GPU Programming

Dr. Farshad Khunjush

Department of Computer Science and Engineering
Shiraz University

Fall 2024

Introduction:
The Multicore Revolution (Third software crisis and its origins and the roadmap to its solutions)

Dr. Farshad Khunjush

Acknowledgement: some slides borrowed from presentations by Jim Demel, Horst Simon, Mark D. Hill, David Patterson, Saman Amarasinghe, Richard Brunner, Luddy Harrison, Jack Dongarra Katherine Yelick, Matei Ripeanu

The Software Crisis

- *E. Dijkstra, 1972 Turing Award Lecture*
  - “To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem."

The First Software Crisis

- Time Frame: ’60s and ’70s
- Problem: Assembly Language Programming
  - Computers could handle larger more complex programs
- Needed to get Abstraction and Portability without losing Performance

Solution to the First Crisis

- High-level languages for von-Neuman machines
  - FORTRAN and C
- Provided "common language" for uni-processors.
  - Common Properties:
    - Single Flow of Control and Single Memory Image
  - Differences:
    - ISA (Instruction Set Architecture), Register Files, and Functional Units

The Second Software Crisis (80’s and 90’s)

- Problem: Inability to build and maintain complex and robust applications requiring multi-million lines of code developed by hundreds of programmers
  - Computers could handle larger more complex programs
- Needed to get Composability and Maintainability
  - High-performance was not an issue → left for Moore’s Law

Solution to the Second Crisis

- Object Oriented Programming
  - C++, C# and Java
- Also…
  - Better tools
    - Component libraries
  - Better software engineering methodology
    - Design patterns, specification, testing, code reviews

Today: Programmers are Oblivious to Processors

- Solid boundary between Hardware and Software
- Programmers don't have to know anything about the processor
  - High level languages abstract away the processors
    - Ex: Java byte code is machine independent
  - Moore's law does not require the programmers to know anything about the processors to get good speedups

Today: Programmers are Oblivious to Processors (Cont' d)

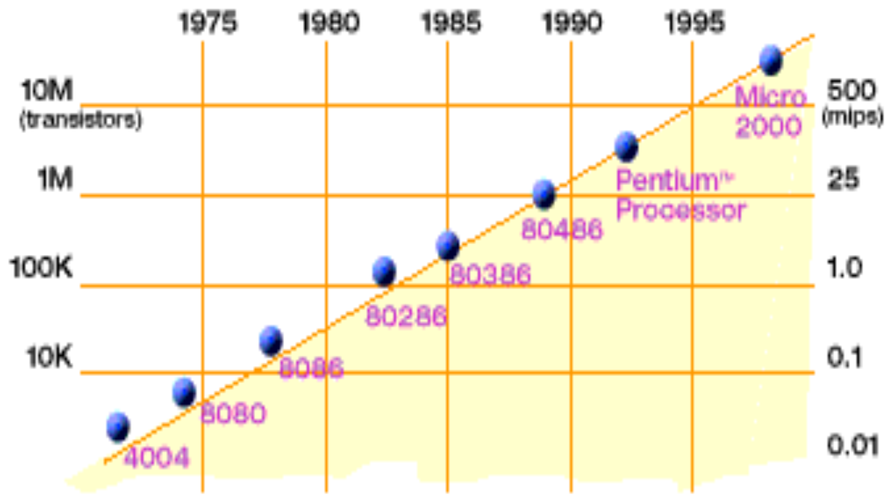
- Programs are oblivious of the processor → work on all processors
  - A program written in '70 using C still works and is much faster today
- This abstraction provides a lot of freedom for the programmers

