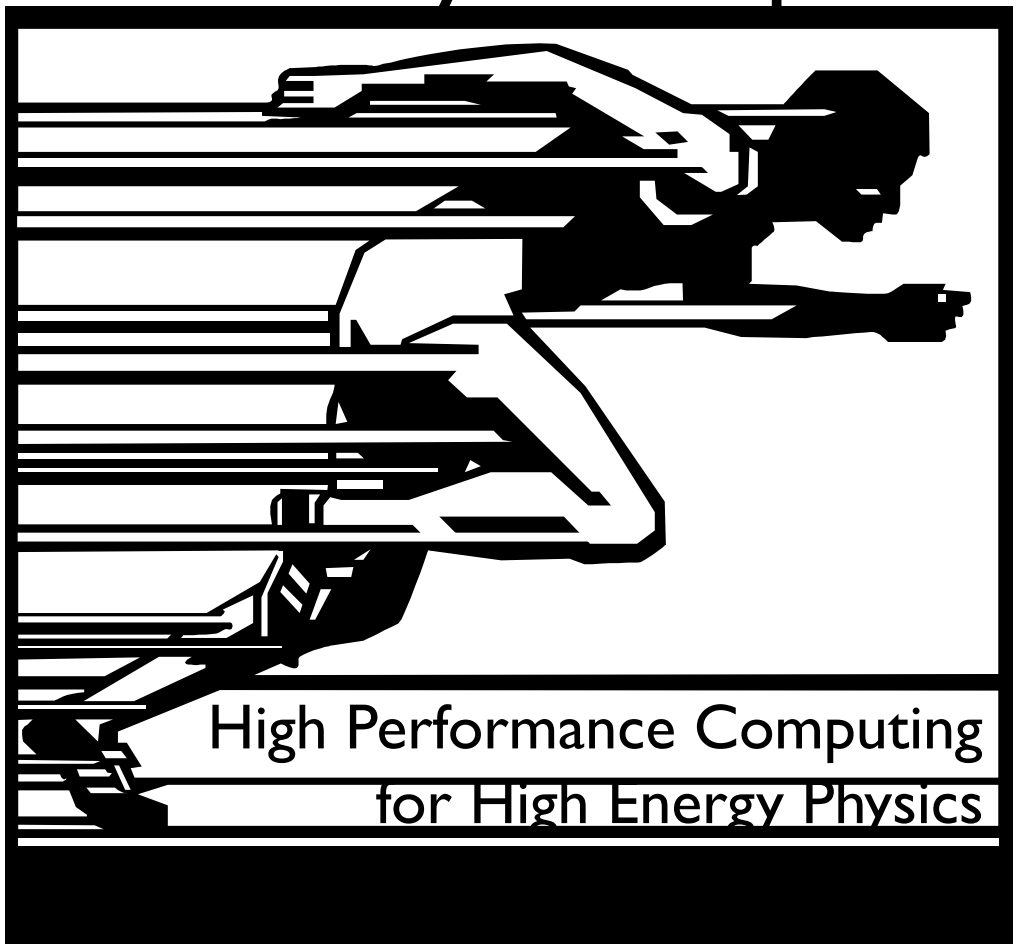
HEP physics software applications on many-core: present and perspectives

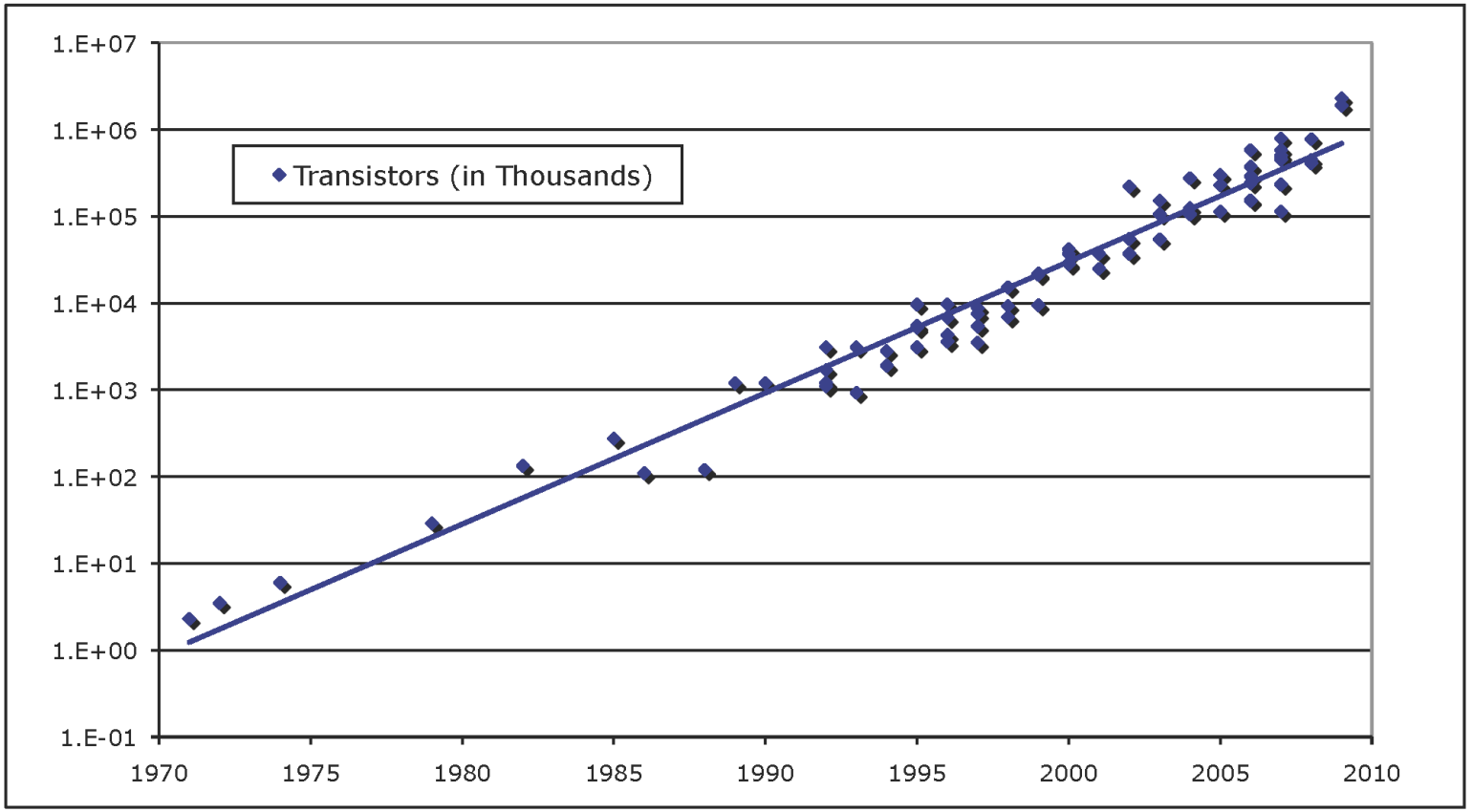


SuperB Workshop, July `11

Vincenzo Innocente
CERN

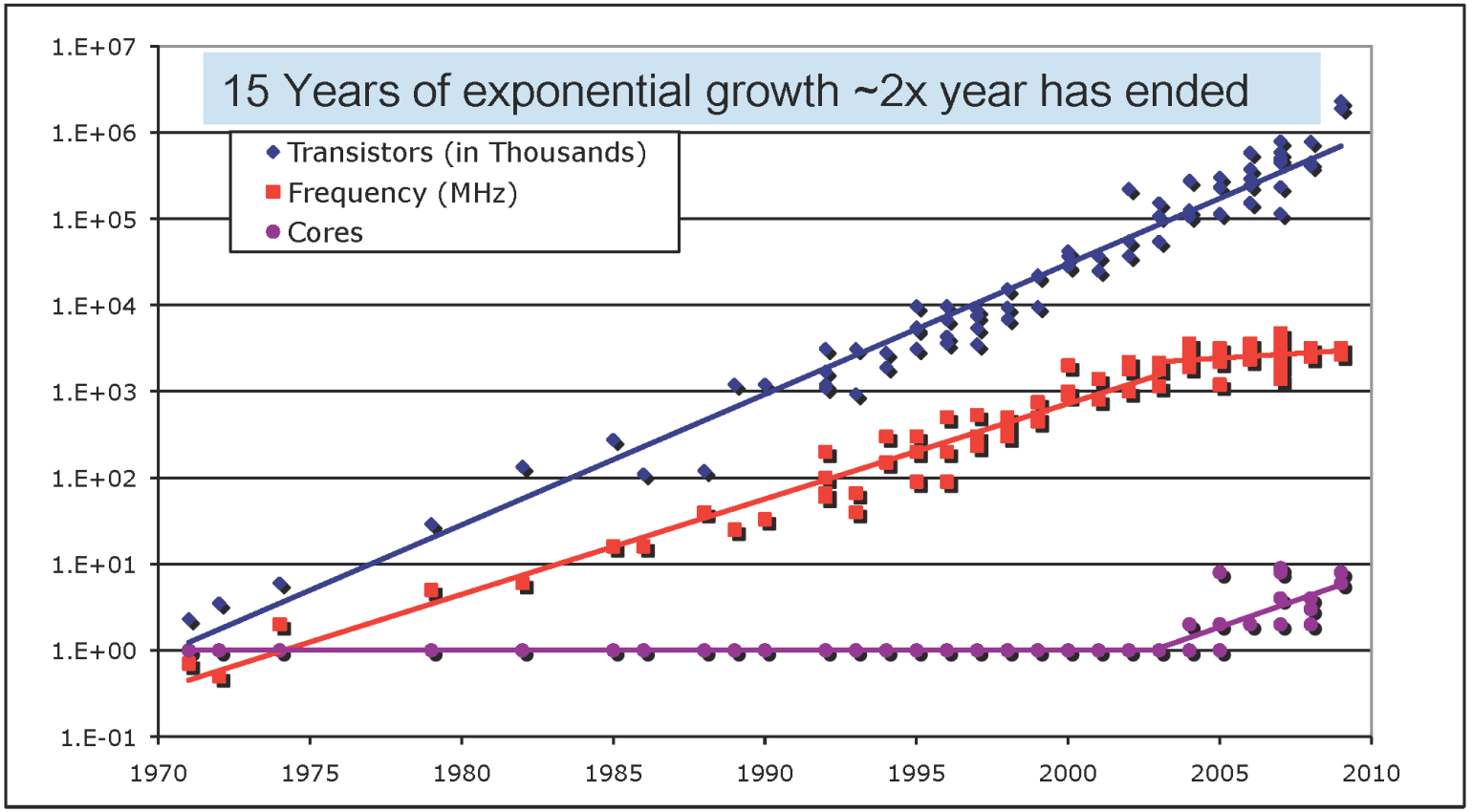
https://twiki.cern.ch/twiki/bin/view/LCG/MultiCoreRD

Moore's Law is Alive and Well



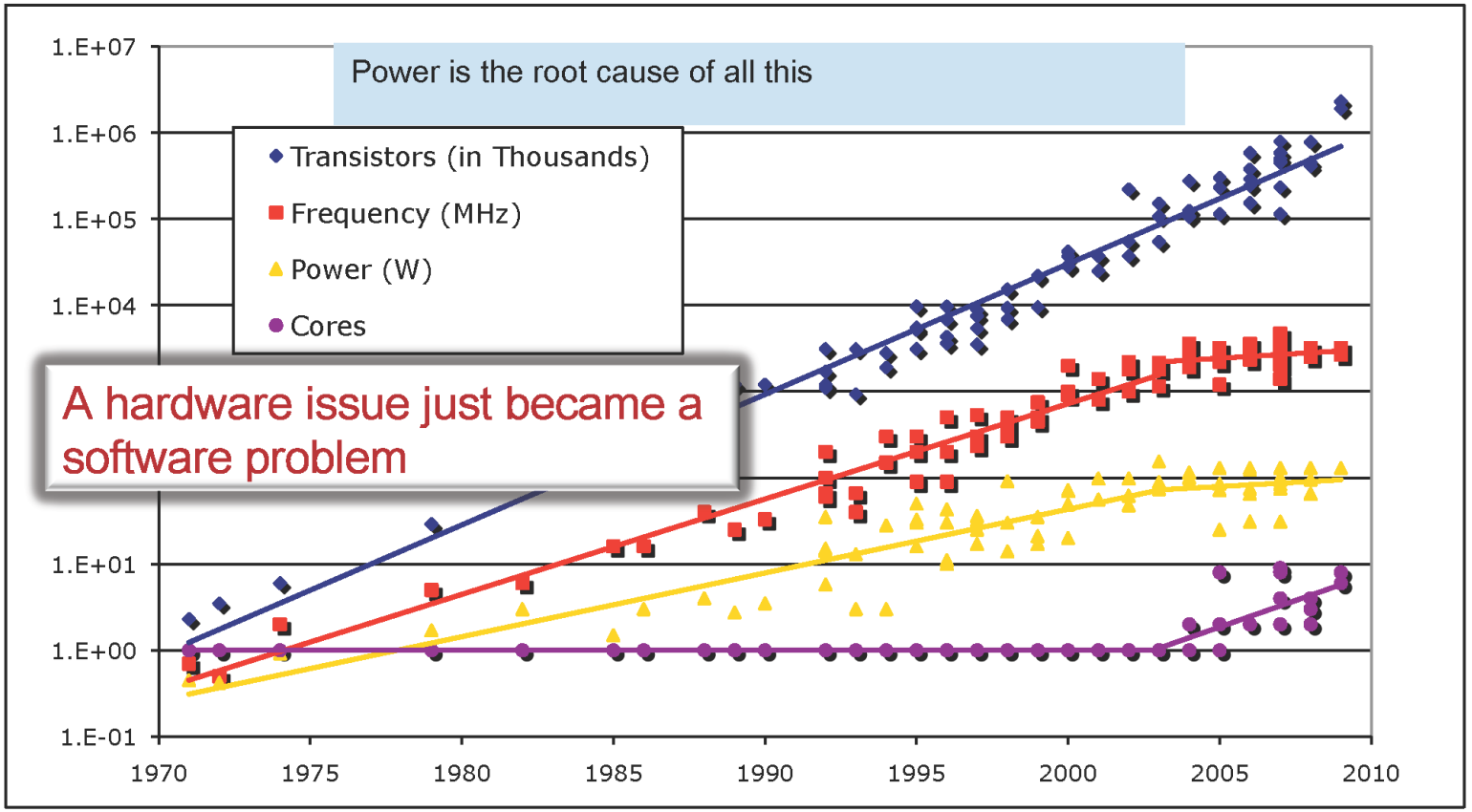
Data from Kunle Olukotun, Lance Hammond, Herb Sutter, Burton Smith, Chris Batten, and Krste Asanoviç

But Clock Frequency Scaling Replaced by Scaling Cores / Chip



Data from Kunle Olukotun, Lance Hammond, Herb Sutter, Burton Smith, Chris Batten, and Krste Asanoviç

Performance Has Also Slowed, Along with Power



Data from Kunle Olukotun, Lance Hammond, Herb Sutter, Burton Smith, Chris Batten, and Krste Asanoviç

Power Cost of Frequency

- Power ∝ Voltage^2 x Frequency (V^{2}F)
- Frequency ∝ Voltage
- Power ∝ Frequency^3

	Cores	V	Freq	Perf	Power	PE (Bops/watt)
Superscalar	1	1	1	1	1	1
"New" Superscalar	1X	1.5X	1.5X	1.5X	3.3X	0.45X
Multicore	2X	0.75X	0.75X	1.5X	0.8X	1.88X

(Bigger # is better)

50% more performance with 20% less power

Preferable to use multiple slower devices, than one superfast device

Moore's Law reinterpreted

- Number of cores per chip will double every two years
- Clock speed will not increase (possibly decrease) because of Power

$$Power \propto Voltage^2 * Frequency$$

$$Voltage \propto Frequency$$

$$Power \propto Frequency^3$$

- Need to deal with systems with millions of concurrent threads
- Need to deal with inter-chip parallelism as well as intra-chip parallelism

WHAT WE WILL FIND IN A BOX?

