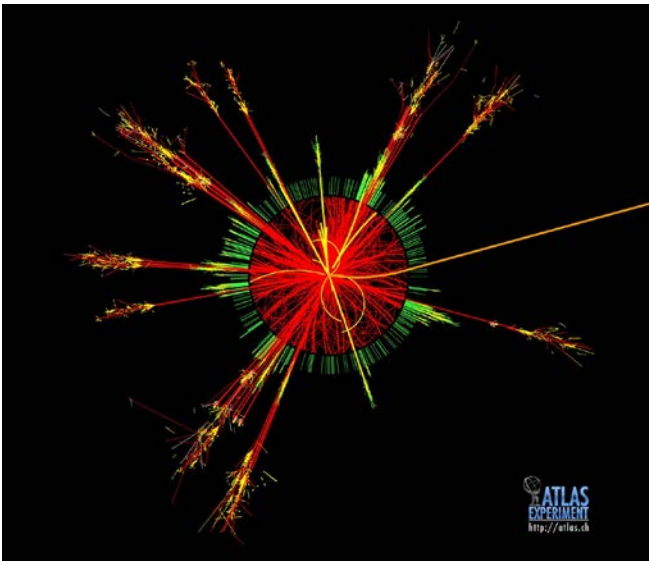


Computer Architecture and Performance Tuning

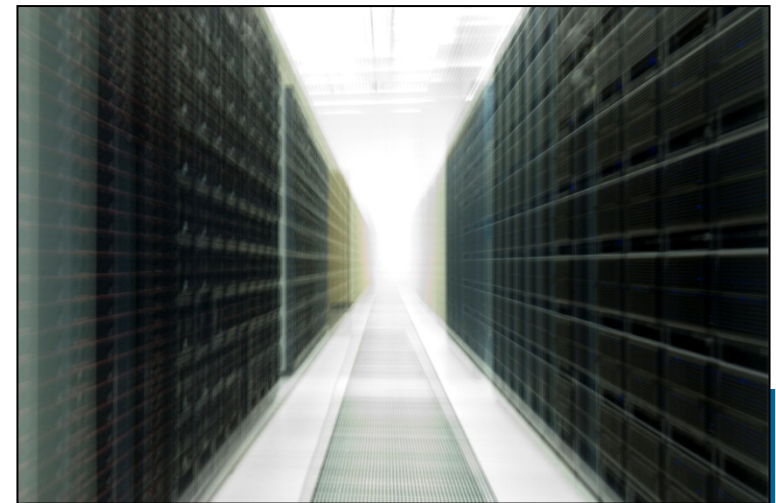
“Modern processors and related optimisation topics, including the seven Performance Dimensions”



Sverre Jarp
CERN
openlab
CTO

IT Dept.

CERN



Goal of this lecture series

- 1. Give an understanding of modern computer architectures from a performance point-of-view**
 - Processor, [Cache, Memory subsystem]
 - Use x86-64 as a de-facto standard
 - But keep an eye on GPUs/accelerators
- 2. Explain hardware factors that improve or degrade program execution speed**
 - Prepare for writing well-performing software
- 3. Teach an approach to detailed performance measurements**
 - Highlight the most important events for such measurements

Contents

- **Introduction:**
 - Setting the scene; Moore's "law"
 - Complexity in Computing
- **Basic Architecture**
- **Performance Dimensions:**
 - Vectorisation
 - Instruction level parallelism
 - Multi-core parallelisation
- **Memory subsystem**
- **Conclusion**

Setting the scene (1)

- Somebody offers you something worth a 100 € Nice!



Setting the scene (2)

- However, when you “measure” the real value of this gift



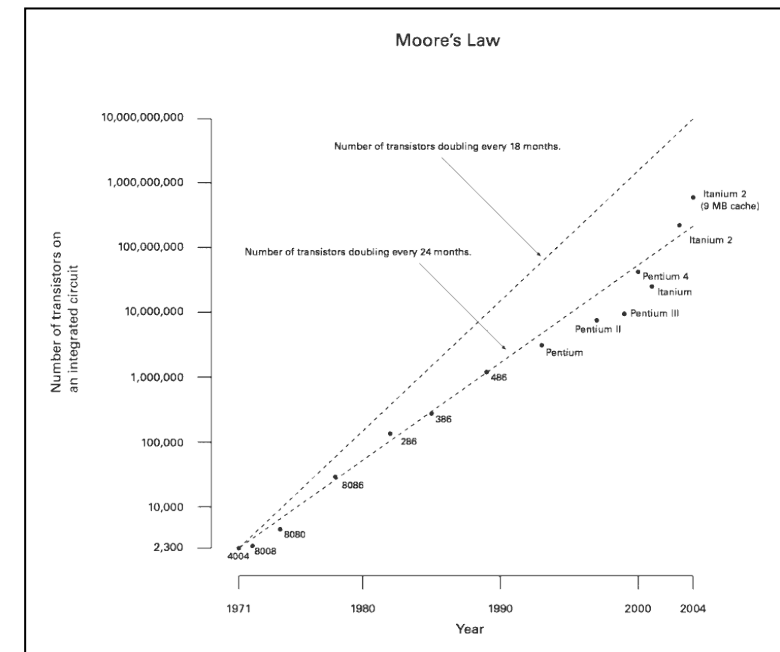
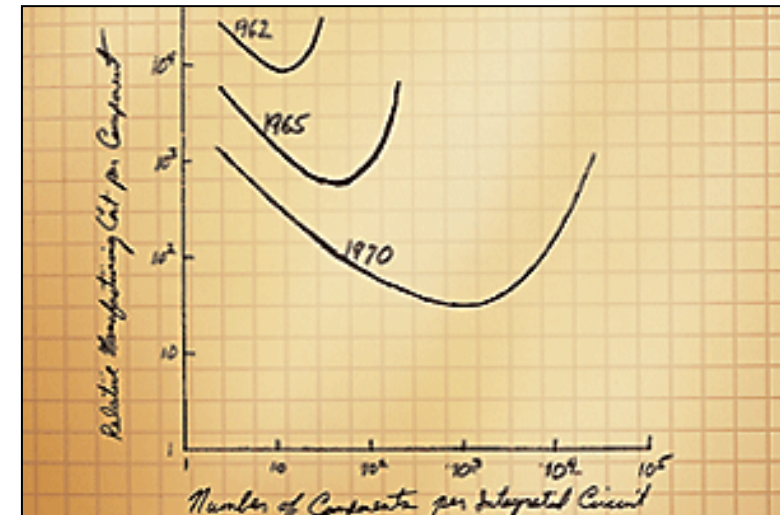
Why worry about performance?

- **My arguments:**

- **Performance per €:** There are important cost issues associated with large scale computing
 - Even when using “commodity equipment”
- **Performance per watt:** There are important thermal issues associated with large scale computing
 - Even when 1W processors exist!
- **The “easy ride” disappeared:** The frequency scaling we enjoyed in the past does not exist any longer. It stopped **10 years ago!**
 - ..and, as a “by-product”, computer architecture is becoming (much) more complicated

Moore's "law"

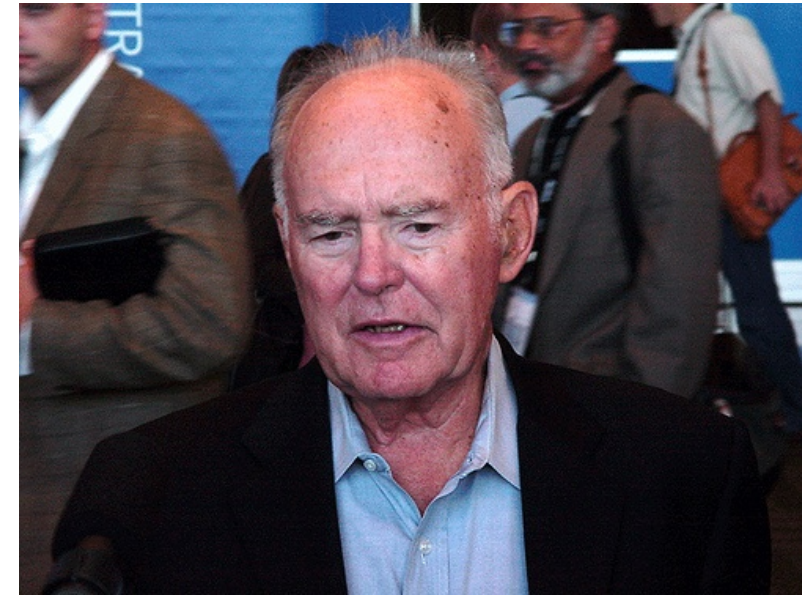
- A marching order established ~50 years ago
- "Let's continue to double the number of transistors every other year!"
- First published as:
 - Moore, G.E.: *Cramming more components onto integrated circuits*. Electronics, 38(8), April 1965.
- Accepted by all partners:
 - Semiconductor manufacturers
 - Hardware integrators
 - Software companies
 - Us, the consumers



From Wikipedia

Moore's "law" (cont'd)

- **The consequences: An incredible level of integration**
 - CPUs: Many-core, Hardware vectors, Hardware threading
 - GPUs: Enormous number of floating-point units
- **Today, we commonly acquire chips with 1'000'000'000 (10^9) transistors!**
 - Server chips and high-end GPU devices have much more
 - Kepler GK110:
 - 7.1 billion transistors



From Wikipedia

Real consequence of Moore's law

- We are being “**snowed under**” 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

“Data Oriented Design”

Semiconductor evolution

- **Today's silicon processes:**

- 32, 28, 22 nm

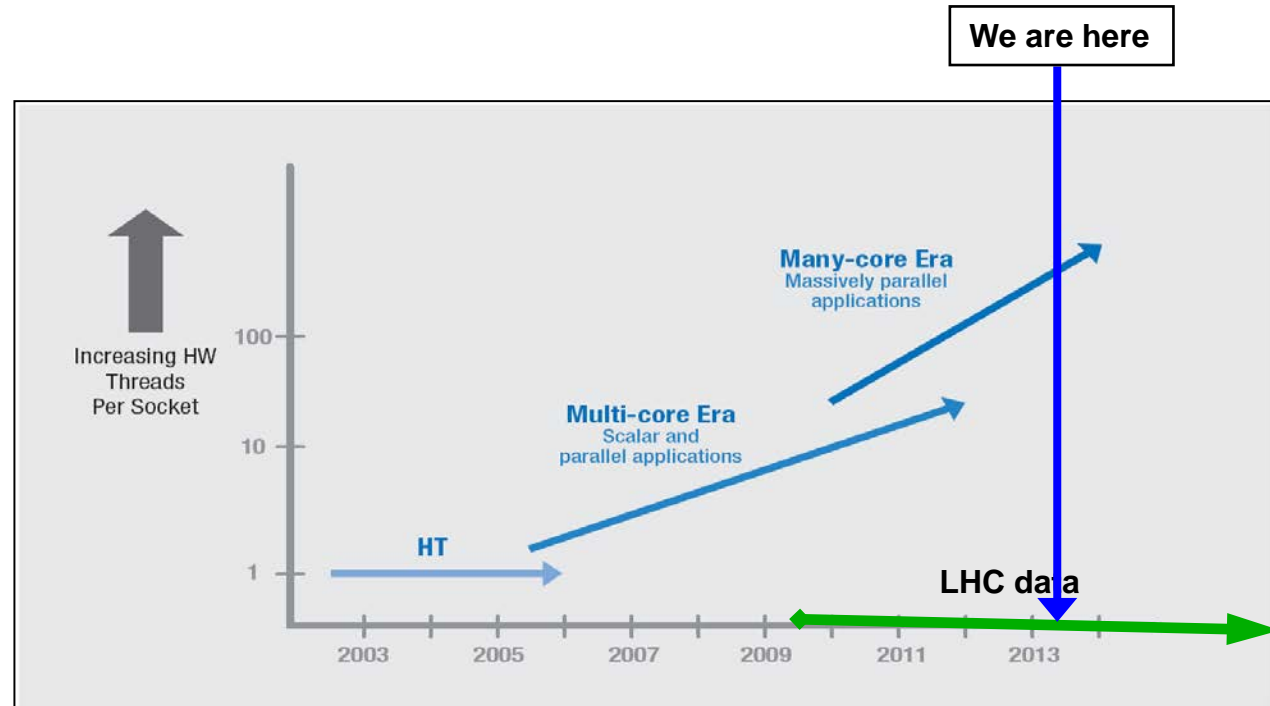
- **Being introduced:**

- 14 nm (2013/14)

- **In research:**

- 10 nm (2015/16)
- 7 nm (2017/18)
- 5 nm (2019/20)

– Source: Intel



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

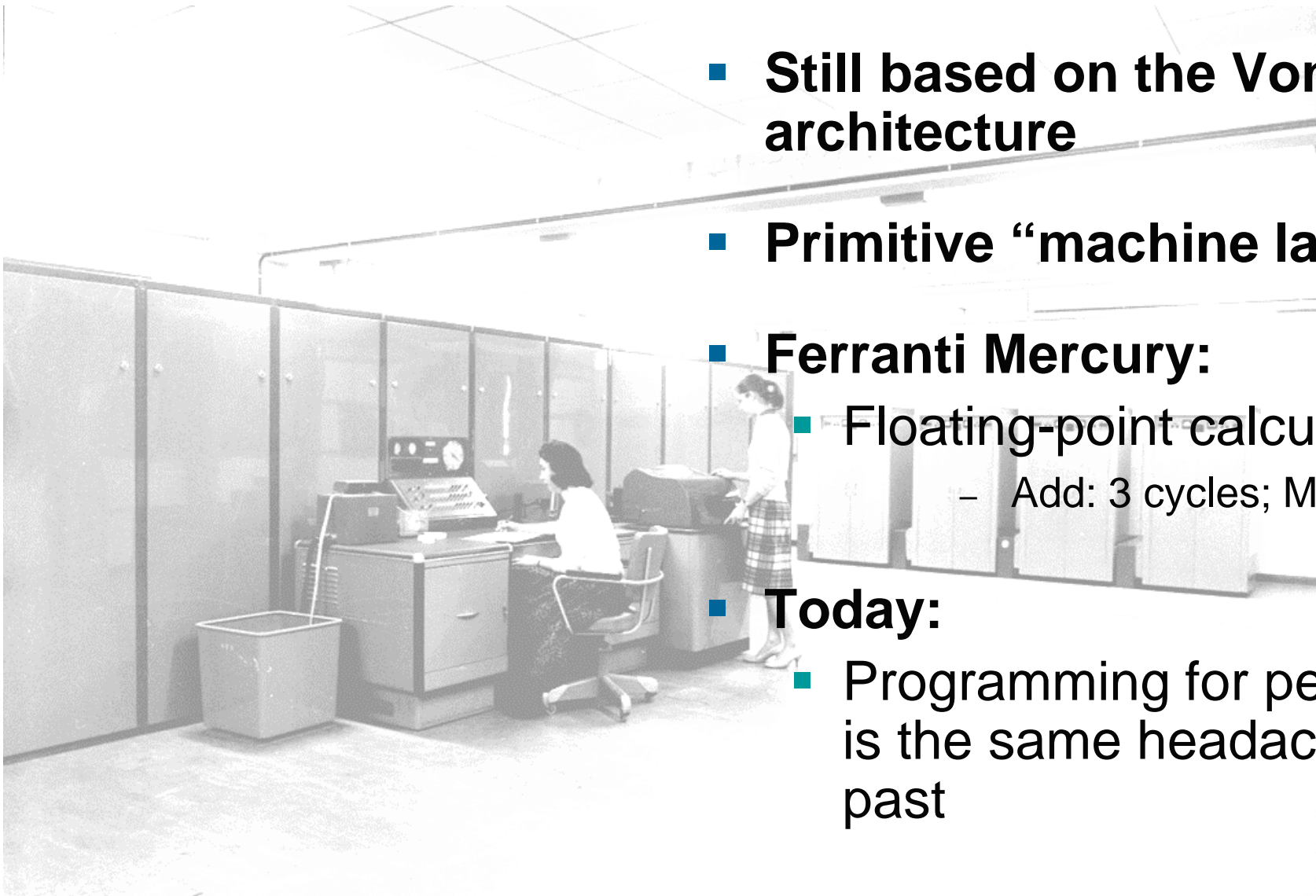
← 2 nm (2028?) TSMC

- **By the end of this decade we will have chips with**
~**100'000'000'000** (10^{11}) transistors!

Complexity in Computing

Archaic Computing Units

- 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:
 - Programming for performance is the same headache as in the past



And the language is ancient, too!

■ Assembly/machine code!

__Z6matmulv (snippet):

```

vmovlhps    %xmm0, %xmm3, %xmm3
vmovss     +_b(%rip), %xmm4
vinserftf128 $1, %xmm3, %ymm3, %ymm3
vinserftps  $0x10, 44+_b(%rip), %xmm7, %xmm7
vmovss      48+_b(%rip), %xmm6
vinserftps  $0x10, 36+_b(%rip), %xmm1, %xmm2
vmovlhps    %xmm0, %xmm2, %xmm2
vinserftps  $0x10, 60+_b(%rip), %xmm4, %xmm0
vxorps      %xmm4, %xmm4, %xmm4
vinserftf128 $1, %xmm2, %ymm2, %ymm2
vinserftps  $0x10, 52+_b(%rip), %xmm6, %xmm1
vmovlhps    %xmm0, %xmm1, %xmm1
vmovaps     _a(%rip), %ymm0
vinserftf128 $1, %xmm1, %ymm1, %ymm1
vpermilps   $0, %ymm0, %ymm7
vmulps      %ymm5, %ymm7, %ymm7
vaddps      %ymm4, %ymm7, %ymm7
vpermilps   $85, %ymm0, %ymm6

```

Good or Bad?

