Lecture 14
GPUs

Daniel Sanchez and Joel Emer

[Incorporates material from Kozyrakis (EE382A), NVIDIA Kepler Whitepaper, Hennesy&Patterson]

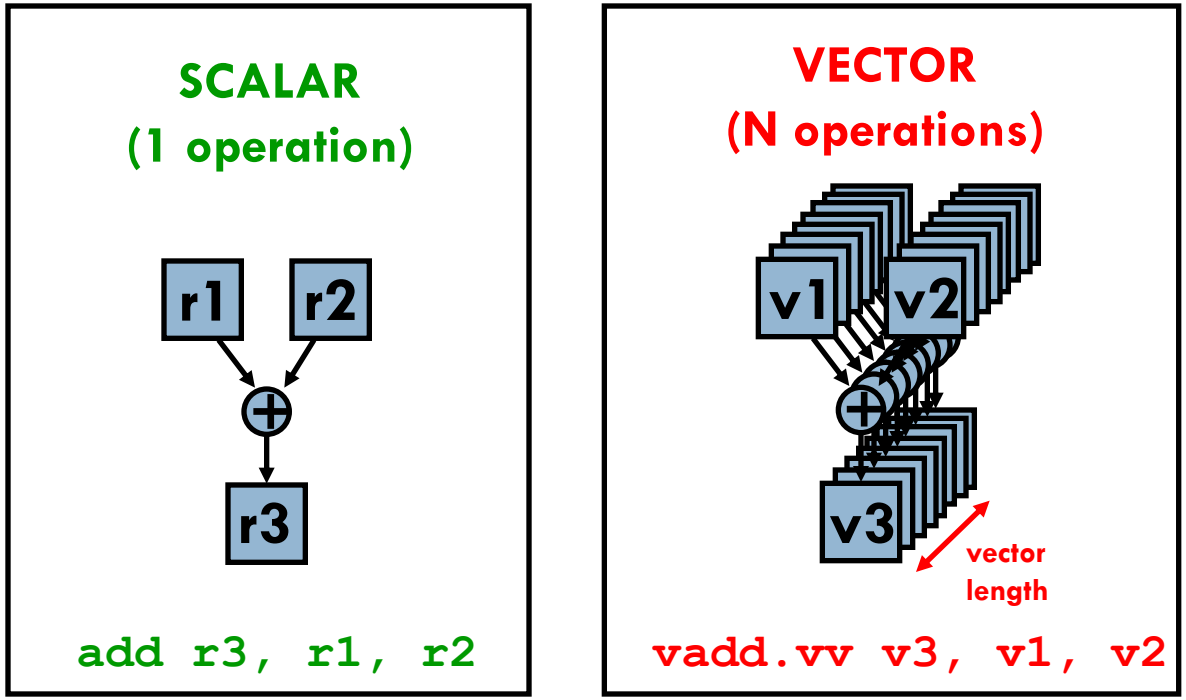
6.888 Parallel and Heterogeneous Computer Architecture
Spring 2013

Massachusetts Institute of Technology

Today's Menu

- Review of vector processors
- Basic GPU architecture
- Paper discussions

Vector Processors

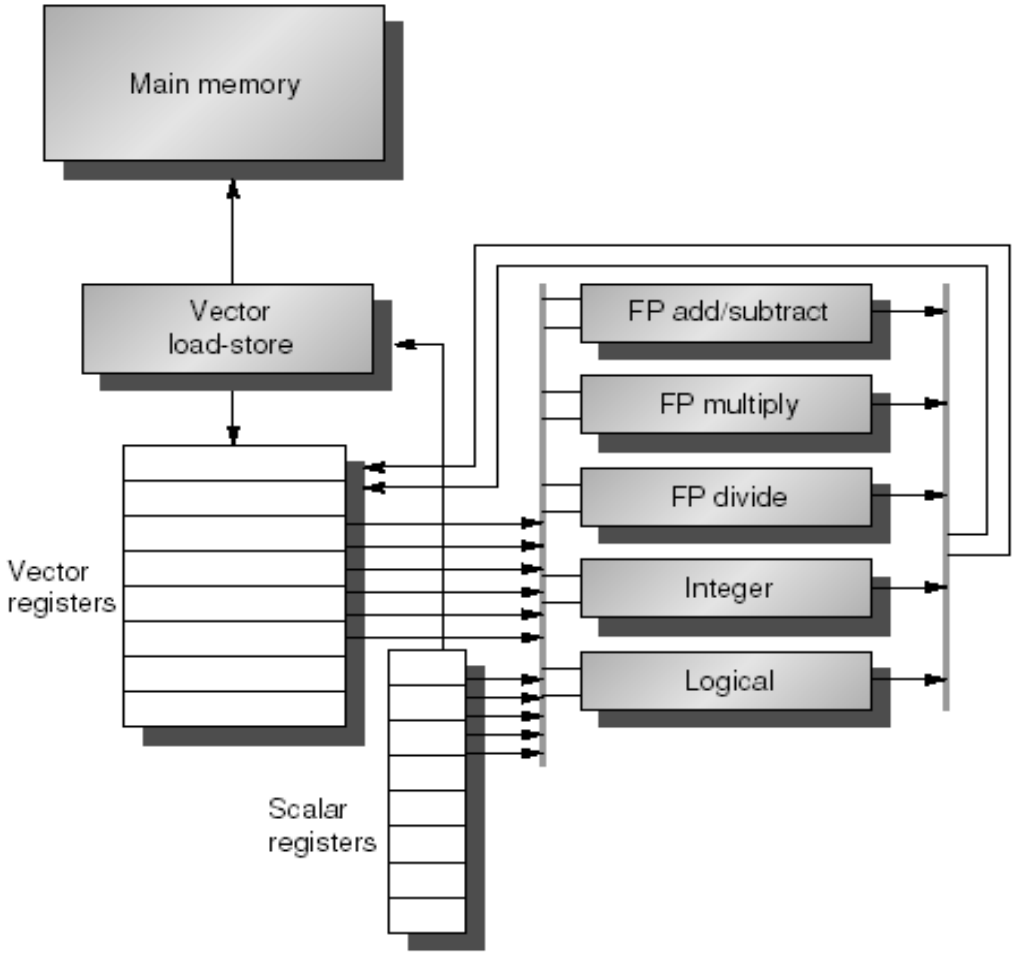


- Scalar processors operate on single numbers (scalars)
- Vector processors operate on linear sequences of numbers (vectors)

What's in a Vector Processor?

