Concepts Introduced in Chapter 4

- vector architectures
- SIMD ISA extensions
- graphics processing units (GPUs)
- loop dependence analysis

SIMD Advantages

- SIMD architectures can significantly improve performance by exploiting DLP when available in applications.
- SIMD processors are more energy efficient than MIMD as they only need to fetch a single instruction to perform the same operation on multiple data items.
- SIMD allows programmers to continue to write algorithms in a sequential manner and sometimes SIMD parallelism can be automatically exploited.

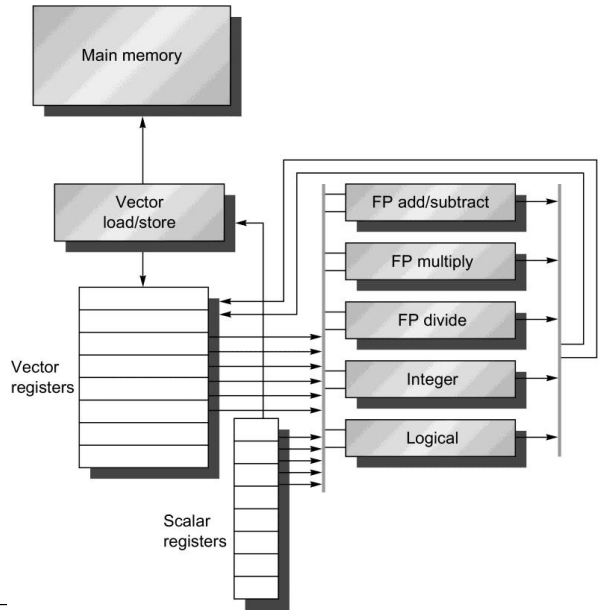
Vector Architectures

- A vector architecture includes instruction set extensions to an ISA to support vector operations, which are deeply pipelined.
  - Vector operations are on vector registers, where each is a fixed-length bank of registers.
  - Data is transferred between a vector register and the memory system.
  - Each vector operation takes two vector registers or a vector register and a scalar value as input.
- A vector architecture can only be effective on applications that have significant data-level parallelism (DLP).
- vector processing advantages
  - Greatly reduces the dynamic instruction bandwidth.
  - Generally execution time is reduced due to (1) significantly decreasing loop overhead, (2) stalls only occurring on the first vector element rather than on each vector element, and (3) performing vector operations in parallel.

Extending RISC-V to Support Vector Operations (RV64V)

- Add 8 vector registers where each register has 32 elements with each element being 64 bits wide.
- After an initial latency each vector functional unit can start a new operation on each clock cycle.
- Vector loads and stores also pay for the memory latency once and afterwards a word is transferred each cycle between the vector register and memory.
- The processor has to detect both structural hazards, which can cause stalls, and data hazards so that chaining (forwarding) can be performed.

Basic Structure of a Vector Architecture



RV64V Vector Instructions

Mnemonic	Name	Description
vadd	ADD	Add elements of V[rs1] and V[rs2], then put each result in V[rd]
vsub	SUBtract	Subtract elements of V[rs2] frpm V[rs1], then put each result in V[rd]
vmul	MULtiply	Multiply elements of V[rs1] and V[rs2], then put each result in V[rd]
vdiv	DIVide	Divide elements of V[rs1] by V[rs2], then put each result in V[rd]
vrem	REMainder	Take remainder of elements of V[rs1] by V[rs2], then put each result in V[rd]
vsqrt	SQuare RooT	Take square root of elements of V[rs1], then put each result in V[rd]
vsll	Shift Left	Shift elements of V[rs1] left by V[rs2], then put each result in V[rd]
vsrl	Shift Right	Shift elements of V[rs1] right by V[rs2], then put each result in V[rd]
vsra	Shift Right Arithmetic	Shift elements of V[rs1] right by V[rs2] while extending sign bit, then put each result in V[rd]
vxor	XOR	Exclusive OR elements of V[rs1] and V[rs2], then put each result in V[rd]
vor	OR	Inclusive OR elements of V[rs1] and V[rs2], then put each result in V[rd]
vand	AND	Logical AND elements of V[rs1] and V[rs2], then put each result in V[rd]
vsgnj	SiGN source	Replace sign bits of V[rs1] with sign bits of V[rs2], then put each result in V[rd]
vsgnjn	Negative SiGN source	Replace sign bits of V[rs1] with complemented sign bits of V[rs2], then put each result in V[rd]
vsgnjx	Xor SiGN source	Replace sign bits of V[rs1] with xor of sign bits of V[rs1] and V[rs2], then put each result in V[rd]

RV64V Vector Instructions (cont.)

