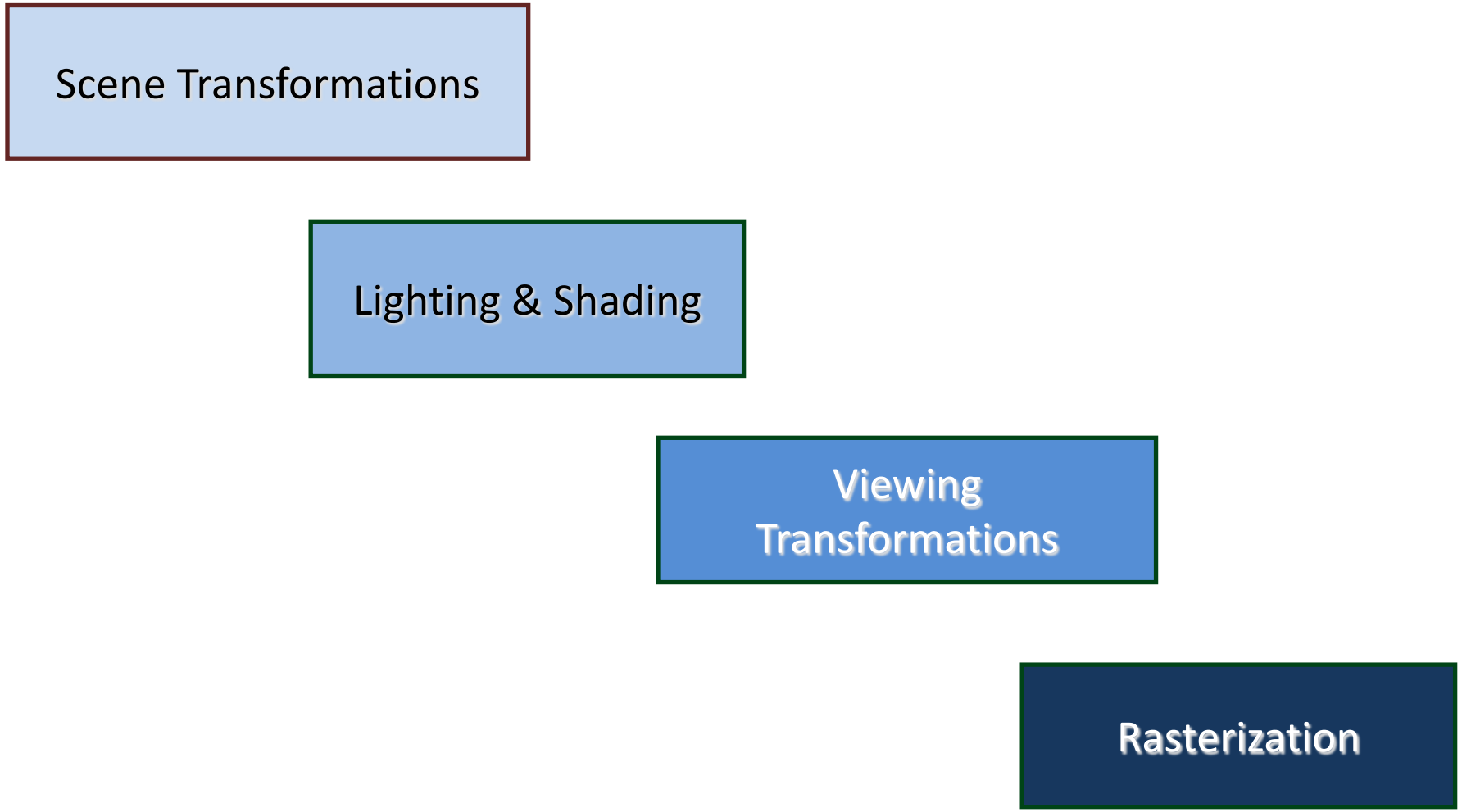
CSE 520
Computer Architecture II

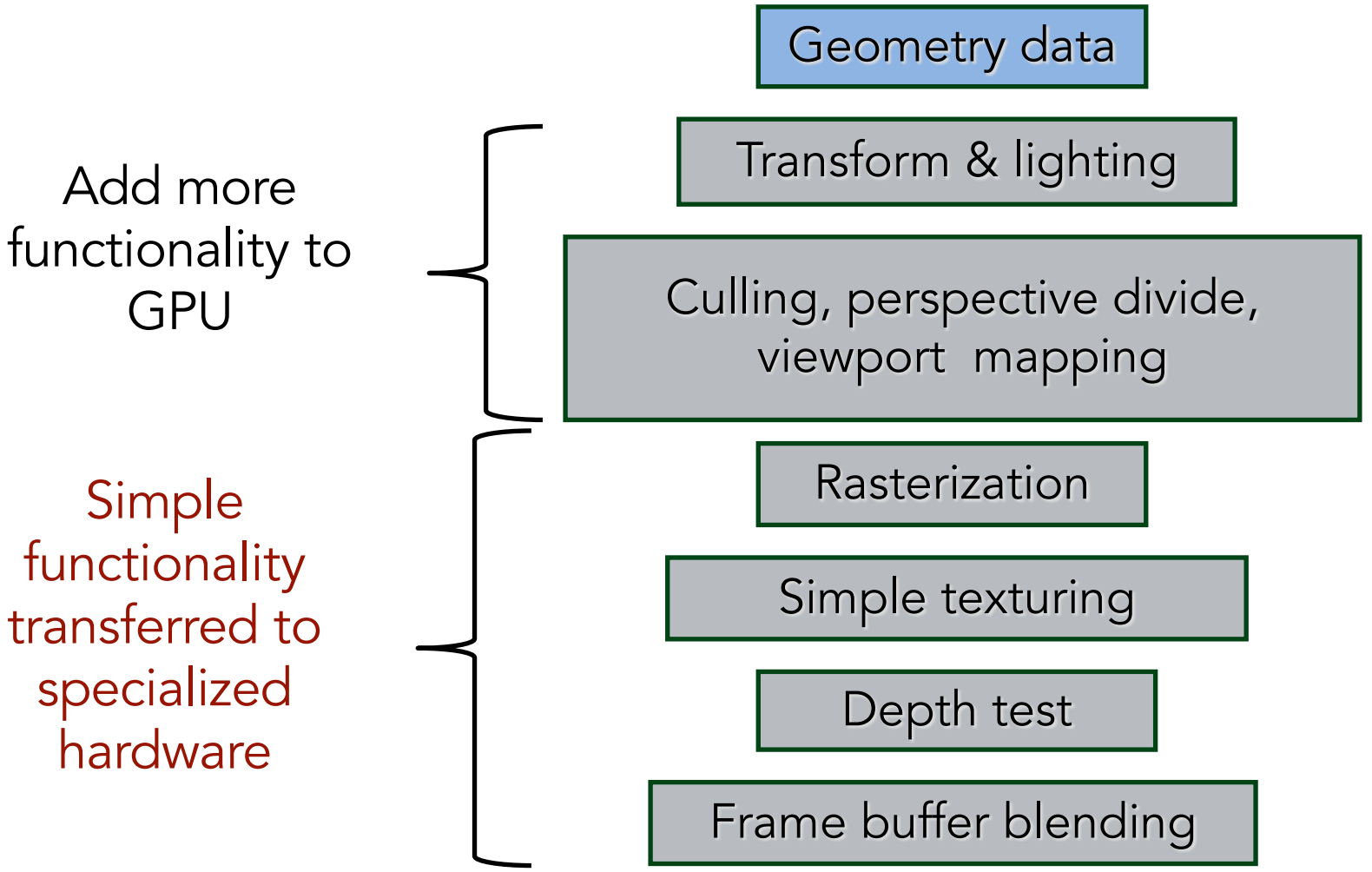
Graphic Processing Units

Prof. Michel A. Kinsy

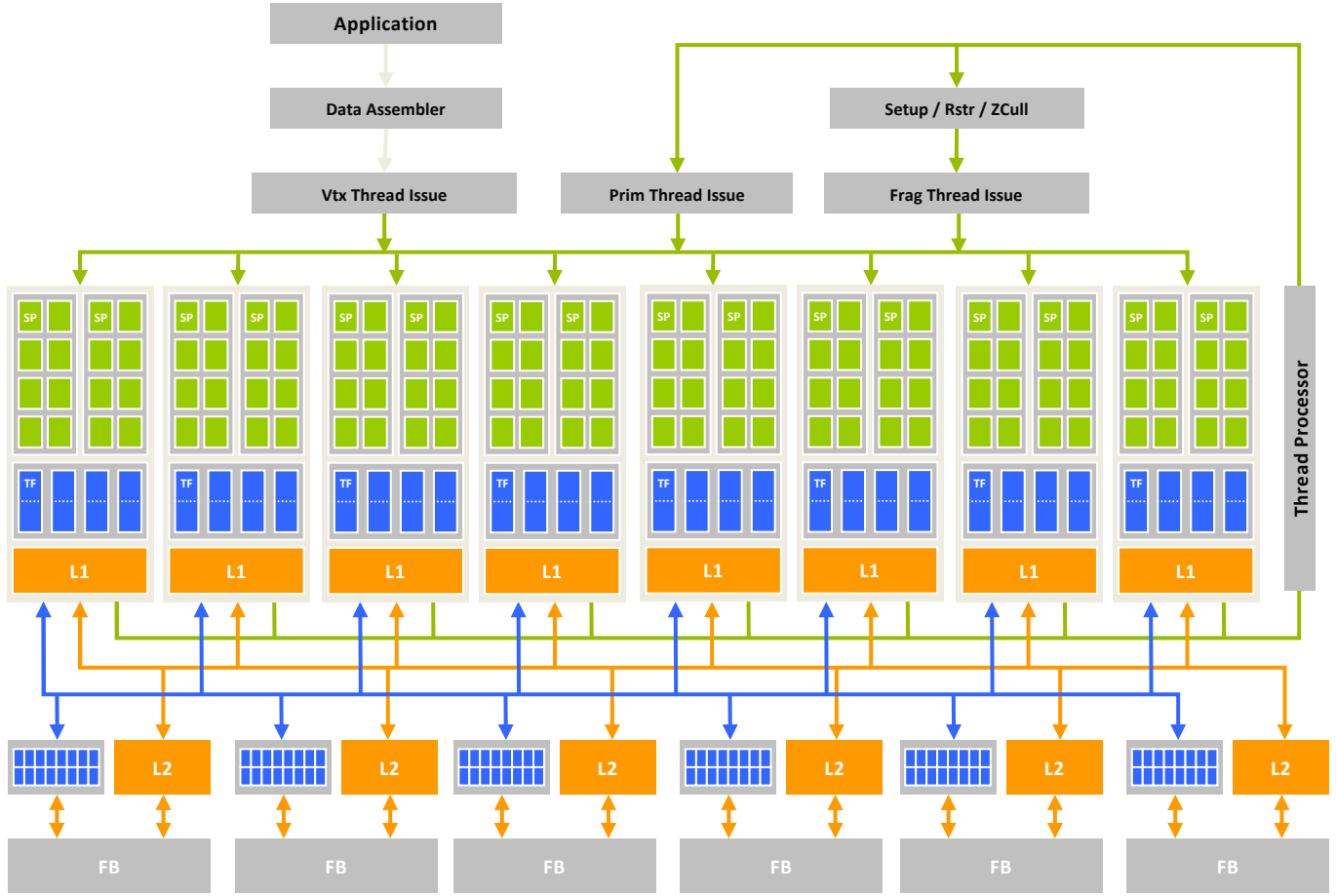
Graphics Processing Units



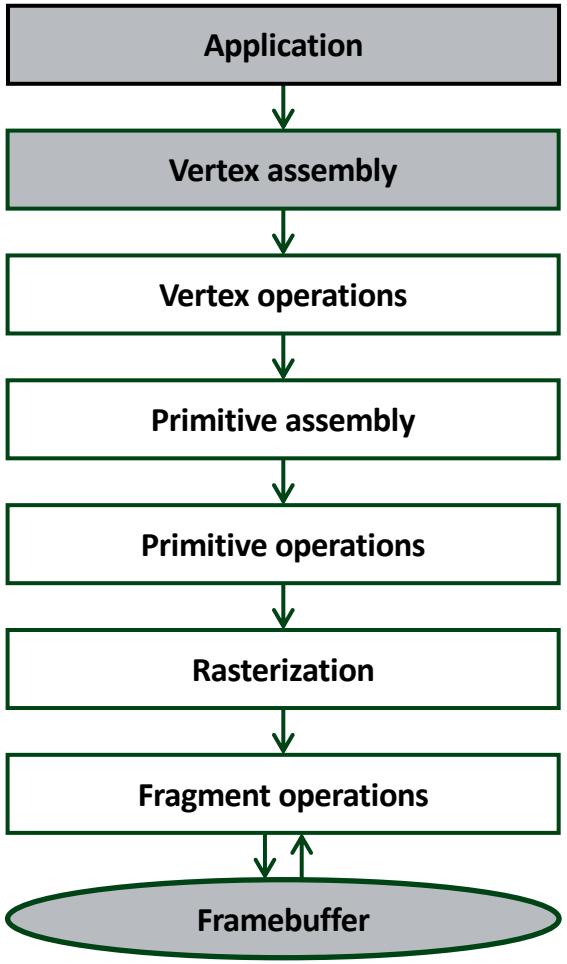
Graphics Processing Units



GPU Architecture



NVIDIA GeForce 8800



GPU Computing

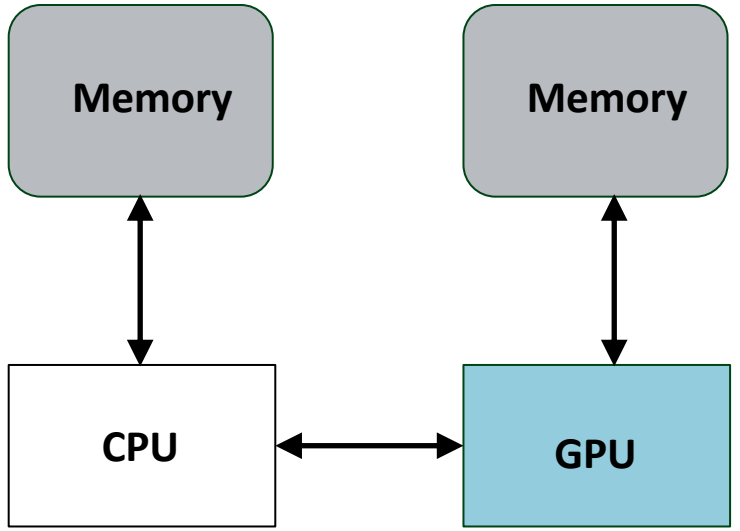
- Most successful commodity accelerator