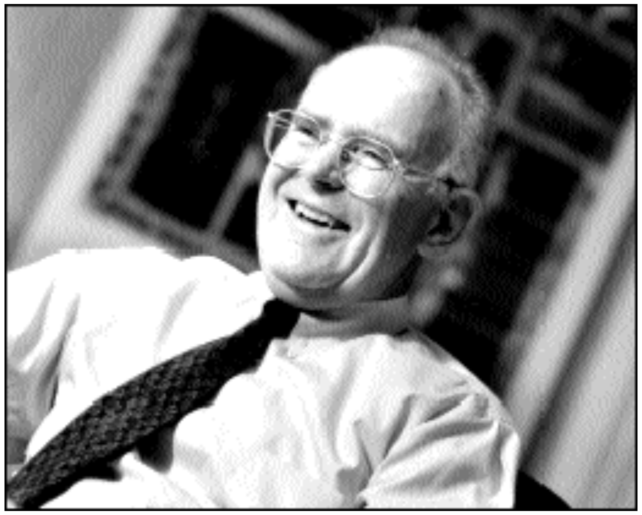
Third Software Crisis (2005-20..?)

- Origins:
  - Problem: Sequential performance is left behind by Moore's law
  - Needed continuous and reasonable performance improvements
    - to support new features
    - to support larger datasets
  - Needed improvements while sustaining portability and maintainability without unduly increasing complexity faced by the programmer

    → critical to keep-up with the current rate of evolution in software

Technology Trends: Microprocessor Capacity





2X transistors/Chip Every 1.5 years

Called "Moore's Law"

Microprocessors have become smaller, denser, and more powerful.

Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.