vld	Load	Load vector register V[rd] from memory starting at address R[rs1]
vlds	Strided Load	Load V[rd] from address at R[rs1] with stride in R[rs2] (i.e., R[rs1]+i × R[rs2])
vldx	Indexed Load (Gather)	Load V[rs1] with vector whose elements are at R[rs2]+V[rs2] (i.e., V[rs2] is an index)
vst	Store	Store vector register V[rd] into memory starting at address R[rs1]
vsts	Strided Store	Store V[rd] into memory at address R[rs1] with stride in R[rs2] (i.e., R[rs1]+i × R[rs2])
vstx	Indexed Store (Scatter)	Store V[rs1] into memory vector whose elements are at R[rs2]+V[rs2] (i.e., V[rs2] is an index)
vpeq	Compare =	Compare elements of V[rs1] and V[rs2]. When equal, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vpne	Compare !=	Compare elements of V[rs1] and V[rs2]. When not equal, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vplt	Compare <	Compare elements of V[rs1] and V[rs2]. When less than, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vpxor	Predicate XOR	Exclusive OR 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
vpor	Predicate OR	Inclusive OR 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
vpand	Predicate AND	Logical AND 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
setvl	Set Vector Length	Set vl and the destination register to the smaller of mvl and the source regsiter

Example of Vector Code

- This is the DAXPY (Double precision A times X Plus Y) loop from the Linpack benchmark.
- Assume x5 and x6 initially contain the beginning addresses of the X and Y arrays.

```
/* Scalar RISC-V Code */      /* Source Code */
   fld    f0,a          <=    for (i = 0; i < 32; i++)
   addi   x28,x5,#256             Y[i] = a * X[i] + Y[i];
Loop:
   fld    f1,0(x5)              /* RV64V Code */
   fmul.d f1,f1,f0              vsetdcfg 4*FP64
   fld    f2,0(x6)              fld      f0,a
   fadd.d f2,f2,f1              vld      v0,x5
   fsd    f2,0(x6)       =>    vmul     v1,v0,f0
   addi   x5,x5,#8              vld      v2,x6
   addi   x6,x6,#8              vadd     v3,v1,v2
   bne    x28,x5,Loop           vst      v3,x6
                                vdisable
```

Chaining, Convoys, and Chimes

- *Chaining* allows the results of one vector operation to be directly used as input to another vector operation.
- A *convoy* is a set of vector instructions that can potentially execute together. Only structural hazards cause separate convoys as true dependences are handled via chaining in the same convoy. The RV64V code below has 3 convoys as there is only one vector memory unit.
- A *chime* is the unit of time taken to execute one convoy, which is the vector length along with the startup cost. The following RV64V code executes in three chimes since there are three convoys.

```
/* RV64V code */          /* convoys */
vld     v0,x5             1. vld     v0,x5
vmul    v1,v0,f0             vmul    v1,v0,f0
vld     v2,x6             2. vld     v2,x6
vadd    v3,v1,v2             vadd    v3,v1,v2
vst     v3,x6             3. vst     v3,x6
```


Startup Time

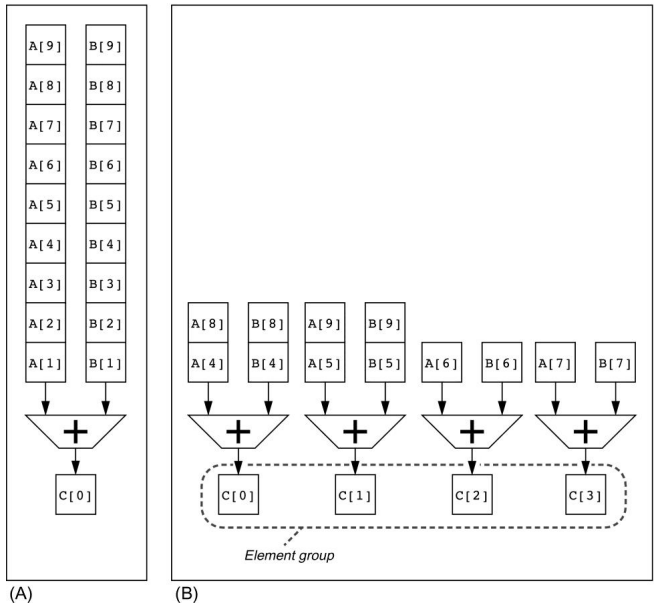
- The startup time for a convoy is primarily affected by the pipelining latency of the vector functional unit associated with the vector operation.
- pipeline latencies in clock cycles for the RV64V
  - FP add - 6
  - FP multiply - 7
  - FP divide - 20
  - load - 12
- Additional cycles need to be added for stalls between the chained vector operations, but only for the first element of each vector operation.
- The cycles to execute the following convoy should be the sum of the *startup time* and the *vector length*, or 51 (19+32).

```
vld     v0,x5
vmul    v1,v0,f0
```

