Introduction to HPC Hardware

Dr. Axel Kohlmeyer

Associate Dean for Scientific Computing, CST
Associate Director, Institute for Computational Science
Assistant Vice President for High-Performance Computing

Temple University
Philadelphia PA, USA

a.kohlmeyer@temple.edu

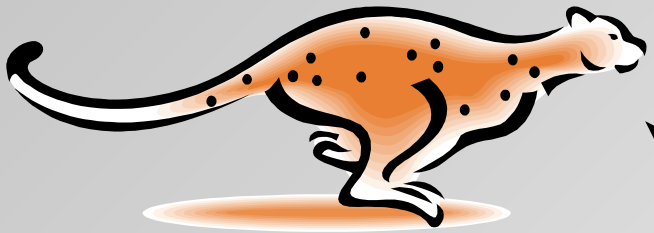
ICMS
Institute for Computational Molecular Science

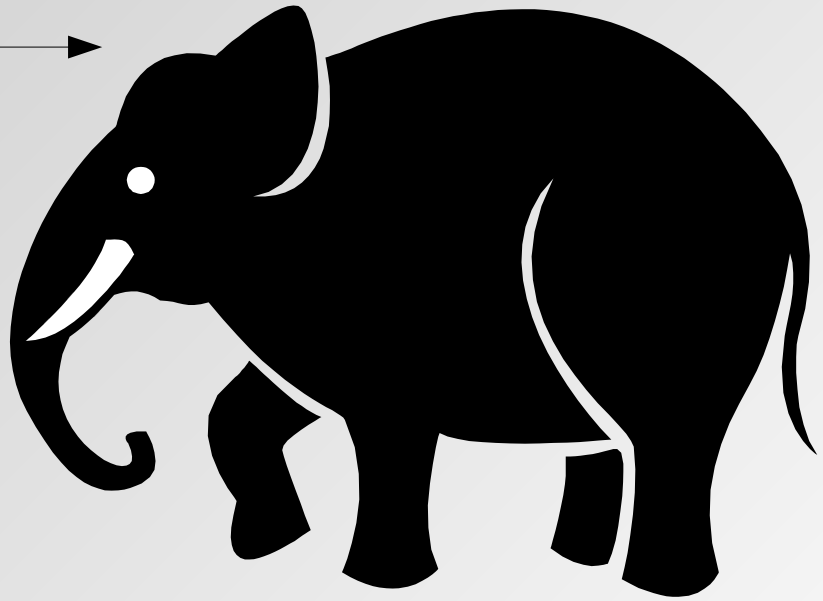
Why use Computers in Science?

- Use complex theories without a closed solution: solve equations or problems that can only be solved numerically, i.e. by inserting numbers into expressions and analyzing the results
- Do “impossible” experiments: study (virtual) experiments, where the boundary conditions are inaccessible or not controllable
- Benchmark correctness of models and theories: the better a model/theory reproduces known experimental results, the better its predictions

Why Would I Care About HPC?

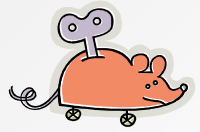
- My problem is big





- My problem is complex

- My computer is too small and too slow



- My software is not efficient and/or not parallel
  -> often scaling with system size the problem

ICMS
Institute for Computational Molecular Science

How to Get My Answers Faster?

- Work harder
  => get faster hardware (get more funding)
- Work smarter
  => use optimized algorithms (libraries!)
  => write faster code (adapt to match hardware)
  => trade performance for convenience
  (e.g. compiled program vs. script program)
- Delegate parts of the work
  => parallelize code, (grid/batch computing)
  => use accelerators (GPU/MIC CUDA/OpenCL)

What Determines Performance?

- How fast is my CPU?
- How fast can I move data around?
- What is the scaling behavior of my algorithm
- How well can I parallelize the work?

  => efficient serial algorithms may not parallelize as well as less efficient (simpler) ones

  => always run benchmarks to understand requirements of your applications and properties of your hardware

How Do We Measure Performance?

- For numerical operations: FLOP/s (or FLOPS)
  = Floating-Point Operations per second
- Theoretical maximum (**peak**) performance:
  clock rate *x* number of double precision addition and/or multiplications completed per clock
  => 2.5 Ghz *x* 8 FLOP/clock = 20 GigaFLOP/s
  => can never be reached (data load/store)
- Real (**sustained**) performance:
  => very application dependent
  => Top500 uses Linpack (linear algebra)