Even assembly is “too high level”

- Intel translates “CISC” x86 assembly instructions
 - into “RISC” micro-operations
 - which can vary with each CPU generation
- NVIDIA translates PTX (parallel thread execution, or virtual assembly)
 - into machine instructions
 - which can vary with each GPU generation
- So, what does it really mean (?) when the hardware tells you:
 - “ XX^N instructions executed”

CISC: Complex Instruction Set Computing

RISC: Reduced Instruction Set Computing

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 clue as to the efficiency of this transformation process**

A Complicated Story (in 9 layers!)

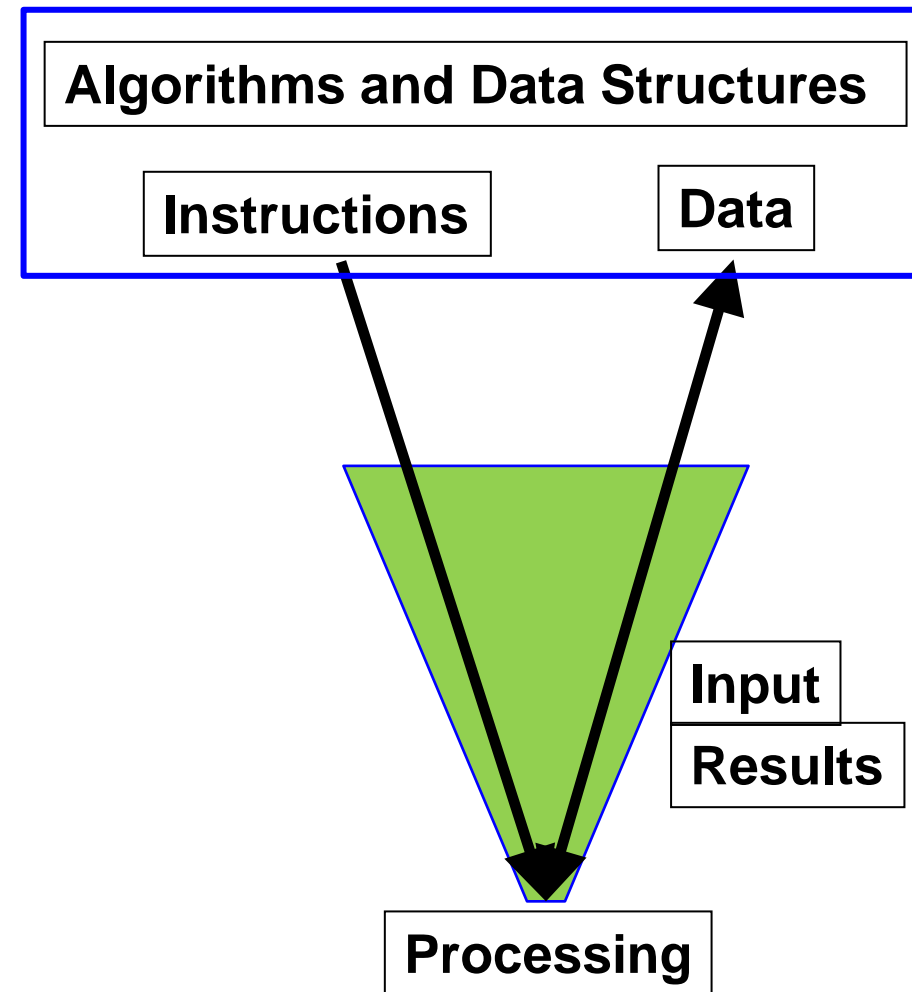
- Computing problems are solved by getting electrons to “dance”

Problem
Design, Algorithms, Data
Language, Source program
Compilers, Libraries
System architecture
Instruction set architecture
μ-architecture
Circuits
Electrons

But, let's start with the basics!

Von Neumann architecture

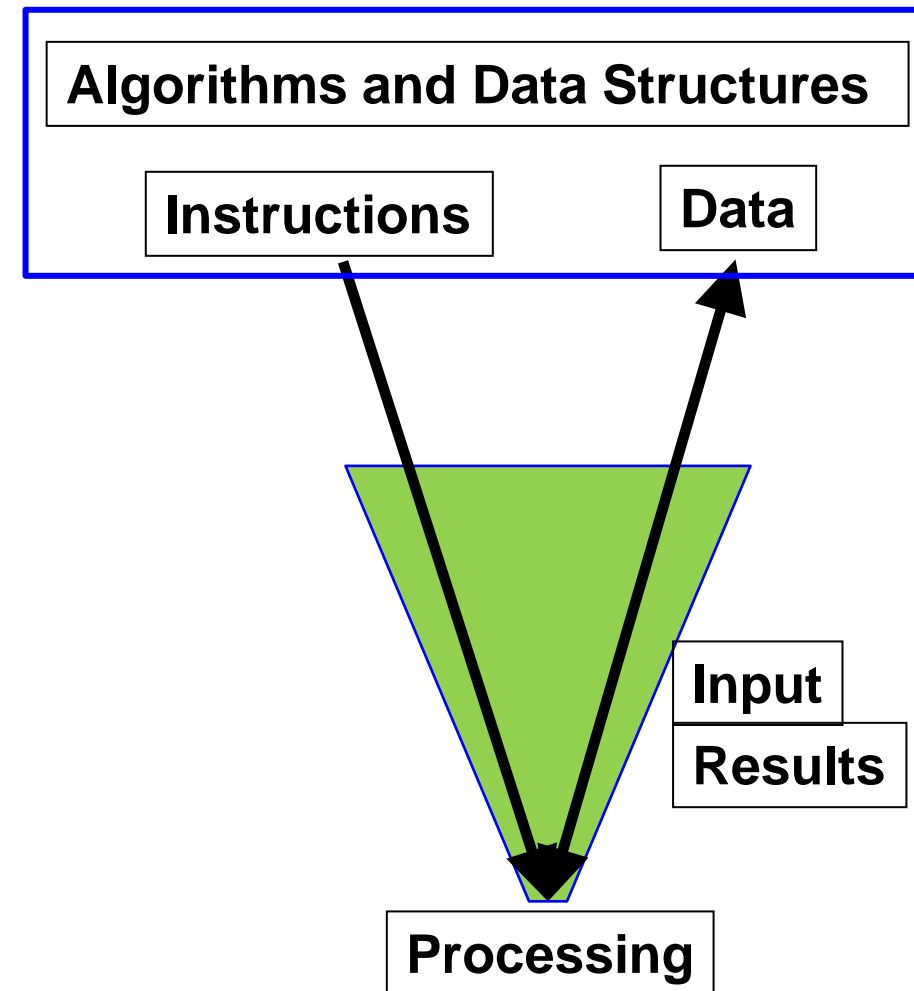
- **From Wikipedia:**
 - The von Neumann architecture is a computer design model that uses a processing unit and a single separate storage structure to hold both instructions and data.
- **It can be viewed as an entity into which one streams instructions and data in order to produce results**



**Some people think the architecture is out-dated.
But nobody has replaced it (yet)**

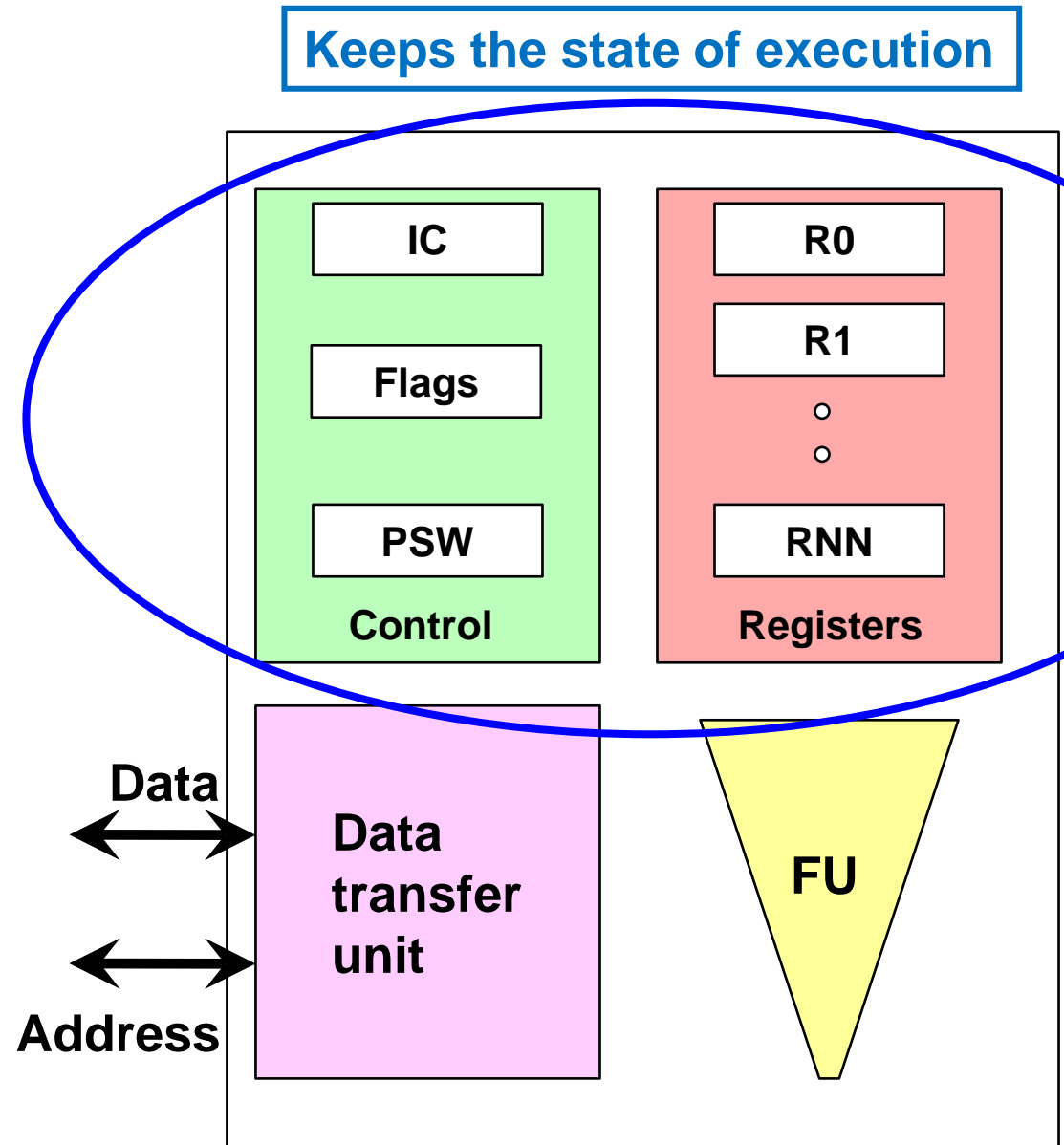
Von Neumann architecture (cont'd)

- The goal is to produce results as fast as possible
- But, lots of problems can occur:
 - Instructions or data don't arrive in time
 - Bandwidth issues?
 - Latency issues?
 - Clashes between input data and output data
 - Other “complexity-based” problems inside an extreme processing parallelism



Simple processor layout

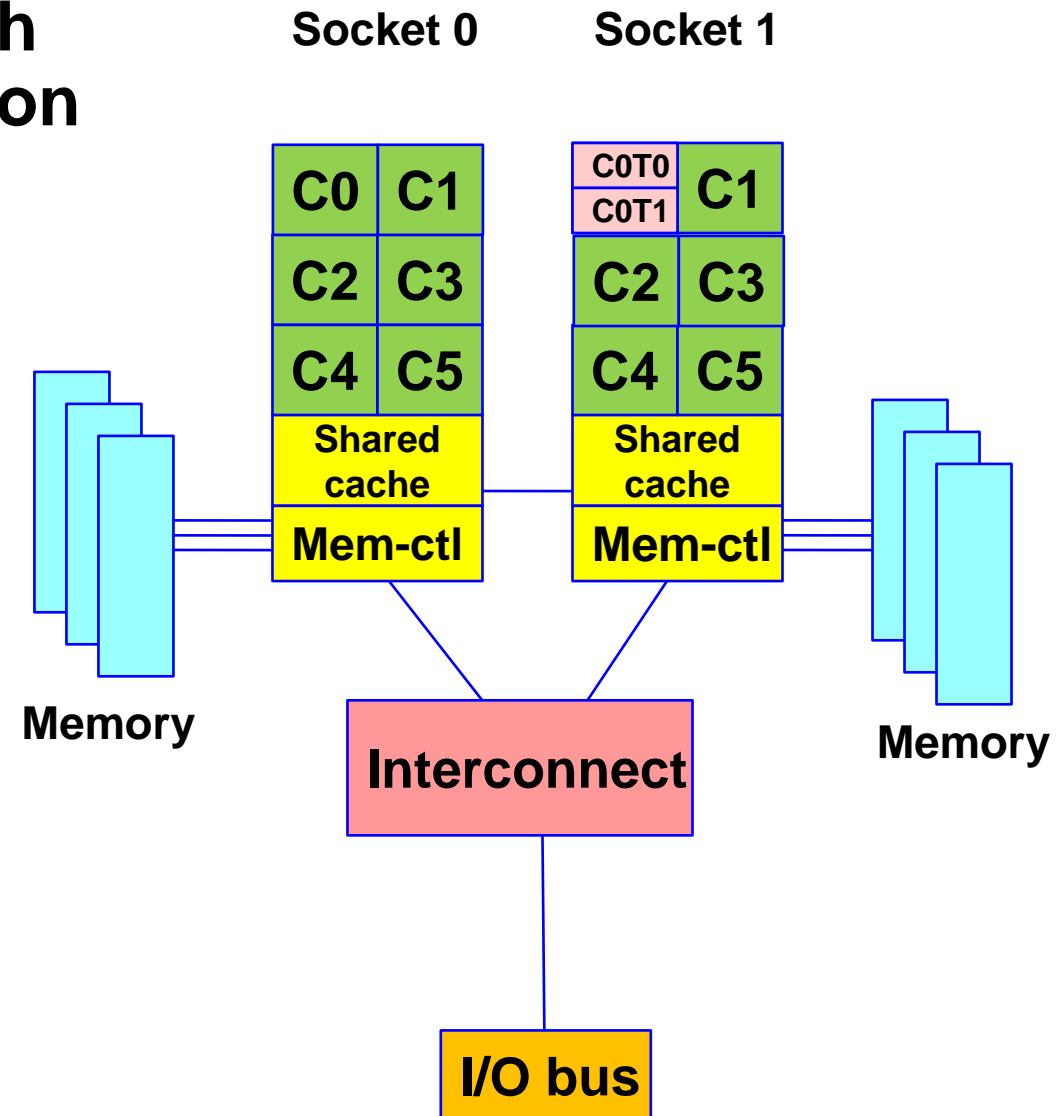
- A simple processor with four key components:
 - Control Logic
 - Instruction Counter
 - Program Status Word
 - Register File
 - Functional Unit
 - Data Transfer Unit
 - Data bus
 - Address bus



Simple server diagram

- **Multiple components which interact during the execution of a program:**

- Processors/cores
 - w/private caches
 - I-cache, D-cache
- Shared caches
 - Instructions and Data
- Memory controllers
- Memory (non-uniform)
- I/O subsystem
 - Network attachment
 - Disk subsystem



Performance Dimensions

In the days of the Pentium

- **Life was really simple:**

- Basically two dimensions
 - The frequency of the pipeline
 - The number of boxes
- The semiconductor industry increased the frequency
- We acquired the right number of (single-socket) boxes

Pipelining

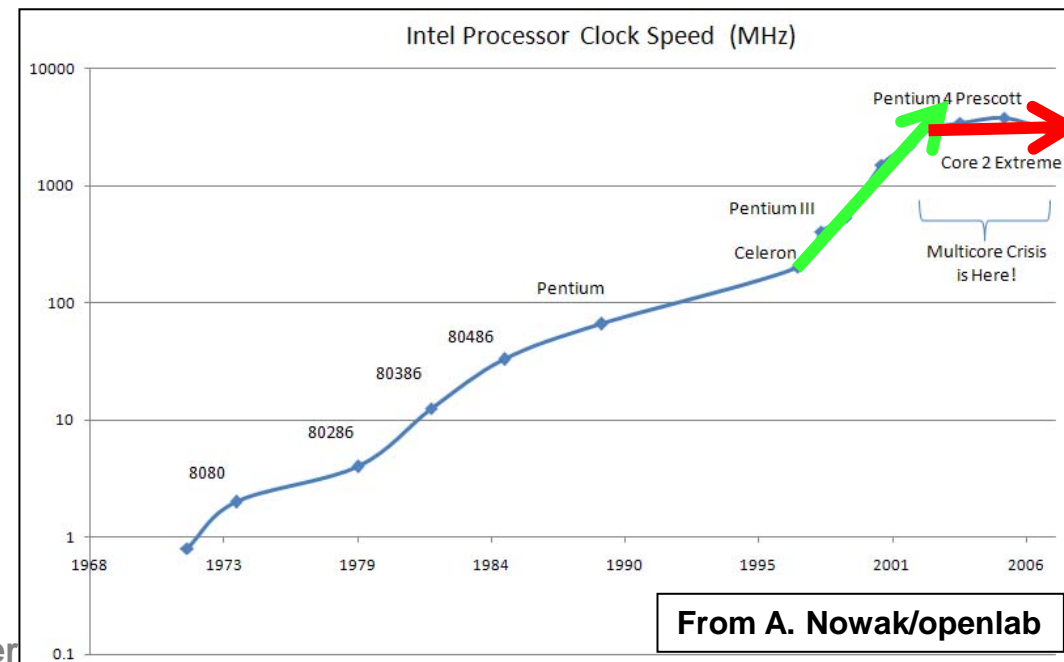
Superscalar

Nodes

Sockets

Frequency scaling

- **The 7 “fat” years of 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 E5-2630L
 - “only” 2.00 GHz