- A scalar processor (e.g. a MIPS processor)
  - Scalar register file (32 registers)
  - Scalar functional units (arithmetic, load/store, etc)
- A vector register file (a 2D register array)
  - Each register is an array of elements
    - E.g. 32 registers with 32 64-bit elements per register
  - MVL = maximum vector length = max # of elements per register
- A set of vector functional units
  - Integer, FP, load/store, etc
  - Some times vector and scalar units are combined (share ALUs)

Example of Simple Vector Processor



Basic Vector ISA

Instr.	Operands	Operation	Comment
VADD.**VV**	V1,V2,V3	V1=V2+V3	vector + vector
VADD.**SV**	V1,**R0**,V2	V1=**R0**+V2	scalar + vector
VMUL.VV	V1,V2,V3	V1=V2*V3	vector x vector
VMUL.SV	V1,R0,V2	V1=R0*V2	scalar x vector
VLD	V1,R1	V1=M[R1...R1+63]	load, stride=1
VLD**S**	V1,R1,**R2**	V1=M[R1...R1**+63*R2**]	load, stride=R2
VLD**X**	V1,R1,**V2**	V1=M[R1**+V2**$_i$,i=0..63]	indexed("gather")
VST	V1,R1	M[R1...R1+63]=V1	store, stride=1
VST**S**	V1,R1,**R2**	V1=M[R1...R1**+63*R2**]	store, stride=R2
VST**X**	V1,R1,**V2**	V1=M[R1**+V2**$_i$,i=0..63]	indexed("scatter")

+ regular scalar instructions…

Advantages of Vector ISAs

- Compact: single instruction defines N operations
  - Amortizes the cost of instruction fetch/decode/issue
  - Also reduces the frequency of branches
- Parallel: N operations are (data) parallel
  - No dependencies
  - No need for complex hardware to detect parallelism (similar to VLIW)
  - Can execute in parallel assuming N parallel datapaths
- Expressive: memory operations describe patterns
  - Continuous or regular memory access pattern
  - Can prefetch or accelerate using wide/multi-banked memory
  - Can amortize high latency for 1st element over large sequential pattern

Vector Length (VL)

- Basic: Fixed vector length (typical in narrow SIMD)
  - Is this efficient for wide SIMD (e.g., 32-wide vectors)?

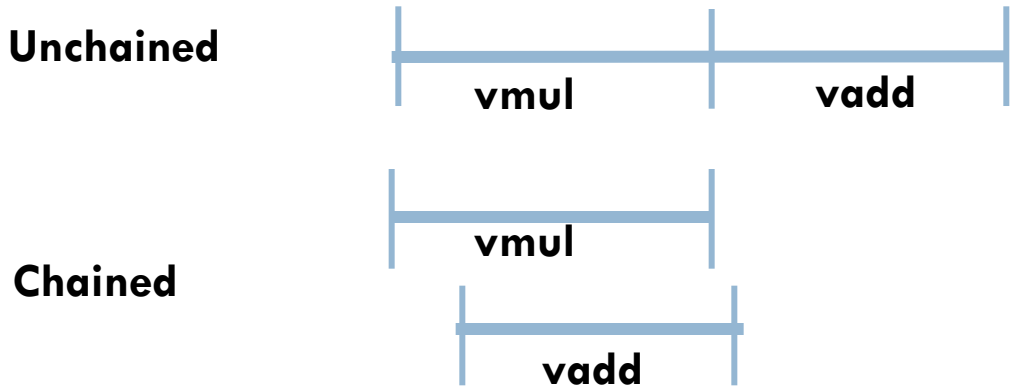
- Vector-length (VL) register: Control the length of any vector operation, including vector loads and stores
  - e.g. vadd.vv with VL=10 ←→ for (i=0; i<10; i++) V1[i]=V2[i]+V3[i]
  - VL can be set up to MVL (e.g., 32)
  - How to do vectors > MVL?
  - What if VL is unknown at compile time?