Intel Sandy Bridge (January 6th 2011)

2nd Gen Intel® Core™ Processor Overview

Integrates CPU, Graphics, MC, PCI Express* On Single Chip

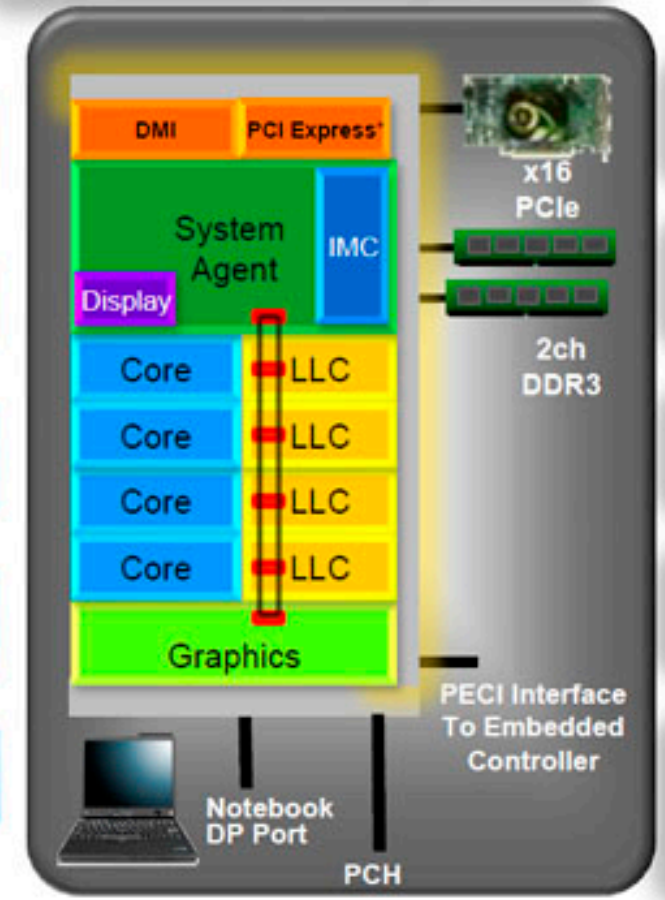
Next Generation Intel® Turbo Boost Technology

High Bandwidth Last Level Cache

Next Generation Processor Graphics and Media

Embedded Display Port

Discrete Graphics Support: 1x16 or 2x8



High BW/low-latency modular core/GFX interconnect

Substantial performance improvement

Intel® Advanced Vector Extension (Intel® AVX)

Integrated Memory Controller 2ch DDR3

Intel® Hyper-Threading Technology
4 Cores / 8 Threads
2 Cores / 4 Threads

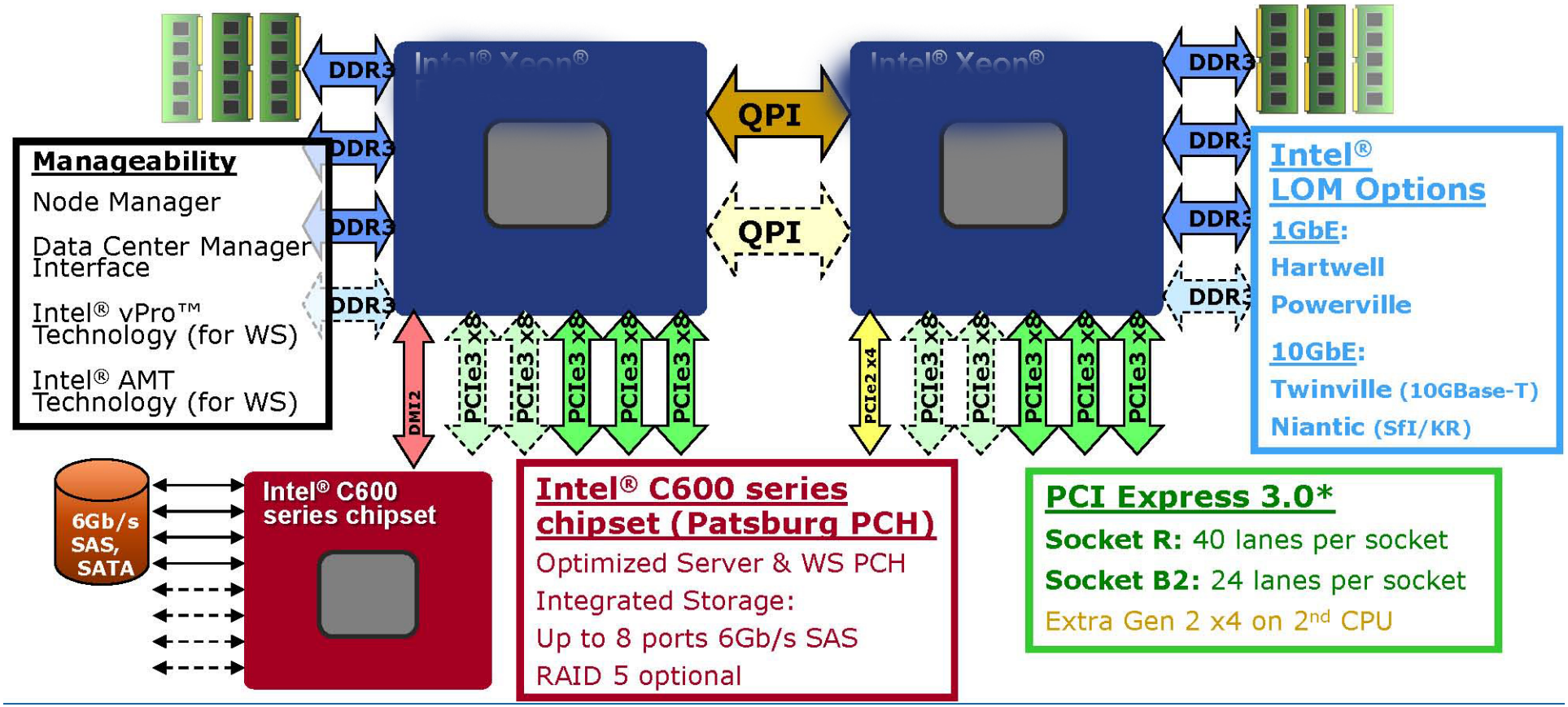
Intel XEON 2012

Read:
4 x 8 cores 8-word wide

Sandy Bridge CPUs
Socket R: Up to 8 cores / socket
Socket B2: Up to 8 cores / socket

QPI
Socket R: 2 QPI links
Socket B2: 1 QPI link

Memory
DDR3 & DDR3L
RDIMMs & UDIMMs, LR DIMMs
Socket R: 4 channels per socket, up to 3 DPC; speeds up to DDR3 1600
Socket B2: 3 channels per socket, up to 2 DPC; speeds up to DDR3 1600



Manageability
Node Manager
Data Center Manager Interface
Intel® vPro™ Technology (for WS)
Intel® AMT Technology (for WS)

Intel® LOM Options
1GbE:
Hartwell
Powerville
10GbE:
Twinville (10GBase-T)
Niantic (SfI/KR)

Intel® C600 series chipset (Patsburg PCH)
Optimized Server & WS PCH
Integrated Storage:
Up to 8 ports 6Gb/s SAS
RAID 5 optional

PCI Express 3.0*
Socket R: 40 lanes per socket
Socket B2: 24 lanes per socket
Extra Gen 2 x4 on 2nd CPU

AMD Buldozer CORE (next quarter)

Bulldozer

What it is:

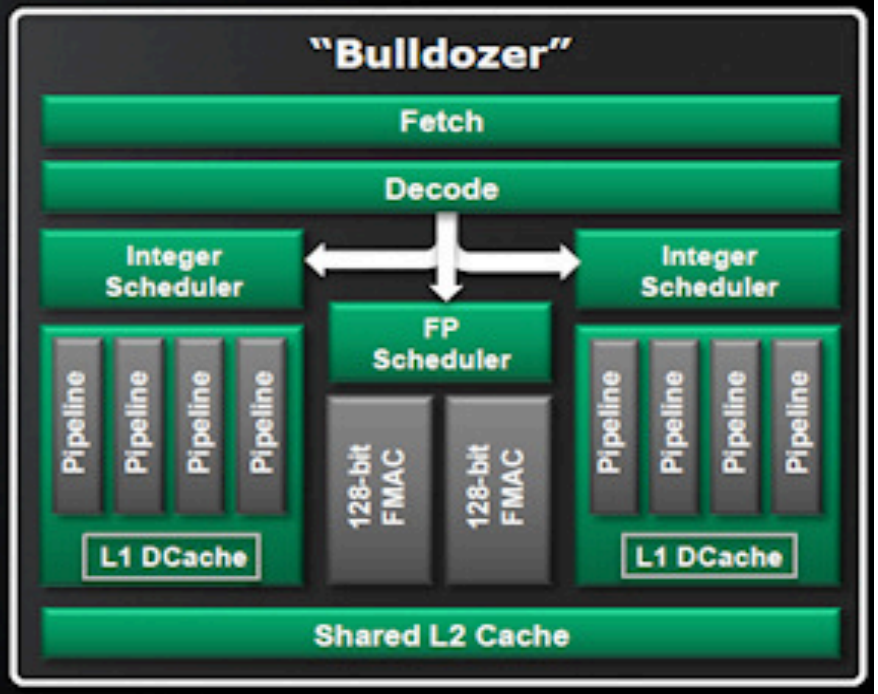
- A monolithic dual core building block that supports two threads of execution

How it works:

- Shares latency-tolerant functionality
- Smoothes bursty/inefficient usage
- Dynamic resource allocation between threads

Customer Benefits:

- Greater scalability and predictability than two threads sharing a single core
- Throughput advantages for multi-threaded workloads without significant loss on serial single-threaded workload components
- When only one thread is active, it has full access to all shared resources
- Estimated average of 80% of the CMP performance with much less area and power *



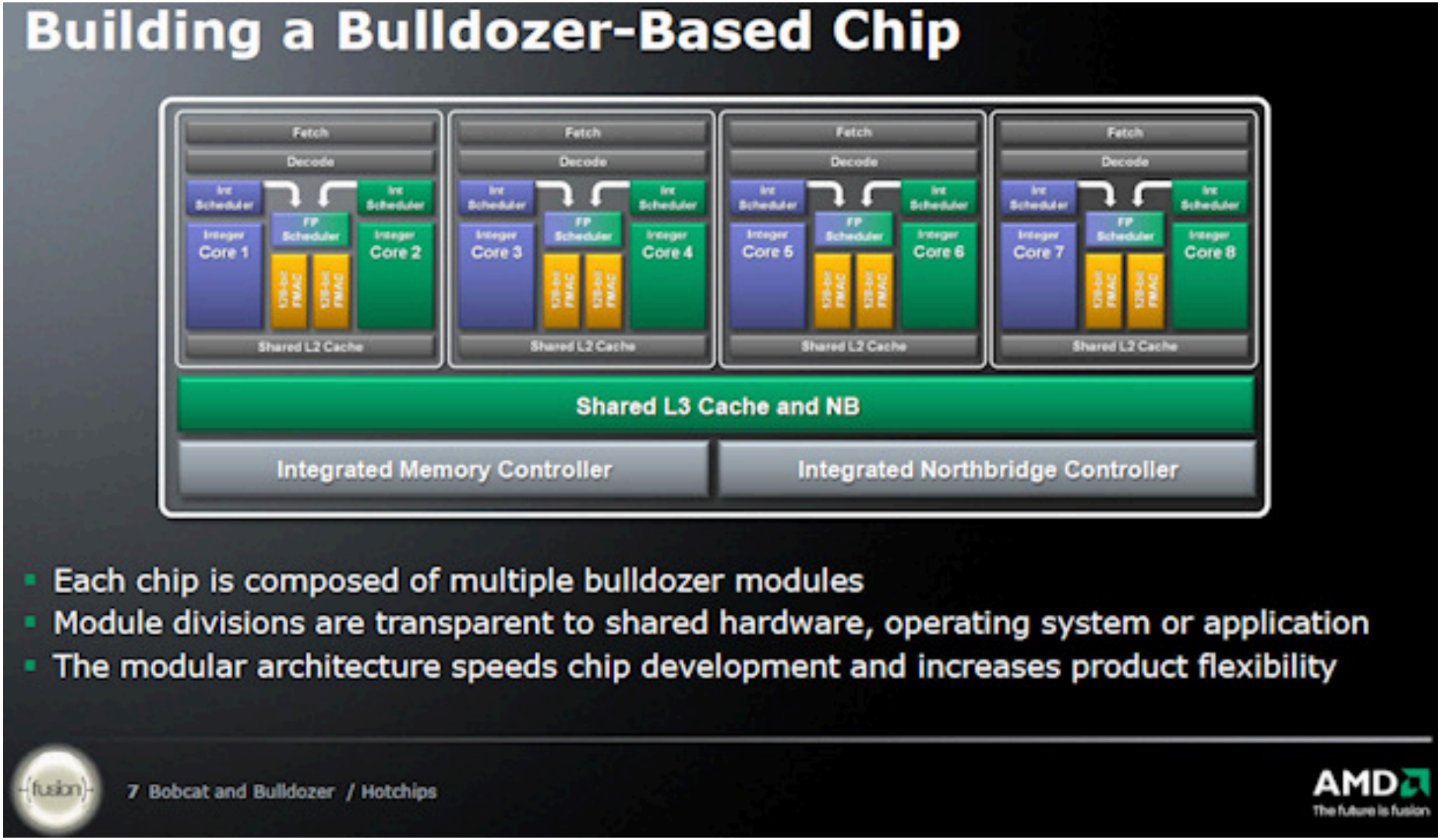
*Based on internal AMD modeling using benchmark simulations

AMD
The future is fusion

AMD Buldozer CHIP (next quarter)

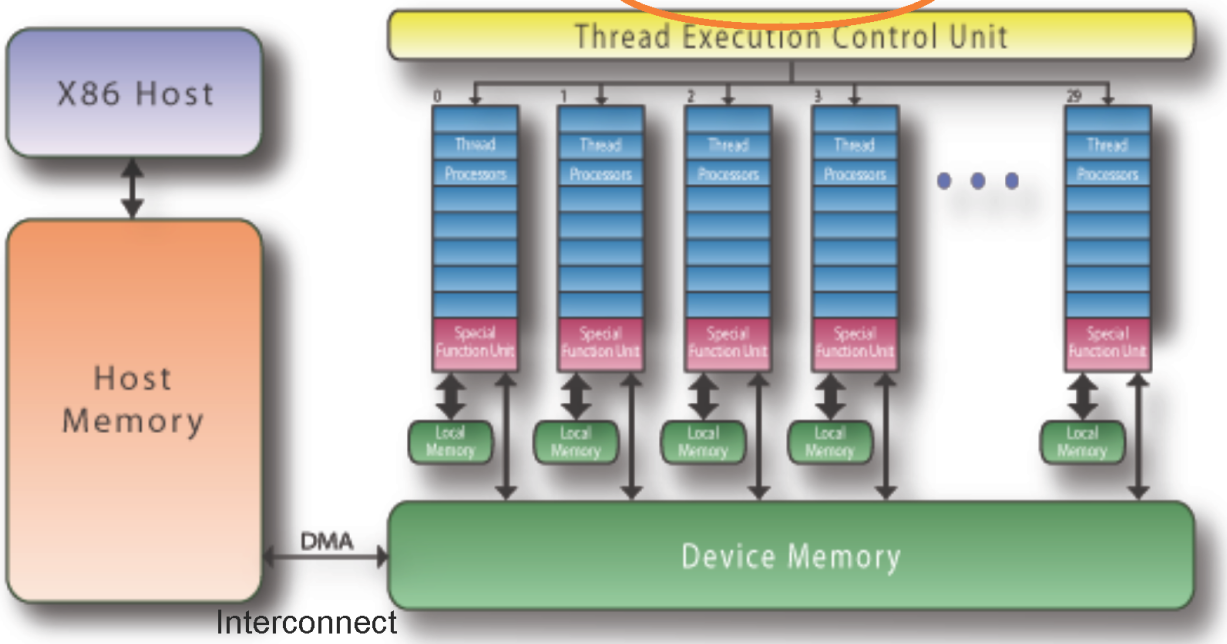
Read:

