6.1 Introduction

> "I swing big, with everything I've got. I hit big or I miss big. I like to live as big as I can.
> *Babe Ruth, American baseball player*

Figure 6.1.1: Multiprocessor or cluster organization.

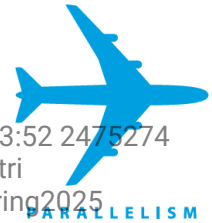
> "Over the Mountains Of the Moon, Down the Valley of the Shadow, Ride, boldly ride the shade replied—If you seek for El Dorado!
> *Edgar Allan Poe, "El Dorado," stanza 4, 1849*

Computer architects have long sought the "The City of Gold" (El Dorado) of computer design: to create powerful computers simply by connecting many existing smaller ones. This golden vision is the fountainhead of *multiprocessors*. Ideally, customers order as many processors as they can afford and receive a commensurate amount of performance. Thus, multiprocessor software must be designed to work with a variable number of processors. As mentioned in COD Chapter 1 (Computer Abstractions and Technology), energy has become the overriding issue for both microprocessors and datacenters. Replacing large inefficient processors with many smaller, efficient processors can deliver better performance per joule both in the large and in the small, if software can efficiently use them. Thus, improved energy efficiency joins scalable performance in the case for multiprocessors.

Since multiprocessor software should scale, some designs support operation in the presence of broken hardware; that is, if a single processor fails in a multiprocessor with *n* processors, these

Multiprocessor: A computer system with at least two processors. This computer is in contrast to a uniprocessor, which has one, and is increasingly hard to find today.

systems would continue to provide service with *n* - 1 processors. Hence, multiprocessors can also improve availability (see COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy)).



High performance can mean high throughput for independent tasks, called *task-level parallelism* or *process-level parallelism*. These tasks are independent single-threaded applications, and they are an important and popular use of multiple processors. This approach is in contrast to running a single job on multiple processors. We use the term *parallel processing program* to refer to a single program that runs on multiple processors simultaneously.

Task-level parallelism or ***process-level parallelism***: Utilizing multiple processors by running independent programs simultaneously.

Parallel processing program: A single program that runs on multiple processors simultaneously.

There have long been scientific problems that have needed much faster computers, and this class of problems has been used to justify many novel parallel computers over the decades. Some of these problems can be handled simply today, using a cluster composed of microprocessors housed in many independent servers (see COD Section 6.7 (Clusters, warehouse scale computers, and other message-passing multiprocessors)). In addition, clusters can serve equally demanding applications outside the sciences, such as search engines, Web servers, email servers, and databases.

Cluster: A set of computers connected over a local area network that function as a single large multiprocessor.

As described in COD Chapter 1 (Computer Abstractions and Technology), multiprocessors have been shoved into the spotlight because the energy problem means that future increases in performance must come from someplace other than much higher clock rates or vastly improved CPI. As we said in COD Chapter 1 (Computer Abstractions and Technology), they are called *multicore microprocessors* instead of multiprocessor microprocessors, presumably to avoid redundancy in naming. Hence, processors are often called *cores* in a multicore chip. The number of cores is expected to increase with improved hardware technology. These multicores are almost always *Shared Memory Processors (SMPs)*, as they usually share a single physical address space. We'll see SMPs more in COD Section 6.5 (Multicore and other shared memory multiprocessors).

Multicore microprocessor: A microprocessor containing multiple processors ("cores") in a single integrated circuit. Virtually all microprocessors today in desktops and servers are multicore.

Shared memory multiprocessor (***SMP***): A parallel processor with a single physical address space.

The state of technology today means that programmers who care about performance must become parallel programmers (see COD Section 6.12).

The tall challenge facing the industry is to create hardware and software that will make it easy to write correct parallel processing programs that will execute efficiently in performance and energy as the number of cores per chip scales.

This abrupt shift in microprocessor design caught many off guard, so there is a great deal of confusion about the terminology and what it means. The figure below tries to clarify the terms serial, parallel, sequential, and concurrent. The columns of this figure represent the software, which is either inherently sequential or concurrent. The rows of the figure represent the hardware, which is either serial or parallel. For example, the programmers of compilers think of them as sequential programs: the steps include parsing, code generation, optimization, and so on. In contrast, the programmers of operating systems normally think of them as concurrent programs: cooperating processes handling I/O events due to independent jobs running on a computer.

Figure 6.1.2: Hardware/software categorization and examples of application perspective on concurrency versus hardware perspective on parallelism (COD Figure 6.1).

		Software	
		Sequential	**Concurrent**
Hardware	Serial	Matrix Multiply written in MatLab running on an Intel Pentium 4	Windows Vista Operating System running on an Intel Pentium 4
	Parallel	Matrix Multiply written in MATLAB running on an Intel Core i7	Windows Vista Operating System running on an Intel Core i7

The point of these two axes of the figure above is that concurrent software can run on serial hardware, such as operating systems for the Intel Pentium 4 uniprocessor, or on parallel hardware, such as an OS on the more recent Intel Core i7. The same is true for sequential software. For example, the MATLAB programmer writes a matrix multiply thinking about it sequentially, but it could run serially on the Pentium 4 or in parallel on the Intel Core i7.

You might guess that the only challenge of the parallel revolution is figuring out how to make naturally sequential software have high performance on parallel hardware, but it is also to make concurrent programs have high performance on multiprocessors as the number of processors increases. With this distinction made, in the rest of this chapter we will use *parallel processing program* or *parallel software* to mean either sequential or concurrent software running on parallel hardware. The next section of this chapter describes why it is hard to create efficient parallel processing programs.

Before proceeding further down the path to parallelism, don't forget our initial incursions from the earlier chapters:

- COD Chapter 2 (Instructions: Language of the Computer), COD Section 2.11: Parallelism and instructions: Synchronization
- COD Chapter 3 (Arithmetic for Computers), COD Section 3.6: Parallelism and computer arithmetic: Subword parallelism
- COD Chapter 4 (The Processor), COD Section 4.11: Parallelism via instructions
- COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy), COD Section 5.10: Parallelism and memory hierarchy: Cache coherence

PARTICIPATION ACTIVITY 6.1.1: Check yourself: Multiprocessors.

1) To benefit from a multiprocessor, an application must be concurrent.
   - ○ True
   - ○ False

6.2 The difficulty of creating parallel processing programs

The challenge with parallelism is not the hardware; it is that too few important application programs have been rewritten to complete tasks sooner on multiprocessors. It is difficult to write software that uses multiple processors to complete one task faster, and the problem gets worse as the number of processors increases.

Why has this been so? Why have parallel processing programs been so much harder to develop than sequential programs?

The first reason is that you *must* get better performance or better energy efficiency from a parallel processing program on a multiprocessor; otherwise, you would just use a sequential program on a uniprocessor, as sequential programming is simpler. In fact, uniprocessor design techniques such

as superscalar and out-of-order execution take advantage of instruction-level parallelism (see COD Chapter 4 (The Processor)), normally without the involvement of the programmer. Such innovations reduced the demand for rewriting programs for multiprocessors, since programmers could do nothing and yet their sequential programs would run faster on new computers.

Why is it difficult to write parallel processing programs that are fast, especially as the number of processors increases? In COD Chapter 1 (Computer Abstractions and Technology), we used the analogy of eight reporters trying to write a single story in hopes of doing the work eight times faster. To succeed, the task must be broken into eight equal-sized pieces, because otherwise some reporters would be idle while waiting for the ones with larger pieces to finish. Another speed-up obstacle could be that the reporters would spend too much time communicating with each other instead of writing their pieces of the story. For both this analogy and parallel programming, the challenges include scheduling, partitioning the work into parallel pieces, balancing the load evenly between the workers, time to synchronize, and overhead for communication between the parties. The challenge is stiffer with the more reporters for a newspaper story and with the more processors for parallel programming.

Our discussion in COD Chapter 1 (Computer Abstractions and Technology) reveals another obstacle, namely Amdahl's Law. It reminds us that even small parts of a program must be parallelized if the program is to make good use of many cores.

Example 6.2.1: Speed-up challenge.

Suppose you want to achieve a speed-up of 90 times faster with 100 processors. What percentag computation can be sequential?

Answer

Amdahl's Law (COD Chapter 1 (Computer Abstractions and Technology)) says

$$\textbf{Execution time after improvement} = \frac{\textbf{Execution time affected by improvement}}{\textbf{Amount of improvement}} +$$

We can reformulate Amdahl's Law in terms of speed-up versus the original execution time:

$$\textbf{Speed-up} = \frac{\textbf{Execution time before}}{\textbf{(Execution time before - Execution time affected)} + \frac{\textbf{Execution}}{\textbf{Amount o}}}$$

This formula is usually rewritten assuming that the execution time before is 1 for some unit of tim affected by improvement is considered the fraction of the original execution time:

$$\text{Speed-up} = \frac{1}{(1-\text{Fraction time affected}) + \frac{\text{Fraction time affected}}{\text{Amount of improvemer}}}$$

Substituting 90 for speed-up and 100 for amount of improvement into the formula above:

$$90 = \frac{1}{(1-\text{Fraction time affected}) + \frac{\text{Fraction time affected}}{100}}$$

Then simplifying the formula and solving for fraction time affected:

$$90 \times (1 - 0.99 \times \text{Fraction time affected}) = 1$$

$$90 - (90 \times 0.99 \times \text{Fraction time affected}) = 1$$

$$90 - 1 = 90 \times 0.99 \times \text{Fraction time}$$

$$\text{Fraction time affected} = 89/89.1 = 0.999$$

Thus, to achieve a speed-up of 90 from 100 processors, the sequential percentage can only be 0.1

Yet, there are applications with plenty of parallelism, as we shall see next.

Example 6.2.2: Speed-up challenge: Bigger problem.

Suppose you want to perform two sums: one is a sum of 10 scalar variables, and one is a matrix s dimensional arrays, with dimensions 10 by 10. For now let's assume only the matrix sum is paralle how to parallelize scalar sums. What speed-up do you get with 10 versus 40 processors? Next, ca assuming the matrices grow to 20 by 20.

Answer

If we assume performance is a function of the time for an addition, *t*, then there are 10 additions t parallel processors and 100 additions that do. If the time for a single processor is 110 *t*, the execu processors is

$$\text{Execution time after improvement} = \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} +$$

$$\text{Execution time after improvement} = \frac{100t}{10} + 10t = 20t$$

so the speed-up with 10 processors is 110*t*/20*t* = 5.5. The execution time for 40 processors is

$$\text{Execution time after improvement} = \frac{100t}{40} + 10t = 12.5t$$

so the speed-up with 40 processors is 110*t*/12.5*t* = 8.8. Thus, for this problem size, we get about 5 speed-up with 10 processors, but only 22% with 40.

Look what happens when we increase the matrix. The sequential program now takes 10*t* + 400*t* = for 10 processors is

$$\text{Execution time after improvement} = \frac{400t}{10} + 10t = 50t$$

so the speed-up with 10 processors is 410*t*/50*t* = 8.2. The execution time for 40 processors is

$$\text{Execution time after improvement} = \frac{400t}{40} + 10t = 20t$$

so the speed-up with 40 processors is 410*t*/20*t* = 20.5. Thus, for this larger problem size, we get 8 speed-up with 10 processors and 51% with 40.

These examples show that getting good speed-up on a multiprocessor while keeping the problem size fixed is harder than getting good speed-up by increasing the size of the problem. This insight allows us to introduce two terms that describe ways to scale up.

Strong scaling means measuring speed-up while keeping the problem size fixed. *Weak scaling* means that the problem size grows proportionally to the increase in the number of processors. Let's assume that the size of the problem, M, is the working set in main memory, and we have P processors. Then the memory per processor for strong scaling is approximately M/P, and for weak scaling, it is approximately M.

Strong scaling: Speed-up achieved on a multiprocessor without increasing the size of the problem.

Weak scaling: Speed-up achieved on a multiprocessor while increasing the size of the problem proportionally to the increase in the number of processors.

Note that the **memory hierarchy** can interfere with the conventional wisdom about weak scaling being easier than strong scaling. For example, if the weakly scaled dataset no longer fits in the last level cache of a multicore microprocessor, the resulting performance could be much worse than by using strong scaling.

Depending on the application, you can argue for either scaling approach. For example, the TPC-C debit-credit database benchmark requires that you scale up the number of

customer accounts in proportion to the higher transactions per minute. The argument is that it's nonsensical to think that a given customer base is suddenly going to start using ATMs 100 times a day just because the bank gets a faster computer. Instead, if you're going to demonstrate a system that can perform 100 times the numbers of transactions per minute, you should run the experiment with 100 times as many customers. Bigger problems often need more data, which is an argument for weak scaling.

This final example shows the importance of load balancing.

Example 6.2.3: Speed-up challenge: Balancing load.

To achieve the speed-up of 20.5 on the previous larger problem with 40 processors, we assumed the load was perfectly balanced. That is, each of the 40 processors had 2.5% of the work to do. Instead, show the impact on speed-up if one processor's load is higher than all the rest. Calculate at twice the load (5%) and five times the load (12.5%) for that hardest working processor. How well utilized are the rest of the processors?

Answer

If one processor has 5% of the parallel load, then it must do 5% × 400 or 20 additions, and the other 39 will share the remaining 380. Since they are operating simultaneously, we can just calculate the execution time as a maximum

$$\textbf{Execution time after improvement} = \textbf{Max}\left(\frac{380t}{39}, \frac{20t}{1}\right) + 10t = 30t$$

The speed-up drops from 20.5 to 410*t*/30*t* = 14. The remaining 39 processors are utilized less than half the time: while waiting 20*t* for hardest working processor to finish, they only compute for 380*t*/39 = 9.7*t*.

If one processor has 12.5% of the load, it must perform 50 additions. The formula is:

$$\textbf{Execution time after improvement} = \textbf{Max}\left(\frac{350t}{39}, \frac{50t}{1}\right) + 10t = 60t$$

The speed-up drops even further to 410*t*/60*t* = 7. The rest of the processors are utilized less than 20% of the time (9*t*/50*t*). This example demonstrates the importance of balancing load, for just a single processor with twice the load of the others cuts speed-up by a third, and five times the load on just one processor reduces speed-up by almost a factor of three.

Now that we better understand the goals and challenges of parallel processing, we give an

overview of the rest of the chapter. The next section (COD Section 6.3 (SISD, MIMD, SIMD, SPMD, and vector)) describes a much older classification scheme than in COD Figure 6.1 (Hardware/ software categorization and examples …). In addition, it describes two styles of instruction set architectures that support running of sequential applications on parallel hardware, namely *SIMD* and *vector*. COD Section 6.4 (Hardware multithreading) then describes *multithreading*, a term often confused with multiprocessing, in part because it relies upon similar concurrency in programs. COD Section 6.5 (Multicore and other shared memory multiprocessors) describes the first the two alternatives of a fundamental parallel hardware characteristic, which is whether or not all the processors in the systems rely upon a single physical address space. As mentioned above, the two popular versions of these alternatives are called *shared memory multiprocessors* (SMPs) and *clusters*, and this section covers the former. COD Section 6.6 (Introduction to graphics processing units) describes a relatively new style of computer from the graphics hardware community, called a *graphics-processing unit* (GPU) that also assumes a single physical address. (COD Appendix C (Graphics and Computing GPUs) describes GPUs in even more detail.) COD Section 6.6.5 introduces Domain Specific Architectures, where the processors are customized to perform well in one domain but need not run all programs well. COD Section 6.8 (Clusters, warehouse scale computers, and other message-passing multiprocessors) describes clusters, a popular example of a computer with multiple physical address spaces. COD Section 6.9 (Introduction to multiprocessor network topologies) shows typical topologies used to connect many processors together, either server nodes in a cluster or cores in a microprocessor. COD Section 6.10 (Communicating to the outside world: Cluster networking) describes the hardware and software for communicating between nodes in a cluster using Ethernet. It shows how to optimize its performance using custom software and hardware. We next discuss the difficulty of finding parallel benchmarks in COD Section 6.10 (Multiprocessor benchmarks and performance models). This section also includes a simple, yet insightful performance model that helps in the design of applications as well as architectures. We use this model as well as parallel benchmarks in COD Section 6.12 (Real stuff: Benchmarking Intel Core i7 versus NVIDIA Tesla GPU) to compare a Domain Specific Architecture to a GPU. COD Section 6.13 (Going faster: Multiple processors and matrix multiply) divulges the final and largest step in our journey of accelerating matrix multiply. Parallel processing uses 48 cores to improve performance by a factor of 12 to 17 if we increase matrix size (weak scaling). We close with fallacies and pitfalls and our conclusions for parallelism.

In the next section, we introduce acronyms that you probably have already seen to identify different types of parallel computers.

PARTICIPATION ACTIVITY 6.2.1: Check yourself: Strong scaling.

1) Strong scaling is not bound by Amdahl's Law.
   - ○ True
   - ○ False

6.3 SISD, MIMD, SIMD, SPMD, and vector

(Original section[1])

One categorization of parallel hardware proposed in the 1960s is still used today. It was based on the number of instruction streams and the number of data streams. The figure below shows the categories. Thus, a conventional uniprocessor has a single instruction stream and single data stream, and a conventional multiprocessor has multiple instruction streams and multiple data streams. These two categories are abbreviated *SISD* and *MIMD*, respectively.

SISD or ***single instruction stream***, ***single data stream***: A uniprocessor.

MIMD or ***multiple instruction streams***, ***multiple data streams***: A multiprocessor.

Figure 6.3.1: Hardware categorization and examples based on number of instruction streams and data streams: SISD, SIMD, MISD, and MIMD (COD Figure 6.2).

		Data Streams	
		Single	**Multiple**
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Multiple	MISD: No examples today	MIMD: Intel Core i7

While it is possible to write separate programs that run on different processors on a MIMD computer and yet work together for a grander, coordinated goal, programmers normally write a single program that runs on all processors of an MIMD computer, relying on conditional statements when different processors should execute different sections of code. This style is called *Single Program Multiple Data (SPMD)*, but it is just the normal way to program a MIMD computer.

SPMD or ***single program, multiple data streams***: The conventional MIMD programming model, where a single program runs across all processors.

The closest we can come to multiple instruction streams and single data stream (*MISD*) processor might be a "stream processor" that would perform a series of computations on a single data stream in a pipelined fashion: parse the input from the network, decrypt the data, decompress it,

search for match, and so on. The inverse of MISD is much more popular. *SIMD* computers operate on vectors of data. For example, a single SIMD instruction might add 64 numbers by sending 64 data streams to 64 ALUs to form 64 sums within a single clock cycle. The subword parallel instructions that we saw in COD Sections 3.6 (Parallelism and computer arithmetic: Subword parallelism) and 3.7 (Real stuff: Streaming SIMD extensions and advanced vector extensions in x86) are another example of SIMD; indeed, the middle letter of Intel's SSE acronym stands for SIMD.

SIMD or ***single instruction stream, multiple data streams***: The same instruction is applied to many data streams, as in a vector processor.

The virtues of SIMD are that all the parallel execution units are synchronized and they all respond to a single instruction that emanates from a single *program counter* (PC). From a programmer's perspective, this is close to the already familiar SISD. Although every unit will be executing the same instruction, each execution unit has its own address registers, and so each unit can have different data addresses. Thus, in terms of COD Figure 6.1 (Hardware/software categorization …), a sequential application might be compiled to run on serial hardware organized as a SISD or in parallel hardware that was organized as a SIMD.

The original motivation behind SIMD was to amortize the cost of the control unit over dozens of execution units. Another advantage is the reduced instruction bandwidth and space—SIMD needs only one copy of the code that is being simultaneously executed, while message-passing MIMDs may need a copy in every processor, and shared memory MIMD will need multiple instruction caches.

SIMD works best when dealing with arrays in `for` loops. Hence, for parallelism to work in SIMD, there must be a great deal of identically structured data, which is called *data-level parallelism*. SIMD is at its weakest in `case` or `switch` statements, where each execution unit must perform a different operation on its data, depending on what data it has. Execution units with the wrong data must be disabled so that units with proper data may continue. If there are *n* cases, in these situations SIMD processors essentially run at 1/*n*th of peak performance.

Data-level parallelism: Parallelism achieved by performing the same operation on independent data.

The so-called array processors that inspired the SIMD category lost popularity until recently (see COD Section 6.7 (Domain specific Architectures) and COD Section 6.16 (Historical Perspective and Further Reading)), but two current interpretations of SIMD have been active for decades.

