Modern Computer Architectures

Dr. Axel Kohlmeyer

Senior Scientific Computing Expert

Information and Telecommunication Section
The Abdus Salam International Centre
for Theoretical Physics

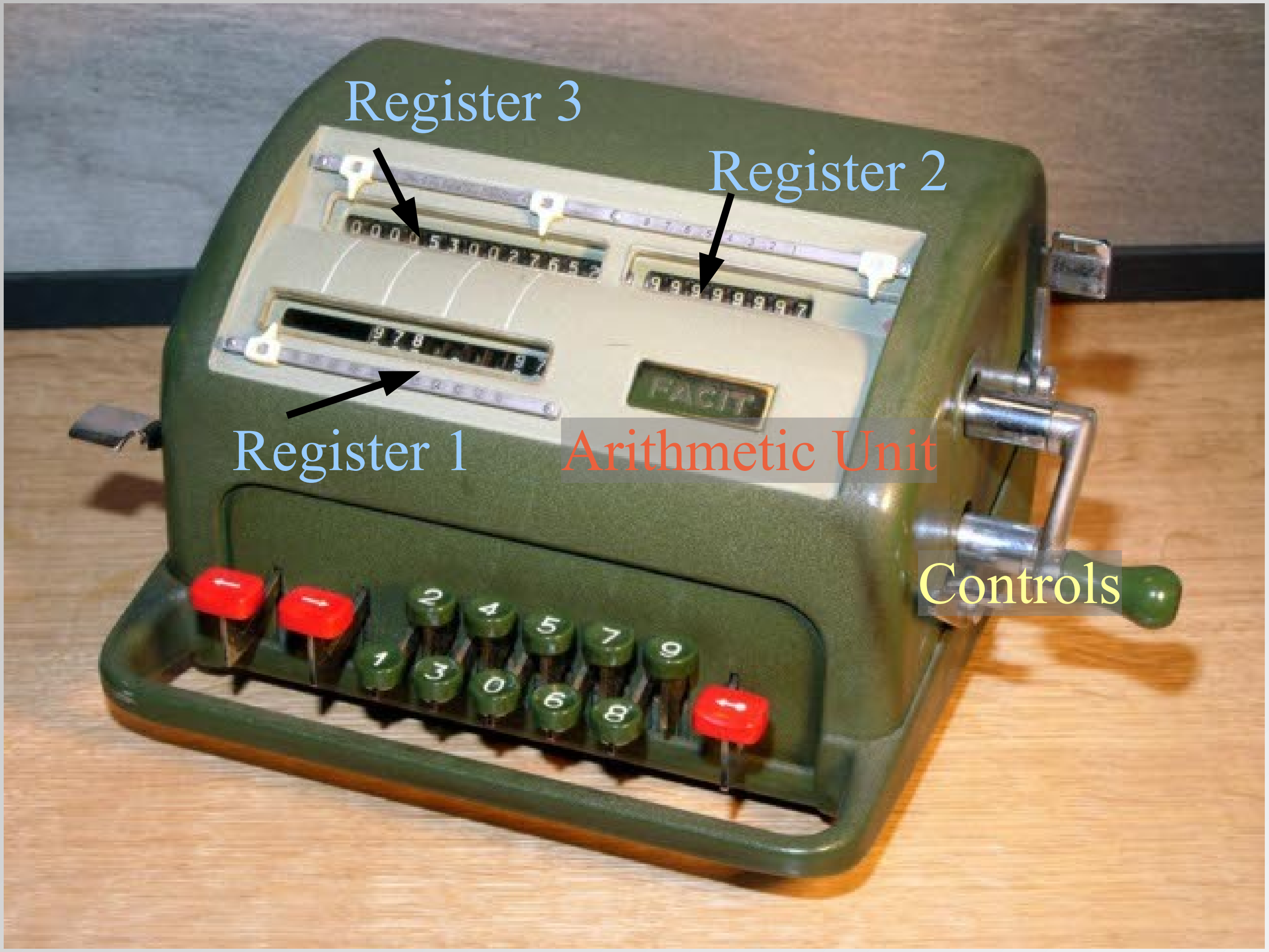
http://sites.google.com/site/akohlmey/

akohlmey@ictp.it

The Abdus Salam
International Centre
for Theoretical Physics

Workshop on Computer Programming and
Advanced Tools for Scientific Research Work

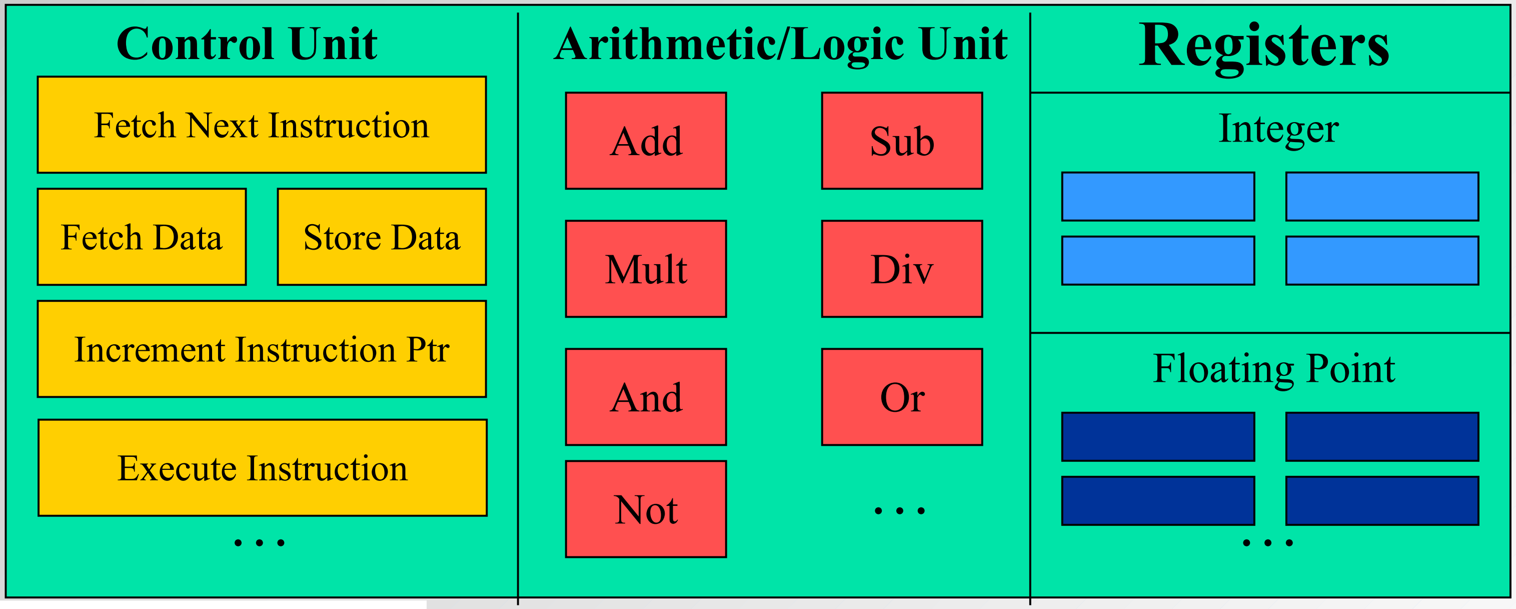
A Simple Calculator



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

