# HPC CODE OPTIMIZATION WORKSHOP

Igor Vorobtsov

Senior Technical Consulting Engineer

PRACE Workshop 8-10 June 2020

## Agenda

| | |
|---|---|
| • 10:30 | Introduction: Code modernization approach |
| • 11.00 | Basic compiler optimizations |
| 11.30 | Lab exercises |
| • 13.00 | Vectorization – part I |
| 13.30 | Lab exercises |
| • 14.00 | Vectorization – part II |
| 14.30 | Lab exercises |
| • 15.00 | Skylake optimizations |
| 15.30 | Lab exercises |

## Overview

This is a unique hands-on workshop where you are going to learn techniques, methods and solutions for **code modernization**.

- *Where does the performance of my application lay?*
  - hardware features for Skylake, Intel compiler, optmization report
- *What is the maximum speed-up achievable on the architecture I am using?*
  - Advisor profiling, roofline model
- *Is my implementation matching the HPC objectives?*
  - Use optimized software on latest Intel hardware

# What is code modernization?

- *The Code Modernization optimization framework takes a systematic approach to application performance improvement.*
- *This framework takes an application though five optimization stages, each stage iteratively improving the application performance.*

  – *Leverage optimization tools and libraries*
  – *Scalar, serial optimization*
  – *Vectorization*
  – *Thread parallelism*
  – *Scale your application from multicore to many core*

*https://software.intel.com/en-us/articles/what-is-code-modernization*

## MODERN COMPUTER ARCHITECTURE

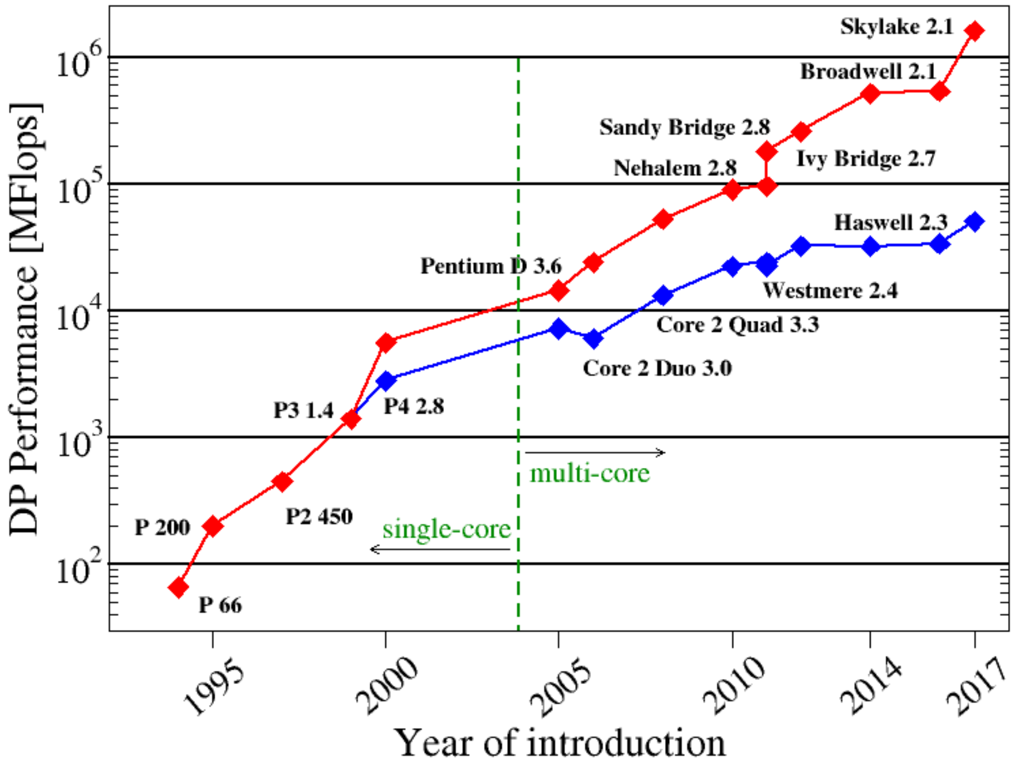
# CPU complexity

The clock frequency cannot increase indefinitely due to the power consumption

Increasing transistor count requires architectural changes which increasing the CPU performance:

- Instruction level parallelism (**ILP**)
  - Pipeline
  - Superscalarity
  - Out of order engine
- Branch prediction and hardware prefetching
- Single instruction multiple data (**SIMD**)
- Simultaneous multi-threading (**SMT**)
- Memory hierarchy (**Caches**)
- Multi-core hardware