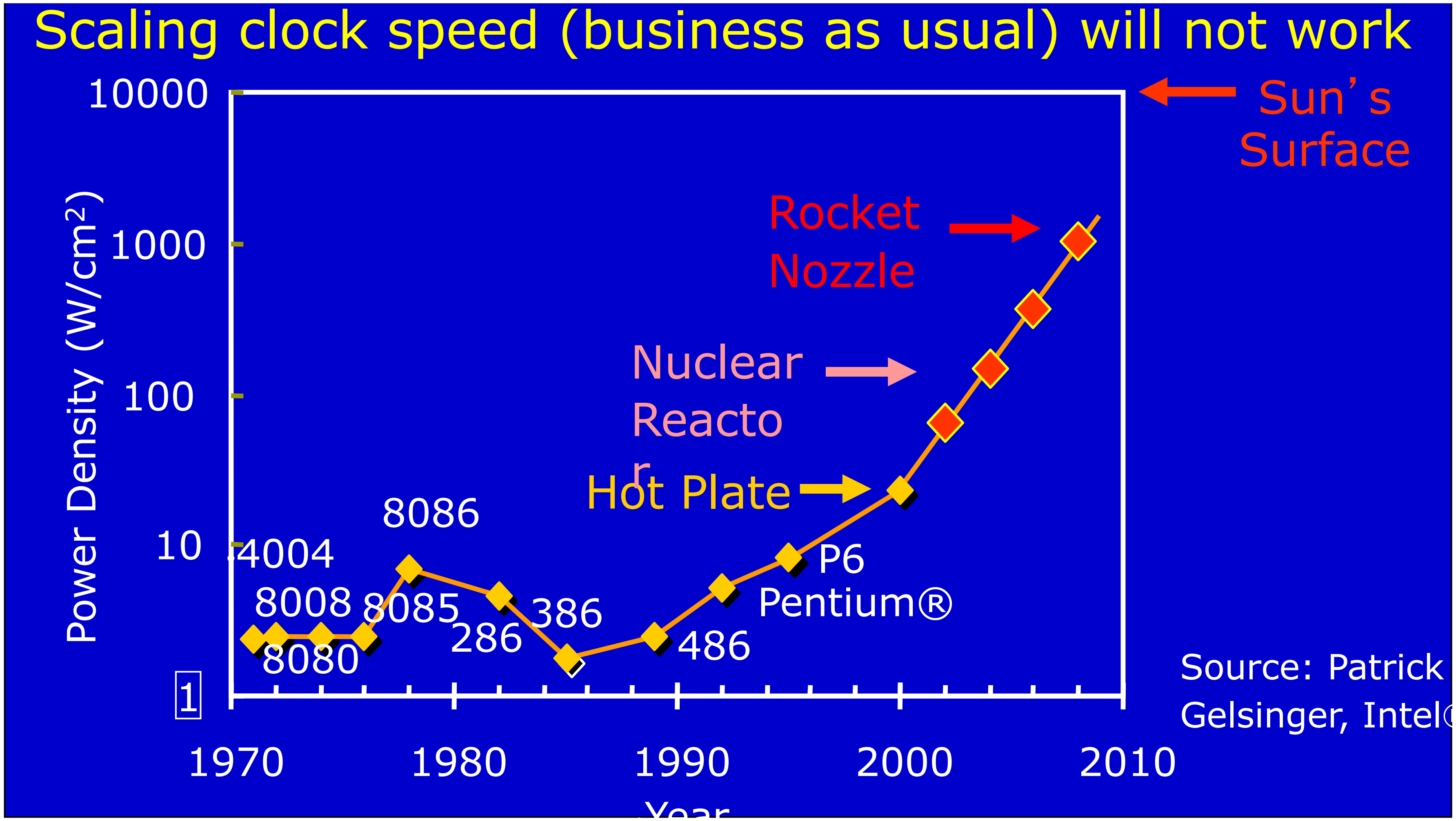
Slide source: Jack Dongarra

But there are limiting forces

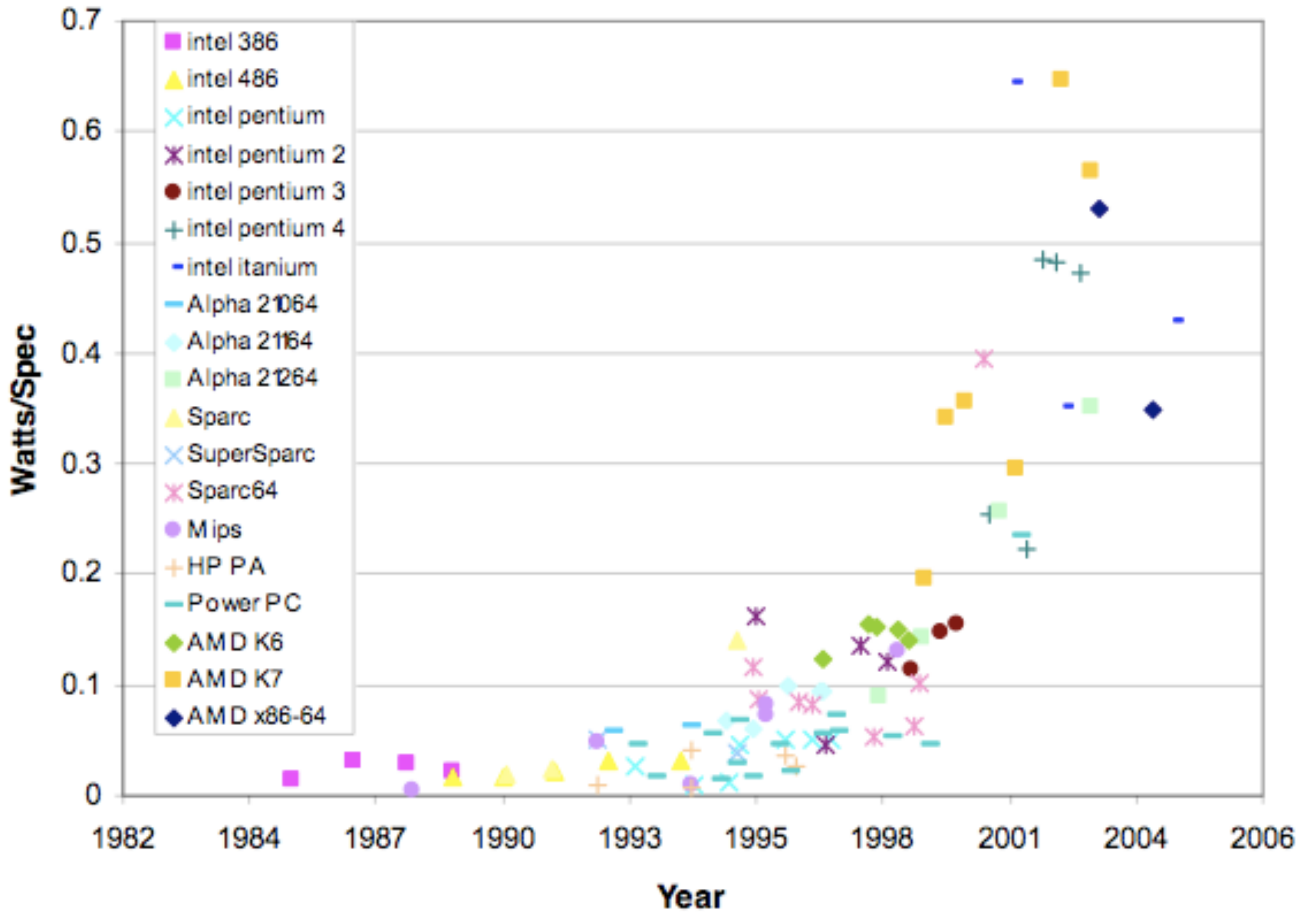