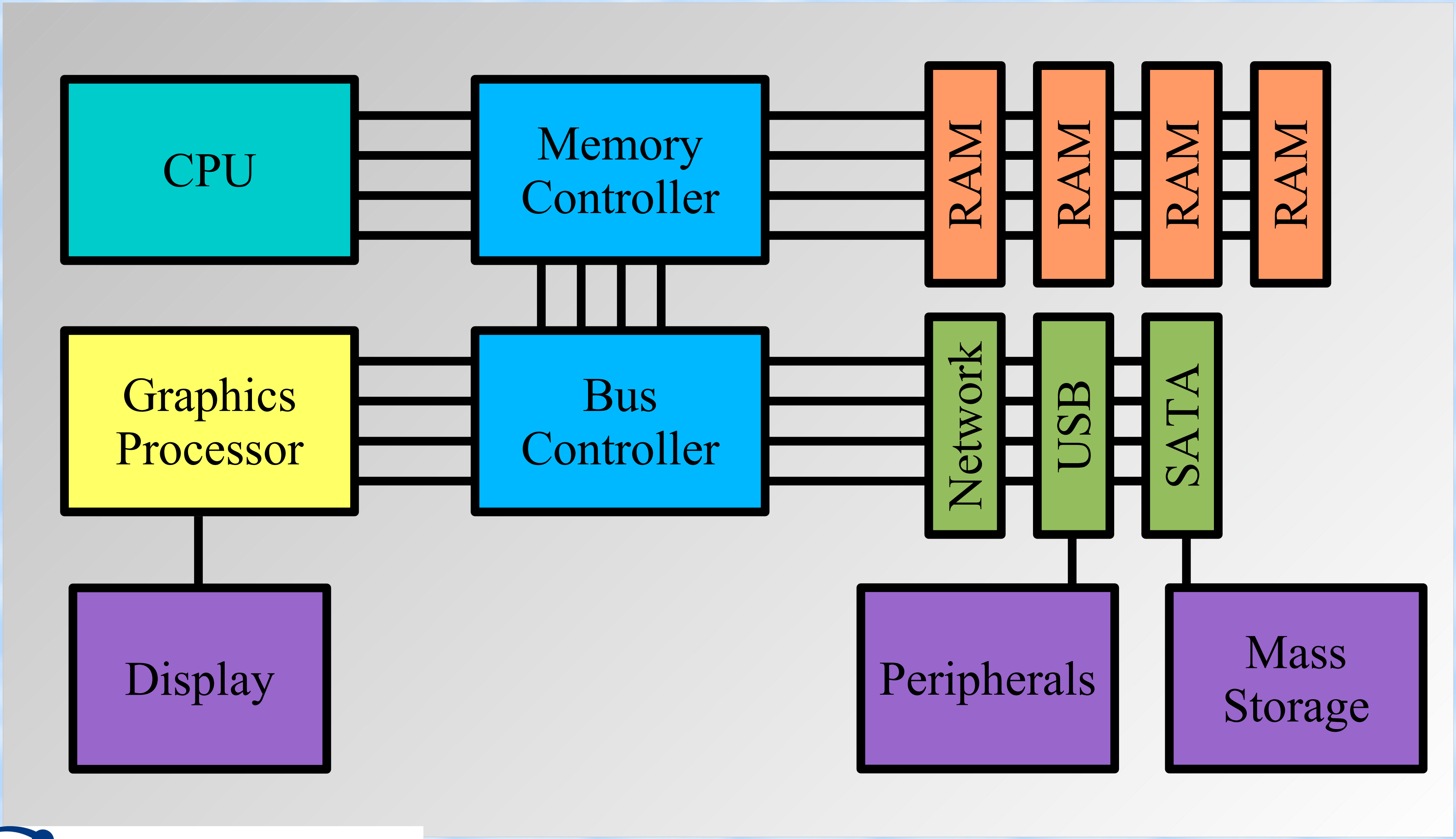
- The basic CPU design is not much different from the mechanical calculator.
  - Data still needs to be fetched into registers first
  - Different operations take different effort to complete



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 (SSE2 etc.)
- IBM Power 5+: 80 general purpose registers, 72 64-bit floating-point registers (or 36x 128-bit)

A Typical Computer

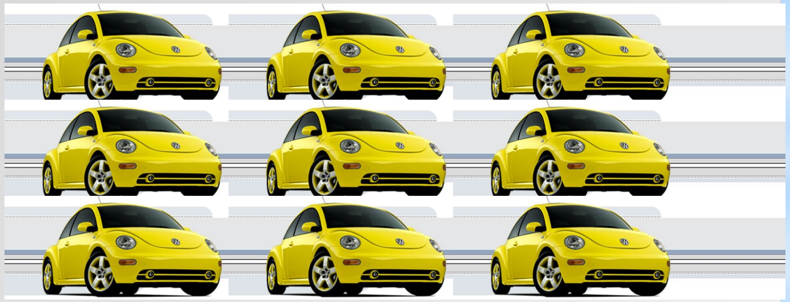


Running Faster v1: Vector CPU

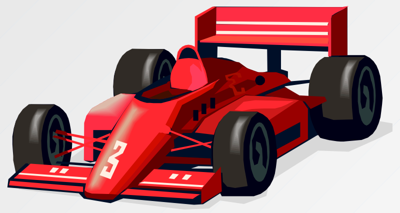
- Reading data from memory (RAM) takes time => performance depends on memory latency
- Typical problems operate on blocks of data => use registers that can hold blocks numbers
- Registers are filled sequentially; arithmetic operations operate number-by-number and thus can start before the register fetch is complete => memory 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
  Vector CPU => “longest loop” as inner loop
  Scalar CPU => re-use data from cache
  => “shortest loop” as inner loop
  => tiling, if inner loop too long

