Introduction to code tuning on modern multi-core processors

Martin Kronbichler

May 19, 2017

How does a processor work?

Consider matrix-vector multiplication $y = Ax$

```
for (int i=0; i<M; ++i)
{
  y[i] = 0;
  for (int j=0; j<N; ++j)
    y[i] += A[j*M+i] * x[j];
}
```

Inside the inner `j` loop, the CPU has to do the following actions:

1	Read next instruction	Compute offset `o=sizeof(double)*(j*M+i)` to array pointer `A`
2	Read next instruction	Load `a_ij=A+o` from memory
3	Read next instruction	Load `x_j=x+j*sizeof(double)` from memory
4	Read next instruction	Multiply `tmp = a_ij * x_j`
5	Read next instruction	Add `y_i = y_i + tmp`
6	Read next instruction	Increment loop counter `j = j+1`
7	Read next instruction	Compare `j < N`
8	Read next instruction	If comparison `true`, jump to back to position 1
	Instructions	Operations on data

Organization of a CPU: Instructions vs. data

Instruction path

- Machine code is decoded and execution resources are allocated
- Scheduling of **operations**
  - Some take longer than others
  - Some depend on previous ones
- Needs to be done (well) in advance of operations
- Branches (`if` statements, loop comparisons) and function calls change instruction flow
- Conditional branches: Jump value depends on data
- Jumps make it difficult to get a constant stream of instructions

Data path

- Integer operations (including pointers)
- Floating point operations ("actual work" in HPC)
- Memory operations (load/store)
- Some operations consist of up to 100 000 binary switches (e.g. floating point multiplication) $\rightarrow$ must be broken in several steps $\rightarrow$ FP multiplication takes 4–5 clock cycles on good processors

Organization of a CPU: Reach high performance

Difficulties: Instructions may take several cycles to complete, some depend on others

Solution: Instruction-level parallelism

- Operations are *pipelined*: Start second multiplication as soon as the first one leaves initial stage but before it is completely ready (pipeline length today: 12–20 stages)
- Input must be ready
- Change order of operations to find independent instructions (either by compiler or by *out-of-order execution* within CPU)

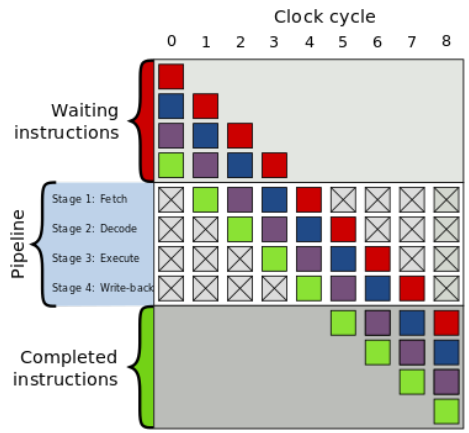


Image source: Wikipedia

- Reduce cost of branches by speculating on the outcome of comparisons (*branch prediction*)
- CPU typically have *several execution units* (Intel Haswell has 8) which can operate in parallel (e.g. memory load and addition)