Now: Seven dimensions of performance

- **First three dimensions:**

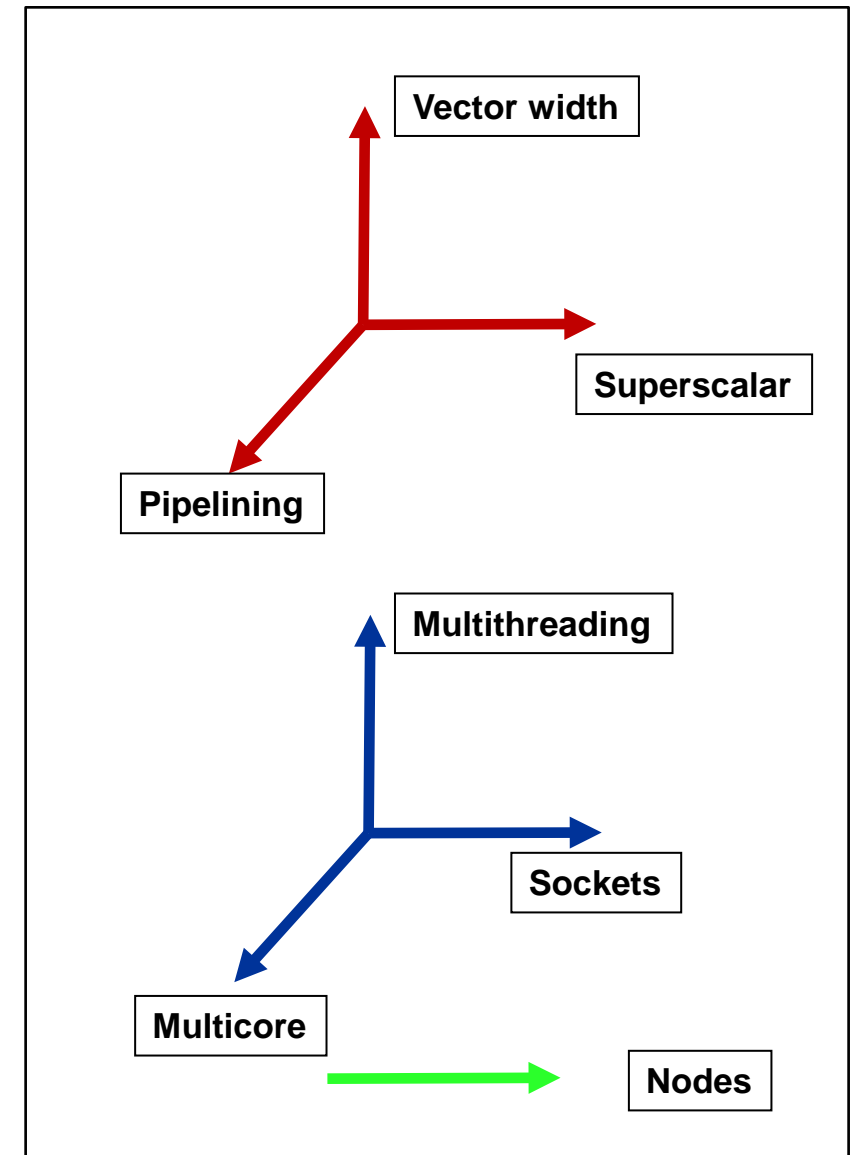
- Hardware vectors/SIMD
- Superscalar
- Pipelining

- **Next dimension is a “pseudo” dimension:**

- Hardware multithreading

- **Last three dimensions:**

- Multiple cores
- Multiple sockets
- Multiple compute nodes



Seven multiplicative dimensions:

■ First three dimensions:

- Hardware vectors/SIMD
- Superscalar
- Pipelining

**Data and Instruction
Level parallelism
(Vectors/Matrices)**

■ Next dimension is a “pseudo” dimension:

- Hardware multithreading

**Task parallelism
(Events/Tracks)**

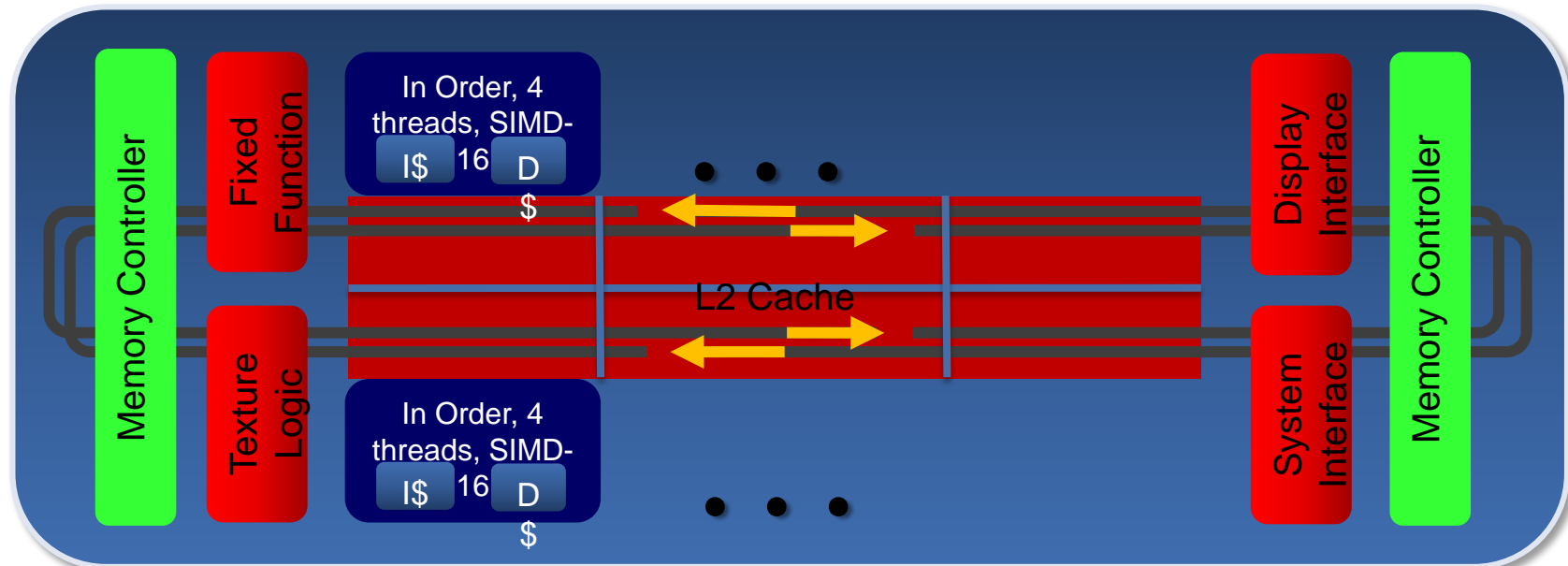
■ Last three dimensions:

- Multiple cores
- Multiple sockets
- Multiple compute nodes

**Task/process
parallelism**

Simple, but illustrative example

- Xeon Phi has 60 cores, 4-way hardware threading, hardware vectors of size 8 (Double Precision):
- Program A: Threaded 60 x 4, vectorised 8x:
 - Performance potential: 1920
- Program B: Not threaded: 1x, not vectorised: 1x
 - Performance potential: 1



GPUs: 7 dimensions of performance

- **First four dimensions:**

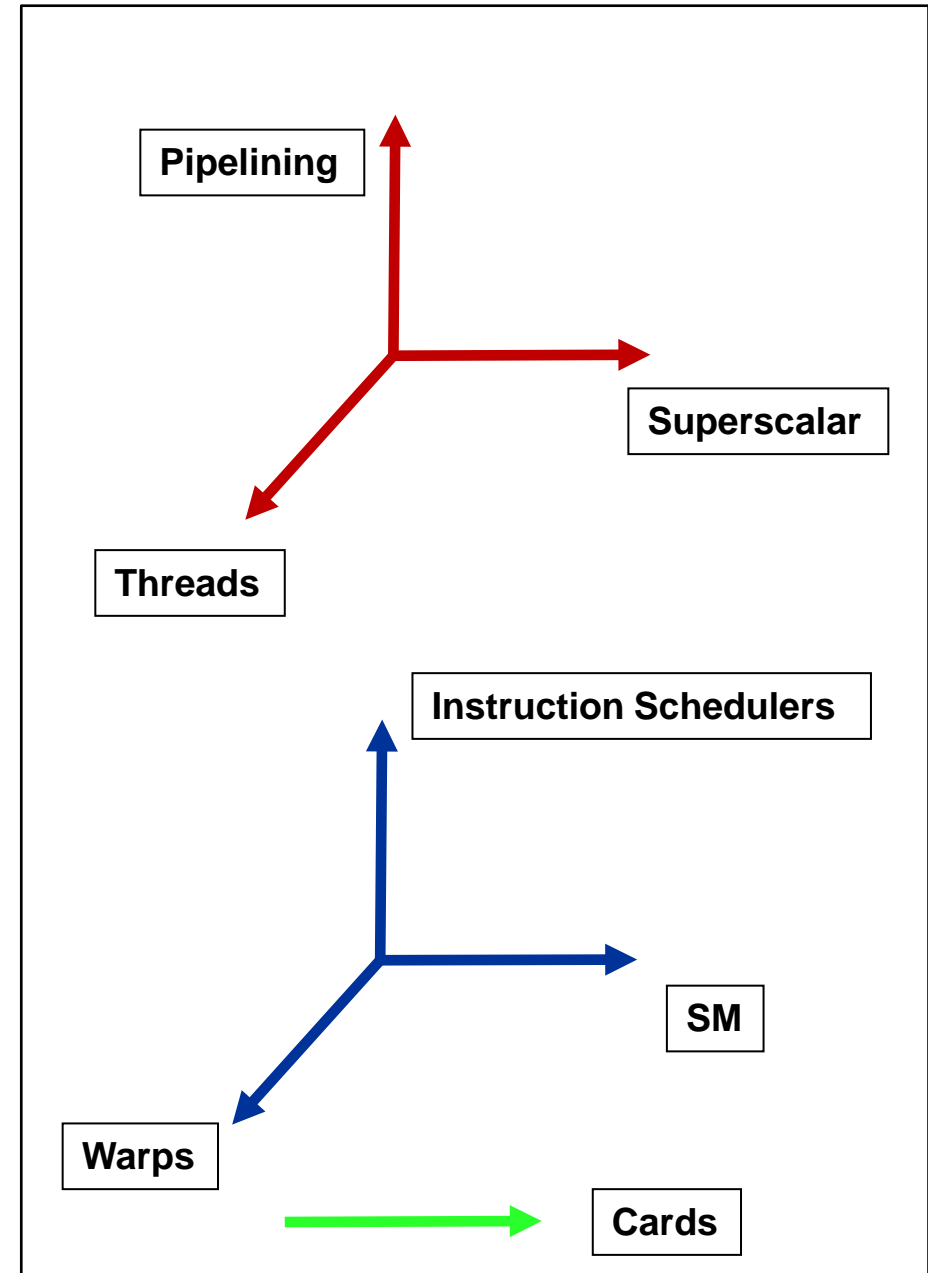
- Superscalar (dual issue)
- Pipelining
- Threads (32)
- Instruction Schedulers (4)

- **Then, there are:**

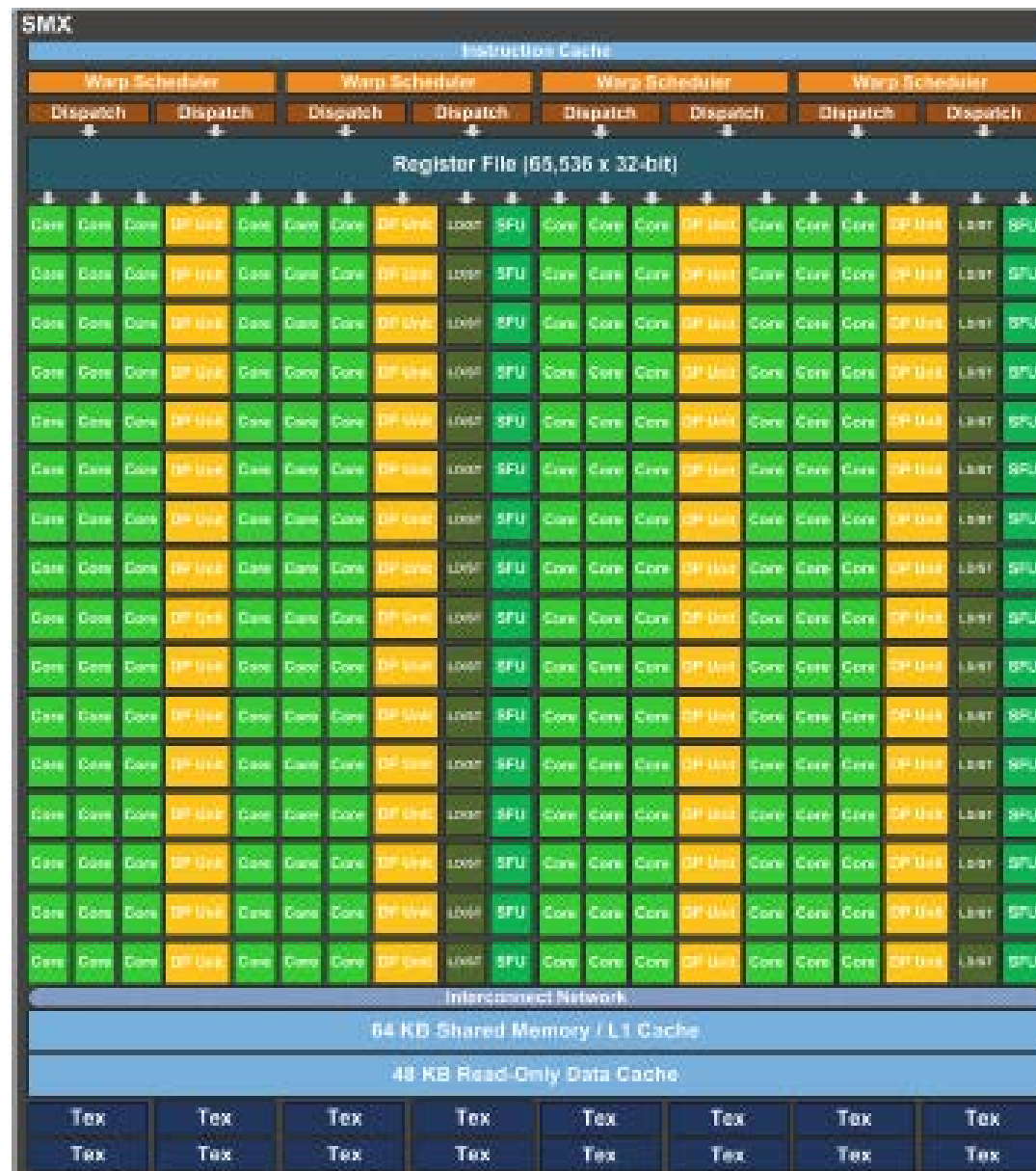
- Warps

- **Last dimensions:**

- Multiple SMs
- Multiple accelerators



Streaming Multiprocessor Architecture

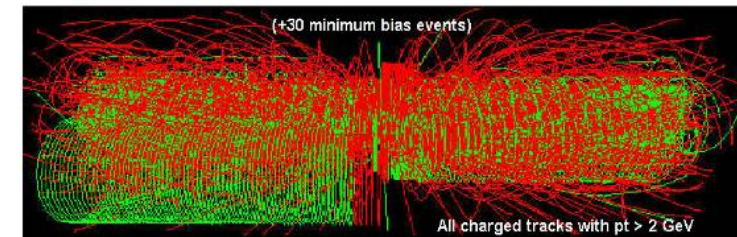
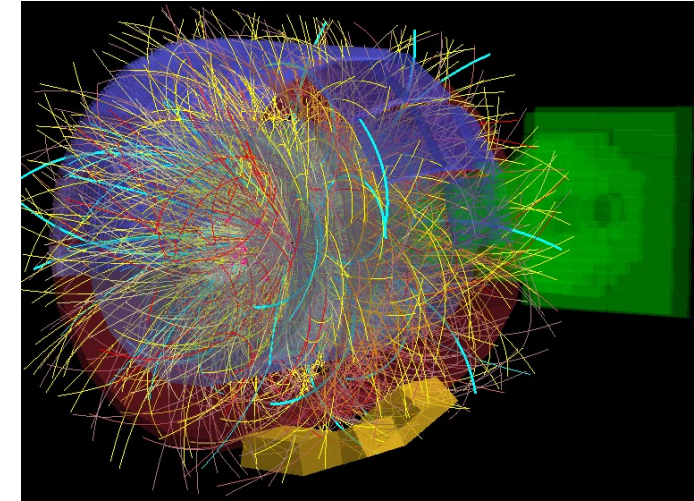


SMX: 192 single-precision CUDA cores, 64 double-precision units, 32 special function units (SFU), and 32 load/store units (LD/ST).

