CS 377P:
Programming for Performance

Administration

- Instructors:
  - Keshav Pingali (Professor, CS, ECE & OI)
    - 4.126 Peter O'Donnell Building (POB)
    - Email: pingali@cs.utexas.edu
- TA:
  - Kaveri Mukundan
    - Email: kaverimukundan2013@gmail.com

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

- 6-7 programming projects
  - These will be more or less evenly spaced through the semester
  - Some projects will require the use of Intel performance analysis tools
- One mid-semester exam
  - Date: TBA (just before spring break)
- Final exam

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 (Intel Vtune, Advisor) and libraries (MPI) but for us, they are a means to an end and not end in themselves.

What this course IS about

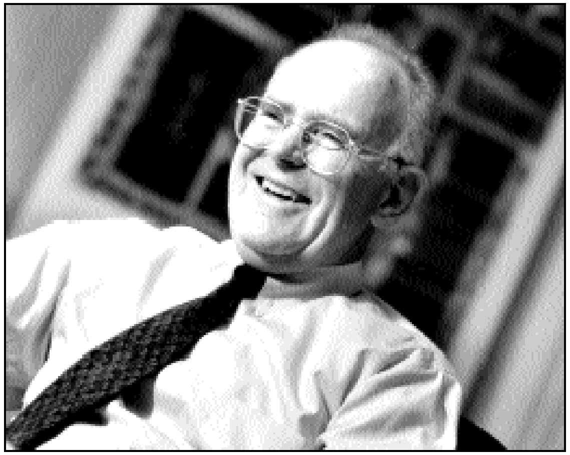
- Architects invent many hardware features for boosting program performance
- Usually, software can benefit from these features only if it is carefully written to exploit them
- Our agenda in CS 377P:
  – Understand key performance-critical architectural features in modern computers
  – Develop general 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 waited and then bought a new computer
  - Small number of 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 evolution?

Moore's Law

- What Moore said [1965]:
  - Number of transistors on a chip double every new generation of technology (~1.5 years)
- What people think Moore said:
  - Processor frequency doubles every 1.5 years

