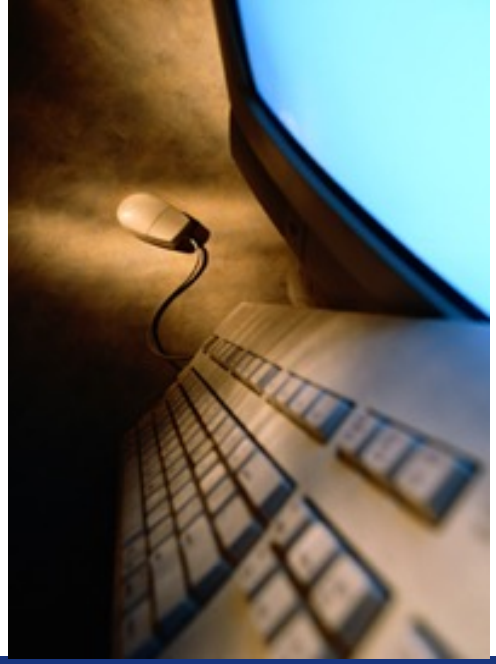
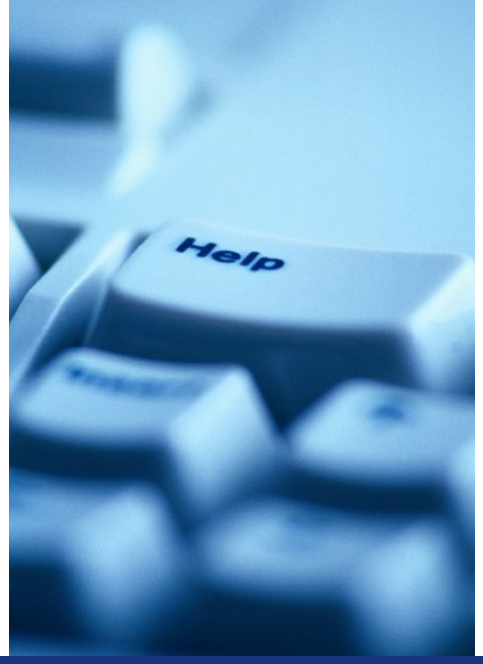
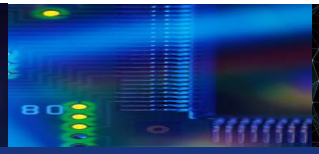



并行与分布式计算

Parallel & Distributed Computing

陈鹏飞
数据科学与计算机学院
2021-09-10



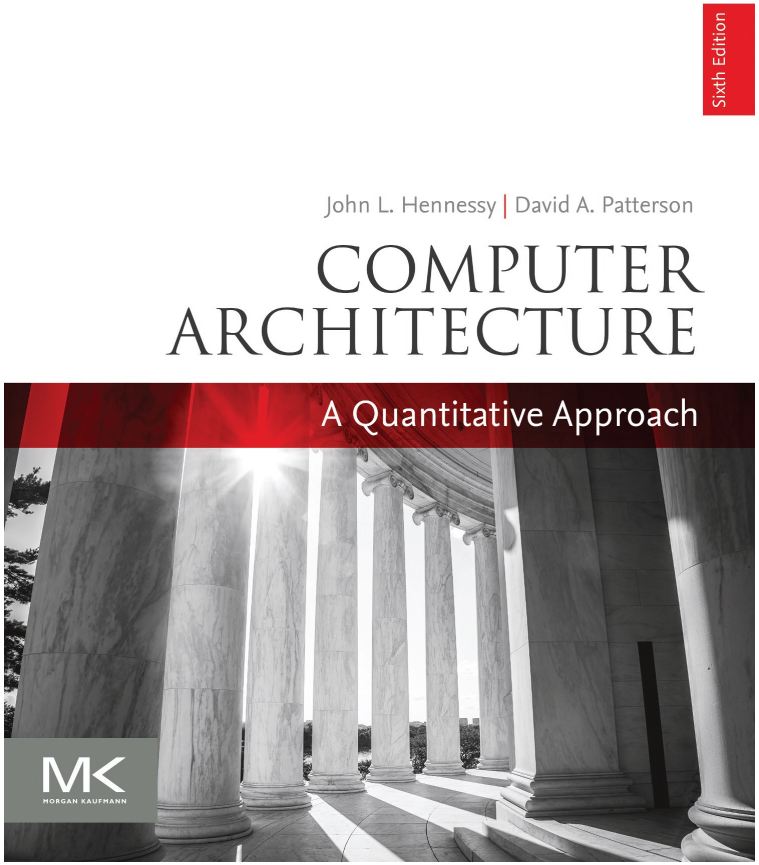
Second lecture — Parallel Architecture

Pengfei Chen

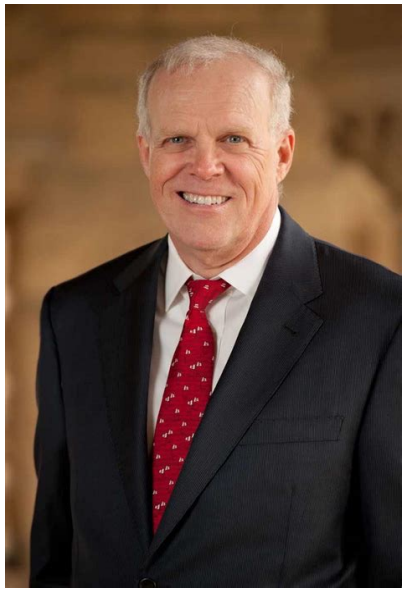
School of Data and Computer Science

Sep 10, 2021

Architecture



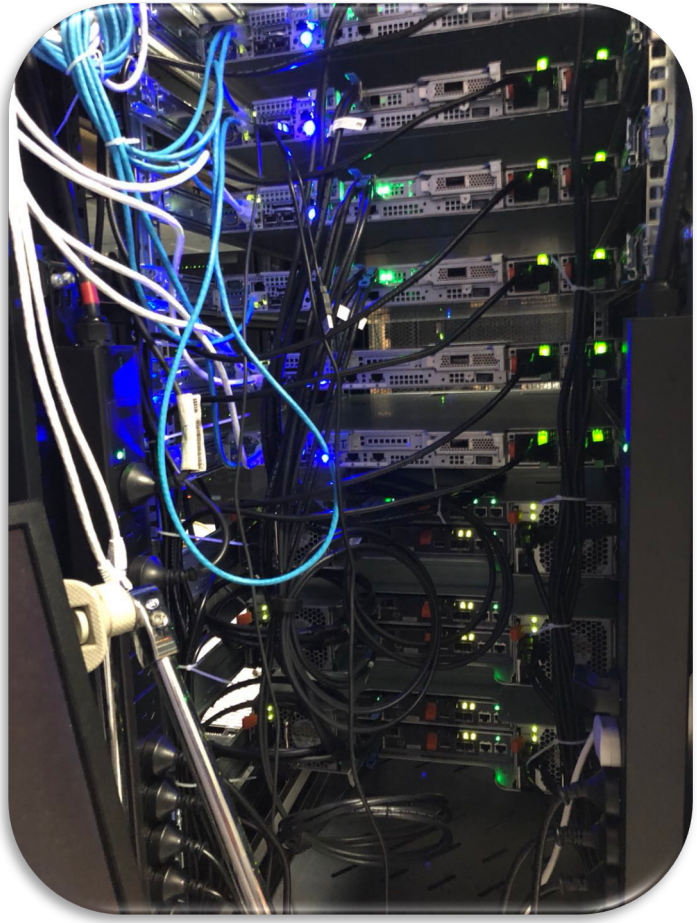
David A. Patterson



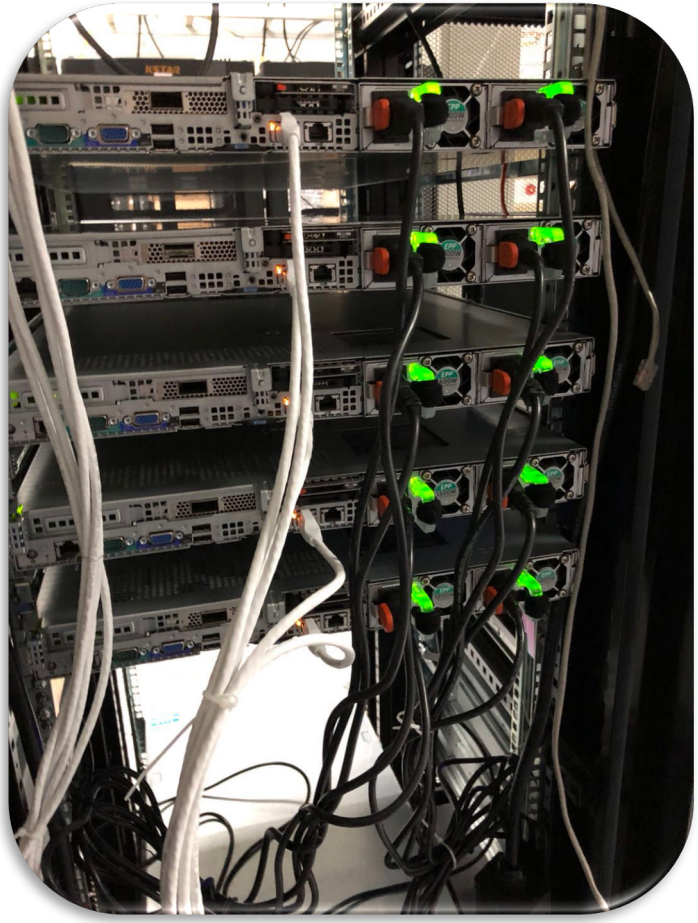
John L. Hennessy

Along with Hennessy, Patterson won the 2017 Turing Award for their work in developing RISC.

Architecture



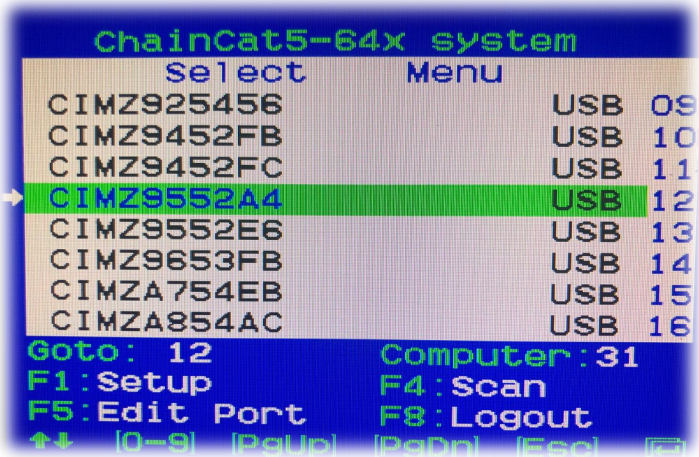
CPU计算集群



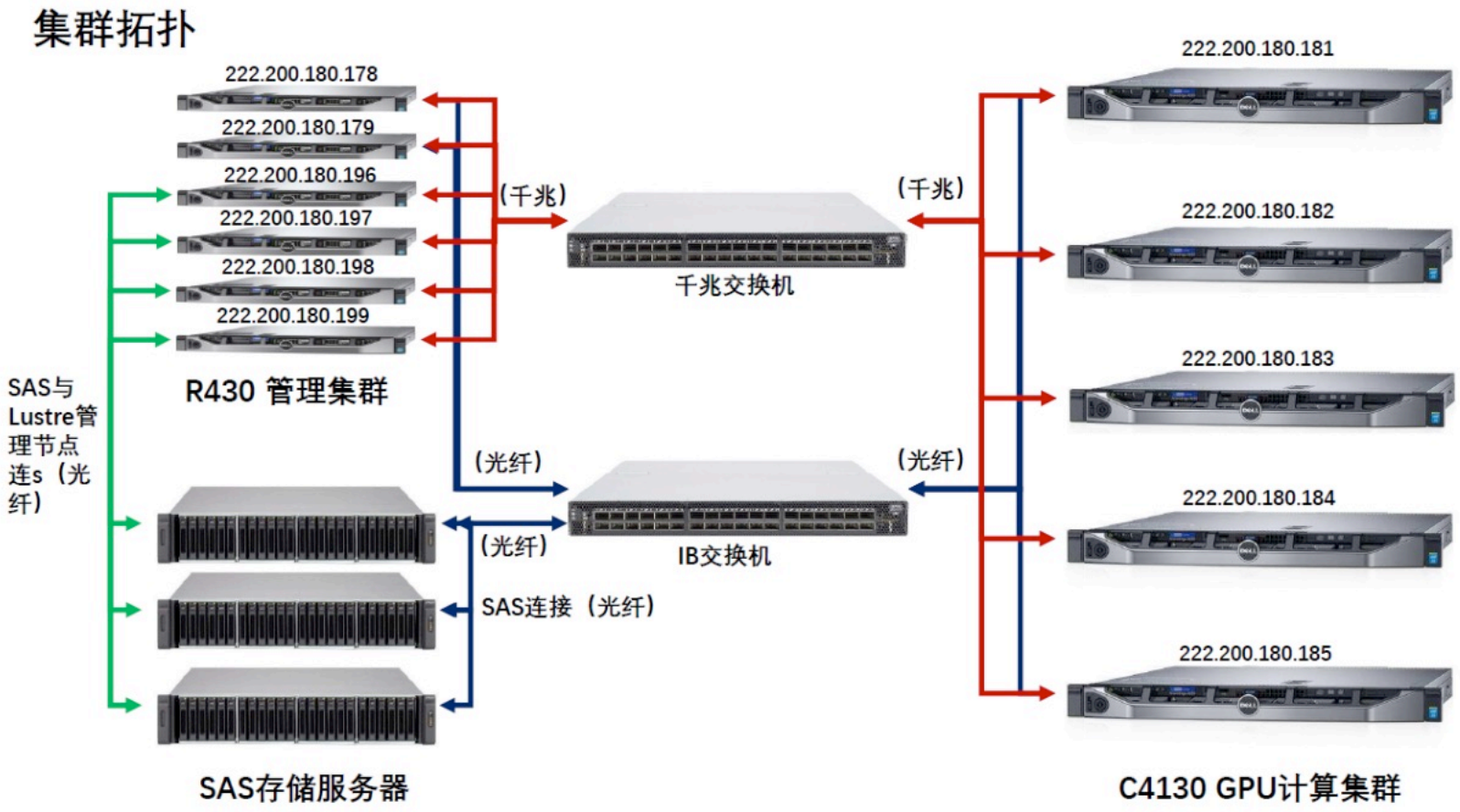
GPU计算集群

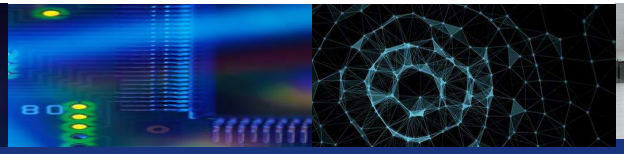
DeepBrain—SYSU Cluster

50个节点，
1.5TB内存，
200TB存储



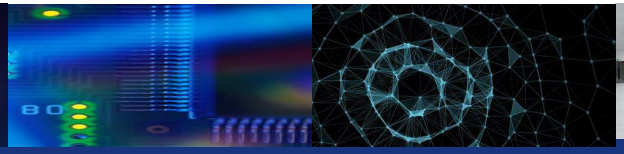
Architecture





Outline

1. Flynn's Taxonomy
2. Uniprocessor Parallelism
3. Multi-core Chips
4. Parallel Computer Architecture



1 Flynn's Taxonomy

Architectural Trends (Recall)

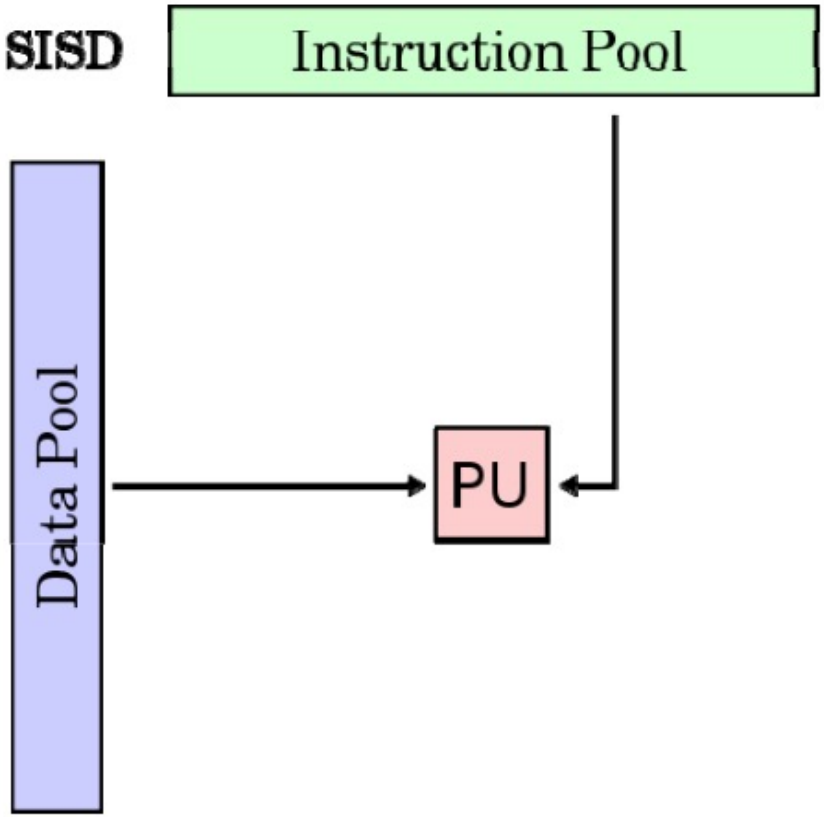
- Greatest trend in VLSI is an increase in the exploited parallelism
  - Up to 1985: bit-level parallelism: 4-bit -> 8-bit -> 16-bit
    - slows after 32 bits
    - adoption of 64-bit now under way
  - Mid 80s to mid 90s: instruction-level parallelism
    - pipelining and simple instruction sets (RISC)
    - on-chip caches and functional units => superscalar execution
    - greater sophistication（复杂性）: out of order execution（乱序执行）
  - Nowadays:
    - hyper-threading
    - multi-core

Flynn's taxonomy

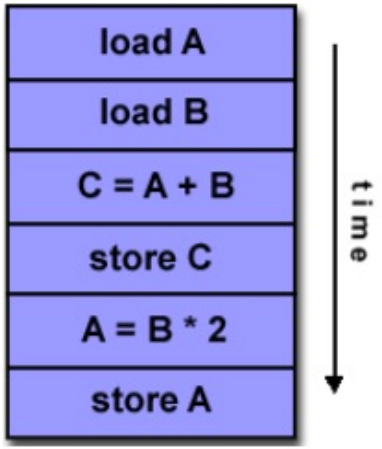
- A classification of computer architectures based on the number of streams of instructions and data

		Instruction stream	
		single	**multiple**
Data stream	**single**	SISD Single Instruction, Single Data	MISD Multiple Instructions, Single Data
	multiple	SIMD Single Instruction, Multiple Data	MIMD Multiple Instructions, Multiple Data

