Advanced Computing Architecture

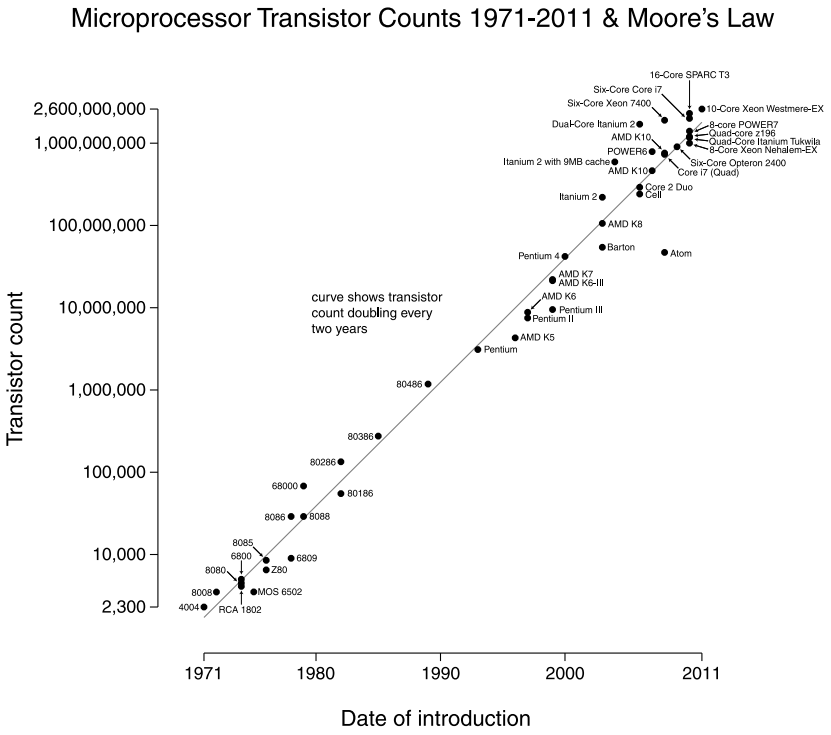
SAPIENZA
UNIVERSITÀ DI ROMA

Alessandro Pellegrini

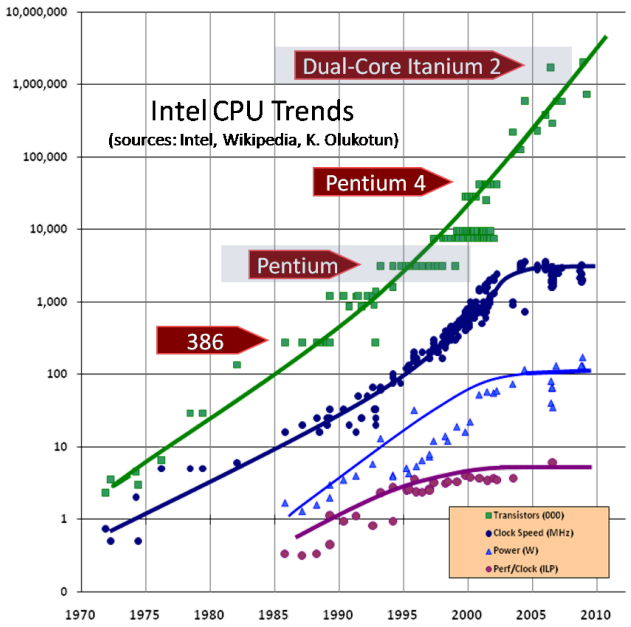
Department of Computer, Control, and Management Engineering
Sapienza, University of Rome

A.Y. 2014/2015

Moore's Law (1965)



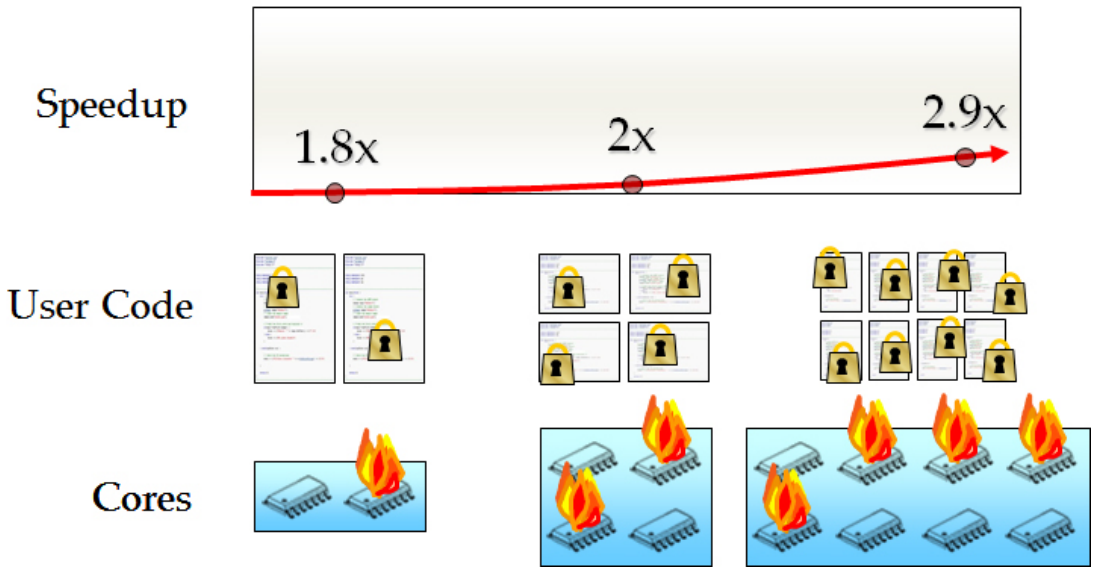
Effect of this Technological Trend



- Implications of Moore's Law have changed since 2003
- 130W is considered an upper bound (the *power wall*)

$$P = ACV^2f$$

Multicore Software Scaling



Why should we parallelize?

Exploitation of Computing Power provided by more processing units allows to solve *more complex/larger* problems

- **SPEED-UP**
  To solve problems of a given size in less time
- **SCALE-UP**
  To solve bigger-sized problems in comparable time

In general parallelism can improve *cost/performance ratio*

Speedup Performance Models

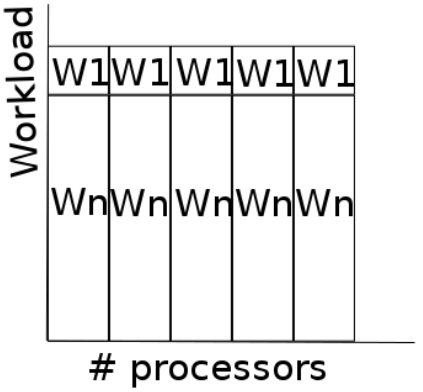
Amdahl Law—Fixed-size Model (1967)

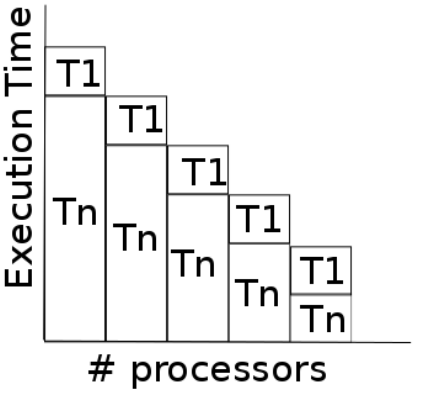
- The workload is fixed: it studies how the behaviour of the *same* program varies when adding more computing power

$$S_{Amdahl} = \frac{T_s}{T_p} = \frac{T_s}{\alpha T_s + (1-\alpha)\frac{T_s}{p}} = \frac{1}{\alpha + \frac{(1-\alpha)}{p}}$$

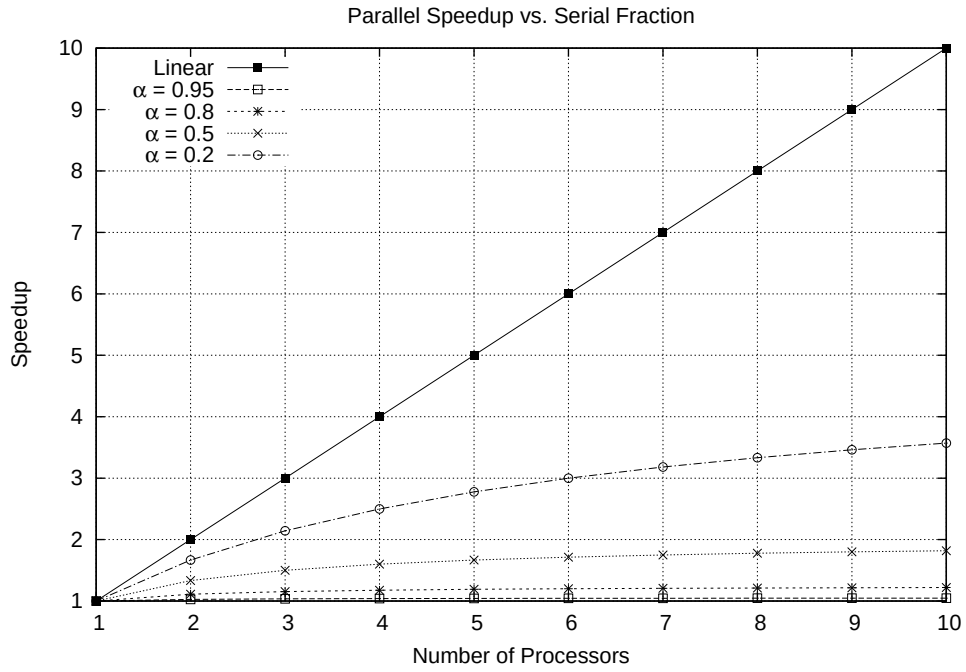
- where:
  - $\alpha \in [0,1]$: Serial fraction of the program
  - $p \in \mathbb{N}$: Number of processors
  - T_s : Serial execution time
  - T_p : Parallel execution time
- It can be expressed as well vs. the *parallel fraction* $P = 1 - \alpha$

Fixed-size Model





Speed-up According to Amdahl



How Real is This?

$$\lim_{p \to \infty} = \frac{1}{\alpha + \frac{(1-\alpha)}{p}} = \frac{1}{\alpha}$$

- So if the sequential fraction is 20%, we have:

$$\lim_{p \to \infty} = \frac{1}{0.2} = 5$$

- Speedup 5 using *infinte* processors!

