HARVARD UNIVERSITY

FAS RC

FAS RESEARCH COMPUTING
DIVISION OF SCIENCES
HTTPS://RC.FAS.HARVARD.EDU

Intro to Code Optimization

FAS Research Computing

https://rc.fas.harvard.edu

Outline

Moore's Law

CPU Architecture 101

Compiler Optimization

General Code Optimization Rules

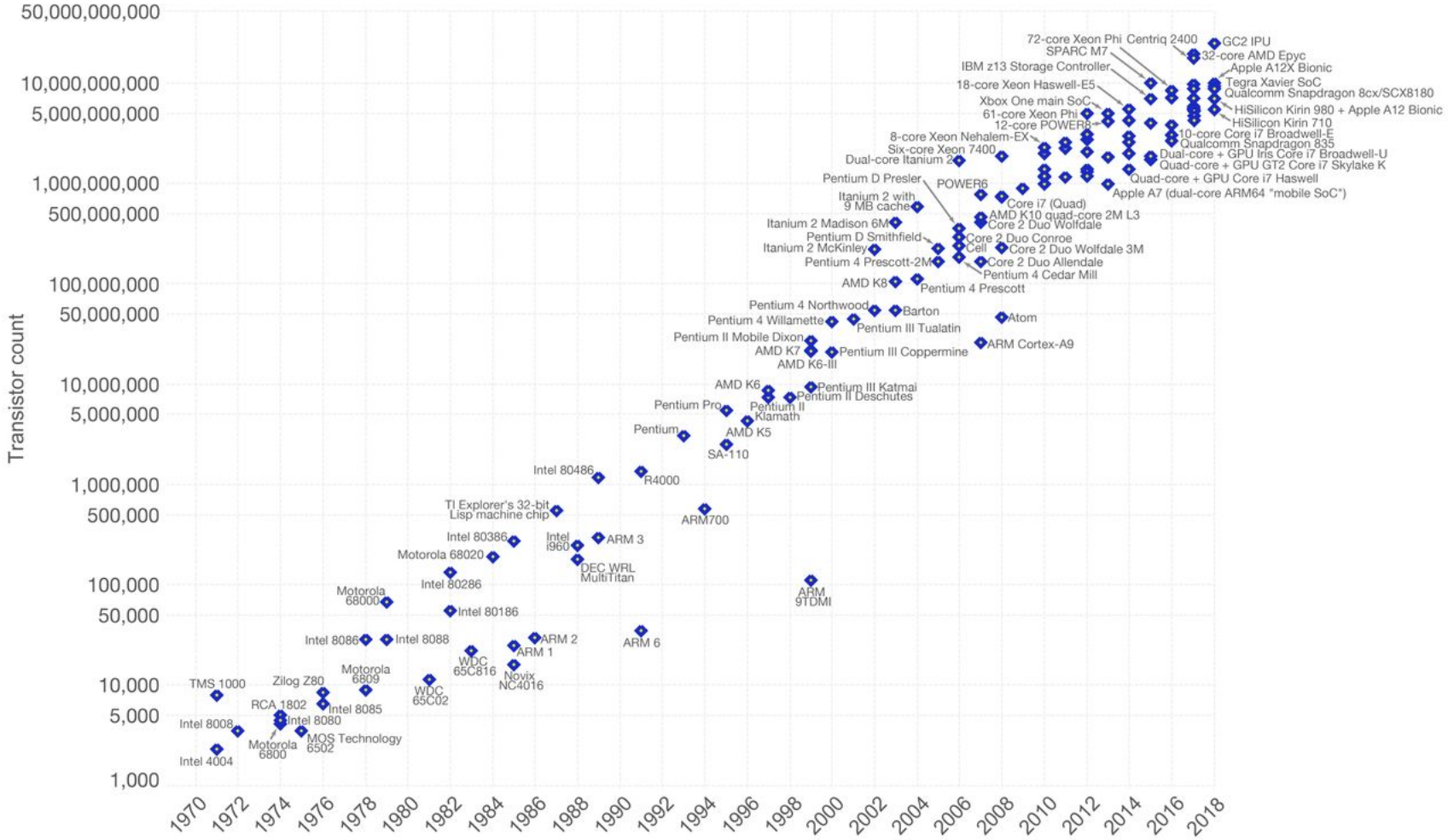
Parallelism

Next Steps

Moore's Law – The number of transistors on integrated circuit chips (1971-2018)

Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are linked to Moore's law.

