4

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

We call these algorithms data parallel algorithms because their parallelism comes from simultaneous operations across large sets of data rather than from multiple threads of control.

W. Daniel Hillis and Guy L. Steele,
"Data parallel algorithms," *Commun. ACM* (1986)

If you were plowing a field, which would you rather use: two strong oxen or 1024 chickens?

Seymour Cray, Father of the Supercomputer
(arguing for two powerful vector processors versus many simple processors)

Computer Architecture. https://doi.org/10.1016/B978-0-443-15406-5.00004-3

4.1 Introduction

A question for the single instruction multiple data (SIMD) architecture, which Chapter 1 introduced, has always been just how wide a set of applications has significant data-level parallelism (DLP). Nearly 60 years after the SIMD classification was proposed (Flynn, 1966), the answer is not only the matrix-oriented computations of scientific computing but also the media-oriented image and sound processing and machine learning algorithms, as we will see in Chapter 7. Note these are all variations of linear algebra. Since a multiple instruction multiple data (MIMD) architecture needs to fetch one instruction per data operation, SIMD is potentially more energy-efficient as a single instruction can launch many data operations. These two answers make SIMD attractive for personal mobile devices as well as for servers. Finally, perhaps the biggest advantage of SIMD versus MIMD is that the programmer continues to think sequentially yet achieves parallel speedup by having parallel data operations.

This chapter covers three variations of SIMD: vector architectures, multimedia SIMD instruction set extensions, and graphics processing units (GPUs).[1]

The first variation, which predates the other two by more than 30 years, extends pipelined execution to many data operations. These *vector architectures* are easier to understand and to compile to than other SIMD variations, but they were considered too expensive for microprocessors until recently. Part of that expense was in transistors, and part was in the cost of sufficient dynamic random access memory (DRAM) bandwidth, given the widespread reliance on caches to meet memory performance demands on conventional microprocessors.

The second SIMD variation borrows from the SIMD name to mean basically simultaneous parallel data operations and is now found in most instruction set architectures that support multimedia applications. For x86 architectures, the SIMD instruction extensions started with the MMX (multimedia extensions) in 1996, which were followed by several SSE (streaming SIMD extensions) versions in the next decade, and they continue until this day with AVX (advanced vector extensions) and AMX (advanced matrix extensions). To get the highest computation rate from an x86 computer, you often need to use these SIMD instructions, especially for floating-point programs.

The third variation on SIMD comes from the graphics accelerator community, offering higher potential performance than is found in traditional multicore computers today. Although GPUs share features with vector architectures, they have their own distinguishing characteristics, in part because of the ecosystem in

[1] This chapter is based on material in Appendix F, "Vector Processors," by Krste Asanovic, and Appendix G, "Hardware and Software for VLIW and EPIC" from the 5th edition of this book; on material in Appendix A, "Graphics and Computing GPUs," by John Nickolls and David Kirk, from the 5th edition of *Computer Organization and Design*; and to a lesser extent on material in "Embracing and Extending 20th-Century Instruction Set Architectures," by Joe Gebis and David Patterson, *IEEE Computer*, April 2007 and "SIMD Considered Harmful," by David Patterson and Andrew Waterman, September 2017.

which they evolved. This environment has a system processor and system memory in addition to the GPU and its graphics memory. In fact, to recognize those distinctions, the GPU community refers to this type of architecture as *heterogeneous*. In addition to graphics, GPUs are also driving machine learning. We cover that role of GPUs in Chapter 7.

For problems with lots of data parallelism, all three SIMD variations share the advantage of being easier on the programmer than classic parallel MIMD programming.

The goal of this chapter is for architects to understand why vector is more general than multimedia SIMD, as well as the similarities and differences between vector and GPU architectures. Because vector architectures are supersets of the multimedia SIMD instructions, including a better model for compilation, and because GPUs share several similarities with vector architectures, we start with vector architectures to set the foundation for the following two sections. The next section introduces vector architectures, and Appendix G goes much deeper into the subject.

4.2 Vector Architecture

The most efficient way to execute a vectorizable application is a vector processor.

Jim Smith,
International Symposium on Computer Architecture (1994)

Vector architectures grab sets of data elements scattered about memory, place them into large sequential register files, operate on data in those register files, and then disperse the results back into memory. A single instruction works on vectors of data, which results in dozens of register-register operations on independent data elements.

These large register files act as compiler-controlled buffers, both to hide memory latency and to leverage memory bandwidth. Because vector loads and stores are deeply pipelined, the program pays the long memory latency only once per vector load or store versus once per element, thus amortizing the latency over, say, 32 elements. Indeed, vector programs strive to keep the memory busy.

The power limits of modern microprocessors lead architects to value architectures that can deliver good performance without the energy and design complexity costs of highly out-of-order superscalar processors. Vector instructions are a natural match to this trend because architects can use them to increase performance of simple in-order scalar processors without greatly raising energy demands and design complexity. In practice, developers can express many of the programs that run well on complex out-of-order designs more efficiently as DLP in the form of vector instructions, as Kozyrakis and Patterson (2002) showed.

RV64V Extension

We begin with a vector processor consisting of the primary components that Figure 4.1 shows. It is loosely based on the 50-year-old Cray-1, which was one of the first supercomputers. We use the RISC-V vector instruction set extension RVV. (The vector extension by itself is called RVV, so RV64V refers to the 64-bit RISC-V base instructions plus the vector extension.) We show a simplified subset of RV64V, trying to capture its essence in a few pages.

The primary components of the instruction set architecture of RV64V are the following:

- *Vector registers*—Each vector register holds a single vector, and RV64V has 32 of them. The vector register file needs to provide enough ports to feed all

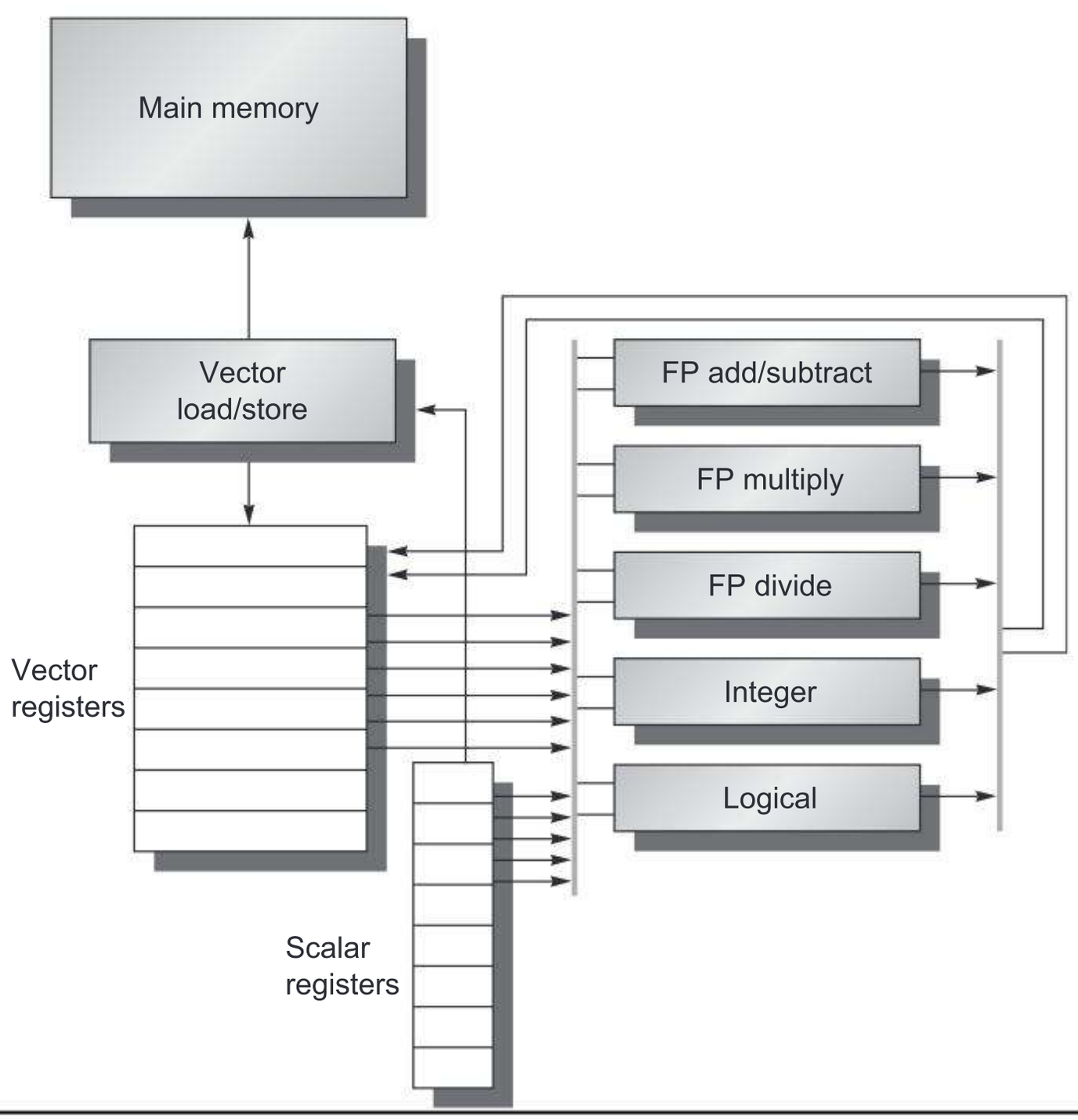

Figure 4.1 The basic structure of a vector architecture, RV64V, which includes a RISC-V scalar architecture. The vector unit has 32 vector registers, and all functional units are vector functional units. The vector and scalar registers have a significant number of read and write ports to allow multiple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

the vector functional units. These ports will allow a high degree of overlap among vector operations to different vector registers. The read and write ports, which total at least 16 read ports and 8 write ports, are connected to the functional unit inputs or outputs by a pair of crossbar switches. One way to increase the register file bandwidth is to compose it from multiple banks, which work well with relatively long vectors.

- *Vector functional units*—Each unit is fully pipelined in our implementation, and it can start a new operation on every clock cycle. A control unit is needed to detect hazards, both structural hazards for functional units and data hazards on register accesses. Figure 4.1 shows that we assume an implementation of RV64V has five functional units. For simplicity, we focus on the floating-point functional units in this section.
- *Vector load/store unit*—The vector memory unit loads or stores a vector to or from memory. The vector loads and stores are fully pipelined in our hypothetical RV64V implementation so that words can be moved between the vector registers and memory with a bandwidth of one word per clock cycle, after an initial latency. This unit would also normally handle scalar loads and stores.
- *A set of scalar registers*—Scalar registers can likewise provide data as input to the vector functional units and compute addresses to pass to the vector load/store unit. These are the normal 31 general-purpose registers and 32 floating-point registers of RV64G. One input of the vector functional units latches scalar values as they are read out of the scalar register file.
- In contrast to SIMD instructions in the next section, the total number of elements in a data-parallel operation is not determined by the instruction opcode. This value is called the *maximum vector length* (*VLMAX*), which is associated with each implementation of the vector architecture rather than with each instruction. Two implementations can have the same vector instruction set but different values for VLMAX.

Figure 4.2 lists the RV64V vector instructions we use in this section. The description in Figure 4.2 assumes that the input operands are all vector registers, but there are also versions of these instructions where an operand can be a scalar register (`xi` or `fi`). There are several types of each vector instruction depending on whether the source operands are all vector registers (`.vv` suffix), a vector register and a scalar integer register (`.vx` suffix), a vector register and a scalar integer immediate (`.vi` suffix), or a vector register and a scalar floating-point register (`.vf` suffix).

Because the operands determine the version of the instruction, we usually let the assembler supply the appropriate suffix. The vector functional unit gets a copy of the scalar value at instruction issue time.

Mnemonic	*Name*	*Description*
`vadd`	ADD	Integer add elements of V[rs1] and V[rs2], then put each result in V[rd]
`vsub`	SUBtract	Integer subtract elements of V[rs2] frpm V[rs1], then put each result in V[rd]
`vmul`	MULtiply	Integer multiply elements of V[rs1] and V[rs2], then put each result in V[rd]
`vdiv`	DIVide	Integer divide elements of V[rs1] by V[rs2], then put each result in V[rd]
`vmacc`	Multiply ACCumulate	Integer multiply elements of V[rs1] and V[rs2], then add each result to V[rd]
`vfadd`	Fp ADD	Fl. pt. add elements of V[rs1] and V[rs2], then put each result in V[rd]
`vfsub`	Fp SUBtract	Fl. pt. subtract elements of V[rs2] frpm V[rs1], then put each result in V[rd]
`vfmul`	Fp MULtiply	Fl. pt. multiply elements of V[rs1] and V[rs2], then put each result in V[rd]
`vfdiv`	Fp DIVide	Fl. pt. divide elements of V[rs1] by V[rs2], then put each result in V[rd]
`vfmacc`	Fp Multiply ACCumulate	Fl. pt. multiply elements of V[rs1] and V[rs2], then add each result to 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]
`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]
`vpeq`	Compare =	Compare elements of V[rs1] and V[rs2]. When equal, put a 1 in the corresponding 1-bit element of V[0]; 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 V[0] otherwise put 0
`vmxor`	Mask XOR	Exclusive OR LSB 1-bit elements of V[rs1] and V[rs2], then put each result in V[0]
`vmor`	Mask OR	Inclusive OR LSB 1-bit elements of V[rs1] and V[rs2], then put each result in V[0]
`vpmand`	Mask AND	Logical AND LSB 1-bit elements of V[rs1] and V[rs2], then put each result in V[0]
vle***i***	Contiguous Load	Load vector to register V[rd] from memory starting at address R[rs1]. (***i***-bit elements, ***i*** = 8, 16, 32, or 64. `vl` sets number of elements.)
vlse***i***	Strided Load	Load V[rd] from address at R[rs1] with stride in R[rs2] . i.e., R[rs1] + j x R[rs2], j = 0, 1, ...)
vloxe***i***	Indexed Load (Gather)	Load V[rs1] with vector whose elements are at R[rs2] + V[rs2] (i.e., V[rs2] is an index)
vse***i***	Contiguous Store	Store vector register V[rd] into memory starting at address R[rs1]
vsse***i***	Strided Store	Store V[rd] into memory at address R[rs1] with stride in R[rs2] (i.e., R[rs1] + j x R[rs2], j = 0, 1, ..)
vsoxe***i***	Indexed Store (Scatter)	Store V[rs1] into memory vector whose elements are at R[rs2] + V[rs2] (i.e., V[rs2] is an index)
`setvl`	Set vector length	Set element width and the number of registers (in `vtype`) and set `vl` and the destination register to the smaller of `mvl` and the source register

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.

Although the traditional vector architectures didn't support narrow data types efficiently, vectors naturally accommodate varying data sizes (Kozyrakis and Patterson, 2002). Thus, if a vector register has 32 64-bit elements, then 128 × 16-bit elements and even 256 × 8-bit elements are equally valid views. Such hardware multiplicity is why a vector architecture can be useful for multimedia applications and for scientific applications.

Note that the RV64V instructions in Figure 4.2 omit the size! An innovation of RV64V is to associate the data size *with each vector register* rather than the normal approach of the instruction supplying that information. Thus, before executing the vector instructions, a program configures the vector registers being used to specify their widths. Figure 4.3 lists the RV64V options. It is specified by setting a status

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.

register with the standard element width field SEW to the number of bits, for example, 64.

One reason for *dynamic register typing* is that many instructions are required for a conventional vector architecture that supports such variety. Given the number of sizes in Figure 4.3, if not for dynamic register typing, Figure 4.2 would be much longer! Also, since the type doesn't change, it's more efficient to specify it once rather than in every instruction.

Dynamic typing also lets programmers optimize their use of the vector register file. A program that doesn't need many vector registers can group registers together to make longer vectors. For example, suppose we have 1024 bytes of vector memory. A program that needs only four vector registers of 64-bit floats can use this storage as four 256-byte, or 256/8 = 32-element, vectors. It is specified by setting a status register with the length multiplier field LMUL to 8, which means 8 registers are grouped together, so we have only 4 active vector registers.

One complaint about vector architectures is that their larger state means slower context switch time. Our implementation of RV64V increases state a factor of 3: from $2 \times 32 \times 8 = 512$ bytes to $2 \times 32 \times 8 + 1024 = 1536$ bytes. To reduce the cost of switching contexts, the processor tracks whether software has written the vector registers since the last context switch.

A third benefit of dynamic register typing is that conversions between different-size operands can be implicit as part of arithmetic instructions depending on the configuration of the registers rather than as additional explicit conversion instructions. We'll see an example of this benefit in the next section.

The names `vle` and `vse` denote vector load and vector store, and they load or store an entire vector of data. One operand is the vector register to be loaded or stored. The other operand, which is an RV64G general-purpose register, is the starting address of the vector in memory. Vectors need more registers beyond the vector registers themselves. The vector-length register `vl` is used when the natural vector length is not equal to VLMAX, the vector-type register `vtype` records register types, and the predicate registers p_i are used when loops involve IF statements. We'll see them in action in the following example.

With a vector instruction, the system can perform the operations on the vector data elements in many ways, including operating on many elements simultaneously. This flexibility lets vector designs use slow but wide execution units to achieve high performance at low power. Furthermore, the independence of elements within a vector instruction set allows scaling of functional units without performing additional costly dependency checks, as superscalar processors require.

How Vector Processors Work: An Example

We can best understand a vector processor by looking at a vector loop for RV64V. Let's take a typical vector problem, which we use throughout this section:

$$Y = a \times X + Y$$

`X` and `Y` are vectors, initially resident in memory, and `a` is a scalar. This problem is the *SAXPY* or *DAXPY* loop that forms the inner loop of the Linpack benchmark (Dongarra et al., 2003). (SAXPY stands for single-precision a × X plus Y, and DAXPY for double-precision a × X plus Y.) Linpack is a collection of linear algebra routines, and the Linpack benchmark consists of routines for performing Gaussian elimination.

For now, let us assume that the number of elements, or *length*, of a vector register (32) matches the length of the vector operation we are interested in. (This restriction will be lifted shortly.)

Example Show the code for RV64G and RV64V for the DAXPY loop. For this example, assume that X and Y have 32 elements and the starting addresses of `X` and `Y` are in `x5` and `x6`, respectively. (A subsequent example covers when they do not have 32 elements.)

Answer Here is the RISC-V code:

```
        fld    f0,a              # Load scalar a
        addi   x28,x5,#256       # Last address to load
Loop:   fld    f1,0(x5)          # Load X[i]
        fmul.d f1,f1,f0          # a × X[i]
        fld    f2,0(x6)          # Load Y[i]
        fadd.d f2,f2,f1          # a × X[i] + Y[i]
        fsd    f2,0(x6)          # Store into Y[i]
        addi   x5,x5,#8          # Increment index to X
        addi   x6,x6,#8          # Increment in
```

Here is the RV64V code for DAXPY:

```
vsetvl a3,a0,e64,m8,ta,ma # 4 64-bit vregs (SEW=64,LMUL=8)
fld    f0,a               # Load scalar a
vle64  v0,(x5)            # Load vector X
vfmul  v8,v0,f0           # Vector-scalar multiply
vle64  v16,(x6)           # Load vector Y
vfadd  v24,v8,v16         # Vector-vector add
vse64  v24,(x6)           # Store the sum
```

Note that the assembler determines which version of the vector operations to generate. Because the multiply has a scalar operand, it generates `vfmul.vf`, whereas the add doesn't have a scalar operand, so it generates `vfadd.vv`. The initial instruction configures the first four vector registers to hold 64-bit data.

The most dramatic difference between the preceding scalar and vector code is that the vector processor greatly reduces the dynamic instruction bandwidth, executing only 8 instructions versus 258 for RV64G. This reduction occurs because the vector operations work on 32 elements and the overhead instructions that constitute nearly half the loop on RV64G are not present in the RV64V code. When the compiler produces vector instructions for such a sequence, and the resulting code spends much of its time running in vector mode, the code is said to be *vectorized* or *vectorizable*. Loops can be vectorized when they do not have dependences between iterations of a loop, which are called *loop-carried dependences* (see Section 4.5).

Another important difference between RV64G and RV64V is the frequency of pipeline interlocks for a simple implementation of RV64G. In the straightforward scalar RV64G code, every `fadd.d` must wait for a `fmul.d`, and every `fsd` must wait for the `fadd.d`. On the vector processor, each vector instruction will stall only for the first element in each vector, and then subsequent elements will flow smoothly down the pipeline. Thus pipeline stalls are required only once per vector *instruction*, rather than once per vector *element*. Vector architects call forwarding of element-dependent operations *chaining*, in that the dependent operations are "chained" together. In this example the pipeline stall frequency on RV64G will be about 32× higher than it is on RV64V. Software pipelining, loop unrolling (Appendix H), or out-of-order execution can reduce the pipeline stalls on RV64G. However, the large difference in instruction bandwidth cannot be reduced substantially.

Vector Execution Time

The execution time of a sequence of vector operations primarily depends on three factors: (1) the length of the operand vectors, (2) structural hazards among the operations, and (3) the data dependences. Given the vector length and the *initiation rate*, which is the rate at which a vector unit consumes new operands and produces new results, we can compute the time for a single vector instruction.

All modern vector computers have vector functional units with multiple parallel pipelines (or *lanes*) that can produce two or more results per clock cycle, but they may also have some functional units that are not fully pipelined. For simplicity, our RV64V implementation has one lane with an initiation rate of one element per clock cycle for individual operations. Thus the execution time in clock cycles for a single vector instruction is approximately the vector length.

To simplify the discussion of vector execution and vector performance, we use the notion of a *convoy*, which is the set of vector instructions that could potentially execute together. The instructions in a convoy *must not* contain any structural hazards. If such hazards were present, the instructions would need to be serialized and initiated in different convoys. Thus, the `vld` and the following `vmul` in the preceding example can be in the same convoy. As we will soon see, you can estimate performance of a section of code by counting the number of convoys. To keep this analysis simple, we assume that a convoy of instructions must complete execution before any other instructions (scalar or vector) can begin execution.

It might seem that in addition to vector instruction sequences with structural hazards, sequences with read-after-write dependency hazards should also be in separate convoys. However, chaining allows them to be in the same convoy since it allows a vector operation to start as soon as the individual elements of its vector source operand become available: the results from the first functional unit in the chain are "forwarded" to the second functional unit. In practice, we often implement chaining by allowing the processor to read and write a particular vector register at the same time, albeit to different elements. Early implementations of chaining worked just like forwarding in scalar pipelining, but this approach restricted the timing of the source and destination instructions in the chain. Recent implementations use *flexible chaining*, which allows a vector instruction to chain to essentially any other active vector instruction if we don't generate a structural hazard. All modern vector architectures support flexible chaining, which we assume throughout this chapter.

To turn convoys into execution time, we need a metric to estimate the timing of a convoy. It is called a *chime*, which is simply the unit of time taken to execute one convoy. Thus a vector sequence that consists of m convoys executes in m chimes. For example, given a vector length of n, for our simple RV64V implementation, the execution time is approximately $m \times n$ clock cycles.

The chime approximation ignores some processor-specific overheads, many of which are dependent on vector length. Therefore measuring time in chimes is a better approximation for long vectors than for short ones. We will use the chime measurement, rather than clock cycles per result, to indicate explicitly that we are ignoring certain overheads.

If we know the number of convoys in a vector sequence, we know the execution time in chimes. One source of overhead ignored in measuring chimes is any limitation on initiating multiple vector instructions in a single clock cycle. If only one vector instruction can be initiated in a clock cycle (the reality in most vector processors), the chime count will underestimate the actual execution time of a convoy. Because the length of vectors is typically much greater than the number of instructions in the convoy, we will simply assume that the convoy executes in one chime.

Example Show how the following code sequence lays out in convoys, assuming a single copy of each vector functional unit:

```
vle64 v0,x5       # Load vector X
vfmul v1,v0,f0    # Vector-scalar multiply
vle64 v2,x6       # Load vector Y
vfadd v3,v1,v2    # Vector-vector add
vse64 v3,x6       # Store the sum
```

How many chimes will this vector sequence take? How many cycles per FLOP (floating-point operation) are needed, ignoring vector instruction issue overhead?

Answer The first convoy starts with the first `vle` instruction. The `vfmul` is dependent on the first `vle`, but chaining allows it to be in the same convoy.

The second `vle` instruction must be in a separate convoy because there is a structural hazard on the load/store unit for the prior `vle` instruction. The `vfadd` is dependent on the second `vld`, but it can again be in the same convoy via chaining. Finally, the `vse` has a structural hazard on the `vld` in the second convoy, so it must go in the third convoy. This analysis leads to the following layout of vector instructions into convoys:

1. `vle` `vfmul`
2. `vle` `vfadd`
3. `vse`

The sequence requires three convoys. Because the sequence takes three chimes and there are two floating-point operations per result, the number of cycles per FLOP is 1.5 (ignoring any vector instruction issue overhead). Note that although we allow the `vld` and `vmul` both to execute in the first convoy, most vector machines will take 2 clock cycles to initiate the instructions.

This example shows that the chime approximation is reasonably accurate for long vectors. For example, for 32-element vectors, the time in chimes is 3, so the sequence would take about 32×3 or 96 clock cycles. The overhead of issuing convoys in two separate clock cycles would be small.

Another source of overhead is far more significant than the issue limitation. The most important source of overhead ignored by the chime model is vector *start-up time*, which is the latency in clock cycles until the pipeline is full. The start-up time is principally determined by the pipelining latency of the vector functional unit. For RV64V, we will use the same pipeline depths as the Cray-1, although latencies in more modern processors have tended to increase, especially for vector loads. All functional units are fully pipelined. The pipeline depths are 6

clock cycles for floating-point add, 7 for floating-point multiply, 20 for floating-point divide, and 12 for vector load.

Given these vector basics, the next several subsections will give optimizations that either improve the performance or increase the types of programs that can run well on vector architectures. In particular, they will answer these questions:

- How can a vector processor execute a single vector faster than one element per clock cycle? Multiple elements per clock cycle improve performance.
- How does a vector processor handle programs where the vector lengths are not the same as the maximum vector length (mvl)? Because most application vectors don't match the architecture vector length, we need an efficient solution to this common case.
- What happens when there is an IF statement inside the code to be vectorized? More code can vectorize if we can efficiently handle conditional statements.
- What does a vector processor need from the memory system? Without sufficient memory bandwidth, vector execution can be futile.
- How does a vector processor handle multiple-dimensional matrices? This popular data structure must vectorize for vector architectures to be widely applicable.
- How does a vector processor handle sparse matrices? This popular data structure must vectorize also.
- How do you program a vector computer? Architectural innovations that are a mismatch to programming languages and their compilers may not get widespread use.

The rest of this section introduces each of these optimizations of the vector architecture, and Appendix G goes into greater depth.