- GPUs combine two useful strategies to increase efficiency
  - Massive parallelism
  - Specialization
- Illustrates tension between performance and programmability in accelerators

Graphics Processors Timeline

- Till mid-90s
  - VGA controllers used to accelerate some display functions
- Mid-90s to mid-2000s
  - Fixed-function accelerators for the OpenGL and DirectX APIs
  - 3D graphics: triangle setup & rasterization, texture mapping & shading

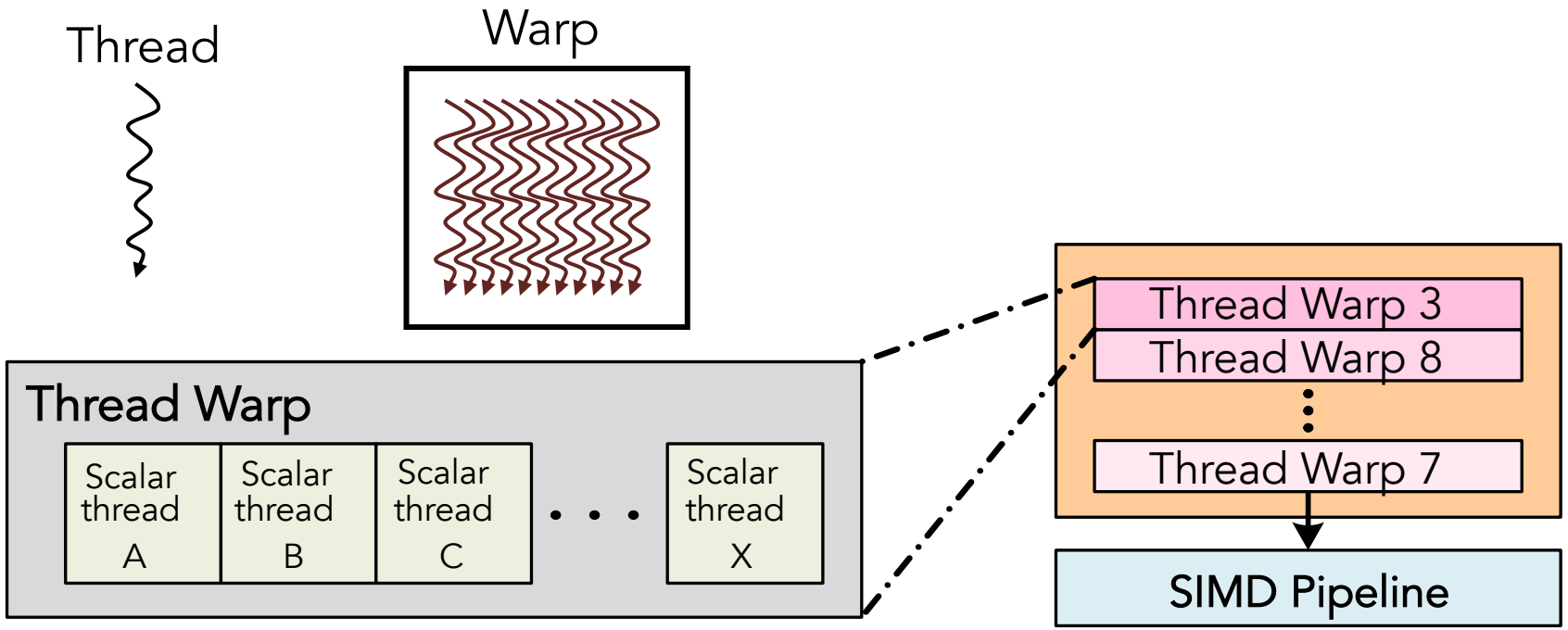
Graphics Processors Timeline

- Modern GPUs
  - Programmable multiprocessors optimized for data-parallelism
    - OpenGL/DirectX and general purpose languages (CUDA, OpenCL, …)
  - Some fixed-function hardware (texture, raster ops, …)
  - GPUs in Modern Systems
    - Discrete GPUs
      - PCIe-based accelerator
      - Separate GPU memory
    - Integrated GPUs
      - CPU and GPU on same die
      - Shared main memory and
      - last-level cache



