高级计算机系统结构

沈海华
shenhh@ucas.ac.cn

第十六讲 Graphics Processing Units

GPU

Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory
M.I.T.

Why Study GPUs?

- Very successful commodity accelerator/co-processor
- GPUs combine two strategies to increase efficiency
  - Massive parallelism
  - Specialization
- Illustrates tension between performance and programmability in accelerators

GPU vs. CPU

- CPU

  -Latency oriented

  -高主频、复杂流水线和控制技术、强大的执行单元，追求绝对性能----即程序执行所需的时间短。

- GPU

  -Throughput oriented

  -通过海量的流处理器实现大规模的线程并行，采用SIMD基本结构扩展，通过大量数据的相同操作，以及不同操作间的流水线并行，提高数据的吞吐率。

GPU vs. CPU

- CPU适合各类复杂的控制或逻辑、算数运算任务，特别是计算复杂度高的单进程任务。
- GPU擅长用海量简单的计算单元去完成大量的计算密集型任务，适合计算量大，计算复杂度低，具有高度重复性的工作。

如果有个工作需要算几亿次一百以内加减乘除.....

- 一种办法是雇上几十甚至上百个小学生一起算，一人算一部分。采用这种办法基于一个前提，就是计算任务彼此之间基本不存在依赖性，相对独立。
- 另一种办法是雇佣一个专家教授，专家从数学分析到矩阵概率都会算。

哪个算得快？

什么类型的程序适合在GPU上运行？

- 计算密集型的程序

  -所谓计算密集型(Compute-intensive)的程序，主要指操作依赖简单寄存器访问和功能单元运算，访存相对较少。

- 易于并行的程序

  -GPU的基础是SIMD(Single Instruction Multiple Data)架构，常常有多达成百上千个运算单元，非常适合运算相关度低的重复且相对独立的运算。

  举例：石油勘探、天气预报、图形图像处理、音视频处理等

An efficient GPU workload ...

- Has thousands of independent pieces of work
  - Uses many ALUs on many cores
  - Supports massive interleaving for latency hiding
- Is amenable to instruction stream sharing
  - Maps to SIMD execution well
- Is compute-heavy: the ratio of math operations to memory access is high
  - Not limited by memory bandwidth

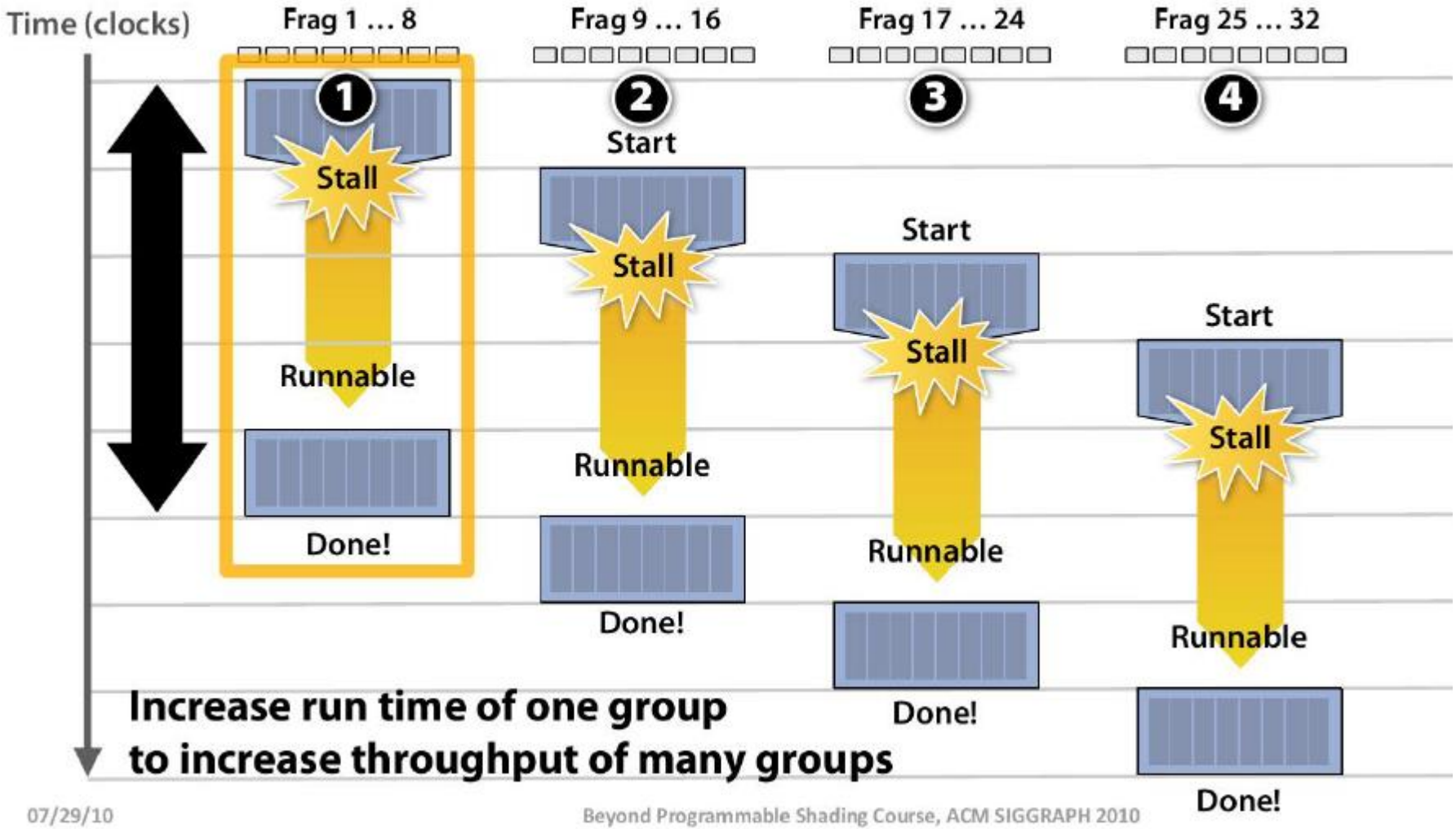
GPU中的SIMD 流程 vs. CPU SIMD指令

Clarification

SIMD processing does not imply SIMD instructions

- Option 1: explicit vector instructions
  - x86 SSE, Intel Larrabee
- Option 2: scalar instructions, implicit HW vectorization
  - HW determines instruction stream sharing across ALUs (amount of sharing hidden from software)
  - NVIDIA GeForce ("SIMT" warps)

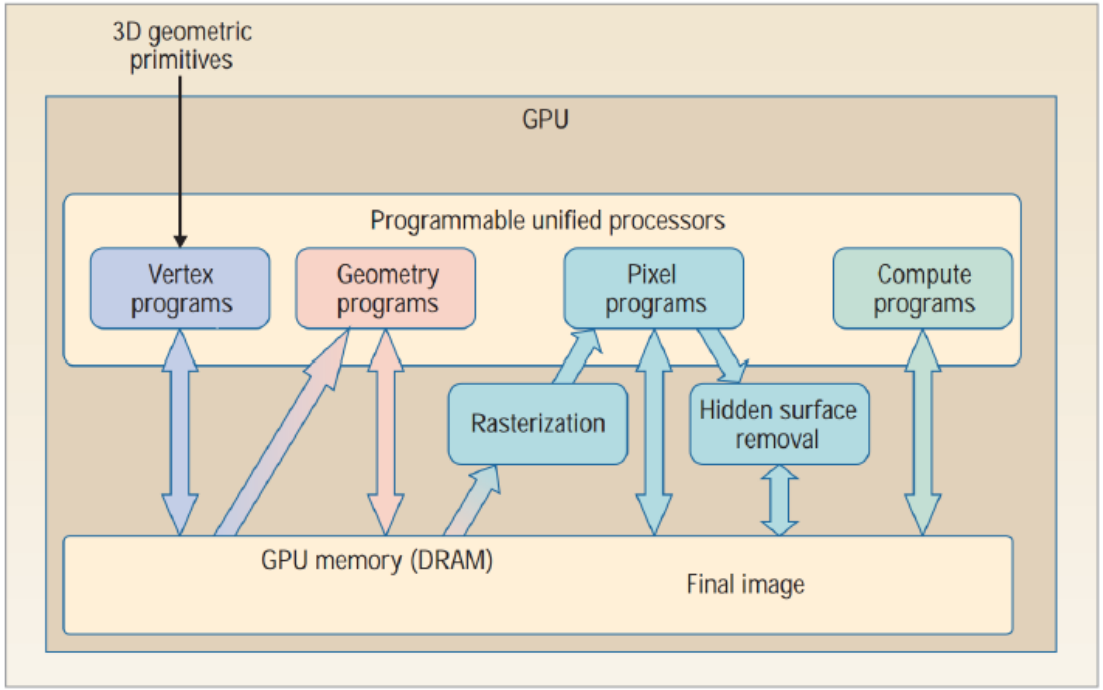
Throughput!


