Roofline by Example

October 2016 - Romain Dolbeau
Introduction (1)

- The « roofline model », named after its appearance similar to a physical roof on a house, is a visual method to evaluate the optimality of a code.

- The fundamental idea is to plot the current situation of a code (or a kernel, a function, etc.) in relation to theoretical upper bounds on performance.

- Usually, the Roofline is made up of a two dimensional graph, with:
  - Horizontally, the arithmetic intensity of the code: the ratio of the amount of computation to the amount of memory accesses. It is typically expressed in “flops/byte”.
  - Vertically, the computational performance of the code. It is typically expressed in “flops/s” (number of floating-point operations per second).
Introduction (2)

- The upper bound a code can reach depending on its arithmetic intensity is limited by two parameters:
  - The theoretical peak of the system, in “flops/s”. On the graph, this is an horizontal line (constant for all arithmetic intensity)
  - The memory bandwidth of the system, in “bytes/s” (bytes of data that can be loaded from memory per second). On the graph, it is a diagonal, going up with arithmetic intensity

- The next graph illustrate those limits with logarithmic-logarithmic scales. It shows theoretical limits for processors of five families from Intel Nehalem (X5570) to Intel Haswell (E5 2640v3)
  - The red diagonal line plots arithmetic intensities for which the performance bound is dictated by memory bandwidth. Here, a code would not be compute-intensive enough and the memory bandwidth is too low to feed the computations
  - The blue horizontal line plots the upper bound of performance (in double precision) of the processor. In this area of high arithmetic intensity, a code would have enough bandwidth to satisfy its demand, and would be limited by the processor’s raw performance
Introduction (3)

We observe that with each newer generation, the inflection point (where arithmetic intensity is in perfect balance with available bandwidth and raw performance for the platform) is moving to the right, that is, toward higher arithmetic intensity.
Creation of the Roofline (1)

Theoretical or practical limits

- The graph from the previous slide uses the theoretical limits of the processors
- Memory bandwidth is limited by the theoretical values of the controllers
  - For instance, 4 channels of DDR3 64 bits at 1866 MHz have an aggregated bandwidth of approximately 60 GigaBytes/second
- The theoretical peak is limited by the saturation of execution units at nominal frequency — an E5-2600v3 has therefore a theoretical bound, according to the formula below, of \( 2 \times 1 \times 2 \times (2.0 \times 10^9) \times 8 \) for a single socket, or 332.8 GigaFlops/s in double precision.

\[
\frac{\text{flops}}{\text{node}} = \frac{\text{operations}}{\text{instruction}} \times \left( \frac{\text{micro}}{\text{architecture}} \right) \times \left( \frac{\text{cycles}}{\text{second}} \right) \times \left( \frac{\text{instructions}}{\text{cycle}} \right) \times \left( \frac{\text{cores}}{\text{socket}} \right) \times \left( \frac{\text{sockets}}{\text{node}} \right) \times \left( \frac{\text{machine}}{\text{architecture}} \right)
\]
Creation of the Roofline (2)

► Those limits are rarely, if ever, attainable by real-life codes

► It is possible to replace – or supplement – those limits by more practical ones
  – i.e., the performance reached by codes known to saturate either one of the parameters

► The most common examples are
  – The benchmark STREAM for a practical memory bandwidth upper bound
  – A highly optimized large, dense matrix multiplication for the compute performance

► But using values from such benchmarks requires some care
  – If they are not sufficiently optimized, they may show limits lower than what could be ultimately achieved
  – If exceedingly specific optimizations are used, the resulting limits might not be reachable by codes that cannot leverage such optimizations
Additional limits

- Other limits may appear on a Roofline graph
  - Memory bandwidth is only a partial view of the issues related to memory accesses
  - Multiple levels of caches have a critical part to play
  - It’s not always easy to pick a cache bandwidth limit, in particular for non-shared caches
  - On a contemporary Intel processor, the third level cache (L3 cache or LLC, for Last Level Cache) is shared between cores and can be displayed like memory
  - But level 1 and 2 caches are replicated per-core, and one should consider the aggregated bandwidth of all caches of the same level when considering multiple cores
Lowering the limits