Result: Modern CPUs and code execute 0.5–3 instructions per cycle

Balancing instructions and data operations

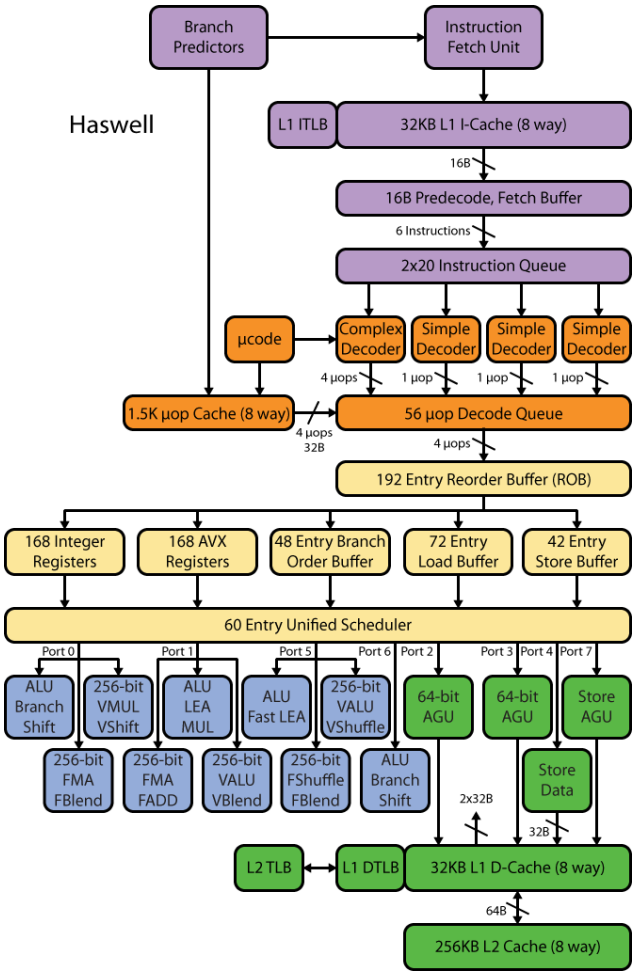


Image source:
http://www.realworldtech.com

Machinery to schedule and control instructions is expensive compared to actual operations

- Scalar code: 50–80% of energy consumption due to instruction control
- Balance this by **vectorization**: Perform operations for several data items at once

```
for (int i=0; i<20; ++i)
  a[i] = b[i]
       + c[i] * d[i];
```

```
for (int i=0; i<20; i+=4)
  a[i:i+3] = b[i:i+3] +
     c[i:i+3] * d[i:i+3];
```

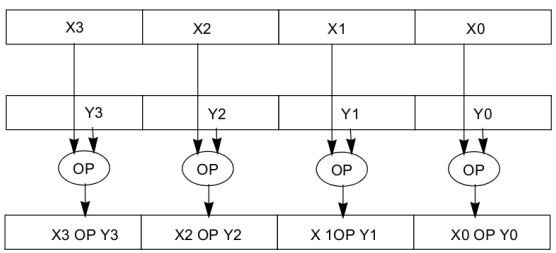


Image source:
Intel optimization manual

Vectorization

- Vectorization is single-instruction/multiple data (SIMD) concept
- Today's standards: 4 data items in double precision (AVX)
- Future: 8 items soon (Xeon Phi, including Knights Landing, Xeon Skylake Server)
- Only effective for an array of data processed by same instruction

How to vectorize?

- Compiler at high optimization levels will look for opportunities (in loop optimization passes and for straight-line code)
- My experience: Most interesting loops typically not vectorized
  - If there is the slightest possibility of data overlap (aliasing), mis-alignment or re-ordering of floating point operations, the compiler will **not vectorize**
  - Happens often in C++ (pointers can cause aliasing)
- Can also write explicitly vectorized code (intrinsics) $\rightarrow$ need to re-design algorithms
- Try to use vectorized libraries (BLAS + LAPACK, etc.)

Should we write code to fit computer architecture?

- **Instruction-level parallelism**
  - Usually not worth looking for re-ordering of operations
  - Integer operations: Compiler and processor do better job than you in 99% of cases
  - Floating point operations: Compiler and processor do better job than you in 90% of cases
  - Exception: Floating point operations cannot be freely re-ordered by compiler (FP addition is not associative) – might gain in special cases
- **Vectorization** (SIMD, data-level parallelism)
  - Hope for compiler or programming framework to provide it
- **Branches**
  - Avoid comparisons at innermost loop level (inside loop over Gauss points and indices of element matrix) if they can be pulled out
  - Processors' branch predictors are really good!

If there is so little to gain from these factors, what is important?