Time (clocks)
Frag 1 … 8
Frag 9 … 16
Frag 17 … 24
Frag 25 … 32
1
2
3
4
Stall
Start
Runnable
Done!
Increase run time of one group
to increase throughput of many groups

Graphics Processors Timeline

- Until mid-90s
  - Most graphics processing in CPU
  - VGA controllers used to accelerate some display functions
- Mid-90s to mid-2000s
  - Fixed-function accelerators for 2D and 3D graphics
    - triangle setup & rasterization, texture mapping & shading
  - Programming:
    - OpenGL and DirectX APIs

Contemporary GPUs



Luebke and Humphreys, 2007

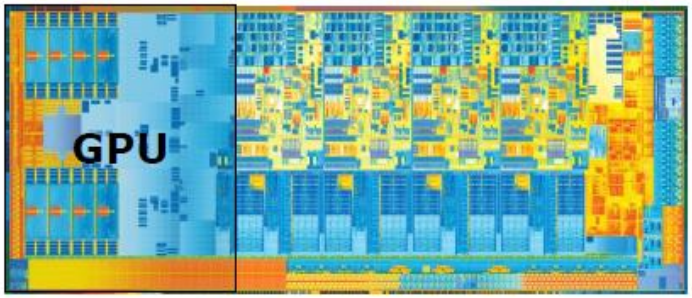
- Modern GPUs
  - Some fixed-function hardware (texture, raster ops, …)
  - Plus programmable data-parallel multiprocessors
  - Programming:
    - OpenGL/DirectX
    - Plus more general purpose languages (CUDA, OpenCL, …)

GPUs in Modern Systems

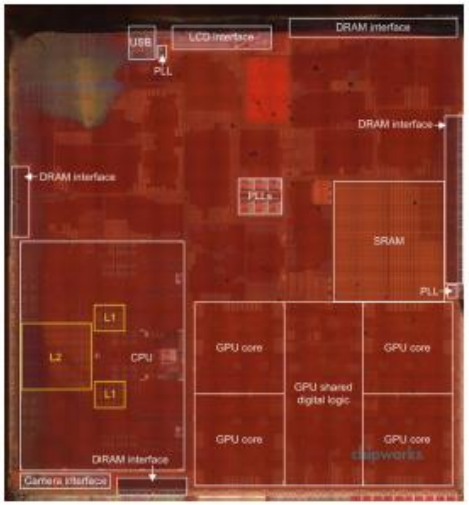
- Discrete GPUs
  - PCIe-based accelerator
  - Separate GPU memory

Nvidia Kepler

- Integrated GPUs
  - CPU and GPU on same die
  - Shared main memory and last-level cache



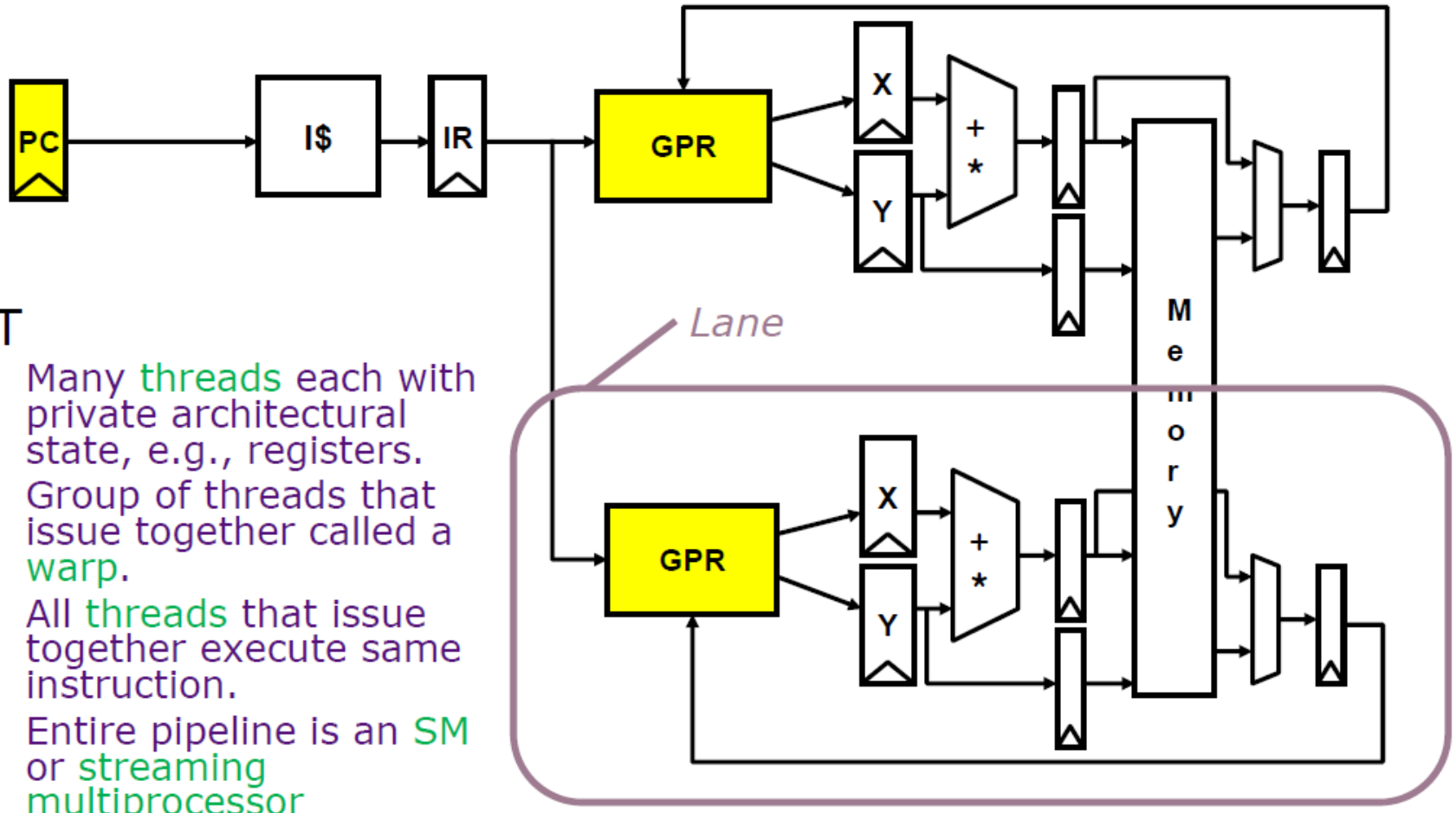
Intel Ivy Bridge, 22nm 160mm^2



Apple A7, 28nm
TSMC, 102mm^2

- Pros/cons?

Single Instruction Multiple Thread

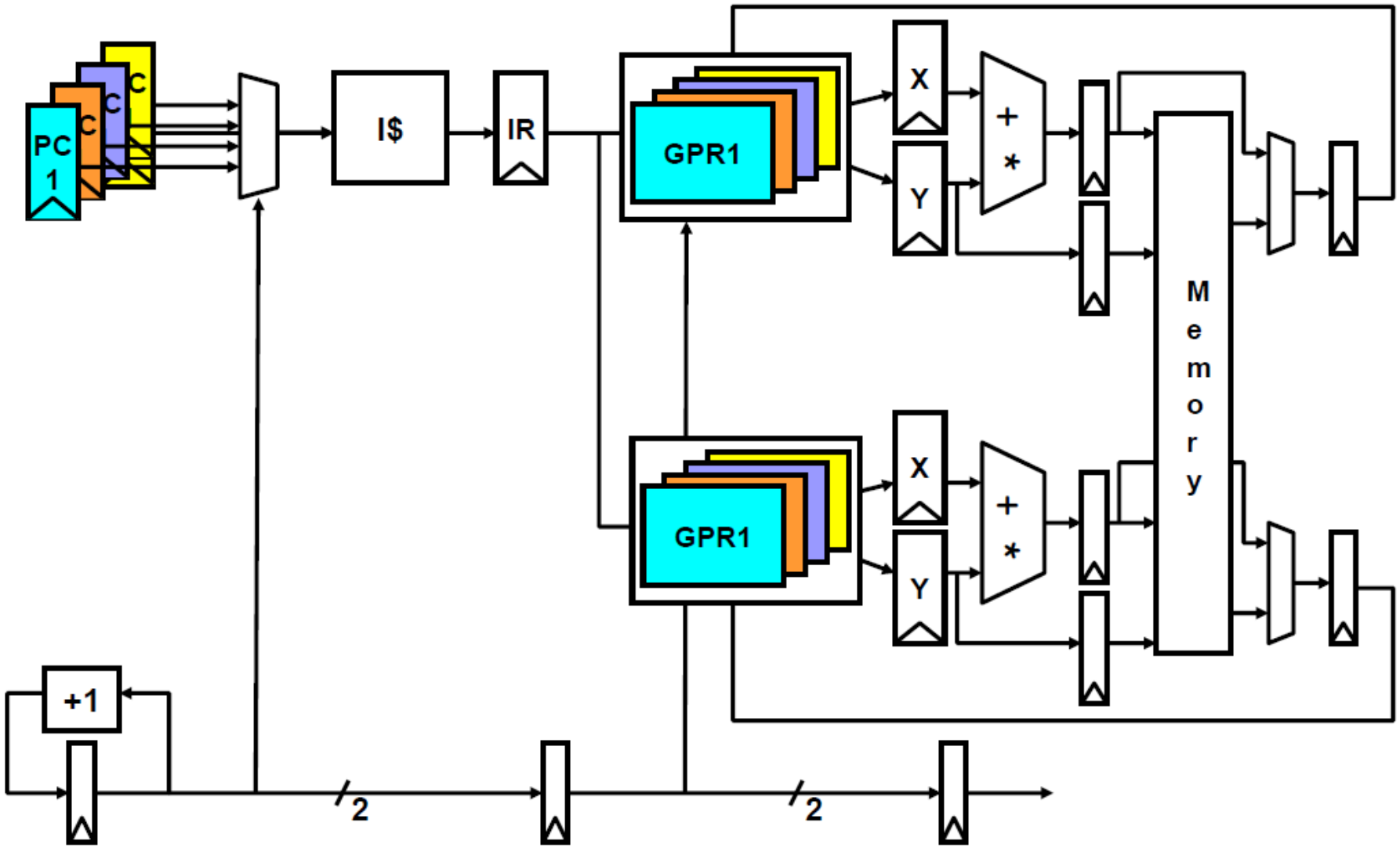


SIMT

- Many threads each with private architectural state, e.g., registers.
- Group of threads that issue together called a warp.
- All threads that issue together execute same instruction.
- Entire pipeline is an SM or streaming multiprocessor

