CS 377P: Programming for Performance

Administration

- Instructor:
  - Keshav Pingali (Professor, CS, ECE & Oden)
    - 4.126 Peter O'Donnell Building (POB)
    - Email: pingali@cs.utexas.edu
- TA:
  - Taeklim Kim (PhD student, CS)
    - Email: tlkim@cs.utexas.edu

Prerequisites

- Basic computer architecture course
  - (e.g.) PC, ALU, cache, memory, instruction-level parallelism (ILP)
- Basic calculus and linear algebra
  - differential equations and matrix operations
- Software maturity
  - assignments will be in C/C++ on Linux computers
  - ability to write medium-sized programs (< 1000 lines)
- Self-motivation
  - willingness to experiment with systems

Coursework

- 6 programming projects
  - These will be more or less evenly spaced through the semester
- One mid-semester exam
  - Date: TBA
- Final exam
  - Monday, May 4 2026, 1:00 pm-3:00 pm

Text-book for course

No official book for course

This book is a useful reference.

"Parallel programming in C with MPI and OpenMP", Michael Quinn, McGraw-Hill Publishers. ISBN 0-07-282256-2

Lots of material on the web

What this course is not about

- This is not a clever hacks course
  - We are interested in general scientific principles for performance programming, not in squeezing out every last cycle for somebody's favorite program
- This is not a tools/libraries course
  - We will use several tools and libraries like MPI but for us, they are a means to an end and not end in themselves.

What this course IS about

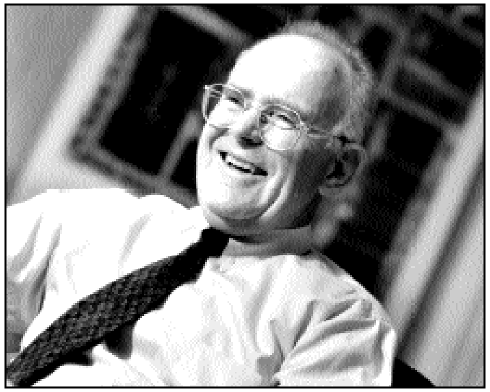
- Architects invent many hardware features for boosting program performance
- Usually, software can benefit from these features only if it is carefully written to exploit them
- Agenda in CS 377P:
  - Understand performance-critical architectural features in modern computers
  - Develop general principles and techniques that can guide us in writing programs to exploit these features
  - Use state-of-the-art tools to put these into practice
- Two major concerns:
  - Exploiting parallelism
  - Exploiting locality

Why worry about performance?

- Until ~2005
  - Most programmers did not worry about performance
    - Programs ran faster on each new generation of computer
    - If you didn’t like the performance, you could wait and buy a new computer
  - Small number of single-processor performance programmers
    - Caches: exploit locality
    - Vectorization
  - Even smaller number of parallel programmers
    - HPC centers: worried about parallelism and locality
- Since then
  - Programs do not run any faster on new hardware unless they exploit parallelism
- What drove this dramatic change?