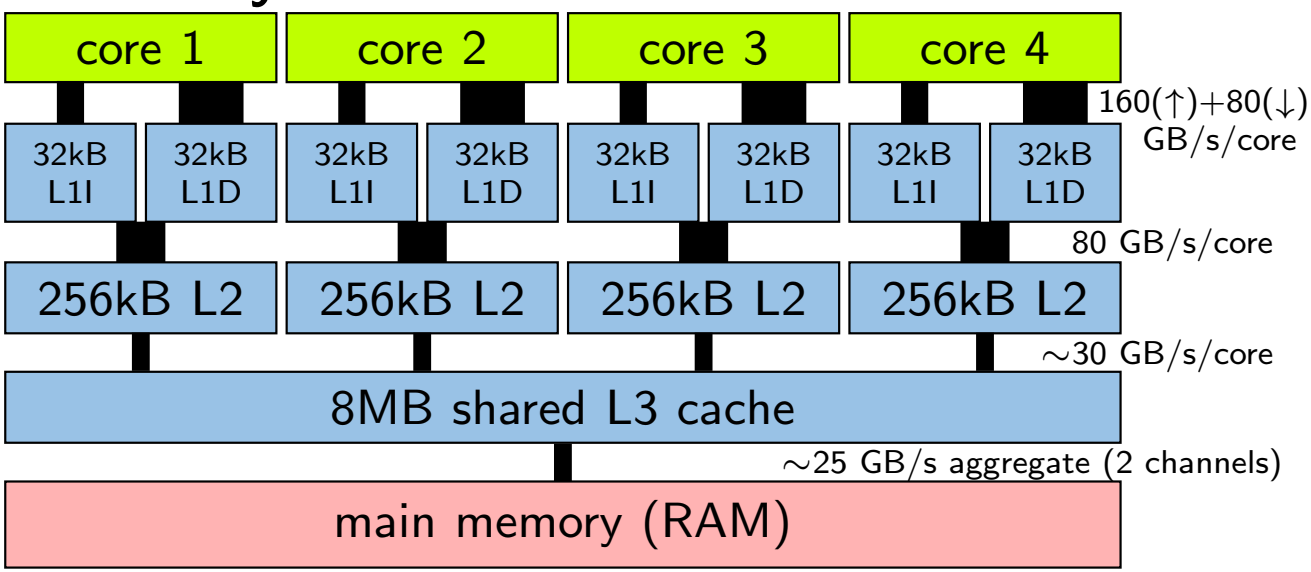
Factors with big effect on performance

- **Redundant code** on a larger level (e.g. re-building expensive data structures such as sparse matrices rather than only filling new data values)
- **Memory allocations**
- **Look-ups through strings** in maps (bad) or **parameter lists** (really bad) in innermost loops
- **Access to main memory** (bad cache usage – no data locality)
- **Access to main memory** (streaming operations, sparse matrix-vector products)
- Access to file system (I/O)
- **Missing parallelism**
- **Synchronization & data exchange** between different processors

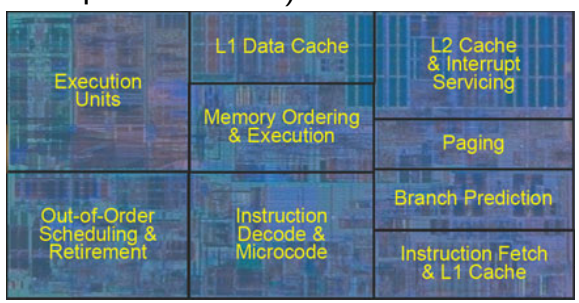
The cost of memory access – memory hierarchy

- Computations are cheap in terms of energy compared to data transport ($\rightarrow$ **memory wall**)
- 1 processor core (at around 10 W) can do **40 billion floating point operations per second** (GFLOPs) in double precision
  ($= 2.5[\text{GHz}] \times 4[\text{AVX}] \times 2[\text{execution units}] \times 2[\text{add+mult}]$)
- Assume one operand per operation comes from main memory
  - One core would need 320 GB/s memory bandwidth
  - Would cost around 100–150 W of power (per core!)

Actual layout of caches on Haswell:



CPU layout (1 core of Intel Nehalem from 2009, image source: Intel press material)



Goals in HPC

- Processors favor algorithms that re-use data (data locality)
- Arithmetic throughput easier to increase than (main) memory bandwidth $\rightarrow$ data locality becomes increasingly important
- Do "more" operations on data that has been loaded from memory

Simple example: Loop fusion

```
for (int i=0; i<1000000; i++)
  a[i] = b[i] + c[i];
for (int i=0; i<1000000; i++)
  d[i] = a[i] * c[i];
```

`a[i]` and `c[i]` loaded twice from RAM

```
for (int i=0; i<1000000; i++)
{
  a[i] = b[i] + c[i];
  d[i] = a[i] * c[i];
}
```

`a[i]` and `c[i]` loaded only once from RAM, second operation re-uses them

Goals in HPC

- Processors favor algorithms that re-use data (data locality)
- Arithmetic throughput easier to increase than (main) memory bandwidth $\rightarrow$ data locality becomes increasingly important
- Do "more" operations on data that has been loaded from memory
- High arithmetic intensity rather than memory intensity

