Chapter 4
Data-Level Parallelism in Vector, SIMD, and GPU Architectures

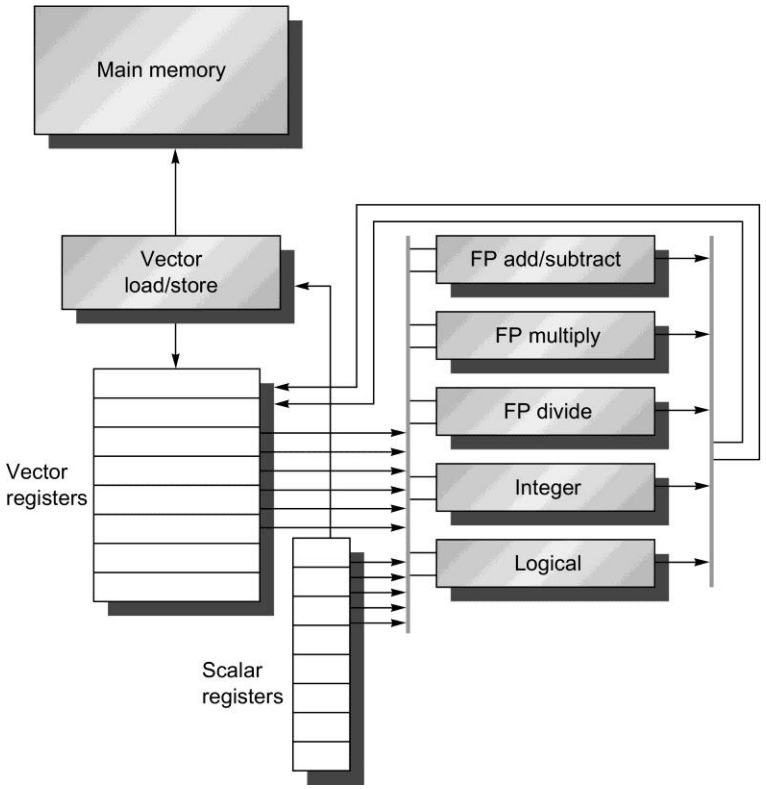

Figure 4.1 The basic structure of a vector architecture, RV64V, which includes a RISC-V scalar architecture. There are also 32 vector registers, and all the functional units are vector functional units. The vector and scalar registers have a significant number of read and write ports to allow multiple simultaneous vector operations. A set of crossbar switches *(thick gray lines)* connects these ports to the inputs and outputs of the vector functional units.

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]
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

Figure 4.2 The RV64V vector instructions. All use the R instruction format. Each vector operation with two operands is shown with both operands being vector (`.vv`), but there are also versions where the second operand is a scalar register (`.vs`) and, when it makes a difference, where the first operand is a scalar register and the second is a vector register (`.sv`). The type and width of the operands are determined by configuring each vector register rather than being supplied by the instruction. In addition to the vector registers and predicate registers, there are two vector control and status registers (CSRs), `vl` and `vctype`, discussed below. The strided and indexed data transfers are also explained later. Once completed, RV64 will surely have more instructions, but the ones in this figure will be included.

Integer	8, 16, 32, and 64 bits	Floating point	16, 32, and 64 bits

Figure 4.3 Data sizes supported for RV64V assuming it also has the single- and double-precision floating-point extensions RVS and RVD. Adding RVV to such a RISC-V design means the scalar unit must also add RVH, which is a scalar instruction extension to support half-precision (16-bit) IEEE 754 floating point. Because RV32V would not have doubleword scalar operations, it could drop 64-bit integers from the vector unit. If a RISC-V implementation didn't include RVS or RVD, it could omit the vector floating-point instructions.

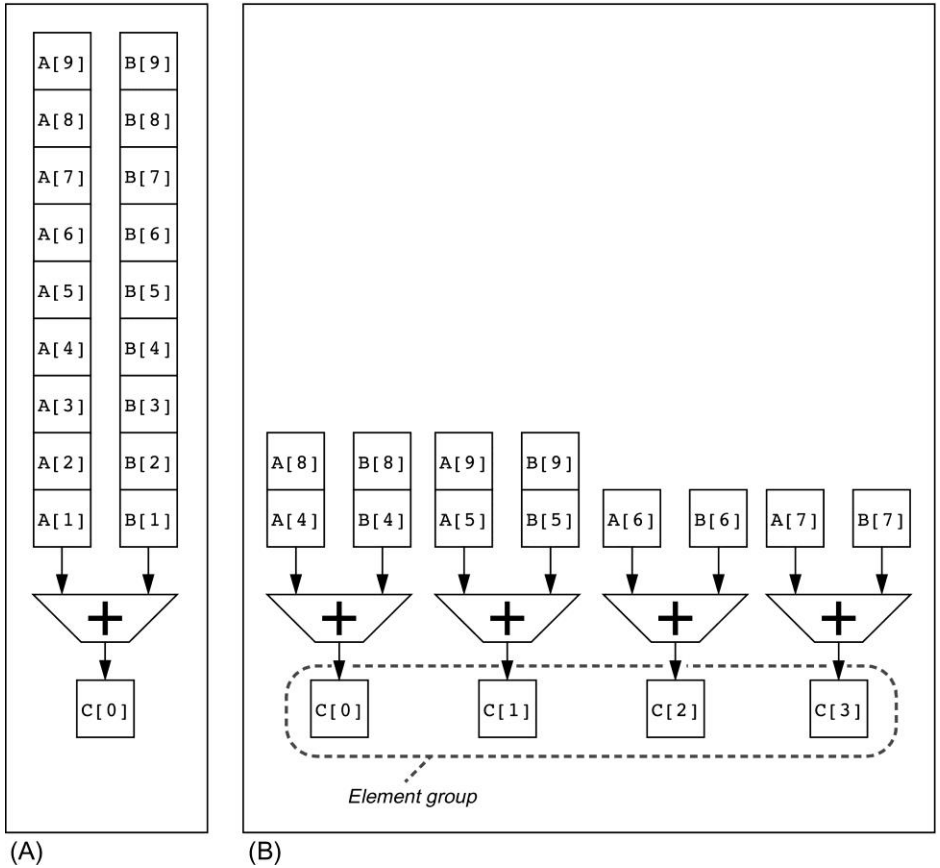

Figure 4.4 Using multiple functional units to improve the performance of a single vector add instruction, C = A + B. The vector processor (A) on the left has a single add pipeline and can complete one addition per clock cycle. The vector processor (B) on the right has four add pipelines and can complete four additions per clock cycle. The elements within a single vector add instruction are interleaved across the four pipelines. The set of elements that move through the pipelines together is termed an *element group*. Reproduced with permission from Asanovic, K., 1998. Vector Microprocessors (Ph.D. thesis). Computer Science Division, University of California, Berkeley.

