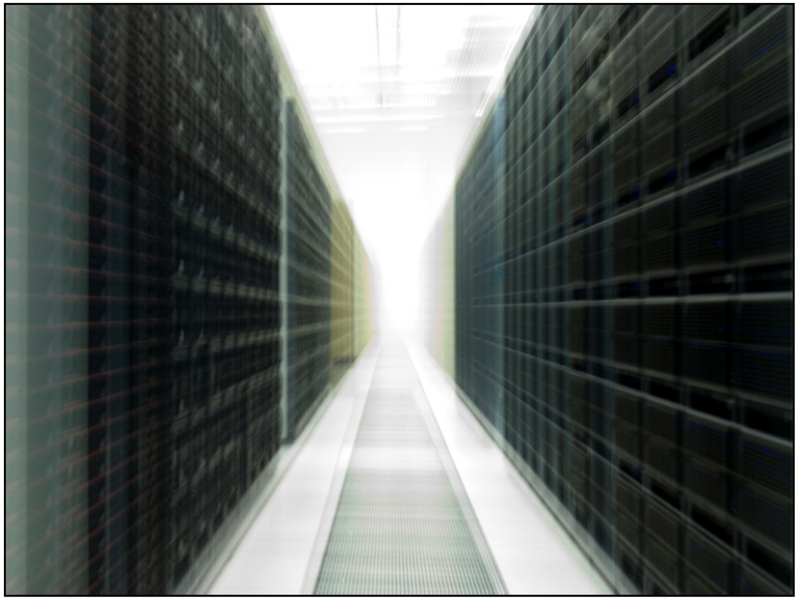
Second INFN International School on Architectures, tools and methodologies for developing efficient large scale scientific computing applications

Ce.U.B. – Bertinoro – Italy, 24 – 29 October 2011

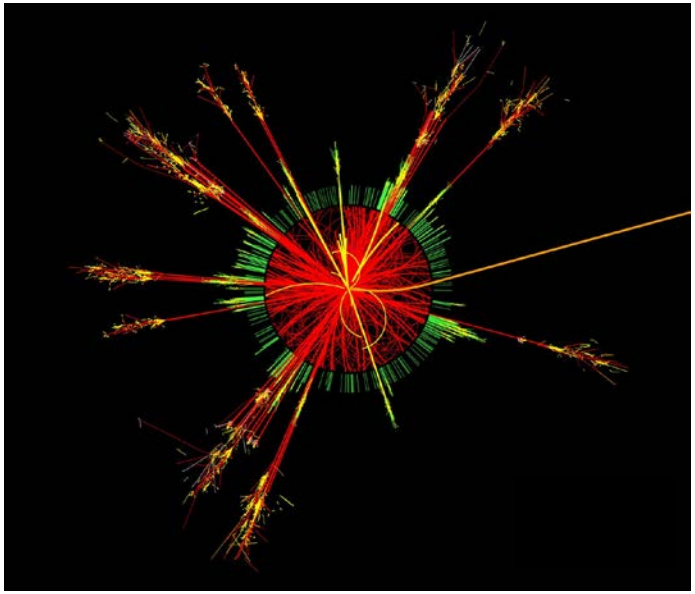
Efficient use of modern CPU architectures

“The 7 dimensions of performance”



Sverre Jarp

CERN
openlab
CTO



Bertinoro, 24 October 2011

Contents

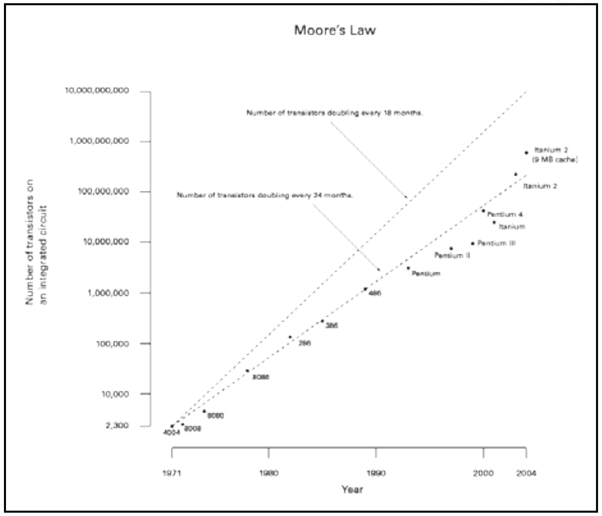
- **The driving force: Moore's law**
- **Review some fundamental architectural principles**
- **Define performance "dimensions"**
- **Scaling "inside-a-core":**
  - First 3 dimensions
  - Causes of execution delays
  - Performance metrics
- **Scaling "across-cores"**
  - Next set of dimensions
  - Parallel programming paradigms
  - Achieving better memory footprints
  - C++ parallelization support
  - Example of parallelization: Track fitting and others
- **Conclusions**

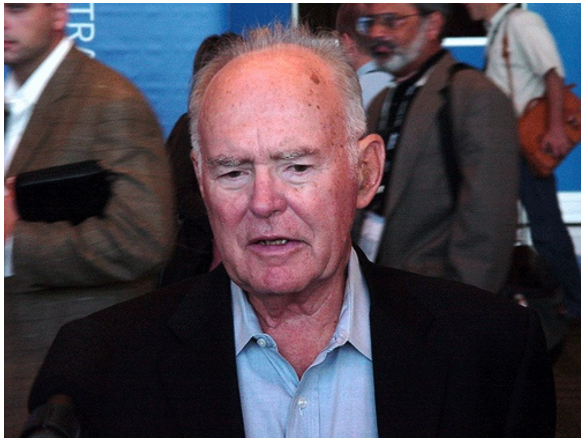
Moore’s law

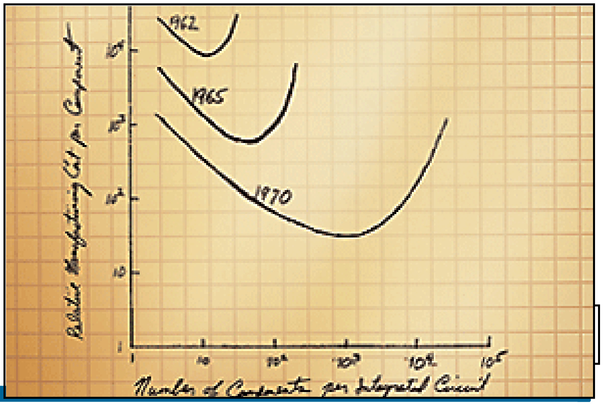
- **We continue to double the number of transistors every other year**

- **The consequences:**
  - CPUs
    - Single core → Multicore → Manycore
    - Hardware vectors
    - Hardware threading
  - GPUs
    - Huge number of FMA units

- **Today, we commonly acquire chips with 1’000’000’000 transistors!**






Relative Manufacturing Cost per Component
1962
1965
1970
1
10
10²
10³
10⁴
Number of Components per Integrated Circuit
1
10
10²
10³
10⁴
10⁵

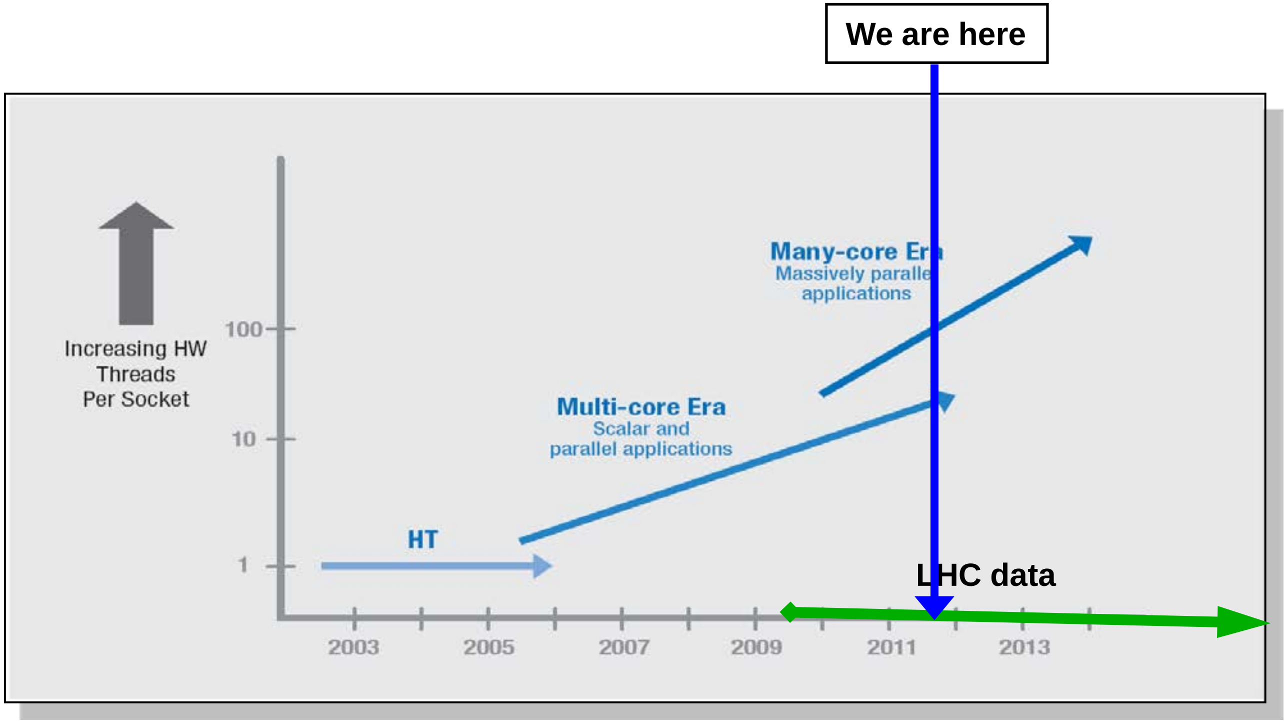
Real consequence of Moore’s law

- **We are being “drowned” by transistors:**
  - More (and more complex) execution units
    - Hundreds of new instructions
  - Longer SIMD/SSE hardware vectors
  - More and more cores
  - More hardware threading
- **In order to profit we need to “think parallel”**
  - Data parallelism
  - Task parallelism

“Intel platform 2015” (and beyond)

- **Today’s silicon processes:**
  - 45, 32 nm
- **On the roadmap:**
  - 22 nm (2011/12)
  - 15 nm (2013/14)
- **In research:**
  - 10 nm (2015/16)
  - 7 nm (2017/18)
  - 5 nm (2019/20)
    - Source: Shekhar Borkar/Intel

We are here



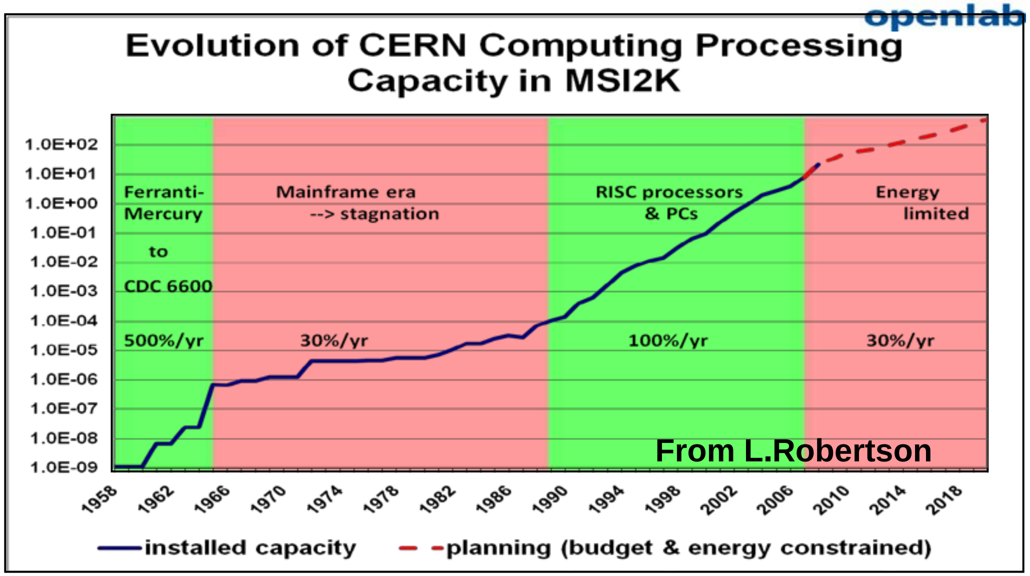
S. Borkar et al. (Intel), "Platform 2015: Intel Platform Evolution for the Next Decade", 2005.

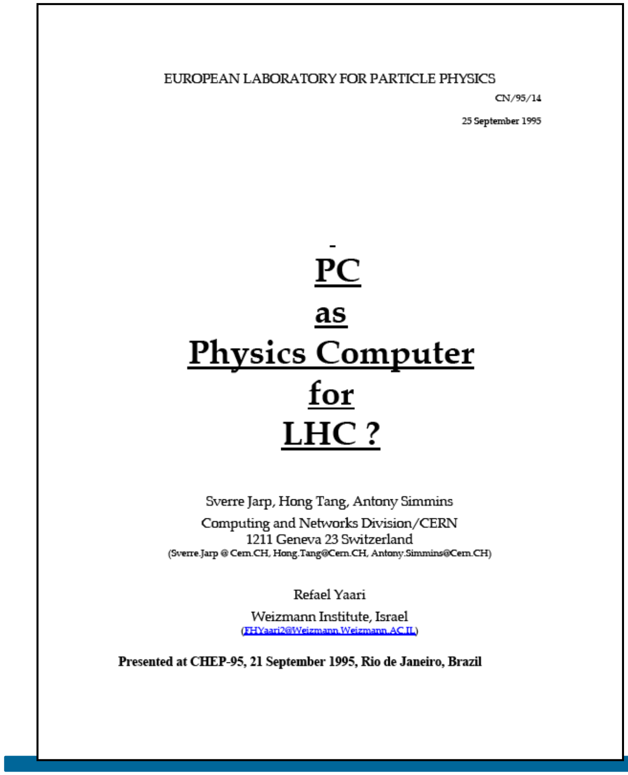
- **Each generation will push the core count:**
  - We are entering the many-core era (whether we like it or not) !

Evolution of CERN's computing capacity

- **During the LEP era (1989 – 2000):**
  - Doubling of total computing capacity every year
  - Initiated with the move from mainframes to RISC systems

- **The PC has been with us for 15 years!**
  - At CHEP-95 I made the first recommendation to move to PCs
    - After a set of encouraging benchmark results





EUROPEAN LABORATORY FOR PARTICLE PHYSICS

CN/95/14

25 September 1995

PC
as
Physics Computer
for
LHC ?

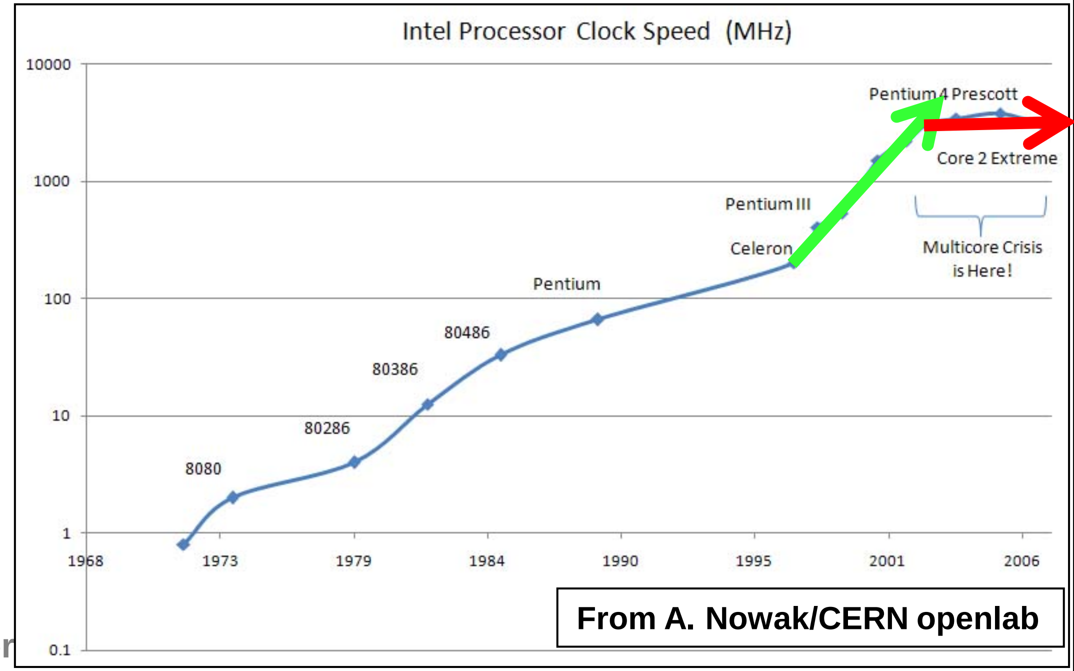
Sverre Jarp, Hong Tang, Antony Simmins
Computing and Networks Division/CERN
1211 Geneva 23 Switzerland
(Sverre.Jarp @ Cern.CH, Hong.Tang@Cern.CH, Antony.Simmins@Cern.CH)

Refael Yaari
Weizmann Institute, Israel
(FHYaari2@Weizmann.Weizmann.AC.IL)

Presented at CHEP-95, 21 September 1995, Rio de Janeiro, Brazil

Frequency scaling

- **The 7 “fat” years of <u>easy</u> frequency scaling in HEP**
  - The Pentium Pro in 1996: 150 MHz
  - The Pentium 4 in 2003: 3.8 GHz (~25x)
- **Since then**
  - Core 2 systems:
    - ~3 GHz
    - Multi-core
- **Recent CERN purchase:**
  - Intel Xeon L5640 CPUs
    - 2.26 GHz



From A. Nowak/CERN openlab

The holy grail: Forward scalability

- **Not only should a program be written in such a way that it extracts maximum performance from today’s hardware**
- **On future processors, performance should scale automatically**
  - In the worst case, one would have to recompile or relink
- **Additional CPU/GPU hardware, be it cores/threads or vectors, would automatically be put to good use**
- **Scaling would be as expected:**
  - If the number of cores (or the vector size) doubled:
    - Scaling would be close to 2x, but certainly not just a few percent
- **We cannot afford to “rewrite” our software for every hardware change!**

Performance: A complicated story!

- **We start with a concrete, real-life problem to solve**
  - For instance, simulate the passage of elementary particles through matter

- **We write programs in high level languages**
  - C++, JAVA, Python, etc.