N x M cores "8-word wide"



Commodity plus Accelerators

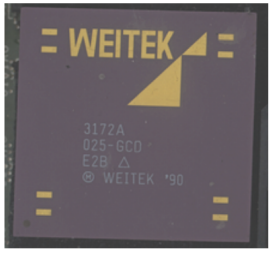
Commodity	Accelerator (GPU)
Intel Xeon	Nvidia C2050 "Fermi"
8 cores	448 "Cuda cores"
3 GHz	1.15 GHz
8*4 ops/cycle	~~448 ops/cycle~~
96 Gflop/s (DP)	515 Gflop/s (DP)

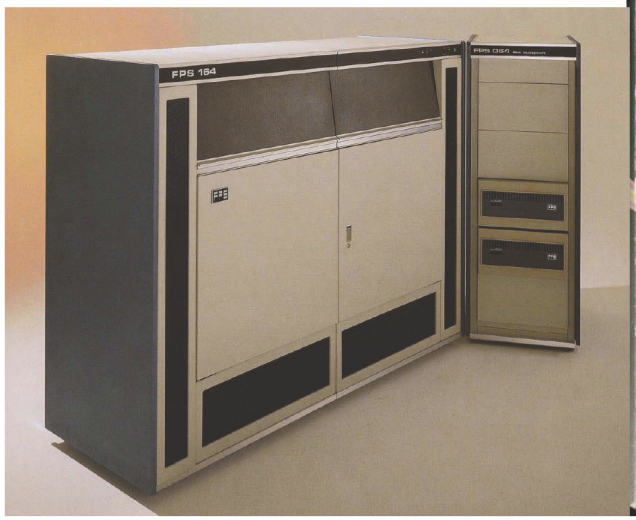


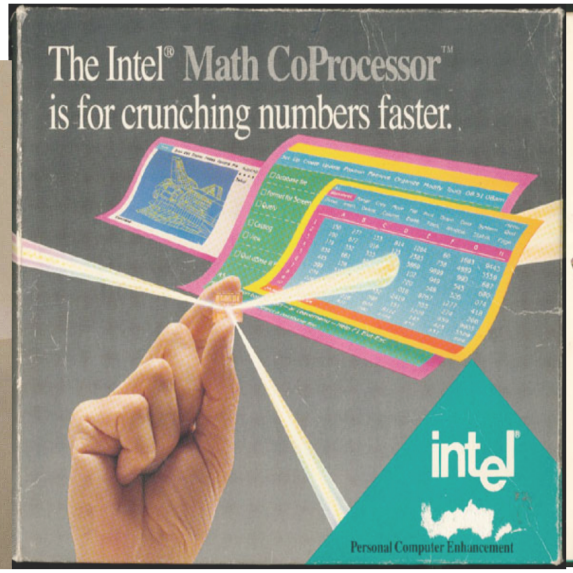
Interconnect
PCI-X 16 lane
64 Gb/s
1 GW/s

We Have Seen This Before

- Floating Point Systems FPS-164 Scientific Computer (1976)
- Intel Math Co-processor (1980)
- Weitek Math Co-processor (1981)





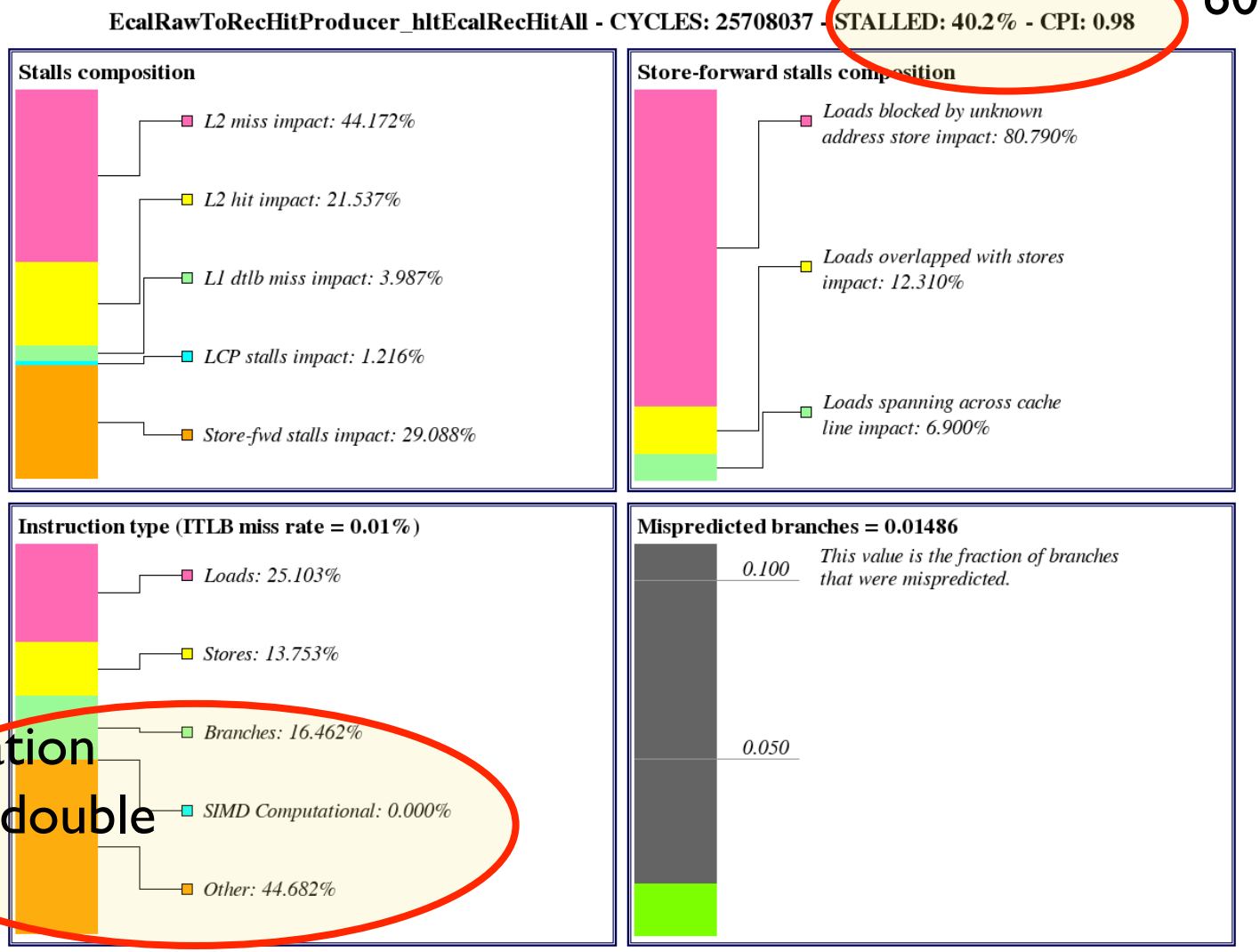


Balance Between Data Movement and Floating point

- FPS-164 and VAX (1976)
  - 11 Mflop/s; transfer rate 44 MB/s
  - Ratio of flops to bytes of data movement: 1 flop per 4 bytes transferred
- Nvidia Fermi and PCI-X to host
  - 500 Gflop/s; transfer rate 8 GB/s
  - Ratio of flops to bytes of data movement: 62 flops per 1 byte transferred
- Flop/s are cheap, so are provisioned in excess

Dominated by data movement NOW!
We use only 15% of availiable "d"flops

60% "active"

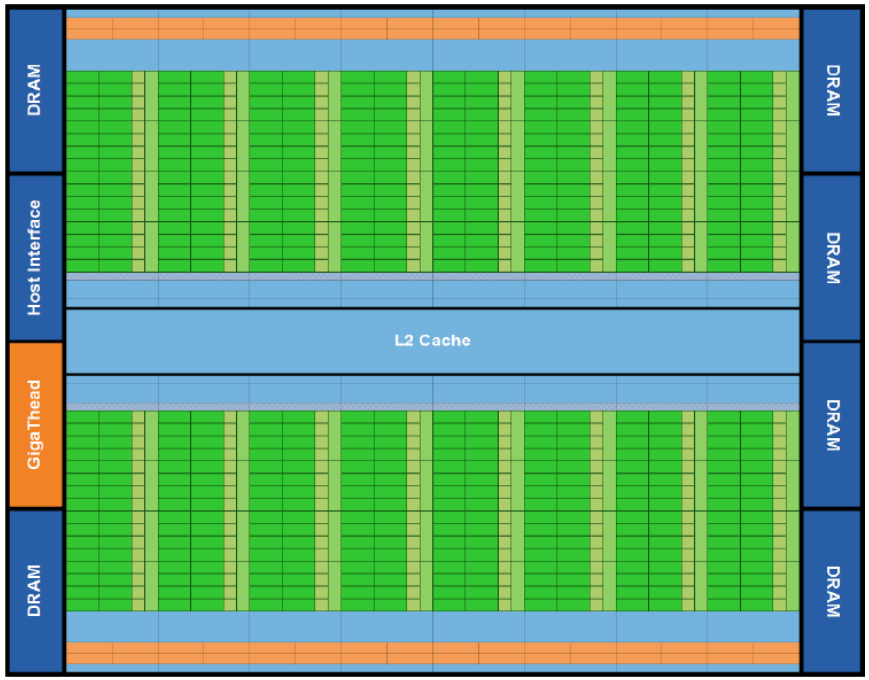


50% "computation on single/double word"

nVidia Fermi Architecture

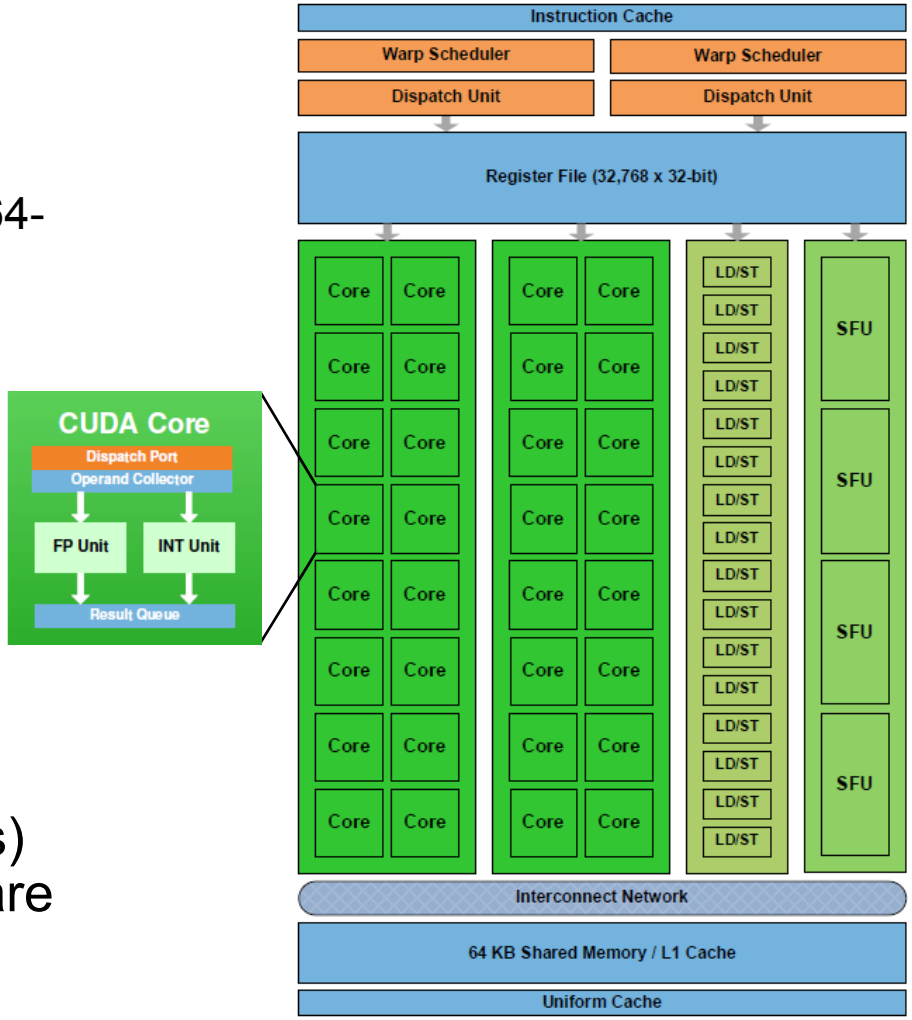
Read:
16 cores 32-world wide

- Up to 512 cores
  - 16 Streaming multiprocessors each with 32 cores @ 1.3GHz
- Parallel DataCache
  - 64 KB Shmem/L1 Cache
  - 768 KB Unified L2 Cache
- Six 64-bit memory partitions
  - 384-bit memory interface
  - Up to 6 GB GDDR5 DRAM
- Up to 16 concurrent kernels
- IEEE floating point math
- ECC memory



Fermi Streaming Multiprocssor Architecture

- 32 Cores
  - 32-bit Integer ALU with 64-bit extensions
  - Full IEEE 754-2008 32-bit and 64-bit precision
- 64 KB Shared Memory/L1 cache
  - 16KB Shmem/48KB cache or 48KB Shmem/16KB L1 cache
- 16 load/store units
- Dual Warp scheduler (dual instruction issue)
- Four Special Function Units (SFUs) for sin, cosine, reciprocal, and square root operations



Comparison to Previous nVidia GPGPUs

GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Special Function Units (SFUs) / SM	2	2	4
Warp schedulers (per SM)	1	1	2
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

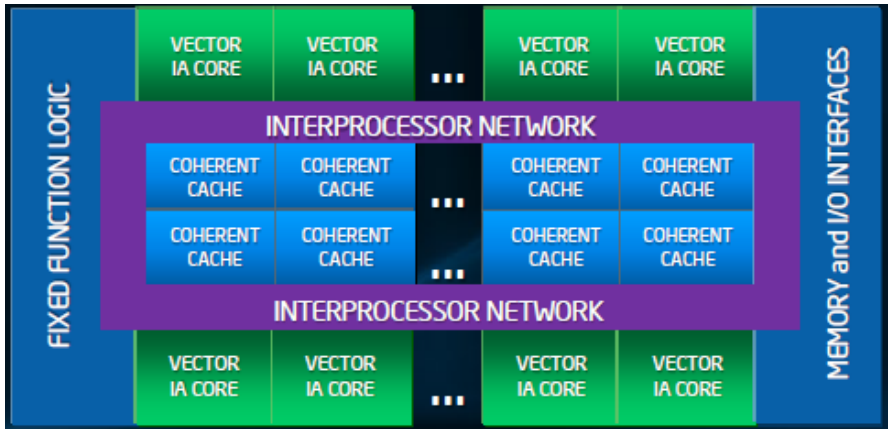
Intel MIC Architecture

Read:
32 cores 16-word wide

Pronounced “Mike”

Many cores with many threads per core

Standard IA programming and memory model



Knights Ferry

- Software development platform
- 1-2GB GDDR5 connected to host memory through PCI DMA operations with virtual addressing
- Intel HPC developer tools

32 Cores @ 1.2 GHz

- ✓ 4 threads/core, 128 total parallel threads
- ✓ 32KB i-cache, 32KB d-cache
- ✓ 256KB coherent L2 cache (8MB total)
- ✓ 512bit vector unit
  - 16 Single precision FLOPs/clock
  - 8 Double precision FLOPS/clock

MIC Programming Environment

- Inherently supports OpenMP.
- Virtual memory environment extends back to host memory.
- Intel Parallel Studio and Cluster Studio support MIC.
- Optimizing performance will take almost as much effort as for CUDA and OpenCL environments.