- A code might not be able to use all the features of the hardware
  - If the compute part of the code cannot be vectorized, the vector-inclusive limit won’t be very useful
    - It might be preferable to add a theoretical peak without vectorization
  - If the compute part of the code cannot exploit specific instructions such as Fused-Multiply-Add (FMA)
    - A code with a strong bias toward either additions or multiplications will not be able to reach the peak if this peak is largely made by the ability of the hardware to compute FMAs
      - For instance, for a code doing two multiplications for each addition on a pure FMA machine such as Intel Haswell, the theoretical peak will only be 75% of the machine peak. ¼ of the computations the hardware is capable of are extra additions the code will never use

- The goal is to create a realistic view of what the code could reach after optimizations
Plotting a code

- Now the goal is to add a point on the roofline graph, representing the state of a code (or a kernel, a function, etc.)
  - This point will be defined by two coordinates: arithmetic intensity of the code (horizontally) and the performance in flops/s of the code (vertically)

Arithmetic intensity

- The arithmetic intensity of the code is a simple ratio - the number of floating-point operations divided by the amount of memory

- The first element is the compute part
  - If the code only contains trivial operations (additions, multiplications) without redundancy in perfectly nested loops, then the theoretical number of operations and the number of operations really executed should be very close
  - If the code contains redundancies, those may or may not be factorized by the compiler
  - The code might also contains more complex operations such as square roots or divisions, etc.

- So there could be several different values for the compute part...
Floating-point operations in a code (1)

- The algorithmic number
  - This is what the theory says about the algorithm
  - For instance, the benchmark "benchfft" counts operations for a size-FFT. It's a useful approximation, since the dynamic number of operations can vary wildly depending on the implementation. Similarly, we can consider that for a real-valued matrix multiplication, there is approximately operations

- The implementation number
  - This is what the implementation of the algorithm in a high-level language shows
  - Typically, the FFTW3 library supports many algorithms – and combinations thereof – to implement a FFT. It is theoretically possible to count the number of operations for a specific implementation

- The dynamic number
  - This is what the processor really computes
  - The compiler has done its job of the high-level implementation, with optimizations and possible inefficiencies. Then we measure, one way or another, the dynamically executed instructions on the processor
Floating-point operations in a code (2)

- The value to be taken into account depends on the context and the purpose of the creation of the roofline
  - The third (dynamic) is the closest to what the hardware does, and therefore should give the best view of the proximity to the limits
  - But if there is an issue in the implementation or the compilation, the dynamic number of flops could be higher (or lower) than necessary, and so could give a biased view of the situation

- If a high-level code contains redundancy, the question is whether to count them or not
  - If those redundancies are counted but the compiler optimize them away, we are going to overestimate the performance of the code
  - If the redundancies are ignored but the compiler keeps generating instructions for them, the code will be seen as less efficient than it is
    - A different interpretation is that this lower-than-expected efficiency from the code compared to the algorithm/the implementation is a clue something went wrong during compilation
Memory accesses (1)

Memory accesses can also be counted several ways
- Like for the compute part, what we are going to count depends on what we expect to achieve by creating the Roofline

Even more than computations, memory accesses can be affected by the behavior of the compiler
- A data element used several times can be loaded one or more time depending on optimizations and registers availability
- Those potential reuse can also be affected by the memory hierarchy - some accesses can be served by the first, second, etc. level cache, in ways not always easy to establish

Like for computations, there at least three ways of counting memory accesses
Memory accesses (2)

The algorithmic number
- This is what the theory says about the algorithm
- For an FFT “out of place” (with an input array different from the output array), all the inputs have to be read once and all the results have to be written once.
- So for an FFT of size $n$, $n$ reads and $n$ writes
- And for a matrix multiplications of square matrices of size $n$ defined by $x \times n$, exactly $2 \times n^2$ reads and $n^2$ writes

