Vector Computers and GPUs

15-740 SPRING'18

NATHAN BECKMANN

BASED ON SLIDES BY DANIEL SANCHEZ, MIT

Today: Vector/GPUs

Focus on throughput, not latency

Programmer/compiler must expose parallel work directly

Works on regular, replicated codes (i.e., data parallel)

Background: Supercomputer Applications

Typical application areas

- Military research (nuclear weapons aka “computational fluid dynamics”, cryptography)
- Scientific research
- Weather forecasting
- Oil exploration
- Industrial design (car crash simulation)
- Bioinformatics
- Cryptography

All involve **huge computations** on **large data sets**

In 70s-80s, Supercomputer == Vector Machine

Vector Supercomputers

Epitomized by Cray-1, 1976:

Scalar Unit

- Load/Store Architecture

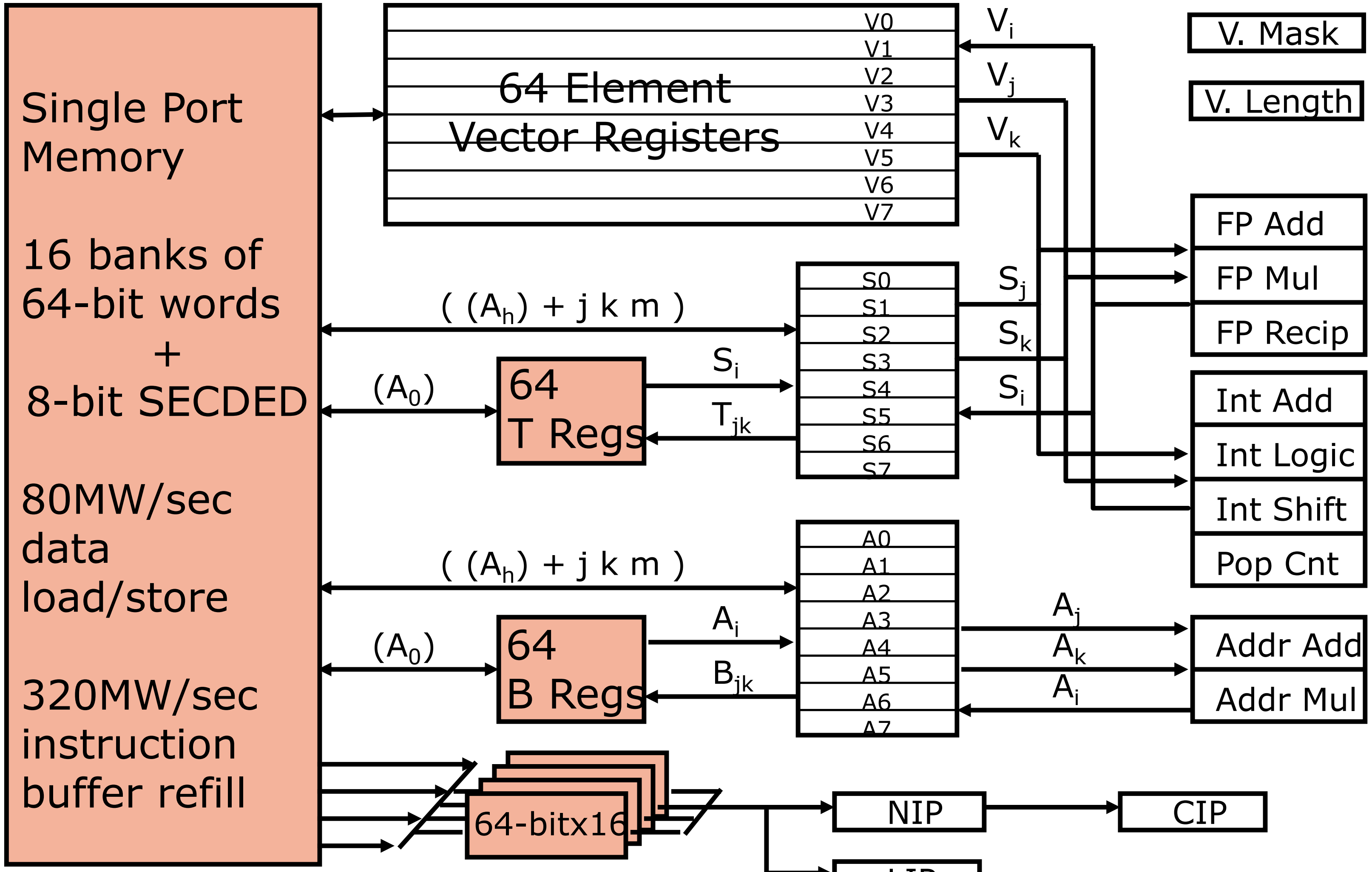
Vector Extension

- Vector Registers
- Vector Instructions

Implementation

- Hardwired Control
- Highly Pipelined Functional Units
- Interleaved Memory System
- No Data Caches
- No Virtual Memory

Cray-1 (1976)



Single Port Memory

16 banks of 64-bit words + 8-bit SECDED

80MW/sec data load/store

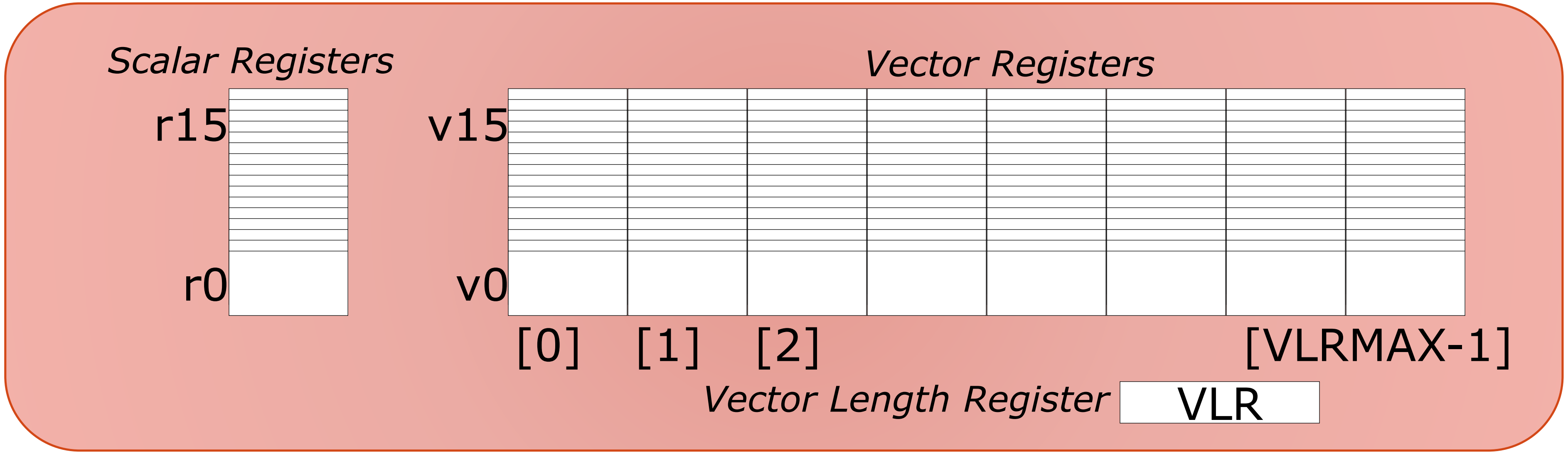
320MW/sec instruction buffer refill

memory bank cycle 50 ns

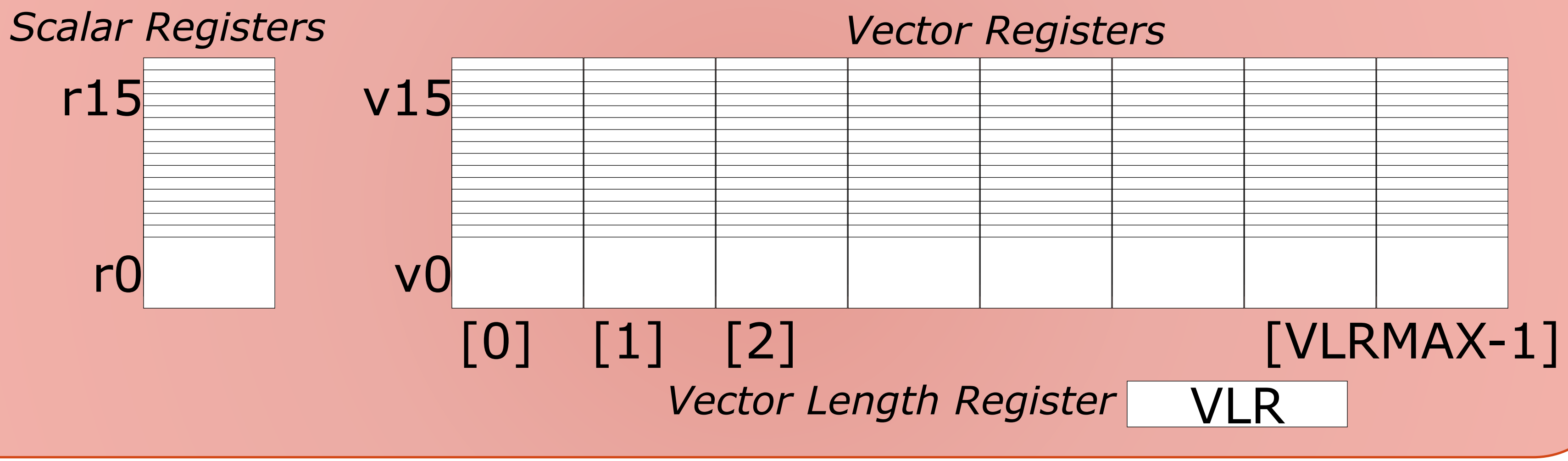
4 Instruction Buffers

processor cycle 12.5 ns (80MHz)