Multiple Lanes: Beyond One Element per Clock Cycle

A critical advantage of a vector instruction set is that it allows software to pass a large amount of parallel work to hardware using only a single short instruction. One vector instruction can include scores of independent operations yet be encoded in the same number of bits as a conventional scalar instruction. The parallel semantics of a vector instruction allow an implementation to execute these elemental operations using a deeply pipelined functional unit, as in the RV64V implementation we've studied so far; an array of parallel functional units; or a combination of parallel and pipelined functional units. Figure 4.4 illustrates

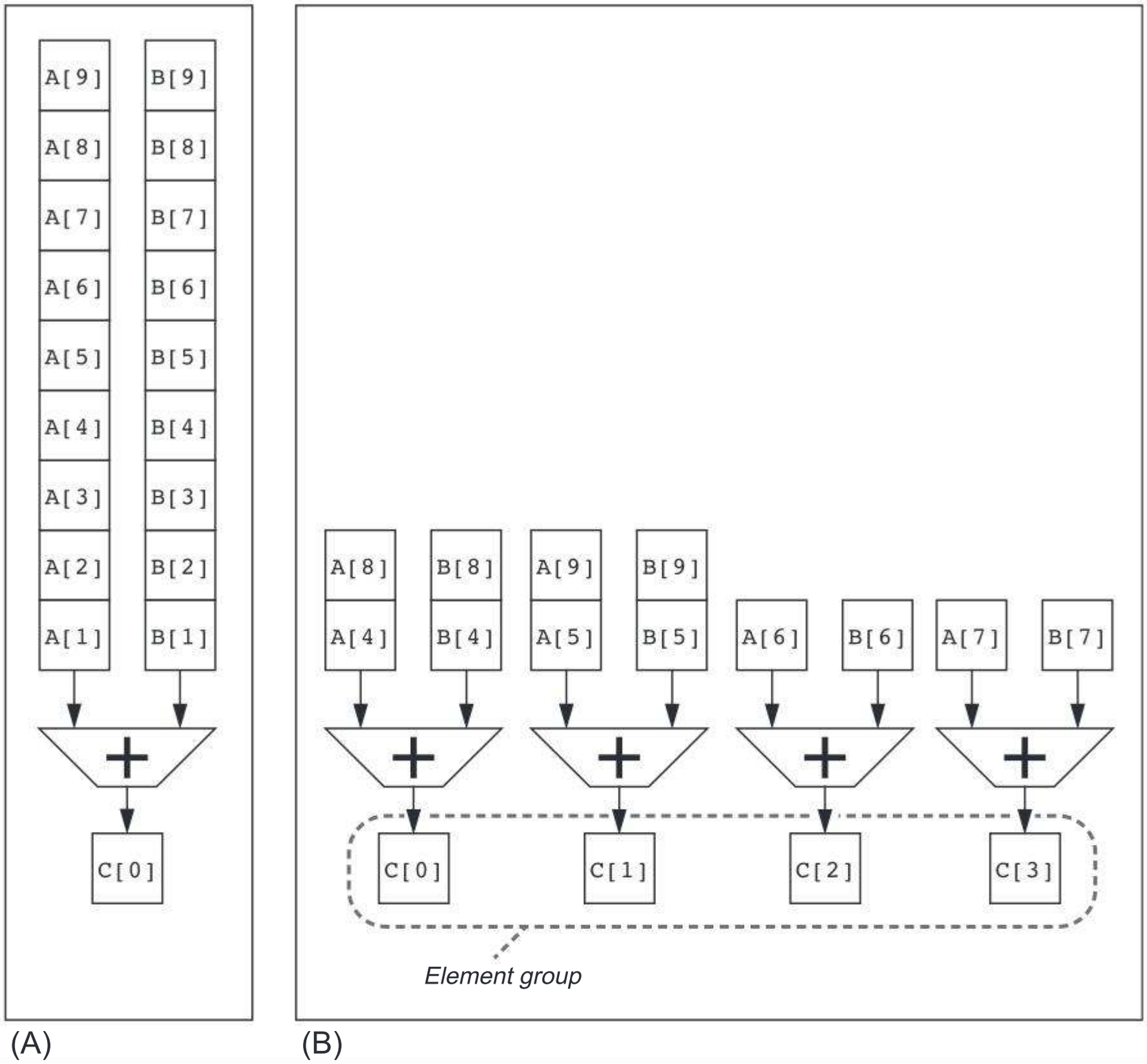

Figure 4.4 Using multiple functional units to improve the performance of a single vector add instruction, C = A + B. The vector processor (A) on the left has a single add pipeline and can complete one addition per clock cycle. The vector processor (B) on the right has four add pipelines and can complete 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.

how to improve vector performance by using parallel pipelines to execute a vector add instruction.

The RV64V instruction set has the property that all vector arithmetic instructions only allow element *N* of one vector register to take part in operations with element *N* from other vector registers. This dramatically simplifies the design of a highly parallel vector unit, which can be structured as multiple parallel *lanes*.

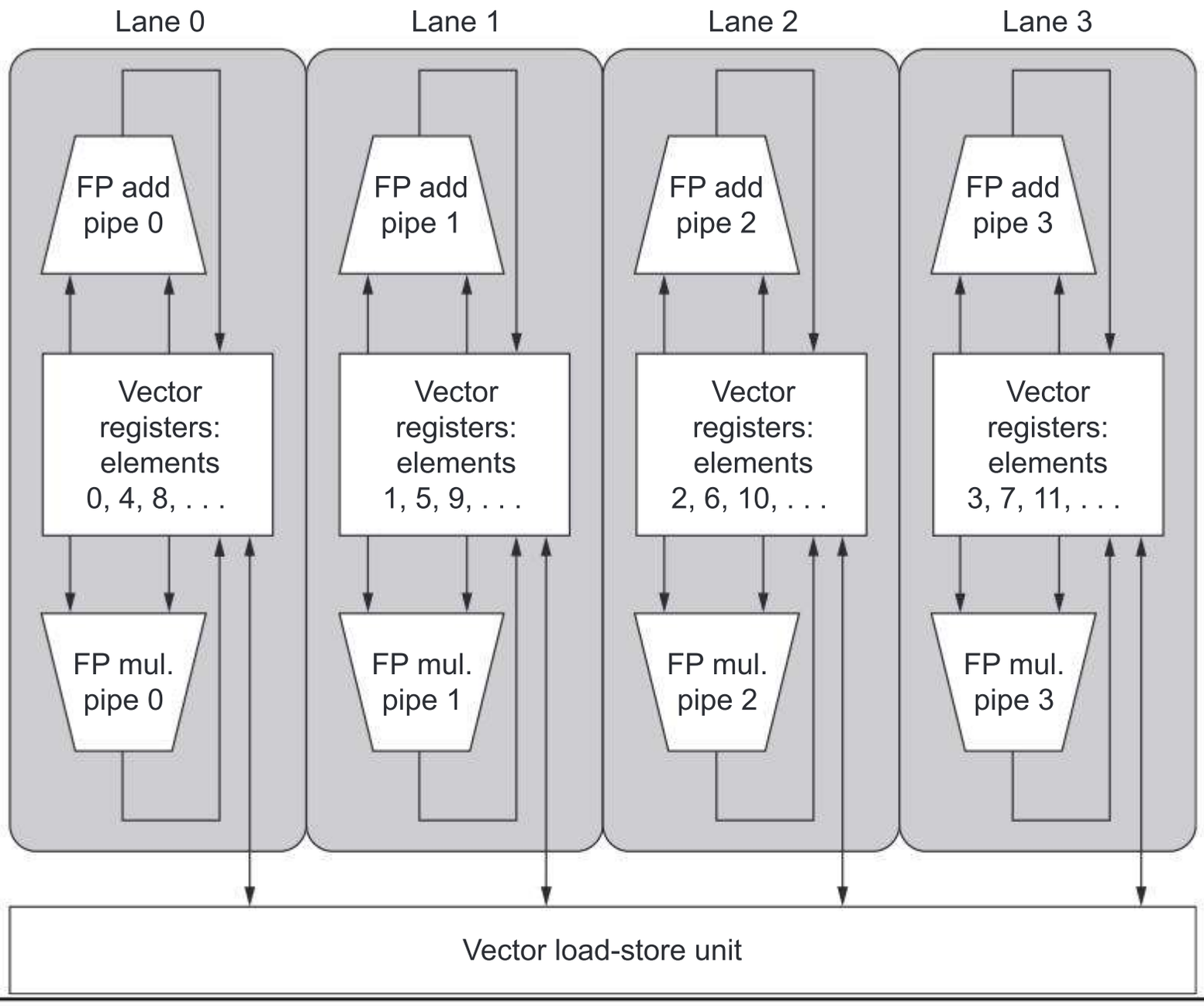

Figure 4.5 Structure of a vector unit containing four lanes. The vector register memory is divided across the lanes, with each lane holding every fourth element of each vector register. The figure shows three vector functional units: an FP add, an FP multiply, and a load-store unit. Each of the vector 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.

As with a traffic highway, we can increase the peak throughput of a vector unit by adding more lanes. Figure 4.5 shows the structure of a four-lane vector unit. Thus going to four lanes from one lane reduces the number of clocks for a chime from 32 to 8 in this example. For multiple lanes to be advantageous, both the applications and the architecture must support long vectors; otherwise, if vector instructions are short they will execute so quickly that you'll run out of instruction bandwidth, requiring ILP techniques (see Chapter 3) to supply enough vector instructions.

Each lane contains one portion of the vector register file and one execution pipeline from each vector functional unit. Each vector functional unit executes vector instructions at the rate of one element group per cycle using multiple pipelines, one per lane. The first lane holds the first element (element 0) for all vector registers, and so the first element in any vector instruction will have its source and destination operands located in the first lane. This allocation allows the arithmetic

pipeline local to the lane to complete the operation without communicating with other lanes. Avoiding interlane communication reduces the wiring cost and register file ports required to build a highly parallel execution unit. It also helps explain why vector computers can complete up to 64 operations per clock cycle (2 arithmetic units and 2 load/store units across 16 lanes).

Adding multiple lanes is a popular technique to improve vector performance as it requires little increase in control complexity and does not require changes to existing machine code. It also allows designers to trade off die area, clock rate, voltage, and energy without sacrificing peak performance. If the clock rate of a vector processor is halved, doubling the number of lanes will retain the same peak performance.

Vector-Length Registers: Handling Loops Not Equal to 32

A vector register processor has a natural vector length determined by the maximum vector length (mvl). This length, which was 32 in our example above, is unlikely to match the real vector length in a program. Moreover, in a real program, the length of a particular vector operation is often *unknown* at compile time. In fact, a single piece of code may require different vector lengths. For example, consider this code:

```
for (i=0; i < n; i=i+1)
 Y[i] = a * X[i] + Y[i];
```

The size of all the vector operations depends on n, which may not even be known until run time. The value of n might also be a parameter to a procedure containing the preceding loop and therefore subject to change during execution.

The solution to these problems is to add a *vector-length register* (`vl`). The `vl` controls the length of any vector operation, including a vector load or store. The value in the `vl`, however, cannot be greater than the maximum vector length (mvl, kept in the `VLMAX` register in RISC-V). This solves our problem as long as the real length is less than or equal to the mvl. This parameter means the length of vector registers can grow in later computer generations without changing the instruction set. As we will see in the next section, multimedia SIMD extensions have no equivalent of mvl, so they expand the instruction set every time they increase their vector length.

What if the value of `n` is not known at compile time and thus may be greater than the mvl? To tackle the second problem where the vector is longer than the maximum length, a technique called *strip mining* is traditionally used (see Figure 4.6). Strip mining is the generation of code such that each vector operation is done for a size less than or equal to the mvl. One loop handles any number of iterations that is a multiple of the mvl and fixup code handles any remaining iterations and must be less than the mvl. RISC-V has a better solution than a separate loop for strip mining. The instruction `setvl` writes the smaller of the `VLMAX` and

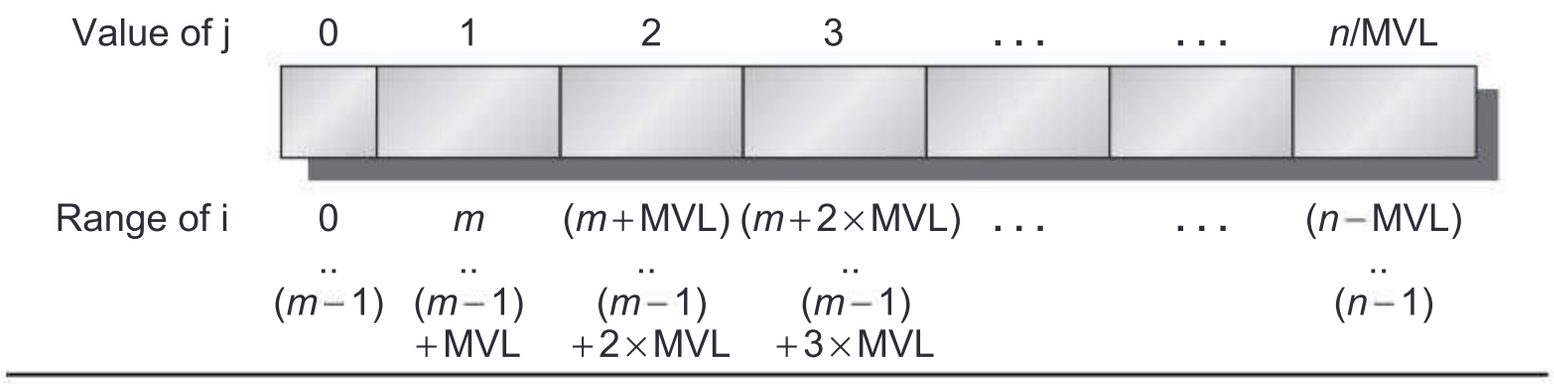

Figure 4.6 A vector of arbitrary length processed with strip mining. All blocks but the first are of length VLMAX, utilizing the full power of the vector processor. In this figure we use the variable *m* for the expression (n % VLMAX). (The C operator % is modulo.)

the loop variable n into vl (and to another register). If the number of iterations of the loop is larger than n, then the fastest the loop can compute is VLMAX values at time, so setvl sets vl to VLMAX. If n is smaller than VLMAX, it should compute only on the last n elements in this final iteration of the loop, so setvl sets VLMAX to n. setvl also writes another scalar register to help with later loop bookkeeping. Below is the RV64V code for vector DAXPY for any value of n.

```
        vsetvl a3,a0,e64,m16,ta,ma  # 2 64-bit vregs (SEW=64,LMUL=16)
        fld    f0,a                 # Load scalar a
loop:   vsetvl t0,a0                # vl = t0 = min(mvl,n)
        vle64  v0,x5                # Load vector X
        slli   t1,t0,3              # t1 = vl * 8 (in bytes)
        add    x5,x5,t1             # Increment pointer to X by vl*8
        vfmul  v0,v0,f0             # Vector-scalar mult
        vle64  v1,x6                # Load vector Y
        vfadd  v1,v0,v1             # Vector-vector add
        sub    a0,a0,t0             # n -= vl (t0)
        vse64  v1,x6                # Store the sum into Y
        add    x6,x6,t1             # Increment pointer to Y by vl*8
        bnez   a0,loop              # Repeat if n != 0
```

Mask Registers: Handling IF Statements in Vector Loops

From Amdahl's law, we know that the speedup on programs with low to moderate levels of vectorization will be very limited. The presence of conditionals (IF statements) inside loops and the use of sparse matrices are two main reasons for lower levels of vectorization. Programs that contain IF statements in loops cannot be run in vector mode using the techniques we have discussed so far because the IF statements introduce control dependences into a loop. Likewise, we cannot implement sparse matrices efficiently using any of the capabilities we have seen so far. We examine strategies for dealing with conditional execution here, leaving the discussion of sparse matrices for later.

Consider the following loop written in C:

```
for (i = 0; i < 64; i=i+1)
  if (X[i] != 0)
    X[i] = X[i] - Y[i];
```

This loop cannot normally be vectorized because of the conditional execution of the body. However, if the inner loop could be run for the iterations for which X [i] ≠ 0, then the subtraction could be vectorized.

The common extension for this capability is *vector-mask control*. In RV64V predicate registers hold the mask and essentially provide conditional execution of each element operation in a vector instruction. These registers use a Boolean vector to control the execution of a vector instruction, just as conditionally executed instructions use a Boolean condition to determine whether to execute a scalar instruction (see Chapter 3). When the mask register is set, all following vector instructions operate only on the vector elements whose corresponding entries in the predicate register are 1. The entries in the destination vector register that correspond to a 0 in the mask register are unaffected by the vector operation. Rather than have separate mask registers, RVV uses vector register v0. When an instruction is masked, the least-significant bit (bit 63) of each element in v0 indicates whether that element operation is active.

We can now use the following code for the previous loop, assuming that the starting addresses of `X` and `Y` are in `x5` and `x6`, respectively:

```
vsetvl   a3,a0,e64,m8,ta,ma # 4 64-bit vreg(SEW=64,LMUL=8)
vle64    v8,(x5)            # Load vector X into v8
vle64    v16,(x6)           # Load vector Y into v16
fmv.d.x  f0,x0              # Put (FP) zero into f0
vfne     v0,v8,f0           # Set v0<63>(i) to 1 if v8(i)!=f0
vfsub    v8,v8,v16, v0.t    # Subtract under vector v0 mask
vse64    v8,x5              # Store the result in X
```

Compiler writers use the term *IF-conversion* to transform an IF statement into a straight-line code sequence using conditional execution.

Using a vector-mask register does have overhead, however. With scalar architectures, conditionally executed instructions still require execution time when the condition is not satisfied. Nonetheless, the elimination of a branch and the associated control dependences can make a conditional instruction faster even if it sometimes does useless work. Similarly, vector instructions executed with a vector mask still take the same execution time, even for the elements where the mask is zero. Despite a significant number of zeros in the mask, using vector-mask control may still be significantly faster than using scalar mode.

As we will see in Section 4.4, one difference between vector processors and GPUs is how they handle conditional statements. Vector processors make the predicate registers part of the architectural state and rely on compilers to manipulate mask registers explicitly. In contrast, GPUs get the same effect using hardware to manipulate internal mask registers that are invisible to GPU software. In both cases the hardware spends the time to execute a vector element whether the corresponding mask bit is 0 or 1, so the FLOPs/second rate drops when masks are used.

Memory Banks: Supplying Bandwidth for Vector Load/Store Units

The behavior of the load/store vector unit is significantly more complicated than that of the arithmetic functional units. The start-up time for a load is the time to get the first word from memory into a register. If the rest of the vector can be supplied without stalling, then the vector initiation rate is equal to the rate at which new words are fetched or stored. Unlike simpler functional units, the initiation rate may not necessarily be 1 clock cycle because memory bank stalls can reduce effective throughput.

Typically, penalties for start-ups on load/store units are higher than those for arithmetic units—over 100 clock cycles on many processors. For RV64V, we assume a start-up time of 12 clock cycles, the same as the Cray-1. (Recent vector computers use caches to bring down latency of vector loads and stores.)

To maintain an initiation rate of one word fetched or stored per clock cycle, the memory system must be capable of producing or accepting this much data. Spreading accesses across multiple independent memory banks usually delivers the desired rate. As we will soon see, having significant numbers of banks is useful for dealing with vector loads or stores that access rows or columns of data. Most vector processors use memory banks—which allow several independent accesses rather than simple memory interleaving—for three reasons:

1. Many vector computers support many loads or stores per clock cycle, and the memory bank cycle time is usually several times larger than the processor cycle time. To support simultaneous accesses from multiple loads or stores, the memory system needs multiple banks and needs to be able to control the addresses to the banks independently.
2. Most vector processors support the ability to load or store data words that are not sequential. In such cases independent bank addressing, rather than interleaving, is required.
3. Most vector computers support multiple processors sharing the same memory system, so each processor will be generating its own separate stream of addresses.

In combination, these features lead to the desire for a large number of independent memory banks, as the following example shows.

Example The largest configuration of a Cray T90 (Cray T932) has 32 processors, each capable of generating 4 loads and 2 stores per clock cycle. The processor clock cycle is 2.167 ns, while the cycle time of the SRAMs used for the memory system is 15 ns. Calculate the minimum number of memory banks required to allow all processors to run at the full memory bandwidth.

Answer The maximum number of memory references each cycle is 192: 32 processors times 6 references per processor. Each SRAM bank is busy for $15/2.167 = 6.92$ clock cycles, which we round up to 7 processor clock cycles. Therefore we require a minimum of $192 \times 7 = 1344$ memory banks!

The Cray T932 actually has 1024 memory banks, so the early models could not sustain the full bandwidth to all processors simultaneously. A subsequent memory upgrade replaced the 15 ns asynchronous SRAMs with pipelined synchronous SRAMs that more than halved the memory cycle time, thereby providing sufficient bandwidth.

Taking a higher-level perspective, vector load/store units play a similar role to prefetch units in scalar processors in that both try to deliver data bandwidth by supplying processors with streams of data.

Stride: Handling Multidimensional Arrays in Vector Architectures

The position in memory of adjacent elements in a vector may not be sequential. Consider this straightforward code for matrix multiplication in C:

```
for (i = 0; i < 100; i=i+1)
  for (j = 0; j < 100; j=j+1) {
    A[i][j] = 0.0;
    for (k = 0; k < 100; k=k+1)
      A[i][j] = A[i][j] + B[i][k] * D[k][j];
   }
```

We could vectorize the multiplication of each row of B with each column of D and strip-mine the inner loop with k as the index variable.

To do so, we must consider how to address adjacent elements in B and adjacent elements in D. When an array is allocated memory, it is linearized and must be laid out in either *row-major order* (as in C) or *column-major order* (as in Fortran). This linearization means that either the elements in the row or the elements in the column are not adjacent in memory. For example, the preceding C code allocates in row-major order, so the elements of D that are accessed by iterations in the inner loop are separated by the row size times 8 (the number of bytes per entry) for a total of 800 bytes. In Chapter 2 we saw that blocking could improve locality in cache-based systems. For vector processors without caches, we need another technique to fetch elements of a vector that are not adjacent in memory.

This distance separating elements to be gathered into a single vector register is called the *stride*. In this example matrix D has a stride of 100 double words (800 bytes), and matrix B would have a stride of 1 double word (8 bytes). For column-major order, which is used by Fortran, the strides would be reversed. Matrix D would have a stride of 1, or 1 double word (8 bytes), separating successive

elements, while matrix B would have a stride of 100, or 100 double words (800 bytes). Thus, without reordering the loops, the compiler can't hide the long distances between successive elements for both B and D.

Once a vector is loaded into a vector register, it acts as if it had logically adjacent elements. Thus a vector processor can handle strides greater than one, called *nonunit strides*, using only vector load and vector store operations with stride capability. This ability to access nonsequential memory locations and to reshape them into a dense structure is one of the major advantages of a vector architecture.

Caches inherently deal with unit-stride data. Increasing block size can help reduce miss rates for large scientific datasets with unit stride, but increasing block size can even have a negative effect on data that are accessed with nonunit strides. While blocking techniques can solve some of these problems (see Chapter 2), the ability to access noncontiguous data efficiently remains an advantage for vector processors on certain problems, as we will see in Section 4.7.

On RV64V, where the addressable unit is a byte, the stride for our example would be 800. The value must be computed dynamically because the size of the matrix may not be known at compile time or—just like vector length—may change for different executions of the same statement. The vector stride, like the vector starting address, can be put in a general-purpose register. Then the RV64V instruction `vlse` (vector load with stride) fetches the vector into a vector register. Likewise, when storing a nonunit stride vector, use the instruction `vsse` (vector store with stride).

Supporting strides greater than one complicates the memory system. Once we introduce nonunit strides, it becomes possible to request accesses from the same bank frequently. When multiple accesses contend for a bank, a memory bank conflict occurs, thereby stalling one access. A bank conflict and thus a stall will occur if

$$\frac{\text{Number of banks}}{\text{Least common multiple}(\text{Stride}, \text{Number of banks})} < \text{Bank busy time}$$

Example Suppose we have 8 memory banks with a bank busy time of 6 clocks and a total memory latency of 12 cycles. How long will it take to complete a 64-element vector load with a stride of 1? With a stride of 32?

Answer Because the number of banks is larger than the bank busy time, for a stride of 1, the load will take $12 + 64 = 76$ clock cycles, or 1.2 clock cycles per element. The worst possible stride is a value that is a multiple of the number of memory banks, as in this case with a stride of 32 and 8 memory banks. Every access to memory (after the first one) will collide with the previous access and will have to wait for the 6-clock-cycle bank busy time. The total time will be $12 + 1 + 6 * 63 = 391$ clock cycles, or 6.1 clock cycles per element, slowing it down by a factor of 5!

Gather-Scatter: Handling Sparse Matrices in Vector Architectures

As previously mentioned, sparse matrices are commonplace, so it is important to have techniques to allow programs with sparse matrices to execute in vector mode. In a sparse matrix the elements of a vector are usually stored in some compacted form and then accessed indirectly. Assuming a simplified sparse structure, we might see code that looks like this:

```
for (i = 0; i < n; i=i+1)
  A[K[i]] = A[K[i]] + C[M[i]];
```

This code implements a sparse vector sum on the arrays A and C, using index vectors K and M to designate the nonzero elements of A and C. (A and C must have the same number of nonzero elements—`n` of them—so `K` and `M` are the same size.)

The primary mechanism for supporting sparse matrices is *gather-scatter operations* using index vectors. The goal of such operations is to support moving between a compressed representation (i.e., zeros are not included) and normal representation (i.e., the zeros are included) of a sparse matrix. A *gather* operation takes an *index vector* and fetches the vector whose elements are at the addresses given by adding a base address to the offsets given in the index vector. The result is a dense vector in a vector register. After these elements are operated on in a dense form, a *scatter* store can store the sparse vector in expanded form using the same index vector. Hardware support for such operations is called *gather-scatter*, and it appears on nearly all modern vector processors. The RV64V instructions are `vloxe` (load vector indexed or gather) and `vsoxe` (store vector indexed or scatter). For example, if `x5`, `x6`, `x7`, and `x28` contain the starting addresses of the vectors in the previous sequence, we can code the inner loop with vector instructions such as:

```
vsetvl   a3,a0,e64,m8,ta,ma # 4 64-bit vregs (SEW=64,LMUL=8)
vle64    v0, x7             # Load K[]
vloxe64  v8,x5,(v0)         # Load A[K[]]
vle64    v16,x28            # Load M[]
vloxe64  v24, x6, (v2)      # Load C[M[]]
vadd     v8, v8, v24        # Add them
vsoxe64  v8, x5,(v0)        # Store A[K[]]
```

This technique allows code with sparse matrices to run in vector mode. A simple vectorizing compiler could not automatically vectorize the preceding source code because the compiler would not know that the elements of K are distinct values, and thus that no dependencies exist. Instead, a programmer directive is required to tell the compiler that it was safe to run the loop in vector mode.

Although indexed loads and stores (gather and scatter) can be pipelined, they typically run much more slowly than nonindexed loads or stores, because the

memory banks are not known from the start of the instruction. The register file must also provide communication between the lanes of a vector unit to support gather and scatter.