green-> Nvidia terminology

Multiple Thread – Single Instruction Multiple Thread

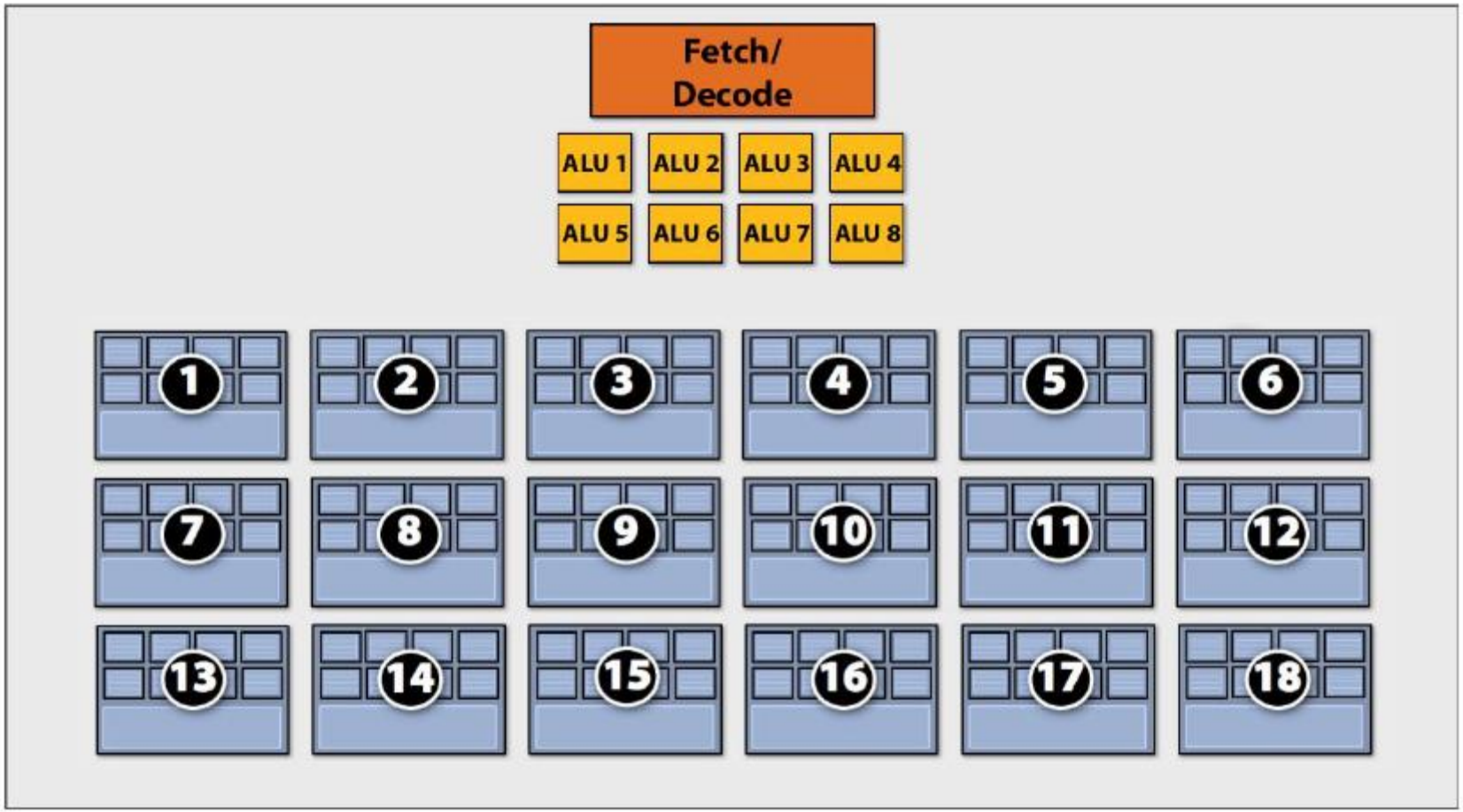


Context Size vs Number of Contexts

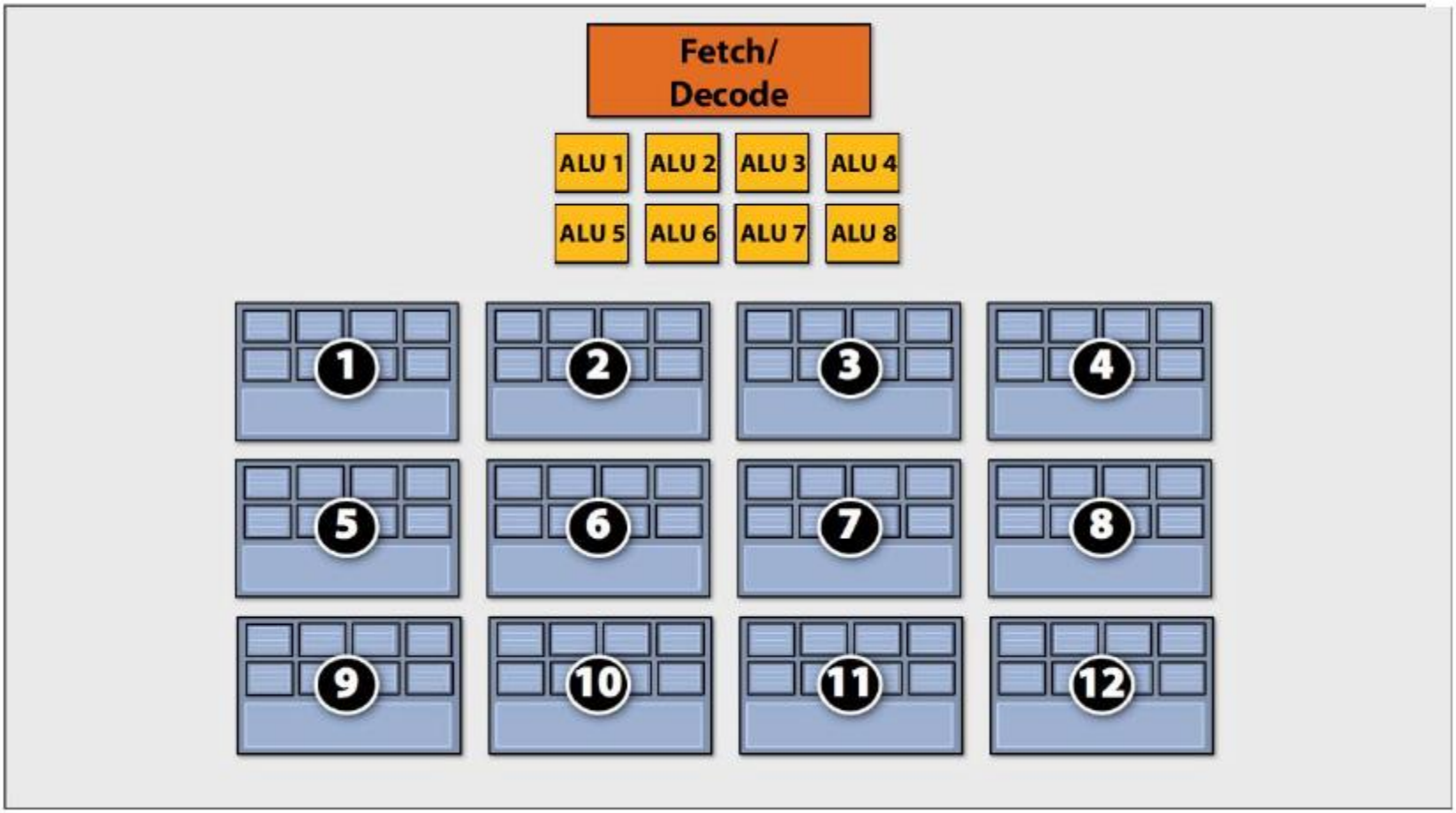
- SMs support a variable number of contexts based on required registers (and shared memory)
  - Few large contexts → Fewer register spills
  - Many small contexts → More latency tolerance
  - Choice left to the compiler

- Example: Kepler supports up to 64 warps
  - Max: 64 warps @ <=32 registers/thread
  - Min: 8 warps @ 256 registers/thread

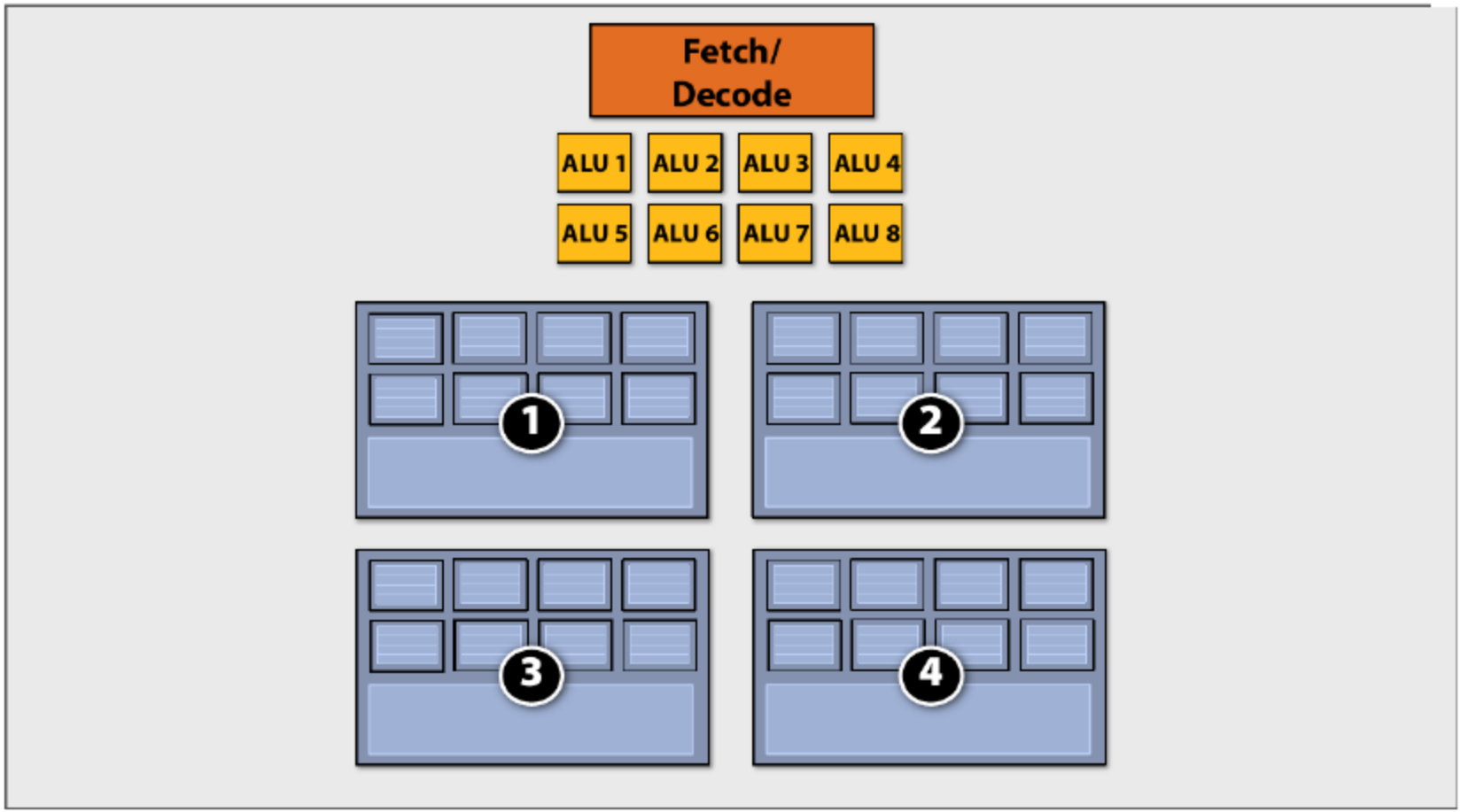
Eighteen small contexts (maximal latency hiding)