SIMD in x86: Multimedia extensions

As described in COD Chapter 3 (Arithmetic for Computers), subword parallelism for narrow integer data was the original inspiration of the Multimedia Extension (MMX) instructions of the x86 in

1996. As Moore's Law continued, more instructions were added, leading first to *Streaming SIMD Extensions* (SSE) and now *Advanced Vector Extensions* (AVX). AVX supports the simultaneous execution of four 64-bit floating-point numbers. The width of the operation and the registers is encoded in the opcode of these multimedia instructions. As the data width of the registers and operations grew, the number of opcodes for multimedia instructions exploded, and now there are hundreds of SSE and AVX instructions (see COD Chapter 3 (Arithmetic for Computers)).

Vector

An older and, as we shall see, more elegant interpretation of SIMD is called a *vector architecture*, which has been closely identified with computers designed by Seymour Cray starting in the 1970s. It is also a great match to problems with lots of data-level parallelism. Rather than having 64 ALUs perform 64 additions simultaneously, like the old array processors, the vector architectures pipelined the ALU to get good performance at lower cost. The basic philosophy of vector architecture is to collect data elements from memory, put them in order into a large set of registers, operate on them sequentially in registers using **pipelined execution units**, and then write the results back to memory. A key feature of vector architectures is then a set of vector registers. Thus, a vector architecture might have 32 vector registers, each with 64 64-bit elements.

Example 6.3.1: Comparing vector to conventional code.

Suppose we extend the MIPS instruction set architecture with vector instructions and vector registers. Vector operations use the same names as MIPS operations, but with the letter "V" appended. For example, `addv.d` adds two double-precision vectors. The vector instructions take as their input either a pair of vector registers (`addv.d`) or a vector register and a scalar register (`addvs.d`). In the latter case, the value in the scalar register is used as the input for all operations—the operation `addvs.d` will add the contents of a scalar register to each element in a vector register. The names `lv` and `sv` denote vector load and vector store, and they load or store an entire vector of double-precision data. One operand is the vector register to be loaded or stored; the other operand, which is a MIPS general-purpose register, is the starting address of the vector in memory. Given this short description, show the conventional MIPS code versus the vector MIPS code for

$$\mathbf{Y} = a \times \mathbf{X} + \mathbf{Y}$$

where *X* and *Y* are vectors of 64 double precision floating-point numbers, initially resident in memory, and *a* is a scalar double precision variable. (This example is the so-called DAXPY loop that forms the inner loop of the Linpack benchmark; DAXPY stands for double precision *a* × *X* plus *Y*.). Assume that the starting addresses of *X* and *Y* are in `$s0` and

`$s1`, respectively.

Answer

Here is the conventional MIPS code for DAXPY:


```
        l.d     $f0, a($sp)        : load scalar a
        addiu   $t0, $s0, #512     : upper bound of what to load
loop:   l.d     $f2, 0($s0)        : load x(i)
        mul.d   $f2, $f2, $f0      : a x x(i)
        l.d     $f4, 0($s1)        : load y(i)
        add.d   $f4, $f4, $f2      : a x x(i) + y(i)
        s.d     $f4, 0($s1)        : store into y(i)
        addiu   $s0, $s0, #8       : increment index to x
        addiu   $s1, $s1, #8       : increment index to y
        subu    $t1, $t0, $s0      : compute bound
        bne     $t1, $zero, loop   : check if done
```

Here is the vector MIPS code for DAXPY:

```
        l.d     $f0, a($sp)        : load scalar a
        lv      $v1, 0($s0)        : load vector x
        mulvs.d $v2, $v1, $f0      : vector-scalar multiply
        lv      $v3, 0($s1)        : load vector y
        addv.d  $v4, $v2, $v3      : add y to product
        sv      $v4, 0($s1)        : store the result
```

There are some interesting comparisons between the two code segments in this example. The most dramatic is that the vector processor greatly reduces the dynamic instruction bandwidth, executing only 6 instructions versus almost 600 for the traditional MIPS architecture. This reduction occurs both because the vector operations work on 64 elements at a time and because the overhead instructions that constitute nearly half the loop on MIPS are not present in the vector code. As you might expect, this reduction in instructions fetched and executed saves energy.

Another important difference is the frequency of **pipeline** hazards (COD Chapter 4 (The Processor)). In the straightforward MIPS code, every `add.d` must wait for a `mul.d`, every `s.d` must wait for the `add.d` and every `add.d` *and* `mul.d` must wait on `l.d`. On the vector processor, each vector instruction will only stall 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 *operation*,

rather than once per vector *element*. In this example, the pipeline stall frequency on MIPS will be about 64 times higher than it is on the vector version of MIPS. The pipeline stalls can be reduced on MIPS by using loop unrolling (see COD Chapter 4 (The Processor)). However, the large difference in instruction bandwidth cannot be reduced.

Since the vector elements are independent, they can be operated on in parallel, much like subword parallelism for AVX instructions. All modern vector computers have vector functional units with multiple parallel pipelines (called *vector lanes*; see COD Figures 6.2 (Hardware categorization and examples …) and 6.3 (Using multiple functional units to improve the performance …)) that can produce two or more results per clock cycle.

Elaboration

The loop in the example above exactly matched the vector length. When loops are shorter, vector architectures use a register that reduces the length of vector operations. When loops are larger, we add bookkeeping code to iterate full-length vector operations and to handle the leftovers. This latter process is called strip mining.

Vector versus scalar

Vector instructions have several important properties compared to conventional instruction set architectures, which are called *scalar architectures* in this context:

- A single vector instruction specifies a great deal of work—it is equivalent to executing an entire loop. The instruction fetch and decode bandwidth needed is dramatically reduced.
- By using a vector instruction, the compiler or programmer indicates that the computation of each result in the vector is independent of the computation of other results in the same vector, so hardware does not have to check for data hazards within a vector instruction.
- Vector architectures and compilers have a reputation of making it much easier than when using MIMD multiprocessors to write efficient applications when they contain data-level parallelism.
- Hardware need only check for data hazards between two vector instructions once per vector operand, not once for every element within the vectors. Reduced checking can save energy as well as time.
- Vector instructions that access memory have a known access pattern. If the vector's elements are all adjacent, then fetching the vector from a set of heavily interleaved memory banks works very well. Thus, the cost of the latency to main memory is seen only once for the entire vector, rather than once for each word of the vector.
- Because an entire loop is replaced by a vector instruction whose behavior is predetermined,

control hazards that would normally arise from the loop branch are nonexistent.
- The savings in instruction bandwidth and hazard checking plus the efficient use of memory bandwidth give vector architectures advantages in power and energy versus scalar architectures.

For these reasons, vector operations can be made faster than a sequence of scalar operations on the same number of data items, and designers are motivated to include vector units if the application domain can often use them.

Vector versus multimedia extensions

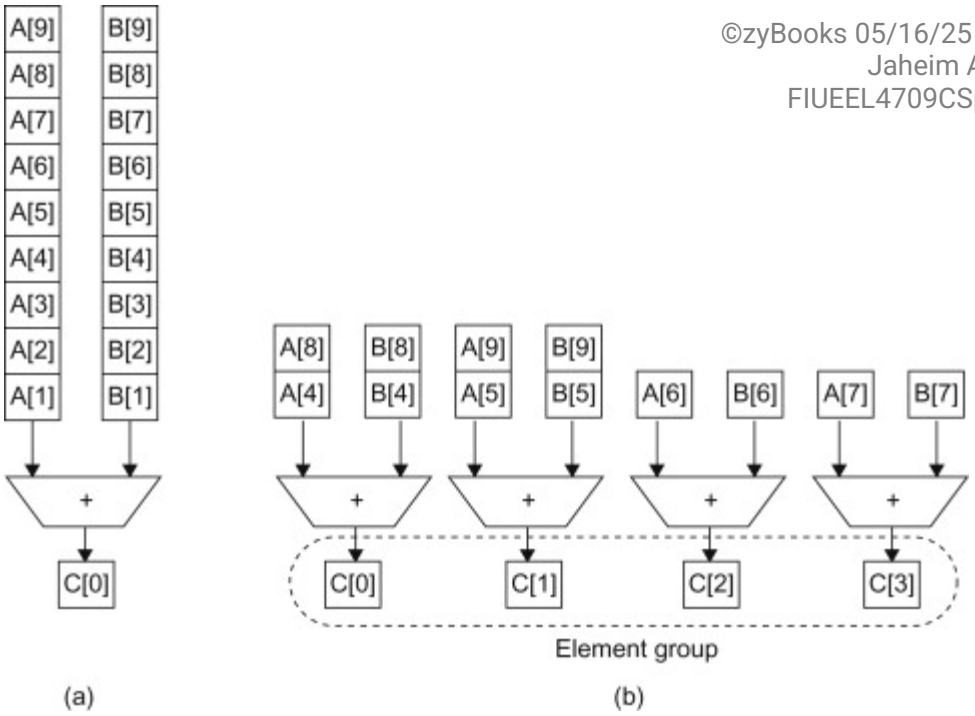
Like multimedia extensions found in the x86 AVX instructions, a vector instruction specifies multiple operations. However, multimedia extensions typically specify a few operations while vector specifies dozens of operations. Unlike multimedia extensions, the number of elements in a vector operation is not in the opcode but in a separate register. This distinction means different versions of the vector architecture can be implemented with a different number of elements just by changing the contents of that register and hence retain binary compatibility. In contrast, a new large set of opcodes is added each time the "vector" length changes in the multimedia extension architecture of the x86: MMX, SSE, SSE2, AVX, AVX2, ….

Also unlike multimedia extensions, the data transfers need not be contiguous. Vectors support both strided accesses, where the hardware loads every *n*th data element in memory, and indexed accesses, where hardware finds the addresses of the items to be loaded in a vector register. Indexed accesses are also called *gather-scatter*, in that indexed loads gather elements from main memory into contiguous vector elements and indexed stores scatter vector elements across main memory.

Like multimedia extensions, vector architectures easily capture the flexibility in data widths, so it is easy to make a vector operation work on 32 64-bit data elements or 64 32-bit data elements or 128 16-bit data elements or 256 8-bit data elements. The parallel semantics of a vector instruction allows an implementation to execute these operations using a deeply **pipelined** functional unit, an array of parallel functional units, or a combination of parallel and pipelined functional units. The figure below illustrates how to improve vector performance by using parallel pipelines to execute a vector add instruction.

Figure 6.3.2: Using multiple functional units to improve the performance of a single vector add instruction, C = A + B (COD Figure 6.3).

The vector processor (a) on the left has a single add pipeline and can complete one addition per cycle. The vector processor (b) on the right has four add pipelines or lanes and can complete four additions per cycle. The elements within a single vector add instruction are interleaved across the four lanes.

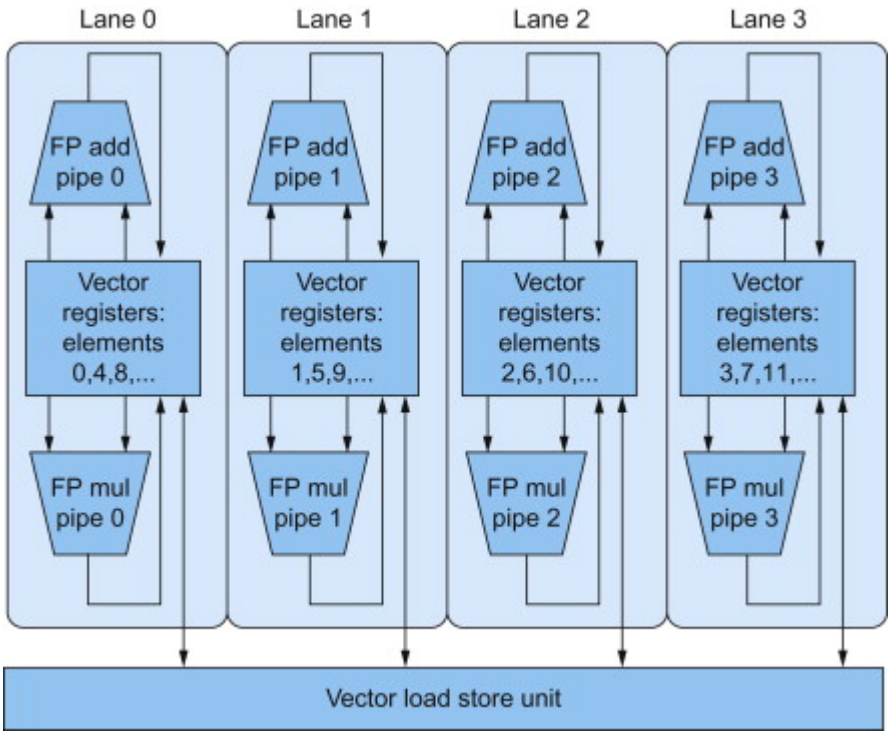


Vector arithmetic instructions usually only allow element N of one vector register to take part in operations with element N from other vector registers. This dramatically simplifies the construction of a highly parallel vector unit, which can be structured as multiple parallel *vector lanes*. As with a traffic highway, we can increase the peak throughput of a vector unit by adding more lanes. The figure below shows the structure of a four-lane vector unit. Thus, going to four lanes from one lane reduces the number of clocks per vector instruction by roughly a factor of four. For multiple lanes to be advantageous, both the applications and the architecture must support long vectors. Otherwise, they will execute so quickly that you'll run out of instructions, requiring instruction level **parallel** techniques like those in COD Chapter 4 (The Processor) to supply enough vector instructions.

Vector lane: One or more vector functional units and a portion of the vector register file. Inspired by lanes on highways that increase traffic speed, multiple lanes execute vector operations simultaneously.

Figure 6.3.3: Structure of a vector unit containing four lanes (COD Figure 6.4).

The vector-register storage 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 acts in concert to complete a single vector instruction. Note how each section of the vector-register file only needs to provide enough read and write ports (see COD Chapter 4 (The Processor)) for functional units local to its lane.



Generally, vector architectures are a very efficient way to execute data parallel processing programs; they are better matches to compiler technology than multimedia extensions; and they are easier to evolve over time than the multimedia extensions to the x86 architecture.

Given these classic categories, we next see how to exploit parallel streams of instructions to improve the performance of a *single* processor, which we will reuse with multiple processors.

PARTICIPATION ACTIVITY 6.3.1: Check yourself: Vector versus multimedia extensions.

1) As exemplified in the x86, multimedia extensions can be thought of as a vector architecture with short vectors

that supports only contiguous vector data transfers.

- ○ True
- ○ False

Elaboration

Given the advantages of vector, why aren't they more popular outside high-performance computing? There were concerns about the larger state for vector registers increasing context switch time and the difficulty of handling page faults in vector loads and stores, and SIMD instructions achieved some of the benefits of vector instructions. In addition, as long as advances in instruction level parallelism could deliver on the performance promise of Moore's Law, there was little reason to take the chance of changing architecture styles.

Elaboration

Another advantage of vector and multimedia extensions is that it is relatively easy to extend a scalar instruction set architecture with these instructions to improve performance of data parallel operations.

Elaboration

The Haswell-generation x86 processors from Intel support AVX2, which has a gather operation but not a scatter operation. Skylake and later generation processors support AVX512, which adds a scatter operation.

(*1) This section is in original form.

6.4 Hardware multithreading

(Original section[1])

A related concept to MIMD, especially from the programmer's perspective, is *hardware multithreading*. While MIMD relies on multiple *processes* or *threads* to try to keep multiple processors busy, hardware multithreading allows multiple threads to share the functional units of a single processor in an overlapping fashion to try to utilize the hardware resources efficiently. To permit this sharing, the processor must duplicate the independent state of each thread. For example, each thread would have a separate copy of the register file and the program counter. The memory itself can be shared through the virtual memory mechanisms, which already support multi-programming. In addition, the hardware must support the ability to change to a different thread relatively quickly. In particular, a thread switch should be much more efficient than a process switch, which typically requires hundreds to thousands of processor cycles while a thread switch can be instantaneous.

Hardware multithreading: Increasing utilization of a processor by switching to another thread when one thread is stalled.

Thread: A thread includes the program counter, the register state, and the stack. It is a lightweight process; whereas threads commonly share a single address space, processes don't.

Process: A process includes one or more threads, the address space, and the operating system state. Hence, a process switch usually invokes the operating system, but not a thread switch.

There are two main approaches to hardware multithreading. *Fine-grained multithreading* switches between threads on each instruction, resulting in interleaved execution of multiple threads. This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that clock cycle. To make fine-grained multithreading practical, the processor must be able to switch threads on every clock cycle. One advantage of fine-grained multithreading is that it can hide the throughput losses that arise from both short and long stalls, since instructions from other threads can be executed when one thread stalls. The primary disadvantage of fine-grained multithreading is that it slows down the execution of the individual threads, since a thread that is ready to execute without stalls will be delayed by instructions from other threads.

Fine-grained multithreading: A version of hardware multithreading that implies switching between threads after every instruction.

Coarse-grained multithreading was invented as an alternative to fine-grained multithreading. Coarse-grained multithreading switches threads only on costly

stalls, such as last-level cache misses. This change relieves the need to have thread switching be extremely fast and is much less likely to slow down the execution of an individual thread, since instructions from other threads will only be issued when a thread encounters a costly stall. Coarse-grained multithreading suffers, however, from a major drawback: it is limited in its ability to overcome throughput losses, especially from shorter stalls. This limitation arises from the **pipeline** start-up costs of coarse-grained multithreading. Because a processor with coarse-grained multithreading issues instructions from a single thread, when a stall occurs, the pipeline must be emptied or frozen. The new thread that begins executing after the stall must fill the pipeline before instructions will be able to complete. Due to this start-up overhead, coarse-grained multithreading is much more useful for reducing the penalty of high-cost stalls, where pipeline refill is negligible compared to the stall time.

Coarse-grained multithreading: A version of hardware multithreading that implies switching between threads only after significant events, such as a last-level cache miss.

Simultaneous multithreading (SMT) is a variation on hardware multithreading that uses the resources of a multiple-issue, dynamically scheduled **pipelined** processor to exploit thread-level parallelism at the same time it exploits instruction-level parallelism (see COD Chapter 4 (The Processor)). The key insight that motivates SMT is that multiple-issue processors often have more functional unit parallelism available than most single threads can effectively use. Furthermore, with register renaming and dynamic scheduling (see COD Chapter 4 (The Processor)), multiple instructions from independent threads can be issued without regard to the dependences among them; the resolution of the dependences can be handled by the dynamic scheduling capability.