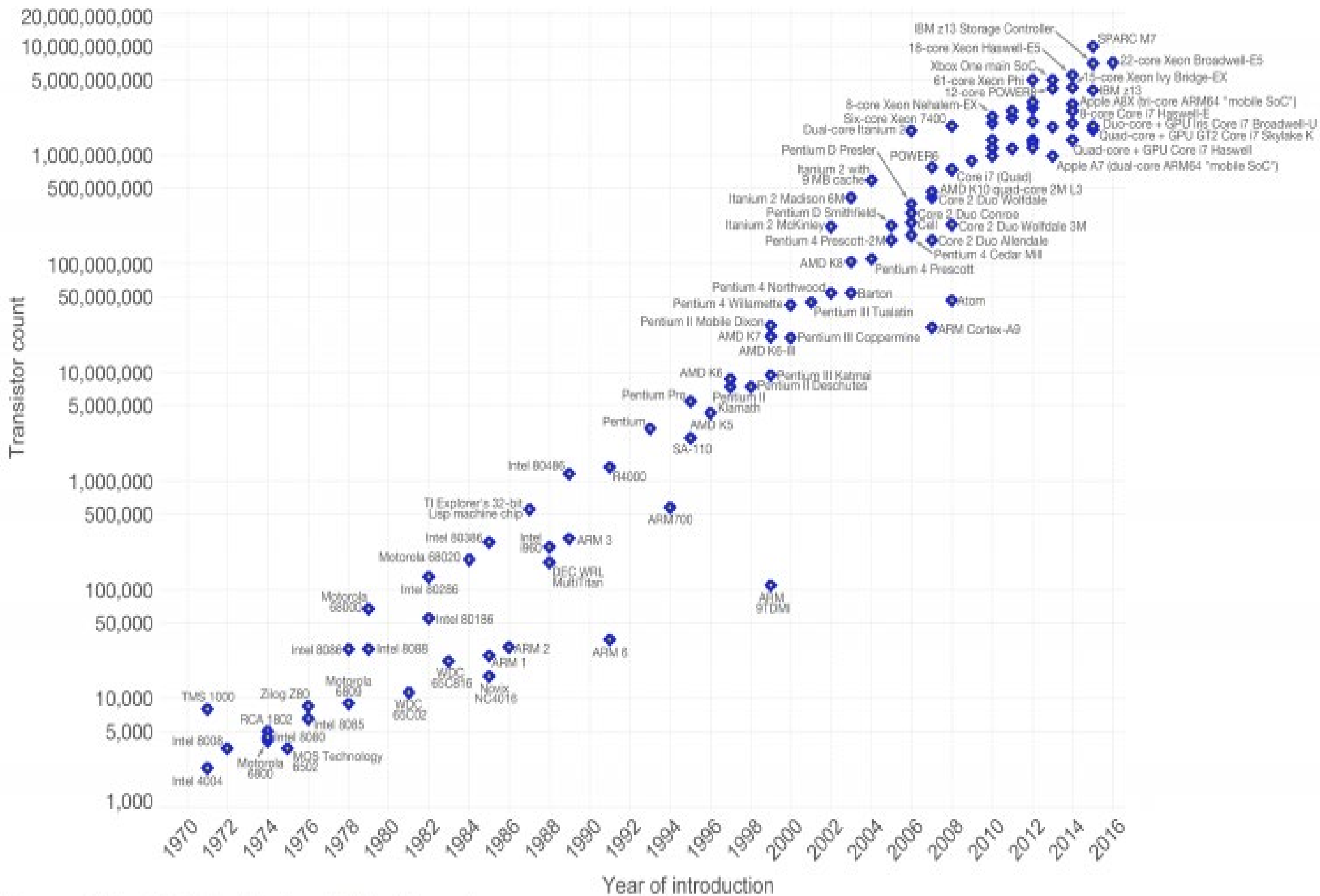


Gordon Moore (Intel)