Gustafson Law—Fixed-time Model (1989)

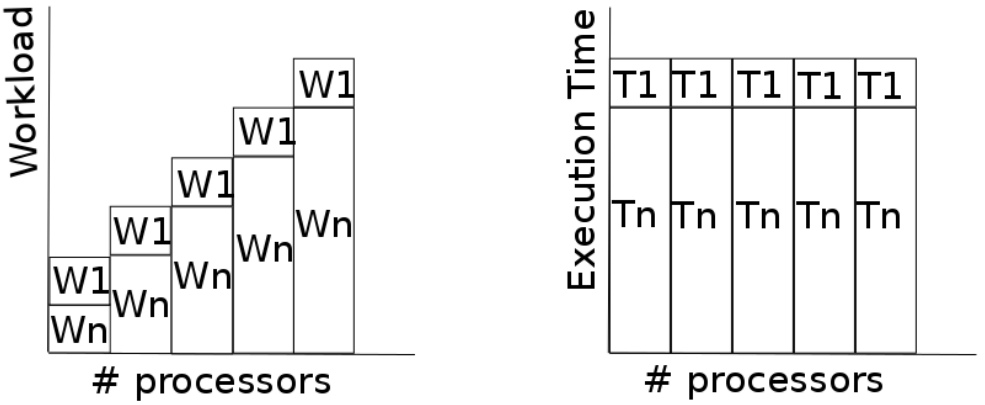
- The execution time is fixed: it studies how the behaviour of a *scaled* program varies when adding more computing power

$$W' = \alpha W + (1 - \alpha) p W$$

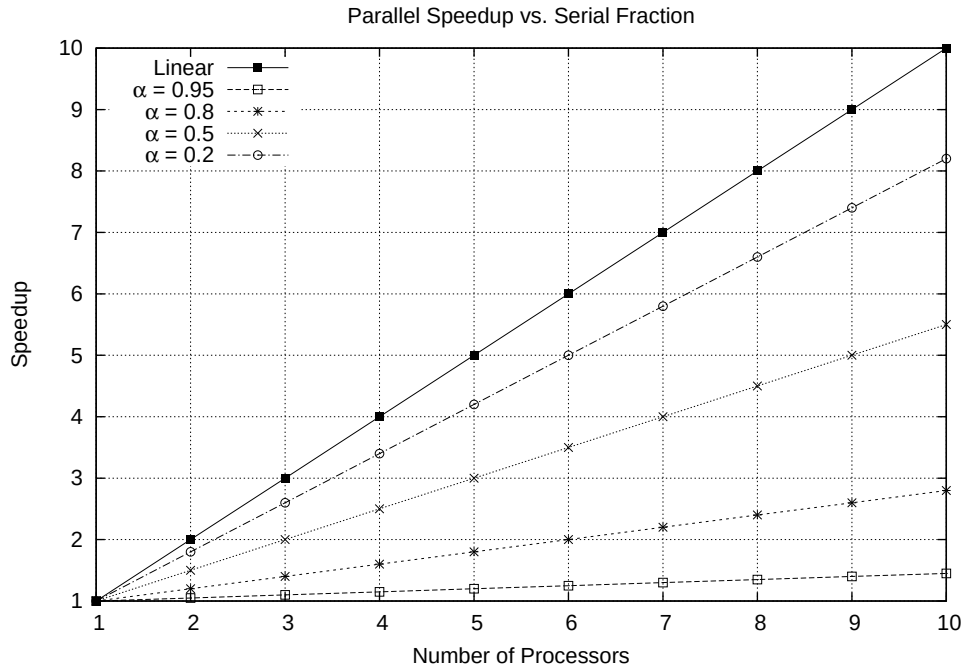
$$S_{Gustafson} = \frac{W'}{W} = \alpha + (1 - \alpha) p$$

- where:
  - $\alpha \in [0, 1]$: Serial fraction of the program
  - $p \in \mathbb{N}$: Number of processors
  - W : Original Workload
  - W' : Scaled Workload

Fixed-time Model



Speed-up According to Gustafson



Amdahl vs. Gustafson—a Driver's Experience

Amdahl Law:

A car is traveling between two cities 60 Kms away, and has already traveled half the distance at 30 Km/h. No matter how fast you drive the last half, it is impossible to achieve 90 Km/h average speed before reaching the second city. It has already taken you 1 hour and you only have a distance of 60 Kms total: Going infinitely fast you would only achieve 60 Km/h.

Gustafson Law:

A car has been travelling for some time at less than 90 Km/h. Given enough time and distance to travel, the car's average speed can always eventually reach 90 Km/h, no matter how long or how slowly it has already traveled. If the car spent one hour at 30 Km/h, it could achieve this by driving at 120 Km/h for two additional hours.

Sun, Ni Law—Memory-bounded Model (1993)

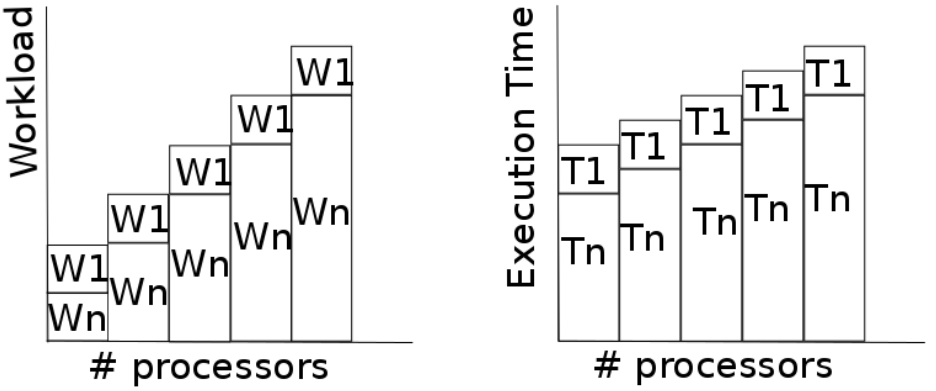
- The workload is scaled, bounded by *memory*

$$S_{Sun-Ni} = \frac{\text{sequential time for Workload, } W^*}{\text{parallel time for Workload, } W^*} =$$

$$= \frac{\alpha W + (1-\alpha)G(p)W}{\alpha W + (1-\alpha)G(p)\frac{W}{p}} = \frac{\alpha + (1-\alpha)G(p)}{\alpha + (1-\alpha)\frac{G(p)}{p}}$$

- where:
  - $G(p)$ describes the workload increase as the memory capacity increases
  - $W^* = \alpha W + (1-\alpha)G(p)W$

Memory-bounded Model



Speed-up According to Sun, Ni

$$S_{Sun-Ni} = \frac{\alpha + (1-\alpha)G(p)}{\alpha + (1-\alpha)\frac{G(p)}{p}}$$

- If G(p) = 1

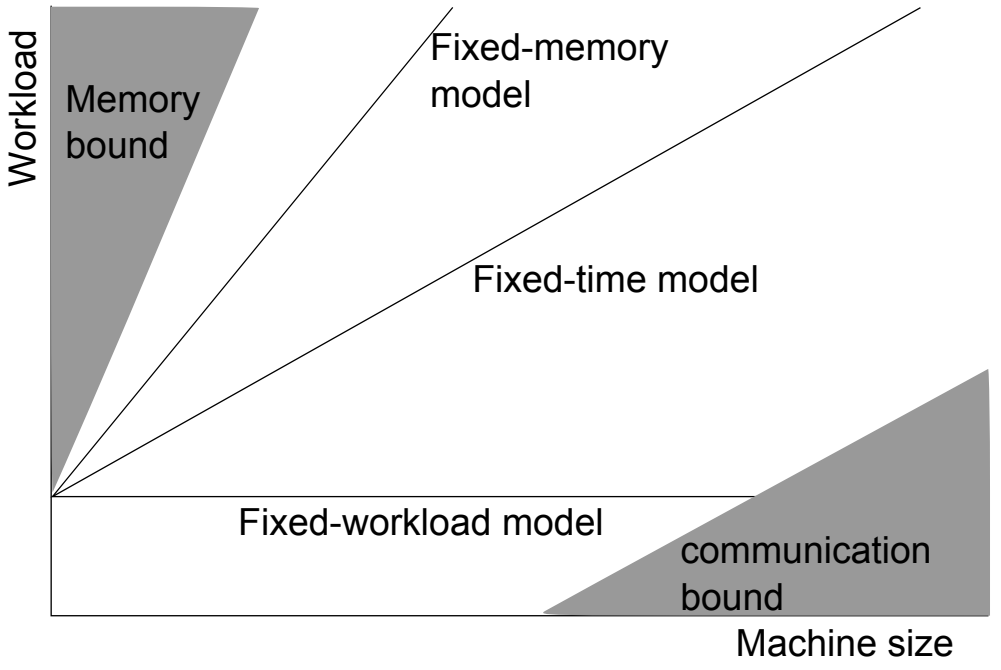
$$S_{Amdahl} = \frac{1}{\alpha + \frac{(1-\alpha)}{p}}$$