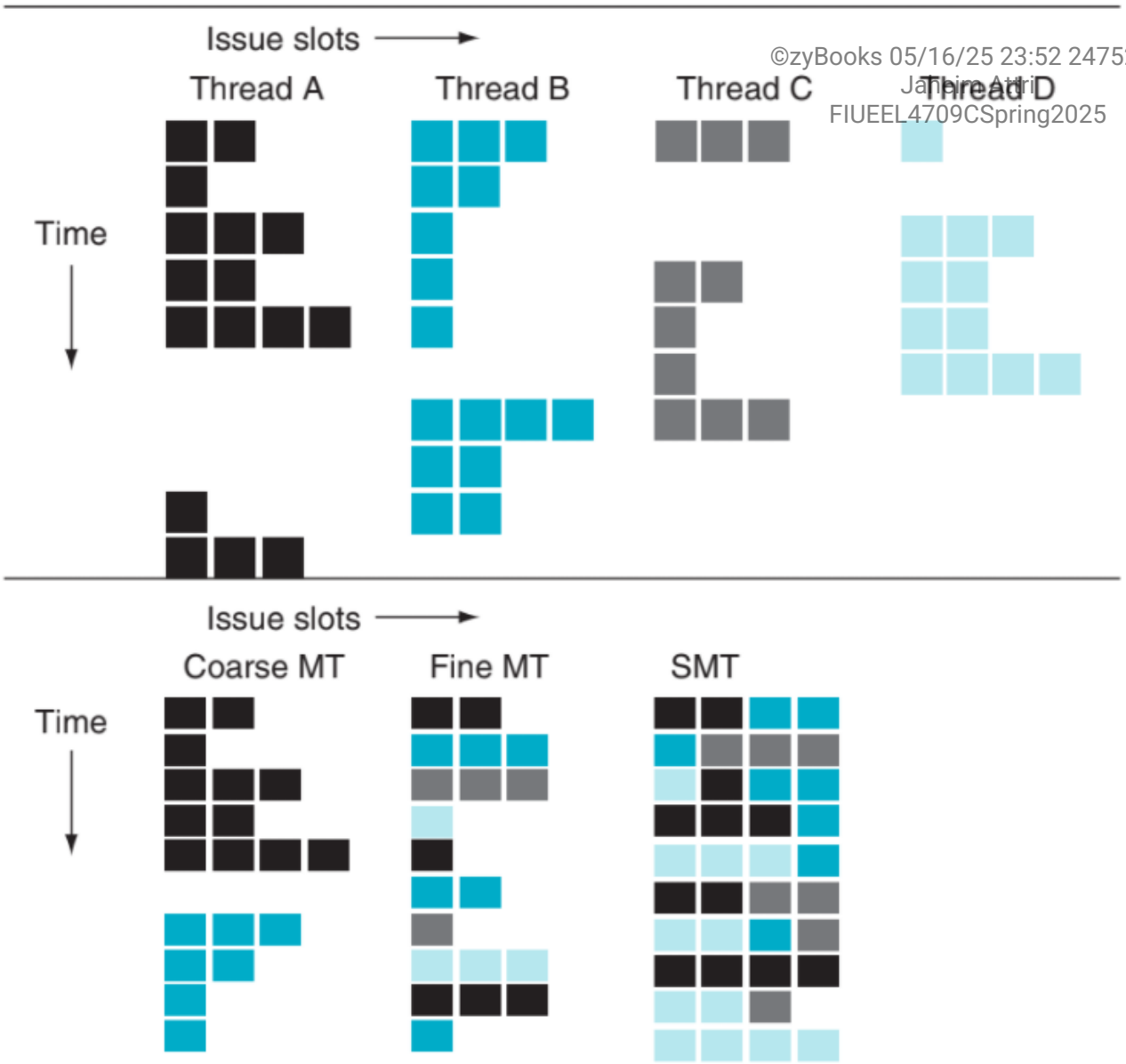
Simultaneous multithreading (***SMT***): A version of multithreading that lowers the cost of multithreading by utilizing the resources needed for multiple issue, dynamically scheduled microarchitecture.

Since SMT relies on the existing dynamic mechanisms, it does not switch resources every cycle. Instead, SMT is *always* executing instructions from multiple threads, leaving it up to the hardware to associate instruction slots and renamed registers with their proper threads.

Figure 6.4.1: How four threads use the issue slots of a superscalar processor in different approaches (COD Figure 6.5).

The four threads at the top show how each would execute running alone on a standard superscalar processor without multithreading support. The three examples at the bottom show how they would execute running together in three multithreading options. The horizontal dimension represents the instruction issue capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (white) box indicates

that the corresponding issue slot is unused in that clock cycle. The shades of gray and color correspond to four different threads in the multithreading processors. The additional pipeline start-up effects for coarse multithreading, which are not illustrated in this figure, would lead to further loss in throughput for coarse multithreading.



The figure above conceptually illustrates the differences in a processor's ability to exploit superscalar resources for the following processor configurations. The top portion shows how four threads would execute independently on a superscalar with no multithreading support. The bottom portion shows how the four threads could be combined to execute on the processor more efficiently using three multithreading options:

- A superscalar with coarse-grained multithreading
- A superscalar with fine-grained multithreading

- A superscalar with simultaneous multithreading

In the superscalar without hardware multithreading support, the use of issue slots is limited by a lack of **instruction-level parallelism**. In addition, a major stall, such as an instruction cache miss, can leave the entire processor idle.

In the coarse-grained multithreaded superscalar, the long stalls are partially hidden by switching to another thread that uses the resources of the processor. Although this reduces the number of completely idle clock cycles, the pipeline start-up overhead still leads to idle cycles, and limitations to ILP means all issue slots will not be used. In the fine-grained case, the interleaving of threads mostly eliminates idle clock cycles. Because only a single thread issues instructions in a given clock cycle, however, limitations in instruction-level parallelism still lead to idle slots within some clock cycles.

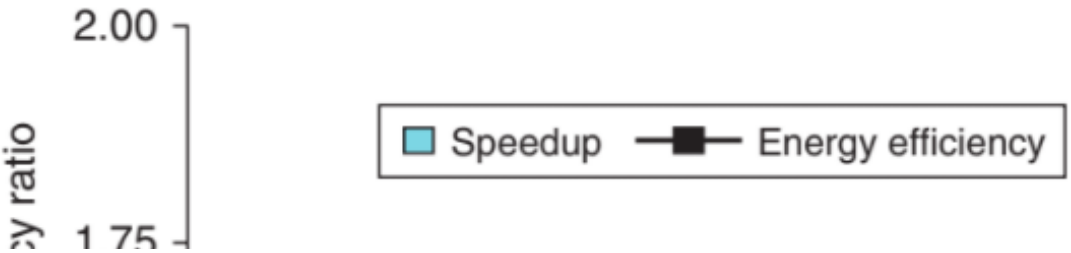
In the SMT case, thread-level parallelism and instruction-level parallelism are both exploited, with multiple threads using the issue slots in a single clock cycle. Ideally, the issue slot usage is limited by imbalances in the resource needs and resource availability over multiple threads. In practice, other factors can restrict how many slots are used. Although the figure above greatly simplifies the real operation of these processors, it does illustrate the potential performance advantages of multithreading in general and SMT in particular.

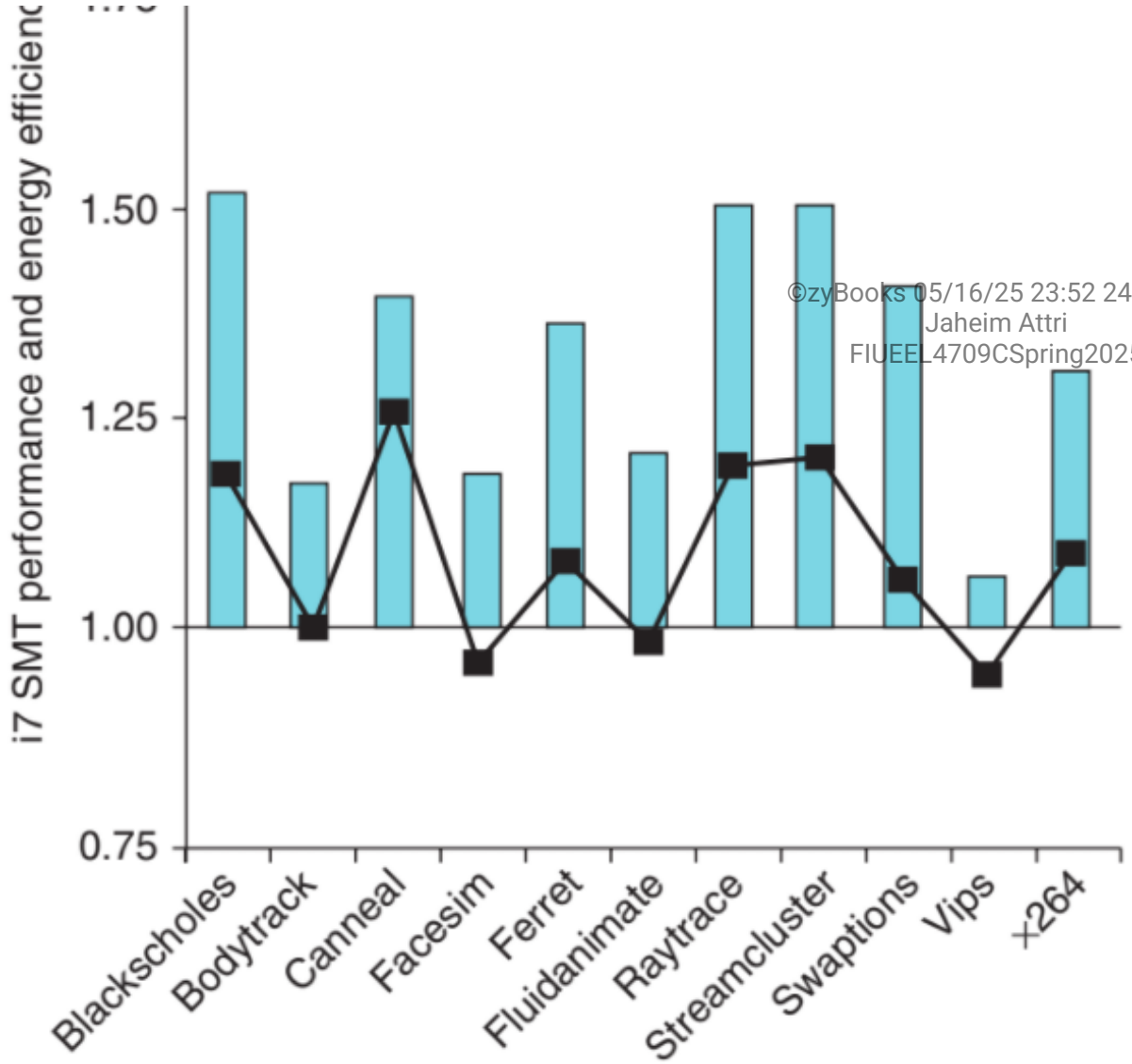
The figure below plots the performance and energy benefits of multithreading on a single processors of the Intel Core i7 960, which has hardware support for two threads, as does the the more recent i7 6700. The changes between the i7 920 and the 6700 are relatively small and are unlikely to significantly change the results in this figure. The average speed-up is 1.31, which is not bad given the modest extra resources for hardware multithreading. The average improvement in energy efficiency is 1.07, which is excellent. In general, you'd be happy with a performance speed-up being energy neutral.

Figure 6.4.2: The speed-up from using multithreading on one core on an i7 processor (COD Figure 6.6).

Processor averages 1.31 for the PARSEC benchmarks (see COD Section 6.9 (Communicating to the outside world: Cluster networking)) and the energy efficiency improvement is 1.07

This data was collected and analyzed by Esmaeilzadeh et. al. [2011].





Now that we have seen how multiple threads can utilize the resources of a single processor more effectively, we next show how to use them to exploit multiple processors.

PARTICIPATION ACTIVITY 6.4.1: Check Yourself: Hardware Multithreading.

1) Both multithreading and multicore rely on parallelism to get more efficiency from a chip.
   - ○ True
   - ○ False

2) Simultaneous multithreading (SMT) uses threads to improve resource utilization of a dynamically scheduled,

out-of-order processor.

- ○ True
- ○ False

(*1) This section is in original form.

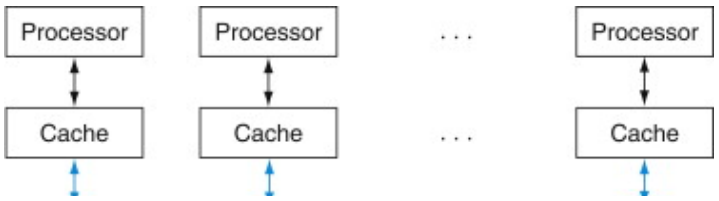
6.5 Multicore and other shared memory multiprocessors

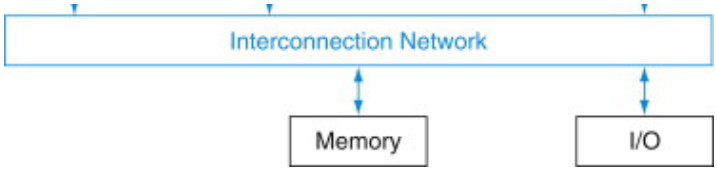
While hardware multithreading improved the efficiency of processors at modest cost, a big challenge has been to deliver on the traditional performance potential of Moore's Law by efficiently programming the increasing number of processors per chip.

Given the difficulty of rewriting old programs to run well on parallel hardware, a natural question is: what can computer designers do to simplify the task? One answer was to provide a single physical address space that all processors can share, so that programs need not concern themselves with where their data is, merely that programs may be executed in parallel. In this approach, all variables of a program can be made available at any time to any processor. The alternative is to have a separate address space per processor that requires that sharing must be explicit; we'll describe this option in the COD Section 6.8 (Clusters, warehouse scale computers, and other message-passing multiprocessors). When the physical address space is common then the hardware typically provides cache coherence to give a consistent view of the shared memory (see COD Section 5.8 (A common framework for memory hierarchy)).

As mentioned above, a *shared memory multiprocessor* (SMP) is one that offers the programmer a *single physical address space* across all processors—which is nearly always the case for multicore chips—although a more accurate term would have been shared-*address* multiprocessor. Processors communicate through shared variables in memory, with all processors capable of accessing any memory location via loads and stores. The figure below shows the classic organization of an SMP. Note that such systems can still run independent jobs in their own virtual address spaces, even if they all share a physical address space.

Figure 6.5.1: Classic organization of a shared memory multiprocessor (COD Figure 6.7).





Single address space multiprocessors come in two styles. In the first style, the latency to a word in memory does not depend on which processor asks for it. Such machines are called *uniform memory access (UMA)* multiprocessors. In the second style, some memory accesses are much faster than others, depending on which processor asks for which word, typically because main memory is divided and attached to different microprocessors or to different memory controllers on the same chip. Such machines are called *nonuniform memory access (NUMA)* multiprocessors. As you might expect, the programming challenges are harder for a NUMA multiprocessor than for a UMA multiprocessor, but NUMA machines can scale to larger sizes and NUMAs can have lower latency to nearby memory.

Uniform memory access (***UMA***): A multiprocessor in which latency to any word in main memory is about the same no matter which processor requests the access.

Nonuniform memory access (***NUMA***): A type of single address space multiprocessor in which some memory accesses are much faster than others depending on which processor asks for which word.

As processors operating in parallel will normally share data, they also need to coordinate when operating on shared data; otherwise, one processor could start working on data before another is finished with it. This coordination is called *synchronization*, which we saw in COD Chapter 2 (Instructions: Language of the Computer). When sharing is supported with a single address space, there must be a separate mechanism for synchronization. One approach uses a *lock* for a shared variable. Only one processor at a time can acquire the lock, and other processors interested in shared data must wait until the original processor unlocks the variable. COD Section 2.11 (Parallelism and instructions: synchronization) of COD Chapter 2 (Instructions: Language of the Computer) describes the instructions for locking in the MIPS instruction set.

Synchronization: The process of coordinating the behavior of two or more processes, which may be running on different processors.

Lock: A synchronization device that allows access to data to only one processor at a time.

Example 6.5.1: A simple parallel processing program for a shared address

space.

Suppose we want to sum 64,000 numbers on a shared memory multiprocessor computer with uniform memory access time. Let's assume we have 64 processors.

Answer

The first step is to ensure a balanced load per processor, so we split the set of numbers into subs of the same size. We do not allocate the subsets to a different memory space, since there is a sin memory space for this machine; we just give different starting addresses to each processor. Pn is number that identifies the processor, between 0 and 63. All processors start the program by runni loop that sums their subset of numbers:

```
sum[Pn] = 0;

for (i = 1000 * Pn; i < 1000 * (Pn + 1); i += 1)
   sum[Pn] += A[i];                 /* sum the assigned areas */
```

(Note the C code `i += 1` is just a shorter way to say `i = i + 1`.)

The next step is to add these 64 partial sums. This step is called a *reduction*, where we divide to conquer. Half of the processors add pairs of partial sums, and then a quarter add pairs of the new partial sums, and so on until we have the single, final sum. The figure below illustrates the hierarch nature of this reduction.

In this example, the two processors must synchronize before the "consumer" processor tries to re the result from the memory location written by the "producer" processor; otherwise, the consume may read the old value of the data. We want each processor to have its own version of the loop counter variable `i`, so we must indicate that it is a "private" variable. Here is the code (`half` is priv also):

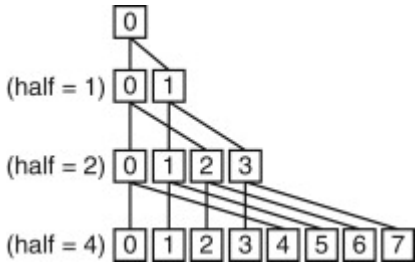
```
half = 64;                          /* 64 processors in multiprocessor

do
   synch();                         /* wait for partial sum completion
   if (half % 2 != 0 && Pn == 0)
      sum[0] += sum[half - 1];
                                    /* Conditional sum needed when half
                                       odd; Processor0 gets missing ele
      half = half / 2;              /* dividing line on who sums */
```

```
      if (Pn < half)
         sum[Pn] += sum[Pn + half];
while (half > 1);                    /* exit with final sum in Sum[0] */
```

Reduction: A function that processes a data structure and returns a single value.

Figure 6.5.2: The last four levels of a reduction that sums results from each processor, from bottom to top (COD Figure 6.8).

For all processors whose number i is less than half, add the sum produced by processor number (i + half) to its sum.



Hardware/Software Interface

Given the long-term interest in parallel programming, there have been hundreds of attempts to build parallel programming systems. A limited but popular example is *OpenMP*. It is just an *Application Programmer Interface* (API) along with a set of compiler directives, environment variables, and runtime library routines that can extend standard programming languages. It offers a portable, scalable, and simple programming model for shared memory multiprocessors. Its primary goal is to parallelize loops and to perform reductions.

Most C compilers already have support for OpenMP. The command to use the OpenMP API with the UNIX C compiler is just:

```
cc -fopenmp foo.c
```

OpenMP extends C using *pragmas*, which are just commands to the C macro preprocessor like `#define` and `#include`. To set the number of processors we want to use to be 64, as we wanted in the example above, we just use the command

```
#define P 64                          /* define a constant that
                                         we'll use a few times */
#pragma omp parallel num_threads(P)
```


That is, the runtime libraries should use 64 parallel threads.

To turn the sequential for loop into a parallel for loop that divides the work equally between all the threads that we told it to use, we just write (assuming `sum` is initialized to 0)

```
#pragma omp parallel for
for (Pn = 0; Pn < P; Pn += 1)
   for (i = 0; 1000 * Pn; i < 1000 * (Pn + 1); i += 1)
      sum[Pn] += A[i];              /* sum the assigned areas */
```

To perform the reduction, we can use another command that tells OpenMP what the reduction operator is and what variable you need to use to place the result of the reduction.

```
#pragma omp parallel for reduction(+ : FinalSum)
for (i = 0; i < P; i += 1)
   FinalSum += sum[i];              /* Reduce to a single number */
```

Note that it is now up to the OpenMP library to find efficient code to sum 64 numbers efficiently using 64 processors.

While OpenMP makes it easy to write simple parallel code, it is not very helpful with debugging, so many parallel programmers use more sophisticated parallel programming systems than OpenMP, just as many programmers today use more productive languages than C.

OpenMP: An API for shared memory multiprocessing in C, C++, or Fortran that runs on UNIX and Microsoft platforms. It includes compiler directives, a library, and runtime directives.

Given this tour of classic MIMD hardware and software, our next step is a more exotic tour of a type of MIMD architecture with a different heritage and thus a very different perspective on the parallel programming challenge.

PARTICIPATION ACTIVITY 6.5.1: Check yourself: Shared memory.

1) Shared memory multiprocessors cannot take advantage of task-level parallelism.
   - ○ True
   - ○ False

Elaboration

Some writers repurposed the acronym SMP to mean symmetric multiprocessor, to indicate that the latency from processor to memory was about the same for all processors. This shift was done to contrast them from large-scale NUMA multiprocessors, as both classes used a single address space. As clusters proved much more popular than large-scale NUMA multiprocessors, in this book we restore SMP to its original meaning, and use it to contrast against that use multiple address spaces, such as clusters.

Elaboration

An alternative to sharing the physical address space would be to have separate physical address spaces but share a common virtual address space, leaving it up to the operating system to handle communication. This approach has been tried, but it has too high an overhead to offer a practical shared memory abstraction to the performance-oriented programmer.

6.6 Introduction to graphics processing units

The original justification for adding SIMD instructions to existing architectures was that many microprocessors were connected to graphics displays in PCs and workstations, so an increasing fraction of processing time was used for graphics. As Moore's Law increased the number of transistors available to microprocessors, it therefore made sense to improve graphics processing.

A major driving force for improving graphics processing was the computer game industry, both on PCs and in dedicated game consoles such as the Sony PlayStation. The rapidly growing game market encouraged many companies to make increasing investments in developing faster graphics hardware, and this positive feedback loop led graphics processing to improve at a faster rate than general-purpose processing in mainstream microprocessors.