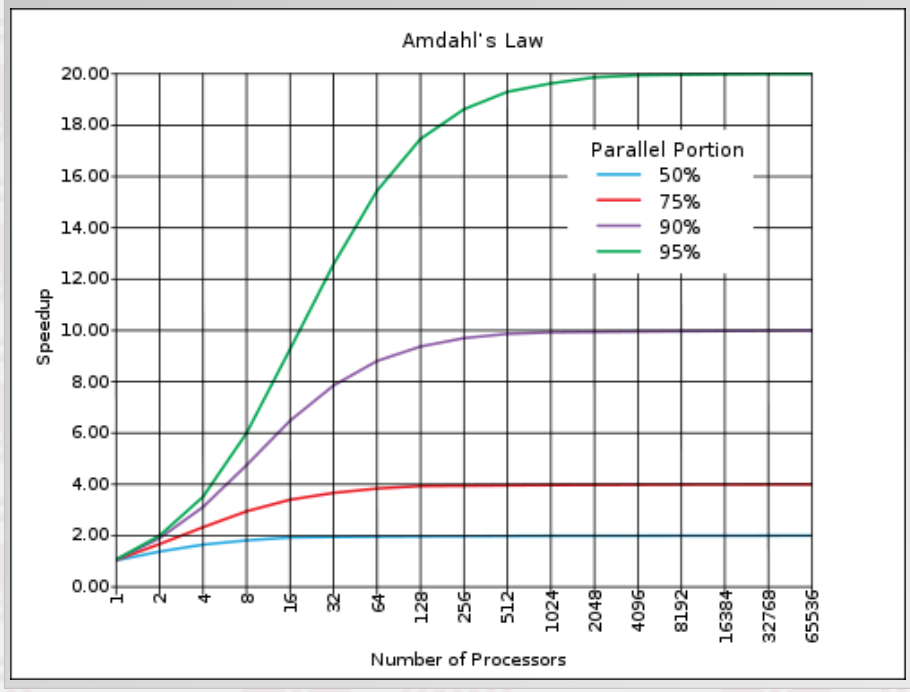
A High-Performance Problem

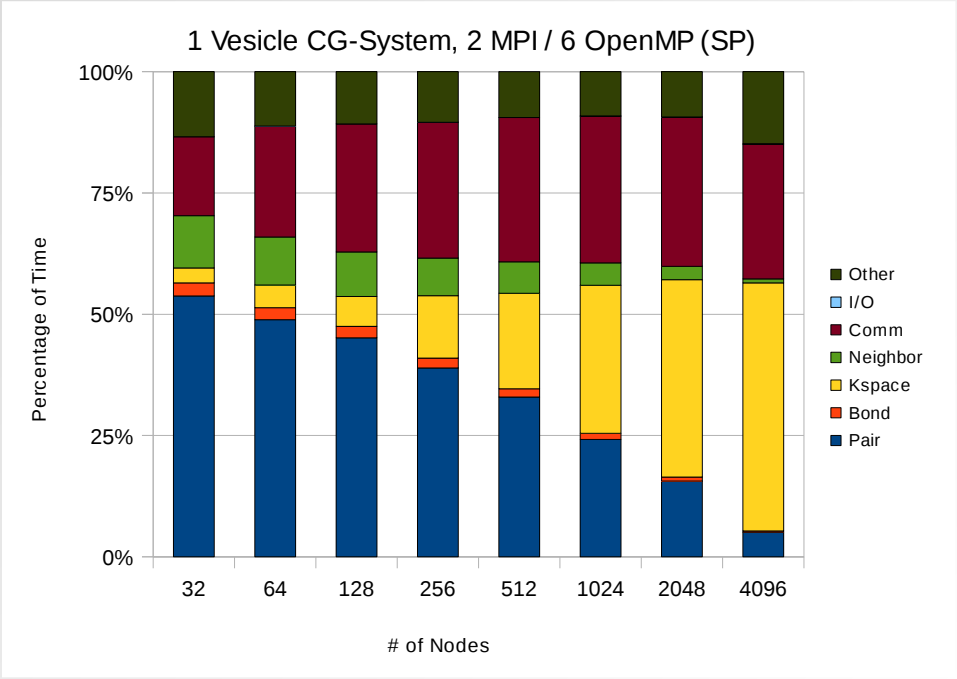
Two Types of Parallelism

- Functional parallelism: different people are performing different tasks at the same time
- Data parallelism: different people are performing the same task, but on different equivalent and independent objects

Amdahl's Law vs. Real Life

- The speedup of a parallel program is limited by the sequential fraction of the program.
- This assumes perfect scaling and no overhead