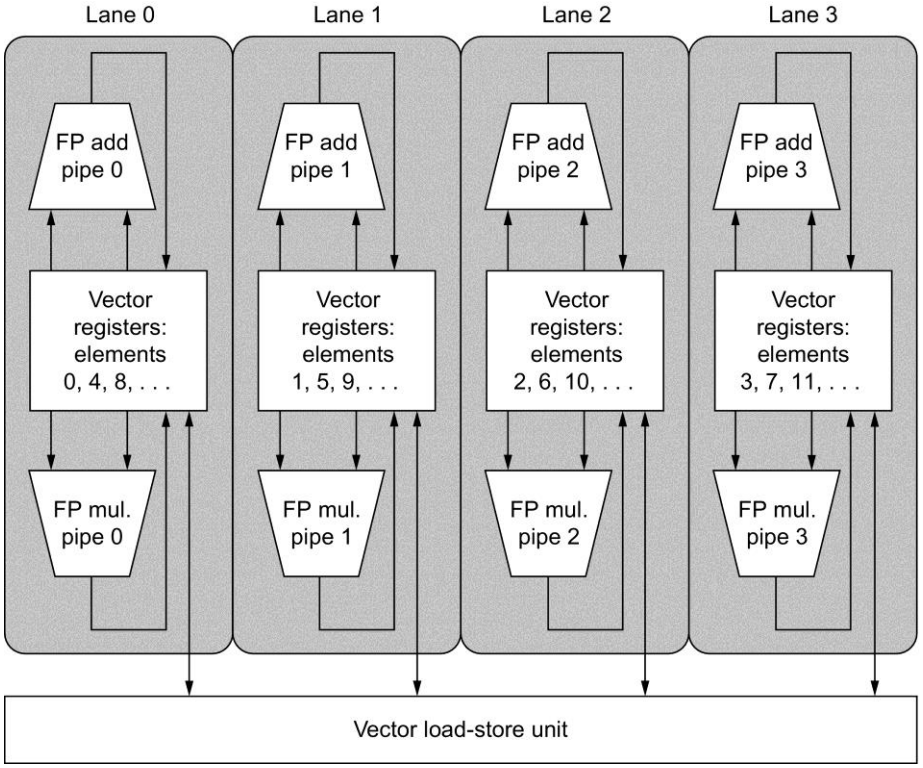

Figure 4.5 Structure of a vector unit containing four lanes. The vector register memory is divided across the lanes, with each lane holding every fourth element of each vector register. The figure shows three vector functional units: an FP add, an FP multiply, and a load-store unit. Each of the vector arithmetic units contains four execution pipelines, one per lane, which act in concert to complete a single vector instruction. Note how each section of the vector register file needs to provide only enough ports for pipelines local to its lane. This figure does not show the path to provide the scalar operand for vector-scalar instructions, but the scalar processor (or Control Processor) broadcasts a scalar value to all lanes.

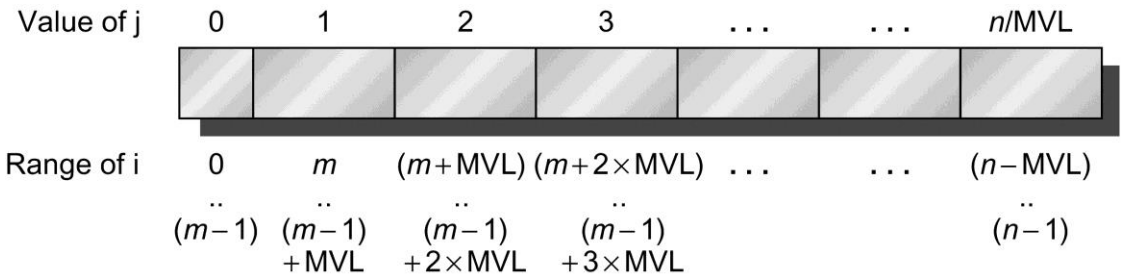

Figure 4.6 A vector of arbitrary length processed with strip mining. All blocks but the first are of length MVL, utilizing the full power of the vector processor. In this figure, we use the variable *m* for the expression (n % MVL). (The C operator % is modulo.)

Benchmark name	Operations executed in vector mode, compiler-optimized	Operations executed in vector mode, with programmer aid	Speedup from hint optimization
BDNA	96.1%	97.2%	1.52
MG3D	95.1%	94.5%	1.00
FLO52	91.5%	88.7%	N/A
ARC3D	91.1%	92.0%	1.01
SPEC77	90.3%	90.4%	1.07
MDG	87.7%	94.2%	1.49
TRFD	69.8%	73.7%	1.67
DYFESM	68.8%	65.6%	N/A
ADM	42.9%	59.6%	3.60
OCEAN	42.8%	91.2%	3.92
TRACK	14.4%	54.6%	2.52
SPICE	11.5%	79.9%	4.06
QCD	4.2%	75.1%	2.15

Figure 4.7 Level of vectorization among the Perfect Club benchmarks when executed on the Cray Y-MP (Vajapeyam, 1991). The first column shows the vectorization level obtained with the compiler without hints, and the second column shows the results after the codes have been improved with hints from a team of Cray Research programmers.

Instruction category	Operands
Unsigned add/subtract	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Maximum/minimum	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Average	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Shift right/left	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Floating point	Sixteen 16-bit, eight 32-bit, four 64-bit, or two 128-bit

Figure 4.8 Summary of typical SIMD multimedia support for 256-bit-wide operations. Note that the IEEE 754-2008 floating-point standard added half-precision (16-bit) and quad-precision (128-bit) floating-point operations.

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

Figure 4.9 AVX instructions for x86 architecture useful in double-precision floating-point programs. Packed-double for 256-bit AVX means four 64-bit operands executed in SIMD mode. As the width increases with AVX, it is increasingly important to add data permutation instructions that allow combinations of narrow operands from different parts of the wide registers. AVX includes instructions that shuffle 32-bit, 64-bit, or 128-bit operands within a 256-bit register. For example, BROADCAST replicates a 64-bit operand four times in an AVX register. AVX also includes a large variety of fused multiply-add/subtract instructions; we show just two here.