Moore's Law

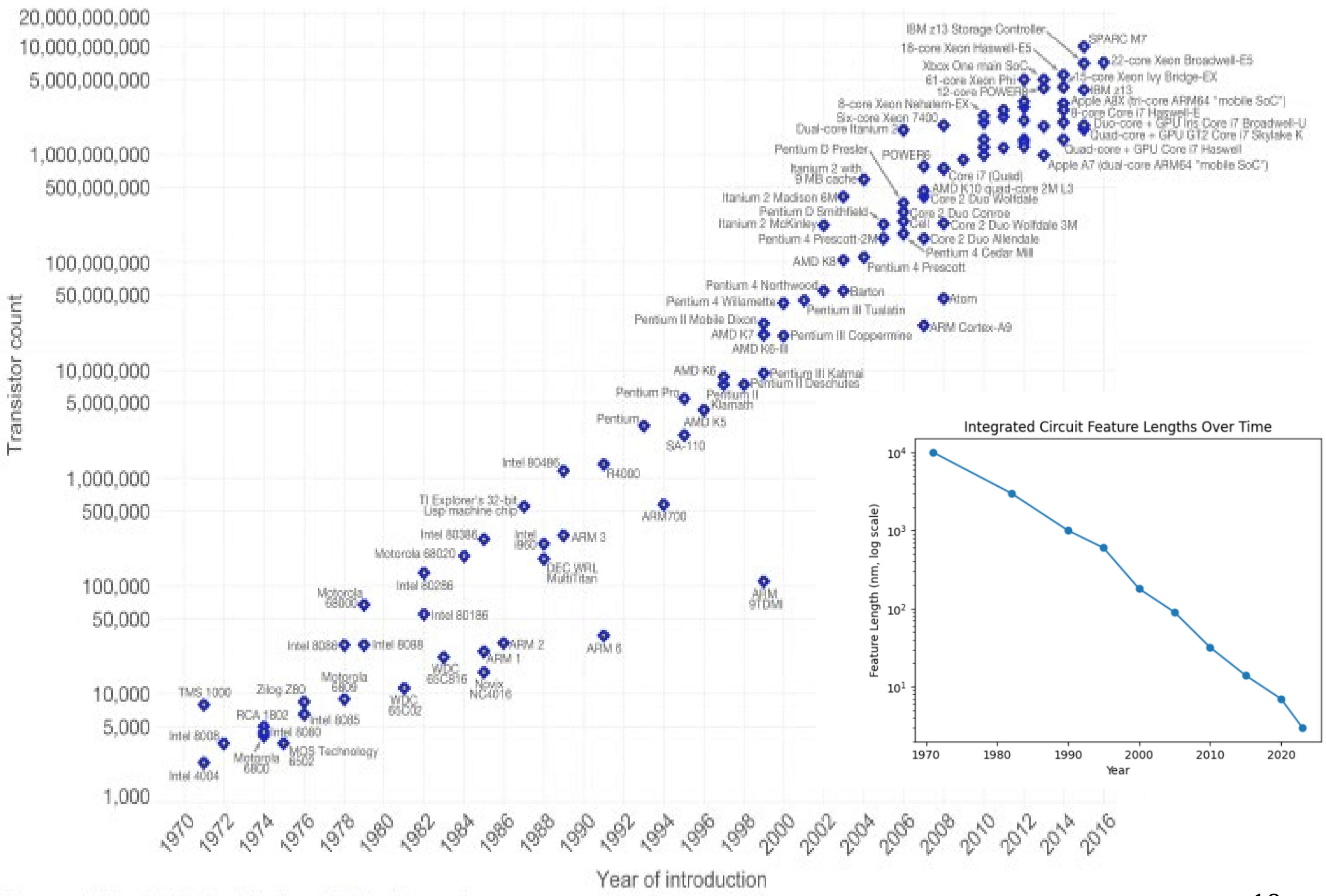
- What Moore said [1965]:
  - Number of transistors on a chip double every new generation of technology (~1.5 years)
  - Empirical observation: how many transistors can be placed on IC wafer economically
- What people think Moore said:
  - Processor frequency doubles every 1.5 years

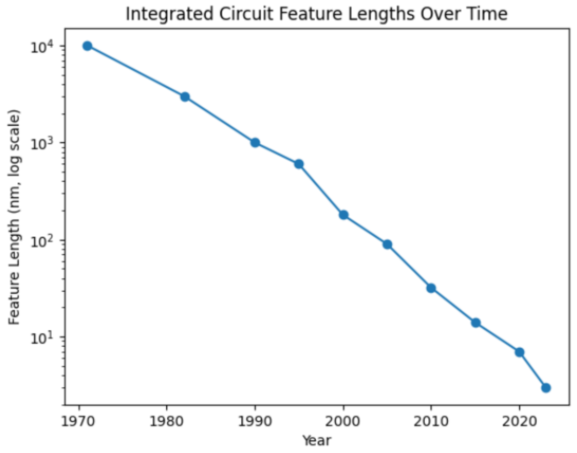

Gordon Moore (Intel)

Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

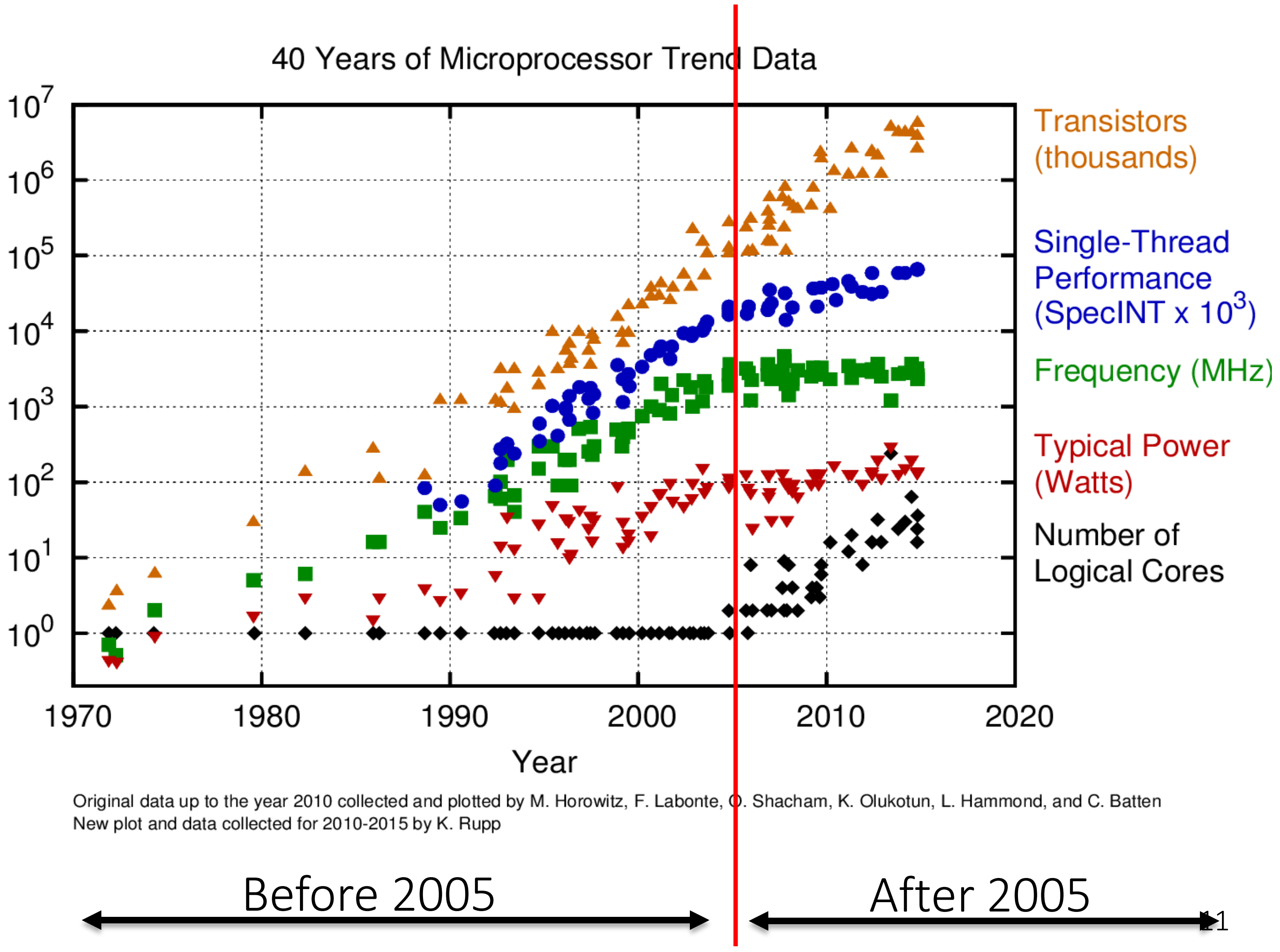
Our World in Data

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 strongly linked to Moore's law.