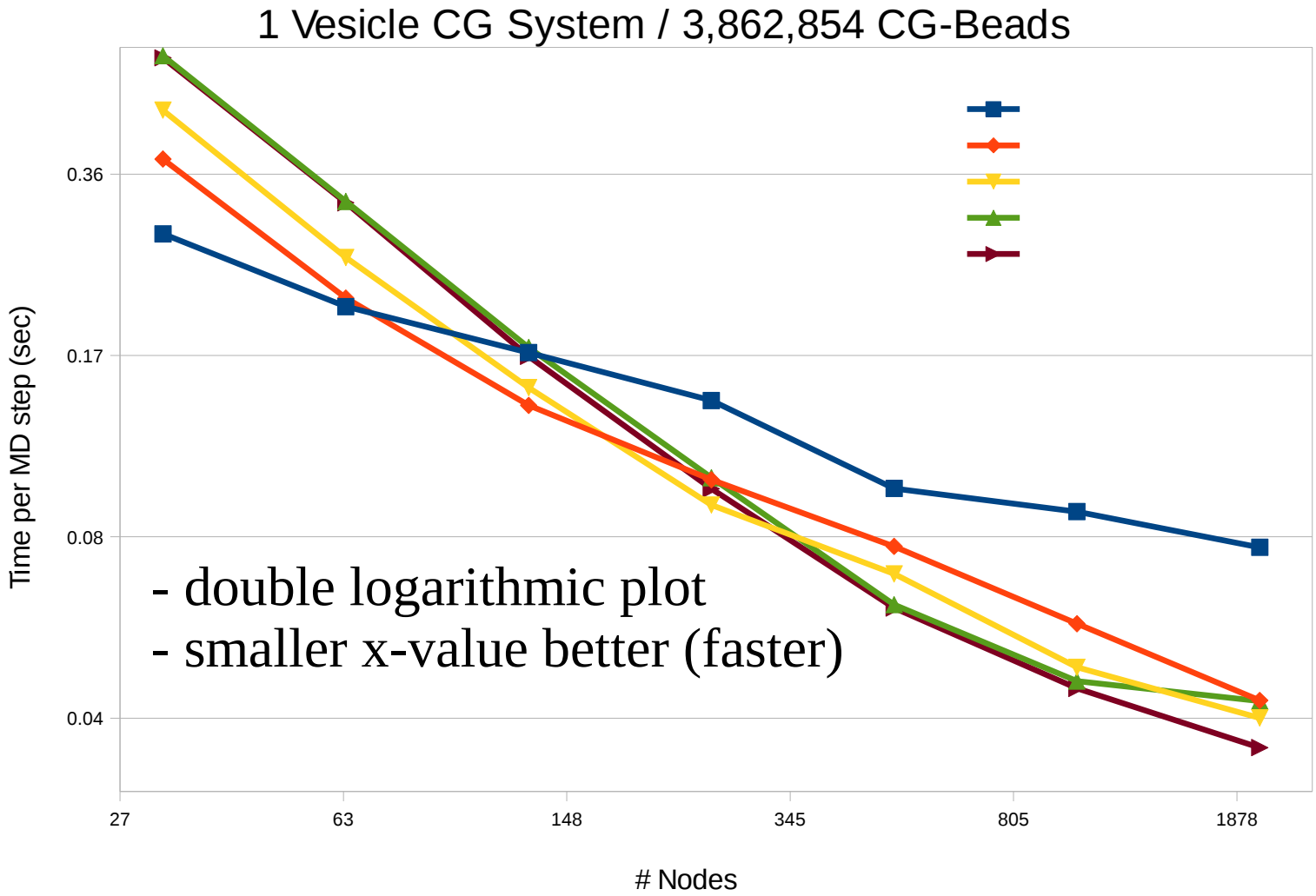


ICMS
Institute for Computational Molecular Science

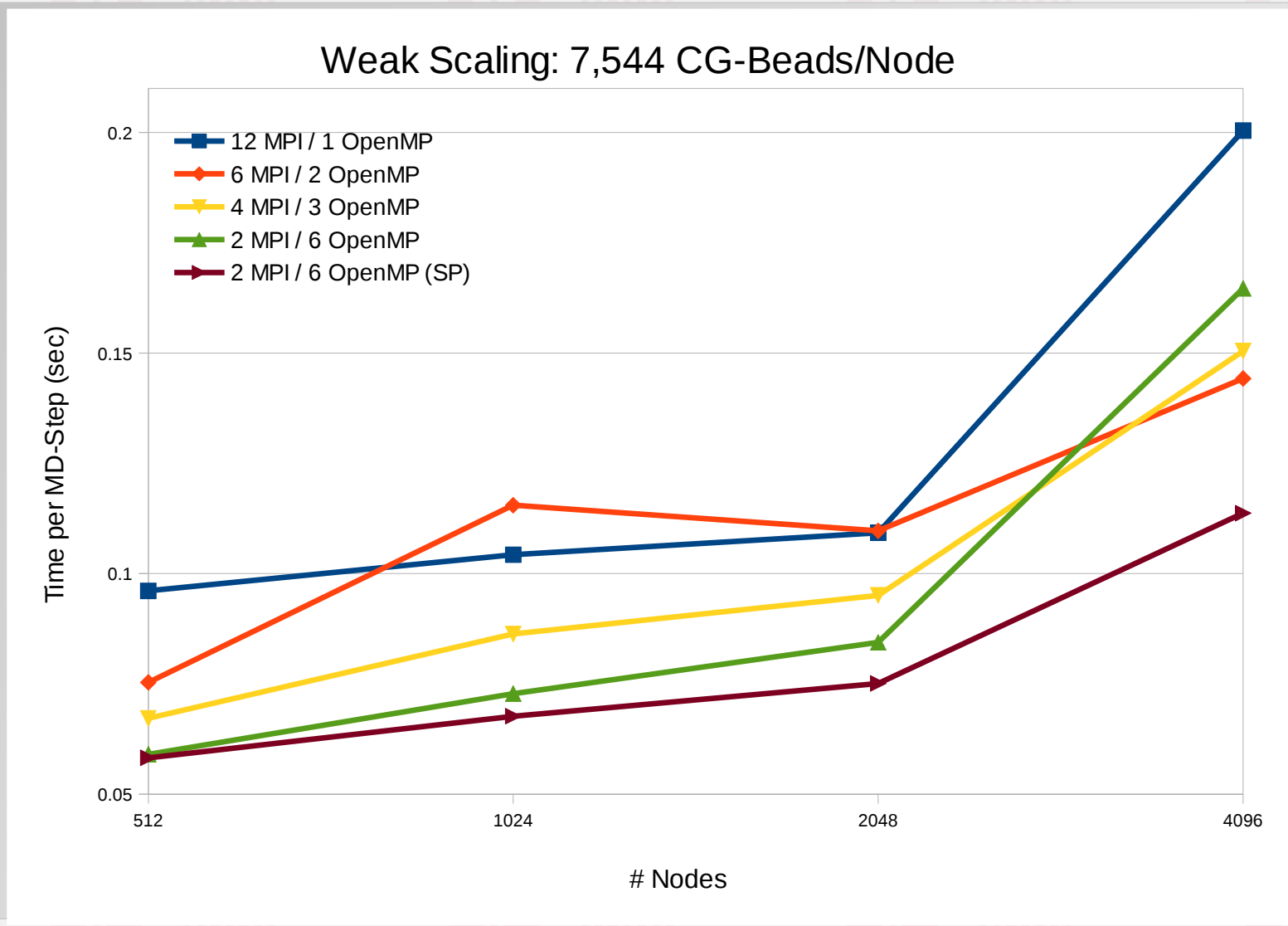
Performance of SC Applications

- Strong scaling: fixed data/problem set; measure speedup with more processors
- Weak scaling: data/problem set increases with more processors; measure if speed is same
- Linpack benchmark: weak scaling test, more efficient with more memory => 60-95% peak
- Climate modeling (WRF): strong scaling test, work distribution limited, load balancing, serial overhead => < 5% peak (similar for MD)