- If G(p) = p

$$S_{Gustafson} = \alpha + (1-\alpha)p$$

In general $G(p) > p$ giving an higher scale-up

Application Model for Parallel Computers



Scalability

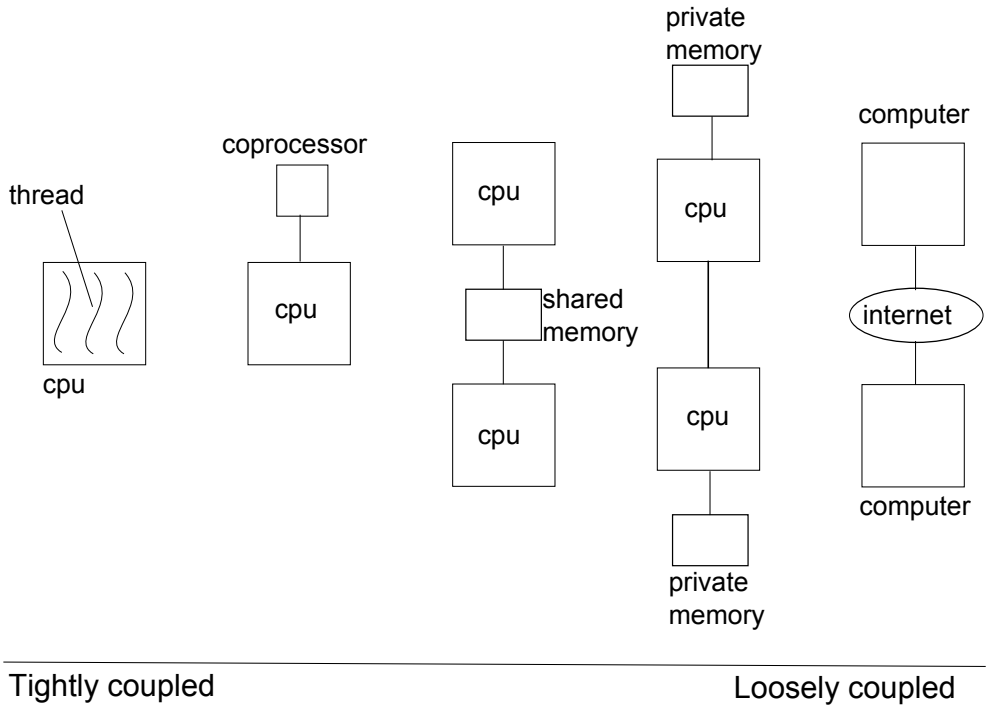
- Efficiency $E = \frac{\text{speed-up}}{\text{number of processors}}$
- **Strong Scalability**: If the efficiency is kept fixed while increasing the number of processes and maintainig fixed the problem size
- **Weak Scalability**: If the efficiency is kept fixed while increasing at the same rate the problem size and the number of processes

Superlinear Speedup

- Can we have a Speed-up $> p$? Yes!
  - Workload increases more than computing power ($G(p) > p$)
  - Cache effect: larger accumulated cache size. More or even all of the working set can fit into caches and the memory access time reduces dramatically
  - RAM effect: enables the dataset to move from disk into RAM drastically reducing the time required, e.g., to search it.
  - The parallel algorithm uses some search like a random walk: the more processors that are walking, the less distance has to be walked in total before you reach what you are looking for.

Parallel Architectures

Processor Coupling



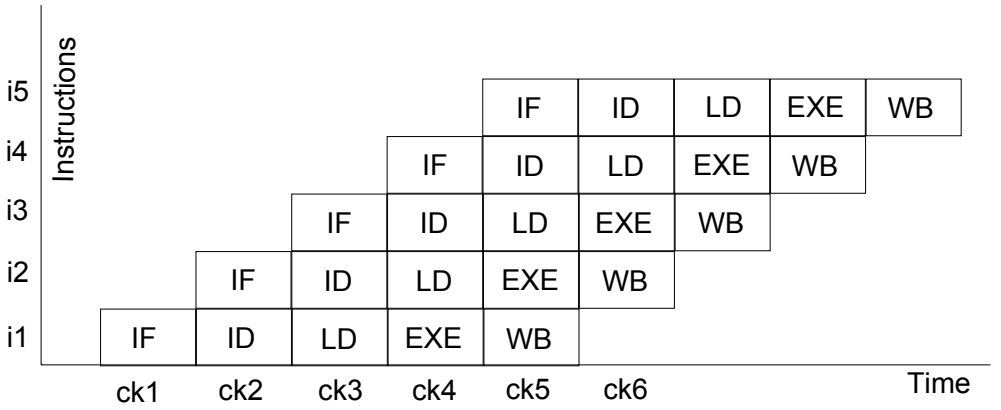
Classification of Parallelism

- **Instruction Level Parallelism**:
  each core can process more instructions per time unit
  - Pipelining
  - Superscalar

- **Thread/Process Level Parallelism**:
  workload is distributed over different threads/processes

Pipelining

- Instruction processing is split into different *stages*
- Each stage executes independently of each other

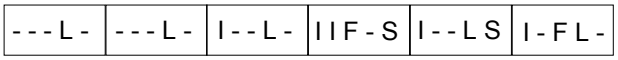


VLIW—Very Long Instruction Word

- An instruction represents multiple operations
- Each operation must operate on different hardware components
- This is done by (more complex!) compilers

Available Components:

- Integer Operations
- Integer Operations
- Floating Point Operation
- Load Operation
- Store Operation

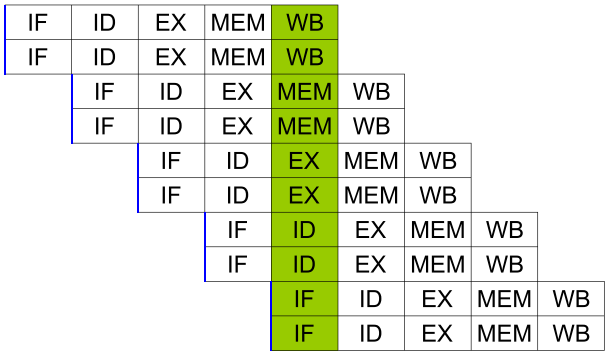


Sequence of VLIW

VLIW Stream boundled

Superscalar Architecture

- More instructions are simultaneously executed on the same CPU
- There are redundant functional units that can operate in parallel
- Run-time scheduling (in contrast to compile-time)
- It might require speculation (for branch prediction)

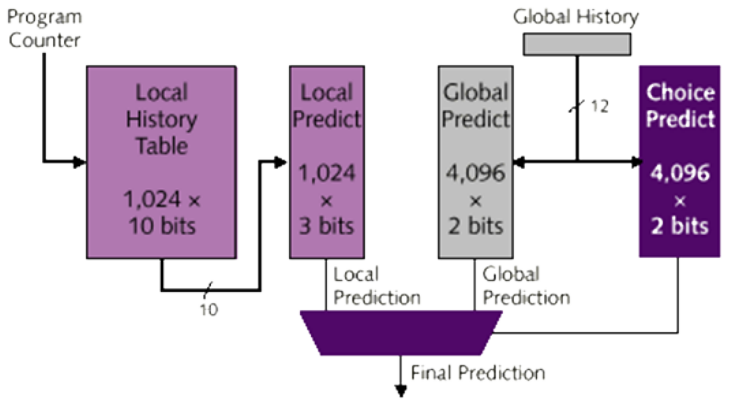


Speculation

- A guess on the outcome of a compare is made
  - if wrong the result is discarded
  - if right the result is flushed

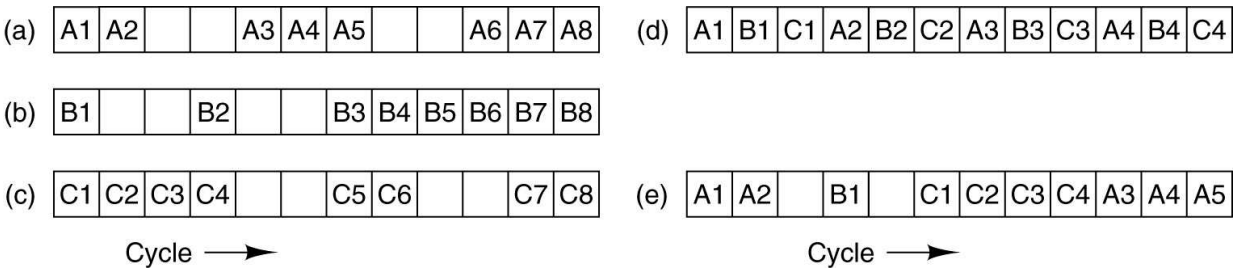
$a \leftarrow b + c$
if $a \geq 0$ **then**
    $d \leftarrow b$
else
    $d \leftarrow c$
end if

Speculation



- DEC Alpha 21264 Branch Prediction Unit
  - Tournament branch prediction algorithm
  - 35Kb of prediction information
  - 2% of total die size
  - Claim 0.7–1.0% misprediction

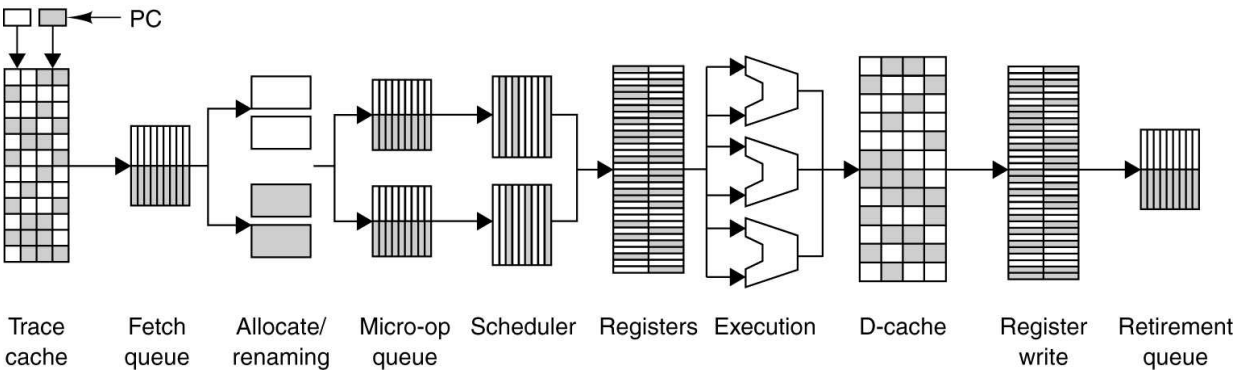
Hardware Thread-Level Parallelism



(a) (b) (c): Three threads. In empty boxes the thread has stalled on a memory access