Vector Programming Model

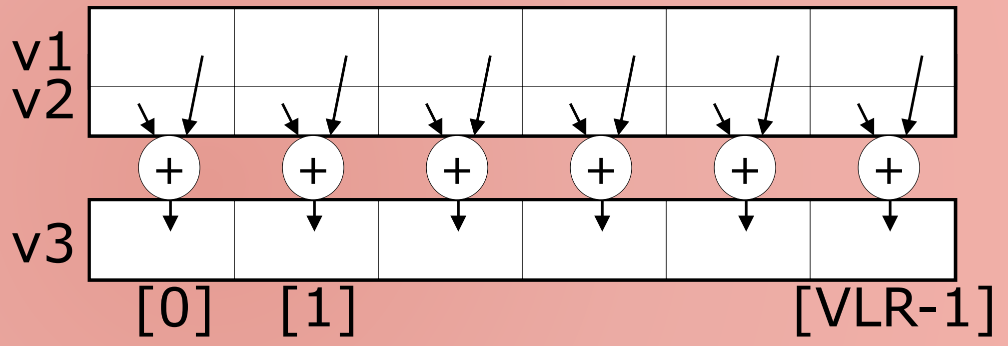


Vector Programming Model

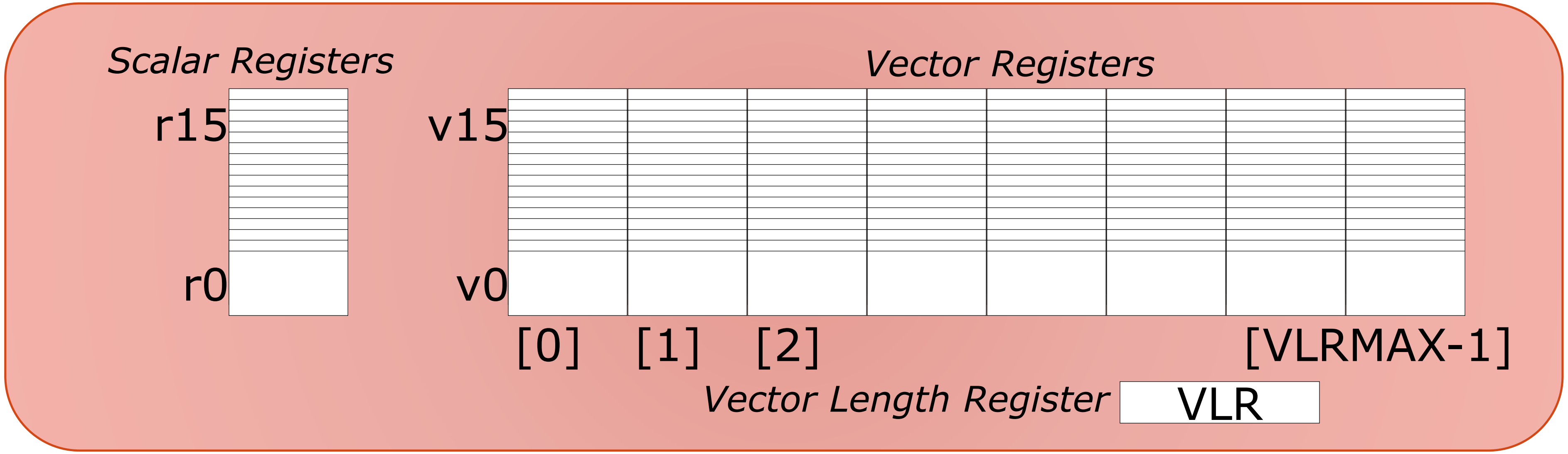


Vector Arithmetic Instructions

ADDV v3, v1, v2



Vector Programming Model



Scalar Registers

r15

r0

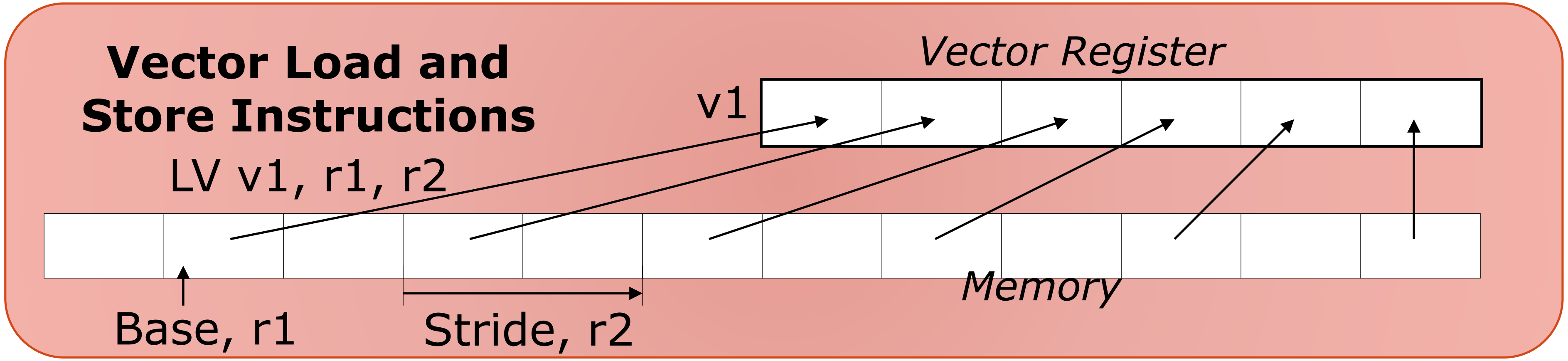
Vector Registers

v15

v0

[0] [1] [2] [VLRMAX-1]

Vector Length Register VLR



Vector Load and Store Instructions

LV v1, r1, r2

Vector Register

v1

Memory

Base, r1

Stride, r2

Vector Code Example

```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];
```

```
# Scalar Code
  LI R4, 64
loop:
  LD F0, 0(R1)
  LD F2, 0(R2)
  ADD F4, F2, F0
  ST F4, 0(R3)
  ADD R1, R1, 8
  ADD R2, R2, 8
  ADD R3, R3, 8
  SUB R4, R4, 1
  BNEZ R4, loop
```

```
# Vector Code
  LI VLR, 64
  LI R4, 4
  LV V1, R1, R4
  LV V2, R2, R4
  ADDV V3, V1, V2
  SV V3, R3, R4
```

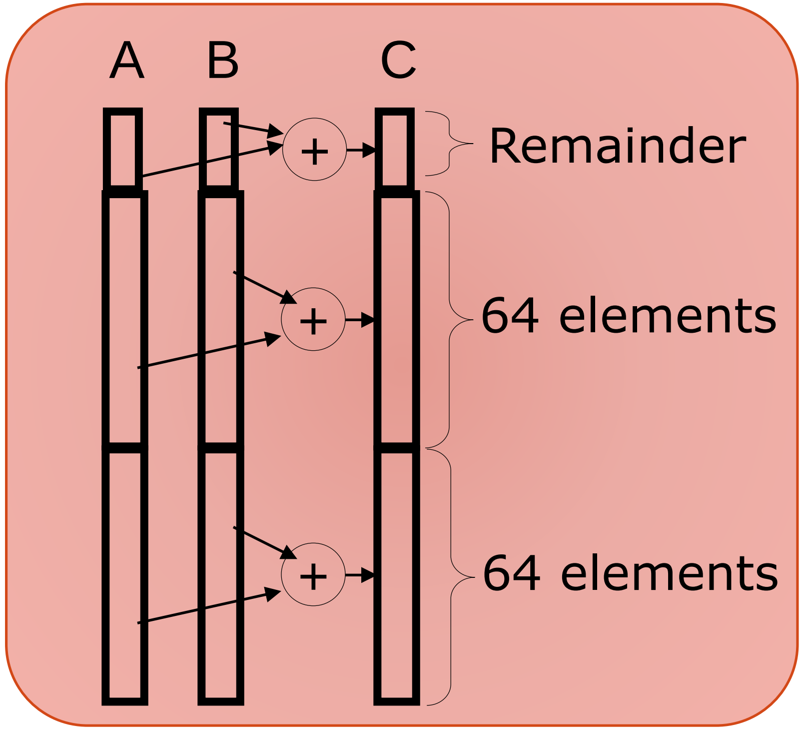
What if we want to execute larger loops than the vector registers?

Vector Stripmining

Problem: Vector registers are finite

Solution: Break loops into chunks that fit into registers, "stripmining"

```
for (i=0; i<N; i++)
    C[i] = A[i]+B[i];
```



```
 AND R1, N, 63      # N mod 64
 MV VLR, R1         # Do remainder
loop:
 SUB N, N, R1
 SLL R2, R1, 3      # Multiply by 8
 LV V1, RA          # Inner loop using vector
 ADD RA, RA, R2
 LV V2, RB
 ADD RB, RB, R2
 ADDV V3, V1, V2
 SV V3, RC
 ADD RC, RC, R2
 LI R1, 64          # Reset full length
 MV VLR, R1
 BGTZ N, loop       # Any more to do?
```

Vector Instruction Set Advantages

Compact

- one short instruction encodes N operations

Expressive, tells hardware that these N operations:

- are independent
- use the same functional unit
- access disjoint registers
- access registers in same pattern as previous instructions
- access a contiguous block of memory (unit-stride load/store)
- access memory in a known pattern (strided load/store)

Scalable & (somewhat) portable

- can run same code on more parallel pipelines (*lanes)*

Vector Arithmetic Execution

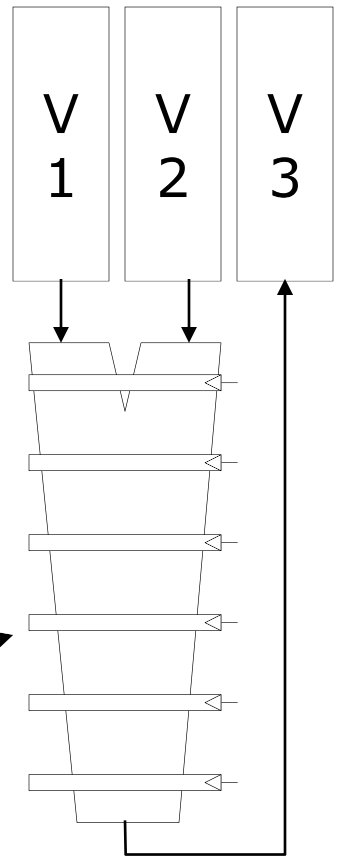
Use deep pipeline to execute element operations

- Deep pipeline ➔ Fast clock!

Much simpler pipeline control!

- Operations are independent ➔ no pipeline hazards

Vector maximizes advantages of pipelining and avoids its downsides

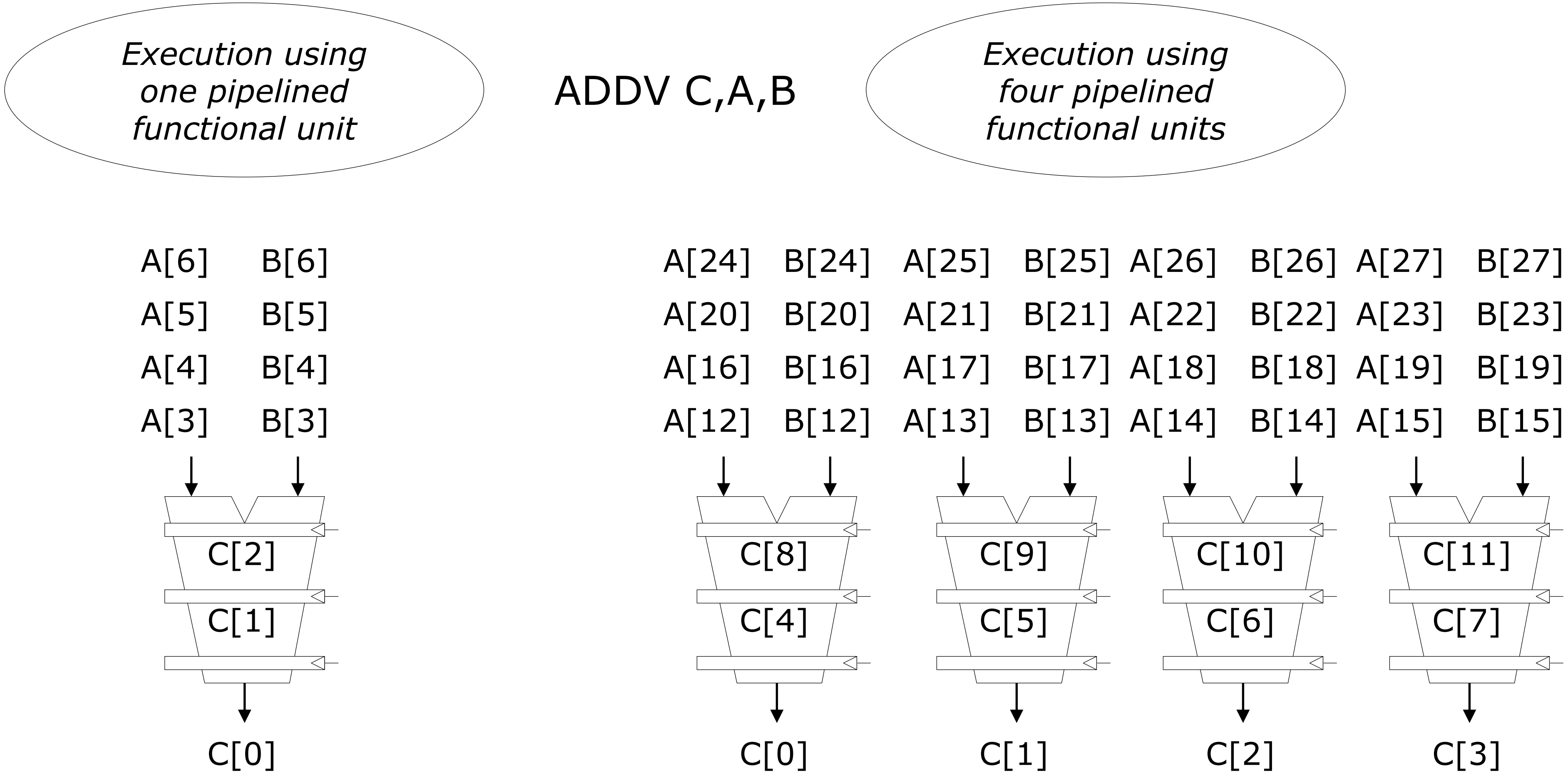


Six stage multiply pipeline

V3 <- V1 * V2

Vector Instruction Execution

Vector machine can microarchitecturally vary the number of “lanes”

