Parallel Processors: Parallel Programming, Multithreading, GPU

Dr. Vincent C. Emeakaroha

27-03-2017

vc.emeakaroha@cs.ucc.ie

Parallelism We've Already Covered

- §2.11: Parallelism and Instructions
  - Synchronization

- §3.6: Parallelism and Computer Arithmetic
  - Subword Parallelism

- §4.10: Parallelism and Advanced Instruction-Level Parallelism

- §5.10: Parallelism and Memory Hierarchies
  - Cache Coherence

Parallel Programming

- Developing parallel software is the problem

- Need to get significant performance improvement
  - Otherwise, just use a faster uniprocessor, since it's easier!

- Difficulties
  - Partitioning
  - Coordination
  - Communications overhead

Amdahl's Law

- Sequential part can limit speedup
- Example: To achieve 90× speedup with 100 processors, what application percentage could be sequential?
    - $T_{new} = T_{parallelizable}/100 + T_{sequential}$
    - $$Speedup = \frac{1}{(1-F_{parallelizable}) + F_{parallelizable}/100} = 90$$
    - Solving: $F_{parallelizable} = 0.999$
- Need sequential part to be 0.1% of original application

Instruction and Data Streams

- An alternate classification based on number of instructions and data streams

		Data Streams	
		Single	Multiple
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Intel Xeon e5345

- # SPMD: Single Program Multiple Data
  - A parallel program on a MIMD computer
  - Conditional code for different processors

Single Instruction Multiple Data (SIMD)

- Operate elementwise on vectors of data
  - E.g., MMX and SSE instructions in x86
    - Multiple data elements in 128-bit wide registers
- All processors execute the same instruction at the same time
  - Each with different data address, etc.
- Simplifies synchronization
- Reduced instruction control hardware
- Works best for highly data-parallel applications
  - Loops in programs

Vector Processors

- Highly pipelined function units
  - An elegant interpretation of SIMD
- Stream data from/to vector registers to units
  - Data collected from memory into registers
  - Results stored from registers to memory
- Example: Vector extension to MIPS
  - 32 × 64-element registers (64-bit elements)
  - Vector instructions
    - `lv, sv`: load/store vector
    - `addv.d`: add vectors of double
    - `addvs.d`: add scalar to each element of vector of double
- Significantly reduces instruction-fetch bandwidth