## Why SIMD vector parallelism?



The vectorization is crucial for keeping perfomance

- 2000: SSE
- 2004: Multi-core chips
- 2011: AVX
- 2014: AVX+FMA
- 2017: AVX-512

***Skylake:***

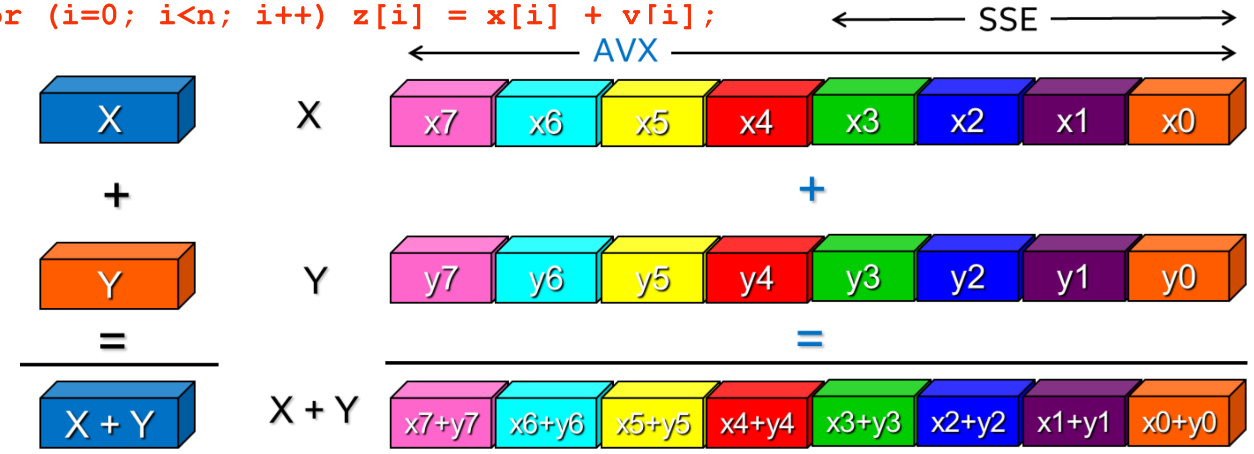
*24 cores, 32 DP FLOPs/cycle: two 8-wide FMA -> **1613GFs***

# SIMD processing

Single instruction multiple data (**SIMD**) allows to execute the same operation on multiple data elements using larger registers.

- Scalar mode
  - one instruction produces one result
  - E.g. vadd**ss**, (vadd**sd**)
- Vector (SIMD) mode
  - one instruction can produce multiple results
  - E.g. vadd**ps**, (vadd**pd**)

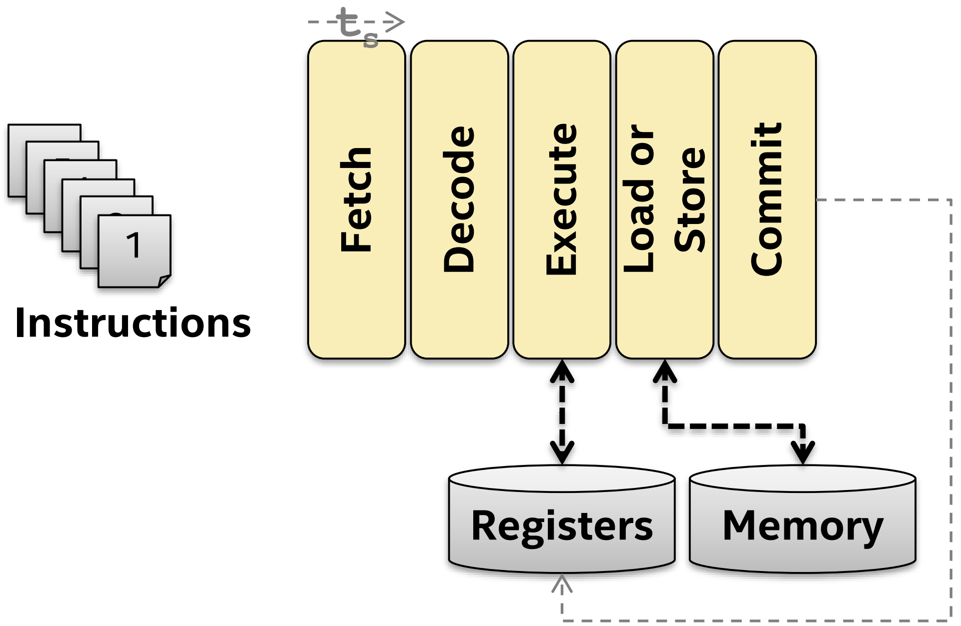
```
for (i=0; i<n; i++) z[i] = x[i] + y[i];
```



- SSE (128 Bits reg.): -> 4 floats
- AVX (256 Bits reg.): -> 8 floats
- AVX512 (512 Bits reg.): -> 16 floats