Source: NVIDIA white paper

Concurrency in High Energy Physics

- We are “blessed” with lots of it:
 - Entire events
 - Particles, hits, 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 the “first-generation” software frameworks**

Autoparallelization/Autovectorization

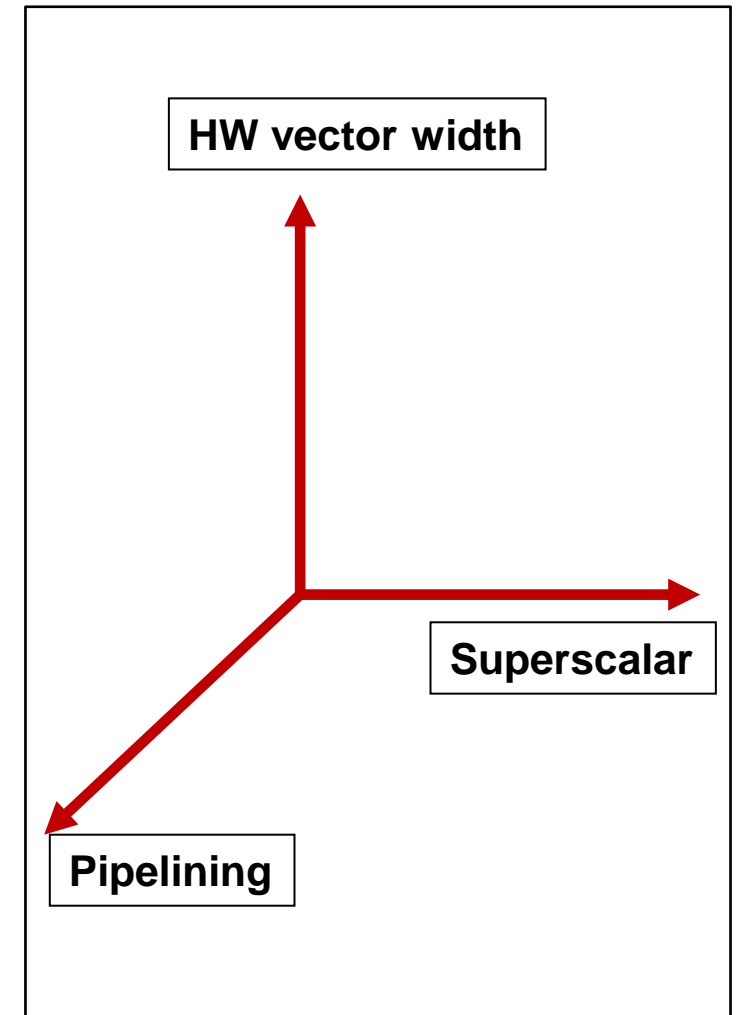
- **Would it not be wonderful if the compilers could do all the (vectorization/parallelisation) work automatically?**
- **GNU compiler (4.3.0 or later):**
 - Autovectorization: **YES**, but needs “-ftree-vectorize”
 - “-ftree-vectorizer-verbose=[0-7]” for reports
 - Autoparallelization support in preparation
 - 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

Use “-guide” for both scenarios to get advice [“Guided Auto Parallelisation” or “GAP”]

Autovectorization is beginning to look serious in recent compiler versions!

Part 1: Opportunities for scaling performance inside a core

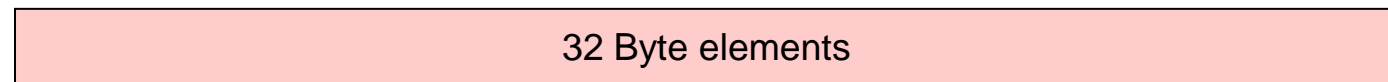
- Here are the first three dimensions
- The resources:
 - HW vectors: Fill the computational width
 - Superscalar: Fill the ports
 - Pipelining: Fill the stages
- Best approach: Data Oriented Design
- In HEP today, we extract less than **10%** of peak execution capability!



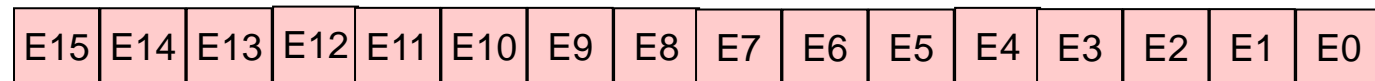
First topic: Vector registers

- **Until recently, Streaming SIMD Extensions (SSE):**
 - 16 “XMM” registers with 128 bits each (in 64-bit mode)
- **New (as of 2011): Advanced Vector eXtensions (AVX 1):**
 - 16 “YMM” registers with 256 bits each

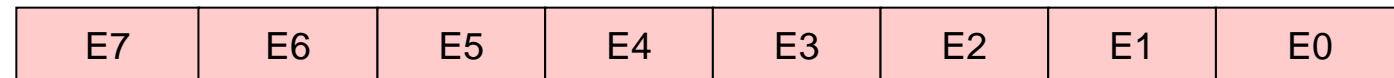
32 Bytes



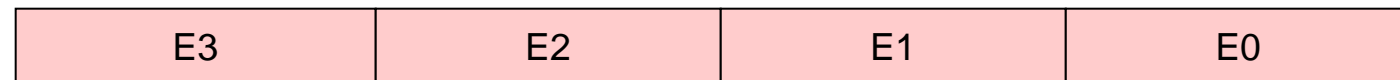
16 Words



8 Dwords/**Single**



4 Qwords/**Double**



Bit 255

Bit 0

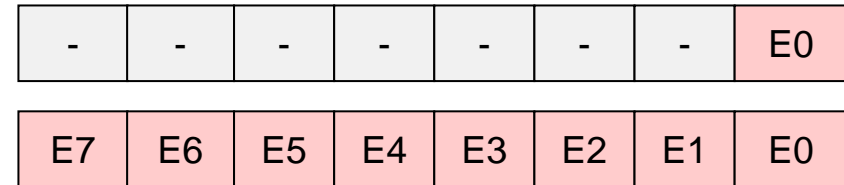
128 bits

256 bits (AVX 1)

Four floating-point data flavours

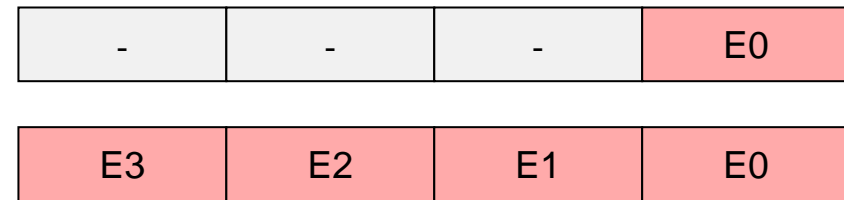
■ Single precision

- Scalar single (SS)
- Packed single (PS)



■ Double precision

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



■ 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 coming!~~ **have been announced !**
 - Definitely 512 bits (already used in the Xeon Phi co-processors)

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 C Extended Array Notation (CEAN) to their 12.0 compiler
 - As well as CILK+
 - 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:

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

Intel® Cilk™ Plus
C/C++ compiler extension for simplified parallelism

Try these first	Reducers	Array Notation
Cilk Keywords cilk_spawn cilk_sync cilk_for Vectorization __declspec(vector) __attribute__((vector)) uniform linear mask #pragma simd reduction(op:var) vectorlength	Lists list_append list_prepend Min/Max max max_index min min_index Math operators add mul Bitwise operators and or xor String concatenation string wstring Files ostream	Array sections Array section operations Section reductions add mul max max_index min min_index all_zero all_nonzero any_zero any_nonzero mutating user-defined Tools Intel® Cilk™ Screen Intel® Cilk™ View


 Simplifies harnessing the power of threading and vector processing on Windows*, Linux* and OS X*
 

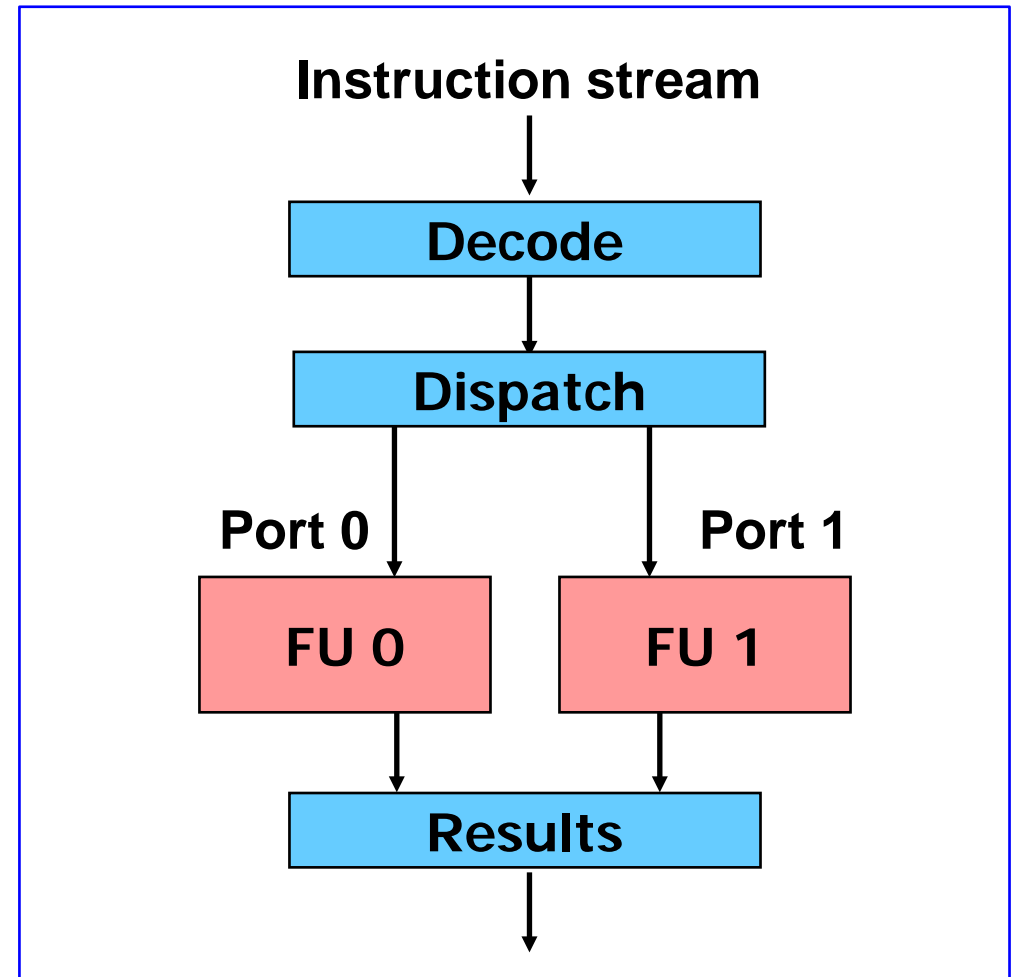
Courtesy: INTEL

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”
- **From time to time, we see a good vector example**
 - For example: Track Fitting code from ALICE trigger
 - → Explained in the HEP examples
- **Interesting development from ALICE (Matthias Kretz):**
 - Vc (Vector Classes) being implemented into ROOT
 - <http://compeng.uni-frankfurt.de/index.php?id=vc>
- **Hopefully, there will be renewed efforts to use vectors efficiently (Geant-V and others)**