SISD Architecture

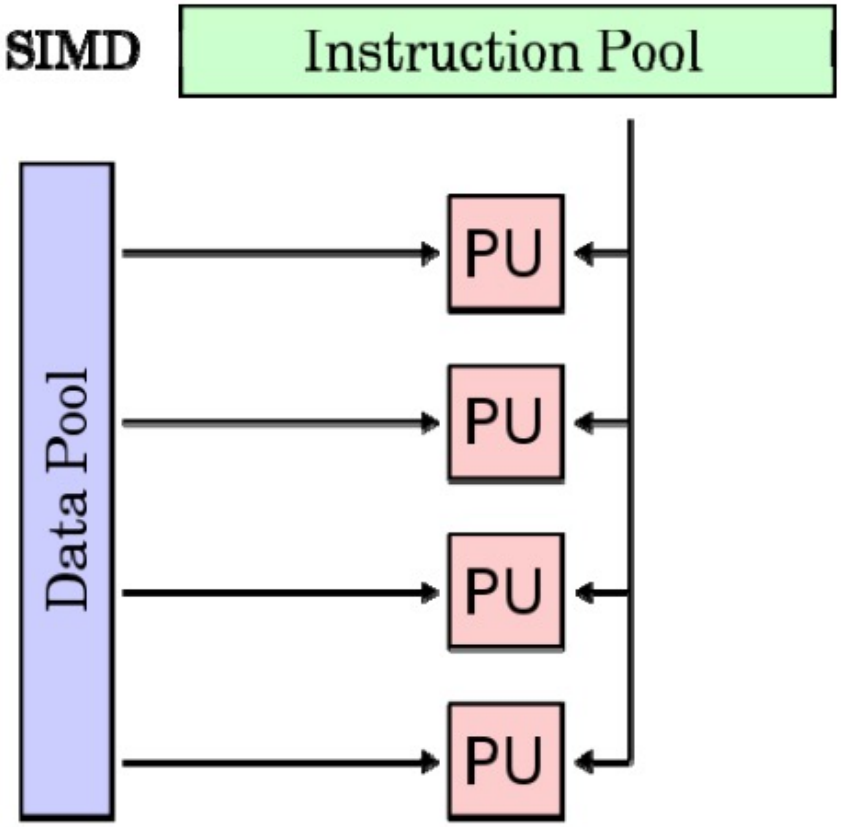


Example: single-core computers

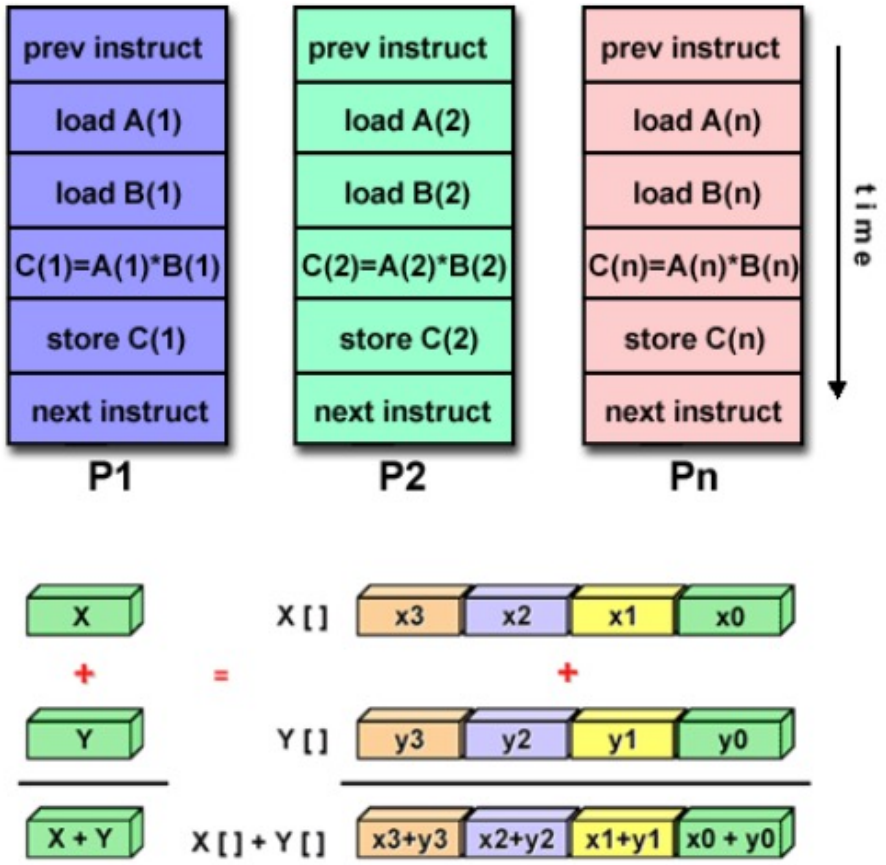


Source: http://en.wikipedia.org/wiki/Flynn's_taxonomy
Source: Blaise Barney (LLNL), Introduction to Parallel Computing

SIMD Architecture



Example: vector processors, GPUs



Source: http://en.wikipedia.org/wiki/Flynn's_taxonomy
Source: Blaise Barney (LLNL), Introduction to Parallel Computing

延伸： SPMD，**Intel SPMD Program Compiler**

Intel AVX

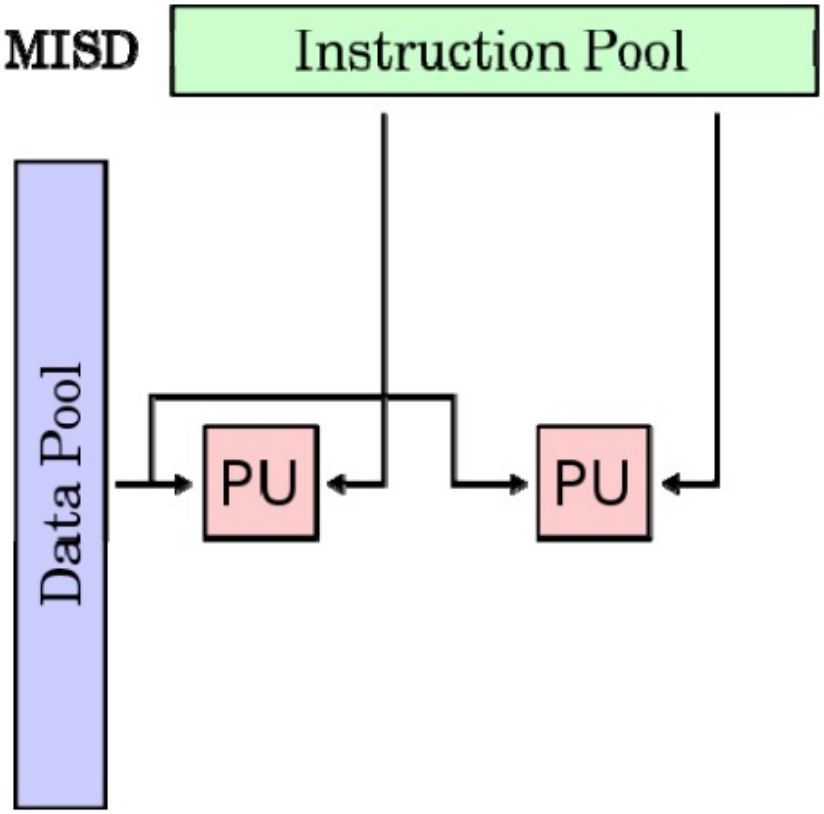
Video: https://www.intel.cn/content/www/cn/zh/architecture-and-technology/avx-512-animation.html

AVX指令集是Sandy Bridge和Larrabee架构下的新指令集。AVX是在之前的128位扩展到和256位的单指令多数据流。而Sandy Bridge的单指令多数据流演算单元扩展到256位的同时数据传输也获得了提升，所以从理论上看CPU内核浮点运算性能提升到了2倍。

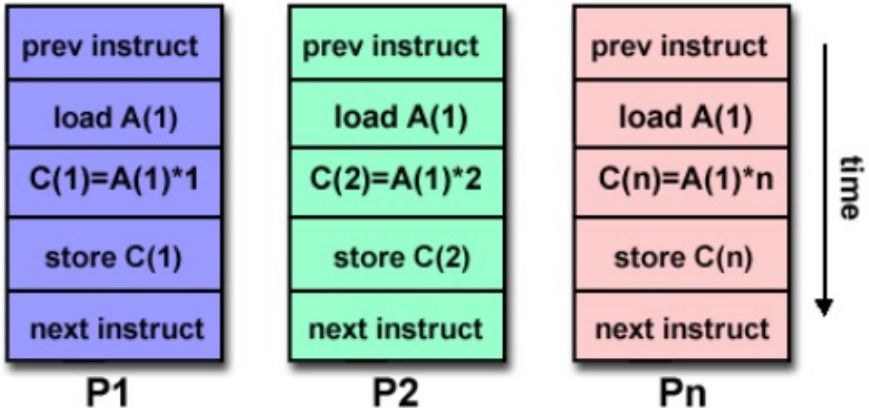
Intel AVX指令集，在单指令多数据流计算性能增强的同时也沿用了的MMX/SSE指令集。不过和MMX/SSE的不同点在于增强的AVX指令，从指令的格式上就发生了很大的变化。x86(IA-32/Intel 64)架构的基础上增加了prefix(Prefix)，所以实现了新的命令，也使更加复杂的指令得以实现，从而提升了x86 CPU的性能。

```
Gcc   -mavx -mavx2 -mfma -msse -msse2 -msse3 -Wall -O
```

MISD Architecture

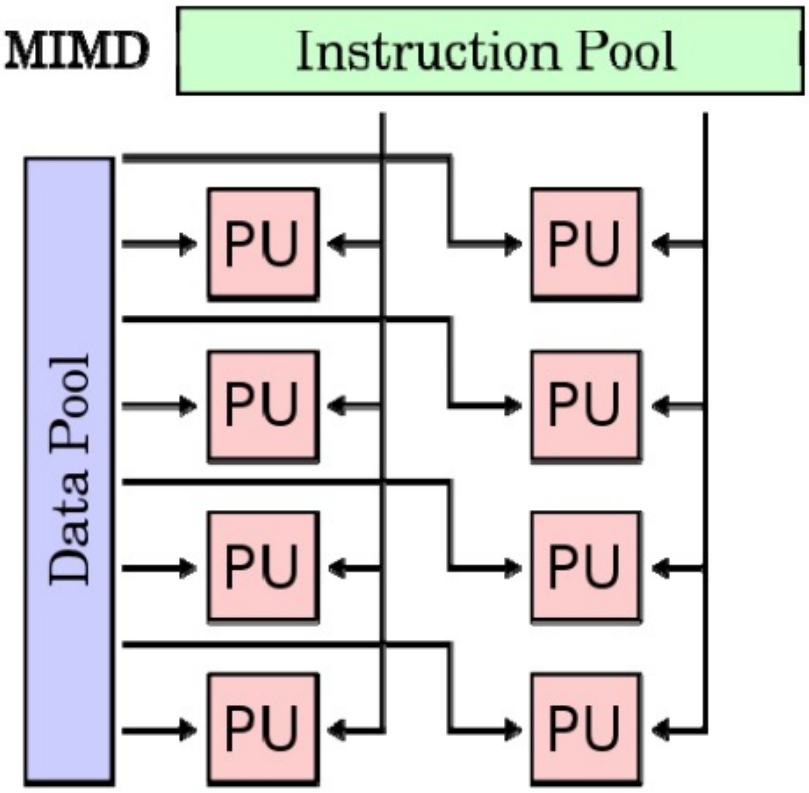


* no well-known systems fit

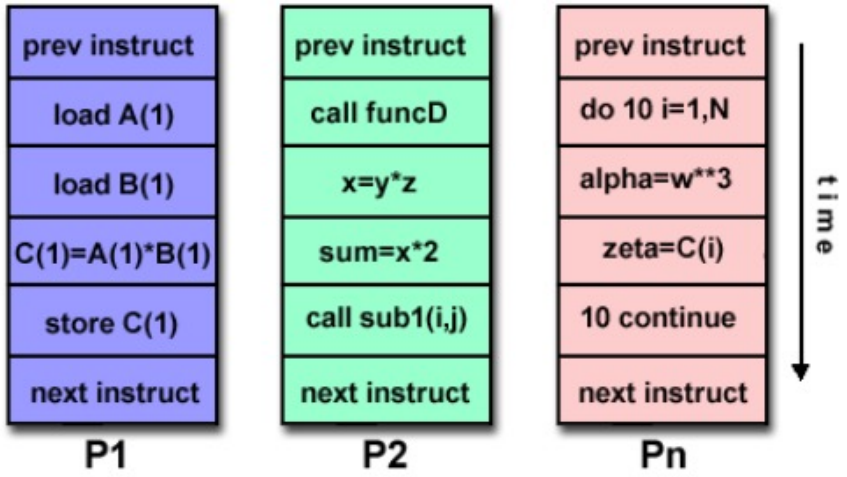


Source: http://en.wikipedia.org/wiki/Flynn's_taxonomy
Source: Blaise Barney (LLNL), Introduction to Parallel Computing

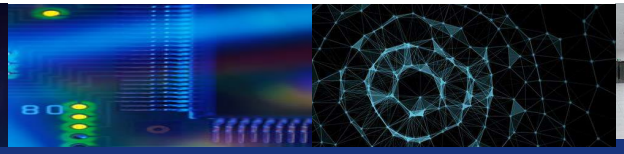
MIMD Architecture



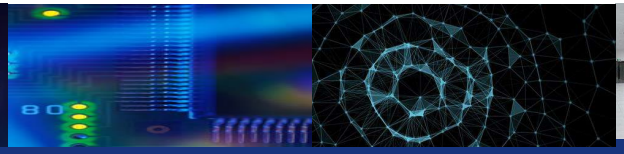
Example: modern parallel systems



Source: http://en.wikipedia.org/wiki/Flynn's_taxonomy
Source: Blaise Barney (LLNL), Introduction to Parallel Computing



2 Uniprocessor Parallelism

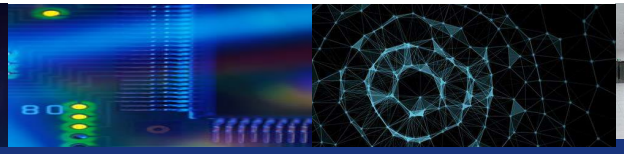


Parallelism is Everywhere

- Modern Processor Chips have ≈ 1 billion transistors
  - Clearly must get them working in parallel
  - Question: how much of this parallelism must programmer understand?
- How do uniprocessor computer architectures extract parallelism?
  - By finding parallelism within instruction stream
  - Called “Instruction Level Parallelism” (ILP)
  - The theory: hide parallelism from programmer

Parallelism is Everywhere

- **Goal of Computer Architects until about 2002:**
  - Hide Underlying Parallelism from everyone: OS, Compiler, Programmer
- **Examples of ILP techniques**
  - Pipelining: Overlapping individual parts of instructions
  - Superscalar execution: Do multiple things at same time
  - VLIW: Let compiler specify which operations can run in parallel
  - Vector Processing: Specify groups of similar (independent) operations
  - Out of Order Execution (OOO): Allow long operations to happen



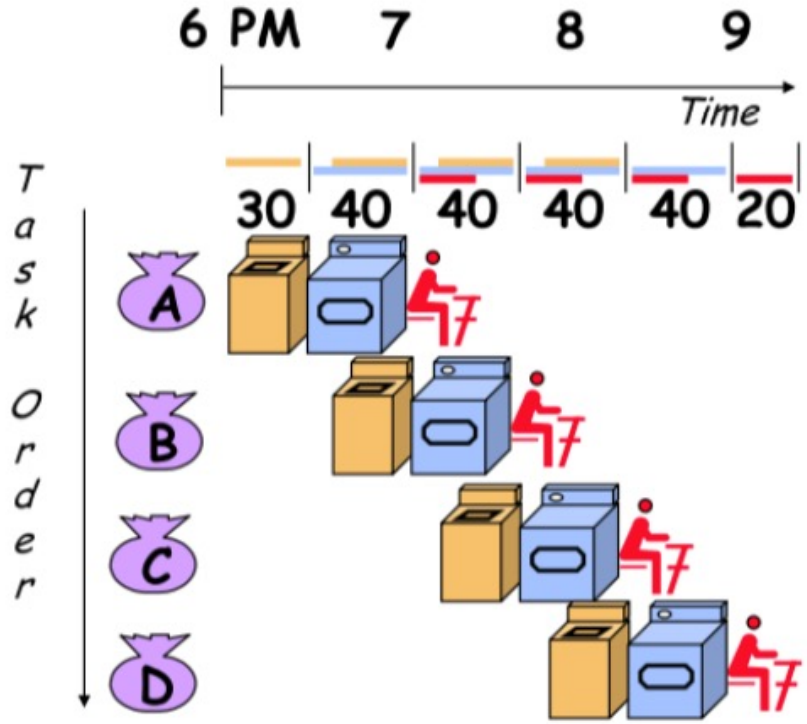
Uniprocessor Parallelism

PIPELINING, SUPERSCALAR, OUT-OF-ORDER EXECUTION

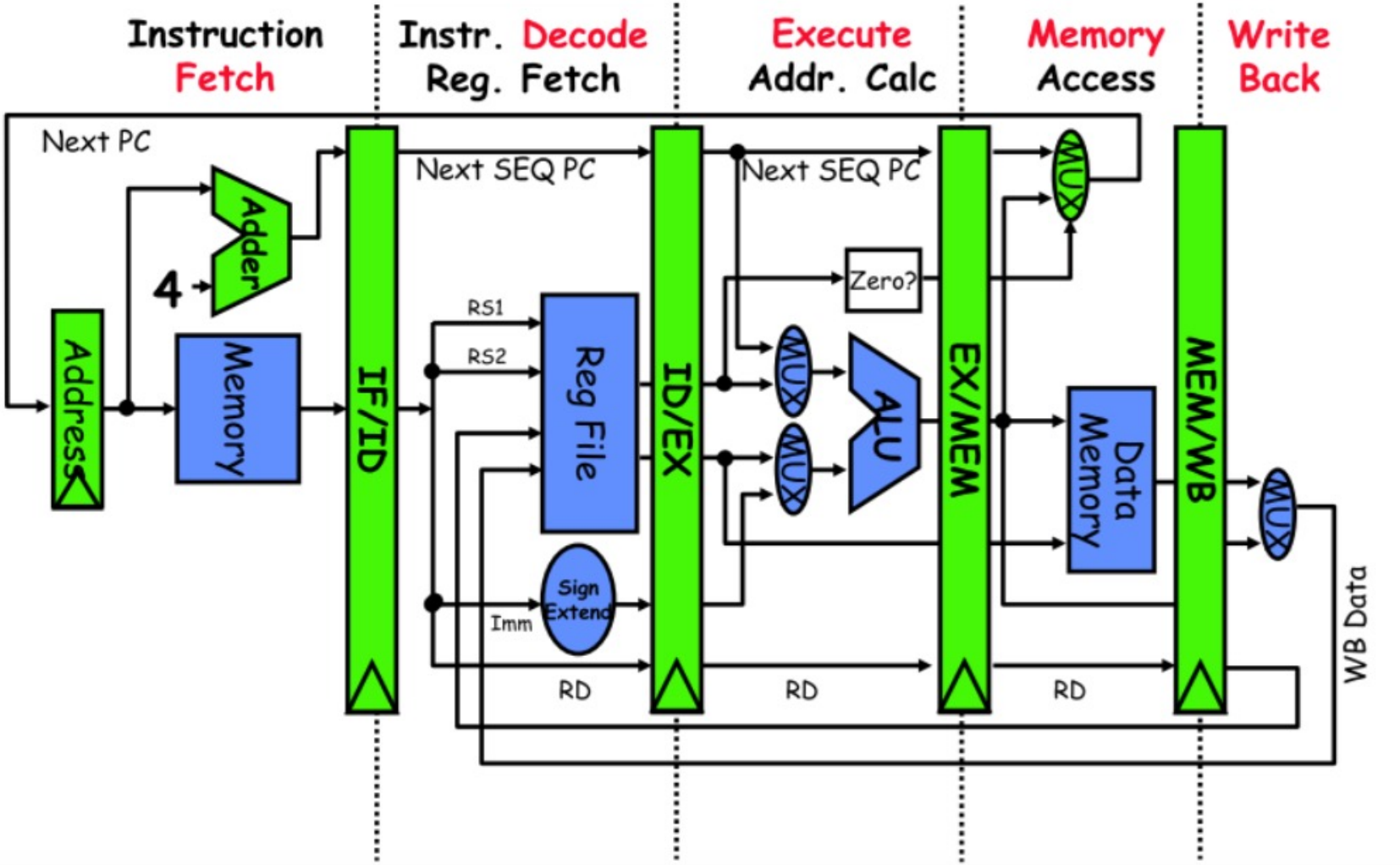
What is Pipelining?