# Processor Architecture Basics

## Pipeline Execution



**Characteristics of pipeline execution:**

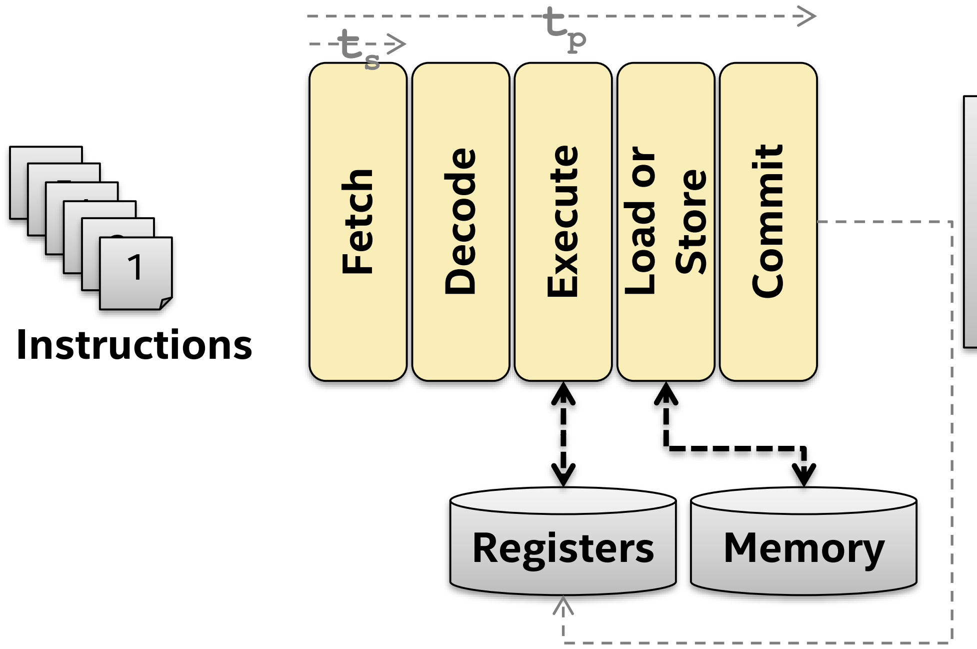
- Multiple instructions for the entire pipeline (one per stage)
- Efficient because all stages kept active at every point in time
- Execution time: $n_{instructions} * t_s$

**Problem:**

- Reality check: What happens if $t_s$ is not constant?

# Processor Architecture Basics

## Pipeline Stalls



```
1: mov $0x1, %rax
2: mov 0x1234, %rbx
3: add %rbx, %rax
4: …
5: …
6: …
```

**Pipeline stalls:**

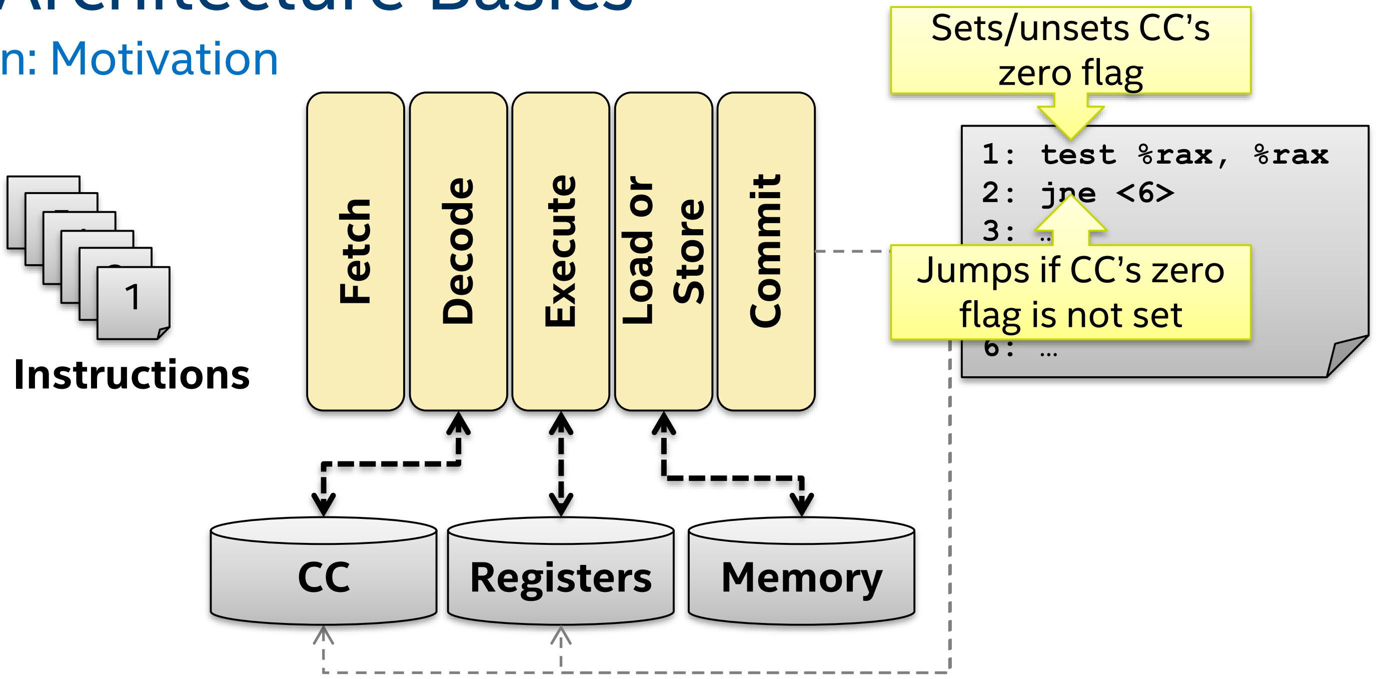
- Caused by pipeline stages to take longer than a cycle
- Caused by dependencies: order has to be maintained
- Execution time: $n_{instructions} * t_{avg}$ with $t_s \leq t_{avg} \leq t_p$

**Problem:**

- Stalls slow down pipeline throughput and put stages idle.

