EEL 5764 Computer Architecture

Lecture 34-35: GPU

Sandip Ray
Department of Electrical and Computer Engineering
University of Florida

Graphical Processing Units

CUDA is an elegant solution to the problem of representing parallelism in algorithms, not all algorithms, but enough to matter. It seems to resonate in some way with the way we think and code, allowing an easier, more natural expression of parallelism beyond the task level.

Vincent Natol

"Kudos for CUDA," *HPC Wire* (2010)

Graphical Processing Units

- Given the hardware invested to do graphics well, how can we supplement it to improve performance of a wider range of applications?

- Basic idea:
  - → Heterogeneous execution model
    - ➤ CPU is the *host*, GPU is the *device*
  - → Develop a C-like programming language for GPU
  - → Unify all forms of GPU parallelism as *CUDA thread*
  - → Programming model is "Single Instruction Multiple Thread"

Sample C-Code and CUDA Code

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 Thread 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];
}
```

Threads and Blocks

- A **thread** (sequence of SIMD lane operations) is associated with each data element
- Threads are organized into **blocks** (body of vectorizable loop)
- Blocks are organized into a **grid** (vectorizable loop)

- GPU hardware handles thread management (assignment of instructions to SIMD lanes), not applications or OS

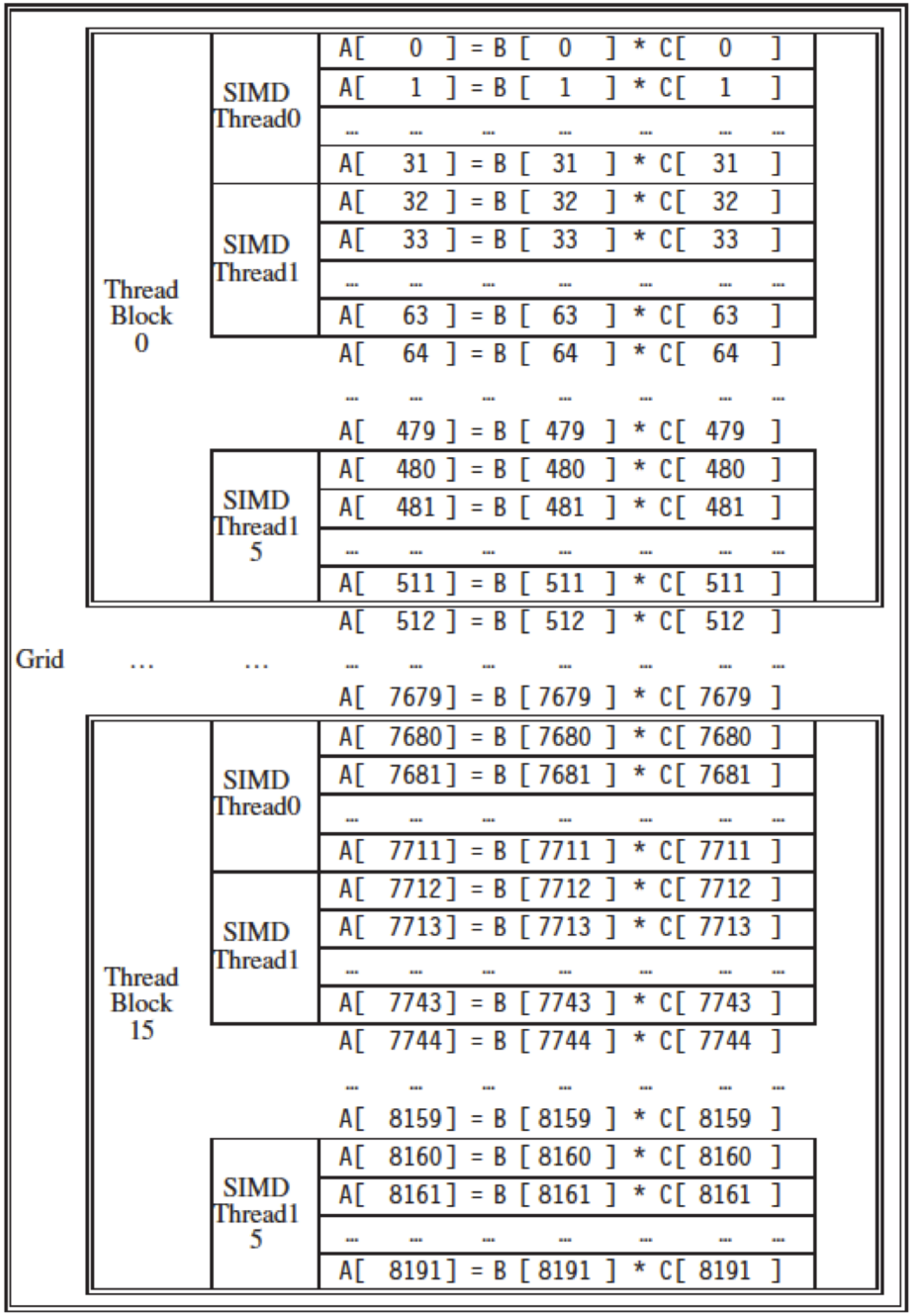


Figure 4.13 The mapping of a Grid (vectorizable loop), Thread Blocks (SIMD basic blocks), and threads of SIMD instructions to a vector–vector multiply, with each vector being 8192 elements long. Each thread of SIMD instruc-

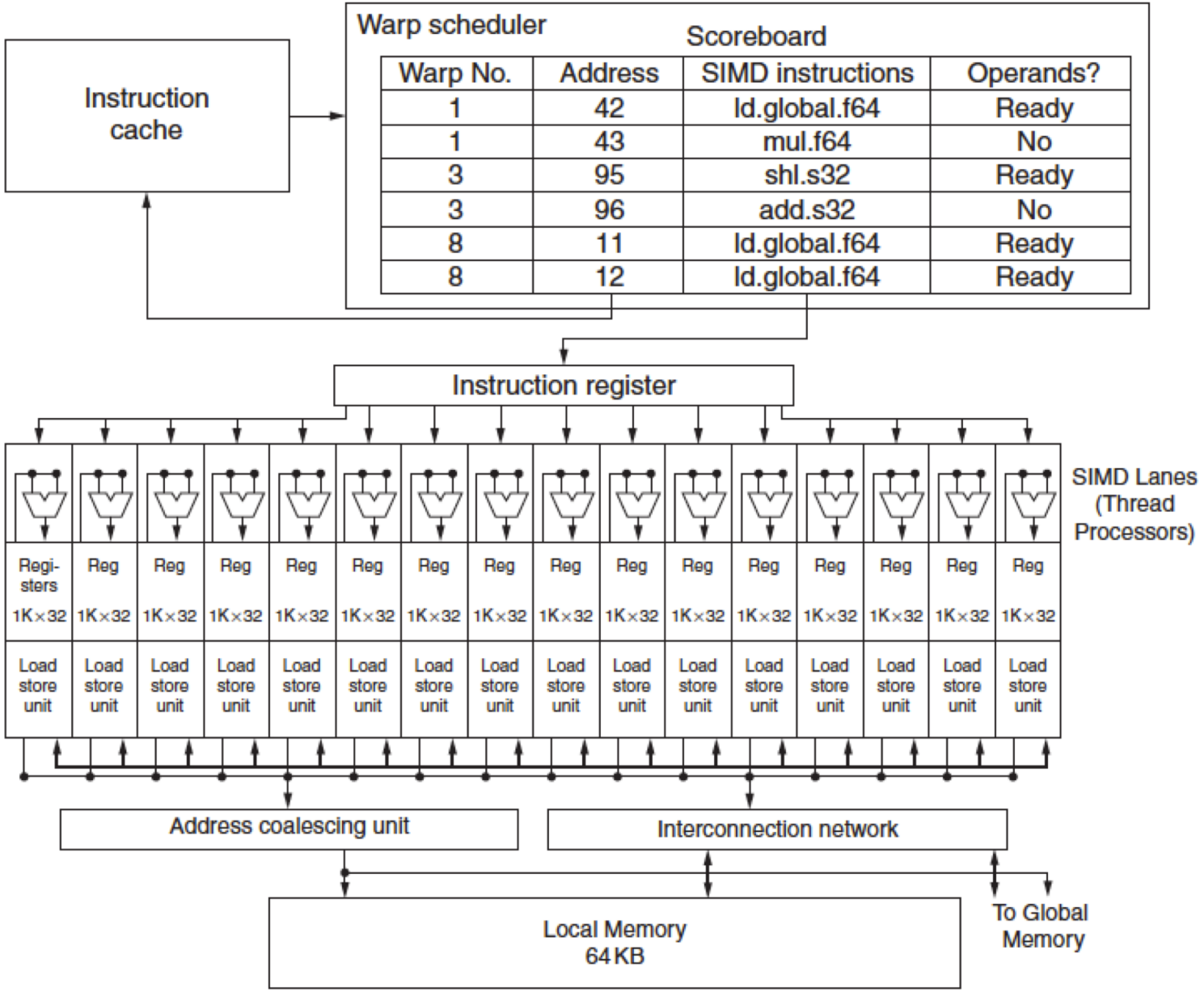


Figure 4.14 Simplified block diagram of a Multithreaded SIMD Processor. It has 16 SIMD lanes. The SIMD Thread Scheduler has, say, 48 independent threads of SIMD instructions that it schedules with a table of 48 PCs.

<u>Warp</u>
Threads of SIMD instructions that can execute OOO

GPU

- Started from graphic accelerators
- Going mainstream
  - → Can’t abandon promises to graphic
- Use SIMT model
  - → Single instruction multiple thread
  - → SIMD not exposable to programmer