Optimization 1: Chaining

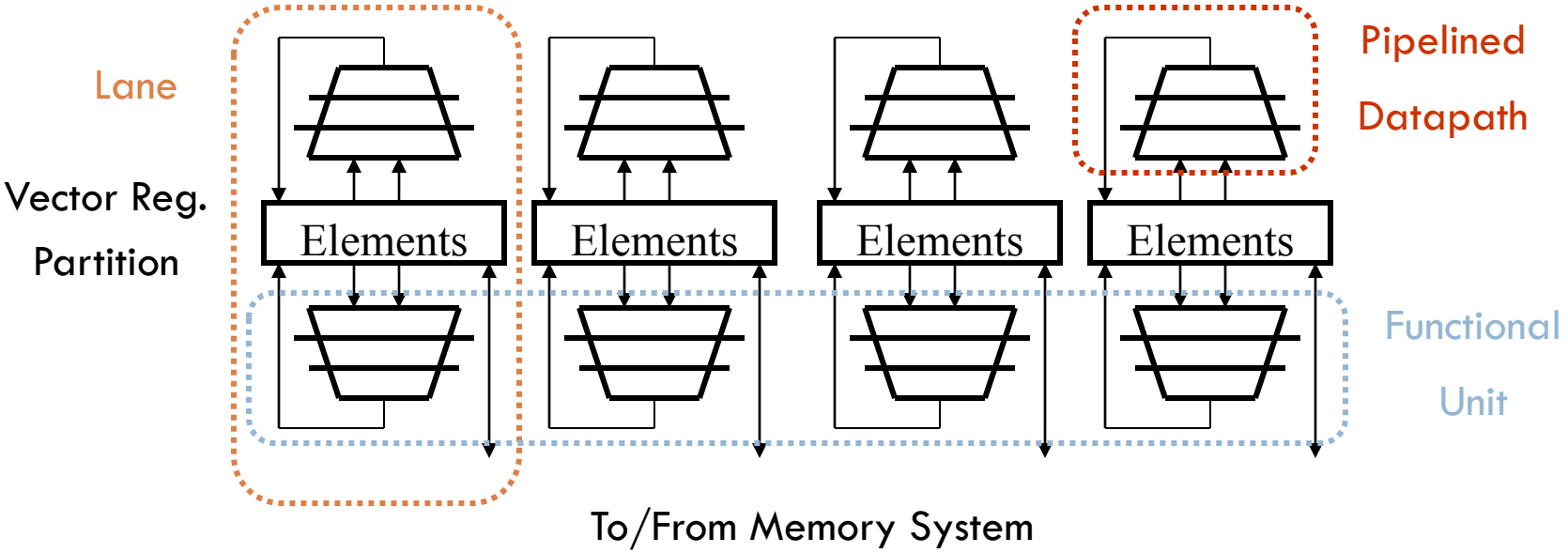
- Suppose the following code with VL=32:

```
vmul.vv    V1,V2,V3
vadd.vv    V4,V1,V5      # very long RAW hazard
```

- Chaining
  - V1 is not a single entity but a group of individual elements
  - Pipeline forwarding can work on an element basis
- Flexible chaining: allow vector to chain to any other active vector operation => more read/write ports

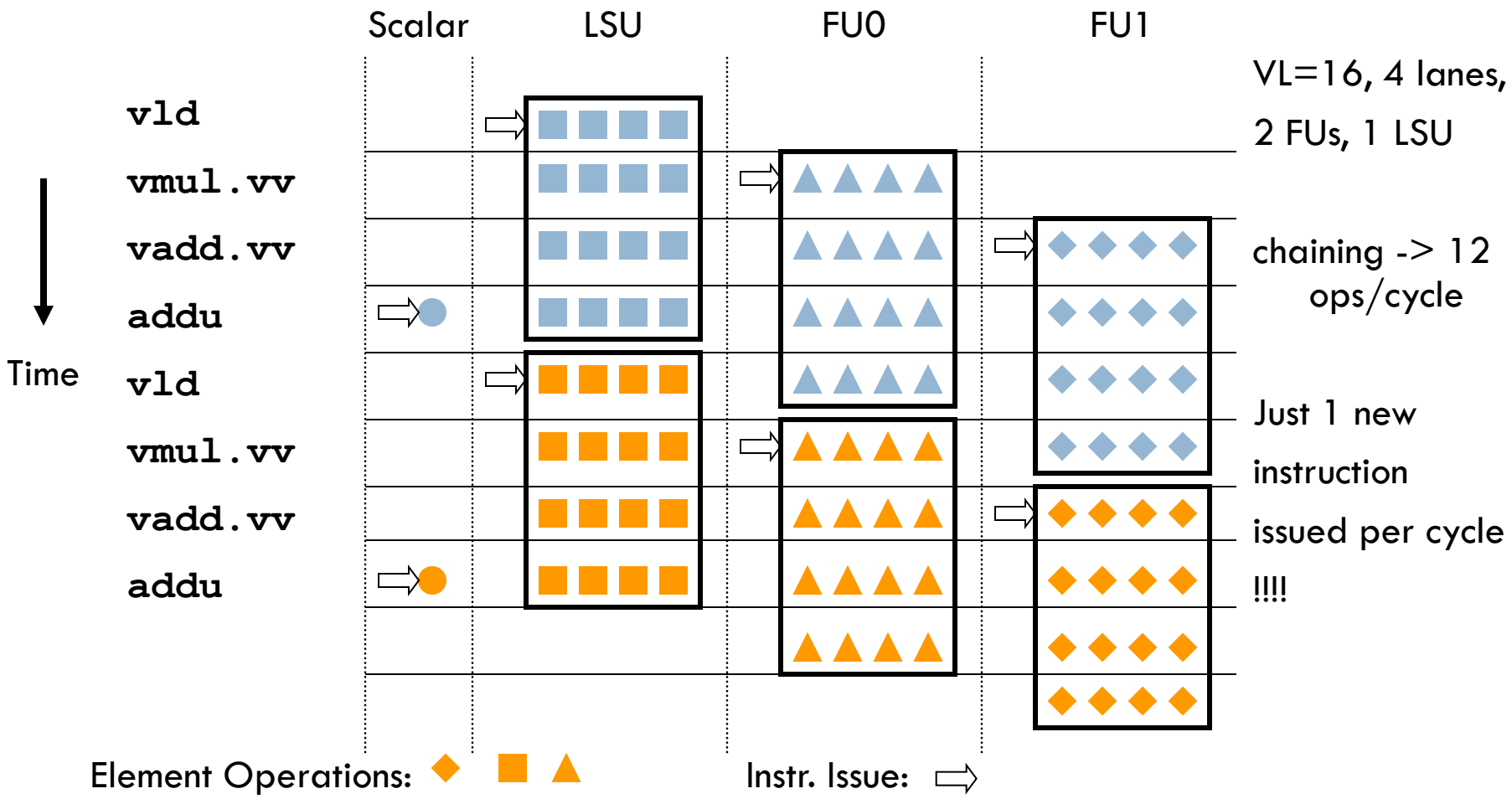


Optimization 2: Multiple Lanes



- Modular, scalable design
  - Elements for each vector register interleaved across the lanes
  - Each lane receives identical control
  - Multiple element operations executed per cycle
  - No need for inter-lane communication for most vector instructions