Limit #1: Power density

Can soon put more transistors on a chip than can afford to turn on.

-- Patterson '07



Power Efficiency (Watt/Spec)



Parallelism Saves Power

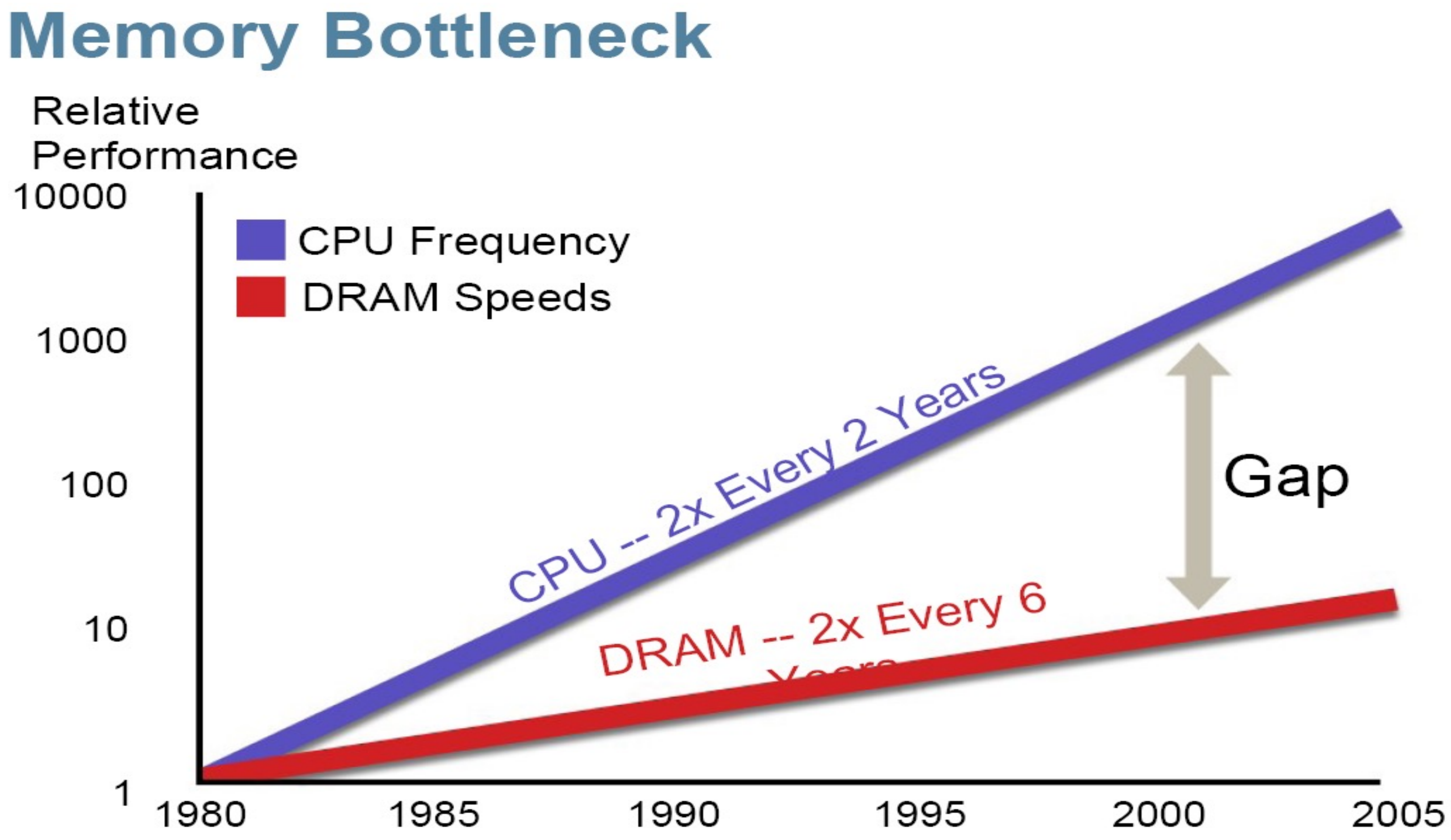
- Exploit explicit parallelism for reducing power