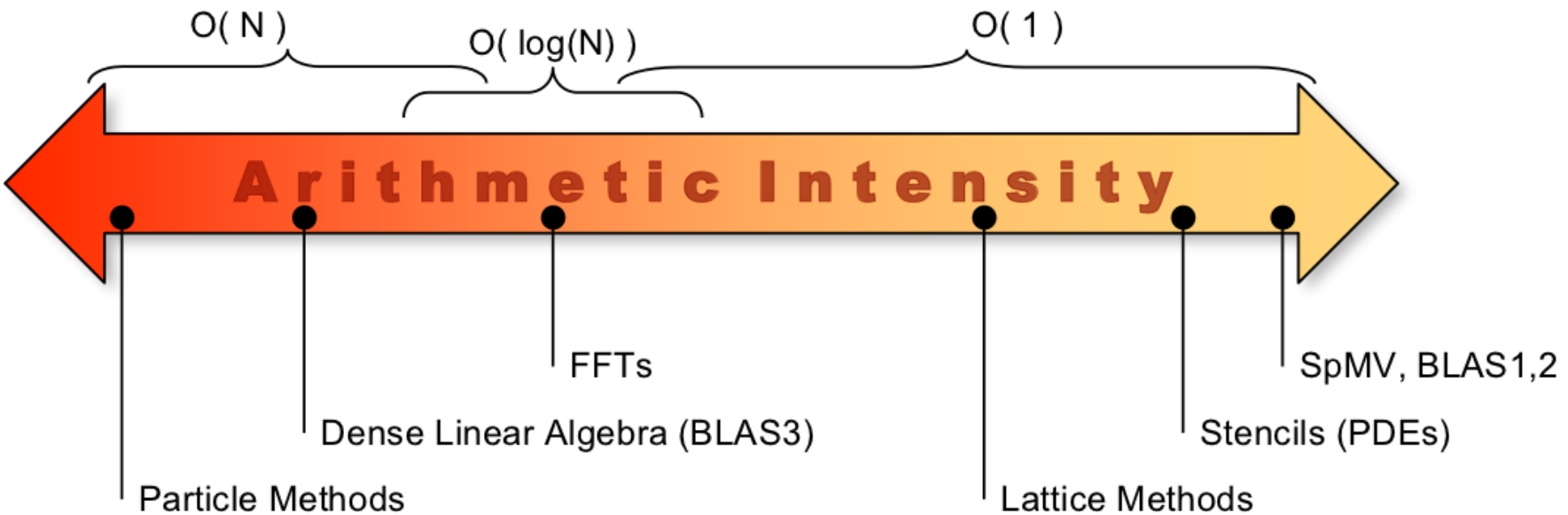


Image source: L. Oliker, Lawrence Berkeley National Laboratories

Operation costs on bruteforce (Haswell, 2.5 GHz)

`double` variables

Throughput (TP): Cost if enough independent operations
Latency: Cost if all operations dependent

Operation	TP vectorized	TP scalar	Latency
$a = a + b * c$ (FMA)	0.05 ns	0.2 ns	2 ns
$a \pm b$ (add, subtract)	0.1 ns	0.4 ns	1.2 ns
$a * b$ (multiply)	0.05 ns	0.2 ns	2 ns
a/b (divide)	1.6–2.8 ns	3.2–5.6 ns	4–8 ns
$\sqrt{a}$	1.6–2.8 ns	3.2–5.6 ns	6.4 ns
exp, log, sin, cos	(needs library)	15–35 ns	15–35 ns
`pow(double, int)`	(needs library)	1–10 ns	2–10 ns
`pow(double, double)`	(needs library)	50–100 ns	50–100 ns
load from L1 cache	0.05 ns	0.2 ns	1.6 ns
store to L1 cache	0.1 ns	0.4 ns	1.6 ns
load/store L2 cache	0.1–0.2 ns	0.4 ns	4.4 ns
load/store L3 cache	0.25–0.5 ns	0.4–0.5 ns	14–16 ns
load/store RAM	0.7–1.9 ns	1.0–1.9 ns	70–80 ns
branch mispredict	—	—	5.6–6.8 ns