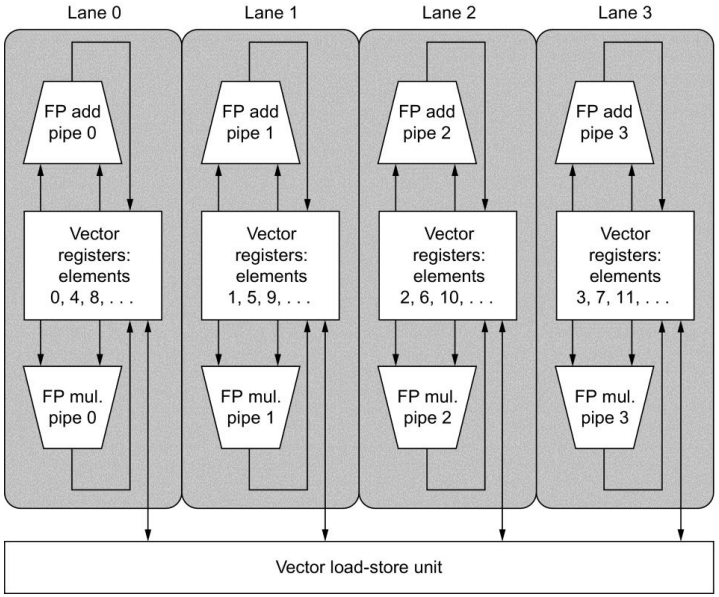

Using Multiple Lanes

- Vector operations on vector register elements can also be executed in parallel when there is an array of parallel pipelined functional units.
- So element *N* of a vector register will take part in operations with element *N* from other vector registers.
- Each *lane* contains one portion of the register file and one execution pipeline from each vector functional unit.
- Each lane *i* of *n* lanes operates on each *k* vector register file element where *k* % *n* is equal to *i*.
- No communication is needed between lanes.
- Convoy time is now *startup time* + ceil(*vector length*/*n*).

Using Multiple Functional Vector Units



Structure of a Vector Unit Containing Four Lanes



Vector Length Register

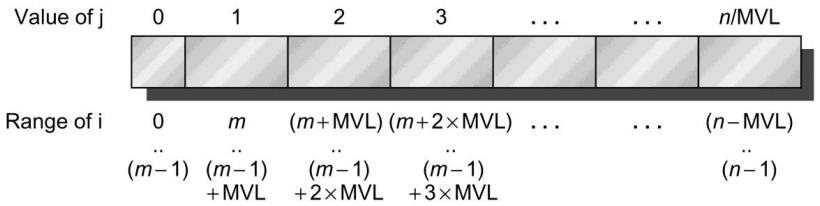
- A vector length register (vl) allows the length of a vector operation to be determined at run time.
- Loops are *strip mined* so that the maximum vector length (MVL) is no more than the length of a vector register.
- The innermost loop in the strip mined loop nest can be vectorized.

```
/* original loop */
for (i = 0; i < n; i++)
   Y[i] = a * X[i] + Y[i];
```

```
/* strip mined loop nest */
low = 0;
vl = n % MVL;
for (j = 0; j <= n/MVL; j++) {
   for (i = low; i < low+vl; i++)
      Y[i] = a * X[i] + Y[i];
   low += vl;
   vl = MVL;
}
```


A Vector of Arbitrary Length Processed with Strip Mining

- The first vector operation processes m = n % MVL elements.
- The remaining vector operations process MVL elements.



Strip Mined Vectorized Code

```
/* stripmined loopnest */
i = 0;
while (n != 0) {
   vl = min(MVL,n);
   for (j = 0; j < vl;
        j++, i++)
      Y[i] =
         a*X[i]+Y[i];
   n -= vl;
}
```

```
/* RV64V code */
   vsetdcfg 2 DP FP  # enable 2 vect regs
   fld      f0,a     # f0=M[a]
loop:
   setvl    t0,a0    # vl=t0=min(MVL,n)
   vld      v0,x5    # load vector x
   slli     t1,t0,3  # t1=t0*8
   add      x5,x5,t1 # x5 += t1
   vmul     v0,v0,f0 # vect-scalar mult
   vld      v1,x6    # load vector y
   vadd     v1,v0,v1 # vect-vect add
   sub      a0,a0,t0 # n -= t0
   vst      v1,x6    # store vector y
   add      x6,x6,t1 # x6 += t1
   bnez     a0,loop  # loop if n != 0
   vdisable          # disable vect regs
```