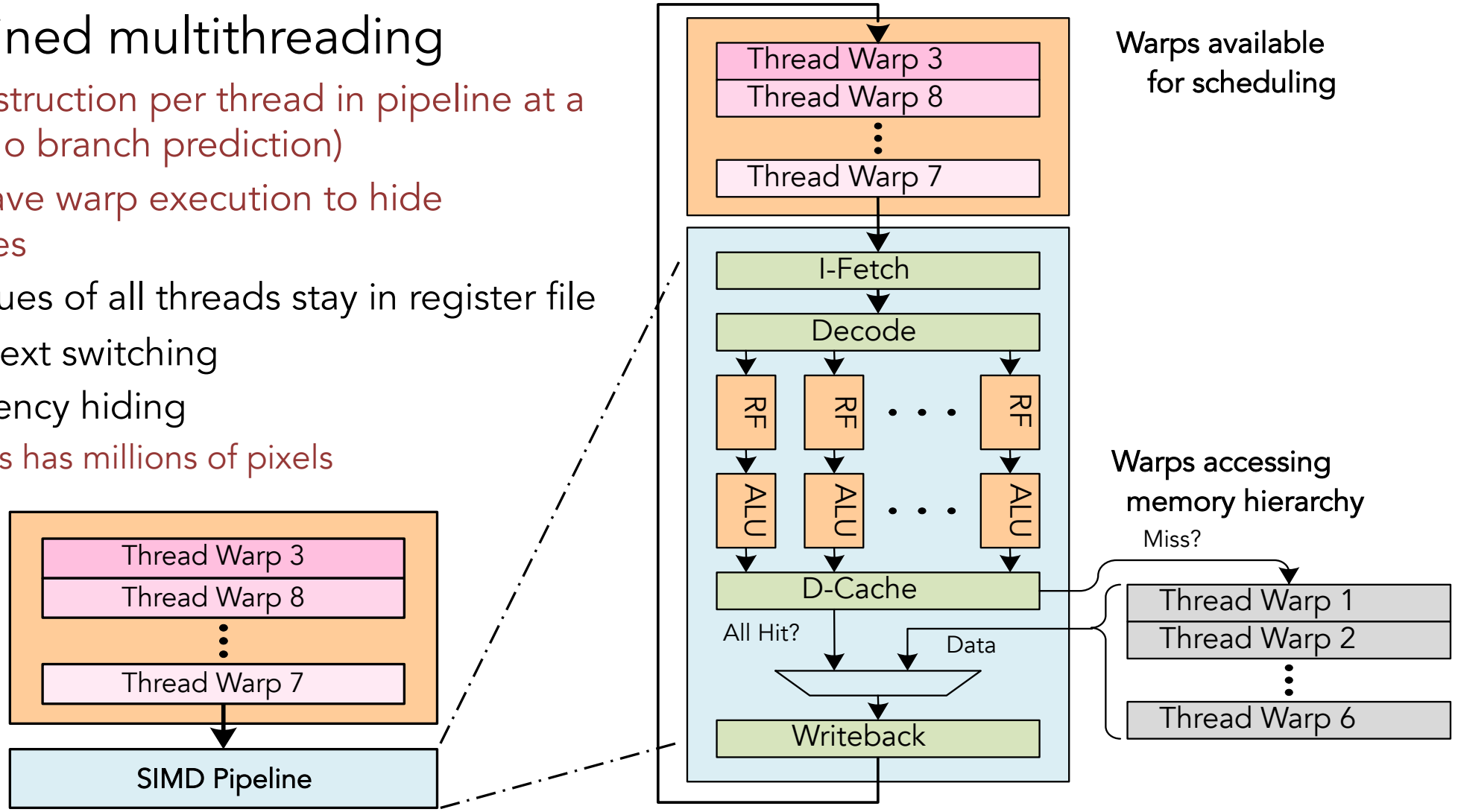
GPU Architecture

- Warp
  - A set of threads that execute the same instruction on different data elements
  - All threads run the same kernel

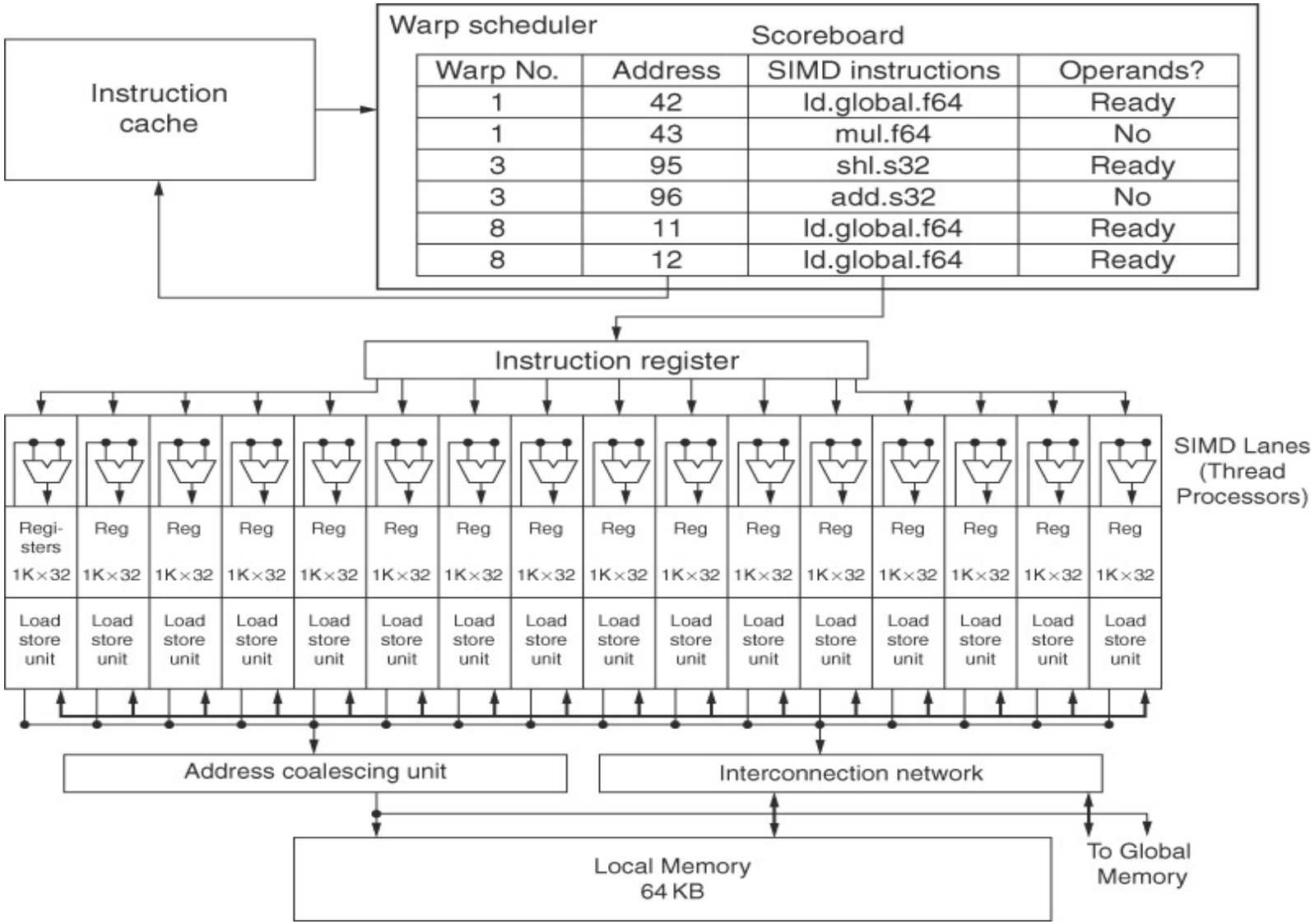


GPU Architecture

- Fine-grained multithreading
  - One instruction per thread in pipeline at a time (No branch prediction)
  - Interleave warp execution to hide latencies
- Register values of all threads stay in register file
- No OS context switching
- Memory latency hiding
  - Graphics has millions of pixels