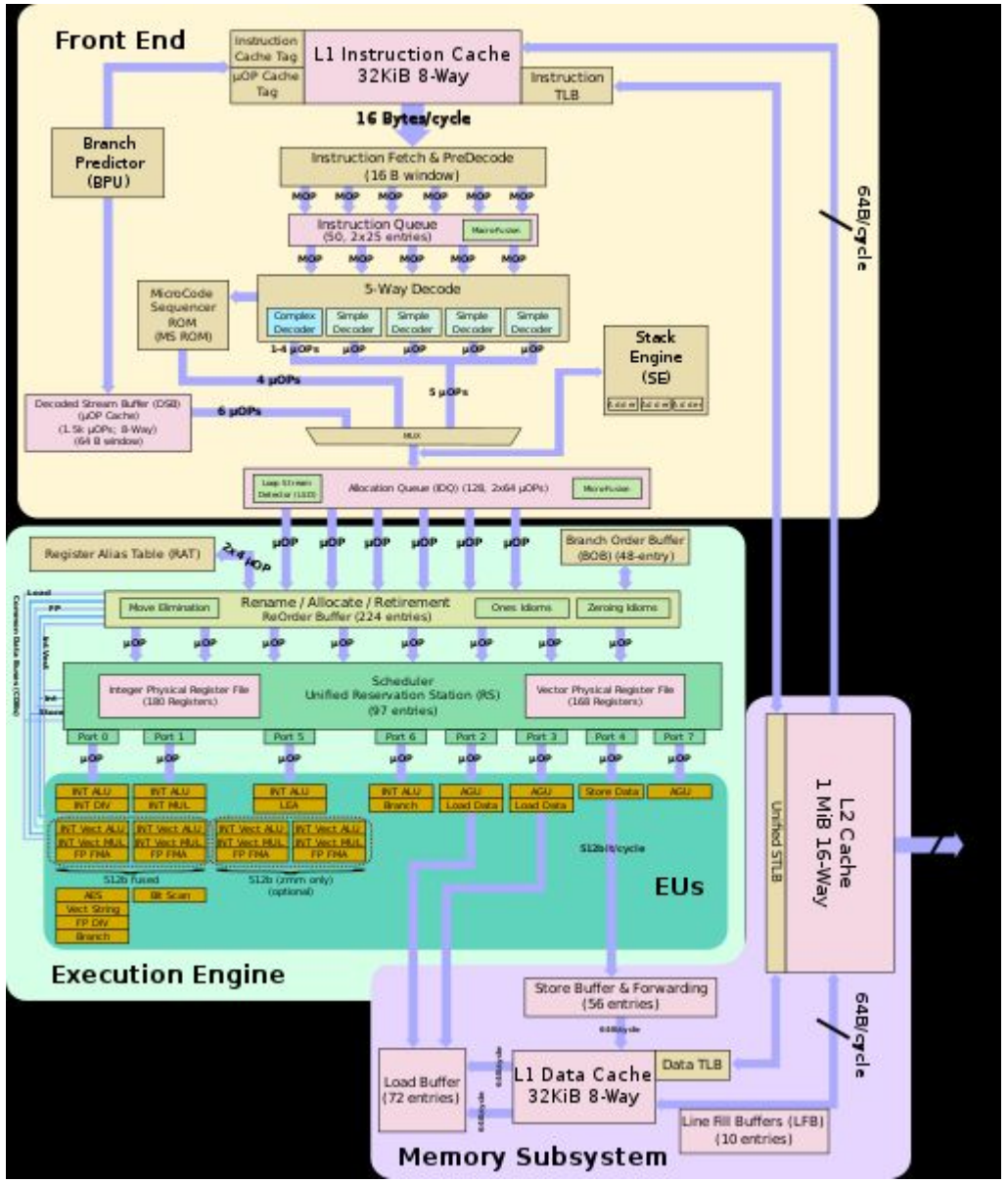


Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

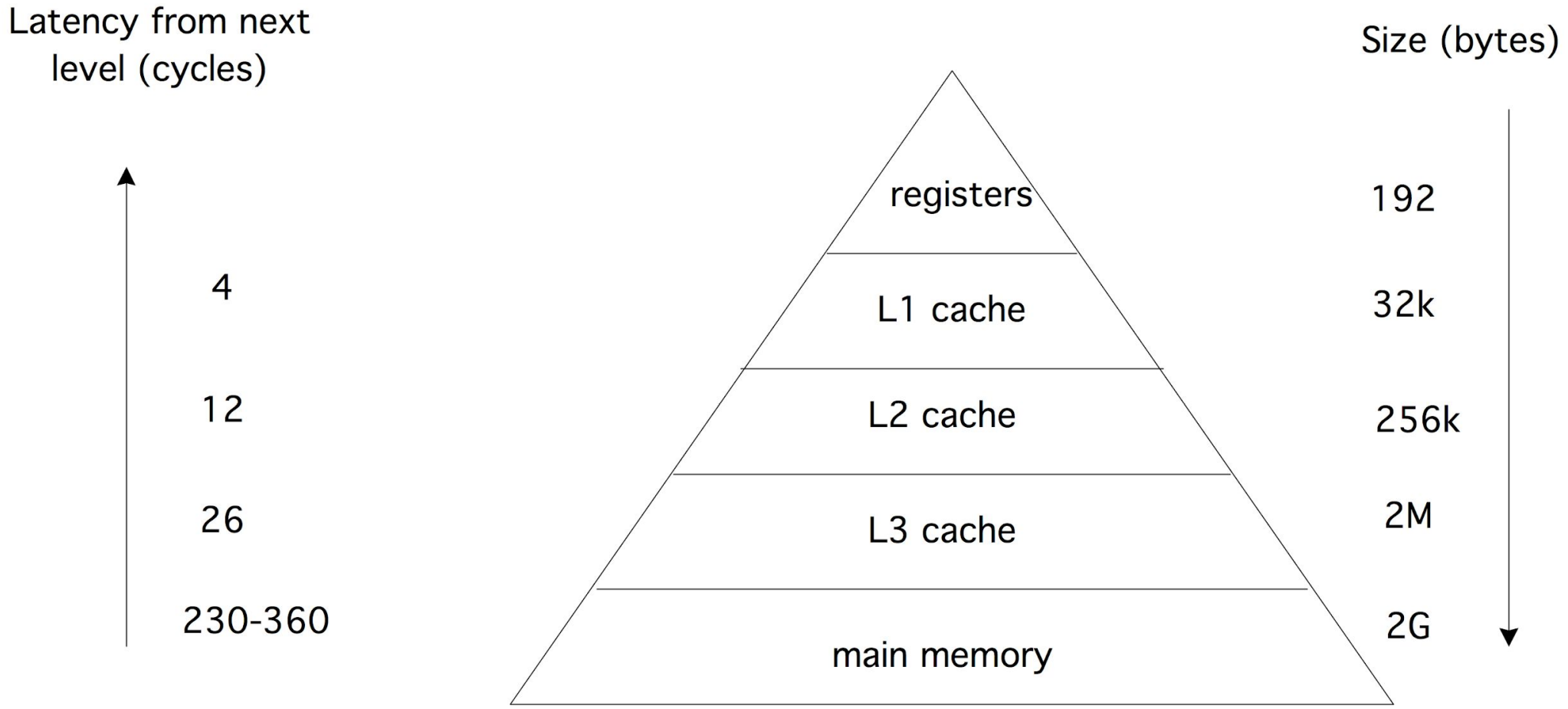
CPU Architecture 101

Core Block Diagram for Intel Cascade Lake (Wikichip)

- µops (micro-ops): basic unit of work, cpu cycle
- Branch Predictor
- Registers
  - Instruction
  - Data
- SIMD: Single Instruction Multiple Data
- ALU: Arithmetic Logic Unit
- AGU: Address Generation Unit
- MUL: Multipiler
- FMA: Fused Multiply Add
- INT: Integer
- FP: Floating Point
- LEA: Load Effective Address
- AES: Advanced Encryption Standard



Memory Hierarchy



Intel Sandy Bridge Memory Hierarchy from Intro to HPC by Victor Eijkhout

Compiler Optimization

- Use the latest compiler
- Remove all debug flags for production runs
  - -g, --traceback, -check-* (Intel), -debug, -W*
- Test code performance at different optimization levels
- Be careful of processor specific optimizations
- Use only optimizations that improve code performance
- Always verify that the result you are getting is still correct when using a new compiler and/or new optimization level
- Read the manual page for the compiler as compilers vary on how they bundle their optimization levels.

-O0

- Disables all optimizations
- This is good for debugging but should never be used
- If you truly want to see how much work the compiler is doing try -O0, but warning your code will run dead slow.
- Default optimization level for gcc

-O1

- Enables optimizations for speed but disables optimizations that impact code size, such as intrinsic inlining
- This optimization level is safe in all cases
- This should be the absolute minimum optimization but should not be the norm. Only use if you absolutely have to for code consistency.

-O2

- Default optimization level for Intel
- Safe in most cases, but can lead to variance in round off error or out of order instructions in edge cases.
- Vectorization (Intel only): -vec, -xCore-AVX512
  - Vectorization report: -qopt-repot=1 -qopt-report-phase=vec
- Inlining of intrinsics
- Intrafile interprocedural optimization
- Loop unrolling
- Variable renaming
- Variable simplification
- Intel Compiler Specific: Codes compiled with O2 start picking up additional Intel chip specific optimizations.