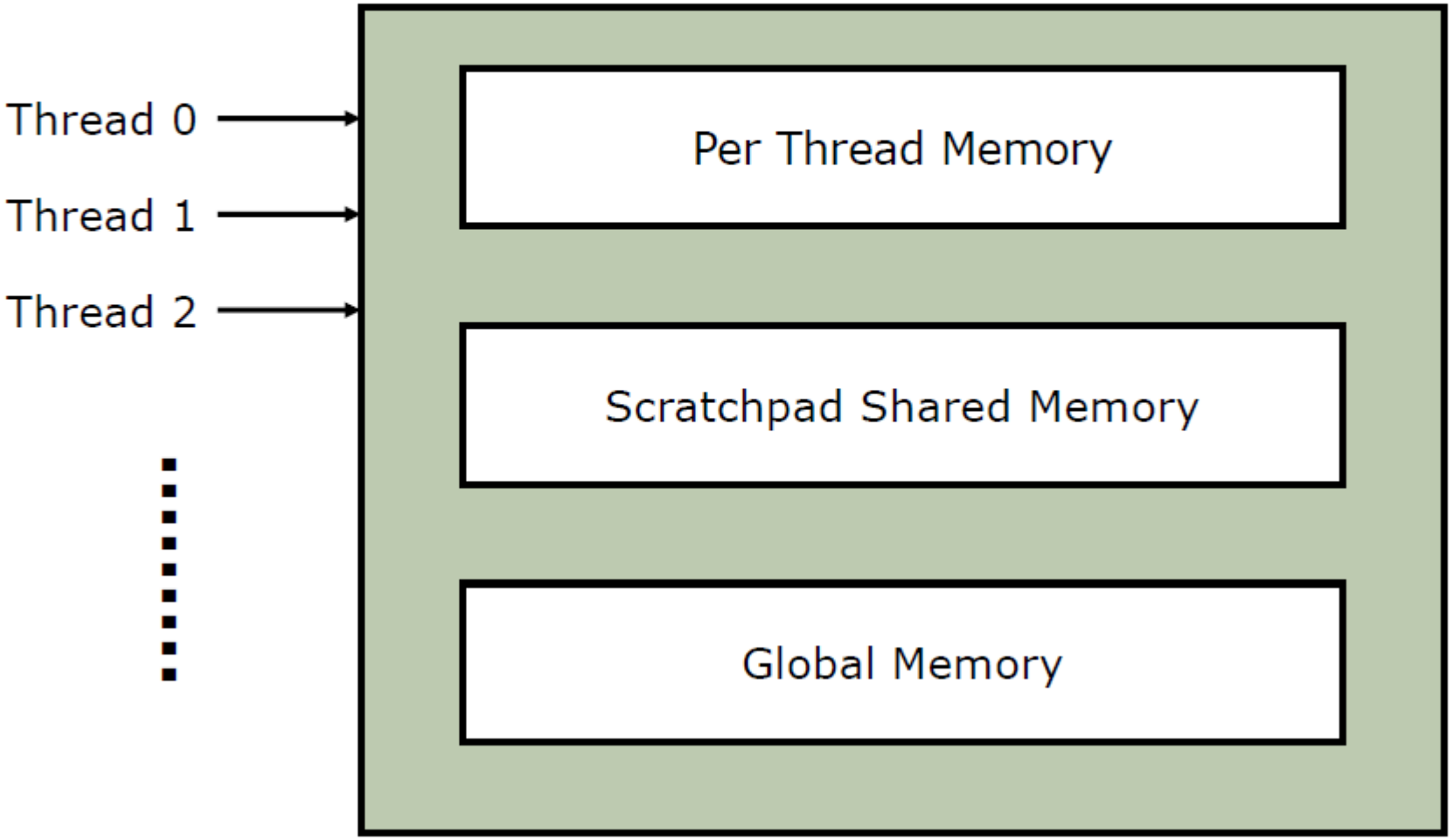
Twelve medium contexts



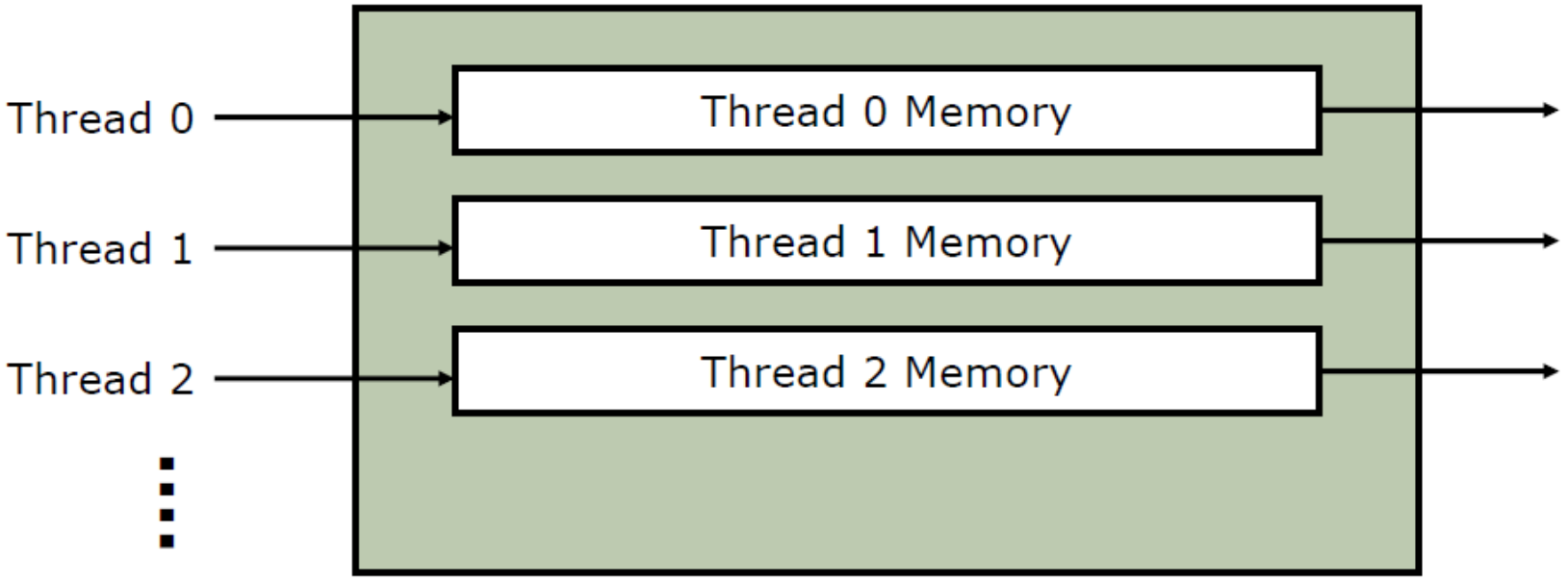
Four large contexts



Many Memory Types

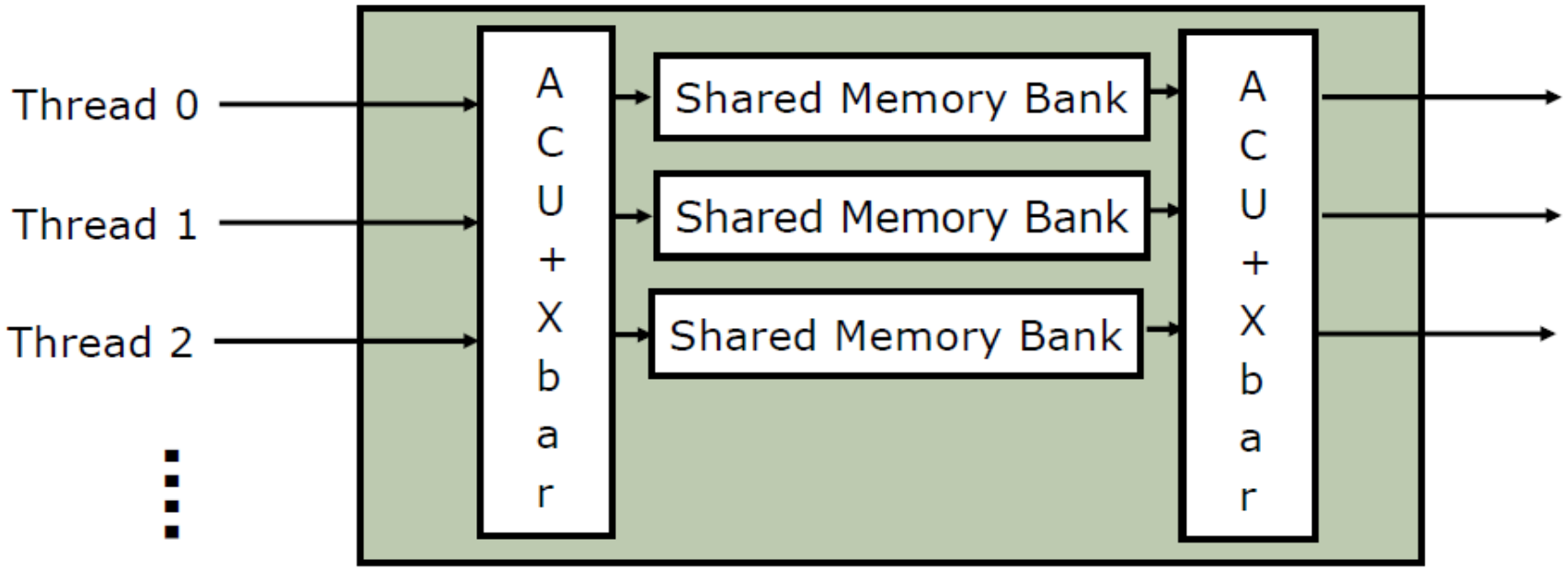


Private Per Thread Memory



- Private memory
  - No cross-thread sharing
  - Small, fixed size memory
    - Can be used for constants
  - Multi-bank implementation (can be in global memory)