- One common language CUDA
  - → Computer unified device architecture

GPU Architecture

- Similarities to vector machines:
  - → Works well with data-level parallel problems
  - → Scatter-gather transfers are automatic
  - → Mask registers
  - → Large register files
- Differences:
  - → No scalar processor
  - → Uses multithreading to hide memory latency
    - ➤ No real priority in instruction scheduling wrt ordering
  - → Has many functional units (shallow pipelining, as opposed to a few deeply pipelined units like a vector processor

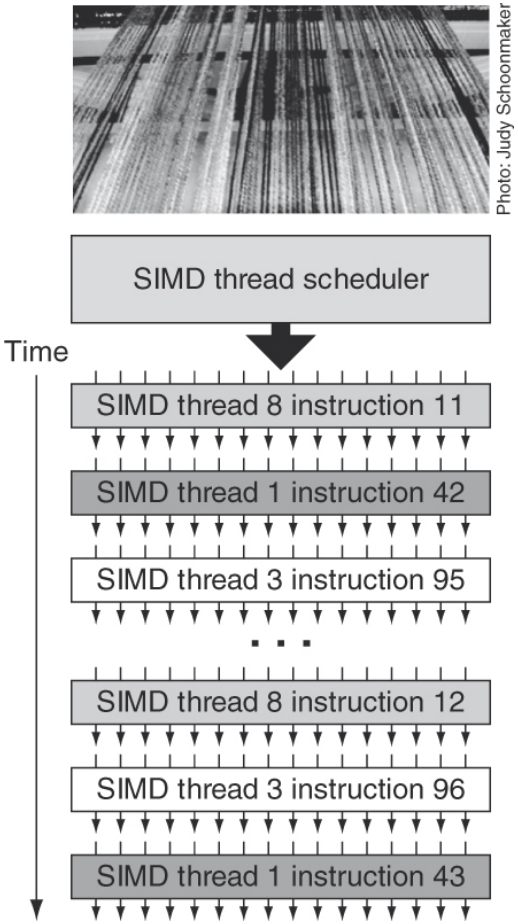


Figure 4.16 Scheduling of threads of SIMD instructions. The scheduler selects a ready thread of SIMD instructions and issues an instruction synchronously to all the SIMD Lanes executing the SIMD thread. Because threads of SIMD instructions are independent, the scheduler may select a different SIMD thread each time.

Terminology

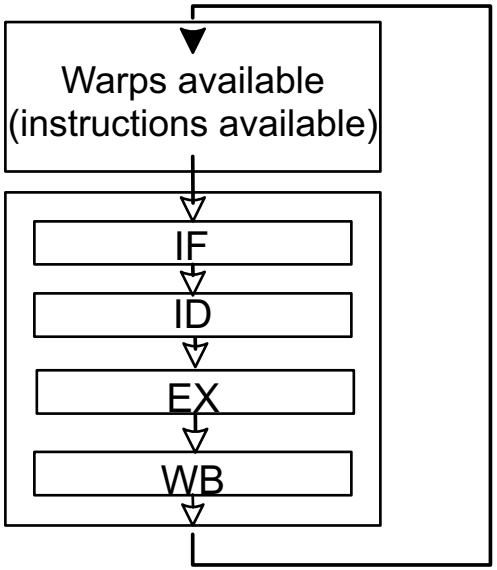
- Grid
  - → Code that works over all elements
  - → Broken down into blocks (in manageable sizes)
- Blocks
  - → Group of threads
  - → Analogous to a strip-mined vector loop
  - → Assigned to a *multithreaded SIMD processor* by the *thread block scheduler*
- Threads
  - → Composed of SIMD instructions
  - → Independent of each other
  - → Can communicate with threads in block only

Terminology contd.

- SIMD instructions
  - → Each has its own program counter
- SIMS thread scheduler
  - ➤ Identical to thread scheduler in conventional processor
  - ➤ Uses scoreboard to dispatch instructions
- Thread block scheduler schedules blocks to SIMD processors
- Within each SIMD processor:
  - → 16 SIMD lanes
  - → Wide and shallow compared to vector processors
- Warp
  - Threads of SIMD instructions that can execute OOO

Pipelining a warp

- Instructions can be pipelined
- Stages vary based on the pipeline design
- Additional consideration
  - One instruction in warp can stall all instruction



Example (2012)

- NVIDIA GPU has 32,768 registers
  - → Divided into SIMD lanes
  - → Each SIMD thread is limited to 64 registers
  - → SIMD thread has up to:
    - ➤ 64 vector registers of 32 32-bit elements
    - ➤ 32 vector registers of 32 64-bit elements