Source: Intel 64 and IA-32 Architectures Optimization Manual, own measurements
Note: memory throughput from L2, L3 and RAM is slower if no full cache line is used (random access: 8 times as long)

Operation costs on Haswell (2.5 GHz), cont'd

Memory allocation, serial (e.g. `std::vector<double> v(17)`, `Epetra_SerialDenseMatrix (3,3)`, `new double[42]` etc.), `new + free`: **50 ns** (independent of size) + initialization (e.g. zeroing)

Memory allocation, fully loaded node : **50–100 ns** + X (all processes must obtain memory address from the global RAM pool)

Reduce number of memory allocations!

- Frequent memory allocations (many small chunks that fill more than ∼10% of physical memory) leads to **memory fragmentation** → some operations are sporadically 10–50 times slower (e.g. creating temporary data structures in evaluation of mapping during matrix assembly); unrelated operations (solver) can take suddenly longer
- Try to reduce memory allocations:
  - Re-use temporary data structures (local matrices) over several element calls
  - Use fixed-size/stack allocations for small data structures
- Try to reduce number of memory movements (do not copy big objects, `std::vector::insert()` into the middle of vector)

Practical example

Horner scheme for evaluation of polynomial $p(x) = \sum_{j=0}^{n} c_i x^i$ by

$$c_0 + x\left(c_1 + x\left(c_2 + \ldots + x\left(c_{n-1} + xc_n\right)\right)\right)$$

```
double
horner (const std::vector<double> &c,
        const double               x)
{
  const unsigned int n = c.size()-1;
  double px = c[n];
  for (int j=n-1; j>=0; --j)
    px = c[j] + x * px;
  return px;
}
```

Function run many times for reliable timings

For small $n < 4000$, data fits in L1 cache

```
0x13a0: mov    %edx,%eax
0x13a2: mov    %edx,%ecx
0x13a4: shr    $0x7,%eax
0x13a7: and    $0x7f,%ecx
0x13aa: vmovsd 0x0(%r13,%rax,8),%xmm1
0x13b1: test   %ebp,%ebp
0x13b3: js     0x1561     // jump out if done
0x13b9: movslq %ebp,%rax
0x13bc: vmovapd %xmm2,%xmm0
// ==== inner loop start ====
0x13c0: vfmadd213sd (%rbx,%rax,8),%xmm1,%xmm0
0x13c6: sub    $0x1,%rax
0x13ca: test   %eax,%eax
0x13cc: jns    0x13c0
// ==== inner loop end ====
0x13ce: mov    %ecx,%eax
0x13d0: lea    (%r12,%rax,8),%rax
0x13d4: vaddsd (%rax),%xmm0,%xmm0
0x13d8: vmovsd %xmm0,(%rax)
0x13dc: lea    0x1(%rdx),%eax
0x13df: mov    %rax,%rdx
0x13e2: cmp    %rax,%r14
0x13e5: ja     0x13a0     // loop around horner
```

Practical example

Horner scheme for evaluation of polynomial $p(x) = \sum_{j=0}^{n} c_i x^i$ by

$$c_0 + x\left(c_1 + x\left(c_2 + \ldots + x\left(c_{n-1} + xc_n\right)\right)\right)$$

```
double
horner (const std::vector<double> &c,
        const double               x)
{
  const unsigned int n = c.size()-1;
  double px = c[n];
  for (int j=n-1; j>=0; --j)
    px = c[j] + x * px;
  return px;
}
```

Function run many times for reliable timings

For small $n < 4000$, data fits in L1 cache

What performance to expect?

- ⋆ 40 GFLOPs? (vectorized)
- ⋆ 10 GFLOPs? (not vectorized)

```
0x13b1: test   %ebp,%ebp
0x13b3: js     0x1561     // jump out if done
0x13b9: movslq %ebp,%rax
0x13bc: vmovapd %xmm2,%xmm0
// ==== inner loop start ====
0x13c0: vfmadd213sd (%rbx,%rax,8),%xmm1,%xmm0
0x13c6: sub    $0x1,%rax
0x13ca: test   %eax,%eax
0x13cc: jns    0x13c0
// ==== inner loop end ====
0x13ce: mov    %ecx,%eax
0x13d0: lea    (%r12,%rax,8),%rax
0x13d4: vaddsd (%rax),%xmm0,%xmm0
0x13d8: vmovsd %xmm0,(%rax)
0x13dc: lea    0x1(%rdx),%eax
0x13df: mov    %rax,%rdx
0x13e2: cmp    %rax,%r14
```