The implementation number
- This is what the implementation of the algorithm in a high-level language shows
- Some duplicate accesses the compiler cannot factorize (not in the same loop, etc.) will show up here. Accesses to intermediate arrays required by the implementation but not the algorithm are counted here as well

The dynamic number
- This is what the processor really computes
- Accesses the compiler could have factorized but didn’t, “register spilling”, etc., are counted here
Memory accesses (3)

- The value to be taken into account depends on the context and the purpose of the creation of the roofline
  - The algorithmic value is often the easiest to evaluate, in particular if the algorithm reads and/or updates the entirety of the arrays in use
  - All other accesses linked to the implementation or the compilation can be considered a source of performance loss, and therefore a potential target for optimizations

- The definition of the algorithmic value can vary depending on the granularity at which we consider the problem
  - If a loop nest reads arrays A & B and writes array T, while the next loop nest reads T and writes C, and T is never used elsewhere in the code
    • T is algorithmically useful when looking at the loop nests independently
    • T is a temporary array linked to the implementation when looking at both loop nests combined in a kernel
Performance of a code

- For the Roofline, we express performance in flops/s

- The flops aspect is identical to the arithmetic intensity
  - Therefore there can be several values than can be used

- Time is theoretically easier as we measure the real execution time of the code (kernel, function ...)
  - Depending on the duration of what we measure, we fall back on the usual methods, from the “time” command for a full code to a cycle hardware counter for a loop nest, or perhaps functions such as “gettimeofday” for long-running functions
A practical example: code

A « n-body » simulation, specifically the interaction between particles

- Many possible ways to implement this part of the algorithm
- We consider this versions of the code
  - Including a spurious sqrt(), kept as it is useful for our teaching purpose

```c
for (i = 0; i < size; i++) {
    // loop 1
    float tmpax=0.0f;
    float tmpay=0.0f;
    float tmpaz=0.0f;
    for (j = 0; j < size; j++) {
        // loop 2
        float distance[3];
        float distanceSqr = 0.0f, distanceInv = 0.0f;
        distance[0] = objects_x[j] - objects_x[i];
        distance[1] = objects_y[j] - objects_y[i];
        distance[2] = objects_z[j] - objects_z[i];
        distanceSqr = sqrtf(distance[0]*distance[0] + distance[1]*distance[1] + distance[2]*distance[2]) + softeningSquared;
        distanceInv = 1.0f / sqrtf(distanceSqr);
        tmpax += distance[0] * G * objects_mass[j] * distanceInv * distanceInv * distanceInv;
        tmpay += distance[1] * G * objects_mass[j] * distanceInv * distanceInv * distanceInv;
        tmpaz += distance[2] * G * objects_mass[j] * distanceInv * distanceInv * distanceInv;
    }
    objects_ax[i] += tmpax;
    objects_ay[i] += tmpay;
    objects_az[i] += tmpaz;
}
for (i = 0; i < size; ++i) {
    // loop 3
    objects_vx[i] += objects_ax[i] * timestep * damping;
    objects_vy[i] += objects_ay[i] * timestep * damping;
    objects_vz[i] += objects_az[i] * timestep * damping;
    objects_x[i] += objects_vx[i] * timestep;
    objects_y[i] += objects_vy[i] * timestep;
    objects_z[i] += objects_vz[i] * timestep;
    meanv += objects_vx[i] + objects_vy[i] + objects_vz[i];
```
A practical example: structure

This code has already been partially optimized, such as the presence of variables like “tmpax” and similar. They allow to explicitly remove access to the “objects_ax” arrays and similar in the innermost loop (loop 2).