Each element of a gather or scatter has an individual address, so they can't be handled in groups, and there can be conflicts at many places throughout the memory system. Thus each individual access incurs significant latency even on cache-based systems. However, as Section 4.7 shows, a memory system can deliver better performance by designing for this case and by using more hardware resources versus when architects have a laissez-faire attitude toward such unpredictable accesses.

As we will see in Section 4.4, all loads are gathers and all stores are scatters in GPUs in that no separate instructions restrict addresses to be sequential. To turn the potentially slow gathers and scatters into the more efficient unit-stride accesses to memory, the GPU hardware tries to recognize the sequential addresses during execution.

Programming Vector Architectures

An advantage of vector architectures is that compilers can tell programmers at compile time whether a section of code will vectorize or not, often giving hints as to why it did not vectorize the code. This straightforward execution model allows experts in other domains to learn how to improve performance by revising their code or by giving hints to the compiler when it's okay to assume independence between operations, such as for gather-scatter data transfers. It is this dialogue between the compiler and the programmer, with each side giving hints to the other on how to improve performance, that simplifies programming of vector computers.

Today, the main factor that affects the success with which a program runs in vector mode is the structure of the program itself: Do the loops have true data dependencies (see Section 4.5), or can they be restructured so as not to have such dependencies? This factor is influenced by the algorithms chosen and, to some extent, by how they are coded.

As an indication of the level of vectorization achievable in scientific programs, let's look at the vectorization levels observed for the Perfect Club benchmarks. Figure 4.7 shows the percentage of operations executed in vector mode for two versions of the code running on the Cray Y-MP. The first version is obtained with just compiler optimization on the original code, while the second version uses extensive hints from a team of Cray Research programmers. Several studies of the performance of applications on vector processors show a wide variation in the level of compiler vectorization.

The hint-rich versions show significant gains in vectorization level for codes that the compiler could not vectorize well by itself, with all codes now above 50% vectorization. The median vectorization improved from about 70% to about 90%.

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.

4.3 SIMD Instruction Set Extensions for Multimedia

SIMD Multimedia Extensions started with the simple observation that many media applications operate on narrower data types than the 32-bit processors were optimized for. Graphics systems would use 8 bits to represent each of the three primary colors plus 8 bits for transparency. Depending on the application, audio samples are usually represented with 8 or 16 bits. By partitioning the carry chains within, say, a 256-bit adder, a processor could perform simultaneous operations on short vectors of thirty-two 8-bit operands, sixteen 16-bit operands, eight 32-bit operands, or four 64-bit operands. The additional cost of such partitioned adders was small. Figure 4.8 summarizes typical multimedia SIMD instructions. Like vector instructions, a SIMD instruction specifies the same operation on vectors of data. Unlike vector machines with large register files such as the RISC-V RV64V vector registers, which can hold, say, thirty-two 64-bit elements in each of 32 vector registers, SIMD instructions tend to specify fewer operands and thus use much smaller register files.

In contrast to vector architectures, which offer an elegant instruction set that is intended to be the target of a vectorizing compiler, SIMD extensions have three major omissions: no vector length register, no strided or gather/scatter data transfer instructions, and no mask registers.

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.

1. Multimedia SIMD extensions fix the number of data operands in the opcode, which has led to the addition of hundreds of instructions in the MMX, SSE, AVX, and AMX extensions of the x86 architecture. Vector architectures have a vector-length register that specifies the number of operands for the current operation. These variable-length vector registers easily accommodate programs that naturally have shorter vectors than the maximum size the architecture supports. Moreover, vector architectures have an implicit maximum vector length in the architecture, which combined with the vector length register avoids the use of many opcodes.
2. Up until recently, multimedia SIMD did not offer the more sophisticated addressing modes of vector architectures, namely strided accesses and gather-scatter accesses. These features increase the number of programs that a vector compiler can successfully vectorize (see Section 4.7).
3. Although this is changing, multimedia SIMD usually did not offer the mask registers to support conditional execution of elements as in vector architectures.

Such omissions make it harder for the compiler to generate SIMD code and increase the difficulty of programming in SIMD assembly language.

For the x86 architecture, the MMX instructions added in 1996 repurposed the 64-bit floating-point registers, so the basic instructions could perform eight 8-bit operations or four 16-bit operations simultaneously. These were joined by parallel MAX and MIN operations, a wide variety of masking and conditional instructions, operations typically found in digital signal processors, and ad hoc instructions that were believed to be useful in important media libraries. Note that MMX reused the floating-point data-transfer instructions to access memory.

The Streaming SIMD Extensions (SSE) successor in 1999 added 16 separate registers (XMM registers) that were 128 bits wide, so now instructions could simultaneously perform sixteen 8-bit operations, eight 16-bit operations, or four 32-bit

operations. It also performed parallel single-precision floating-point arithmetic. Because SSE had separate registers, it needed separate data transfer instructions. Intel soon added double-precision SIMD floating-point data types via SSE2 in 2001, SSE3 in 2004, and SSE4 in 2007. Instructions with four single-precision floating-point operations or two parallel double-precision operations increased the peak floating-point performance of the x86 computers, as long as programmers placed the operands side by side. With each generation, they also added ad hoc instructions whose aim was to accelerate specific multimedia functions perceived to be important.

The Advanced Vector Extensions (AVX), added in 2010, doubled the width of the registers again to 256 bits (YMM registers) and thereby offered instructions that doubled the number of operations on all narrower data types. Figure 4.9 shows AVX instructions useful for double-precision floating-point computations. AVX2 in 2013 added 30 new instructions such as gather (VGATHER) and vector shifts (VPSLL, VPSRL, VPSRA). AVX-512 in 2017 doubled the width again to 512 bits (ZMM registers), doubled the number of the registers again to 32, and added about 250 new instructions including scatter (VPSCATTER) and mask registers (OPMASK). AVX includes preparations to extend registers to 1024 bits in future editions of the architecture.

The latest extension is the Advanced Matrix Extension (AMX), added to microprocessors that appeared in 2022. The goal is to support the matrix multiply operations that are fundamental to machine learning (see Chapter 7). It adds 8 two-dimensional vector registers, called *tiles*. Each tile is 16 × 256: 16 rows, each 256 bits wide. The total state added for AMX is 8 × 16 × 256/8 or 4096 bytes. The Tile Matrix Multiply (TMUL) unit can perform matrix multiplication

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.

on either 8-bit integers or 16-bit floating point numbers using the BF16 format (see Chapter 7). Thus a 256-bit row holds either 32 int8 or 16 BF16 data.

In general, the goal of these many extensions has been to accelerate carefully written libraries rather than for the compiler to generate them (see Appendix H), but recent x86 compilers are trying to generate such code, particularly for floating-point-intensive applications. Since the opcode determines the width of the SIMD register, every time the width doubles, so must the number of SIMD instructions.

Given these weaknesses, why are multimedia SIMD extensions so popular? First, they initially cost little to add to the standard arithmetic unit and they were easy to implement. Second, they require scant extra processor state compared to vector architectures, which is always a concern for context switch times. Third, you need a lot of memory bandwidth to support a vector architecture, which many computers don't have. Fourth, SIMD does not have to deal with problems in virtual memory when a single instruction can generate 32 memory accesses, any of which can cause a page fault. The original SIMD extensions used separate data transfers per SIMD group of operands that are aligned in memory, and so they cannot cross page boundaries. Another advantage of short, fixed-length "vectors" of SIMD is that it is easy to introduce instructions that can help with new media standards, such as instructions that perform permutations or instructions that consume either fewer or more operands than vectors can produce. Finally, there was concern about how well vector architectures can work with caches. More recent vector architectures have addressed all these problems. The overarching issue, however, is that due to the overriding importance of backward binary compatibility, once an architecture starts down the SIMD path, it's very hard to get off it.

Example To get an idea about what multimedia instructions look like, suppose we added a 256-bit SIMD multimedia instruction extension to RISC-V, tentatively called RVP for "packed." We concentrate on floating-point in this example. We add the suffix "`4D`" on instructions that operate on four double-precision operands at once. Like vector architectures, you can think of a SIMD Processor as having lanes, four in this case. RV64P expands the F registers to be the full width, in this case 256 bits. This example shows the RISC-V SIMD code for the `DAXPY` loop, with the changes to the RISC-V code for SIMD underlined. Assume that the starting addresses of `X` and `Y` are in `x5` and `x6`, respectively.

Answer Here is the RISC-V SIMD code:

```
        fld       f0,a              #Load scalar a
        splat.4D  f0,f0             #Make 4 copies of a
        addi      x28,x5,           #256 #Last address to load
Loop:   fld.4D    f1,0(x5)          #Load X[i] … X[i+3]
        fld.4D    f2,0(x6)          #Load Y[i] … Y[i+3]
                                    # a×X[i+3]+Y[i+3]
        fsd.4D    f2,0(x6)          #Store Y[i]… Y[i+3]
        addi      x5,x5,#32         #Increment index to X
        addi      x6,x6,#32         #Increment index to Y
        bne       x28,x5,Loop       #Check if done
```

The changes were:

- replacing every RISC-V double-precision instruction with its `4D` equivalent,
- increasing the increment from 8 to 32, and
- adding the `splat` instruction that makes 4 copies of `a` in the 256 bits of `f0`.

While not as dramatic as the 32× reduction of dynamic instruction bandwidth of RV64V, RISC-V SIMD does get almost a 4× reduction: 67 versus 258 instructions executed for RV64G. This code knows the number of elements. That number is often determined at run time, which would require an extra strip-mine loop to handle the case when the number is not a modulo of 4.

Programming Multimedia SIMD Architectures

Given the ad hoc nature of the SIMD multimedia extensions, the easiest way to use these instructions has been through libraries or by writing in assembly language.

Recent extensions have become more regular, giving compilers a more reasonable target. By borrowing techniques from vectorizing compilers, compilers are starting to produce SIMD instructions automatically. For example, advanced compilers today can generate SIMD floating-point instructions to deliver much higher performance for scientific codes. However, programmers must be sure to align all the data in memory to the width of the SIMD unit on which the code is run to prevent the compiler from generating scalar instructions for otherwise vectorizable code.

The Roofline Visual Performance Model

One visual, intuitive way to compare potential floating-point performance of variations of SIMD architectures is the Roofline model (Williams et al., 2009). The horizontal and diagonal lines of the graphs it produces give this simple model its name and indicate its value (see Figure 4.11). It ties together floating-point performance, memory performance, and arithmetic intensity in a two-dimensional graph. *Arithmetic intensity* is the ratio of floating-point operations per byte of memory accessed. It can be calculated by taking the total number of floating-point operations for a program divided by the total number of data bytes transferred to main memory during program execution. Figure 4.10 shows the relative arithmetic intensity of several example kernels.

Peak floating-point performance can be found using the hardware specifications. Many of the kernels in this case study do not fit in on-chip caches, so peak memory performance is defined by the memory system behind the caches. Note that we need the peak memory bandwidth that is available to the processors,

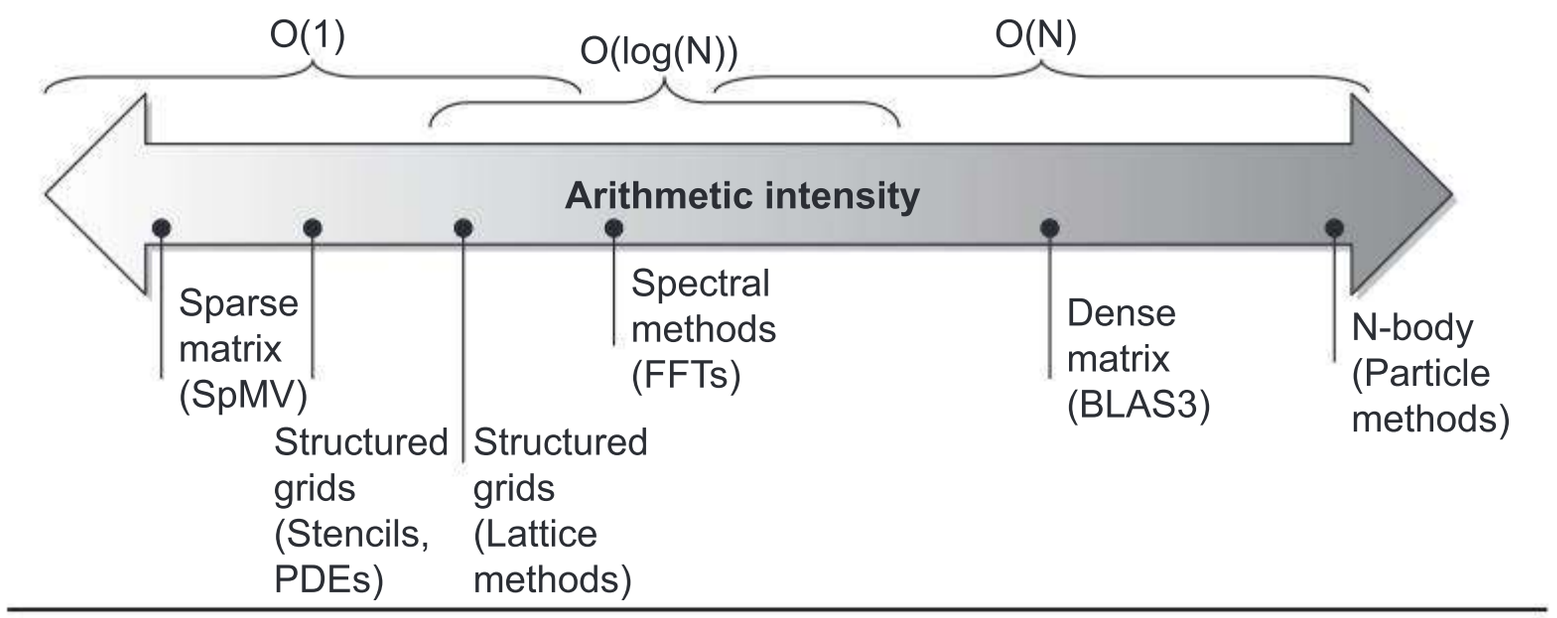

Figure 4.10 Arithmetic intensity, specified as the number of floating-point operations to run the program divided by the number of bytes accessed in main memory (Williams et al., 2009). Some kernels have an arithmetic intensity that scales with problem size, such as a dense matrix, but there are many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

not just at the DRAM pins as in Figure 4.30 on page 328. One way to find the (delivered) peak memory performance is to run the Stream benchmark.

Figure 4.11 shows the Roofline model for the NEC SX-9 vector processor on the left and the Intel Core i7 920 multicore processor on the right. The vertical y-axis is achievable floating-point performance from 2 to 256 GFLOPS/s. The horizontal x-axis is arithmetic intensity, varying from 1/8 FLOP/DRAM byte accessed to 16 FLOP/DRAM byte accessed in both graphs. Note that the graph is a log-log scale, and that Rooflines are done just once for a computer.

For a given kernel, we can find a point on the x-axis based on its arithmetic intensity. If we drew a vertical line through that point, the performance of the kernel on that computer must lie somewhere along that line. We can plot a horizontal line showing peak floating-point performance of the computer. Obviously, the actual floating-point performance can be no higher than the horizontal line because that is a hardware limit.

How could we plot the peak memory performance? Because the x-axis is FLOPs/byte and the y-axis is FLOPs/second, bytes/second is just a diagonal line at a 45-degree angle in this figure.

Thus, in the roofline model we can plot a third line that gives the maximum floating-point performance that the memory system of that computer can support for a given arithmetic intensity. We can express the limits as a formula to plot these lines in the graphs in Figure 4.11:

$$\begin{aligned} \textit{Attainable GFLOPs/s} &= \textit{Min}(\textit{Peak Memory BW} \\ &\quad \times \textit{Arithmetic Intensity}, \textit{Peak FloatingPoint Perf}.) \end{aligned}$$

If that conclusion was too quick, here is another try. Suppose we want to plot a curve, y = f(x), showing the performance limit when memory bandwidth is

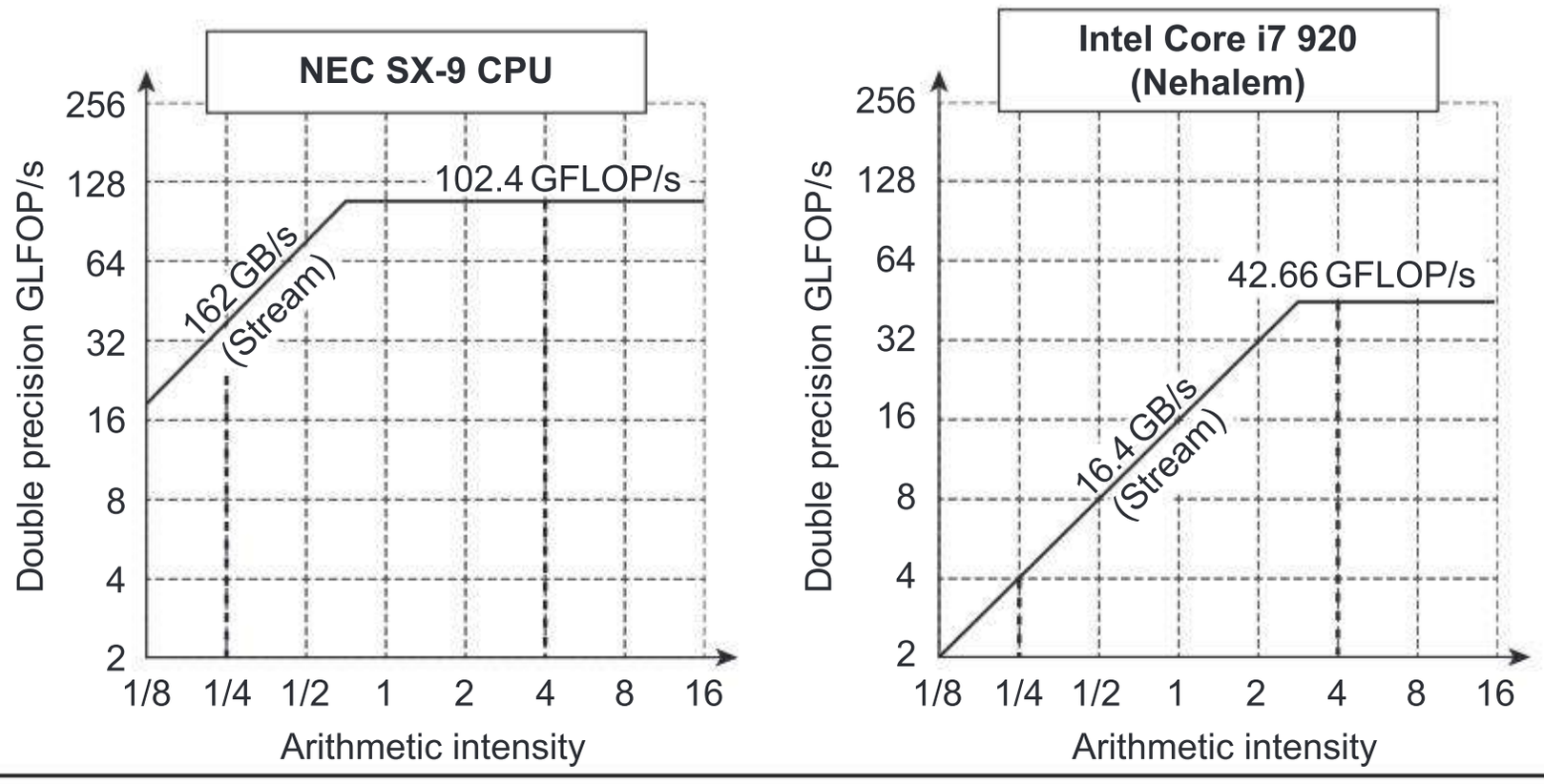

Figure 4.11 Roofline model for one NEC SX-9 vector processor on the left and the Intel Core i7 920 multicore computer with SIMD extensions on the right (Williams et al., 2009). This Roofline is for unit-stride memory accesses and double-precision floating-point (DP FP) performance. NEC SX-9 is a vector 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.

fully used. Any point (x,y) on the curve has associated with it a computation rate (y-axis) and a memory bandwidth. The memory bandwidth of the point is implicit and a consequence of the fact the x-axis (FLOPs/byte) captures the tradeoff between an amount of computation and the amount of memory required to perform that amount of computation. Specifically, we can find the memory bandwidth "m" for a given point (x,y) on the chart as m = y/x because the y-axis measures FLOPs/second, the x-axis *measures* FLOPs/byte, and FLOPs/second divided by FLOPs/byte gives bytes/second, the units for memory bandwidth. Next, note we can rearrange this equation to be y = m * x. Finally, if we let "m" be the constant peak memory bandwidth of the system, this equation describes a straight line with a slope m.

The "Roofline" sets an upper bound on performance of a kernel depending on its arithmetic intensity. If we think of arithmetic intensity as a pole that hits the roof, either it hits the flat part of the roof, which means performance is computationally limited, or it hits the slanted part of the roof, which means performance is ultimately limited by memory bandwidth. In Figure 4.11 the vertical dashed line on the right (arithmetic intensity of 4) is an example of the former and the vertical

dashed line on the left (arithmetic intensity of 1/4) is an example of the latter. Given a Roofline model of a computer, you can apply it repeatedly, because it doesn't vary by kernel.

Note that the "ridge point," where the diagonal and horizontal roofs meet, offers an interesting insight into a computer. If it is far to the right, then only kernels with very high arithmetic intensity can achieve the maximum performance of that computer. If it is far to the left, then almost any kernel can potentially hit the maximum performance. As we will see, this vector processor has both much higher memory bandwidth and a ridge point far to the left as compared to other SIMD Processors.

Figure 4.11 shows that the peak computational performance of the SX-9 is 2.4× faster than Core i7, but the memory performance is 10× faster. For programs with an arithmetic intensity of 0.25, the SX-9 is 10× faster (40.5 versus 4.1 GFLOP/s). The higher memory bandwidth moves the ridge point from 2.6 in the Core i7 down to 0.6 on the SX-9, which means many more programs can reach the peak computational performance on the vector processor.

Similarities and Differences Between Vector Architectures and Multimedia SIMD Computer

As mentioned above, an advantage of SIMD is that the initial support for data parallelism is very easy and low cost. An architect partitions the existing registers and ALU into many 8-, 16-, or 32-bit pieces and then computes them in parallel. The opcode provides data width and the operation. Data transfers are simply loads and stores of a single register.

To accelerate SIMD by supporting even more data parallelism, architects subsequently double the width of the registers to compute more partitions concurrently. Because ISAs traditionally embrace backward binary compatibility, and the opcode specifies the data width, expanding the SIMD registers also expands the SIMD instruction set. Each subsequent step of doubling the width of SIMD registers and the number of SIMD instructions leads instruction set architectures (ISAs) down the path of escalating size, which is borne by processor designers, compiler writers, and assembly language programmers. The x86 instruction set has grown from 80 to around 1400 instructions since 1978, largely fueled by SIMD.

Vector computers gather objects from main memory and put them into long, sequential vector registers. Pipelined execution units compute very efficiently on these vector registers. Vector architectures then scatter the results back from the vector registers to main memory. The data loaded/stored can be from operands located in sequential, nonsequential (nonunit stride), and random (gather/scatter). Within the vector register the operands are sequential. While a simple vector processor might execute one vector element at a time, element operations are independent by definition, and so a processor could theoretically compute all of them

simultaneously. Today's vector processors typically execute two, four, or eight word elements per clock cycle. Hardware handles the fringe cases when the vector length is not a multiple of the number of elements executed per clock cycle. Like SIMD, vectors can operate on variable data widths. A vector processor with N 64-bit elements per register also computes on vectors with 2N 32-bit, 4N 16-bit, and 8N 8-bit elements.

Figure 4.12 summarizes the number of instructions in DAXPY of programs for the x86 AVX2 and RV32V (Patterson and Waterman, 2017). The SIMD computation code is dwarfed by the bookkeeping code. Two-thirds to three-fourths of the code for x86 AVX2 is SIMD overhead. The bookkeeping code includes replicating the scalar variable a for use in SIMD execution, the strip mining code to handle when n is not a multiple of the SIMD register width, and to skip the main loop if n is 0. RV32V code doesn't need such bookkeeping code, which slashes the static number of instructions. Unlike SIMD, it has a vector length register `vl`, which makes the vector instructions work at any value of `n`. It also has the `vsetvl` instructions, which automatically handles the case when `n` is not a multiple of the number of vector lanes and when `n` is 0.

However, the biggest difference between SIMD and vector processing is not the static code size. The SIMD instructions execute 10 to 20 times more instructions than RV32V because each SIMD loop does only 2 or 4 elements instead of 64 in the vector case. The extra instruction fetches and instruction decodes means higher energy to perform the same task.

Comparing the results to the scalar versions of DAXPY, SIMD roughly doubles the size of the code in instructions and bytes, despite the main loop remaining the same size. The reduction in the dynamic number of instructions executed is a factor of 2 or 4, depending on the width of the SIMD registers. However, the RV32V vector static code size increases by a factor of only 1.2, yet the dynamic instruction count is a factor of 43 smaller.

While dynamic instruction count is a large difference, it may not be the most significant disparity between SIMD and vector. It may be the mushrooming size of the ISA and the bookkeeping code. ISAs like x86 that follow the backward binary compatible doctrine and evolutionary instruction set design must duplicate

ISA	IA-32 AVX2	RV32 V
Instructions (static)	29	13
Instructions per Main Loop	6	10
Bookkeeping Instructions	23	3
Results per Main Loop	4	64
Instructions (dynamic n = 1000)	1517	163

Figure 4.12 Number of instructions and total number of instructions executed for n = 1000 for the DAXPY program.

all the old SIMD instructions defined for narrower SIMD registers every time they change the SIMD width. Hundreds of x86 instructions were created over many generations of SIMD ISAs and hundreds more are in their future. The cognitive load on the assembly language programmer is large. What does `vfmadd213pd` mean and when do you use it? This also means that third-party applications have to be recoded to use these new instructions. In many cases this is a nontrivial task.

In comparison, RV64V code is not affected by memory size for vector registers. Not only is RV64V unchanged if vector memory size expands, but you don't even have to recompile. Since the processor supplies the value of maximum vector length `VLMAX` for RV64V, the code is untouched if the processor designer doubles or halves the vector memory.

As SIMD ISAs dictate the hardware, achieving greater DLP means changing the instruction set and the compiler. In contrast, vector ISAs allow processor designers to choose the resources for data parallelism for their application without affecting the programmer or the compiler. Vector architectures do a much better job of following the design principle of isolating architecture from implementation.

4.4 Graphics Processing Units

People can buy a GPU chip with thousands of parallel floating-point units for a few hundred dollars and plug it into their desk-side PC. Such affordability and convenience make high-performance computing available to many. The interest in GPU computing blossomed when this potential was combined with a programming language that made GPUs easier to program.

When people talk about computing using GPUs today, they can be referring to

1. GPUs for graphics generation
2. GPUs for scientific or multimedia computation
3. GPUs for machine learning computation

For graphics generation, GPUs are the unquestioned winner. Many programmers of scientific and multimedia applications today are pondering whether to use GPUs or CPUs. For example, 30% of the top 500 supercomputers rely on GPUs to do the bulk of their calculations as of 2022. Programmers interested in machine learning, which is the subject of Chapter 7, prefer GPUs over CPUs.