Knights Corner
1st Production MIC Co-processor

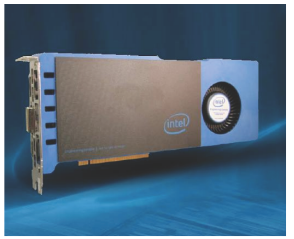
- Second Half 2012
  - Knowns:
    - 50+ cores
    - 22nm manufacturing process
  - Unknowns:
    - Core frequency
    - Size of GDDR5 memory on board
    - ECC support

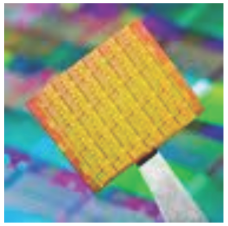
Co-processor Comparison

	AMD Firestream	NVIDIA Fermi	Intel Knights Ferry	Intel Knights Corner Speculation	Intel Knights Corner Speculation2
Cores	1600	512	32*4 threads/core = 128	50*4 threads/core = 200	64*4 threads/core = 256
Core Frequency	700/825 MHz	1.3 GHz	1.2 GHz	1.2 GHz	2 GHz
Thread Granularity	fine	fine	coarse	coarse	coarse
Single Precision Floating Point Capability GFLOPs	2000/2640	1024	614	960	2048
Double Precision Floating Point Capability GFLOPs	400/528	512	307	480	1024
GDDR5 RAM	2/4 GB	3-6 GB	1-2 GB	?	?
L1 cache/processor		64KB (16KB Shmem, 48KB L1 or 48KB Shmem, 16KB L1)	64KB (32KB icache, 32KB dcache)	64KB (32KB icache, 32KB dcache)	64KB (32KB icache, 32KB dcache)
L2 cache/processor		768KB shared L2	8MB coherent total (256KB/core)	12MB coherent total (256KB/core)	16MB coherent total (256KB/core)
programming model		CUDA kernels	posix threads	posix threads	posix threads
virtual memory		no	yes	yes	yes
memory shared with host		no	no	no	no
Software	OpenCL, DirectCompute	C, C++, CUDA, OpenCL, DirectCompute	C, C++, FORTRAN, OpenMP, CUDA, OpenCL, DirectCompute	C, C++, FORTRAN, OpenMP, CUDA, OpenCL, DirectCompute	C, C++, FORTRAN, OpenMP, CUDA, OpenCL, DirectCompute

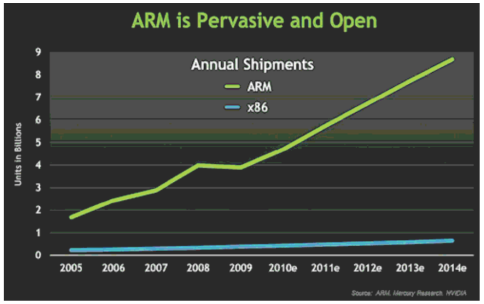
Future Computer Systems

- Most likely be a hybrid design
  - Think standard multicore chips and accelerator (GPUs)
- Today accelerators are attached
- Next generation more integrated
- Intel's MIC architecture "Knights Ferry" and "Knights Corner" to come.
  - 48 x86 cores
- AMD's Fusion in 2012 - 2013
  - Multicore with embedded graphics ATI
- Nvidia's Project Denver plans to develop an integrated chip using ARM architecture in 2013.





AMD
The future is fusion



Content of a box (server)

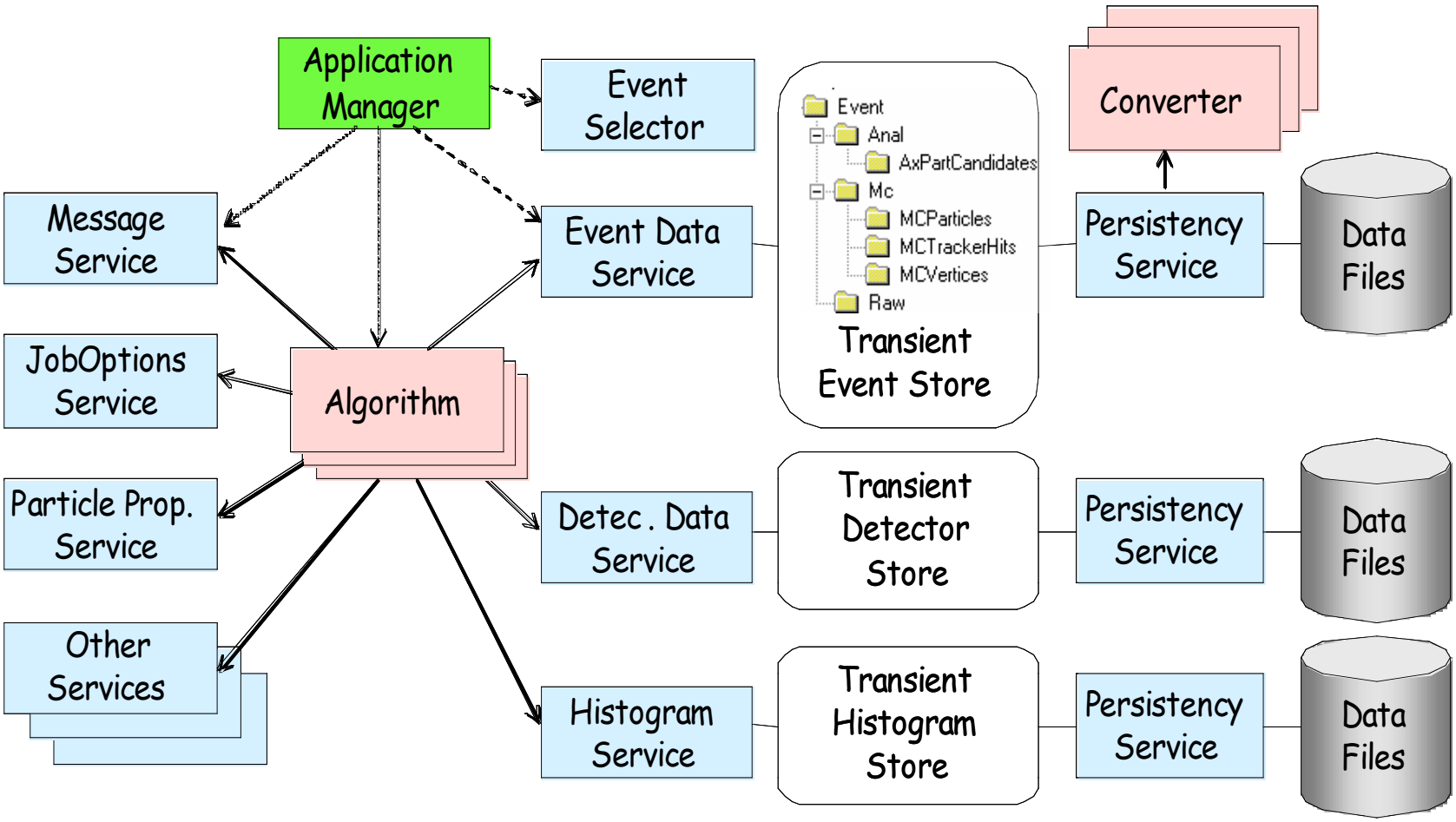
- Very soon
  - » 4 highly interconnected chips with 8 “vector”-core each
  - » Fast access to peripherals
- Soon
  - » As above + one (or more?) coprocessor(s)
  - » Faster and faster communication among host and coprocessor
  - » Better and better sharing of resources (memory)
- Not so far in future
  - » Core and coprocessors integrated in a chip or very close together
  - » Seamless instruction and data sharing among them

WHAT TO DO WITH SUCH A BOX?

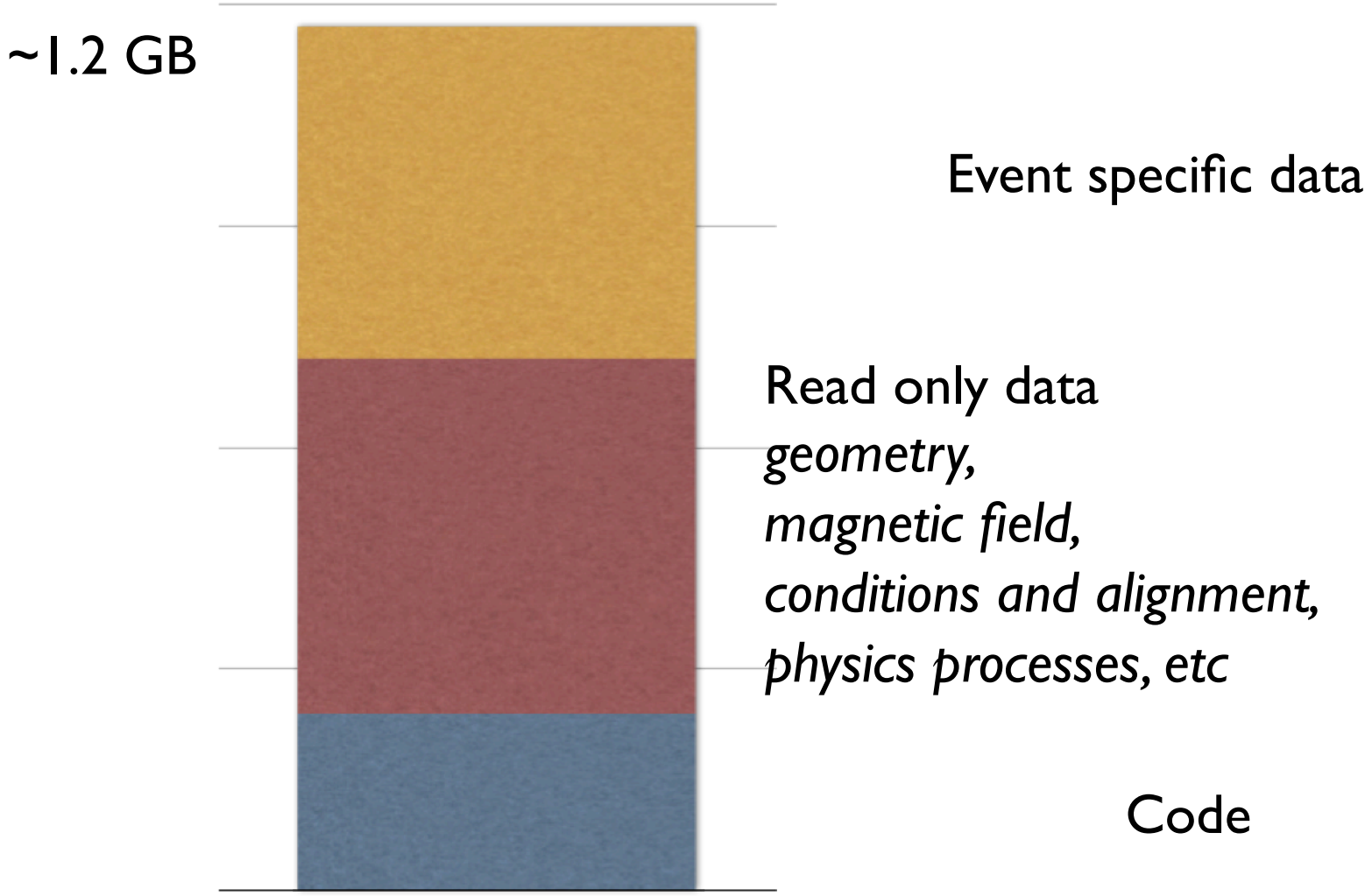
Optimization on many core

- Efficient use of shared resources
  - » Main memory, shared caches, I/O
- Minimize communication
  - » Including back-&-forth to main memory
- Remove synchronization-barriers
  - » Mostly implicit in traditional sequential scheduling
- Streamline code to allow vectorization
- New programming paradigm
  - » Think local and parallel!
  - » Decompose a problem vertically (parallel) first, then horizontally (sequentially)
  - » Consider speculative computation in place of likely miss-predicted branches
  - » Prefer deterministic algorithm to recursion, hit/miss

HEP Application

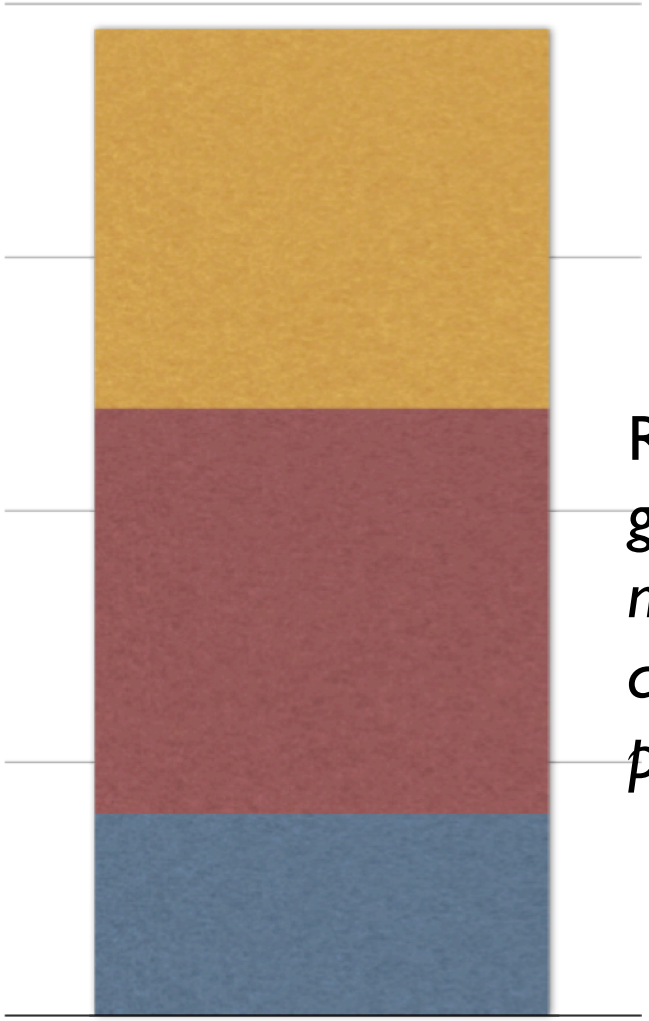


CMS offline software memory budget



CMS offline software memory budget

~1.2 GB



Event specific data

Read only data
geometry,
magnetic field,
conditions and alignment,
physics processes, etc

Code

COMMON!