The global structure of the code is the following:

```c
for (i = 0; i < size; i++) { // loop 1
    // COMPUTE LOOP 1, BEGIN
    for (j = 0; j < size; j++) { // loop 2
        // COMPUTE LOOP 2
    }
    // COMPUTE LOOP 1, END
}
for (i = 0; i < size; ++i) { // loop 3
    // COMPUTE LOOP 3
}
```
FLOPS high level (1)

- We can easily count floating-point operations in each different block. We know the blocks in loop 1 and 3 are computed “size” times, while the block in loop 2 is computed “size$^2$” times
  - Compute loop 1, begin : 0 flops
  - Compute loop 1, end : 3 additions
  - Compute loop 3 : 9 additions & 9 multiplications
  - Compute loop 2 :
    - 3 subtractions
    - 3 additions, 3 multiplications, 1 square root
    - 1 reciprocal of square root
    - 3 additions & 15 multiplications

- First remark: no division in this list
  - One of the peculiarity of the x86-64 architecture is the availability of an instruction “approximate the reciprocal of a square root”.
  - We could count this line of code as a division (or a reciprocal) and a square root, but a close look at the assembly code produced by the compiler confirms it uses the specific instructions for the square roots and their reciprocals
FLOPS high level (2)

- Second remark: the last block has 15 multiplications
  - This count, apparently exact, is actually an upper bound
  - If we were to pre-compute “G * objects_mass[j] * distanceInv * distanceInv * distanceInv”, there is only 7 multiplications left in this block. 4 are in pre-computation, and 3 are in the update of “tmpax” and similar
  - Using the code without explicit pre-computations, the compiler could optimize all or some of that code, and therefore produce between 7 and 15 multiplications for this block

- Third remark: similar to the second but for block 3
  - The expression “timestep * damping” in this function is in fact a product of constant
  - Not only can it be factorized (removing two multiplications), but it can be moved out of the loop, removing 3 of the 9 multiplications
FLOPS high level (3)

So from a high language point of view there is two bounds for this count:

- Upper bound:
  - additions
  - Subtractions are assimilated to additions
    - \( \text{subtractions} = \text{additions} \)
  - multiplications
    - \( \text{multiplications} = \text{size} \times 16 + \text{size} \times 9 \)
    - square roots and reciprocal of square roots
      - \( \text{roots} = \text{size}^2 + 6 \)

- Lower bound:
  - additions
    - \( \text{additions} = \text{size} \times 3 + \text{size}^2 \times 9 + \text{size} \times 9 \)
  - multiplications
    - \( \text{multiplications} = \text{size} \times 10 + \text{size} \times 6 \)
    - square roots and reciprocal of square roots
      - \( \text{roots} = \text{size}^2 + 0 \)
FLOPS measured (1)

For a known “size” parameter, we can compare the theory with what is really measured while the code is running.

In this example, which is part of a synthetic test case, the total number of FLOPS is easy to compute: “size” is going to be 5000, and the function will be called 200 times. So we should have additions and between multiplications and square roots and reciprocals of square root.

To get the exact number of FLOPS, it could be possible to exploit hardware performance counters.

For this example, we are going to use the Intel® Software Development Emulator.

- Initially designed to simulate future instruction sets, the SDE can also be used to count dynamic instructions in a running code.
- As it is originally an emulator, it counts instructions with perfect accuracy.
FLOPS measured (2)

- The SDE outputs
  - A detailed list of all the basic blocks in the code, the assembly instruction in the basic blocks, the number of times they were executed, etc.
  - A summary of the number of instructions of each type, globally and per-function
  - Knowing the number of instructions and the size of their operands, it’s easy to compute the number of Flops

- In the table on the next slide
  - The first two columns are from the output of the SDE, and indicate the names and numbers of compute instructions
  - The third column is the size of the vector for the instruction, 1 for scalar operations (name ends in SS) and 8 for vector operations
    - Their name ends in PS, registers are AVX with a width of 256 bits so 8 single-precision floating point values
  - The next three columns indicate the number of additions, multiplications and other operations encoded in the instruction semantic
    - A FMA includes both an addition and a multiplication
  - The last three columns indicate the total number of Flops of each type, plus the sums in red at the bottom
Sample SDE output (extracts)