  $\text{Power} = C * V^2 * F$ $\qquad$ $\text{Performance} = \text{Cores} * F * 1$

  Capacitance Voltage Frequency

- Using additional cores
  - Increase density (= more transistors = more capacitance)
  - Can increase cores (2x) and performance (2x)
  - Or increase cores (2x), but decrease frequency (1/2): same performance at ¼ the power
- Additional benefits
  - Small/simple cores → more predictable performance

Limit #2: Memory Wall



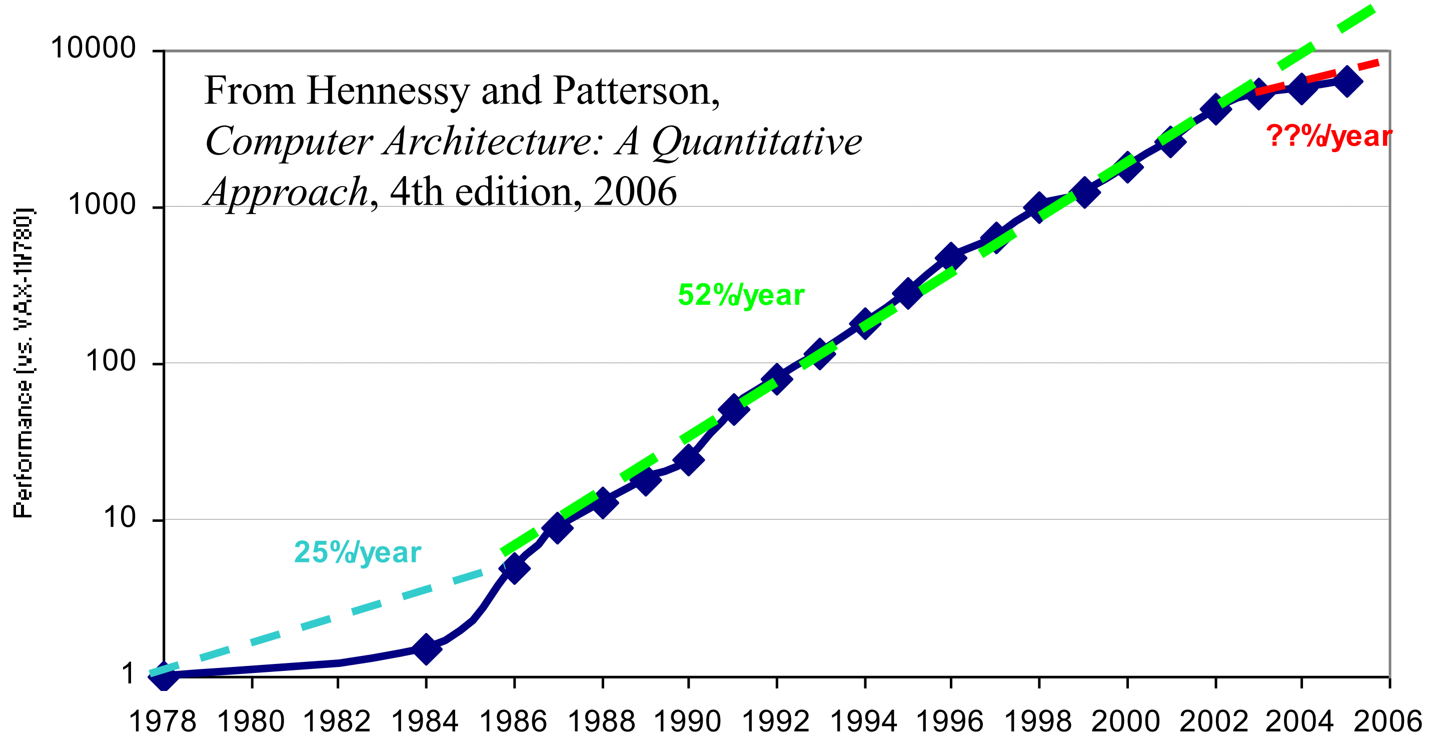
source: www.opensparc.net

Memory Wall Implications

- Each memory access takes hundreds of CPU cycles
- Increasing clock frequency doesn't automatically improve performance anymore!

Limit #3: Hidden Parallelism Tapped Out

Application performance was increasing by 52% per year as measured by the SpecInt benchmarks here



VAX: **25%/year 1978 to 1986**

RISC + x86: **52%/year 1986 to 2002**

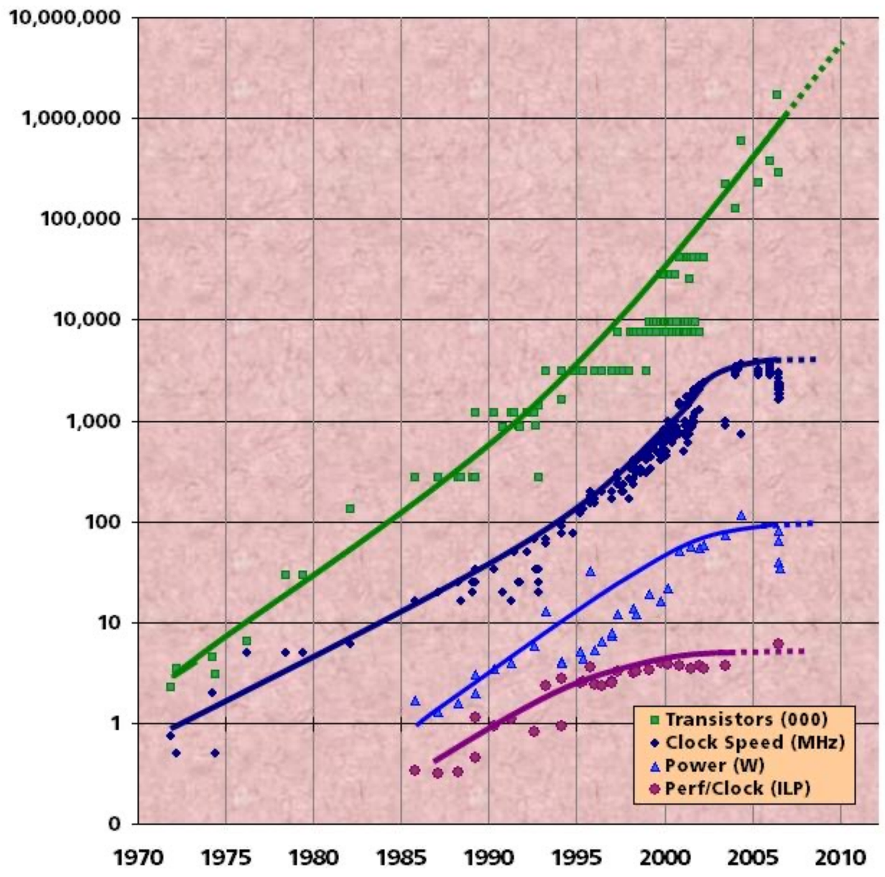
Limit #3: Hidden Parallelism Tapped Out

- Superscalar (SS) designs were the state of the art; many forms of parallelism not visible to programmer
  - multiple instruction issue
  - dynamic scheduling: hardware discovers parallelism between instructions
  - speculative execution: look past predicted branches
  - non-blocking caches: multiple outstanding memory ops
- You may have heard of these in ADV COMP. ARCH., but you haven't needed to know about them to write software !!
- Unfortunately, these sources have been used up