Microprocessor trend data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

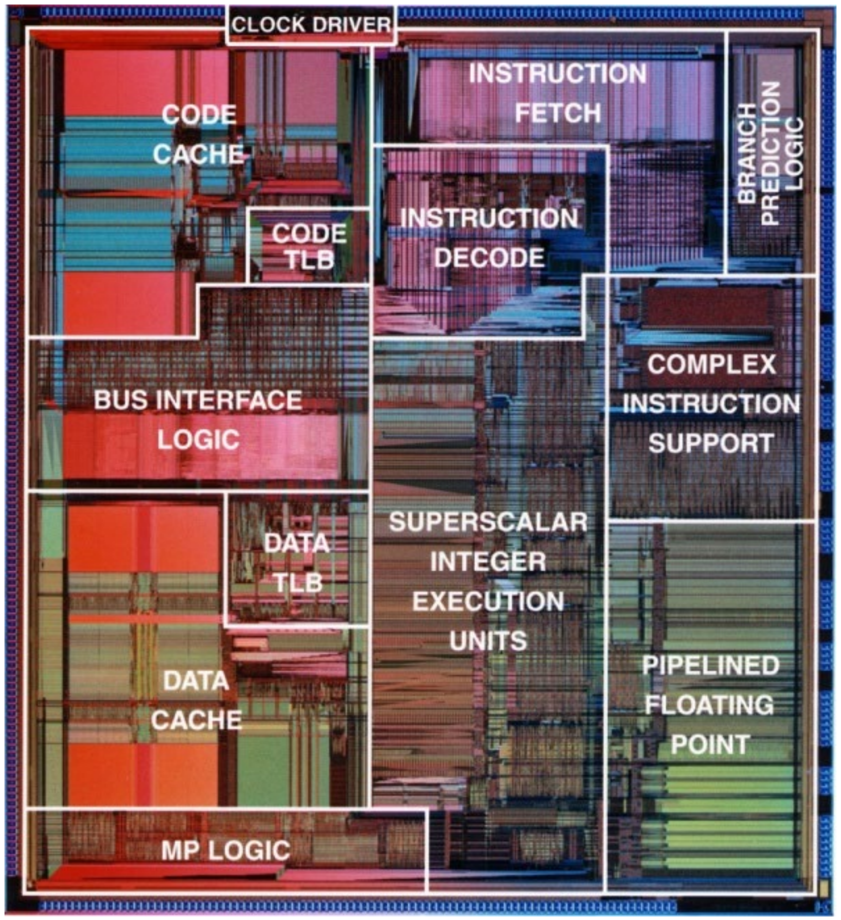
Before 2005

After 2005

BEFORE 2005

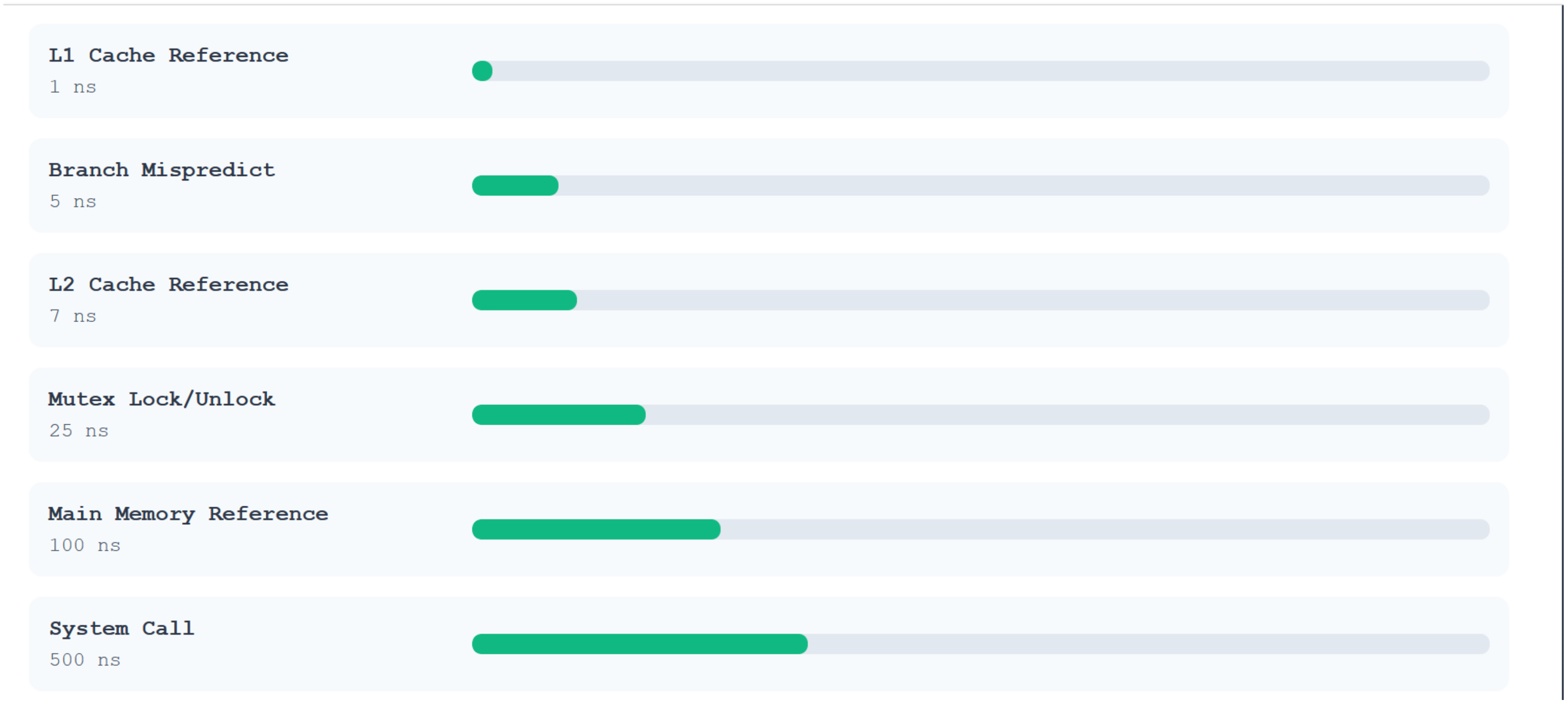
What were all those transistors used for?

- On-chip caches
- Pipelined instruction execution
  - Instruction-level parallelism (ILP)
- Many functional units
  - VLIW or superscalar to keep functional units busy
- Vector units
  - (e.g.) Intel's AVX 512
- Wider on-chip data-paths
  - 8bit → 16 bit → 32 bit → 64 bit