Vector vs. Scalar

- Vector architectures and compilers
  - Single vector instruction is equivalent to executing entire scalar loop
  - Explicit statement of absence of loop-carried dependences
    - Reduced checking in hardware
  - Regular access patterns benefit from interleaved and burst memory
  - Avoid control hazards by avoiding loops
- Better power and energy savings than scalar architecture
  - Less instruction and memory bandwidths and less hazard checking
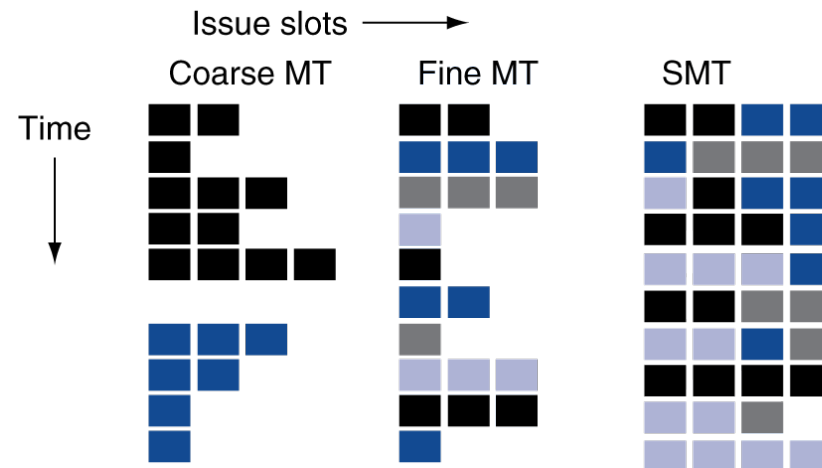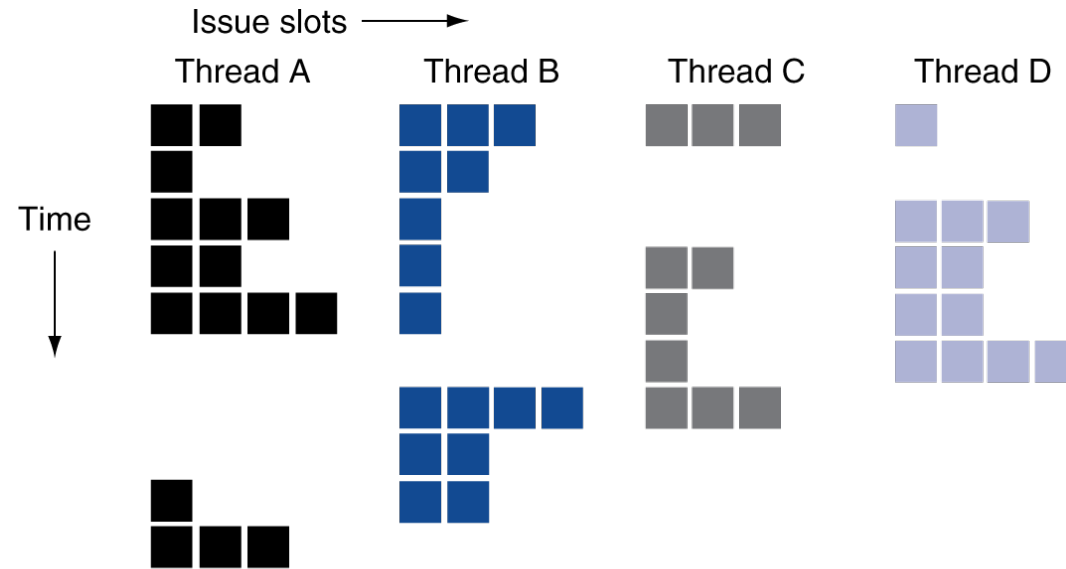
Multithreading

- Performing multiple threads of execution in parallel for a single processor
  - Replicate registers, PC, etc.
  - Fast switching between threads as compared to processes
  - Uses virtual memory mechanism to share memory
- Fine-grain multithreading
  - Switch threads after each instruction
  - Interleave instruction execution
  - If one thread stalls, others are executed
- Coarse-grain multithreading
  - Only switch on long stall (e.g., L2-cache miss)
  - Simplifies hardware, but doesn't hide short stalls (eg, data hazards)

Simultaneous Multithreading (SMT)

- In multiple-issue dynamically scheduled processor
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when function units are available
  - Within threads, dependencies handled by scheduling and register renaming
- Example: Intel Pentium-4 HT
  - Two threads: duplicated registers, shared function units and caches

# Multithreading Example

Issue slots ⟶

Future of Multithreading

- Will it survive? In what form?
- Power considerations $\Rightarrow$ simplified microarchitectures
  - Simpler forms of multithreading
- Tolerating cache-miss latency
  - Thread switch may be most effective
- Multiple simple cores might share resources more effectively