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

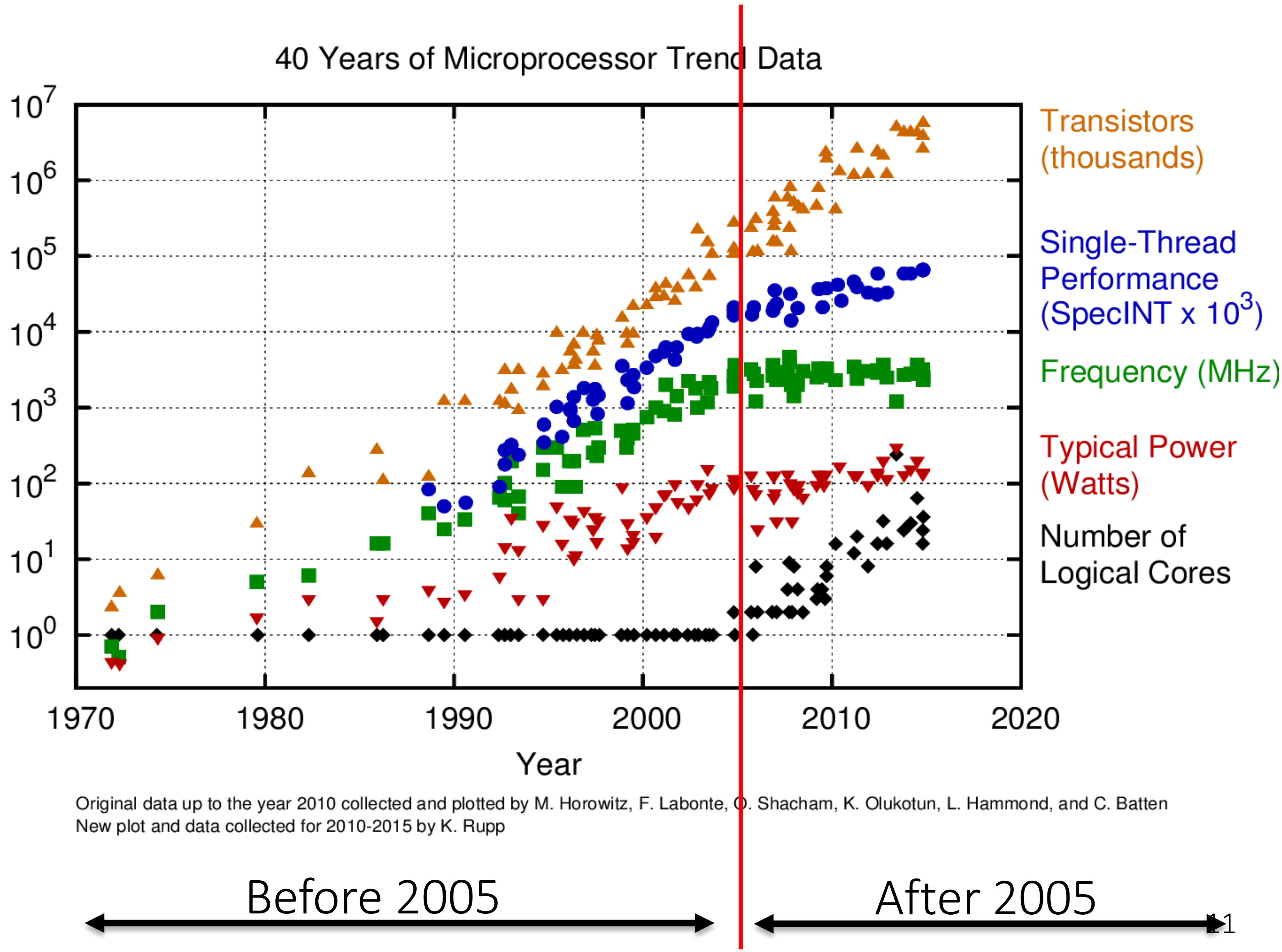
Our World in Data

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.



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.

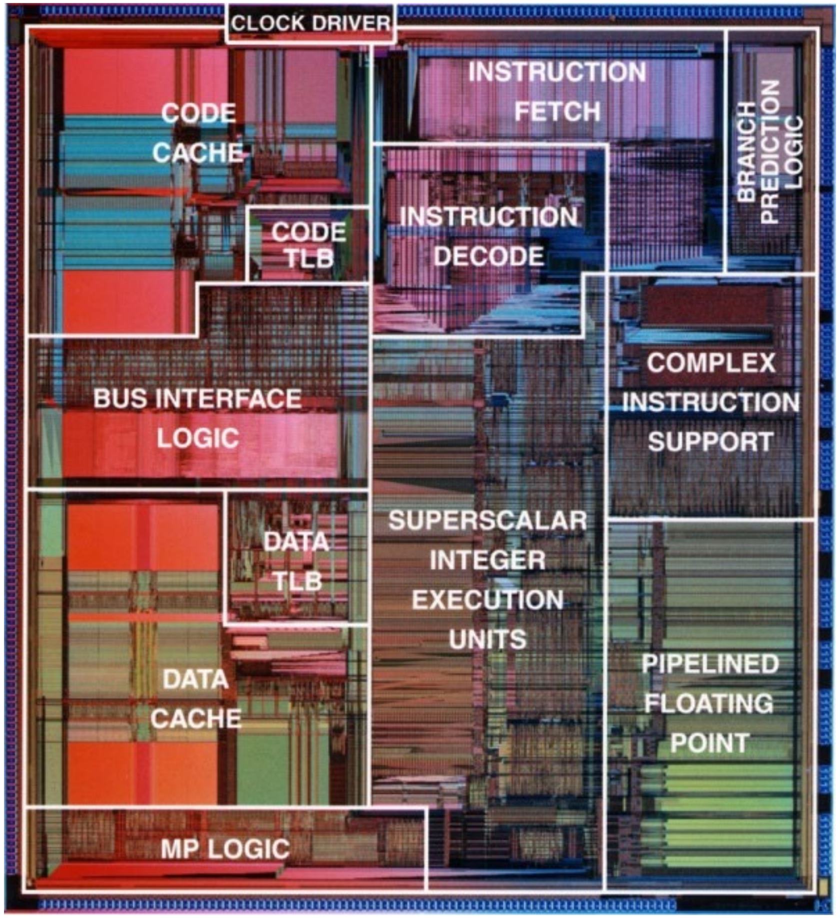
Microprocessor trend data



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 (today)