# Processor Architecture Basics

## Branch Prediction: Motivation



```
1:  test %rax, %rax
2:  jne <6>
3:  …
6:  …
```

**Conditional execution causes pipeline stalls:**

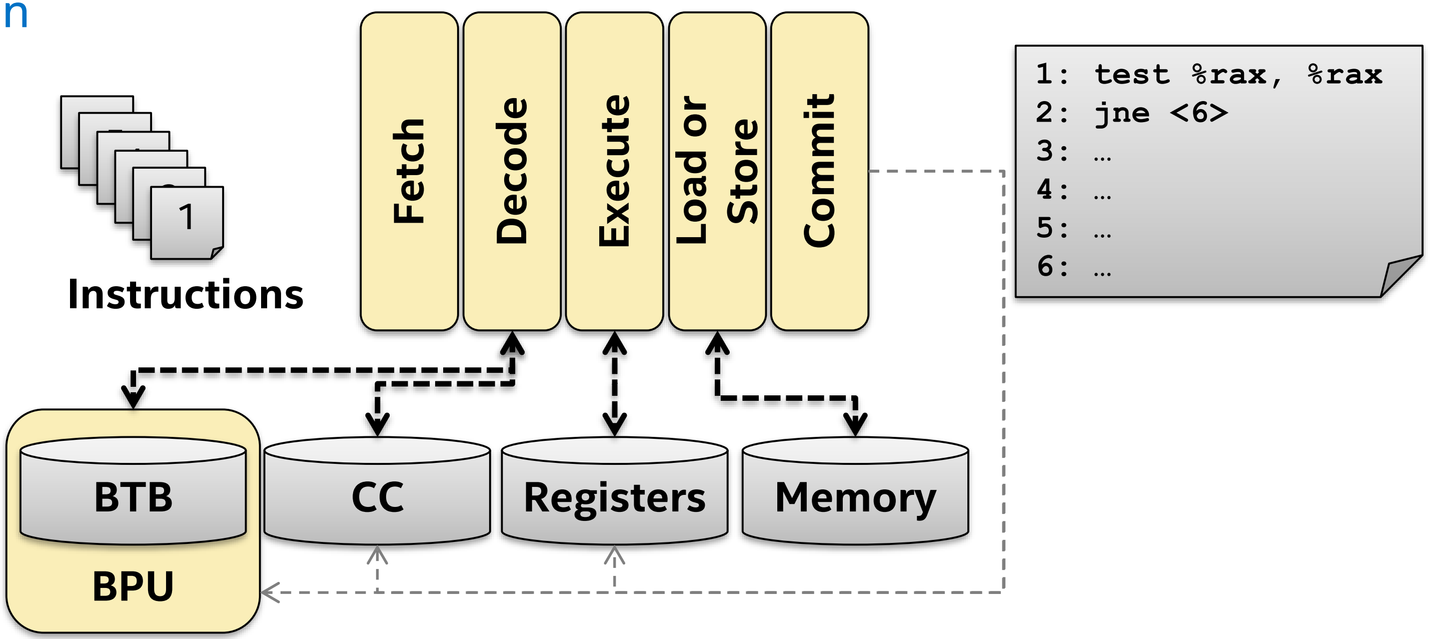
- Conditional branches/jumps require condition codes (CC)
- Those codes (flags) are set by previous instruction once committed
- Leave most of the pipeline idle

**Problem:**

- Large amount of instructions in code can be conditional branches/jumps!

# Processor Architecture Basics

## Branch Prediction



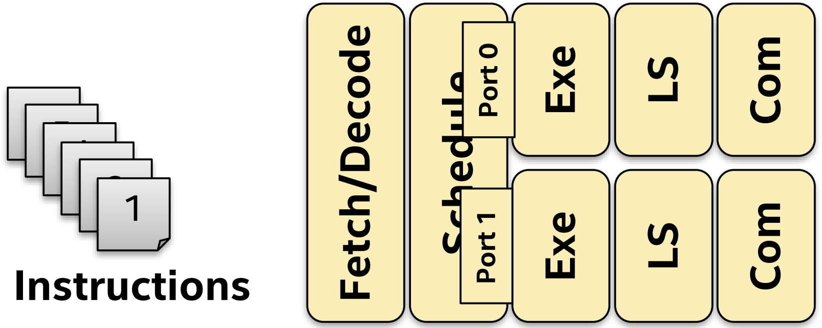
```
1:  test %rax, %rax
2:  jne <6>
3:  …
4:  …
5:  …
6:  …
```

**Branch prediction reduced pipeline stalls:**

- Branch prediction unit (BPU) holds logic which determines likelihood of the branch being taken or not
- Branch target buffer (BTB) remembers branch targets (state of the BPU logic)
- Correct prediction: no latency; incorrect prediction: large latency
- Probability of correct branch prediction can be **more than 90%**!