GPUs and CPUs do not go back in computer architecture genealogy to a common ancestor; there is no "missing link" that explains both. As Section 4.10 describes, the primary ancestors of GPUs are graphics accelerators, as doing graphics well is the reason why GPUs exist. While GPUs are moving toward

mainstream computing, they can't abandon their responsibility to continue to excel at graphics. Thus the design of GPUs may make more sense when architects ask, given the hardware invested to do graphics well, how can we supplement it to improve the performance of a wider range of applications?

Note that this section concentrates on using GPUs for computing. To see how GPU computing combines with the traditional role of graphics acceleration, see "Graphics and Computing GPUs," by John Nickolls and David Kirk (Appendix C in the 6th edition of *Computer Organization and Design* by the same authors as this book).

Because the terminology and some hardware features are quite different from vector and SIMD architectures, we believe it will be easier if we start with the simplified programming model for GPUs before we describe the architecture.

Programming the GPU

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

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

The challenge for the GPU programmer is not simply getting good performance on the GPU but also coordinating the scheduling of computation on the system processor and the GPU and the transfer of data between system memory and GPU memory. Moreover, as we will see later in this section, GPUs have virtually every type of parallelism that can be captured by the programming environment: multithreading, MIMD, SIMD, and even instruction-level.

NVIDIA decided to develop a C-like language and programming environment that would improve the productivity of GPU programmers by attacking both the challenges of heterogeneous computing and of multifaceted parallelism. The name of their system is *CUDA*, for Compute Unified Device Architecture. CUDA produces C/C++ for the system processor (*host*) and a C and C++ dialect for the GPU (*device*, thus the D in CUDA). NVIDIA decided that the unifying theme of all these forms of parallelism is the *CUDA Thread*. Using this lowest level of parallelism as the programming primitive, the compiler and the hardware can gang thousands of CUDA Threads together to utilize the various styles of parallelism within a GPU: multithreading, MIMD, SIMD, and instruction-level parallelism. Therefore NVIDIA classifies the CUDA programming model as single instruction, multiple thread (*SIMT*). For reasons we will soon see, these threads are blocked together and executed in groups of threads, called a *Thread Block*. We call the hardware that executes a whole block of threads a *multithreaded SIMD Processor*.

We need just a few details before we can give an example of a CUDA program:

- To distinguish between functions for the GPU (device) and functions for the system processor (host), CUDA uses `__device__` or `__global__` for the former and `__host__` for the latter.
- CUDA variables declared with `__device__` are allocated to the GPU Memory (see below), which is accessible by all multithreaded SIMD Processors.
- The extended function call syntax for the function *name* that runs on the GPU is

  name < <<`dimGrid`, `dimBlock`>> > (… *parameter list*…)

  where `dimGrid` and `dimBlock` specify the dimensions of the code (in Thread Blocks) and the dimensions of a block (in threads).
- Besides the identifier for blocks (`blockIdx`) and the identifier for each thread in a block (`threadIdx`), CUDA provides a keyword for the number of threads per block (`blockDim`), which comes from the `dimBlock` parameter in the preceding bullet.

Before seeing the CUDA code, let's start with conventional C code for the `DAXPY` loop from Section 4.2:

```
// 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];
}
```

Following is the CUDA version. We launch `n` threads, one per vector element, with 256 CUDA Threads per Thread Block in a multithreaded SIMD Processor. The GPU function starts by calculating the corresponding element index `i` based on the block ID, the number of threads per block, and the thread ID. As long as this index is within the array (`i < n`), it performs the multiply and add.

```
// Invoke DAXPY with 256 threads per Thread Block
__host__int nblocks = (n + 255) / 256;
daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__global__
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];
}
```

Comparing the C and CUDA codes, we see a common pattern to parallelizing data-parallel CUDA code. The C version has a loop where each iteration is independent of the others, allowing the loop to be transformed straightforwardly into a parallel code where each loop iteration becomes a separate thread. (As previously mentioned and described in detail in Section 4.5, vectorizing compilers also rely on a lack of dependences between iterations of a loop, which are called *loop-carried dependences*.) The programmer determines the parallelism in CUDA explicitly by specifying the grid dimensions and the number of threads per SIMD Processor. By assigning a single thread to each element, there is no need to synchronize between threads when writing results to memory.

The GPU hardware handles parallel execution and thread management; it is not done by applications or by an operating system. To simplify scheduling by the hardware, CUDA requires that Thread Blocks be able to execute independently and in any order. Different Thread Blocks cannot communicate directly, although they can *coordinate* using atomic memory operations in global memory.

As we will soon see, many GPU hardware concepts are not obvious in CUDA. Performance programmers must keep the GPU hardware in mind when writing in CUDA. Writing efficient GPU code requires that programmers think in terms of SIMD operations, even though the CUDA programming model looks like MIMD. That perspective could hurt programmer productivity, but most programmers are already using GPUs instead of CPUs to get performance. For reasons explained shortly, they know that they need to keep groups of 32 threads together in control flow to get the best performance from multithreaded SIMD Processors and to create many more threads per multithreaded SIMD Processor to hide latency to DRAM. They also need to keep the data addresses localized to one or a few blocks of memory to get the expected memory performance.

Like many parallel systems, a compromise between productivity and performance is for CUDA to include intrinsics to give programmers explicit control over the hardware. The struggle between productivity on the one hand and allowing the programmer to be able to express anything that the hardware can do on the other hand happens often in parallel computing. It will be interesting to see how the language evolves in this classic productivity-performance battle and to see whether CUDA becomes popular for other GPUs or even other architectural styles.

NVIDIA GPU Computational Structures

The uncommon heritage mentioned above helps explain why GPUs have their own architectural style and their own terminology independent from CPUs. One obstacle to understanding GPUs has been the jargon, with some terms even having misleading names. This obstacle has been surprisingly difficult to overcome, as the many rewrites of this chapter can attest.

To try to bridge the twin goals of making the architecture of GPUs understandable *and* learning the many GPU terms with nontraditional definitions, our approach is to use the CUDA terminology for software but initially use more descriptive terms for the hardware, sometimes borrowing terms from OpenCL. Once we explain the GPU architecture in our terms, we'll translate them into the official jargon of NVIDIA GPUs.

From left to right, Figure 4.13 lists the descriptive name used in this section, the closest term from mainstream computing, the official NVIDIA GPU jargon in case you are interested, and then a short explanation of the term. The rest of this section explains the microarchitectural features of GPUs using the descriptive terms in the left column of the figure.

We use NVIDIA systems as our example as they are representative of GPU architectures. Specifically, we follow the terminology of the preceding CUDA parallel programming language and use the NVIDIA Pascal GPU as the example (see Section 4.7). Subsequent NVIDIA GPUs have the same underlying microarchitecture, although generally copies of some of Pascal's features.

Like vector architectures, NVIDIA GPUs work well only with data-level parallel problems. Both styles have gather-scatter data transfers and mask registers, and NVIDIA GPU processors have even more registers than do vector processors. Sometimes, NVIDIA GPUs implement certain features in hardware that vector processors would implement in software. This difference is because vector processors have a scalar processor that can execute a software function. Unlike most vector architectures, NVIDIA GPUs also rely on multithreading within a single multithreaded SIMD Processor to hide memory latency (see Chapters 2 and 3). Efficient code for both vector architectures and GPUs, however, requires programmers to think in groups of SIMD operations.

A *Grid* is the code that runs on an NVIDIA GPU that consists of a set of *Thread Blocks*. Figure 4.13 draws the analogy between a grid and a vectorized loop and between a Thread Block and the body of that loop (after it has been strip-mined so that it is a full computation loop). To give a concrete example, let's suppose we want to multiply two vectors together, each 8192 elements long: `A = B * C`. We'll return to this example throughout this section. Figure 4.14 shows the relationship between this example and these first two GPU terms. The GPU code that works on the whole 8192 element multiply is called a *Grid* (similar to a vectorized loop). To break it down into more manageable sizes, a Grid is composed of *Thread Blocks* (like a body of a vectorized loop), each with up to 512 elements. Note that a GPU SIMD instruction executes 32 elements at a time. With 8192 elements in the vectors, this example thus has 16 Thread Blocks because $16 = 8192 \div 512$. The Grid and Thread Block are programming abstractions implemented in GPU hardware that help programmers organize their CUDA code. (The Thread Block is analogous to a strip-mined vector loop with a vector length of 32.)

A Thread Block is assigned to a processor that executes that code, which we call a *multithreaded SIMD Processor*, by the *Thread Block Scheduler*. The

	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 4.13 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.

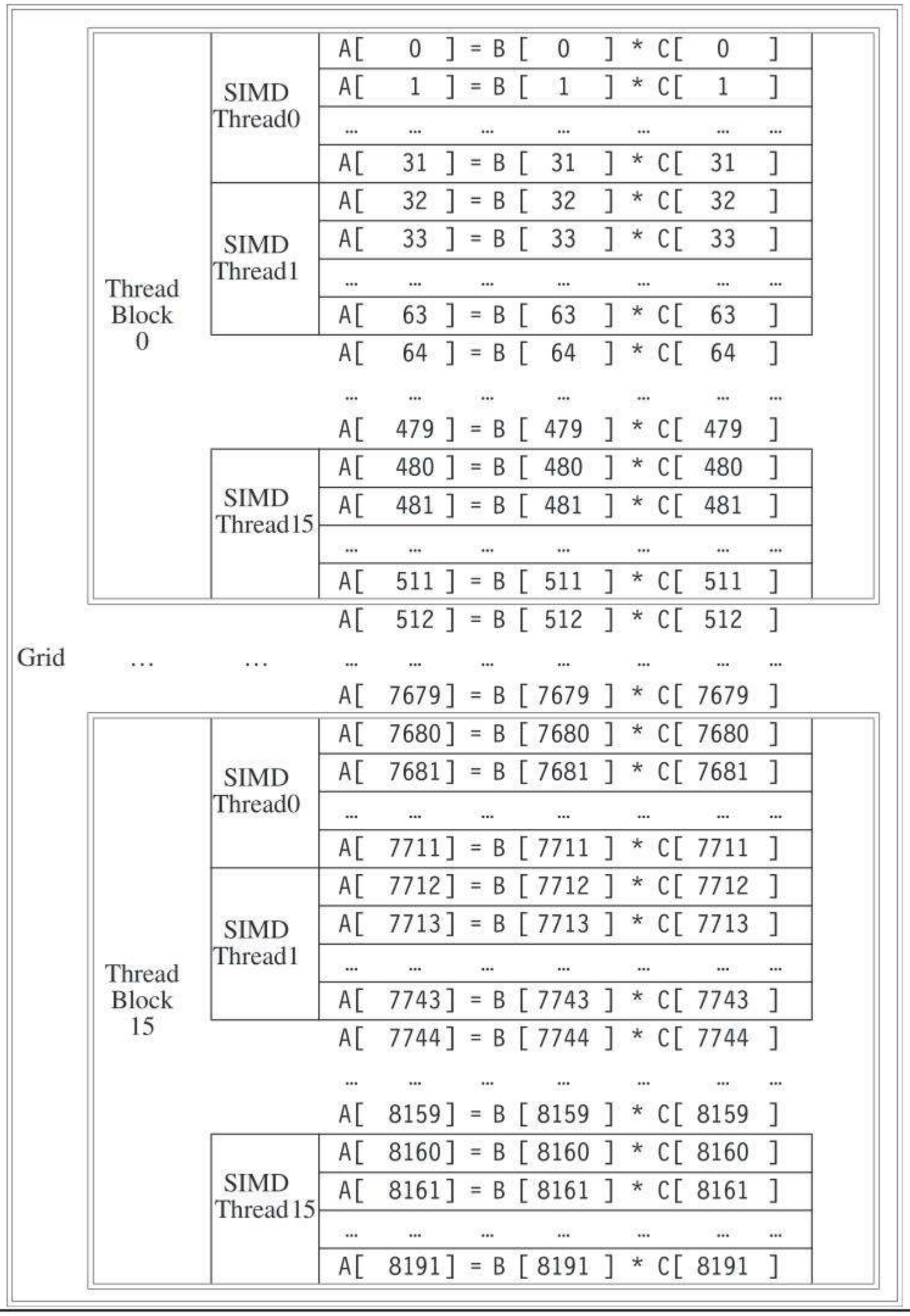

Figure 4.14 The mapping of a Grid (vectorizable loop), Thread Blocks (SIMD basic blocks), and threads of SIMD instructions to a vector-vector multiply, with each vector being 8192 elements long. Each thread of SIMD instructions calculates 32 elements per instruction, and in this example each Thread Block 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.)

programmer tells the Thread Block Scheduler, which is implemented in hardware, how many Thread Blocks to run. In this example it would send 16 to multithreaded SIMD Processors to compute all 8192 elements of this loop using 16 Thread Blocks.

Figure 4.15 shows a simplified block diagram of a multithreaded SIMD Processor. It is similar to a vector processor, but it has many parallel functional units instead of a few that are deeply pipelined, as in a vector processor. In the programming example in Figure 4.14 each multithreaded SIMD Processor is assigned 512 elements of the vectors to work on. SIMD Processors are full processors with separate PCs and are programmed using threads (see Chapter 3).

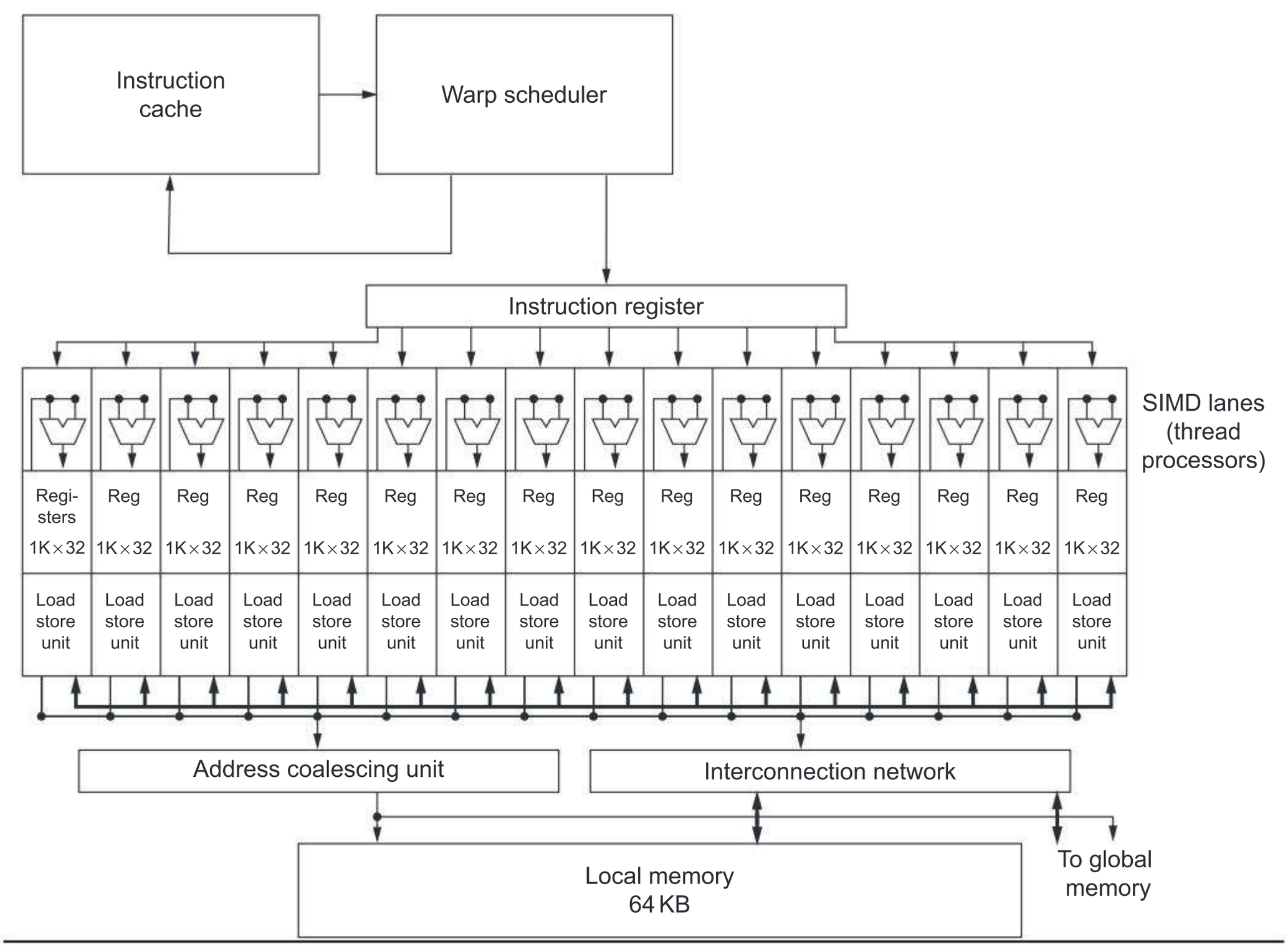

Figure 4.15 Simplified block diagram of a multithreaded SIMD Processor. It has 16 SIMD Lanes. The SIMD Thread Scheduler has, say, 64 independent threads of SIMD instructions that it schedules with a table of 64 program counters (PCs). Note that each lane has 1024 32-bit registers.

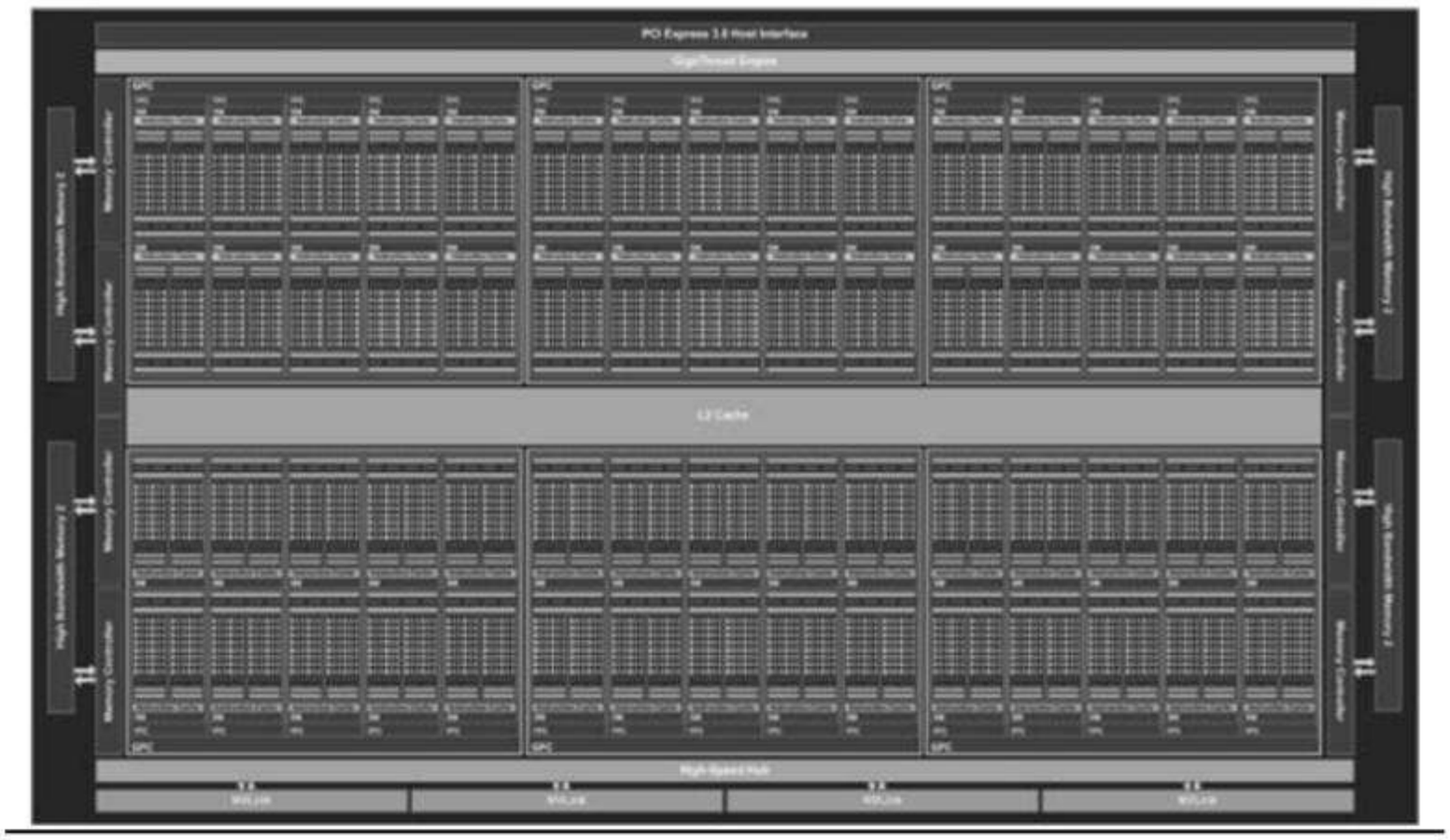

Figure 4.16 Full-chip block diagram of the Pascal P100 GPU. It has 56 multithreaded SIMD Processors, each with an L1 cache and local memory, 32 L2 units, and a memory-bus width of 4096 data wires. (It has 60 blocks, with four spares to improve yield.) The P100 has 4 HBM2 ports supporting up to 16 GB of capacity. It contains 15.4 billion transistors.

The GPU hardware contains a collection of multithreaded SIMD Processors that execute a Grid of Thread Blocks (equivalent to bodies of vectorized loop). That is, a GPU is a multiprocessor composed of multithreaded SIMD Processors.

A GPU can have from one to several dozen multithreaded SIMD Processors. For example, the Pascal P100 system has 56 and the Hopper H100 has 132, while the smaller chips may have as few as one or two. To provide transparent scalability across models of GPUs with a differing number of multithreaded SIMD Processors, the Thread Block Scheduler assigns Thread Blocks (bodies of a vectorized loop) to multithreaded SIMD Processors. Figure 4.16 shows the floor plan of the P100 implementation of the Pascal architecture.

Dropping down one more level of detail, the machine object that the hardware creates, manages, schedules, and executes is a *thread of SIMD instructions*. It is a traditional thread containing exclusively SIMD instructions. These threads of SIMD instructions have their own PCs, and they run on a multithreaded SIMD Processor. The *SIMD Thread Scheduler* knows which threads of SIMD instructions are ready to run and then sends them off to a dispatch unit to be run on the multithreaded SIMD Processor. Thus GPU hardware has two levels of hardware schedulers:

(1) the *Thread Block Scheduler* that assigns Thread Blocks (bodies of vectorized loops) to multithreaded SIMD Processors, and

(2) the SIMD Thread Scheduler *within* a SIMD Processor, which schedules when threads of SIMD instructions should run inside the processor.

The SIMD instructions of these threads are 32 wide, so each thread of SIMD instructions in this example would compute 32 of the elements of the computation. In this example Thread Blocks would contain 512/32 = 16 SIMD Threads (see Figure 4.14).

Because the thread consists of SIMD instructions, the SIMD Processor must have parallel functional units to perform the operation. We call them *SIMD Lanes*, and they are quite similar to the Vector Lanes in Section 4.2.

For the Pascal GPU, each 32-wide thread of SIMD instructions is mapped to 16 physical SIMD Lanes, so each SIMD instruction in a thread of SIMD instructions takes 2 clock cycles to complete. Each thread of SIMD instructions is executed in lock step and scheduled only at the beginning. Staying with the analogy of a SIMD Processor as a vector processor, you could say that it has 16 lanes, the vector length is 32, and the chime is 2 clock cycles. (This wide but shallow nature is why we use the more accurate analogy of SIMD Processor rather than vector processor.)

Note that the number of lanes in a GPU SIMD Processor can be anything up to the number of threads in a Thread Block, just as the number of lanes in a vector processor can vary between 1 and the maximum vector length. For example, across GPU generations, the number of lanes per SIMD Processor has fluctuated between 8 and 32.

Because by definition the threads of SIMD instructions are independent, the SIMD Thread Scheduler can pick whatever thread of SIMD instructions is ready and need not stick with the next SIMD instruction in the sequence within a thread. The SIMD Thread Scheduler includes a scoreboard (see Chapter 3) to keep track of up to 64 threads of varying execution time SIMD instructions to see which SIMD instruction is ready to go. The latency of memory instructions is variable because of hits and misses in the caches and the TLB, thus the requirement of a scoreboard to determine when these instructions are complete. Figure 4.17 shows the SIMD Thread Scheduler picking threads of SIMD instructions in a different order over time. The assumption of GPU architects is that GPU applications have so many threads of SIMD instructions that multithreading can both hide the latency to DRAM and increase utilization of multithreaded SIMD Processors.

Continuing our vector multiply example, each multithreaded SIMD Processor must load 32 elements of two vectors from memory into registers, perform the multiply by reading and writing registers, and store the product back from registers into memory. To hold these memory elements, a SIMD Processor has between an impressive 32,768–65,536 32-bit registers (1024 per lane in Figure 4.15), depending on the model of the Pascal GPU. Just like a vector processor, these registers are divided logically across the Vector Lanes or, in this case, SIMD Lanes.