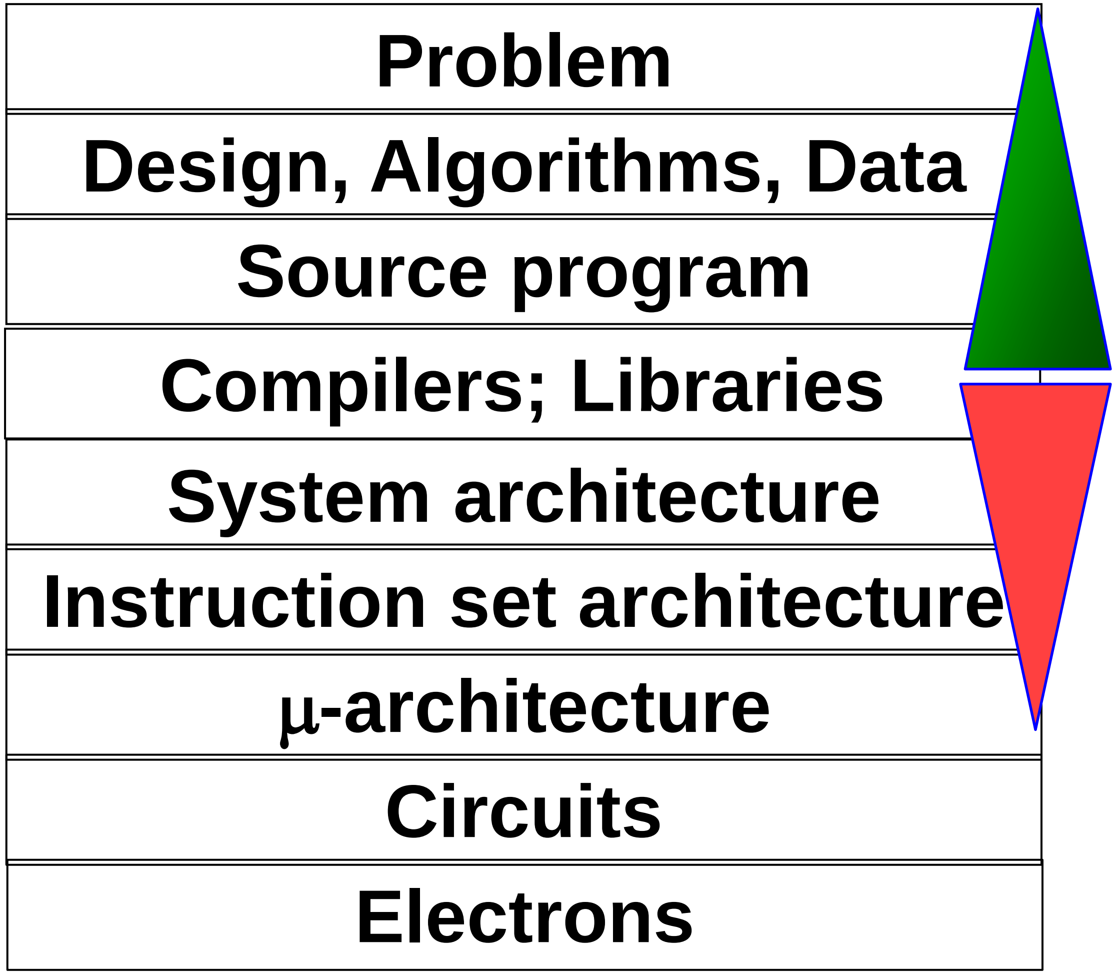
- **A compiler (or an interpreter) transforms the high-level code to machine-level code**

- **We link in external libraries**

- **A sophisticated processor with a complex architecture and even more complex micro-architecture executes the code**

- **In most cases, we have little (or no) clue as to the efficiency of this transformation process**

A Complicated Story (in layers!)



Problem

Design, Algorithms, Data

Source program

Compilers; Libraries

System architecture

Instruction set architecture

μ-architecture

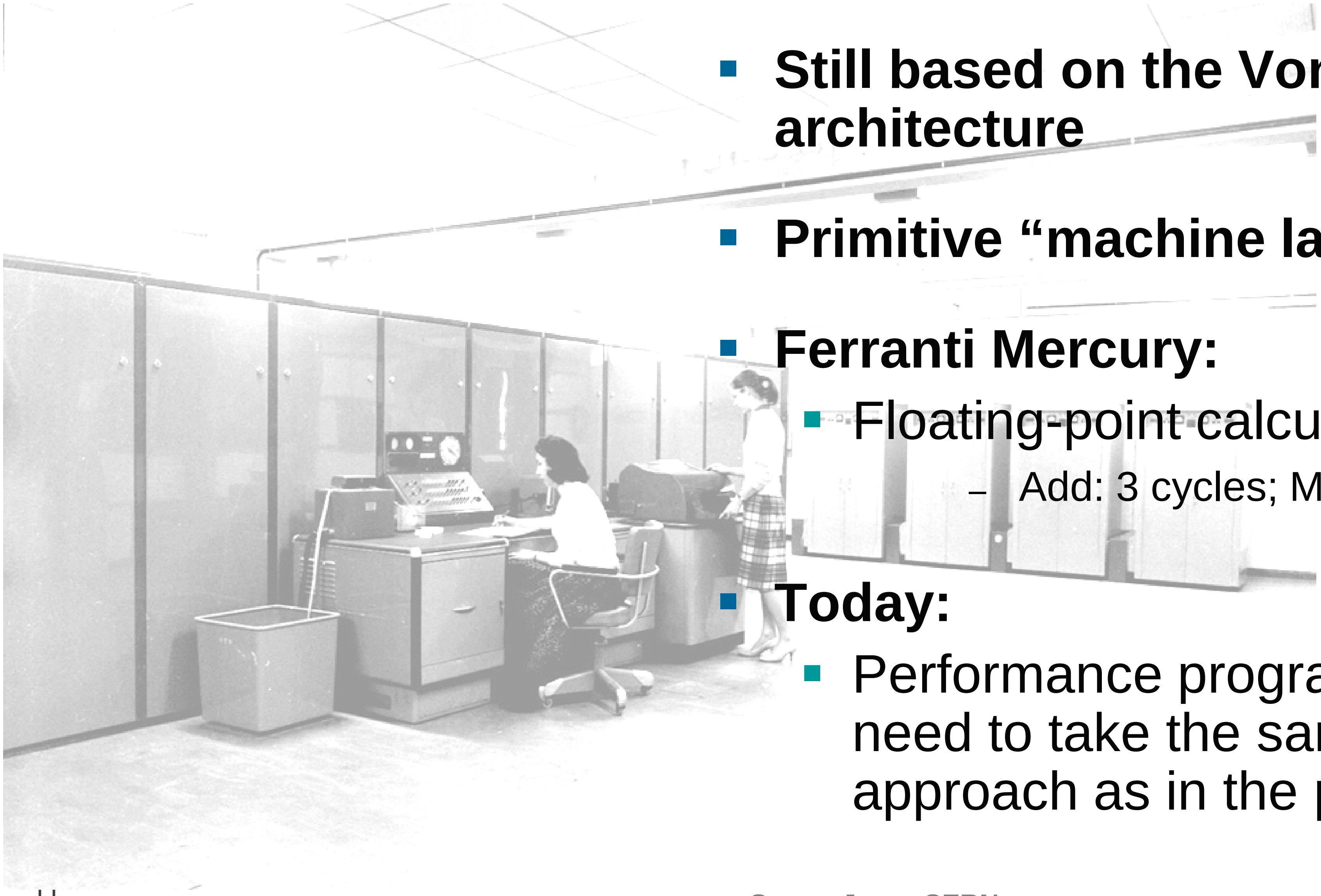
Circuits

Electrons

- **We must avoid being fenced into a single layer!**

Adapted from Y.Patt, U-Austin

Archaic CPUs



- **As “stupid” as 50 years ago**
- **Still based on the Von Neumann architecture**
- **Primitive “machine language”**
- **Ferranti Mercury:**
  - Floating-point calculations
    - Add: 3 cycles; Multiply: 5 cycles
- **Today:**
  - Performance programmers need to take the same approach as in the past

And the spoken language is

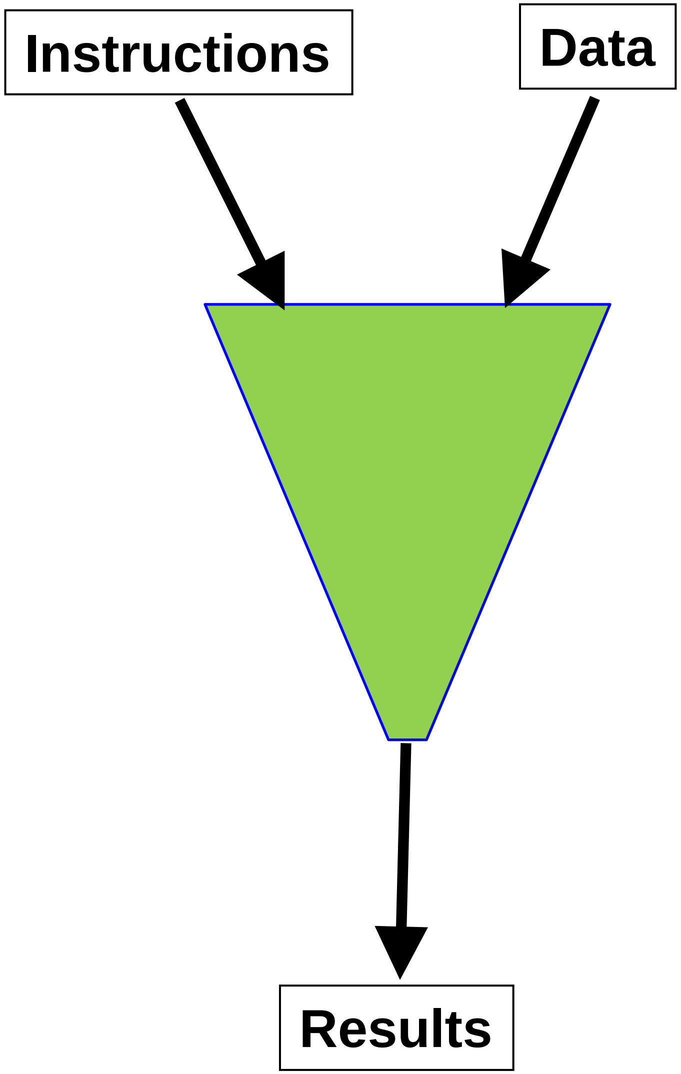
- **Assembly/machine code!**

```
..B1.31:                      # Preds ..B1.31 ..B1.30         # Infreq
        movsd     (%rsp), %xmm3                                 #94.17
        lea       (%rbx,%rbx,2), %rcx                           #94.36
        movsd     (%rsi,%rcx,8), %xmm2                          #94.40
        incl      %eax                                          #93.42
        movsd     8(%rsi,%rcx,8), %xmm0                         #94.40
        cmpl      %edx, %eax                                    #93.39
        mulsd     %xmm2, %xmm2                                  #94.40
        mulsd     %xmm0, %xmm0                                  #94.40
        movsd     16(%rsi,%rcx,8), %xmm1                        #94.40
        addsd     %xmm0, %xmm2                                  #94.40
        mulsd     %xmm1, %xmm1                                  #94.40
        movl      %eax, %ebx                                    #93.42
        addsd     %xmm1, %xmm2                                  #94.40
        sqrtsd    %xmm2, %xmm2                                  #94.40
        addsd     %xmm2, %xmm3                                  #94.17
        movsd     %xmm3, (%rsp)                                 #94.17
        jb        ..B1.31       # Prob 82%                      #93.39
```

But, let’s start with the basics!

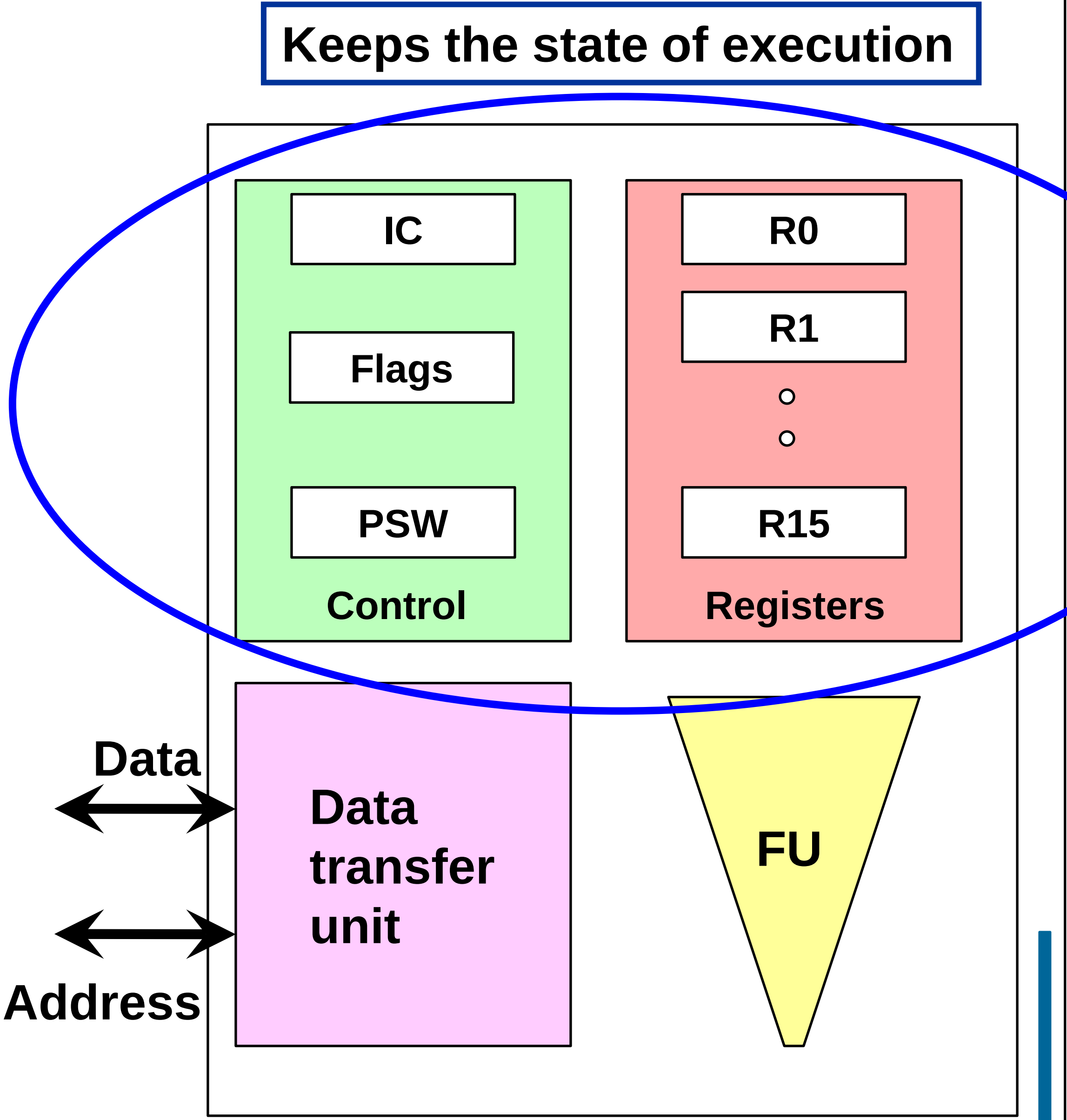
Von Neumann architecture

- **From Wikipedia:**
  - The von Neumann architecture is a computer design model that uses a processing unit and a single (separate) storage structure to hold both instructions and data.
- **It can be viewed as an entity into which one streams instructions and data in order to produce results**
- **Our goal is to produce results as fast as possible**



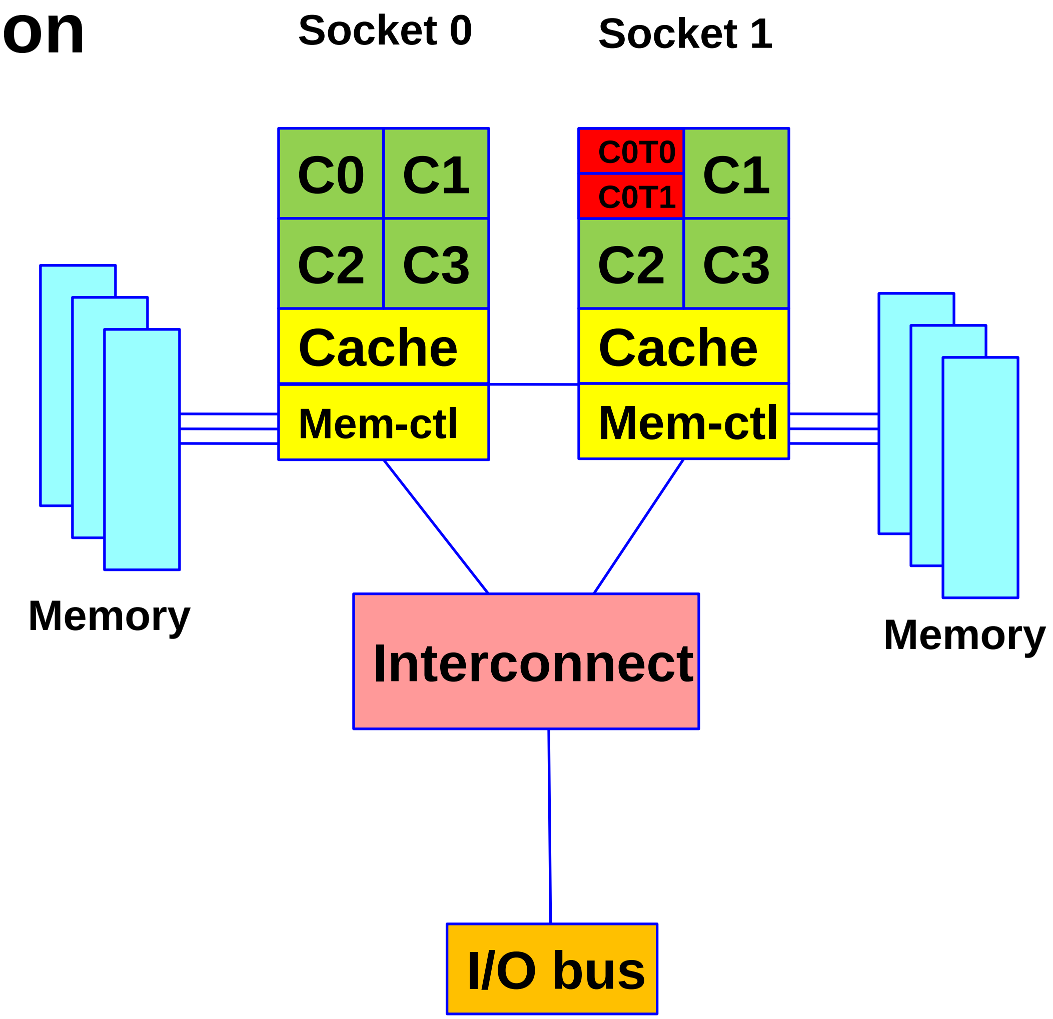
Simple CPU layout

- **A simple processor with four key components:**
  - Control Logic
    - Instruction Counter
    - Program Status Word
  - Register File
  - Functional Unit
  - Data Transfer Unit
    - Data bus
    - Address bus



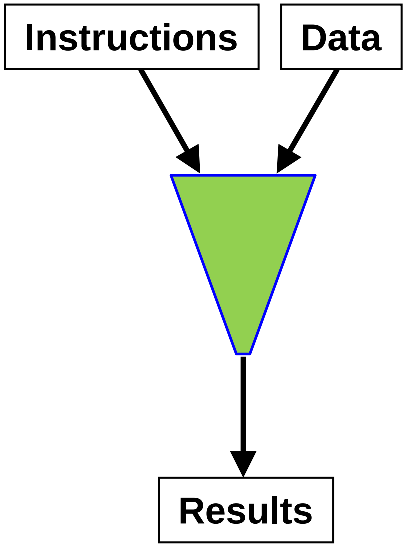
Simple server diagram

- **Multiple components which interact during the execution of a program:**
  - Processors/cores
  - Caches
    - Instructions (I-cache)
    - Data (D-cache)
  - Memory controllers
  - Memory (non-uniform)
  - I/O subsystem
    - Network attachment
    - Disk subsystem