- **Fine-grain thread**, switch thread at each clock cycle (d)
- **Coarse-grain thread**, switch thread upon slow operations (e)
- **Simultaneous Multithreading**, multiple threads can execute simultaneously (superscalar processors)

Hyperthreading



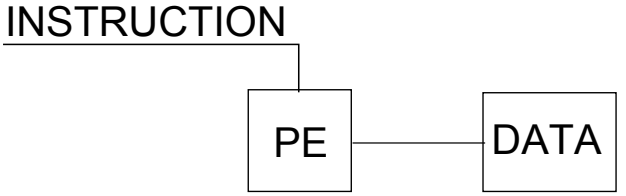
Taxonomy (Flynn 1966)

Flynn classification is based on the number of concurrent instructions (or controls) and data streams available in the architecture

- SISD: Single Instruction Stream, Single Data Stream
- SIMD: Single Instruction Stream, Multiple Data Stream
- MISD: Multiple Instruction Stream, Single Data Stream
- MIMD: Multiple Instruction Stream, Multiple Data Stream

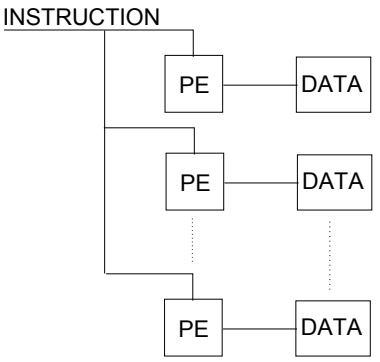
SISD—Single Instruction Stream, Single Data Stream

- One operation executed at a time on a single data item
- Classical Von Neumann architecture



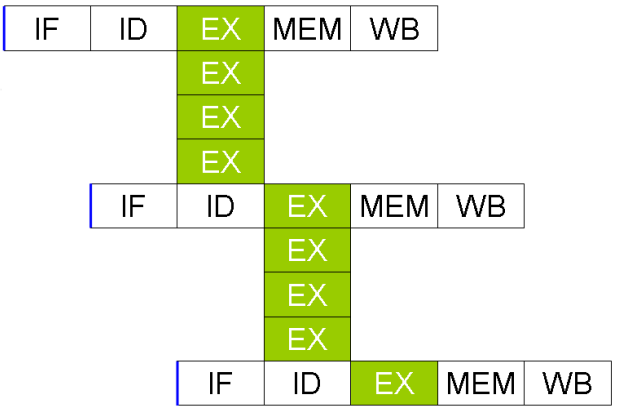
SIMD—Single Instruction Stream, Multiple Data Stream

- One operation executed on a set of data (e.g., matrix operations)
- Data-level Parallelism
- Synchronous operation
- Available also in commodity processors as well



Vector Processor (or Array Processor)

- Vector registers
- Vectorized and pipelined functional units
- Vector instructions
- Interleaved memory
- Strided memory access and hardware scatter/gather

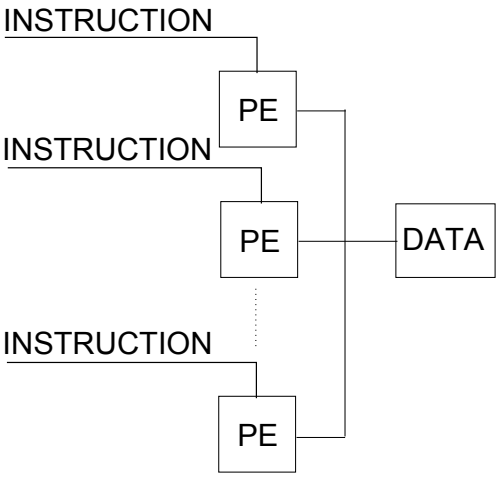


Graphics Processing Unit (GPU)

- Born specifically for graphics, then re-adapted for scientific computation
- The same operation is performed on a large set of objects (points, lines, triangles)
- Large number of ALUs ($\sim$100)
- Large number of registers ($\sim$32k)
- Large bandwidth

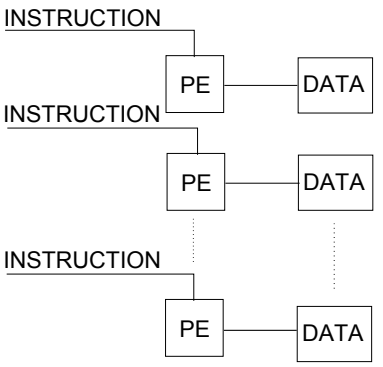
MISD—Multiple Instruction Stream, Single Data Stream

- More operations are executed at a time on a single data item
- Not of great interest, never became a real commercial product



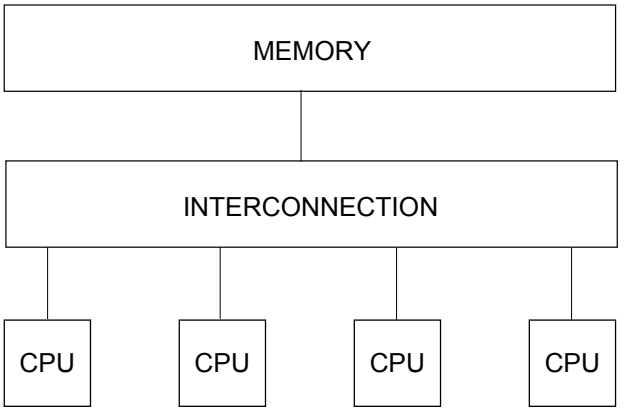
MIMD—Multiple Instruction Stream, Multiple Data Stream

- More operations executed at a time on a set of data
- The winning category for high performance computing
- Asynchronous
- MIMD machines are mainly:
  - Multiprocessors
  - Multicomputers



Multiprocessors

- Shared Memory
- Each processor can see the whole memory
- Communication supported by shared memory
- Synchronization is mandatory to consistently access memory

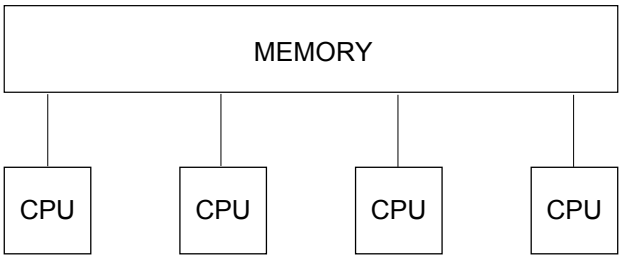


Multiprocessors Classification

- **UMA (Uniform Memory Access)**:
  Processors can access any memory area uniformly (e.g., same latency)
- **NUMA (Non-Uniform Memory Access)**:
  Processors have "nearer" memory areas which are accessed faster than others
  - **NC-NUMA** (Not-Caching NUMA)
  - **CC-NUMA** (Coherent Caches NUMA)
- **COMA (Cache-Only Memory Access)**:
  The local memories (typically DRAM) at each node are used as cache

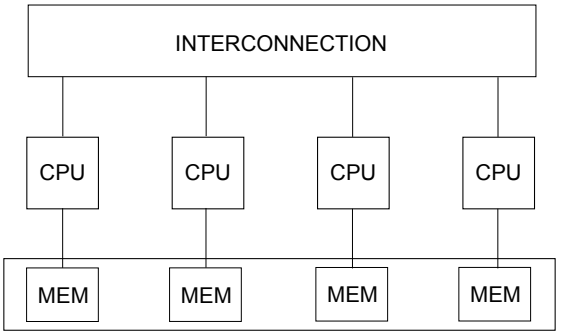
Uniform Memory Access

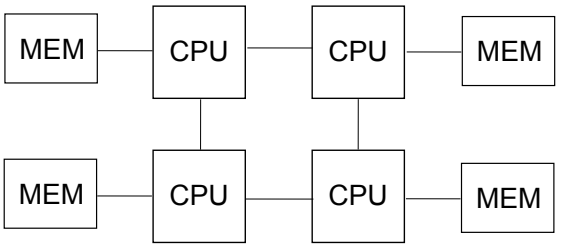
- Typical SMP (Symmetric Multiprocessing) memory access
- Each CPU can directly see the whole memory
- Tightly coupled system



Non-Uniform Memory Access

- Each CPU has its own *local* memory which is accessed faster
- Shared memory is the union of *local* memories
- The latency to access *remote* memory depends on the 'distance'





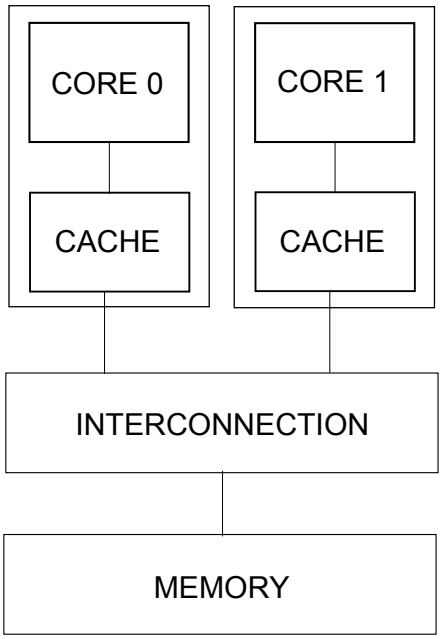
Cache Only Memory Access

- Each CPU has a private memory that operates as a cache
- The shared memory is the union of the caches
- A data element is owned by a single cache
- Data elements can migrate during the execution
- Search for a data element position is needed