Each SIMD Thread is limited to no more than 256 registers, so you might think of a SIMD Thread as having up to 256 vector registers, with each vector register having 32 elements and each element being 32 bits wide. (Because

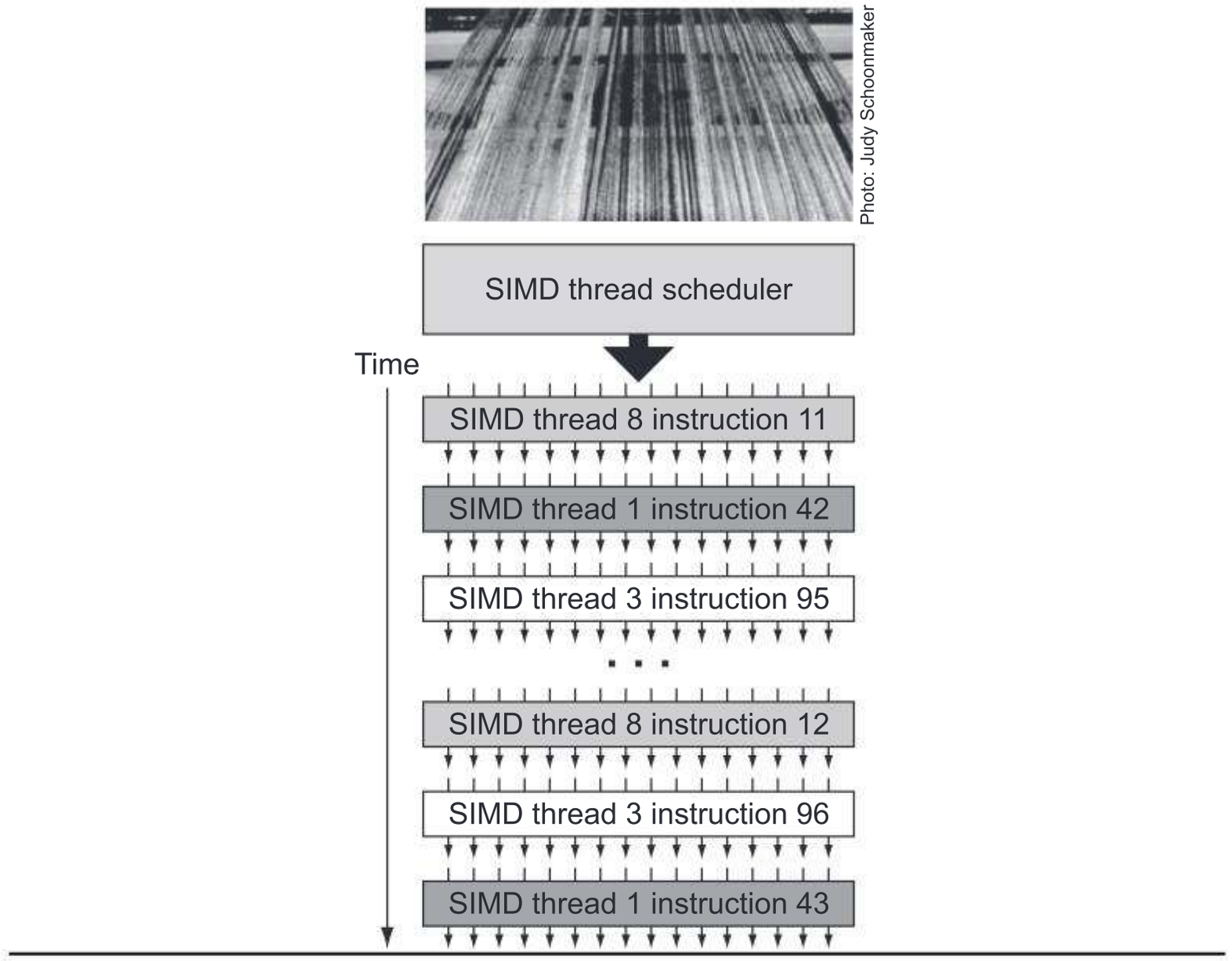

Figure 4.17 Scheduling of threads of SIMD instructions. The scheduler selects a ready thread of SIMD instructions and issues an instruction synchronously to all the SIMD Lanes executing the SIMD Thread. Because threads of SIMD instructions are independent, the scheduler may select a different SIMD Thread each time.

double-precision floating-point operands use two adjacent 32-bit registers, an alternative view is that each SIMD Thread has 128 vector registers of 32 elements, each of which is 64 bits wide.)

There is a trade-off between register use and maximum number of threads. Fewer registers per thread means more threads are possible, and more registers mean fewer threads. That is, not all SIMD Threads need to have the maximum number of registers. Pascal architects believe much of this precious silicon area would be idle if all threads had the maximum number of registers.

To be able to execute many threads of SIMD instructions, each is dynamically allocated a set of the physical registers on each SIMD Processor when threads of SIMD instructions are created and freed when the SIMD Thread exits. For example, a programmer can have a Thread Block that uses 36 registers per thread with, say, 16 SIMD Threads alongside another Thread Block that has 20 registers per thread with 32 SIMD Threads. Subsequent Thread Blocks may show up in any order, and the registers have to be allocated on demand. While this variability can lead to fragmentation and make some registers unavailable, in practice, most Thread Blocks use the same number of registers for a given vectorizable loop ("grid"). The hardware must know where the registers for each Thread Block

are in the large register file, and this is recorded on a per Thread-Block basis. This flexibility requires routing, arbitration, and banking in the hardware because a specific register for a given Thread Block could end up in any location in the register file.

Note that a CUDA Thread is just a vertical cut of a thread of SIMD instructions, corresponding to one element executed by one SIMD Lane. Beware that CUDA Threads are very different from POSIX Threads; you can't make arbitrary system calls from a CUDA Thread.

We're now ready to see what GPU instructions look like.

NVIDIA GPU Instruction Set Architecture

Unlike most system processors, the instruction set target of the NVIDIA compilers is an abstraction of the hardware instruction set. *PTX* (*Parallel Thread Execution*) provides a stable instruction set for compilers and compatibility across generations of GPUs. The hardware instruction set is hidden from the programmer. PTX instructions describe the operations on a single CUDA Thread and usually map one-to-one with hardware instructions, but one PTX instruction can expand to many machine instructions, and vice versa. PTX uses an unlimited number of write-once registers and the compiler must run a register allocation procedure to map the PTX registers to a fixed number of read-write hardware registers available on the actual device. The optimizer runs subsequently and can reduce register use even further. This optimizer also eliminates dead code, folds instructions together, and calculates places where branches might diverge and places where diverged paths could converge.

Although there is some similarity between the x86 microarchitecture and PTX—in that both translate to an internal form (microinstructions for x86)—the difference is that this translation happens in hardware at runtime during execution on the x86 versus in software and load time on a GPU.

The format of a PTX instruction is

```
opcode.type d, a, b, c ; comment
```

where `d` is the destination operand; `a`, `b`, and `c` are source operands; and everything to the right of semicolon is a comment. The operation type (`opcode.type`) is one of the following:

Type	.type specifier
Untyped bits 8, 16, 32, and 64 bits	`.b8, .b16, .b32, .b64`
Unsigned integer 8, 16, 32, and 64 bits	`.u8, .u16, .u32, .u64`
Signed integer 8, 16, 32, and 64 bits	`.s8, .s16, .s32, .s64`
Floating Point 16, 32, and 64 bits	`.f16, .f32, .f64`

Source operands are 32-bit or 64-bit registers or a constant value. Destinations are registers, except for store instructions.

Figure 4.18 shows the basic PTX instruction set. All instructions can be predicated by 1-bit predicate registers, which can be set by a set predicate instruction (`setp`). The control flow instructions are functions `call` and `return`, thread `exit`, `branch`, and barrier synchronization for threads within a Thread Block (`bar.sync`). Placing a predicate before a branch instruction gives us conditional branches. The compiler or PTX programmer declares virtual registers as 32-bit or 64-bit typed or untyped values. For example, `R0`, `R1`, … are for 32-bit values and `RD0`, `RD1`, … are for 64-bit registers. Recall that the assignment of virtual registers to physical registers occurs at load time with PTX.

The following sequence of PTX instructions is for one iteration of our DAXPY loop on page 292:

```
shl.u32 R8, blockIdx, 8     ; Thread Block ID * Block size
                            ; (256 or 28)
add.u32 R8, R8, threadIdx   ; R8 = i = my CUDA Thread ID
shl.u32 R8, R8, 3           ; byte offset
ld.global.f64 RD0, [X+R8]   ; RD0 = X[i]
ld.global.f64 RD2, [Y+R8]   ; RD2 = Y[i]
mul.f64 RD0, RD0, RD4       ; Product in RD0 = RD0 * RD4
                            ; (scalar a)
add.f64 RD0, RD0, RD2       ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0   ; Y[i] = sum (X[i]*a + Y[i])
```

As demonstrated above, the CUDA programmer assigns one CUDA Thread to each loop iteration and CUDA offers a unique identifier number to each Thread Block (`blockIdx`) and one to each CUDA Thread within a block (`threadIdx`). Thus it creates 8192 CUDA Threads and uses the unique number to address each element within the array, so there is no incrementing or branching code. The first three PTX instructions calculate that unique element byte offset in `R8`, which is added to the base of the arrays. The following PTX instructions load two double-precision floating-point operands, multiply and add them, and store the sum. (We'll describe the PTX code corresponding to the CUDA code "`if (i < n)`" below.)

What the example shows is that a GPGPU programmer initially tries to create as many threads as possible and possibly further optimizes by going after known sources of inefficiencies like uncoalesced accesses. Some of the CUDA examples that NVIDIA provides start with simple but not necessarily efficient kernels and progressively improve these to more efficient code, such as reductions.

Note that, unlike vector architectures, GPUs don't have separate instructions for sequential data transfers, strided data transfers, and gather-scatter data transfers. All data transfers are gather-scatter! To regain the efficiency of sequential (unit-stride) data transfers, GPUs include special Address Coalescing hardware to recognize when the SIMD Lanes within a thread of SIMD instructions are collectively issuing sequential addresses. That runtime hardware then notifies the Memory Interface Unit to request a block transfer of 32 sequential words.

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 4.18 Basic PTX GPU thread instructions.

To get this important performance improvement, the GPU programmer must ensure that adjacent CUDA Threads access nearby addresses at the same time so that they can be coalesced into one or a few memory or cache blocks, which our example does.

Conditional Branching in GPUs

Just like the case with unit-stride data transfers, there are strong similarities between how vector architectures and GPUs handle IF statements, with the former implementing the mechanism largely in software with limited hardware support and the latter making use of even more hardware. As we will see, in addition to explicit predicate registers, GPU branch hardware uses internal masks, a branch synchronization stack, and instruction markers to manage when a branch diverges into multiple execution paths and when the paths converge.

At the PTX assembler level, control flow of one CUDA Thread is described by the PTX instructions branch, call, return, and exit, plus individual per-thread-lane predication of each instruction, specified by the programmer with per-thread-lane 1-bit predicate registers. The PTX assembler analyzes the PTX branch graph and optimizes it to the fastest GPU hardware instruction sequence. Each thread can make its own decision on a branch and does not need to be in lock step.

At the GPU hardware instruction level, control flow includes branch, jump, jump indexed, call, call indexed, return, exit, and special instructions that manage the branch synchronization stack. Some earlier GPUs provide each SIMD Thread with its own stack. A stack entry contains an identifier token, a target instruction address, and a target thread-active mask. There are GPU special instructions that push stack entries for a SIMD Thread and special instructions and instruction markers that pop a stack entry or unwind the stack to a specified entry and branch to the target instruction address with the target thread-active mask. GPU hardware instructions also have an individual per-lane predication (enable/disable), specified with a 1-bit predicate register for each lane. The Volta GPU and successors drop the stack and allow warps to split.

The PTX assembler typically optimizes a simple outer-level IF-THEN-ELSE statement coded with PTX branch instructions to solely predicated GPU instructions, without any GPU branch instructions. A more complex control flow often results in a mixture of predication and GPU branch instructions with special instructions and markers that use the branch synchronization stack to push a stack entry when some lanes branch to the target address while others fall through. NVIDIA says a branch *diverges* when this happens. This mixture is also used when a SIMD Lane executes a synchronization marker or *converges*, which pops a stack entry and branches to the stack-entry address with the stack-entry thread-active mask.

The PTX assembler identifies loop branches and generates GPU branch instructions that branch to the top of the loop, along with special stack instructions to handle individual lanes breaking out of the loop and converging the SIMD

Lanes when all lanes have completed the loop. GPU indexed jump and indexed call instructions push entries on the stack so that when all lanes complete the switch statement or function call, the SIMD Thread converges.

A GPU set predicate instruction (`setp` in Figure 4.18) evaluates the conditional part of the IF statement. The PTX branch instruction then depends on that predicate. If the PTX assembler generates predicated instructions with no GPU branch instructions, it uses a per-lane predicate register to enable or disable each SIMD Lane for each instruction. The SIMD instructions in the threads inside the THEN part of the IF statement broadcast operations to all the SIMD Lanes. Those lanes with the predicate set to 1 perform the operation and store the result, and the other SIMD Lanes don't perform an operation or store a result. For the ELSE statement, the instructions use the complement of the predicate (relative to the THEN statement), so the SIMD Lanes that were idle now perform the operation and store the result while their formerly active siblings don't. At the end of the ELSE statement, the instructions are not predicated, so the original computation can proceed. Thus, for equal length paths, an IF-THEN-ELSE operates at 50% efficiency or less.

IF statements can be nested, thus the use of a stack, and the PTX assembler typically generates a mix of predicated instructions and GPU branch and special synchronization instructions for complex control flow. Note that deep nesting can mean that most SIMD Lanes are idle during execution of nested conditional statements. Thus doubly nested IF statements with equal-length paths run at 25% efficiency, triply nested at 12.5% efficiency, and so on. The analogous case would be a vector processor operating where only a few of the mask bits are ones, which is what would occur with nested if-statements, since the vector mask register would be set by combining the conditions.

Dropping down a level of detail, the PTX assembler sets a "branch synchronization" marker on appropriate conditional branch instructions that pushes the current active mask on a stack inside each SIMD Thread. If the conditional branch diverges (some lanes take the branch but some fall through), it pushes a stack entry and sets the current internal active mask based on the condition. A branch synchronization marker pops the diverged branch entry and flips the mask bits before the ELSE portion. (If there are nested IF-ELSE statements, adjusting the mask bits is a little more complicated than simply inverting them.) At the end of the IF statement, the PTX assembler adds another branch synchronization marker that pops the prior active mask off the stack into the current active mask.

If all the mask bits are set to 1, then the branch instruction at the end of the THEN skips over the instructions in the ELSE part. There is a similar optimization for the THEN part in case all the mask bits are 0 because the conditional branch jumps over the THEN instructions. Parallel IF statements and PTX branches often use branch conditions that are unanimous (all lanes agree to follow the same path) such that the SIMD Thread does not diverge into a different individual lane control flow. The PTX assembler optimizes such branches to skip

over blocks of instructions that are not executed by any lane of a SIMD Thread. This optimization is useful in conditional error checking, for example, where the test must be made but is rarely taken.

The code for a conditional statement similar to the one in Section 4.2 is

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

This IF statement could compile to the following PTX instructions (assuming that `R8` already has the scaled thread ID), with `*Push`, `*Comp`, `*Pop` indicating the branch synchronization markers inserted by the PTX assembler that push the old mask, complement the current mask, and pop to restore the old mask:

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

Once again, a SIMD Processor normally executes all instructions in the IF-THEN-ELSE statement. It's just that only some of the SIMD Lanes are enabled for the THEN instructions and some lanes for the ELSE instructions. As previously mentioned, in the surprisingly common case that the individual lanes agree on the predicated branch—such as branching on a parameter value that is the same for all lanes so that all active mask bits are 0s or all are 1s—the branch skips the THEN instructions or the ELSE instructions.

This flexibility makes it appear that an element has its own program counter. However, in the slowest case, only one SIMD Lane could store its result every 2 clock cycles, with the rest idle. The analogous slowest case for vector architectures is operating with only one mask bit set to 1. This flexibility can lead naive GPU programmers to poor performance, but it can be helpful in the early stages of program development. Keep in mind, however, that the only choice for a SIMD Lane in a clock cycle is to perform the operation specified in the PTX instruction or be idle; two SIMD Lanes cannot simultaneously execute different instructions. (In a few paragraphs below we'll contrast this solution to the vector approach.)

This flexibility also helps explain the name *CUDA Thread* given to each element in a thread of SIMD instructions, because it gives the illusion of acting independently. A naive programmer may think that this thread abstraction means GPUs handle conditional branches more gracefully. Some threads go one way, the rest go another, which seems true as long as you're not in a hurry. Each CUDA

Thread is either executing the same instruction as every other thread in the Thread Block or it is idle. This synchronization makes it easier to handle loops with conditional branches because the mask capability can turn off SIMD Lanes and it detects the end of the loop automatically.

The resulting performance sometimes belies that simple abstraction. Writing programs that operate SIMD Lanes in this highly independent MIMD mode is like writing programs that use lots of virtual address space on a computer with a smaller physical memory. Both are correct, but they will likely run so slowly that the programmer will not be pleased with the result.

Conditional execution is a case where GPUs do in runtime hardware what vector architectures do at compile time. Vector compilers do a double IF-conversion, generating four different masks. The execution is basically the same as GPUs, but there are some more overhead instructions executed for vectors. Vector architectures have the advantage of being integrated with a scalar processor, allowing them to avoid the time for the 0 cases when they dominate a calculation. Although it will depend on the speed of the scalar processor versus the vector processor, the crossover when it's better to use scalar might be when less than 20% of the mask bits are 1s. One optimization available at runtime for GPUs, but not at compile time for vector architectures, is to skip the THEN or ELSE parts when mask bits are all 0s or all 1s.

Thus, the efficiency with which GPUs execute conditional statements comes down to how frequently the branches will diverge. For example, one calculation of eigenvalues has deep conditional nesting, but measurements of the code show that around 82% of clock cycle issues have between 29 and 32 out of the 32 mask bits set to 1, so GPUs execute this code more efficiently than one might expect.

The same mechanism handles the strip-mining of vector loops—when the number of elements doesn't perfectly match the hardware. The example at the beginning of this section shows that an IF statement checks to see if this SIMD Lane element number (stored in `R8` in the preceding example) is less than the limit (`i < n`), and it sets masks appropriately. The key difference is that the vector compiler sets the mask register and generates the different loops—up to 2^n for n-depth nesting—versus the hardware tracking the masks and the nestings in GPUs. The run times of both approaches are similar.

NVIDIA GPU Memory Structures

Figure 4.19 shows the memory structures of an NVIDIA GPU. Each SIMD Lane in a multithreaded SIMD Processor is given a private section of off-chip DRAM, which we call the *private memory*. It is used for the stack frame, for spilling registers, and for private variables that don't fit in the registers. SIMD Lanes do *not* share private memories. GPUs cache this private memory in the L1 and L2 caches to aid register spilling and to speed up function calls.

We call the on-chip memory that is local to each multithreaded SIMD Processor *local memory*. It is a small scratchpad memory with low latency (a few dozen clocks) and high bandwidth (128 bytes/clock) where the programmer can store

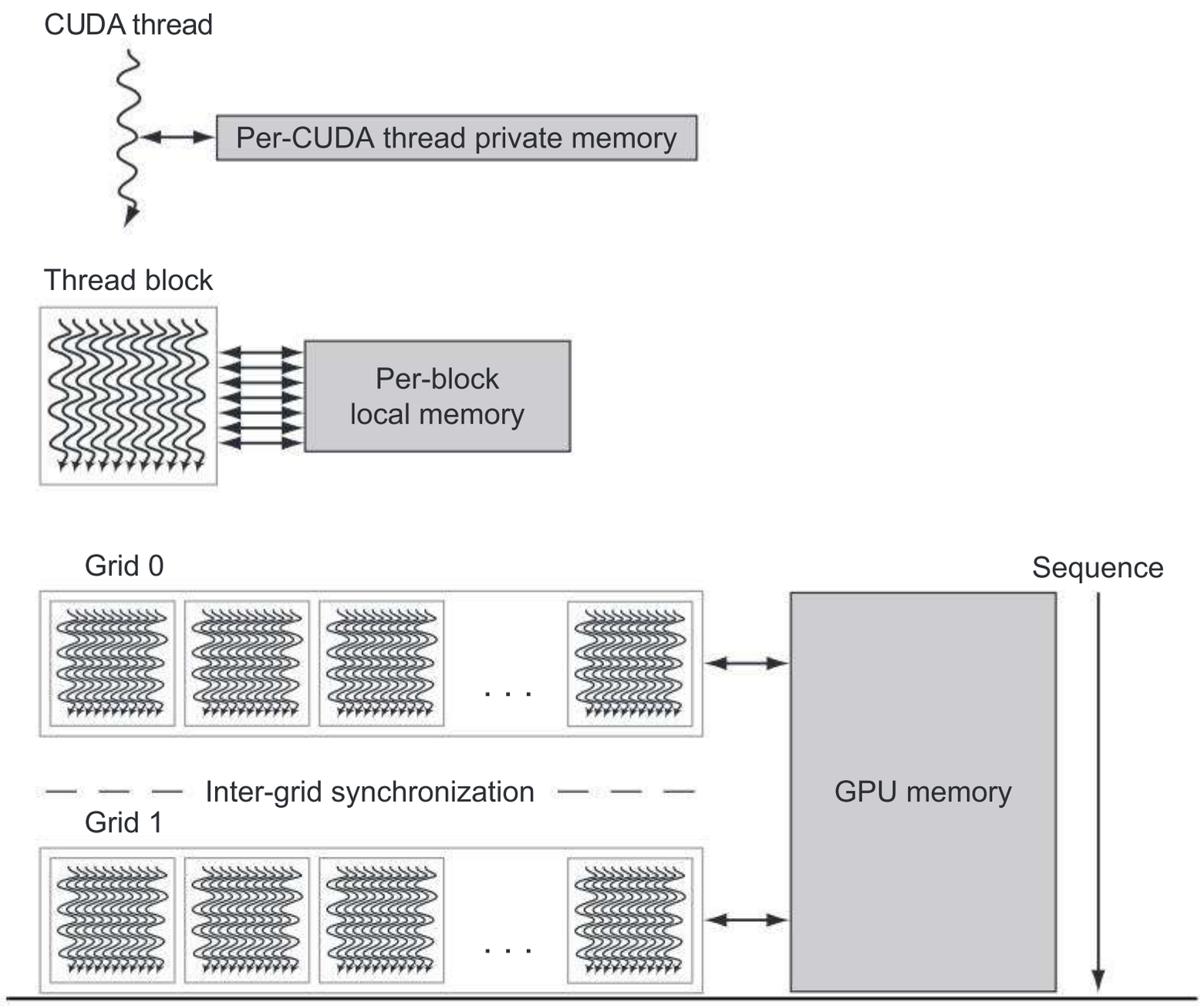

Figure 4.19 GPU memory structures. All Grids (vectorized loops) share GPU memory, local memory is shared by all threads of SIMD instructions within a Thread Block (body of a vectorized loop), and private memory is private to a single CUDA Thread. Pascal allows preemption of a Grid, which requires that all local and private memory be able to be saved in and restored from global memory. For the sake of completeness, 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.

data that needs to be reused, either by the same thread or another thread in the same Thread Block. Local memory is limited in size, typically to 48 KiB. It carries no state between Thread Blocks executed on the same processor. It is shared by the SIMD Lanes within a multithreaded SIMD Processor, but this memory is not shared between multithreaded SIMD Processors (hence our name, local memory). The multithreaded SIMD Processor dynamically allocates portions of the local memory to a Thread Block when it creates the Thread Block and frees the memory when all the threads of the Thread Block exit. That portion of local memory is private to that Thread Block.

Finally, we call the section of off-chip DRAM shared by the whole GPU and all Thread Blocks *GPU Memory*. Our vector multiply example used only GPU Memory.

The system processor, called the *host*, can read or write GPU Memory. Local memory is unavailable to the host, as it is restricted to each multithreaded SIMD Processor. Private memories are unavailable to the host as well.

Rather than rely on large caches to contain the whole working sets of an application, GPUs traditionally use smaller streaming caches and, because their working sets can be hundreds of megabytes, rely on extensive multithreading of threads of SIMD instructions to hide the long latency to DRAM. Given the use of multithreading to hide DRAM latency, the chip area used for large L2 and L3 caches in system processors is spent instead on computing resources and on the large number of registers to hold the state of many threads of SIMD instructions. In contrast, as mentioned, vector loads and stores amortize the latency across many elements because they pay the latency only once and then pipeline the rest of the accesses.

Although hiding memory latency behind many threads was the original philosophy of GPUs, all recent GPUs (and vector processors) have caches to reduce latency. The argument follows Little's law from queuing theory: the longer the latency, the more threads need to run during a memory access, which in turn requires more registers. Thus GPU caches are added to lower average latency and thereby reduce potential shortages of the number of registers. Note that GPU caches are designed assuming much lower hit rates than CPU caches, as GPUs often operate on data streaming directly from high bandwidth memory.

To improve memory bandwidth and reduce overhead, as mentioned, PTX data transfer instructions in cooperation with the memory controller coalesce individual parallel thread requests from the same SIMD Thread together into a single memory block request when the addresses fall in the same block. These restrictions are placed on the GPU program, somewhat analogous to the guidelines for system processor programs to engage hardware prefetching (see Chapter 2). The GPU memory controller will also hold requests and send them together to the same open page to improve DRAM bandwidth (see Section 4.6). (Chapter 2 describes DRAM in sufficient detail for readers to understand the potential benefits of grouping related addresses.)

Innovations in the Recent GPU Architectures

The real multithreaded SIMD Processor of GPUs is a bit more complicated than the simplified version in Figure 4.15. To increase hardware utilization, each SIMD Processor has two SIMD Thread Schedulers, each with multiple instruction dispatch units (some GPUs have even more thread schedulers). A dual SIMD Thread Scheduler selects two threads of SIMD instructions and issues one instruction from each to two sets of 16 SIMD Lanes, 16 load/store units, or 8 special function units. With multiple execution units available, two threads of SIMD instructions are scheduled each clock cycle, allowing 64 lanes to be active. Because the threads are independent, there is no need to check for data dependences in the instruction stream. This innovation would be analogous to a multithreaded vector

processor with double the number of vector lanes that can issue vector instructions from two independent threads. Figure 4.20 shows the Dual Scheduler issuing instructions, and Figure 4.21 shows the block diagram of the multithreaded SIMD Processor of a Pascal GP100 GPU.