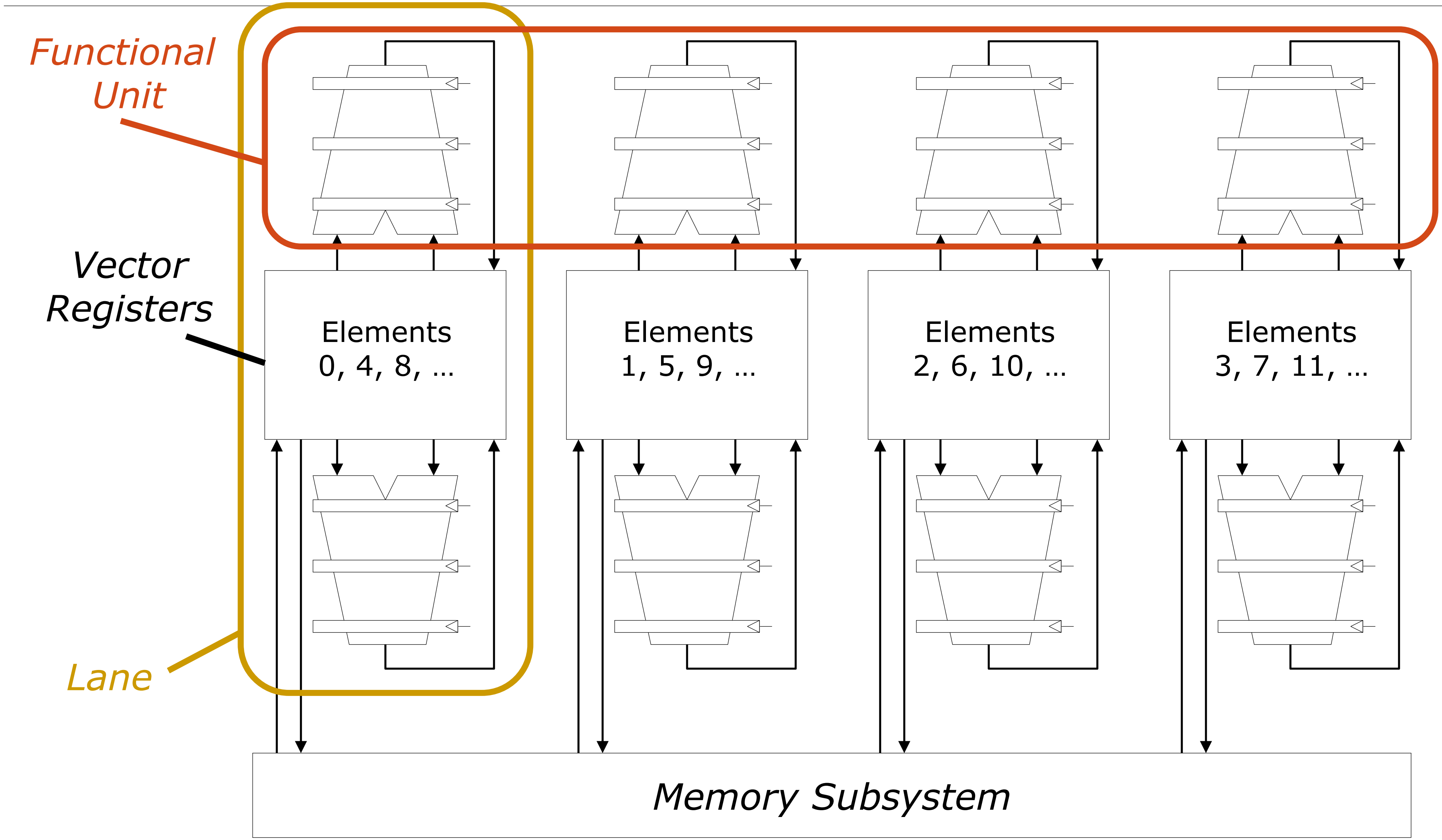


Execution using one pipelined functional unit

ADDV C,A,B

Execution using four pipelined functional units

Vector Unit Structure

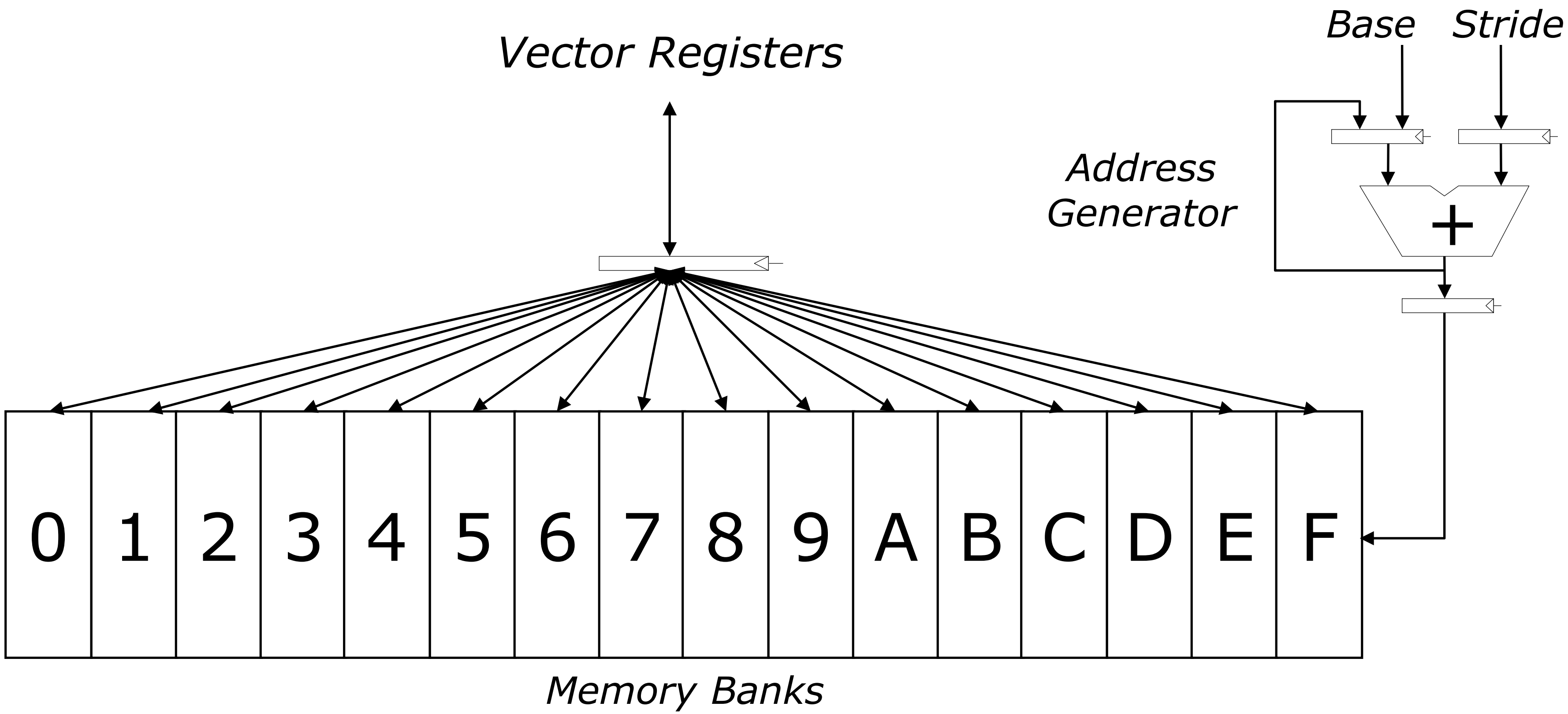


Vector Memory System

Challenge is **bandwidth** ➔ aggressive banking

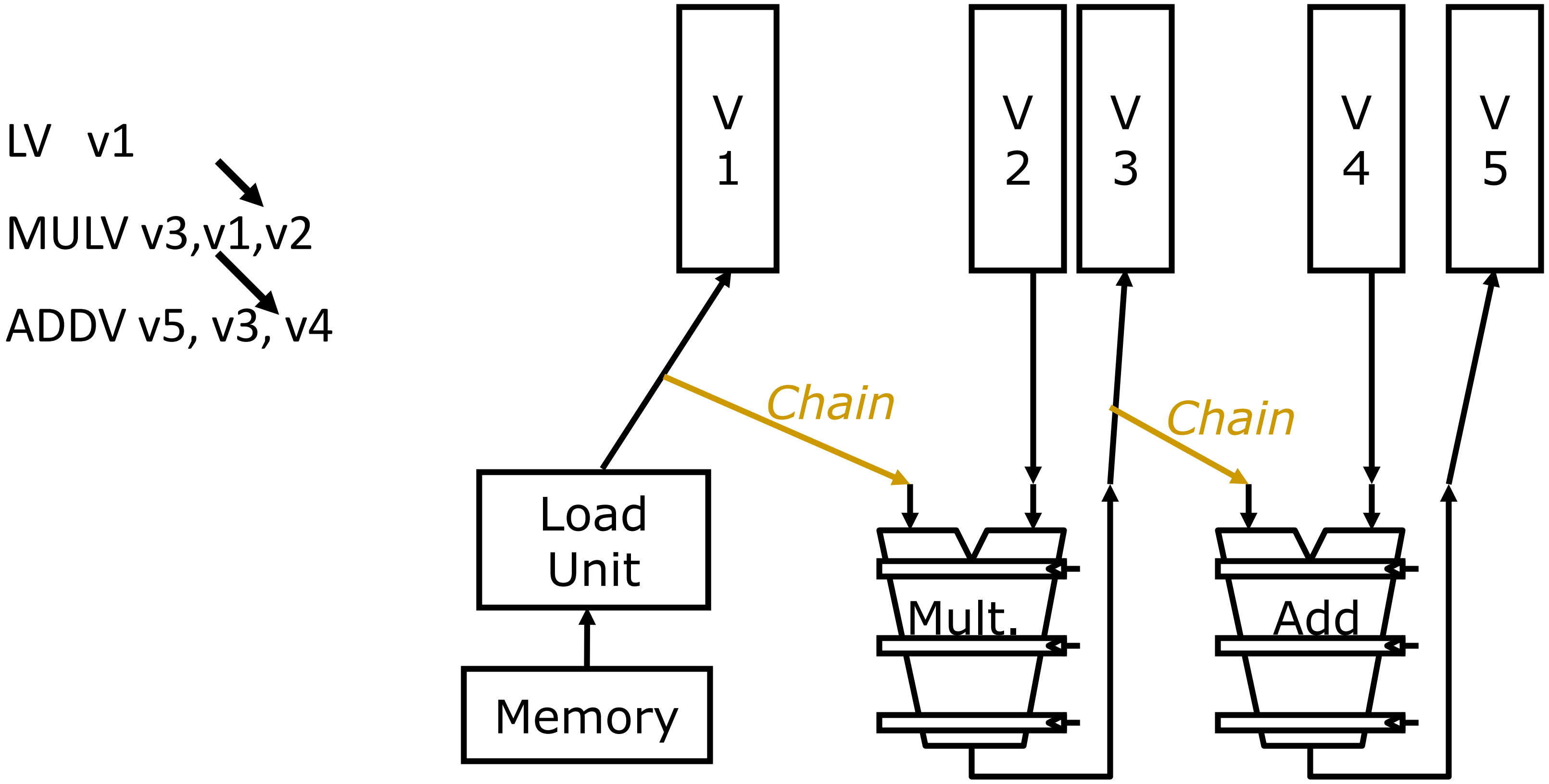
Cray-1: 16 banks, 4 cycle bank busy time, 12 cycle latency