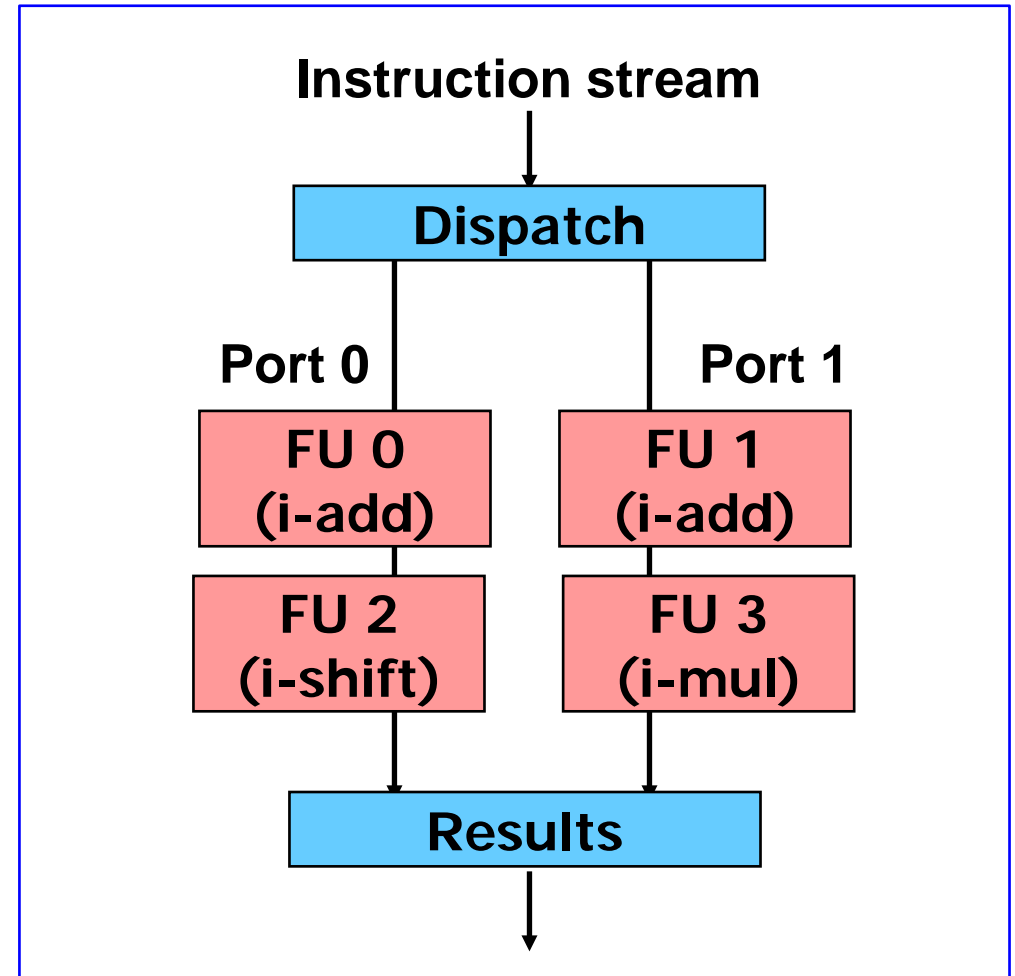
Second topic: Superscalar architecture

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

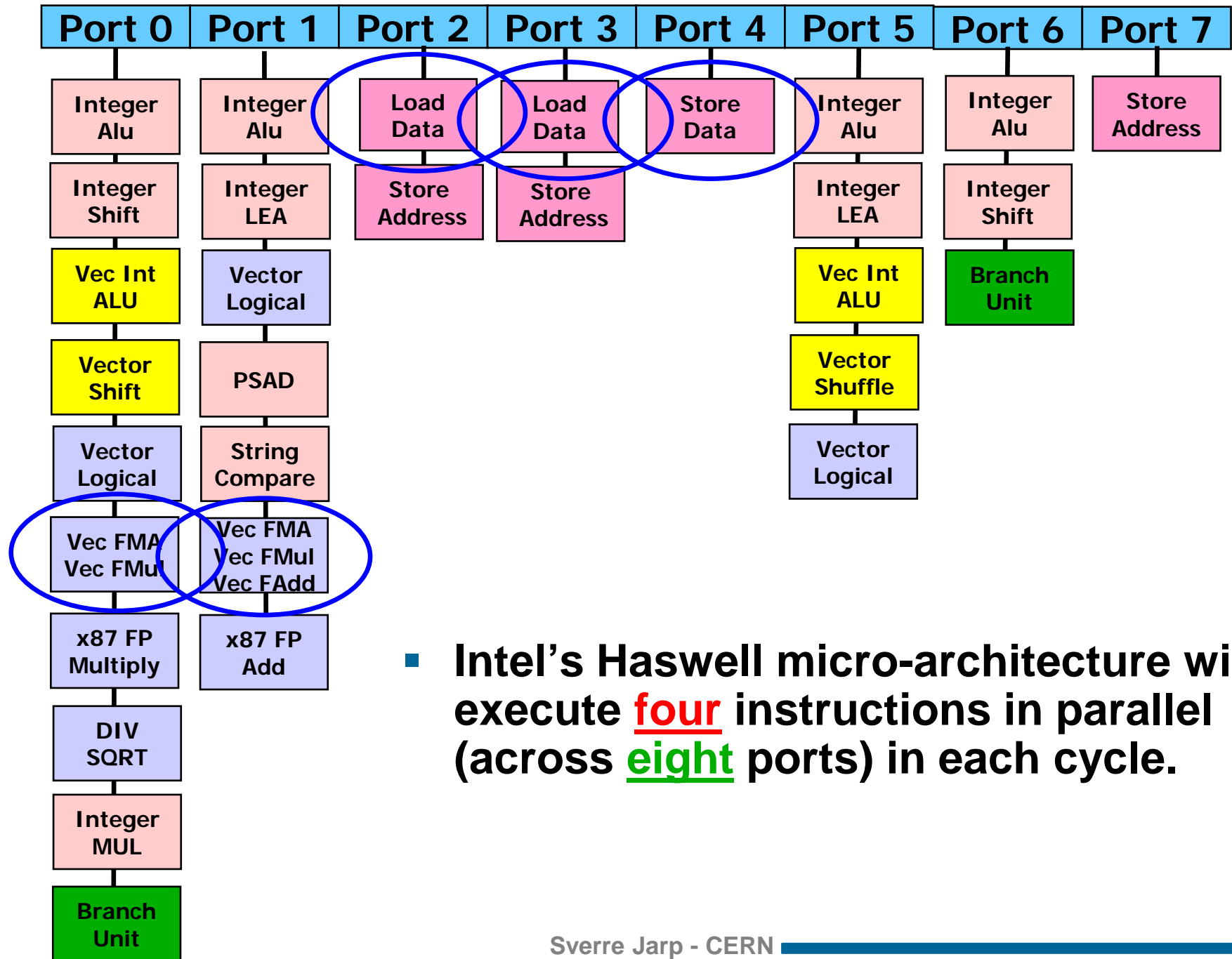


Enhanced superscalar architecture

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



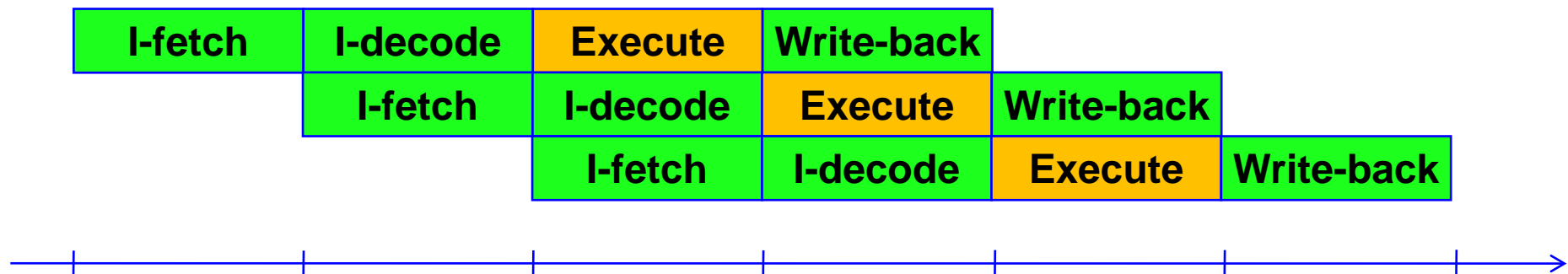
Latest superscalar architecture



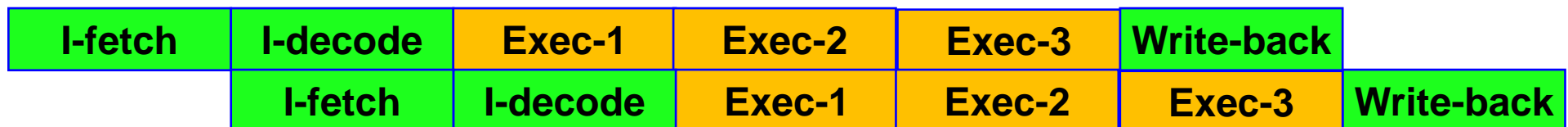
- Intel's Haswell micro-architecture will execute **four** instructions in parallel (across **eight** ports) in each cycle.

Third topic: Instruction pipelining

- Instructions are broken up into stages.
 - With a **one-cycle** execution latency (simplified):



- With a **three-cycle** execution latency:



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: **FDIV** (20), **FSQRT** (27)
 - Other math functions: even more

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

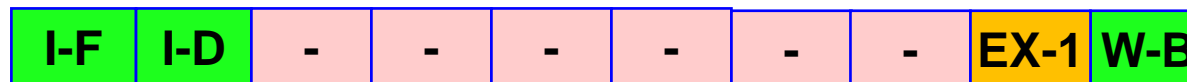
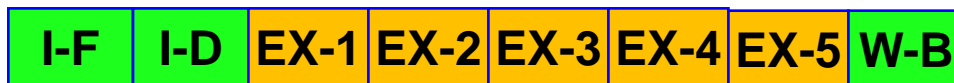
Latencies and serial code (1)

- 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;

b = 2.0; c = 3.0; e = 4.0;