Chaining & Multi-lane Example



Optimization 3: Conditional Execution

- Suppose you want to vectorize this:

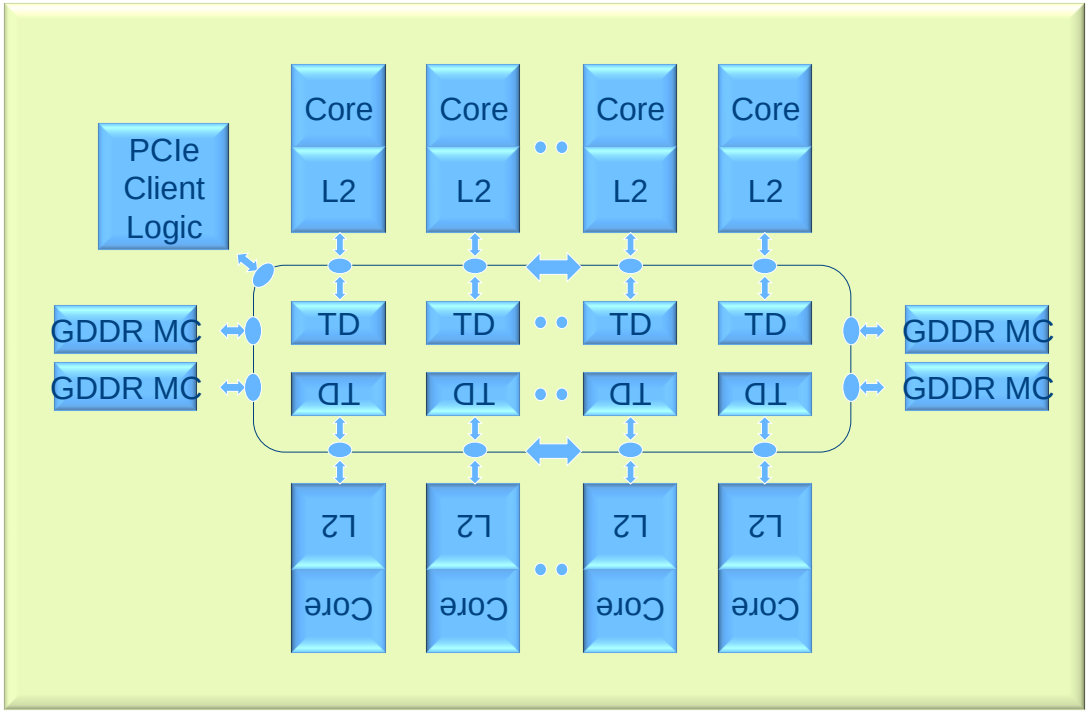
```
for (i=0; i<N; i++) if (A[i]!= B[i]) A[i] -= B[i];
```

- Solution: Vector conditional execution (predication)
  - Add vector flag registers with single-bit elements (masks)
  - Use a vector compare to set the a flag register
  - Use flag register as mask control for the vector sub
    - Add executed only for vector elements with corresponding flag element set

- Vector code

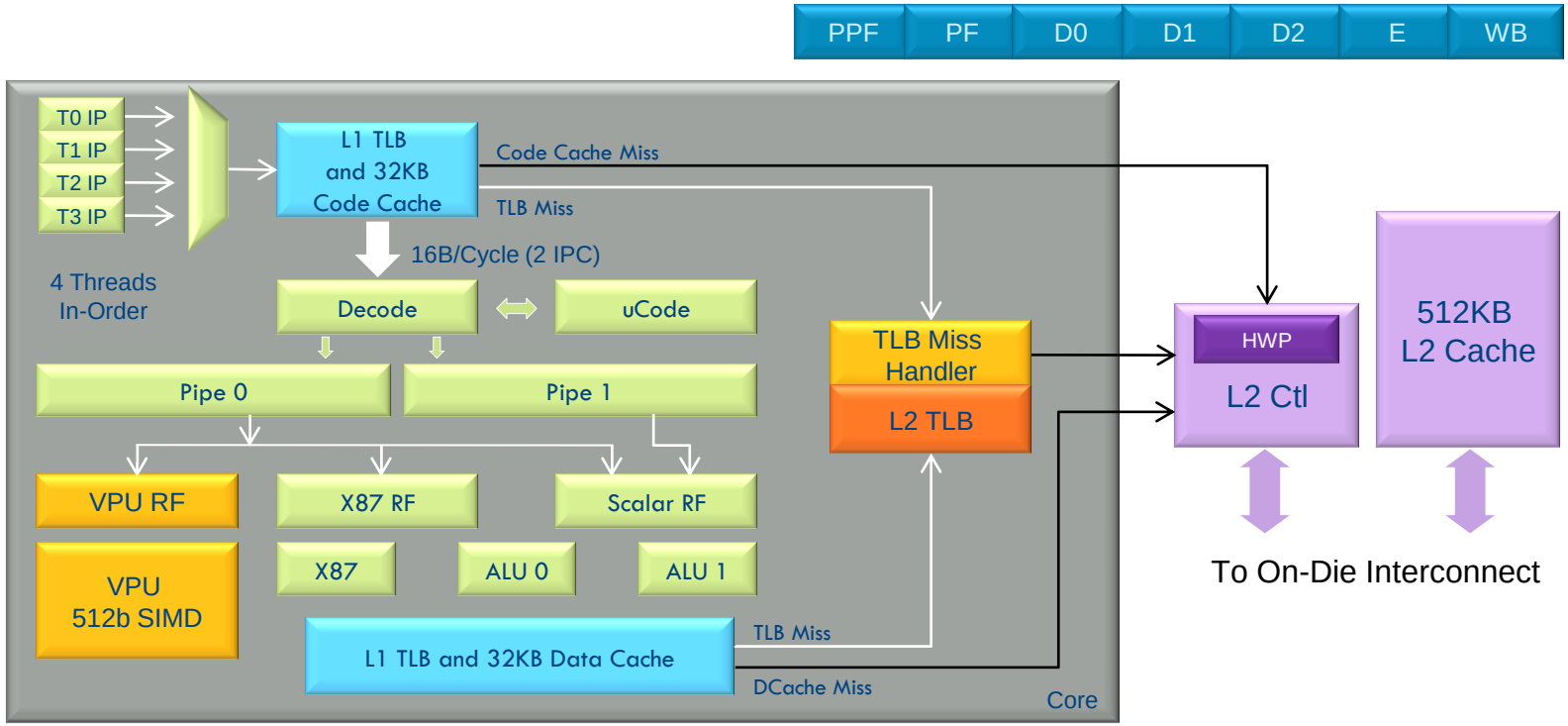
```
vld           V1, Ra
vld           V2, Rb
vcmp.neq.vv   M0, V1, V2      # vector compare
vsub.vv       V3, V2, V1, M0  # conditional vadd
vst           V3, Ra
```