- More on this in GPUs…



Vector Chaining

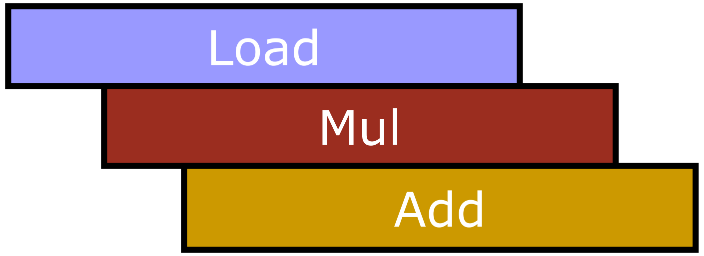
Vector analog of bypassing



Vector Chaining Advantage

- Without chaining, must wait for last element of result to be written before starting dependent instruction

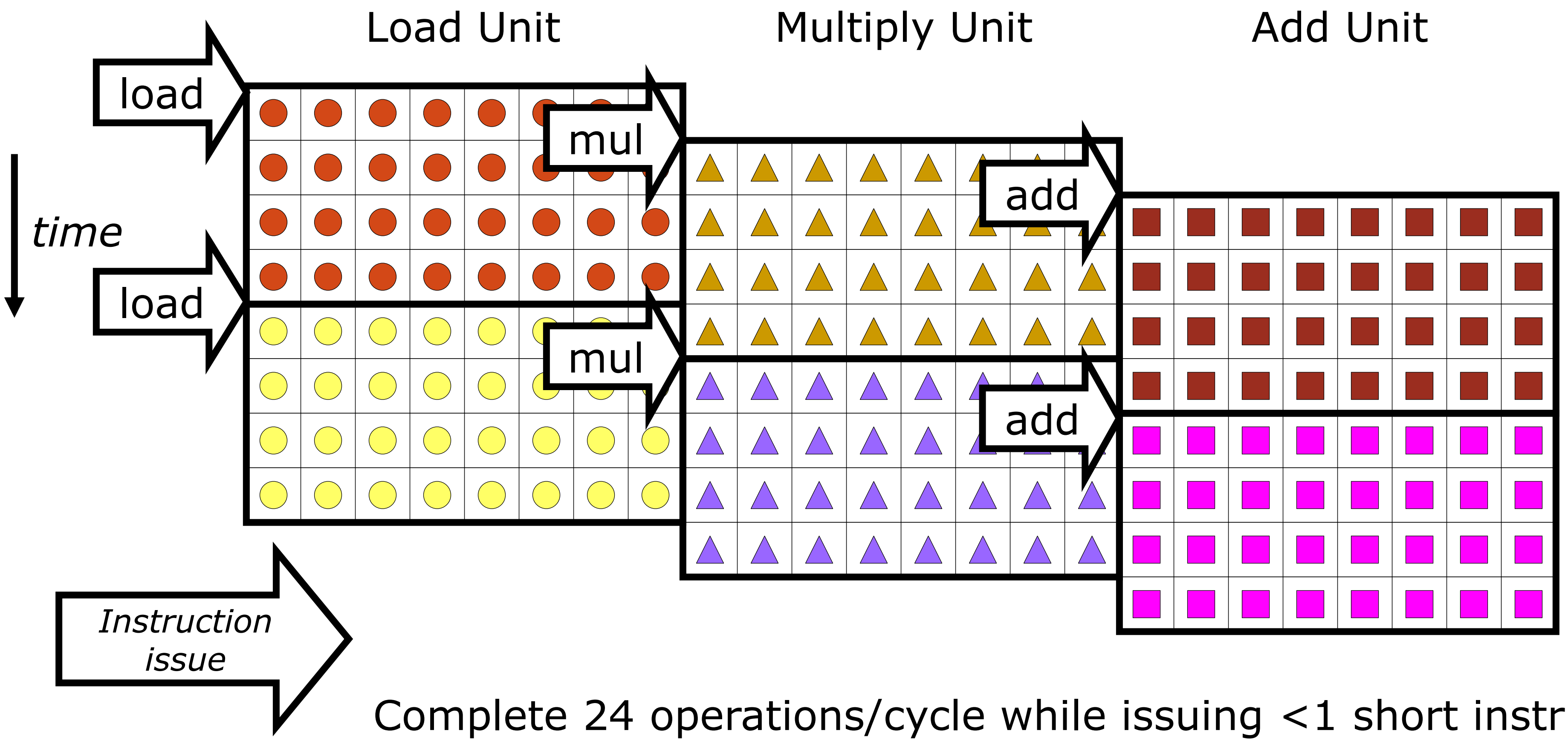
- With chaining, can start dependent instruction as soon as first result appears



Vector Instruction-Level Parallelism

Can overlap execution of multiple vector instructions

- Example machine has 32 elements per vector register and 8 lanes



Complete 24 operations/cycle while issuing <1 short instruction/cycle

Vector Conditional Execution

Problem: Want to vectorize loops with conditional code:

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

Solution: Add vector mask (or flag) registers

- vector version of predicate registers, 1 bit per element

…and maskable vector instructions

- vector operation becomes NOP at elements where mask bit is clear

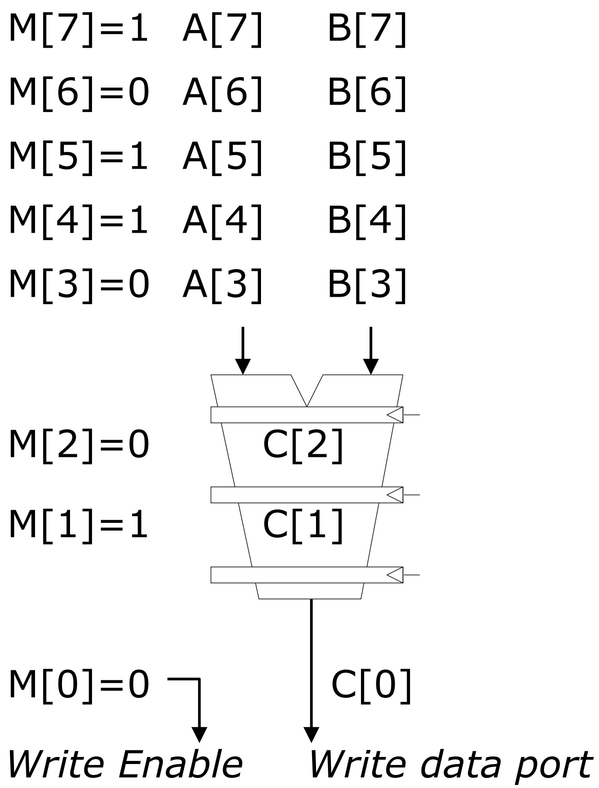
Code example:

```
CVM                 # Turn on all elements
LV vA, rA           # Load entire A vector
SGTV vA, F0         # Set bits in mask register where A>0
LV vA, rB          # Load B vector into A under mask
SV vA, rA          # Store A back to memory under mask
```

Masked Vector Instructions

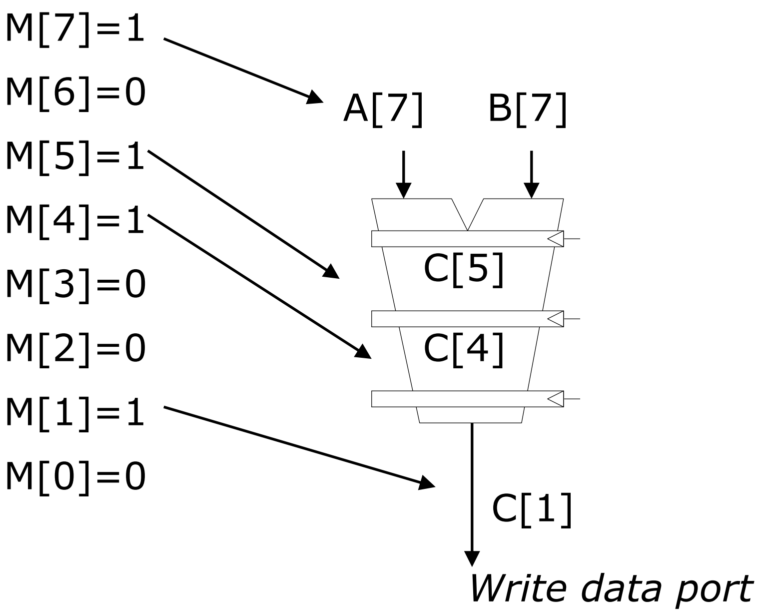
Simple Implementation

- execute all N operations, turn off result writeback according to mask



Efficient Implementation

- scan mask vector and only execute elements with non-zero masks



Vector Scatter/Gather

Want to vectorize loops with indirect accesses:

```
for (i=0; i<N; i++)
    A[i] = B[i] + C[D[i]]
```

Indexed load instruction (*Gather*)

```
LV vD, rD       # Load indices in D vector
LVI vC, rC, vD  # Load indirect from rC base
LV vB, rB       # Load B vector
ADDV vA, vB, vC # Do add
SV vA, rA       # Store result
```

Vector Scatter/Gather

Scatter example:

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

Is following a correct translation?

```
LV vB, rB         # Load indices in B vector
LVI vA, rA, vB    # Gather initial A values
ADDV vA, vA, 1    # Increment
SVI vA, rA, vB    # Scatter incremented values
```

Multimedia Extensions

Short vectors added to existing general-purpose ISAs

Initially, 64-bit registers split into 2x32b or 4x16b or 8x8b

Limited instruction set:

- No vector length control
- No strided load/store or scatter/gather
- Unit-stride loads must be aligned to 64-bit boundary

Limitation: Short vector registers

- Requires superscalar dispatch to keep multiply/add/load units busy
- Loop unrolling to hide latencies increases register pressure

Trend towards fuller vector support in microprocessors

- e.g. x86: MMX → SSEx (128 bits) → AVX (256 bits) → AVX-512 (512 bits)

Intel Larrabee Motivation

Design experiment: not a real 10-core chip!