Shared Scratchpad Memory

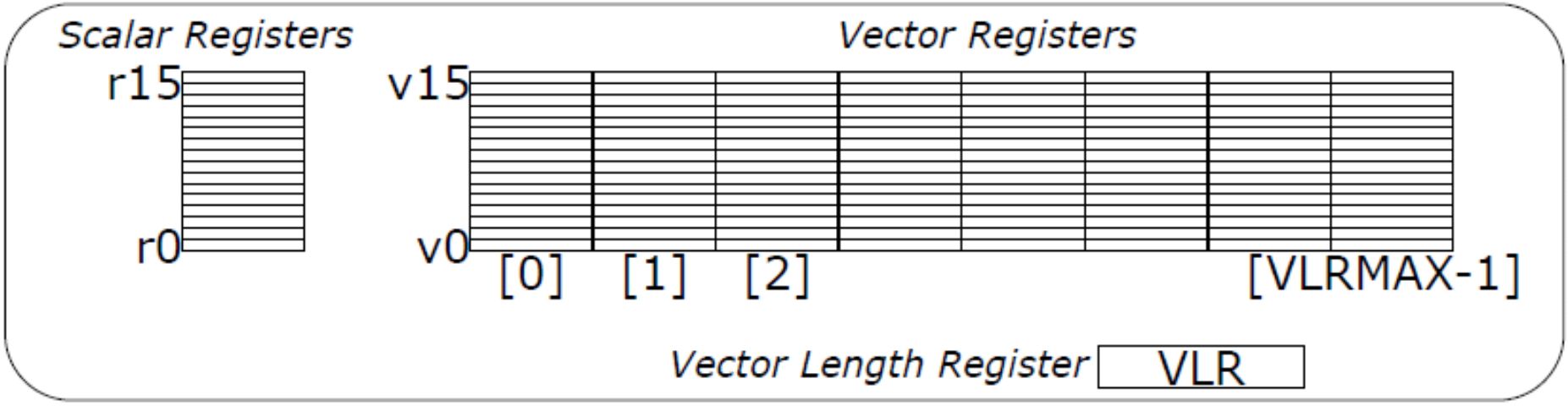


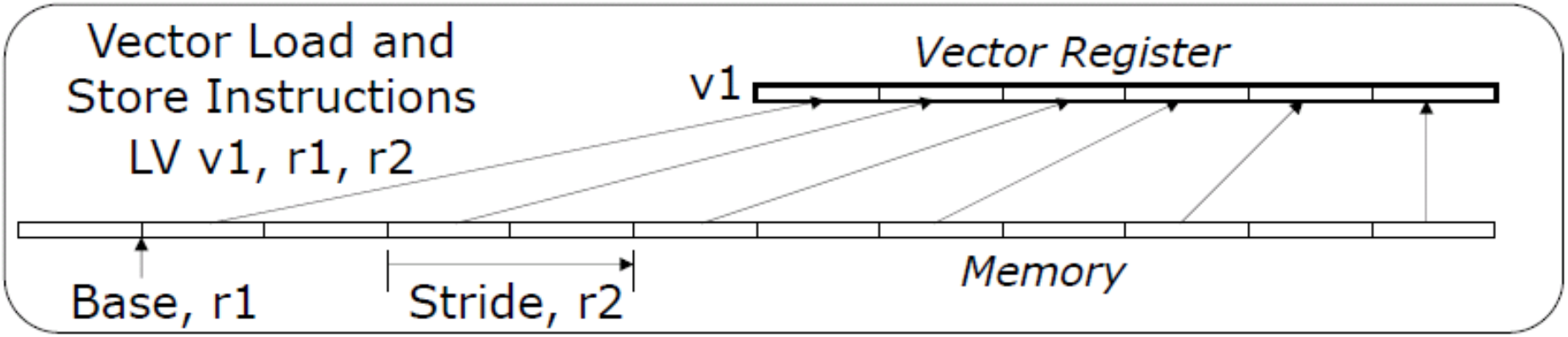
- Shared scratchpad memory (threads share data)
  - Small, fixed size memory (16K-64K / 'core')
  - Banked for high bandwidth
  - Fed with address coalescing unit +crossbar
    - ACU can buffer/coalesce requests

Memory Access Divergence

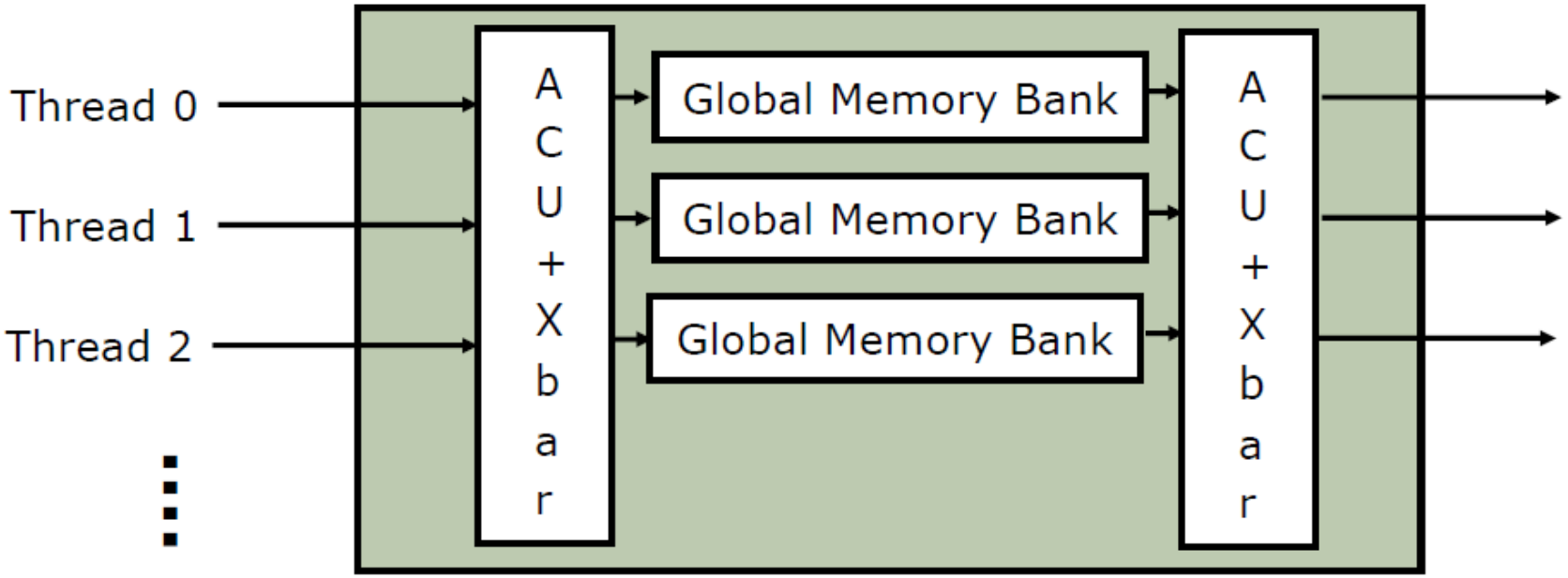
- All loads are gathers, all stores are scatters
- Address coalescing unit detects sequential and strided patterns, coalesces memory requests, but complex patterns can result in multiple lower bandwidth requests (memory divergence)
- Writing efficient GPU code requires most accesses to not conflict, even though programming model allows arbitrary patterns!