David Patterson's Laundry example: 4 people doing laundry wash (30 min) + dry (40 min) + fold (20 min) = 90 min Latency

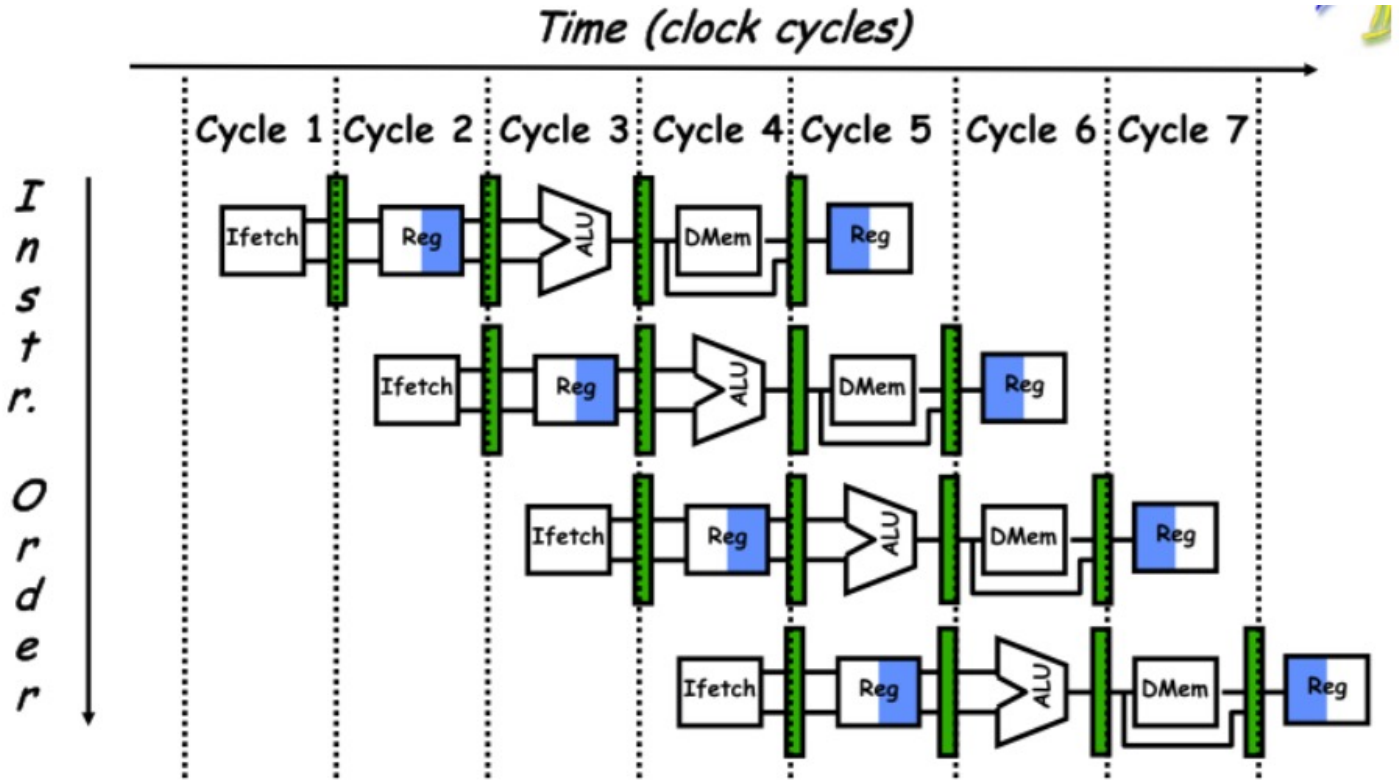
- In this example:
  - Sequential execution take 4 * 90min = 6 hours
  - Pipelined execution takes 30+4*40+20 = 3.5 hou
- Bandwidth = loads/hour
  - BW = 4/6 l/h w/o pipelining
  - BW = 4/3.5 l/h w pipelining
  - BW <= 1.5 l/h w pipelining, more total loads
- Pipelining helps bandwidth but not latency (90 min)
- Bandwidth limited by slowest pipeline stage
- Potential speedup = Number of pipe stages



5 Steps of MIPS Pipeline

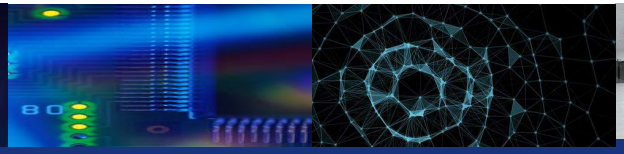


Visualizing The Pipeline



- In ideal case: CPI (cycles/instruction) = 1!
  - On average, put one instruction into pipeline, get one out
  - Superscalar: Launch more than one instruction/cycle
  - In real case, CPI < 1

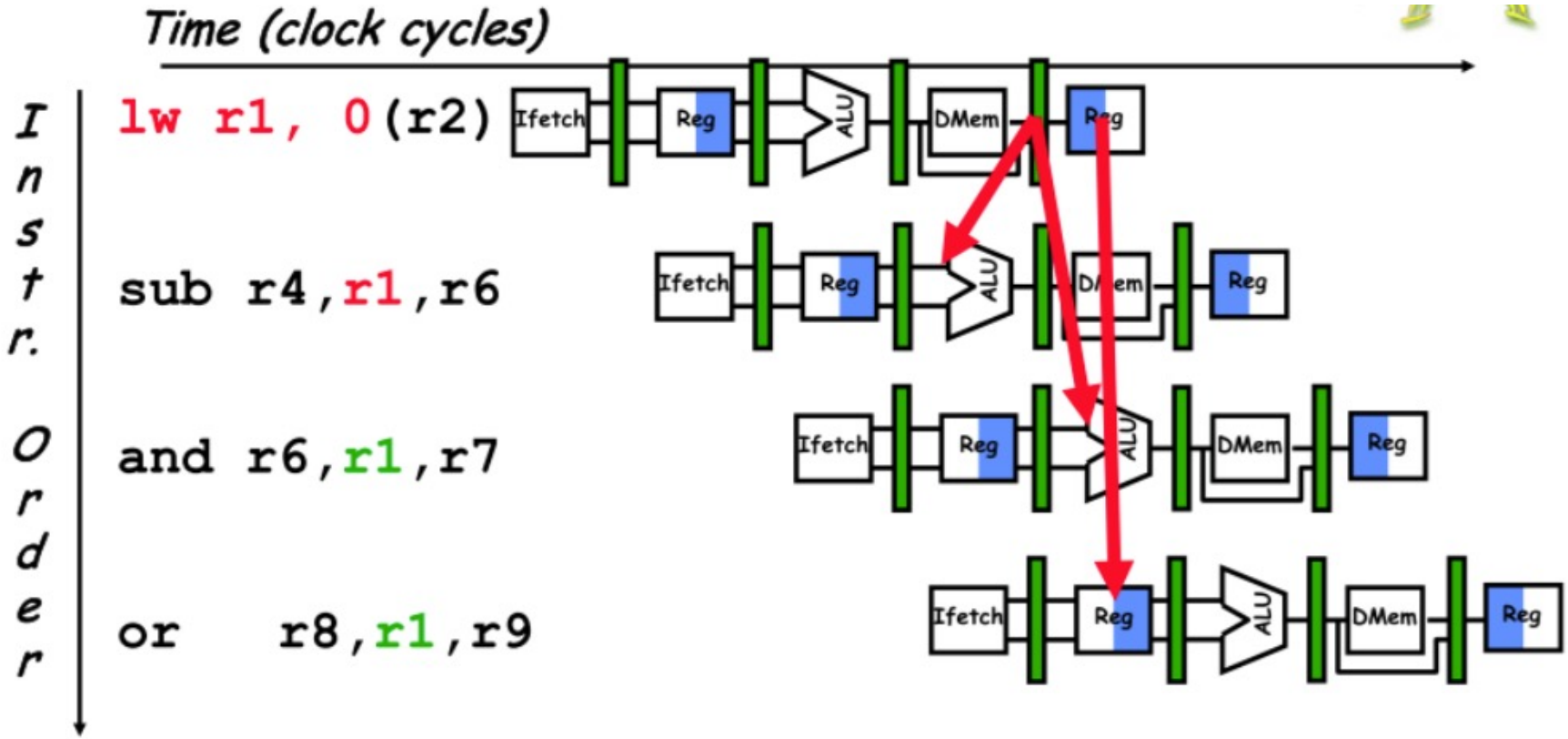
Demo



Limits to Pipelining

- Overhead prevents arbitrary division
  - Cost of **latches** (between stages) limits what can do within stage
  - Sets minimum amount of work/stage
- Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: Attempt to use the same hardware to do two different things at once
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps)
- Superscalar increases occurrence of hazards
  - More conflicting instructions/cycle

Data Hazard: Must go Back in Time?



- Data dependencies between adjacent instructions
  - Must wait ("stall") for result to be done (No "back in time" exists!)
  - Net result is that CPI > 1
- Superscalar increases frequency of hazards

Out-of-Order (OOO) Execution

- Key idea: Allow instructions behind stall to proceed

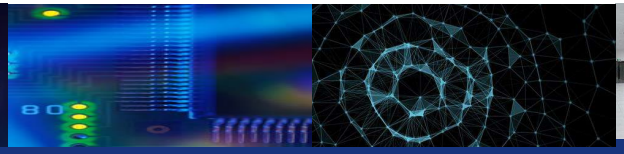
DIVD F0,F2,F4
ADDD F10,F0,F8
SUBD F12,F8,F14

- Out-of-order execution → out-of-order completion
- Dynamic Scheduling Issues from OOO scheduling
  - Must match up results with consumers of instructions
  - Precise Interrupts

Instruction	Clock Cycle Number																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
LD F6,34(R2)	IF	ID	EX	MEM	WB												
LD F2,45(R3)		IF	ID	EX	MEM	WB											
MULTD F0,F2,F4			IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	MEM	WB
SUBD F8,F6,F2				IF	ID	A1	A2	MEM	WB								
DIVD F10,F0,F6					IF	ID	stall	stall	stall	stall	stall	stall	stall	stall	stall	D1	D2
ADDD F6,F8,F2						IF	ID	A1	A2	MEM	WB						

RAW

WAR



Modern ILP

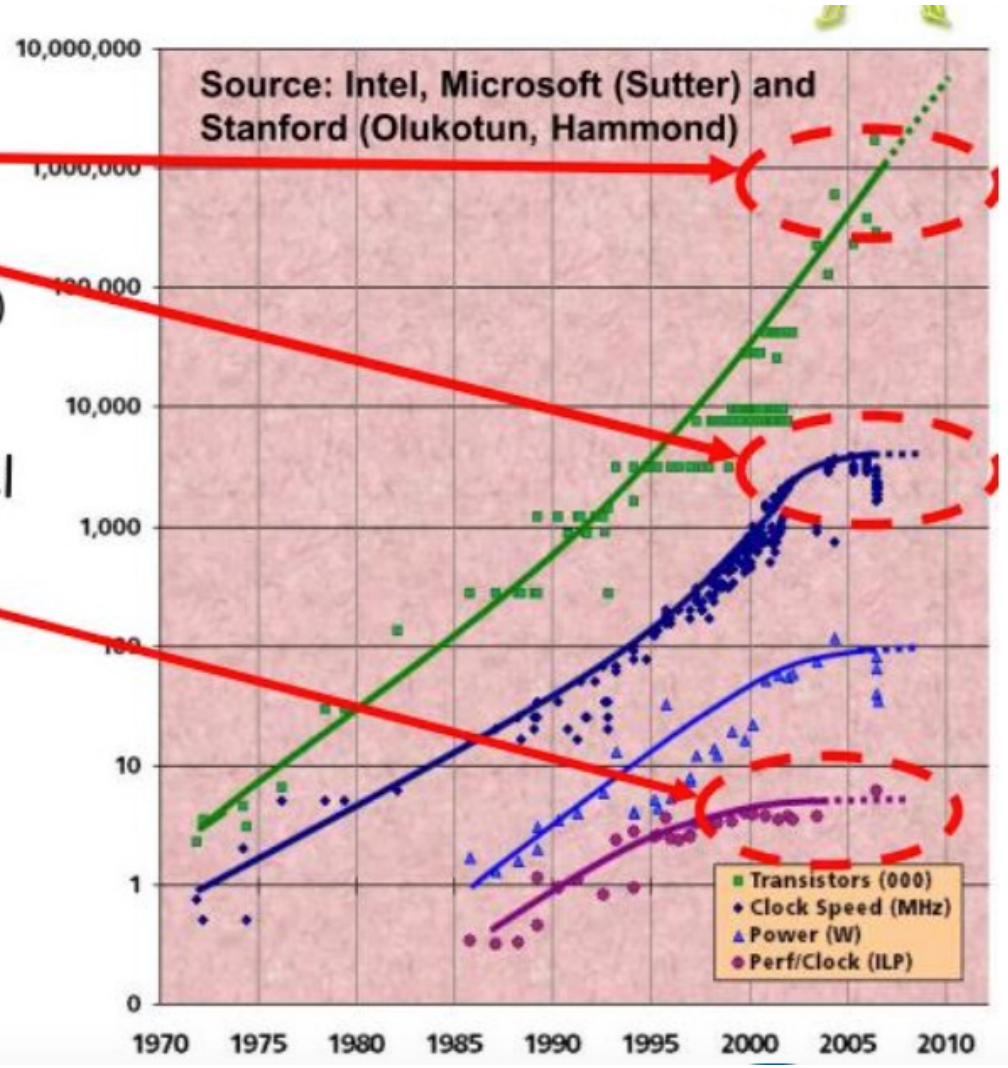
- Dynamically scheduled, out-of-order execution
  - Current microprocessors fetch 6-8 instructions per cycle
  - Pipelines are 10s of cycles deep → many overlapped instructions in execution at once, although work often discarded
- What happens
  - Grab a bunch of instructions, determine all their dependences, eliminate dep's wherever possible, throw them all into the execution unit, let each one move forward as its dependences are resolved
  - Appears as if executed sequentially

Modern ILP(Cont.)

- Dealing with Hazards: May need to *guess!*
  - Called “Speculative Execution（推测执行）”
  - Speculate on Branch results, Dependencies, even Values!
    - If correct, don’t need to stall for result → yields performance
    - **If not correct, waste time *and power* *(How do we observer?)***
  - Must be able to UNDO a result if guess is wrong
  - **Problem**: accuracy of guesses decreases with the number of simultaneous instructions in pipeline
- Huge complexity
  - Complexity of many components scales as n^2 (issue width)
  - Power consumption big problem

Limiting Forces: Clock Speed and ILP

- Chip density is continuing increase ~2x every 2 years
- Clock speed is not
  - # processors/chip (cores) may double instead
- There is little or no more Instruction Level Parallelism (ILP) to be found
  - Can no longer allow programmer to think in terms of a serial programming model
- Conclusion:
  Parallelism must be exposed to software!



Uniprocessor Parallelism

VECTOR PROCESSING/SIMD

Vector Code Example

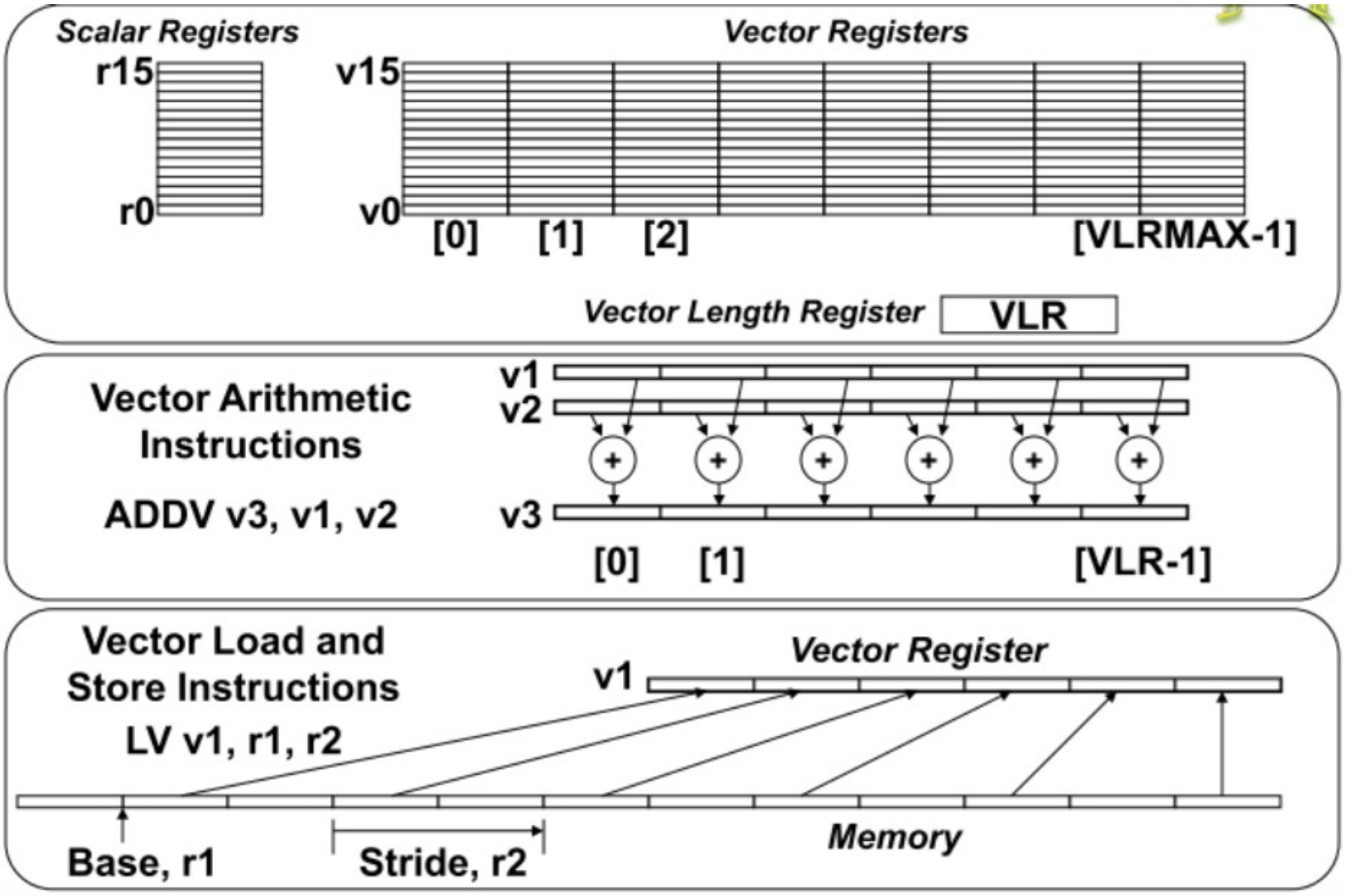
```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];
```

```
# Scalar Code
  LI R4, 64
loop:
  L.D F0, 0(R1)
  L.D F2, 0(R2)
  ADD.D F4, F2, F0
  S.D F4, 0(R3)
  DADDIU R1, 8
  DADDIU R2, 8
  DADDIU R3, 8
  DSUBIU R4, 1
  BNEZ R4, loop
```

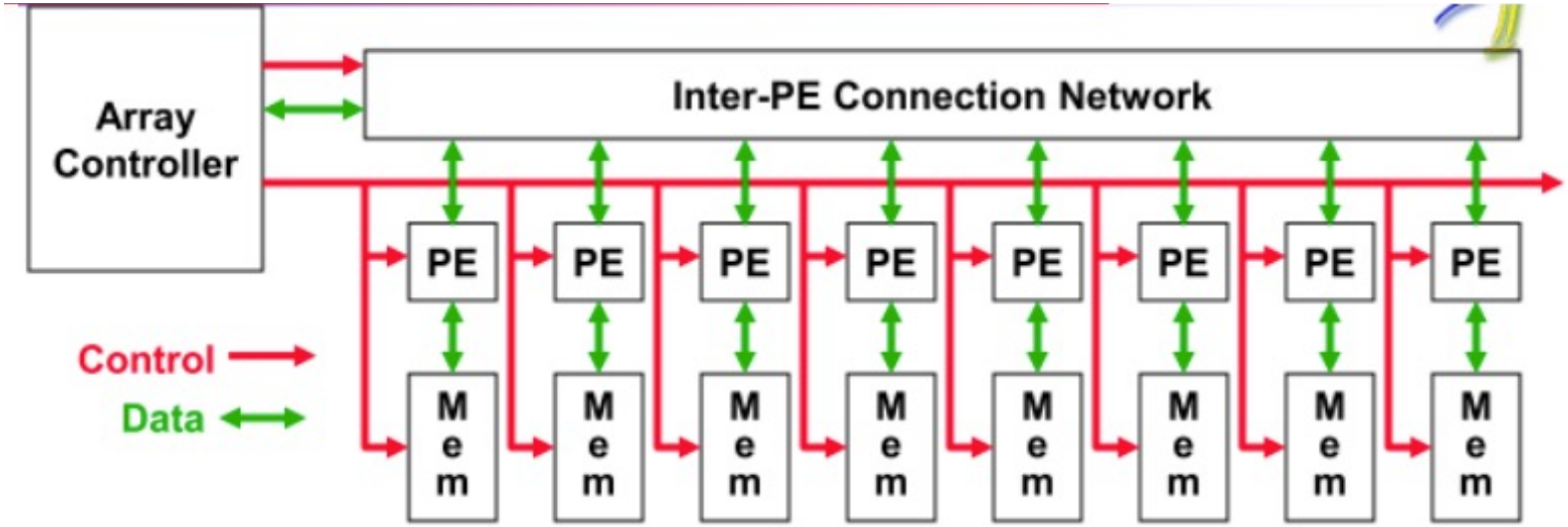
```
# Vector Code
  LI VLR, 64
  LV V1, R1
  LV V2, R2
  ADDV.D V3, V1, V2
  SV V3, R3
```

- Require programmer (or compiler) to identify parallelism
  - Hardware does not need to re-extract parallelism
- Many multimedia/HPC applications are natural consumers of vector processing