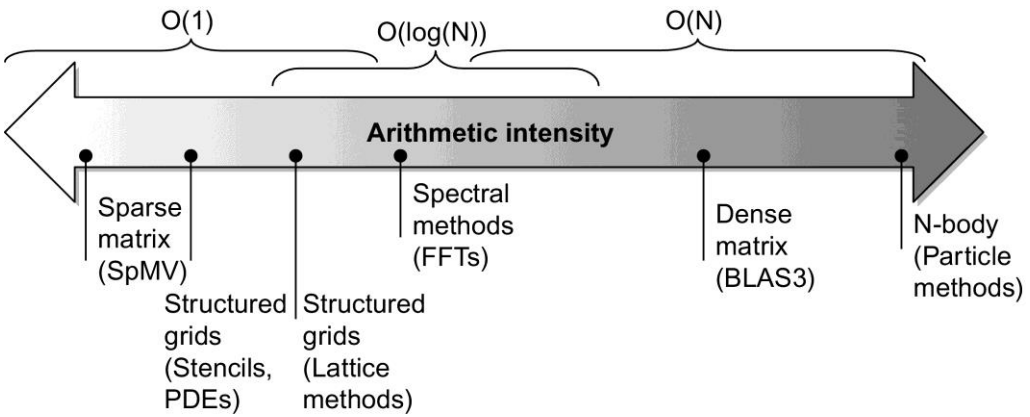

Figure 4.10 Arithmetic intensity, specified as the number of floating-point operations to run the program divided by the number of bytes accessed in main memory (Williams et al., 2009). Some kernels have an arithmetic intensity that scales with problem size, such as a dense matrix, but there are many kernels with arithmetic intensities independent of problem size.

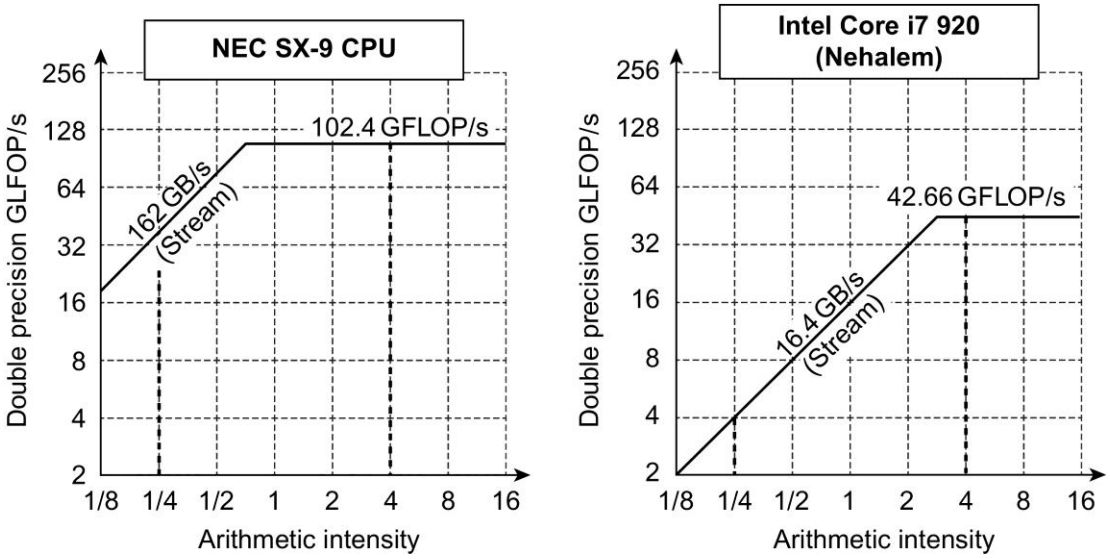

Figure 4.11 Roofline model for one NEC SX-9 vector processor on the left and the Intel Core i7 920 multicore computer with SIMD extensions on the right (Williams et al., 2009). This Roofline is for unit-stride memory accesses and double-precision floating-point performance. NEC SX-9 is a vector supercomputer announced in 2008 that cost millions of dollars. It has a peak DP FP performance of 102.4 GFLOP/s and a peak memory bandwidth of 162 GB/s from the Stream benchmark. The Core i7 920 has a peak DP FP performance of 42.66 GFLOP/s and a peak memory bandwidth of 16.4 GB/s. The dashed vertical lines at an arithmetic intensity of 4 FLOP/byte show that both processors operate at peak performance. In this case, the SX-9 at 102.4 FLOP/s is 2.4 × faster than the Core i7 at 42.66 GFLOP/s. At an arithmetic intensity of 0.25 FLOP/byte, the SX-9 is 10 × faster at 40.5 GFLOP/s versus 4.1 GFLOP/s for the Core i7.

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
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)

Figure 4.12 Quick guide to GPU terms used in this chapter. We use the first column for hardware terms. Four groups cluster these 11 terms. From top to bottom: program abstractions, machine objects, processing hardware, and memory hardware. Figure 4.21 on page 312 associates vector terms with the closest terms here, and Figure 4.24 on page 317 and Figure 4.25 on page 318 reveal the official CUDA/NVIDIA and AMD terms and definitions along with the terms used by OpenCL.

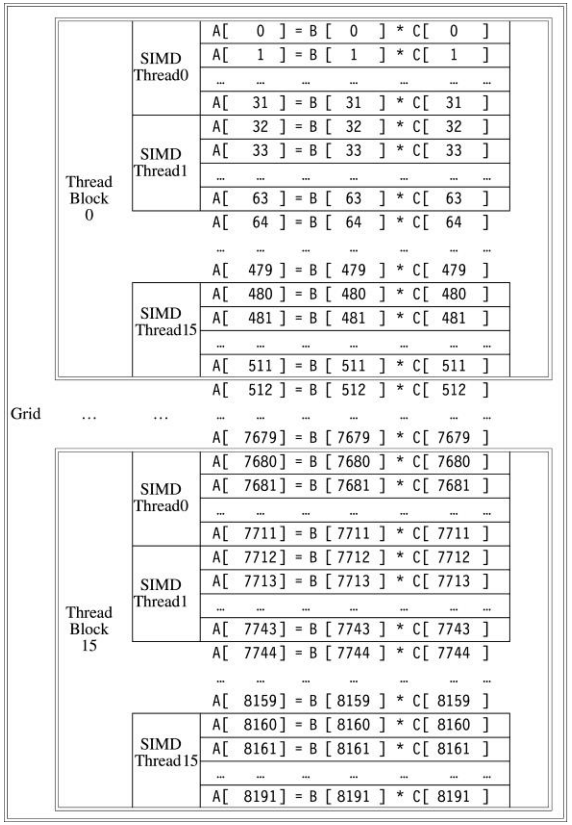

Figure 4.13 The mapping of a Grid (vectorizable loop), Thread Blocks (SIMD basic blocks), and threads of SIMD instructions to a vector-vector multiply, with each vector being 8192 elements long. Each thread of SIMD instructions calculates 32 elements per instruction, and in this example, each Thread Block contains 16 threads of SIMD instructions and the Grid contains 16 Thread Blocks. The hardware Thread Block Scheduler assigns Thread Blocks to multithreaded SIMD Processors, and the hardware Thread Scheduler picks which thread of SIMD instructions to run each clock cycle within a SIMD Processor. Only SIMD Threads in the same Thread Block can communicate via local memory. (The maximum number of SIMD Threads that can execute simultaneously per Thread Block is 32 for Pascal GPUs.)