Example: Intel Xeon Phi (Knights Corner)



- A multi-core chip with x86-based vector processors
  - Ring interconnect, private L2 caches, coherent
- Targeting the HPC market
  - Goal: high GFLOPS, GFLOPS/Watt

Xeon Phi Core Design



- 4-way threaded + vector processing
- In-order (why?), short pipeline
- Vector ISA: 32 vector registers (512b), 8 mask registers, scatter/gather

Graphics Processors Timeline

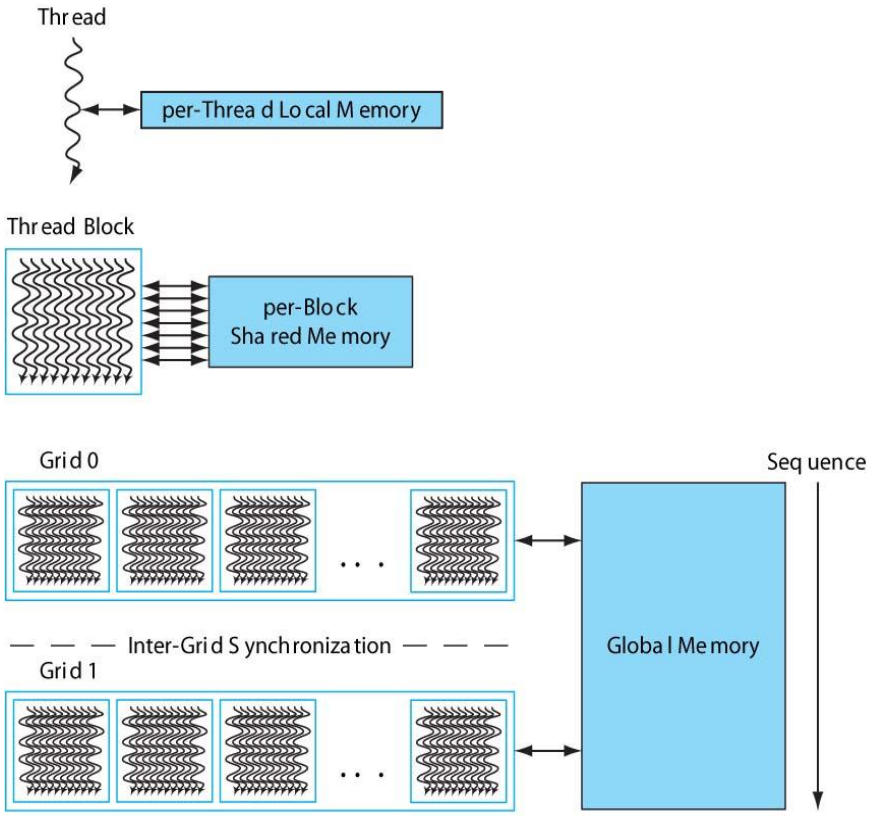
- Till mid 90s
  - VGA controllers used to accelerate some display functions
- Mid 90s to mid 00s
  - Fixed-function graphics accelerators for the OpenGL and DirectX APIs
    - Some GP-GPU capabilities by on top of the interfaces
  - 3D graphics: triangle setup & rasterization, texture mapping & shading
- Modern GPUs
  - Programmable multiprocessors optimized for data-parallel ops
    - OpenGL/DirectX and general purpose languages (CUDA, OpenCL, …)
  - Some fixed-function hardware (texture, raster ops, …)
  - Either a PCIe accelerator (discrete), or in same die as CPU (integrated)
    - Tradeoffs?

Our Focus

- GPU hardware architecture

- Good high-level mental model
  - GPU = Multicore chip, with highly-threaded vector cores
  - Not 100% accurate, but helpful as a SW developer

Refresh: Software GPU Thread Model (CUDA)



- Single-program multiple data (SPMD) model
- Each thread has local memory
- Parallel threads packed in blocks
  - Access to per-block shared memory
  - Can synchronize with barrier
- Grids include independent groups
  - May execute concurrently