Vector Programming Model

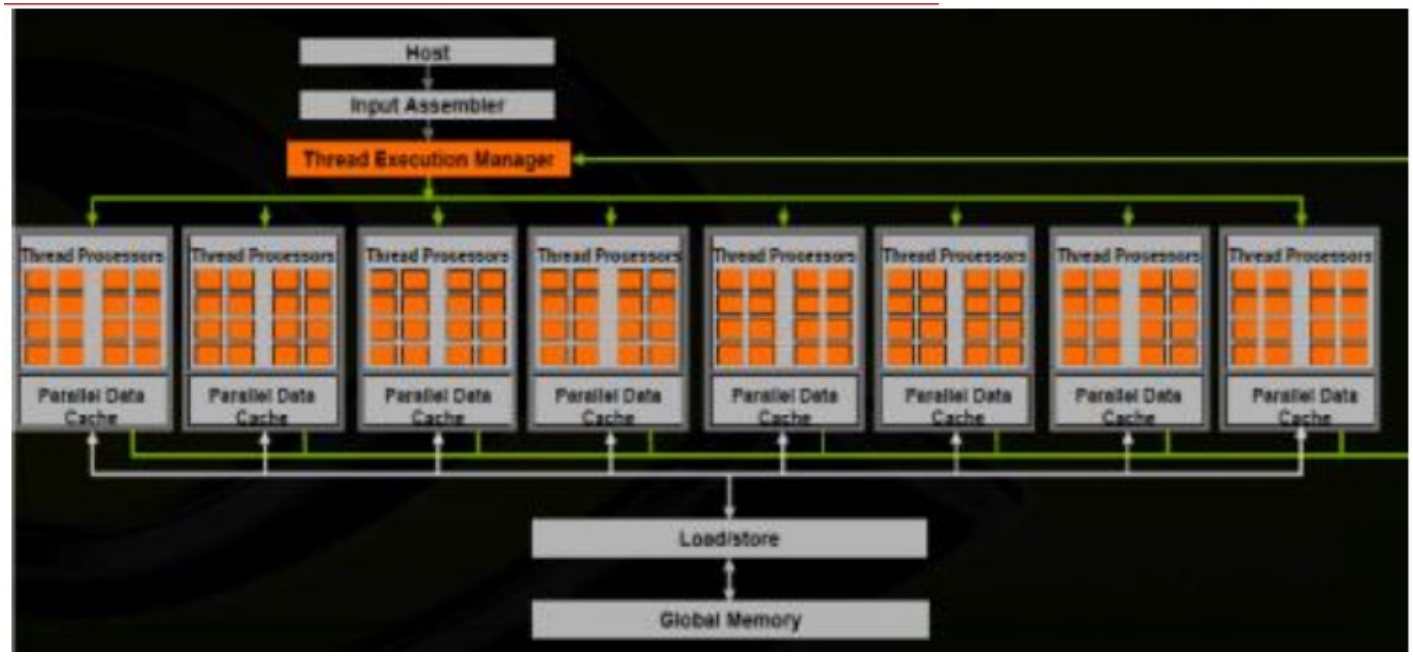


SIMD Architecture



- Single Instruction Multiple Data (SIMD)
- Central controller broadcasts instructions to multiple processing elements (PEs)
  - Only requires one controller for whole array
  - Only requires storage for one copy of program
  - All computations are fully synchronized
- Recent return to popularity
  - GPU (Graphics Processing Units) have SIMD properties
  - However, also multicore behavior, so mix of SIMD and MIMD (more later)
- Dual between Vector and SIMD execution

General-Purpose GPUs (GP-GPUs)



- In 2006, Nvidia introduced GeForce 8800 GPU supporting a new programming language: CUDA
  - Compute Unified Device Architecture
  - OpenCL is a vendor-neutral version of same ideas
- Idea: Take advantage of GPU computational performance and memory bandwidth to accelerate some kernels for general-purpose computing
- Attached processor model: Host CPU issues data-parallel kernels to GP-GPU for execution

How about vector processing in CPU?

Intel SPMD Program Compiler (ISPC)

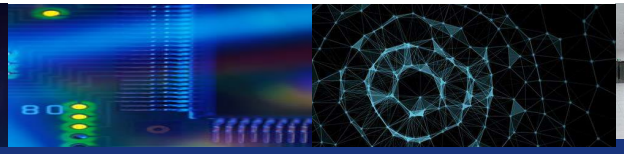
intel Look Inside.™

ISPC: The Intel® SPMD Program Compiler

For Xeon® and Xeon Phi™

Overview & Tutorial

Reference: ISPC



Intel SPMD Program Compiler (ISPC)

Motivation

Performance keeps increasing.

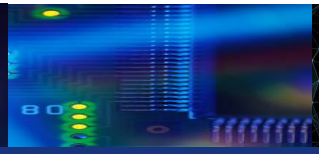
- Clock speeds governed by power considerations.

Processors instead possess more and more resources:

- Multiple cores
  - 2,4,8,...,60+
- Increasingly wider SIMD vector units
  - MMX, SSE, AVX, Xeon Phi™
  - *2,4,8,16-wide*!

It has become *necessary* to exploit parallelism inside computations in order to achieve the peak performance and efficiency available from modern machines.

However, legacy 'SIMD-unaware' languages lack semantics to allow predictably good SIMD code generation.



Intel SPMD Program Compiler (ISPC)

What is ISPC?

ISPC is a LLVM-based language and compiler that provides a SPMD programming model for Intel SIMD architectures.

Goals:

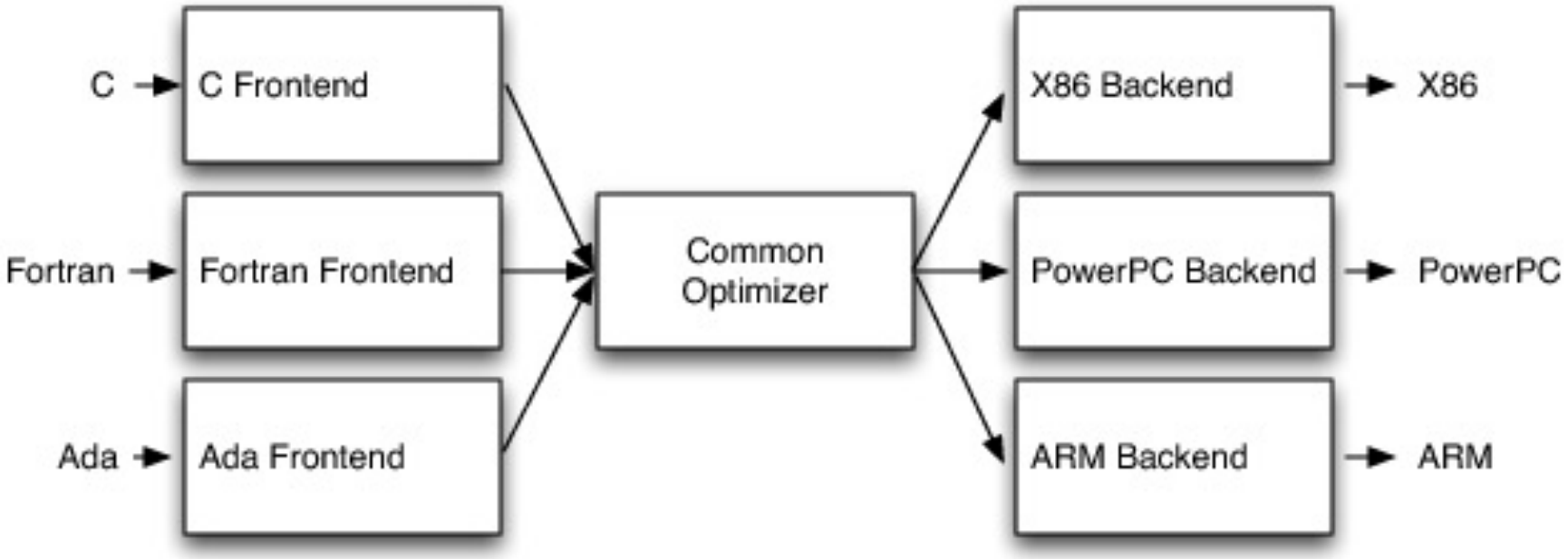
- High performance code for SSE/AVX/Xeon Phi
- Scale with additional resources: core count and SIMD vector width
- Ease of adoption and integration
- Use minimal input from the programmer to outperform loop-based autovectorizing compilers

Low Level Virtual Machine (LLVM)

- **The LLVM Compiler Infrastructure**
  - Provides reusable components for building compilers
  - Reduce the time/cost to build a new compiler
  - Build static compilers, JITs, trace-based optimizers, ...

- **The LLVM Compiler Framework**
  - End-to-end compilers using the LLVM infrastructure
  - C and C++ are robust and aggressive:
    - Java, Scheme and others are in development
  - Emit C code or native code for X86, Sparc, PowerPC

Low Level Virtual Machine (LLVM)

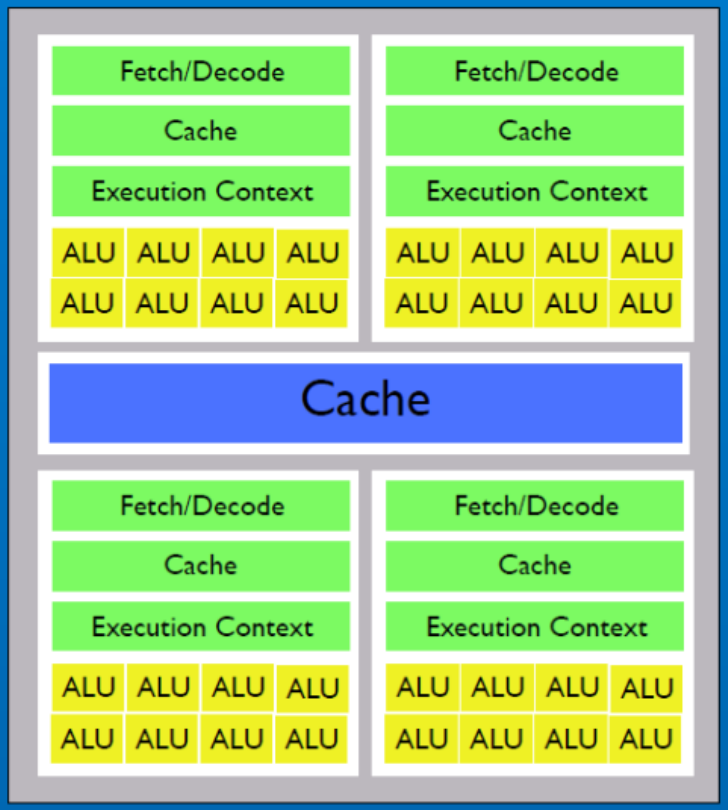


LLVM Architecture

Basic Idea

Filling the Machine (CPU and GPU)

- *Task parallelism* across cores: run different programs (if wanted) on different cores
- *Data-parallelism* across SIMD lanes in a single core: run the same program on different input values



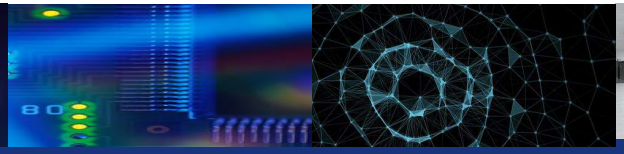
Intel SPMD Program Compiler (ISPC)

```
export void simple(uniform float vin[], uniform float vout[],
                   uniform int count) {
    foreach (index = 0 ... count) {
        // Load the appropriate input value for this program instance.
        float v = vin[index];

        // Do an arbitrary little computation, but at least make the
        // computation dependent on the value being processed
        if (v < 3.)
            v = v * v;
        else
            v = sqrt(v);

        // And write the result to the output array.
        vout[index] = v;
    }
}
```

ISPC program example

```
[====================== 100 % ======================]
[sort ispc]:    [7290.543] million cycles
[====================== 100 % ======================]
[sort ispc + tasks]:    [1987.837] million cycles
[====================== 100 % ======================]
[sort serial]:          [14207.416] million cycles
                        (1.95x speedup from ISPC, 7.15x speedup from ISPC + tasks)
```

The improvement of Sort program with ISPC

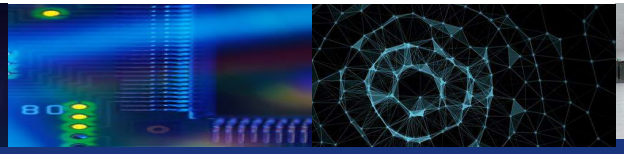


Intel SPMD Program Compiler (ISPC)

Recall: example program from last class

Compute $\sin(x)$ using Taylor expansion: $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! +$ for each element of an array of N floating-point numbers

```
void sinx(int N, int terms, float* x, float* result)
{
   for (int i=0; i<N; i++)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      int denom = 6;  // 3!
      int sign = -1;

      for (int j=1; j<=terms; j++)
      {
         value += sign * numer / denom;
         numer *= x[i] * x[i];
         denom *= (2*j+2) * (2*j+3);
         sign *= -1;
      }

      result[i] = value;
   }
}
```



Intel SPMD Program Compiler (ISPC)

sin(x) in ISPC

Compute $\sin(x)$ **using Taylor expansion:** $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \ldots$

C++ code: `main.cpp`

```
#include "sinx_ispc.h"

int N = 1024;
int terms = 5;
float* x = new float[N];
float* result = new float[N];

// initialize x here

// execute ISPC code
sinx(N, terms, x, result);
```

ISPC code: `sinx.ispc`

```
export void sinx(
    uniform int N,
    uniform int terms,
    uniform float* x,
    uniform float* result)
{
    // assume N % programCount = 0
    for (uniform int i=0; i<N; i+=programCount)
    {
        int idx = i + programIndex;
        float value = x[idx];
        float numer = x[idx] * x[idx] * x[idx];
        uniform int denom = 6;  // 3!
        uniform int sign = -1;

        for (uniform int j=1; j<=terms; j++)
        {
            value += sign * numer / denom
            numer *= x[idx] * x[idx];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }
        result[idx] = value;
    }
}
```

SPMD programming abstraction:

Call to ISPC function spawns "gang" of ISPC "program instances"

All instances run ISPC code concurrently

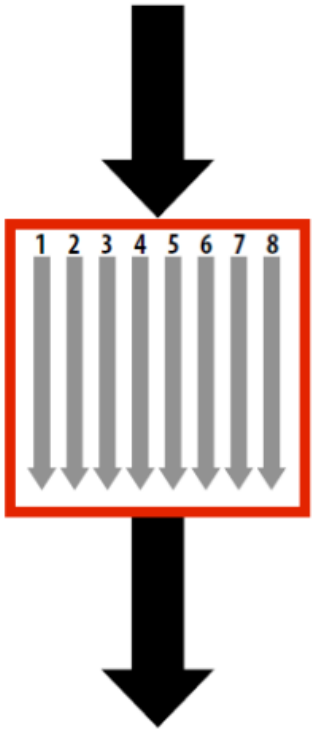
Upon return, all instances have completed

sin(x) in ISPC

Compute $\sin(x)$ using Taylor expansion: $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \ldots$

C++ code: `main.cpp`

```
#include "sinx_ispc.h"

int N = 1024;
int terms = 5;
float* x = new float[N];
float* result = new float[N];

// initialize x here

// execute ISPC code
sinx(N, terms, x, result);
```



Sequential execution (C code)

Call to `sinx()`
Begin executing `programCount` instances of `sinx()` (ISPC code)

`sinx()` returns.
Completion of ISPC program instances.
Resume sequential execution

Sequential execution
(C code)

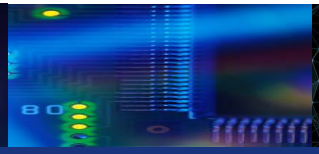
SPMD programming abstraction:

Call to ISPC function spawns "gang" of ISPC "program instances"

All instances run ISPC code concurrently

Upon return, all instances have completed

In this illustration `programCount` = 8



Uniprocessor Parallelism

MULTITHREADING: INCLUDING PTHREADS

Thread Level Parallelism (TLP)

- ILP exploits implicit parallel operations within a loop or straight-line code segment
- TLP explicitly represented by the use of multiple threads of execution that are inherently parallel
  - Threads can be on a single processor
  - Or, on multiple processors
- Concurrency vs Parallelism
  - Concurrency is when two tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean they'll ever both be running at the same instant
    - For instance, multitasking on a single-threaded machine
  - Parallelism is when tasks literally run at the same time, eg. on a multicore processor
- Goal: Use multiple instruction streams to improve
  - Throughput of computers that run many programs
  - Execution time of multi-threaded programs

Common Notions of Thread Creation

- cobegin/coend

```
cobegin
    job1(a1);
    job2(a2);
coend
```

  - Statements in block may run in parallel
  - cobegins may be nested
  - Scoped, so you cannot have a missing coend

- fork/join

```
tid1 = fork(job1, a1);
job2(a2);
join tid1;
```

  - Forked procedure runs in parallel
  - Wait at join point if it's not finished

- future

```
v = future(job1(a1));
... = ...v...;
```

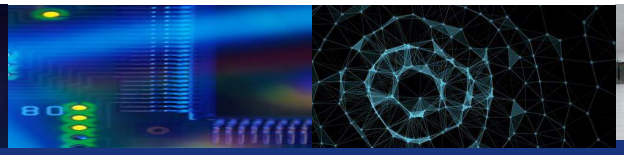
  - Future expression possibly evaluated in parallel
  - Attempt to use return value will wait

➢ Threads expressed in the code may not turn into independent Computations

- ◆ Only create threads if processors idle
- ◆ Example: Thread-stealing runtimes

Overview of POSIX Threads

- POSIX: Portable Operating System Interface for UNIX
  - Interface to Operating System utilities
- Pthreads: The POSIX threading interface
  - System calls to create and synchronize threads
  - Should be relatively uniform across UNIX-like OS platforms
  - Originally IEEE POSIX 1003.1c
- Pthreads contain support for
  - Creating parallelism
  - Synchronizing
  - No explicit support for communication, because shared memory is implicit; a pointer to shared data is passed to a thread
    - Only for HEAP! Stacks not shared



Simple Threading Example (pThreads)

E.g., compile using gcc -lpthread

```
void* SayHello(void *foo) {
  printf( "Hello, world!\n" );
  return NULL;
}

int main() {
  pthread_t threads[16];
  int tn;
  for(tn=0; tn<16; tn++) {
    pthread_create(&threads[tn], NULL, SayHello, NULL);
  }
  for(tn=0; tn<16 ; tn++) {
    pthread_join(threads[tn], NULL);
  }
  return 0;
}
```

Shared Data and Threads

- Variables declared outside of main are shared
- Objects allocated on the heap may be shared (if pointer is passed)
- Variables on the stack are private: passing pointer to these around to other threads can cause problems

- Often done by creating a large "thread data" struct, which is passed into all threads as argument

```
char *message = "Hello World!\n";

pthread_create(&thread1, NULL, print_fun,(void*) message);
```

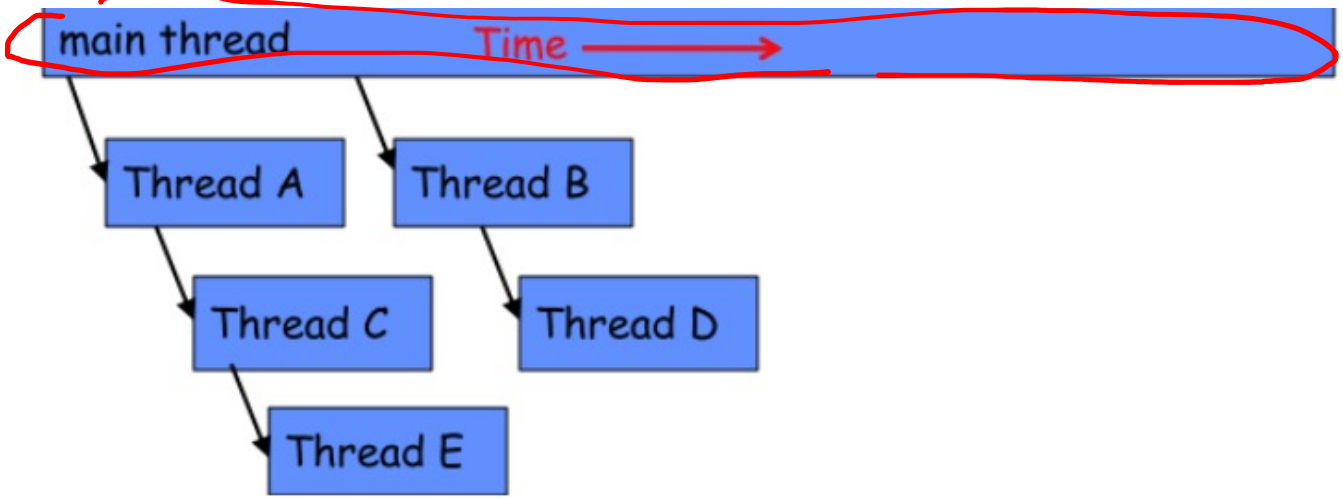
Loop Level Parallelism

- Many application have parallelism in loops

```
double stuff [n][n];
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        ... pthread_create (..., update_stuff, ..., &stuff[i][j]);
```

- But overhead of thread creation is nontrivial
  - *update_stuff* should have a significant amount of work
- Common Performance Pitfall: Too many threads
  - The cost of creating a thread is 10's of thousands of cycles on modern architectures
  - Solution: Thread blocking: use a small # of threads, often equal to the number of cores/processors or hardware threads