  - → Fermi has 16 physical SIMD lanes, each containing 2048 registers

Example

- AMD Radeon Pro Duo Series
  - → Released to the public April 2016
  - → Two Fiji XT GPUs each with a frequency of up to 1 GHz
  - → 8192 stream processors
  - → 8GB HBM (high bandwidth memory) total (4GB per GPU)
  - → Memory interface 8192 bits total (4096b per GPU)
  - → Memory frequency of 500 MHz
  - → Effective mem speed of 1 Gbps
  - → Memory bandwidth of 1 TB/s
  - → 32b floating point operations: 16.38 TFLOPs
  - → TDP (thermal design power) of 350 W

Example

- A NVIDIA GTX1080 GP104 GPU has:
  - → 20 streaming multiprocessors (SM)
  - → 4 processing blocks per SM
  - → Each block has 32 CUDA cores (32 thread warps)
  - → Single warp scheduler per block with two dispatch ports per scheduler
  - → (128 x 20) = 2,560 CUDA cores (single precision ALUs) total
  - → 16K registers per block
  - → (20 * 4 * 16k) = 1,280,000 registers
  - → (256KB x 20) = 5MB of total register file capacity
  - → 8 memory controllers
  - → 8GB DDR5X with 256-bit memory interface
  - → Thermal design power of 180 W

Example

- PASCAL GPU Architecture.(Used by NVIDIA)
  - → Max warps per SIMD processor: 64
  - → Max threads per SIMD processor : 2048
  - → Max thread blocks per SIMD processor : 32
  - → Max 32 bit registers: 65536
  - → CUDA cores: 1024

NVIDIA Instruction Set Arch.

- ISA is an abstraction of the hardware instruction set
  - → "Parallel Thread Execution (PTX)" – stable ISA for compilers and compatibility across GPU generations
  - → Uses virtual registers – compiler determines physical registers
  - → Translation to machine code is performed in software
    - ➤ Compared to in hardware for x86

Instruction format

- Opcode.type d, a, b, c;
  - → d, destination, a, b, c are source operands
  - → Can be 32 bit or 64 bit or constant
- Type
  - → Untyped bits
  - → Unsigned integer
  - → Signed integer
  - → Floating point

NVIDIA Instruction Set Arch.

- Example for one loop iteration (1 CUDA thread per iteration)
- These first three instructions calculate the unique element offset in R8
- No branching or incrementing, just directly accesses each element
- Creates 8192 CUDA threads per iteration