Cache Coherency problem

time	Core 0	Core 1
0	$x \leftarrow 2$	$y \leftarrow 1$
1	$y \leftarrow 2 * x$	$h \leftarrow 0$
2	$a++$	$z \leftarrow y + h$

$$z = ?$$



Snooping Cache Coherency

- Same principle as of bus systems. Each device connected to the bus sees signals sent by others
- Upon a memory update the cache broadcasts update informations
- Other cores are snooping for updates
- Notifies have 'cache line' granularity

Directory-based Cache Coherence

- Snooping does not scale because of the broadcast cost in large networks
- (Distributed) directory structures store cache line information
- During a read, the directory is updated reflecting that the core owns a new copy
- A write operation makes the copy of each core invalid
- Only the owners of a copy are contacted

False Cache Sharing Problem

- Caches work at cache line granularity (typical values: 64B–256B)
- The problem arises whenever two cores access **different** data items that lie on the **same** cache line
- It produces an invalidation although accessed data items are different

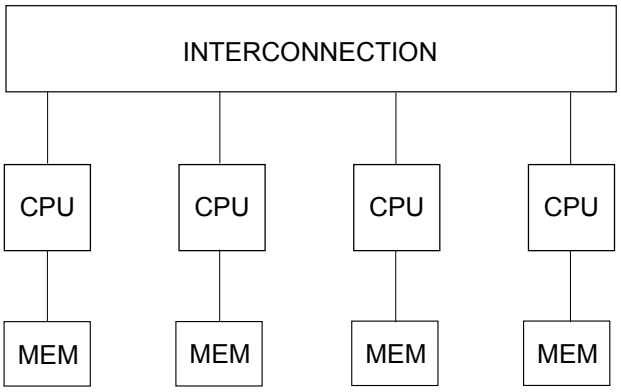
```
1 struct foo {
2     int x;
3     int y;
4 };
5
6 struct foo f;
```

```
1 void inc_x(void)
2 {
3     int i;
4     for(i = 0; i <
          1000000; i++)
5         f.x++;
6 }
```

```
1 int sum_y(void) {
2     int s = 0;
3     int i;
4     for (i = 0; i <
          1000000; i++)
5         s += f.y;
6     return s;
7 }
```

Multicomputers

- Distributed Memory
- Memories are private to single processors
- Communication is explicit through Message Passing
- Required support for send/receive primitives

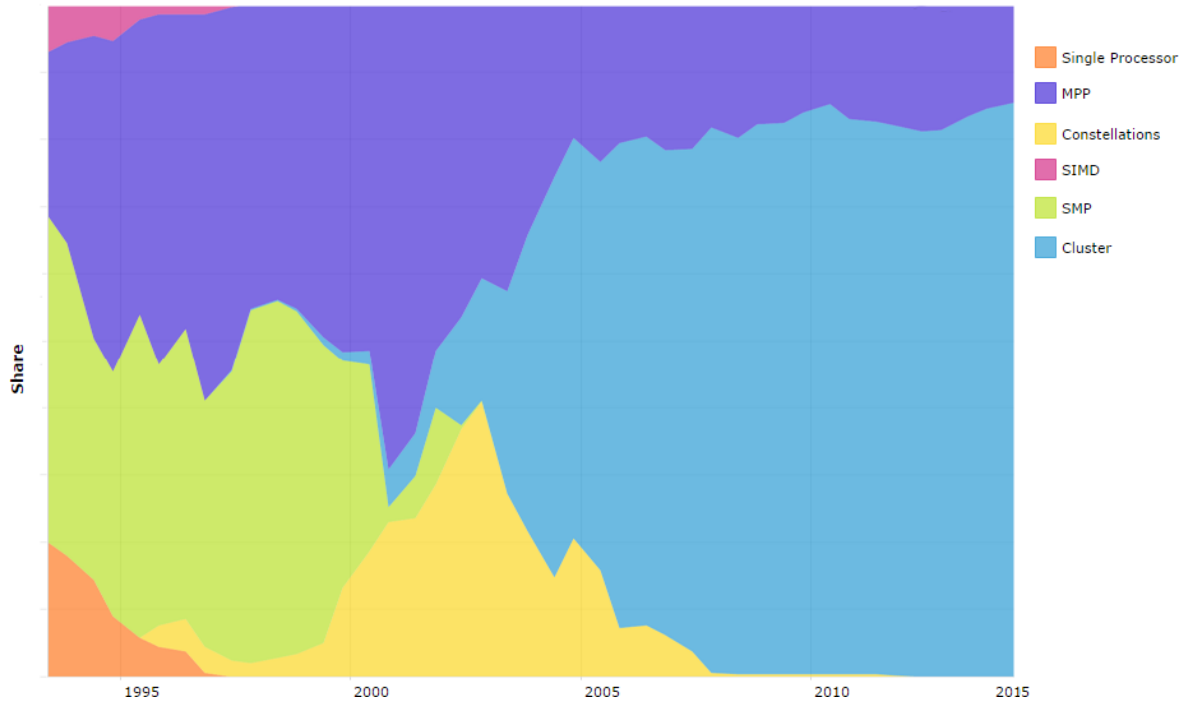


Multicomputer Classification

- **MPP (Massively Parallel Processors)**: A huge number of nodes connected by (in general) proprietary networks equipped with high bandwidth and low latency

- **Clusters** (also known as Cluster of Workstation (COW) or Network of Workstation (NOW)): Large number of commodity computers connected through a commodity network

Top 500—Architecture Types



Multicomputer Interconnections

Network performance

- **Latency**: Time to transmit the first single bit of a message in the worst case
- **Bandwidth**: Network Data-Rate from the receipt of the first bit
- **Hardware Complexity**: Implementation cost, Links, Switch connectors, Arbitration
- **Scalability**: Ability of the network to be expansible while maintaining the same performance

$$\text{Transmit Time} = \text{latency} + \text{size of the message} \,/\, \text{bandwidth}$$

Network Properties

Geometry:

- **Node Degree**: Number of edges coming into a node
- **Diameter**: Maximum shortest path between any two nodes
- **Bisection**: (Worst-case) number of links cut off to split the network into two equal halves
- **Bisection width**: The bandwidth of the channels cut off by the bisection
- **Regularity**: All nodes have the same degree
- **Symmetry**: Each node sees the same net

Network Properties

Routing:

- Static Networks:
  - Point-to-point networks
  - Paths do not change during the lifetime of the system
- Dynamic Networks:
  - Implemented through switched channels
    - Single-stage
    - Multi-stage
  - Paths can change during the system operation (e.g., depending on software)

Connectivity:

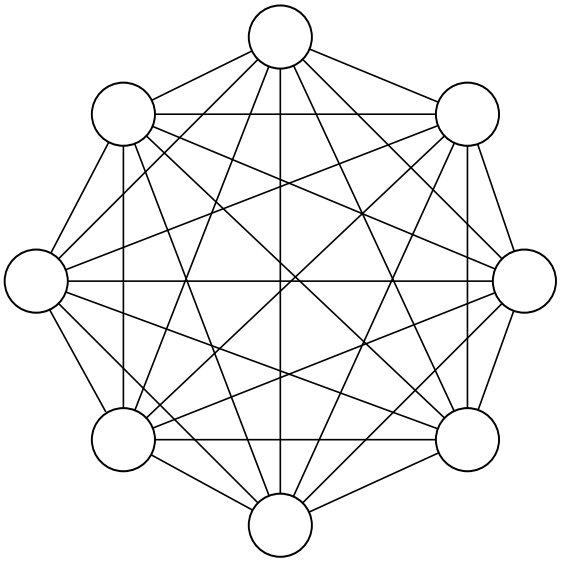
- Blocking Network:
  - Not any input/output combination is allowed
- Non-blocking Network:
  - All input/output combinations are allowed

Interconnections

- Shared Medium
  - All the nodes are directly connected to the medium
- Direct
  - Nodes are connected to each other through point-to-point links
  - The *message* is the information unit exchanged
- Indirect
  - Nodes are connected through a network of interconnections
  - The *message* is the information unit exchanged

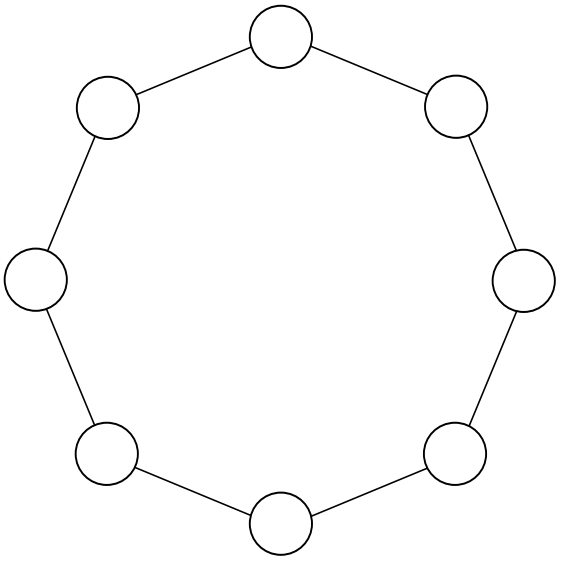
Fully Connected Mesh

- Each Node is connected to all the others
- Static
- Symmetric
- Node degree $= n - 1$
- Diameter $= 1$
- Number of links $= n(\frac{n-1}{2})$
- Bisection $= (\frac{n}{2})^2$



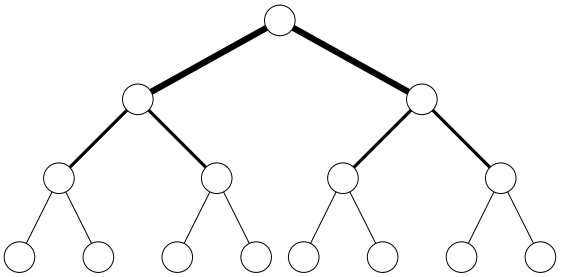
Ring

- Simple
- Static
- Symmetric
- Node degree $= 2$
- Diameter $= \lfloor \frac{n}{2} \rfloor$
- Number of links $= n$
- Bisection $= 2$