GPU Architecture

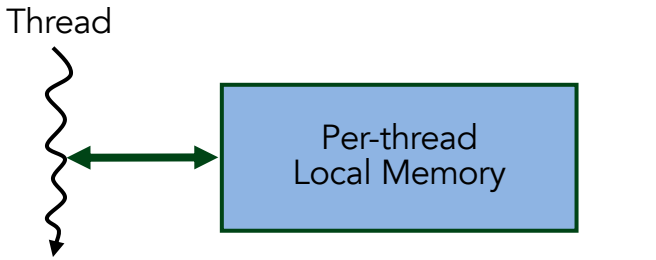


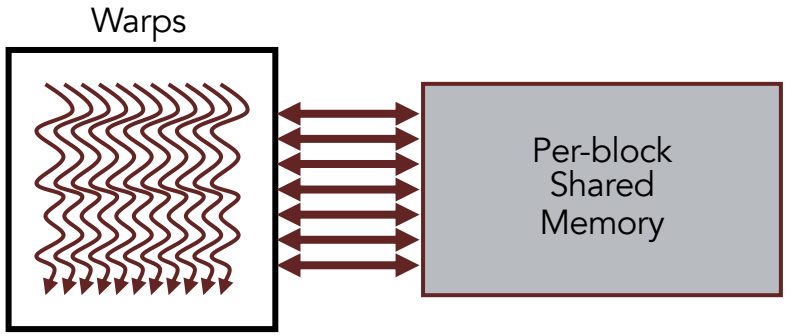
GPU Architecture

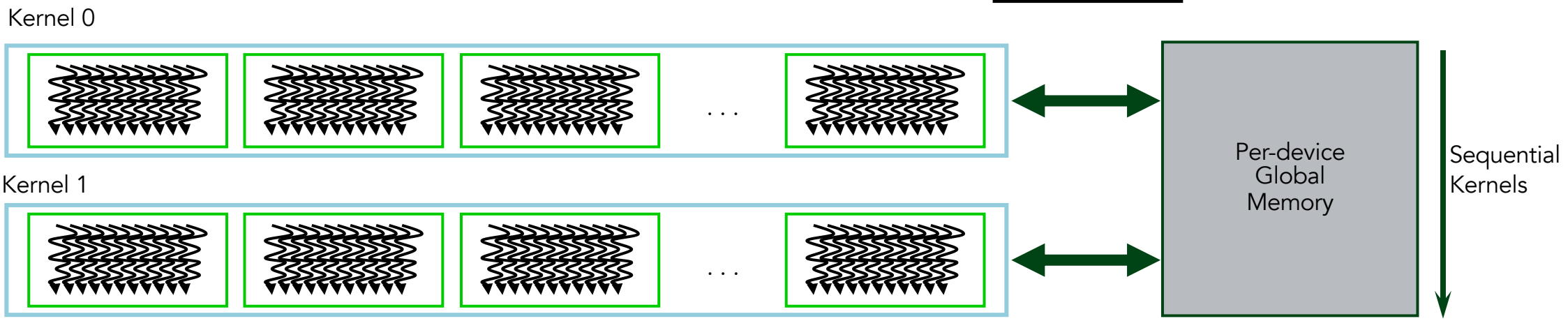
- Similarities to vector machines
  - Works well with data-level parallel problems
  - Scatter-gather transfers
  - Mask registers
  - Large register files
- Differences:
  - No scalar processor
  - Uses multithreading to hide memory latency
  - Has many functional units, as opposed to a few deeply pipelined units like a vector processor

GPU Architecture

- Each thread has local memory
- Parallel threads packed in warps
  - Access to per-warp shared memory
  - Can synchronize with barrier
- Grids include independent warps
  - May execute concurrently