```
shl.s32     R8, blockIdx, 9     ; Thread Block ID * Block size (512 or 2^9)
                                 - shift left signed 32-bit int, blockIdx unique per
                                 block
add.s32     R8, R8, threadIdx   ; R8 = i = CUDA thread ID
                                - add 32-bit signed int, threadIdx unique per
                                thread
shl.s.32    R8, R8, 3           ; R8 = byte offset of element
```

NVIDIA Instruction Set Arch.

- Load the operands, two double-precision floating point values
- Multiply, add the, and store sum

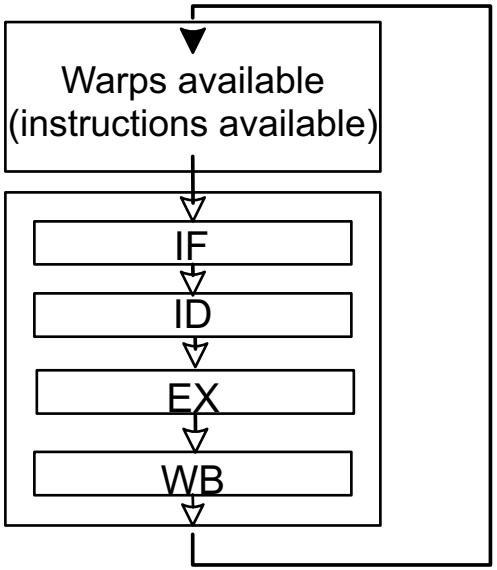
```
ld.global.f64      RD0, [X+R8]          ; RD0 = X[i]
ld.global.f64      RD2, [Y+R8]          ; RD2 = Y[i]
                                        load from memory space
mul.f64 R0D, RD0, RD4      ; Product in RD0 = RD0 * RD4 (scalar a)
                                        multiply floating point
add.f64 R0D, RD0, RD2      ; Sum in RD0 = RD0 + RD2 (Y[i])
                                        add floating point
st.global.f64 [Y+R8], RD0  ; Y[i] = sum (X[i]*a + Y[i])
                                        store 64-bit floating point to memory
```

NVIDIA Instruction Set Arch.

- Unlike vector machines, no separate instructions for
  - → Sequential data transfers
  - → Strided data transfers
  - → Gather-scatter data transfers
- All data transfers are gather-scatter
  - → Uses special Address Coalescing hardware to recognize when SIMD lane within a thread of SIMD instructions are collectively issuing sequential addresses
  - → Runtime hardware signals memory interface to request a block transfer of 32 sequential words
  - → GPU programmer only must ensure that adjacent CUDA threads access nearby addresses at the same time that can be coalesced into one or a few memory or cache blocks

Pipelining a warp

- Instructions can be pipelined
- Stages vary based on the pipeline design
- Additional consideration
  - One instruction in warp can stall all instruction



Conditional Branching

- Like vector architectures, GPU branch hardware uses internal masks
  - → SIMD lanes have code for both branch outcomes, then masks the incorrect branch outcome to suppress commit
- Also uses
  - → Branch synchronization stack
    - ➤ Entries consist of masks for each SIMD lane, stack allows nesting
    - ➤ I.e. which threads commit their results (all threads execute) based on mask and conditional outcome
  - → Instruction markers to manage when a branch diverges into multiple execution paths
    - ➤ Push on divergent branch
  - → …and when paths converge
    - ➤ Act as barriers
    - ➤ Pops stack

Conditional Branching

- Per-thread-lane 1-bit predicate register, specified by programmer
  - → So have to share this bit with all nested outstanding branches
- Inefficient
  - → For equal length paths, an IF-THEN-ELSE operates at 50% efficiency
    - ➤ Half of lanes executing incorrect instructions
  - → Doubly nested IF statements with equal-length paths run at 25% efficiency
  - → Triply nested at 12.5% efficiency, and so on
  - → Worse for not equal length paths
    - ➤ Have to wait for longest path

Example