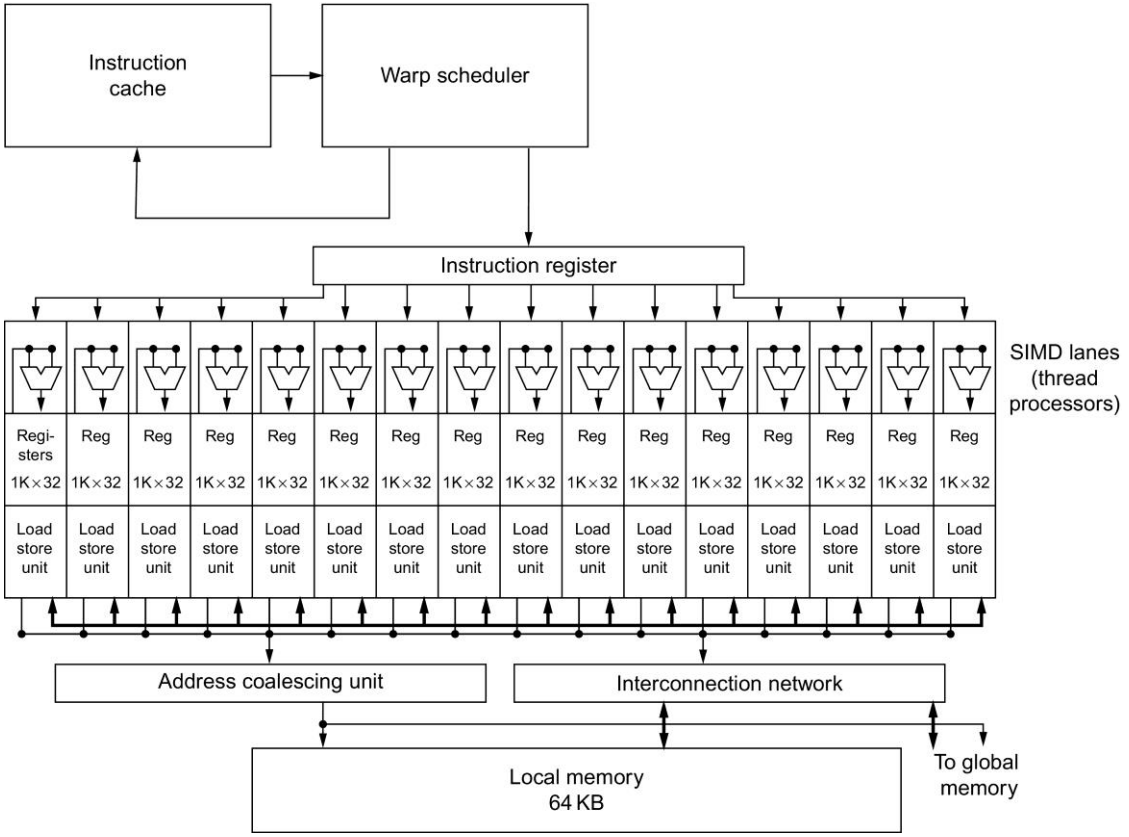

Figure 4.14 Simplified block diagram of a multithreaded SIMD Processor. It has 16 SIMD Lanes. The SIMD Thread Scheduler has, say, 64 independent threads of SIMD instructions that it schedules with a table of 64 program counters (PCs). Note that each lane has 1024 32-bit registers.

Figure 4.15 Full-chip block diagram of the Pascal P100 GPU. It has 56 multithreaded SIMD Processors, each with an L1 cache and local memory, 32 L2 units, and a memory-bus width of 4096 data wires. (It has 60 blocks, with four spares to improve yield.) The P100 has 4 HBM2 ports supporting up to 16 GB of capacity. It contains 15.4 billion transistors.

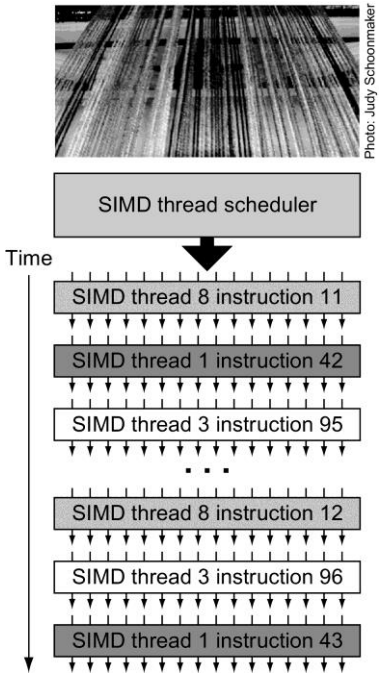

Figure 4.16 Scheduling of threads of SIMD instructions. The scheduler selects a ready thread of SIMD instructions and issues an instruction synchronously to all the SIMD Lanes executing the SIMD Thread. Because threads of SIMD instructions are independent, the scheduler may select a different SIMD Thread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

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
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

Figure 4.17 Basic PTX GPU thread instructions.

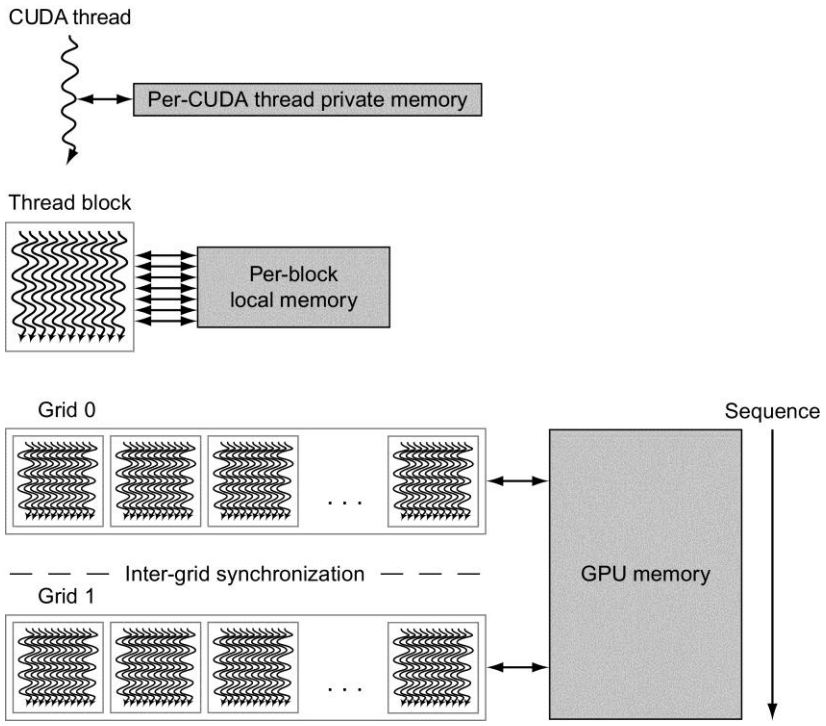

Figure 4.18 GPU memory structures. GPU memory is shared by all Grids (vectorized loops), local memory is shared by all threads of SIMD instructions within a Thread Block (body of a vectorized loop), and private memory is private to a single CUDA Thread. Pascal allows preemption of a Grid, which requires that all local and private memory be able to be saved in and restored from global memory. For completeness sake, the GPU can also access CPU memory via the PCIe bus. This path is commonly used for a final result when its address is in host memory. This option eliminates a final copy from the GPU memory to the host memory.