# CPU cores	2 out of order	10 in-order
Instructions per issue	4 per clock	2 per clock
VPU lanes per core	4-wide SSE	16-wide
L2 cache size	4 MB	4 MB
Single-stream	4 per clock	2 per clock
Vector throughput	**8 per clock**	**160 per clock**

20 times the multiply-add operations per clock

Data in chart taken from Seiler, L., Carmean, D., et al. 2008. *Larrabee: A many-core x86 architecture for visual computing*.

Larrabee/Xeon Phi: x86 with vectors

64 cores w/ short, in-order pipeline

- Pentium core + vectors

4 “hyper”-threads / core

- 288 threads total
- Time-multiplexed, skipping stalled threads
- Cannot issue from same thread consecutively

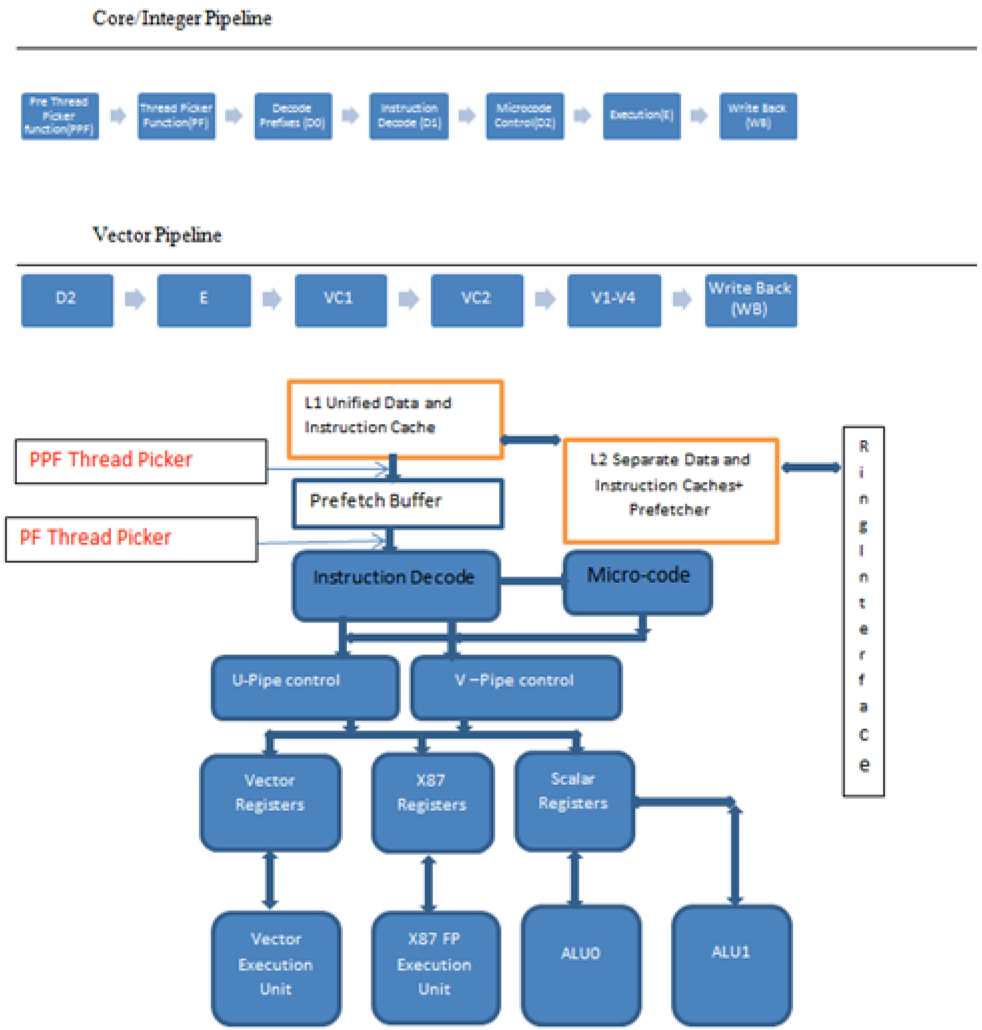
Separate scalar and vector units and register sets

- Vector unit: 16 x 32-bit ops/clock

Fast access to L1 cache

L1 connects to core’s portion of the L2 cache

Latest Xeon Phi have 72 “wimpy” out-of-order cores



Larrabee/Xeon Phi Vector ISA

Data types: 32-bit integer, 32- and 64-bit floating point

Vector operations

- Two input/one output operations
- Full complement of arithmetic and media operations
  - Fused multiply-add (three input arguments)
- Mask registers select lanes to write
- Swizzle the vector elements on register read

Memory access

- Vector load/store including scatter/gather
- Data replication on read from memory
- Numeric type conversion on memory read

Graphics Processing Units (GPUs)

Why Study GPUs?

GPUs combine two useful strategies to increase efficiency

- **Massive parallelism:** hide latency with other independent work
- **Specialization:** optimize architecture for particular workload

All to avoid architectural overheads & scaling limits of OoO
➔ More resources available for useful computation

Most successful commodity accelerator

- Tension between performance and programmability

Culmination of many design techniques

- Multicore, vector, superscalar, VLIW, etc

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

Modern GPUs

- Programmable multiprocessors optimized for data-parallelism
  - OpenGL/DirectX and general purpose languages (CUDA, OpenCL, …)
- Still some fixed-function hardware for graphics (texture, raster ops, …)
- Converging to vector processors

GPUs in Modern Systems

Discrete GPUs

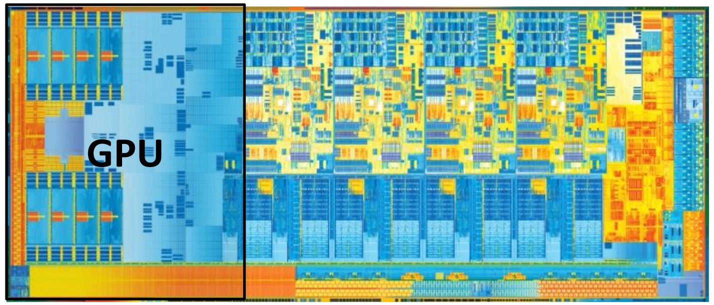
- PCIe-based accelerator
- Separate GPU memory

Nvidia Kepler

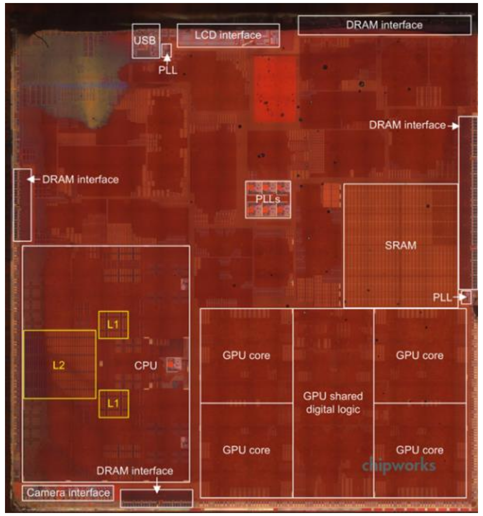
Integrated GPUs

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

Pros/cons?



Intel Ivy Bridge, 22nm 160mm^2



Apple A7, 28nm TSMC, 102mm^2

Our Focus

GPUs as programmable multicores

- Vastly different design point than CPUs
- Software model
- Hardware architecture

Good high-level mental model

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

Will use Nvidia programming model (CUDA) and terminology (like Hennessy & Patterson)

CUDA GPU Thread Model

Single-program multiple data (SPMD) model

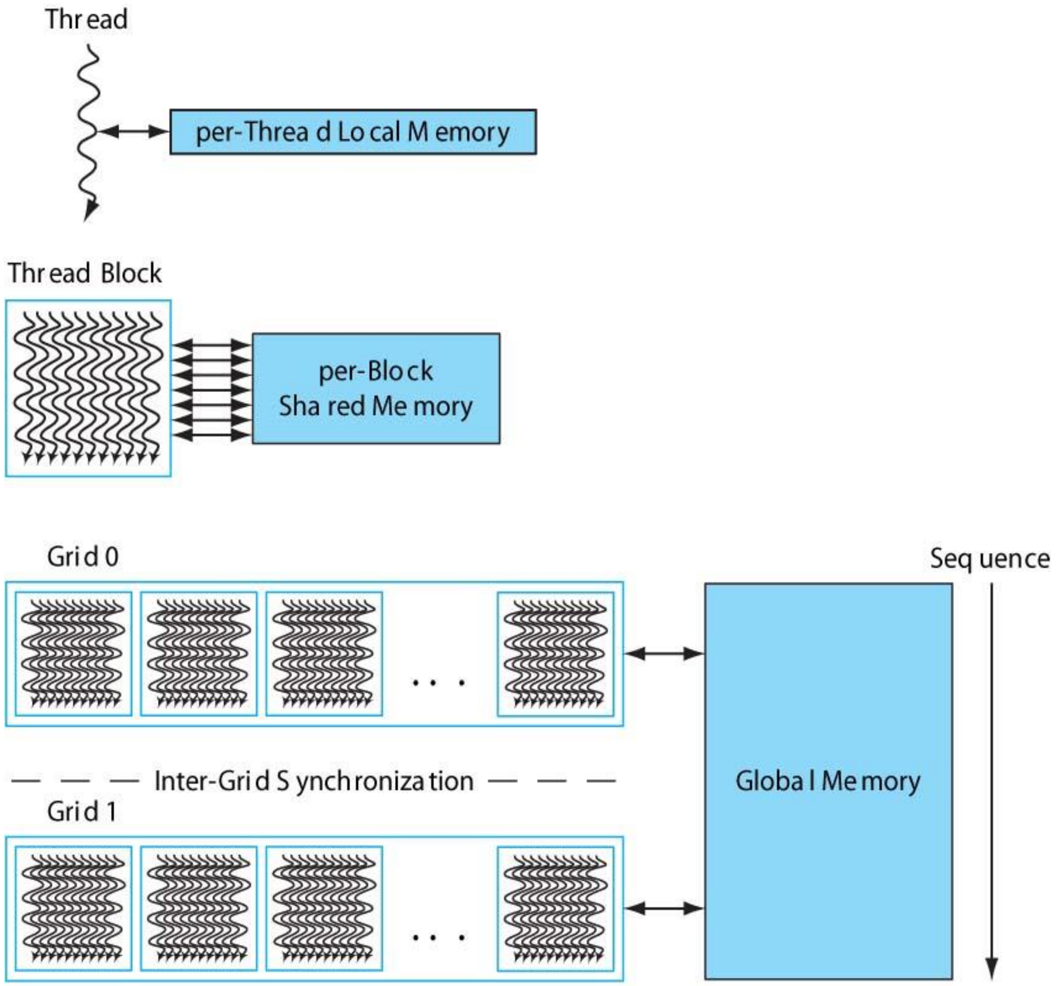
Each **thread** has local memory

Parallel threads packed in **blocks**

- Access to per-block shared memory
- Can synchronize with barrier

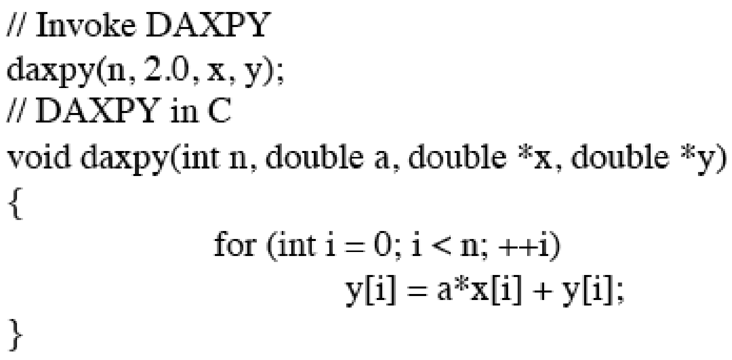
Grids include independent blocks

Vector analog: Program a single lane;
HW dynamically schedules



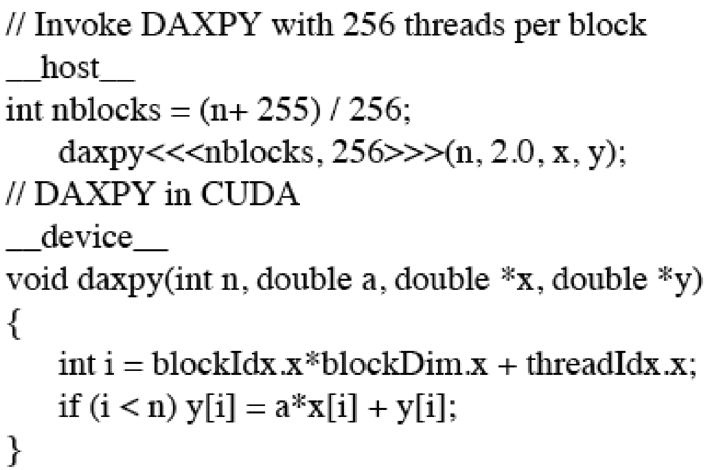
Code Example: DAXPY

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 (with VL=256 in this example)
- Grid = Vectorizable loop