# END_TOP_BLOCK_STATS
# EMIT_DYNAMIC_STATS FOR TID 0 OS-TID 38125 EMIT #1
# $dynamic-counts

(…)
TEST 2000400
VADDPS 6000000
VADDSS 9000000
(…)
VEXTRACTF128 3000000
VFMADD132PS 625000000
VFMADD213SS 6000000
VFMADD231PS 3125000000
(…)
VMOVAPS 625000000
VMOVHLPS 3000000
VMOVSS 24000600
VMOVUPS 1876000400
VMULPS 10000000000
VRSQRTPS 1250000000
## FLOPS measured (3)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Number</th>
<th>Vector width</th>
<th>Additions</th>
<th>Multiplications</th>
<th>Other</th>
<th>Total Add.</th>
<th>Total Mul.</th>
<th>Total Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>VADDPS</td>
<td>6 000 000</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>48 000 000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VADDSS</td>
<td>9 000 000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>9 000 000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VFADD132PS</td>
<td>625 000 000</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>5 000 000</td>
<td>5 000 000</td>
<td>0</td>
</tr>
<tr>
<td>VFADD213SS</td>
<td>6 000 000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6 000 000</td>
<td>6 000 000</td>
<td>0</td>
</tr>
<tr>
<td>VFADD231PS</td>
<td>3 125 000 000</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>25 000 000</td>
<td>25 000 000</td>
<td>0</td>
</tr>
<tr>
<td>VFMSUB213PS</td>
<td>625 000 000</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>5 000 000</td>
<td>5 000 000</td>
<td>0</td>
</tr>
<tr>
<td>VFMSUB231PS</td>
<td>625 000 000</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>5 000 000</td>
<td>5 000 000</td>
<td>0</td>
</tr>
<tr>
<td>VMULPS</td>
<td>10 000 000 000</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>80 000 000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VRSQRTPS</td>
<td>1 250 000 000</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10 000 000</td>
<td>0</td>
</tr>
<tr>
<td>VSUBPS</td>
<td>1 875 000 000</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>15 000 000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>55 063 000</td>
<td>120 006 000</td>
<td>10 000 000</td>
</tr>
</tbody>
</table>
FLOPS measured (4)

- If we make the hypothesis that the reciprocal of square root VRSQRTPS is used for the square root as well (it is, the compiler uses the equivalence \( \frac{1}{\sqrt{x}} = \frac{\sqrt{x}}{x} \)), number is fine.
- It's unfortunately the only one, since we have additions in excess, and between et multiplications in excess, or up to 140% excess multiplications...
- A large chunk of the excess comes from the fact the instruction VRSQRTPS computes an approximation, and not an exact value. This approximation must be refined.
  - And for the square root itself, at least an extra multiplication is needed.
  - If we examine carefully what the compiler does, it in fact adds an extra 3 multiplications and an extra addition for every (reciprocal of) square root.
- We have to add those operations in our theoretical formula.
- This raise the theoretical number of additions to 55 012 000 000, and the theoretical number of multiplications between 90 006 000 000 and 130 009 000 000.
FLOPS measured (5)

From those values we can observe three facts:

- There still some excess additions, but not in the inner loop
  - The order of magnitude matches with loop 1 or 3, in millions rather than billions with our “size” of 5000 and the number of calls of 200

- Two multiplications in the inner loop have been optimized away
  - One of the four redundant operations has been factorized, but not the other three
    - The factorized multiplication has been computed once instead of thrice, so two have been removed

- The redundant multiplication in loop 3 has been completely removed from the loop
  - Only 6 millions multiplications instead of 9 millions
FLOPS measured (6)

Additional remark: the ratio between vector operations (ending in “PS”) and scalar operations (ending in “SS”) is excellent
   All of the inner loop has been vectorized, the number of scalar operations matches the amount of computations in loops 1 and 3

The remaining overhead are the 6 millions vector additions and 3 millions of the 9 millions of scalar additions, for 51 millions additions
   The source seems to be the way the compiler generates some initialization phases in loop 1 and 3
   Those are “inefficiency” of the compiler