Initial premise

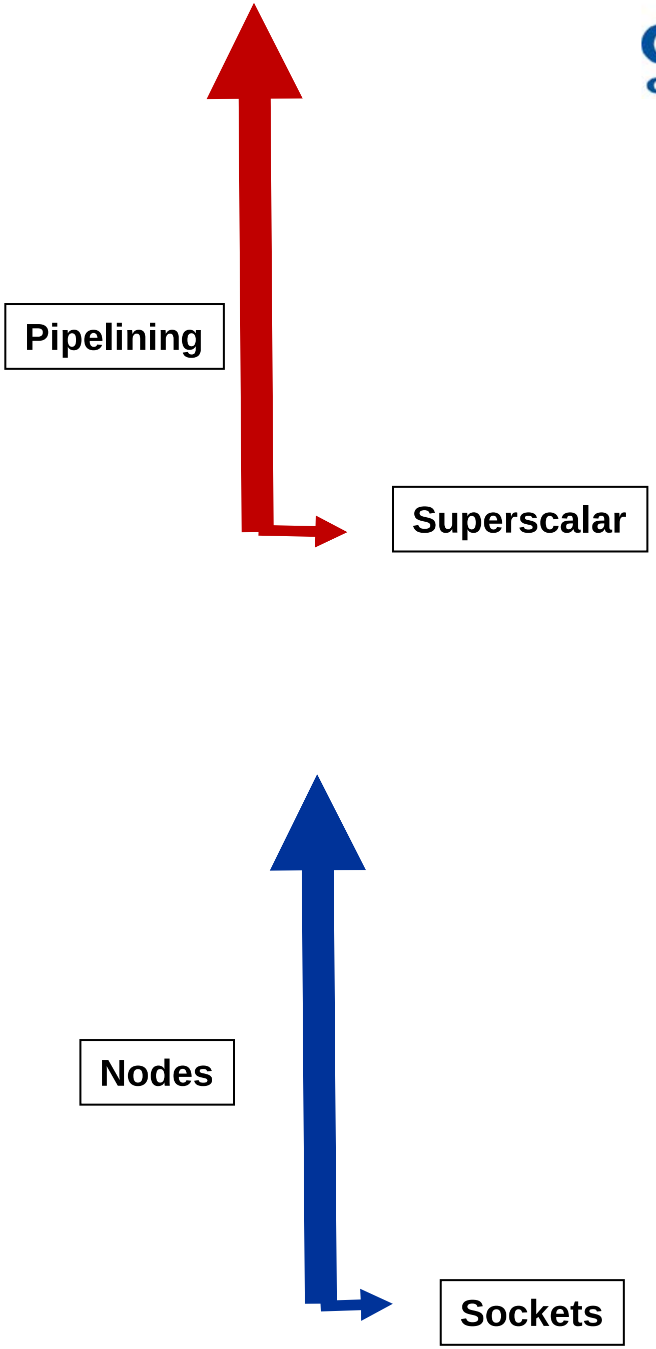
- **We want the process to complete in the shortest possible time**
  - Our compute job (a process) will require the execution of a given number of (machine-level) instructions
    - Dictated by the algorithms inside (and the compiler)
  - This time corresponds to a given number of machine cycles
- **Simple example:**
  - A program consists of 10^{10} instructions
  - We measure an execution time of 6 seconds on a processor running at 2.0 GHz
  - We can now compute a key value:
    - Cycles per Instruction (CPI)
    - Our result: $(6 * 2.0 * 10^9) / 10^{10} = 1.2$



In the days of the Pentium

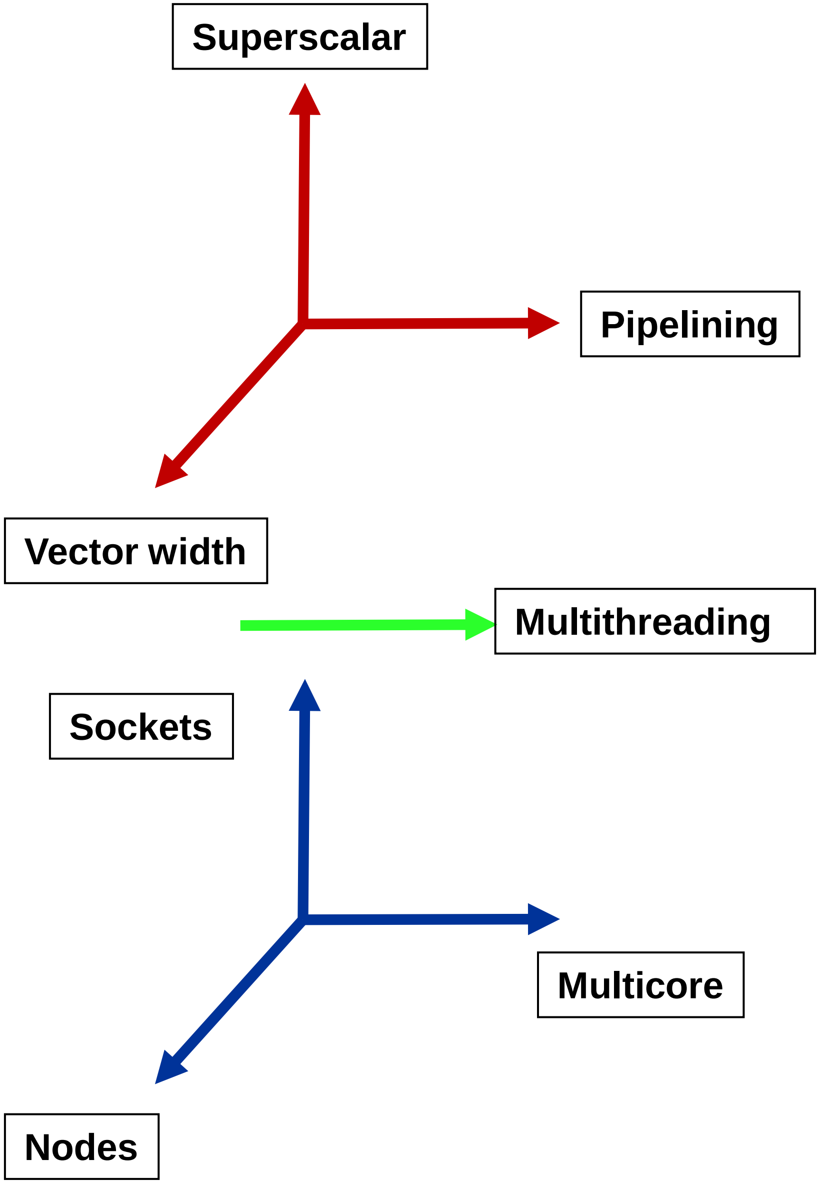
- **Life was really simple:**
  - Basically two dimensions
    - The frequency (of the pipeline)
    - The number of boxes

1. The semiconductor industry increased the frequency
2. We acquired the right number of (single-socket) boxes



Now: Seven dimensions of performance

- **First three dimensions:**
  - Hardware vectors/SIMD
  - Pipelining
  - Superscalar
- **Next dimension is a “pseudo” dimension:**
  - Hardware multithreading
- **Last three dimensions:**
  - Multiple cores
  - Multiple sockets
  - Multiple compute nodes



SIMD = Single Instruction Multiple Data

Seven multiplicative dimensions:

- **First three dimensions:**
  - Hardware vectors/SIMD
  - Pipelining
  - Superscalar

Data parallelism (Vectors/Matrices)

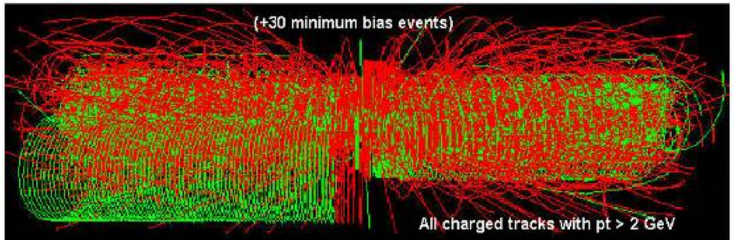
- **Next dimension is a “pseudo” dimension:**
  - Hardware multithreading
- **Last three dimensions:**
  - Multiple cores
  - Multiple sockets
  - Multiple compute nodes

Task parallelism (Events/Tracks)

Task/process parallelism

Concurrency in HEP

- **We are "blessed" with lots of it:**
  - Entire events
  - Particles, tracks and vertices
  - Physics processes
  - I/O streams (ROOT trees, branches)
  - Buffer handling (also data compaction, etc.)
  - Fitting variables
  - Partial sums, partial histograms
  - and many others …..



- **Usable for both data and task parallelism!**

- **But, fine-grained parallelism is not well exposed in today's software frameworks**

Autoparallelization/Autovectorization

- **Would it not be wonderful if the compilers could do all the (vectorization/parallelisation) work automatically?**

- **GNU compiler (4.3.0 or later):**
  - Autovectorization: YES, but needs “-ftree-vectorize”
    - “-ftree-vectorizer-verbose=[0-7]” for reports
  - Autoparallelization support in preparation
    - OpenMP support available

- **Intel compiler (10.1 or later):**
  - Autovectorization: YES, included in “-O”
    - “-vec-reportN” for reports
  - Autoparallelization: YES with “-parallel”
    - “-par-reportN” for reports

In addition, both compilers support intrinsics:
“higher-level assembly instructions” for explicit vectorization

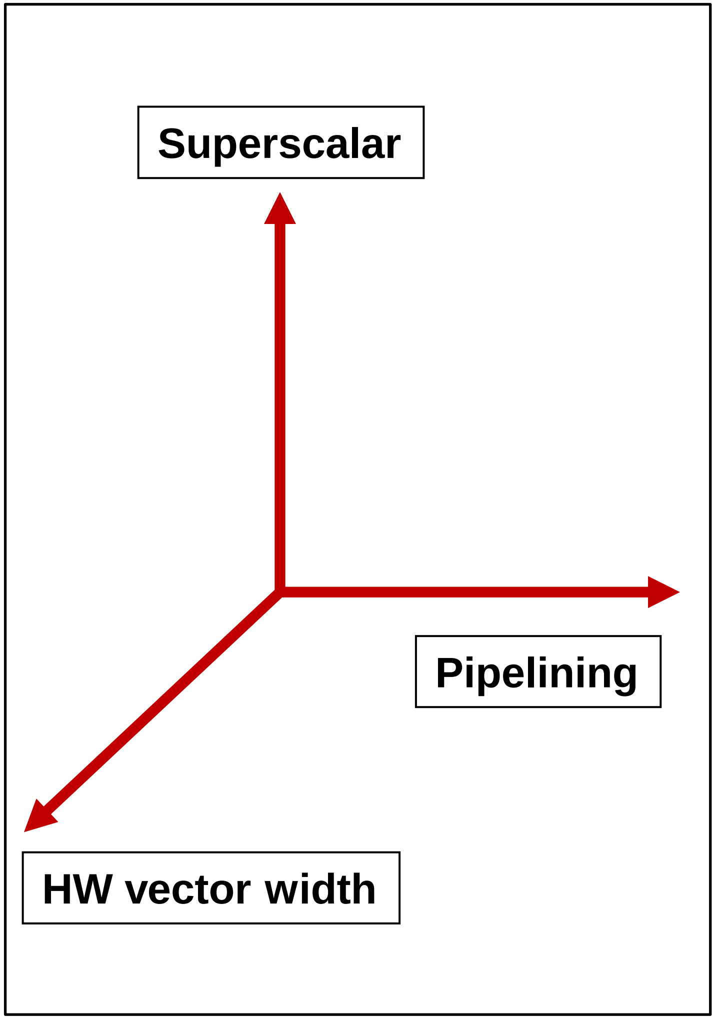
Part 1: Opportunities for scaling performance inside a core

- **Here are the first three dimensions**

- **The resources:**
  - HW vectors: Fill the computational width
  - Pipelining: Fill the stages
  - Superscalar: Fill the ports

- **Best approach: Data Oriented Design**

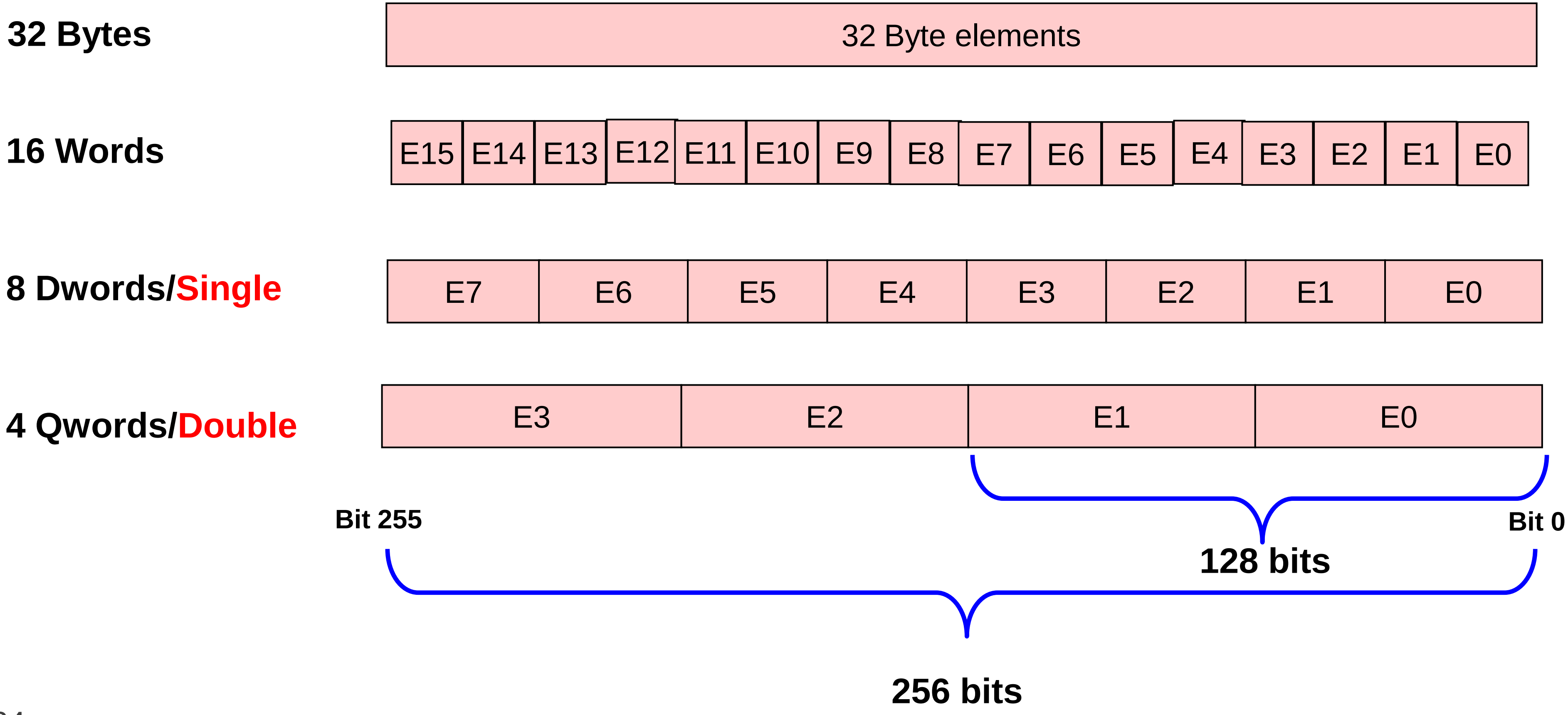
- **In HEP today, we probably extract only 10% of peak execution capability!**



First topic: Vector registers

- **Until now Steaming SIMD Extensions (SSE):**
  - 16 “XMM” registers with 128 bits each (in 64-bit mode)

- **New (as of 2011): Advanced Vector eXtensions (AVX):**
  - 16 “YMM” registers with 256 bits each



32 Bytes

32 Byte elements

16 Words

E15 E14 E13 E12 E11 E10 E9 E8 E7 E6 E5 E4 E3 E2 E1 E0

8 Dwords/Single

E7 E6 E5 E4 E3 E2 E1 E0

4 Qwords/Double

E3 E2 E1 E0

Bit 255

Bit 0

128 bits

256 bits

Four floating-point data flavours

- **Single precision**
  - Scalar single (SS)
  - Packed single (PS)

-	-	-	-	-	-	-	E0
E7	E6	E5	E4	E3	E2	E1	E0

- **Double precision**
  - Scalar Double (SD)
  - Packed Double (PD)

-	-	-	E0
E3	E2	E1	E0

- **Note:**
  - Scalar mode (with AVX) means using only:
    - 1/8 of the width (single precision)
    - 1/4 of the width (double precision)
  - Even longer vectors are (most likely) coming!
    - 512 bits (and maybe even 1024 bits one day)

Scalable programming inside a core

- **Easiest way to fill the execution capabilities is to program software vectors**
- **But, which ones?**
  - Standard C arrays
    - Intel has added Extended Array Notation to their 12.0 compiler
  - STL vectors
  - TBB vectors (thread-safe)
  - Intrinsics
  - etc.

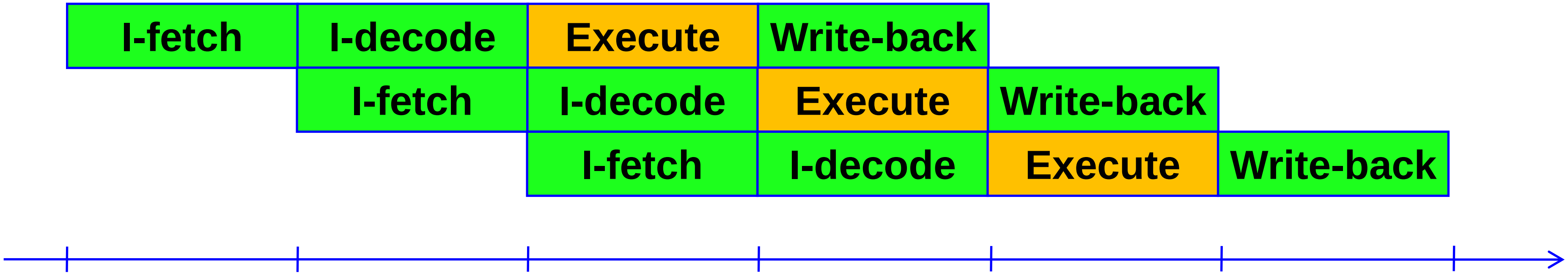
```
float  u[100], v[100];

for (int i = 0; i<50; ++i) u[i] = 0.0;

for (i = 0; i<50; ++i) u[i] = sin(v[i]);

for (int i = 0; i<50; ++i) u[i] = v[i*2+1];
```

CEAN example:

```
A[i:n] = 2.5 * B[j:n] ;
```

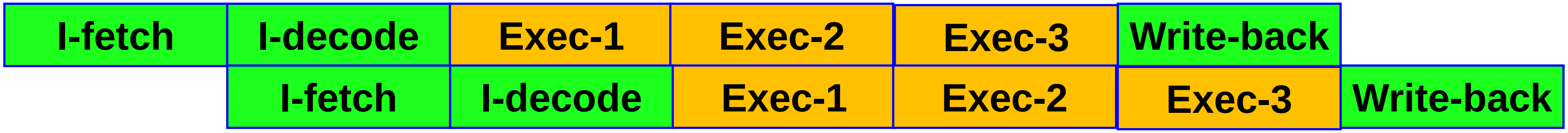
Next topic: Instruction pipelining

- **Instructions are <u>always</u> broken up into stages.**
  - With a one-cycle execution latency (simplified):



I-fetch	I-decode	Execute	Write-back		
	I-fetch	I-decode	Execute	Write-back	
		I-fetch	I-decode	Execute	Write-back

  - With a three-cycle execution latency:



I-fetch	I-decode	Exec-1	Exec-2	Exec-3	Write-back	
	I-fetch	I-decode	Exec-1	Exec-2	Exec-3	Write-back

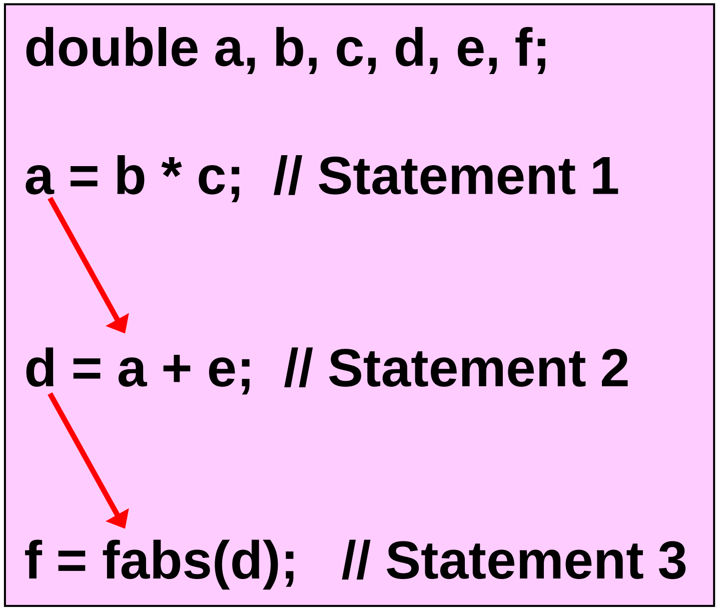
Real-life latencies

- **Most integer/logic instructions have a one-cycle execution latency:**
  - For example:
    - ADD, AND, SHL (shift left), ROR (rotate right)
  - Amongst the exceptions:
    - IMUL (integer multiply): 3
    - IDIV (integer divide): 13 – 23
- **Floating-point latencies are typically multi-cycle**
  - FADD (3), FMUL (5)
    - Same for both x87 and SIMD double-precision variants
  - Exception: FABS (absolute value): 1
  - Many-cycle: FSQRT (27), FDIV (20)

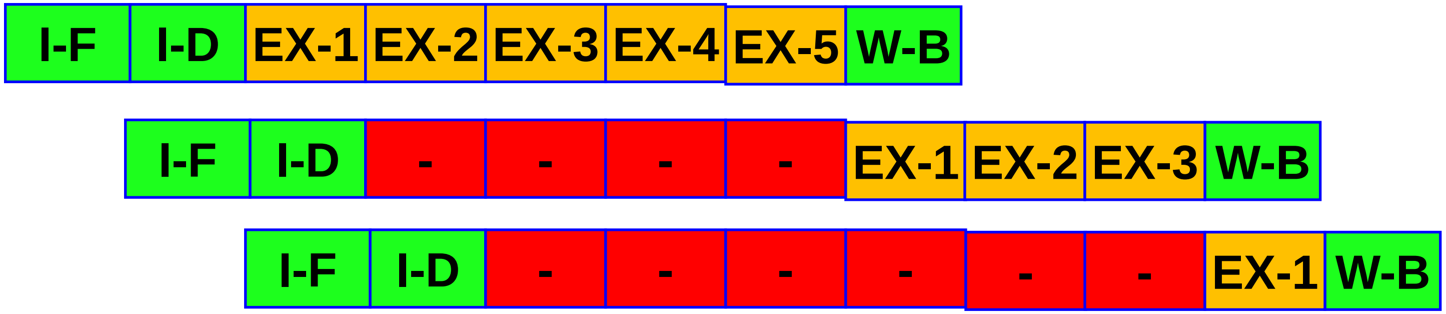
Latencies in the Core micro-architecture (Intel Manual No. 248966-020 or later). AMD processor latencies are similar.