Strong Scaling Graph

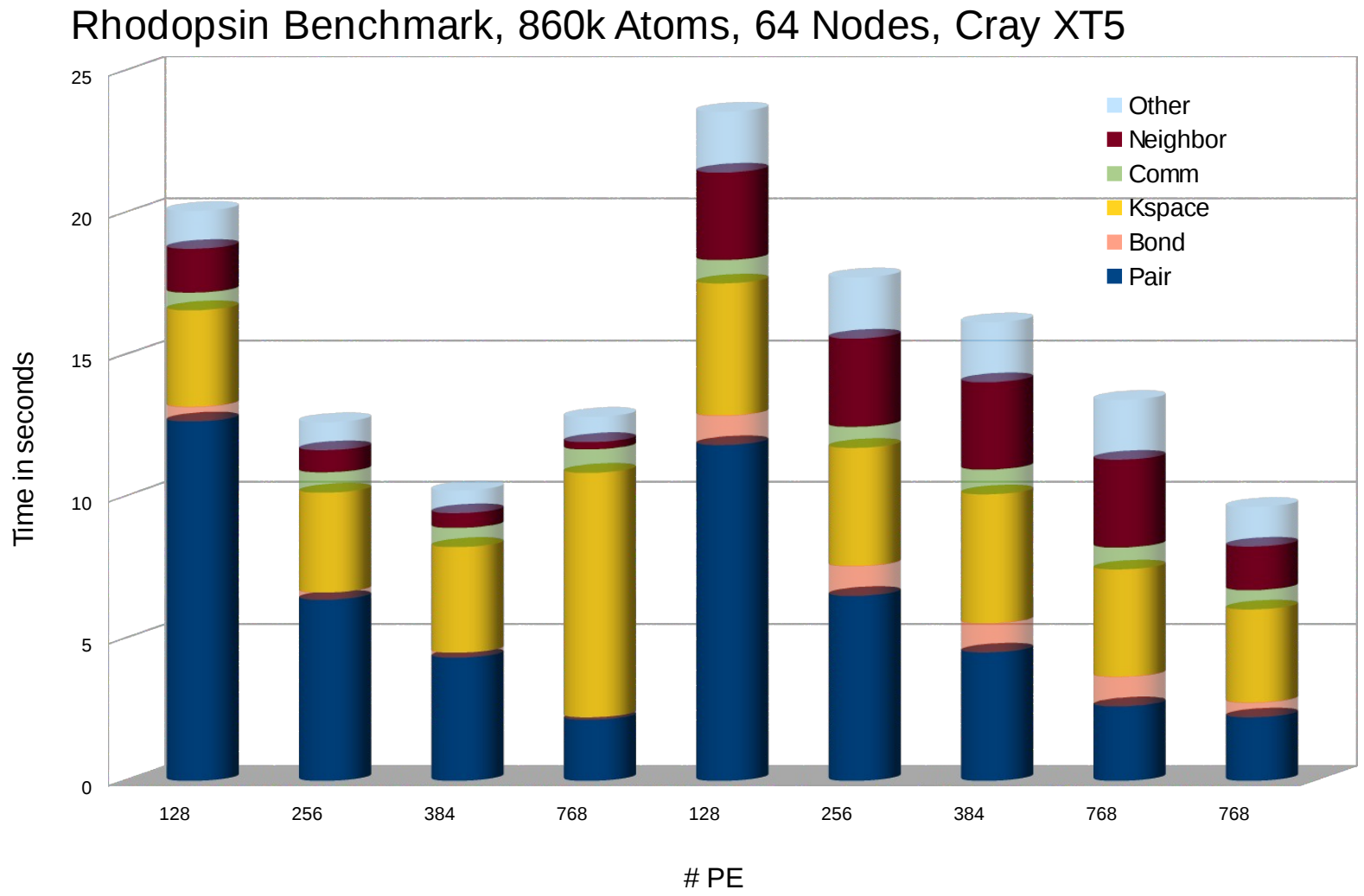


Weak Scaling Graph



ICMS
Institute for Computational Molecular Science

Performance within an Application



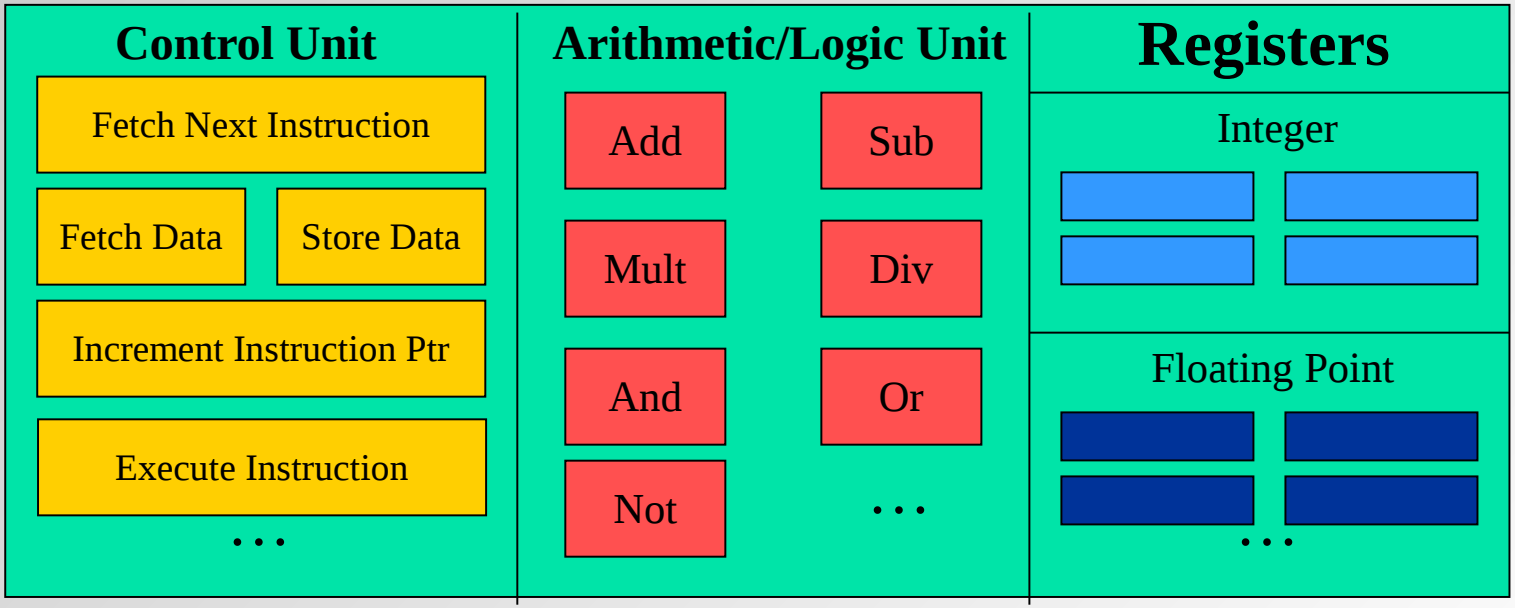
ICMS
Institute for Computational Molecular Science

A Simple Calculator

1) Enter number on keyboard => register 1
2) Turn handle forward = add backward = subtract
3) Multiply = add register 1 with shifts until register 2 is 0
4) Register 3 = result

A Simple CPU

- The basic CPU design is not much different from the mechanical calculator.
- Data still needs to be fetched into registers for the CPU to be able to operate on it.



How Many Registers?

- Minimum: 2
  > very inefficient, need many load/store ops
- 32-bit x86: 4 general purpose (integer) registers
  > more flexible. e.g. “indirect” load/store ops
  > “width” of register defines “bitness” of CPU
  > 8 floating-point registers (80-/64-/32-bit FPU)
- 64-bit x86 (AMD64,EM64T): 8 integer registers
  > same FPU as 32-bit, SIMD unit (SSE, AVX)
- IBM Power 5+: 80 general purpose registers,
  72 64-bit floating-point registers (or 36x 128-bit)

Fast and Slow Compute Operations

- Fast: add, subtract, multiply
- Medium: divide, modulus, sqrt()
- Slow: most transcendental functions
- Very slow: power (x^y for real x and y)

CPUs are often optimized for only the “fast” operations, so code will be the fastest when using only add and multiply => linear algebra