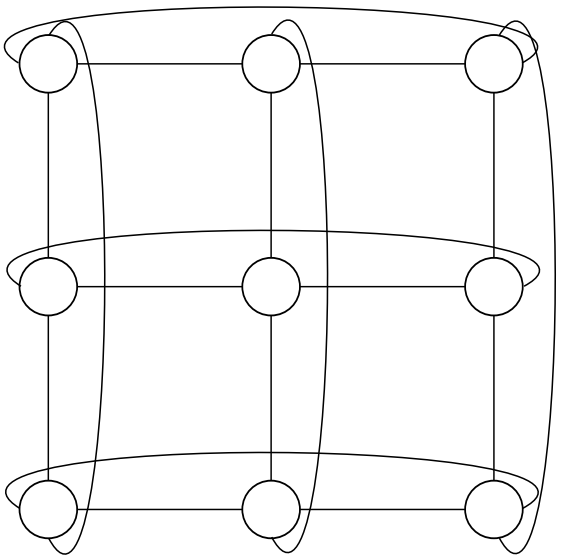
Fat-Tree

- Faces the problem of classical tree's root bottleneck
- Static
- Asymmetric
- Node degree $= 3$
- Diameter $= 2(\log n - 1)$
- Number of links $= n - 1$
- Bisection $= 1$



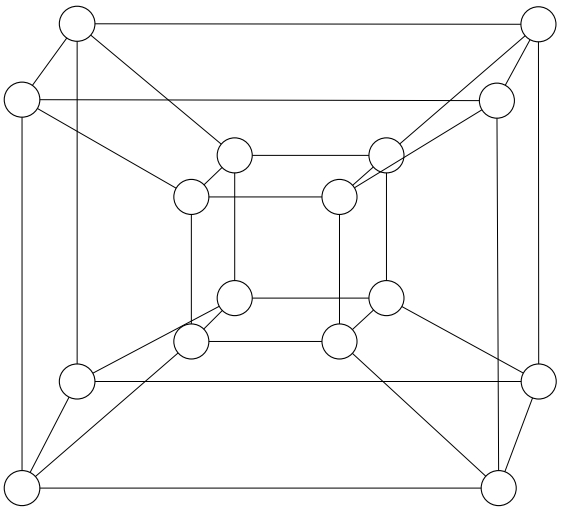
2D Torus

- Quite Simple
- Static
- Symmetric
- Node degree $= 4$
- Diameter $= 2\lfloor \frac{\sqrt{n}}{2} \rfloor$
- Number of links $= 2n$
- Bisection $= 2\sqrt{n}$



Hypercube

- d-sized Hypercube manages 2^d nodes
- Static
- Symmetric
- Node degree $= d$
- Diameter $= d + 1$
- Number of links $= d\frac{n}{2}$
- Bisection $= \frac{n}{2}$

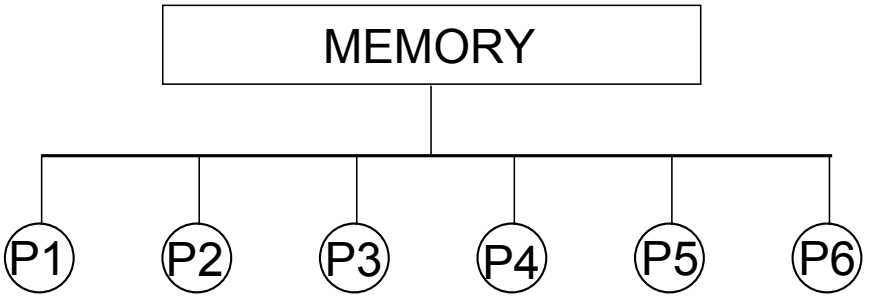


Static Networks Summary

	Fully C	Ring	Fat-tree	2D-torus	Hypercube
Node degree	n - 1	2	3	4	d
Diameter	1	$\lfloor \frac{n}{2} \rfloor$	$2(\log n - 1)$	$2\lfloor \frac{\sqrt{n}}{2} \rfloor$	d + 1
Links	$n(\frac{n-1}{2})$	n	n - 1	2n	$d\frac{n}{2}$
Bisection	$(\frac{n}{2})^2$	2	1	$2\sqrt{n}$	$\frac{n}{2}$
Simmetry	Y	Y	N	Y	Y

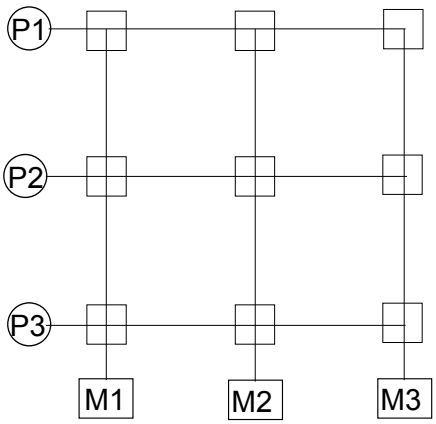
Bus

- Dynamic
- All devices are linked to the same wires
- Presence of arbitration
- Increasing the number of connected devices:
  - Increases contention
  - Decreases performance



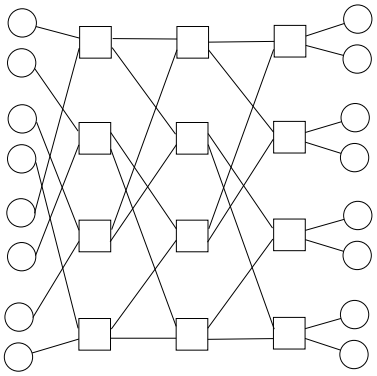
Crossbar

- Dynamic
- Matrix with switching points
  - Number of switches = pm
- There is contention only if two cores access the same memory bank



Omega

- Dynamic
- Multistage Intectonnection Network (MIN)
- Each connection is a 2x2 crossbar
- Relies on perfect shuffle interconnection algorithm
- Some communications cannot happen simultaneously



Dynamic Networks Summary

	Bus	**MultiStage**	**Crossbar**
Minumum Latency	Costant	$O(log_k n)$	costant
CPU Bandwidth	$O(\frac{w}{n})$ to $O(w)$	$O(w)$ to $O(nw)$	$O(w)$ to $O(nw)$
Wiring Complexity	$O(w)$	$O(nw \log_k n)$	$O(n^2 w)$
Switching Complexity	$O(n)$	$O(n \log_k n)$	$O(n^2)$
Connectivity and routing capability	Only one to one at time	Some permutations and broadcast if network is unblocked	All permutation, one at time

Scalable Manycore Architecture

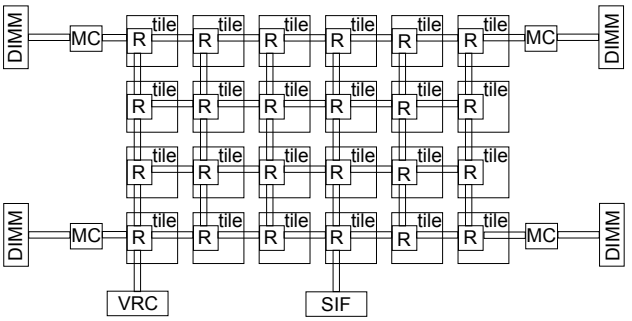
Intel^{TM}48 Cores
Single Chip Cloud Computing

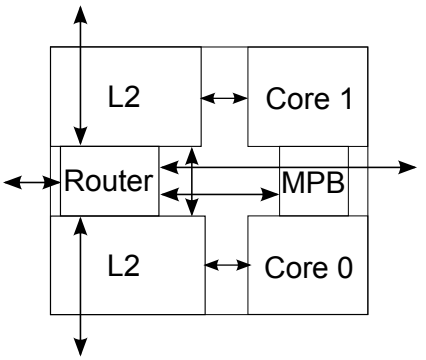
Single-chip Cloud Computing—SCC

- 48 Intel cores on a single die
- Power 125W cores @ 1GHz, Mesh @ 2Ghz
- Message Passing Architecture
- No coherent shared memory
- Proof of Concept of a scalable many-core solution

Tiles and Cores

- 24 Tiles with 2 Intel cores each
- 32 KB (16KB I + 16KB D) L1 Cache per Core (inside each core)
- 256 KB L2 Cache per Core (inside each tile)
- Each tile has a Message Passing Buffer (MPB)
- Each tile has a Router



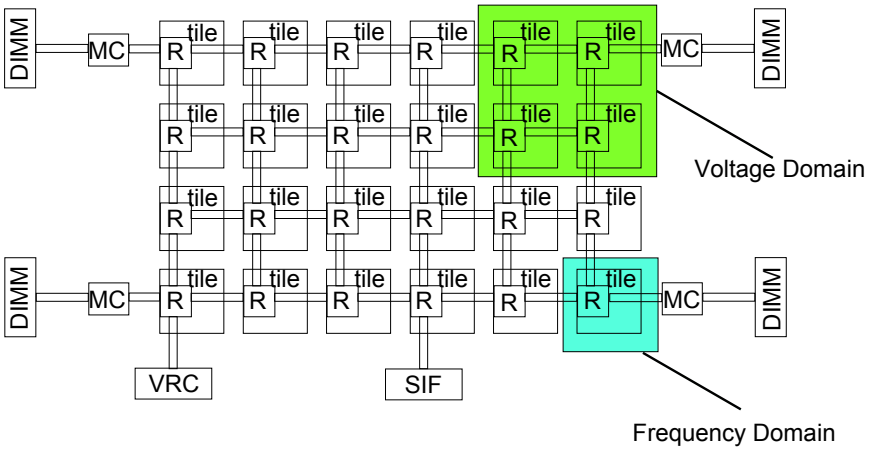


Mesh Interface Unit—MIU

- 2D mesh (6 x 4)
- 2Tb/s Bisection bandwidth @ 2Ghz
- Connects the tile to the mesh
- Packs/Unpacks data in the tile
- Round-robin to arbitrate in-tile cores