Latencies and serial code (1)

- **In serial programs, we typically pay the penalty of a multi-cycle latency during execution:**
  - In this example:
    - Statement 2 cannot be started before statement 1 has finished
    - Statement 3 cannot be started before statement 2 has finished

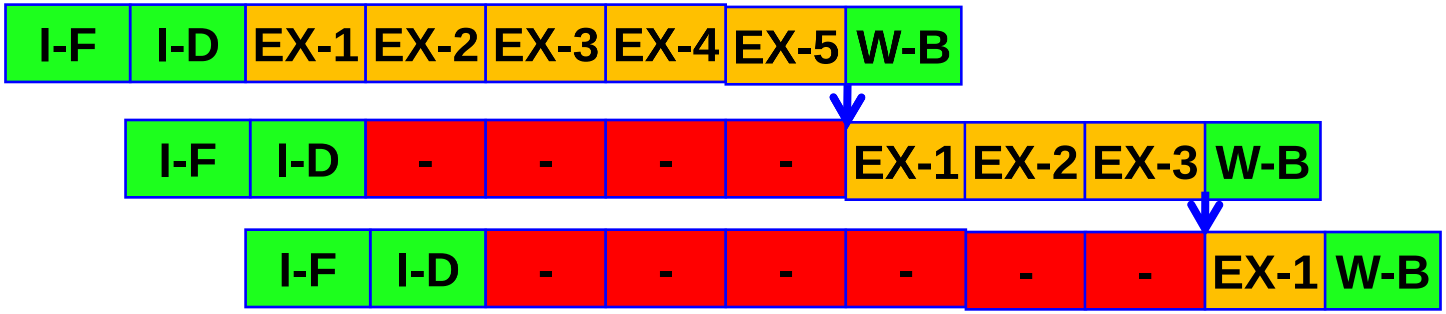


```
double a, b, c, d, e, f;

a = b * c;  // Statement 1

d = a + e;  // Statement 2

f = fabs(d);   // Statement 3
```



I-F	I-D	EX-1	EX-2	EX-3	EX-4	EX-5	W-B				
	I-F	I-D	-	-	-	-	EX-1	EX-2	EX-3	W-B	
		I-F	I-D	-	-	-	-	-	-	EX-1	W-B

Latencies and serial code (2)

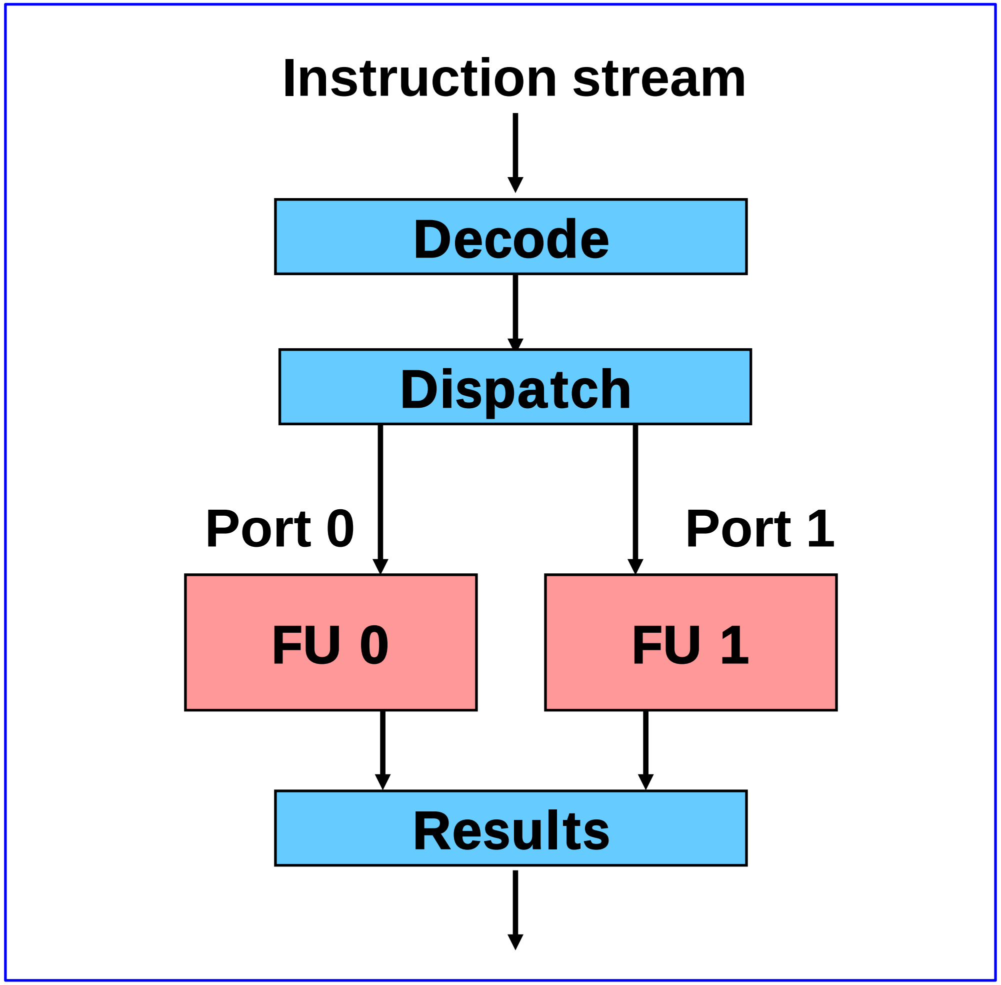


I-F	I-D	EX-1	EX-2	EX-3	EX-4	EX-5	W-B				
	I-F	I-D	-	-	-	-	EX-1	EX-2	EX-3	W-B	
		I-F	I-D	-	-	-	-	-	-	EX-1	W-B

- **Observations:**
  - Even if the processor can fetch and decode a new instruction every cycle, it must wait for the previous result to be made available
    - Fortunately, the result takes a ‘bypass’, so that the write-back stage does not cause even further delays
  - The result: CPI is equal to 3
    - 9 execution cycles are needed for 3 instructions!
- **A good way to hide latency is to unroll (vector) loops !**

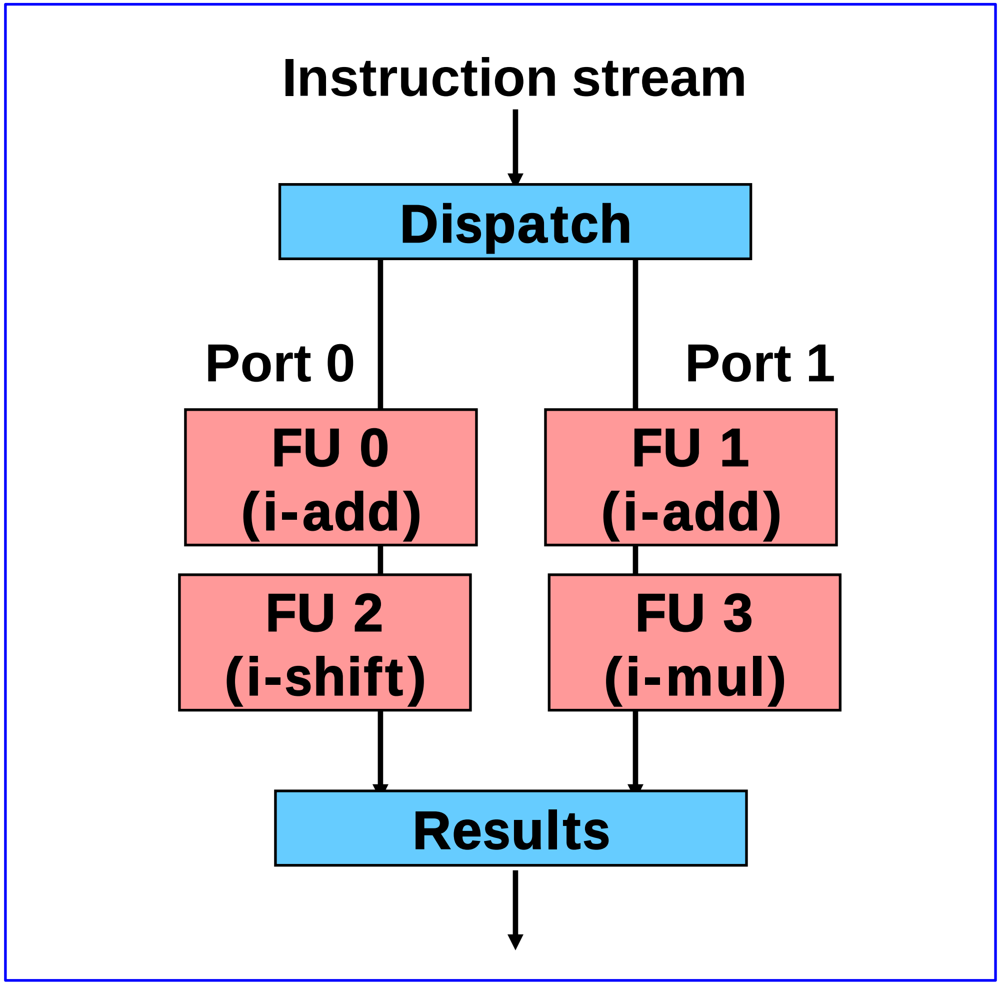
Third topic: Superscalar architecture

- **In this simplified design, instructions are decoded in sequence, but dispatched to two Function Units.**
  - The decoder and dispatcher must be able to handle two instructions per cycle
  - The FUs can have identical or different execution capabilities



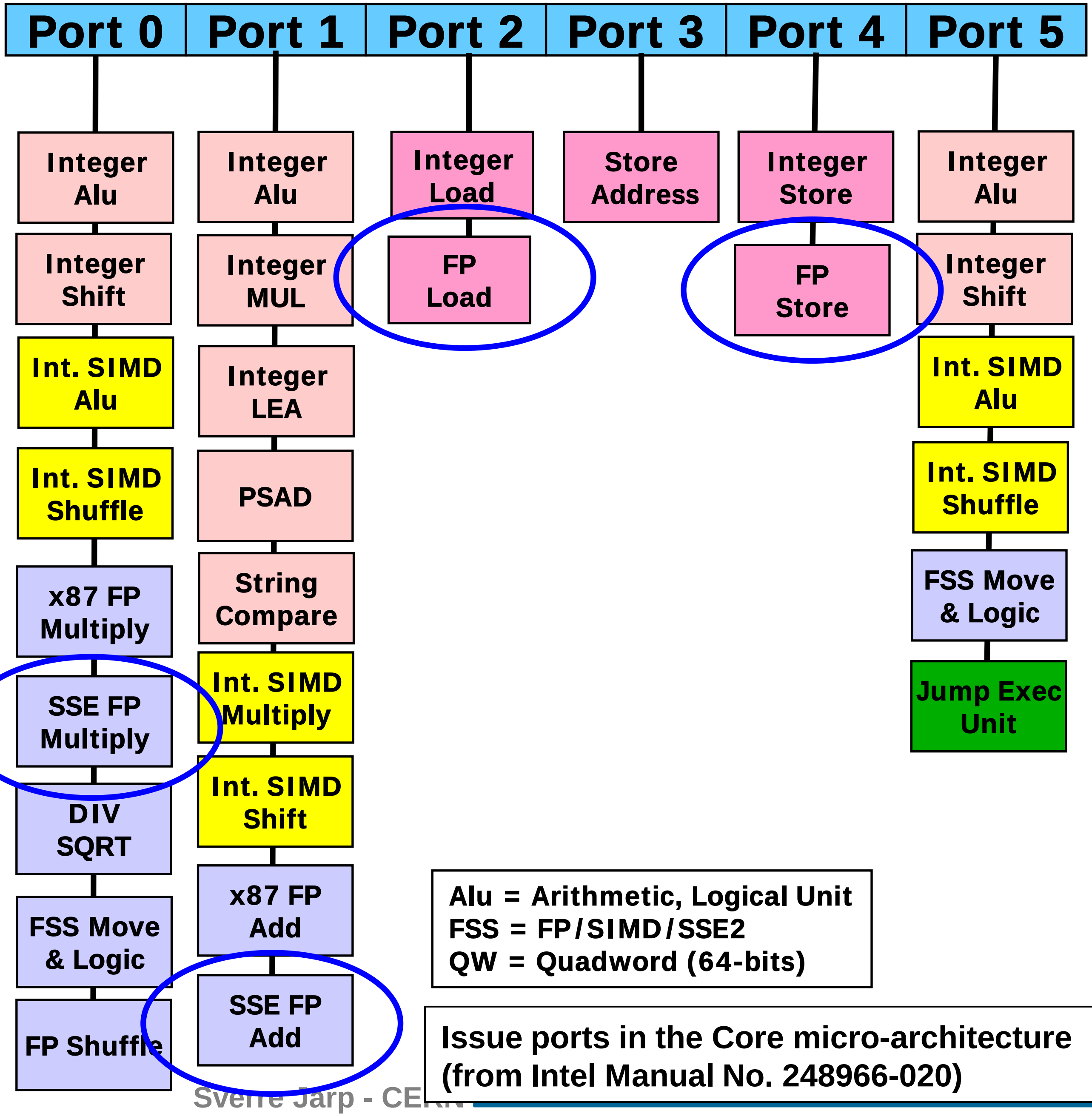
Enhanced superscalar architecture

- **A more realistic architecture will have multiple FUs attached to the same port**
  - An instruction can be dispatched to either <u>matching</u> execution unit on a given port, but not to both units on the same port in a given cycle



Modern superscalar architecture

- For instance, Intel’s Nehalem microarchitecture can dispatch/execute/retire four instructions in parallel (across six ports) in each cycle:



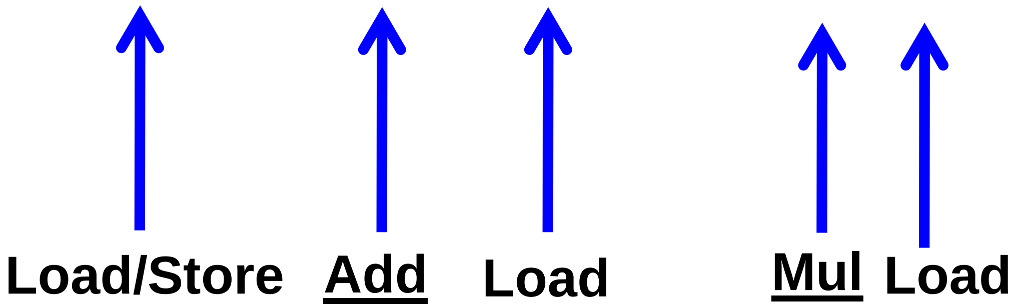
Alu = Arithmetic, Logical Unit
FSS = FP/SIMD/SSE2
QW = Quadword (64-bits)

Issue ports in the Core micro-architecture (from Intel Manual No. 248966-020)

Matrix multiply superscalar example

- **For a given algorithm, we can understand exactly which functional execution units (and ports) will be used in each cycle:**
  - For instance, in the innermost loop of matrix multiplication

```
unsigned long i,j,k;
for(i=0;i<N;i++) {
        for(k=0;k<N;k++) {
            for(j=0;j<N;j++) {
                c[i][j]    +=   a[i][k]   *    b[k][j];             }}}
```

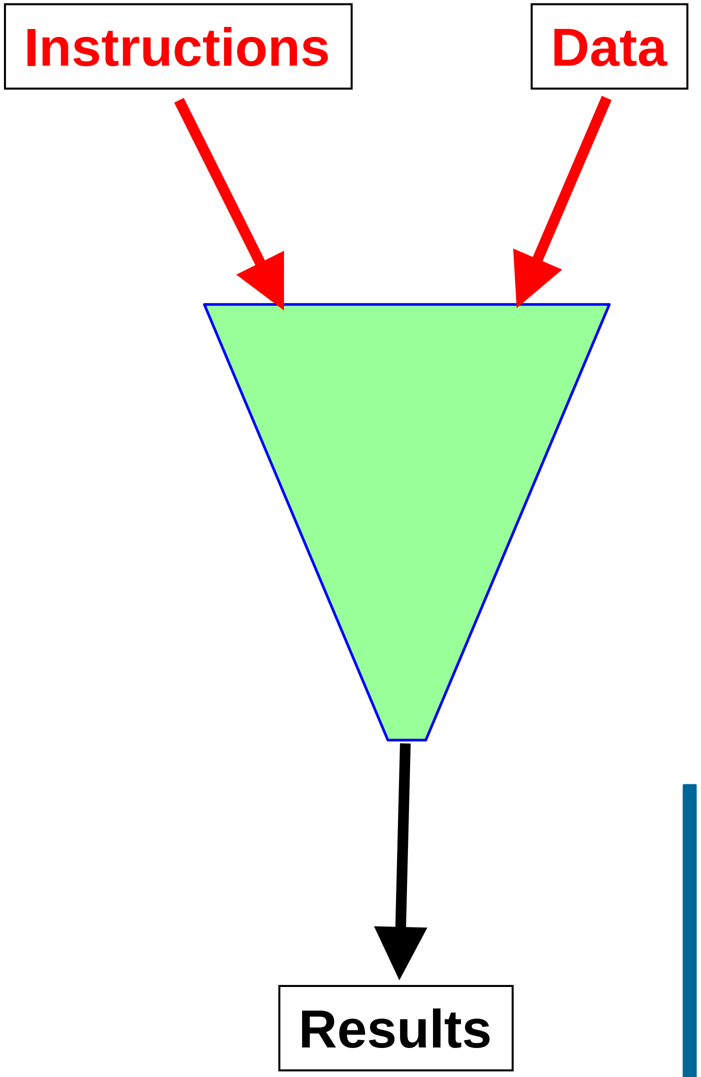


Load/Store **Add** Load **Mul** Load

Possible causes of execution delays (1)