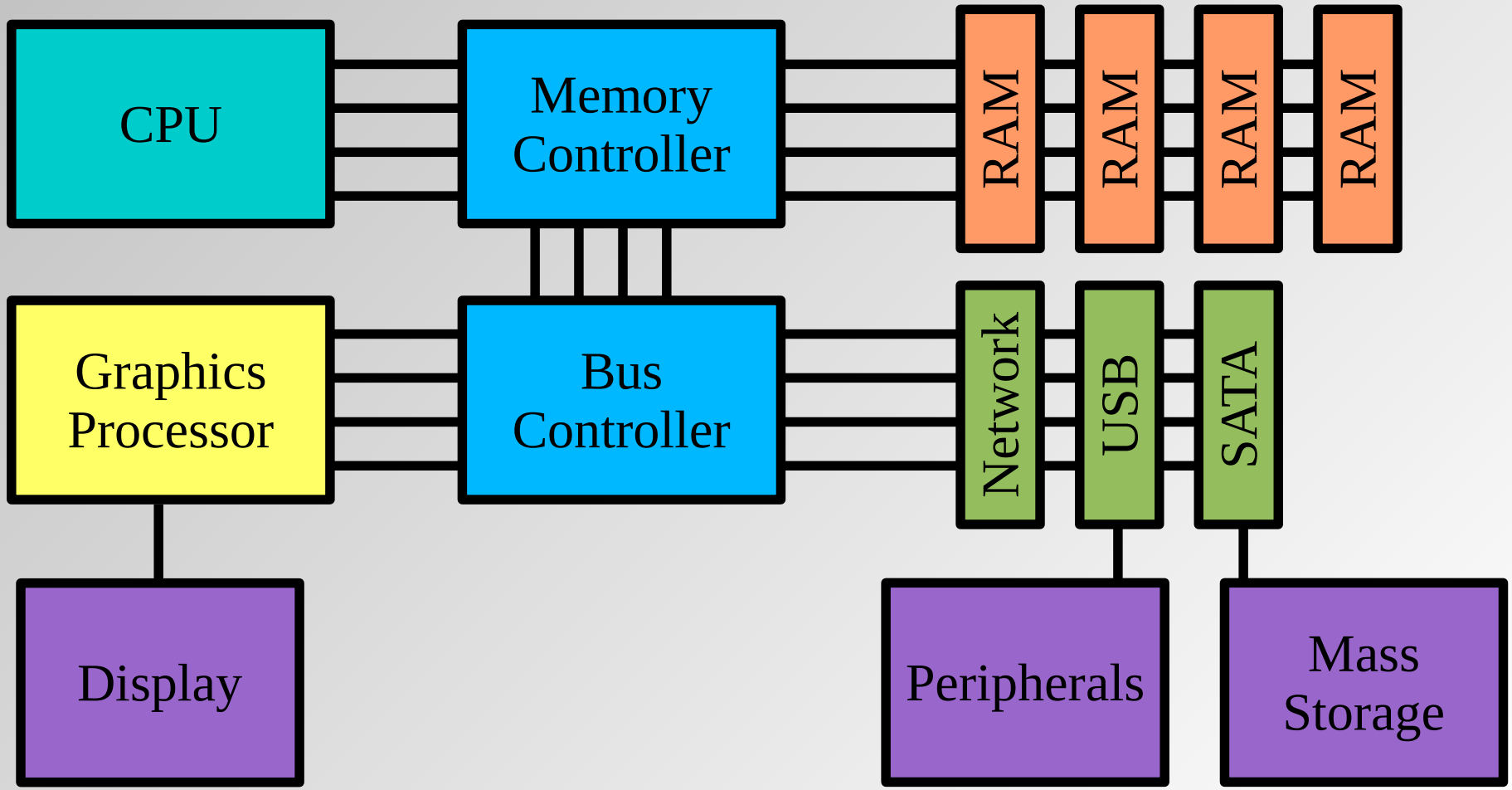
GPUs have fast approximate versions of library functions as needed by graphics (sin, cos, sqrt)

Software Optimization

- Writing maximally efficient code is hard:
  => most of the time it will not be executed exactly as programmed, not even for assembly
- Maximally efficient code is not very portable:
  => cache sizes, pipeline depth, registers, instruction set will be different between CPUs
- Compilers are smart (but not too smart!) and can do the dirty work for us, but can get fooled

  => modular programming: generic code for most of the work plus well optimized kernels

A Typical Computer



Running Faster v1: Vector CPU

- Reading data from memory (RAM) takes time => performance depends on memory latency
- Big calculations often operate on blocks of data => use registers that hold multiple numbers
- Registers are filled sequentially; arithmetic operations operate number-by-number and thus can commence before the register is full => (some) memory access latency is hidden
- Problems: data dependencies, memory speed

Running Faster v2: Cache Memory

- Registers are very fast, but very **expensive**
- Loading data from memory is slow, but is is cheap and there can be a lot of it

- => Cache memory = small buffer of fast memory between regular memory and CPU; buffers blocks of data

- Cache can come in multiple “levels”, L#: L1: fastest/smallest <-> L3: slowest/largest can be within CPU, or external

Cache vs. Vector Registers

- Cache is much cheaper to implement
- Vector processors are easier to program, particularly on large multi-dimensional data
  => weather and climate, finite element models
- Programs have to be written differently, especially for nested loops
  Vector CPU => “longest loop” as inner loop
  Scalar CPU => re-use data from cache
  => inner loop should fit into cache
  => use tiling, if inner loop too long