```
if (X[i] != 0)
        X[i] = X[i] – Y[i];
else X[i] = Z[i];

ld.global.f64      RD0, [X+R8]                  ; RD0 = X[i]
setp.neq.s32       P1, RD0, #0          ; P1 is predicate register 1
@!P1, bra          ELSE1, *Push                 ; Push old mask, set new mask bits
                                                ; if P1 false, go to ELSE1
ld.global.f64      RD2, [Y+R8]                  ; RD2 = Y[i]
sub.f64            RD0, RD0, RD2                ; Difference in RD0
st.global.f64      [X+R8], RD0                  ; X[i] = RD0
@P1, bra           ENDIF1, *Comp                ; complement mask bits
                                                ; if P1 true, go to ENDIF1
ELSE1:             ld.global.f64 RD0, [Z+R8]    ; RD0 = Z[i]
                   st.global.f64 [X+R8], RD0    ; X[i] = RD0
ENDIF1:    <next instruction>, *Pop       ; pop to restore old mask
```

NVIDIA GPU Memory Structures

- Each SIMD Lane has private section of off-chip DRAM
  - → "Private memory"
  - → Contains stack frame, spilling registers, and private variables
- Each multithreaded SIMD processor also has local memory
  - → Shared by SIMD lanes / threads within a block
- Memory shared by SIMD processors is GPU Memory
  - → CPU host can read and write GPU memory

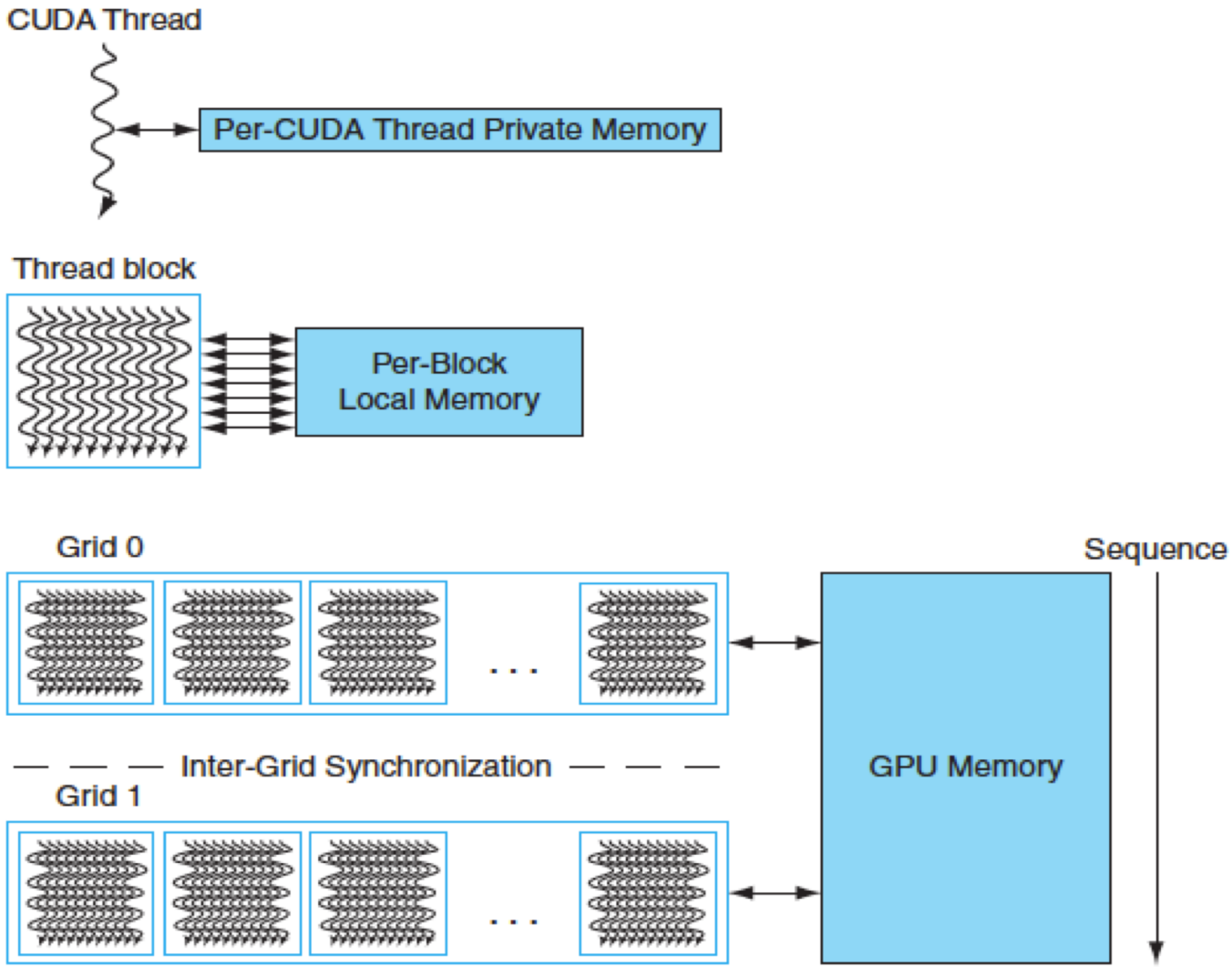


Figure 4.18 GPU Memory structures. GPU Memory is shared by all Grids (vectorized loops), Local Memory is shared by all threads of SIMD instructions within a thread block (body of a vectorized loop), and Private Memory is private to a single CUDA Thread.

Fermi Architecture

- Each SIMD processor has
  - → Two SIMD thread schedulers, two instruction dispatch units
  - → 2 sets of 16 SIMD lanes (SIMD width=32, chime=2 cycles), 16 load-store units, or 4 special function units
  - → Thus, two threads of SIMD instructions are scheduled every two clock cycles
  - → Threads are independent (same warp) so no data dependencies
- Caches for GPU memory
  - → 16KB L1cache, 768KB L2 cache
- 64-bit addressing and unified address space
- Error correcting codes for memory and registers
- Faster context switching, 25 microseconds which is 10x faster than predecessor
- Faster atomic instructions
  - → Special hardware in L2 cache

Vector Processors vs. GPUs

Type	Vector term	Closest CUDA/NVIDIA GPU term	Comment
Program abstractions	Vectorized Loop	Grid	Concepts are similar, with the GPU using the less descriptive term.