Vector Mask Registers

- Mask registers provide support for conditional execution of each element within a vector register in a vector instruction.
- When the vector-mask register is enabled, vector instructions update results only for vector elements where the corresponding bit in the vector-mask register is set.
- No execution time is saved for the elements where the bits in the vector-mask register are zero.

```
/* original loop */
for (i = 0; i < 32; i++)
   if (X[i] != 0)
      X[i] -= Y[i];
```

```
/* RV64V assembly code */
vsetdcfg  2*FP64    # enable 2 vect regs
vsetpcfgi 1         # enable 1 pred reg
vld       v0,x5     # load X into v0
vld       v1,x6     # load Y into v1
fmv.d.x   f0,x0     # f0 = 0.0
vpne      p0,v0,f0  # p0 = v0 != f0
vsub      v0,v0,v1  # if (p0) v0 -= v1
vst       v0,x5     # if (p0) M[X] = v0
vdisable            # disable vect regs
vpdisable           # disable pred reg
```


Using Cache/Memory Banks

- The more recent vector computers use caches to reduce the latency of vector loads and stores.
- Word-interleaved banks for cache and main memory often provide the ability for simultaneous independent accesses.
  - Supporting multiple vector load or store operations to avoid a structural hazard.
  - Supporting vector loads or stores that are not sequential.
  - Supporting multiple processor cores sharing the same L3 cache and main memory.

Handling Non-Unit Strides

- The distance separating elements in memory can be nonsequential, which is called a *non-unit stride*.
- The vector stride can be put in a general-purpose register and can be accessed with vector load/store instructions.
- Supporting non-unit strides may cause more bank contention and cache misses, which complicates the vector load/store operations.
- Assume the addresses of *B* and *D* are in *x7* and *x8*, respectively.

```
/* matrix multiply loop nest */
for (i = 0; i < 100; i++)
   for (j = 0; j < 100; j++) {
      A[i][j] = 0.0;
      for (k = 0; k < 100; k++)
         A[i][j] +=
            B[i][k]*D[k][j];
   }
```

```
/* inner loop RV64V code */
vld  v1,x7      # load B into v1
mov  x5,#800    # stride = 800
vlds v2,(x8,x5) # strided load of
                #    D into v2
vmul v3,v1,v2   # vect B * vect D
...
```


Gather-Scatter Operations

- Sparse matrices are common and are usually stored in some compacted form and indirectly accessed.
- An index vector contains the indices of nonzero array elements.
- A *gather/scatter* operation uses the *index vector* along with a base address to fetch/store elements in an array.
- Assume the addresses of *K*, *M*, *A*, and *C* are in *x7*, *x28*, *x5*, and *x6*, respectively.

```
/* sparse array loop */
for (i = 0; i < n; i++)
   A[K[i]] += C[M[i]];
```

```
/* RV64V code */
vsetdcfg 4*FP64     # enable 4 vect regs
vld      v0,x7      # load K[]
vldx     v1,(x5,v0) # load A[K[]]
vld      v2,x28     # load M[]
vldx     v3,(x6,v2) # load C[M[]]
vadd     v1,v1,v3   # v1 += v3
vstx     v1,(x5,v0) # store A[K[]]
vdisable            # disable vect regs
```

SIMD Extensions to GP Processors

- Many GP processors now have SIMD extensions to support simultaneous operations on applications, including for multimedia.
- SIMD extensions are simpler than vector operations.
  - Operate on a fixed number of operands (no *vl* register).
  - Do not support non-unit strides or gather-scatter access.
  - Do not support conditional execution of operations (no vector mask register).
- SIMD operations work on shorter vectors and all operations are typically performed in parallel, as opposed to being pipelined.
- Examples include the x86 SIMD extensions.
  - MultiMedia eXtensions (MMX) in 1996 - used FP registers
  - Streaming Simd Extensions (SSE) 1999 - separate 128-bit registers
  - Advanced Vector eXtensions (AVX) 2010 - separate 256-bit registers
  - Extended Advanced Vector eXtensions (AVX-512) 2016 - separate 512-bit registers

AVX DP Instructions for the x86 Architecture

AVX instruction	Description
VADDPD	Add four packed double-precision operands
VSUBPD	Subtract four packed double-precision operands
VMULPD	Multiply four packed double-precision operands
VDIVPD	Divide four packed double-precision operands
VFMADDPD	Multiply and add four packed double-precision operands
VFMSUBPD	Multiply and subtract four packed double-precision operands
VCMPxx	Compare four packed double-precision operands for EQ, NEQ, LT, LE, GT, GE, ...
VMOVAPD	Move aligned four packed double-precision operands
VBROADCASTSD	Broadcast one double-precision operand to four locations in a 256-bit register

SIMD Extensions Easier to Implement

- Can be added with little cost. For instance, an option to a conventional integer adder can be to not perform carries across specific partitions (e.g. parallel 8-bit additions).
- Require little state as compared to vector architectures, which means it is easier to implement context switches.
- Need much less memory bandwidth.
- Operands in memory for SIMD extensions on many architectures have to be aligned within a L1 DC line, which means one instruction only needs one access to the memory system. However, due to this SIMD alignment problem, it is much harder for compilers to automatically exploit these SIMD extensions.