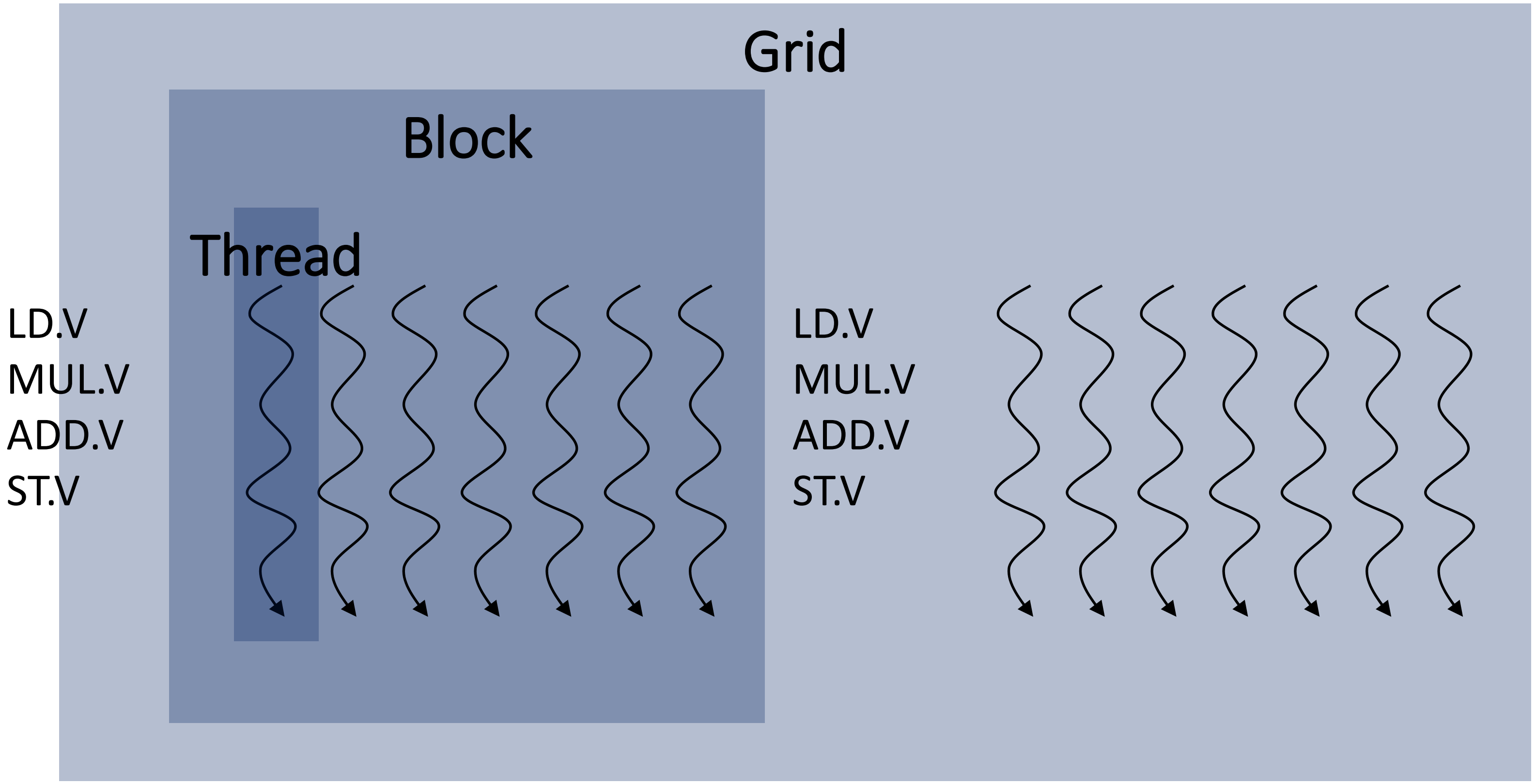
GPU Terminology

In classical terms,

GPUs are superscalar, vector, multithreaded, multiprocessors

but GPUs have developed their own (confusing) nomenclature…

Vector vs GPU Terminology



Grid

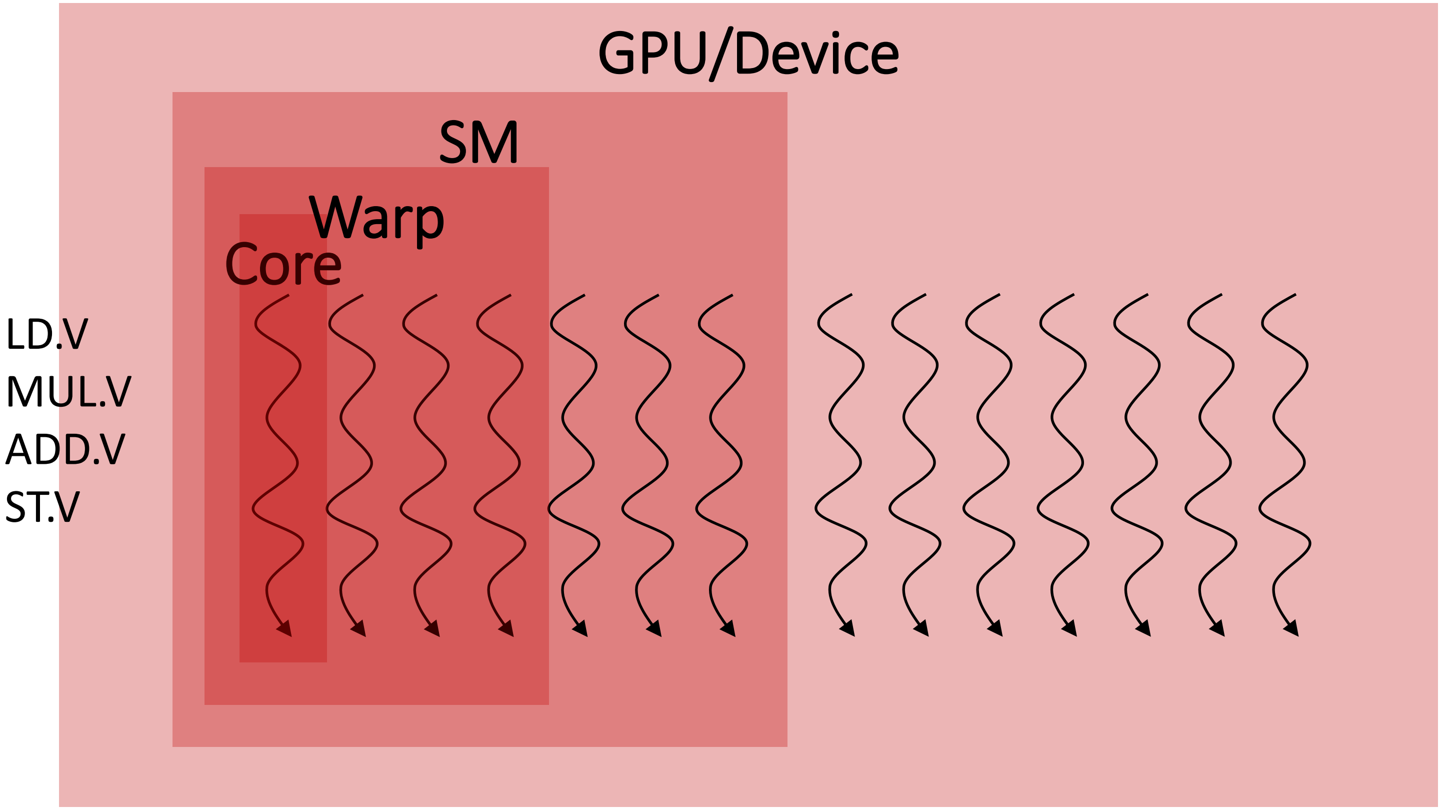
Block

Thread

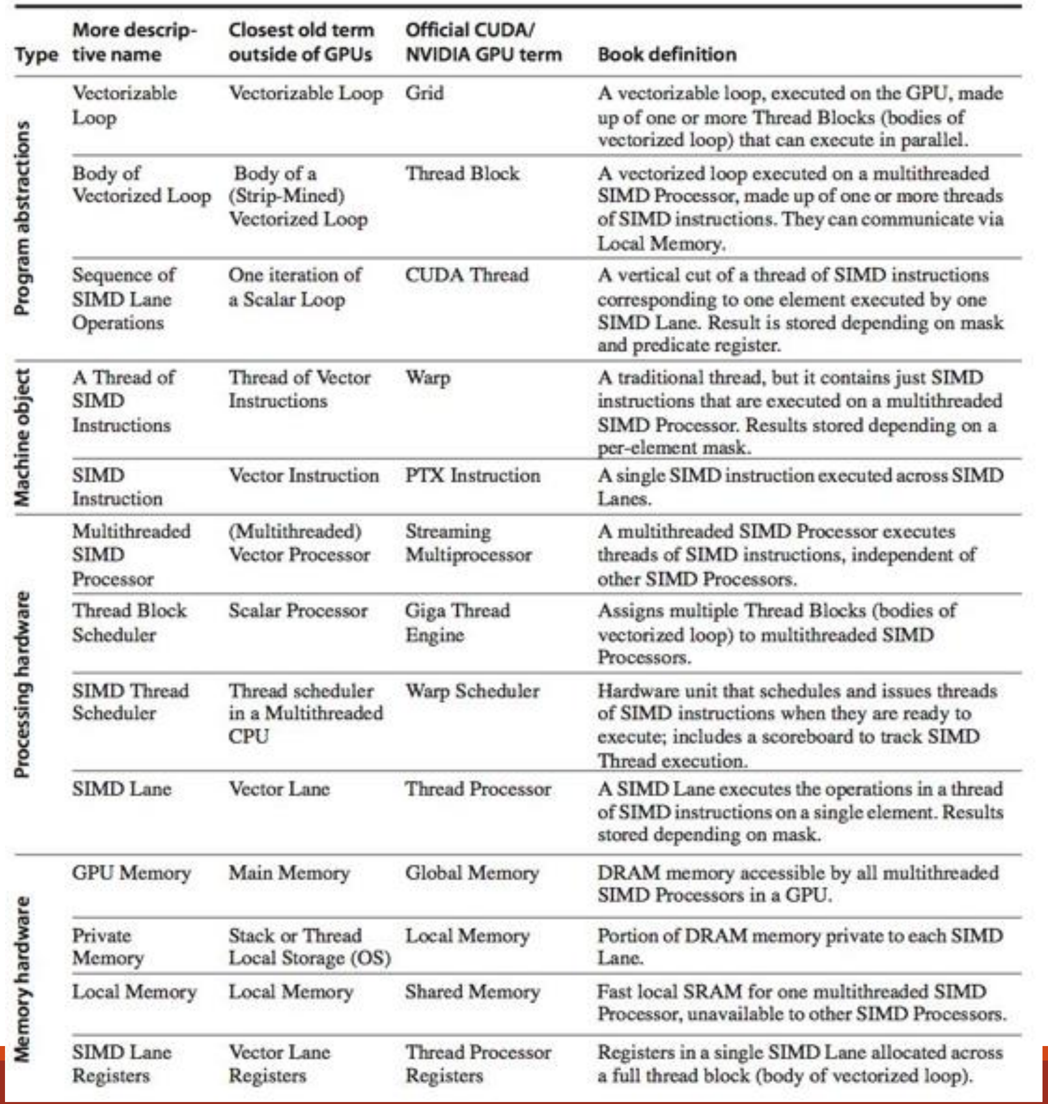
LD.V
MUL.V
ADD.V
ST.V

LD.V
MUL.V
ADD.V
ST.V

Vector vs GPU Terminology



Vector vs GPU Terminology



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).