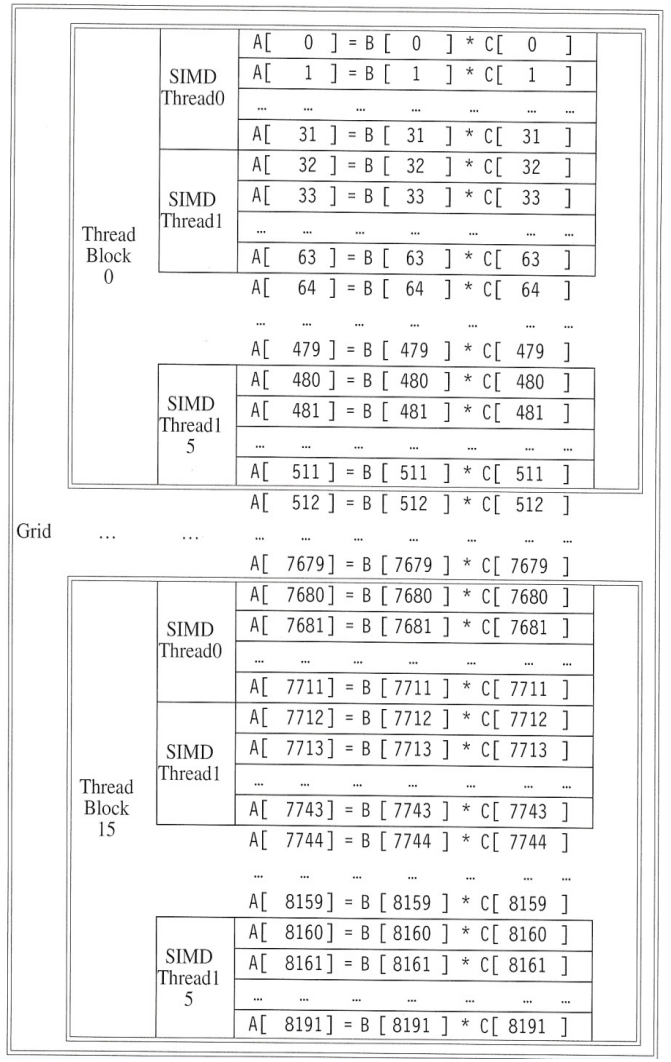






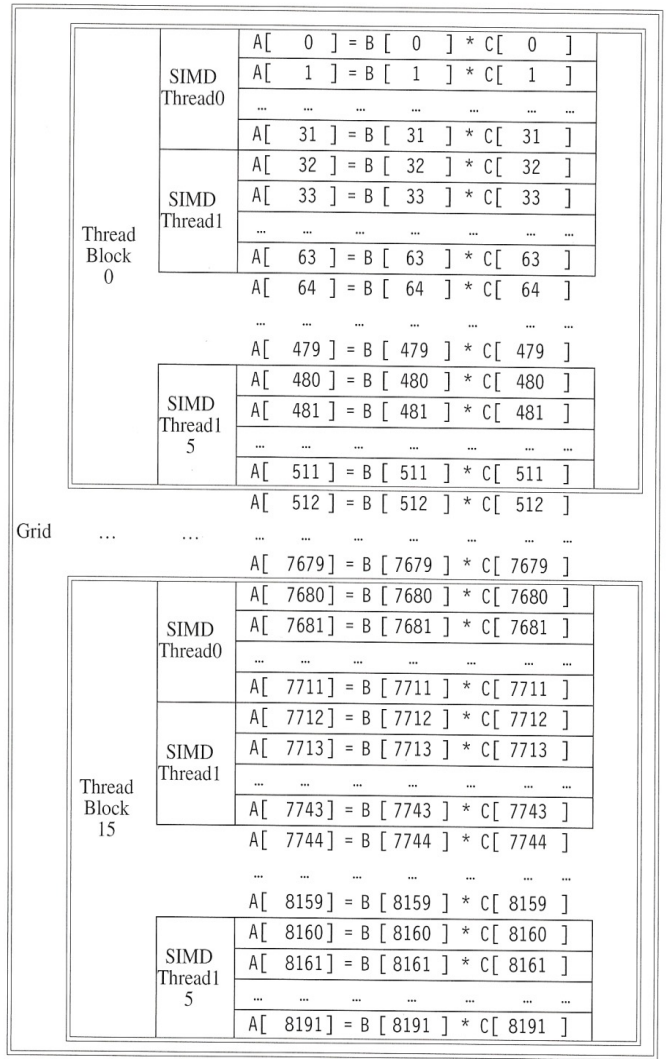
GPU Architecture

- A thread is associated with each data element
  - *CUDA threads*, with thousands of which being utilized to various styles of parallelism: multithreading, SIMD, MIMD, ILP
- Threads are organized into blocks
  - *Thread Blocks*: groups of up to 512 elements
  - *Multithreaded SIMD Processor*: hardware that executes a whole thread block (32 elements executed per thread at a time)



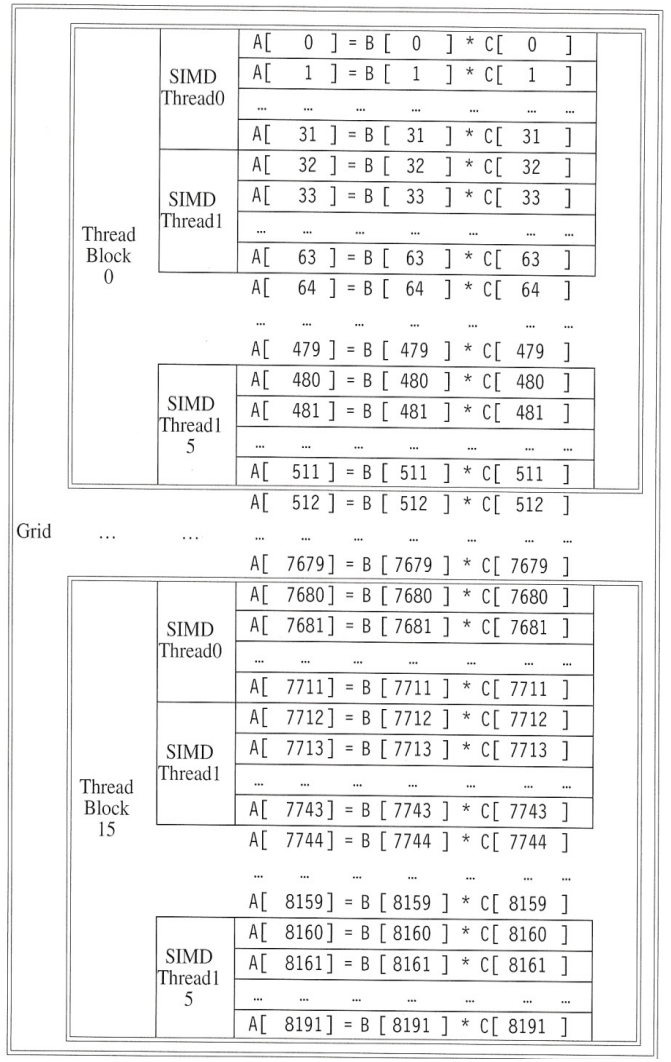
GPU Architecture

- A thread is associated with each data element
  - *CUDA threads*, with thousands of which being utilized to various styles of parallelism: multithreading, SIMD, MIMD, ILP
- Threads are organized into blocks
  - *Thread Blocks*: groups of up to 512 elements
  - *Multithreaded SIMD Processor*: hardware that executes a whole thread block (32 elements executed per thread at a time)
- Blocks are organized into a grid
  - Blocks are executed independently and in any order
  - Different blocks cannot communicate directly but can *coordinate* using atomic memory operations in Global Memory
- GPU hardware handles thread management, not applications or OS
  - A multiprocessor composed of multithreaded SIMD processors
  - A Thread Block Scheduler



GPU Architecture

- Multiply two vectors of length 8192
  - Code that works over all elements is the grid
  - Thread blocks break this down into manageable sizes
    - 512 elements/block, 16 SIMD threads/block → 32 ele/thread
  - SIMD instruction executes 32 elements at a time
  - Thus grid size = 16 blocks
  - Block is analogous to a strip-mined vector loop with vector length of 32
  - Block is assigned to a *multithreaded SIMD processor* by the *thread block scheduler*