	Chime	--	Since a vector instruction (PTX Instruction) takes just two cycles on Fermi and four cycles on Tesla to complete, a chime is short in GPUs.
Machine objects	Vector Instruction	PTX Instruction	A PTX instruction of a SIMD thread is broadcast to all SIMD Lanes, so it is similar to a vector instruction.
	Gather/Scatter	Global load/store (ld.global/st.global)	All GPU loads and stores are gather and scatter, in that each SIMD Lane sends a unique address. It's up to the GPU Coalescing Unit to get unit-stride performance when addresses from the SIMD Lanes allow it.
	Mask Registers	Predicate Registers and Internal Mask Registers	Vector mask registers are explicitly part of the architectural state, while GPU mask registers are internal to the hardware. The GPU conditional hardware adds a new feature beyond predicate registers to manage masks dynamically.

Processing and memory hardware	Vector Processor	Multithreaded SIMD Processor	These are similar, but SIMD Processors tend to have many lanes, taking a few clock cycles per lane to complete a vector, while vector architectures have few lanes and take many cycles to complete a vector. They are also multithreaded where vectors usually are not.
	Control Processor	Thread Block Scheduler	The closest is the Thread Block Scheduler that assigns Thread Blocks to a multithreaded SIMD Processor. But GPUs have no scalar-vector operations and no unit-stride or strided data transfer instructions, which Control Processors often provide.
	Scalar Processor	System Processor	Because of the lack of shared memory and the high latency to communicate over a PCI bus (1000s of clock cycles), the system processor in a GPU rarely takes on the same tasks that a scalar processor does in a vector architecture.
	Vector Lane	SIMD Lane	Both are essentially functional units with registers.
	Vector Registers	SIMD Lane Registers	The equivalent of a vector register is the same register in all 32 SIMD Lanes of a multithreaded SIMD Processor running a thread of SIMD instructions. The number of registers per SIMD thread is flexible, but the maximum is 64, so the maximum number of vector registers is 64.
	Main Memory	GPU Memory	Memory for GPU versus System memory in vector case.

Vector Processors vs. GPUs

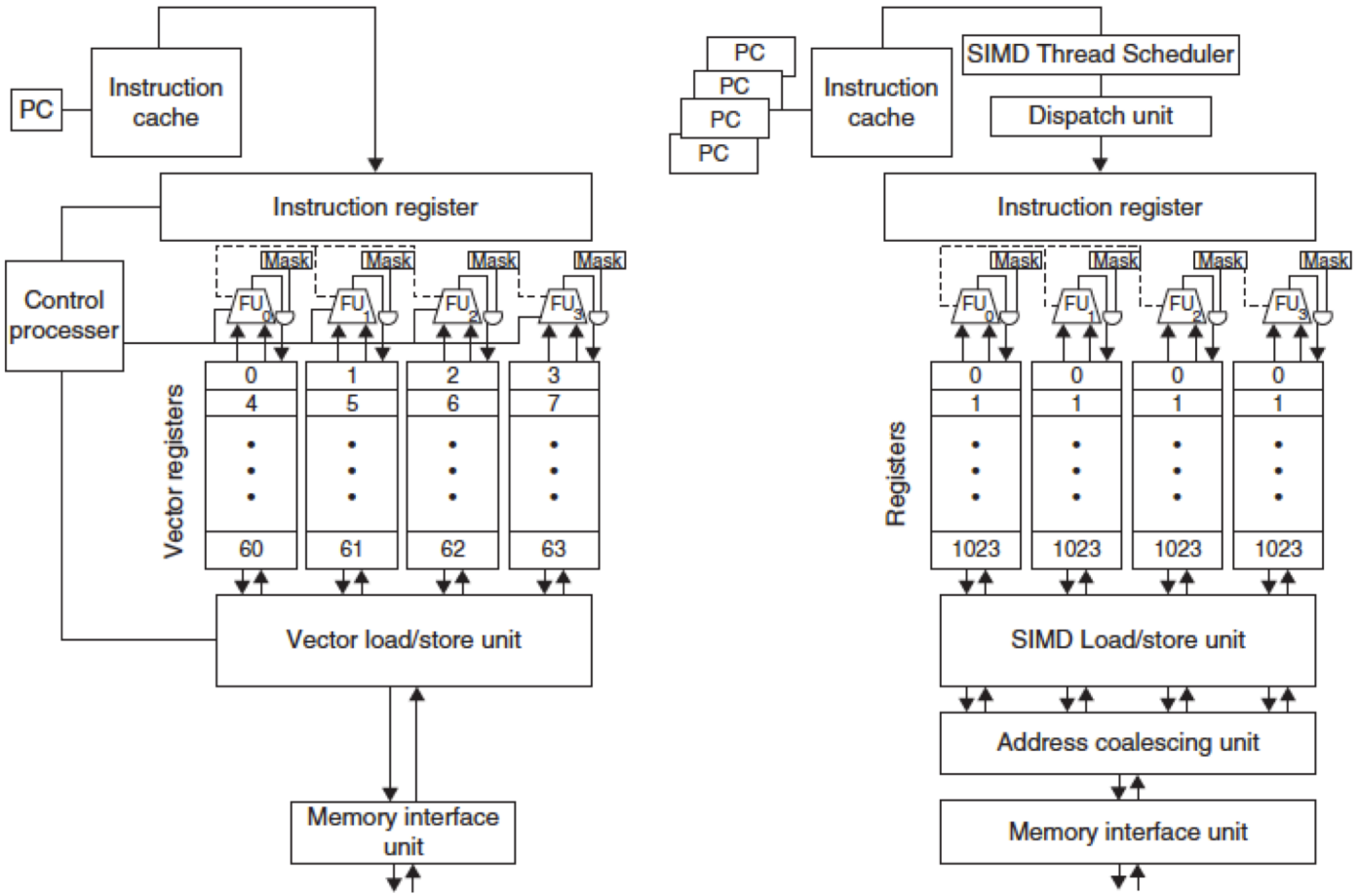


Figure 4.22 A vector processor with four lanes on the left and a multithreaded SIMD Processor of a GPU with four SIMD Lanes on the right. (GPUs typically have 8 to 16 SIMD Lanes.)

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 */
}
```