[H&P5, Fig 4.25]

Vector vs GPU Terminology

	Vector term	**GPU term**
Programming	Vectorizable loop	Grid
	Body of (strip-mined) loop	Thread block
	Scalar loop iteration	Thread
	Thread of vector instructions	Warp
Compute	Vector lane	Core/Thread processor
	Vector processor (multithreaded)	Streaming processor
	Scalar processor	Giga thread engine
	Thread scheduler (hw)	Warp scheduler
Memory	Main memory	Global memory
	Private memory	Local memory
	Local memory	Shared memory
	Vector lane registers	Thread registers

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
- E.g., “Driver updates” for newly released games

GPU Architecture Overview

A highly multithreaded multicore chip

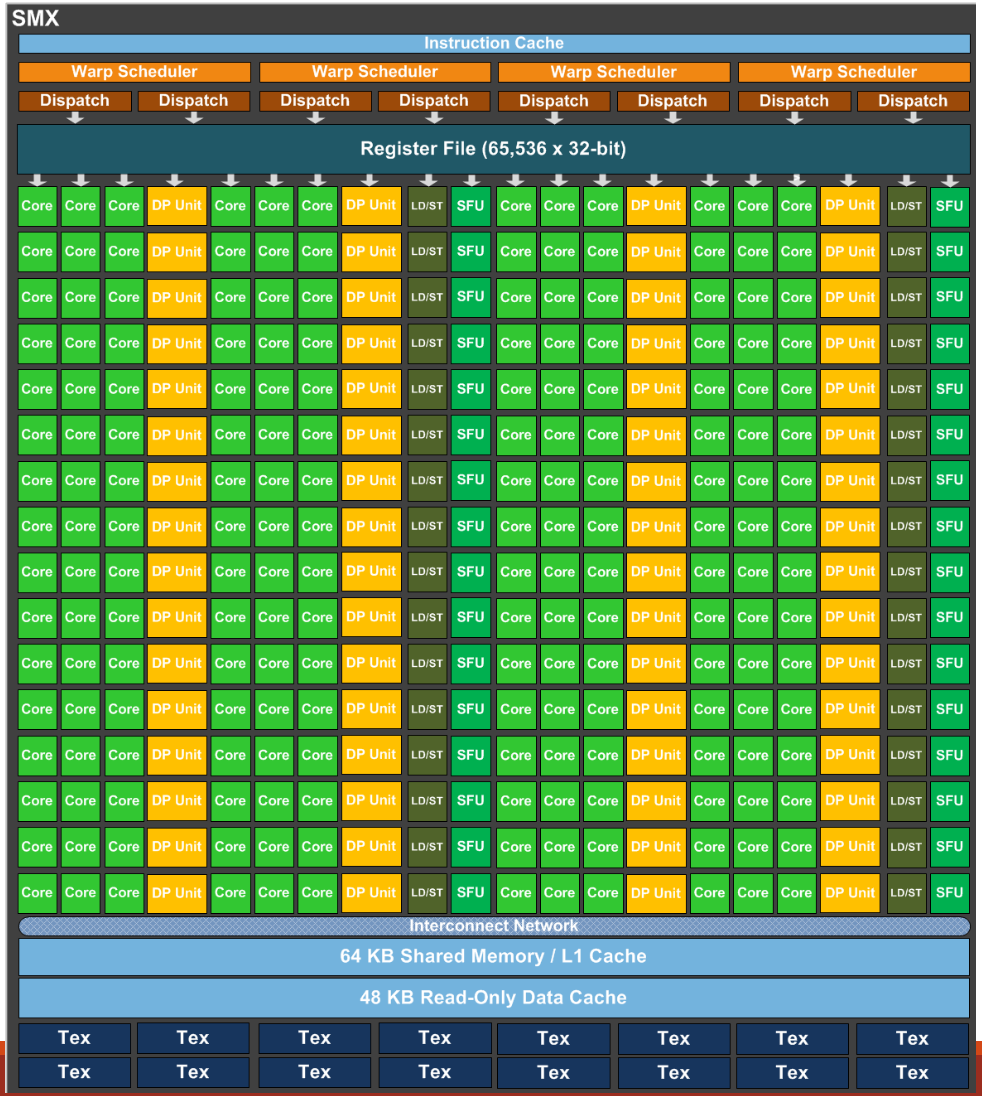
Example: Nvidia Kepler GK110

- 15 cores or streaming multiprocessors (SMX)
- 1.5MB Shared L2 cache
- 6 memory channels
- Fixed-function logic for graphics (texture units, raster ops, …)

- **Scalability → change number of cores and memory channels**

- Scheduling mostly controlled by hardware

Zoom In: Kepler Streaming Multiprocessor



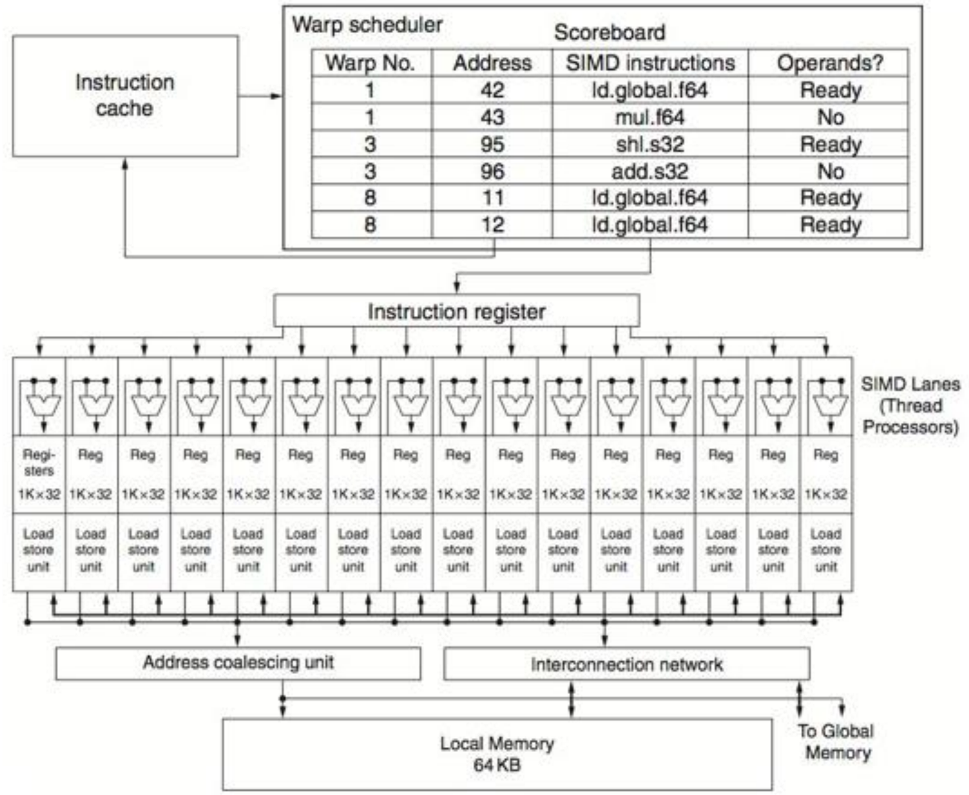
Execution units

- 192 simple FUs (int and single-precision FP)
- 64 double-precision FUs
- 32 load-store FUs
- 32 special-function FUs (e.g., sqrt, sin, cos, …)

Memory structures

- 64K 32-bit registers
- 64KB data memory, split between shared memory (scratchpad) and L1
- 48KB read-only data/texture cache

Streaming Multiprocessor Execution Overview



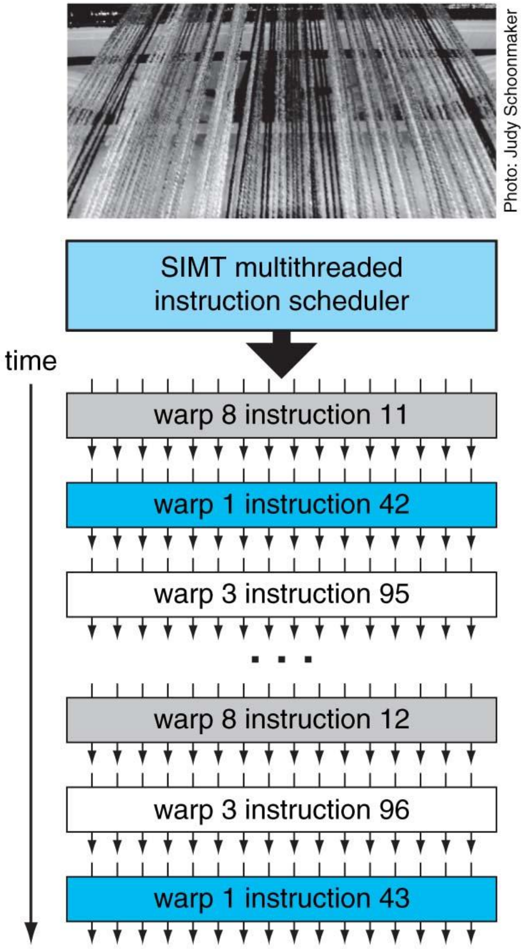
Warp No.	Address	SIMD instructions	Operands?
1	42	ld.global.f64	Ready
1	43	mul.f64	No
3	95	shl.s32	Ready
3	96	add.s32	No
8	11	ld.global.f64	Ready
8	12	ld.global.f64	Ready

Each SM supports 10s of warps (e.g., 64 in Kepler)

- I.e., HW multithreading

Multithreading is a GPU's main latency-hiding mechanism

Thread Scheduling & Parallelism



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 (like vector instructions we saw earlier)

Individual threads may be inactive

- Because they branched differently
- Equivalent of conditional execution (but **implicit**)
- Loss of efficiency if not data parallel

Software thread blocks mapped to warps

- When HW resources are available

Context Size vs Number of Contexts

SMs support a variable number of thread 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
- Constraint: All warps of a thread block must be scheduled on same SM