Multiprocessor Shared Memory

- SMP: shared memory multiprocessor
  - Hardware provides single physical address space for all processors
  - Synchronize shared variables using locks and cache coherence
- Two styles of SMP memory access time
  - Uniform memory access: latency does not depend on processor
  - Non-uniform memory access: variable access time depending processor
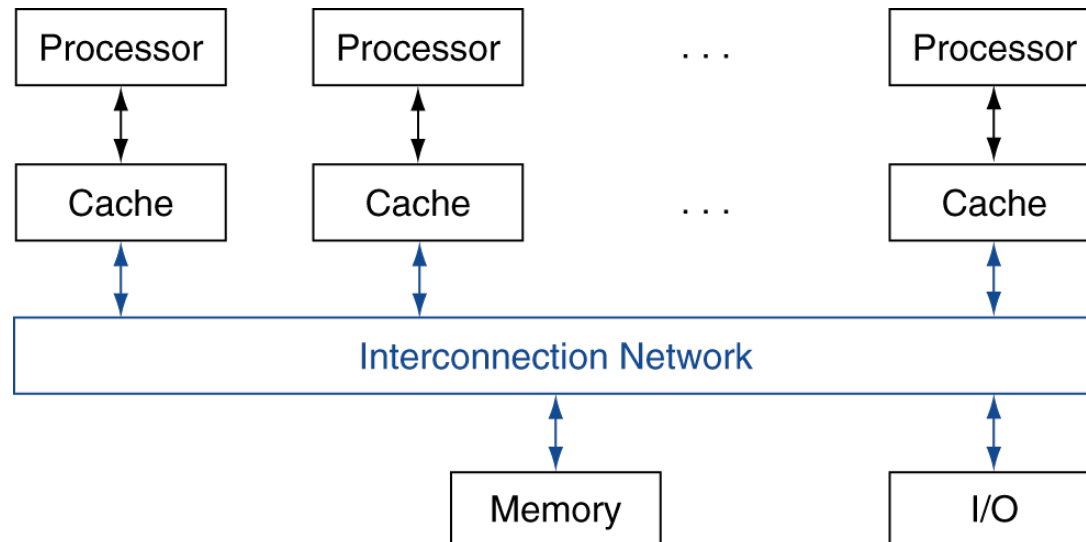
Example: Sum Reduction

- Sum 100,000 numbers on 100 processor UMA
  - Each processor has ID: $0 \le Pn \le 99$
  - Partition 1000 numbers per processor
  - Initial summation on each processor

```
sum[Pn] = 0;
  for (i = 1000*Pn;
     i < 1000*(Pn+1); i = i + 1)
    sum[Pn] = sum[Pn] + A[i];
```

- Now need to add these partial sums
  - Reduction: divide and conquer
  - Half the processors add pairs, then quarter, …
  - Need to synchronize between reduction steps

Example: Sum Reduction


```
half = 100;
repeat
  synch();
  if (half%2 != 0 && Pn == 0)
    sum[0] = sum[0] + sum[half-1];
    /* Conditional sum needed when half is odd;
       Processor0 gets missing element */
  half = half/2; /* dividing line on who sums */
  if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);
```
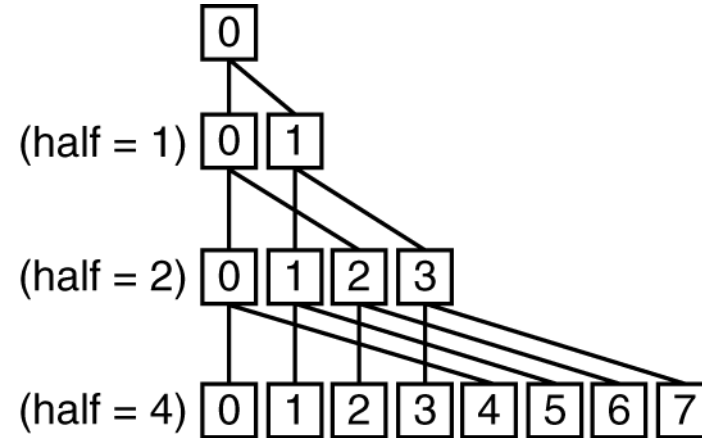
History of GPUs

- Early video cards
  - Frame buffer memory with address generation for video output

- 3D graphics processing
  - Originally high-end computers (e.g., SGI)
  - Moore's Law $\Rightarrow$ lower cost, higher density
  - 3D graphics cards for PCs and game consoles

- Graphics Processing Units
  - Accelerators that supplement CPU
  - Graphics and game community
 - Different goals than the microprocessor community