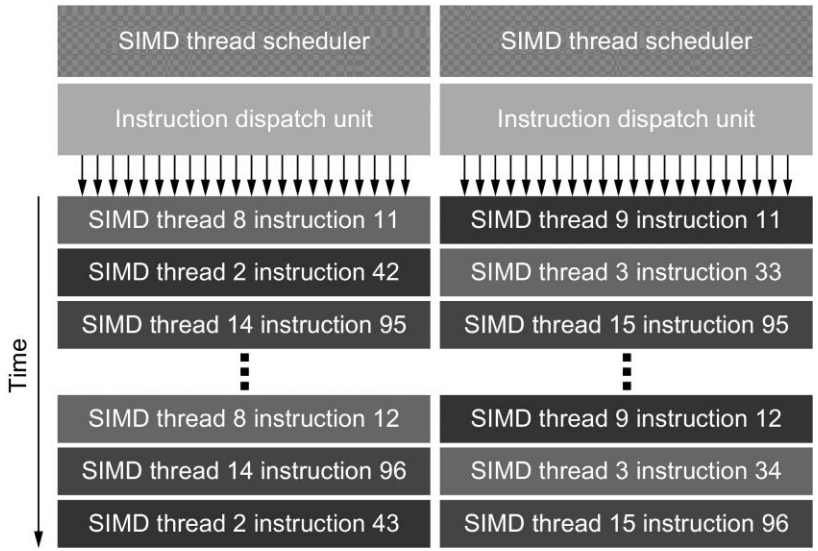

Figure 4.19 Block diagram of Pascal's dual SIMD Thread scheduler. Compare this design to the single SIMD Thread design in Figure 4.16.

Figure 4.20 Block diagram of the multithreaded SIMD Processor of a Pascal GPU. Each of the 64 SIMD Lanes (cores) has a pipelined floating-point unit, a pipelined integer unit, some logic for dispatching instructions and operands to these units, and a queue for holding results. The 64 SIMD Lanes interact with 32 double-precision ALUs (DP units) that perform 64-bit floating-point arithmetic, 16 load-store units (LD/STs), and 16 special function units (SFUs) that calculate functions such as square roots, reciprocals, sines, and cosines.

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	—	Because a vector instruction (PTX instruction) takes just 2 cycles on Pascal to complete, a chime is short in GPUs. Pascal has two execution units that support the most common floating-point instructions that are used alternately, so the effective issue rate is 1 instruction every clock cycle
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 in vector architectures
	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	Very similar; both are essentially functional units with registers
	Vector Registers	SIMD Lane Registers	The equivalent of a vector register is the same register in all 16 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 256 in Pascal, so the maximum number of vector registers is 256
	Main Memory	GPU Memory	Memory for GPU versus system memory in vector case

Figure 4.21 GPU equivalent to vector terms.

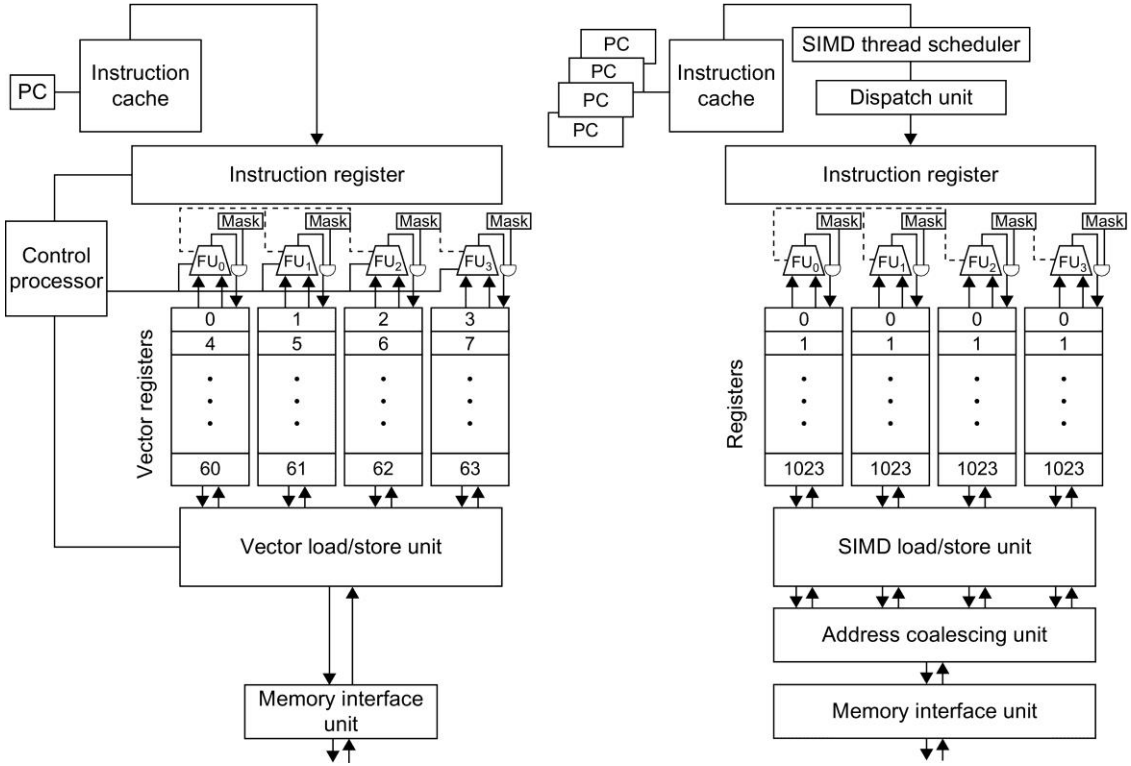

Figure 4.22 A vector processor with four lanes on the left and a multithreaded SIMD Processor of a GPU with four SIMD Lanes on the right. (GPUs typically have 16 or 32 SIMD Lanes.) The Control Processor supplies scalar operands for scalar-vector operations, increments addressing for unit and nonunit stride accesses to memory, and performs other accounting-type operations. Peak memory performance occurs only in a GPU when the Address Coalescing Unit can discover localized addressing. Similarly, peak computational performance occurs when all internal mask bits are set identically. Note that the SIMD Processor has one PC per SIMD Thread to help with multithreading.