SIMD Example

- X and Y have to be aligned on a 32 byte boundary.

```
/* sparse array
   loop */
for (i = 0;
     i < 32;
     i++)
   Y[i] =
      a*X[i]+Y[i];
```

```
      /* RISC-V SIMD code */
      fld      f0,a        # load scalar a
      splat.4D f0,f0       # make 4 copies of a
      addi     x28,x5,#256 # address after X
Loop:
      fld.4D   f1,0(x5)    # load X[i]..X[i+3]
      fmul.4D  f1,f1,f0    # a*X[i]..a*X[i+3]
      fld.4D   f2,0(x6)    # load Y[i]..Y[i+3]
      fadd.4D  f2,f2,f1    # a*X[i]+Y[i]..
                           # a*X[i+3]+Y[i+3]
      fsd.4D   f2,0(x6)    # store Y[i]..Y[i+3]
      addi     x5,x5,#32   # incr X index
      addi     x6,x6,#32   # incr Y index
      bne      x28,x5,Loop # loop if not done
```

Graphics Processing Units (GPUs)

- GPUs were first developed as graphics accelerators, where the main emphasis was for the video game industry. But now GPUs are also starting to be used in mainstream computing (GPGPUs).
- GPUs support many types of parallelism (ILP, SIMD, multithreading, MIMD), but work best with DLP applications.
- Some GPUs have their own programming language.
  - CUDA is offered by NVIDIA.
  - OpenCL is vendor-independent for multiple platforms.

NVIDIA GPU Overview

- heterogeneous execution model
  - CPU is the host.
  - GPU is the device.
- CUDA is a C-like programming language to exploit GPU features.
- The programming model is called single instruction, multiple thread (SIMT).

NVIDIA Terminology

- programming abstractions
  - A vectorizable loop is called a *grid*.
  - A *grid* is composed of *thread blocks*, which is equivalent to the body of a strip-mined loop.
  - A *thread block* consists of a set of *CUDA threads*.
  - Each *CUDA thread* processes one element of the vector registers and is equivalent to one iteration of a scalar loop.
- machine object
  - A *warp* is a thread of *PTX* instructions.
  - A *PTX* (Parallel Thread eXecution) instruction is a SIMD instruction.
- processing hardware
  - A *SIMD lane* executes the operations in a *CUDA thread* of SIMD instructions.
  - Multiple *SIMD lanes* within a *thread block* all simultaneously execute the same instruction or are all idle.

GPU Terms Used in this Chapter

Type	Descriptive name	Closest old term outside of GPUs	Official CUDA/NVIDIA GPU term	Short explanation
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 only contains 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

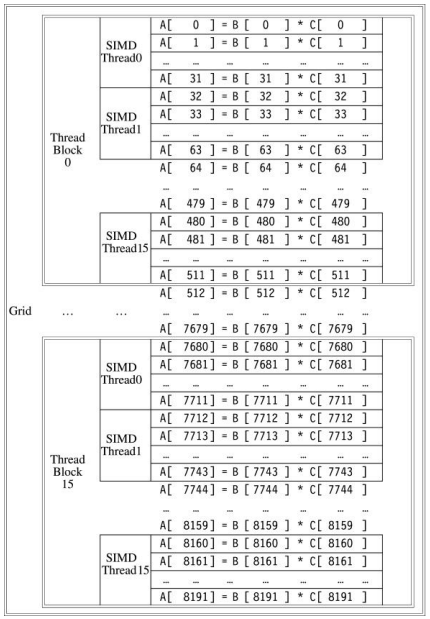
GPU Terms Used in this Chapter (cont.)

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)