# Processor Architecture Basics

## Superscalar



**Characteristics of a superscalar architecture:**

- Improves throughput by covering latency (ports are independent)
- Ports can have different functionalities (floating point, integer, addressing, …)
- Requires multiple issue fetch & decode (here: 2 issue)
- Execution time: $n_{instructions} * t_{avg} / n_{ports}$

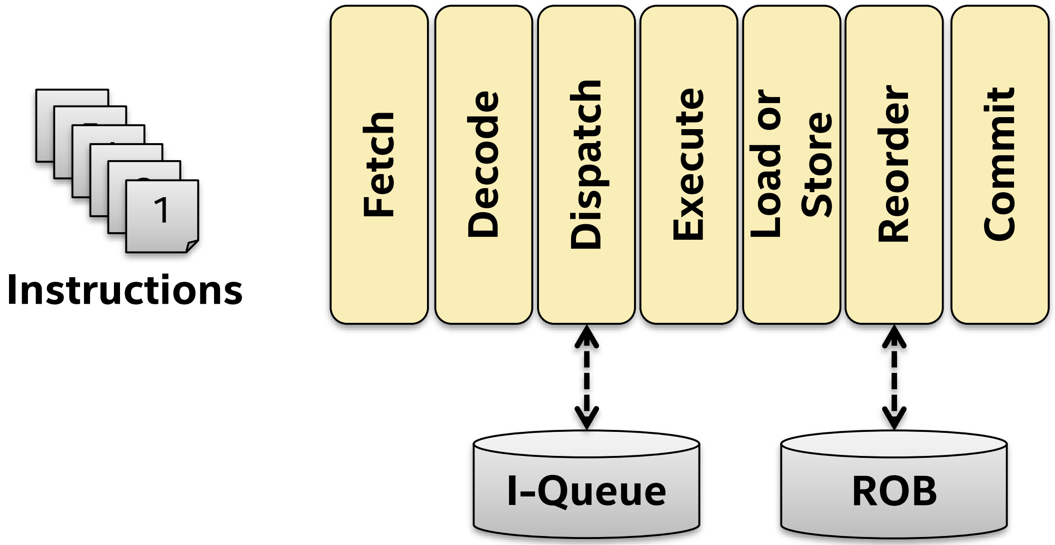
**Problem:**

- More complex and prone in case of dependencies

  ⇨ Solution: Out of Order Execution

# Processor Architecture Basics

## Out of Order Execution



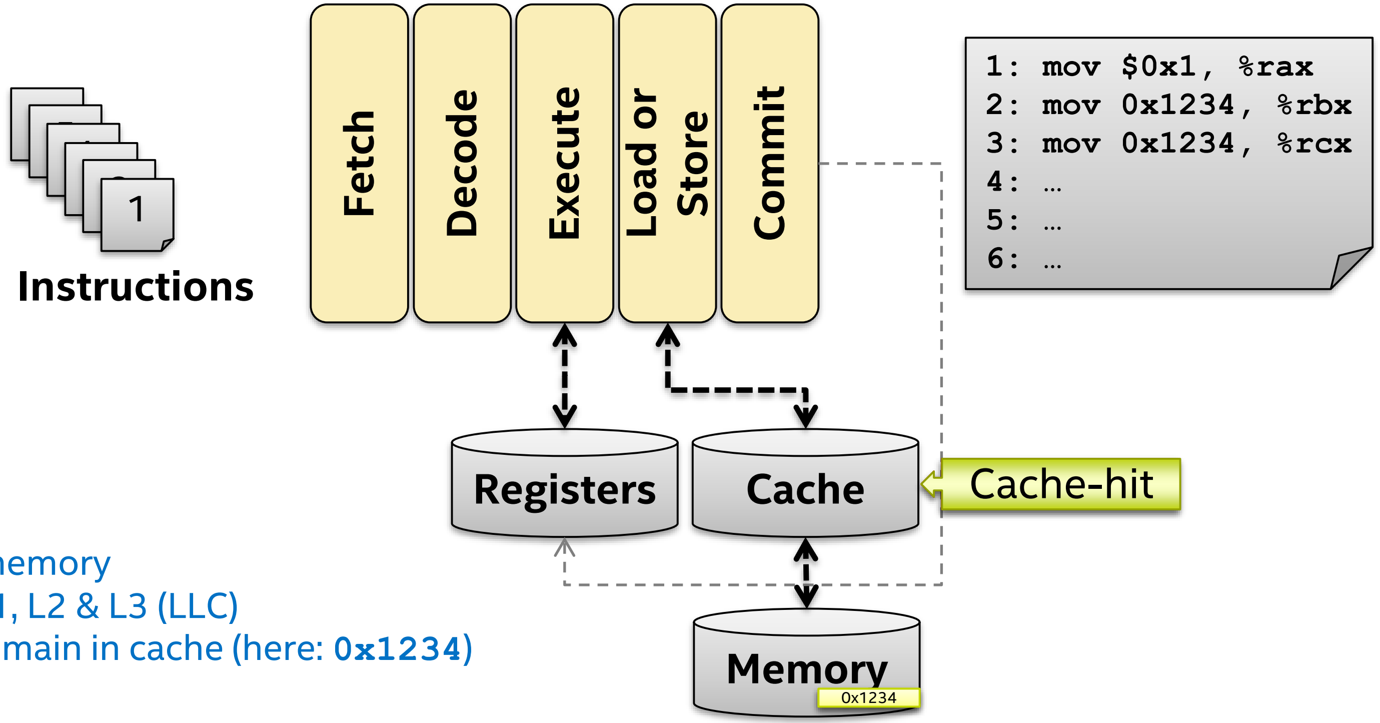
```
1: mov $0x1, %rax
2: mov $0x2, %rbx
3: add %rbx, %rax
4: add $0x0, $r8
5: add $0x1, $r9
6: add $0x2, $r10
```