Vector Programming Model



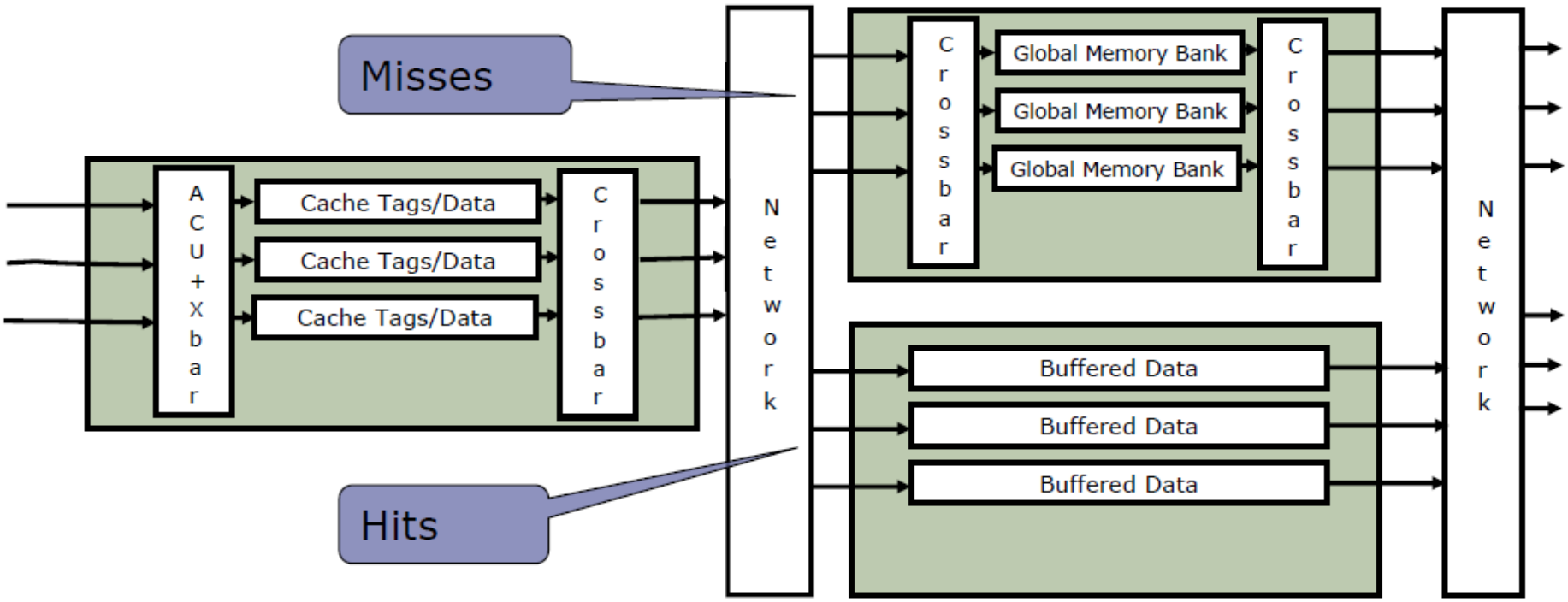


Shared Global Memory



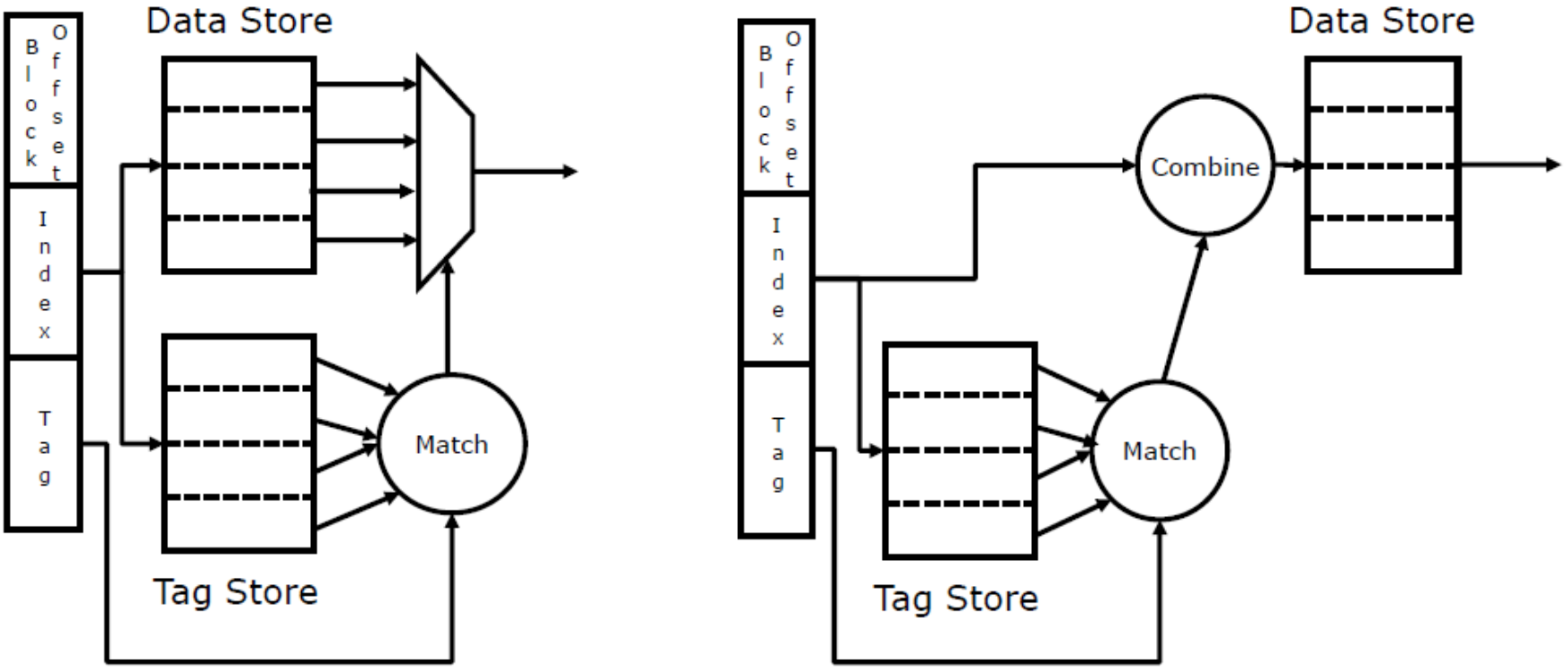
- Shared global memory
  - Large shared memory
  - Will suffer also from memory divergence

Shared Global Memory



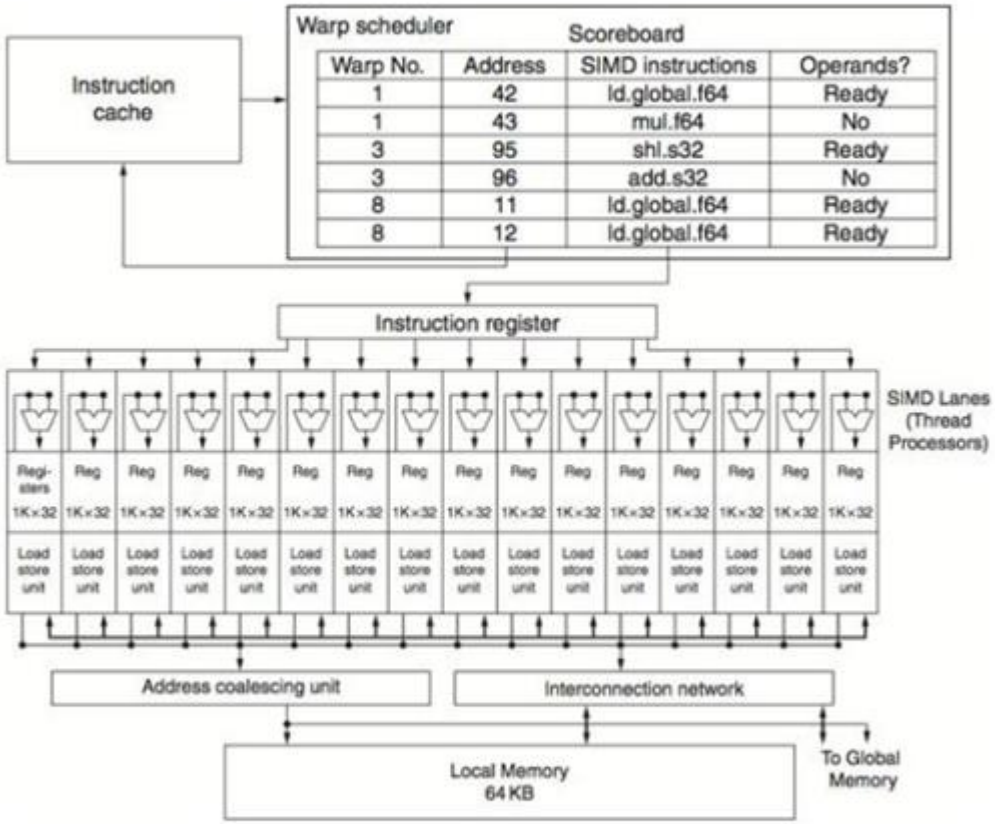
- Memory hierarchy with caches
  - Cache to save memory bandwidth
  - Caches also enable compression/decompression of data

Serialized cache access



- Trade latency for power/flexibility
  - Only access data bank that contains data
  - Facilitate more sophisticated cache organizations
    - e.g., greater associatively

Streaming Multiprocessor Overview



- Each SM supports 10s of warps (e.g., 64 in Kepler) with 32 threads/warp
- Fetch 1 instr/cycle
- Issue 1 ready instr/cycle
  - Simple scoreboarding: all warp elements must be ready
- Instruction broadcast to all lanes
- Multithreading is the main latency-hiding mechanism

Handling Branch Divergence

- Similar to vector processors, but masks are handled internally
  - Per-warp stack stores PCs and masks of non-taken paths
- On a conditional branch
  - Push the current mask onto the stack
  - Push the mask and PC for the non-taken path
  - Set the mask for the taken path
- At the end of the taken path
  - Pop mask and PC for the non-taken path and execute
- At the end of the non-taken path
  - Pop the original mask before the branch instruction
- If a mask is all zeros, skip the block