CMS near future multicore strategy:
forking

Event specific data

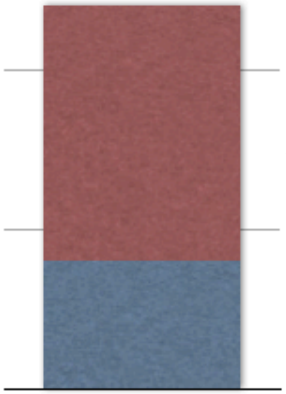
Event specific data

Event specific data

Event specific data

Event specific data

Event specific data



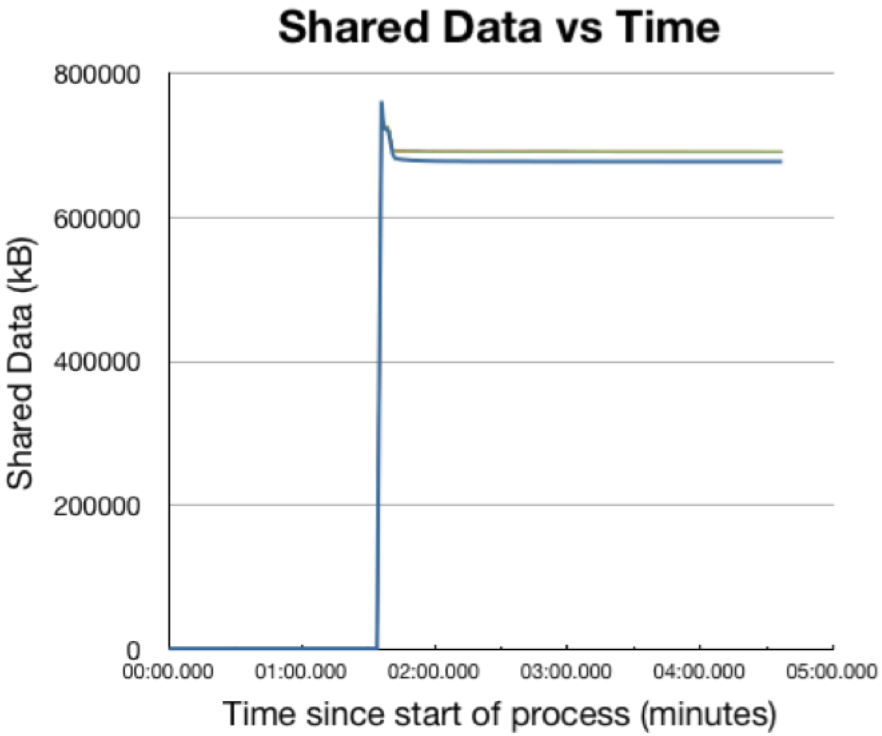
Shared common data

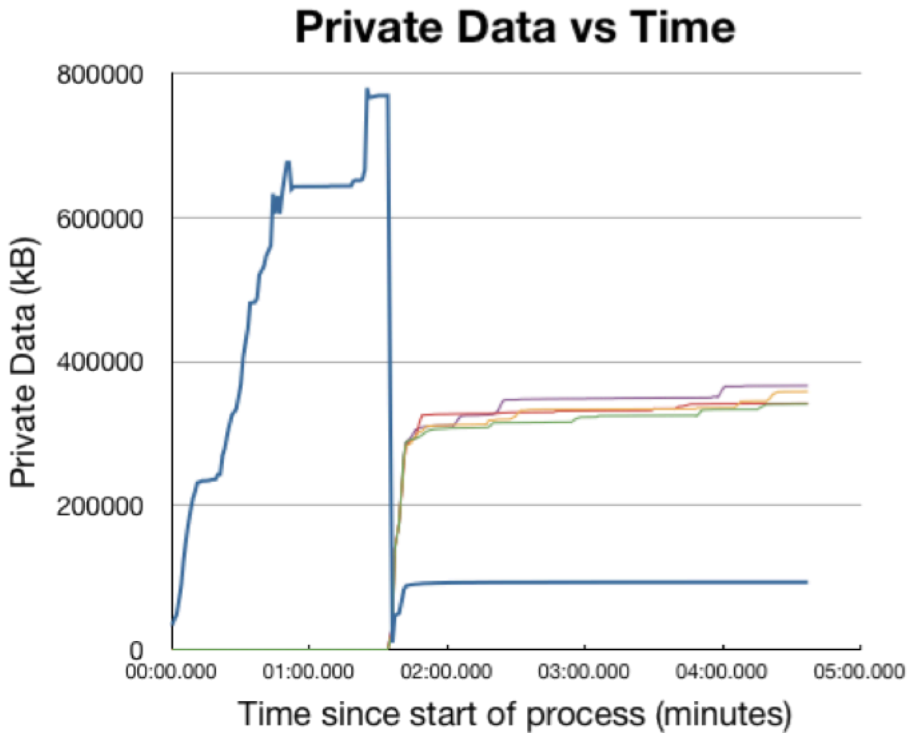
C-o-W*

- Most (all?) of the common const data / code can actually be brought in the application very early
- If you fork at that point, the kernel is actually smart enough to share the common data memory pages between parent and the children
- The kernel “un-shares” the memory pages only when one of the processes writes to them
- New allocations (i.e. event data) end up in non shared pages

* Copy-on-Write

Forking: memory sharing



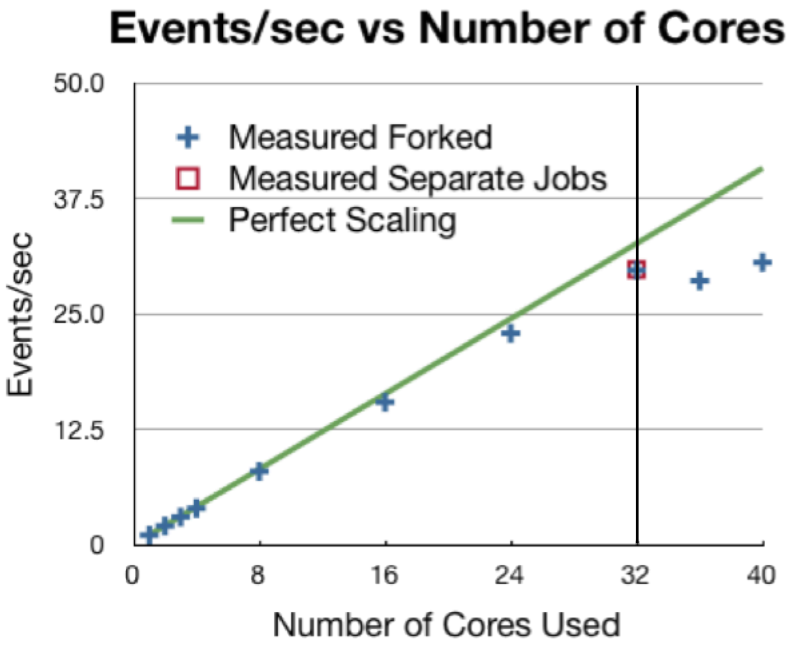


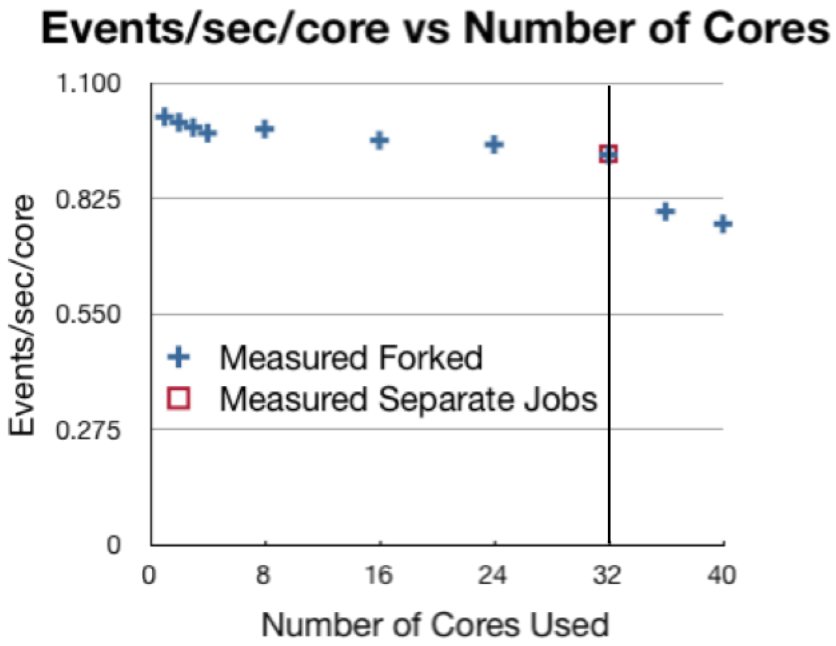
Measurements done using reconstruction with 64bit software on 4 CPU, 8 core/CPU 2GHz AMD Opteron(tm) Processor 6128

Shared memory per child: ~700MB
Private memory per child: ~375MB
Total memory used by 32 children: 13GB
Total memory used by 32 separate jobs: 34 GB

We suddenly have lots of free memory available

Forking: throughput





Problems

– I/O “consolidation” non solved yet

» There are still multiple input and output buffers plus independent output streams

» Will require introducing explicit distributor and collector processes.

– Memory accounting gets more complicated

*All the nice accounting tools we had for **RSS** memory are now useless. We need something which is capable to keep sharing of pages into account. See http://www.selenic.com/smem/ and http://lwn.net/Articles/230975/ for some ideas.*

– Deleting “common” objects make them non-shared!

This actually a problem when you are border-line with memory usage. The final deletion of common part has to be avoided to prevent a swap-storm.

A newly designed framework may prefer an explicit shared memory model

“Whole-node” scheduling

Exploiting this new processing model requires a new model in computing resources allocation as well

Experiments need to have control over a larger quantum of resources

as multi-core aware jobs require scheduling of multiple cores at the same time

Correct resource accounting fundamental (and gets trickier)

Whole-Node Job Submission Task Force*

whole-node-task-force@cern.ch

(mandated by WLCG-MB, chaired by Peter Elmer)

“Whole-node” scheduling

One natural unit in the system is the “whole node”: the physical thing running one copy of the OS and sharing a set of resources (CPU, disk, network, etc.)

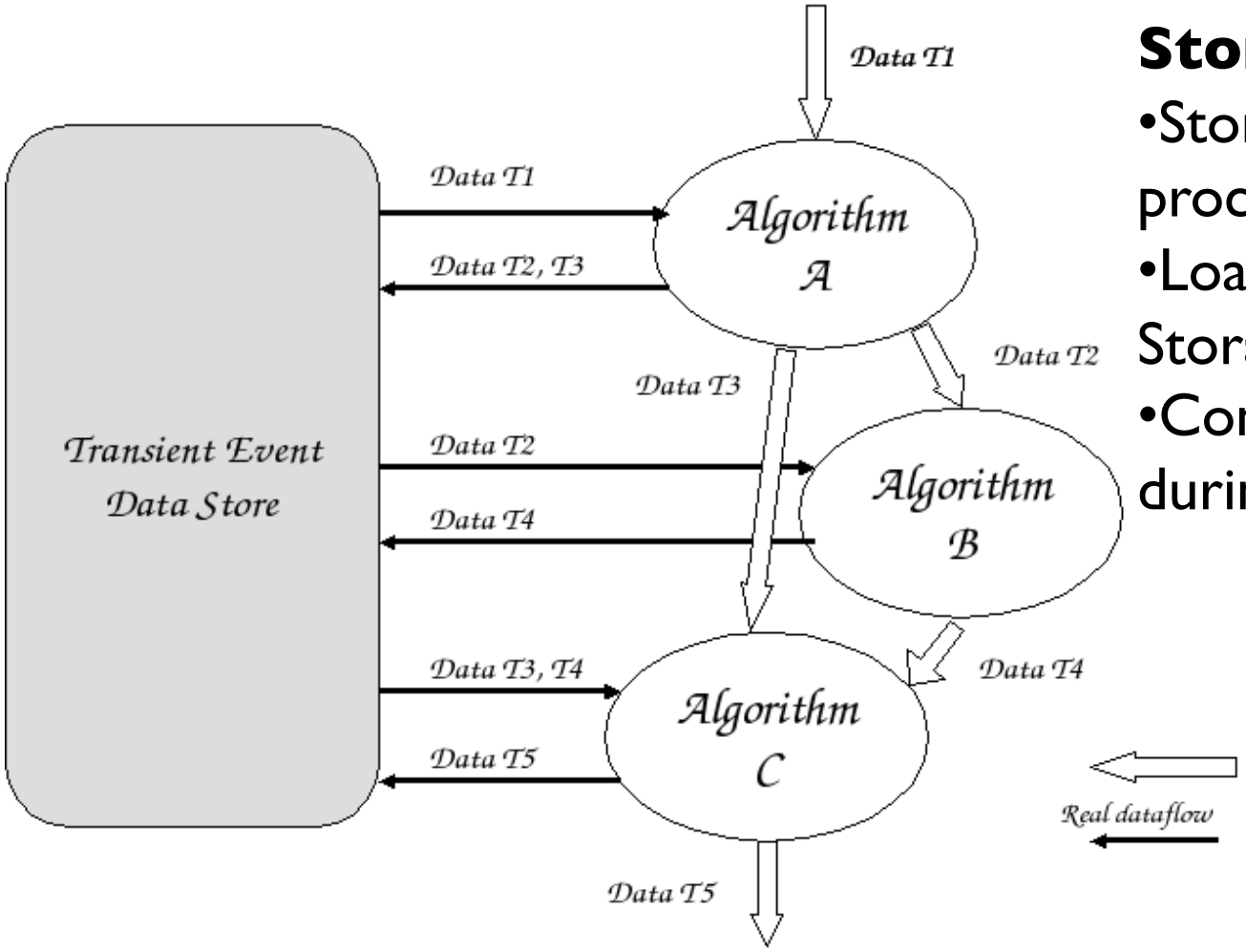
The applications explicitly take over the management of the sharing of resources within the “whole node” quantum of resources

Compatible with current *modus-operandi*, will allow moving to **forking** / multi-threading, allowing for optimization of data/workflow management: I/O caching, local merging, etc

Sites only need to care about the whole node, not individual processes

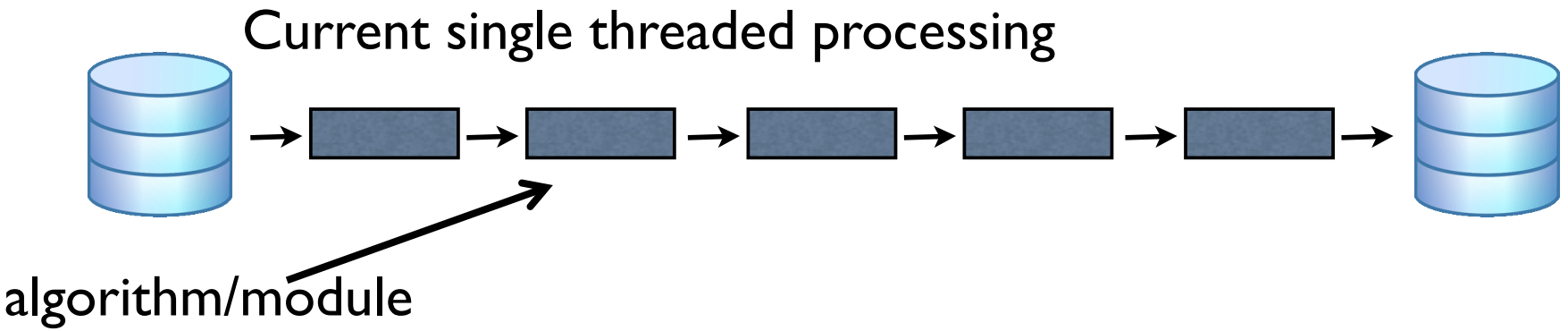
A move to a proper “whole node” accounting for CPU / memory use, etc. recognizes the role of the OS in optimizing access to resources

Gaudi : HEP Event Processing

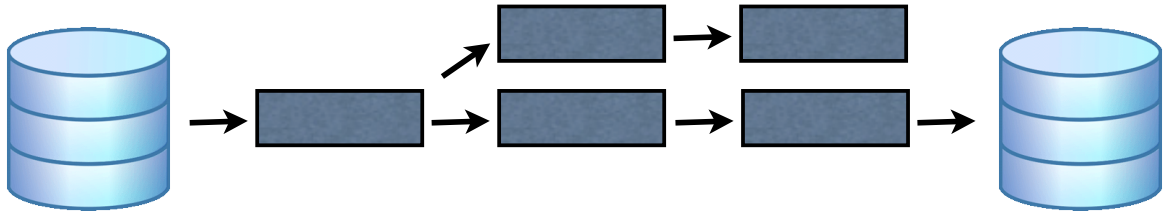


- **Transient Event Store** : Part of Framework
- Stores *DataObjects* during processing
- Loaded from Persistent Storage at Start
- Constantly modified during run