**Characteristics of out of order (OOO) execution:**

- Instruction queue (I-Queue) moves stalling instructions out of pipeline
- Reorder buffer (ROB) maintains correct order of committing instructions
- Reduces pipeline stalls, but not entirely!
- Speculative execution possible
- Opposite of OOO execution is in order execution

# Processor Architecture Basics

## Cache



```
1: mov $0x1, %rax
2: mov 0x1234, %rbx
3: mov 0x1234, %rcx
4: …
5: …
6: …
```

**Cache:**

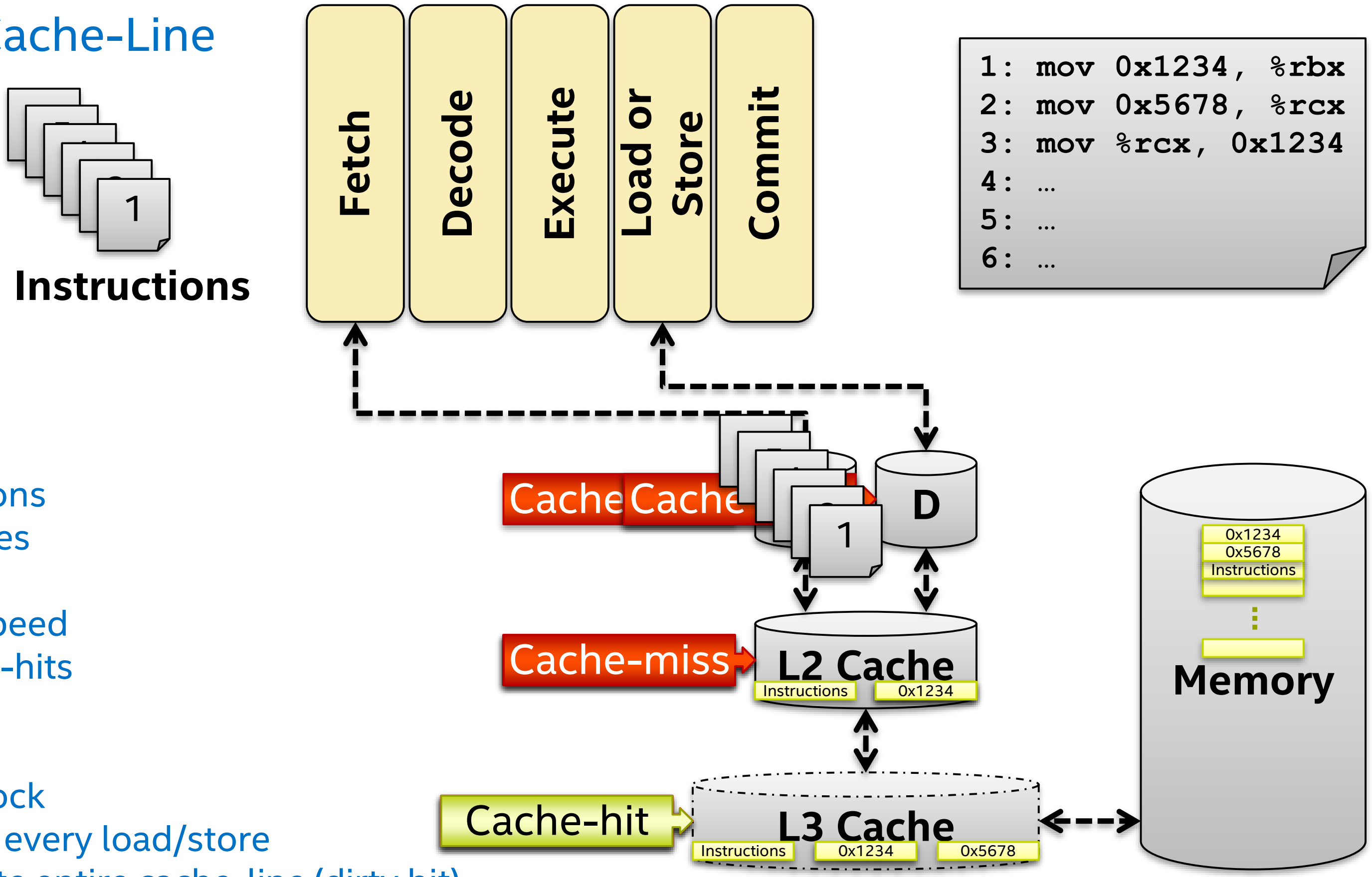
- Small, some KiB
- Faster than main memory
- Cache hierarchy: L1, L2 & L3 (LLC)
- Reused data can remain in cache (here: **`0x1234`**)