GFLOPs $= 10^9$ floating point ops / second

Practical example

Horner scheme for evaluation of polynomial $p(x) = \sum_{j=0}^{n} c_i x^i$ by

$$c_0 + x\left(c_1 + x\left(c_2 + \ldots + x\left(c_{n-1} + xc_n\right)\right)\right)$$

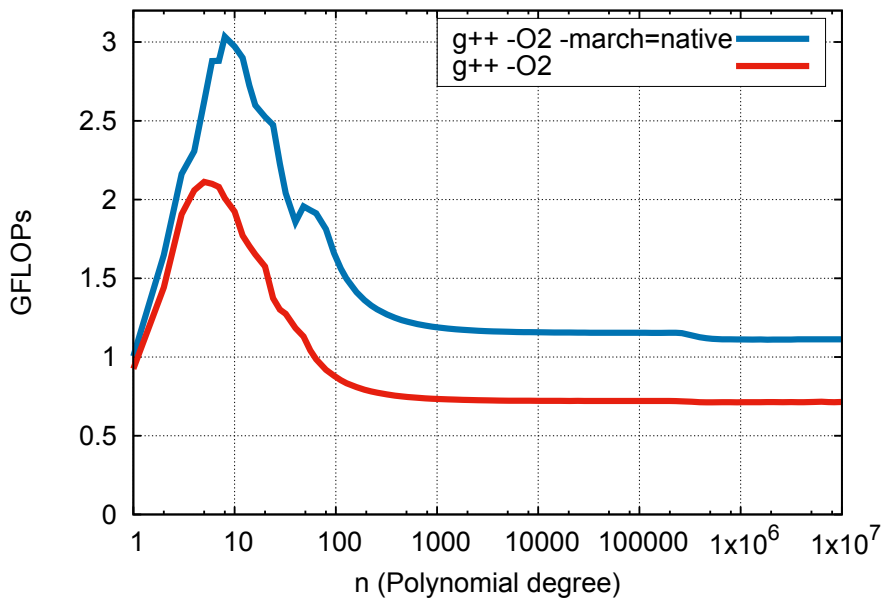
```
double
horner (const std::vector<double> &c,
        const double                x)
{
  const unsigned int n = c.size()-1;
  double px = c[n];
  for (int j=n-1; j>=0; --j)
    px = c[j] + x * px;
  return px;
}
```

Function run many times for reliable timings

For small $n < 4000$, data fits in L1 cache

What performance to expect?

- ⋆ 40 GFLOPs? (vectorized)
- ⋆ 10 GFLOPs? (not vectorized)



GFLOPs = 10^9 floating point ops / second

Practical example

Horner scheme for evaluation of polynomial $p(x) = \sum_{j=0}^{n} c_i x^i$ by

$$c_0 + x\left(c_1 + x\left(c_2 + \ldots + x\left(c_{n-1} + xc_n\right)\right)\right)$$

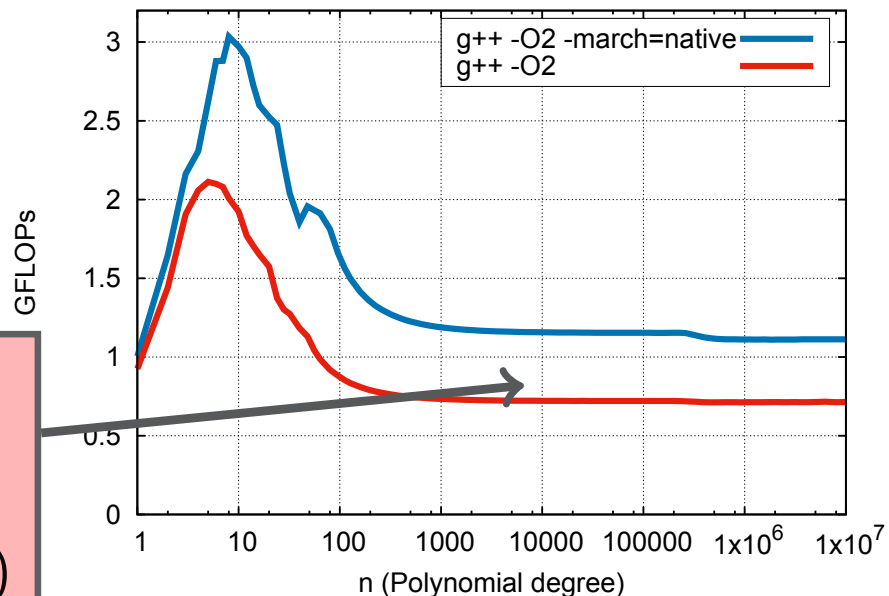
```
double
horner (const std::vector<double> &c,
        const double               x)
{
  const unsigned int n = c.size()-1;
  double px = c[n];
  for (int j=n-1; j>=0; --j)
    px = c[j] + x * px;
  return px;
}
```