Figure 4.22 shows that each new GPU generation typically adds some new features that increase performance or make it easier for programmers. For example, here are the four main innovations of the Pascal GPU:

- *Fast single-precision, double-precision, and half-precision floating-point arithmetic*—Pascal GP100 chip has significant floating-point performance in three sizes, all part of the IEEE standard for floating-point. The single-precision floating-point of the GPU runs at a peak of 10 TeraFLOP/s. Double-precision is roughly half-speed at 5 TeraFLOP/s, and half-precision is about double-speed at 20 TeraFLOP/s when expressed as 2-element vectors. The atomic memory operations include floating-point add for all three sizes. Pascal GP100 was the first GPU with such high performance for half-precision.
- *High-bandwidth memory*—The next innovation of the Pascal GP100 GPU is the use of stacked, high-bandwidth memory (*HBM2*). This memory has a wide bus with 4096 data wires running at 0.7 GHz, offering a peak bandwidth of 732 GB/s, which is more than twice as fast as previous GPUs. Subsequent GPUs use newer and faster versions of HBM.
- *High-speed chip-to-chip interconnect*—Given the coprocessor nature of GPUs, the PCI bus can be a communications bottleneck when trying to use multiple GPUs with one CPU. Pascal GP100 introduced the *NVLink* communications channel that supports data transfers of up to 20 GB/s in each direction. Each

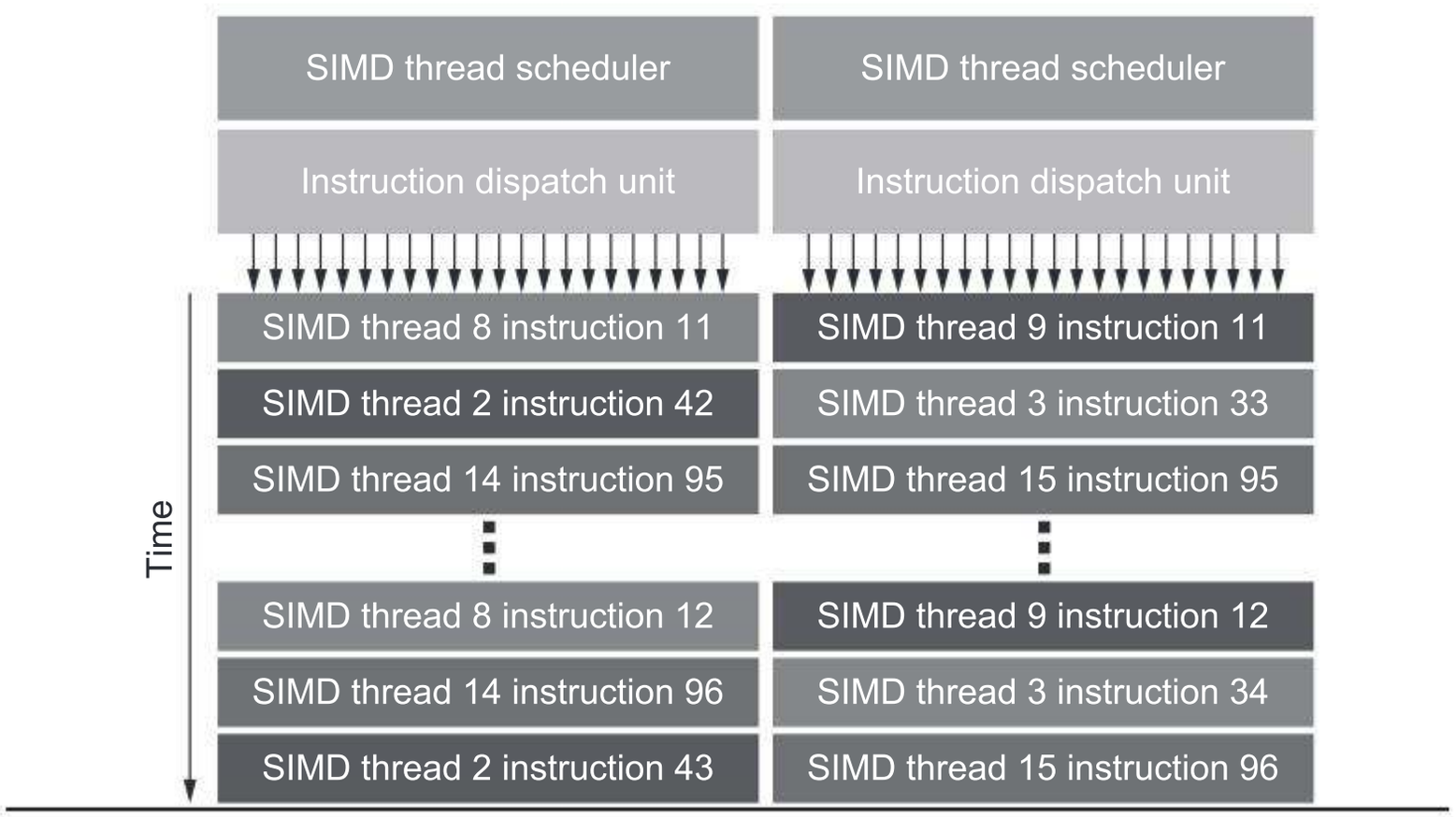

Figure 4.20 Block diagram of Pascal's dual SIMD Thread scheduler. Compare this design to the single SIMD Thread design in Figure 4.16.

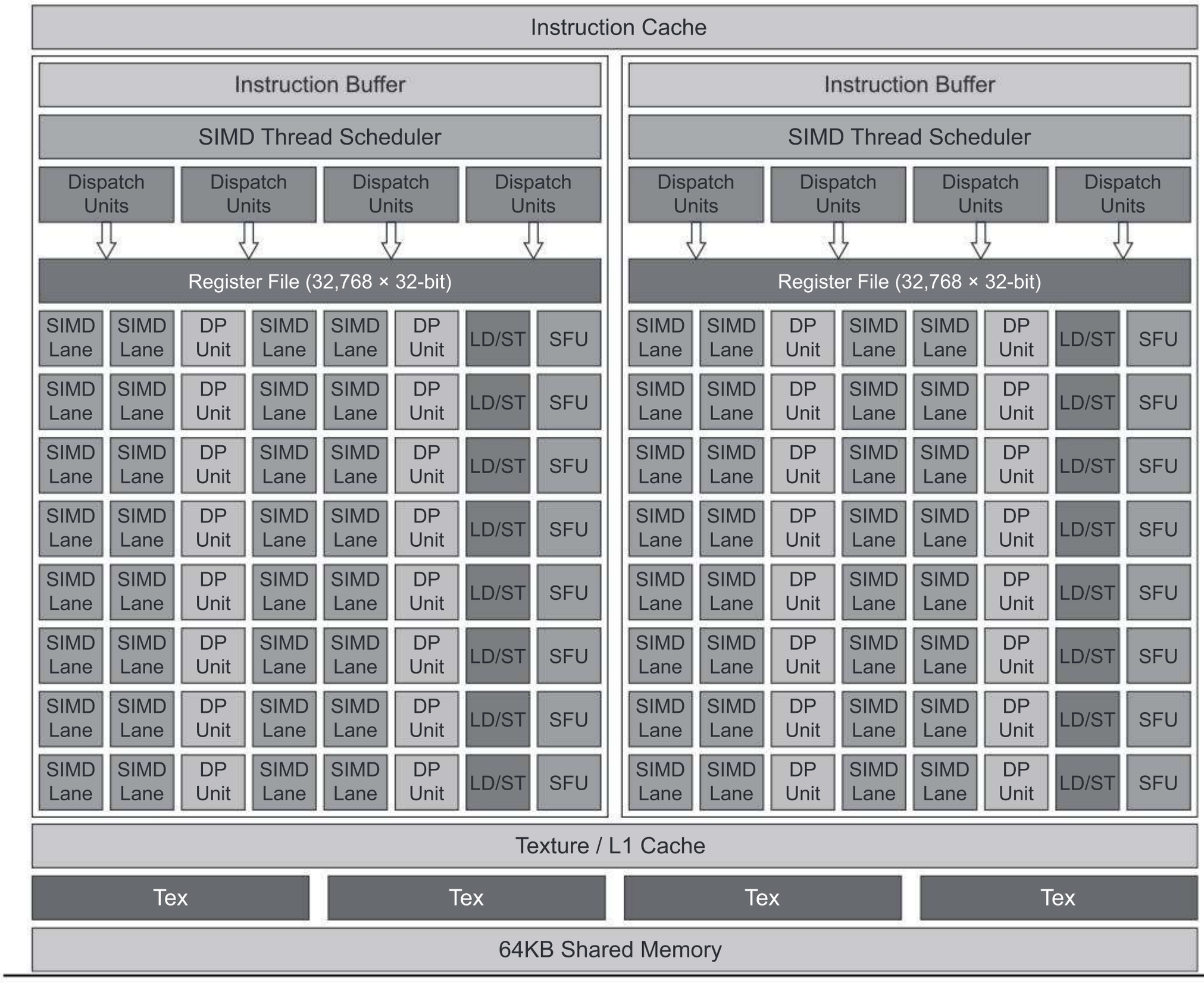

Figure 4.21 Block diagram of the multithreaded SIMD Processor of a Pascal GPU. Each of the 64 SIMD Lanes (cores) has a pipelined floating-point unit, a pipelined integer unit, some logic for dispatching instructions and operands to these units, and a queue for holding results. The 64 SIMD Lanes interact 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.

GP100 has 4 NVLink channels, providing a peak aggregate chip-to-chip bandwidth of 160 GB/s per chip. Systems with 2, 4, and 8 GPUs are available for multi-GPU applications, where each GPU can perform load, store, and atomic operations to any GPU connected by NVLink. Additionally, an NVLink channel can communicate with the CPU in some cases. For example, the IBM Power9 and later CPUs support CPU-GPU communication. In this chip NVLink provides a coherent view of memory between all GPUs and CPUs connected together. It also provides cache-to-cache communication instead of memory-to-memory communication.

GPU Model	P100	V100	A100	H100
Year Announced	2016	2017	2020	2022
Number of SMs	56	80	108	132
Base Clock frequency (GHz)	1.19	1.12	1.10	1.13
Turbo Mode Clock frequency (GHz)	1.48	1.53	1.41	1.98
Die size (mm2)	610	815	826	814
Technology (TSMC)	16 nm	12 nm FinFET+	7 nm N7	4N
Power (Watts)	300	300	400	700
Transistors (B)	15.3	21.1	54.2	80
Memory bandwidth (GB/s)	732	900	2000	3350
L2 Cache Size (MB)	4	6	40	50
Single-precision SIMD width	8	32	32	32
Double-precision SIMD width	4	32	32	32
Peak single-precision FLOPs/sec(GFLOP/s)	10,608	15,700	19,500	66,900
Peak double-precision FLOPs/sec(GFLOP/s)	5,304	7,800	9,700	33,500
Peak 16-bit FLOPs/sec (GFLOP/s for ML)	N.A.	125,000	312,000	989,000
Peak 8-bit FLOPs/sec (GFLOP/s for ML)	N.A.	N.A.	N.A.	1,979,000

Figure 4.22 Key parameters over four generations of NVIDIA GPUs. Chapter 7 describes the use of narrow floating point operations (8-bit and 16-bit) that are useful for ML that were introduced by more recent GPUs.

- *Unified virtual memory and paging support*—The Pascal GP100 GPU adds page-fault capabilities within a unified virtual address space. This feature allows a single virtual address for every data structure that is identical across all the GPUs and CPUs in a single system. When a thread accesses an address that is remote, a page of memory is transferred to the local GPU for subsequent use. Unified memory simplifies the programming model by providing demand paging instead of explicit memory copying between the CPU and GPU or between GPUs. It also allows allocating far more memory than exists on the GPU to solve problems with large memory requirements. As with any virtual memory system, care must be taken to avoid excessive page movement.

Figure 4.22 lists key parameters of four generations of GPUs, and Figure 4.23 displays them graphically relative to the Pascal P100 GPU. The biggest change is the increase in L2 cache size, followed by peak single- and double-precision FLOPs/second and the number of transistors. The A100 has modest increase in FLOPs/second compared to its increase in transistors because the focus was primarily on improving performance for machine learning—see the last two rows of Figure 4.22—which we discuss in Chapter 7. The large increase in FLOPs/second for H100 comes in part from using almost twice the power and from using 4 nm technology.

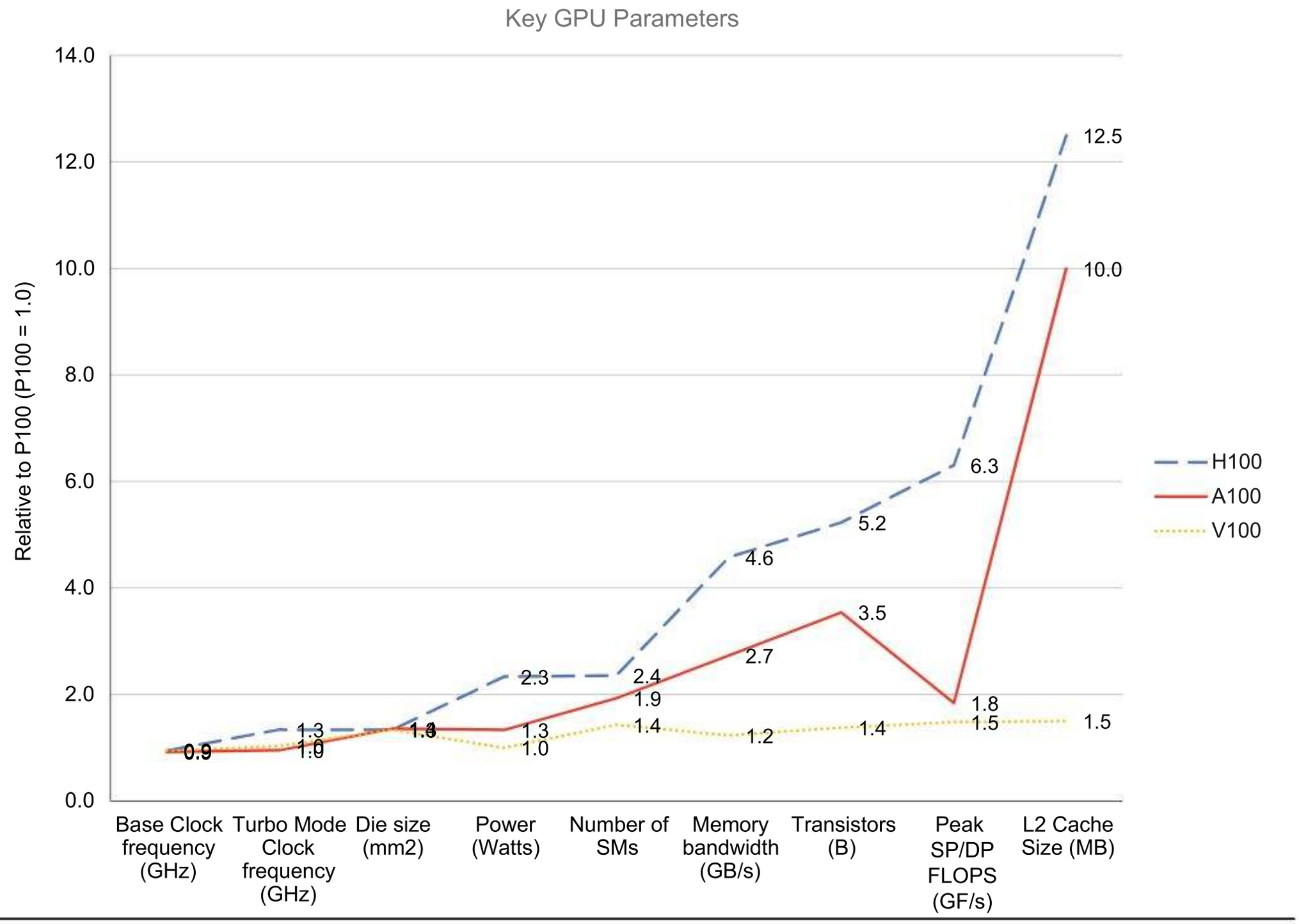

Figure 4.23 Graphical display of ratio of newer GPUs versus P100 for some parameters.

Similarities and Differences Between Vector Architectures and GPUs

As we have seen, there really are many similarities between vector architectures and GPUs. Along with the quirky jargon of GPUs, these similarities have contributed to the confusion in some architecture circles about how novel GPUs really are. Now that you've seen what is under the covers of vector computers and GPUs, you can appreciate both the similarities and the differences. Because both architectures are designed to execute data-level parallel programs but take different paths, this comparison is in depth to provide a better understanding of what is needed for DLP hardware. Figure 4.24 shows the vector term first and then the closest equivalent in a GPU.

A SIMD Processor is like a vector processor. The multiple SIMD Processors in GPUs act as independent MIMD cores, just as many vector computers have multiple vector processors. This view characterizes the NVIDIA H100 as a 132-core machine with hardware support for multithreading, where each core

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 4.24 GPU equivalent to vector terms.

has 64 lanes. The biggest difference is multithreading, which is fundamental to GPUs and missing from most vector processors.

Looking at the registers in the two architectures, the RV64V register file in our implementation holds entire vectors—that is, a contiguous block of elements. In contrast, a single vector in a GPU will be distributed across the registers of all SIMD Lanes. An RV64V processor has 32 vector registers with perhaps 32 elements, or 1024 elements total. A GPU thread of SIMD instructions has up to 256 registers with 32 elements each, or 8192 elements. These extra GPU registers support multithreading.

Figure 4.25 is a block diagram of the execution units of a vector processor on the left and a multithreaded SIMD Processor of a GPU on the right. For pedagogic purposes, we assume the vector processor has four lanes and the multithreaded SIMD Processor also has four SIMD Lanes. This figure shows that the four

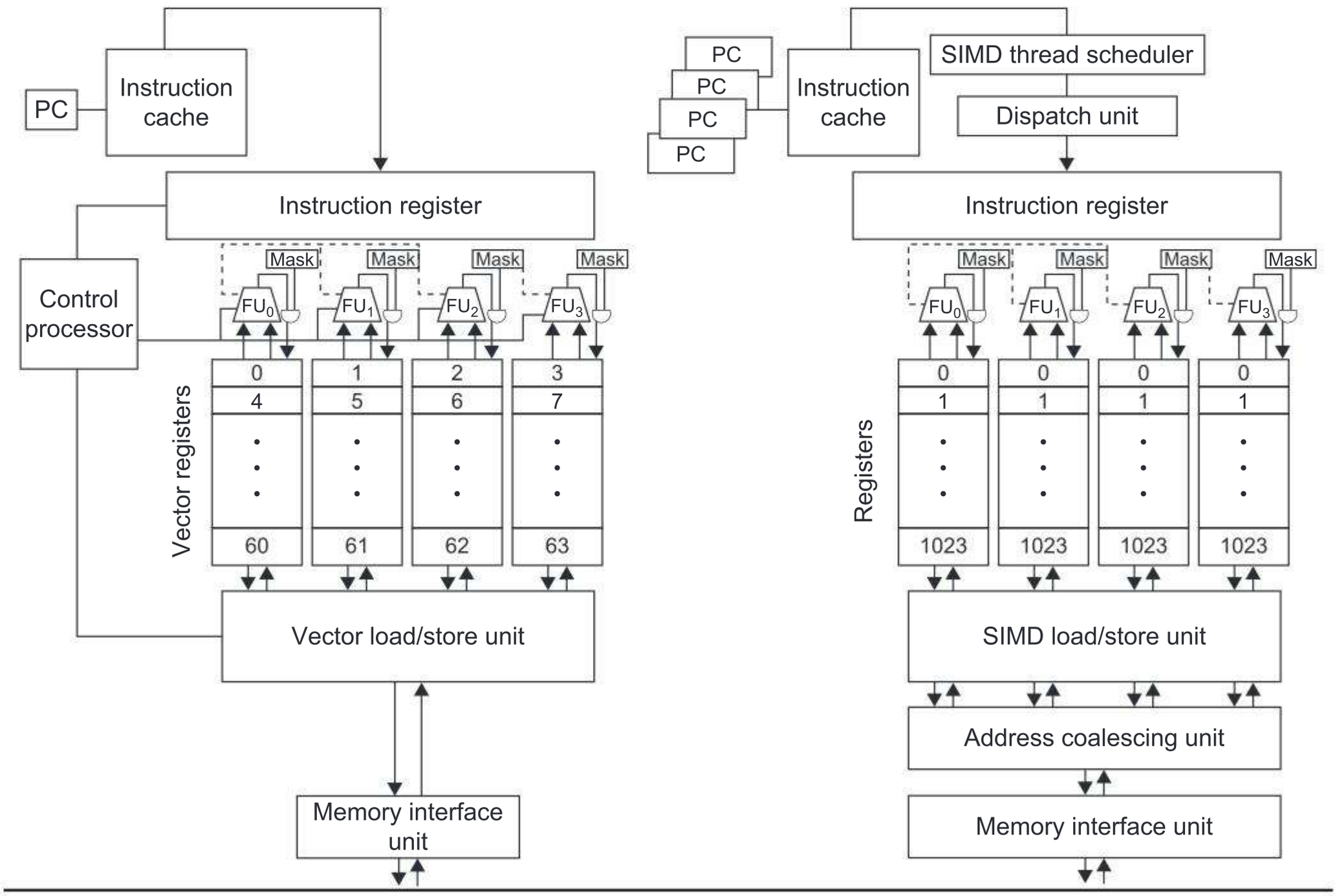

Figure 4.25 A vector processor with four lanes on the left and a multithreaded SIMD Processor of a GPU with four SIMD Lanes on the right. (GPUs typically have 16 or 32 SIMD Lanes.) The Control Processor supplies scalar operands for scalar-vector operations, increments addressing for unit and nonunit 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.

SIMD Lanes act in concert much like a four-lane vector unit, and that a SIMD Processor acts much like a vector processor.

In reality, there are many more lanes in GPUs, so GPU "chimes" are shorter. While a vector processor might have 2 to 8 lanes and a vector length of, say, 32—making a chime 4 to 16 clock cycles—a multithreaded SIMD Processor might have 8 or 16 lanes. A SIMD Thread is 32 elements wide, so a GPU chime would just be 2 or 4 clock cycles. This difference is why we use "SIMD Processor" as the more descriptive term because it is closer to a SIMD design than it is to a traditional vector processor design.

The closest GPU term to a vectorized loop is Grid, and a PTX instruction is the closest to a vector instruction because a SIMD Thread broadcasts a PTX instruction to all SIMD Lanes.

As for memory access instructions in the two architectures, all GPU loads are gather instructions and all GPU stores are scatter instructions. If data addresses of CUDA Threads refer to nearby addresses that fall into the same cache/memory block at the same time, the Address Coalescing Unit of the GPU will ensure high memory bandwidth. The *explicit* unit-stride load and store instructions of vector architectures versus the *implicit* unit stride of GPU programming is why writing efficient GPU code requires that programmers think in terms of SIMD operations, even though the CUDA programming model looks like MIMD. Because CUDA Threads can generate their own addresses—strided as well as gather-scatter—addressing vectors are found in both vector architectures and GPUs.

As we mentioned repeatedly, the two architectures take very different approaches to hiding memory latency. Vector architectures amortize it across all the elements of the vector by having a deeply pipelined access, so you pay the latency only once per vector load. Therefore vector loads and stores are like a block transfer between memory and the vector registers. In contrast, GPUs hide memory latency using multithreading. (Some researchers are investigating adding multithreading to vector architectures to try to capture the best of both worlds.)

Regarding conditional branch instructions, both architectures implement them using mask registers. Both conditional branch paths occupy time and/or space even when they do not store a result. The difference is that the vector compiler manages mask registers explicitly in software, while the GPU hardware and assembler manages them implicitly using branch synchronization markers and an internal stack to save, complement, and restore masks.

The Control Processor of a vector computer plays an important role in the execution of vector instructions. It broadcasts operations to all the Vector Lanes and broadcasts a scalar register value for vector-scalar operations. It also does implicit calculations that are explicit in GPUs, such as automatically incrementing memory addresses for unit-stride and nonunit-stride loads and stores. The Control Processor is missing in the GPU. The closest analogy is the Thread Block Scheduler, which assigns Thread Blocks (bodies of vector loop) to multithreaded SIMD Processors. The runtime hardware mechanisms in a GPU that both generate addresses and then discover if they are adjacent, which is commonplace in many

DLP applications, are likely less power-efficient than using a Control Processor, especially for the common case of unit stride accesses.

The scalar processor in a vector computer executes the scalar instructions of a vector program; that is, it performs operations that would be too slow to do in the vector unit. Although the system processor that is associated with a GPU is the closest analogy to a scalar processor in a vector architecture, the separate address spaces plus transferring over a PCIe bus means thousands of clock cycles of overhead to use them together. The scalar processor can be slower than a vector processor for floating-point computations in a vector computer, but not by the same ratio as the system processor versus a multithreaded SIMD Processor (given the overhead).

Therefore each "vector unit" in a GPU must do computations that you would expect to do using a scalar processor in a vector computer. That is, rather than calculate on the system processor and communicate the results, it can be faster to disable all but one SIMD Lane using the predicate registers and built-in masks and do the scalar work with one SIMD Lane. The relatively simple scalar processor in a vector computer is likely to be faster and more power-efficient than the GPU solution. If system processors and GPUs become more closely tied together in the future, such as in the NVIDIA Grace processor, it will be interesting to see if system processors can play the same role as scalar processors do for vector and multimedia SIMD architectures.

Similarities and Differences Between Multimedia SIMD Computers and GPUs

At a high level, multicore computers with multimedia SIMD instruction extensions do share similarities with GPUs. Figure 4.26 summarizes the similarities and differences.

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

Both are multiprocessors whose processors use multiple SIMD Lanes, although GPUs have more processors and many more lanes. Both use hardware multithreading to improve processor utilization, although GPUs have hardware support for many more threads. Both have roughly 2:1 performance ratios between peak performance of single-precision and double-precision floating-point arithmetic. Both use caches, although GPUs use smaller streaming caches, and multicore computers use large multilevel caches that try to contain whole working sets completely. Both use a 64-bit address space, although the physical main memory is much smaller in GPUs. Both support memory protection at the page level and demand paging, which allows them to address far more memory than they have on board.

Besides the large numerical differences in processors, SIMD Lanes, hardware thread support, and cache sizes, there are many architectural differences. The scalar processor and multimedia SIMD instructions are tightly integrated in traditional computers; they are separated by an I/O bus in GPUs, and they even have separate main memories. The multiple SIMD Processors in a GPU use a single address space and can support a coherent view of all memory on some systems given support from CPU vendors (such as the IBM Power9). Unlike GPUs, multimedia SIMD instructions historically did not support gather-scatter memory accesses, which Section 4.7 shows is a significant omission.

A key for vector architectures is getting reports from the compiler saying what loops vectorized and using that feedback to drive changes to the code. GPU compilers say almost nothing about how efficiently one's code will run. Instead, programmers must use performance analysis tools (e.g., NVIDIA Parallel Nsight) to get feedback on dynamic execution behavior and thus what could be done to improve the code.

CUDA/SIMT succeeded better than some thought because CUDA/SIMT divides the work of exploiting parallelism up between programmer and system in a way that leaves the easy but tedious optimizations to the system but makes the programmer focus on the hard problem of finding sufficient outer loop parallelism. This division of labor might encourage the programmer to look for an entirely new algorithm.

When CUDA/SIMT first arrived, there was a fair amount of skepticism from many computer architects because they assumed the program or algorithm was held (relatively) constant, and the job was to optimize the rest of the system for that constant input. From this perspective, it seemed unlikely that many developers would invest time to rewrite large bodies of code, so GPU computing would remain a niche within computing. The focus on standard benchmarks written for existing platforms made it difficult for architects to imagine the arrival of new economically important applications being written from scratch primarily for new architecture different than CPUs.

Fortunately for GPUs, the machine learning application is not based on thousands of programmers writing millions of lines of code, which makes writing a new application from scratch more feasible. The cleverness comes from a

relatively small program that learns from data rather than trying to code cleverness as a huge program. Valuable DNNs can come from small kernels written in standard Python frameworks (JAX, PyTorch, TensorFlow) that may train for weeks by analyzing huge amounts of data on thousands of GPUs.

Summary

Now that the veil has been lifted, we can see that GPUs are really just multithreaded SIMD Processors, although they have more processors, more lanes per processor, and more multithreading hardware than do traditional multicore computers. For example, the Hopper H100 GPU has 132 SIMD Processors with 64 lanes per processor and hardware support for 64 SIMD Threads. The Pascal GPU embraces instruction-level parallelism by issuing instructions from two SIMD Threads to two sets of SIMD Lanes. GPUs also have less cache memory—the L2 cache is 50 MiB in Hopper—and it can be coherent with a cooperative distant scalar processor or distant GPUs.

The CUDA programming model wraps up all these forms of parallelism around a single abstraction, the CUDA Thread. The CUDA programmer can think of programming thousands of threads, although they are really executing each block of 32 threads on the many lanes of the many SIMD Processors. The CUDA programmer who wants good performance keeps in mind that these threads are organized in blocks and executed 32 at a time and that addresses need to be to adjacent addresses to get good performance from the memory system.