Power Management

- VRC (voltage regulator controller) is the SCC power controller
- It can change voltage and frequency of cores, as well as of other parts of the die, upon request
- 7 voltage domain
- 24 frequency domain



Message Passing Buffer—MPB

- 16 KB buffer per tile
- 384 KB in the SCC
- Shared among all the cores on the chip
- Low latency Message Passing Support (through shared memory)

Lookup Table—LUT

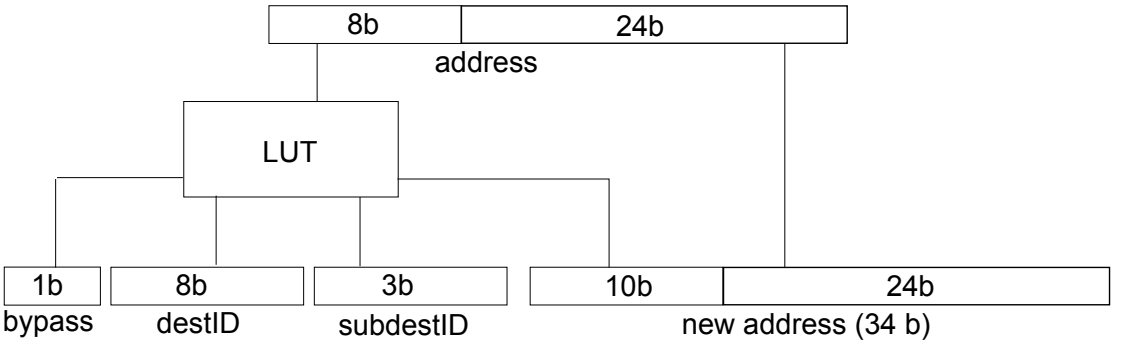
- Each core has a *core address* (up to 4 GB)
- The SCC has a *system space* (up to 64GB)
- MIU uses the LUT to translate the core address to a system address
- Each core has its own LUT
- 256 entries for 4 GB addressable (each entry addressed 16 MB)
- An address request is placed in one of these queues:
  - Router queue $\rightarrow$ memory controller $\rightarrow$ DDR3 memory
  - MPB queue
  - A queue to access local configuration registers

SCC Memory

- On-chip SRAM (used for MPB)
- Off-chip DDR3 DRAM (accessed through 4 on-die memory controller)
- Off-chip divided in core-private and shared memory
- The division is determined by LUT and configurable at boot time, or dynamically after boot
- Tiles are organized in 4 regions, each one mapped to a different memory controller (when accessing private off-chip memory)
- When accessing shared off-chip memory messages might pass through each controller

Address Translation

- **bypass bit:** if 1 the address is local to the tile's MPB
- **destID:** identifies the tile containing the router (x, y coords)
- **subdestId:** specifies the port in the router
- DDR Memory controllers adjacent to some exterior tiles



Address Translation table

Sub Dest	`subdestID`	Description
Core0	0x0	Core 0 in the tile
Core1	0x1	Core 1 in the tile
CRB	0x2	Configuration Register
MPB	0x3	Message Passing Buffer
E-port	0x4	(0,5) and (2,5) select DDR3 MC
S-port	0x5	(0,3) System interface (SIF), (0,0) VRC
W-port	0x6	(0,0) and (2,0) select DDR3 MC
N-port	0x7	Nothing on this port of any edge router

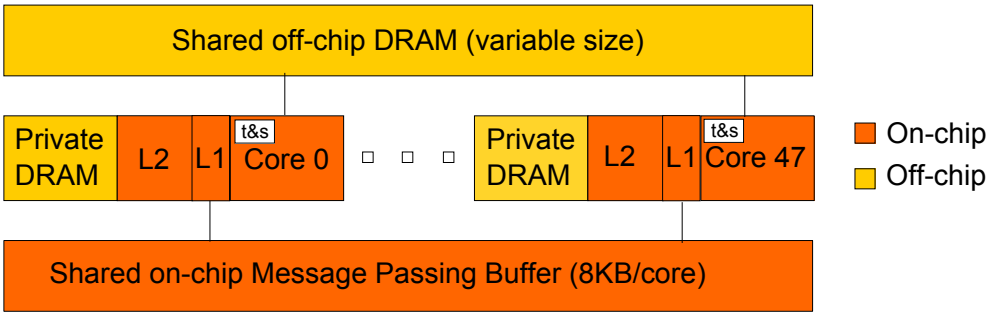
Cache Behaviour

- Core private Off-chip DRAM is cached through L1 and L2 accordind to classical rules
- No cache coherence among the cores
- Coherence is left to software:
  - RCCE Library takes care of it
  - Special tag and Instruction for developers customized solutions
    - **MPBT** (Message Passing Buffer Type) tag identifies a cache line for a data in the shared memory region
    - **CL1INVMB** instruction marks all MPBT type data as invalid L1 lines
    - Accessing an invalid data forces the update of the data in the L1

How RCCE deals with cache incoherence

- MPB is of type MPBT
- If data is changed by another core and image is still in L1?
  - Invalidate before read
- If the image of a line to write to is already in L1 (thus not to be flushed in memory)?
  - Invalidate before write
- L1 has write-combining buffer
  - Always push whole cache lines
- MPB should be used only for data movement

Programmer's View of Memory



SIMD Architectures for Supercomputing

NVIDIA^{TM} FERMI

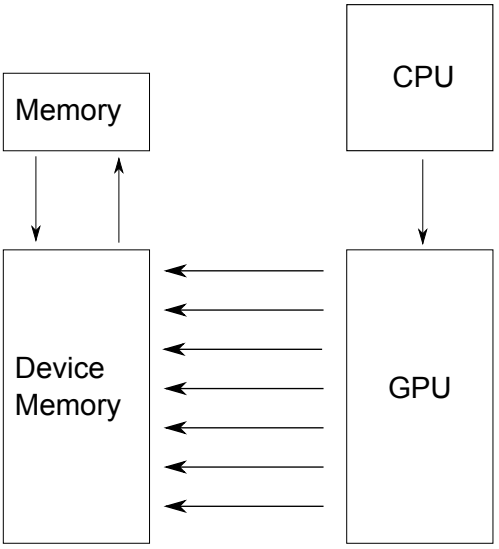
GP-GPU

- GPGPU is the usage of a GPU to perform general-purpose scientific computation
- The model for GPU computing is to use a CPU and GPU(s) together in a heterogeneous co-processing computing architecture
- The serial part of the program is left to the CPU, the parallel one is deployed on the GPU

From GPGPU to GPU Computing Model

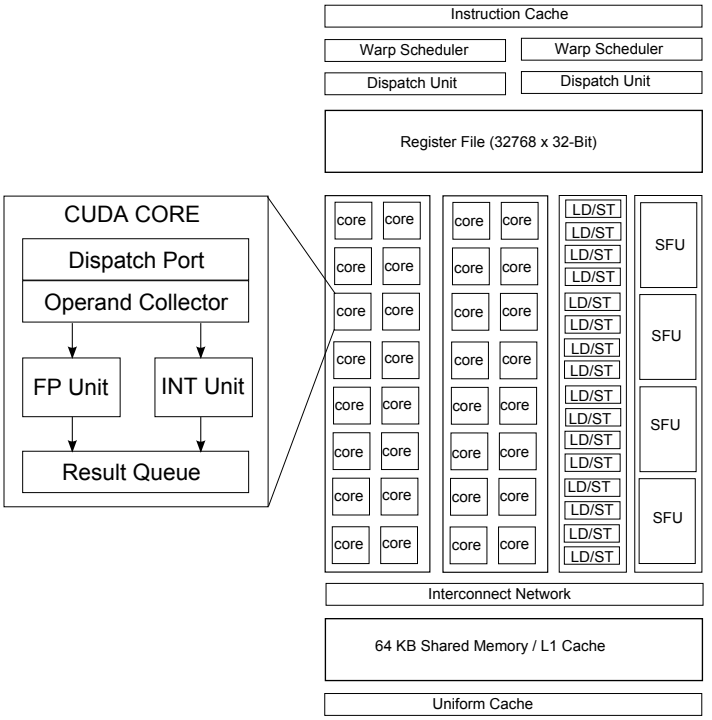
- Classic GPUs had a reduced programmability, oriented only to graphic computation
- All problems to be solved must be mapped to graphic ones
- CUDA (Compute Unified Device Architecture):
  - Straight support to C and Fortran languages
  - SIMT (Single Istruction Multiple Thread) Execution model
  - Shared memory and barrier synchronization
  - ...