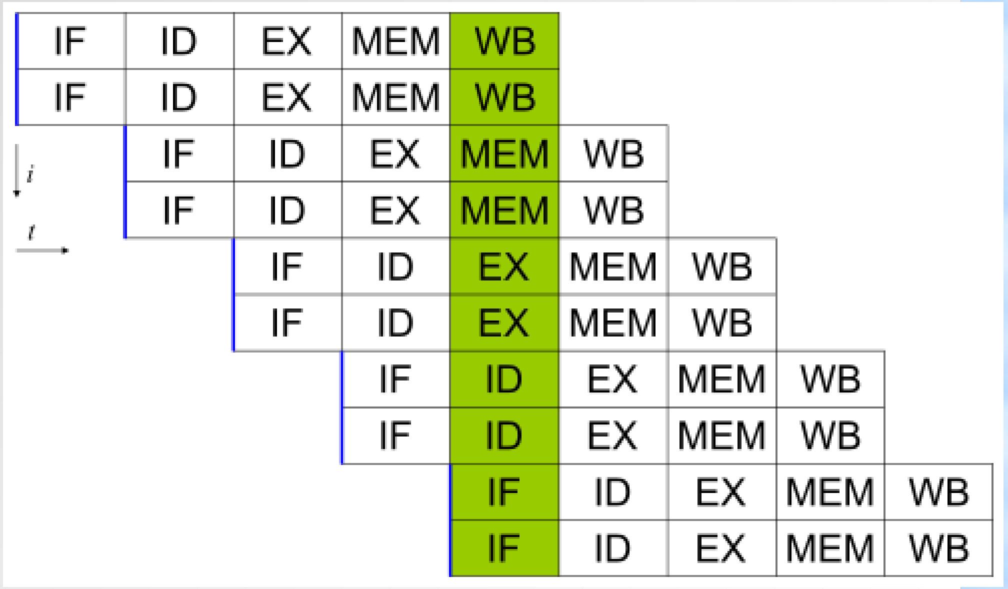
Running Faster v3: Pipelining

- Multiple steps in one CPU “operation”:
  fetch, decode, execute, memory, write back
  => multiple functional units
- 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
- Often combined with pipelined CPU design
- No data dependencies, no branches
- Not SIMD/SSE/MMX
- Optimization:
  => loop unrolling



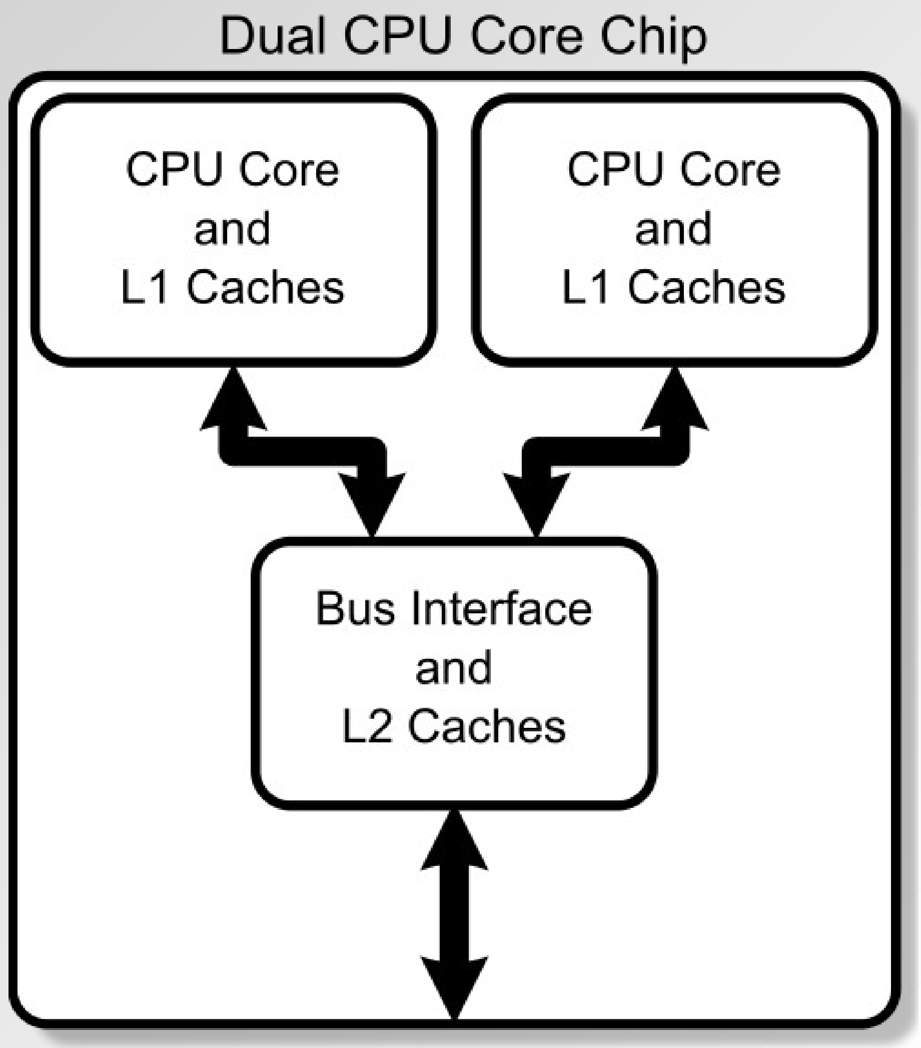
IF	ID	EX	MEM	WB				
IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB
				IF	ID	EX	MEM	WB