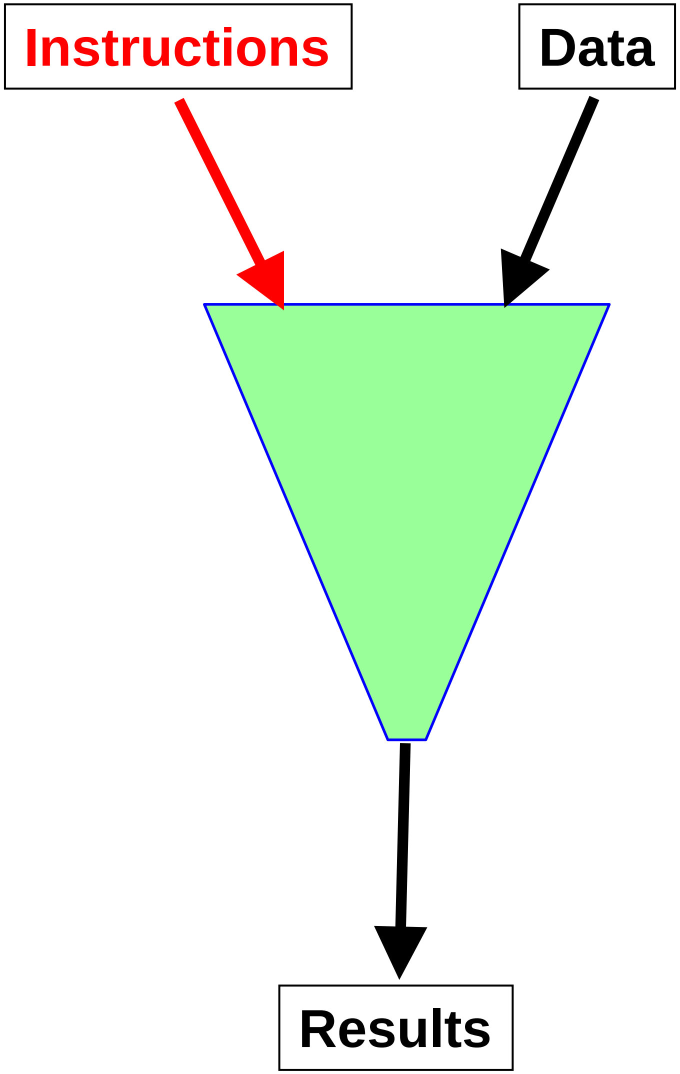
- **We already stated that the aim is to keep instructions and data flowing, so that results are generated optimally**

- **First issue:**
  - Instructions and/or data stop flowing
    - Instructions are not found in the I-cache
    - Data is not found in the D-cache
  - Before execution can continue, instructions and data must be fetched from a lower level of the memory hierarchy



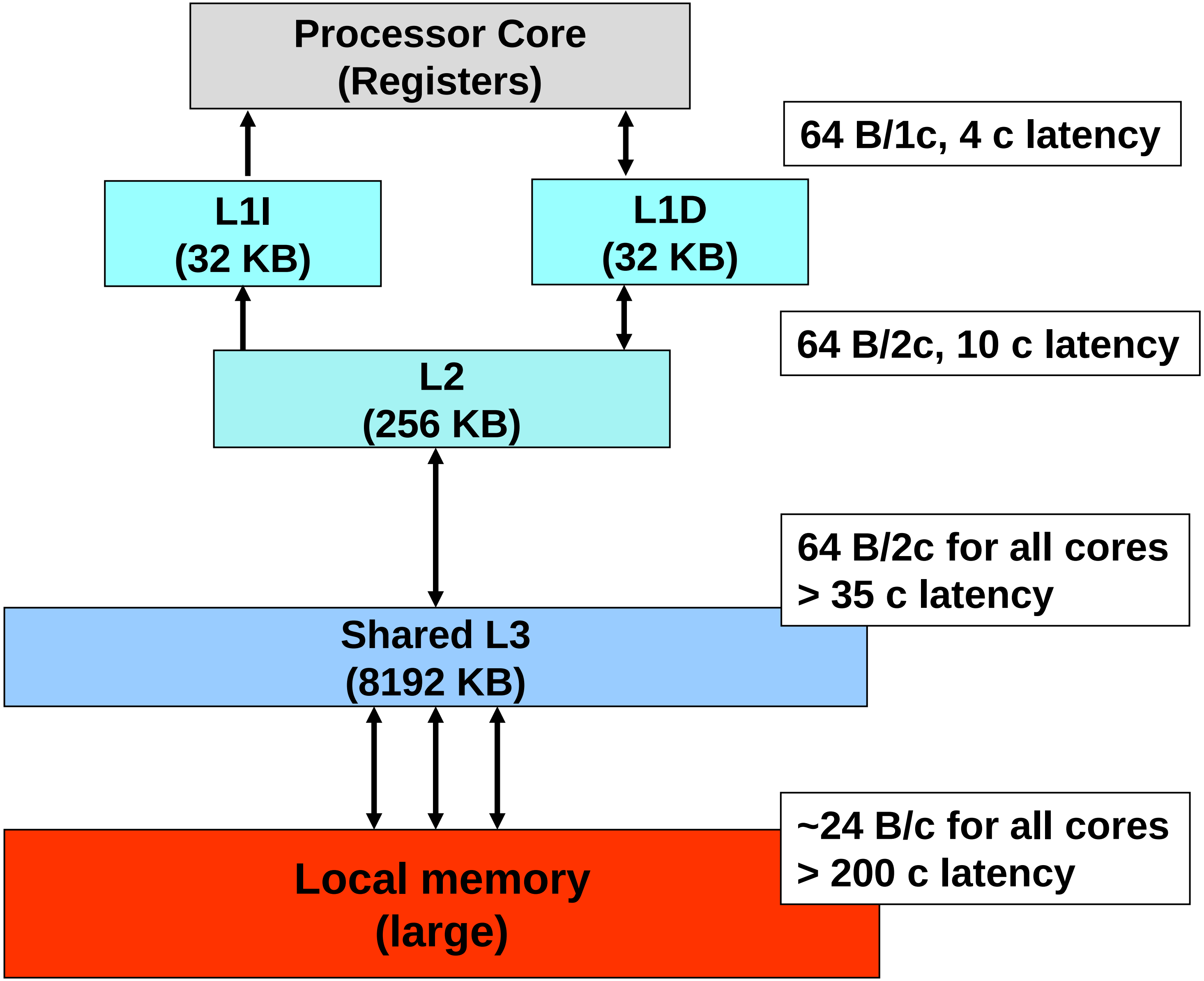
Possible causes of execution delays (2)

- **Second issue:**
  - Instructions are not ready in time for execution (Front-end stalls)
    - Typically caused by branching
    - If the branch is mispredicted, we suffer a stall (cycles add up, but no work gets done)
    - We typically find that 10% of all instructions are branch instructions
      - – Or even more
        - And, unavoidingly, some of them will be mispredicted



Memory Hierarchy

- **From CPU to main memory on a Nehalem processor**
  - With multicore, memory bandwidth is shared between cores in the same processor (socket)



c = cycle

Cache lines (1)

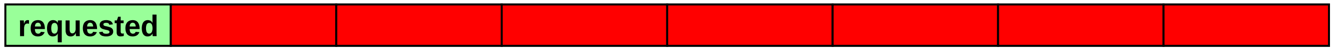
- **When a data element or an instruction is requested by the processor, a <u>cache line</u> is ALWAYS moved (as the minimum quantity) to Level-1**

requested							

- **A cache line is a contiguous section of memory, typically 64B in size (8 * double)**
  - A 32KB level-1 cache holds 512 (64B) lines
- **When cache lines have to be moved come from memory**
  - Latency is long (>200 cycles, as already mentioned)
    - It is even longer if the memory is remote
  - Memory controller stays busy (~6-8 cycles)

Cache lines (2)

- **Space <u>locality</u> is vital**
  - When only one element (4B or 8B) element is used inside the cache line:
    - A lot of bandwidth is wasted!



- **Multidimensional arrays should be accessed with the last index changing fastest:**

```
for (i = 0; i < rows; ++i)
        for (j = 0; j < columns; ++j)
                mymatrix [i] [j]   += increment;
```

- **Pointer chasing (in linked lists) can easily lead to “cache thrashing” (too much memory traffic)**

Programming the memory hierarchy is an art in itself.

Inside-the-core: HEP and vectors

- **Too little common ground!**
  - Practically all attempts in the past failed.
    - w/CRAY, CYBER 205, IBM 3090-Vector Facility, etc.
    - Interesting reading: Dekeyser J 1987 “Vectorization of the GEANT3 geometrical routines for a Cyber 205” Nuclear Instruments and Methods in Physics Research Section A, Volume 264, Issue 2-3, p. 291-296

- **From time to time, we see a good vector example**
  - For example: Track Fitting code from ALICE trigger
    - → See one of the next slides

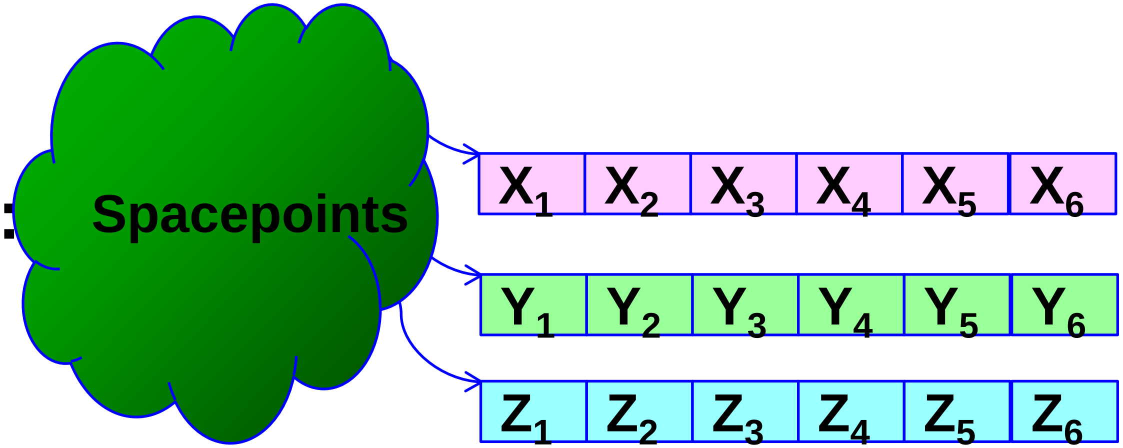
- **Interesting development from ALICE (Matthias Kretz):**
  - Vc (Vector Classes)
    - http://compeng.uni-frankfurt.de/index.php?id=vc

- **Other examples: Use of STL vectors; small matrices**

SoA versus AoS

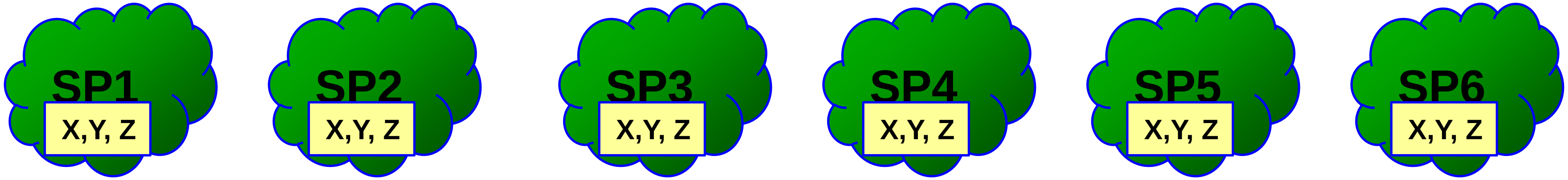
- **In general, compilers and hardware prefer the former!**

- **Structure of Arrays (SoA):**



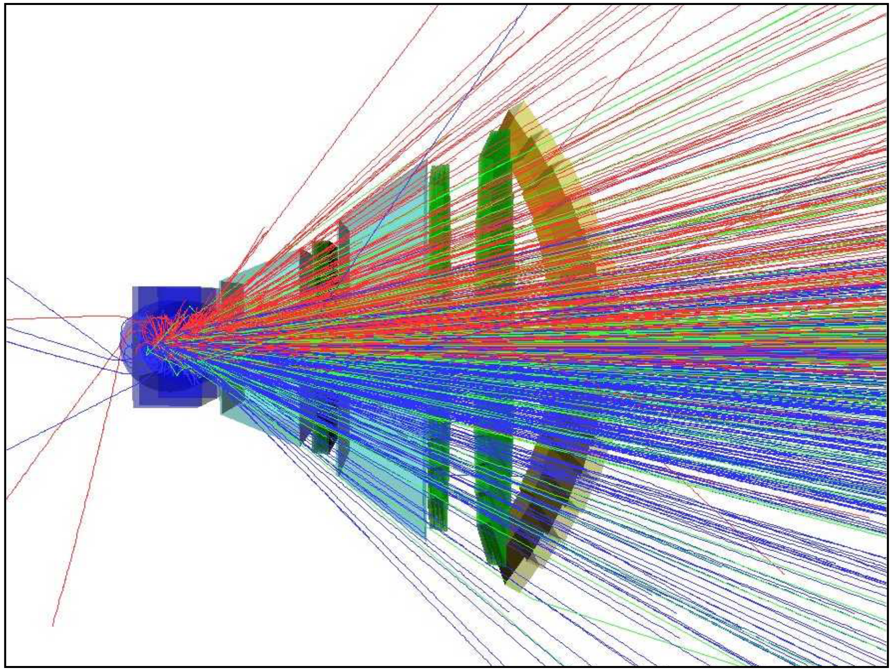
X_1	X_2	X_3	X_4	X_5	X_6
Y_1	Y_2	Y_3	Y_4	Y_5	Y_6
Z_1	Z_2	Z_3	Z_4	Z_5	Z_6

- **Array of Structures (AoS):**



Examples of parallelism: CBM/ALICE track fitting

- **Extracted from their High Level Trigger (HLT) Code**
  - Originally ported to IBM’s Cell processor
- **Tracing particles in a magnetic field**
  - Embarrassingly parallel code
- **Re-optimization on x86-64 systems**
  - Using vectors instead of scalars



I.Kisel/GSI: “Fast SIMDized Kalman filter based track fit”
http://www-linux.gsi.de/~ikisel/17_CPC_178_2008.pdf

“Compressed Baryonic Matter”

CBM/ALICE track fitting

- **Re-optimization on x86-64 systems**
  - First: use SSE vectors instead of scalars
    - Operator overloading allows seamless change of data types
    - Intrinsics (from Intel/GNU header file): Map directly to instructions:
      - __mm_add_ps corresponds directly to ADDPS, the instruction that operates on **four** packed, single-precision FP numbers
        - 128 bits in total
    - Classes
      - P4_F32vec4 – packed single class with overloaded operators
        - F32vec4 operator +(const F32vec4 &a, const F32vec4 &b) { return _mm_add_ps(a,b); }
    - Result: 4x speed increase from x87 scalar to packed SSE (single precision)

Important performance counters

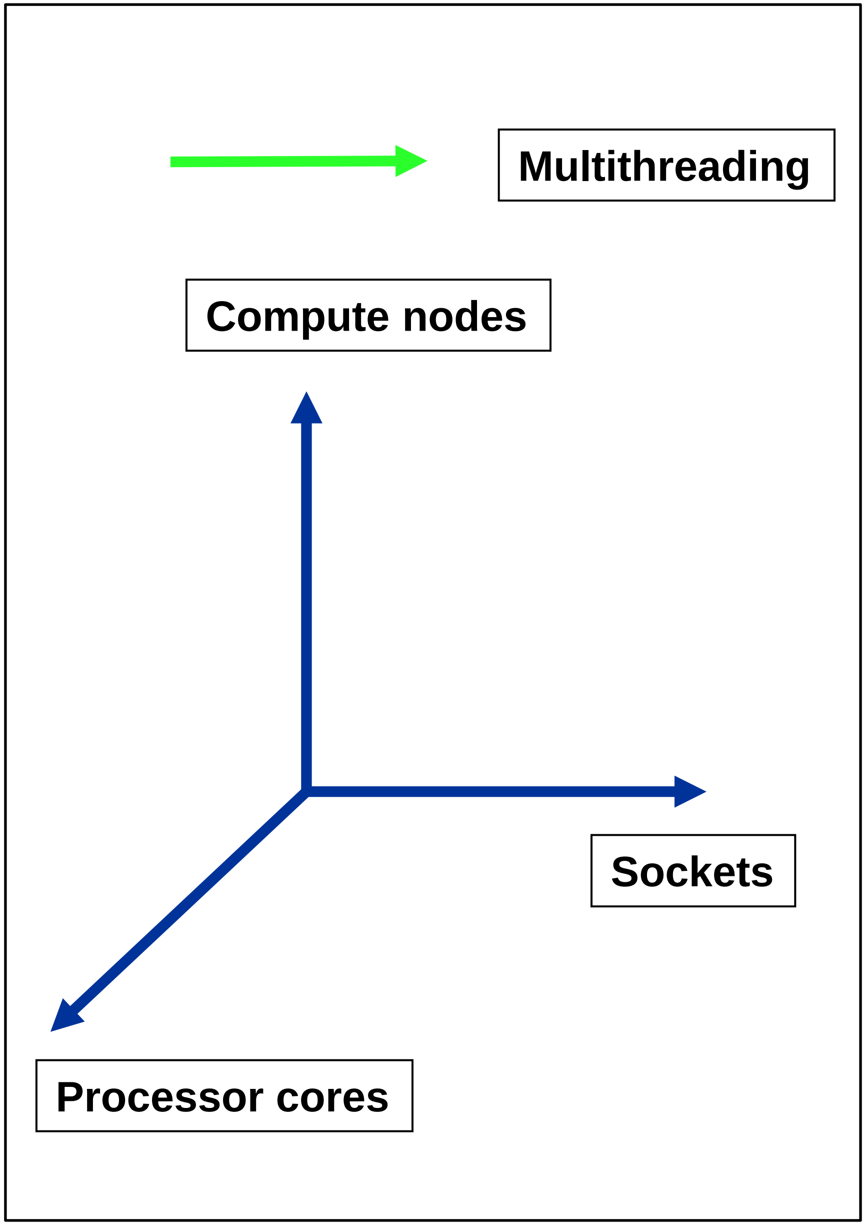
(that can tell you if things go wrong)

- **Related to what we have discussed:**
  - The total cycle count (C)
  - The total instruction count (I)
  - Derived value: CPI
  - Resource Stall count: Cycles when no execution occurred
  - Total number of executed branch instructions
  - Total number of mispredicted branches

- **Plus:**
  - Total number of cache accesses
  - Total number of (last-level) cache misses
  - The total number (and the type) of computational SSE/AVX instructions
  - The total number of SSE/AVX instructions

Part 2: Parallel execution across hw-threads and cores

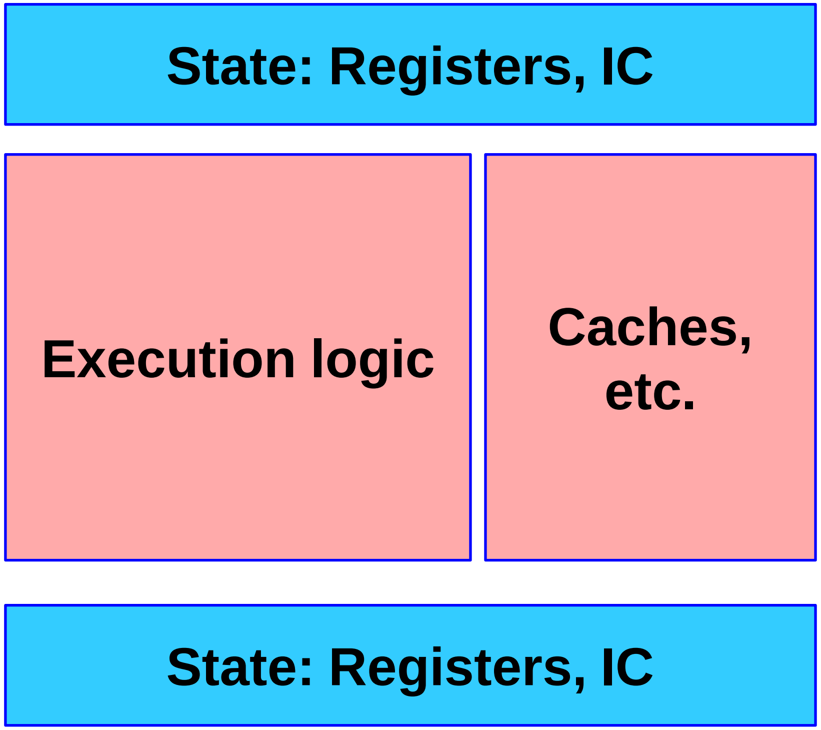
- **Next dimension is a "pseudo" dimension:**
  - Hardware multithreading
- **Last three dimensions:**
  - Multiple cores
  - Multiple sockets
  - Multiple compute nodes
- **Multiple nodes will not be discussed here**
  - Our focus is scalability inside a node



Definition of a hardware core/thread

- **Core**
  - A complete ensemble of execution logic, and cache storage as well as register files plus instruction counter (IC) for executing a software process or thread

- **Hardware thread**
  - Addition of a set of register files plus IC



The sharing of the execution logic can be coarse-grained or fine-grained.