L1 cache reference/hit 1.5 ns 4 cycles
Floating-point add/mult/FMA operation 1.5 ns 4 cycles

L2 cache reference/hit 5 ns 12 ~ 17 cycles

L3 cache hit 16-40 ns 40-300 cycles

256MB main memory reference 75-120 ns TinyMemBench on "Broadwell" E5-2690v4

Read 1MB sequentially from disk 5,000,000 ns 5,000 us 5 ms ~200MB/sec hard disk (seek time would be additional latency)
Random Disk Access (seek+rotation) 10,000,000 ns 10,000 us 10 ms

Send packet CA->Netherlands->CA 150,000,000 ns 150,000 us 150 ms

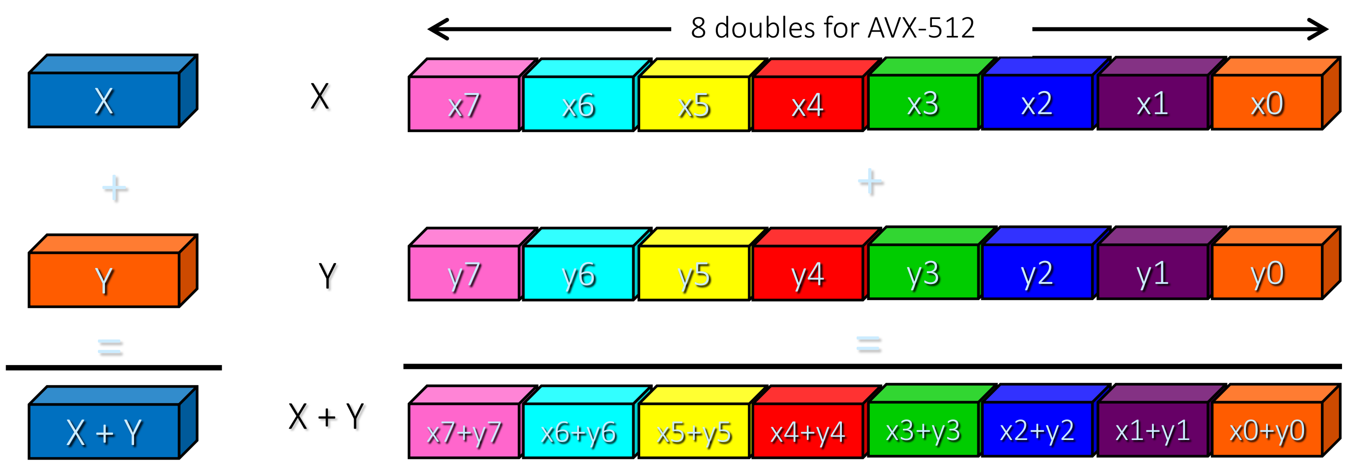
Locality is important.

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
- 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

