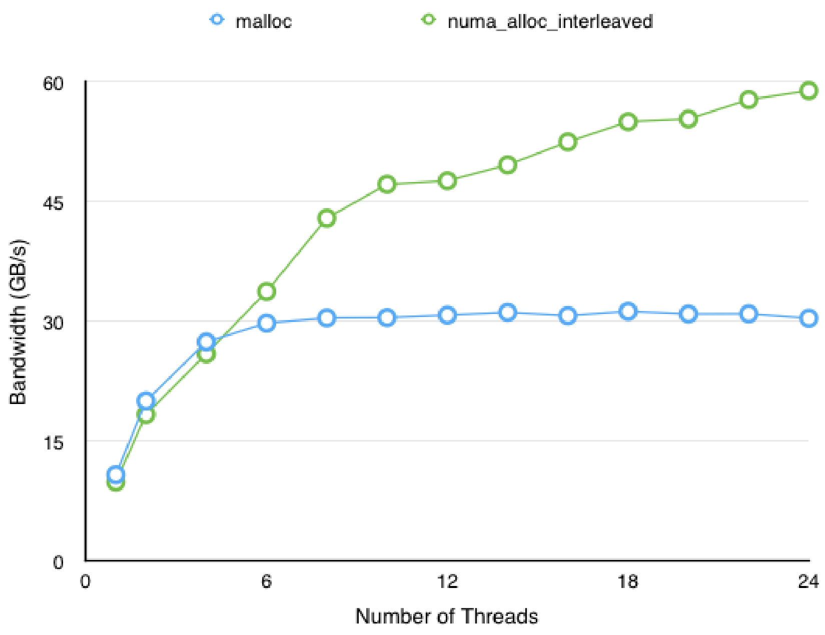

Figure 3: Measured memory bandwidth vs number of threads, using malloc() and numa_alloc_interleaved().

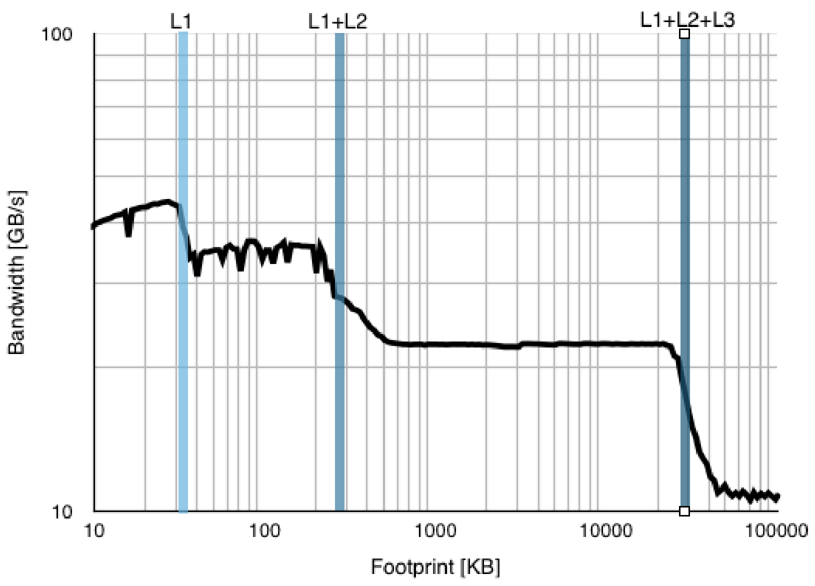

Figure 4: Memory bandwidth versus memory footprint. The blue vertical lines denote the cumulative cache sizes: light blue marks the size of the L1 cache (32 kB), blue marks the size of the sum of L1 and L2 caches (32 kB + 256 kB), dark blue marks the total of L1, L2 and L3 caches (32 kB + 256 kB + 30 MB).

Summary

Summarize your answers, results and plots into a PDF document. Furthermore, elucidate the main structure of the code and report possible code details that are relevant in terms of accuracy or performance. Send the PDF document and source code to your assigned teaching assistant.