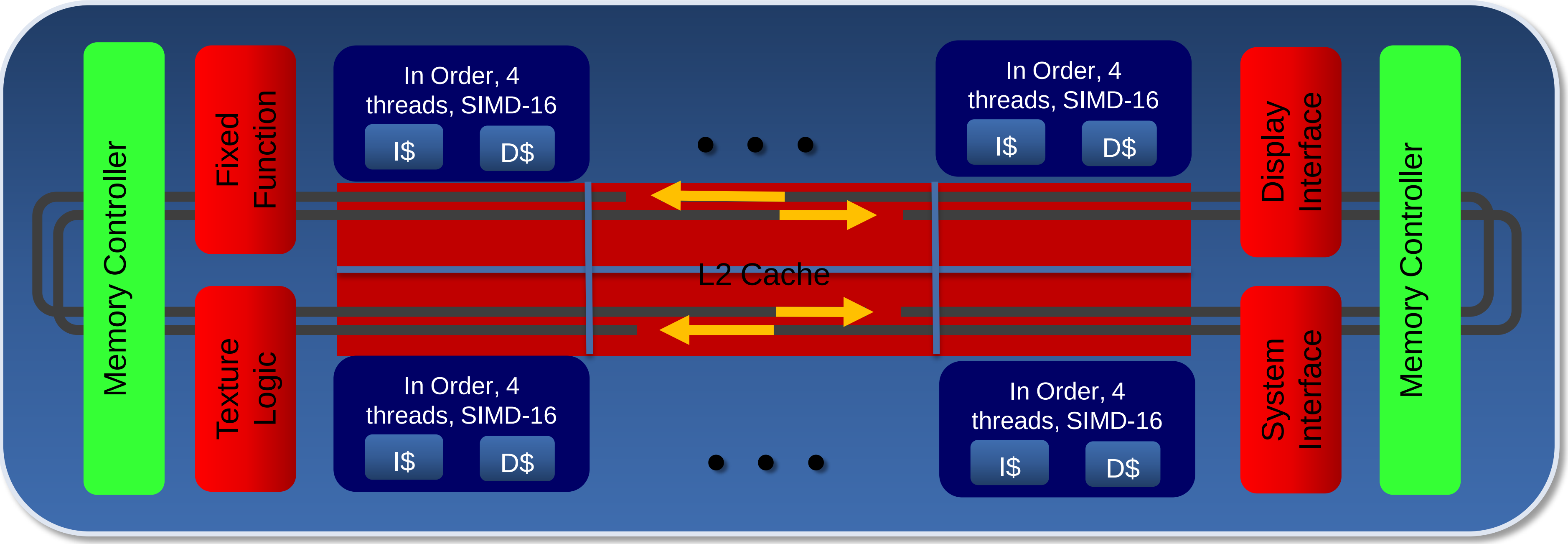
Please refer to slide 14

The move to many-core systems

- **Examples of “dispatch slots”: Sockets * Cores * HW-threads**
  - Basically what you observe in “cat /proc/cpuinfo”
- Conservative:
  - Dual-socket AMD six-core (Istanbul): 2 * 6 * 1 = 12
  - Dual-socket Intel six-core (Westmere-EP): 2 * 6 * 2 = 24
- More aggressive:
  - Quad-socket AMD Interlagos (16-core) 4 * 16 * 1 = 64
  - Quad-socket Westmere-EX “deca-core”: 4 * 10 * 2 = 80
- **In the near future: Hundreds of CPU slots !**
  - Quad-socket Oracle/Sun Niagara (T3) processors w/16 cores and 8 threads (each): 4 * 16 * 8 = 512
- **And, by the time new software is ready: Thousands !!**

Accelerators (1): Intel MIC

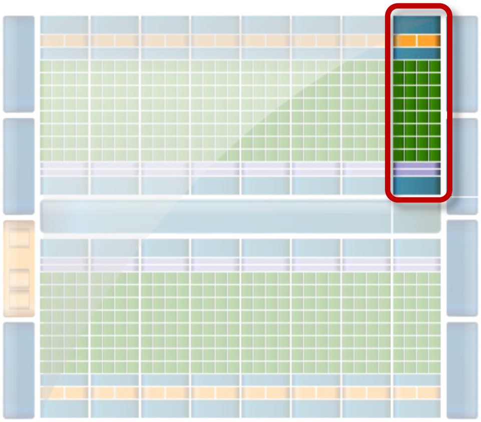
- **Many Integrated Core architecture:**
  - Announced at ISC10 (end-May 2010)
  - Based on the x86 architecture, 22nm (in 2012?)
  - Many-core (> 50 cores) + 4-way multithreaded + 512-bit vector unit
  - Limited memory: A few Gigabytes



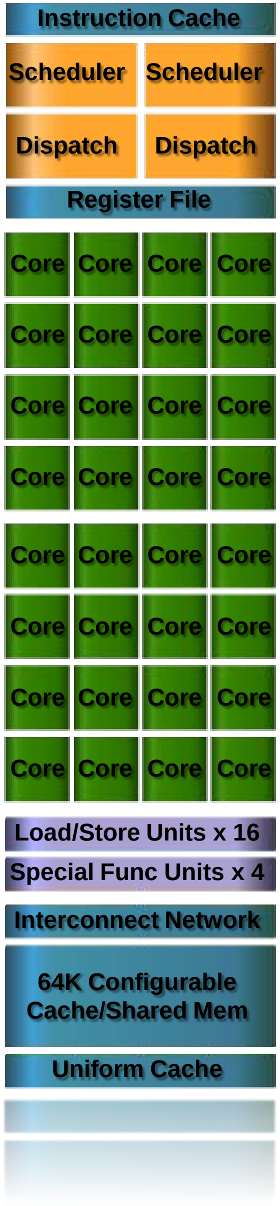
Accelerators (2): Nvidia Fermi GPU

- **Streaming Multiprocessing (SM) Architecture**

- **32 “CUDA cores” per SM (512 total)**
- **Peak single precision floating point performance (at 1.15 GHz):**
  - **Above 1 Tflop**
- **Double-precision: 50%**
- **Dual Thread Scheduler**
- **64 KB of RAM for shared memory and L1 cache (configurable)**
- **A few Gigabytes of main memory**



Lots of interest in the HEP on-line community



Adapted from Nvidia

Definition of a software process and thread

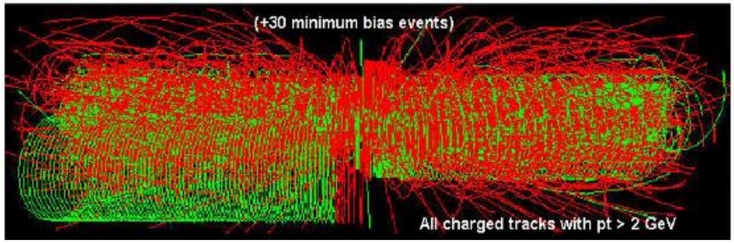
- **Process (OS process):**
  - An instance of a computer program that is being executed (sequentially). It typically runs as a program with its private set of operating system resources, i.e. in its own “address space” with all the program code and data, its own file descriptors with the operating system permissions, its own heap and its own stack.

- **Thread:**
  - A process may have multiple threads of execution. These threads run in the same address space, share the same program code, the operating system resources as the process they belong to. Each thread gets its own stack.

Adapted from Wikipedia

HEP programming paradigm

- **Event-level parallelism has been used for decades**
- **And, we should not lose this advantage:**
  - Large jobs can be split into N efficient “chunks”, each responsible for processing M events
  - Has been our “forward scalability”



- **Disadvantage with current approach:**
  - Memory must be made available to each <u>process</u>
    - A dual-socket server with six-core processors needs 24 – 36 GB (or more)
    - Today, SMT is often switched off in the BIOS (!)
- **We must <u>not</u> let memory limitations decide our ability to compute efficiently!**

What are the multi-core options?

- **There is currently a discussion in the community about the best way forward:**

1) Stay with event-level parallelism (and entirely independent processes)
   - Assume that the necessary memory remains affordable
   - Or rely on tools, such as KSM, to help share pages
2) Rely on forking:
   - Start the first process; Run through the first "event"
   - Fork N other processes
   - Rely on the OS to do "copy on write", in case pages are modified
3) Move to a fully multi-threaded paradigm
   - Still using coarse-grained (event-level) parallelism
     - But, watch out for increased complexity

Mini-example of real-life serial code

- **Suffers long latencies:**

High level C++ code →

```
if (abs(point[0] - origin[0]) > xhalfsz) return FALSE;
```

Machine instructions →

```
movsd 16(%rsi), %xmm0
subsd 48(%rdi), %xmm0   // load & subtract
andpd _2il0floatpacket.1(%rip), %xmm0 // and with a mask
comisd 24(%rdi), %xmm0 // load and compare
jbe ..B5.3      # Prob 43%  // jump if FALSE
```

Same instructions laid out according to latencies on the Core 2 processor →

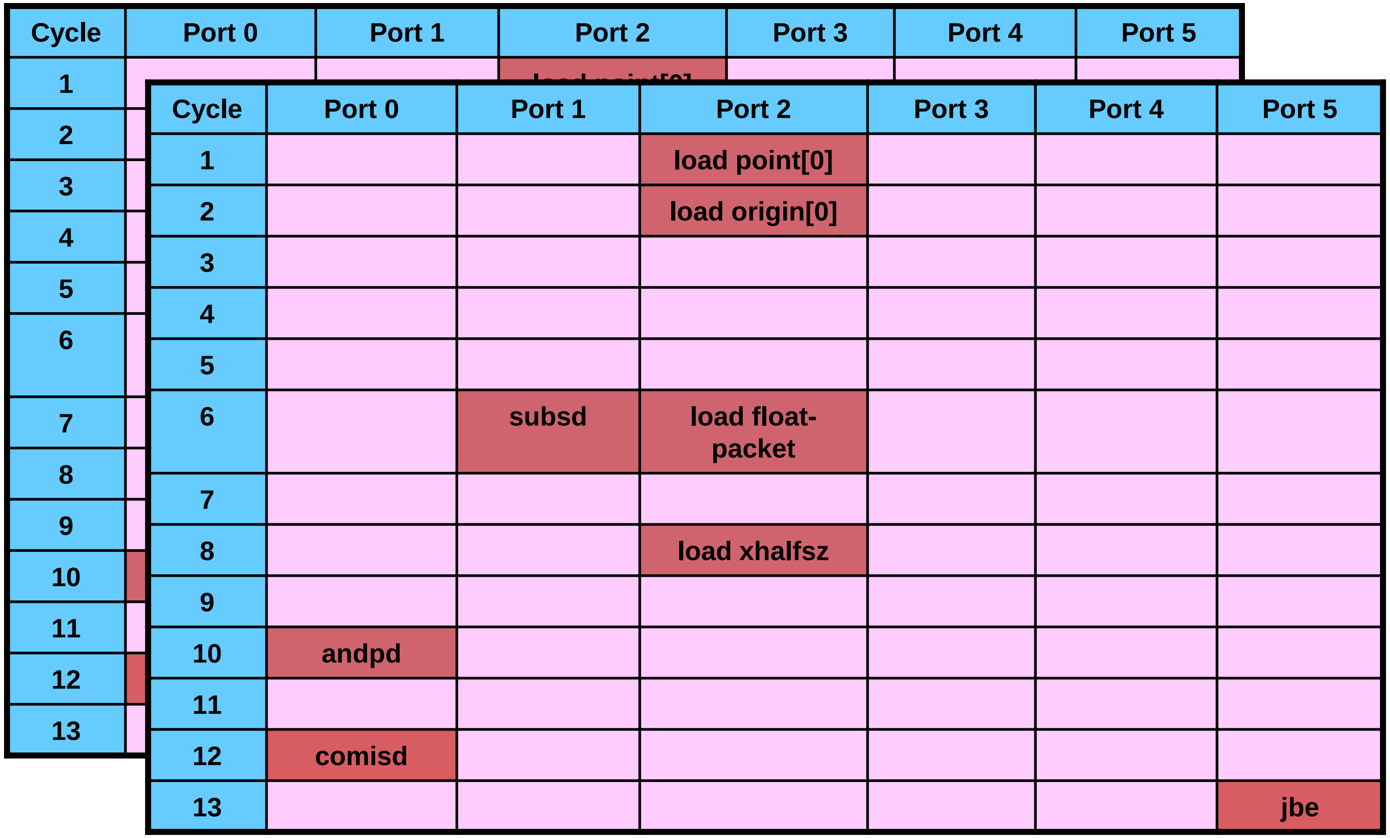
NB: Out-of-order scheduling not taken into account.

Cycle	Port 0	Port 1	Port 2	Port 3	Port 4	Port 5
1			load point[0]			
2			load origin[0]			
3						
4						
5						
6		subsd	load float-packet			
7						
8			load xhalfsz			
9						
10	andpd					
11						
12	comisd					
13						jbe

HEP and Symmetric Multi-Threading

- **Because we have “thin” instruction streams, we ought to profit from SMT, provided the memory issue is under control**
  - It would seem that we could easily tolerate up to 4 hardware threads!

Unfortunately, on Xeon 5600, we currently get max 25% from the second hardware thread !



Cycle	Port 0	Port 1	Port 2	Port 3	Port 4	Port 5
1			load point[0]			
2						
3						
4						
5						
6						
7						
8						
9						
10						
11						
12						
13						

Cycle	Port 0	Port 1	Port 2	Port 3	Port 4	Port 5
1			load point[0]			
2			load origin[0]			
3						
4						
5						
6		subsd	load float-packet			
7						
8			load xhalfsz			
9						
10	andpd					
11						
12	comisd					
13						jbe

SMT (Symmetric Multi-Threading)

Let’s look more closely at parallelism

From Concurrency to Parallel Execution

- **Multiple steps must be kept in mind:**
  - Concurrency
  - Decomposition
  - Communication
  - Synchronization
  - Mapping to hardware resources
  - Execution

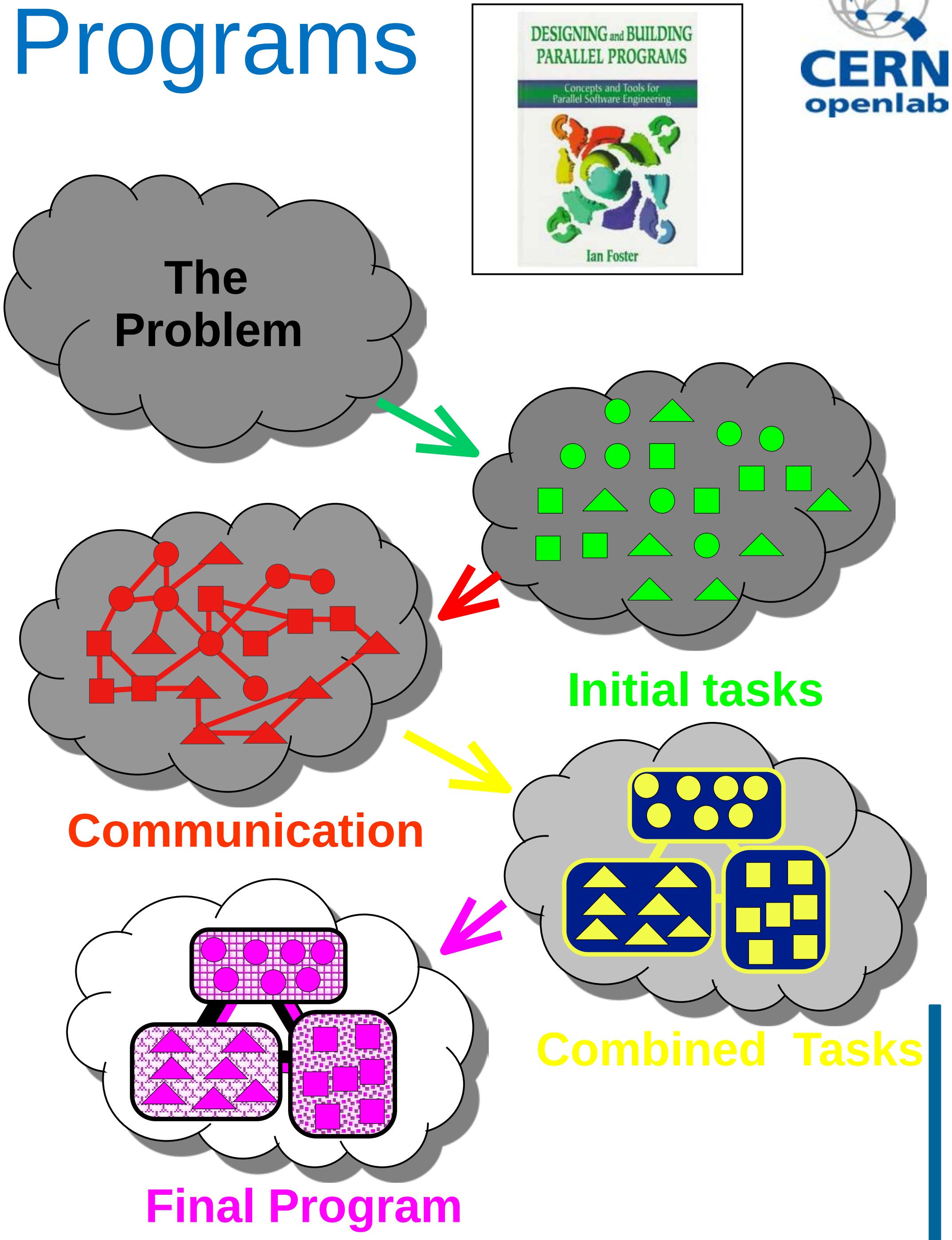
- **Keeping Amdahl's law for max speedup in mind**

$$S_p^{\max}(n) = \frac{1}{1-p+\frac{p}{n}}$$

where:
p (parallel portion)
s (serial portion)
p + s = 1.0

Designing Threaded Programs

- **Partition**
  - Divide problem into tasks
- **Communicate**
  - Determine amount and pattern of communication
- **Agglomerate**
  - Combine tasks
- **Map**
  - Assign agglomerated tasks to created threads



DESIGNING and BUILDING PARALLEL PROGRAMS

Concepts and Tools for Parallel Software Engineering

Ian Foster

More on decomposition

- **Divide the total work into smaller parts,**
  - Which can be executed concurrently

- **Some techniques:**
  - Data decomposition
    - Partition the data domain
  - Task/functional decomposition
    - Split according to “logical” tasks/functions
  - Recursive decomposition
    - Divide-and-conquer strategy
  - Exploratory decomposition
    - Search for a configuration space for a solution
      - Not guaranteed to reduce amount of work

Recommendations

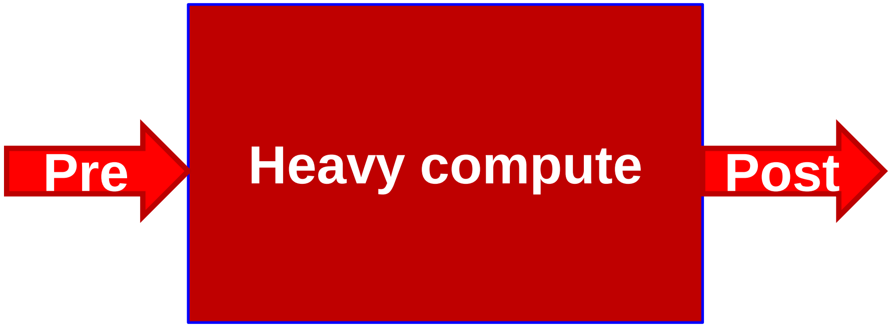
(based on observations in openlab)

A proposal for “agile” software:

1) **Create compute-intensive kernels**
2) **Parallelism at all levels**
   a. Events, tracks, vertices, etc.
   b. Perform “chunk” processing (removing event separation)
3) **Built-in forward scalability**
4) **Efficient memory footprint**
5) **Locality-optimised data layout**
6) **Performance-oriented C++**
7) **Broad Programming Talent**
8) **Best-of-breed Tools**

Performance guidance

- **Take the whole program and its execution behaviour into account**
  - Get yourself a global overview as soon as possible
    - Via early prototypes
    - Influence early the design and definitely the implementation
- **Foster clear split:**
  - Prepare to compute
  - Do the heavy computation
    - In kernels, where you go after the all the available parallelism
  - Post-processing



Pre

Heavy compute

Post

- **Consider exploiting the entire server**
  - Using affinity scheduling

Performance recommendations

- **Control memory usage (both in a multi-core and an accelerator environment)**
  - Optimize malloc/free
  - Optimize the cache hierarchy
  - Forking is good; it may cut memory consumption in half
  - Don’t be afraid of threading; it may perform miracles !
  - Be aware of NUMA (Non-Uniform Memory Access): The “new” blessing (possibly a curse?)