Although we've used CUDA and the NVIDIA GPU in this section, rest assured that the same ideas are found in the OpenCL programming language and in GPUs from other companies.

Now that you understand better how GPUs work, we reveal the real jargon. Figures 4.27 and 4.28 match the descriptive terms and definitions of this section with the official CUDA/NVIDIA and AMD terms and definitions. We also include the OpenCL terms. We believe the GPU learning curve is steep in part because of using terms such as "streaming multiprocessor" for the SIMD Processor, "thread processor" for the SIMD Lane, and "shared memory" for local memory—especially because local memory is *not* shared between SIMD Processors! We hope that this two-step approach gets you up that curve quicker, even if it's a bit indirect.

4.5 Detecting and Enhancing Loop-Level Parallelism

Loops in programs are the fountainhead of many of the types of parallelism we previously discussed here and in Chapter 5. In this section, we discuss compiler technology used for discovering the amount of parallelism for vector and SIMD architectures that we can exploit in a program and hardware support for these compiler techniques. (As discussed above, GPUs leave this optimization to the programmer rather than to the CUDA compiler.) We define precisely when a

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

loop is parallel (or vectorizable), how a dependence can prevent a loop from being parallel, and techniques for eliminating some types of dependences. Finding and manipulating loop-level parallelism is critical to exploiting both DLP and TLP, as well as the more aggressive static ILP approaches (e.g., VLIW) that we examine in Appendix H.

Loop-level parallelism is normally investigated at the source level or close to it, while most analysis of ILP is done once instructions have been generated by the compiler. Loop-level analysis involves determining what dependences exist among the operands in a loop across the iterations of that loop. For now, we will consider only data dependences, which arise when an operand is written at some point and read at a later point. Name dependences also exist and may be removed by the renaming techniques discussed in Chapter 3.

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 4.28 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.

The analysis of loop-level parallelism focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations; such dependence is called a *loop-carried dependence*. Most of the examples we considered in Chapters 2 and 3 had no loop-carried dependences and thus are loop-level parallel. To see that a loop is parallel, let us first look at the source representation:

```
for (i=999; i>=0; i=i-1)
      x[i] = x[i] + s;
```

In this loop the two uses of `x[i]` are dependent, but this dependence is within a single iteration and is not loop carried. There is a loop-carried dependence between successive uses of `i` in different iterations, but this dependence involves an induction variable that can be easily recognized and eliminated. We saw examples of how to eliminate dependences involving induction variables during loop unrolling in Section 2.2 of Chapter 2, and we will look at additional examples later in this section.

Because finding loop-level parallelism involves recognizing structures such as loops, array references, and induction variable computations, a compiler can do this analysis more easily at or near the source level, in contrast to the machine-code level. Let's look at a more complex example.

Example Consider a loop like this one:

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

Assume that A, B, and C are distinct, nonoverlapping arrays. (In practice, the arrays may sometimes be the same or may overlap. Because the arrays may be passed as parameters to a procedure that includes this loop, determining whether arrays overlap or are identical often requires sophisticated, interprocedural analysis of the program.) What are the data dependences among the statements `S1` and `S2` in the loop?

Answer

1. There are two different dependences: `S1` uses a value computed by `S1` in an earlier iteration, because iteration `i` computes `A[i + 1]`, which is read in iteration `i + 1`. The same is true of `S2` for `B[i]` and `B[i + 1]`.
2. `S2` uses the value `A[i + 1]` computed by `S1` in the same iteration.

These two dependences are distinct and have different effects. To see how they differ, let's assume that only one of these dependences exists at a time. Because the dependence of statement `S1` is on an earlier iteration of `S1`, this dependence is loop carried. It forces successive iterations of this loop to execute in series.

The second dependence (S2 depending on S1) is within an iteration and is not loop carried. Thus, if this were the only dependence, multiple iterations of the loop would execute in parallel, as long as each pair of statements in an iteration were kept in order. We saw this type of dependence in an example in Section 2.2, where unrolling could expose the parallelism. These intraloop dependences are common. For example, a sequence of vector instructions that uses chaining exhibits exactly this sort of dependence.

It is also possible to have a loop-carried dependence that does not prevent parallelism, as the next example shows.

Example Consider a loop like this one:

What are the dependences between S1 and S2? Is this loop parallel? If not, show how to make it parallel.

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

Answer Statement S1 uses the value assigned in the previous iteration by statement S2, so there is a loop-carried dependence between S2 and S1. Despite this loop-carried dependence, this loop can be made parallel. Unlike the earlier loop, this dependence is not circular; neither statement depends on itself, and although S1 depends on S2, S2 does not depend on S1. A loop is parallel if it can be written without a cycle in the dependences because the absence of a cycle means that the dependences give a partial ordering on the statements.

Although there are no circular dependences in the preceding loop, it must be transformed to conform to the partial ordering and expose the parallelism. Two observations are critical to this transformation:

1. There is no dependence from S1 to S2. If there were, then there would be a cycle in the dependences and the loop would not be parallel. Because this other dependence is absent, interchanging the two statements will not affect the execution of S2.
2. On the first iteration of the loop, statement S2 depends on the value of B[0] computed *prior* to initiating the loop.

These two observations allow us to replace the preceding loop with the following code sequence:

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

The dependence between the two statements is no longer loop carried so that iterations of the loop may be overlapped, provided the statements in each iteration are kept in order.

Our analysis needs to begin by finding all loop-carried dependences. This dependence information is *inexact*, in the sense that it tells us that such dependence *may* exist. Consider the following example:

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

The second reference to A in this example need not be translated to a load instruction because we know that the value is computed and stored by the previous statement. Thus the second reference to A can simply be a reference to the register into which A was computed. Performing this optimization requires knowing that the two references are *always* to the same memory address and that there is no intervening access to the same location. Normally, data dependence analysis tells that only one reference *may* depend on another; a more complex analysis is required to determine that two references *must be* to the exact same address. In the preceding example, a simple version of this analysis suffices because the two references are in the same basic block.

Often loop-carried dependences are in the form of a *recurrence*. A recurrence occurs when a variable is defined based on the value of that variable in an earlier iteration, usually the one immediately preceding, as in the following code fragment:

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

Detecting a recurrence can be important for two reasons: some architectures (especially vector computers) have special support for executing recurrences, and in an ILP context it may still be possible to exploit a fair amount of parallelism.

Finding Dependences

Clearly, finding the dependences in a program is important both to determine which loops might contain parallelism and to eliminate name dependences. The complexity of dependence analysis arises also because of the presence of arrays and pointers in languages such as C or C++, or pass-by-reference parameter passing in Fortran. Because scalar variable references explicitly refer to a name, they can usually be analyzed quite easily with aliasing because of pointers and reference parameters causing few complications and uncertainty in the analysis.

How does the compiler detect dependences in general? Nearly all dependence analysis algorithms work on the assumption that array indices are *affine*. In simplest terms, a one-dimensional array index is affine if it can be written in the form $a \times i + b$, where a and b are constants and i is the loop index variable. The index of a multidimensional array is affine if the index in each dimension is affine. Sparse array accesses, which typically have the form `x[y[i]]`, are one of the major examples of nonaffine accesses.

Determining whether there is a dependence between two references to the same array in a loop is thus equivalent to determining whether two affine functions can have identical value for different indices between the bounds of the loop. For example, suppose we have stored to an array element with index value $a \times i + b$ and loaded from the same array with index value $c \times i + d$, where i is the for-loop index variable that runs from m to n. A dependence exists if two conditions hold:

1. There are two iteration indices, j and k, that are both within the limits of the for-loop. That is, $m \leq j \leq n$, $m \leq k \leq n$.
2. The loop stores into an array element indexed by $a \times j + b$ and later fetches from that *same* array element when it is indexed by $c \times k + d$, that is, $a \times j + b = c \times k + d$.

In general, we cannot determine whether dependence exists at compile time. For example, the values of a, b, c, and d may not be known (they could be values in other arrays), making it impossible to tell if a dependence exists. In other cases the dependence testing may be very expensive but decidable at compile time. For example, the accesses may depend on the iteration indices of multiple nested loops. Many programs, however, contain primarily simple indices where a, b, c, and d are all constants. For these cases, it is possible to devise reasonable compile time tests for dependence.

As an example, a simple and sufficient test for the absence of a dependence is the *greatest common divisor* (*GCD*) test. It is based on the observation that if a loop-carried dependence exists, then GCD (c, a) must divide ($d - b$). (Recall that an integer, x, *divides* another integer, y, if we get an integer quotient when we do the division y/x and there is no remainder.)

Example Use the GCD test to determine whether dependences exist in the following loop:

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

Answer Given the values $a = 2$, $b = 3$, $c = 2$, and $d = 0$, then GCD(a, c) = 2, and $d - b = -3$. Because 2 does not divide -3, no dependence is possible.

The GCD test is sufficient to guarantee that no dependence exists. However, there are cases where the GCD test succeeds but no dependence exists. This can arise, for example, because the GCD test does not consider the loop bounds.

In general, determining whether a dependence actually exists is NP-complete. In practice, however, many common cases can be analyzed precisely at low cost. Recently, approaches using a hierarchy of exact tests increasing in generality and cost have been shown to be both accurate and efficient. (A test is *exact* if it precisely determines whether a dependence exists. Although the general case is NP-complete, there exist exact tests for restricted situations that are much cheaper.)

Besides detecting the presence of a dependence, a compiler wants to classify the type of dependence. This classification allows a compiler to recognize name dependences and eliminate them at compile time by renaming and copying.

Example The following loop has multiple types of dependences. Find all the true dependences, output dependences, and antidependences, and eliminate the output dependences and antidependences by renaming.

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

Answer

1. The following dependences exist among the four statements: There are true dependences from `S1` to `S3` and from `S1` to `S4` because of Y[i]. These are not loop carried, so they do not prevent the loop from being considered parallel. These dependences will force `S3` and `S4` to wait for S1 to complete.
2. There is an antidependence from `S1` to `S2`, based on X[i].
3. There is an antidependence from `S3` to `S4` for Y[i].
4. There is an output dependence from `S1` to `S4`, based on Y[i].

The following version of the loop eliminates these false (or pseudo) dependences.

```
for (i=0; i<100; i=i+1 {
      T[i] = X[i] / c; /* Y renamed to T to remove output dependence */
      X1[i] = X[i] + c;/* X renamed to X1 to remove antidependence */
      Z[i] = T[i] + c;/* Y renamed to T to remove antidependence */
      Y[i] = c - T[i];
}
```

After the loop, the variable X has been renamed X1. In code that follows the loop the compiler can simply replace the name X by X1. In this case renaming does not require an actual copy operation, as it can be done by substituting names or by register allocation. In other cases, however, renaming will require copying.

Dependence analysis is a critical technology for exploiting parallelism, as well as for the transformation-like blocking that Chapter 2 covers. For detecting loop-level parallelism, dependence analysis is the basic tool. Effectively compiling programs for vector computers, SIMD computers, or multiprocessors depends critically on this analysis. The major drawback of dependence analysis is that it applies only under a limited set of circumstances, namely, among references within a single loop nest and using affine index functions. Thus there are many situations where array-oriented dependence analysis *cannot* tell us what we want to know. For example, analyzing accesses done with pointers, rather than with array indices can be much harder. (This is one reason why Fortran is still preferred over C and C++ for many scientific applications designed for parallel computers.) Similarly, analyzing references across procedure calls is extremely difficult. Thus, while analysis of code written in sequential languages remains important, we also need approaches such as OpenMP and CUDA that write explicitly parallel loops.

Eliminating Dependent Computations

As previously mentioned, one of the most important forms of dependent computations is a recurrence. A dot product is a perfect example of a recurrence:

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

This loop is not parallel because it has a loop-carried dependence on the variable `sum`. We can, however, transform it to a set of loops, one of which is completely parallel and the other partly parallel. The first loop will execute the completely parallel portion of this loop. It looks like this:

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

Notice that sum has been expanded from a scalar into a vector quantity (a transformation called *scalar expansion*) and that this transformation makes this new loop completely parallel. When we are done, however, we need to do the reduce step, which sums up the elements of the vector. It looks like this:

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

Although this loop is not parallel, it has a very specific structure called a *reduction*. Reductions are common in linear algebra, and as we will see in Chapter 6, they are also a key part of the primary parallelism primitive MapReduce used in warehouse-scale computers. In general, any function can be used as a reduction operator, and common cases include operators such as max and min.

Reductions are sometimes handled by special hardware in a vector and SIMD architecture that allows the reduce step to be done much faster than it could be

done in scalar mode. These accelerators work by implementing a technique similar to what can be done in a multiprocessor environment. While the general transformation works with any number of processors, suppose for simplicity we have 10 processors. In the first step of reducing the sum each processor executes the following (with `p` as the processor number ranging from 0 to 9):

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

This loop, which sums up 1000 elements on each of the 10 processors, is completely parallel. A simple scalar loop can then complete the summation of the last 10 sums. Similar approaches are used in vector processors and SIMD Processors.

It is important to observe that the preceding transformation relies on associativity of addition. Although arithmetic with unlimited range and precision is associative, computer arithmetic is not associative, for either integer arithmetic, because of limited range, or floating-point arithmetic, because of both range and precision. Thus using these restructuring techniques can sometimes lead to erroneous behavior, although such occurrences are rare. For this reason, most compilers require that optimizations that rely on associativity be explicitly enabled.

4.6 Cross-Cutting Issues

Energy and DLP: Slow and Wide Versus Fast and Narrow

A fundamental power advantage of data-level parallel architectures comes from the energy equation in Chapter 1. Assuming ample DLP, the performance is the same if we halve the clock rate and double the execution resources: twice the number of lanes for a vector computer, wider registers and ALUs for multimedia SIMD, and more SIMD Lanes for GPUs. If we can lower the voltage while dropping the clock rate, we can actually reduce energy and the power for the computation while maintaining the same peak performance. Thus GPUs tend to have lower clock rates than system processors, which rely on high clock rates for performance (see Section 4.7).

Compared to out-of-order processors, DLP processors can have simpler control logic to launch a large number of operations per clock cycle; for example, the control is identical for all lanes in vector processors, and there is no logic to decide on multiple instruction issues or speculative execution logic. They also fetch and decode far fewer instructions. Vector architectures can also make it easier to turn off unused portions of the chip. Each vector instruction explicitly describes all the resources it needs for a number of cycles when the instruction issues.

Banked Memory and High Bandwidth Memory

Section 4.2 noted the importance of substantial memory bandwidth for vector architectures to support unit stride, nonunit stride, and gather-scatter accesses.

To achieve the highest memory performance, stacked DRAMs are used in the top-end GPUs from AMD and NVIDIA. Intel also uses stacked DRAM in its Xeon Phi product. Also known as *high bandwidth memory* (*HBM*, *HBM2, HBM3, HBM4*), the memory chips are stacked and stacks are placed in the same socket next to the processing chip. The extensive width (typically 4096−5120 data wires) provides high bandwidth while placing the memory chips in the same package as the processor chip reduces latency and power consumption. The capacity of stacked DRAM is typically 8−32 GB.

Given all the potential demands on the memory from both the computation tasks and the graphics acceleration tasks, the memory system could see a large number of uncorrelated requests. Unfortunately, this diversity hurts memory performance. To cope, the GPU's memory controller maintains separate queues of traffic bound for different banks, waiting until there is enough traffic to justify opening a row and transferring all requested data at once. This delay improves bandwidth but stretches latency, and the controller must ensure that no processing units starve while waiting for data, for otherwise neighboring processors could become idle. Section 4.7 shows that gather-scatter techniques and memory bank−aware access techniques can deliver substantial increases in performance versus conventional cache-based architectures.

Strided Accesses and TLB Misses

One problem with strided accesses is how they interact with the translation lookaside buffer (TLB) for virtual memory in vector architectures or GPUs. (GPUs also use TLBs for memory mapping.) Depending on how the TLB is organized and the size of the array being accessed in memory, it is even possible to get one TLB miss for every access to an element in the array! The same type of collision can happen with caches, but the performance impact is probably less.

4.7 Putting It All Together: Embedded Versus Server GPUs and Tesla Versus Core i7

Given the popularity of graphics applications, GPUs are now found in mobile clients, laptops, heavy-duty desktop computers, and traditional servers. Figure 4.29 lists the key characteristics of the Intel Xe GPU system on a chip for tablet and laptop clients, and the NVIDIA Hopper GPU for servers.

The Iris Xe ICX is the low-power member of a family of GPU microarchitectures from Intel. Its building block is the Execution Unit (EU), which is a seven-way multithreaded SIMD processor. Each thread has 128 registers that are 32 bytes wide. The primary computation unit is an eight-wide SIMD unit for floating

GPU Model	Intel Xe GPU in Core i7-1195G7	NVIDIA Hopper H100
Year Announced	2021	2022
Number of cores	96	132
Base Clock frequency (GHz)	0.30	1.13
Turbo Mode Clock frequency (GHz)	1.40	1.98
Die size (mm2)	N.A.	814
Technology	Intel 10 nm SF	TSMC 4N
Power (Watts)	up to 28 (including CPU, caches, ...)	700
Transistors (B)	N.A.	80
Memory technology	LPDDR4x-4267	HBM3
Memory capacity (GB)	32	80
Memory bandwidth (GB/s)	68	3350
On-chip memory (MB)	12	50
Single-precision SIMD width	8	32
Double-precision SIMD width	N.A.	32
Peak single-precision FLOPS (GFLOP/s)	N.A.	66,900
Peak double-precision FLOPS (GFLOP/s)	N.A.	33,500
Peak 16-bit FLOPS (GFLOP/s for ML)	4,300	989,000

Figure 4.29 Key features of the GPUs for embedded clients and servers.

point and integer computation and the secondary unit is a two-wide SIMD unit for extended math operations. Each EU can execute sixteen 16-bit and eight 32-bit floating point operations per clock cycle. The instance in Figure 4.29 has 96 EUs with 12 MB of L3 cache and 128 byte data transfer per clock cycle to DRAM.

The NVIDIA Hopper H100 turbo clock rate is 2.0 GHz, and it includes 132 SIMD Processors. The path to HBM3 memory is 5120 bits wide, and it transfers data on both the rising and falling edge of a 2.62 GHz clock, which means a peak memory bandwidth of 3330 GB/s.

All physical characteristics of the H100 die are impressively large. It contains 80 billion transistors, the die size is 814 mm^2 in a 4-nm TSMC process, and the typical power is 700 W.

Comparison of a GPU and a MIMD With Multimedia SIMD

A group of Intel researchers published a paper (Lee et al., 2010) comparing a quad-core Intel i7 with multimedia SIMD extensions to the Tesla GTX 280. Although the study did not compare the latest versions of CPUs and GPUs, it was the most in-depth comparison of the two styles in that it explained the reasons behind the differences in performance. Moreover, the current versions of these architectures share many similarities to the ones in the study.

Figure 4.30 lists the characteristics of both systems. Both products were purchased in fall of 2009. The Core i7 is in Intel's 45-nanometer semiconductor technology, while the GPU is in TSMC's 65-nanometer technology. Although it might have been fairer to have a comparison done by a neutral party or by both interested parties, the purpose of this section is *not* to determine how much faster one product is than the other but to try to understand the relative value of features of these two contrasting architecture styles.

The rooflines of the Core i7 920 and GTX 280 in Figure 4.31 illustrate the differences in the computers. The 920 has a slower clock rate than the 960 (2.66

	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 4.30 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/second 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.

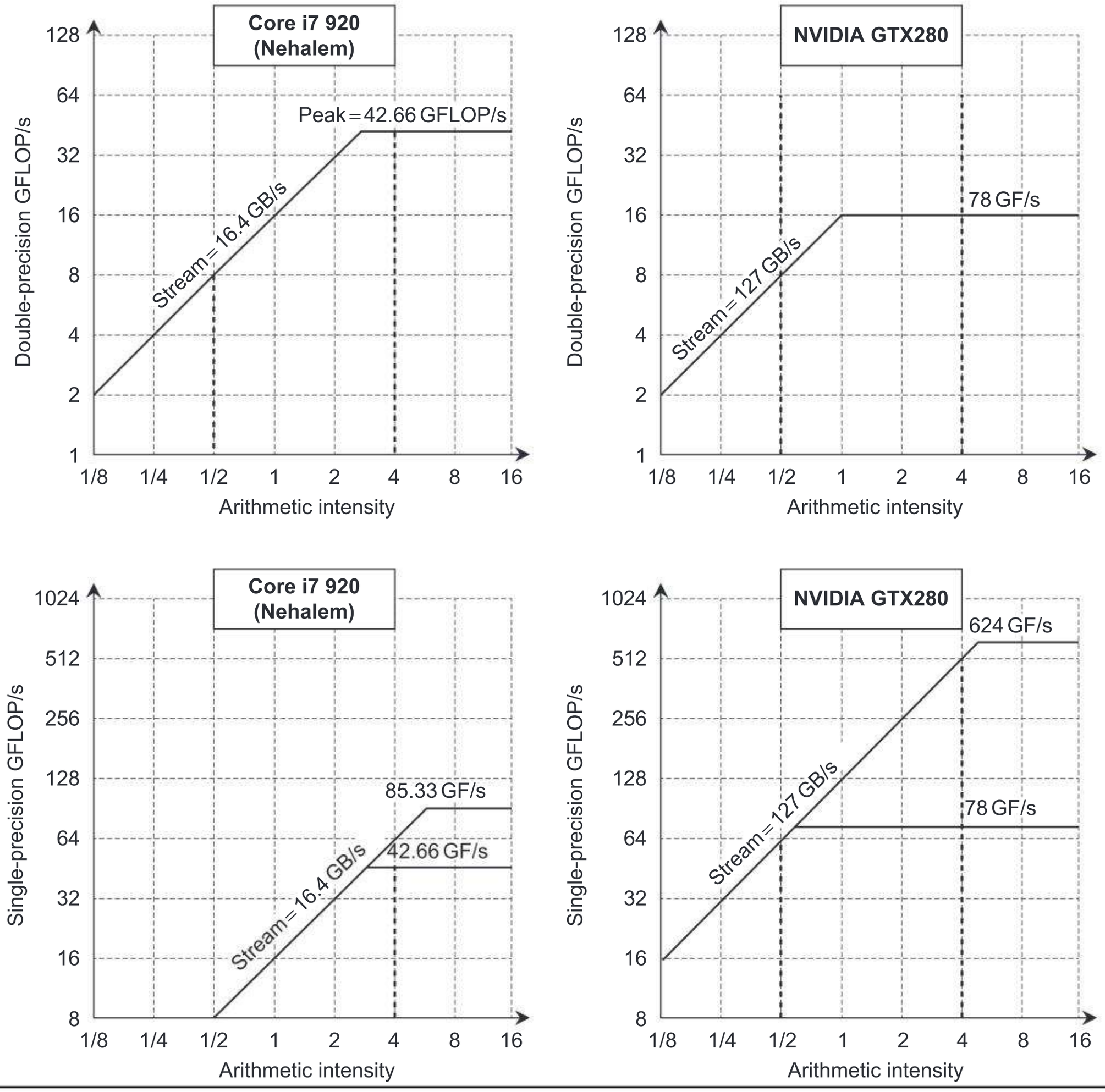

Figure 4.31 Roofline model (Williams et al. 2009). These rooflines show double-precision floating-point (DP FP) performance in the top row and single-precision performance in the bottom row. (The DP FP performance ceiling is also in the bottom row to give perspective.) The Core i7 920 on the left has a peak DP FP performance of 42.66 GFLOP/s, an single-precision floating-point (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, an 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.

GHz vs. 3.2 GHz), but the rest of the system is the same. Not only does the GTX 280 have much higher memory bandwidth and double-precision floating-point performance, but also its double-precision ridge point is considerably to the left. As previously mentioned, it is much easier to hit peak computational performance the further the ridge point of the roofline is to the left. The double-precision ridge point is 0.6 for the GTX 280 versus 2.6 for the Core i7. For single-precision performance, the ridge point moves far to the right, as it's considerably harder to hit the roof of single-precision performance because it is so much higher. Note that the arithmetic intensity of the kernel is based on the bytes that go to main memory, not the bytes that go to cache memory. Thus caching can change the arithmetic intensity of a kernel on a particular computer, presuming that most references really go to the cache. The Rooflines help explain the relative performance in this case study. Note also that this bandwidth is for unit-stride accesses in both architectures. Real gather-scatter addresses that are not coalesced are slower on the GTX 280 and on the Core i7, as we will see.

The researchers said that they selected the benchmark programs by analyzing the computational and memory characteristics of four recently proposed benchmark suites and then "formulated the set of *throughput computing kernels* that capture these characteristics." Figure 4.32 describes these 14 kernels, and Figure 4.33 shows the performance results, with larger numbers meaning faster.

Given that the raw performance specifications of the GTX 280 vary from 2.5× slower (clock rate) to 7.5× faster (cores per chip) while the performance varies from 2.0× slower (Solv) to 15.2× faster (GJK), the Intel researchers explored the reasons for the differences:

- *Memory bandwidth.* The GPU has 4.4× the memory bandwidth, which helps explain why LBM and SAXPY run 5.0× and 5.3× faster; their working sets are hundreds of megabytes and thus don't fit into the Core i7 cache. (To access memory intensively, they did not use cache blocking on SAXPY.) Thus the slope of the rooflines explains their performance. SpMV also has a large working set, but it only runs 1.9× because the double-precision floating point of the GTX 280 is just 1.5× faster than the Core i7.
- *Compute bandwidth.* Five of the remaining kernels are compute bound: SGEMM, Conv, FFT, MC, and Bilat. The GTX is faster by 3.9, 2.8, 3.0, 1.8, and 5.7, respectively. The first three of these use single-precision floating-point arithmetic, and GTX 280 single-precision is 3—6 × faster. (The 9 × faster than the Core i7 as shown in Figure 4.30 occurs only in the very special case when the GTX 280 can issue a fused multiply-add and a multiply per clock cycle.) MC uses double-precision, which explains why it's just 1.8× faster since DP performance is only 1.5× faster. Bilat uses transcendental functions, which the GTX 280 supports directly (see Figure 4.18). The Core i7 spends two-thirds of its time calculating transcendental functions, so the GTX 280 is 5.7× faster. This observation helps point out the value of

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 4.32 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.

hardware support for operations that occur in your workload: double-precision floating-point and perhaps even transcendentals.

- *Cache benefits.* Ray casting (RC) is only 1.6× faster on the GTX because cache blocking with the Core i7 caches prevents it from becoming memory bandwidth bound, as it is on GPUs. Cache blocking can help Search, too. If the index trees are small so that they fit into the cache, the Core i7 is twice as fast. Larger index trees make them memory bandwidth bound. Overall,

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 4.33 Raw and relative performance measured for both 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.

the GTX 280 runs Search 1.8× faster. Cache blocking also helps Sort. While most programmers wouldn't run Sort on a SIMD Processor, it can be written with a 1-bit Sort primitive called *split*. However, the split algorithm executes many more instructions than a scalar sort does. As a result, the GTX 280 runs only 0.8× as fast as the Core i7. Note that caches also help other kernels on the Core i7 because cache blocking allows SGEMM, FFT, and SpMV to become compute bound. This observation reemphasizes the importance of cache blocking optimizations in Chapter 2.