Given that the graphics and game community had different goals than the microprocessor development community, it evolved its own style of processing and terminology. As the graphics processors increased in power, they earned the name *Graphics Processing Units* or *GPUs* to distinguish themselves from CPUs.

For a few hundred dollars, anyone can buy a GPU today with hundreds of parallel floating-point units, which makes high-performance computing more accessible. The interest in GPU computing blossomed when this potential was combined with a programming language that made GPUs easier to program. Hence, many programmers of scientific and multimedia applications today are pondering whether to use GPUs or CPUs.

(This section concentrates on using GPUs for computing. To see how GPU computing combines with the traditional role of graphics acceleration, see COD Appendix C (Graphics and Computing GPUs).)

Here are some of the key characteristics as to how GPUs vary from CPUs:

- GPUs are accelerators that supplement a CPU, so they do not need to be able to perform all the tasks of a CPU. This role allows them to dedicate all their resources to graphics. It's fine for GPUs to perform some tasks poorly or not at all, given that in a system with both a CPU and a GPU, the CPU can do them if needed.
- The GPU problems sizes are typically hundreds of megabytes to gigabytes, but not hundreds of gigabytes to terabytes.

These differences led to different styles of architecture:

- Perhaps the biggest difference is that GPUs do not rely on multilevel caches to overcome the long latency to memory, as do CPUs. Instead, GPUs rely on hardware multithreading (COD Section 6.4 (Hardware multithreading)) to hide the latency to memory. That is, between the time of a memory request and the time that data arrives, the GPU executes hundreds or thousands of threads that are independent of that request.

- The GPU memory is thus oriented toward bandwidth rather than latency. There are even special graphics DRAM chips for GPUs that are wider and have higher bandwidth than DRAM chips for CPUs. In addition, GPU memories have traditionally had smaller main memories than conventional microprocessors. In 2020, GPUs typically have 4 to 16 GiB or less, while CPUs have 64 to 512 GiB. Finally, keep in mind that for general-purpose computation, you must include the time to transfer the data between CPU memory and GPU memory, since the GPU is a coprocessor.
- Given the reliance on many threads to deliver good memory bandwidth, GPUs can accommodate many parallel processors (MIMD) as well as many threads. Hence, each GPU processor is more highly multithreaded than a typical CPU, plus they have more processors.

Hardware/Software Interface

Although GPUs were designed for a narrower set of applications, some programmers wondered if they could specify their applications in a form that would let them tap the high potential performance of GPUs. After tiring of trying to specify their problems using the graphics APIs and languages, they developed C-inspired programming languages to allow them to write programs directly for the GPUs. An example is NVIDIA's CUDA (Compute Unified Device Architecture), which enables the programmer to write C programs to execute on GPUs, albeit with some restrictions. COD Appendix C (Graphics and Computing GPUs) gives examples of CUDA code. (OpenCL is a multi- company initiative to develop a portable programming language that provides many of the benefits of 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. These threads are blocked together and executed in groups of 32 at a time. A multithreaded processor inside a GPU executes these blocks of threads, and a GPU consists of 8 to 32 of these multithreaded processors.

An introduction to the NVIDIA GPU architecture

We use NVIDIA systems as our example as they are representative of GPU architectures. Specifically, we follow the terminology of the CUDA parallel programming language and use the Tesla as the example.

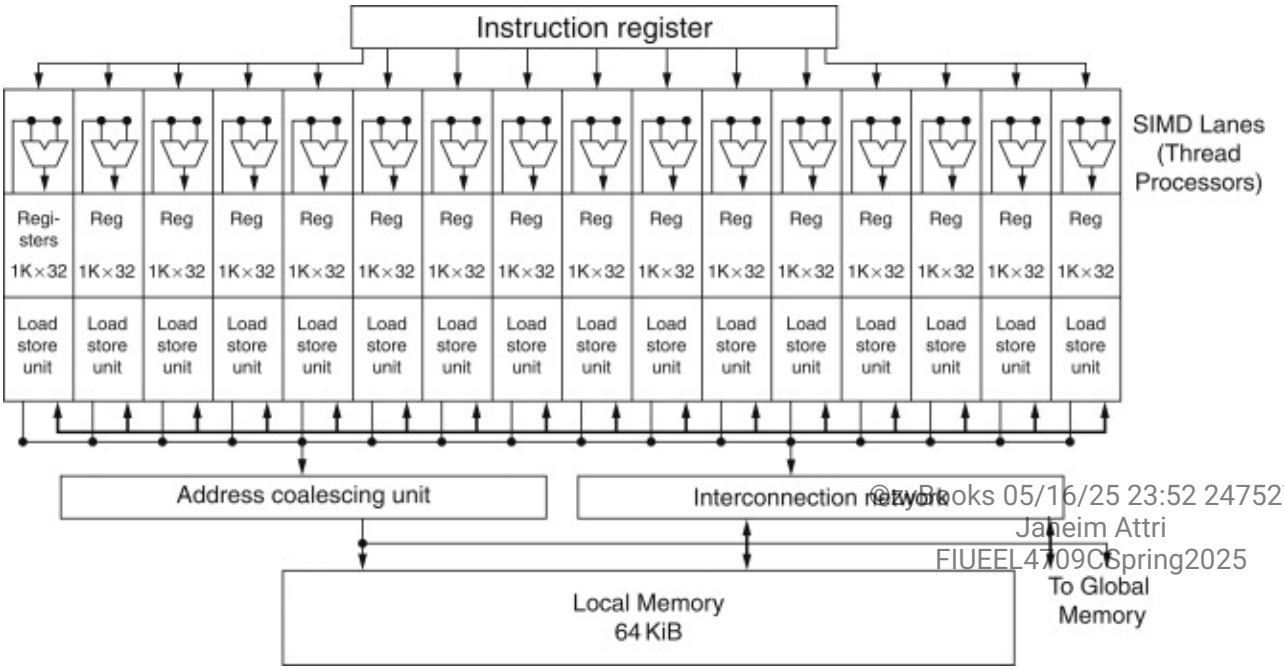
Like vector architectures, GPUs work well only with data-level parallel problems. Both styles have gather-scatter data transfers, and GPU processors have even more registers than do vector processors. Unlike most vector architectures, GPUs also rely on hardware multithreading within a single multi-threaded SIMD processor to hide memory latency (see COD Section 6.4 (Hardware multithreading)).

A multithreaded SIMD processor is similar to a Vector Processor, but the former has many parallel functional units instead of just a few that are deeply pipelined, as does the latter.

As mentioned above, a GPU contains a collection of multithreaded SIMD processors; that is, a GPU is a MIMD composed of multithreaded SIMD processors. For example, NVIDIA has four implementations of the Tesla at different price points with 15, 24, 56, or 80 multithreaded SIMD processors. To provide transparent scalability across models of GPUs with differing number of multithreaded SIMD processors, the Thread Block Scheduler hardware assigns blocks of threads to multithreaded SIMD processors. The figure below shows a simplified block diagram of a multithreaded SIMD processor.

Figure 6.6.1: Simplified block diagram of the datapath of a multithreaded SIMD Processor (COD Figure 6.9).

It has 16 SIMD lanes. The SIMD Thread Scheduler has many independent SIMD threads that it chooses from to run on this processor.



Dropping down one more level of detail, the machine object that the hardware creates, manages, schedules, and executes is a *thread of SIMD instructions*, which we will also call a *SIMD thread*. It is

a traditional thread, but it contains exclusively SIMD instructions. These SIMD threads have their own program counters and they run on a multithreaded SIMD processor. The *SIMD Thread Scheduler* includes a controller that lets it know which threads of SIMD instructions are ready to run, and then it sends them off to a dispatch unit to be run on the multithreaded SIMD processor. It is identical to a hardware thread scheduler in a traditional multithreaded processor (see COD Section 6.4 (Hardware multithreading)), except that it is scheduling threads of SIMD instructions. Thus, GPU hardware has two levels of hardware schedulers:

1. The *Thread Block Scheduler* that assigns blocks of threads to multithreaded SIMD processors, and
2. the SIMD Thread Scheduler *within* a SIMD processor, which schedules when SIMD threads should run.

The SIMD instructions of these threads are 32 wide, so each thread of SIMD instructions would compute 32 of the elements of the computation. Since 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 COD Section 6.3 (SISD, MIMD, SIMD, SPMD, and vector).

Elaboration

Each 32-wide thread of SIMD instructions is mapped to 16 SIMD Lanes, so each SIMD instruction in a thread of SIMD instructions takes two clock cycles to complete. Each thread of SIMD instructions is executed in lock step. Staying with the analogy of a SIMD processor as a vector processor, you could say that it has 16 lanes, and the vector length would be 32. This wide but shallow nature is why we use the term SIMD processor instead of vector processor, as it is more intuitive.

Since 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 single thread. Thus, using the terminology of COD Section 6.4 (Hardware multithreading), it uses fine-grained multithreading.

To hold these memory elements, a SIMD processor has an impressive 32,768 32-bit registers. 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 64 registers, so you might think of a SIMD Thread as having up to 64 vector registers, with each vector register having 32 elements and each element being 32 bits wide.

Since it has 16 SIMD Lanes, each contains 2048 registers. Each CUDA Thread gets one element of each of the vector registers. Note that a CUDA thread is just a vertical cut of a thread of SIMD instructions, corresponding to one element executed by one

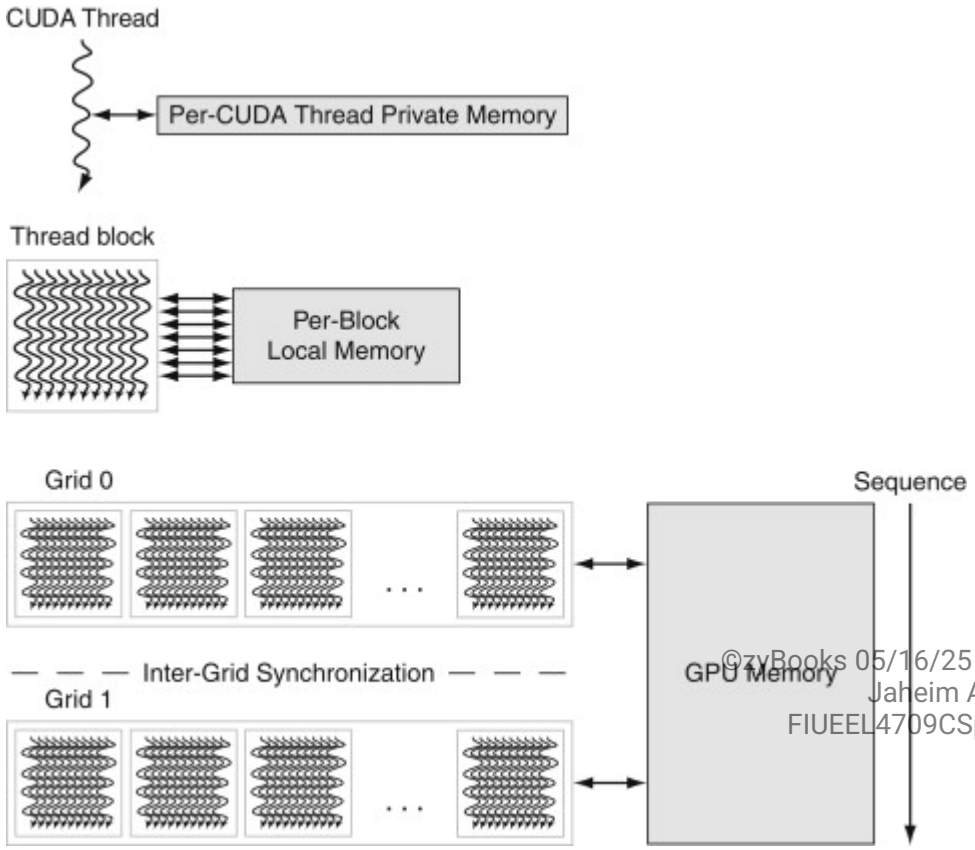
SIMD Lane. Beware that CUDA Threads are very different from POSIX threads; you can't make arbitrary system calls or synchronize arbitrarily in a CUDA Thread.

NVIDIA GPU memory structures

The figure below shows the memory structures of an NVIDIA GPU. We call the on-chip memory that is local to each multithreaded SIMD processor *Local Memory*. It is shared by the SIMD Lanes within a multithreaded SIMD processor, but this memory is not shared between multithreaded SIMD processors. We call the off-chip DRAM shared by the whole GPU and all thread blocks *GPU Memory*.

Figure 6.6.2: GPU Memory structures (COD Figure 6.10).

GPU Memory is shared by the vectorized loops. All threads of SIMD instructions within a thread block share Local Memory.



Rather than rely on large caches to contain the whole working sets of an application, GPUs

traditionally use smaller streaming caches and rely on extensive multithreading of threads of SIMD instructions to hide the long latency to DRAM, since their working sets can be hundreds of megabytes. Thus, they will not fit in the last level cache of a multicore microprocessor. Given the use of hardware multithreading to hide DRAM latency, the chip area used for caches in system processors is spent instead on computing resources and on the large number of registers to hold the state of the many threads of SIMD instructions.

Elaboration

While hiding memory latency is the underlying philosophy, note that the latest GPUs and vector processors have added caches. They are thought of as either bandwidth filters to reduce demands on GPU Memory or as accelerators for the few variables whose latency cannot be hidden by multithreading. Local memory for stack frames, function calls, and register spilling is a good match to caches, since latency matters when calling a function. Caches can also save energy, since on-chip cache accesses take much less energy than accesses to multiple, external DRAM chips.

Putting GPUs into perspective

At a high level, multicore computers with SIMD instruction extensions do share similarities with GPUs. The figure below summarizes the similarities and differences. Both are MIMDs 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 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. While GPUs support memory protection at the page level, they do not yet support demand paging.

Figure 6.6.3: Similarities and differences between multicore and Multimedia SIMD extensions and recent GPUs (COD Figure 6.11).

Feature	Multicore with SIMD	GPU
SIMD processors	8 to 32	15 to 128
SIMD lanes/processor	2 to 4	8 to 16
Multithreading hardware support for SIMD threads	2 to 4	16 to 32
Largest cache size	48 MiB	6 MiB
Size of memory address	64-bit	64-bit
Size of main memory	64 GiB to 1024 GiB	4 GiB to 16 GiB

Memory protection at level of page	Yes	Yes
Demand paging	Yes	No
Cache coherent	Yes	No

SIMD processors are also similar to vector processors. The multiple SIMD processors in GPUs act as independent MIMD cores, just as many vector computers have multiple vector processors. This view would consider the Volta V100 as an 80-core machine with hardware support for multithreading, where each core has 16 lanes. The biggest difference is multithreading, which is fundamental to GPUs and missing from most vector processors.

GPUs and CPUs do not go back in computer architecture genealogy to a common ancestor; there is no Missing Link that explains both. As a result of this uncommon heritage, GPUs have not used the terms common in the computer architecture community, which has led to confusion about what GPUs are and how they work. To help resolve the confusion, the figure below (from left to right) lists the more descriptive term used in this section, the closest term from mainstream computing, the official NVIDIA GPU term in case you are interested, and then a short description of the term. This "GPU Rosetta Stone" may help relate this section and ideas to more conventional GPU descriptions, such as those found in COD Appendix C (Graphics and Computing GPUs).

Figure 6.6.4: Quick guide to GPU terms (COD Figure 6.12).

We use the first column for hardware terms. Four groups cluster these 12 terms. From top to bottom: Program Abstractions, Machine Objects, Processing Hardware, and Memory Hardware.

Type	More descriptive name	Closest old term outside of GPUs	Official CUDA/ NVIDIA GPU term	Book definition
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel.
	Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via Local Memory.
	Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register.
Machine object	A Thread of SIMD Instructions	Thread of Vector Instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask.
	SIMD Instruction	Vector Instruction	PTX Instruction	A single SIMD instruction executed across SIMD Lanes.
re	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

Processing hardware				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.
	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).

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?

GPUs are the first example of accelerators justified by improving the performance of a single domain, in this case computer graphics. The next section gives more examples, with the domain of machine learning getting the most attention.

PARTICIPATION ACTIVITY 6.6.1: Check yourself: GPUs.

1) GPUs rely on graphics DRAM chips to reduce memory latency and thereby increase performance on graphics applications.
   - ○ True
   - ○ False

6.7 Domain specific architectures

The combination of the slowing of Moore's Law, the end of Dennard Scaling, and the practical limits to multicore performance due to Amdahl's Law moved the prevailing wisdom that the only path left to improved performance and energy efficiency is *Domain Specific Architectures (DSAs)*. Like GPUs, DSAs do only a narrow range of tasks, but they do them extremely well. Thus, just as the field switched from uniprocessors to multiprocessors in the past decade out of necessity, desperation is the reason architects are now working on DSAs.

The new normal is that a computer will consist of standard processors to run conventional large programs such as operating systems along with domain-specific processors. We expect that computers will be much more heterogeneous than the homogeneous multicore chips of the past. This section new to this book is based on a new 80-page chapter on DSAs in *Computer Architecture: A Quantitative Approach*, sixth edition, if you're interested in going into greater depth.

DSAs follow five principles:

1. *Use dedicated memories to minimize the distance over which data is moved*. The many levels of caches in general-purpose microprocessors use a great deal of area and energy trying to move data optimally for a program. For example, a two-way set associative cache uses 2.5 times as much energy as an equivalent software-controlled scratchpad memory. By definition, the compiler writers and programmers of DSAs understand their domain, so there is no need for the hardware to try to move data for them. Instead, data movement is reduced with software-controlled memories that are dedicated to and tailored for specific functions within the domain.
2. *Invest the resources saved from dropping advanced microarchitectural optimizations into more arithmetic units or bigger memories*. Architects turned the bounty from Moore's Law into the resource-intensive optimizations for CPUs and GPUs: out-of-order execution, speculation, multithreading, multiprocessing, prefetching, multi-level caches, and so on. Given the superior understanding of program execution in these narrower domains, these resources are much better spent on more processing units or larger on-chip memories.
3. *Use the easiest form of parallelism that matches the domain*. Target domains for DSAs almost always have inherent parallelism. A key decision for a DSA is how to take advantage of that parallelism and how to expose it to software. The goal is to design the DSA around the natural granularity of the parallelism of the domain and expose that parallelism simply in the programming model. For example, with respect to data-level parallelism, if SIMD works in the domain, it's certainly easier for the programmer and the compiler writer than MIMD. Similarly, if VLIW can express the instruction-level parallelism for the domain, the design can be smaller and more energy-efficient than out-of-order execution.
4. *Reduce data size and type to the simplest needed for the domain*. Applications in many domains are memory-bound, so you can increase the effective memory bandwidth and on-chip memory utilization by using narrower data types. Narrower and simpler data also lets you pack more arithmetic units into the same chip area or in the same energy budget.
5. *Use a domain-specific programming language to port code to the DSA*. A classic challenge for special purpose architectures is getting applications to run on your novel architecture. Fortunately, domain-specific programming languages became popular even before architects were forced to switch their attention to DSAs, such as Halide for vision processing and TensorFlow for machine learning. Raising the level of programming abstraction makes porting applications to a DSA much more feasible.

Examples of domains that have been accelerated beyond graphics include bioinformatics, image processing, and simulation, but the most popular by far has been *Artificial intelligence (AI)*. Instead

of building AI as a large set of logical rules, in the past decade the focus switched to *machine learning* (ML) from example data as the most promising path to AI. The amount of data and computation needed to learn was much greater than thought. The warehouse scale computers (WSCs) of this century, which harvest and store peta-bytes of information found on the Internet from the billions of users and their smartphones, supply the ample data. We also underestimated the amount of computation needed to learn from the massive data, but GPUs--which have excellent single-precision floating-point cost-performance--embedded in the thousands of servers of WSCs deliver sufficient computing.