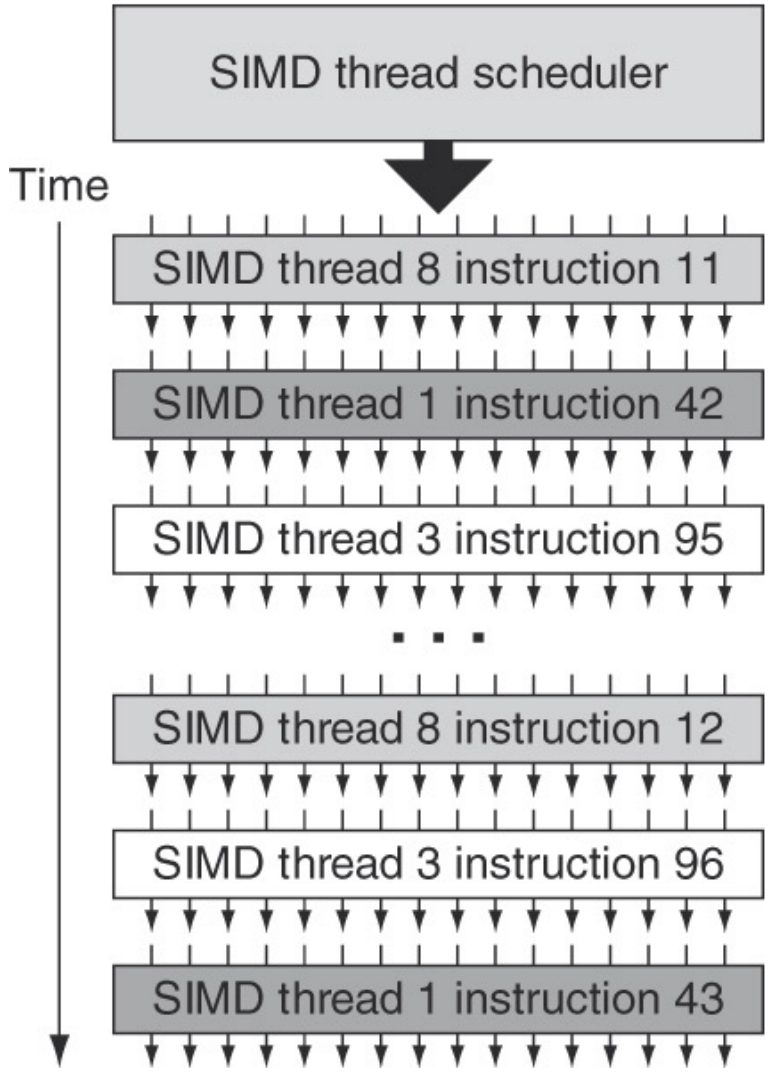


GPU ISA and Compilation

- 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

Thread Scheduling

- The scheduler selects a ready thread of SIMD instructions and issues an instruction synchronously to all the SIMD Lanes executing the SIMD thread
- Because threads of SIMD instructions are independent, the scheduler can select a different SIMD thread each time



Instruction & Thread Scheduling

- In theory, all threads can be independent
- For efficiency, 32 threads packed in warps
  - Warp: set of parallel threads that execute the same instruction
    - Warp = a thread of vector instructions
    - Warps introduce data parallelism
  - 1 warp instruction keeps cores busy for multiple cycles

Instruction & Thread Scheduling

- Individual threads may be inactive
  - Because they branched differently
  - This is the equivalent of conditional execution (but implicit)
  - Loss of efficiency if not data parallel
- Software thread blocks mapped to warps
  - When hardware resources are available

Handling Branch Divergence

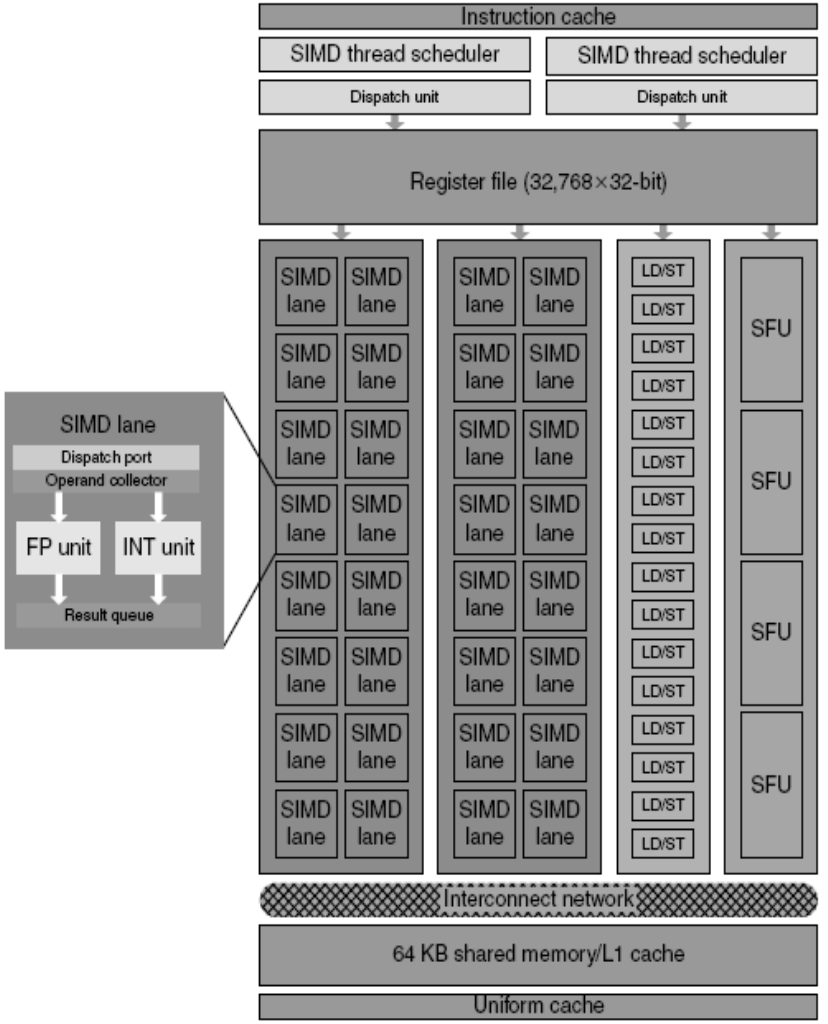
- Similar to vector processors, but masks are handled internally
  - Per-warp stack stores PCs and masks of non-taken paths
- On a conditional branch
  - Push the current mask onto the stack
  - Push the mask and PC for the non-taken path
  - Set the mask for the taken path
- At the end of the taken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

Handling Branch Divergence

- 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
- Dynamic Warp Formation
  - Dynamically merge threads executing the same instruction (after branch divergence)
  - Form new warp at divergence
  - Enough threads branching to each path to create full new warps

Fermi Architecture

- Each SIMD processor has
  - Two SIMD thread schedulers, two instruction dispatch units
  - 16 SIMD lanes (SIMD width=32, chime=2 cycles), 16 load-store units, 4 special function units
  - Thus, two threads of SIMD instructions are scheduled every two clock cycles
- Fast double precision
  - 515 GFLOPs for DAXPY
- Caches for GPU memory: I/D L1/SIMD proc and shared L2
  - 64-bit addressing and unified address space - C/C++ ptrs
- Error correcting codes
- Faster context switching
- Faster atomic instructions



Fermi streaming multiprocessor (SM)