Graphics in the System
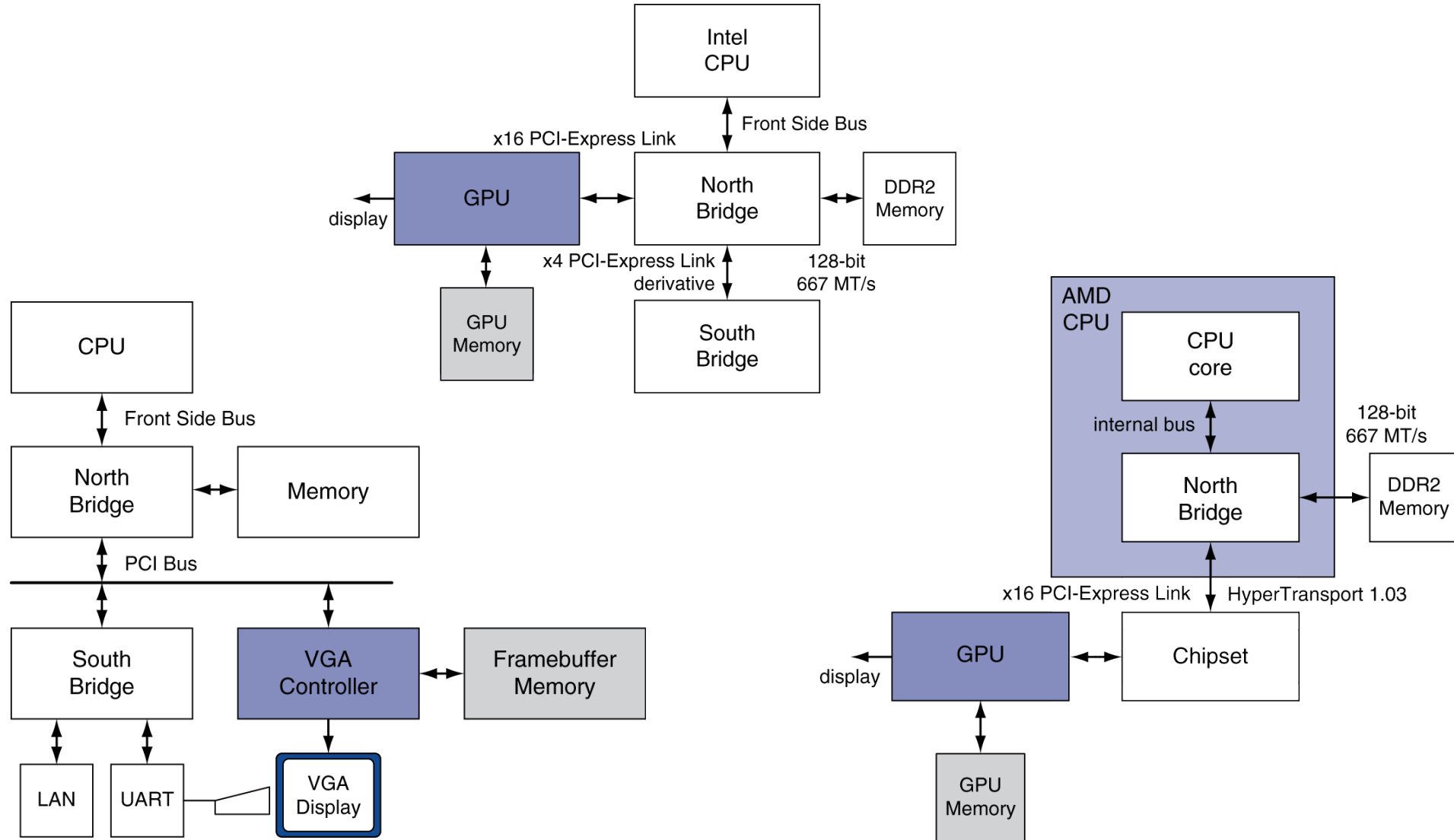