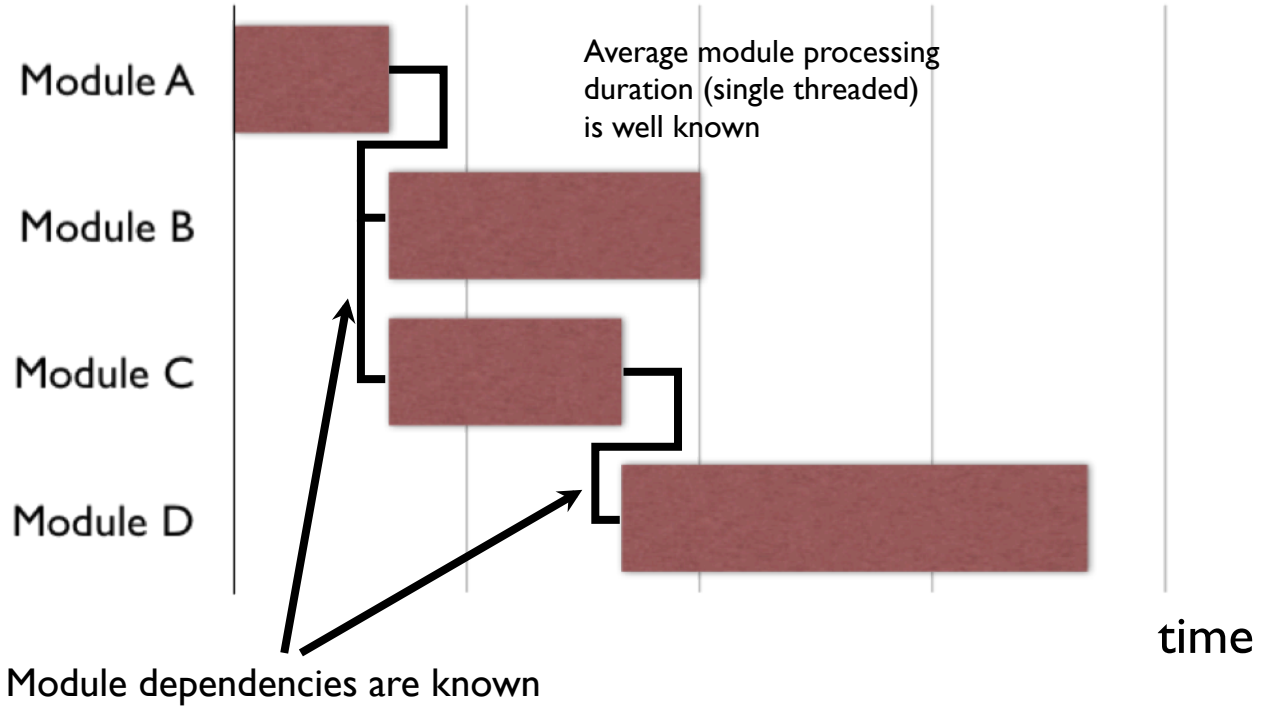
Multi-threading scheduler

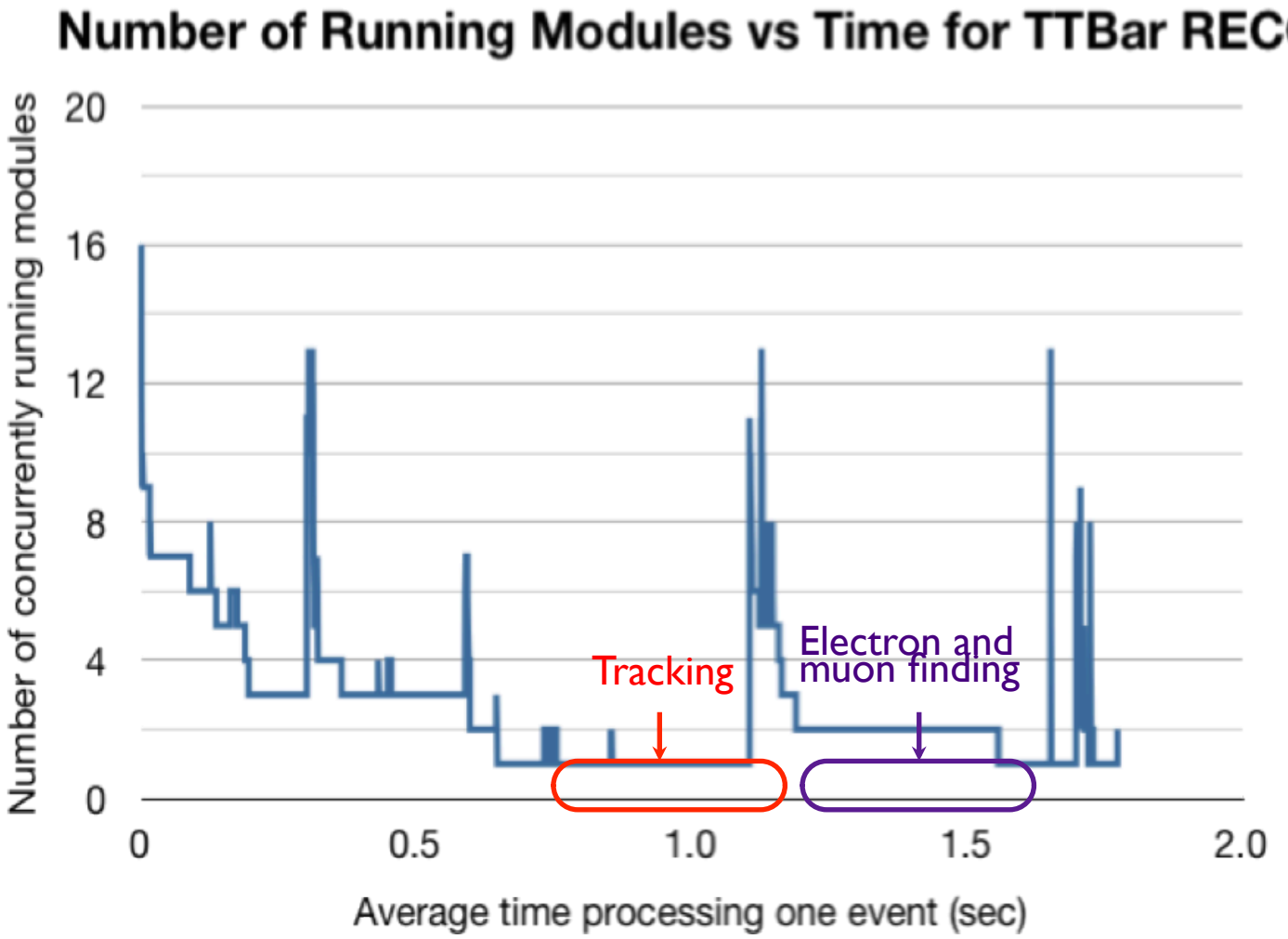


Unrelated parts could be elaborated by separate threads to increase throughput



Behavior / bottlenecks can be “estimated” even now

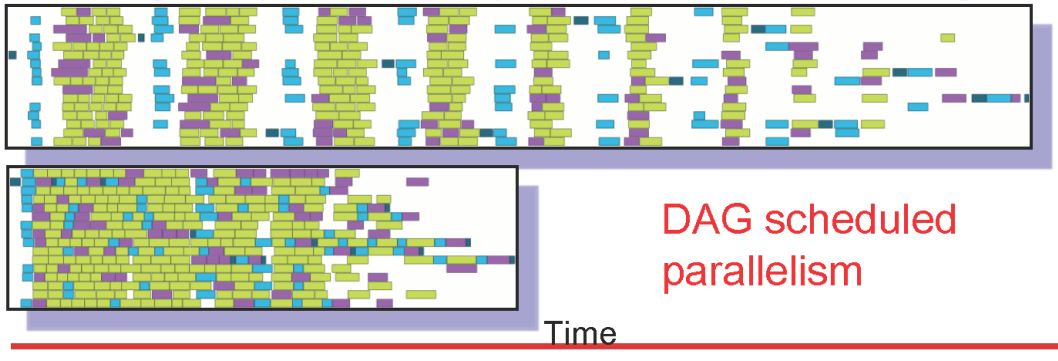


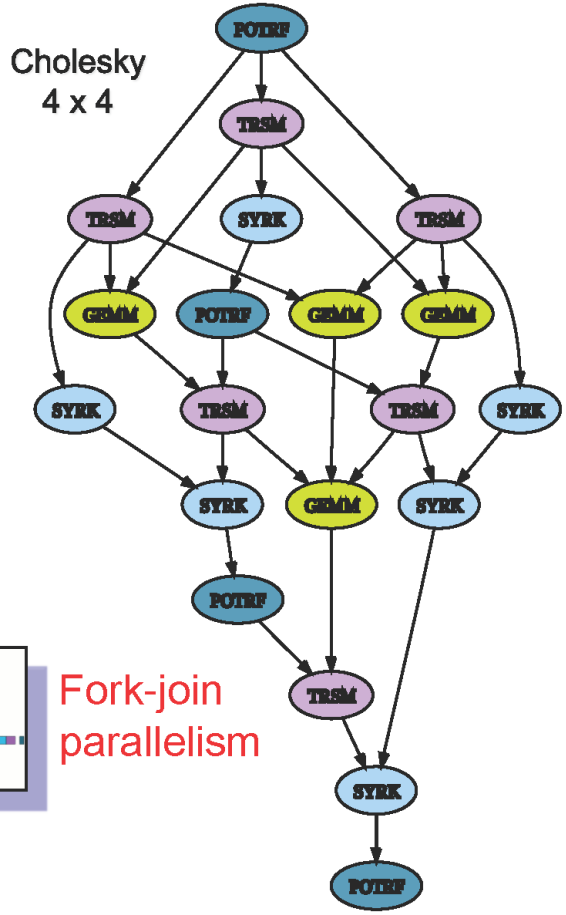


Not worth with current tracking algorithms.

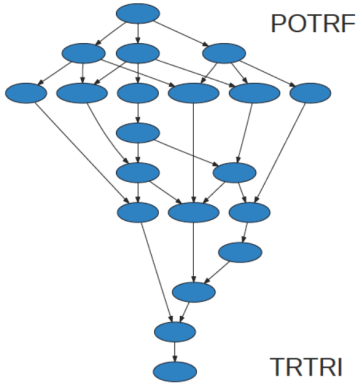
PLASMA: Parallel Linear Algebra s/w for Multicore Architectures

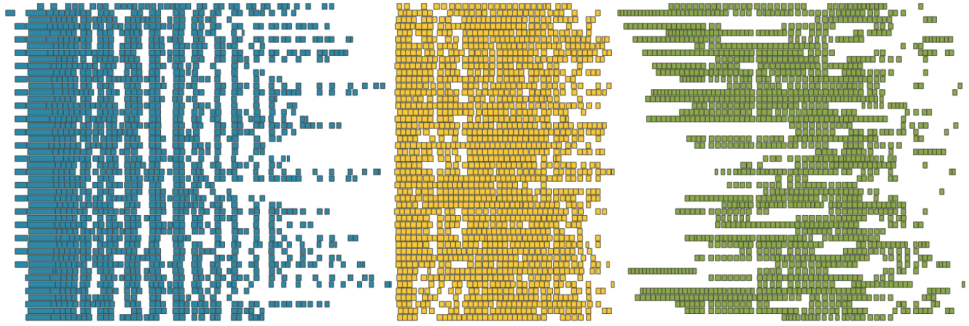
- Objectives
  - High utilization of each core
  - Scaling to large number of cores
  - Shared or distributed memory
- Methodology
  - Dynamic DAG scheduling (QUARK)
  - Explicit parallelism
  - Implicit communication
  - Fine granularity / block data layout
- Arbitrary DAG with dynamic scheduling

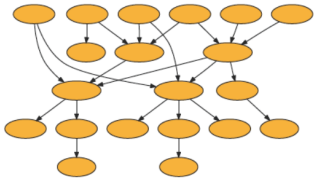


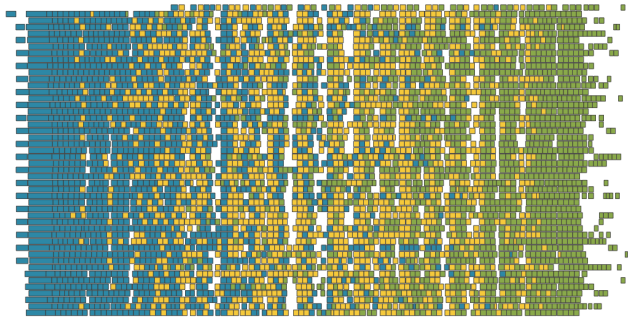


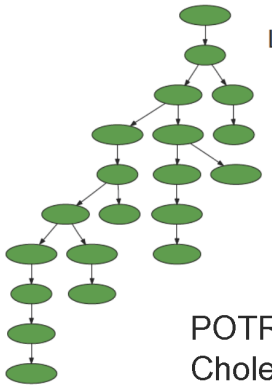
Pipelining: Cholesky Inversion
3 Steps: Factor, Invert L, Multiply L's





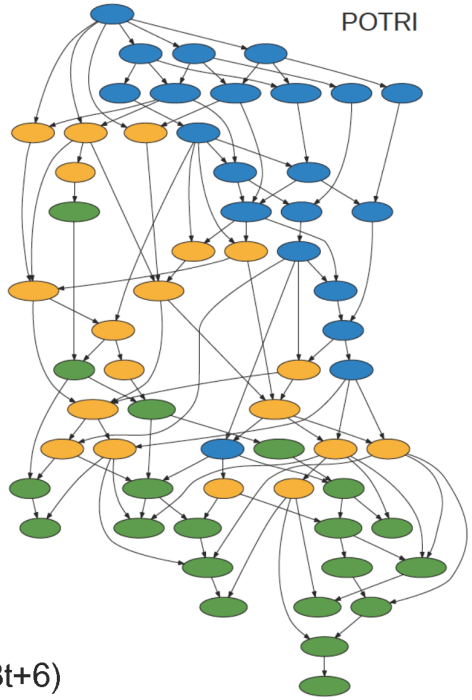






48 cores
POTRF, TRTRI and LAUUM.
The matrix is 4000 x 4000,tile size is 200 x 200,