Revolution is Happening Now

- Chip density is continuing increase ~2x every 2 years
  - Clock speed is not
  - Number of processor cores may double instead
- There is little or no hidden parallelism (ILP) to be found
- Parallelism must be exposed to and managed by software

Source: Intel, Microsoft (Sutter) and Stanford (Olukotun, Hammond)



Moore's Law reinterpreted

- Number of cores per chip will double every two years
- Clock speed will not increase (possibly decrease)
- Need to deal with systems with large number of concurrent threads
  - Millions for HPC systems
  - Thousands for servers
  - Hundreds for workstations and notebooks
- Need to deal with inter-chip parallelism as well as intra-chip parallelism

Why Parallelism?

- These arguments are no longer theoretical
- All major processor vendors are producing multi-core chips
  - Every machine will soon be a parallel machine
  - All programmers will be parallel programmers???
- New software model
  - Want a new feature? Hide the "cost" by speeding up the code first
  - All programmers will be performance programmers???
- Some may eventually be hidden in libraries, compilers, and high level languages
  - But a lot of work is needed to get there
- Big open questions:
  - What will be the killer apps for multicore machines
  - How should the chips be designed, and how will they be programmed?

Why writing (fast) parallel programs is hard

Principles of Parallel Computing

- Finding enough parallelism (Amdahl's Law)
- Granularity
- Locality
- Load balance
- Coordination and synchronization
- Performance modeling

➡ All of these things makes parallel programming even harder than sequential programming.

“Automatic” Parallelism in Modern Machines

- Bit level parallelism
  - within floating point operations, etc.
- Instruction level parallelism (ILP)
  - multiple instructions execute per clock cycle
- Memory system parallelism
  - overlap of memory operations with computation
- OS parallelism
  - multiple jobs run in parallel on commodity SMPs

Limits to all of these -- for very high performance, need user to identify, schedule and coordinate parallel tasks

Amdahl' s Law

- Simple software assumption
  - Fraction F of execution time perfectly parallelizable
  - No Overhead for
    - Scheduling
    - Synchronization
    - Communication, etc.
  - Fraction 1 – F Completely Serial
- Time on 1 core = (1 – F) / 1 + F / 1 = 1
- Time on N cores = (1 – F) / 1 + F / N

Finding Enough Parallelism

$$\text{Amdahl's Speedup} = \frac{1}{\frac{1-F}{1} + \frac{F}{N}}$$

- Implications:
  - Attack the common case when introducing parallelization's: When *F* is small, optimizations will have little effect.
  - Even if the parallel part speeds up perfectly performance is limited by the sequential part
    - As N approaches infinity, speedup is bound by 1/(1 – F).
  - The aspects you ignore will limit speedup
- Discussion:
  - Can you ever obtain super-linear speedups?