Thread Scheduling



- Once created, when will a given thread run?
  - It is up to the operating system or hardware, but it will run eventually, even if you have more threads than cores
  - But scheduling may be non-ideal for your application
- Programmer can provide hints or affinity in some cases
  - E.g., create exactly *P* threads and assign to *P* cores
- Can provide user-level scheduling for some systems
  - Application-specific tuning based on programming model

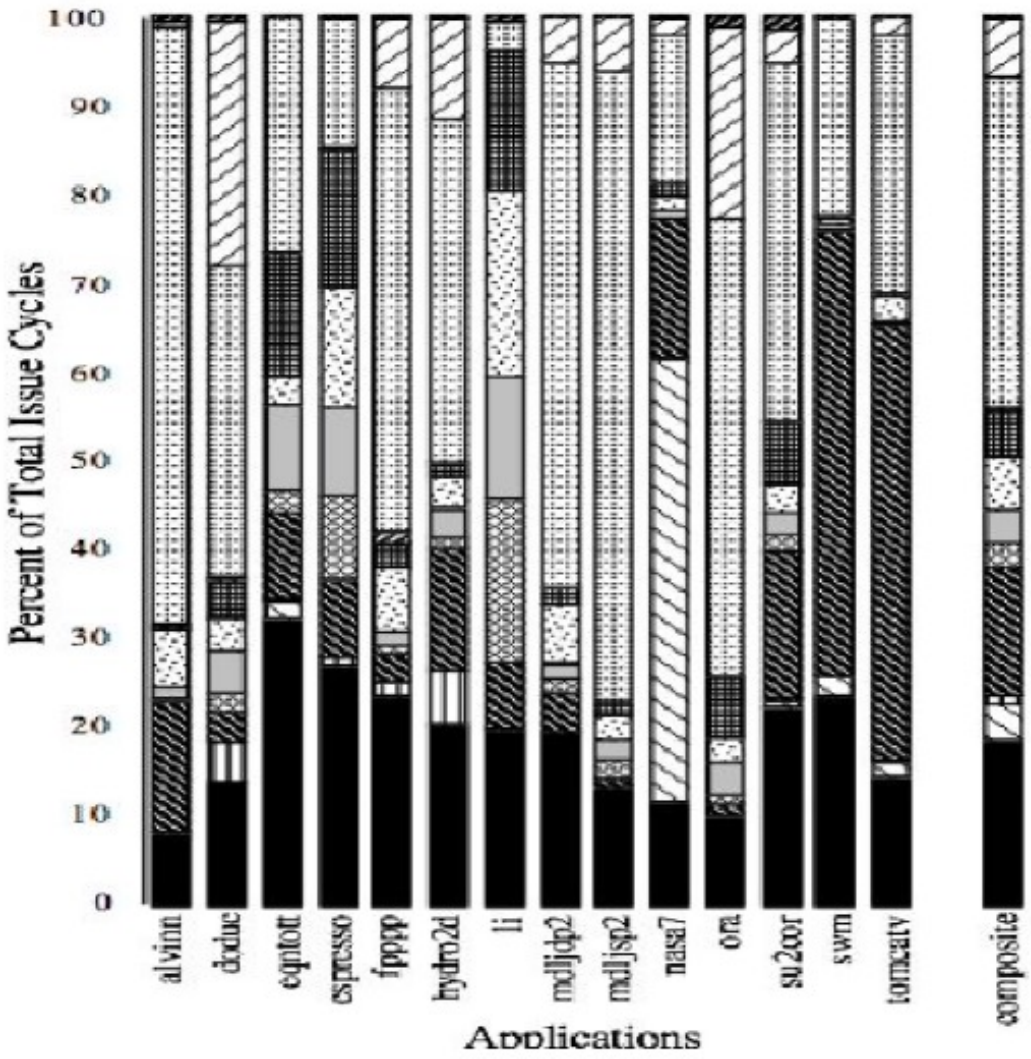
Multithreaded Execution

- Multitasking operating system
  - Gives "illusion" that multiple things happen at same time
  - Switches at a coarse-grained time (for instance: 10ms)
- Hardware Multithreading: multiple threads share processor simultaneously (with little OS help)
  - Hardware does switching
    - HW for fast thread switch in small number of cycles
    - much faster than OS switch which is 100s to 1000s of clocks
  - Processor duplicates independent state of each thread
    - e.g., a separate copy of register file, a separate PC, and for running independent programs, a separate page table
  - Memory shared through the virtual memory mechanisms, which already support multiple processes
- When to switch between threads?
  - Alternate instruction per thread (fine grain)
  - When a thread is stalled, perhaps for a cache miss, another thread can be executed (coarse grain)

What about combining ILP and TLP?

- TLP and ILP exploit two different kinds of parallel structure in a program
- Could a processor oriented at ILP benefit from exploiting TLP?
  - functional units are often idle in data path designed for ILP because of either stalls or dependences in the code
  - TLP used as a source of independent instructions that might keep the processor busy during stalls
  - TLP be used to occupy functional units that would otherwise lie idle when insufficient ILP exists
- Called "Simultaneous Multithreading"
  - Intel renamed this "Hyperthreading"

Quick Recall: Many Resources IDLE!

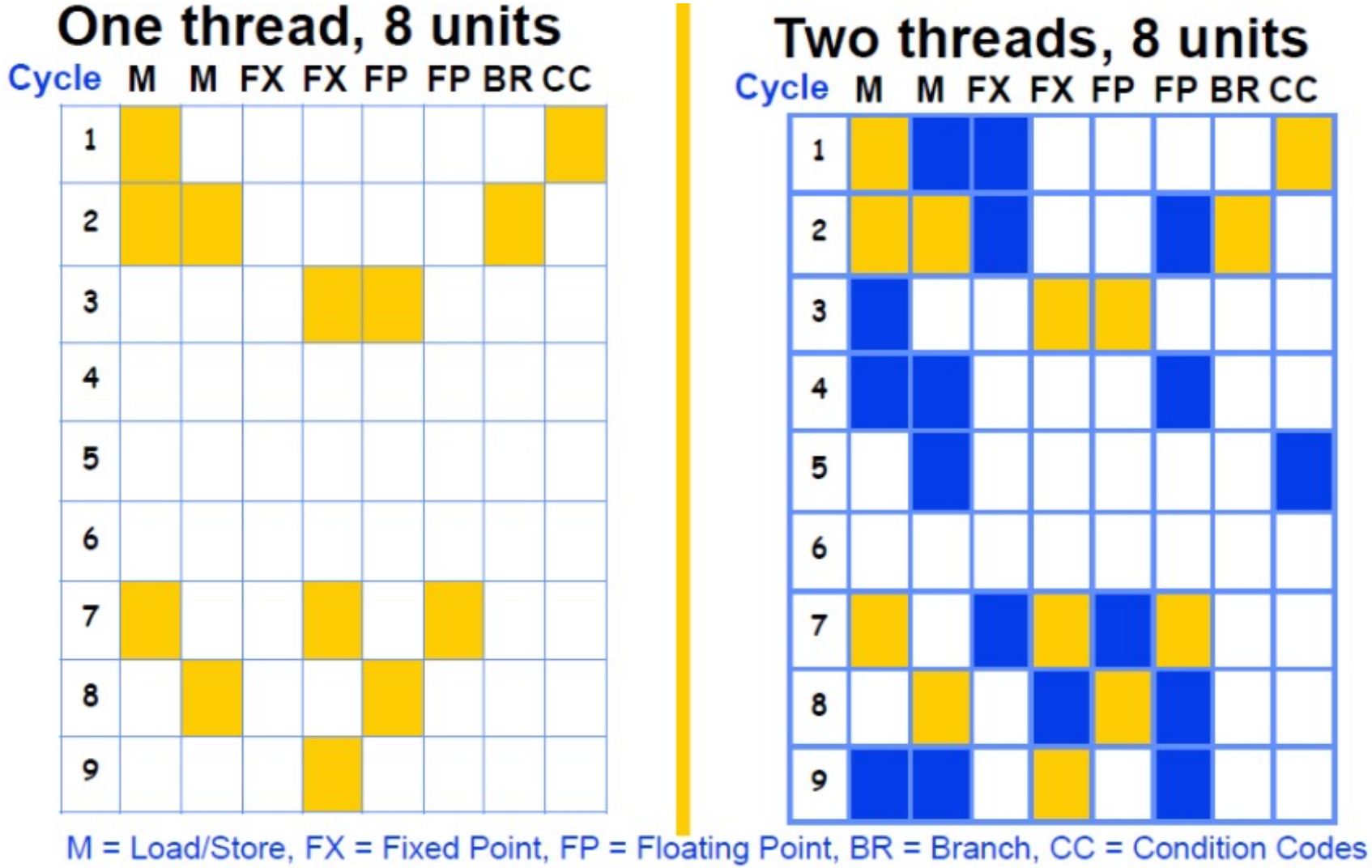


For an 8-way superscalar

- memory conflict
- long fp
- short fp
- long integer
- short integer
- load delays
- control hazards
- branch misprediction
- dcache miss
- icache miss
- dtlb miss
- itlb miss
- processor busy

From: Tullsen, Eggers, and Levy, "Simultaneous Multithreading: Maximizing On-chip Parallelism, ISCA, 1995

Simultaneous Multi-threading



Uniprocessor Parallelism

UNIPROCESSOR MEMORY SYSTEMS

Limitations of Memory System Performance

- Memory system, and not processor speed, is often the bottleneck for many applications.
- Memory system performance is largely captured by two parameters, latency and bandwidth.
- Latency is the time from the issue of a memory request to the time the data is available at the processor.
- Bandwidth is the rate at which data can be pumped to the processor by the memory system.

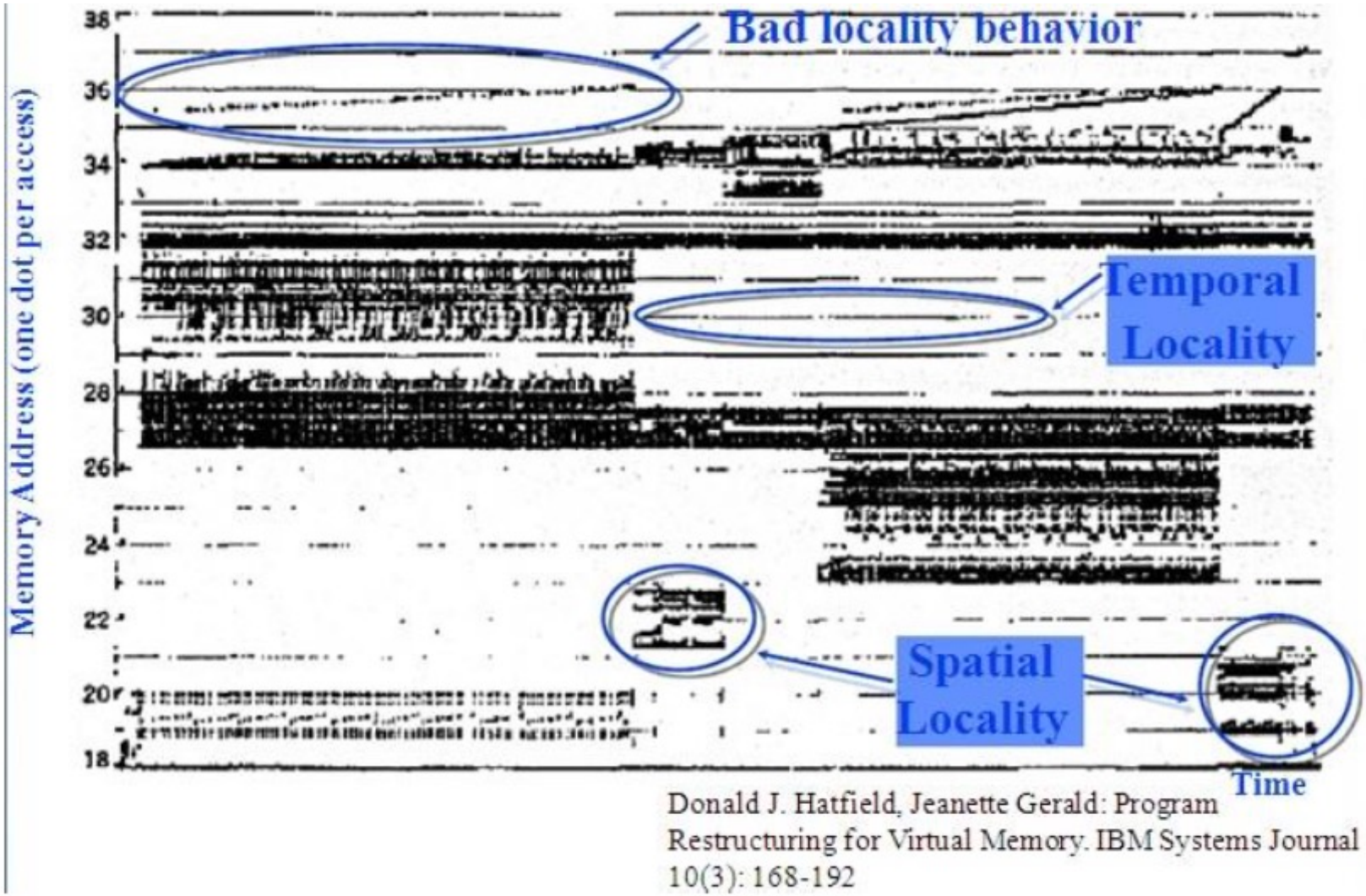
Limiting Force: Memory Wall

- How do architects address this gap?
  - Put small, fast "cache" memories between CPU and DRAM (Dynamic Random Access Memory).
  - Create a "memory hierarchy"

Principle of Locality

- Principle of Locality
  - Program access a relatively small portion of the address space at any instant of time
- Two Different Types of Locality
  - *Temporal Locality* (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
  - *Spatial Locality* (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straight line code, **array access**)
- Last 25 years, HW relied on locality for speed

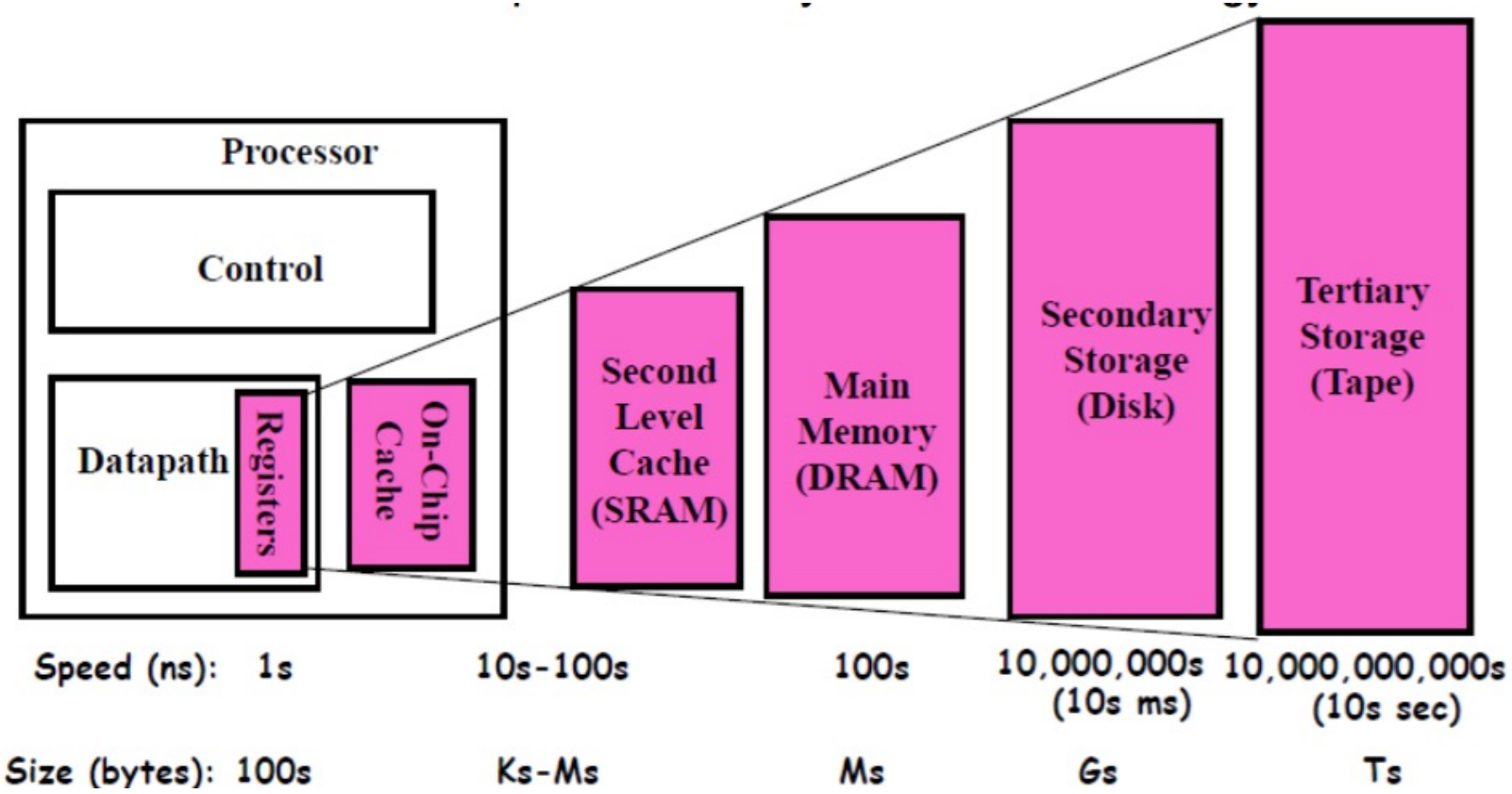
Programs with locality cache well



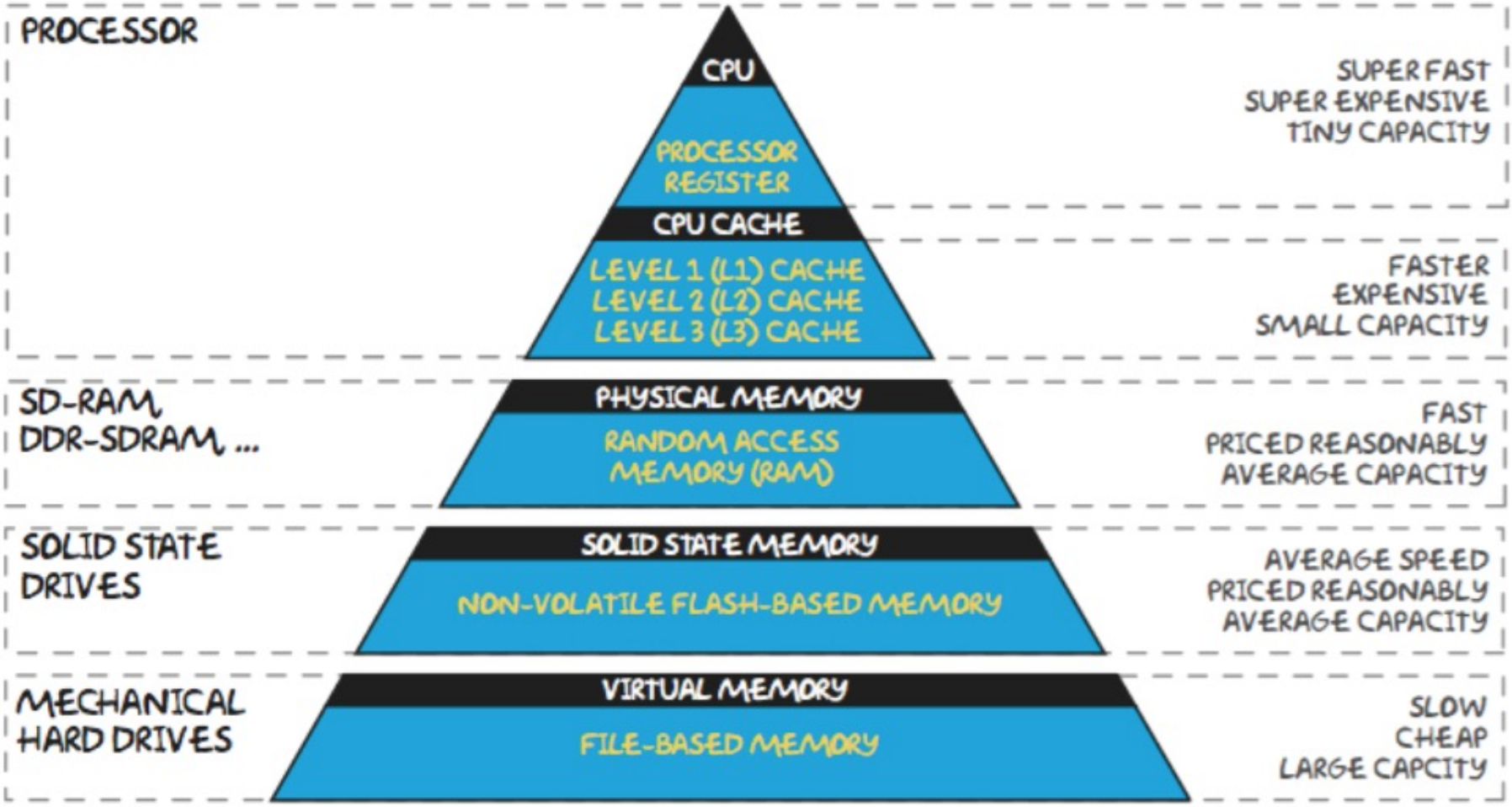
Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192