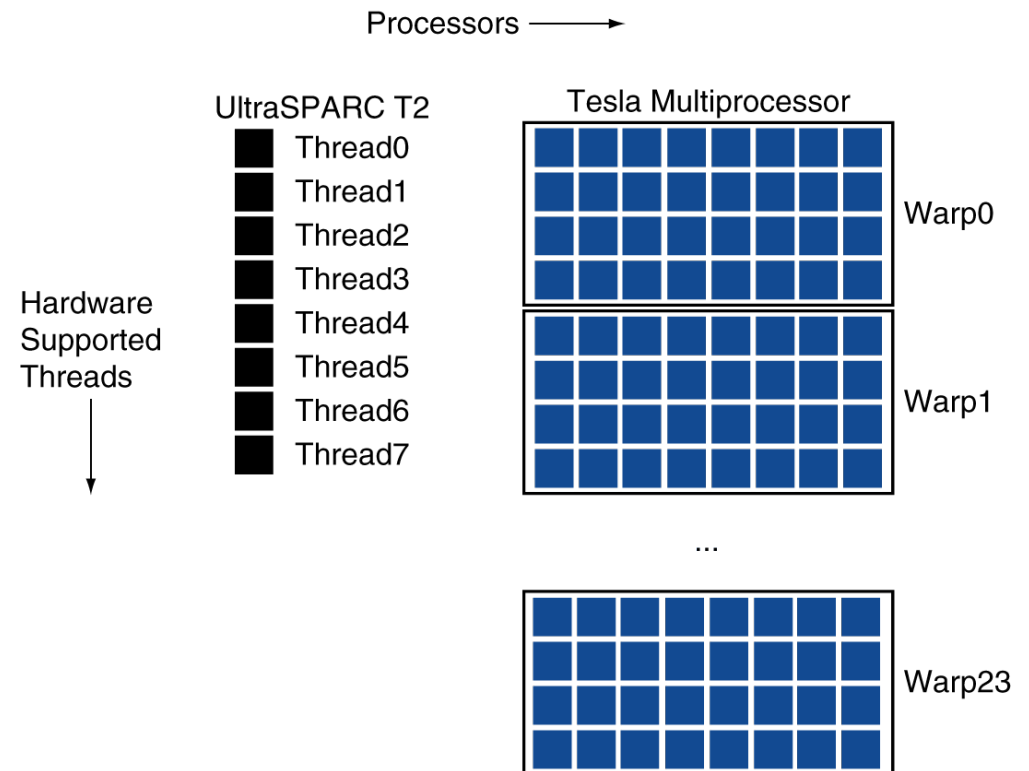
GPU Architectures