Intel Pentium floorplan

Caches: typical latency numbers



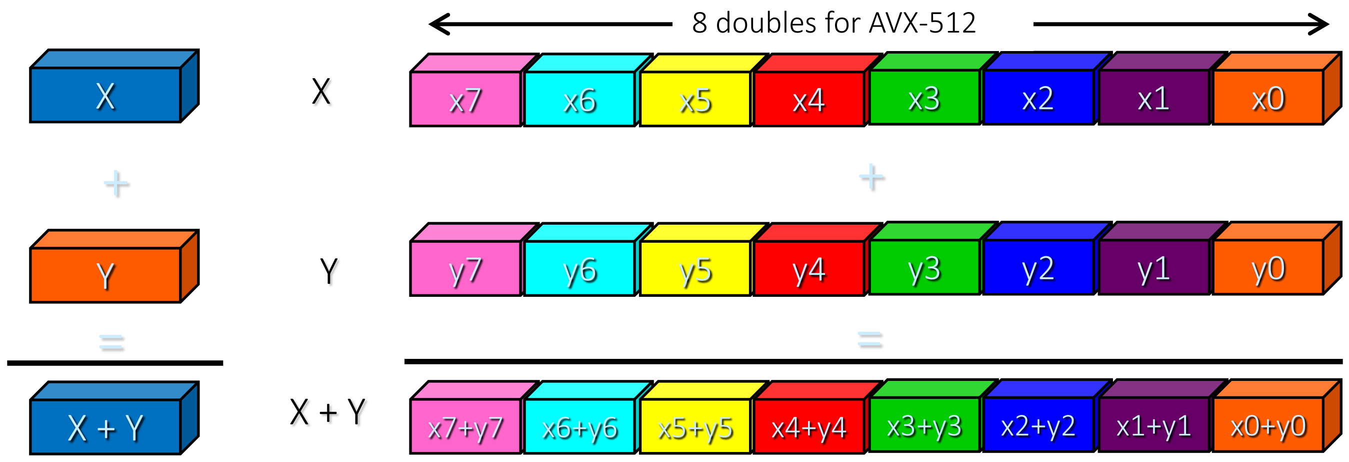
Software must exploit locality to make effective use of caches

From: Latency numbers every HPC programmer should know

Vector instructions

for (I=0; i<n; i++) Z[i] = X[i] + Y[i];

- ❑ Scalar mode
  - one instruction produces one result
  - E.g. vaddss, (vaddsd)
- ❑ Vector (SIMD) mode
  - one instruction can produce multiple results
  - E.g. vaddps, (vaddpd)



Note: AVX was introduced in 2011
Before that, MMX and SSE.

Software challenges for performance programmers before 2005

- Exploiting instruction-level parallelism
  - (e.g.) loop unrolling to create long basic blocks (see later)
- Exploiting vector parallelism
  - (e.g.) vectorization of innermost loops
- Exploiting memory hierarchy
  - exploit spatial and temporal locality
  - code and data transformations for enhancing spatial and temporal locality
  - (e.g.) blocking of loops

Getting performance is hard

- Amdahl’s Law
  - Simple observation that shows that unless most program operations can be optimized, the benefits of performance optimization are limited
  - Unoptimized portions of program become bottleneck
- Analogy: suppose I go from Austin to Houston at 60 mph, and return “infinitely” fast. What is my average speed?
  - Answer: 120 mph, not infinity

Amdahl's Law (details)

- In general, program will have both optimized and unoptimized portions
  - Suppose program has N operations
    - r*N operations in optimized portion
    - (1-r)*N operations in unoptimized portion
- Assume
  - Unoptimized portion requires one time unit per operation
  - Optimized portion can be executed infinitely fast so it takes zero time to execute.
- Speed-up:

$$\frac{\text{Original execution time}}{\text{Optimized execution time}} = \frac{N}{(1-r)*N} = \frac{1}{(1-r)}$$

- Even if r = 0.99, speed-up is only 100.

Unless most of your program is performance-optimized, you won't see much benefit.