Code Example: SAXPY

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

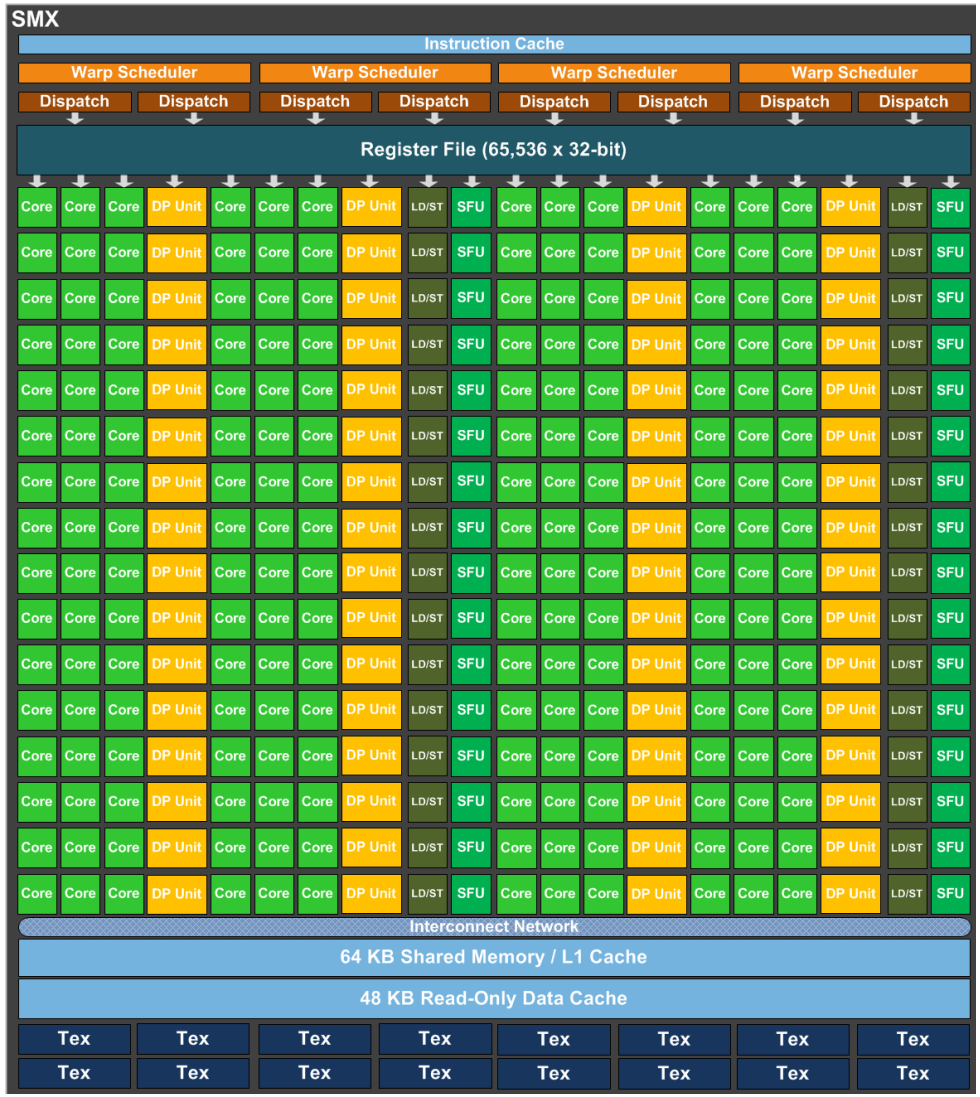
```
// Invoke DAXPY with 256 threads per block
__host__
int nblocks = (n+ 255) / 256;
    daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__device__
void daxpy(int n, double a, double *x, double *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
```

- CUDA code launches 256 threads per block
  - Thread = 1 iteration of scalar loop (1 element in vector loop)
  - Block = body of vectorized loop (with VL=256 in this example)
  - Grid = vectorizable loop

Example: Nvidia Kepler GK110

- 15 SMX processors, shared L2, 6 memory controllers
  - 1TFLOP DP
- HW thread scheduling

Streaming Multiprocessor (SMX)



- Cores are
  - Multithreded
  - Data parallel
- Capabilities
  - 64K registers
  - 192 simple cores
    - Int and SP FPU
  - 64 DP FPUs
  - 32 LSUs, 32 SFUs
- Scheduling
  - 4 warp schedulers
  - 2 inst dispatch per warp

Instruction & Thread Scheduling: Thread + Data Parallelism

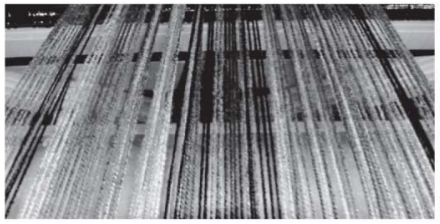
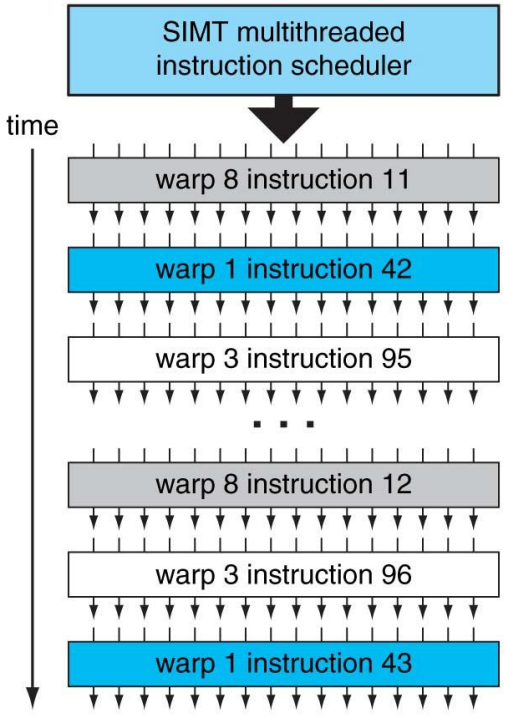
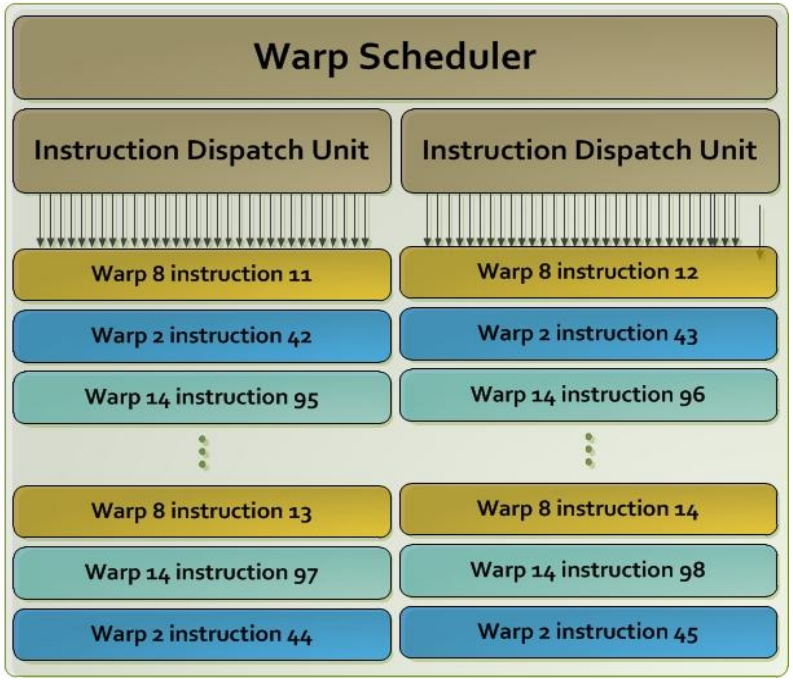