Feature	Multicore with SIMD	GPU
SIMD Processors	4–8	8–32
SIMD Lanes/Processor	2–4	up to 64
Multithreading hardware support for SIMD Threads	2–4	up to 64
Typical ratio of single-precision to double-precision performance	2:1	2:1
Largest cache size	40 MB	4 MB
Size of memory address	64-bit	64-bit
Size of main memory	up to 1024 GB	up to 24 GB
Memory protection at level of page	Yes	Yes
Demand paging	Yes	Yes
Integrated scalar processor/SIMD Processor	Yes	No
Cache coherent	Yes	Yes on some systems

Figure 4.23 Similarities and differences between multicore with multimedia SIMD extensions and recent GPUs.

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

Figure 4.24 Conversion from terms used in this chapter to official NVIDIA/CUDA and AMD jargon. OpenCL names are given in the book's definitions.

Type	More descriptive name used in this book	Official CUDA/ NVIDIA term	Short explanation and AMD and OpenCL terms	Official CUDA/NVIDIA definition
Processing hardware	Multithreaded SIMD processor	Streaming multiprocessor	Multithreaded SIMD Processor that executes thread of SIMD instructions, independent of other SIMD Processors. Both AMD and OpenCL call it a "compute unit." However, the CUDA programmer writes program for one lane rather than for a "vector" of multiple SIMD Lanes	A streaming multiprocessor (SM) is a multithreaded SIMT/SIMD Processor that executes warps of CUDA Threads. A SIMT program specifies the execution of one CUDA Thread, rather than a vector of multiple SIMD Lanes
	Thread Block Scheduler	Giga Thread Engine	Assigns multiple bodies of vectorized loop to multithreaded SIMD Processors. AMD name is "Ultra-Threaded Dispatch Engine"	Distributes and schedules Thread Blocks of a grid to streaming multiprocessors as resources become available
	SIMD Thread scheduler	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. AMD name is "Work Group Scheduler"	A warp scheduler in a streaming multiprocessor schedules warps for execution when their next instruction is ready to execute
	SIMD Lane	Thread processor	Hardware SIMD Lane that executes the operations in a thread of SIMD instructions on a single element. Results are stored depending on mask. OpenCL calls it a "processing element." AMD name is also "SIMD Lane"	A thread processor is a datapath and register file portion of a streaming multiprocessor that executes operations for one or more lanes of a warp
Memory hardware	GPU Memory	Global memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU. OpenCL calls it "global memory"	Global memory is accessible by all CUDA Threads in any Thread Block in any grid; implemented as a region of DRAM, and may be cached
	Private memory	Local memory	Portion of DRAM memory private to each SIMD Lane. Both AMD and OpenCL call it "private memory"	Private "thread-local" memory for a CUDA Thread; implemented as a cached region of DRAM
	Local memory	Shared memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors. OpenCL calls it "local memory." AMD calls it "group memory"	Fast SRAM memory shared by the CUDA Threads composing a Thread Block, and private to that Thread Block. Used for communication among CUDA Threads in a Thread Block at barrier synchronization points
	SIMD Lane registers	Registers	Registers in a single SIMD Lane allocated across body of vectorized loop. AMD also calls them "registers"	Private registers for a CUDA Thread; implemented as multithreaded register file for certain lanes of several warps for each thread processor

Figure 4.25 Conversion from terms used in this chapter to official NVIDIA/CUDA and AMD jargon. Note that our descriptive terms "local memory" and "private memory" use the OpenCL terminology. NVIDIA uses SIMT (single-instruction multiple-thread) rather than SIMD to describe a streaming multiprocessor. SIMT is preferred over SIMD because the per-thread branching and control flow are unlike any SIMD machine.

	NVIDIA Tegra 2	NVIDIA Tesla P100
Market	Automotive, Embedded, Console, Tablet	Desktop, server
System processor	Six-Core ARM (2 Denver2 +4 A57)	Not applicable
System interface	Not applicable	PCI Express × 16 Gen 3
System interface bandwidth	Not applicable	16 GB/s (each direction), 32 GB/s (total)
Clock rate	1.5 GHz	1.4 GHz
SIMD multiprocessors	2	56
SIMD Lanes/SIMD multiprocessor	128	64
Memory interface	128-bit LP-DDR4	4096-bit HBM2
Memory bandwidth	50 GB/s	732 GB/s
Memory capacity	up to 16 GB	up to 16 GB
Transistors	7 billion	15.3 billion
Process	TSMC 16 nm FinFET	TSMC 16 nm FinFET
Die area	147 mm^2	645 mm^2
Power	20 W	300 W

Figure 4.26 Key features of the GPUs for embedded clients and servers.

	Core i7-960	GTX 280	Ratio 280/i7
Number of processing elements (cores or SMs)	4	30	7.5
Clock frequency (GHz)	3.2	1.3	0.41
Die size	263	576	2.2
Technology	Intel 45 nm	TSMC 65 nm	1.6
Power (chip, not module)	130	130	1.0
Transistors	700 M	1400 M	2.0
Memory bandwidth (GB/s)	32	141	4.4
Single-precision SIMD width	4	8	2.0
Double-precision SIMD width	2	1	0.5
Peak single-precision scalar FLOPS (GFLOP/S)	26	117	4.6
Peak single-precision SIMD FLOPS (GFLOP/S)	102	311–933	3.0–9.1
(SP 1 add or multiply)	N.A.	(311)	(3.0)
(SP 1 instruction fused multiply-adds)	N.A.	(622)	(6.1)
(Rare SP dual issue fused multiply-add and multiply)	N.A.	(933)	(9.1)
Peak double-precision SIMD FLOPS (GFLOP/S)	51	78	1.5

Figure 4.27 Intel Core i7-960 and NVIDIA GTX 280. The rightmost column shows the ratios of GTX 280 to Core i7. For single-precision SIMD FLOPS on the GTX 280, the higher speed (933) comes from a very rare case of dual issuing of fused multiply-add and multiply. More reasonable is 622 for single fused multiply-adds. Note that these memory bandwidths are higher than in Figure 4.28 because these are DRAM pin bandwidths and those in Figure 4.28 are at the processors as measured by a benchmark program. From Table 2 in Lee, W.V., et al., 2010. Debunking the 100 × GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In: Proc. 37th Annual Int'l. Symposium on Computer Architecture (ISCA), June 19–23, 2010, Saint-Malo, France.