Code is latency bound
theory at 2.9 GHz (turbo):
$2.9 * 2 / 5 = 1.12$ GF (FMA),
$2.9 * 2 / (5 + 3) = 0.72$ (mul+add)

Fu… ab… Fo… cache

What performance to expect?

- ⋆ 40 GFLOPs? (vectorized)
- ⋆ 10 GFLOPs? (not vectorized)



GFLOPs = 10^9 floating point ops / second

Practical example

Horner scheme for evaluation of polynomial $p(x) = \sum_{j=0}^{n} c_i x^i$ by

$$c_0 + x\left(c_1 + x\left(c_2 + \ldots + x\left(c_{n-1} + xc_n\right)\right)\right)$$

```
double
ho
{
  const unsigned int n = c.size() - 1;
  double px = c[n];
  for (int j=n-1; j>=0; --j)
    px = c[j] + x * px;
  return px;
}
```

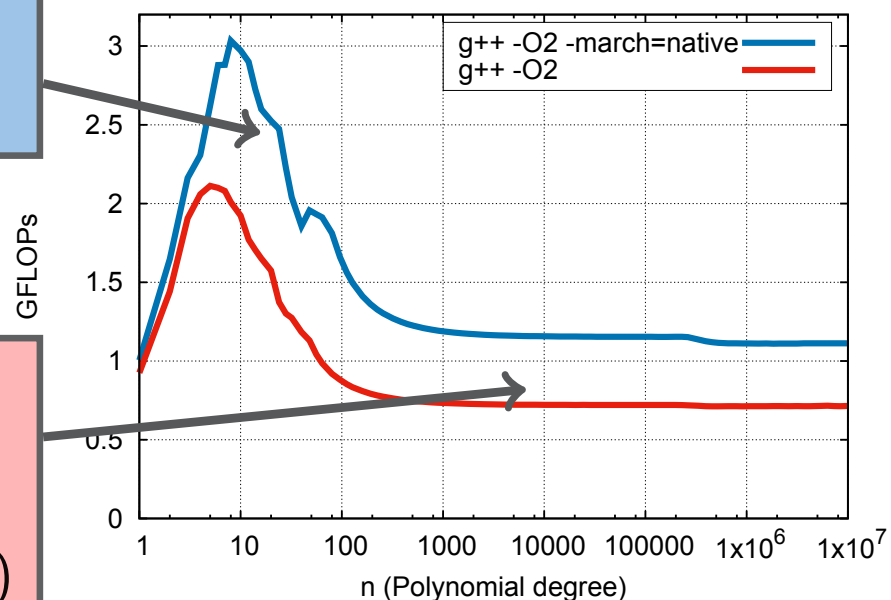
Processor executes **out of order** → overlaps 2–3 polynomial evaluations
Out of order window size: 192 instructions → reasonable overlap up to degree ~ 40

Fu
ab
Fo
cache

Code is latency bound
theory at 2.9 GHz (turbo):
$2.9 * 2 / 5 = 1.12$ GF (FMA),
$2.9 * 2 / (5 + 3) = 0.72$ (mul+add)

What performance to expect?