Memory Hierarchy

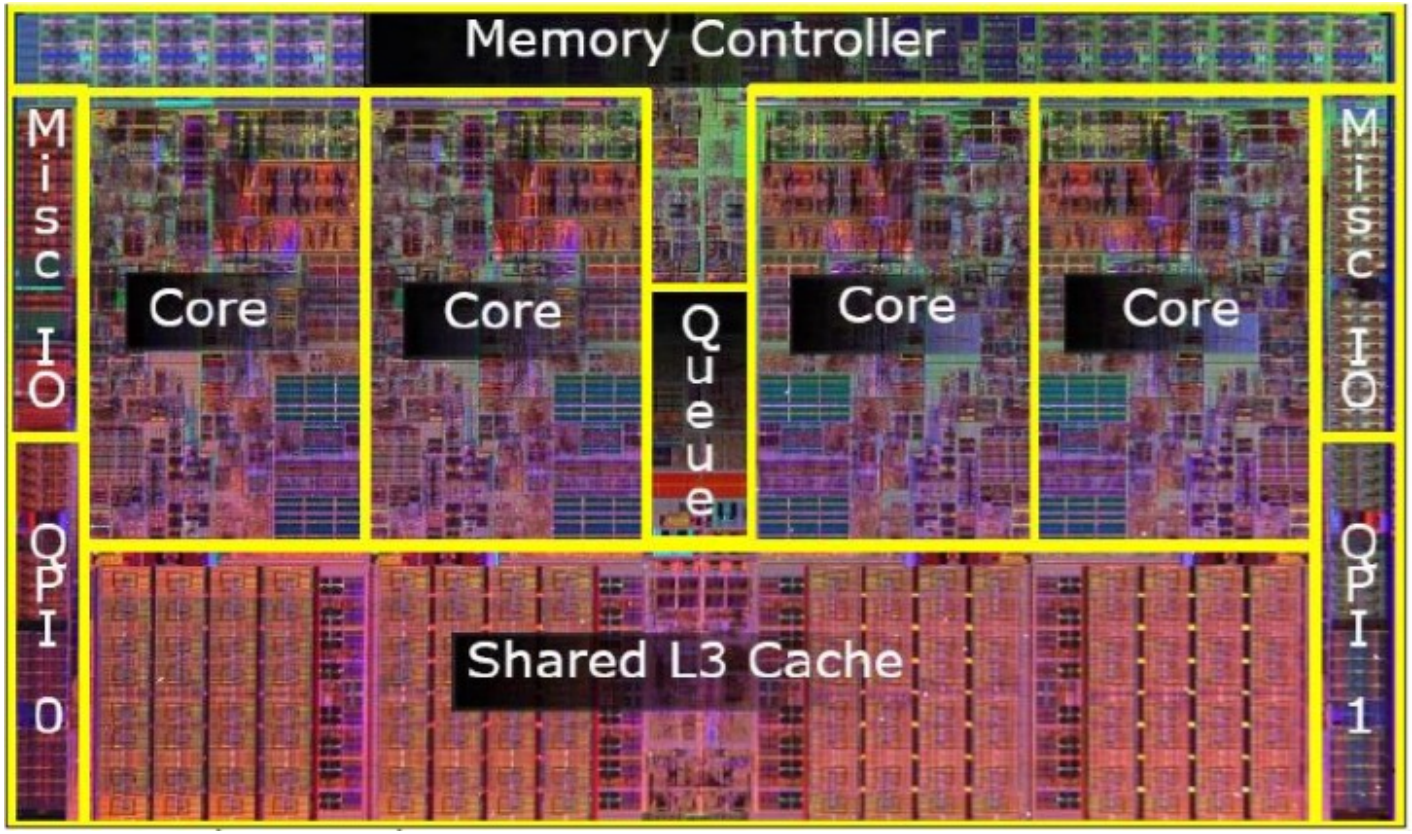
- Take advantage of the principle of locality to
  - Present as much memory as in the cheapest technology
  - Provide access at speed offered by the fastest technology



Memory Hierarchy



Example of Modern Core: Nehalem



- ON-chip cache resources
  - For each core: L1: 32KB instruction and 32KB data cache, L2: 1MB, L3: 8MB shared among all 4 cores
- Integrated, on-chip memory controller (DDR3)

Memory Latency: An Example

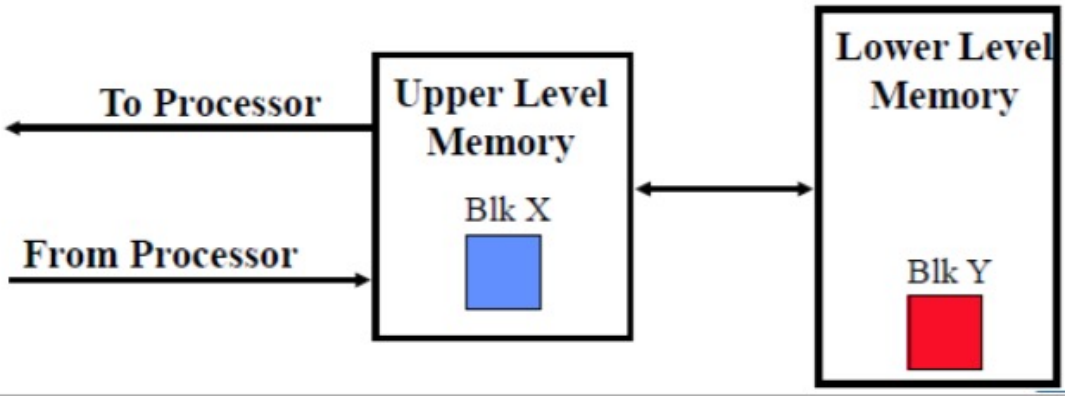
- A processor operating at 1 GHz (1 ns clock) connected to a DRAM with a latency of 100 ns (no caches).
- Assume the processor has 2 multiply-add units; capable of executing 4 instructions in each clock cycle
  - The peak processor rating is 4 GFLOPS.
  - Every time a memory request is made, the processor must wait 100 cycles before it can process the data.

Memory Latency: An Example (Cont'd)

- On the previous architecture, consider the problem of computing a dot-product of two vectors.
  - Performs one multiply-add on each pair of vector elements
    - Each floating point operation requires one data fetch.
  - Limited to one floating point operation every 100 ns, or a speed of 10 MFLOPS,
    - A very small fraction of the peak processor rating!

Memory Access Performance

- Hit: data appears in some blocks in the upper level (example: Block X)
  - Hit Rate: the fraction of memory access found in the upper level
  - Hit Time: Time to access the upper level which consists of RAM access time + Time to determine hit/miss
- Miss: data needs to be retrieve from a block in the lower level (Block Y)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Time to replace a block in the upper level + Time to deliver the block to the processor
- Hit Time << Miss Penalty



Improving Effective Memory Latency Using Caches

- Caches are small and fast memory elements between the processor and DRAM.
  - Acts as a low-latency high-bandwidth storage.
  - If a piece of data is repeatedly used, the effective latency of this memory system can be reduced by the cache.
- The fraction of data references satisfied by the cache is called the cache *hit ratio.*
- Cache hit ratio achieved by a code on a memory system often determines its performance.

Impact of Caches: Example

- Introduce a cache of size 32 KB with a latency of 1 ns or one cycle in previous example.
- Assume we multiply two matrices A and B of dimensions 32 × 32.
  - Both A and B (1K each), as well as the result matrix C can be in cache

Fetching the two matrices into the cache corresponds to fetching 2K words, which takes approximately 200 µs.

- Multiplying two $n \times n$ matrices takes $2n^3$ operations.
  - => 64K operations, which can be performed in 16K cycles (or 16 µs) at 4 instructions per cycle.
- The total time for the computation = 200 + 16 µs.
  - A peak computation rate of 64K/216 or 303 MFLOPS
  - Much better!

Impact of Memory Bandwidth

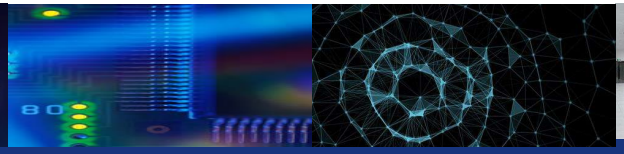
- Memory bandwidth is determined by the bandwidth of the memory bus as well as the memory units.
  - Memory bandwidth can be improved by increasing the size of memory blocks.
    - The underlying system takes *l* time units (where *l* is the latency of the system) to deliver *b* units of data (where *b* is the block size).
- Bandwidth has improved more than latency 23% per year vs 7% per year

Impact of Memory Bandwidth

- Note that increasing block size does not change latency of the system.
- Physically, it can be viewed as a wide data bus (4 words or 128 bits) connected to multiple memory banks.
- In practice, such wide buses are expensive to construct.
- In a more practical system, consecutive words are sent on the memory bus on subsequent bus cycles after the first word is retrieved.
- In order to take advantage of full bandwidth
  - The data layouts were assumed to be such that consecutive data words in memory were used by successive instructions
  - Spatial locality of reference
- If spatial locality is poor
  - Computations often need to be reordered to enhance spatial locality of reference.

Memory Hierarchy Lessons

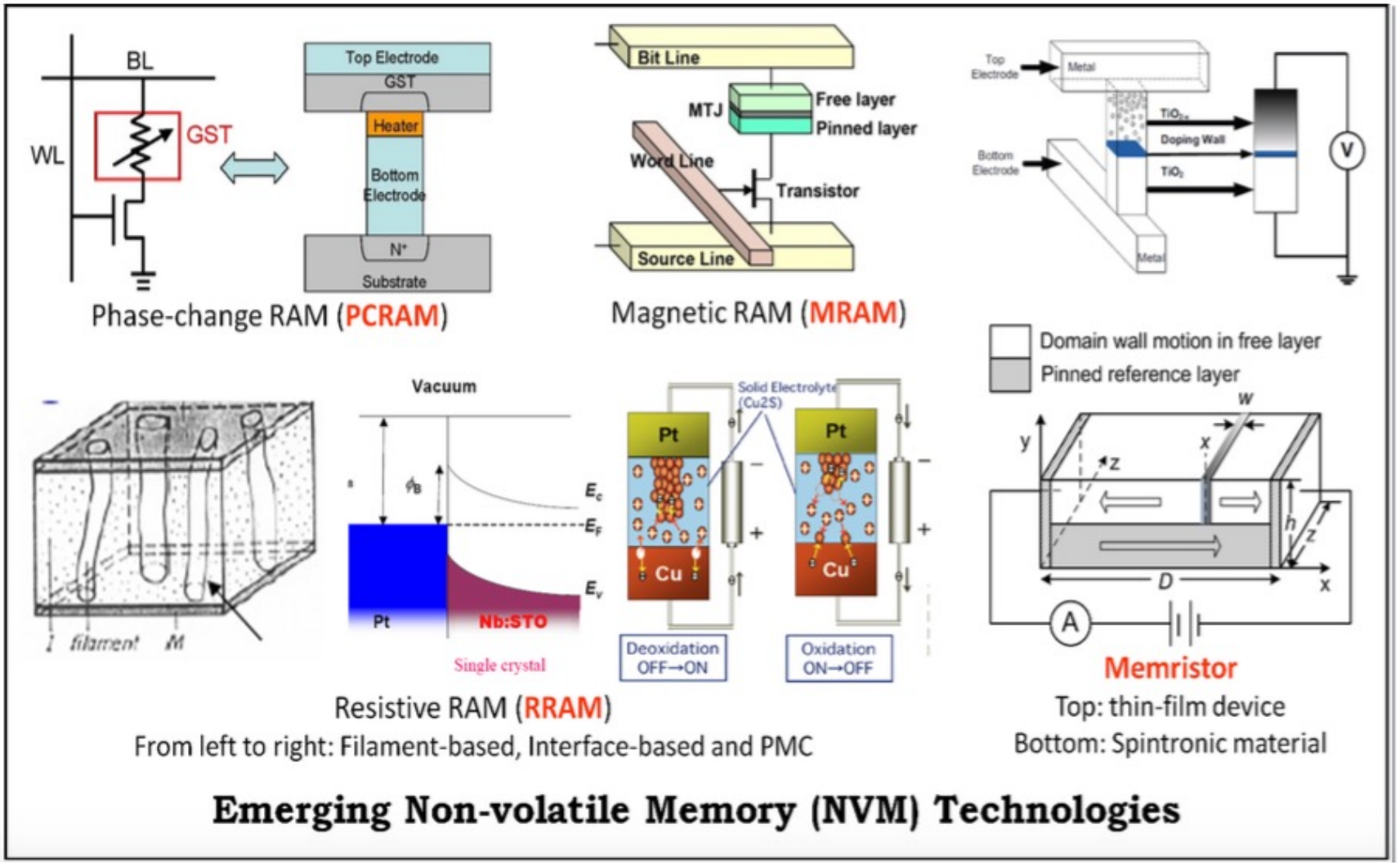
- Caches vastly impact performance
  - Cannot consider performance without considering memory hierarchy
- Actual performance of a simple program can be a complicated function of the architecture
  - Slight changes in the architecture or program change the performance Significantly
  - To write fast programs, need to consider architecture
    - True on sequential or parallel processor
  - We would like simple models to help us design efficient algorithms



Memory Hierarchy Lessons

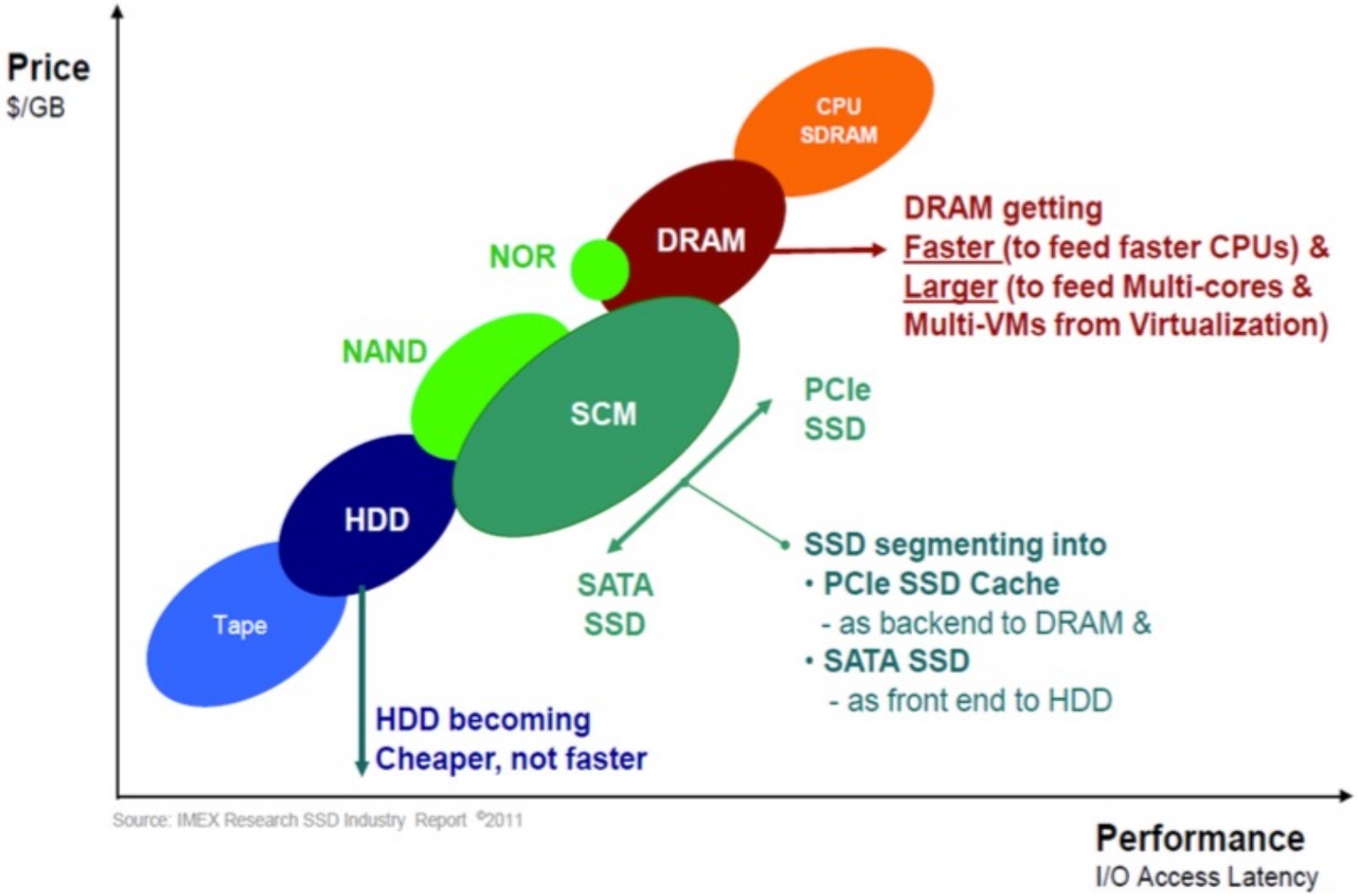
- Common technique for improving cache performance, called blocking or tiling
  - Idea: used divide-and-conquer to define a problem that fits in register/L1-cache/L2-cache
- Autotuning: Deal with complexity through experiments
  - Produce several different versions of code
    - Different algorithms, Blocking Factors, Loop orderings, etc
  - For each architecture, run different versions to see which is fastest
  - Can (in principle) navigate complex design options for optimum

NVM Technologies



New Advances on Memory and Storage

—Storage Class Memory

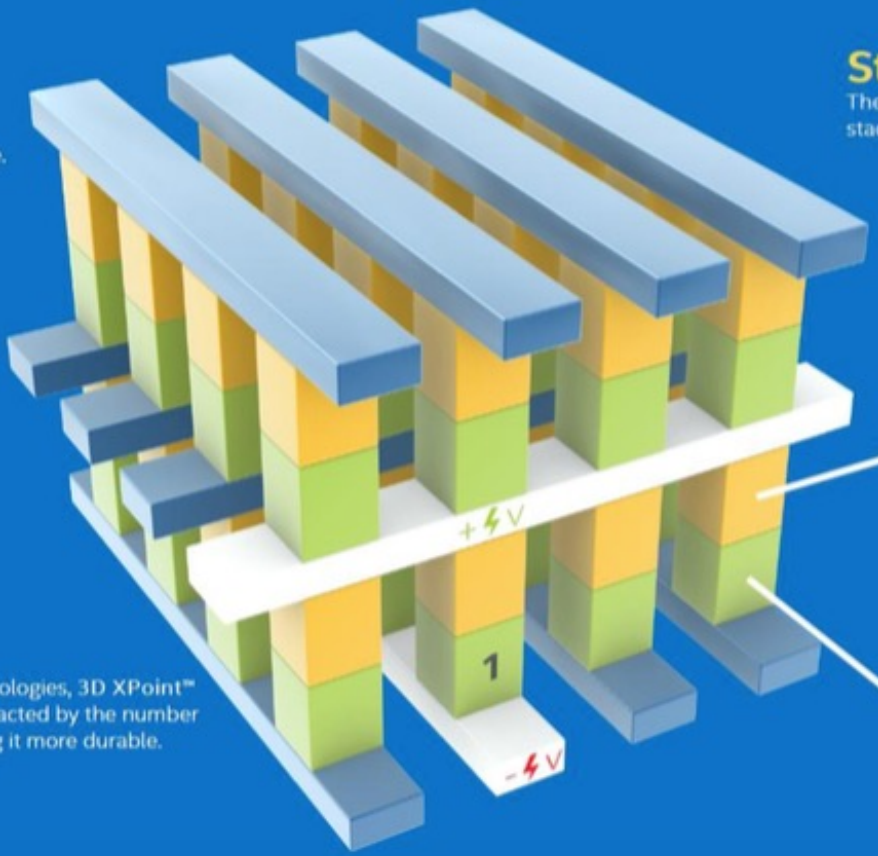


Intel-Micron 3D XPoint Technology

3D XPoint™ Technology:
An Innovative, High-Density Design

Cross Point Structure
Perpendicular wires connect submicroscopic columns. An individual memory cell can be addressed by selecting its top and bottom wire.

Stackable
These thin layers of memory can be stacked to further boost density.