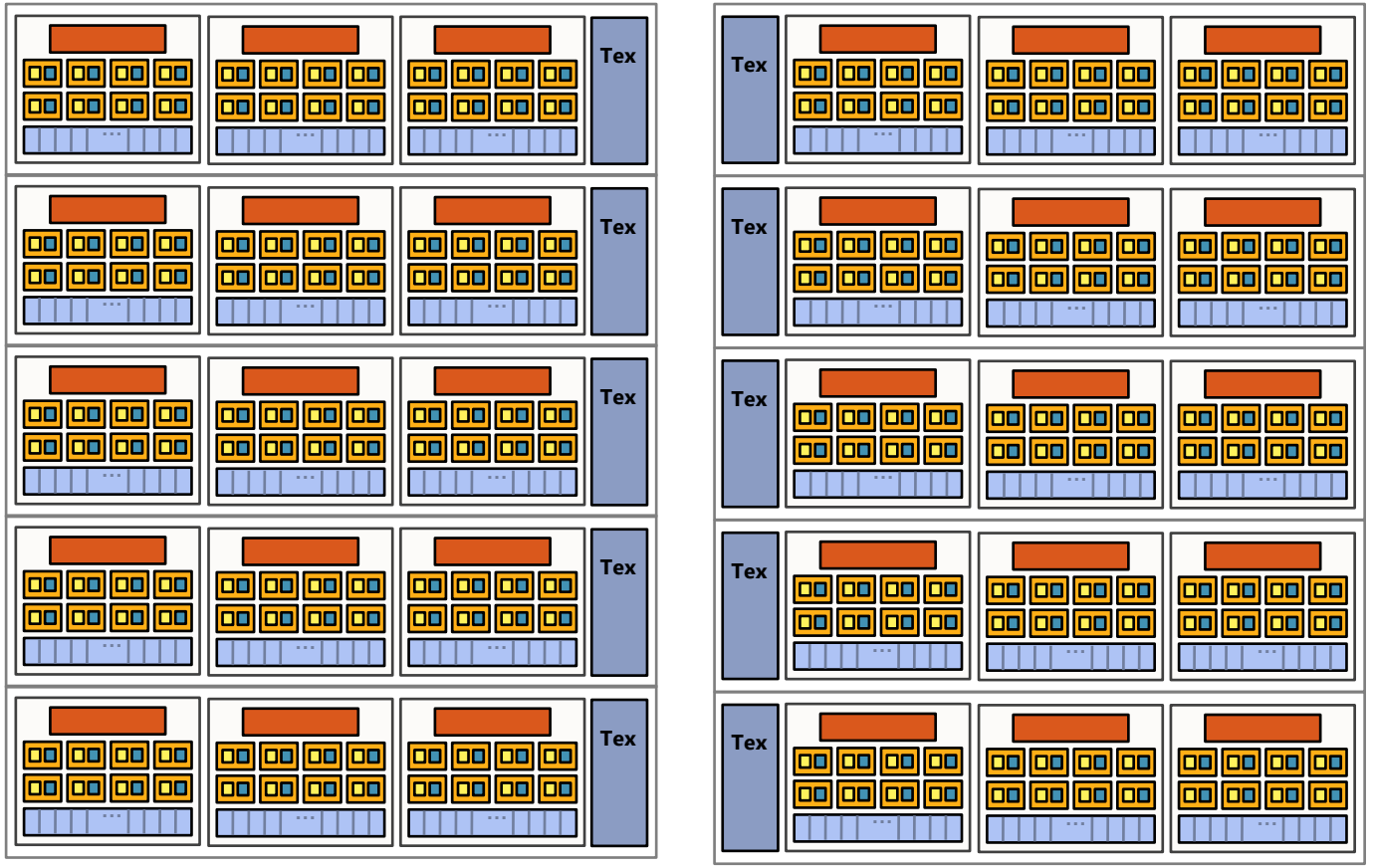
Kepler GK110 Architecture

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K
Max X Grid Dimension	2^16-1	2^16-1	2^32-1	2^32-1
Hyper-Q	No	No	No	Yes
Dynamic Parallelism	No	No	No	Yes

- Kepler GK110 Full chip block diagram
  - Highly multithreaded multicore chip
  - 15 cores or streaming
  - multiprocessors (SMX)

NVIDIA GeForce GTX 285 Core

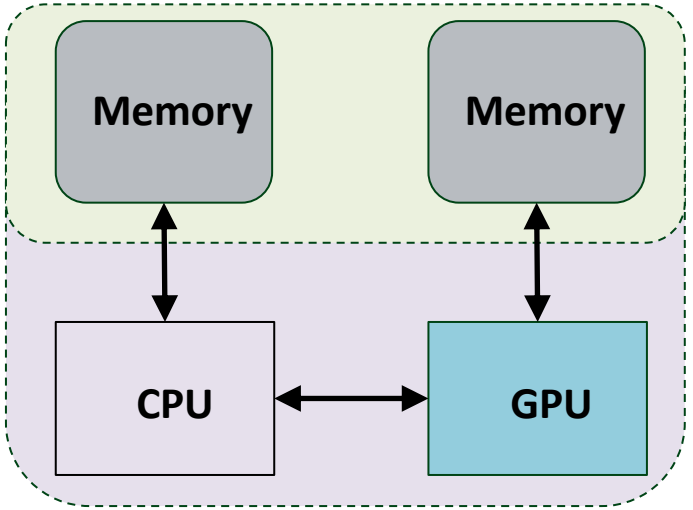
- 30 Cores
  - 30,720 threads
- Each Core
  - Groups of 32 threads share instruction stream (each group is a Warp)
  - Up to 32 warps are simultaneously interleaved
  - Up to 1024 thread contexts can be stored



GPU Execution Model

- Data transfers can dominate execution time
- Integrated GPUs with unified address space
  - No copies

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



Program Execution

- Same instruction in different threads uses thread id to index and access different data elements

GPU code

```
for (int i = 0; i < 100; i++) {
C[i] = A[i] + B[i];
}
```

Program Execution

- Same instruction in different threads uses thread id to index and access different data elements

GPU code

```
for (int i = 0; i < 100; i++) {
C[i] = A[i] + B[i];
}
```

CUDA code

```
// there are 100 threads
__global__ void KernelFunction(…) {
  int tid = blockDim.x * blockIdx.x + threadIdx.x;
  int varA = aa[tid];
  int varB = bb[tid];
  C[tid] = varA + varB;
}
```

Loop-Level Parallelism

- Focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations
  - Loop-carried dependence
- Example 1

```
for (i=999; i>=0; i=i-1)
 x[i] = x[i] + s;
```

- No loop-carried dependence

Loop-Level Parallelism

- Example 2

```
for (i=0; i<100; i=i+1) {
 A[i+1] = A[i] + C[i]; /* S1 */
 B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

- S1 and S2 use values computed by S1 in previous iteration
- S2 uses value computed by S1 in same iteration

Loop-Level Parallelism

- Example 3

```
for (i=0; i<100; i=i+1) {
 A[i] = A[i] + B[i];    /* S1 */
 B[i+1] = C[i] + D[i]; /* S2 */
}
```

- S1 uses value computed by S2 in previous iteration but dependence is not circular so loop is parallel
- Transform to:

```
A[0] = A[0] + B[0];
for (i=0; i<99; i=i+1) {
 B[i+1] = C[i] + D[i];
 A[i+1] = A[i+1] + B[i+1];
}
B[100] = C[99] + D[99];
```

Loop-Level Parallelism

- Example 4

```
for (i=0;i<100;i=i+1)  {
 A[i] = B[i] + C[i];
 D[i] = A[i] * E[i];
}
```

  - No loop-carried dependence

- Example 5

```
for (i=1;i<100;i=i+1)  {
 Y[i] = Y[i-1] + Y[i];
}
```

  - Loop-carried dependence in the form of *recurrence*

Next Class

- Memory Organization