- S2 uses value computed by S1 in same iteration
  - → Does not affect parallelism <u>if it was</u> the only dependency
    - ➤ Can be alleviated with chaining if necessary
- S1 and S2 use values computed in previous iteration
  - → Compute i+1 in iteration i
  - → Loop carried, forces serial execution

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];
}
```

  → If you reference the register into which A is computed, rather than load

- Example 5:

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

  → Need to detect if recurrence exists. Occurs when variable value is defined based on the value of that variable in an earlier iteration.

Finding dependencies

- Assume indices are affine, which means a 1D array index can be written as :
  - → $a \times i + b$
  - → a and b are constants and i is the loop index
- Assume loop that:
  - → Stores to $a \times i + b$, then
  - → Loads from same array at $c \times i + d$
  - → i runs from m to n
  - → A dependence exists if:
    - There are two iteration indices, j and k, that are both within the limits of the loop
      j, k such that $m \leq j \leq n$, $m \leq k \leq n$
    - Store to $a \times j + b$, load from $c \times k + d$, and $a \times j + b = c \times k + d$

Finding dependencies

- Generally cannot determine at compile time
- Test for absence of a dependence:
  - → E.g., GCD test:
    - ➤ If a dependency exists, GCD(*c*,*a*) must evenly divide (*d*-*b*)
- Example:

```
for (i=0; i<100; i=i+1) {
   X[2*i+3] = X[2*i] * 5.0;
}
```

Given: a =2, b=3, c=2, d=0.

GCD(a,c)=2, and d-b = -3

Since 2 does not divide -3, no dependence is possible

Finding dependencies

- Example 2 with multiple dependencies

```
for (i=0; i<100; i=i+1) {
    Y[i] = X[i] / c; /* S1 */
    X[i] = X[i] + c; /* S2 */
    Z[i] = Y[i] + c; /* S3 */
    Y[i] = c - Y[i]; /* S4 */
}
```

- Watch for antidependencies and output dependencies
- True dependencies
  - → (s1, s3), (s1,s4) due to Y[i], not loop carried so iterations are parallel but forces s3, s4 to wait on S1
- Anti-dependencies
  - → s1 to s2, based on x[i]; s3 to s4 based on y[i]
- Output dependencies
  - → S1 to s4 based on y[i]

Eliminating dependent computation

- Reduction Operation, iterations not parallel due to variable *sum*:

```
for (i=9999; i>=0; i=i-1)
        sum = sum + x[i] * y[i];
```

- Transform to…

```
for (i=9999; i>=0; i=i-1)
        sum [i] = x[i] * y[i];
for (i=9999; i>=0; i=i-1)
        finalsum = finalsum + sum[i];
```

- Do on p processor:

```
for (i=999; i>=0; i=i-1)
        finalsum[p] = finalsum[p] + sum[i+1000*p];
```

- Note: assumes associativity!