GPGPU—GPU Computing Programming model



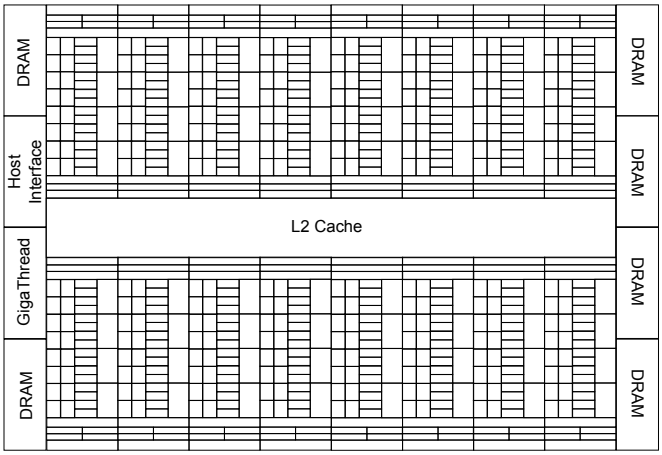
Fermi Streaming Multiprocessor

- Each SM can take up to 1536 *CUDA threads*
- Each SM can take up to 8 *CUDA blocks*



Fermi Architecture

- Up to 512 CUDA cores
- 16 SMs, 32 cores each
- A FP or Integer Istruction per clock per thread



Cuda Programming model

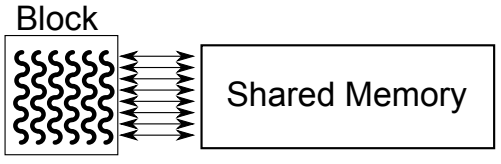
- CUDA is a hardware/software architecture used to provide general programming model support
- Data-Parallel portions are executed on the GPUs as *kernels*
- A kernel executes in parallel across a set of parallel threads
- CUDA threads are extremely lightweight
- The programmer (or the compiler) organizes these threads in thread blocks and grids of thread blocks

CUDA Thread

- Executed by one scalar processor
- Has its own Program counter
- Has its own Registers
- Has a Private memory
- Inputs data and output results
- Identified by a thread ID within a thread block

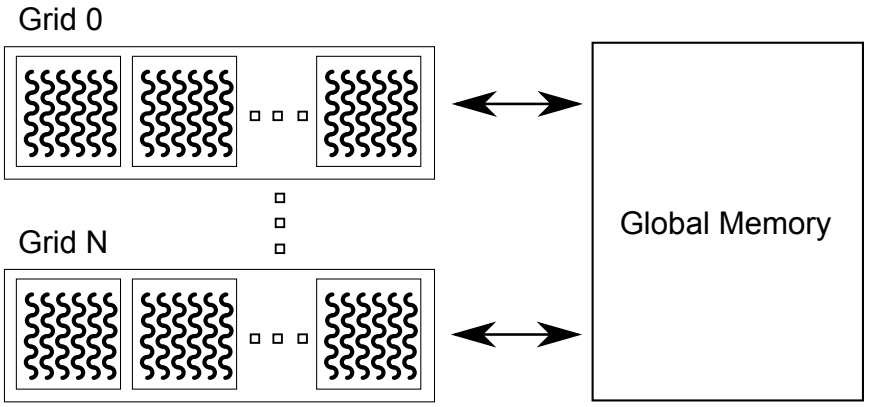
CUDA Block

- Is a set of threads that share:
  - (Shared) Memory
  - Synchronization Barriers
- Up to 1024 threads
- Executed on a Multiprocessor
- Cannot migrate across Multiprocessors
- Several concurrent thread blocks can reside on a multiprocessor
- Identified by a block ID within a thread Grid



CUDA Grid

- Is an array of blocks that execute the same kernel:
  - Shared Global Memory
  - Synchronization between dependent kernel calls
- Kernels are launched as grids of threads



CUDA Hardware Mapping

- A GPU can execute one or more kernel grids
  - A kernel can execute as a 1D, 2D or 3D grid of thread blocks
- Each SM executes one or more thread blocks
  - A thread block is a 1D, 2D or 3D batch of threads
- CUDA cores and other execution units in the SM execute threads

How to Manage Grids and Blocks

- Cuda defines dim3 datatype

```
struct dim3 {
  unsigned int x, y, z;
};
```

- **gridDim:** dimension of the Grid
- **blockDim:** dimension of the Block
- **blockIdx:** block index
- **threadIdx:** thread index

GigaThread Scheduler

- At chip level thread blocks are scheduled onto various SMs
- At SM level The code is actually executed in groups of 32 threads, what NVIDIA calls a *warp*
  - Threads in a warp execute the same instructions at the same time
  - Each SM can issue and execute 2 warps concurrently
    - Managing warps faces performance not correctness
    - One instruction per warp is executed
    - No contention from two different warps

Fermi Register File

- Up to 32768 32-bit register in each SM
- Dynamically partitioned among all the blocks assigned to the SM
- Once assigned to a block, it is not accessible by threads in other blocks
- Each thread in a block can only access registers assigned to it

Memory Hierarchy

- Local Memory—Thread Scope
  - Private to threads
- Shared Memory—Block Scope
  - Shared among threads within the same block
  - Extremely fast, on-chip memory
  - Not visible to threads in other blocks running concurrently
- Global—Application Scope
  - Shared by all threads
  - Allows inter-grid communication

Considerations on Blocks

Example: Matrix multiplication via multiple blocks.

What is the best-suited size?

- 8x8 = 64 threads per block
  Each SM can run at most 8 blocks
  Only 256 threads will be run by the same SM
- 32x32 = 1024 threads per block
  Each SM can run at most 1536 threads
  Only 1 block can fit the SM
- 16x16 = 256 threads per block
  With 6 blocks, SM works at its best

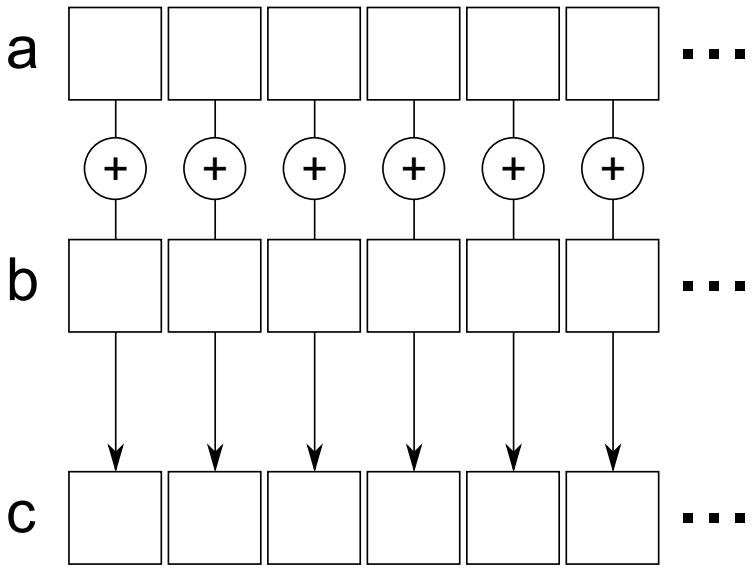
Considerations on Registers

- How many threads can run on each SM with a block of 16x16 threads on register constraints?
  - Each thread requires 30 registers
    Each block requires 30x256 = 7680 registers
    32768/7680 = 4 + remainder
    Each SM can run 4 blocks
  - Each thread requires 33 registers (+10%)
    Each block require 33x256 = 8448 registers
    32768/8448 = 3 + remainder
    Each SM can run 3 blocks (-25% parallelism)

Programming with CUDA (at a glance)

- Computation partitioning
  - Functions are declared as: `__host__`, `__global__` and `__device__`
  - Mapping of thread programs to device: `compute<<<gs,bs>>>(<args>)`
- Data partitioning
  - Data are declared as: `__shared__`, `__device__`, `__constant__`
- Data management and orchestration
  - Functions to copy data from/to Host
- Concurrency management
  - Functions like: `__synchthreads()`

Programming with CUDA: Summing two vectors



Programming with CUDA: Summing two vectors

```
1  #define N 10
2  int main(void) {
3      int a[N], b[N], c[N];
4      int *dev_a, *dev_b, *dev_c;
5
6      // allocate the memory on the GPU
7      cudaMalloc((void**)&dev_a, N * sizeof(int));
8      cudaMalloc((void**)&dev_b, N * sizeof(int));
9      cudaMalloc((void**)&dev_c, N * sizeof(int));
10
11     // fill the arrays 'a' and 'b' on the CPU
12     for (int i=0; i<N; i++) {
13         a[i] = -i;
14         b[i] = i * i;
15     }
```


Programming with CUDA: Summing two vectors (2)

```
16
17     // copy the arrays 'a' and 'b' to the GPU
18     cudaMemcpy(dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice);
19     cudaMemcpy(dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice);
20
21     // Perform addition on the GPU
22     add<<<N,1>>>(dev_a, dev_b, dev_c);
23
24     // copy the array 'c' back from the GPU to the CPU
25     cudaMemcpy(c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost);
26
27     // display the results
28     for(int i = 0; i < N; i++) {
29         printf("%d + %d = %d\n", a[i], b[i], c[i]);
30     }
31
```

Programming with CUDA: Summing two vectors (3)

```
32      // free the memory allocated on the GPU
33      cudaFree(dev_a);
34      cudaFree(dev_b);
35      cudaFree(dev_c);
36
37      return 0;
38  }
39
40  __global__ void add(int *a, int *b, int *c) {
41      int tid = blockIdx.x; // handle the data at this index
42      if(tid < N)
43          c[tid] = a[tid] + b[tid];
44  }
```


Vector Sum as seen by GPU

Block 0:

```
1  __global__ void add(int *a, int *
       b, int *c) {
2      int tid = 0;
3      if (tid < N)
4          c[tid] = a[tid] + b[tid];
5  }
```

Block 1:

```
1  __global__ void add(int *a, int *
       b, int *c) {
2      int tid = 1;
3      if (tid < N)
4          c[tid] = a[tid] + b[tid];
5  }
```

Block 2:

```
1  __global__ void add(int *a, int *
       b, int *c) {
2      int tid = 2;
3      if (tid < N)
4          c[tid] = a[tid] + b[tid];
5  }
```

Block 3:

```
1  __global__ void add(int *a, int *
       b, int *c) {
2      int tid = 3;
3      if (tid < N)
4          c[tid] = a[tid] + b[tid];
5  }
```

Bibliography

- Amdahl, G. M.: *Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities*. In AFIPS Conference Proceedings. (1967)
- Gustafson, J. L.: *Reevaluating Amdahl's Law*. Communications of the ACM. (1988)
- Sun, X.-H., Ni, L.: *Scalable Problems and Memory-Bounded Speedup*. Journal of Parallel and Distributed Computing. (1993)
- Kai Hwang - *Advanced Computer Architecture Parallelism, Scalability, Programmability*
- Intel *SCC Platform Overview*
- NVIDIA *NVIDIA Fermi Architecture Whitepaper*