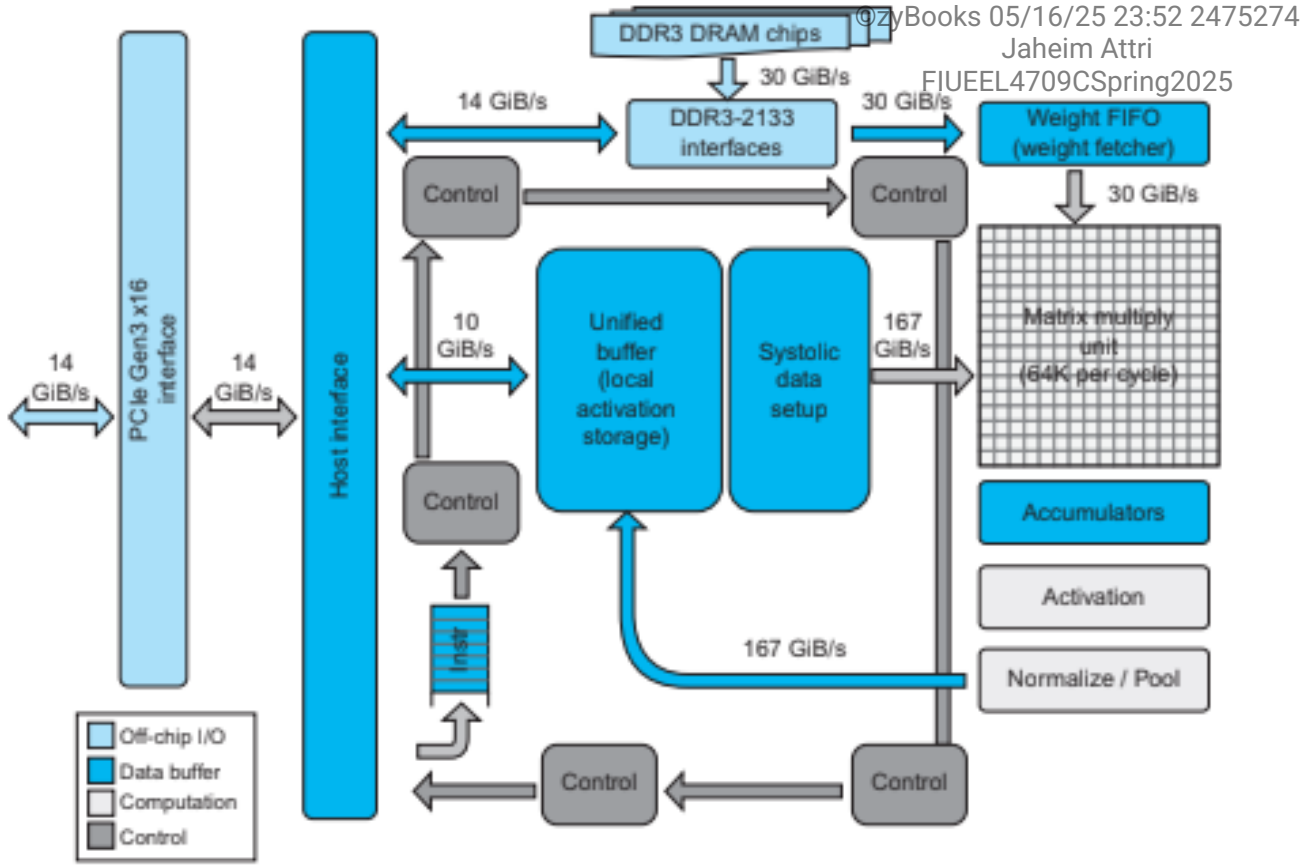
One part of machine learning, called *deep neural networks* (DNNs), has been the ML star since 2012. There seems to be a new breakthrough enabled by DNNs publicized almost every month, such as in object recognition, language translation, and enabling a computer program for the first time to beat a human champion at Go.

A prominent example of a DNN DSA is Google's Tensor Processing Unit (TPUv1). Starting as far back as 2006, Google engineers had discussions about deploying GPUs, FPGAs, or custom chips in their data centers. They concluded that the few applications that could run on special hardware could be done virtually for free using the excess capacity of the large data centers, and it's hard to improve on free. The conversation changed in 2013 when it was projected that if people used voice search for three minutes a day using speech recognition DNNs, it would have required Google's data centers to double in order to meet computation demands. That would be very expensive and time consuming to satisfy with conventional CPUs. Google started a high-priority project to quickly produce a custom chip for DNNs. The goal was to improve cost-performance tenfold over CPUs or GPUs. Given this urgent mandate, the TPU was designed, verified, built, and deployed in data centers in only 15 months. If you use Google applications, you've been using TPUv1s, as they have been deployed since 2015.

The figure below shows the block diagram of the TPUv1. The internal blocks are typically connected together by 256-byte-wide paths. Starting in the upper-right corner, the Matrix Multiply Unit is the heart of the TPU. It follows the DSA guideline of investing the resources saved from dropping CPU features into more arithmetic units, as it contains an array of 256x256 ALUs. That is 250 times as many ALUs as a contemporary server CPU and 25 times as many as a contemporary GPU. Using SIMD parallelism for the 65,536 ALUs follows the guideline of using the simplest form of parallelism to fit the domain. Moreover, TPUv1 reduces the data size and type to 8-bit and 16-bit integers from 32-bit floating point type used in the contemporary GPU, which is sufficient for this DNN domain. Following the guideline utilizing dedicated memories, the matrix unit products are collected in the 4 MiB of Accumulators and the intermediate results are held in the 24 MiB Unified Buffer, which can serve as inputs to the Matrix Multiply Unit. TPUv1 has almost four times the on-chip memory as the equivalent GPU. Finally, it is programmed using TensorFlow, which simplifies porting DNN applications to this DSA.

Figure 6.7.1: TPUv1 Block Diagram (COD figure 6.13).

The main computation part is the Matrix Multiply Unit in the upper-right corner. Its inputs are the Weight FIFO and the Unified Buffer and its output is the Accumulators. The 24 MiB Unified Buffer is almost a third of the TPUv1 die, and the Matrix Multiply Unit with 65,536 multiple-accumulate ALUs is a quarter, so the datapath is nearly two-thirds of the TPUv1 die. For CPUs, multi-level caches are often two-thirds of the die



The TPUv1 clock rate 700 MHz, which despite the modest clock rate gives 65,536 ALU yield a peak performance of 90 Tera-operations per second. The die area is less than half the size of a contemporary CPU or GPU and at 75 watts it's less than half their power.

Using the average of six production DNN applications, TPUv1 is 29.2 times as fast as a contemporary CPU and 15.3 times as fast as a contemporary GPU. In a datacenter we care about cost-performance as well as performance. The best measure of datacenter cost is total cost of ownership (TCO): the cost of purchase plus the cost of operation over several years for power, cooling, and space. Indeed, the original goal for TPUv1 was ten times the performance per TCO dollar of CPUs or GPUs. Alas, TCO numbers are closely guarded secrets and so unavailable for comparisons. The good news is that TCO is correlated with power, which is available. TPUv1 has 29 times the performance per Watt of the contemporary GPUs and 83 times the performance per Watt of contemporary CPUs, thereby overshooting its original target.

We return to more traditional architectures in the next section, introducing parallel processors

where each processor has its own private address space, which makes it much easier to build much larger systems. The Internet services that you use every day depend on these large-scale systems, and Google does indeed use these large-scale systems to deploy its TPUv1s.

PARTICIPATION ACTIVITY 6.7.1: Check yourself: DSAs.

1) DSAs are more effective than CPUs or GPUs in their domains primarily because you can justify using a much larger die for a domain.
   - ○ True
   - ○ False

6.8 Clusters, warehouse scale computers, and other message-passing multiprocessors

The alternative approach to sharing an address space is for the processors to each have their own private physical address space. The figure below shows the classic organization of a multiprocessor with multiple private address spaces. This alternative multiprocessor must communicate via explicit *message passing*, which traditionally is the name of such style of computers. Provided the system has routines to *send* and *receive messages*, coordination is built in with message passing, since one processor knows when a message is sent, and the receiving processor knows when a message arrives. If the sender needs confirmation that the message has arrived, the receiving processor can then send an acknowledgment message back to the sender.

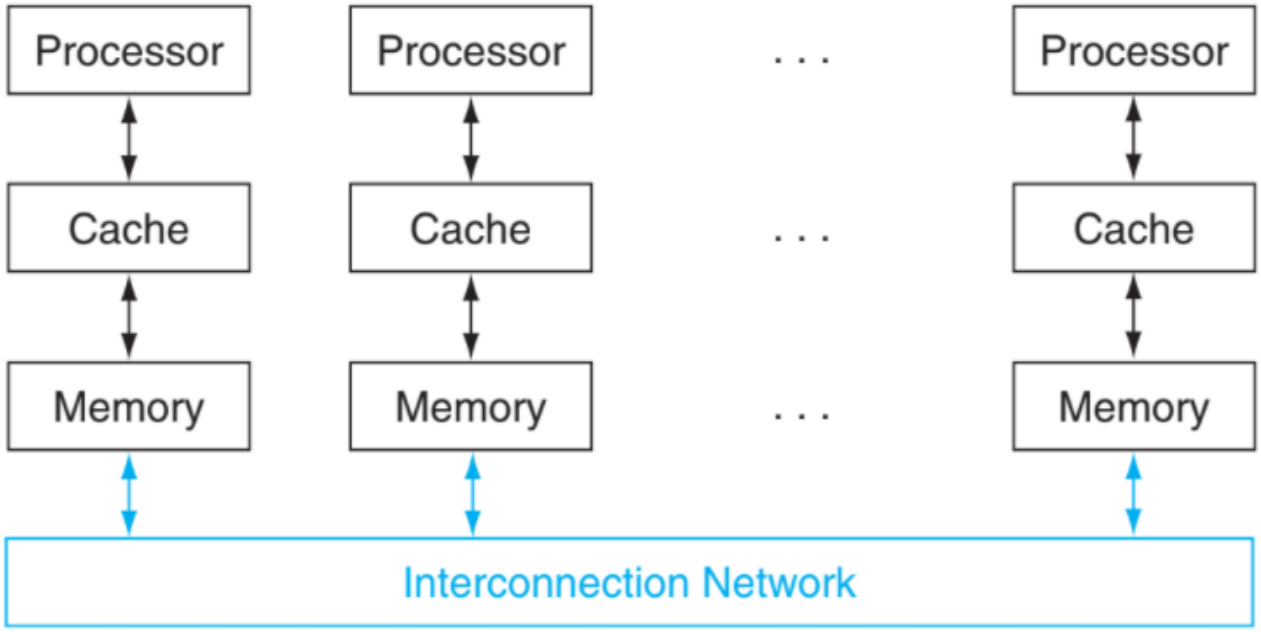
Message passing: Communicating between multiple processors by explicitly sending and receiving information.

Send message routine: A routine used by a processor in machines with private memories to pass a message to another processor.

Receive message routine: A routine used by a processor in machines with private memories to accept a message from another processor.

Figure 6.8.1: Classic organization of a multiprocessor with multiple private address spaces, traditionally called a message-passing multiprocessor (COD Figure 6.13).

Note that unlike the SMP in COD Figure 6.7 (Classic organization of a shared memory multiprocessor), the interconnection network is not between the caches and memory but is instead between processor-memory nodes.



There have been several attempts to build large-scale computers based on high-performance message-passing networks, and they do offer better absolute communication performance than clusters built using local area networks. Indeed, many supercomputers today use custom networks. The problem is that they are much more expensive than local area networks like Ethernet. Few applications today outside of high performance computing can justify the higher communication performance, given that costs are usually much higher.

Hardware/Software Interface

Computers that rely on message passing for communication rather than cache coherent shared memory are much easier for hardware designers to build (see COD Section 5.8 (A common framework for memory hierarchy)). There is an advantage for programmers as well, in that communication is explicit, which means there are fewer performance surprises than with the implicit communication in cache-coherent shared memory computers. The downside for programmers is that it's

harder to port a sequential program to a message-passing computer, since every communication must be identified in advance or the program doesn't work. Cache-coherent shared memory allows the hardware to figure out what data needs to be communicated, which makes porting easier. There are differences of opinion as to which is the shortest path to high performance, given the pros and cons of implicit communication, but there is no confusion in the marketplace today. Multicore microprocessors use shared physical memory and nodes of a cluster communicate with each other using message passing.

Some concurrent applications run well on parallel hardware, independent of whether it offers shared addresses or message passing. In particular, task-level parallelism and applications with little communication—like Web search, mail servers, and file servers—do not require shared addressing to run well. As a result, *clusters* have become the most widespread example today of the message-passing parallel computer. Given the separate memories, each node of a cluster runs a distinct copy of the operating system. In contrast, the cores inside a microprocessor are connected using a high-speed network inside the chip, and a multichip shared-memory system uses the memory interconnect for communication. The memory interconnect has higher bandwidth and lower latency, allowing much better communication performance for shared memory multiprocessors.

Clusters: Collections of computers connected via I/O over standard network switches to form a message-passing multiprocessor.

The weakness of separate memories for user memory from a parallel programming perspective turns into a strength in system dependability (see COD Section 5.5 (Dependable memory hierarchy)). Since a cluster consists of independent computers connected through a local area network, it is much easier to replace a computer without bringing down the system in a cluster than in a shared memory multiprocessor. Fundamentally, the shared address means that it is difficult to isolate a processor and replace it without heroic work by the operating system and in the physical design of the server. It is also easy for clusters to scale down gracefully when a server fails, thereby improving **dependability**. Since the cluster software is a layer that runs on top of the local operating systems running on each computer, it is much easier to disconnect and replace a broken computer.

Given that clusters are constructed from whole computers and independent, scalable networks, this isolation also makes it easier to expand the system without bringing down the application that runs on top of the cluster.

Their lower cost, higher availability, and rapid, incremental expandability make clusters attractive to service Internet providers, despite their poorer communication performance when compared to large-scale shared memory multiprocessors. The search engines that hundreds of millions of us

use every day depend upon this technology. Amazon, Facebook, Google, Microsoft, and others all have multiple datacenters each with clusters of tens of thousands of servers. Clearly, the use of multiple processors in Internet service companies has been hugely successful.

> "Anyone can build a fast CPU. The trick is to build a fast system.
> *Seymour Cray, considered the father of the supercomputer.*

Warehouse-scale computers

Internet services, such as those described above, necessitated the construction of new buildings to house, power, and cool 50,000 servers. Although they may be classified as just large clusters, their architecture and operation are more sophisticated. They act as one giant computer and cost on the order of $150M for the building, the electrical and cooling infrastructure, the servers, and the networking equipment that connects and houses 50,000 servers. We consider them a new class of computer, called *Warehouse-Scale Computers* (WSC).

Hardware/Software Interface

The most popular framework for batch processing in a WSC is MapReduce [Dean, 2008] and its open-source twin Hadoop. Inspired by the Lisp functions of the same name, Map first applies a programmer-supplied function to each logical input record. Map runs on thousands of servers to produce an intermediate result of key-value pairs. Reduce collects the output of those distributed tasks and collapses them using another programmer-defined function. With appropriate software support, both are highly parallel yet easy to understand and to use. Within 30 minutes, a novice programmer can run a MapReduce task on thousands of servers.

For example, one MapReduce program calculates the number of occurrences of every English word in a large collection of documents. Below is a simplified version of that program, which shows just the inner loop and assumes just one occurrence of all English words found in a document:


```
map(String key, String value):
     // key: document name
     // value: document contents
     for each word w in value:
     EmitIntermediate(w, "1");   // Produce list of all words
reduce(String key, Iterator values):
// key: a word
```

```
// values: a list of counts
     int result = 0;
     for each v in values:
     result += ParseInt(v);       // get integer from key-value pair
     Emit(AsString(result));
```

The function `EmitIntermediate` used in the Map function emits each word in the document and the value one. Then the Reduce function sums all the values per word for each document using `ParseInt()` to get the number of occurrences per word in all documents. The MapReduce runtime environment schedules map tasks and reduce tasks to the servers of a WSC.

At this extreme scale, which requires innovation in power distribution, cooling, monitoring, and operations, the WSC is a modern descendant of the 1970s supercomputers—making Seymour Cray the godfather of today's WSC architects. His extreme computers handled computations that could be done nowhere else, but were so expensive that only a few companies could afford them. This time the target is providing information technology for the world instead of high performance computing for scientists and engineers. Hence, WSCs surely play a more important societal role today than Cray's supercomputers did in the past.

While they share some common goals with servers, WSCs have three major distinctions:

1. *Ample, easy parallelism:* A concern for a server architect is whether the applications in the targeted marketplace have enough parallelism to justify the amount of parallel hardware and whether the cost is too high for sufficient communication hardware to exploit this parallelism. A WSC architect has no such concern. First, batch applications like MapReduce benefit from the large number of independent data sets that need independent processing, such as billions of Web pages from a Web crawl. Second, interactive Internet service applications, also known as *Software as a Service (SaaS)*, can benefit from millions of independent users of interactive Internet services. Reads and writes are rarely dependent in SaaS, so SaaS rarely needs to synchronize. For example, search uses a read-only index and email is normally reading and writing independent information. We call this type of easy parallelism *Request-Level Parallelism*, as many independent efforts can proceed in parallel naturally with little need for communication or synchronization.

2. *Operational Costs Count:* Traditionally, server architects design their systems for peak performance within a cost budget and worry about energy only to make sure they don't exceed the cooling capacity of their enclosure. They usually ignored operational costs of a server, assuming that they pale in comparison to purchase costs. WSC have longer lifetimes—the building and electrical and cooling infrastructure are often amortized over 10 or more

years—so the operational costs add up: energy, power distribution, and cooling represent more than 30% of the costs of a WSC over 10 years.

3. *Scale and the Opportunities/Problems Associated with Scale*: To construct a single WSC, you must purchase 50,000 servers along with the supporting infrastructure, which means volume discounts. Hence, WSCs are so massive internally that you get economy of scale even if there are not many WSCs. These economies of scale led to *cloud computing*, as the lower per unit costs of a WSC meant that cloud companies could rent servers at a profitable rate and still be below what it costs outsiders to do it themselves. The flip side of the economic opportunity of scale is the need to cope with the failure frequency of scale. Even if a server had a Mean Time To Failure of an amazing 25 years (200,000 hours), the WSC architect would need to design for 5 server failures every day. COD Section 5.15 (Fallacies and pitfalls) mentioned annualized disk failure rate (AFR) was measured at Google at 2% to 4%. If there were 4 disks per server and their annual failure rate was 4%, the WSC architect should expect to see one disk fail every *hour*. Thus, fault tolerance is even more important for the WSC architect than the server architect.

Software as a service (***SaaS***) : Rather than selling software that is installed and run on customers' own computers, software is run at a remote site and made available over the Internet typically via a Web interface to customers. SaaS customers are charged based on use versus on ownership.

The economies of scale uncovered by WSC have realized the long dreamed of goal of computing as a utility. Cloud computing means anyone anywhere with good ideas, a business model, and a credit card can tap thousands of servers to deliver their vision almost instantly around the world.

To put the growth rate of cloud computing into perspective, in 2012 Amazon Web Services (AWS) announced that it adds enough new server capacity *every day* to support all of Amazon's global infrastructure as of 2003, when Amazon was a $5.2Bn annual revenue enterprise with 6,000 employees. In 2020, the majority of the profits of Amazon come from cloud computing despite it representing only 10% of Amazon's revenues. AWS is growing at 40% annually.

Now that we understand the importance of message-passing multiprocessors, especially for cloud computing, we next cover ways to connect the nodes of a WSC together. Thanks to the increasing number of cores per chip, we now need networks inside a chip as well, so these topologies are important in the small as well as in the large.

Elaboration

The MapReduce framework shuffles and sorts the key-value pairs at the end of the Map phase to produce groups that all share the same key. These groups are then passed to the Reduce phase.

Elaboration

Another form of large scale computing is grid computing, where the computers are spread across large areas, and then the programs that run across them must communicate via long haul networks. The most popular and unique form of grid computing was pioneered by the SETI@home project. As millions of PCs are idle at any one time doing nothing useful, they could be harvested and put to good uses if someone developed software that could run on those computers and then gave each PC an independent piece of the problem to work on. The first example was the Search for ExtraTerrestrial Intelligence (SETI), which was launched at UC Berkeley in 1999. Over 5 million computer users in more than 200 countries have signed up for SETI@home, with more than 50% outside the US. In June 2013, the average performance of the SETI@home grid was 668 PetaFLOPS, faster than the best supercomputer of 2013.

PARTICIPATION ACTIVITY 6.8.1: Check Yourself: Clusters.