SINCE 2005

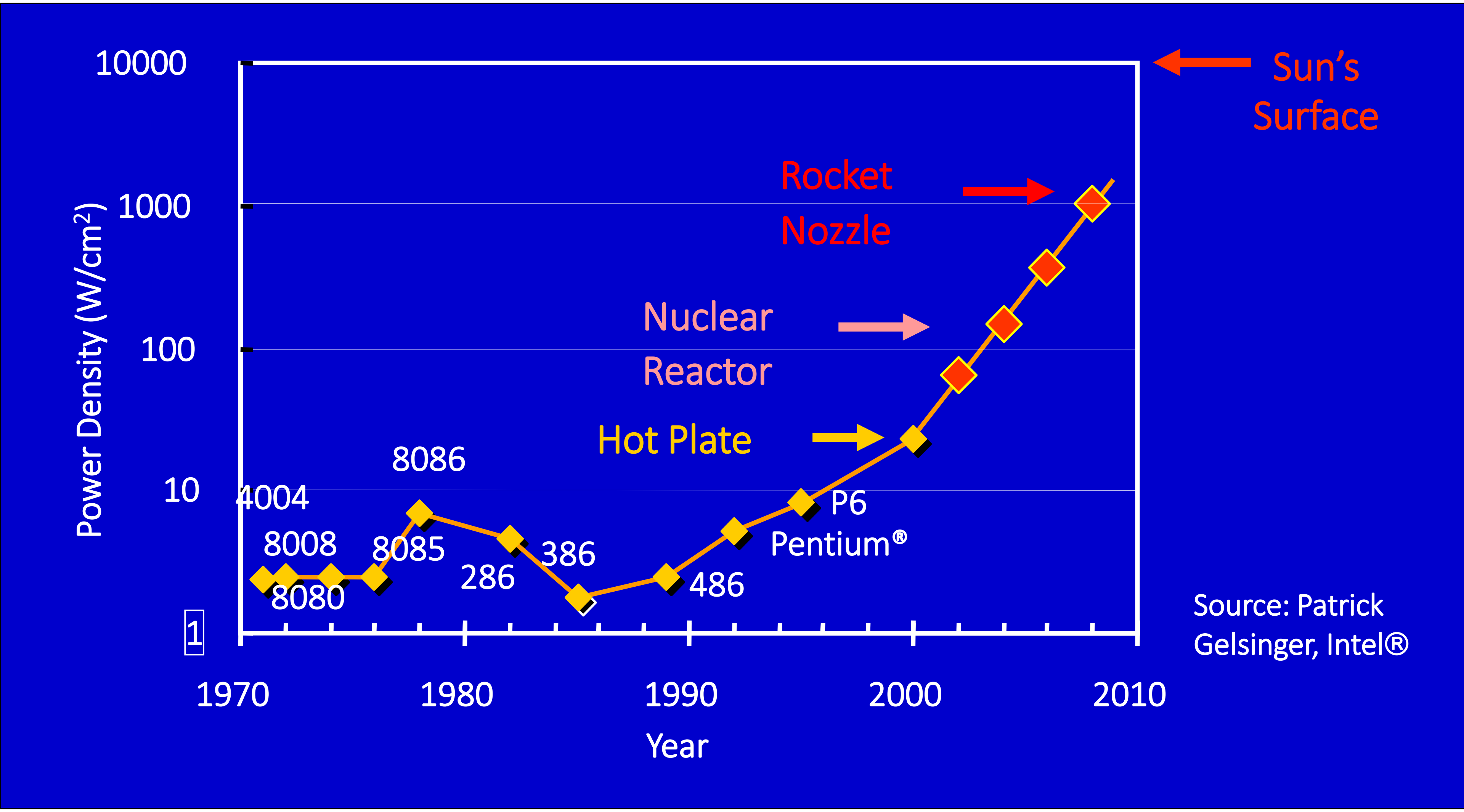
Fundamental change since ~2005

- Moore’s Law still holds
  - We get more transistors in each new technology generation
- However
  1. Architects have run out of ideas for how to use these transistors to speed up single-thread performance
  2. Processor clock speed have stalled at roughly 1-3 GHz

(1) Using the additional transistors: old ideas have run out of steam

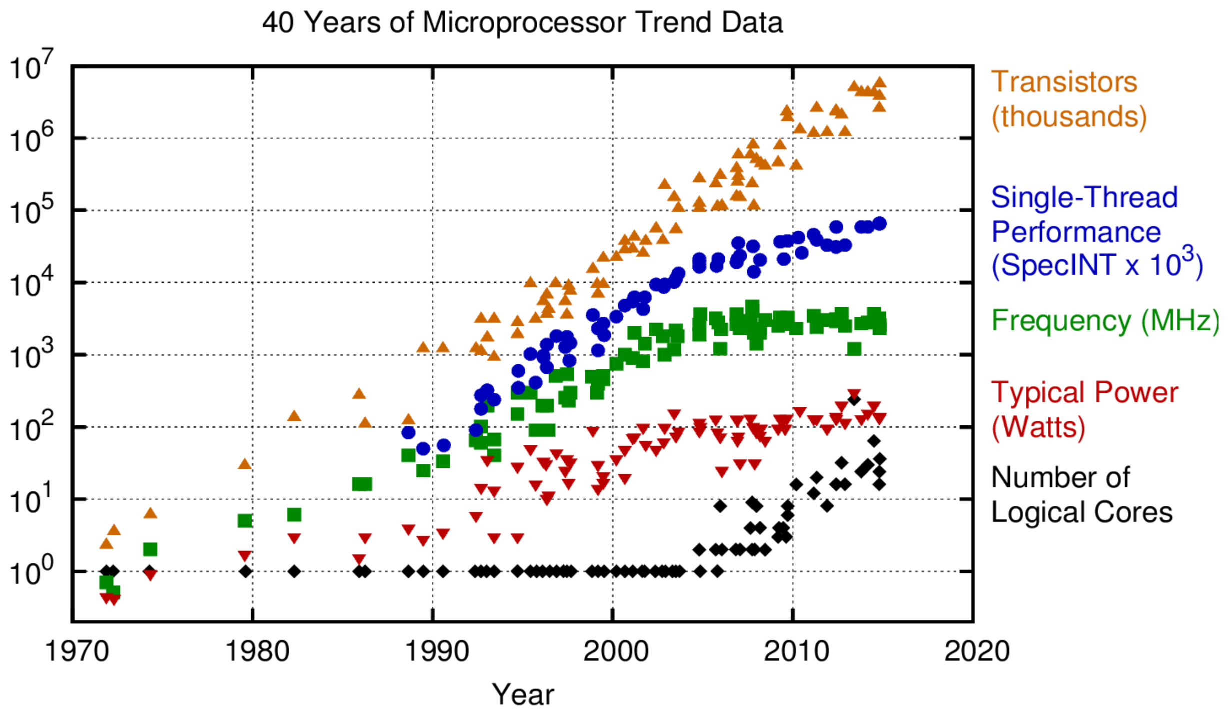
- More cache
  - More cache buys performance until working set of program fits in cache
- Deeper pipeline
  - Deeper pipeline buys frequency at expense of increased branch mis-prediction penalty
  - Deeper pipelines => higher clock frequency => more power
- Add more functional units/vector units
  - Diminishing returns for adding more units
- Wider data paths
  - Increases bandwidth between functional units in a core but we now have comprehensive 64-bit designs