a = b * c; // Statement 1
          ↘
d = a + e; // Statement 2
          ↘
f = fabs(d); // Statement 3
```



Everything moves slowly!

Summary of Last Two Dimensions

- **Commonly referred to as:**
 - Instruction level parallelism (ILP)
- **Very dependent on algorithms and/or data structures**
- **Issues are equally valid for vector and scalar computing**
- **Multiplies with what we get from all the other dimensions**
 - Vectorisation
 - Threading
- **But, difficult to understand or manipulate**
 - Both micro-architecture and compilers get in the way

Mini-example of real-life scalar, 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 Nehalem 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

Out-of-order (OOO) scheduling

- **Modern x86 processors use OOO scheduling**
 - This means that they will speculatively execute instructions ahead of time (inside a “window” of ~150 instructions)
 - In certain cases the results of such executed instructions must be discarded
- **At the end, there is a difference between “executed instructions” and “retired instructions”**
 - One typical reason for this is mispredicted branches
- **Potential problem with OOO:**
 - A lot of extra energy is needed!

Important performance measurements

(that can tell you if things go wrong)



- **Related to what we have discussed:**

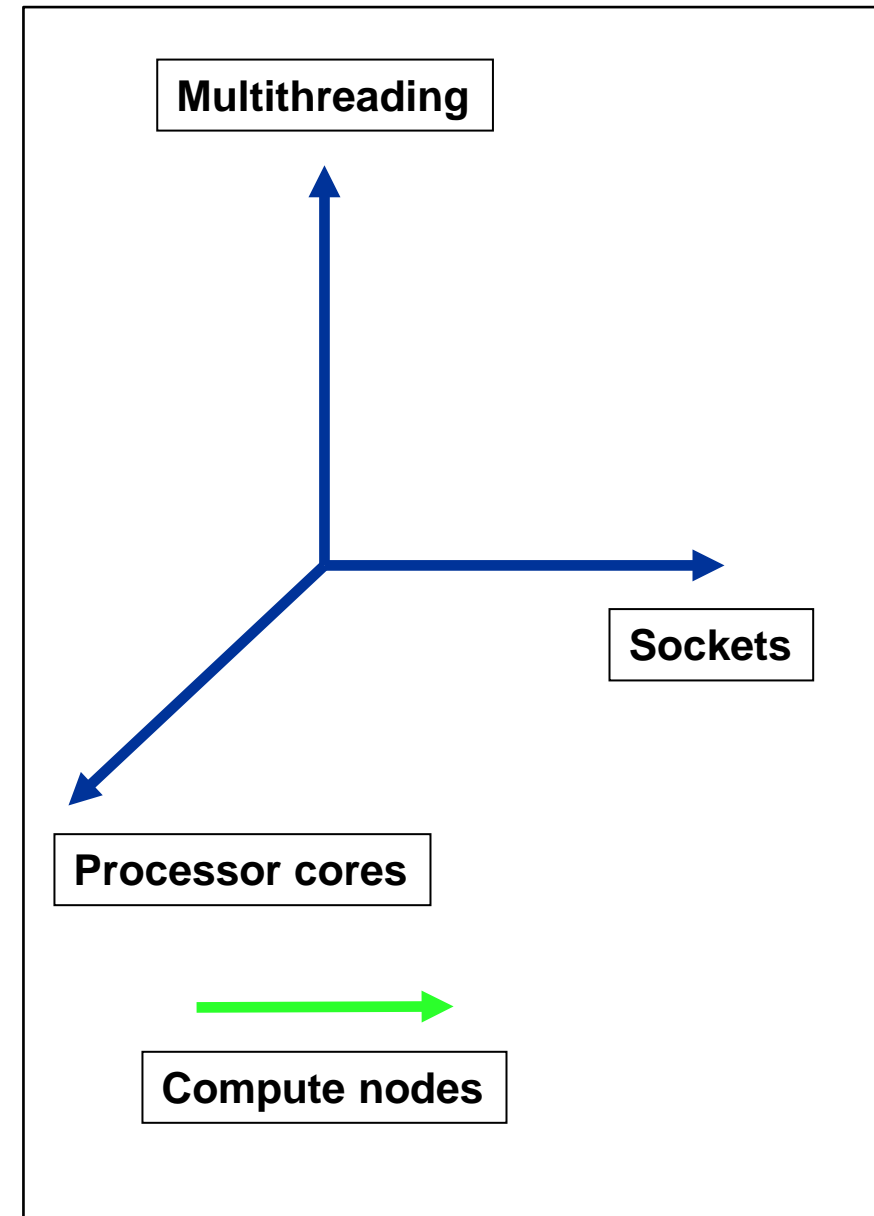
- The total cycle count (C)
- The total instruction count (I)
- Derived value: CPI
- Resource Stall count: Cycles when no execution occurred
- Total number of executed branch instructions
- Total number of mispredicted branches

- **Plus:**

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

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



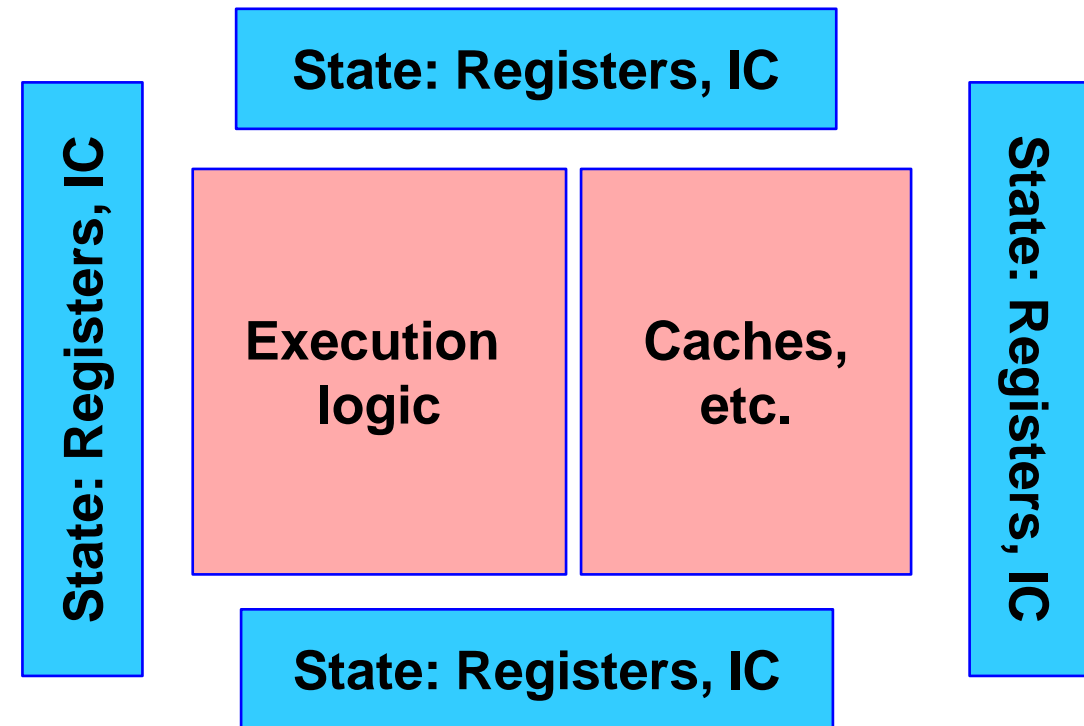
Definition of a hardware core/thread

■ 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.

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.

Seven multiplicative dimensions:

■ First three dimensions:

- Hardware vectors/SIMD
- Superscalar
- Pipelining



**Data and Instruction
Level parallelism
(Vectors/Matrices)**

■ Next dimension is a “pseudo” dimension:

- Hardware multithreading



**Task parallelism
(Events/Tracks)**

■ Last three dimensions:

- Multiple cores
- Multiple sockets
- Multiple compute nodes



**Task/process
parallelism**

The move to many-core systems

- **Examples of “CPU 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 “octo-core”: $4 * 10 * 2 = 80$

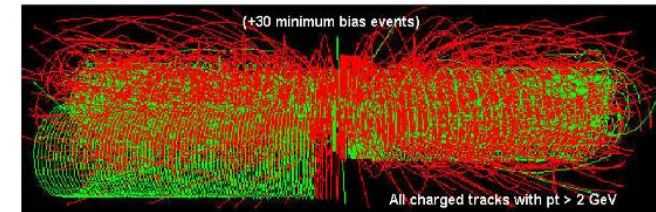
- **In the near future: Hundreds of CPU slots !**

- 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 !!**

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 process
 - A dual-socket server with eight-core processors needs 32 – 48 GB (or more)
 - The double (64 – 96 GB), if SMT is enabled!
- **Although large memories are now coming, we must not let memory limitations decide our ability to compute efficiently!**



What are the options?

- **There is currently a discussion in the community about the best way forward (in a many-core world):**
 - 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; Fork N others
 - 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

Let's briefly introduce parallelism

Getting to Parallel Execution

- **Multiple steps must be kept in mind:**
 - Concurrency
 - Decomposition
 - Communication
 - Synchronization
 - Mapping
 - 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

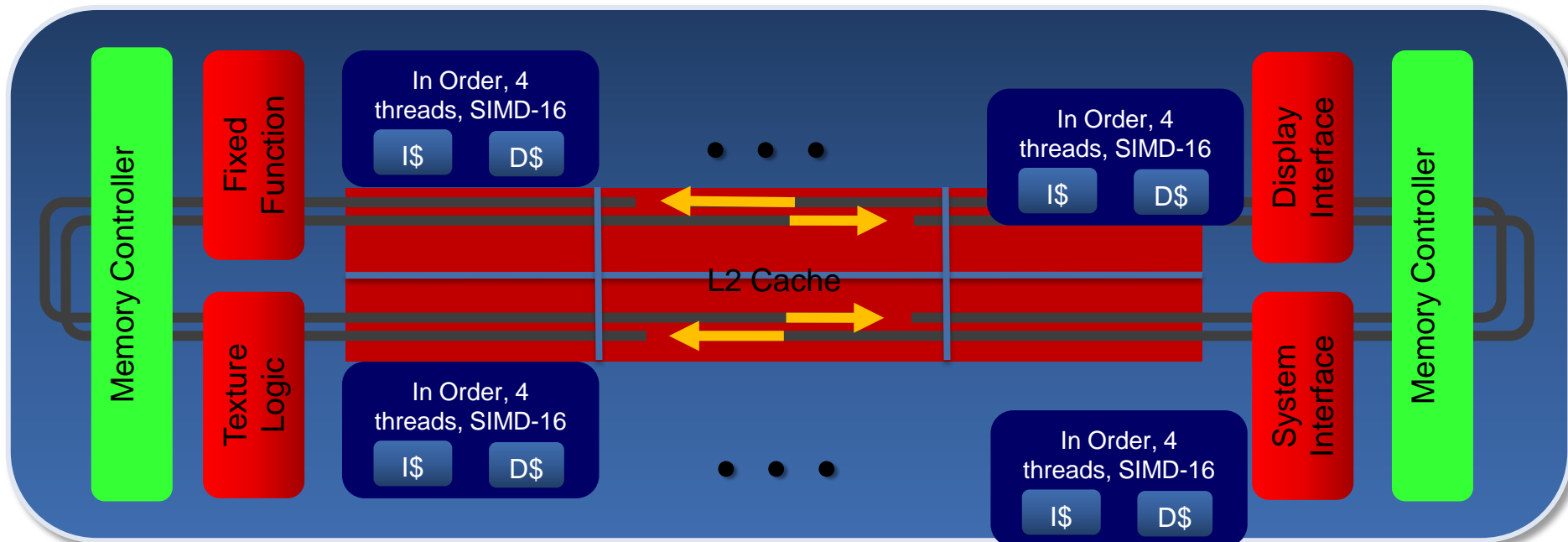
Parallelization support (C++ and others)

- **Large selection of tools (inside the compiler or as additions):**
 - Native: pthreads/Windows threads
 - New C++ standard: `std::thread`
 - OpenMP
 - Intel Threading Building Blocks (TBB; also open source)
 - Intel CILK+
 - OpenACC
 - Thread wrapper classes
 - MPI (from multiple providers), etc.
 - CUDA (from Nvidia)
 - OpenCL

Accelerators (1): Intel Xeon Phi

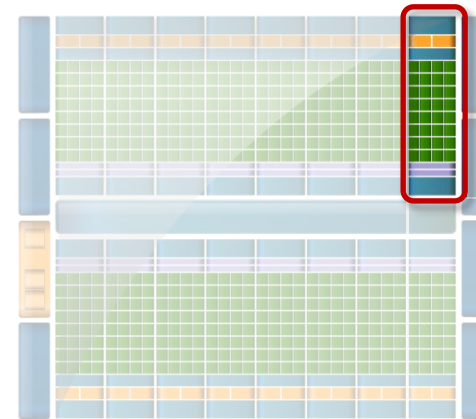
- **Intel Many Integrated Cores (MIC):**
 - Announced at ISC10 (end-May 2010)
 - Based on the x86 architecture, 22nm
 - Many-core (up to 62 cores) + 4-way multithreaded + **512-bit vector unit**
 - **Limited memory: 8 – 16 Gigabytes**

48'000 such accelerators are used in the world's fastest supercomputer (Tianhe-2 Xeon-cluster in China)



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 Tflops
- Double-precision: ~300 Gigaflops
- Dual Thread Scheduler
- 64 KB of RAM for shared memory and L1 cache (configurable)
- A few Gigabytes of main memory



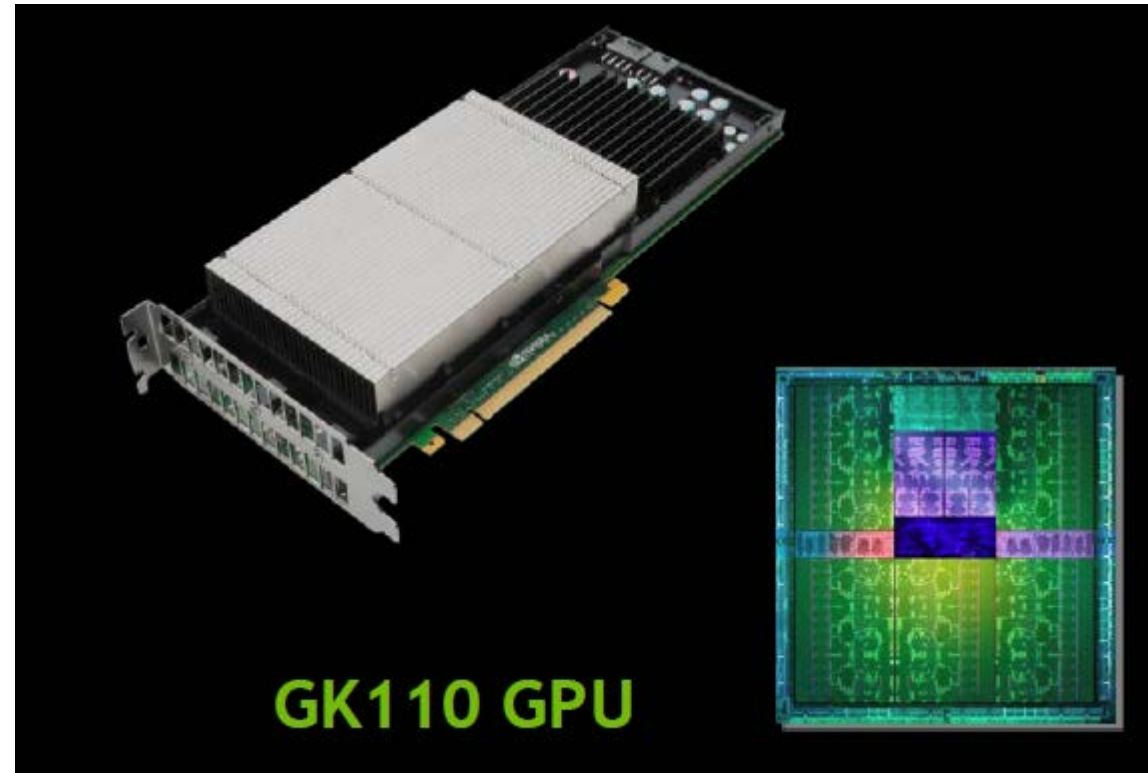
Considerable interest in the physics community



Adapted from Nvidia

Accelerators (3): Nvidia Kepler GPU

- Made available in 4Q2012
 - GK110 GPU
 - 3x DP performance:
 - 1 Teraflops
 - Innovative design:
 - **SMX** (streaming multiprocessors)
 - **Dynamic parallelism** for spawning new threads