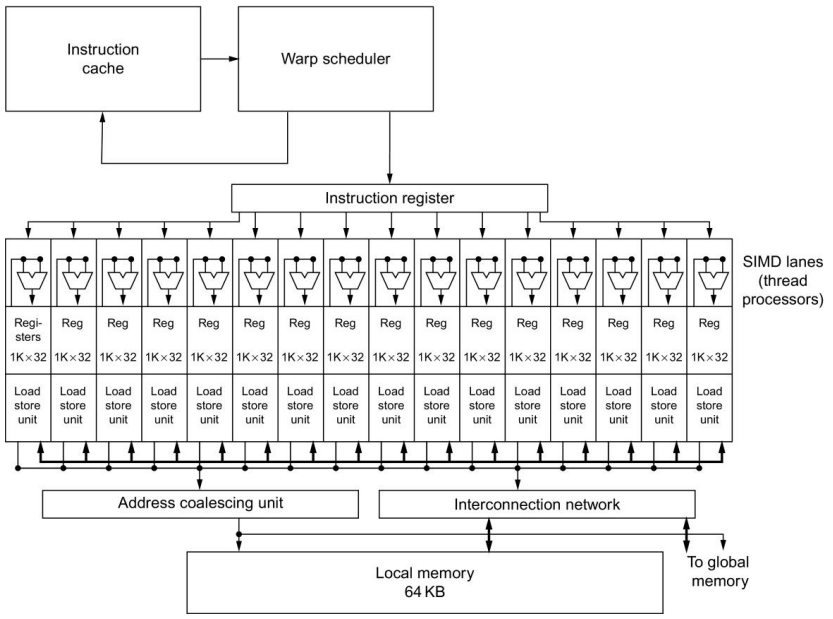
Descriptive Terms to NVIDIA Terms

Type	More descriptive name used in this book	Official CUDA/ NVIDIA term	Short explanation and AMD and OpenCL terms	Official CUDA/NVIDIA definition
Program abstractions	Vectorizable loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more "Thread Blocks" (or bodies of vectorized loop) that can execute in parallel. OpenCL name is "index range." AMD name is "NDRange"	A Grid is an array of Thread Blocks that can execute concurrently, sequentially, or a mixture
	Body of Vectorized loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. These SIMD Threads can communicate via local memory. AMD and OpenCL name is "work group"	A Thread Block is an array of CUDA Threads that execute concurrently and can cooperate and communicate via shared memory and barrier synchronization. A Thread Block has a Thread Block ID within its Grid
	Sequence of SIMD Lane operations	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. AMD and OpenCL call a CUDA Thread a "work item"	A CUDA Thread is a lightweight thread that executes a sequential program and that can cooperate with other CUDA Threads executing in the same Thread Block. A CUDA Thread has a thread ID within its Thread Block
Machine object	A thread of SIMD instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results are stored depending on a per-element mask. AMD name is "wavefront"	A warp is a set of parallel CUDA Threads (e.g., 32) that execute the same instruction together in a multithreaded SIMT/SIMD Processor
	SIMD instruction	PTX instruction	A single SIMD instruction executed across the SIMD Lanes. AMD name is "AMDIL" or "FSAIL" instruction	A PTX instruction specifies an instruction executed by a CUDA Thread

Vector-Vector Multiply Mapping to an NVIDIA Grid



Block Diagram of a Multithreaded SIMD Processor



CUDA Source Example

```
// Invoke DAXPY in C.
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];
}
=>
// Invoke DAXPY in CUDA with 256 CUDA 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];
}
```


Scheduling GPU Instructions

- GPU hardware handles the thread management, not the OS or the application, to improve performance.
- A *thread block* scheduler assigns *thread blocks* to SIMD processors.
- A SIMD thread scheduler allocates SIMD threads within a multithreaded SIMD processor.
- SIMD threads are used to hide memory latency.

GPU Threads

- There are often more SIMD threads on a SIMD processor than can run at one time, which is useful for hiding memory latency.
- Uses a scoreboard to detect SIMD threads ready to run.
- Each SIMD thread has its own PC and each SIMD instruction within a thread simultaneously executes up to *n* operations.
- The *n* parallel functional units to perform a SIMD operation are called *lanes*.
- No dependences can exist between different SIMD threads.
- A *CUDA thread* (vertical cut of SIMD instructions within a SIMD thread) is typically assigned for each loop iteration.
- For each *CUDA thread*, virtual registers are assigned to distinct physical registers and a unique identifier number is used to determine the offsets into arrays so the same code can be invoked both within and across different threads.