Photo: Judy Schoonmaker



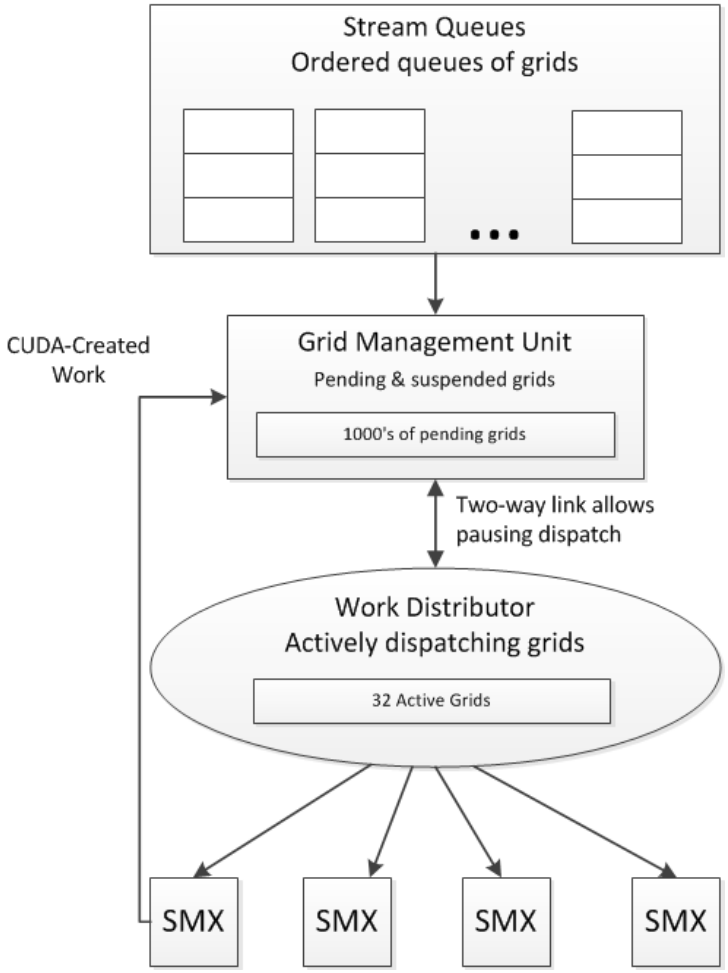
- In theory, all threads can be independent
  - HW implements zero-overhead switching
- For efficiency, 32 threads are packed in warps
  - Warp: set of parallel threads the execute same instruction
    - Wrap = a thread of vector instructions
    - Warps introduce data parallelism
  - 1 warp instruction keeps cores busy for multiple cycles
- 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
- SW thread blocks mapped to warps
  - When HW resources are available

Warp Scheduling



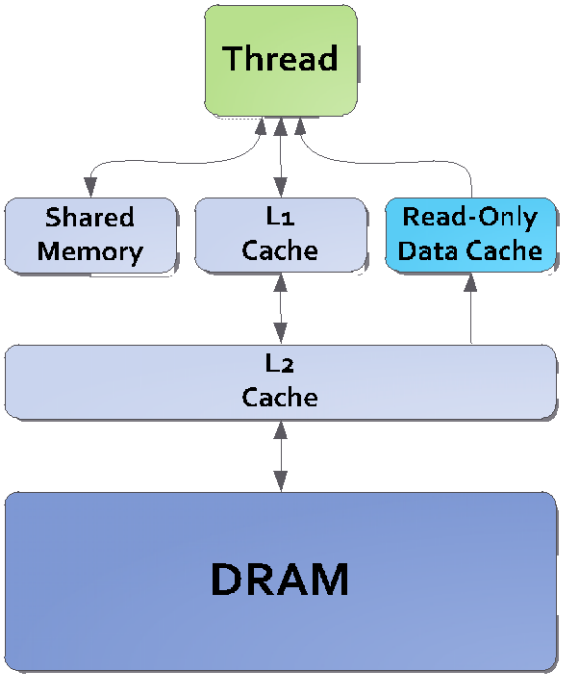
- 64 warps per SMX
- 32 threads per warp
  - 64K registers/SMX
  - Up to 255 registers per thread (8 warps)
- Scheduling
  - 4 schedulers select 1 warp per cycle
  - 2 independent instructions issued per warp (double-pumped FUs)
  - Total bandwidth = 4 * 2 * 32 = 256 ops per cycle
- Register scoreboarding
  - To track ready instructions
  - Simplified using static latencies
    - Binary incompatibility?