Running Faster v5: Hyperthreading

- Method to keep functional units in CPU busy
- Two sets of registers share functional units
- Hardware manages access transparently
- Operating system “sees” two processors
- Performance gain application dependent
  - scheduling overhead for 2x # processors
  - independent data access => cache trashing
  - applications need to use mix of functional units (load/store, integer, floating-point)

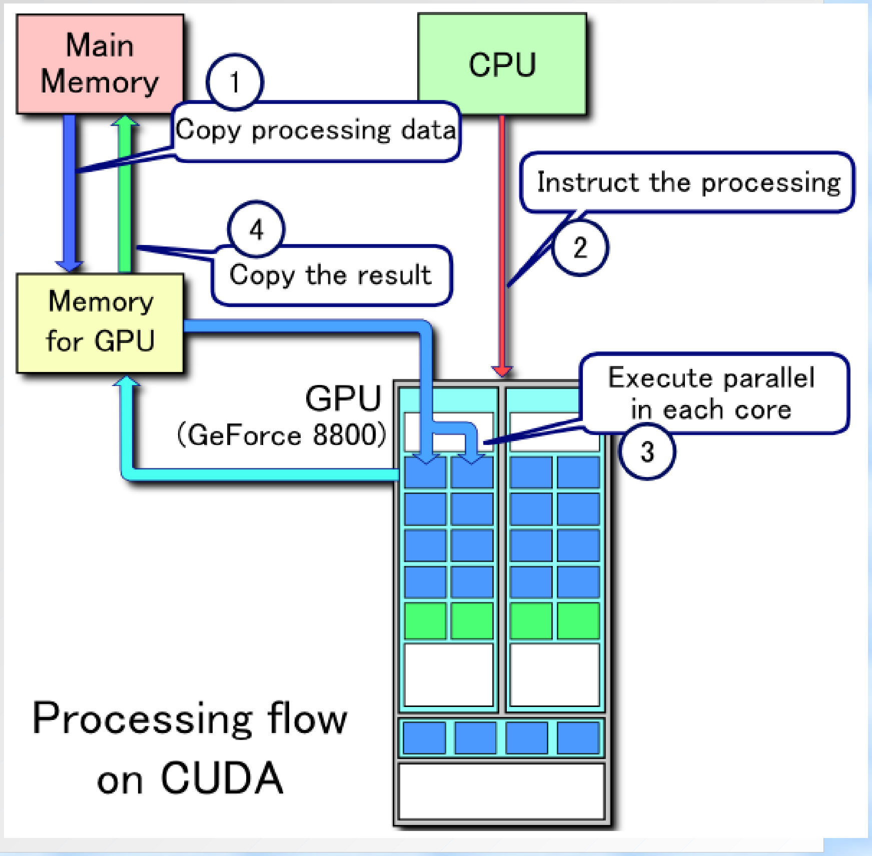
Running Faster v6: Multi-core

- 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: GPGPU

- Offload compute intensive or data intensive tasks to add-on card with GPGPU
- Fast, wide memory bus
- Single precision FP
  fast FP math ops
- SMT (hyper-threading)
  hardware thread manager
- Close-to-the-metal
  => no OS/kernel on GPU



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
- Use processor and memory affinity in OS kernel when possible for maximum efficiency

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

What is Coming Down the Line?