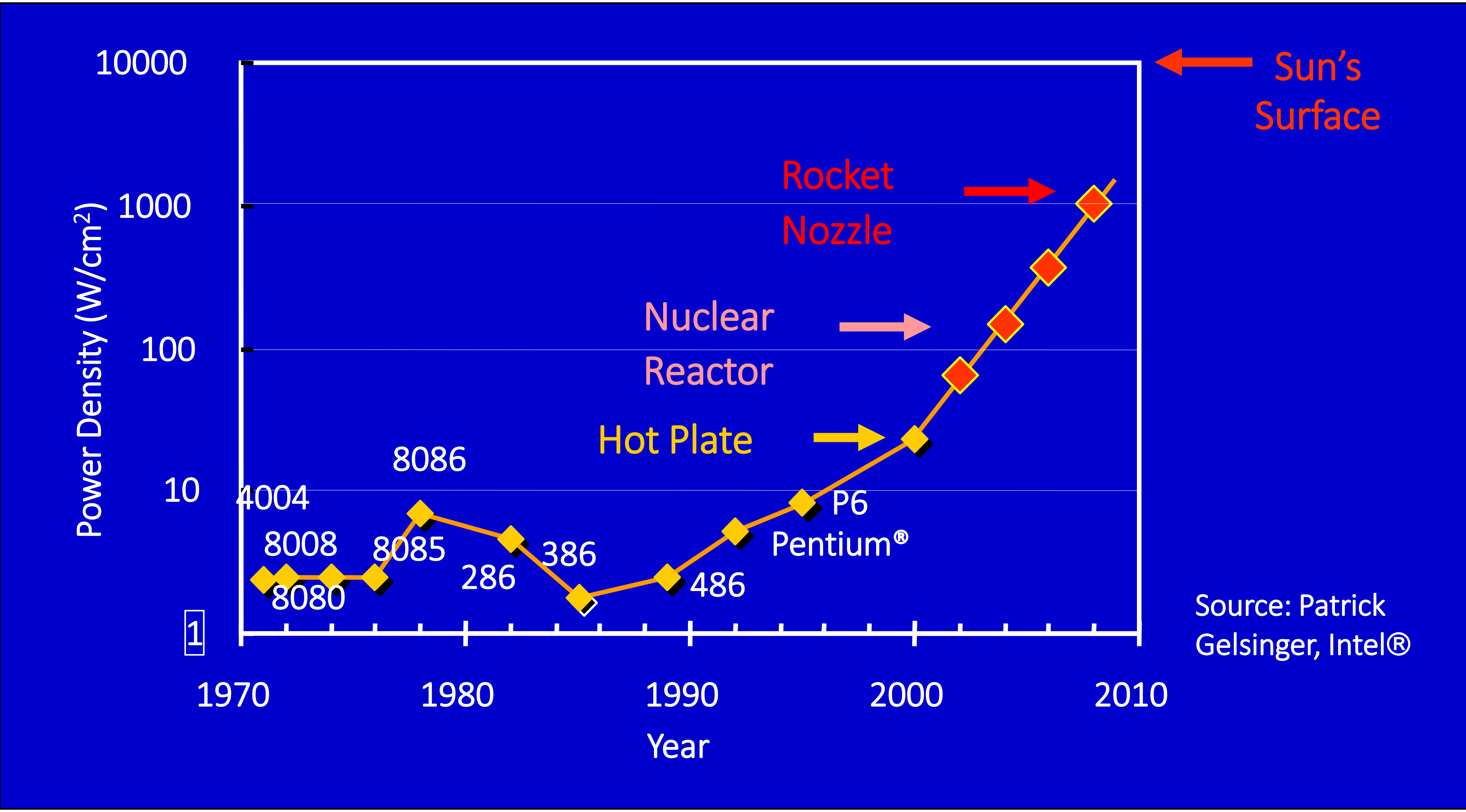
Fundamental change since ~2005

- 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

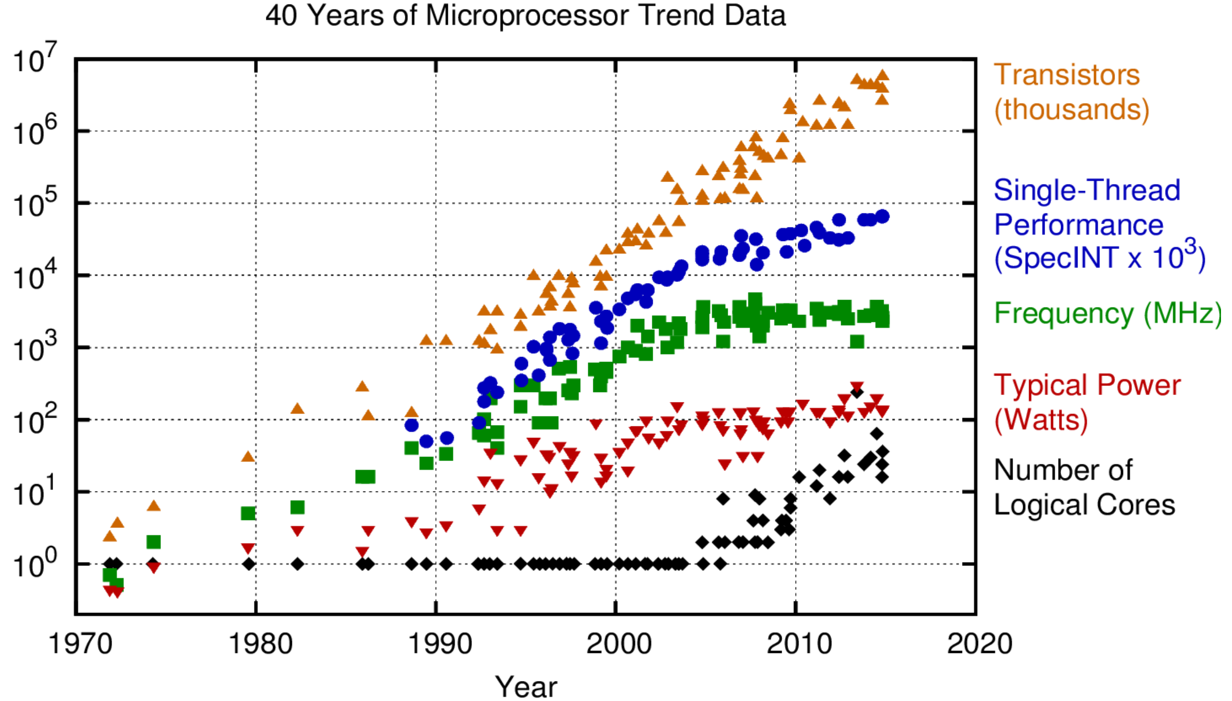
- More cache
  - More cache buys performance until working set of program fits in cache
- Deeper pipeline
  - Deeper pipeline buys frequency at expense of increased branch mis-prediction penalty
  - Deeper pipelines => higher clock frequency => more power
- Add more functional units/vector units
  - 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