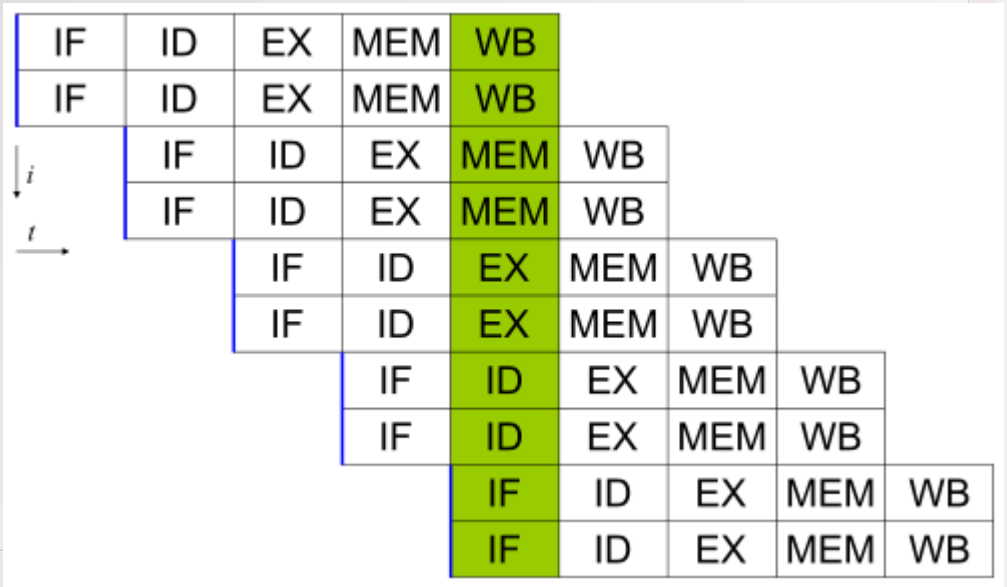
Running Faster v3: Pipelining

- Multiple steps in one CPU “operation”: e.g. fetch, decode, execute, memory, write back
  => multiple units with out-of-order execution
- Using a pipeline can improve their utilization, allows for faster clock
- Dependencies and branches can stall the pipeline
  => branch prediction
  => no “if” in inner loop

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Running Faster v4: Superscalar

- Superscalar CPU => instruction level parallelism
- Redundant functional units in single CPU
  => multiple instructions executed at same time,
  if there are no data dependencies
- Often combined with pipelined CPU design
- no branches
- Not SIMD/SSE/MMX
- Optimization:
  => loop unrolling



ICMS
Institute for Computational Molecular Science

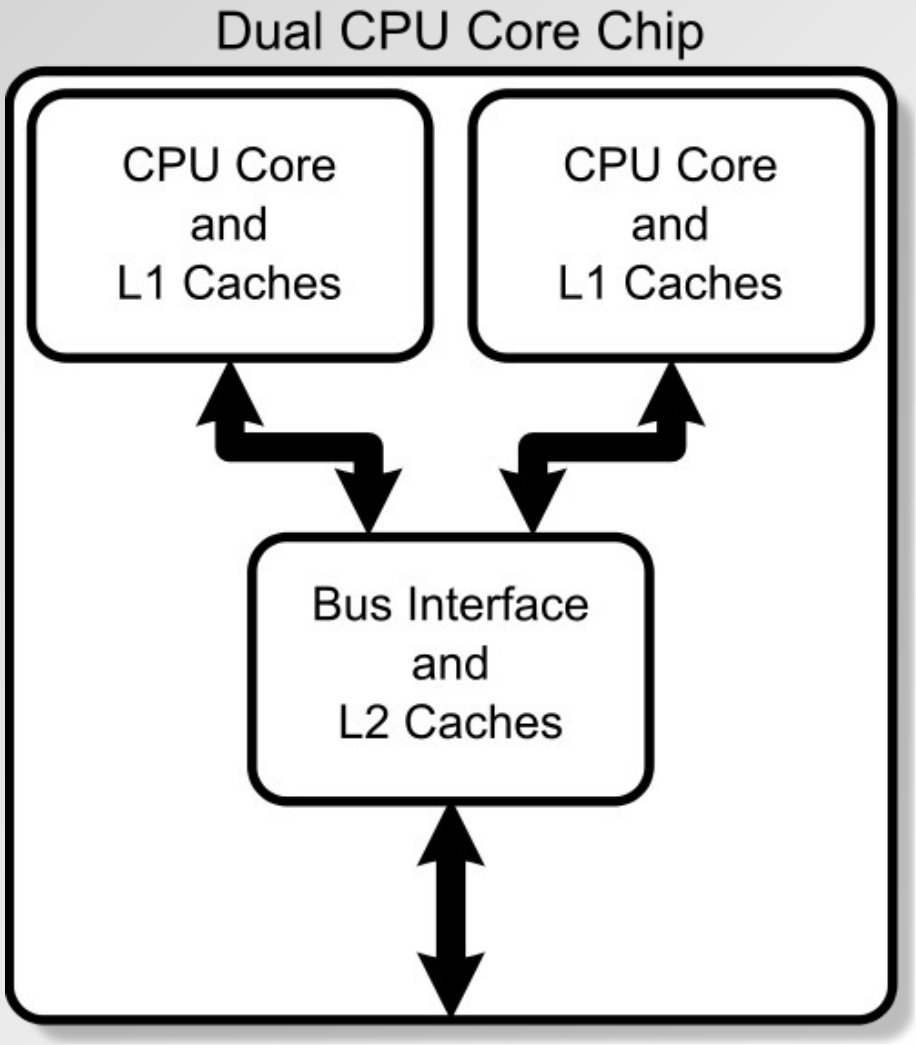
Running Faster v5: Hyperthreading

- Method to keep functional units in CPU busy
- Two sets of registers share functional units
- Improves utilization, but not individual speed
- Operating system “sees” two processors