- Hybrid CPUs, new architectures, low power
- “System on a Chip” type hardware
- Smartphones as commodity hardware platform
- Virtualization and “reverse virtualization”
- Remote data access will be a fast/slow as local
- We are drowning in data (streaming to analysis immediately instead of post processing)

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 (they still do fail)
- RAID (=redundant array of inexpensive disks) allows for parallelism or reliability or both

Storage Hierarchy

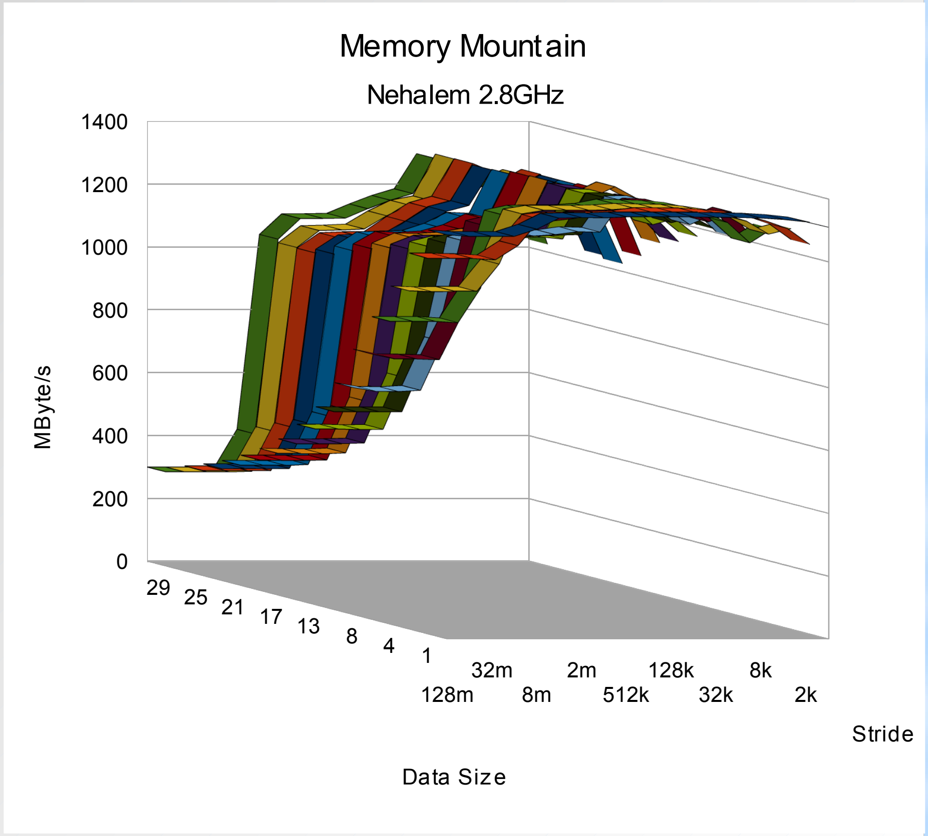
- Register
  integer/floating point, single/vector
- Cache
  multiple levels, shared/exclusive
- Main memory
  Local/NUMA
- External storage
  Solid state, hard drive, networked file server

Conclusions for Software

- It is getting complicated!
  => multiple combined strategies for faster processing means that a failure at one of them will limit the overall performance
- Follow the “dance of the data”, keep data “local”
- Multi-level parallelism, fine grained (SIMD, threading) and coarse grained (MPI), required
- Compiler technology can help, but not for all
- Optimized libraries; domain specific languages

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


Memory Mountain
Nehalem 2.8GHz
MByte/s
32m
2m
128k
8k
128m
8m
512k
32k
2k
Stride
Data Size