**Problem:**

- Maintain data coherency
  ⇨ Cache coherency protocol (e.g. MESI)

# Processor Architecture Basics

## Cache-Hierarchy & Cache-Line



```
1: mov 0x1234, %rbx
2: mov 0x5678, %rcx
3: mov %rcx, 0x1234
4: …
5: …
6: …
```

**Cache-hierarchy:**

- For data and instructions
- Usually inclusive caches
- Races for resources
- Can improve access speed
- Cache-misses & cache-hits

**Cache-line:**

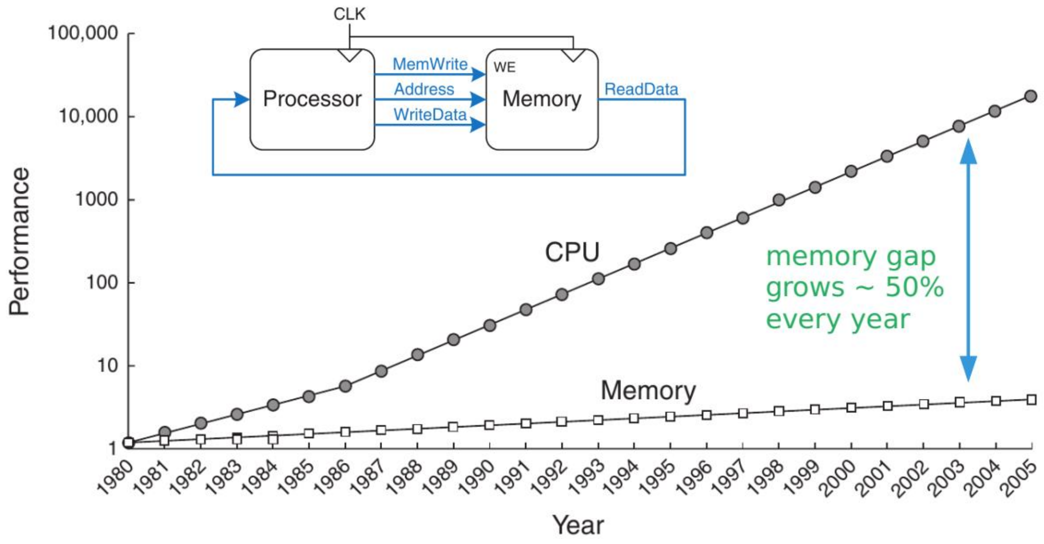
- Always full 64 byte block
- Minimal granularity of every load/store
- Modifications invalidate entire cache-line (dirty bit)

# Critical questions

- What is the real bottleneck?
- What is the theoretical instruction throughput ?
- What is the impact of the memory transfer ?

# Memory system

Performance always depends on which is the slowest: processor or memory



*source*: Hennessy and Patterson. Computer Architecture: A Quantitative Approach (2006)

- CPU performance doubles every 18 months
- Access to RAM doubles every 120 months
- Loading data is very slow

- Intermediate fast memory layers improves performance

## Basic performance considerations

### Let’s calculate some performance numbers:

- Intel® Xeon E5-2630 v4 : **10** cores at **2.2 GHz** in a 2 sockets

  Theoretical peak performance: (2.2 x 10 cores x 16 DP Flops/cycle x 2 sockets) = **704 GF/s**
  704 x 8 bytes = **5.6 TB/s**

  Theoretical memory bandwidth: **68.3 GB/s**

  The peak throughput is: **82 FP / memory access** !!!

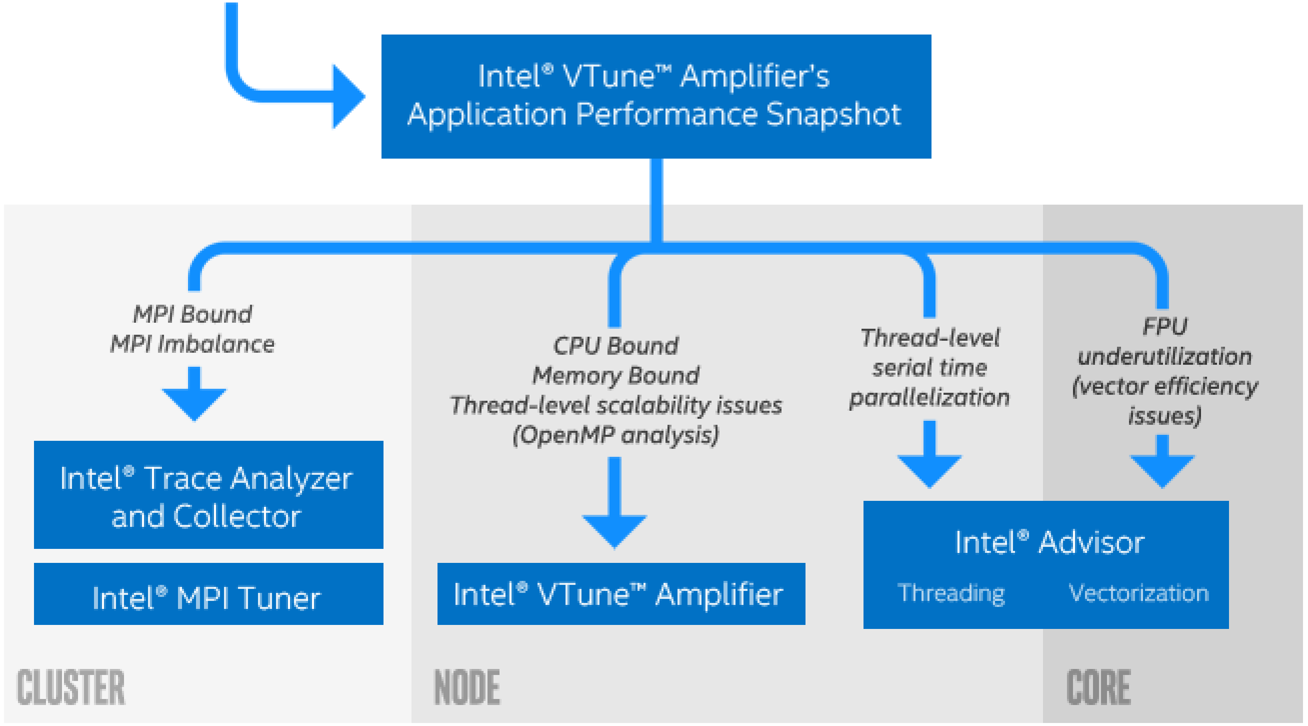
- > 82 FP / mem is a compute bound application
- < 82 FP / mem is a bandwidth bound application