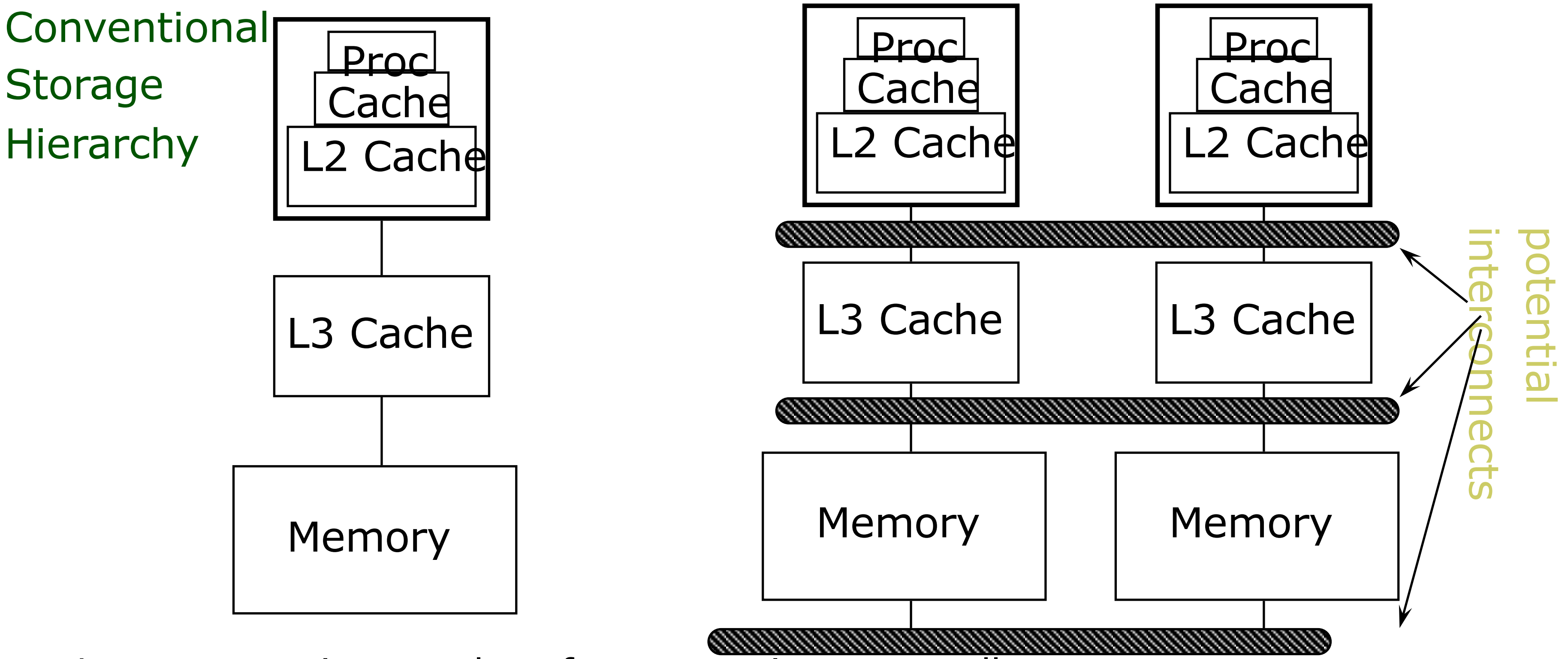
Overhead of Parallelism

- Given enough parallel work, this is the biggest barrier to getting desired speedup
- Parallelism overheads include:
  - cost of starting a thread or process
  - cost of communicating shared data
  - cost of synchronizing
  - extra (redundant) computation

Overhead of Parallelism (Cont' d)

- Each of these can be in the range of milliseconds (=millions of flops) on some systems
- Tradeoff: Algorithm needs sufficiently large units of work to run fast in parallel (i.e. large granularity), but not so large that there is not enough parallel work

Locality and Parallelism



- Large memories are slow, fast memories are small
- Storage hierarchies are large and fast <u>on average</u>
- Parallel processors, collectively, have large, fast cache
  - the slow accesses to “remote” data we call “communication”
- Algorithm should do most work on local data

Course Organization (Tentative)

- The Multicore Revolution (Third software crisis and its origin and roadmap to its solutions)
- Different Parallel Architectures & Multicore Architectures (From pipeline to Homogeneous & Heterogeneous Multi-core architectures)
- Concurrency Programming Principles
- Shared Memory and Pthread Programming
- Introduction to GPGPU Architectures
- CUDA programming Concepts
- Reduction and FFT Implementation
- Profiling CUDA programs
- Interesting new features
- Machine Learning Libraries using CUDA
- …

Grading (tentative)

Assignment	25%
1st Assignment	5%
2nd Assignment	5%
3rd Assignment	5%
4th Assignment	5%
5th Assignment	5%
Final Project	25%
Report & Presentation	25%
Mid-Term Exam	20%
Final Exam	30%