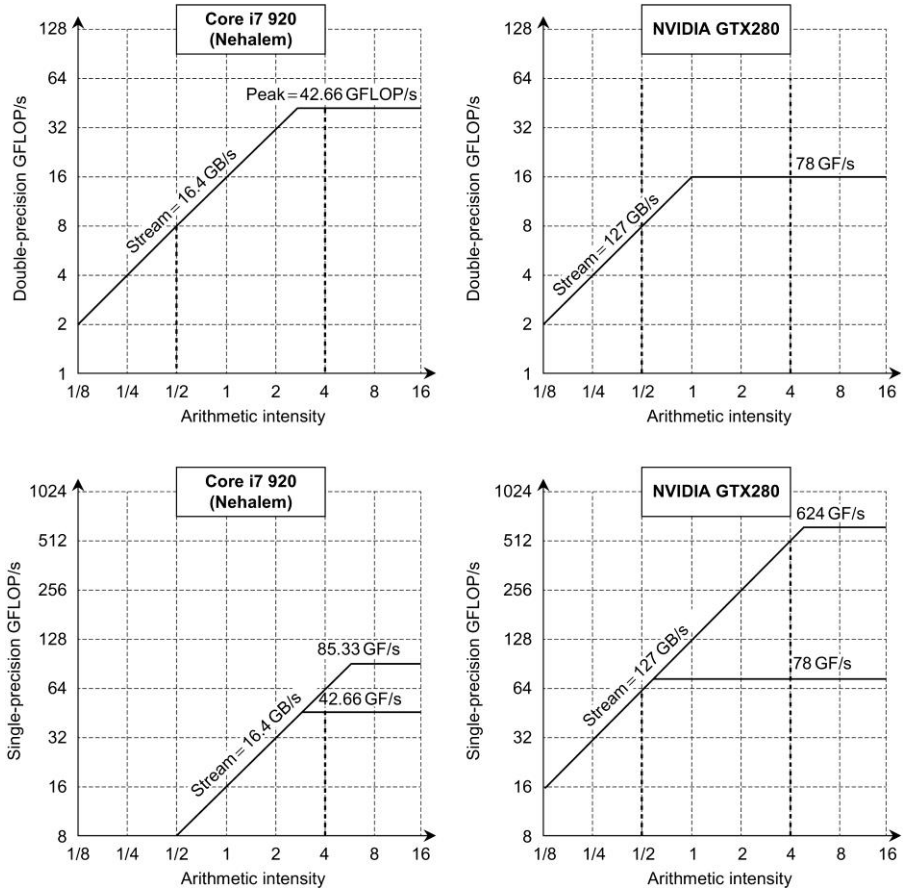

Figure 4.28 Roofline model (Williams et al. 2009). These rooflines show double-precision floating-point performance in the top row and single-precision performance in the bottom row. (The DP FP performance ceiling is also in the bottom row to give perspective.) The Core i7 920 on the left has a peak DP FP performance of 42.66 GFLOP/s, a SP FP peak of 85.33 GFLOP/s, and a peak memory bandwidth of 16.4 GB/s. The NVIDIA GTX 280 has a DP FP peak of 78 GFLOP/s, SP FP peak of 624 GFLOP/s, and 127 GB/s of memory bandwidth. The dashed vertical line on the left represents an arithmetic intensity of 0.5 FLOP/byte. It is limited by memory bandwidth to no more than 8 DP GFLOP/s or 8 SP GFLOP/s on the Core i7. The dashed vertical line to the right has an arithmetic intensity of 4 FLOP/byte. It is limited only computationally to 42.66 DP GFLOP/s and 64 SP GFLOP/s on the Core i7 and to 78 DP GFLOP/s and 512 DP GFLOP/s on the GTX 280. To hit the highest computation rate on the Core i7, you need to use all 4 cores and SSE instructions with an equal number of multiplies and adds. For the GTX 280, you need to use fused multiply-add instructions on all multithreaded SIMD Processors.

Kernel	Application	SIMD	TLP	Characteristics
SGEMM (**SGEMM**)	Linear algebra	Regular	Across 2D tiles	Compute bound after tiling
Monte Carlo (**MC**)	Computational finance	Regular	Across paths	Compute bound
Convolution (**Conv**)	Image analysis	Regular	Across pixels	Compute bound; BW bound for small filters
FFT (**FFT**)	Signal processing	Regular	Across smaller FFTs	Compute bound or BW bound depending on size
SAXPY (**SAXPY**)	Dot product	Regular	Across vector	BW bound for large vectors
LBM (**LBM**)	Time migration	Regular	Across cells	BW bound
Constraint solver (**Solv**)	Rigid body physics	Gather/ Scatter	Across constraints	Synchronization bound
SpMV (**SpMV**)	Sparse solver	Gather	Across nonzero	BW bound for typical large matrices
GJK (**GJK**)	Collision detection	Gather/ Scatter	Across objects	Compute bound
Sort (**Sort**)	Database	Gather/ Scatter	Across elements	Compute bound
Ray casting (**RC**)	Volume rendering	Gather	Across rays	4–8 MB first level working set; over 500 MB last level working set
Search (**Search**)	Database	Gather/ Scatter	Across queries	Compute bound for small tree, BW bound at bottom of tree for large tree
Histogram (**Hist**)	Image analysis	Requires conflict detection	Across pixels	Reduction/synchronization bound
Bilateral (**Bilat**)	Image analysis	Regular	Across pixels	Compute bound

Figure 4.29 Throughput computing kernel characteristics. The name in parentheses identifies the benchmark name in this section. The authors suggest that code for both machines had equal optimization effort. From Table 1 in Lee, W.V., et al., 2010. Debunking the 100 × GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In: Proc. 37th Annual Int'l. Symposium on Computer Architecture (ISCA), June 19–23, 2010, Saint-Malo, France.

Kernel	Units	Core i7-960	GTX 280	GTX 280/ i7-960
SGEMM	GFLOP/s	94	364	3.9
MC	Billion paths/s	0.8	1.4	1.8
Conv	Million pixels/s	1250	3500	2.8
FFT	GFLOP/s	71.4	213	3.0
SAXPY	GB/s	16.8	88.8	5.3
LBM	Million lookups/s	85	426	5.0
Solv	Frames/s	103	52	0.5
SpMV	GFLOP/s	4.9	9.1	1.9
GJK	Frames/s	67	1020	15.2
Sort	Million elements/s	250	198	0.8
RC	Frames/s	5	8.1	1.6
Search	Million queries/s	50	90	1.8
Hist	Million pixels/s	1517	2583	1.7
Bilat	Million pixels/s	83	475	5.7

Figure 4.30 Raw and relative performance measured for the two platforms. In this study, SAXPY is used only as a measure of memory bandwidth, so the right unit is GB/s and not GFLOP/s. Based on Table 3 in Lee, W.V., et al., 2010. Debunking the 100 × GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In: Proc. 37th Annual Int'l. Symposium on Computer Architecture (ISCA), June 19–23, 2010, Saint-Malo, France.