  => added overhead for managing processes
- Performance gain application dependent
  - independent data access => cache trashing
  - applications need to use mix of functional units (load/store, integer, floating-point)

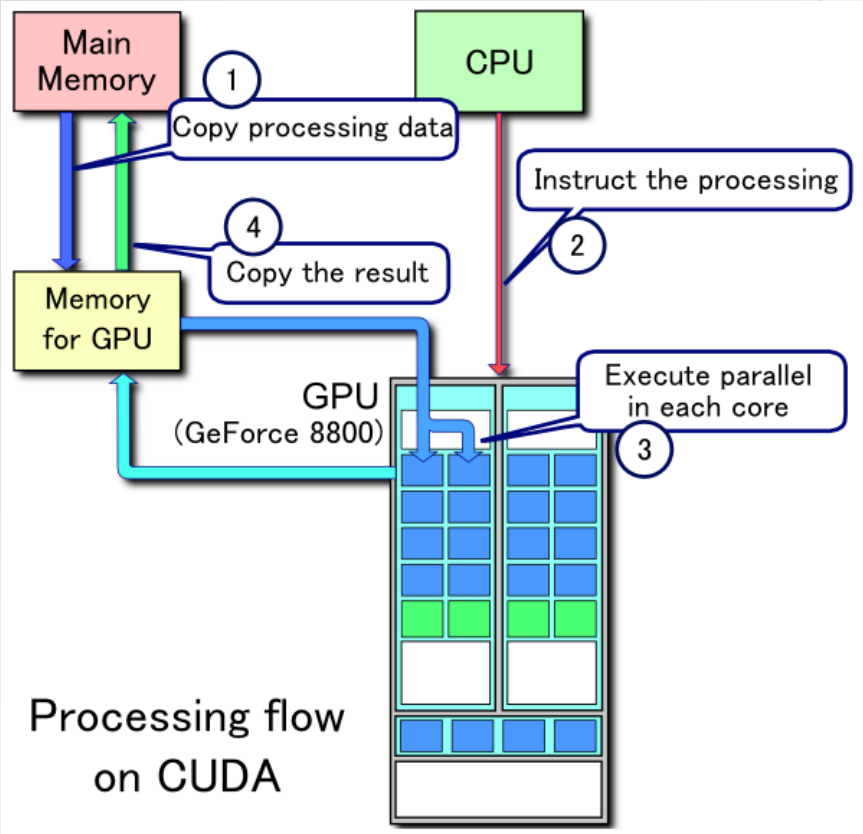
Running Faster v6: Multi-core

- Maximum CPU clock rate limited by physics
- Implement multiple complete, pipelined, and superscalar CPUs into one processor
- Need parallel software to take advantage
- Memory speed limiting



Running Faster v7: Accelerator

- Offload compute intensive or data intensive tasks to add-on card with GPU/Xeon Phi/FPGA
- Fast, wide memory bus
- Simple CPU design, but large number of cores
- SMT (hyper-threading), hardware thread manager
- Close-to-the-metal
  => no OS/kernel on GPU
  => Exception: Xeon Phi



What About Memory Access?

- Memory access faster through multi-channel memory bus (need multiple RAM modules)
  => memory itself not much faster
- Memory controller integrated in CPU
  => Multi-processor machines are NUMA
  (= non-uniform memory access)
  => total (shared memory) larger, but some memory is faster than other
- Need to use processor & memory affinity where possible for maximum efficiency (→numactl)

External Storage

- Hard drive storage has grown in capacity, but not so much in performance
  => large performance gap: RAM vs. HD
  => virtual memory (swap to disk) mostly useless
- Solid state drives combine lots of (slow) non-volatile RAM with hard drive-like interface
  => fast search times, higher transfer rate
  => “no” mechanical wear
- RAID (=redundant array of inexpensive disks) allows for parallelism and reliability

Storage Hierarchy

- Register
  integer/floating point, single/vector
- Cache
  multiple levels, shared/exclusive
- Main memory
  Local/NUMA
- External storage
  Solid state, hard drive, networked file server

Many “Levels” of CPU Hardware We Need to Worry About

- x86_64 has 16 SIMD/Vector registers (SSE => 2x DP, 4x SP; AVX => 4x DP, 8x SP; Xeon Phi 8x DP, 16x SP; ...)
- 2-3 Cache levels, L1 is per core, higher levels shared between varying amounts of cores
- Hybrid hyper-threading (some functional units are shared between two cores, others not)
- NUMA for multi-core, multi-processor machines
- Hybrid hardware (CPU/GPU hybrids)

How to Optimize For All of This?

- Vector registers: compiler auto-vectorization (plus directives), vector intrinsics, libraries
  -> loops without data dependencies & branches
  -> struct of arrays instead of array of structs
- Caches: maximize data reuse
  -> tiling, short inner loops, data reorganization
- Superscalar, pipelined architectures
  -> predictable data flows, concurrent execution
- Multi-core, NUMA: multi-level parallelism with shared and distributed/replicated data as needed

Memory Mountain

- Memory performance with “strided” access stride = distance between two data locations
- Shows cache sizes and performances
- Stride 1 best
- Try with and without compiler optimization => prefetch instruction, vectorization