- Processing is highly data-parallel
  - GPUs are highly multithreaded
  - Use thread switching to hide memory latency
    - Less reliance on multi-level caches
    - Hardware multithreading
  - Graphics memory is wide and high-bandwidth
- Trend toward general purpose GPUs
  - Heterogeneous CPU/GPU systems
  - CPU for sequential code, GPU for parallel code
- Programming languages/APIs
  - DirectX, OpenGL
  - C for Graphics (Cg), High Level Shader Language (HLSL)
  - Compute Unified Device Architecture (CUDA)

Example: NVIDIA Tesla

Streaming multiprocessor

8 × Streaming processors
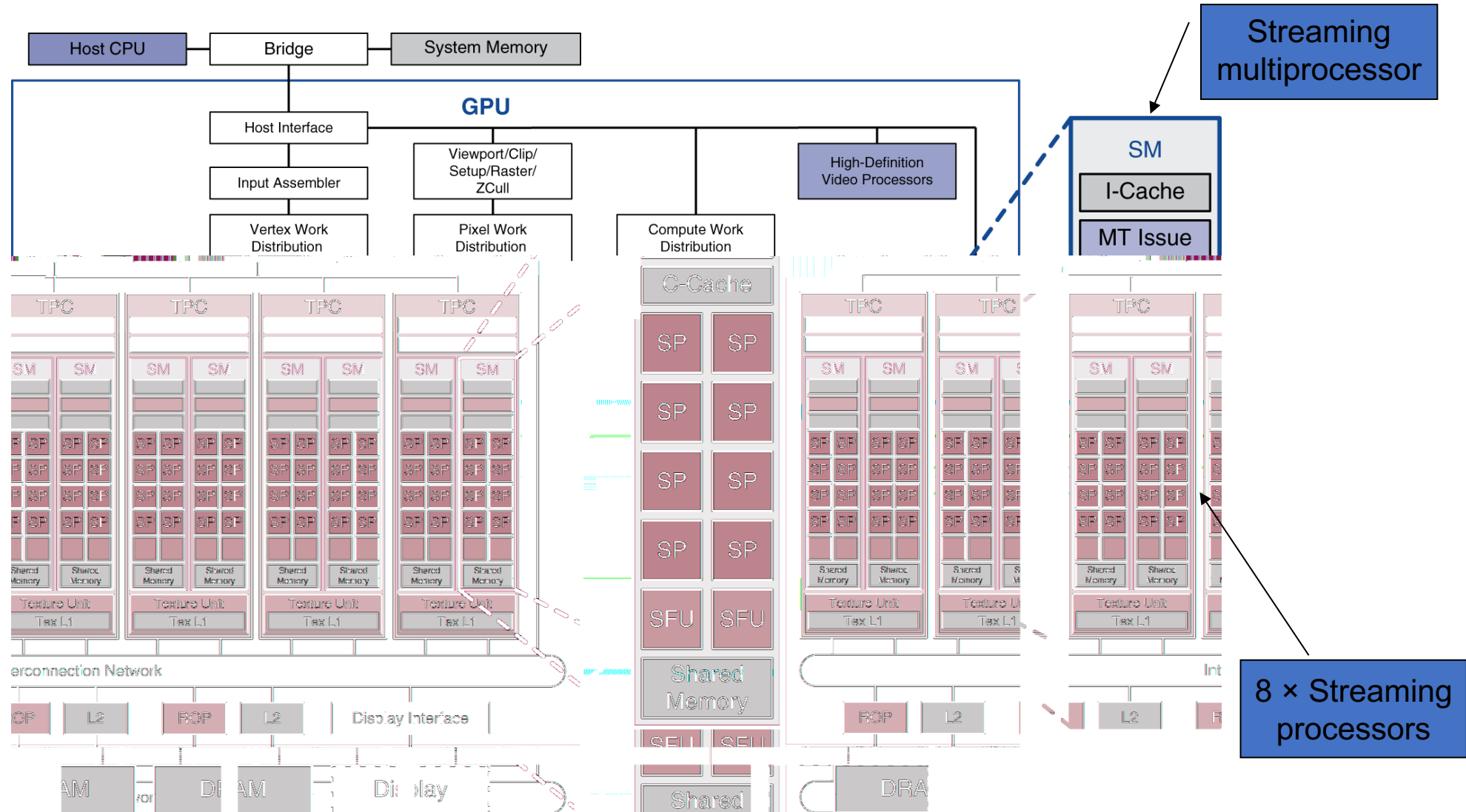
Example: NVIDIA Tesla

- Streaming Processors
  - Single-precision FP and integer units
  - Each SP is fine-grained multithreaded

- Warp: group of 32 threads
  - Executed in parallel, SIMD style
    - 8 SPs × 4 clock cycles
  - Hardware contexts for 24 warps
    - Registers, PCs, …