Basic PTX GPU Thread Instructions

Group	Instruction	Example	Meaning	Comments
Arithmetic	arithmetic .type = .s32, .u32, .f32, .s64, .u64, .f64			
	add.type	add.f32 d, a, b	d = a + b;	
	sub.type	sub.f32 d, a, b	d = a - b;	
	mul.type	mul.f32 d, a, b	d = a * b;	
	mad.type	mad.f32 d, a, b, c	d = a * b + c;	multiply-add
	div.type	div.f32 d, a, b	d = a / b;	multiple microinstructions
	rem.type	rem.u32 d, a, b	d = a % b;	integer remainder
	abs.type	abs.f32 d, a	d = |a|;	
	neg.type	neg.f32 d, a	d = 0 - a;	
	min.type	min.f32 d, a, b	d = (a < b)? a:b;	floating selects non-NaN
	max.type	max.f32 d, a, b	d = (a > b)? a:b;	floating selects non-NaN
	setp.cmp.type	setp.lt.f32 p, a, b	p = (a < b);	compare and set predicate
	numeric .cmp = eq, ne, lt, le, gt, ge; unordered cmp = equ, neu, ltu, leu, gtu, geu, num, nan			
	mov.type	mov.b32 d, a	d = a;	move
	selp.type	selp.f32 d, a, b, p	d = p? a: b;	select with predicate
	cvt.dtype.atype	cvt.f32.s32 d, a	d = convert(a);	convert atype to dtype
Special function	special .type = .f32 (some .f64)			
	rcp.type	rcp.f32 d, a	d = 1/a;	reciprocal
	sqrt.type	sqrt.f32 d, a	d = sqrt(a);	square root
	rsqrt.type	rsqrt.f32 d, a	d = 1/sqrt(a);	reciprocal square root
	sin.type	sin.f32 d, a	d = sin(a);	sine
	cos.type	cos.f32 d, a	d = cos(a);	cosine
	lg2.type	lg2.f32 d, a	d = log(a)/log(2)	binary logarithm
	ex2.type	ex2.f32 d, a	d = 2 ** a;	binary exponential

Basic PTX GPU Thread Instructions (cont.)

Logical	logic.type = .pred,.b32, .b64			
	and.type	and.b32 d, a, b	d = a & b;	
	or.type	or.b32 d, a, b	d = a | b;	
	xor.type	xor.b32 d, a, b	d = a ^b;	
	not.type	not.b32 d, a, b	d = ~a;	one's complement
	cnot.type	cnot.b32 d, a, b	d = (a==0)? 1:0;	C logical not
	shl.type	shl.b32 d, a, b	d = a << b;	shift left
	shr.type	shr.s32 d, a, b	d = a >> b;	shift right
Memory access	memory.space = .global, .shared, .local, .const; .type = .b8, .u8, .s8, .b16, .b32, .b64			
	ld.space.type	ld.global.b32 d, [a+off]	d = *(a+off);	load from memory space
	st.space.type	st.shared.b32 [d+off], a	*(d+off) = a;	store to memory space
	tex.nd.dtyp.btype	tex.2d.v4.f32.f32 d, a, b	d = tex2d(a, b);	texture lookup
	atom.spc.op.type	atom.global.add.u32 d,[a], b atom.global.cas.b32 d,[a], b, c	atomic { d = *a; *a = op(*a, b); }	atomic read-modify-write operation
	atom.op = and, or, xor, add, min, max, exch, cas; .spc = .global; .type = .b32			
Control flow	branch	@p bra target	if (p) goto target;	conditional branch
	call	call (ret), func, (params)	ret = func(params);	call function
	ret	ret	return;	return from function call
	bar.sync	bar.sync d	wait for threads	barrier synchronization
	exit	exit	exit;	terminate thread execution

CUDA PTX Assembly Code Example

- The first three parallel thread execution (PTX) instructions below determine a unique byte offset that is added to the base of the arrays.
- Special address coalescing hardware recognizes when SIMD lanes within different CUDA threads are collectively issuing sequential addresses and requests a block transfer from the memory system.

```
/* code for one loop iter */
Y[i] = a * X[i] + Y[i];
```

```
/* CUDA PTX code */
shl.u32        R8,blockIdx,9
add.u32        R8,R8,threadIdx
shl.u32        R8,R8,3
ld.global.f64  RD0,[X+R8]
ld.global.f64  RD2,[Y+R8]
mult.f64       RD0,RD0,RD4
add.f64        RD0,RD0,RD2
st.global.f64  [Y+R8],RD0
```


PTX Conditional Branching

- Predicate mask registers are used to handle conditional branches as conditionally executed code.
- Also uses a branch synchronization stack for complex control flow.
  - A branch synchronization entry is pushed when a conditional branch is executed and some lanes diverge (IF-THEN portion), which causes mask bits to be set based on the condition.
  - A branch synchronization marker is used to complement the mask bits (ELSE portion).
  - Another branch synchronization marker is used to pop the stack when the paths converge (end of IF).

PTX Conditional Branching Example

- Assume R8 already has the appropriate offset and that *Push, *Comp, and *Pop indicate the branch synchonization markers inserted by the assembler.

```
/* conditional construct */
...
if (X[i] != 0)
   X[i] = X[i] - Y[i];
else
   X[i] = Z[i];
...
```

```
/* CUDA PTX code */
   ld.global.f64 RD0,[X+R8]
   setp.neq.s32  P1,RD0,#0
   @!P1,bra      ELSE1,*Push
   ld.global.f64 RD2,[Y+R8]
   sub.f64       RD0,RD0,RD2
   st.global.f64 [X+R8],RD0
   @P1,bra       ENDIF1,*Comp
ELSE1:
   ld.global.f64 RD0,[Z+R8]
   st.global.f64 [X+R8],RD0
ENDIF1:
   <next inst>   *Pop
```

Comparison with Vector Computers

- similarities to vector computers
  - Works well on data-level parallel problems.
  - Supports scatter-gather memory operations.
  - Uses mask registers to support conditional execution.
- differences with vector computers
  - Scalar instructions are not intermixed with GPU instructions.
  - Uses multithreading to hide memory latency.
  - Has many functional units, as opposed to a few deeply pipelined vector functional units.