C++ parallelization support

- **Large selection of tools (inside the compiler or as additions):**
  - Native: pthreads/Windows threads
  - Forthcoming C++ standard: std::thread
  - OpenMP
  - Intel Array Building Blocks (beta version from Intel; integrating RapidMind)
  - Intel Threading Building Blocks (TBB)
  - CUDA (from Nvidia) ← Not exactly C++, but…
  - MPI (from multiple providers), etc.

We must also keep a close eye on OpenCL (www.khronos.org/opencl)

Performance-oriented C++

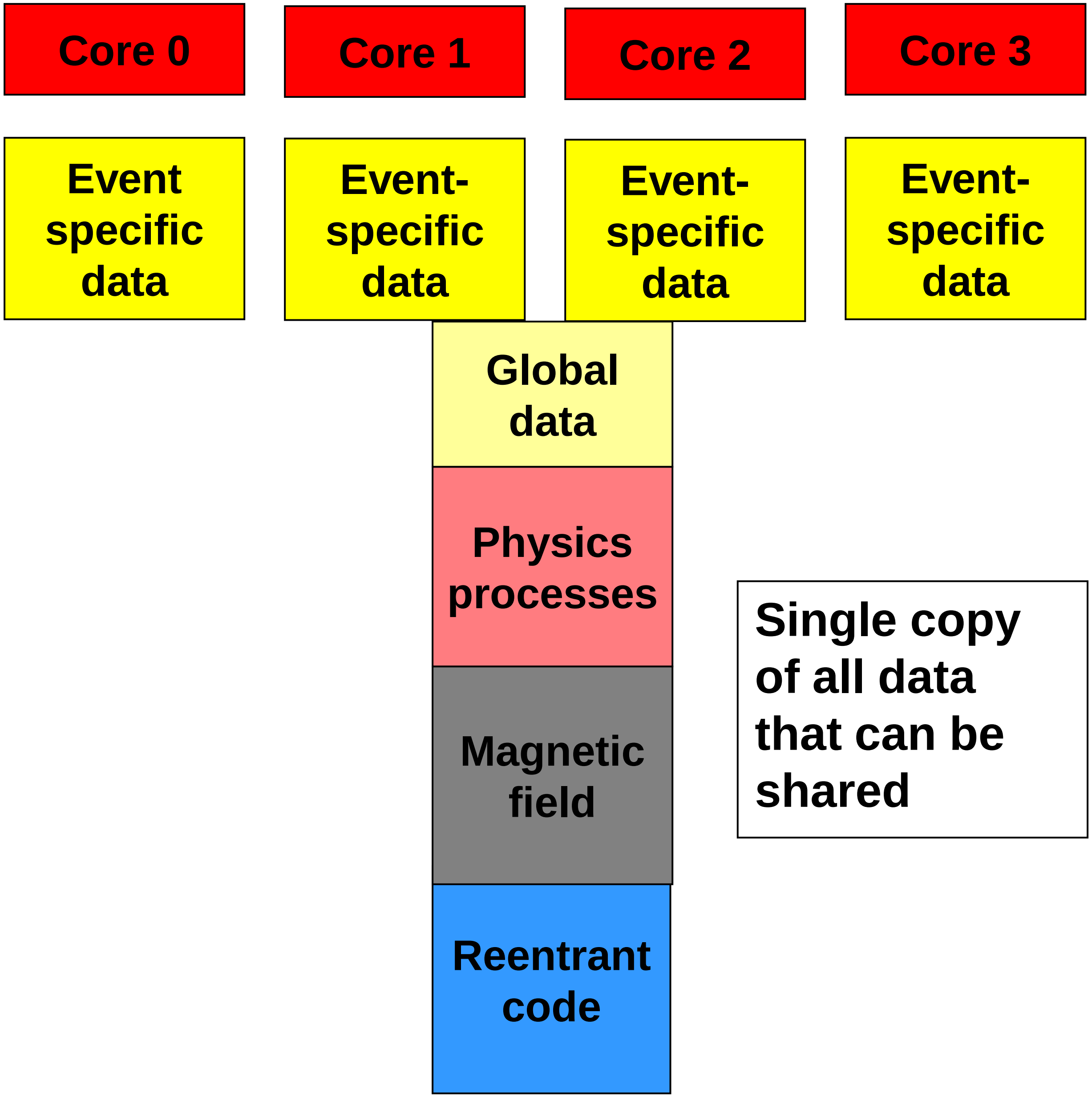
- Use light-weight C++ constructs
- Minimize virtual functions
- Inline whenever important
- Optimize the use of math functions
  - SQRT, DIV
  - LOG, EXP, POW
  - SIN, COS, ATAN2

Vectorize whenever possible

Learn how to inspect the compiler-generated assembly, especially of kernels

Achieving efficient memory footprint

- **As follows:**



Core 0

Core 1

Core 2

Core 3

Event specific data

Event-specific data

Event-specific data

Event-specific data

Global data

Physics processes

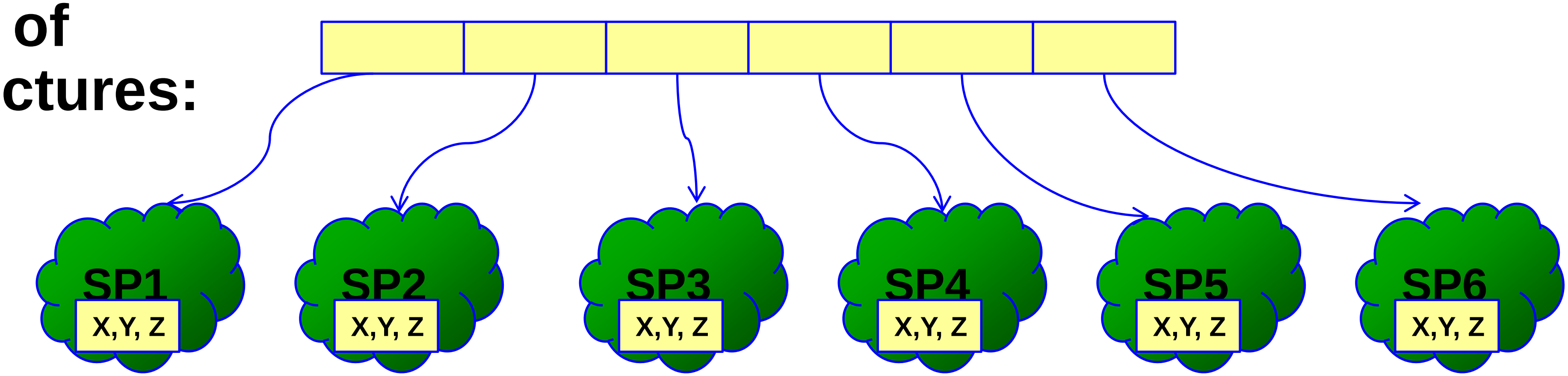
Magnetic field

Reentrant code

Single copy of all data that can be shared

Organization of data: AoS vs SoA

Arrays of Structures:



SP1 X,Y, Z

SP2 X,Y, Z

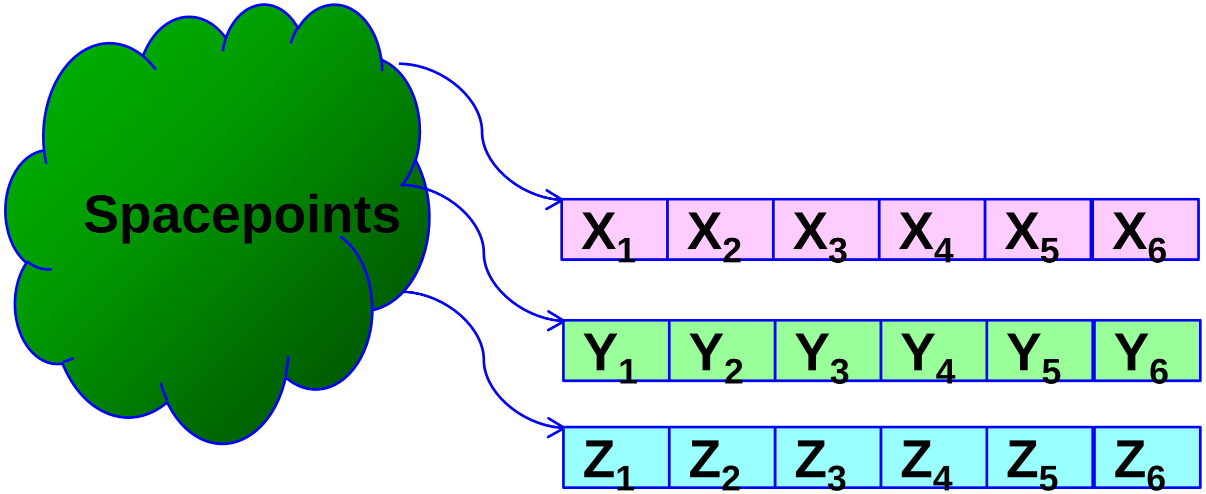
SP3 X,Y, Z

SP4 X,Y, Z

SP5 X,Y, Z

SP6 X,Y, Z

Structure of Arrays:



Spacepoints

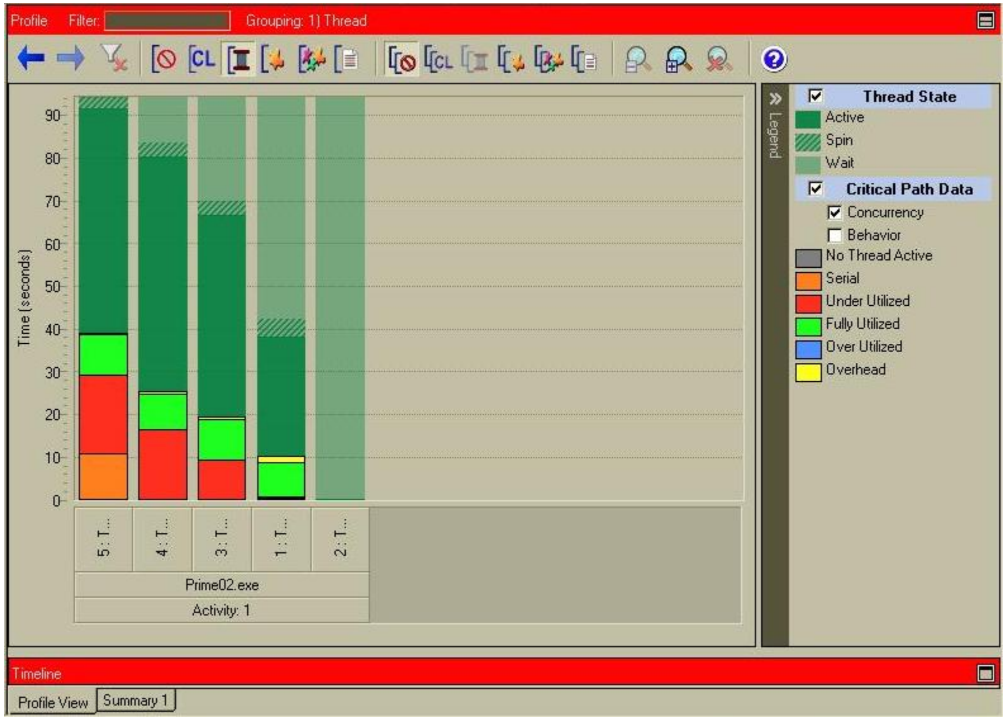
X_1 X_2 X_3 X_4 X_5 X_6

Y_1 Y_2 Y_3 Y_4 Y_5 Y_6

Z_1 Z_2 Z_3 Z_4 Z_5 Z_6

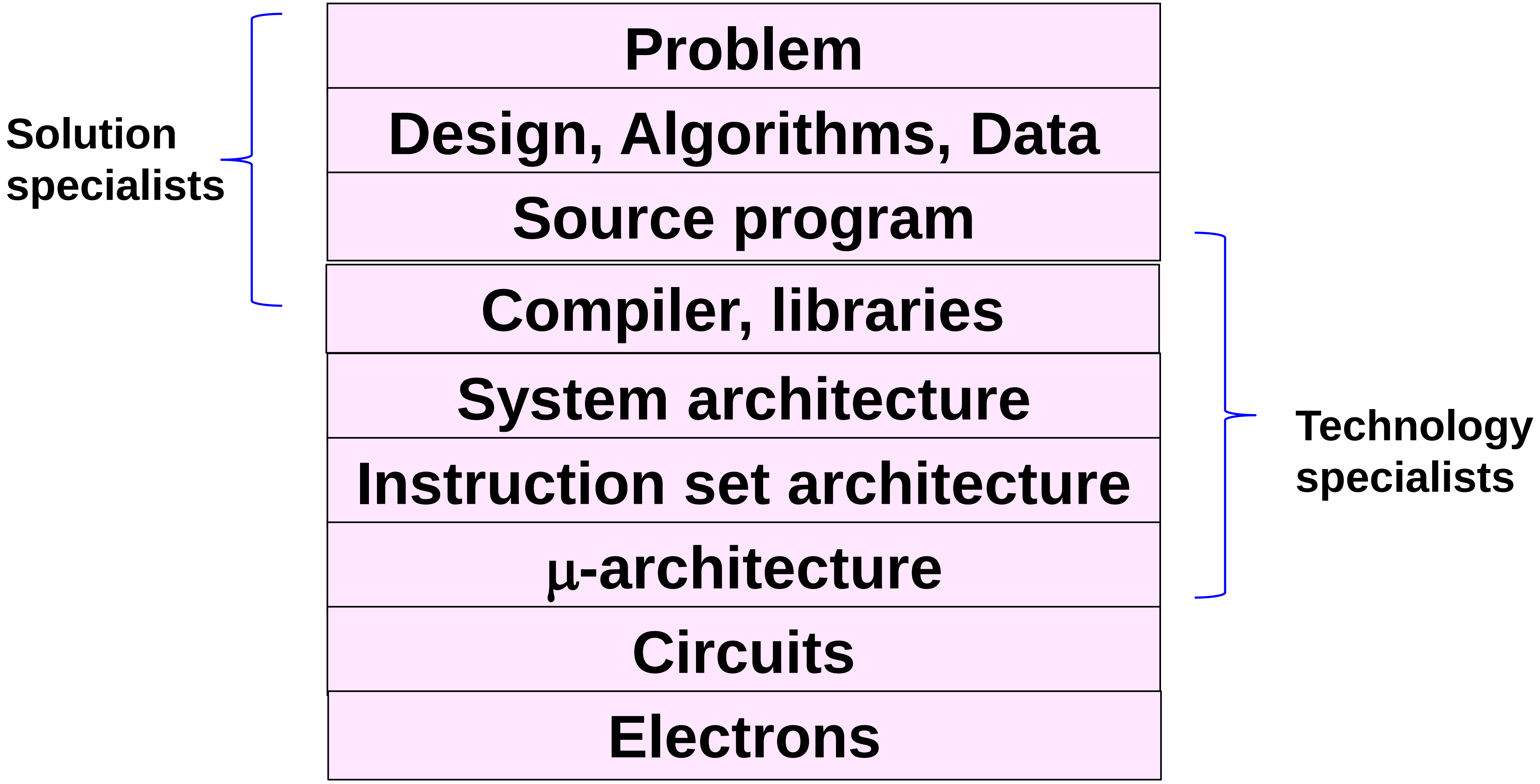
Performance guidance (cont’d)

- **Surround yourself with good tools:**
  - Compilers
  - Libraries
  - Profilers
  - Debuggers
  - Thread checkers
  - Thread profilers



Broad Programming Talent

- **In order to cover as many layers as possible**



Solution specialists

Problem

Design, Algorithms, Data

Source program

Compiler, libraries

System architecture

Instruction set architecture

μ-architecture

Circuits

Electrons

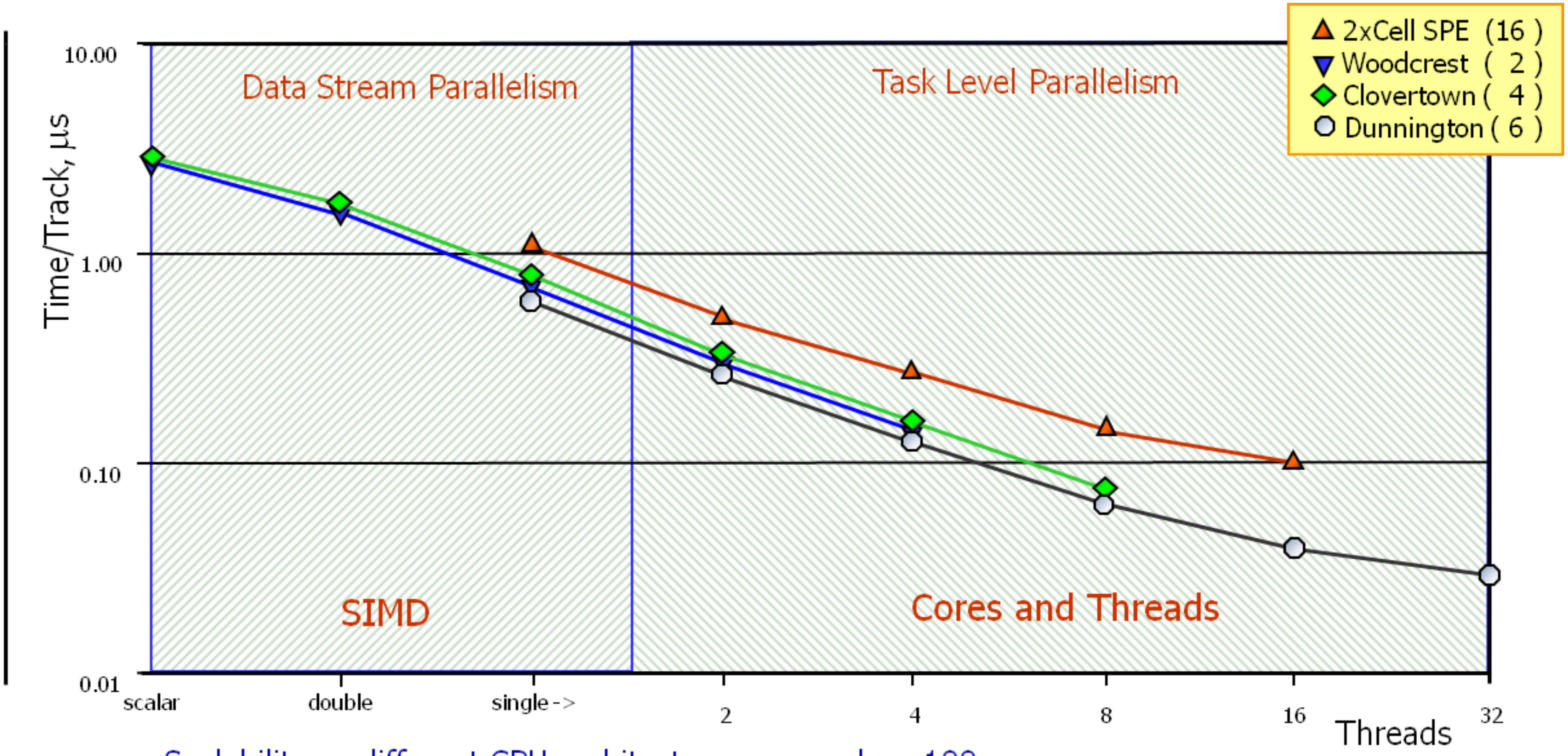
Technology specialists

Adapted from Y.Patt, U-Austin

HEP examples

Examples of parallelism: CBM/ALICE track fitting

- **Re-optimization on x86-64 systems**
  - Part1: Data parallelism using SIMD instructions
  - Part 2: use TBB (or OpenMP) to scale across cores



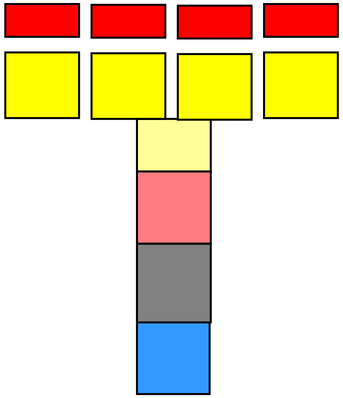
Scalability on different CPU architectures – speed-up 100

From H.Bjerke/CERN openlab, I.Kisel/GSI

Examples of parallelism: GEANT4

- **Initially: ParGeant4 (Gene Cooperman/NEU)**
  - implemented event-level parallelism to simulate separate events across remote nodes.