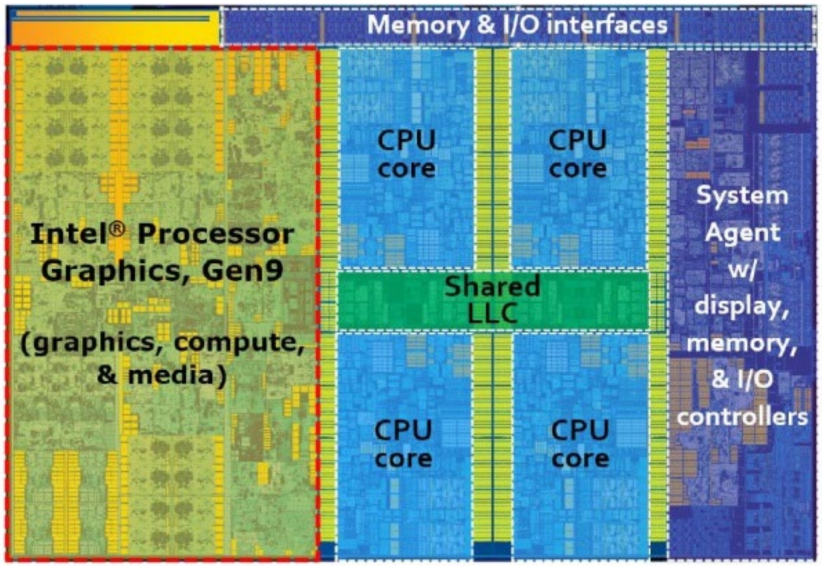
One use of transistors: go multicore

- 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

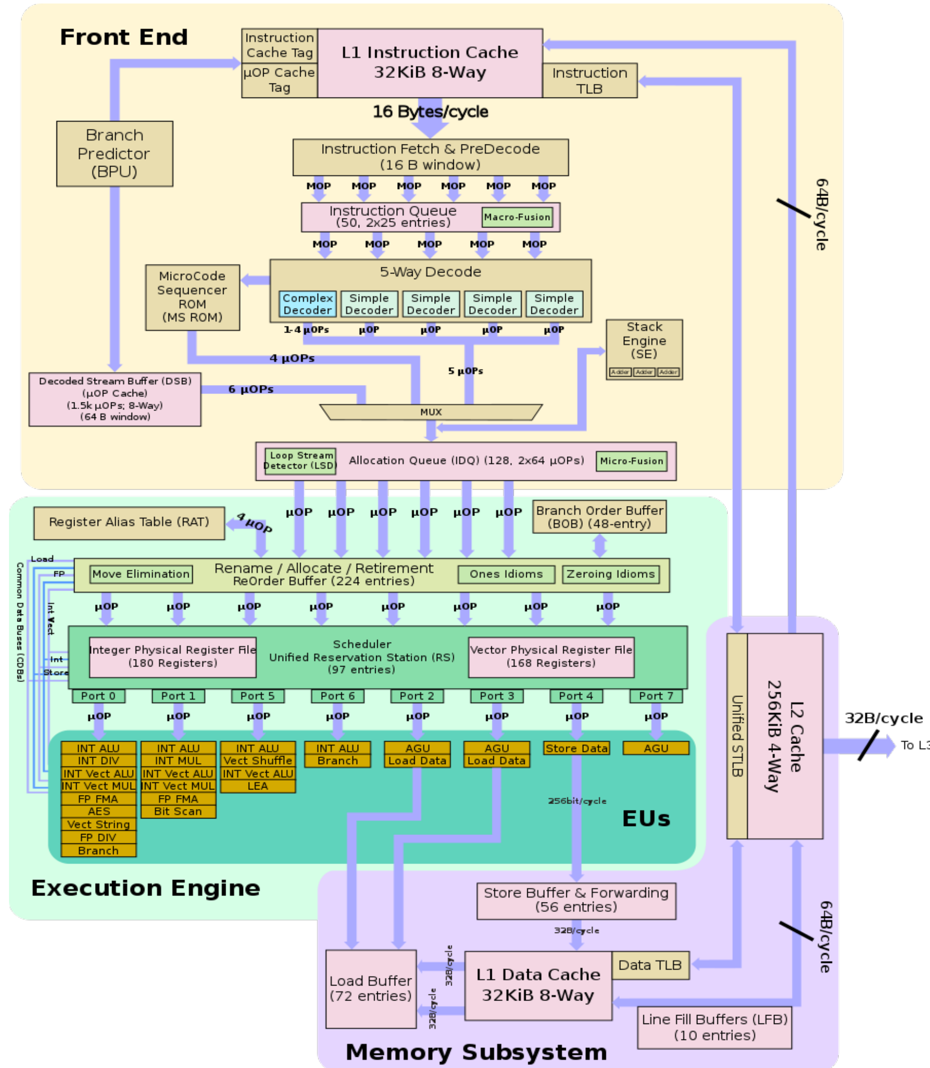


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Intel Skylake chip



Chip



Block diagram of each core

Clusters and data-centers

TACC Stampede 2 cluster

- 4,200 Intel Knights Landing nodes, each with 68 cores
- 1,736 Intel Xeon Skylake nodes, each with 48 cores
- 100 Gb/sec Intel Omni-Path network with a fat tree topology employing six core switches

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

Intel lectures

- Some lectures will be taught by Intel researchers
- Special focus on using Intel tools for writing, debugging and tuning parallel programs
- If you are registered for the course, you will get a license later for these tools