# Tuning for the Intel® Xeon® Scalable processor

## All the new features requires good use of the available resources

- Core
  - Vectorization is critical with 512bit FMA vector units (32 DP ops/cycle)
  - Targeting the current ISA is fundamental to fully exploit vectorization
- Socket
  - Using all cores in a processor requires parallelization (MPI, OMP, … )
  - Up to 28 Physical cores and 56 logical processors per socket!
- Node
  - Minimize remote memory access (extra Intel® UPI hops)
  - Minimize resource sharing (tune local memory access, disk IO and network traffic)

# Tuning Workflow


Intel® VTune™ Amplifier's Application Performance Snapshot
MPI Bound MPI Imbalance
Intel® Trace Analyzer and Collector
Intel® MPI Tuner
CPU Bound Memory Bound Thread-level scalability issues (OpenMP analysis)
Intel® VTune™ Amplifier
Thread-level serial time parallelization
FPU underutilization (vector efficiency issues)
Intel® Advisor
Threading
Vectorization
CLUSTER
NODE
CORE

## Code Modernization

Stage 1: Use Optimized Libraries

Stage 2: Compile with Architecture-specific Optimizations

Stage 3: Analysis and Tuning

Stage 4: Check Correctness

# Resources for Lab exercises

RRZE Meggie cluster - cshpc.rrze.fau.de

Intel DevCloud - http://devcloud.intel.com/

Source files for exercises:

https://github.com/fbaru-dev/nbody-demo.git

https://github.com/fbaru-dev/hpc-workshop.git

https://github.com/ivorobts/compiler-optimization.git

# Legal Disclaimer & Optimization Notice

# Jump on DevCloud

http://devcloud.intel.com/

```
$ ssh devcloud
...login to the devcloud...

$ pbsnodes -l free
...list of free nodes...

$ pbsnodes s001-145
...information about node s001-145...

$ pbsnodes | more
...lots more detail...
$ pbsnodes | grep properties
...useful properties list...
$ pbsnodes | grep fpga
...useful fpga oriented list...
```

## Hello qsub

```
$ mkdir mytst
$ cd mytst

$ cat - > myhello.sh
echo "HELLO, WORLD!"
^D

$ qsub myhello.sh

$ qstat
Job ID Name       ...
------ ----------
3463   myhello.sh ...

$ qstat
```

## Hello qsub

```
$ ls -l
total 8
-rw-r--r-- 1 u27938 u27938  21 Oct 14 22:58 myhello.sh
-rw------- 1 u27938 u27938   0 Oct 14 22:58 myhello.sh.e3463
-rw------- 1 u27938 u27938 603 Oct 14 22:58 myhello.sh.o3463

$ cat myhello.sh.o3463

########################################################################
#      Date:           Mon Oct 14 22:58:58 PDT 2019
#    Job ID:           3463.v-qsvr-nda.aidevcloud
#      User:           u27938
# Resources:           neednodes=1:ppn=2,nodes=1:ppn=2,walltime=06:00:00
########################################################################

HELLO, WORLD!

########################################################################
# End of output for job 3463.v-qsvr-nda.aidevcloud
# Date: Mon Oct 14 22:58:59 PDT 2019
########################################################################
```

# Hello qsub

```
...use a particular node...
$ qsub -lnodes=s001-n155:ppn=2

...use a node based on a property...
$ qsub -lnodes=1:ppn=2:fpga_compile
$ qsub -lnodes=1:ppn=2:gpu
$ qsub -lnodes=1:ppn=2:skl
$ qsub -lnodes=1:ppn=2:cfl
```