 - **Hyper-Q** enables multiple CPU cores to utilise CUDA cores



18'688 such accelerators are used in the world's second-fastest supercomputer (Titan Cray XK7)

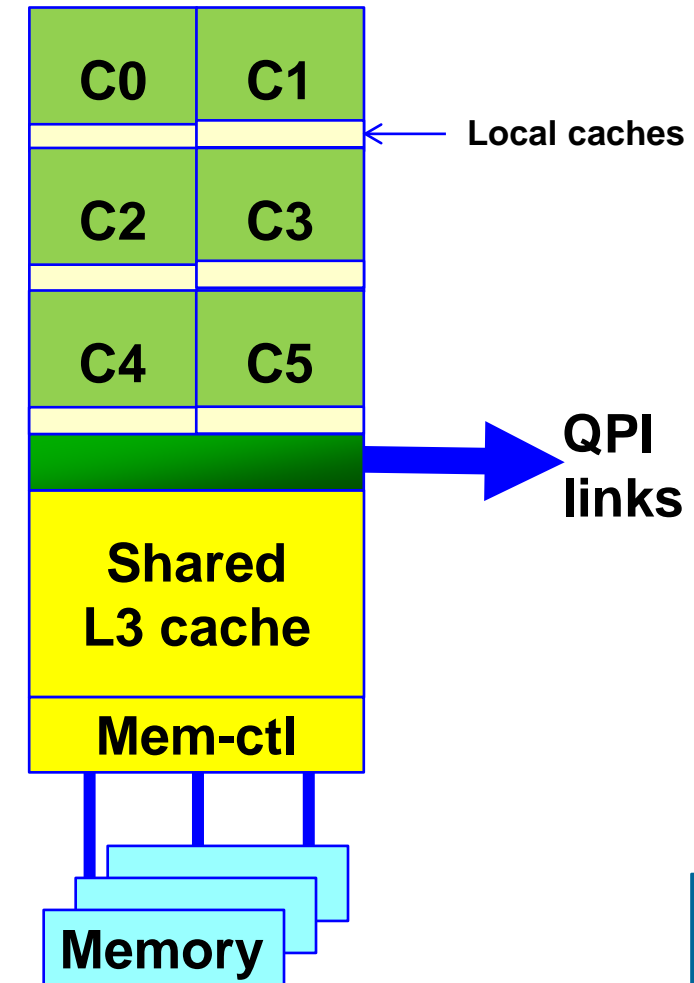
Memory Subsystem

Optimal Memory Programming

- **What needs to be understood:**
 - The memory hierarchy
 - Caches
 - Physical layout, Line sizes
 - Levels/Sharing
 - Latency
 - Main memory
 - Physical layout
 - Latency
 - Bandwidth
 - Programmer/Compiler
 - Data Layout
 - Data Locality
 - Execution environment:
 - Affinity

Recent Intel Server Architecture

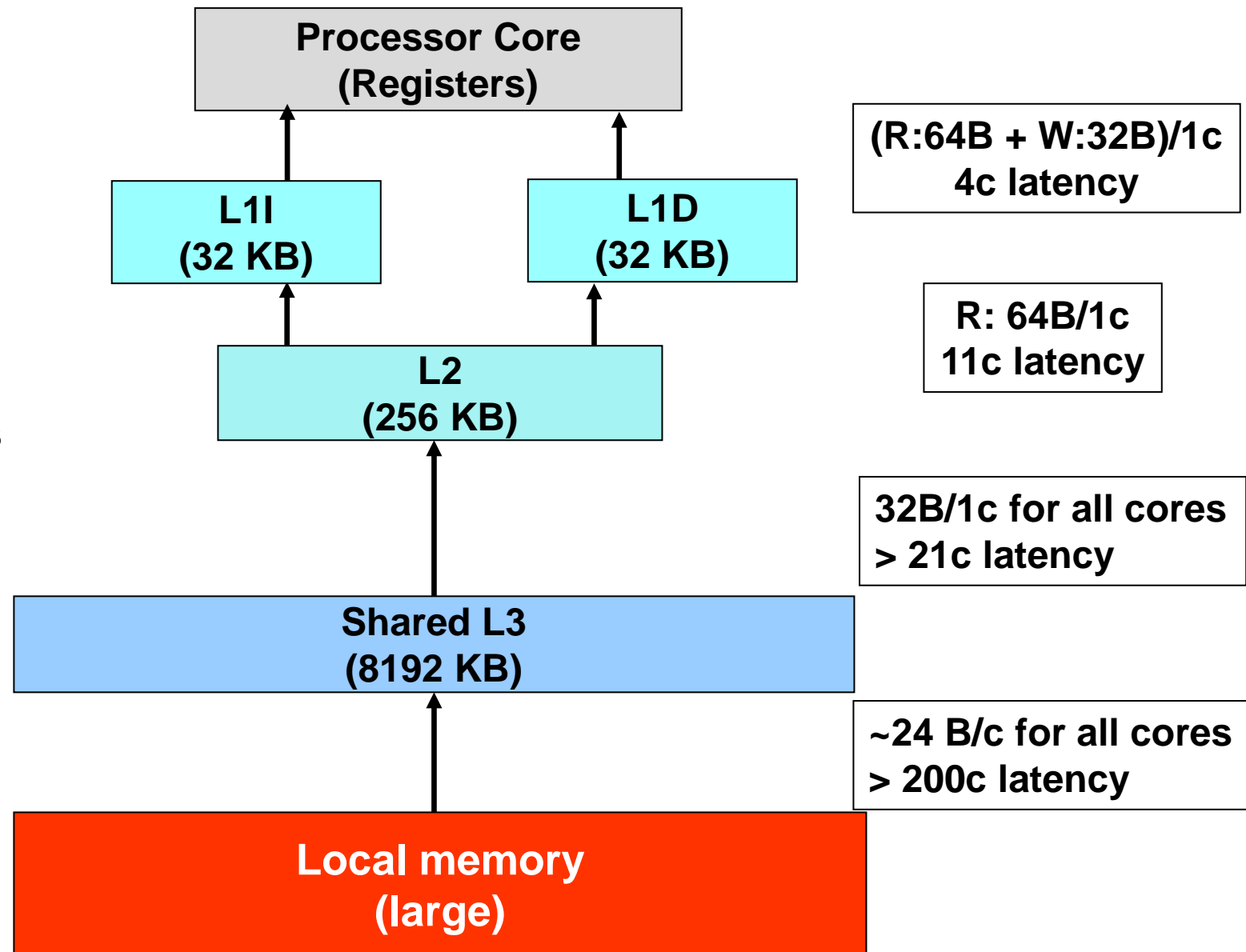
- **Basic building block (as shown in the introduction):**
 - Multiple cores with dedicated Level1 (I&D) and Level2 (unified) caches
 - Shared Level-3 cache
 - Integrated Memory Controller attaches memory directly to the multi-core processor
 - Multiple memory channels
 - Intel Quick Path Interconnect (QPI) or AMD HyperTransport (HT)



AMD was the first company to introduce an on-chip memory controller and processor interconnection (HyperTransport)

Cache Hierarchy

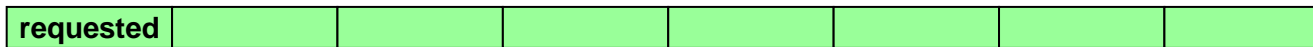
- From CPU to main memory on a Haswell processor
 - With multicore, memory bandwidth is shared between cores in the same processor (socket)



c = cycle

Cache lines (1)

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

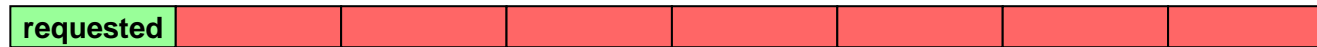


- A cache line is a contiguous section of memory, typically 64B in size ($8 * \text{double}$) and 64B aligned
 - A 32KB Level-1 cache can hold 512 lines
- When cache lines have to be moved come from memory
 - Latency is long (>200 cycles)
 - It is even longer if the memory is remote
 - Memory controller stays busy (~ 8 cycles)

Cache lines (2)

- **Good utilisation is vital**

- When only one element (4B or 8B) element is used inside the cache line:
 - A lot of bandwidth is wasted!



- **Multidimensional C 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)**

Cache lines (3)

- **Prefetching:**

- Fetch a cache line before it is requested
 - Hiding latency
- Normally done by the hardware
 - Especially if processor executes Out-of-order
- Done also by software instructions
 - Especially when In-order (IA-64, Xeon Phi, etc.)

- **Locality is vital:**

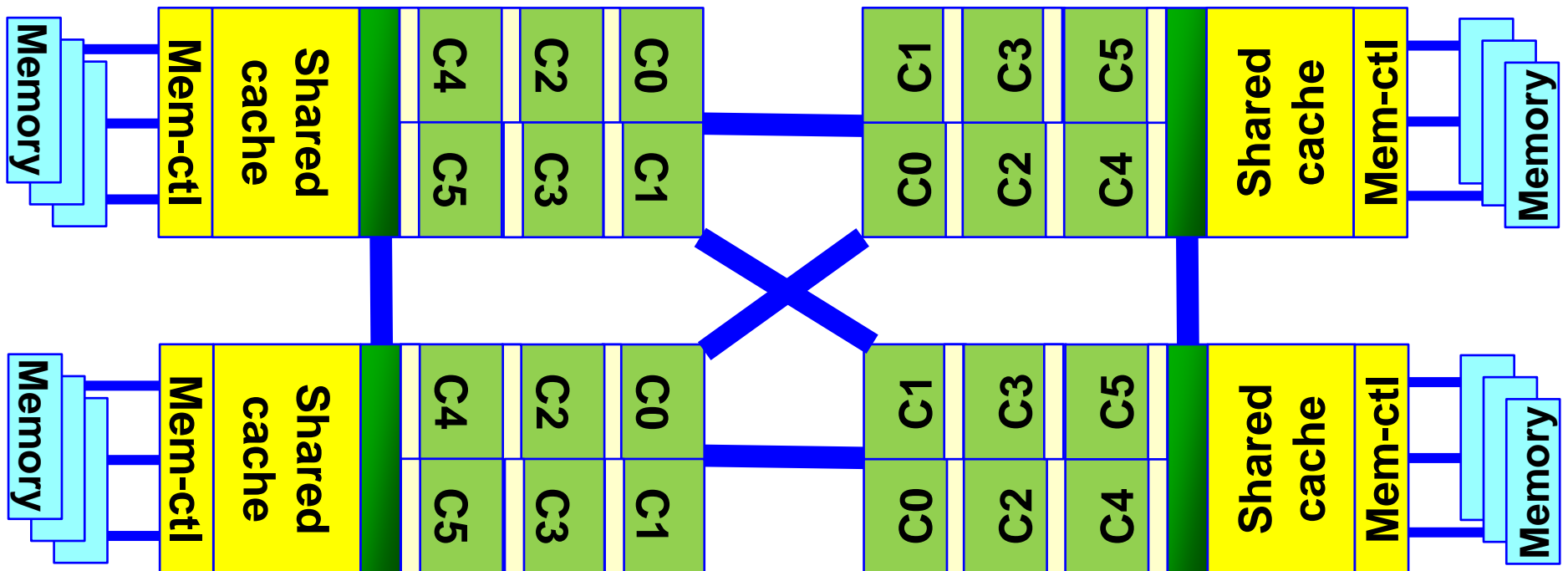
- Spatial locality – Use all elements in the line
- Temporal locality – Complete the execution whilst the element is certain to be in the cache

Programming the memory hierarchy is an art in itself.

Non-Uniform Memory Access

Quad Socket Design

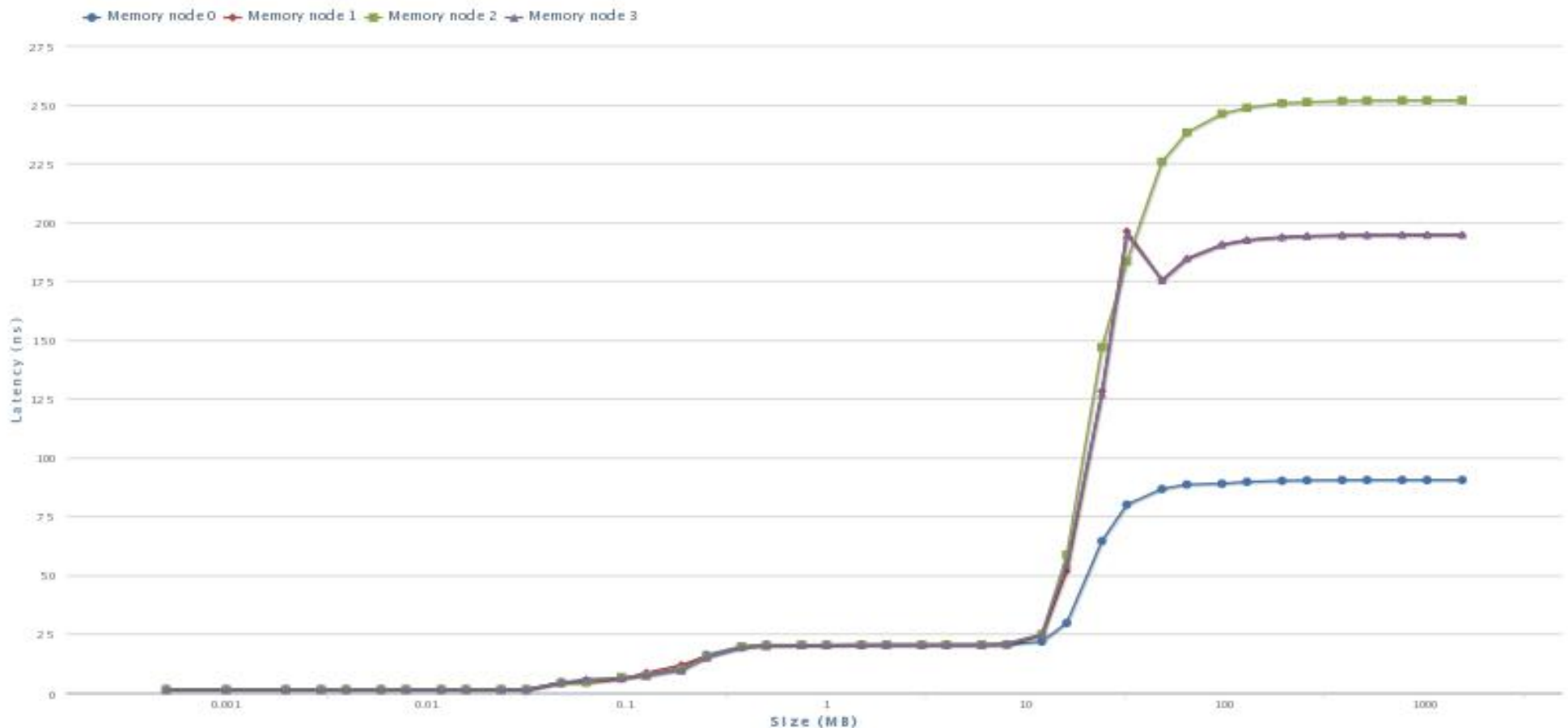
- All processors are interconnected via QPI links (in blue):



Latency Measurements (3)

Memory Latency on Sandy Bridge-EP 4650 (quad-socket):

Sandy Bridge-EP E5-4650 memory latency



Some Recommendations

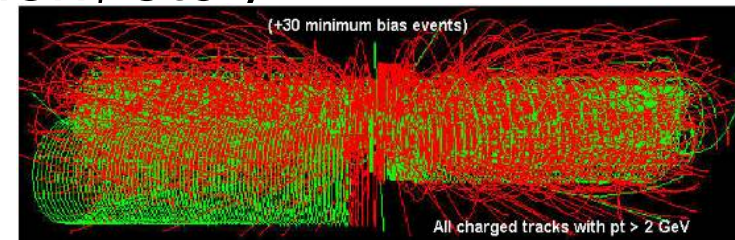
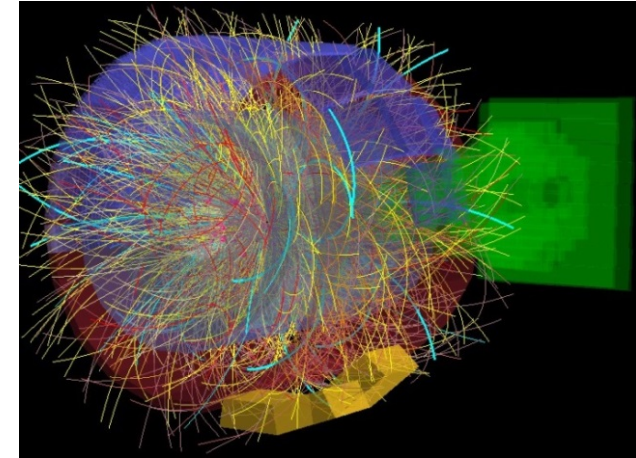
(based on observations in openlab)

A proposal for “agile” software

- 1) **Seek out parallelism at all levels**
 - a. Events, tracks, vertices, etc.
 - b. Perform “chunk” processing (removing event separation)
- 2) **Build forward scalability**
- 3) **Create compute-intensive kernels**
- 4) **Optimise the Memory Hierarchy**
- 5) **Create Performance-oriented Code**
- 6) **Combine broad programming talents**
- 7) **Use best-of-breed tools**

Concurrency in High Energy Physics

- We are “blessed” with lots of it:
 - Entire events
 - Particles, hits, 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!

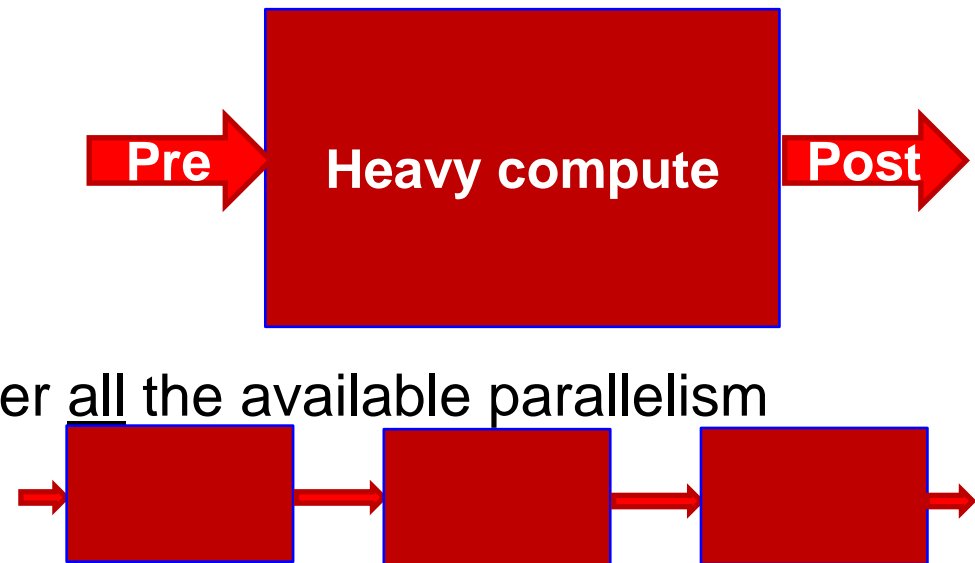


The holy grail: Forward scalability

- 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!

Kernel-oriented Programming

- **Take the whole program and its execution behaviour into account**
 - Get yourself a global overview as soon as possible
 - Via early prototyping with realistic algorithms/data
 - Influence early the design and definitely the implementation
- **Foster clear split:**
 - Prepare to compute
 - Do the heavy computation
 - In kernels, where you go after all the available parallelism
 - Post-processing
- **Often, a single kernel is not sufficient**
 - A sequence of kernels may be needed



The 90 – 10 rule

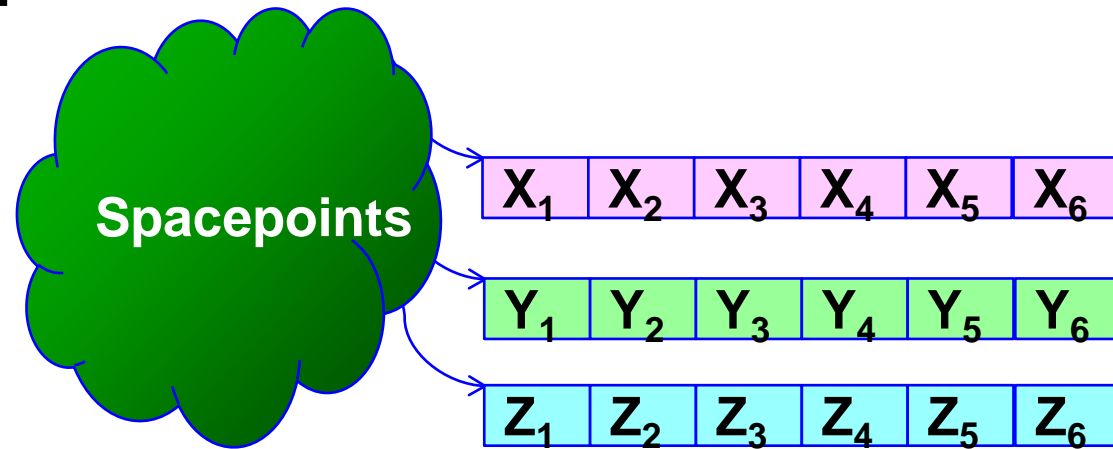
CPU / GPU co-existence

- **What I would like to see happen to a (possibly dusty, sequential) x86 application:**
- **A strong porting effort to move it to the GPU**
 - A good “kernel-oriented design” that aims for a triple-digit speed-up
- **Then, a solid port back to the CPU servers**
 - Exploiting vectors and cores
- **Outcome:**
 - Applications that can profit from new breakthroughs on either side of the fence

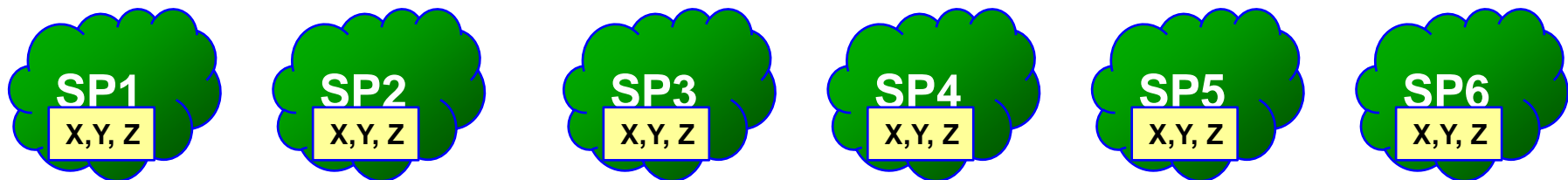
Data layout: SoA versus AoS

- In general, both GPUs and CPUs prefer the former!

- Structure of Arrays (SoA):



- Array of Structures (AoS):



- Also possible: AoSoA