1) Like SMPs, message-passing computers rely on locks for synchronization.
   - ○ True
   - ○ False

2) Clusters have separate memories and thus need many copies of the operating system.
   - ○ True
   - ○ False

6.9 Introduction to multiprocessor network topologies

(Original section[1])

Multicore chips require on-chip networks to connect cores together, and clusters require local area networks to connect servers together. This section reviews the pros and cons of different interconnection network topologies.

Network costs include the number of switches, the number of links on a switch to connect to the network, the width (number of bits) per link, and length of the links when the network is mapped into silicon. For example, some cores or servers may be adjacent and others may be on the other side of the chip or the other side of the datacenter. Network performance is multifaceted as well. It includes the latency on an unloaded network to send and receive a message, the throughput in terms of the maximum number of messages that can be transmitted in a given time period, delays caused by contention for a portion of the network, and variable performance depending on the pattern of communication. Another obligation of the network may be fault tolerance, since systems may be required to operate in the presence of broken components. Finally, in this era of energy-limited systems, the energy efficiency of different organizations may trump other concerns.

Networks are normally drawn as graphs, with each edge of the graph representing a link of the communication network. In the figures in this section, the processor-memory node is shown as a black square and the switch is shown as a colored circle. We assume here that all links are *bidirectional*; that is, information can flow in either direction. All networks consist of *switches* whose links go to processor-memory nodes and to other switches. The first network connects a sequence of nodes together:

This topology is called a *ring*. Since some nodes are not directly connected, some messages will have to hop along intermediate nodes until they arrive at the final destination.

Unlike a bus—a shared set of wires that allows broadcasting to all connected devices—a ring is capable of many simultaneous transfers.

Because there are numerous topologies to choose from, performance metrics are needed to distinguish these designs. Two are popular. The first is total *network bandwidth*, which is the bandwidth of each link multiplied by the number of links. This represents the peak bandwidth. For the ring network above, with P processors, the total network bandwidth would be P times the bandwidth of one link; the total network bandwidth of a bus is just the bandwidth of that bus.

To balance this best bandwidth case, we include another metric that is closer to the worst case: the

Network bandwidth: Informally, the peak transfer rate of a network; can refer to the speed of a single link or the collective transfer rate of all links in the network.

bisection bandwidth. This metric is calculated by dividing the machine into two halves. Then you sum the bandwidth of the links that cross that imaginary dividing line. The bisection bandwidth of a ring is two times the link bandwidth. It is one times the link bandwidth for the bus. If a single link is as fast as the bus, the ring is only twice as fast as a bus in the worst case, but it is P times faster in the best case.

Bisection bandwidth: The bandwidth between two equal parts of a multiprocessor. This measure is for a worst case split of the multiprocessor.

Since some network topologies are not symmetric, the question arises of where to draw the imaginary line when bisecting the machine. Bisection bandwidth is a worst-case metric, so the answer is to choose the division that yields the most pessimistic network performance. Stated alternatively, calculate all possible bisection bandwidths and pick the smallest. We take this pessimistic view because parallel programs are often limited by the weakest link in the communication chain.

Fully connected network: A network that connects processor-memory nodes by supplying a dedicated communication link between every node.

At the other extreme from a ring is a *fully connected network*, where every processor has a bidirectional link to every other processor. For fully connected networks, the total network bandwidth is $P \times (P - 1)/2$, and the bisection bandwidth is $(P/2)^2$.

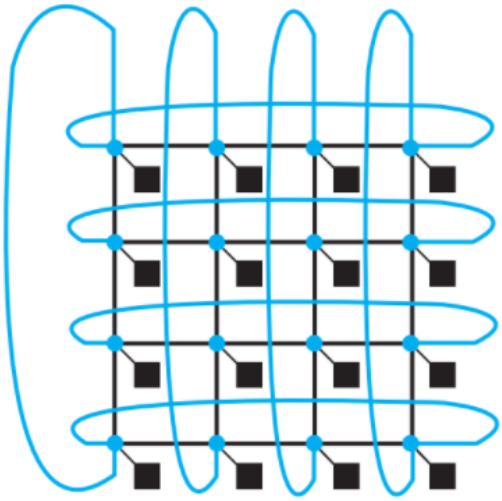
The tremendous improvement in performance of fully connected networks is offset by the tremendous increase in cost. This consequence inspires engineers to invent new topologies that are between the cost of rings and the performance of fully connected networks. The evaluation of success depends in large part on the nature of the communication in the workload of parallel programs run on the computer.

The number of different topologies that have been discussed in publications would be difficult to count, but only a few have been used in commercial parallel processors. The figure below illustrates two of the popular topologies.

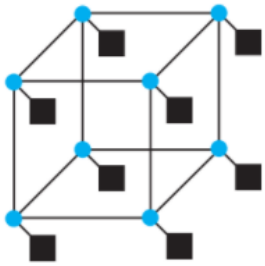
Figure 6.9.1: Network topologies that have appeared in commercial parallel processors (COD Figure 6.15).

The colored circles represent switches and the black squares represent processor-memory nodes. Even though a switch has many links, generally only one goes to the

processor. The Boolean n-cube topology is an n-dimensional interconnect with 2^n nodes, requiring n links per switch (plus one for the processor) and thus n nearest-neighbor nodes. Frequently, these basic topologies have been supplemented with extra arcs to improve performance and reliability.



a. 2-D grid or torus of 16 nodes



b. *n*-cube tree of 8 nodes ($8 = 2^3$ so $n = 3$)

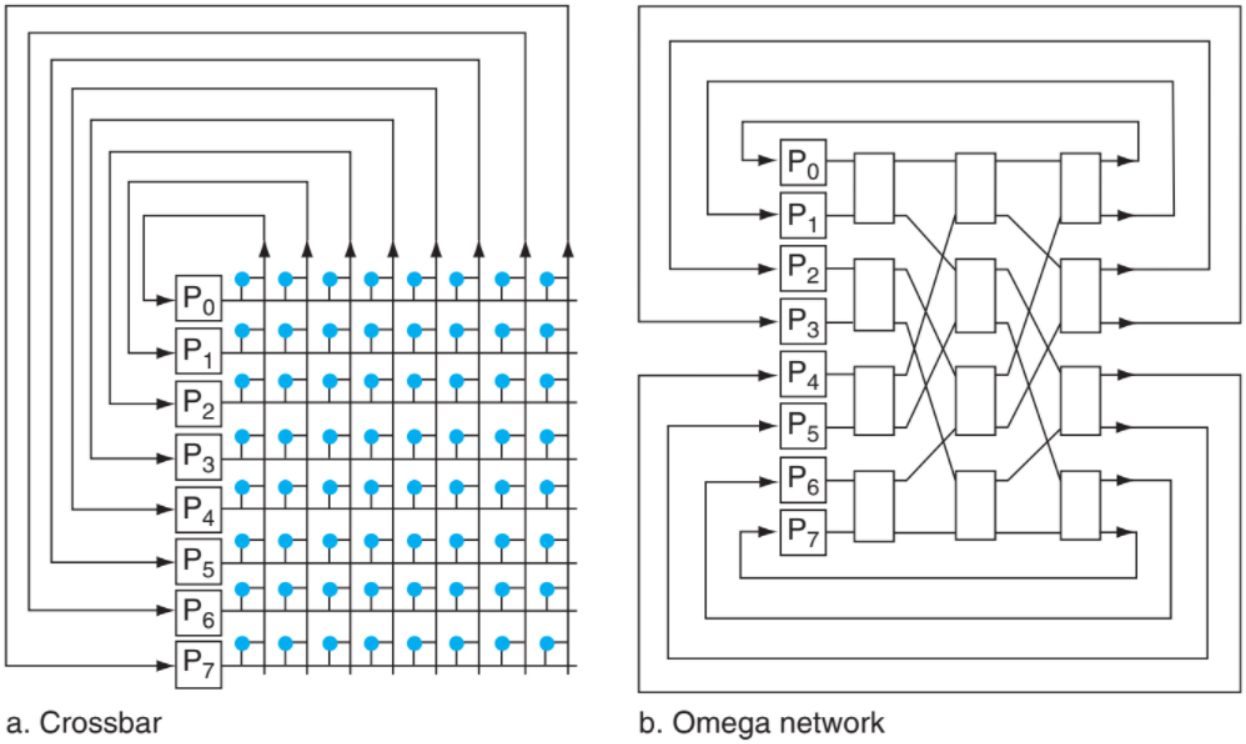
An alternative to placing a processor at every node in a network is to leave only the switch at some of these nodes. The switches are smaller than processor-memory-switch nodes, and thus may be packed more densely, thereby lessening distance and increasing performance. Such networks are frequently called *multistage networks* to reflect the multiple steps that a message may travel. Types of multistage networks are as numerous as single-stage networks; the figure below illustrates two of the popular multistage organizations. A *fully connected* or *crossbar network* allows any node to communicate with any other node in one pass through the network. An Omega network uses less hardware than the crossbar network ($2n \log_2 n$ versus n^2 switches), but contention can occur between messages, depending on the pattern of communication. For example, the Omega network in the figure below cannot send a message from P_0 to P_6 at the same time that it sends a message from P_1 to P_4.

Multistage network: A network that supplies a small switch at each node.

Crossbar network: A network that allows any node to communicate with any other node in one pass through the network.

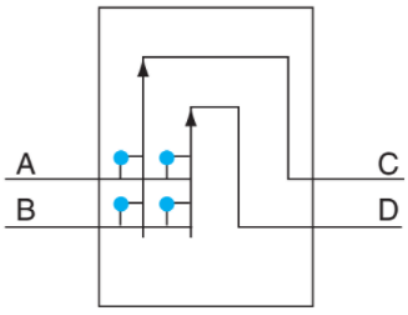
Figure 6.9.2: Popular multistage network topologies for eight nodes (COD Figure 6.16).

The switches in these drawings are simpler than in earlier drawings because the links are unidirectional; data comes in at the left and exits out the right link. The switch box in c can pass A to C and B to D or B to C and A to D. The crossbar uses n^2 switches, where n is the number of processors, while the Omega network uses $2n \log_2 n$ of the large switch boxes, each of which is logically composed of four of the smaller switches. In this case, the crossbar uses 64 switches versus 12 switch boxes, or 48 switches, in the Omega network. The crossbar, however, can support any combination of messages between processors, while the Omega network cannot.



a. Crossbar

b. Omega network



c. Omega network switch box

Implementing network topologies

This simple analysis of all the networks in this section ignores important practical considerations in the construction of a network. The distance of each link affects the cost of communicating at a

high clock rate—generally, the longer the distance, the more expensive it is to run at a high clock rate. Shorter distances also make it easier to assign more wires to the link, as the power to drive many wires is less if the wires are short. Shorter wires are also cheaper than longer wires. Another practical limitation is that the three-dimensional drawings must be mapped onto chips that are essentially two-dimensional media. The final concern is energy. Energy concerns may force multicore chips to rely on simple grid topologies, for example. The bottom line is that topologies that appear elegant when sketched on the blackboard may be impractical when constructed in silicon or in a datacenter.

Now that we understand the importance of clusters and have seen topologies that we can follow to connect them together, we next look at the hardware and software of the interface of the network to the processor.

PARTICIPATION ACTIVITY 6.9.1: Check yourself: multiprocessor network topologies.

1) For a ring with P nodes, the ratio of the total network bandwidth to the bisection bandwidth is P/2.
   - ○ True
   - ○ False

(*1) This section is in original form.

6.10 Communicating to the outside world: Cluster networking

(Original section[1])

This section describes the networking hardware and software used to connect the nodes of cluster together. As there are whole books and courses just on networking, this section only introduces the main terms and concepts. While our example is networking, the techniques we describe apply to storage controllers and other I/O devices as well.

Ethernet has dominated local area networks for decades, so it is not surprising that clusters primarily rely on Ethernet as the cluster interconnect. It became commercially popular at 10 Megabits per second link speed in the 1980s, but today 10 Gigabitsecond Ethernet is standard and 100 Gigabitsecond is being deployed in datacenters. The figure below shows a network interface card (NIC) for 10 Gigabit Ethernet.

Figure 6.10.1: The NetFPGA 10-Gigabit Ethernet card (COD Figure e6.9.1).

The NetFPGA 10-Gigabit Ethernet card (see http://netfpga.org/), which connects up to four 10-Gigabit/sec Ethernet links. It is an FPGA-based open platform for network research and classroom experimentation.

The DMA engine and the four "MAC chips" in the figure below are just portions of the Xilinx Virtex FPGA in the middle of the board. The four PHY chips in the figure below are the four black squares just to the right of the four white rectangles on the left edge of the board, which is where the Ethernet cables are plugged in.

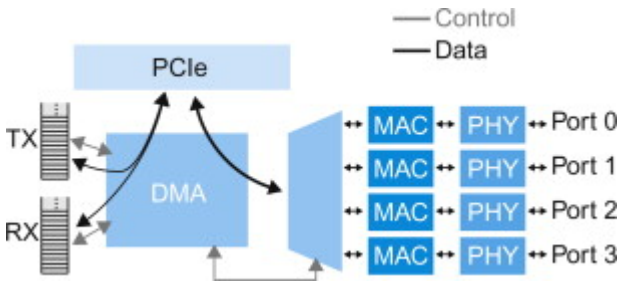
Computers offer high-speed links to plug in fast I/O devices like this NIC. While there used to be separate chips to connect the microprocessor to the memory and high-speed I/O devices, thanks to Moore's Law these functions have been absorbed into the main chip in recent offerings like Intel's Skylake. A popular high-speed link today is **PCIe**, which stands for **Peripheral Component Interconnect Express**. It is called a *link* in that the basic building block, called a *serial lane*, consists of just four wires: two for receiving data and two for transmitting data. This small number contrasts with an earlier version of PCI that consisted of 64 wires, which was called a *parallel bus*. PCIe allows anywhere from 1 to 32 lanes to be used to connect to I/O devices, depending on its needs. This NIC uses PCI 1.1, so each lane transfers at 2 Gigabits/second.

The NIC in the figure above connects to the host computer over an 8-lane PCIe link, which offers 16 Gigabits/second in both directions. To communicate, a NIC must both send or transmit messages and receive them, often abbreviated as TX and RX, respectively. For this NIC, each 10 G link uses separate transmit and receive queues, each of which can store two full-length Ethernet packets, used between the Ethernet links and the NIC. The figure below is a block diagram of the NIC showing the TX and RX queues. The NIC also has two 32-entry queues for transmitting and receiving between the host computer and the NIC.

Figure 6.10.2: Block diagram of the NetFPGA Ethernet card (COD Figure e6.9.2).

Block diagram of the NetFPGA Ethernet card in the figure above showing the control paths and the data path

The control path allows the DMA engine to read the status of the queues, such as empty vs. on-empty, and the content of the next available queue entry. The DMA engine also controls port multiplexing. The data path simply passes through the DMA block to the TX/RX queues or to main memory. The "MAC chips" are described below. The PHY chips, which refer to the physical layer, connect the "MAC chips" to physical networking medium, such as copper wire or optical fiber.



To give a command to the NIC, the processor must be able to address the device and to supply one or more command words. In *memory-mapped I/O*, portions of the address space are assigned to I/O devices. During initialization (at boot time), PCIe devices can request to be assigned an address region of a specified length. All subsequent processor reads and writes to that address region are forwarded over PCIe to that device. Reads and writes to those addresses are interpreted as commands to the I/O device.

Memory-mapped I/O: An I/O scheme in which portions of the address space are assigned to I/O devices, and reads and writes to those addresses are interpreted as commands to the I/O device.

For example, a write operation can be used to send data to the network interface where the data will be interpreted as a command. When the processor issues the address and data, the memory system ignores the operation because the address indicates a portion of the memory space used for I/O. The NIC, however, sees the operation and records the data. User programs are prevented from issuing I/O operations directly, because the OS does not provide access to the address space assigned to the I/O devices, and thus the addresses are protected by the address translation. Memory-mapped I/O can also be used to transmit data by writing or reading to select addresses. The device uses the address to determine the type of command, and the data may be provided by a write or obtained by a read. In any event, the address encodes both the device identity and the

type of transmission between processor and device.

While the processor could transfer the data from the user space into the I/O space by itself, the overhead for transferring data from or to a high-speed network could be intolerable, since it could consume a large fraction of the processor. Thus, computer designers long ago invented a mechanism for offloading the processor and having the device controller transfer data directly to or from the memory without involving the processor. This mechanism is called *direct memory access* (DMA).

Direct memory access (***DMA***): A mechanism that provides a device controller with the ability to transfer data directly to or from the memory without involving the processor.

DMA is implemented with a specialized controller that transfers data between the network interface and memory independent of the processor, and in this case the DMA engine is inside the NIC.

To notify the operating system (and eventually the application that will receive the packet) that a transfer is complete, the DMA sends an *I/O interrupt*.

Interrupt-driven I/O: An I/O scheme that employs interrupts to indicate to the processor that an I/O device needs attention.

An I/O interrupt is just like the exceptions we saw in COD Chapters 4 (The Processor) and 5 (Large and Fast: Exploiting Memory Hierarchy), with two important distinctions:

1. An I/O interrupt is asynchronous with respect to the instruction execution. That is, the interrupt is not associated with any instruction and does not prevent the instruction completion, so it is very different from either page fault exceptions or exceptions such as arithmetic overflow. Our control unit needs only check for a pending I/O interrupt at the time it starts a new instruction.
2. In addition to the fact that an I/O interrupt has occurred, we would like to convey further information, such as the identity of the device generating the interrupt. Furthermore, the interrupts represent devices that may have different priorities and whose interrupt requests have different urgencies associated with them.

To communicate information to the processor, such as the identity of the device raising the interrupt, a system can use either vectored interrupts or an exception identification register, called the Cause register in MIPS (see COD Section 4.9 (Exceptions)). When the processor recognizes the interrupt, the device can send either the vector address or a status field to place in the Cause register. As a result, when the OS gets control, it knows the identity of the device that caused the interrupt and can immediately interrogate the device. An interrupt mechanism eliminates the need for the processor to keep checking the device and instead allows the processor to focus on executing programs.

The role of the operating system in networking

The operating system acts as the interface between the hardware and the program that requests I/O. The network responsibilities of the operating system arise from three characteristics of networks:

1. Multiple programs using the processor share the network.
2. Networks often use interrupts to communicate information about the operations. Because interrupts cause a transfer to kernel or supervisor mode, they must be handled by the operating system (OS).
3. The low-level control of an network is complex, because it requires managing a set of concurrent events and because the requirements for correct device control are often very detailed.

Hardware/Software Interface

These three characteristics of networks specifically and I/O systems in general lead to several different functions the OS must provide:

- The OS guarantees that a user's program accesses only the portions of an I/O device to which the user has rights. For example, the OS must not allow a program to read or write a file on disk if the owner of the file has not granted access to this program. In a system with shared I/O devices, protection could not be provided if user programs could perform I/O directly.
- The OS provides abstractions for accessing devices by supplying routines that handle low-level device operations.
- The OS handles the interrupts generated by I/O devices, just as it handles the exceptions generated by a program.
- The OS tries to provide equitable access to the shared I/O resources, as well as schedule accesses to enhance system throughput.

The software inside the operating system that interfaces to a specific I/O device like this NIC is called a *device driver*. The driver for this NIC follows five steps when transmitting or receiving a message. The figure below shows the relationship of these steps as an Ethernet packet is sent from one node of the cluster and received by another node in the cluster.

Device driver: A program that controls an I/O device that is attached to the computer.

Figure 6
an Ethe
node (C

First, the transmit steps:

1. The driver first prepares a packet buffer in host memory. It copies a packet from the user address space into a buffer that it allocates in the operating system address space.
2. Next, it "talks" to the NIC. The driver writes an I/O descriptor to the appropriate NIC register that gives the address of the buffer and its length.
3. The DMA in the NIC next copies the outgoing Ethernet packet from the host buffer over PCIe.
4. When the transmission is complete, the DMA interrupts the processor to notify the processor that the packet has been successfully transmitted.
5. Finally, the driver de-allocates the transmit buffer.

Next, the receive steps:

1. First, the driver prepares a packet buffer in host memory, allocating a new buffer in which to place the received packet.
2. Next, it "talks" to the NIC. The driver writes an I/O descriptor to the appropriate NIC register that gives the address of the buffer and its length.
3. The DMA in the NIC next copies the incoming Ethernet packet over PCIe into the allocated host buffer.
4. When the transmission is complete, the DMA interrupts the processor to notify the host of the newly received packet and its size.