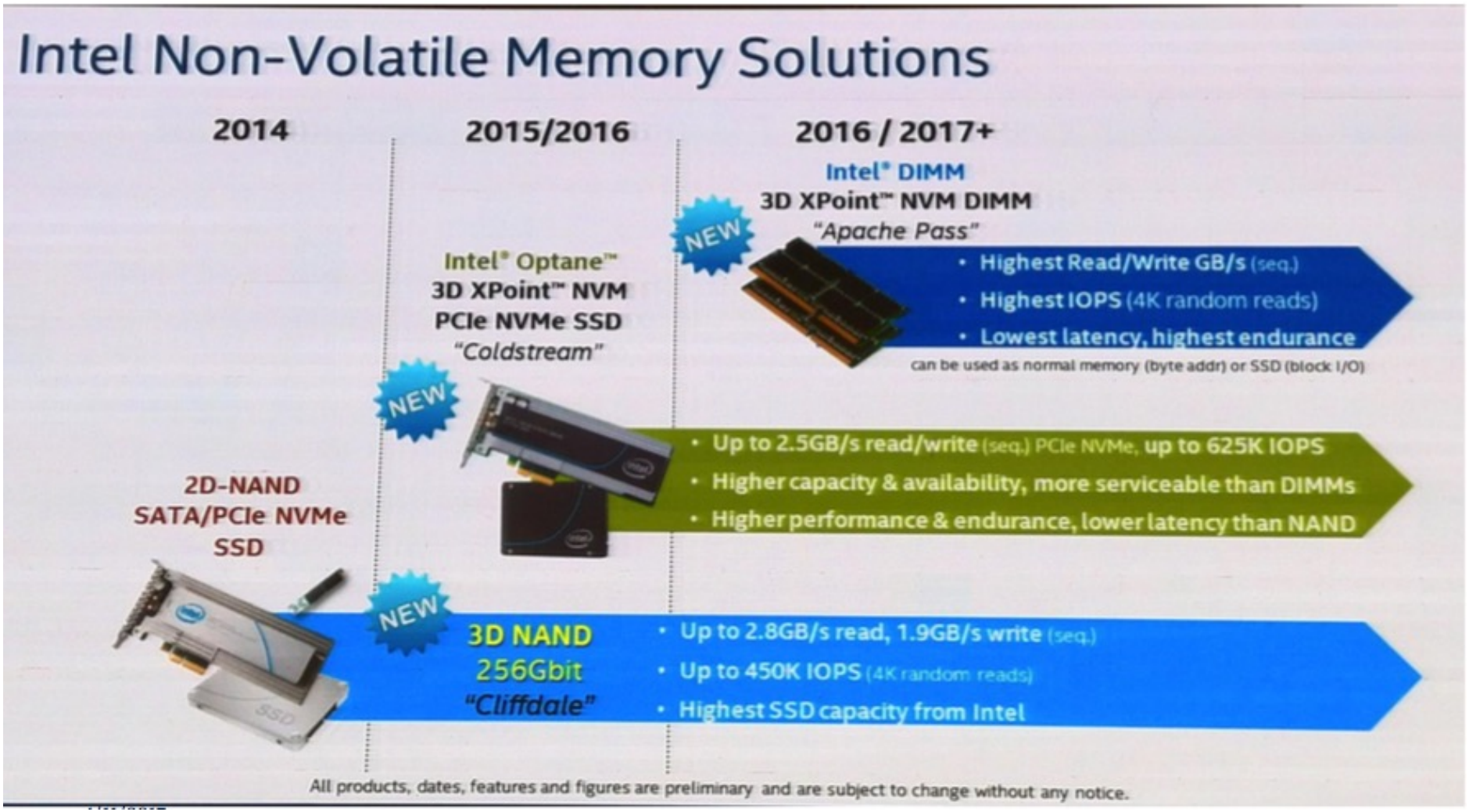
Non-Volatile
3D XPoint™ Technology is non-volatile—which means your data doesn't go away when your power goes away—making it a great choice for storage.

Selector
Whereas DRAM requires a transistor at each memory cell—making it big and expensive—the amount of voltage sent to each 3D XPoint™ Technology selector enables its memory cell to be written to or read without requiring a transistor.

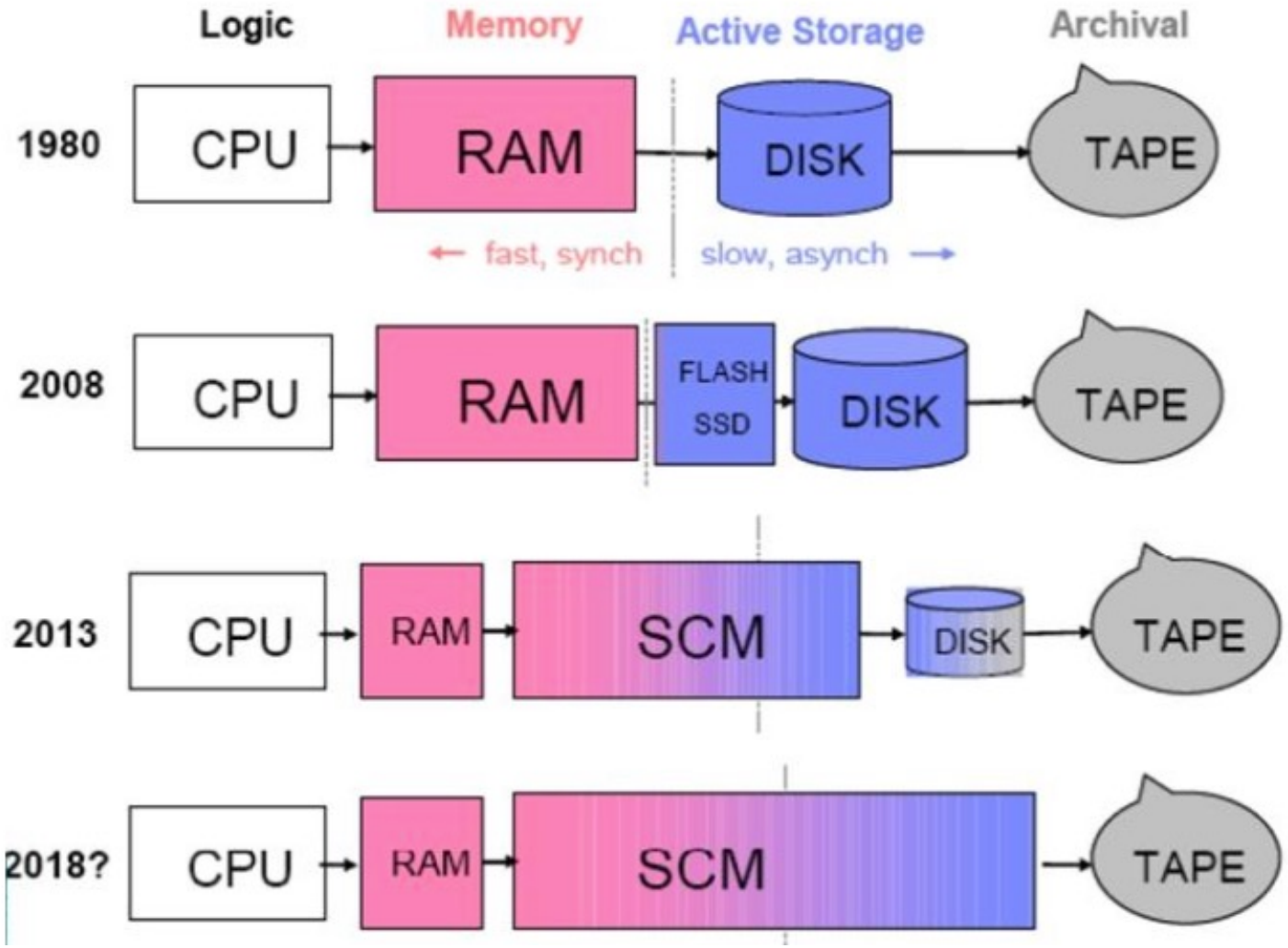
High Endurance
Unlike other storage memory technologies, 3D XPoint™ Technology is not significantly impacted by the number of write cycles it can endure, making it more durable.

Memory Cell
Each memory cell can store a single bit of data.

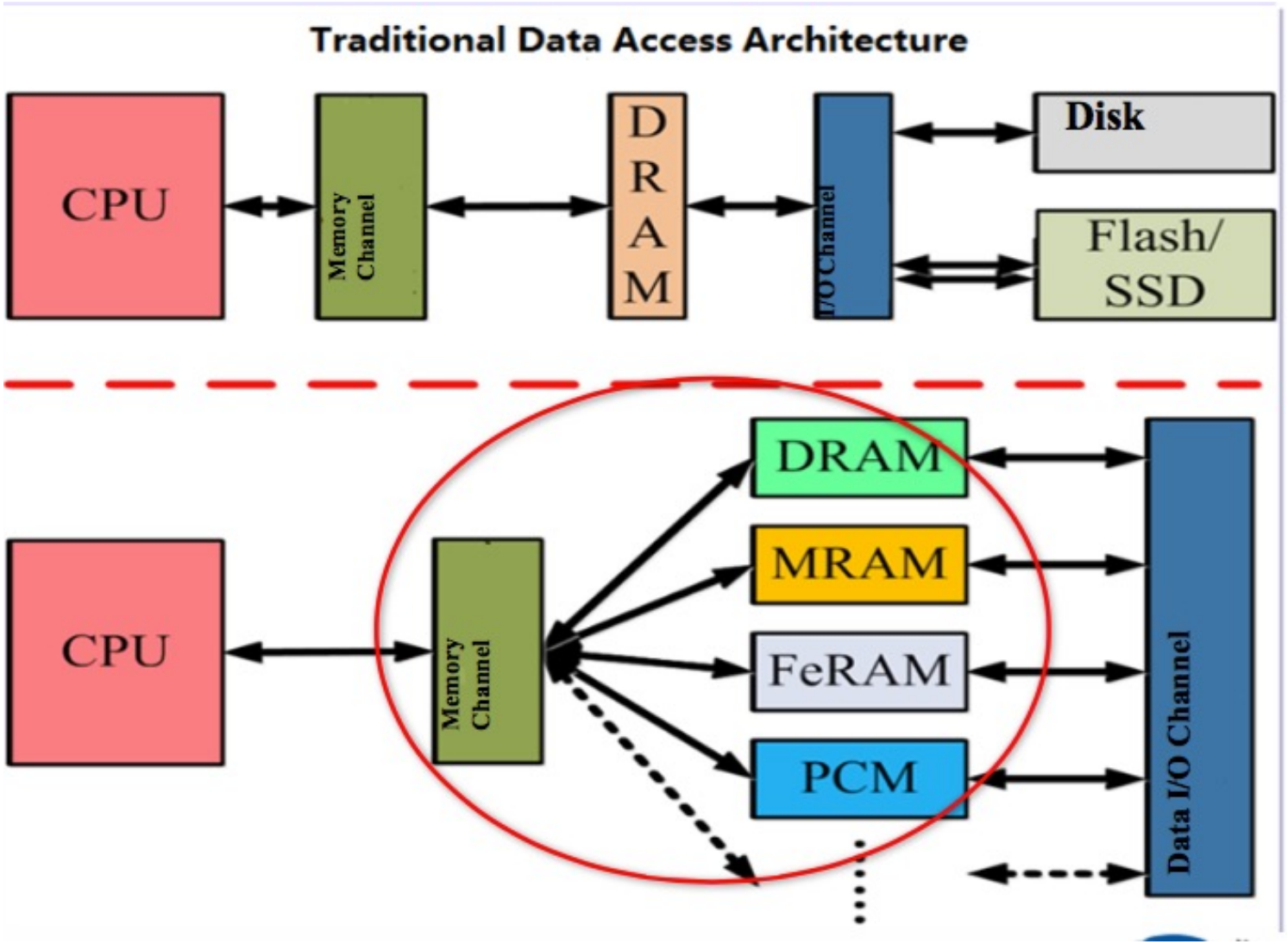
Intel Non-Volatile Memory Solutions



Break the I/O Bottleneck

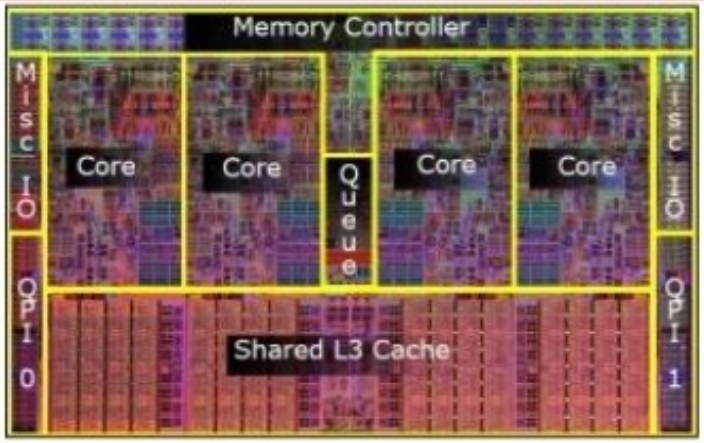


New In-Memory Computing Architecture

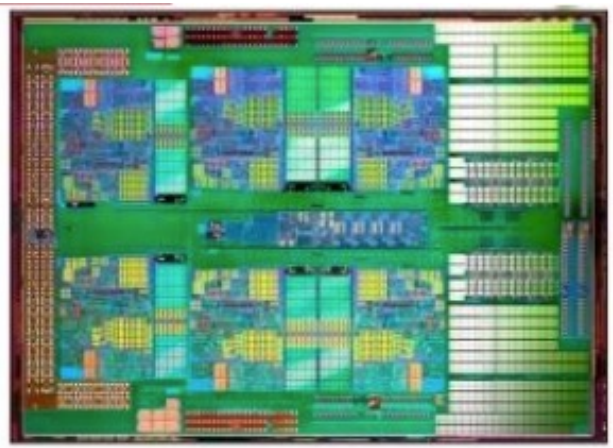


MULTICORE CHIPS

Parallel Chip-Scale Processors



Intel Core 2 Quad: 4 Cores



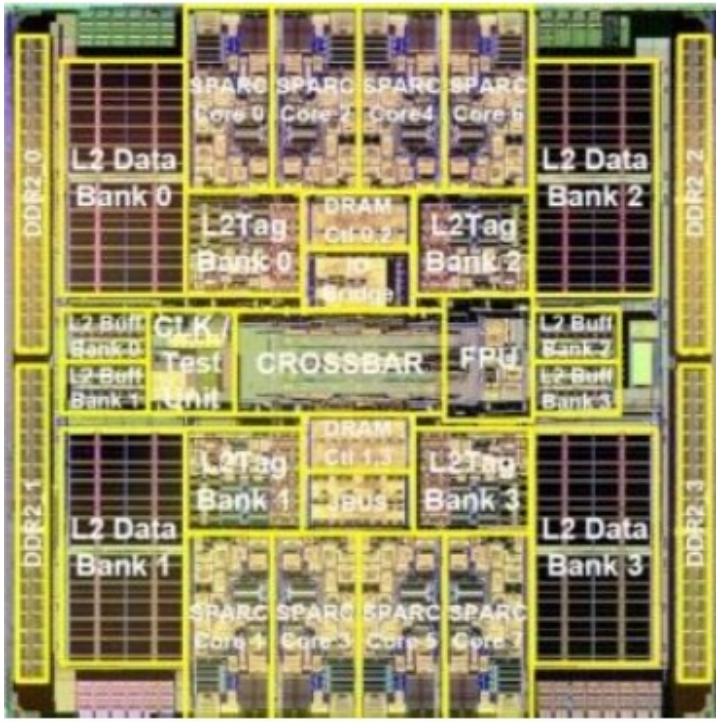
AMD Opteron: 6 Cores

- Multicore processors emerging in general-purpose market due to power limitations in single-core performance scaling
  - 4-16 cores in 2009, connected as cache-coherent SMP
  - Cache-coherent shared memory
- Embedded applications need large amounts of computation
  - Recent trend to build “extreme” parallel processors with dozens to hundreds of parallel processing elements on one die
  - Often connected via on-chip networks, with no cache coherence
  - Examples: 188 core “Metro” chip from CISCO

Sun's T1 ("Niagara")

- Highly Threaded
  - 8 Cores
  - 4 Threads/Core
- Target: Commercial server Applications
  - High thread level parallelism (TLP)
    - Large numbers of parallel client requests
  - Low instruction level parallelism (ILP)
    - High cache miss rates
    - Many unpredictable branches
    - Frequent load-load dependencies
- Power, cooling, and space are major concerns for data centers
- Metric: Performance/Watt/Sq. Ft.
- Approach: Multicore, Fine-grain multithreading, Simple

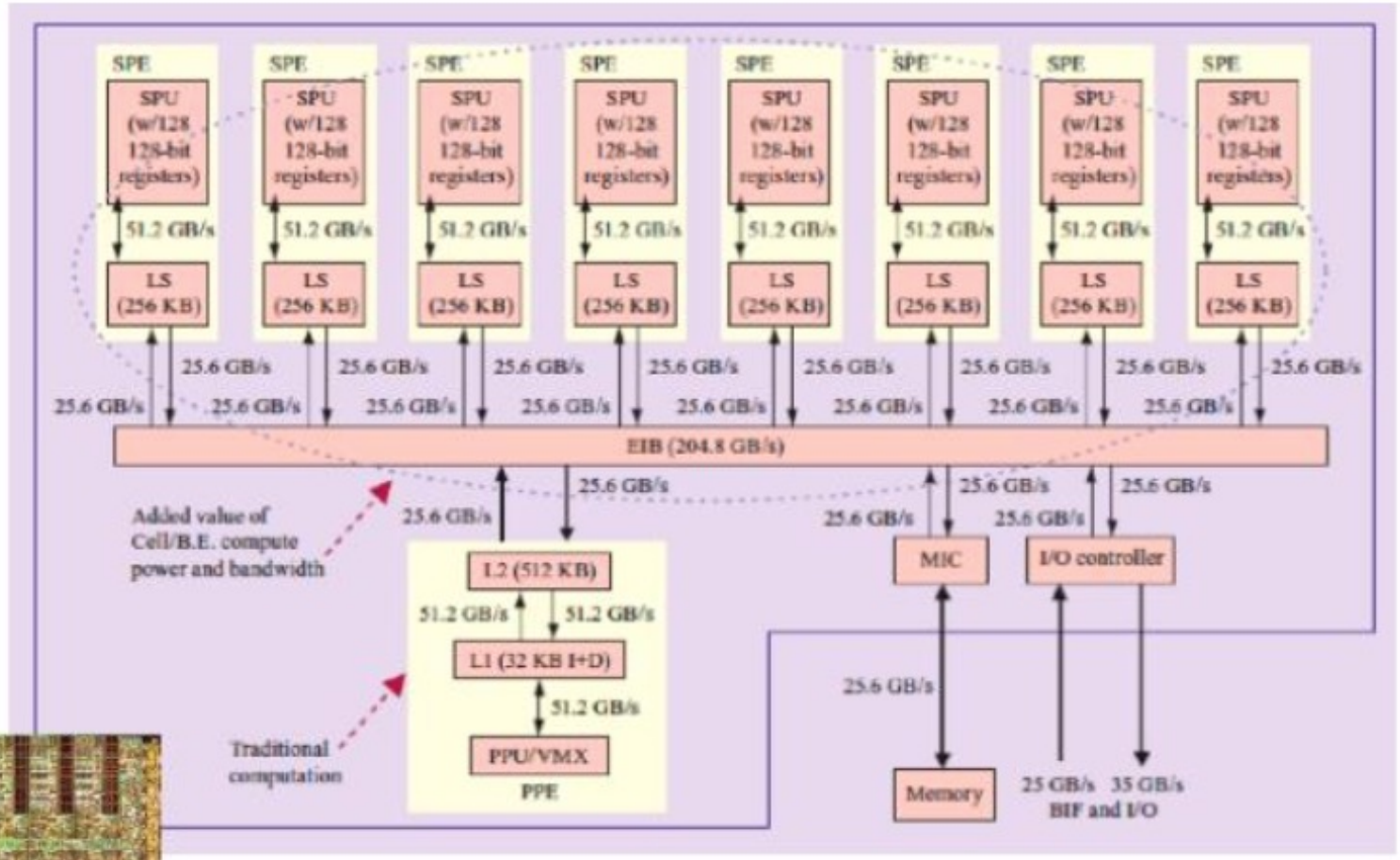
pipeline, Small L1 caches, Shared L2

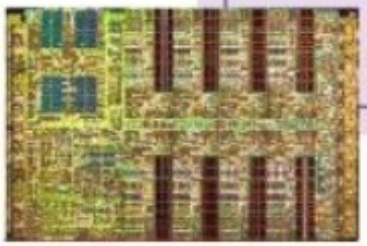


Embedded Parallel Processors

- Often embody a mixture of old architectural styles and ideas
- Exposed memory hierarchies and interconnection networks
  - Programmers code to the "metal" to get best cost/power/performance
  - Portability across platforms less important
- Customized synchronization mechanisms
  - Interlocked communication channels (processor blocks on read if data not ready)
  - Barrier signals
  - Specialized atomic operation units
- Many more, simpler cores