Hardware Scheduling



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

Memory Hierarchy

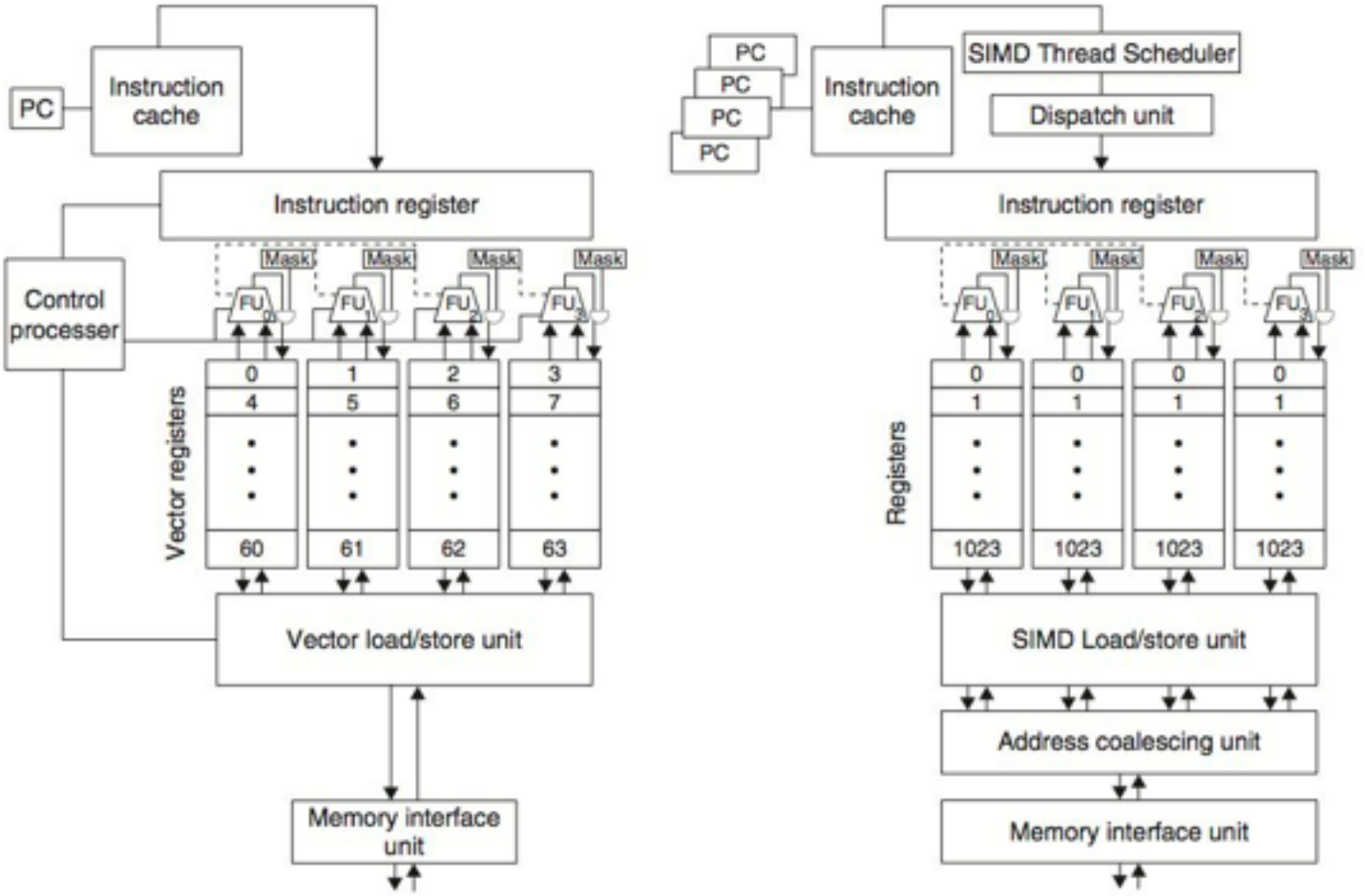


- Each SMX has 64KB of memory
  - Split between shared mem and L1 cache
    - 16/48, 32/32, 48/16
  - 256B per access
- 48KB read-only data cache
- 1.5MB shared L2
  - Supports synchronization operations (atomicCAS, atomicADD, …)
- Throughput-oriented main memory
  - Memory scheduling? TCM-like?
  - GDDRx standards

Paper Discussions

- DWF, Fung et al., MICRO'07
- RF/WS, Gebhart et al., ISCA'11

Lost in Translation: Vector vs GPU



Lost in Translation: Vector vs GPU

Type	More descriptive name	Closest old term outside of GPUs	Official CUDA/ NVIDIA GPU term	Book definition
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel.
	Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via Local Memory.
	Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register.
Machine object	A Thread of SIMD Instructions	Thread of Vector Instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask.
	SIMD Instruction	Vector Instruction	PTX Instruction	A single SIMD instruction executed across SIMD Lanes.
Processing hardware	Multithreaded SIMD Processor	(Multithreaded) Vector Processor	Streaming Multiprocessor	A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors.
	Thread Block Scheduler	Scalar Processor	Giga Thread Engine	Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors.
	SIMD Thread Scheduler	Thread scheduler in a Multithreaded CPU	Warp Scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution.
	SIMD Lane	Vector Lane	Thread Processor	A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask.
Memory hardware	GPU Memory	Main Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU.
	Private Memory	Stack or Thread Local Storage (OS)	Local Memory	Portion of DRAM memory private to each SIMD Lane.
	Local Memory	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors.
	SIMD Lane Registers	Vector Lane Registers	Thread Processor Registers	Registers in a single SIMD Lane allocated across a full thread block (body of vectorized loop).

- From Computer Architecture, 4th edition by J. Hennessy and D. Patterson