(2) Processor clock speed increase has stalled



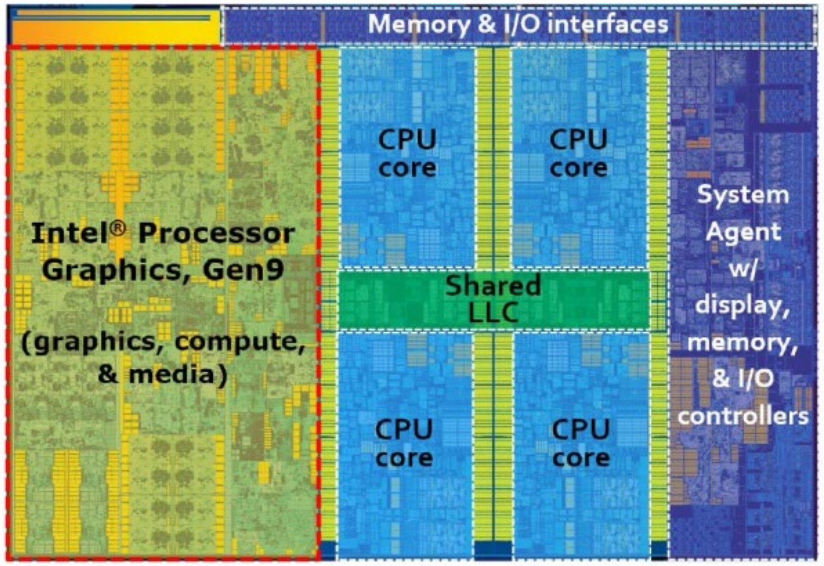
One use of transistors: go multicore

- Use transistors to build multiple cores without increasing clock frequency
  - does not require micro-architectural breakthroughs
  - non-linear scaling of power density with frequency will not be a problem