IBM Cell Processor (Playstation-3)

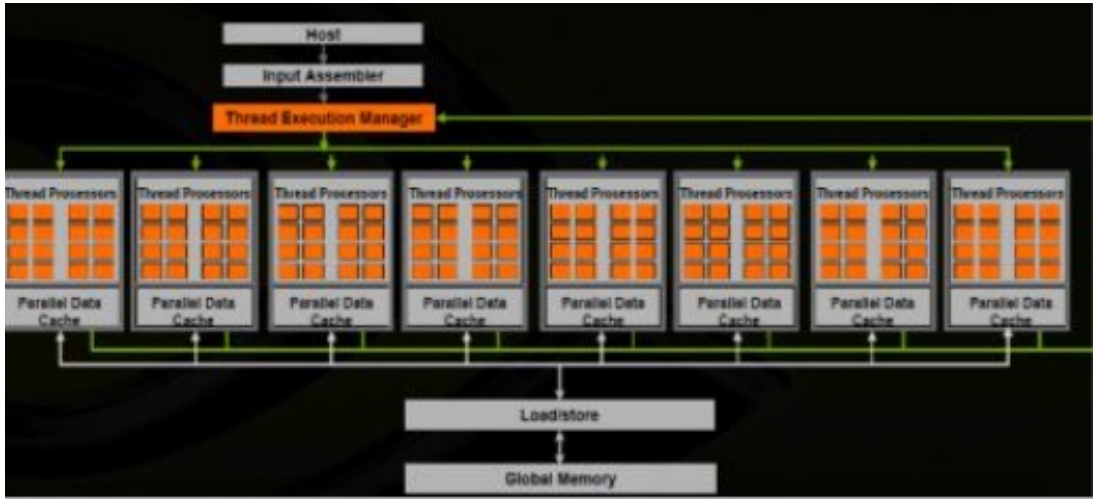




One 2-way threaded PowerPC core (PPE), plus eight specialized short-SIMD cores (SPE)

GPU

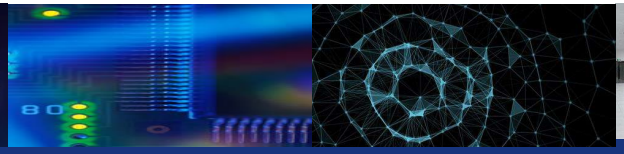
- This is a GPU (Graphics Processor Unit)
  - Available in many desktops
- Example: 16 cores similar to a vector processor with 8 lanes (128 stream processors total)
  - Processes threads in SIMD groups of 32 (a “warp”)
  - Some stripining done in hardware
- Threads can branch, but loses performance compared to when all threads are running same code
- Complete parallel programming environment (CUDA)
  - A lot of parallel codes have been ported to these GPUs
  - For some data parallel applications, GPUs provide the fastest implementations



Nvidia Tesla GPU

Features	Tesla K80[1]	Tesla K40
GPU	2x Kepler GK210	1 Kepler GK110B
Peak double precision floating point performance	2.91 Tflops (GPU Boost Clocks) 1.87 Tflops (Base Clocks)	1.66 Tflops (GPU Boost Clocks) 1.43 Tflops (Base Clocks)
Peak single precision floating point performance	8.74 Tflops (GPU Boost Clocks) 5.6 Tflops (Base Clocks)	5 Tflops (GPU Boost Clocks) 4.29 Tflops (Base Clocks)
Memory bandwidth (ECC off)[2]	480 GB/sec (240 GB/sec per GPU)	288 GB/sec
Memory size (GDDR5)	24 GB (12GB per GPU)	12 GB
CUDA cores	4992 (2496 per GPU)	2880

1 Tesla K80 specifications are shown as aggregate of two GPUs.
2 With ECC on, 6.25% of the GPU memory is used for ECC bits. For example, 6 GB total memory yields 5.25 GB of user available memory with ECC on.



WHAT IS PARALLEL ARCHITECTURE

What is Parallel Architecture?

- Machines with multiple processors

Intel's Quad Core i7

Apple MacPro

Blacklight at the PSC (4096 cores)

Hadoop cluster at Yahoo!

What is Parallel Architecture?

- A parallel computer is a collection of processing elements that cooperate to solve large problems fast Some broad issues
- Resource Allocation
  - how large a collection?
  - how powerful are the elements?
  - how much memory?
- Data access, Communication and Synchronization
  - how do the elements cooperate and communicate?
  - how are data transmitted between processors?
  - what are the abstractions and primitives for cooperation?
- Performance and Scalability
  - how does it all translate into performance? how does it scale?

Types of Parallelism

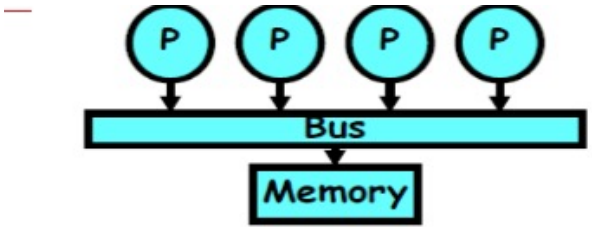
A PARALLEL ZOO OF ARCHITECTURES

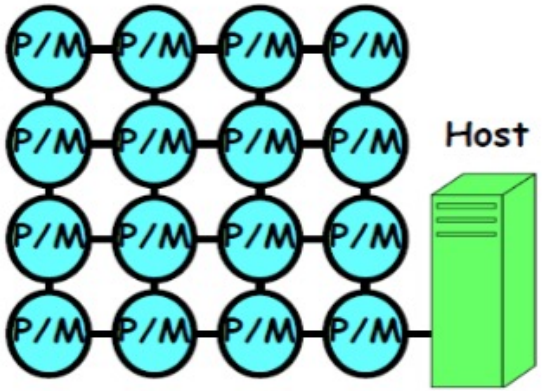
MIMD Machines

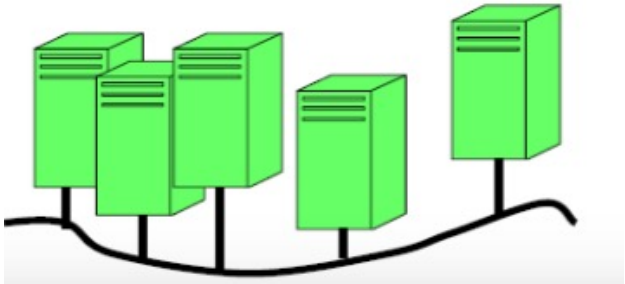
- Multiple Instruction, Multiple Data (MIMD)
  - Multiple independent instruction streams, program counters, etc
  - Called “multiprocessing” instead of “multithreading”
    - Although, each of the multiple processors may be multithreaded
  - When independent instruction streams confined to single chip, becomes a “multicore” processor
- Shared memory: Communication through Memory
  - Option 1: no hardware global cache coherence
  - Option 2: hardware global cache coherence
- Message passing: Communication through Messages
  - Applications send explicit messages between nodes in order to communicate
- For most machines, shared memory built on top of message- passing network
  - Bus-based machines are “exception”

Examples of MIMD Machines

- Symmetric Multiprocessor
  - Multiple processors in box with shared memory communication
  - Current MultiCore chips like this
  - Every processor runs copy of OS
- Non-uniform shared-memory with separate I/O through host
  - Multiple processors
    - Each with local memory
    - general scalable network
  - Extremely light “OS” on node provides simple services
    - Scheduling/synchronization
  - Network-accessible host for I/O
- Cluster
  - Many independent machine connected with general network
  - Communication through messages



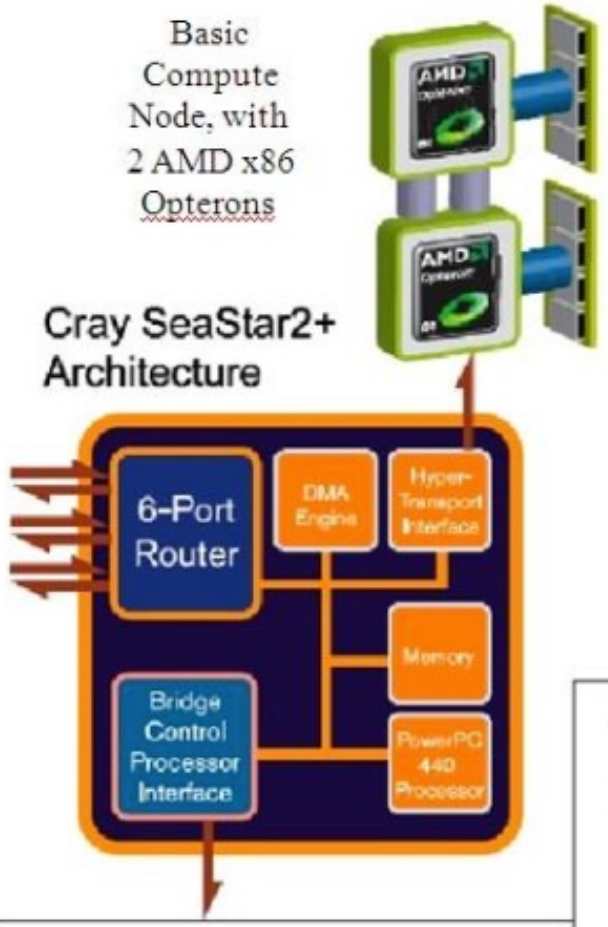




Cray XT5 (2007)

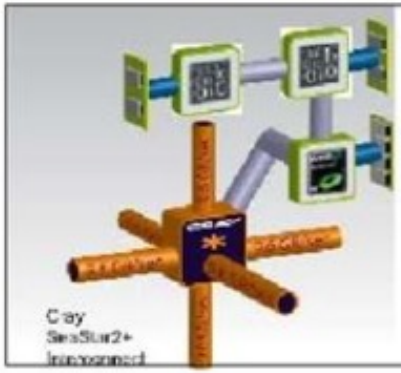
Basic Compute Node, with 2 AMD x86 Opterons

Cray SeaStar2+ Architecture



Vector Node

4-way SMP of SX2 Vector CPUs (8 lanes each)



Reconfigurable Logic Node

2 FPGAs + Opteron

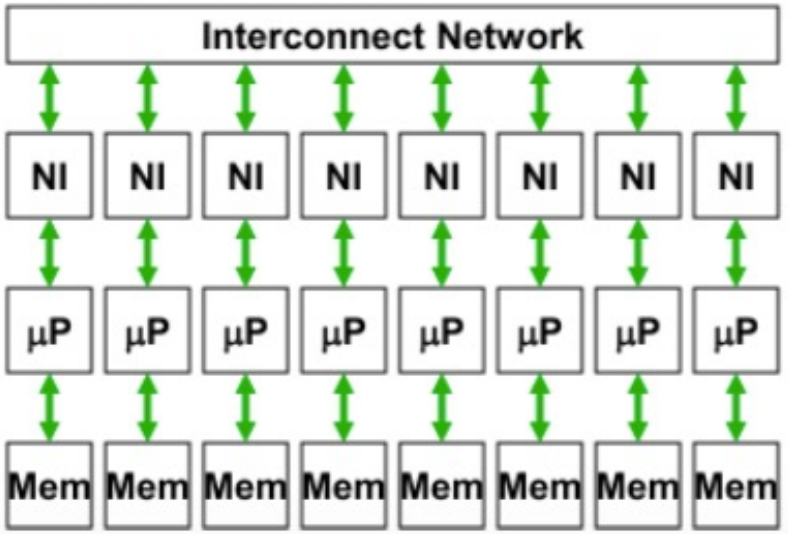
Also, XMT Multithreaded Nodes based on MTA design (128 threads per processor)

Processor plugs into

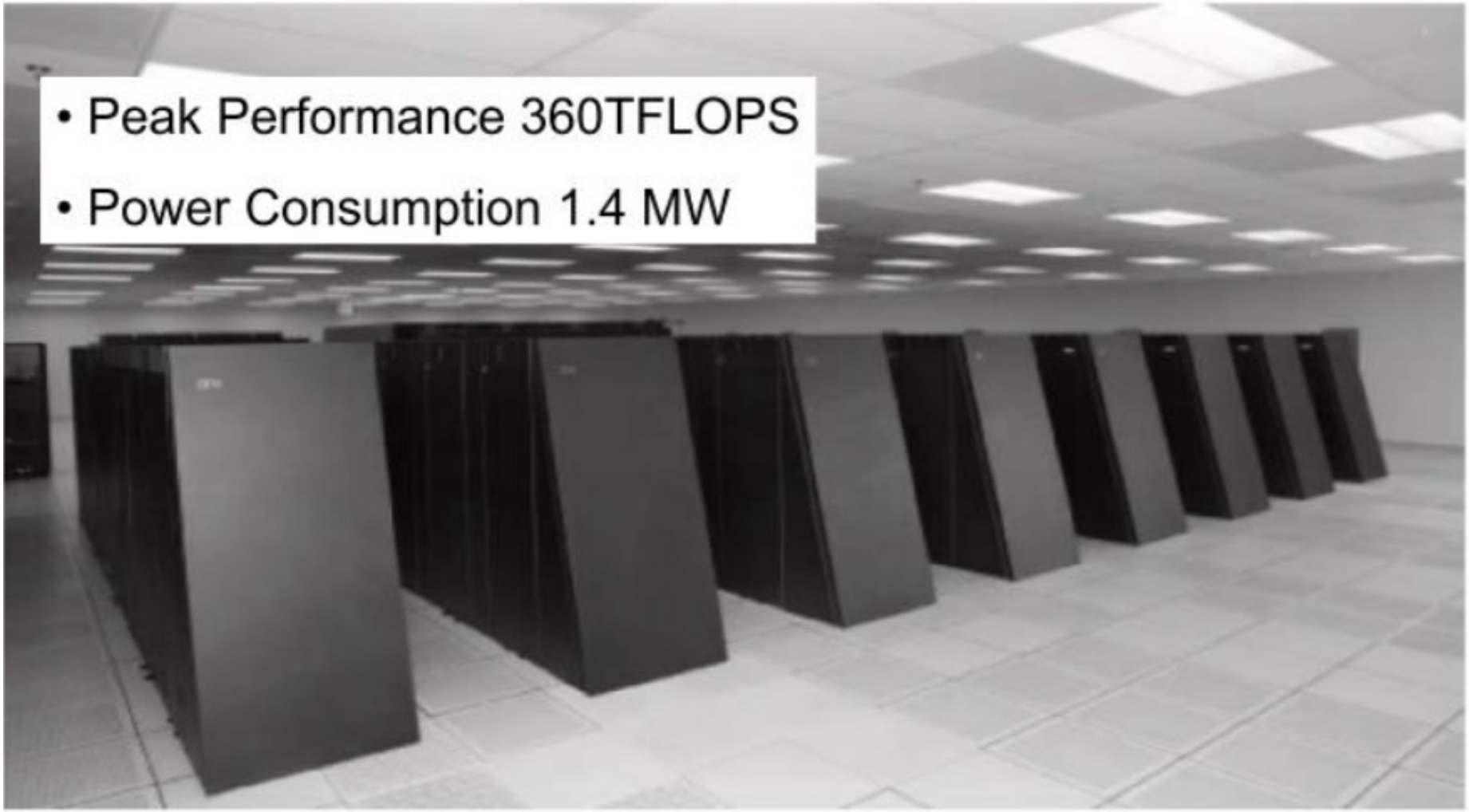
Massively Parallel Processors

- Initial Research Projects
  - Caltech Cosmic Cube (early 1980s) using custom Mosaic processors
  - J-Machine (early 1990s) MIT
- Commercial Microprocessors including MPP Support
  - Transputer (1985)
  - nCube-1(1986) /nCube-2 (1990)
- Standard Microprocessors + Network Interfaces
  - Intel Paragon/i860 (1991)
  - TMC CM-5/SPARC (1992)
  - Meiko CS-2/SPARC (1993)
  - IBM SP-1/POWER (1993)
- MPP Vector Supers
  - Fujitsu VPP500 (1994)

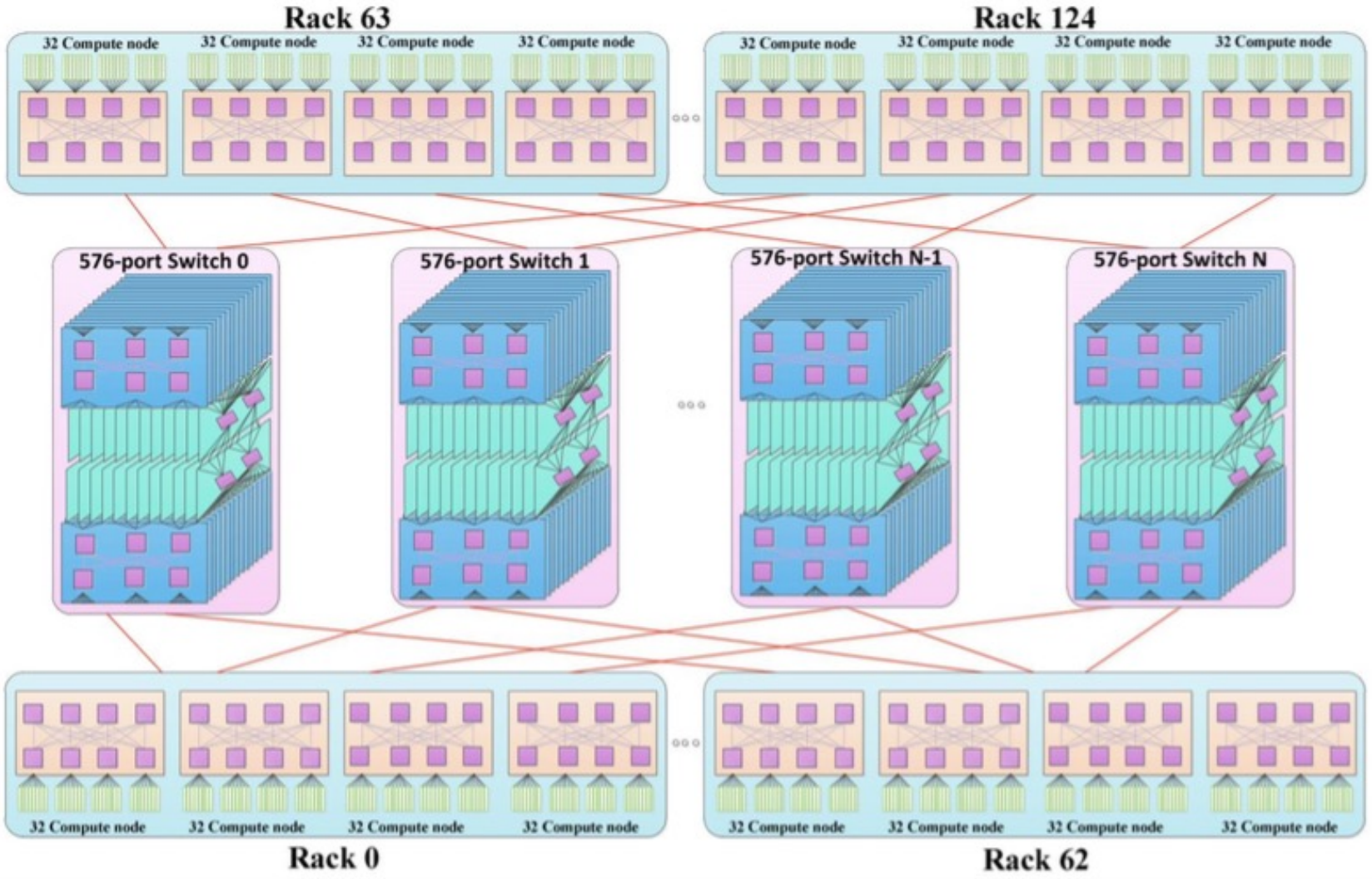
Designs scale to 100s-10,000s of nodes



BG/L 64K Processor System



Tianhe-2


Rack 63
Rack 124
32 Compute node
576-port Switch 0
576-port Switch 1
576-port Switch N-1
576-port Switch N
Rack 0
Rack 62

Homework

1. Write an essay describing a research problem in your major that would benefit from the use of parallel computing. Provide a rough outline of how parallelism would be used. Would you use task- or data-parallelism?

2. Assume that you have the following mix of instructions with average CPIs:

	% of Mix	Average CPI
ALU	47%	6.7
Load	19%	7.9
Branch	20%	5.0
Store	14%	7.1

The clock rate for this machine is 1 GHz.

You want to improve the performance of this machine, and are considering redesigning your multiplier to reduce the average CPI of multiply instructions. (Digress – why do multiplies take longer than adds?) If you make this change, the CPI of multiply instructions would drop to 6 (from 8). The percentage of ALU instructions that are multiply instructions is 23%. How much will performance improve by?

Thank You !