Example: Kepler SMX supports up to 64 warps

- Max: 64 warps @ <= 32 registers/thread
- Min: 8 warps @ 255 registers/thread

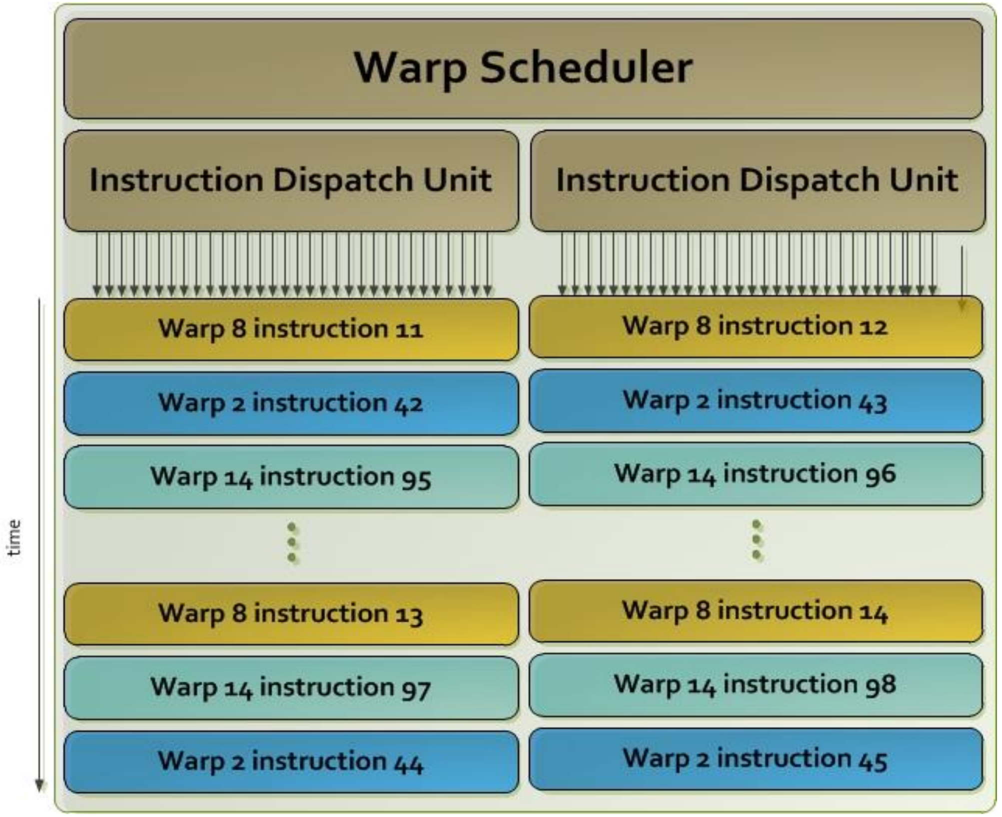
Kepler Warp Scheduler & Instruction Dispatch

Scheduling

- 4 schedulers select 1 warp/cycle
- 2 independent instructions issued per warp
- Total throughput = 4 * 2 * 32 = 256 ops per cycle

Register scoreboarding

- To track ready instructions
- Simplified using static latencies from compiler (a la VLIW)



Conditional Execution & Branch Divergence

Similar to vector masking, 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

```
if (m != 0) {
  if (a > b) {
    y = a - b;
  } else {
    y = b - a;
  }
} else {
  y = 0;
}
```

Assume 4 threads/warp,

initial mask 1111

M = [1, 1, 0, 0]

A = [5, 4, 2, 6]

B = [3, 7, 3, 1]

How efficient is this execution?

Memory Access Divergence

All loads are gathers, all stores are scatters

SM address coalescing unit detects sequential and strided patterns, coalesces memory requests

- Optimizes for memory **bandwidth**, not latency

Warps **stall** until all operands ready

- Must limit memory divergence to keep cores busy
- ➔ Good GPU code requires regular access patterns, even though programming model allows arbitrary patterns!

Memory System

Within a single SM:

- Instruction and constant data caches
- Multi-banked shared memory (scratchpad, not cache)
- **No inter-SM coherence** (unlike, say, Xeon Phi)

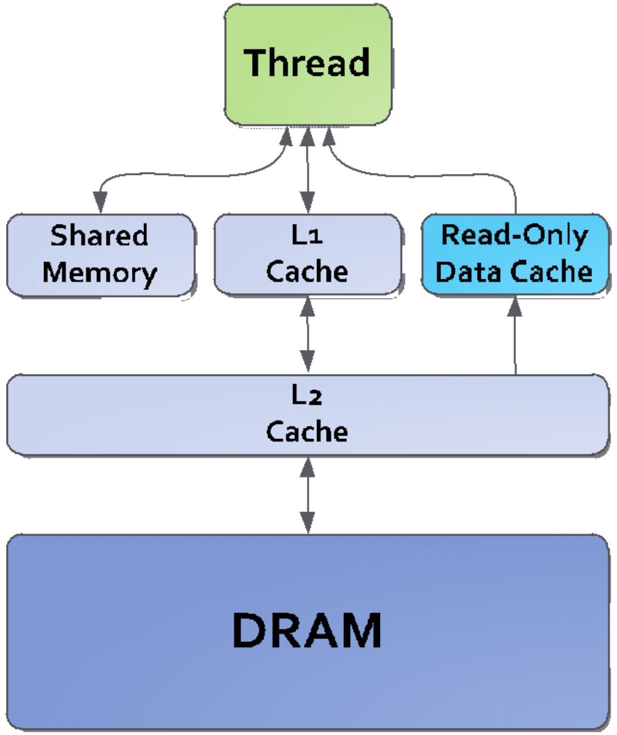
GPUs now include a small, shared L2 cache

- Reduce energy, amplify bandwidth
- Faster atomic operations

Bandwidth-optimized main memory

- Interleaved addresses
- Aggressive access scheduling & re-ordering
- Lossless and lossy compression (e.g., for textures)

Example: Kepler Memory Hierarchy



Each SM 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 (texture memory)

1.5MB shared L2
- Supports synchronization operations (atomicCAS, atomicADD, …)
- **How many bytes/thread?**

GDDR5 main memory
- 384-bit interface (6x 64-bit channels) @ 1.75 GHz (x4 T/cycle)
- 336 GB/s peak bandwidth

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

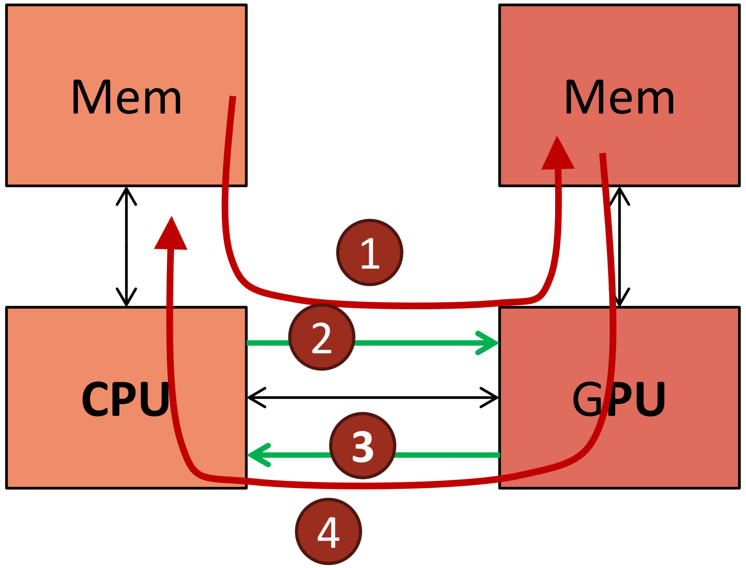
Atomic operations to global memory

- Read-modify-write operations (add, exchange, compare-and-swap, …)
- Performed at the memory controller or at the L2

Limited inter-block synchronization!

- **Can’t wait for other blocks to finish**

GPU Kernel Execution



① Transfer input data from CPU to GPU memory

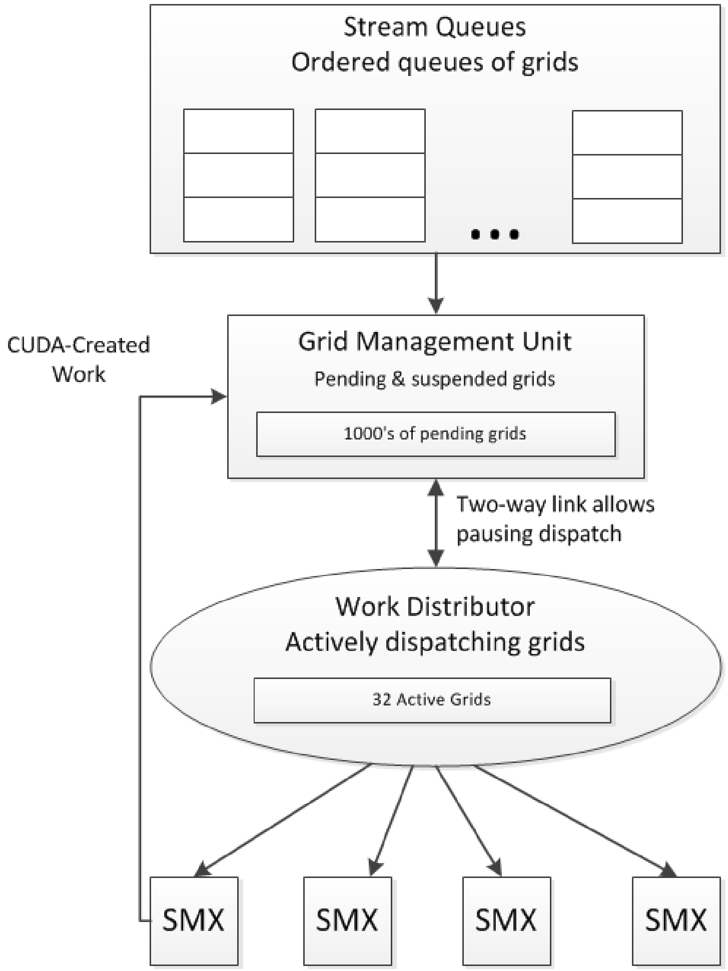
② Launch kernel (grid)

③ Wait for kernel to finish (if synchronous)

④ Transfer results to CPU memory

- Data transfers can dominate execution
  - Pipeline: Overlap next transfer & current execution
  - Integrated GPUs with unified address space ➔ no copies

Hardware Scheduling



HW unit schedules grids on SMX

- Priority-based scheduling

32 active grids

- More queued/paused

Grids can be launched by CPU or GPU

- Work from multiple CPU threads and processes

System-Level Issues

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)

Scheduling

- Each kernel is non-preemptive (but can be aborted)
- Resource management and scheduling left to GPU driver, opaque to OS

GPU Programmability

GPUs are historically accelerators, with general-purpose programming added after-the-fact

- Original GPGPU codes hijacked fixed-function graphics pipeline
- CUDA gives C++ interface, but many legacy limitations are still prominent
- E.g., incoherent memory between SMs, costly synchronization, graphics-optimized primitives like texture memory & FUs

Irregular programs with divergent branches or loads perform badly **by design**

- GPUs choose not to pay overheads of running these well

Rapid development of better programming features

- Open question: what’s a good consistency model?
- Xeon Phi’s big marketing advantage

Vector/GPU Summary

Force programmers to write (implicitly or explicitly) parallel code

Simple hardware can find lots of work to execute in parallel ➔ more compute per area/energy/cost

Solves memory latency problem by overlapping it with useful work

- Must architect for memory bandwidth instead of latency
- Less focus on caches, more on banking etc

GPUs are modern incarnation of this old idea