A small rewrite of the code to explicitly factorize the redundant multiplications in the inner loop allows to remove some multiplications, and to dynamically reach the lower bound of 90 006 000 000 multiplications displayed by the SDE
   This variant of the code is later called variant « without redundancy »
   This variant of the code is later called variant « without redundancy »
High-level memory accesses (1)

- The algorithmic number is very easy to compute in this case
  - Each body is described by 10 arrays: three \((x, y, z)\) for each of the three movement properties (position, speed, acceleration) so 9 arrays plus an array for the mass
  - The mass is read-only, all the others are fully updated in the function
  - Therefore we have bytes read and bytes written at every call
  - So for our dynamic example, with a “size” of 500 and 200 calls, 76000000 bytes transferred to or from memory

- If we compute arithmetic intensity counting 1 flops per multiplication, addition or reciprocal square root (the reciprocal square root is actually costlier, but this approximation doesn’t change the conclusion)
  - An arithmetic intensity in flops/bytes using the minimum number of Flops of about 2000
  - A value which while algorithmically exact, has little to do with the reality of the actual code
High-level memory accesses (2)

▶ The code computes interactions between every pair of particles, a computation quadratic in complexity (i.e. \( O(n^2) \))
- The storage is in turn linear (i.e. \( O(n) \))
- The ratio between the two is therefore in \( O(n) \), meaning the algorithmic arithmetic intensity is size-dependent and can be arbitrarily high. This is similar to matrix multiplication (compute in \( O(n^3) \), storage in \( O(n^2) \))

▶ From this property we can draw two conclusions:
- We require an implementation value to get an estimate closer to the reality of the current code
- There is in the implementation some memory re-use, which can be an optimization target in the future to bring the implementation value closer to the algorithmic value
Implemented memory accesses

- The two-loops implementation goes through every element in the “position” arrays in the inner loop.
- Others are only read or written in the outer loops.
- Because of this inner loop sweeping, there is no re-use between iteration of the inner loop.
- Therefore, there is a quadratic amount of memory read: \( n \times \text{sizeof(float)} \), or \( 300 \times 10^6 \) bytes per iteration (of the outer loop).
- Even ignoring all other arrays, the arithmetic intensity drops to only \( \frac{300}{2.6} \) or less than 2.6. But still a high value that should give good performance.
- But we can’t ignore the mass array “objects_mass”, also read sequentially in the inner loop and also contributing quadratically to memory accesses.
- Other arrays are only required to be accessed in the outer loops, and we can ignore their linear contribution.
- The final arithmetic intensity is now \( \frac{300}{2.6} \), or less than 2.
- Still a reasonable value, but a far cry from 2000!

- The two-loops implementation goes through every element in the “position” arrays in the inner loop.
- Others are only read or written in the outer loops.
- Because of this inner loop sweeping, there is no re-use between iteration of the inner loop.
- Therefore, there is a quadratic amount of memory read: \( n \times \text{sizeof(float)} \), or \( 300 \times 10^6 \) bytes per iteration (of the outer loop).
- Even ignoring all other arrays, the arithmetic intensity drops to only \( \frac{300}{2.6} \) or less than 2.6. But still a high value that should give good performance.
- But we can’t ignore the mass array “objects_mass”, also read sequentially in the inner loop and also contributing quadratically to memory accesses.
- Other arrays are only required to be accessed in the outer loops, and we can ignore their linear contribution.
- The final arithmetic intensity is now \( \frac{300}{2.6} \), or less than 2.
- Still a reasonable value, but a far cry from 2000!
Dynamic memory accesses (1)

► If we look back at the SDE output
  – We observe four instructions that could be a memory load or store: VMOVAPS, VMOVHLPS, VMOVSS et VMOVUPS
  – Unfortunately, the SDE summary does not discriminate between memory-register operations (in which we are interested) and register-register operations (of no interest for memory operations, though as they are useless from an algorithmic point-of-view, they could be a target for optimizations)