POTRF+TRTRI+LAUUM: 25 (7t-3)
Cholesky Factorization alone: 3t-2

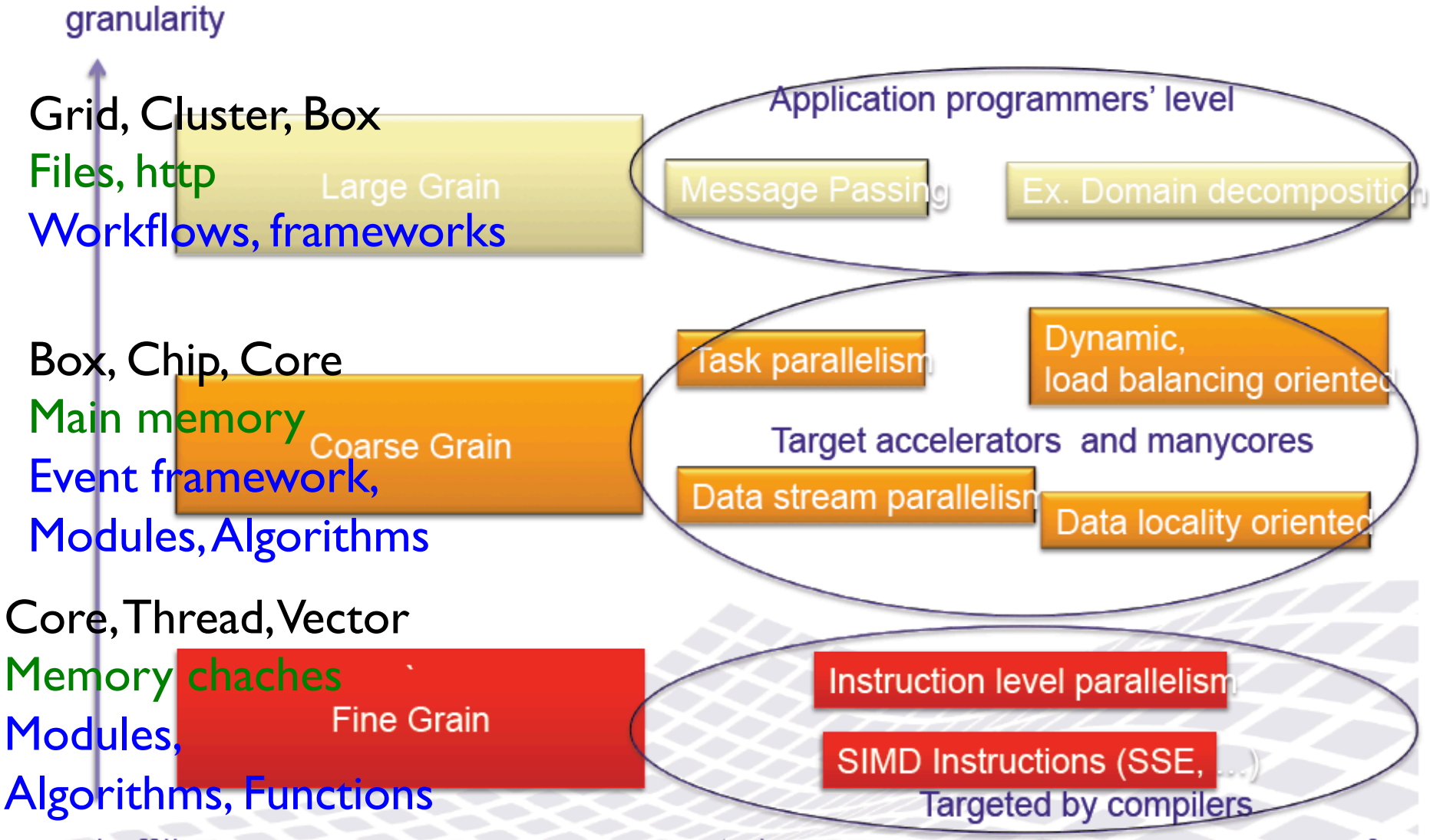


Pipelined: 18 (3t+6)

Toward an Effective Parallel Architecture

Heterogeneous hardware, multiple parallelism forms

CAPS



Hybrid Programming for Future Manycores

CAPS

- Agnostic programming is paramount
  - Highlight parallelism not its implementation
- Use the right parallelism level for each part
  - Software engineering is important
  - Separate application issues from performance issues
    - Specialized components, libraries, …
- Do no expect a common programming API for all levels
  - API always makes some underlying architecture assumptions
    - Fixing API makes hypothesis on the future of architectures
  - No low level programming API common to all devices
  - An API addresses a specific hardware component as a consequence we need many
- Plan for debugging and tuning
  - Parallel bugs are nasty
  - Tuning is target specific

Design for efficient parallelism

- Fuzzy boundaries among "computational domains"
  - » Workflows, applications, event, detector, "data-object"
- Look for innovative (revolutionary?) problem decompositions
  - » Identify *sizable computational chucks* that replicates on "identical" data
    - **down to the lowest possible granularity!**
- Compilers and cpu can be quite smart, still…
  - » They still need "help" from the software-developers
  - » Naïve OO design can easily obfuscate the actual "communication" (memory access) and computation pattern

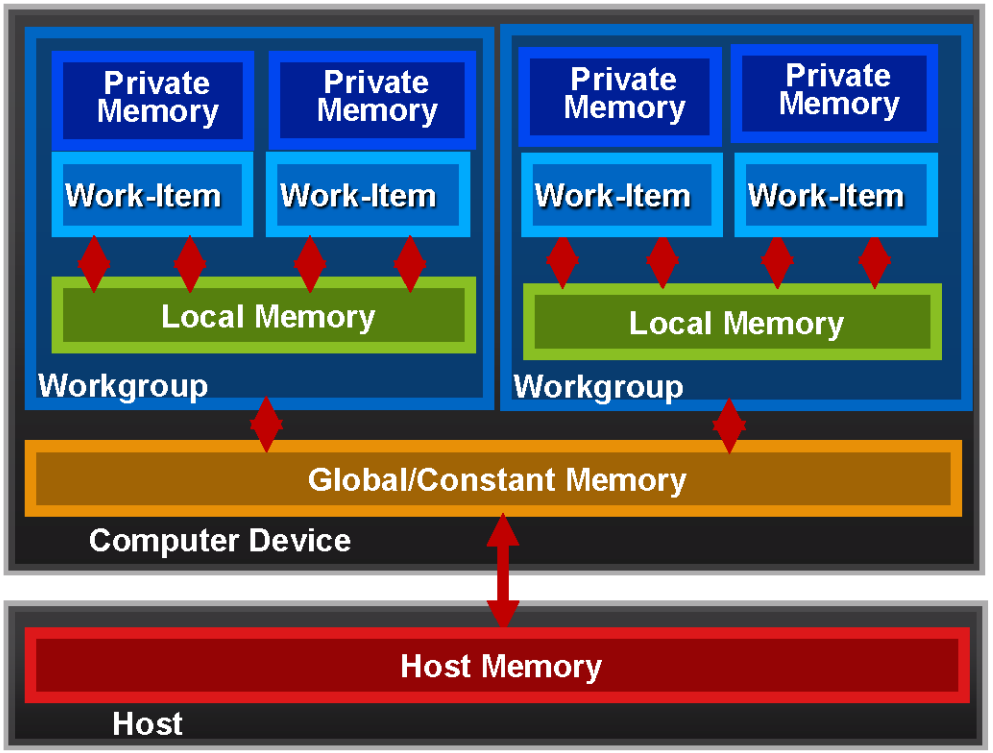
Parallel programming is "tricky":

Better to get inspiration from a model that works. Look at OpenCL

- » Memory Model ("notebooks" vs "whiteboard")
- » Scheduling (commands queued to devices in a context)
- » Task synchronization using "events" (effective DAG)

OpenCL Memory Model

- **Private Memory**
  - Per work-item
- **Local Memory**
  - Shared within a workgroup
- **Local Global/Constant Memory**
  - Visible to all workgroups
- **Host Memory**
  - On the CPU



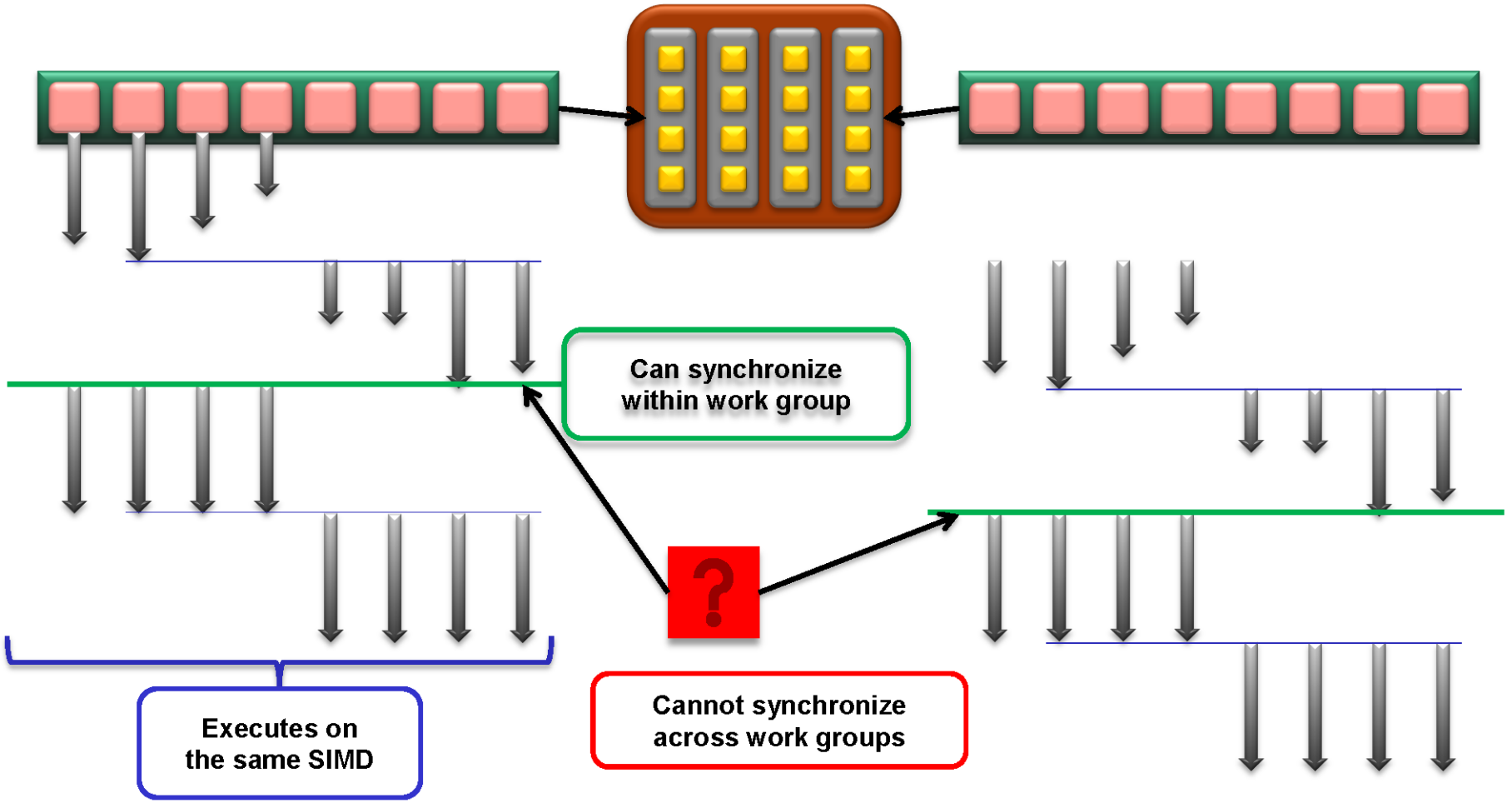
- **Memory management is explicit**
  You must move data from host -> global -> local *and* back

Whiteboard => Notebooks => Whiteboad

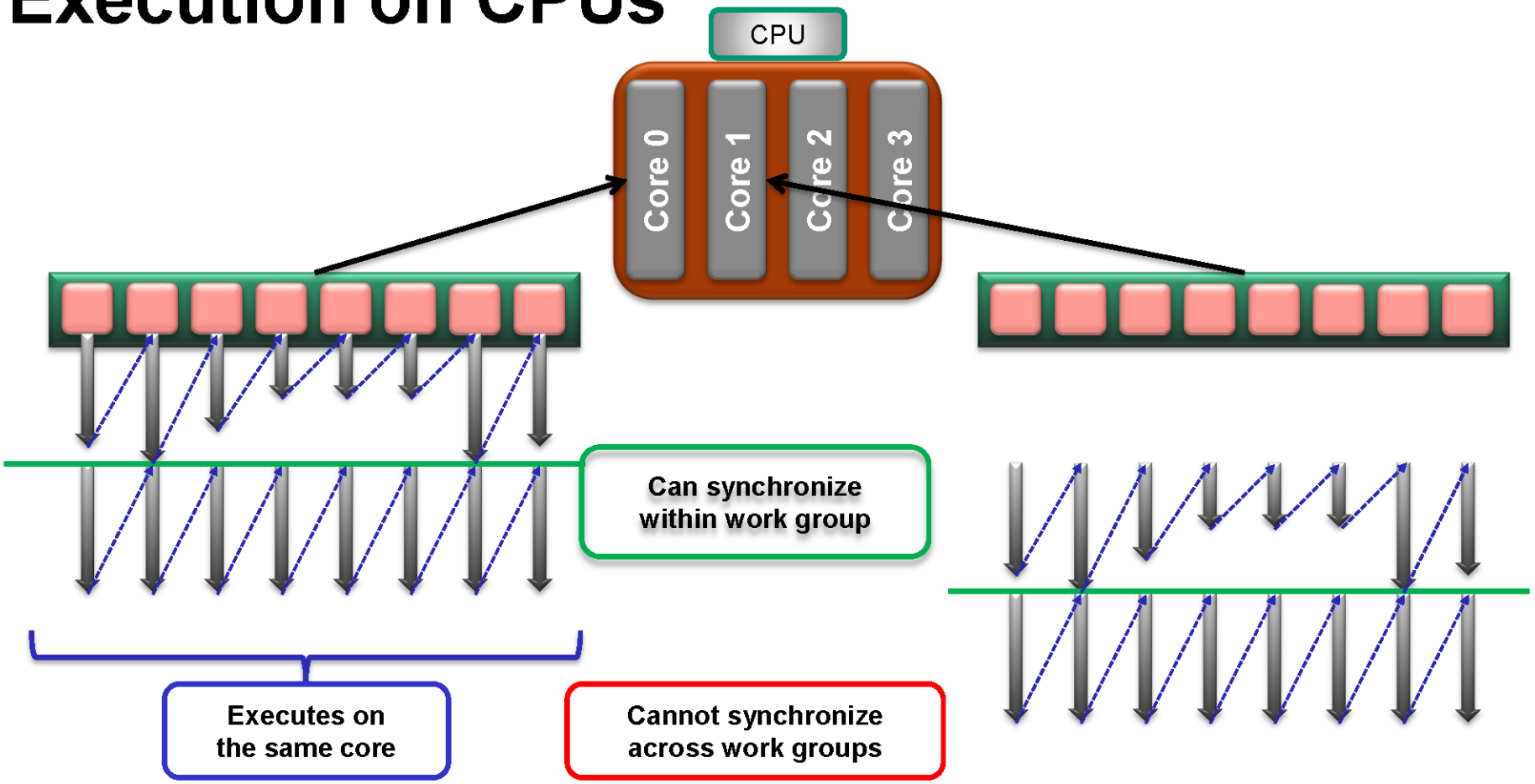
Memory Consistency

- **"OpenCL uses a relaxed consistency memory model; i.e.**
  - the state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times."
- **Within a work-item:**
  - Memory has load/store consistency to its private view of memory
- **Within a work-group:**
  - Local memory is consistent between work-items at a barrier
- **Global memory is consistent within a work-group, at a barrier, but not guaranteed across different work-groups**
- **Consistency of memory shared between commands (e.g. kernel invocations) are enforced through synchronization (events)**