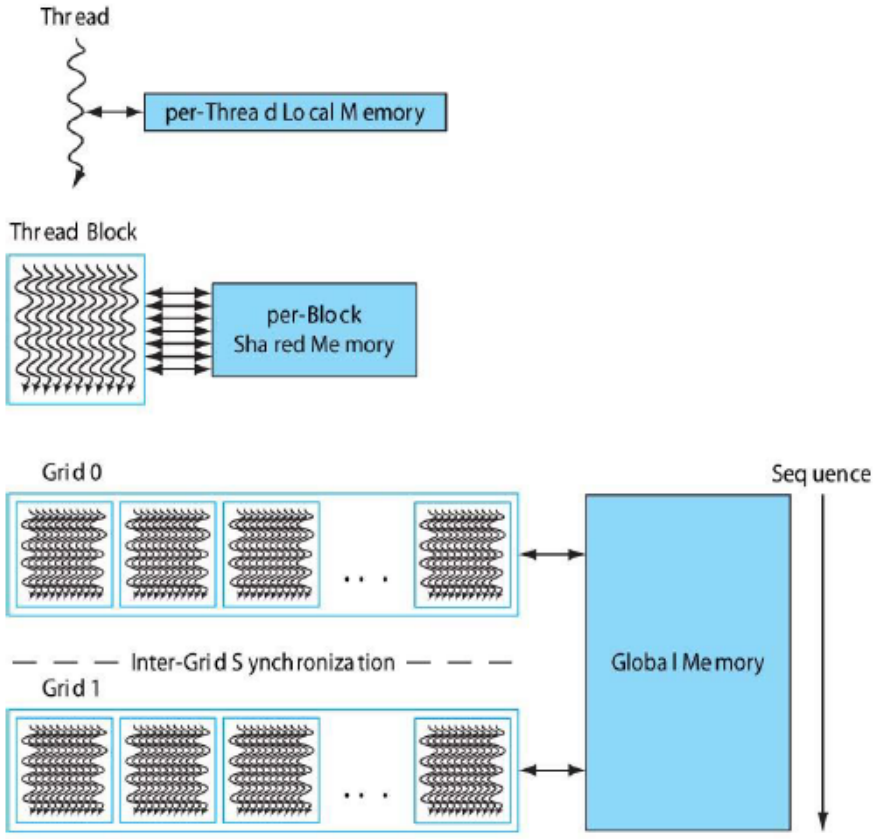
Example: Branch Divergence

Assume 4 threads/warp,
initial mask 1111

```
if (m[i] != 0) {                    1
  if (a[i] > b[i]) {                2
    y[i] = a[i] - b[i];
  } else {                          3
    y[i] = b[i] - a[i];
  }
} else {                            4
  y[i] = 0;
}                                   5
```

1. Push mask 1111
   Push mask 0011
   Set mask 1100
2. Push mask 1100
   Push mask 0100
   Set mask 1000
3. Pop mask 0100
4. Pop mask 0011
5. Pop mask 1111

CUDA GPU Thread Model



- Single-program multiple data (SPMD) model
- Each context is a thread
  - Threads have registers
  - Threads have local memory
- Parallel threads packed in blocks
  - Blocks have shared memory
  - Threads synchronize with barrier
  - Blocks run to completion (or abort)
- Grids include independent blocks
  - May execute concurrently
  - Share global memory, but
  - Have limited inter-block synchronization

Code Example: DAXPY

C Code

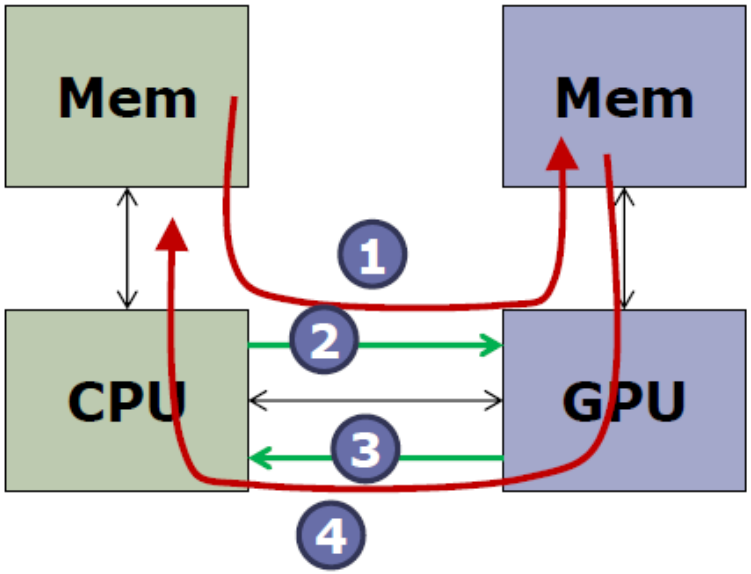
```
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n, double a, double *x, double *y)
{
        for (int i = 0; i < n; ++i)
                y[i] = a*x[i] + y[i];
}
```

CUDA Code

```
// Invoke DAXPY with 256 threads per block
__host__
int nblocks = (n+ 255) / 256;
    daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__device__
void daxpy(int n, double a, double *x, double *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
```

- CUDA code launches 256 threads per block
- CUDA vs vector terminology:
  - Thread = 1 iteration of scalar loop (1 element in vector loop)
  - Block = Body of vectorized loop (VL=256 in this example)
  - Grid = Vectorizable loop

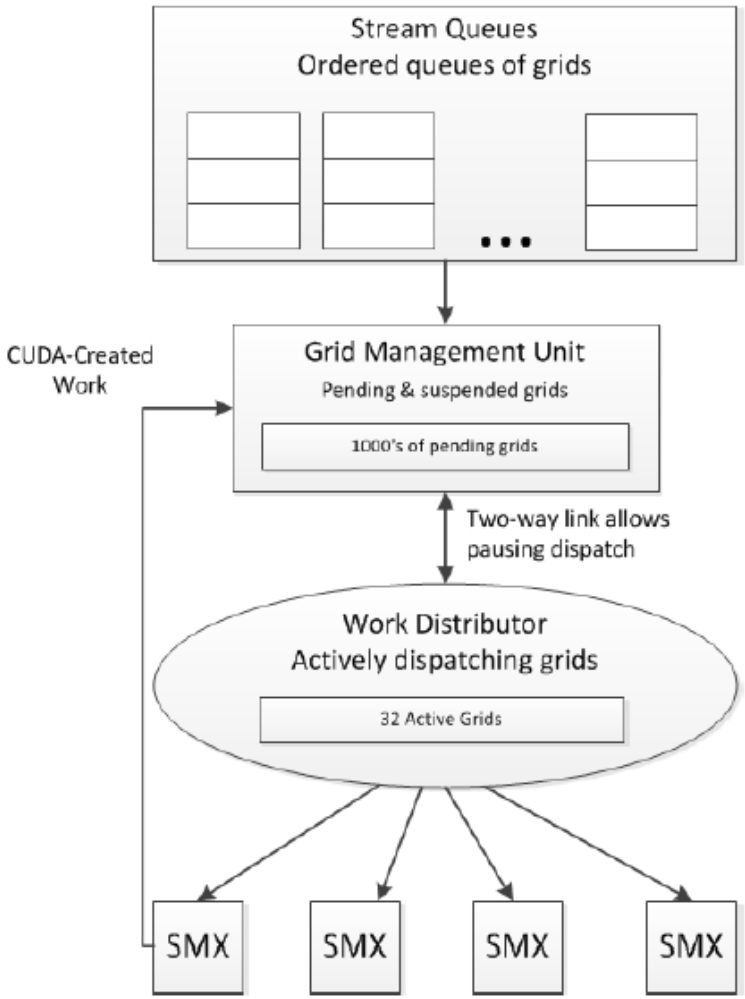
GPU Kernel Execution



1. Transfer input data from CPU to GPU memory
2. Launch kernel (grid)
3. Wait for kernel to finish (if synchronous)
4. Transfer results to CPU memory

- Data transfers can dominate execution time
- Integrated GPUs with unified address space
  → no copies, but…

Hardware Scheduling



- Grids can be launched by CPU or GPU
  - Work from multiple CPU threads and processes
- HW unit schedules grids on SMX
  - Priority-based scheduling
- 32 active grids
  - More queued/paused

Synchronization

- Barrier synchronization within a thread block (__syncthreads())
  - Tracking simplified by grouping threads into warps
  - Counter tracks number of warps that have arrived to barrier
- Atomic operations to global memory
  - Read-modify-write operations
  - Performed at the memory controller or at the L2
- Limited inter-block synchronization!
  - Can't wait for other blocks to finish

GPU ISA and Compilation

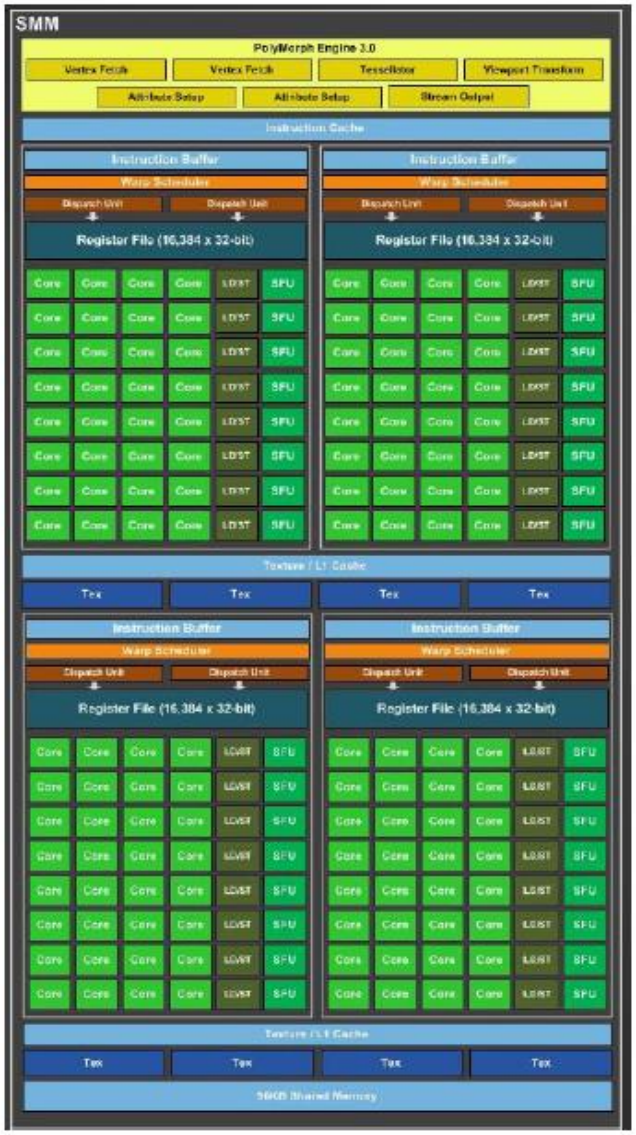
- GPU microarchitecture and instruction set change very frequently
- To achieve compatibility:
  - Compiler produces intermediate pseudo-assembler language (e.g., Nvidia PTX)
  - GPU driver JITs kernel, tailoring it to specific microarchitecture
- In practice, little performance portability
  - Code is often tuned to specific GPU architecture