5. Finally, the driver copies the received packet into the user address space.

As you can see in the figure above, the first three steps are time critical when transmitting a packet (since the last two occur after the packet is sent), and the last three steps are time critical when receiving a packet (since the first two occur before a packet arrives). However, these non-critical steps must be completed before individual nodes run out of resources, such as memory space. Failure to do so negatively affects network performance.

Improving network performance

The importance of networking in clusters means it is certainly worthwhile to try to improve performance. We show both software and hardware techniques.

Starting with software optimizations, one performance target is reducing the number of times the packet is copied, which you may have noticed happening repeatedly in the five steps of the driver above. The *zero-copy* optimization allows the DMA engine to get the message directly from the user program data space during transmission and be placed where the user wants it when the message is received, rather than go through intermediary buffers in the operating system along the way.

A second software optimization is to cut out the operating system almost entirely by moving the communication into the user address space. By not invoking the operating system and not causing a context switch, we can reduce the software overhead considerably.

In this more radical scenario, a third step would be to drop interrupts. One reason is that modern processors normally go into lower power mode while waiting for an interrupt, and it takes time to come out of low power to service the interrupt as well for the disruption to the pipeline, which increases latency. The alternative to interrupts is for the processor to periodically check status bits to see if an I/O operation is complete, which is called *polling*. Hence, we can require the user program to poll the NIC continuously to see when the DMA unit has delivered a message, and as a side effect the processor does not go into low power mode.

Polling: The process of periodically checking the status of an I/O device to determine the need to service the device.

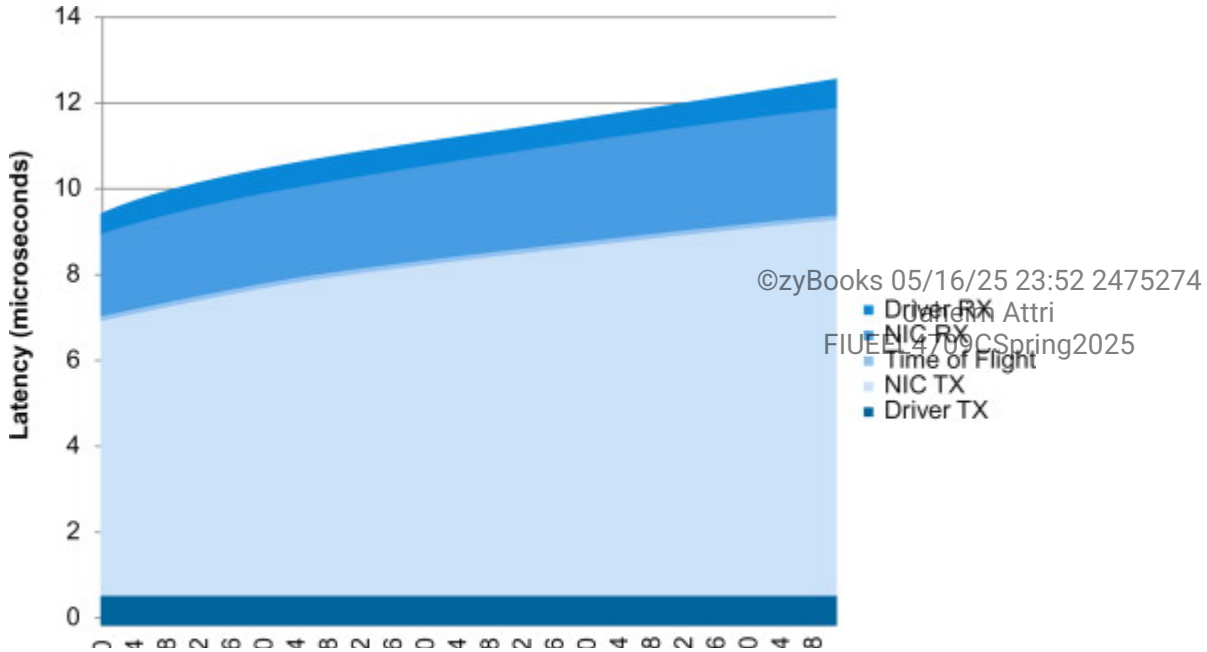
Looking at hardware optimizations, one potential target for improvement is in calculating the values of the fields of the Ethernet packet. The 48-bit Ethernet address, called the *Media Access Control address* or *MAC address*, is a unique number assigned to each Ethernet NIC. To improve performance, the "MAC chip"—actually just a portion of the FPGA on this NIC—calculates the value for the preamble fields and the CRC field (see COD Section 5.5 (Dependable memory hierarchy)). The driver is left with placing the MAC destination address, MAC source address, message type, the data payload, and padding if needed. (Ethernet requires that the minimum packet, including the header and CRC fields but not the preamble, be 64 bytes.) Note that even the least expensive Ethernet NICs do CRC calculation in hardware today.

A second hardware optimization, available on the most recent Intel processors such as Ivy Bridge, improves the performance of the NIC with respect to the memory hierarchy. *Direct Data IO (DDIO)* allowing up to 10% of the last level cache is used as a fast scratchpad for the DMA engine. Data is copied directly into the last level cache rather than to DRAM by the DMA, and only written to DRAM upon eviction from the cache. This optimization helps with latency, but also with bandwidth; some memory regions used for control might be written by the NIC repeatedly, and these writes no longer need to go to DRAM. Thus, DDIO offers benefits similar to those of a write back cache versus a write through cache (COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy)).

Let's look at an object store that follows a client-server architecture and uses most of the optimizations above: zero copy messaging, user space communication, polling instead of interrupts, and hardware calculation of preamble and CRC. The driver operates in user address space as a library that the application invokes. It grants this application exclusive and direct access to the NIC. All of the I/O register space on the NIC is mapped into the application, and all of the driver state is kept in the application. The OS kernel doesn't even see the NIC as such, which avoids the overheads of context switching, the standard kernel network software stack, and interrupts.

Figure 6.10.4: Time to send an object broken into transmit drive and NIC hardware time vs. receive driver and NIC hardware time (COD Figure e6.9.4).

NIC transmit time is much larger than the NIC receive time because transmit requires more PCIe round-trips. The NIC does PCIe reads to read the descriptor and data, but on receive the NIC does PCIe writes of data, length of data, and interrupt. PCIe reads incur a round trip latency because NIC waits for the reply, but PCIe writes require no response because PCIe is reliable, so PCIe writes can be sent back-to-back.



The figure above shows the time to send an object from one node to another. It varies from about 9.5 to 12.5 microseconds, depending on the size of the object. Here is the time for each step in microseconds:

- *0.7 — for the client "driver" (library) to make the request (Driver TX in the figure above).*
- *6.4 to 8.7 — for the NIC hardware to transmit the client's request over the PCIe bus to the Ethernet, depending on the size of the object (NIC TX).*
- *0.02 — to send object over the 10 G Ethernet (Time of Flight). The time of flight is limited by speed of light to 5 ns per meter. The three-meter cables used in this measurement mean the time of flight is 15 ns, which is too small to be clearly visible in the figure.*
- *1.8 to 2.5 — for the NIC hardware to receive the object, depending on its size (NIC RX).*
- *0.6 — for the server "driver" to transmit the message with the requested object to the app (Driver RX).*

Now that we have seen how to measure the performance of network at a low level of detail, let's raise the perspective to see how to benchmark multiprocessors of all kinds with much higher level programs.

Elaboration

There are many versions of PCIe. This NIC uses PCIe 1.1, which transfers at 2 gigabits per second per lane, so this NIC transfers at up to 16 gigabits per second in each direction. PCIe 2.0, which is found on most PC motherboards today, doubles the lane bandwidth to 4 gigabits per second. PCIe 3.0 doubles again to 8 gigabits per second, and it is starting to be found on some motherboards. We applaud the standard committee's logical rate of bandwidth improvement, which has been about $2^{\text{version number}}$ gigabits/second. The limitations of the Virtex 5 FPGA prevented the NIC from using faster versions of PCIe.

Elaboration

While Ethernet is the foundation of cluster communication, clusters commonly use higher-level protocols for reliable communication. Transmission Control Protocol and Internet Protocol (TCP/IP), although invented for planet-wide communication, is often

used inside a warehouse scale computer, due in part to its dependability. While IP makes no delivery guarantees in the protocol, TCP does. The sender keeps the packet sent until it gets the acknowledgment message back that it was received correctly from the receiver. The receiver knows that the message was not corrupted along the way, by double-checking the contents with the TCP CRC field. To ensure that IP delivers to the right destination, the IP header includes a checksum to make sure the destination number remains unchanged. The success of the Internet is due in large part to the elegance and popularity of TCP/IP, which allows independent local area networks to communicate dependably. Given its importance in the Internet and in clusters, many have accelerated TCP/IP, using techniques like those listed in this section [Regnier, 2004].

Elaboration

Adding DMA is another path to the memory system—one that does not go through the address translation mechanism or the cache hierarchy. This difference generates some problems both in virtual memory and in caches. These problems are usually solved with a combination of hardware techniques and software support. The difficulties in having DMA in a virtual memory system arise because pages have both a physical and a virtual address. DMA also creates problems for systems with caches, because there can be two copies of a data item: one in the cache and one in memory. Because the DMA issues memory requests directly to the memory rather than through the processor cache, the value of a memory location seen by the DMA unit and the processor may differ. Consider a read from a NIC that the DMA unit places directly into memory. If some of the locations into which the DMA writes are in the cache, the processor will receive the old value when it does a read. Similarly, if the cache is write-back, the DMA may read a value directly from memory when a newer value is in the cache, and the value has not been written back. This is called the stale data problem or coherence problem (see COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy)). Similar solutions for coherence are used with DMA.

Elaboration

Virtual Machine support clearly can negatively impact networking performance. As a result, microprocessor designers have been adding hardware to reduce the

performance overhead of virtual machines for networking in particular and I/O in general. Intel offers Virtualization Technology for Directed I/O (VT-d) to help virtualize I/O. It is an I/O memory management unit that enables guest virtual machines to directly use I/O devices, such as Ethernet. It supports DMA remapping, which allows the DMA to read or write the data directly in the I/O buffers of the guest virtual machine, rather than into the host I/O buffers and then copy them into the guest I/O buffers. It also supports interrupt remapping, which lets the virtual machine monitor route interrupt requests directly to the proper virtual machine.

PARTICIPATION ACTIVITY 6.10.1: Check yourself: Interrupts and polling.

Two options for networking are using interrupts or polling, and using DMA or using the processor via load and store instructions.

1) If we want the lowest latency for small packets, which combination is likely best?
   - ○ interrupts with DMA
   - ○ polling with load/store

2) If we want the lowest latency for large packets, which combination is likely best?
   - ○ interrupts with DMA
   - ○ polling with load/store

(*1) This section is in original form.

6.11 Multiprocessor benchmarks and performance models

As we saw in COD Chapter 1 (Computer Abstractions and Technology), benchmarking systems is always a sensitive topic, because it is a highly visible way to try to determine which system is better. The results affect not only the sales of commercial systems, but also the reputation of the designers of those systems. Hence, all participants want to win the competition, but they also want to be sure that if someone else wins, they deserve to win because they have a genuinely better

system. This desire leads to rules to ensure that the benchmark results are not simply engineering tricks for that benchmark, but are instead advances that improve performance of real applications.

To avoid possible tricks, a typical rule is that you can't change the benchmark. The source code and data sets are fixed, and there is a single proper answer. Any deviation from those rules makes the results invalid.

Many multiprocessor benchmarks follow these traditions. A common exception is to be able to increase the size of the problem so that you can run the benchmark on systems with a widely different number of processors. That is, many benchmarks allow weak scaling rather than require strong scaling, even though you must take care when comparing results for programs running different problem sizes.

The figure below gives a summary of several parallel benchmarks, also described below:

- *Linpack* is a collection of linear algebra routines, and the routines for performing Gaussian elimination constitute what is known as the Linpack benchmark. The DGEMM routine in the example in COD Section 3.5 (Floating Point) represents a small fraction of the source code of the Linpack benchmark, but it accounts for most of the execution time for the benchmark. It allows weak scaling, letting the user pick any size problem. Moreover, it allows the user to rewrite Linpack in almost any form and in any language, as long as it computes the proper result and performs the same number of floating point operations for a given problem size. Twice a year, the 500 computers with the fastest Linpack performance are published at www.top500.org. The first on this list is considered by the press to be the world's fastest computer. Given the importance of energy efficiency today, the same organization also publishes a Green500 list, where they report the Top500 list based on performance per Watt running Linpack to celebrate the most efficient supercomputer.
- *SPECrate* is a throughput metric based on the SPEC CPU benchmarks, such as SPEC CPU 2006 (see COD Chapter 1 (Computer Abstractions and Technology)). Rather than report performance of the individual programs, SPECrate runs many copies of the program simultaneously. Thus, it measures task-level parallelism, as there is no communication between the tasks. You can run as many copies of the programs as you want, so this is again a form of weak scaling.
- *SPLASH* and *SPLASH 2* (Stanford Parallel Applications for Shared Memory) were efforts by researchers at Stanford University in the 1990s to put together a parallel benchmark suite similar in goals to the SPEC CPU benchmark suite. It includes both kernels and applications, including many from the high-performance computing community. This benchmark requires strong scaling, although it comes with two data sets.
- The *NAS (NASA Advanced Supercomputing) parallel benchmarks* were another attempt from the 1990s to benchmark multiprocessors. Taken from computational fluid dynamics, they consist of five kernels. They allow weak scaling by defining a few data sets. Like Linpack, these benchmarks can be rewritten, but the rules require that the programming language can only be C or Fortran.
- The *PARSEC (Princeton Application Repository for Shared Memory Computers) benchmark*

suite consists of multithreaded programs that use *Pthreads* (POSIX threads) and OpenMP (Open MultiProcessing; see COD Section 6.5 (Multicore and other shared memory multiprocessors)). They focus on emerging computational domains and consist of nine applications and three kernels. Eight rely on data parallelism, three rely on pipelined parallelism, and one on unstructured parallelism.

- On the cloud front, the goal of the *Yahoo! Cloud Serving Benchmark* (YCSB) is to compare performance of cloud data services. It offers a framework that makes it easy for a client to benchmark new data services, using Cassandra and HBase as representative examples. [Cooper, 2010]

Pthreads: A UNIX API for creating and manipulating threads. It is structured as a library.

Figure 6.11.1: Examples of parallel benchmarks (COD Figure 6.16).

Benchmark	Scaling?	Reprogram?	Description
Linpack	Weak	Yes	Dense matrix linear algebra [Dongarra, 1979]
SPECrate	Weak	No	Independent job parallelism [Henning, 2007]
Stanford Parallel Applications for Shared Memory SPLASH 2 [Woo et al., 1995]	Strong (although offers two problem sizes)	No	Complex 1D FFT Blocked LU Decomposition Blocked Sparse Cholesky Factorization Integer Radix Sort Barnes-Hut Adaptive Fast Multipole Ocean Simulation Hierarchical Radiosity Ray Tracer Volume Renderer Water Simulation with Spatial Data Structure Water Simulation without Spatial Data Structure
NAS Parallel Benchmarks [Bailey et al., 1991]	Weak	Yes (C or Fortran only)	EP: embarrassingly parallel MG: simplified multigrid CG: unstructured grid for a conjugate gradient method FT: 3-D partial differential equation solution using FFTs IS: large integer sort
PARSEC Benchmark Suite [Bienia et al., 2008]	Weak	No	Blackscholes—Option pricing with Black-Scholes PDE Bodytrack—Body tracking of a person Canneal—Simulated cache-aware annealing to optimize routing Dedup—Next-generation compression with data deduplication Facesim—Simulates the motions of a human face Ferret—Content similarity search server Fluidanimate—Fluid dynamics for animation with SPH method Freqmine—Frequent itemset mining Streamcluster—Online clustering of an input stream Swaptions—Pricing of a portfolio of swaptions Vips—Image processing x264—H.264 video encoding
Berkeley Design	Strong or		Finite-State Machine Combinational Logic Graph Traversal Structured Grid Dense Matrix Sparse Matrix

Patterns [Asanovic et al., 2006]	Strong or Weak	Yes	Spectral Methods (FFT) Dynamic Programming N-Body MapReduce Backtrack/Branch and Bound Graphical Model Inference Unstructured Grid

The downside of such traditional restrictions to benchmarks is that innovation is chiefly limited to the architecture and compiler. Better data structures, algorithms, programming languages, and so on often cannot be used, since that would give a misleading result. The system could win because of, say, the algorithm, and not because of the hardware or the compiler.

While these guidelines are understandable when the foundations of computing are relatively stable—as they were in the 1990s and the first half of this decade—they are undesirable during a programming revolution. For this revolution to succeed, we need to encourage innovation at all levels.

Researchers at the University of California at Berkeley have advocated one approach. They identified 13 design patterns that they claim will be part of applications of the future. Frameworks or kernels implement these design patterns. Examples are sparse matrices, structured grids, finite-state machines, map reduce, and graph traversal. By keeping the definitions at a high level, they hope to encourage innovations at any level of the system. Thus, the system with the fastest sparse matrix solver is welcome to use any data structure, algorithm, and programming language, in addition to novel architectures and compilers.

While not primarily a parallel computing benchmark, MLPerf is a recent benchmark for ML that usually runs on parallel computers. It includes programs, data sets, and ground rules. New versions of the MLPerf benchmarks occur every 3 months to keep up with the rapid advances in ML. To normalize for different-sized computers, MLPerf includes the power to run the benchmarks. A novel benchmarking feature is to offer both closed and open divisions of the benchmarks. The closed division has tightly controlled rules for submission to try to ensure fair comparisons between systems. The open division encourages innovation, including better data structures, algorithms, programming systems, and so on. Open division submissions just need to perform the same task using the same data sets. We use MLPerf to evaluate DSAs in the next section.

Performance models

A topic related to benchmarks is performance models. As we have seen with the increasing architectural diversity in this chapter—multithreading, SIMD, GPUs, TPUs—it would be especially helpful if we had a simple model that offered insights into the performance of different architectures. It need not be perfect, just insightful.

The 3Cs for cache performance from COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy) is an example performance model. It is not a perfect performance model, since it ignores potentially important factors like block size, block allocation policy, and block replacement

policy. Moreover, it has quirks. For example, a miss can be ascribed due to capacity in one design and to a conflict miss in another cache of the same size. Yet 3Cs model has been popular for 30 years, because it offers insight into the behavior of programs, helping both architects and programmers improve their creations based on insights from that model.

To find such a model for parallel computers, let's start with small kernels, like those from the 13 Berkeley design patterns in the figure above. While there are versions with different data types for these kernels, floating point is popular in several implementations. Hence, peak floating-point performance is a limit on the speed of such kernels on a given computer. For multicore chips, peak floating-point performance is the collective peak performance of all the cores on the chip. If there were multiple microprocessors in the system, you would multiply the peak per chip by the total number of chips.

The demands on the memory system can be estimated by dividing this peak floating-point performance by the average number of floating-point operations per byte accessed:

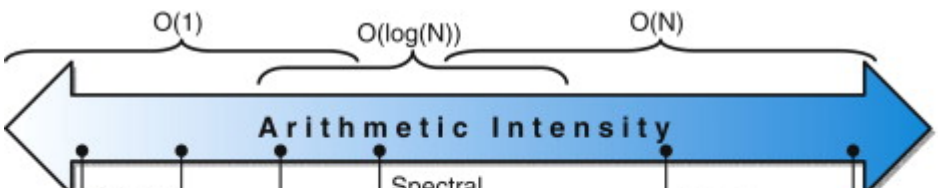
$$\frac{\textbf{Floating-Point Operations/Sec}}{\textbf{Floating-Point Operations/Byte}} = \textbf{Bytes/Sec}$$

The ratio of floating-point operations per byte of memory accessed is called the *arithmetic intensity*. 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. The figure below shows the arithmetic intensity of several of the Berkeley design patterns from the figure above.

Arithmetic intensity: The ratio of floating-point operations in a program to the number of data bytes accessed by a program from main memory.

Figure 6.11.2: Arithmetic intensity, specified as the number of float-point operations to run the program divided by the number of bytes accessed in main memory [Williams, Waterman, and Patterson 2009] (COD Figure 6.17).