► In the code we are interested in, the compiler cannot prove data alignment
  – Since the VMOVAPS instruction requires a properly aligned address, they are in our case only used for register-register operations
  – Similarly, the VMOVHLPS which rearrange data inside vector registers are only used for register-register operations
  – The VMOVSS are few and are scalar operations; they only appear in outer loops
  – Only VMOVUPS are used as load operations
  – The number of memory store we expect is linear (outer loops) and not quadratic, and are taken care of by the VMOVSS

► But…
Dynamic memory accesses(2)

A peculiarity of the x86-64 architecture (by contrast with RISC architectures such as PowerPC, ARM, MIPS, SPARC, …) is that memory operations are not limited to loads and stores

- Any instruction may reference memory as an operand
- So we actually need to look at other instructions to really count all memory accesses

- In our code of interest, if we look at the assembly, all the memory accesses in the inner loop other than VMOVUPS are reads relative to general purpose register %RIP, the instruction pointer, with only one exception: Accesses relative to %RIP are usually accesses to constant values in the function
- The one exception is a multiplication with a simple vector access. This access is for the sequential access to the “objects_mass” array

The Intel SDE dynamically observes VMOVUPS, including in the innermost loop
- 875 is the number of vector elements
- This matches, with 32 bytes per loaded vector (AVX), to exactly 280 bytes. This is the theoretical value for our implementation for arrays “objects[xyz]”
- The compiler did not factorize accesses between iterations. It did not add any extra accesses to the code either
- The compiler did not factorize accesses between iterations. It did not add any extra accesses to the code either
Plotting the code on the Roofline (1)

- We still need to place on the roofline the points for our versions of the code (original and “without redundancy”) after measuring the execution time
  - So we need to pick which Flops and Memory values to use

- For memory, we’re going to pick the “real life” value, as seen by the SDE
  - Picking the theoretical value would guarantee such as high arithmetic intensity that the limit could only be the raw computational power and not memory bandwidth
  - This value is the same for both versions of the code, the comparison is easy

- For Flops, the “without redundancy” version of the code is easy: it explicitly eliminates redundancy, and the theoretical & measured values are almost identical

- So, for the original version with redundancies in the source code, should we pick the measured value or the lower theoretical value...
Plotting the code on the Roofline (2)

First, let’s plot the graph with measured values
- On the graph below is the Roof for a single core of a Xeon E5-2690v4 « Broadwell »
- The pink vertical cross is the original code
- The blue diagonal cross is the code “without redundancy”

On this roofline, the original code has a higher arithmetic intensity and a better “speed” (as in better exploiting the hardware)
- We could think the original code is “better”
- But its runtime is actually higher!
- The “speed” is higher, but this doesn’t make up for the higher amount of computations which (artificially) raise the arithmetic intensity
Placer le code sur le Roofline (3)

- We now plot on the graph (below colored cyan) the position of the original code, but with the theoretical number of Flops
- Naturally the arithmetic intensity is the same as the “without redundancy” code colored blue (same Flops, same Memory) but the “speed” is now lower
Improving the code thanks to the Roofline

- While studying memory accesses, we noticed the memory accesses were a potential target for optimization
  - All of the position arrays were used for every outer iteration, whose number is the “size” of the problem
- Theoretically, if we could factorize those accesses, we could raise arithmetic intensity and potentially speed-up the code, as long as associated costs of doing so are not too high
- Practically, there’s a very simple solution to test the theory: the “unroll_and_jam” compilation directive from the Intel compiler
  - Applied on a loop containing another loop, it asks the compiler to unroll the outer loop (the “unroll” bit) and then fusion the two copies of the inner loop produced by this unrolling (the “jam” bit)
- Theoretically, accesses in the inner loop should be used for two iterations of the outer loop but for the same amount of computations, doubling arithmetic intensity
- Practically, the amount of computations reaches the limits of available registers and there is some “spilling”
- Nonetheless, the code does become faster as is shown as a yellow point on the Roofline in the next slide
Final Roofline

- Original counting redundant flops
- Original ignoring redundant flops
- Code without redundancy
- Code without redundancy with unroll-and-jam
Thanks

For more information please contact:

romain.dolbeau@atos.net