Execution on GPUs

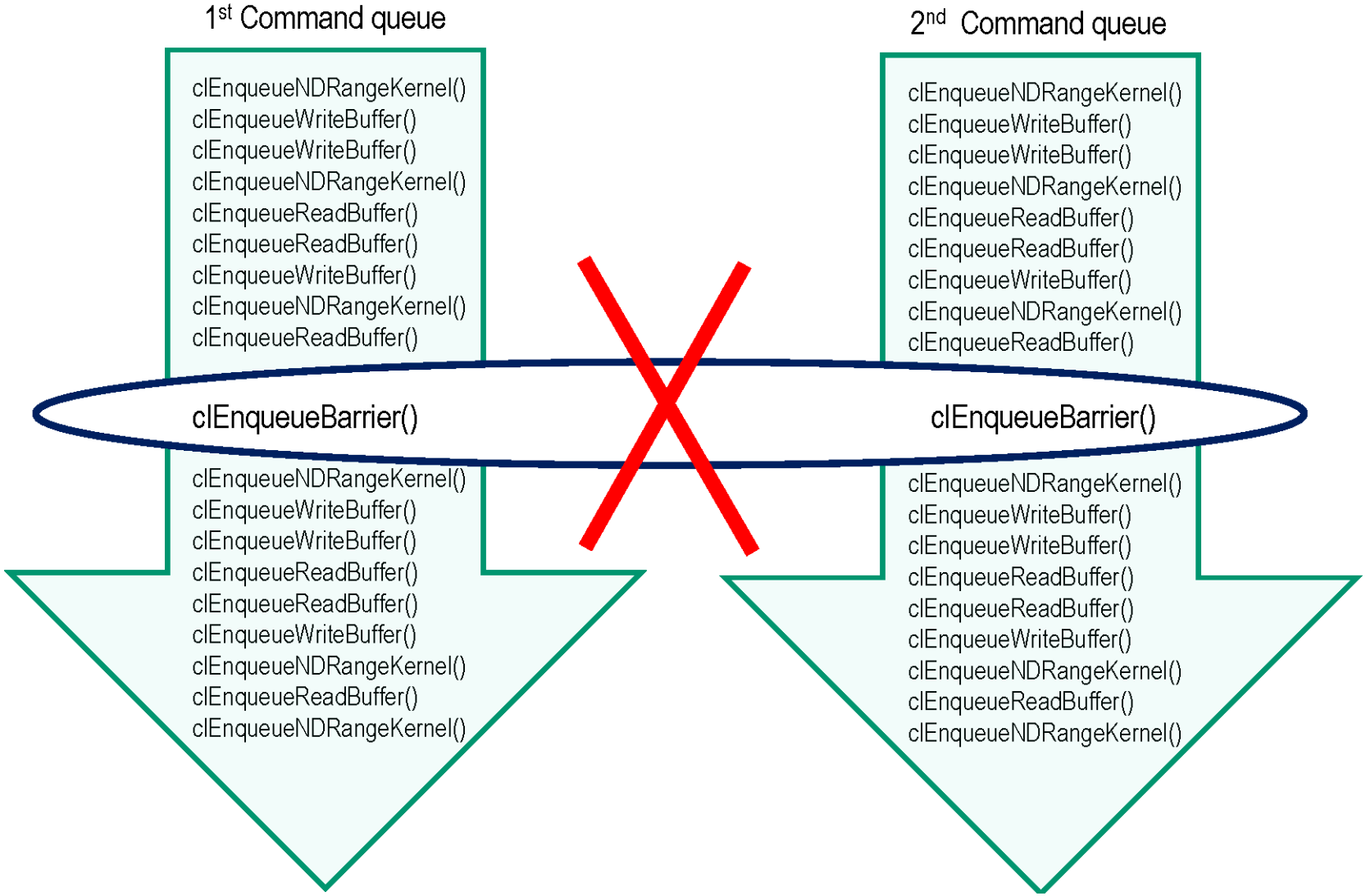


Execution on CPUs

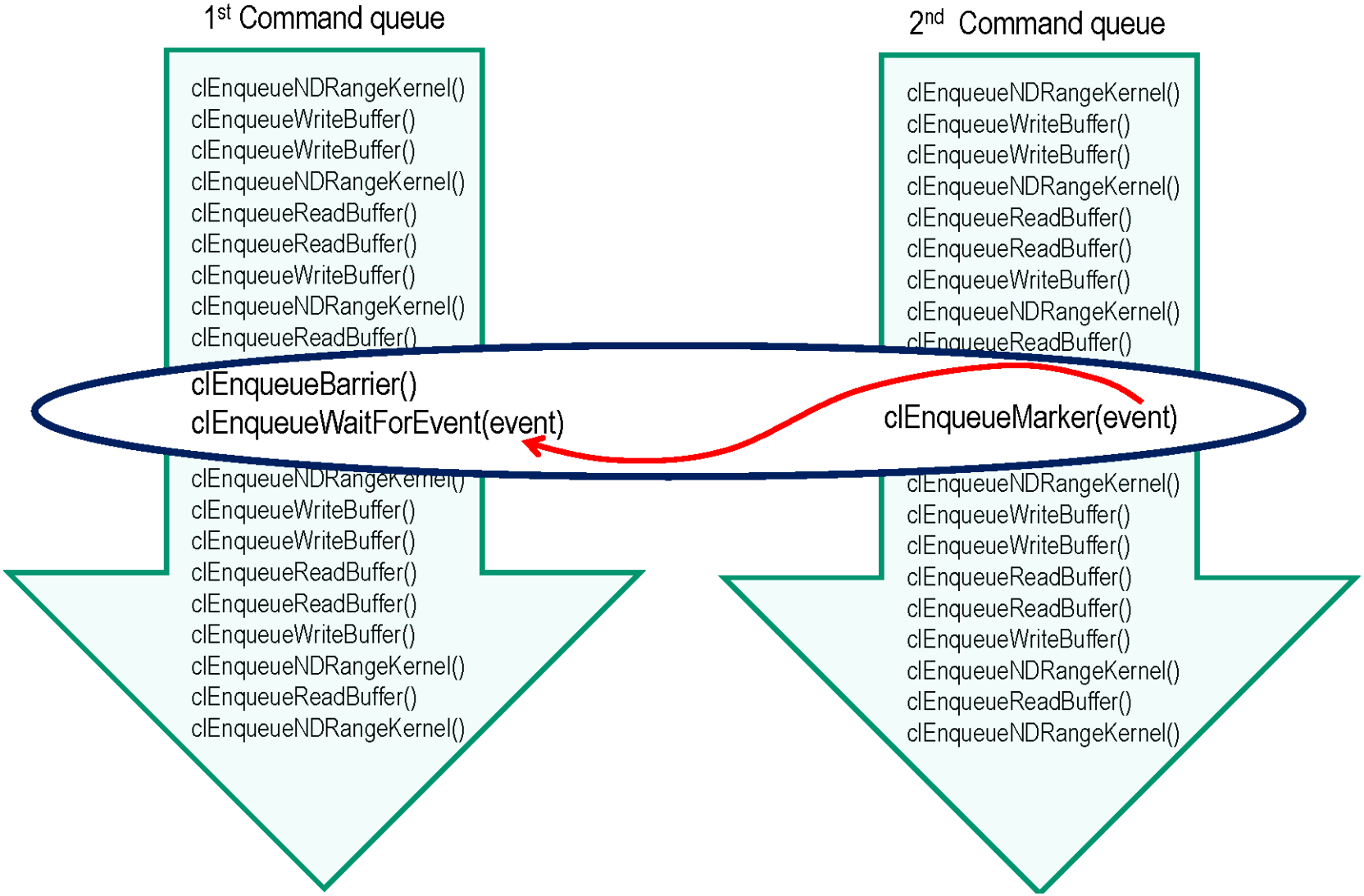


Note: Work-items may be merged by the compiler for parallel execution on SSE/AVX lanes.

Barriers between queues: clEnqueueBarrier doesn't work



Barriers between queues: this works!



Vector Addition - Host Program

Define platform and queues

```
// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0,
// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0,
                                        NULL, &cb);
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb,
    devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0],
    0, NULL);
```

Define Memory objects

```
    CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcA,
    NULL);}
memobjs[1] = clCreateBuffer(context,CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcB,
    NULL);
memobjs[2] = clCreateBuffer(context,CL_MEM_WRITE_ONLY,
                            sizeof(cl_float)*n, NULL,
    NULL);
```

Create the program

```
// create the program
program =
    &progra
```

Build the program

```
//
er                                    , NULL, NULL,
    NULL);
```

Create and setup kernel

```
/
k                                                  LL);

// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
                                  sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1],
                                  sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2],
                                   sizeof(cl_mem));
```

Execute the kernel

```
// set work-item dimensions
global_w

// execu
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1,
    NULL, global_work_size, NULL, 0, NULL, NULL);
```

Read results on the host

```
// rea
err =                                              TRUE,
    0,
```

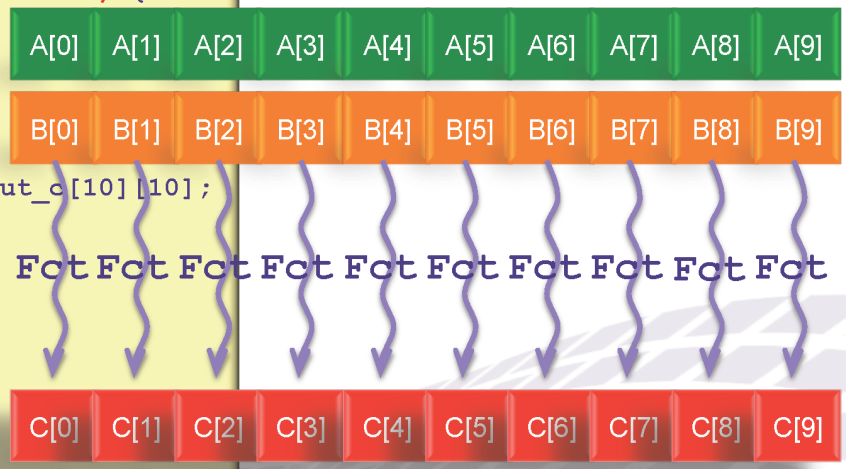
It's complicated, but most of this is "boilerplate" and not as bad as it looks.

What is Stream Computing?

- A similar computation is performed on a collection of data (*stream*)
  - There is no data dependence between the computation on different stream elements
- Stream programming is well suited to GPU *and vector-cpu!*

```
kernel void Fct(float a<>, float b<>, out float c<>) {
   c = a + b;
 }
int main(int argc, char** argv) {
   int i, j;
   float a<10, 10>, b<10, 10>, c<10, 10>;
   float input_a[10][10],input_b[10][10], input_c[10][10];
  for(i=0; i<10; i++) {
     for(j=0; j<10; j++) {
      input_a[i][j] = (float) i;
      input_b[i][j] = (float) j;
    }
  }
 streamRead(a, input_a);
 streamRead(b, input_b);
 Fct(a, b, c);
 streamWrite(c, input_c);
 ...
}
```

Brook+ example



atan2: sequential vs vector optimization

Traditional sequential

```
code = 0;
if( x < 0.0 ) code = 2;
if( y < 0.0 ) code |= 1;
if( x == 0.0 ) {
  if( code & 1 ) return( -PIO2F );
  if( y == 0.0 ) return( 0.0 );
  return( PIO2F );
}
if( y == 0.0 ) {
  if( code & 2 ) return( PIF );
  return( 0.0 );
}
switch( code ) {
 default:
 case 0:
 case 1: w = 0.0; break;
 case 2: w = PIF; break;
 case 3: w = -PIF; break;
}
return w + atanf( y/x ); // more if,div + a poly
```

Vector kernel

```
float xx = fabs(x);
float yy = fabs(y);
float tmp =0.0f;
if (yy>xx) {   // pi/4
  tmp = yy; yy=xx; xx=tmp;
}
float t=yy/xx;
float z=t;
if( t > 0.4142135623730950f)   // pi/8
    z = (t-1.0f)/(t+1.0f); // always computed
// 2 divisions will cost more than the poly
float ret = poly(z);

if (y==0) ret=0.0f;
if( t > 0.4142135623730950f ) ret += PIO4F;
if (tmp!=0) ret = PIO2F - ret;
if (x<0) ret = PIF - ret;
if (y<0) ret = -ret;

return ret;
```

sin-cos: sequential vs vector optimization

Traditional sequential

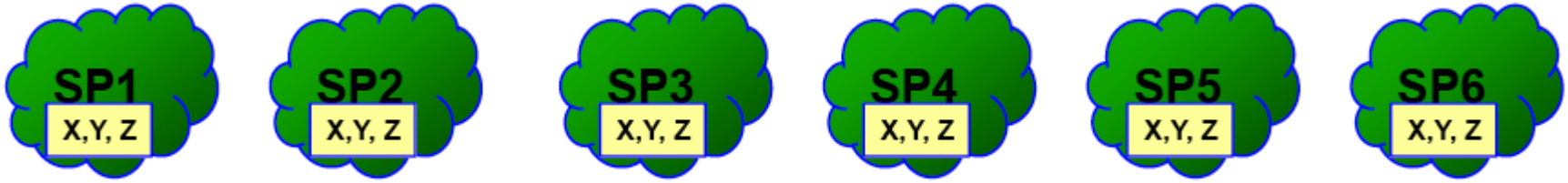
```
int sign = 1;
if( x < 0 ) x = -x;

if( x > T24M1 ) return(0.0);
int j = FOPI * x;
float y = j;
if( j & 1 )  { j += 1;  y += 1.0; }
j &= 7;
if( j > 3) { j -=4; sign = -sign;}
if( j > 1 ) sign = -sign;

if( x > lossth ) x = x - y * PIO4F;
else x = ((x - y * DP1) - y * DP2) - y * DP3;
if( (j==1) || (j==2) ) cos = poly1(x);
else cos = poly2(x);
if(sign < 0) cos = -cos;
return cos;
```

Vector kernel

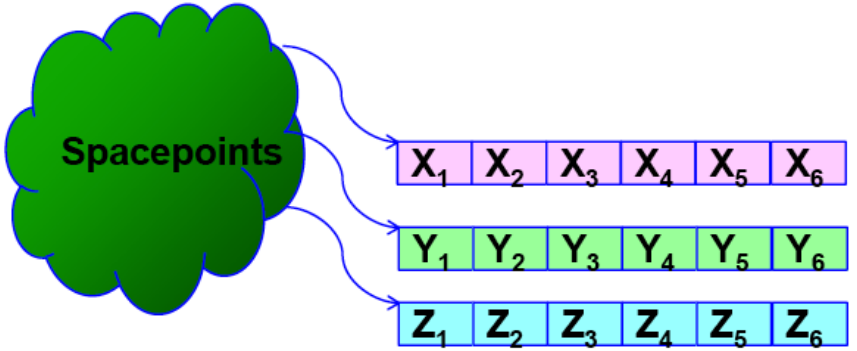
```
float x = fabs(xx);
// x =  (x > T24M1) ?   T24M1 : x;
int j = FOPI * x;
j = (j+1) & (~1);
float y = j;
float signS = (j&4);
j-=2;
float signC = (j&4);
float poly = (j&2);

x = ((x - y * DP1) - y * DP2) - y * DP3;
cos = poly1(x);
sin = poly2(x);
if( poly!=0 ) { swap(cos,sin);}
if(signC == 0) cos = -cos;
if(signS == 0) sin = -sin;
if (xx<0)  sin = -sin;
```

Data Organization AoS vs SoA

- Traditional Object organization is an Array of Structure
  - » Abstraction often used to hide implementation details at object level



- Difficult to fit stream computing
- Better to use a Structure of Arrays



- OO can wrap SoA as the AoS
  - » Move abstraction higher
  - » Expose data layout to the compiler
- Explicit copy in many cases more efficient
  - » (notebooks vs whiteboard)

Summary

- Next generations of generic computers will contain several multicore vector cpus connected with manycore accelerators
- Efficient software will require a design that highlights parallelism
  - » Novel problem decomposition
  - » High granularity task (allow global optimization of DAG)
  - » Explicit memory model ("notebooks" vs "whiteboard")
  - » SoA instead of AoS (ease stream computing)
  - » Long stream-kernels (maximize (G)flops over (G)B/s)
- The Event Processing Framework will have to enable such an approach
  - » Task scheduling
  - » Memory Model & Data transformation
  - » Library of optimized algorithms

https://twiki.cern.ch/twiki/bin/view/LCG/MultiCoreRD