- ⋆ 40 GFLOPs? (vectorized)
- ⋆ 10 GFLOPs? (not vectorized)



GFLOPs = 10^9 floating point ops / second

Practical example

Horner scheme for evaluation of polynomial $p(x) = \sum_{j=0}^{n} c_i x^i$ by

$$c_0 + x\,(c_1 + x\,(c_2 + \ldots + x\,(c_{n-1} + x c_n)))$$

```
do
ho

{


  for (int j=n-1; j>=0; --j)
    px = c[j] + x * px;
  return px;
}
```

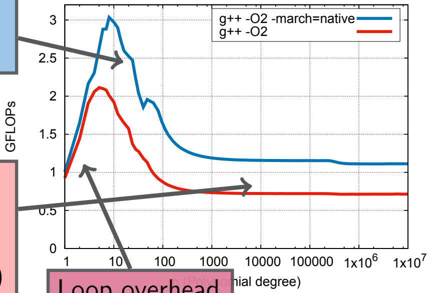
Processor executes **out of order** → overlaps 2–3 polynomial evaluations
Out of order window size: 192 instructions → reasonable overlap up to degree ∼ 40

Code is latency bound
theory at 2.9 GHz (turbo):
2.9 * 2 / 5 = 1.12 GF (FMA),
2.9 * 2 / (5 + 3) = 0.72 (mul+add)

Fu ab Fo cache

What performance to expect?

- ⋆ 40 GFLOPs? (vectorized)
- ⋆ 10 GFLOPs? (not vectorized)



Loop overhead

GFLOPs = 10^9 floating point ops / second

Steps in code optimization I

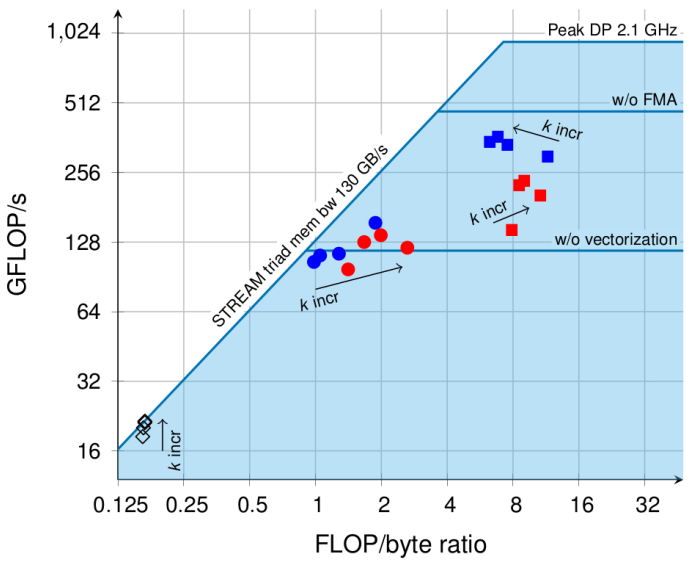
- Identify whether the code is bound by main memory or computations
- How to know the memory access of the code?
  - Measure the relevant algorithm with either a tool like `likwid`[1], analysis of type `CACHES` (look at operational intensity or memory transfer versus arithmetic operations)
  - Compute the required access from main memory by looking at the data structures involved
  - Ideally do both
- How to know the arithmetics done by your code?
  - Measure the relevant algorithm with `likwid`, analysis of type `FLOPS` or `MEM_DP` (double precision) or similar
  - Check whether the compiler has vectorized: Look into assembly code, on some CPUs `likwid` can show both (scalar MUOPS vs packed MUOPS)

[1] https://github.com/RRZE-HPC/likwid

Steps in code optimization II

How to know which one of memory transfer and arithmetics is more limiting?

- Use data on memory transfer and FLOP/s and compare with machine limits (roofline model)
- Look at the output of `clocks per instruction` (CPI) printed by `likwid` or other looks like `perf`; if CPI is higher than 0.6–0.8, the CPU is likely memory bound (or you have bad branches), if it is less it is more likely computation bound



Summary

- Cost lists can guide you on which operations to avoid in important loops
- Lists are only guideline—even good programmers do not know where the hotspots of their codes are unless they profile the code
- Prioritize performance in case simple but unoptimized parts make up for more than $\sim$5% of compute time
- Rule of thumb: Programmer hour in academia $\sim$ 35–50 EUR/hour, the same as 5 000–10 000 core hours on a supercomputer (electricity)
- Prefer easy maintainability and cleanliness of code over performance unless really necessary
- **Only parallel programs will see speedup on future hardware**