GPU: Multithreaded Multicore Chip

- Example: Nvidia Maxwell GM204

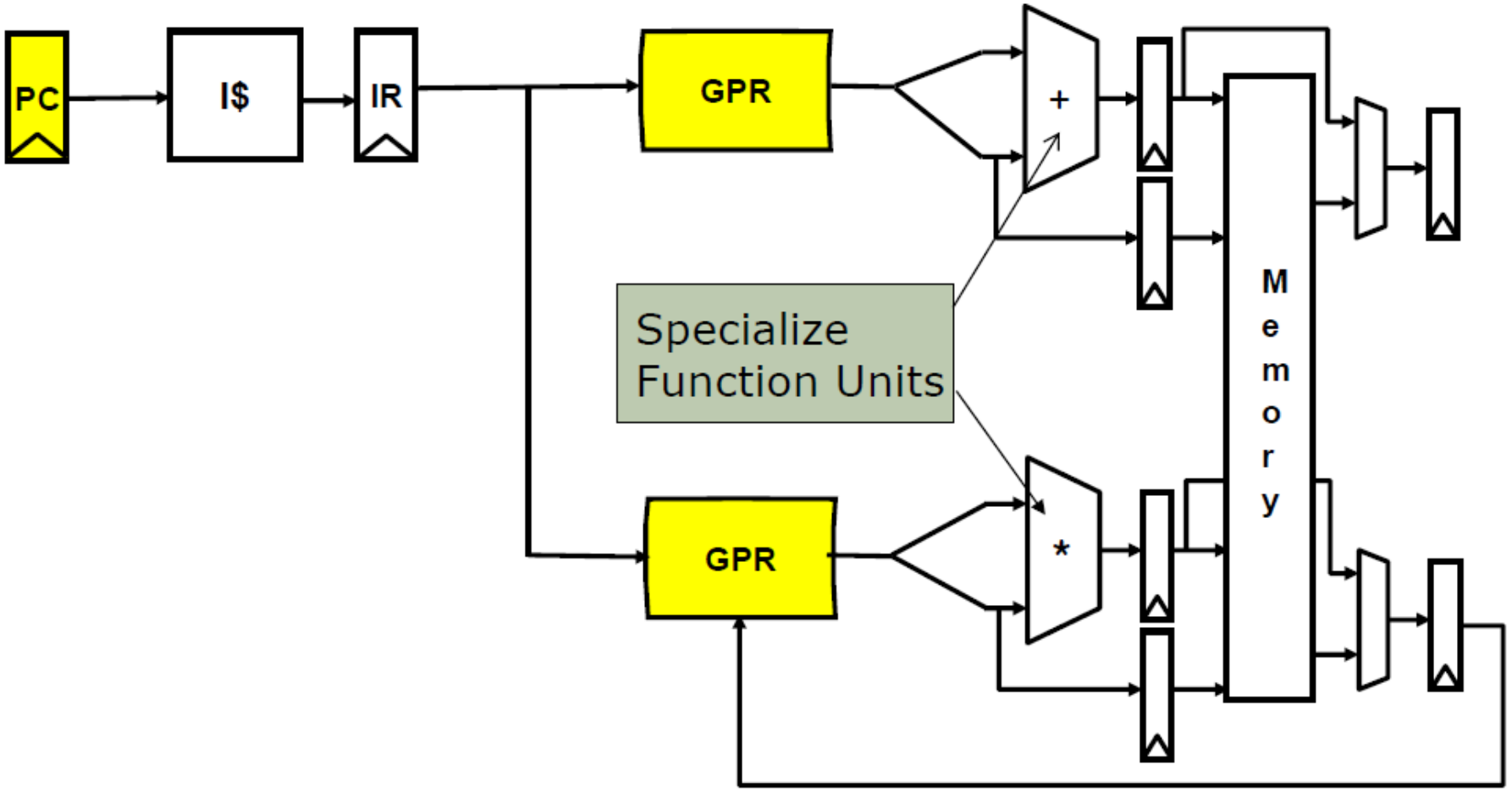
- 16 streaming multiprocessor clusters (SMM)
- 2MB Shared L2 cache
- 4 memory controllers
  - 244 GB/s
- Fixed-function logic for graphics (texture units, raster ops, …)
- Scalability → change number of cores and memory channels
- Scheduling mostly controlled by hardware

Maxwell Streaming Multiprocessor (SMM)

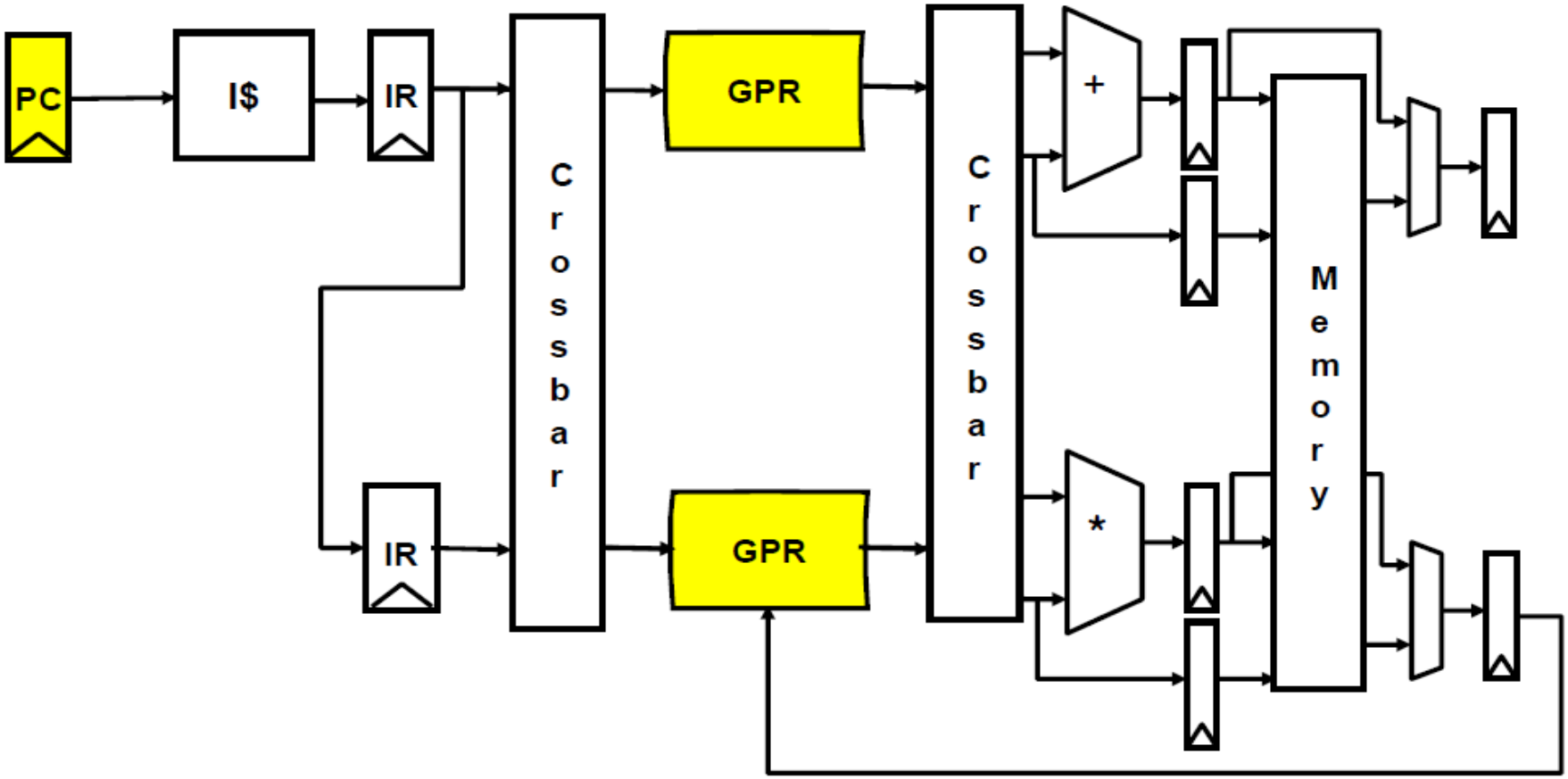


- Execution units
  - 128 FUs (int and FP)
  - 32 load-store FUs
  - 32 special-function FUs (e.g., sqrt, sin, cos, …)
- Memory structures
  - 64K 32-bit registers
  - 96KB shared memory

Function unit optimization



Function unit optimization



Restriction: Can't issue same operation twice in a row

Function unit optimization

Cycle

Unit	0	1	2	3	4	5
Fetch	$Add_{0,0-1}$	$Mul_{1,0-1}$	$Add_{2,0-1}$	$Mul_{3,0-1}$		...
GPR0		$Add_{0,0}$	$Add_{0,1}$	$Add_{2,0}$	$Add_{2,1}$	...
GPR1			$Mul_{1,0}$	$Mul_{1,1}$	$Mul_{3,0}$	...
Adder			$Add_{0,0}$	$Add_{0,1}$	$Add_{2,0}$	...
Mul				$Mul_{1,0}$	$Mul_{1,1}$	...

Key
$Opcode_{inum,thread(s)}$

System-Level Issues

- Instruction semantics
  - Exceptions
- Scheduling
  - Each kernel is non-preemptive (but can be aborted)
  - Resource management and scheduling left to GPU driver, opaque to OS
- Memory management
  - First GPUs had no virtual memory
  - Recent support for basic virtual memory (protection among grids, no paging)
  - Host-to-device copies with separate memories (discrete GPUs)

GPU总结

- CPU通用且功能强大，通常利用复杂的运算控制逻辑来减少任务的运行时间。
- GPU则利用海量简单的流处理器及计算存储单元，通过提高简单重复运算的并行性，来提高总的数据吞吐量。
- CPU和GPU趋于融合
- GPU功耗问题比CPU更严重