Performance-oriented code

■ C++ for performance

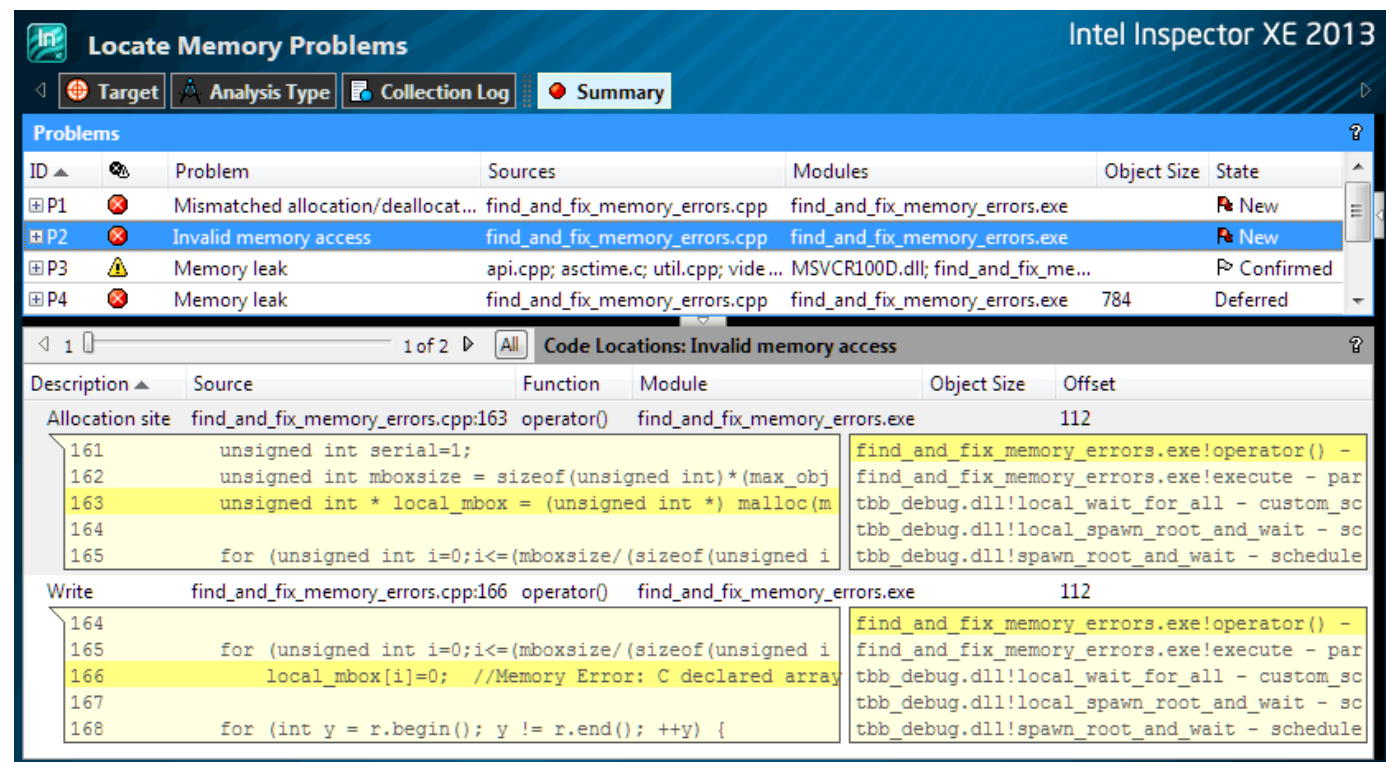
- Use light-weight C++ constructs
- Minimize virtual functions
- Inline whenever important
- Optimize the use of math functions
 - SQRT, DIV
 - LOG, EXP, POW
 - SIN, COS, ATAN2

**Use vector
libraries
whenever
possible,
but master
the
accuracy!**

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

Performance tools

- Surround yourself with good tools:
 - Compilers (not just one!)
 - Libraries
 - Profilers
 - Debuggers
 - Thread checkers
 - Thread profilers



Intel Inspector XE 2013

Locate Memory Problems

Target Analysis Type Collection Log Summary

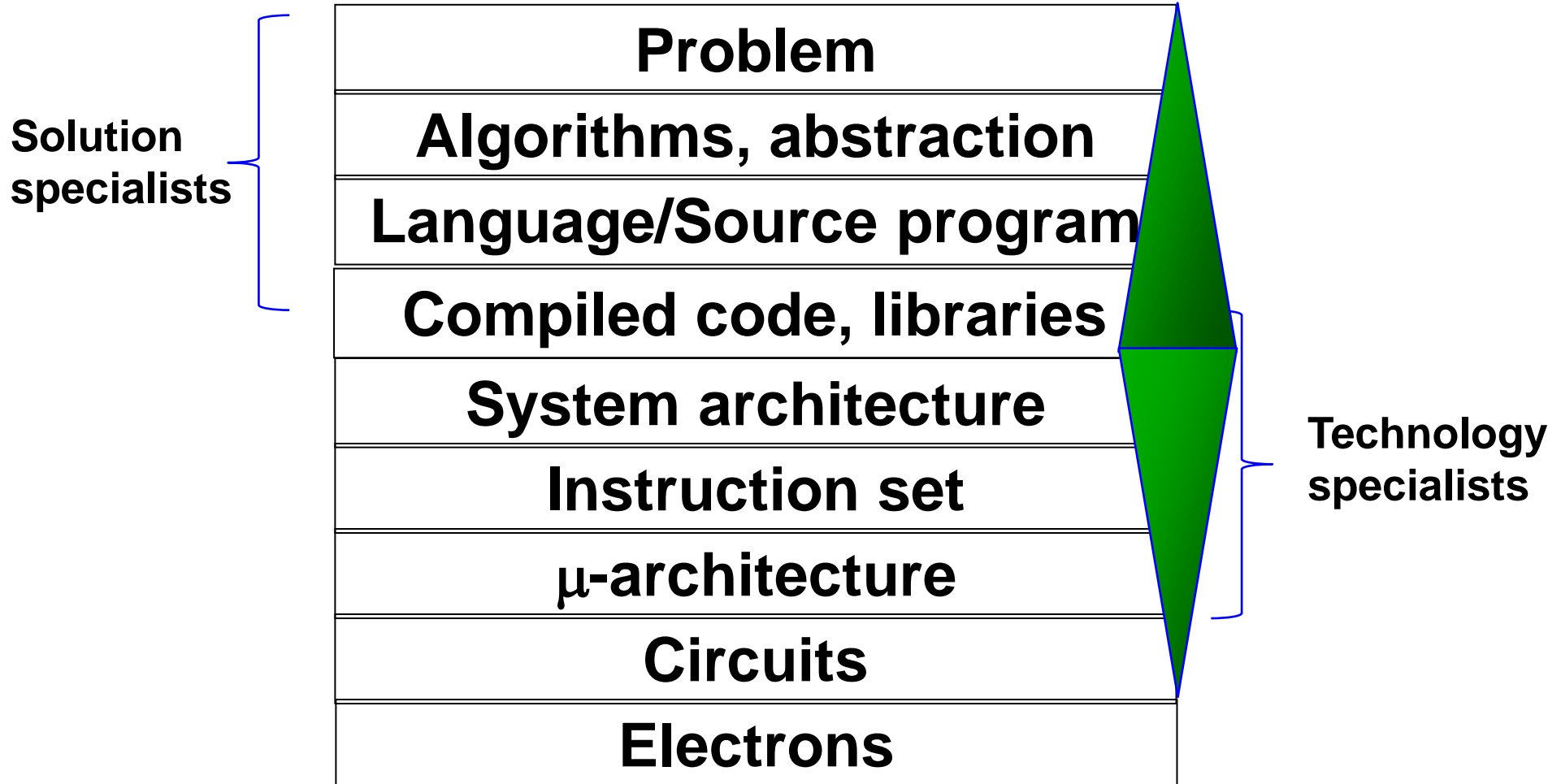
ID	Problem	Sources	Modules	Object Size	State
P1	Mismatched allocation/deallocat...	find_and_fix_memory_errors.cpp	find_and_fix_memory_errors.exe		New
P2	Invalid memory access	find_and_fix_memory_errors.cpp	find_and_fix_memory_errors.exe		New
P3	Memory leak	api.cpp; asctime.c; util.cpp; vide...	MSVCR100D.dll; find_and_fix_me...		Confirmed
P4	Memory leak	find_and_fix_memory_errors.cpp	find_and_fix_memory_errors.exe	784	Deferred

1 of 2 All Code Locations: Invalid memory access

Description	Source	Function	Module	Object Size	Offset
Allocation site	find_and_fix_memory_errors.cpp:163	operator()	find_and_fix_memory_errors.exe		112
161	unsigned int serial=1;				
162	unsigned int mboxsize = sizeof(unsigned int)*(max_obj				
163	unsigned int * local_mbox = (unsigned int *) malloc(m				
164					
165	for (unsigned int i=0;i<=(mboxsize/(sizeof(unsigned i				
Write	find_and_fix_memory_errors.cpp:166	operator()	find_and_fix_memory_errors.exe		112
164					
165	for (unsigned int i=0;i<=(mboxsize/(sizeof(unsigned i				
166	local_mbox[i]=0; //Memory Error: C declared array				
167					
168	for (int y = r.begin(); y != r.end(); ++y) {				

Broad Programming Talent

- In order to cover as many layers as possible

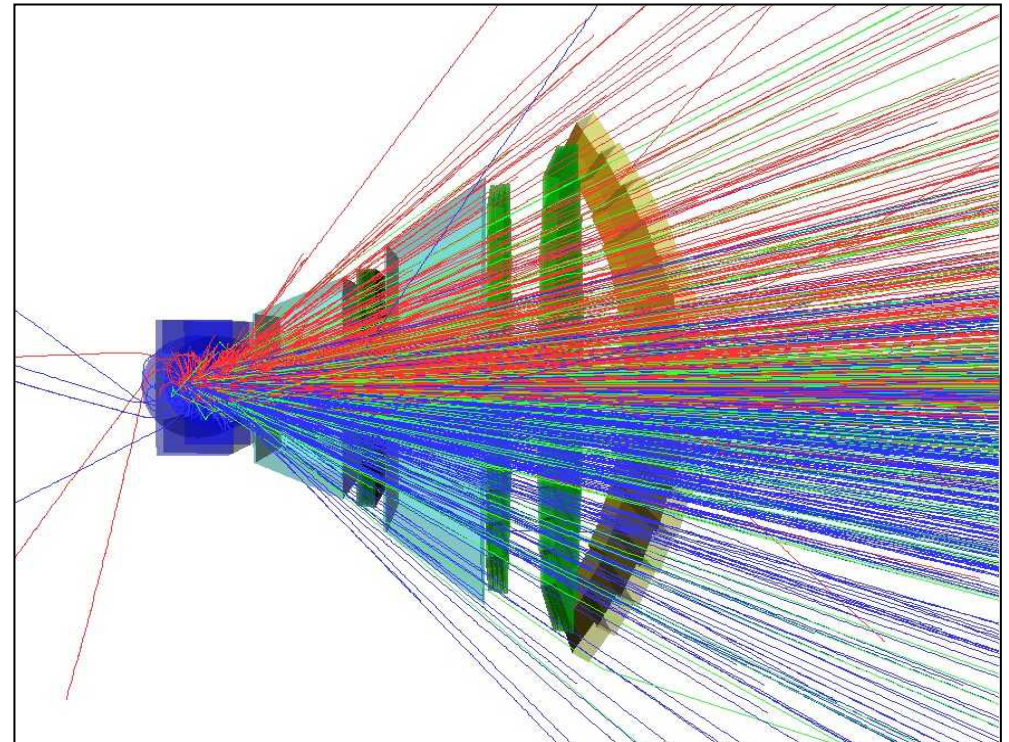


Two HEP examples

Examples of parallelism: CBM/ALICE track-fitting

- **Extracted from the 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"

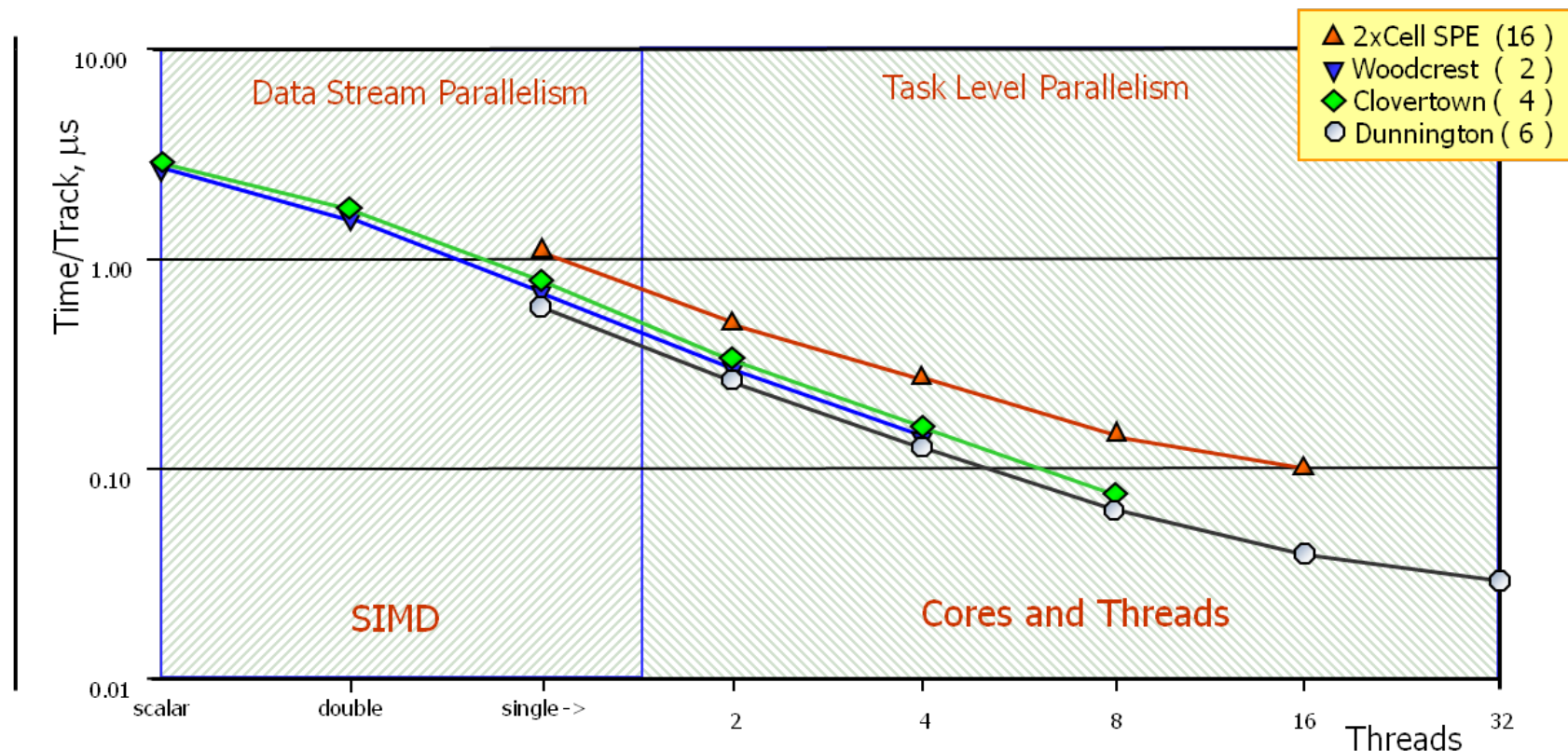
CBM/ALICE track-fitting

- **Details of the re-optimization on x86-64:**
 - **Part 1:** 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)



Examples of parallelism: CBM track-fitting

- **Re-optimization on x86-64 systems**
 - Step 1: Data parallelism using SIMD instructions
 - **Step 2**: use TBB (or OpenMP) to scale across cores



Scalability on different CPU architectures – speed-up 100

Analysis of Track-fitting speed-up

- From paper of I.Kisel (2008):

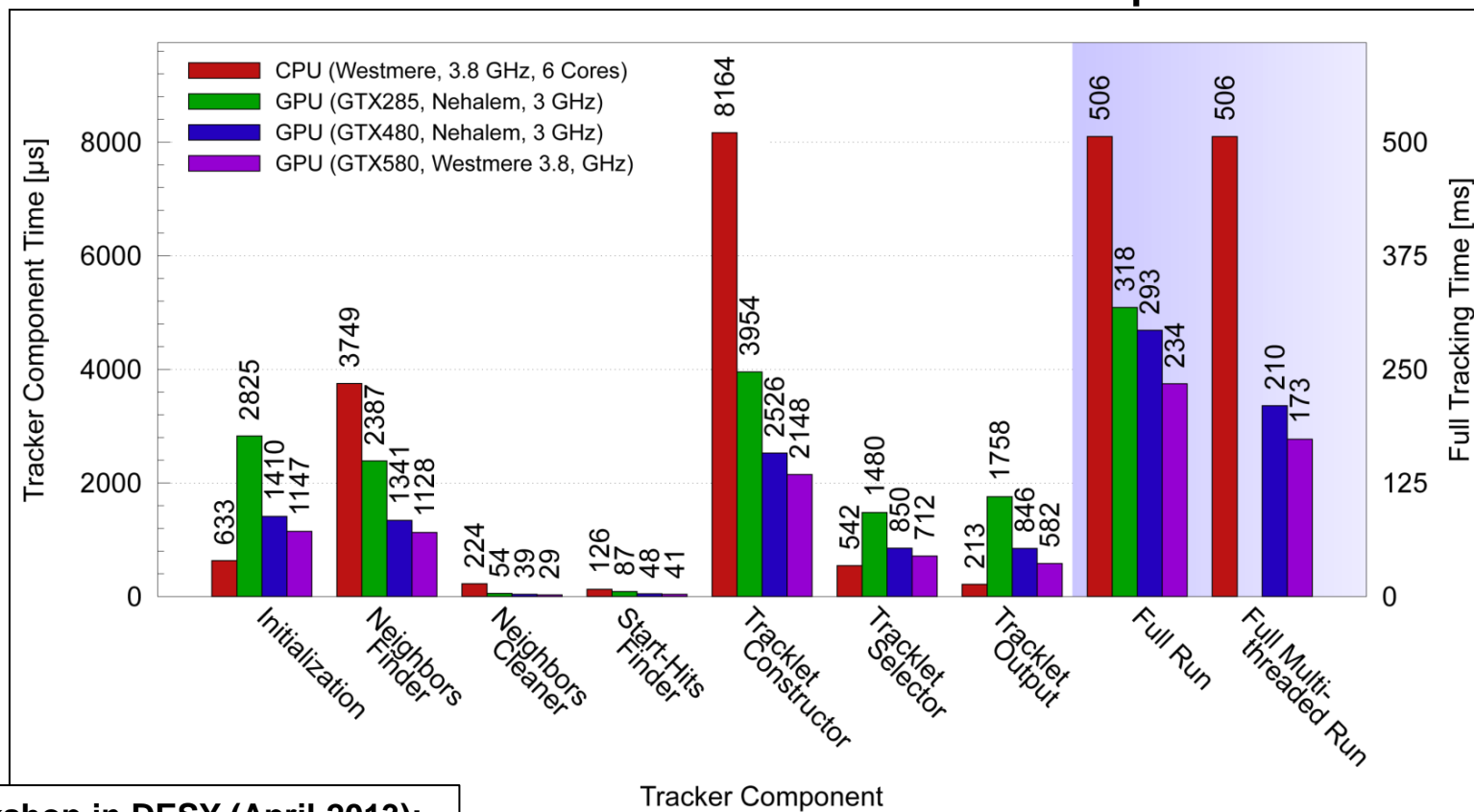
Stage	Description	Time/track	Speed-up	
	Initial scalar version	12 ms	-	1667x
1	Approximation of magnetic field	240 μ s	50	
2	Optimisation of the algorithm	7.2 μ s	33.3	
3	Vectorisation	1.6 μ s	4.5	72x
4	Porting to IBM Cell/SPE	1.1 μ s	1.45	
5	Parallelisation on 16 SPEs	0.1 μ s	10	
	Final SIMD parallel version	0.1 μ s	120'000	

Don't underestimate software optimisation!

Track-fitting: Port to NVIDIA GPU

■ David Rohr/ALICE:

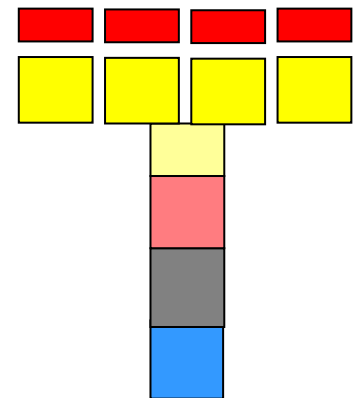
- Integration: GPU and CPU tracker share a common set of source files
- Performance Comparison: GTX580 GPU is almost three times faster than a six-core Westmere processor



Examples of parallelism: GEANT4

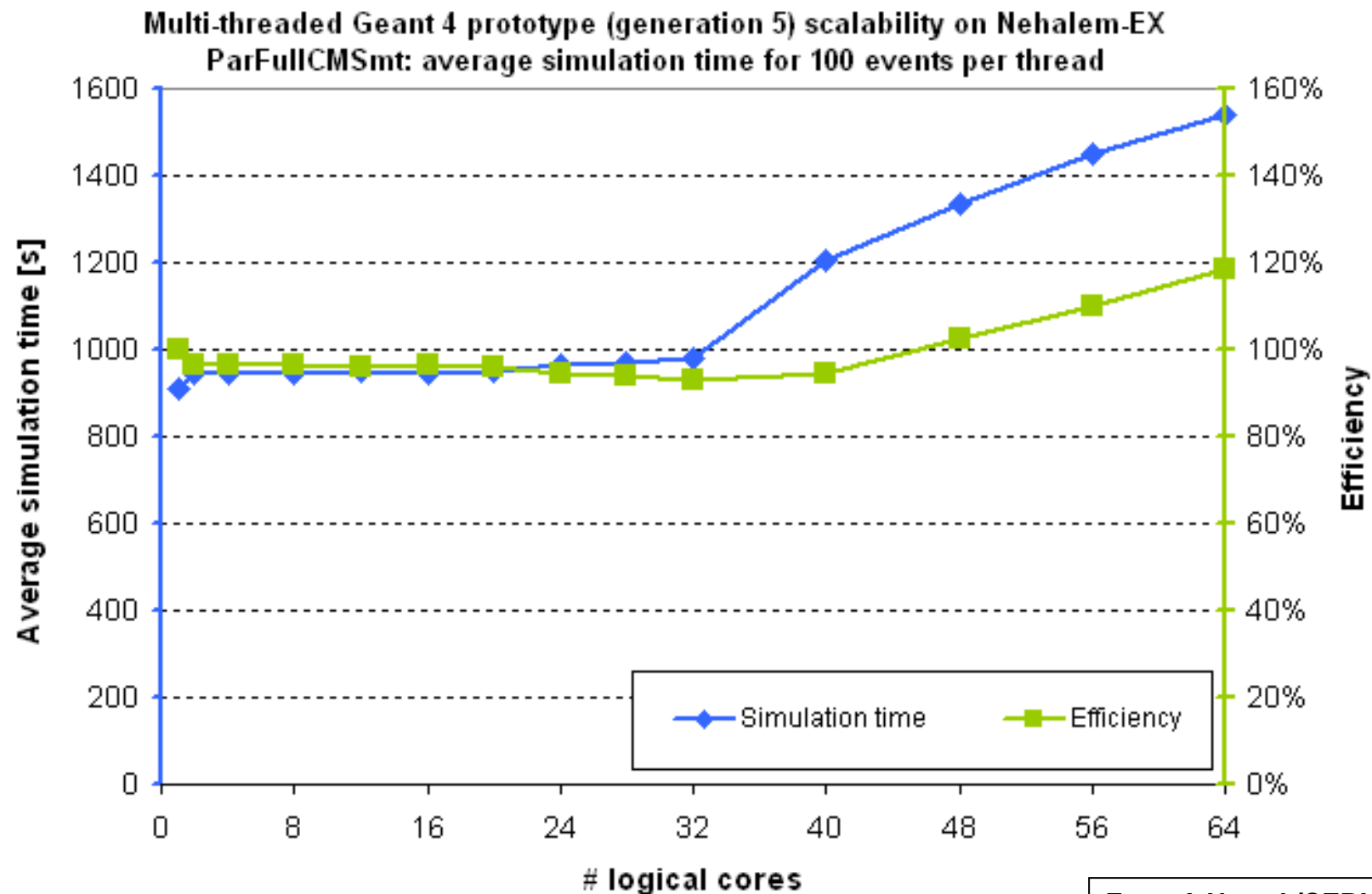


- **Initially; ParGeant4 (Gene Cooperman/NEU)**
 - implemented event-level parallelism to simulate separate events across remote nodes.
- **New [prototype](#) re-implements thread-safe [event-level](#) parallelism inside a multi-core node**
 - Done by NEU PhD student Xin Dong:
 - Using [FullCMS](#) and [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:
 - Now in separate branch in the G4 source tree
- **Additional memory: [Only 25 MB/thread \(!\)](#)**



Multithreaded GEANT4 benchmark

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



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 $C = C - A \times B^T$ operation and the $C = C - A \times B$ 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.



So, what does it all mean?

- **Here is what we tried to say in these lectures:**
- **You must get your code to use vectors**
- **You must understand if your ILP is seriously limited by serial code, complex math functions, and other contracts**
- **You must parallelise across all “CPU slots”**
 - **Hardware threads, Cores, Sockets**

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**
 - In some case, “multi-event” could be even better!
- **Will you be ready for 1000 cores and long vectors?**
 - Are you thinking “parallel, parallel, parallel” ?
- **It helps to know the seven hardware dimensions in order to get close to the best software design !**

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**

Thank you!

FastFlow