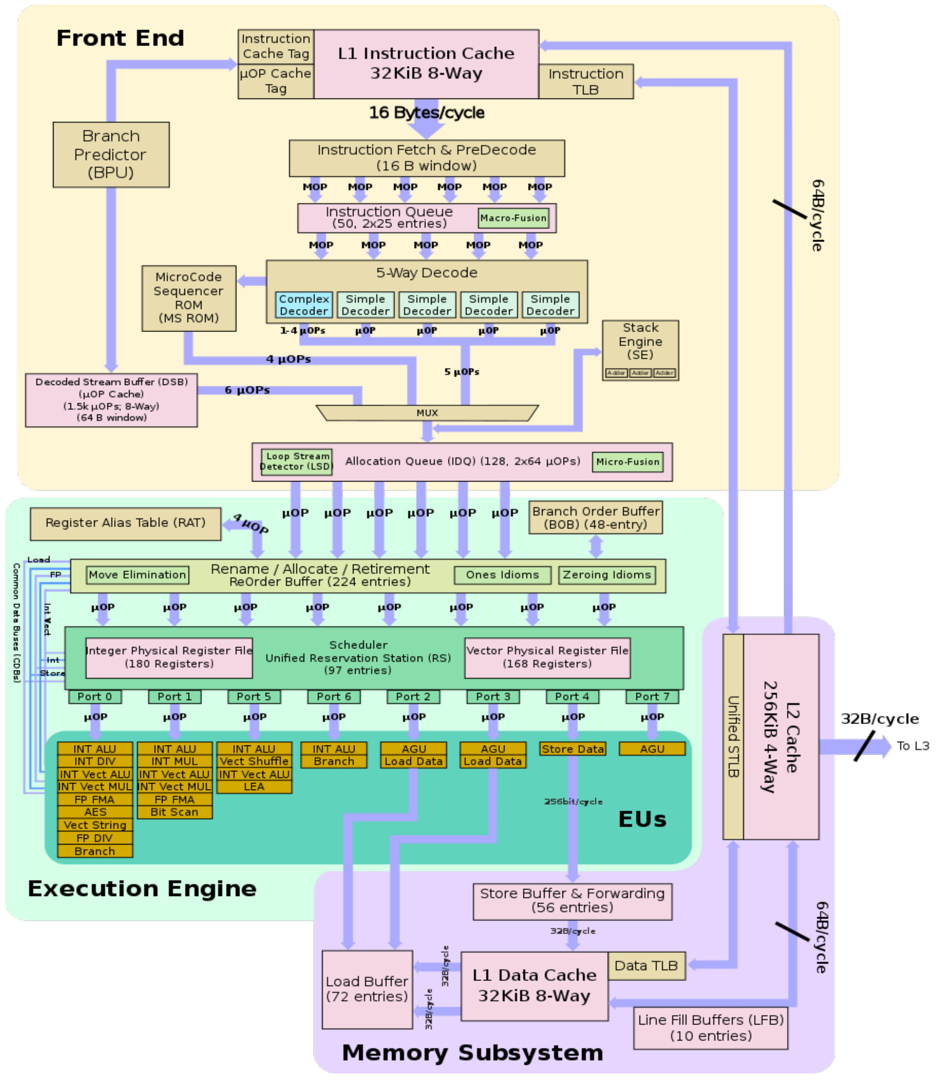


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Intel Skylake chip


Memory & I/O interfaces
CPU core
CPU core
CPU core
CPU core
Shared LLC
Intel® Processor Graphics, Gen9 (graphics, compute, & media)
System Agent w/ display, memory, & I/O controllers

Chip


Front End
Instruction Cache Tag
µOP Cache Tag
L1 Instruction Cache 32KiB 8-Way
Instruction TLB
16 Bytes/cycle
Branch Predictor (BPU)
Instruction Fetch & PreDecode (16 B window)
MOP
Instruction Queue (50, 2x25 entries)
Macro-Fusion
5-Way Decode
MicroCode Sequencer ROM (MS ROM)
Complex Decoder
Simple Decoder
1-4 µOPs
4 µOPs
5 µOPs
6 µOPs
Stack Engine (SE)
Decoded Stream Buffer (DSB) (µOP Cache) (1.5k µOPs; 8-Way) (64 B window)
MUX
Loop Stream Detector (LSD)
Allocation Queue (IDQ) (128, 2x64 µOPs)
Micro-Fusion
64B/cycle
Register Alias Table (RAT)
4 µOP
µOP
Branch Order Buffer (BOB) (48-entry)
Move Elimination
Rename / Allocate / Retirement ReOrder Buffer (224 entries)
Ones Idioms
Zeroing Idioms
Common Data Buses (CDBs)
Load
FP
Int Vect
Int Store
Integer Physical Register File (180 Registers)
Scheduler Unified Reservation Station (RS) (97 entries)
Vector Physical Register File (168 Registers)
Port 0
Port 1
Port 5
Port 6
Port 2
Port 3
Port 4
Port 7
INT ALU, INT DIV, INT Vect ALU, INT Vect MUL, FP FMA, AES, Vect String, FP DIV, Branch
INT ALU, INT MUL, INT Vect ALU, INT Vect MUL, FP FMA, Bit Scan
INT ALU, Vect Shuffle, INT Vect ALU, LEA
INT ALU, Branch
AGU, Load Data
AGU, Load Data
Store Data
AGU
256bit/cycle
EUs
Execution Engine
Unified STLB
L2 Cache 256KiB 4-Way
32B/cycle
To L3
Store Buffer & Forwarding (56 entries)
32B/cycle
Load Buffer (72 entries)
L1 Data Cache 32KiB 8-Way
Data TLB
Line Fill Buffers (LFB) (10 entries)
64B/cycle
Memory Subsystem

Block diagram of each core

Clusters and data-centers

TACC Frontera cluster

- 8,368 nodes
- Intel 8280 Cascade Lake processors with 56 cores/socket

Software challenges post-2005

- Exploiting parallelism: keep the cores busy
  - Node-level and thread-level parallelism
  - Load-balancing
- Exploiting memory hierarchy
  - Spatial and temporal locality
  - Avoid sharing data with other cores as far as possible
- New kinds of bugs:
  - race conditions, deadlocks

Parallel programming

- Shared-memory programming
  - Architecture: processor has some number of cores (e.g., Intel Skylake has up to 18 cores depending on the model)
  - Application program is decomposed into a number of threads, which run on these cores
  - Threads communicate by reading and writing memory locations
  - We will study pThreads and OpenMP for shared-memory programming

- Distributed-memory programming
  - Architecture: network of machines (Stampede II: 4,200 KNL hosts)
  - Application program and data structures are partitioned into processes, which run on machines
  - Processes communicate by sending and receiving messages since they have no memory locations in common
  - We will study MPI for distributed-memory programming

Major Lecture Topics

- Applications
  - Parallelism and locality in important algorithms
- Locality
  - Memory hierarchy, code and data transformations
- Vector parallelism
  - Vectorizing compilers
- Shared-memory parallelism
  - Multicore architectures, pThreads, OpenMP, TBB
- Distributed-memory parallelism
  - Clusters, MPI
- GPUs
  - CUDA