Processors →

UltraSPARC T2
Thread0
Thread1
Thread2
Thread3
Thread4
Thread5
Thread6
Thread7

Hardware
Supported
Threads

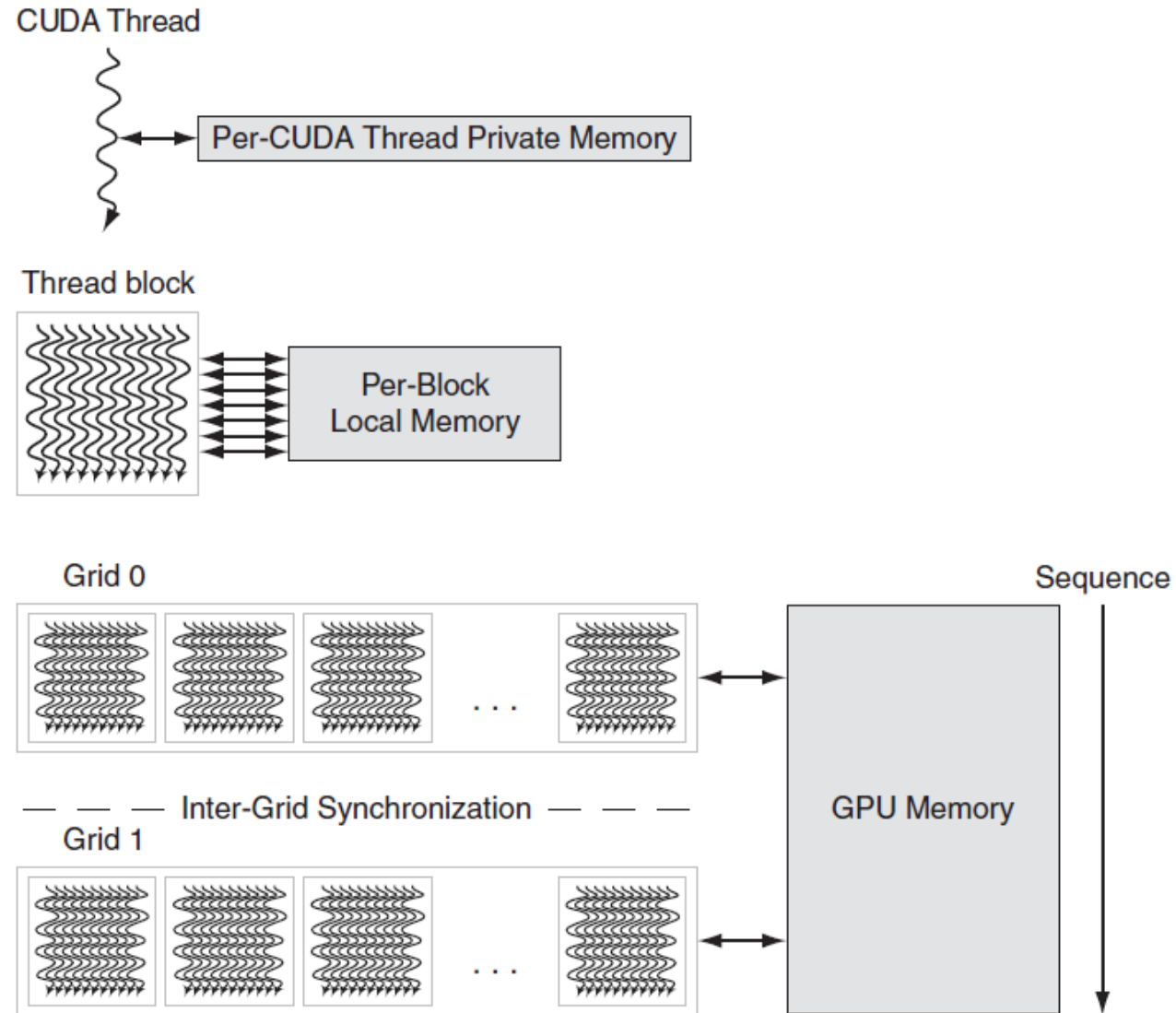Tesla Multiprocessor

Warp0

Warp1

…

Warp23

Classifying GPUs

- Don't fit nicely into SIMD/MIMD model
  - Conditional execution in a thread allows an illusion of MIMD
    - But with performance degredation
    - Need to write general purpose code with care

	Static: Discovered at Compile Time	Dynamic: Discovered at Runtime
Instruction-Level Parallelism	VLIW	Superscalar
Data-Level Parallelism	SIMD or Vector	**Tesla Multiprocessor**

GPU Memory Structures

CUDA Thread

Per-CUDA Thread Private Memory

Thread block

Per-Block
Local Memory

Grid 0

Sequence

. . .

GPU Memory

— — — Inter-Grid Synchronization — — —

Grid 1

. . .

Similarities Between GPUs and CPUs

Feature	Multicore with SIMD	GPU
SIMD processors	4 to 8	8 to 16
SIMD lanes/processor	2 to 4	8 to 16
Multithreading hardware support for SIMD threads	2 to 4	16 to 32
Typical ratio of single precision to double-precision performance	2:1	2:1
Largest cache size	8 MB	0.75 MB
Size of memory address	64-bit	64-bit
Size of main memory	8 GB to 256 GB	4 GB to 6 GB
Memory protection at level of page	Yes	Yes
Demand paging	Yes	No
Integrated scalar processor/SIMD processor	Yes	No
Cache coherent	Yes	No