	Xeon Platinum 8180	P100	Ratio P100/Xeon
Number of processing elements (cores or SMs)	28	56	2.0
Clock frequency (GHz)	2.5	1.3	0.52
Die size	N.A.	610 mm^2	–
Technology	Intel 14 nm	TSMC 16 nm	1.1
Power (chip, not module)	80 W	300 W	3.8
Transistors	N.A.	15.3 B	–
Memory bandwidth (GB/s)	199	732	3.7
Single-precision SIMD width	16	8	0.5
Double-precision SIMD width	8	4	0.5
Peak single-precision SIMD FLOPS (GFLOP/s)	4480	10,608	2.4
Peak double-precision SIMD FLOPS (GFLOP/s)	2240	5304	2.4

Figure 4.31 Intel Xeon ?? and NVIDIA P100. The rightmost column shows the ratios of P100 to the Xeon. Note that these memory bandwidths are higher than in Figure 4.28 because these are DRAM pin bandwidths and those in Figure 4.28 are at the processors as measured by a benchmark program.

Kernel	Units	Xeon Platinum 8180	P100	P100/Xeon	GTX 280/i7-960
SGEMM	GFLOP/s	3494	6827	2.0	3.9
DGEMM	GFLOP/s	1693	3490	2.1	—
FFT-S	GFLOP/s	410	1820	4.4	3.0
FFT-D	GFLOP/s	190	811	4.2	—
SAXPY	GB/s	207	544	2.6	5.3
DAXPY	GB/s	212	556	2.6	—

Figure 4.32 Raw and relative performance measured for modern versions of the two platforms as compared to the relative performance of the original platforms. Like Figure 4.30, SAXPY and DAXPY are used only as a measure of memory bandwidth, so the proper unit is GB/s and not GFLOP/s.

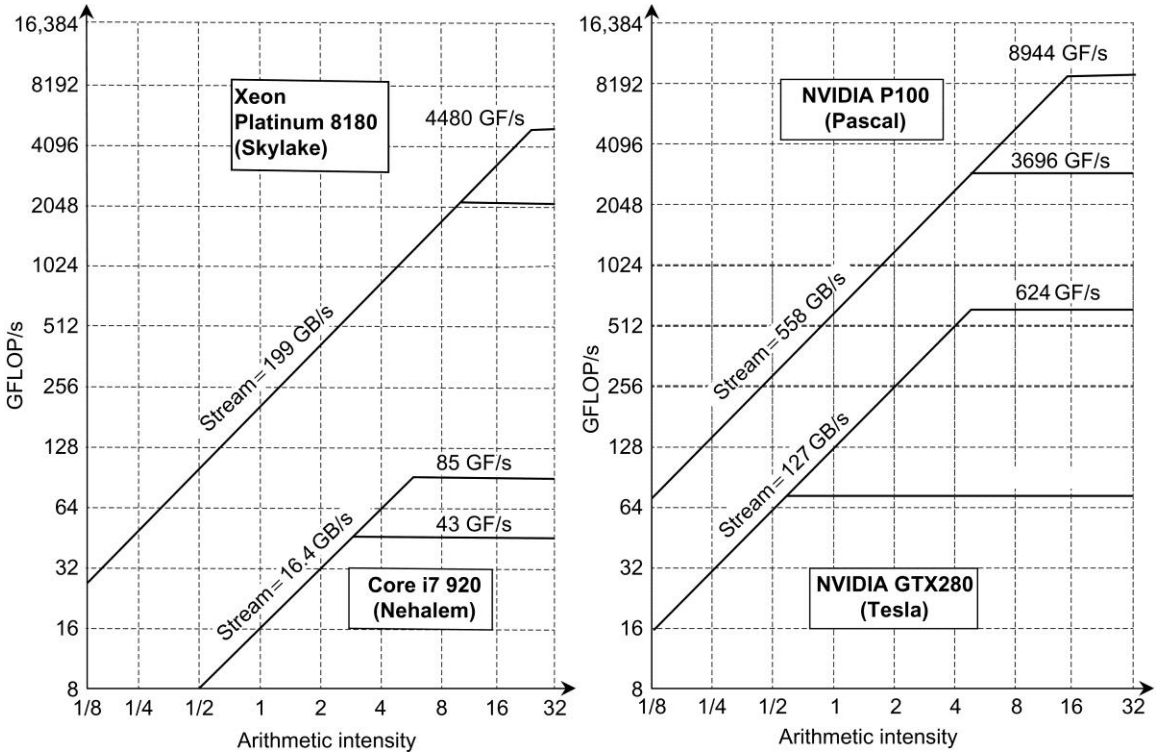

Figure 4.33 Roofline models of older and newer CPUs versus older and newer GPUs. The higher roofline for each computer is single-precision floating-point performance, and the lower one is double-precision performance.

Processor	Minimum rate for any loop (MFLOPS)	Maximum rate for any loop (MFLOPS)	Harmonic mean of all 24 loops (MFLOPS)
MIPS M/120-5	0.80	3.89	1.85
Stardent-1500	0.41	10.08	1.72

Figure 4.34 Performance measurements for the Livermore Fortran kernels on two different processors. Both the MIPS M/120-5 and the Stardent-1500 (formerly the Ardent Titan-1) use a 16.7 MHz MIPS R2000 chip for the main CPU. The Stardent-1500 uses its vector unit for scalar FP and has about half the scalar performance (as measured by the minimum rate) of the MIPS M/120-5, which uses the MIPS R2010 FP chip. The vector processor is more than a factor of 2.5 × faster for a highly vectorizable loop (maximum rate). However, the lower scalar performance of the Stardent-1500 negates the higher vector performance when total performance is measured by the harmonic mean on all 24 loops.

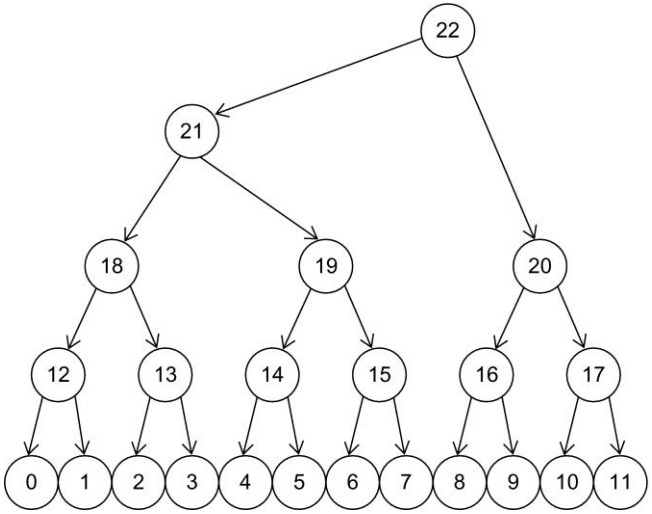

Figure 4.35 Sample tree.