- *Gather-Scatter*. The multimedia SIMD extensions are of little help if the data are scattered throughout main memory. Optimal performance comes only when data are aligned on 16-byte boundaries. Thus GJK gets little benefit from SIMD on the Core i7. As previously mentioned, GPUs offer gather-scatter addressing that is found in a vector architecture but omitted from SIMD extensions. The Address Coalescing Unit helps as well by combining accesses to the same DRAM line, thereby reducing the number of gathers and scatters. The memory controller also batches together accesses to the identical DRAM page. This combination means the GTX 280 runs GJK a startling 15.2× faster than the Core i7, which is larger than any single physical parameter in Figure 4.30. This observation reinforces the importance of gather-scatter to vector and GPU architectures that are missing from SIMD extensions.

- *Synchronization.* The performance synchronization of Hist is limited by atomic updates, which are responsible for 28% of the total runtime on the Core i7 despite its having a hardware fetch-and-increment instruction. Thus Hist is only 1.7× faster than the GTX 280. Solv solves a batch of independent constraints in a small amount of computation followed by barrier synchronization. The Core i7 benefits from the atomic instructions and a memory consistency model that ensures the right results even if not all previous accesses to memory hierarchy have completed. Without the memory consistency model, the GTX 280 version launches some batches from the system processor, which leads to the GTX 280 running half as fast as the Core i7. This observation points out how synchronization performance can be important for some data-parallel problems.

It was interesting that the gather-scatter support of vector architectures, which predate the SIMD instructions by decades, was so important to the effective usefulness of these SIMD extensions, which some had predicted before the comparison (Gebis and Patterson, 2007). The Intel researchers noted that 6 of the 14 kernels would exploit SIMD better with more efficient gather-scatter support on the Core i7.

Note that an important feature missing from this comparison was describing the level of effort to get the results for the two systems. Ideally, future comparisons would release the code used on both systems so that others could recreate the same experiments on different hardware platforms and possibly improve on the results.

Comparison Update

In the intervening years the weaknesses of the Core i7 and Tesla GTX 280 have been addressed by their successors. Intel's ACV2 added gather instructions, and AVX/512 added scatter instructions, both of which are found in the Intel Skylake series. NVIDIA Pascal has double-precision floating-point performance that is one-half instead of one-eighth the speed of single precision, fast atomic operations, and caches.

Figure 4.34 lists the characteristics of these two successors, Figure 4.35 compares performance using 3 of the 14 benchmarks in the original paper (those were the ones for which we could find source code), and Figure 4.36 shows the two new roofline models. The newer GPU chip is 15 to 50 times faster, the newer CPU chips are 50 times faster than their predecessors, and the new GPU is 2–5 times faster than the new CPU.

From a historical perspective, the 3.9× speedup for SGEMM on GPUs versus CPUs and the availability of CUDA proved to be crucial for the future of both Intel and NVIDIA. SGEMM stands for Single precision GEneral Matrix Multiply, which is the core calculation for Deep Neural Networks. Alex Krizhevsky was a PhD student at the University of Toronto who took a graduate course on computing using GPUs. In 2012 he trained his neural network called AlexNet

	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 4.34 Intel Xeon 8180 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 4.35 Raw and relative performance measured for modern versions of both platforms versus 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.

on a GPU, which he submitted to the ImageNet competition on computer vision. AlexNet not only took first place, but it also dramatically reduced the error rate in vision recognition. While AlexNet was the only DNN entered in 2012, just two years later 100% of the entries relied on DNNs. This result helped make GPUs the engines of choice for DNNs, whose importance has grown dramatically since.

In 2012 the market capitalization for Intel was $>10\times$ larger than for NVIDIA. As we revise this chapter in 2024, the rising popularity of DNNs reversed roles, with NVIDIA now worth $>30\times$ as much as Intel. Note that NVIDIA was not targeting DNNs as part of some master plan in 2012. Success came because they enabled a killer app to emerge on its own from anyone on any topic by opening up the potential of GPUs to as many programmers as possible with CUDA.

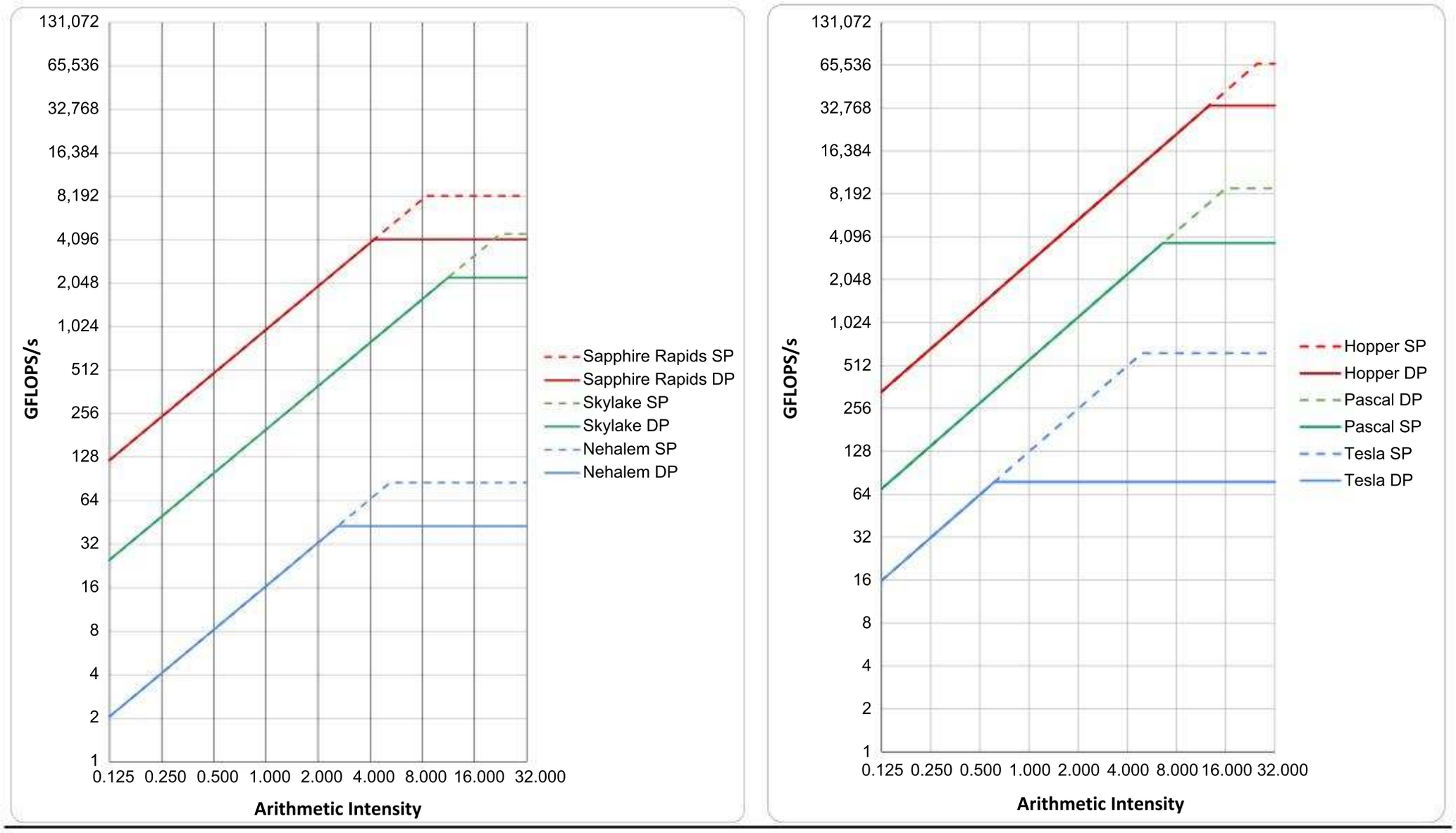

Figure 4.36 Roofline models of older and newer CPUs versus older and newer GPUs. The higher roofline for each computer is single-precision floating-point performance, and the lower one is double-precision performance.

4.8 Fallacies and Pitfalls

While DLP is the easiest form of parallelism after ILP from the programmer's perspective and plausibly the simplest from the architect's perspective, it still has many fallacies and pitfalls.

Fallacy *GPUs suffer from being coprocessors.*

Although the split between main memory and GPU memory has disadvantages, there are advantages to being at a distance from the CPU.

For example, PTX exists in part because of the I/O device nature of GPUs. This level of indirection between the compiler and the hardware gives GPU architects much more flexibility than system processor architects. It's often hard to know in advance whether an architecture innovation will be well supported by compilers and libraries and be important to applications. Sometimes a new mechanism will even prove useful for one or two generations and then fade in importance as the IT world changes. PTX allows GPU architects to try innovations speculatively and drop them in subsequent generations if they disappoint or fade in importance, which encourages experimentation. The justification for

inclusion is understandably considerably higher for system processors—and thus much less experimentation can occur—as distributing binary machine code normally implies that all future generations of that architecture must support the new features.

A demonstration of the value of PTX is that the different generation architecture radically changed the hardware instruction set—from being memory-oriented like x86 to being register oriented like RISC-V *and* doubling the address size to 64 bits—without disrupting the NVIDIA software stack.

Pitfall *Concentrating on peak performance in vector architectures and ignoring start-up overhead.*

Early memory-memory vector processors such as the TI ASC and the CDC STAR-100 had long start-up times. For some vector problems, vectors had to be longer than 100 for the vector code to be faster than the scalar code! On the CYBER 205—derived from the STAR-100—the start-up overhead for DAXPY is 158 clock cycles, which substantially increases the break-even point. If the clock rates of the Cray-1 and the CYBER 205 were identical, the Cray-1 would be faster until the vector length was greater than 64. Because the Cray-1 clock rate was also higher (even though the 205 was newer), the crossover point was a vector length over 100.

Pitfall *Increasing vector performance, without comparable increases in scalar performance.*

This imbalance was a problem on many early vector processors, and a place where Seymour Cray (the architect of the Cray computers) rewrote the rules. Many of the early vector processors had comparatively slow scalar units (and large start-up overheads). Even today, a processor with lower vector performance but better scalar performance can outperform a processor with higher peak vector performance. Good scalar performance keeps down overhead costs (strip mining, for example) and reduces the impact of Amdahl's law.

An excellent example of this comes from comparing a fast scalar processor and a vector processor with lower scalar performance. The Livermore Fortran kernels are a collection of 24 scientific kernels with varying degrees of vectorization. Figure 4.37 shows the performance of two different processors on this benchmark. Despite the vector processor's higher peak performance, its low scalar performance makes it slower than a fast scalar processor as measured by the harmonic mean.

The flip of this danger today is increasing vector performance—say, by increasing the number of lanes—without increasing scalar performance. Such myopia is another path to an unbalanced computer.

The next fallacy is closely related.

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 4.37 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.

Fallacy *You can get good vector performance without providing memory bandwidth.*

As we saw with the DAXPY loop and the Roofline model, memory bandwidth is quite important to all SIMD architectures. DAXPY requires 1.5 memory references per floating-point operation, and this ratio is typical of many scientific codes. Even if the floating-point operations took no time, a Cray-1 could not increase the performance of the vector sequence used, because it is memory limited. The Cray-1 performance on Linpack jumped when the compiler used blocking to change the computation so that values could be kept in the vector registers. This approach lowered the number of memory references per FLOP and improved the performance by nearly a factor of two! Thus the memory bandwidth on the Cray-1 became sufficient for a loop that formerly required more bandwidth.

Fallacy *On GPUs, just add more threads if you don't have enough memory performance.*

GPUs use many CUDA Threads to hide the latency to main memory. If memory accesses are scattered or not correlated among CUDA Threads, the memory system will get progressively slower in responding to each individual request. Eventually, even many threads will not cover the latency. For the "more CUDA Threads" strategy to work, not only do you need lots of CUDA Threads, but the CUDA Threads themselves also must be well behaved in terms of locality of memory accesses.

4.9 Concluding Remarks

Data level parallelism (DLP) is increasing in importance for personal mobile devices, given the popularity of applications showing the importance of audio, video, and games on these devices. When combined with a model that is easier

to program than task-level parallelism and with potentially better energy efficiency, it's easy to see why there has been a renaissance for DLP in this decade.

We are seeing system processors take on more of the characteristics of GPUs, and vice versa. One of the biggest differences in performance between conventional processors and GPUs has been for gather-scatter addressing. Traditional vector architectures show how to add such addressing to SIMD instructions, and we expect to see more ideas added from the well-proven vector architectures to SIMD extensions over time.

As we said in the opening of Section 4.4, the GPU question is not simply which architecture is best, but given the hardware investment to do graphics well, how can it be enhanced to support computation that is more general? Although vector architectures have many advantages on paper, it remains to be proven whether vector architectures can be as good a foundation for graphics as GPUs. RISC-V has embraced vector over SIMD. Like architecture debates of the past, the marketplace will help determine the importance of the strengths and weaknesses of two styles of data-parallel architectures. Thus far, the growing popularity of DNNs and their reliance on NVIDIA GPUs and CUDA for training DNNs, which we describe in Chapter 7, has tipped the balance of DLP to GPUs.

4.10 Historical Perspective and References

Section M.6 (available online) features a discussion on the ILLIAC IV (a representative of the early SIMD architectures) and the Cray-1 (a representative of vector architectures). We also look at multimedia SIMD extensions and the history of GPUs.

Case Study and Exercises by Jason D. Bakos

GPU vs Vector Processor for Low-Arithmetic-Intensity Machine Learning Tasks

Concepts illustrated by this case study

- Arithmetic intensity and performance bottlenecks
- GPU (SIMT) programming model and parallelism
- Vector processor programming model and parallelism

Consider the case where we are executing a kernel that performs a batch normalization and the rectified linear unit (ReLU) layer for a neural network. The corresponding C code for these two layers, combined, is shown below:

```
for (uint32_t i=0 ; i < n_channels; i++) {
      temp1 = max(in[i],0);
      temp2 = (temp1 - mean[i]) / stddev[i];
      out[i] = temp2 * scale[i] + offset[i];
}
```

Assume all variables are of type float16.

4.1. [5] <4.3> How many arithmetic operations are performed per byte accessed from the input and output arrays?

4.2. [15] <4.4> Implement the loop body in PTX. Assume the "in" and "out" arrays are stored in global memory and the "mean," "stddev," "scale," and "offset" arrays are stored in shared memory.

4.3. [5] <4.4> Assume that, for your GPU, loads have a latency of 10 cycles and all other floating-point instructions have a latency of 4 cycles. How many warps must be executing on the GPU core to hide all instruction latencies, that is, keep the core busy at 100% utilization? Assume that the core has an unlimited number of registers and the memory system has unlimited bandwidth. Assuming a warp size of 32 threads, what is the resulting thread count?

4.4. [4] <4.4> Assume the code structure shown below and that, within a warp, P(cond1) = 35%, P(cond2) = 10%, and P(cond3) = 40%, and that codeblock1 has a latency of 100 cycles, codeblock2 has a latency of 200 cycles, and codeblock3 has a latency of 300 cycles. What is the expected execution time?

```
if (cond1) {
  if (cond2) {
       codeblock1
  } else {
       codeblock2
  }
} else {
   if (cond3) {
        codeblock3
   } else {
        codeblock4
   }
}
```

4.5. [15] <4.2> Write an implementation of the code from part a in RV64V assembly. Assume that the architecture supports 16-bit floats. Also assume the following initial register values:

- x1: base address of "in" array
- x2: base address of "mean" array
- x3: base address of "stddev" array
- x4: base address of "scale" array
- x5: base address of "offset" array
- x6: base address of "out" array
- x7: n_channels

You may need the following RV64V instruction that was not included in Table 4.2.

```
vfmax.vf vd, vs2, rs1, vm # vector-scalar max, performs vd(*) = max([vs2(*)],
[rs1]), set vm=1 to disable masking
```

4.6. [10] <4.2> Assume there are two load-store units and five vector floating point units. Rearrange your code from part e to group instructions into convoys.

4.7. [5] <4.2> For a vector length of 32, how many chimes are required and how many cycles per FLOP are achieved for the vector sequence in part f?

4.8 [10/20/20/15/15] <4.2> Consider the following code, which multiplies two vectors that contain single-precision complex values:

```
for (i=0;i < 300;i++) {
 c_re[i] = a_re[i] * b_re[i] - a_im[i] * b_im[i];
 c_im[i] = a_re[i] * b_im[i] + a_im[i] * b_re[i];
}
```

4.9 Assume that the processor runs at 700 MHz and has a maximum vector length of 64. The load/store unit has a start-up overhead of 15 cycles; the multiply unit 8 cycles; and the add/subtract unit 5 cycles.

a. [10] <4.3> What is the arithmetic intensity of this kernel? Justify your answer.
b. [20] <4.2> Convert this loop into RV64V assembly code using strip mining.
c. [20] <4.2> Assuming chaining and a single memory pipeline, how many chimes are required? How many clock cycles are required per complex result value, including start-up overhead?
d. [15] <4.2> If the vector sequence is chained, how many clock cycles are required per complex result value, including overhead?
e. [15] <4.2> Now assume that the processor has three memory pipelines and chaining. If there are no bank conflicts in the loop's accesses, how many clock cycles are required per result?

4.10 [30] <4.2,4.3,4.4> In this problem we will compare the performance of a vector processor with a hybrid system that contains a scalar processor and a GPU-based coprocessor. In the hybrid system the host processor has superior scalar performance to the GPU, so in this case all scalar code is executed on the host processor while all vector code is executed on the GPU. We will refer to the first system as the vector computer and the second system as the hybrid computer. Assume that your target application contains a vector kernel with an arithmetic intensity of 0.5 FLOPs per DRAM byte accessed; however, the application also has a scalar component that must be performed before and after the kernel in order to prepare the input vectors and output vectors, respectively. For a sample dataset, the scalar portion of the code requires 400 ms of execution time on both the vector processor and the host processor in the hybrid system. The kernel reads input vectors

consisting of 200 MB of data and has output data consisting of 100 MB of data. The vector processor has a peak memory bandwidth of 30 GB/s and the GPU has a peak memory bandwidth of 150 GB/s. The hybrid system has an additional overhead that requires all input vectors to be transferred between the host memory and GPU local memory before and after the kernel is invoked. The hybrid system has a direct memory access (DMA) bandwidth of 10 GB/s and an average latency of 10 ms. Assume that both the vector processor and GPU are performance bound by memory bandwidth. Compute the execution time required by both computers for this application.

4.11 [15/25/25] <4.4, 4.5> Section 4.5 discussed the reduction operation that reduces a vector down to a scalar by repeated application of an operation. A reduction is a special type of a loop recurrence. An example is shown as follows:

```
dot=0.0;
for (i=0;i < 64;i++) dot = dot + a[i] * b[i];
```

A vectorizing compiler might apply a transformation called *scalar expansion*, which expands dot into a vector and splits the loop such that the multiply can be performed with a vector operation, leaving the reduction as a separate scalar operation:

```
for (i=0;i < 64;i++) dot[i] = a[i] * b[i];
for (i=1;i < 64;i++) dot[0] = dot[0] + dot[i];
```

As mentioned in Section 4.5, if we allow the floating-point addition to be associative, there are several techniques available for parallelizing the reduction.

```
unsigned int tid = threadIdx.x;
for(unsigned int s=1; s < blockDim.x; s * = 2) {
  if ((tid % (2*s)) == 0) {
    sdata[tid] += sdata[tid + s];
  }
__syncthreads();
}
```

Rewrite the loop to meet these guidelines and eliminate the use of the modulo operator. Assume that there are 32 threads per warp and a bank conflict occurs whenever two or more threads from the same warp reference an index whose modulo by 32 are equal.

a. [15] <4.4, 4.5> One technique is called recurrence doubling, which adds sequences of progressively shorter vectors (i.e., two 32-element vectors, then two 16-element vectors, and so on). Show how the C code would look for executing the second loop in this way.
b. [25] <4.4, 4.5> In some vector processors the individual elements within the vector registers are addressable. In this case, the operands to a vector operation may be two different parts of the same vector register. This flexibility allows another solution for the reduction called *partial sums*. The idea is to reduce

the vector to *m* sums where *m* is the total latency through the vector functional unit, including the operand read and write times. Assume that the RV64 vector registers are addressable (e.g., you can initiate a vector operation with the operand `V1(16)`, indicating that the input operand begins with element 16). Also, assume that the total latency for adds, including the operand read and result write, is eight cycles. Write an RV64V code sequence that reduces the contents of `V1` to eight partial sums.

c. [25] <4.4, 4.5> When performing a reduction on a GPU, one thread is associated with each element in the input vector. The first step is for each thread to write its corresponding value into shared memory. Next, each thread enters a loop that adds each pair of input values. This loop reduces the number of elements by half after each iteration, meaning that the number of active threads also reduces by half after each iteration. In order to maximize the performance of the reduction, the number of fully populated warps should be maximized throughout the course of the loop. In other words the active threads should be contiguous. Also, each thread should index the shared array in such a way as to avoid bank conflicts in the shared memory. The following loop violates only the first of these guidelines and uses the modulo operator, which is very expensive for GPUs:

4.12 [10/10/10/10] <4.3> The following kernel performs a portion of the finite-difference time-domain (FDTD) method for computing Maxwell's equations in a three-dimensional space, part of one of the SPEC06fp benchmarks:

```
for (int x=0; x < NX - 1; x++) {
  for (int y=0; y < NY - 1; y++) {
    for (int z=0; z < NZ - 1; z++) {
      int index = x*NY*NZ + y*NZ + z;
      if (y > 0 && x > 0) {
        material = IDx[index];
        dH1 = (Hz[index] - Hz[index-incrementY])/dy[y];
        dH2 = (Hy[index] - Hy[index-incrementZ])/dz[z];
        Ex[index] = Ca[material]*Ex[index]+Cb[material]* (dH2 - dH1);
      }}}}
```

Assume that `dH1`, `dH2`, `Hy`, `Hz`, `dy`, `dz`, `Ca`, `Cb`, and `Ex` are all single-precision floating-point arrays. Assume `IDx` is an array of unsigned int.

a. [10] <4.3> What is the arithmetic intensity of this kernel?
b. [10] <4.3> Is this kernel amenable to vector or SIMD execution? Why or why not?
c. [10] <4.3> Assume this kernel is to be executed on a processor that has 30 GB/s of memory bandwidth. Will this kernel be memory bound or compute bound?
d. [10] <4.3> Develop a roofline model for this processor, assuming it has a peak computational throughput of 85 GFLOP/s.

4.13 [10/15] <4.4> Assume a GPU architecture that contains 10 SIMD processors. Each SIMD instruction has a width of 32 and each SIMD processor contains 8 lanes for single-precision arithmetic and load/store instructions, meaning that each nondiverged SIMD instruction can produce 32 results every 4 cycles. Assume a kernel that has divergent branches that causes, on average, 80% of threads to be active. Assume that 70% of all SIMD instructions executed are single-precision arithmetic and 20% are load/store. Because not all memory latencies are covered, assume an average SIMD instruction issue rate of 0.85. Assume that the GPU has a clock speed of 1.5 GHz.

4.14 What is speedup in throughput for each of these improvements?

a. [10] <4.4> Compute the throughput, in GFLOPs/second, for this kernel on this GPU.

b. [15] <4.4> Assume that you have the following choices:

(1) Increasing the number of single-precision lanes to 16
(2) Increasing the number of SIMD processors to 15 (assume this change doesn't affect any other performance metrics and that the code scales to the additional processors)
(3) Adding a cache that will effectively reduce memory latency by 40%, which will increase instruction issue rate to 0.95

4.15 [10/15/15] <4.5> In this exercise we will examine several loops and analyze their potential for parallelization.

a. [10] <4.5> Does the following loop have a loop-carried dependency?

```
for (i=0;i < 100;i++) {
  A[i] = B[2*i+4];
  B[4*i+5] = A[i];
}
```

b. [15] <4.5> In the following loop find all the true dependences, output dependences, and antidependences. Eliminate the output dependences and antidependences by renaming.

```
for (i=0;i < 100;i++) {
  A[i] = A[i] * B[i]; /* S1 */
  B[i] = A[i] + c; /* S2 */
  A[i] = C[i] * c; /* S3 */
  C[i] = D[i] * A[i]; /* S4 */
}
```

c. [15] <4.5>Consider the following loop

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

Are there dependencies between S1 and S2? Is this loop parallel? If not, show how to make it parallel.

4.16 [10] <4.4> List and describe at least four factors that influence the performance of GPU kernels. In other words, which runtime behaviors that are caused by the kernel code cause a reduction in resource utilization during kernel execution?

4.17 [10] <4.4> Assume a hypothetical GPU with the following characteristics:

- Clock rate 1.5 GHz
- Contains 16 SIMD processors, each containing 16 single-precision floating-point units
- Has 100 GB/s off-chip memory bandwidth

Without considering memory bandwidth, what is the peak single-precision floating-point throughput for this GPU in GFLOPs/second, assuming that all memory latencies can be hidden? Is this throughput sustainable given the memory bandwidth limitation?

4.18 [60] <4.4> For this programming exercise, you will write and characterize the behavior of a CUDA kernel that contains a high amount of data-level parallelism but also contains conditional execution behavior. Use the NVIDIA CUDA Toolkit along with GPU-SIM from the University of British Columbia (http://www.gpgpu-sim.org/) or the CUDA Profiler to write and compile a CUDA kernel that performs 100 iterations of Conway's Game of Life for a 256 × 256 game board and returns the final state of the game board to the host. Assume that the board is initialized by the host. Associate one thread with each cell. Make sure you add a barrier after each game iteration. Use the following game rules:

- Any live cell with fewer than two live neighbors dies.
- Any live cell with two or three live neighbors lives on to the next generation.
- Any live cell with more than three live neighbors dies.
- Any dead cell with exactly three live neighbors becomes a live cell.

After finishing the kernel answer the following questions:

a. [60] <4.4> Compile your code using the −ptx option and inspect the PTX representation of your kernel. How many PTX instructions make up the PTX implementation of your kernel? Did the conditional sections of your kernel include branch instructions or only predicated nonbranch instructions?

b. [60] <4.4> After executing your code in the simulator, what is the dynamic instruction count? What is the achieved instructions per cycle (IPC) or instruction issue rate? What is the dynamic instruction breakdown in terms of control

instructions, arithmetic-logical unit (ALU) instructions, and memory instructions? Are there any shared memory bank conflicts? What is the effective off-chip memory bandwidth?

c. [60] <4.4> Implement an improved version of your kernel where off-chip memory references are coalesced and observe the differences in runtime performance.

References

David Patterson and Andrew Waterman, 2017. SIMD Instructions Considered Harmful. Computer Architecture Today Blog.

This page intentionally left blank