Loop Dependences

- A loop can be parallelized if its iterations are all independent.
- A *loop-carried dependence* is when a data item in one loop iteration depends on a value produced in an earlier iteration.
- The loop below has two loop-carried dependences that prevent it from being parallelized.

```
for (i = 1; i < 100; i++) {
   A[i] = A[i-1] * 2;         /* S1 */
   B[i+1] = B[i] + A[i];      /* S2 */
}
```


Dependence Distance

- The distance in iterations for the loop-carried dependence is called the *dependence distance*. The following loop has a loop-carried dependence with a dependence distance of 4.

```
for (i = 4; i < 100; i++)
   A[i] = A[i-4] * 2 + A[i]; /* S1 */
```

- The greater the dependence distance, the greater the potential ILP by unrolling the loop.
- All four statements in the unrolled loop are independent of each other.

```
for (i = 4; i < 100; i += 4) {
   A[i] = A[i-4] * 2 + A[i];     /* S1 */
   A[i+1] = A[i-3] * 2 + A[i+1]; /* S2 */
   A[i+2] = A[i-2] * 2 + A[i+2]; /* S3 */
   A[i+3] = A[i-1] * 2 + A[i+3]; /* S4 */
}
```


Transforming Loops to Be Parallelizable

- A loop with a loop-carried dependence can be parallelized if the dependences in a loop do not form a cycle.

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

- S1 (use of *B[i]*) is dependent on S2 (set of *B[i+1]*) from the previous iteration. This loop can be transformed so the only dependences are within a single iteration.

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

Eliminating Reductions

- A *reduction* is where a vector is reduced to a single value.
- The following loop cannot be parallelized due to the recurrence on the variable *sum*.

```
for (i = 0; i < 1000; i++)
   sum = sum + x[i]*y[i];
```

- Scalar expansion can be used to parallelize the loop at the expense of adding a simpler loop that cannot be parallelized afterwards.

```
for (i = 0; i < 1000; i++)
   sum[i] = x[i]*y[i];
for (i = 0; i < 1000; i++)
   finalsum = finalsum + sum[i];
```


Pipelining Reductions

- The following loop cannot even be effectively pipelined due to the recurrence on the variable *sum* that results in stalls between iterations.

```
for (i = 0; i < 1000; i++)
   sum = sum + x[i];
```

- Accumulator expansion can be used to minimize these stalls.

```
for (i = 0; i < 1000; i += 4) {
   sum1 = sum1 + x[i];
   sum2 = sum2 + x[i+1];
   sum3 = sum3 + x[i+2];
   sum4 = sum4 + x[i+3];
}
finalsum = sum1 + sum2 + sum3 + sum4;
```


Dependence Analysis

- Dependence analysis attempts to determine if two references can ever access the same variable. Array-oriented dependence analysis is performed when array references can be represented as affine functions of the form *a*i + b*, where *i* is typically a loop index variable, *a* is a constant, and *b* is a constant.
- One simple test is the GCD test, where if we have two elements to the same array indexed by a*j+b and c*k+d, then a loop-carried dependence may exist if GCD(c,a) divides d-b with no remainder.

```
for (i = 0; i < 100; i++)
   X[2*i+3] = X[4*i];
```

- Here, a=2, b=3, c=4, and d=0. So GCD(a,c) = 2, and d-b = -3. -3/2 does not produce an integer quotient, so these two references are not dependent.

Crosscutting Issues

- DLP processors tend to have lower clock rates and simpler issue logic than GP OoO processors.
- GPUs often have special DRAM chips, called GDRAM, that provide higher bandwidth at lower capacity. Today the top-end GPUs use stacked DRAMs, known as high bandwidth memory, to achieve the higher bandwidth. GPU memory controllers maintain separate queues of traffic for different GDRAM banks.
- GPUs currently transfer data between I/O devices and system memory and then between system memory and GPU memory, which can degrade I/O performance.

Fallacies and Pitfalls

- Pitfall: Concentrating on peak performance in vector architectures and ignoring startup-overhead.
- Pitfall: Increasing vector performance, without comparable increases in scalar performance.
- Fallacy: On GPUs, just add more threads if you don't have enough memory performance.