- **New prototype re-implements thread-safe event-level parallelism inside a multi-core node**
  - Done by NEU PhD student Xin Dong: Using FullCMS and TestEM examples
  - Required change of lots of existing classes (10% of 1 MLOC):
    - Especially *global*, “*extrn*”, and *static* declarations
    - Preprocessor used for automating the work.
  - Major reimplementation:
    - Physics tables, geometry, stepping, etc.

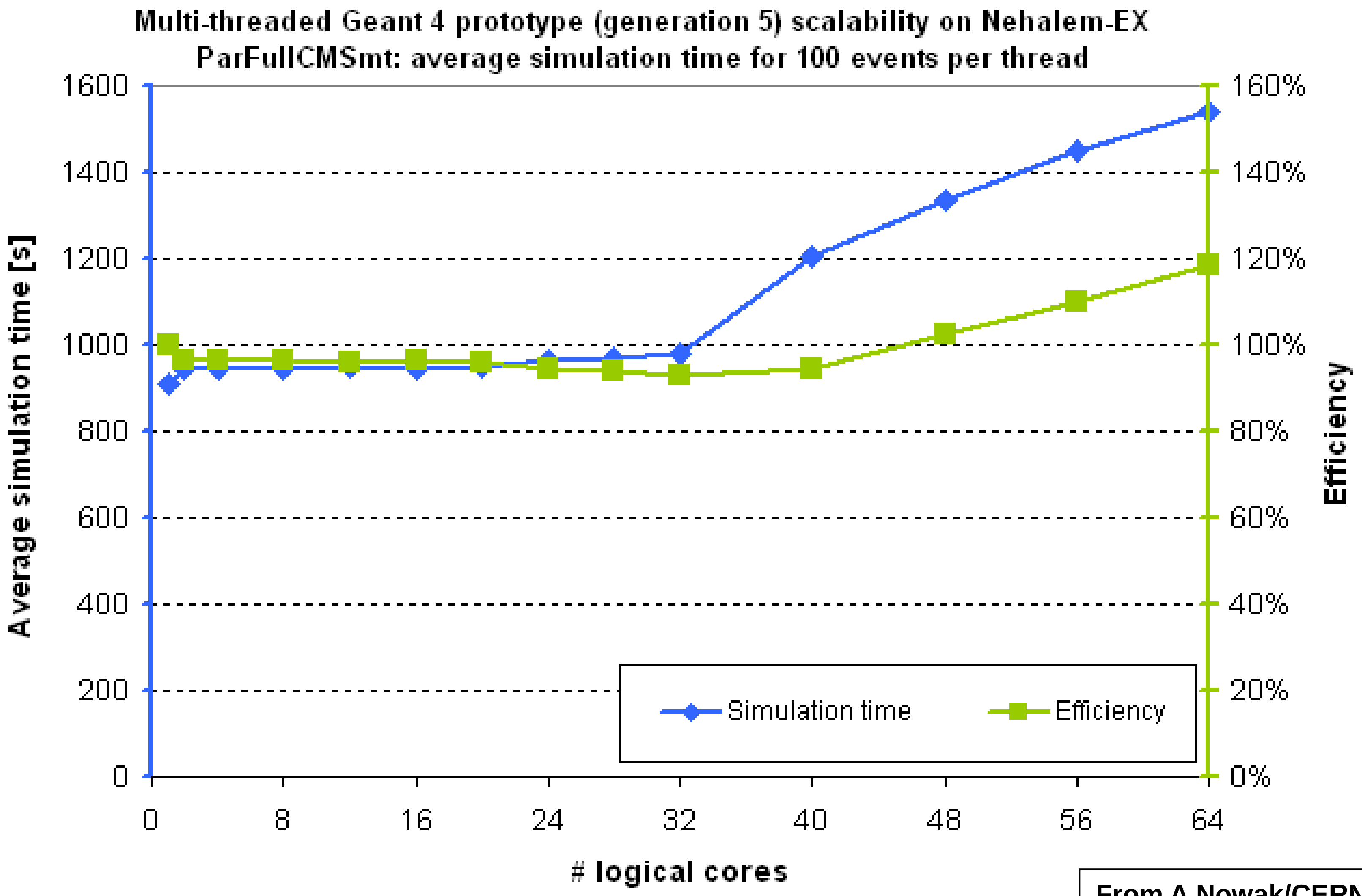


- **Additional memory: Only 25 MB/thread (!)**

Dong, Cooperman, Apostolakis: “Multithreaded Geant4: Semi-Automatic Transformation into Scalable Thread-Parallel Software”, Europar 2010

Multithreaded GEANT4 benchmark

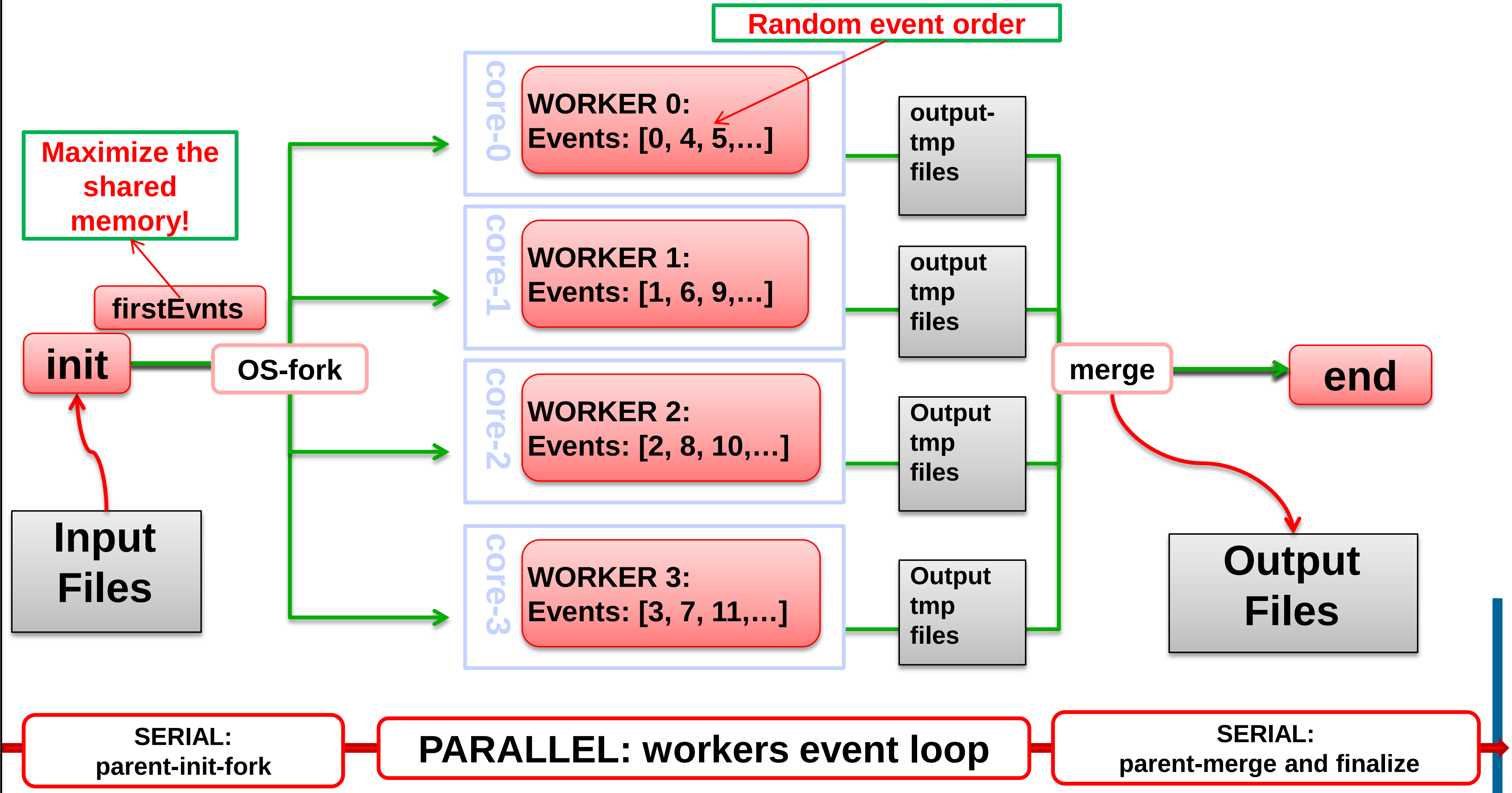
- **Excellent scaling on 32 (real) cores**
  - With a 4-socket server



From A.Nowak/CERN openlab

AthenaMP: event level parallelism

```
$> Athena.py  --nprocs=4  -c  EvtMax=100  Jobo.py
```



Random event order

Maximize the shared memory!

firstEvnts

init

OS-fork

Input Files

core-0

WORKER 0:
Events: [0, 4, 5,…]

core-1

WORKER 1:
Events: [1, 6, 9,…]

core-2

WORKER 2:
Events: [2, 8, 10,…]

core-3

WORKER 3:
Events: [3, 7, 11,…]

output-tmp files

output tmp files

Output tmp files

Output tmp files

merge

end

Output Files

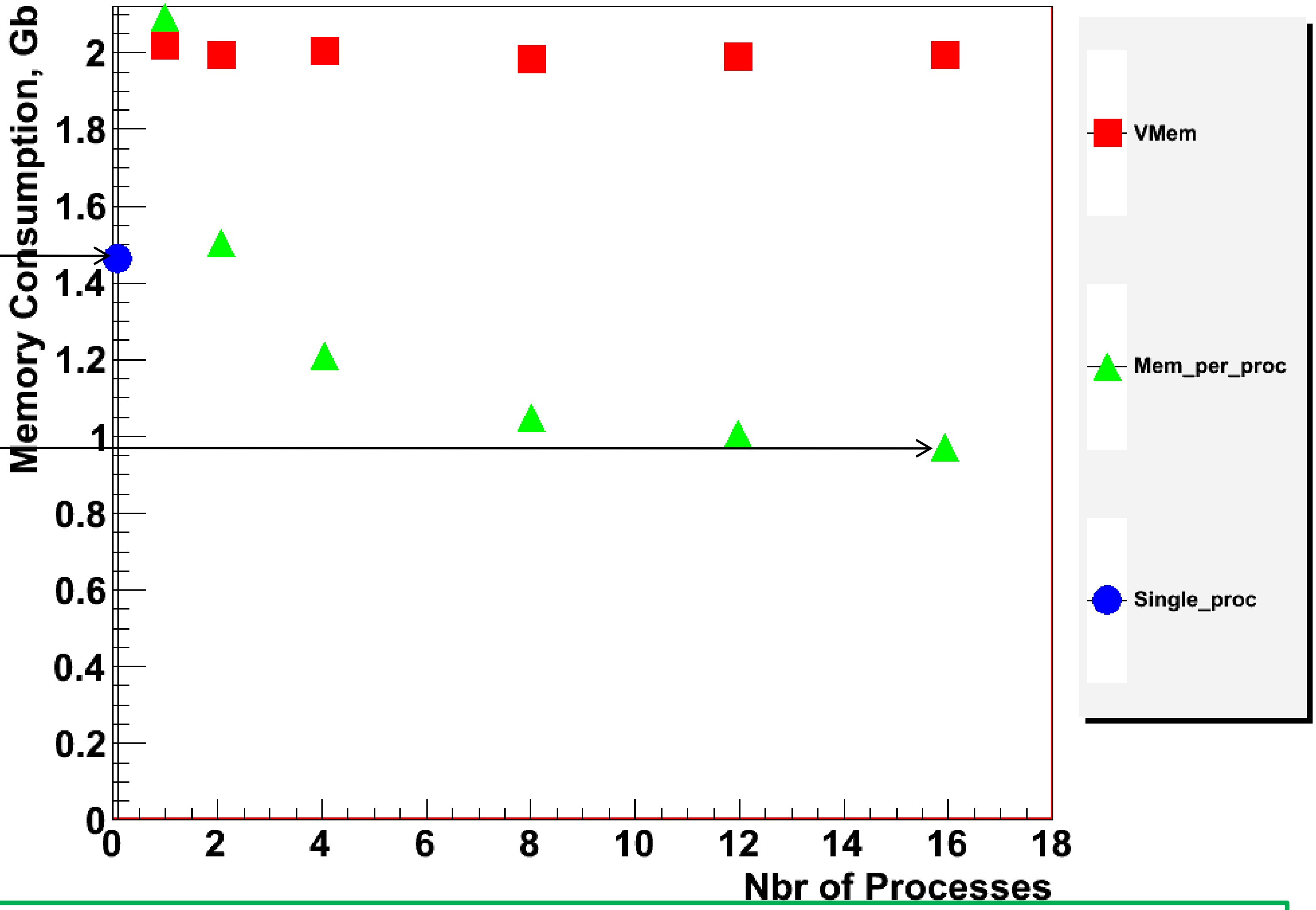
SERIAL:
parent-init-fork

PARALLEL: workers event loop

SERIAL:
parent-merge and finalize

From: Mous TATARKHANOV/May 2010

Memory footprint of AthenaMP



From
~1.5 GB

To
~1.0 GB

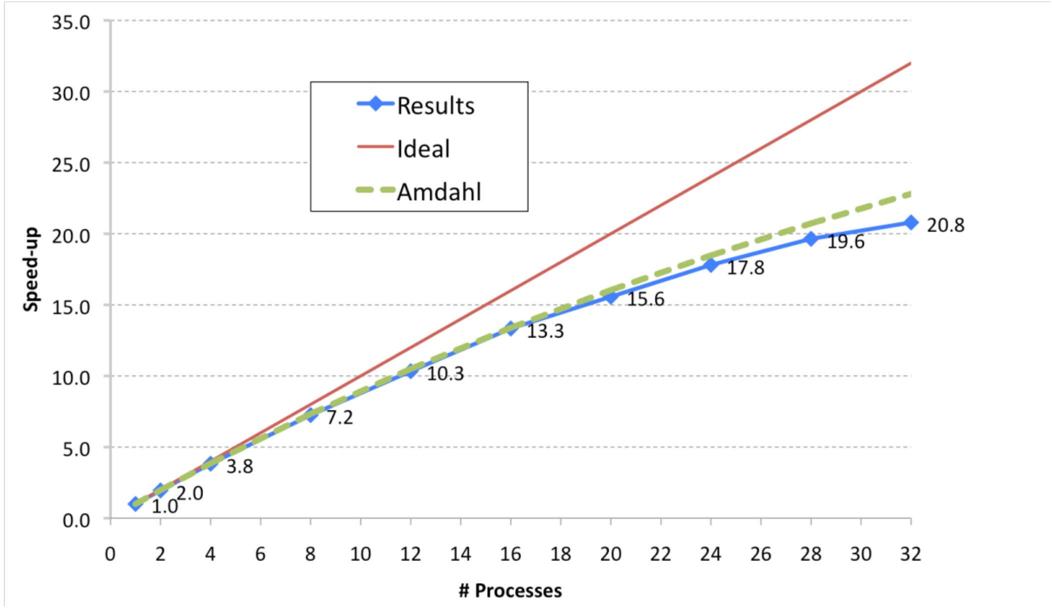
AthenaMP ~0.5 GB physical memory saved per process

From: Mous TATARKHANOV/May 2010

Example: ROOT minimization and fitting

- **Minuit parallelization is independent of user code**
- **Log-likelihood parallelization (splitting the sum) is quite efficient**

- **Example on a 32-core server:**



complex BaBar fitting provided by A. Lazzaro and parallelized using MPI

- **In principle, we can have combination of:**
  - parallelization via multi-threading in a multi-core CPU
  - multiple processes in a distributed computing environment

If you think that all of this is “crazy”

- **Please read:**
- **“Optimizing matrix multiplication for a short-vector SIMD architecture – CELL processor”**
  - J.Kurzak, W.Alvaro, J.Dongarra
  - Parallel Computing 35 (2009) 138–150

In this paper, single-precision matrix multiplication kernels are presented implementing the C = C – A x B^T operation and the C = C – A x B operation for matrices of size 64x64 elements. For the latter case, the performance of 25.55 Gflop/s is reported, or 99.80% of the peak, using as little as 5.9 kB of storage for code and auxiliary data structures.

Concluding remarks

- **The aim of these lectures was to help understand:**
  - Changes in modern computer architecture
  - Impact on our programming methodologies
  - Keeping in mind that there is not always a straight path to reach (all of) the available performance by our programming community.
- **In most HEP programming domains event-level processing will (continue to) dominate**
  - Provided we get the memory requirements under control
- **Will you be ready for 100+ cores and long vectors?**
  - Are you thinking “parallel, parallel, parallel” ?
- **It helps to know the seven hardware dimensions and how appropriate software constructs can assist you !**

Thank you!

Further reading:

- “Designing and Building Parallel Programs”, I. Foster, Addison-Wesley, 1995
- “Foundations of Multithreaded, Parallel and Distributed Programming”, G.R. Andrews, Addison-Wesley, 1999
- “Computer Architecture: A Quantitative Approach”, J. Hennessy and D. Patterson, 3rd ed., Morgan Kaufmann, 2002
- “Patterns for Parallel Programming”, T.G. Mattson, Addison Wesley, 2004
- “Principles of Concurrent and Distributed Programming”, M. Ben-Ari, 2nd edition, Addison Wesley, 2006
- “The Software Vectorization Handbook”, A.J.C. Bik, Intel Press, 2006
- “The Software Optimization Cookbook”, R. Gerber, A.J.C. Bik, K.B. Smith and X. Tian; Intel Press, 2nd edition, 2006
- “Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism”, J. Reinders, O’Reilly, 1st edition, 2007
- “Inside the Machine”, J. Stokes, Ars Technica Library, 2007

BACKUP I