-O3

- O3 does everything in O2 plus more aggressive optimizations including:
  - aggressive loop transformations such as fusion, block-unroll-and-jam
  - collapsing IF statements
  - arithmetic reordering as well as less rigorous but faster methods for math may take place.

- Automatic Vectorization for gcc: -mavx512*, -ftree-vectorize
  - Vectorization report: -fopt-info=vec

- Unlike O2 which generally safe and may not change your results, O3 is considered the kitchen sink approach and thus may change your result. O3 may also slow down your code. Test and verify this option.

- -fast (or -Ofast -march=native) will throw in everything including processor specific optimizations

Additional Flags/Notes

- Each compiler has a wealth of options and differences, so it is worth your time to look at them, especially what options are being applied for O1-3

- -xHost/-ax*/-march: These can be useful for chip specific optimization. However this makes the code non-portable. If you can, grab the individual optimizations you want rather than use -xHost

- Intel vs. gcc vs. something else? Generally if you tune it correctly most compilers give similar performance but the options may vary. That said some compilers, such as Cray's compiler, are superior in most cases but only available on Cray machines. In general the Intel compiler works best out of the box, as gcc tends to include a bunch of options as defaults that drag down code performance.

General Code Optimization Rules

Use the latest compilers and libraries

Leave informative comments in your code

Make sure your loops are ordered appropriately for your arrays

Avoid if statements buried in loops

Use temporary variables to hold math constants (such as pi), especially divisions or exponentials

Set your variables to the right precision/size: Single Precision math is faster than Double Precision math, Integer math can be faster than Floating Point

General Code Optimization Rules

Use timing print statements or profiler to isolate hot spots (i.e. where your code is spending most of its time).

- Intel compiler: -profile-loops, -profile-functions
- Intel VTune
- Tau
- Totalview

If you have a heavy arithmetic section consider using small temporary arrays for the data that you are manipulating

Lower your cache miss rate and try to operate in L2 cache as much as possible

Take advantage of automatic vectorization by writing your code to make it obvious to the compiler. Be sure to check the optimization report and see if it worked as expected. Also make sure your loops contain as much math as possible.

- https://software.intel.com/content/www/us/en/develop/articles/vectorization-essential.html

General Code Optimization Rules

Be aware of the first touch rule for memory allocation and allocate arrays and variables you need frequently first.

Cut down your memory footprint as much as possible by removing extraneous temporary arrays and variables.

Avoid over abstraction, (i.e. pointers to pointers to pointers)

Be specific and well defined, amorphous or poorly structured code is hard to optimize.

Be up on the latest numerical methods and libraries for your discipline. Pick the right one for your code, this may not be the most popular or most cutting edge. Brute force may actually be faster than a more nuanced approach.

When all else fails use Fortran with a good compiler...

Parallelism

Three Types of Parallelism

- SIMD: Single Instruction Multiple Data
- Thread: Shared Memory
  - OpenMP
  - OpenACC
  - Cuda
- Rank: Message Passing Interface

Hybrid: Thread + Rank

Code Improvement vs. Code Overhaul

There are two basic methods for optimizing your code

1. Scrutinize your current code base and make changes to improve speed either by changing compilers or making tweaks to the code.
2. Complete overhaul of your code base.

Each has their advantages and disadvantages

Code Improvement

Advantages

- Low Hanging Fruit
- Fast
- Minimal Changes

Disadvantages

- Cannot deal with structural issues
- Whack-a-mole
- Cascade of changes
- Can lead to spaghetti code that is disjointed and unreadable
- Can introduce unanticipated bugs

Code Overhaul

Advantages

- Can deal with real structural issues
- Allows changes to new numerical methods and techniques
- Allows for adding parallelism
- Allows for a more cohesive whole that takes a holistic view

Disadvantages

- Overhaul work can take 6 months to a year to complete depending on code complexity
- Neverending code development
- Requires a fundamental knowledge of the code and its inner workings

Next Steps

Research

- Man pages
- Intro to HPC: https://pages.tacc.utexas.edu/~eijkhout/istc/istc.html
- Numerical Methods for Scientists and Engineers
- Code Libraries

Research Computing Help

- RC Consulting
- Research Software Engineering Services: https://www.rc.fas.harvard.edu/research-software-engineering-rse/

Questions? Comments?

FAS Research Computing

https://rc.fas.harvard.edu