Some kernels have an arithmetic intensity that scales with problem size, such as Dense Matrix, but there are many kernels with arithmetic intensities independent of problem size. For kernels in this former case, weak scaling can lead to different results since it puts much less demand on the memory system.



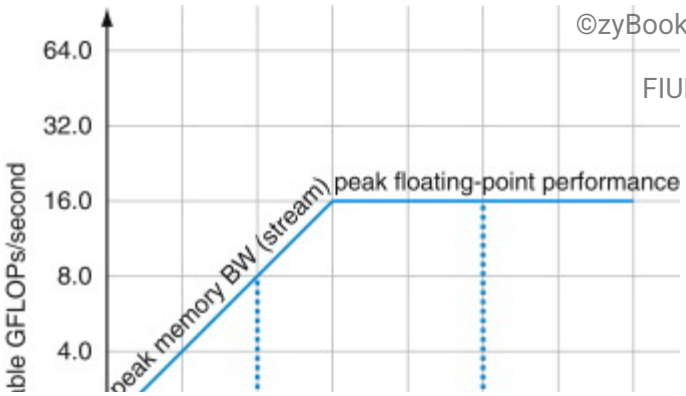
The roofline model

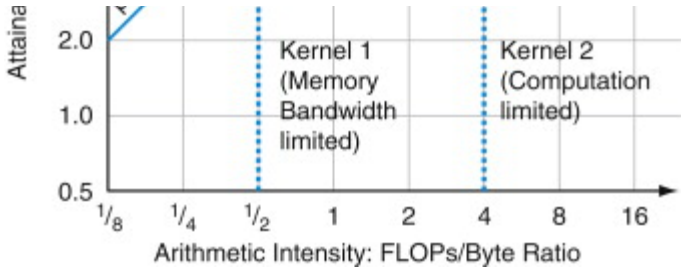
This simple model ties floating-point performance, arithmetic intensity, and memory performance together in a two-dimensional graph [Williams, Waterman, and Patterson 2009]. Peak floating-point performance can be found using the hardware specifications mentioned above. The working sets of the kernels we consider here do not fit in on-chip caches, so peak memory performance may be defined by the memory system behind the caches. One way to find the peak memory performance is the Stream benchmark. (See the *Elaboration* in COD Section 5.2 (Memory technologies)).

The figure below shows the model, which is done once for a computer, not for each kernel. The vertical Y-axis is achievable floating-point performance from 0.5 to 64.0 GFLOPs/second. The horizontal X-axis is arithmetic intensity, varying from 1/8 FLOPs/DRAM byte accessed to 16 FLOPs/DRAM byte accessed. Note that the graph is a log-log scale.

Figure 6.11.3: Roofline Model [Williams, Waterman, and Patterson 2009] (COD Figure 6.18).

This example has a peak floating-point performance of 16 GFLOPS/sec and a peak memory bandwidth of 16 GB/sec from the Stream benchmark. (Since Stream is actually four measurements, this line is the average of the four.) The dotted vertical line in color on the left represents Kernel 1, which has an arithmetic intensity of 0.5 FLOPs/ byte. It is limited by memory bandwidth to no more than 8 GFLOPS/sec on this Opteron X2. The dotted vertical line to the right represents Kernel 2, which has an arithmetic intensity of 4 FLOPs/byte. It is limited only computationally to 16 GFLOPS/s. (This data is based on the AMD Opteron X2 (Revision F) using dual cores running at 2 GHz in a dual socket system.)





For a given kernel, we can find a point on the X-axis based on its arithmetic intensity. If we draw 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, since that is a hardware limit.

How could we plot the peak memory performance, which is measured in bytes/second? Since the X-axis is FLOPs/byte and the Y-axis FLOPs/second, bytes/second is just a diagonal line at a 45-degree angle in this figure. Hence, 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 the line in the graph in the figure above:

$$\textbf{Attainable GFLOPs/sec} = \textbf{Min (Peak Memory BW} \times \textbf{Arithmetic Intensity, Peak F}$$

The horizontal and diagonal lines give this simple model its name and indicate its value. The "roofline" sets an upper bound on performance of a kernel depending on its arithmetic intensity. Given a roofline of a computer, you can apply it repeatedly, since it doesn't vary by kernel.

If we think of arithmetic intensity as a pole that hits the roof, either it hits the slanted part of the roof, which means performance is ultimately limited by memory bandwidth, or it hits the flat part of the roof, which means performance is computationally limited. In the figure above, kernel 1 is an example of the former, and kernel 2 is an example of the latter.

Note that the "ridge point", where the diagonal and horizontal roofs meet, offers an interesting insight into the 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.

Comparing two generations of opterons

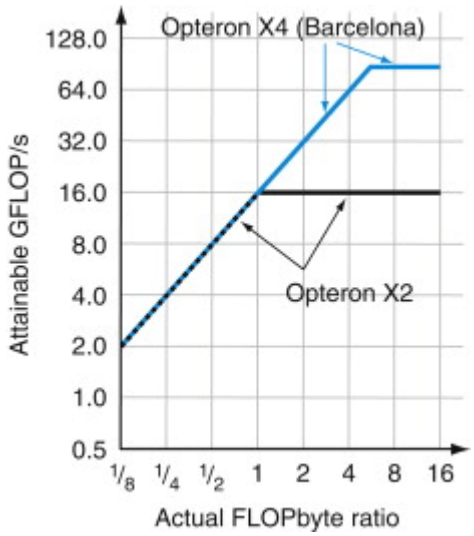
The AMD Opteron X4 (Barcelona) with four cores is the successor to the Opteron X2 with two cores. To simplify board design, they use the same socket. Hence, they have the same DRAM channels and thus the same peak memory bandwidth. In addition to doubling the number of cores, the Opteron X4 also has twice the peak floating-point performance per core: Opteron X4 cores can issue two floating-point SSE2 instructions per clock cycle, while Opteron X2 cores issue at most one. As the two systems we're comparing have similar clock rates—2.2 GHz for Opteron X2 versus 2.3 GHz for Opteron X4—the Opteron X4 has about four times the peak floating-point performance

of the Opteron X2 with the same DRAM bandwidth. The Opteron X4 also has a 2MiB L3 cache, which is not found in the Opteron X2.

In the figure below the roofline models for both systems are compared. As we would expect, the ridge point moves to the right, from 1 in the Opteron X2 to 5 in the Opteron X4. Hence, to see a performance gain in the next generation, kernels need an arithmetic intensity higher than 1, or their working sets must fit in the caches of the Opteron X4.

Figure 6.11.4: Rooflines models of two generations of Opterons (COD Figure 6.19).

The Opteron X2 roofline, which is the same as in the figure above, is in black, and the Opteron X4 roofline is in color. The bigger ridge point of Opteron X4 means that kernels that were computationally bound on the Opteron X2 could be memory-performance bound on the Opteron X4.



The roofline model gives an upper bound to performance. Suppose your program is far below that bound. What optimizations should you perform, and in what order?

To reduce computational bottlenecks, the following two optimizations can help almost any kernel:

1. *Floating-point operation mix.* Peak floating-point performance for a computer typically requires an equal number of nearly simultaneous additions and multiplications. That balance is necessary either because the computer supports a fused multiply-add instruction (see the Elaboration in COD Section 3.5 (Floating Point)) or because the floating-point unit has an equal number of floating-point adders and floating-point multipliers. The best performance also requires that a significant fraction of the instruction mix is floating- point operations and not integer instructions.

2. *Improve* ***instruction-level parallelism*** *and apply SIMD.* For modern architectures, the highest performance comes when fetching, executing, and committing three to four instructions per clock cycle (see COD Section 4.10 (Parallelism via instructions)). The goal for this step is to improve the code from the compiler to increase ILP. One way is by unrolling loops, as we saw in COD Section 4.12 (Going faster: Instruction-level parallelism and matrix multiply). For the x86 architectures, a single AVX instruction can operate on four double precision operands, so they should be used whenever possible (see COD Sections 3.7 (Real stuff: Streaming SIMD extensions and advanced vector extensions in x86) and 3.8 (Going faster: Subword parallelism and matrix multiply)).

To reduce memory bottlenecks, the following two optimizations can help:

1. *Software prefetching.* Usually the highest performance requires keeping many memory operations in flight, which is easier to do by performing **predicting** accesses via software prefetch instructions rather than waiting until the data is required by the computation.
2. *Memory affinity.* Microprocessors today include a memory controller on the same chip with the microprocessor, which improves performance of the **memory hierarchy**. If the system has multiple chips, this means that some addresses go to the DRAM that is local to one chip, and the rest require accesses over the chip interconnect to access the DRAM that is local to another chip. This split results in non-uniform memory accesses, which we described in COD Section 6.5 (Multicore and other shared memory multiprocessors). Accessing memory through another chip lowers performance. This second optimization tries to allocate data and the threads tasked to operate on that data to the same memory-processor pair, so that the processors rarely have to access the memory of the other chips.

The roofline model can help decide which of these two optimizations to perform and the order in which to perform them. We can think of each of these optimizations as a "ceiling" below the appropriate roofline, meaning that you cannot break through a ceiling without performing the associated optimization.

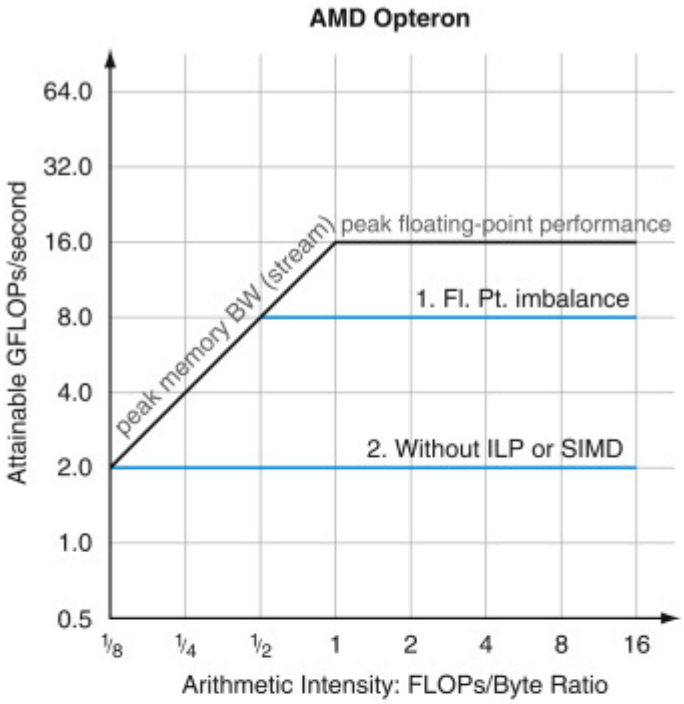
The computational roofline can be found from the manuals, and the memory roofline can be found from running the Stream benchmark. The computational ceilings, such as floating-point balance, can also come from the manuals for that computer. A memory ceiling, such as memory affinity, requires running experiments on each computer to determine the gap between them. The good news is that this process only need be done once per computer, for once someone characterizes a computer's ceilings, everyone can use the results to prioritize their optimizations for that computer.

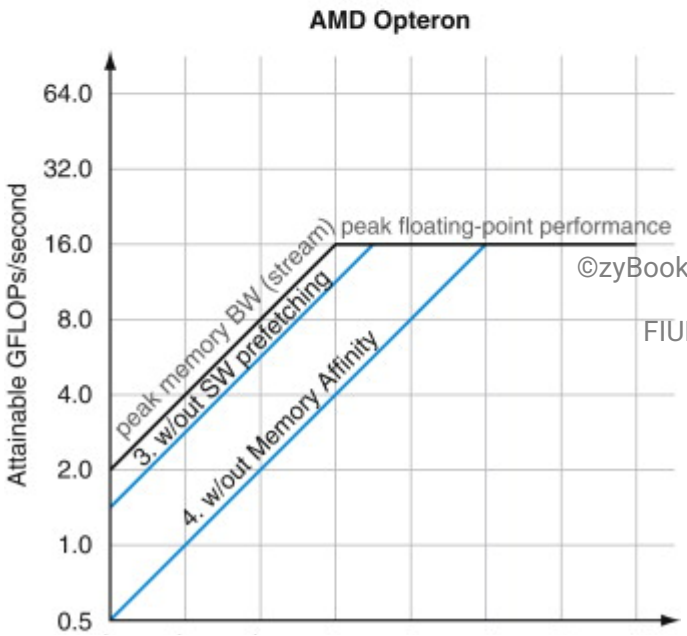
The figure below adds ceilings to the roofline model in COD Figure 6.18 (Roofline model …), showing the computational ceilings in the top graph and the memory bandwidth ceilings on the bottom graph. Although the higher ceilings are not labeled with both optimizations, they are implied

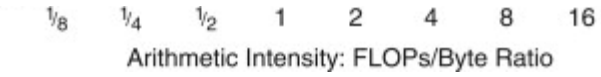
in this figure; to break through the highest ceiling, you need to have already broken through all the ones below.

Figure 6.11.5: Roofline model with ceilings (COD Figure 6.20).

The top graph shows the computational "ceilings" of 8 GFLOPs/sec if the floating-point operation mix is imbalanced and 2 GFLOPs/sec if the optimizations to increase ILP and SIMD are also missing. The bottom graph shows the memory bandwidth ceilings of 11 GB/ sec without software prefetching and 4.8 GB/sec if memory affinity optimizations are also missing.





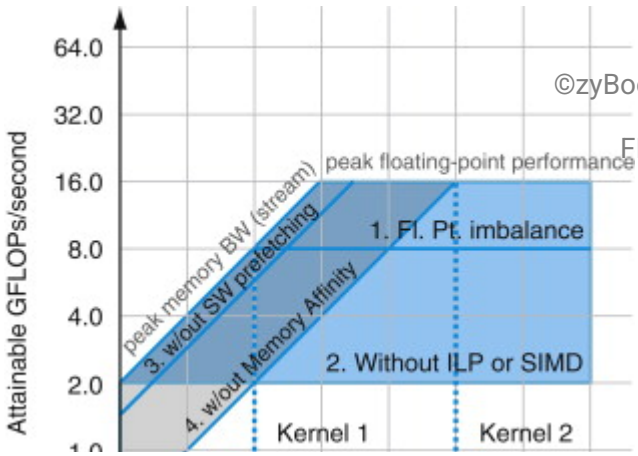


The width of the gap between the ceiling and the next higher limit is the reward for trying that optimization. Thus, the figure above suggests that optimization 2, which improves ILP, has a large benefit for improving computation on that computer, and optimization 4, which improves memory affinity, has a large benefit for improving memory bandwidth on that computer.

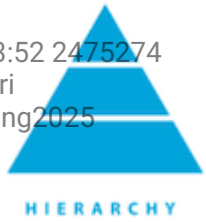
The figure below combines the ceilings of the figure above into a single graph. The arithmetic intensity of a kernel determines the optimization region, which in turn suggests which optimizations to try. Note that the computational optimizations and the memory bandwidth optimizations overlap for much of the arithmetic intensity. Three regions are shaded differently in the figure below to indicate the different optimization strategies. For example, Kernel 2 falls in the blue trapezoid on the right, which suggests working only on the computational optimizations. Kernel 1 falls in the blue-gray parallelogram in the middle, which suggests trying both types of optimizations. Moreover, it suggests starting with optimizations 2 and 4. Note that the Kernel 1 vertical lines fall below the floating-point imbalance optimization, so optimization 1 may be unnecessary. If a kernel fell in the gray triangle on the lower left, it would suggest trying just memory optimizations.

Figure 6.11.6: Roofline model with ceiling, overlapping areas shaded, and the two kernels from COD Figure 6.18 (Roofline Model …) (COD Figure 6.21).

Kernels whose arithmetic intensity land in the blue trapezoid on the right should focus on computation optimizations, and kernels whose arithmetic intensity land in the gray triangle in the lower left should focus on memory bandwidth optimizations. Those that land in the blue-gray parallelogram in the middle need to worry about both. As Kernel 1 falls in the parallelogram in the middle, try optimizing ILP and SIMD, memory affinity, and software prefetching. Kernel 2 falls in the trapezoid on the right, so try optimizing ILP and SIMD and the balance of floating-point operations.



Thus far, we have been assuming that the arithmetic intensity is fixed, but that is not really the case. First, there are kernels where the arithmetic intensity increases with problem size, such as for Dense Matrix and N-body problems (see COD Figure 6.17 (Arithmetic intensity, specified as the number of float-point operations …)). Indeed, this can be a reason that programmers have more success with weak scaling than with strong scaling. Second, the effectiveness of the **memory hierarchy** affects the number of accesses that go to memory, so optimizations that improve cache performance also improve arithmetic intensity. One example is improving temporal locality by unrolling loops and then grouping together statements with similar addresses. Many computers have special cache instructions that allocate data in a cache but do not first fill the data from memory at that address, since it will soon be over-written. Both these optimizations reduce memory traffic, thereby moving the arithmetic intensity pole to the right by a factor of, say, 1.5. This shift right could put the kernel in a different optimization region.



While the examples above show how to help programmers improve performance, architects can also use the model to decide where they should optimize hardware to improve performance of the kernels that they think will be important.

The next section includes the roofline model to compare the performance difference between a DSA and a GPU and to see whether these differences reflect performance of real programs.

Elaboration

The ceilings are ordered so that lower ceilings are easier to optimize. Clearly, a programmer can optimize in any order, but following this sequence reduces the chances of wasting effort on an optimization that has no benefit due to other constraints. Like the 3Cs model, as long as the roofline model delivers on insights, a model can have assumptions that may prove optimistic. For example, roofline assumes the load is balanced between all processors.

Elaboration

An alternative to the Stream benchmark is to use the raw DRAM bandwidth as the

roofline. While the raw bandwidth definitely is a hard upper bound, actual memory performance is often so far from that boundary that it's not that useful. That is, no program can go close to that bound. The downside to using Stream is that very careful programming may exceed the Stream results, so the memory roofline may not be as hard a limit as the computational roofline. We stick with Stream because few programmers will be able to deliver more memory bandwidth than Stream discovers.

Elaboration

Although the roofline model shown is for multicore processors, it clearly would work for a uniprocessor as well.

PARTICIPATION ACTIVITY 6.11.1: Check yourself: Benchmarks for parallel computers.

1) The main drawback with conventional approaches to benchmarks for parallel computers is that the rules that ensure fairness also slow software innovation.
   - ○ True
   - ○ False

6.12 Real stuff: TPUv3 Volta

Deep neural networks DNNs, introduced in COD Section 6.7, have two phases: *training*, which constructs accurate models, and *inference*, which serves those models. Training can take days or weeks to compute, while inference often runs in milliseconds. TPUv1 was designed for inference. This section explores how Google built a production DSA for the much harder training problem. It is based on the paper, "A Domain-Specific Supercomputer for Training Deep Neural Networks," *Communications of the ACM*, 2020 by N. P. Jouppi, D. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. A. Patterson.

DNN Training versus Inference

Let us quickly review of DNNs. Training starts with a huge training data set of known-correct (`input, result`) pairs. Pairs might be an image and what it depicts. It also starts with a neural network *model*, which transforms the input into the result through an intensive calculation of *weights*; the weights are random initially. Models are typically defined as a graph of layers, where a layer contains a linear algebra part (often a matrix multiplication or convolution using the weights) followed by a nonlinear *activation function* (often a scalar function, applied element wise, we call the results *activations*). Training "learns" weights that raise the likelihood of correctly mapping from input to result.

How do we get from random initial weights to trained weights? Current best practices use variants of stochastic gradient descent (*SGD*). SGD consists of much iteration of three steps: forward propagation, back-propagation, and weight update.

1. *Forward propagation* takes a randomly chosen training example, applies its inputs to the model, and runs the calculation through the layers to produce a result (which, with the random initial weights, is garbage the first time). Forward propagation is functionally similar to DNN inference, and if we were building an inference accelerator, we could stop there. For training, this is less than a third of the story. SGD next measures the difference or error between the model's result and the known good result from the training set using a *loss function*.
2. Then *back-propagation* runs the model in reverse, layer-by-layer, to produce a set of error/loss values for each layer's output. These losses measure the deviation from the desired output.
3. Last, *weight update* combines the input of each layer with the loss value to calculate a set of deltas—changes to weights—which, when added to the weights, would have resulted in nearly zero loss. Updates can have small magnitude.

Each SGD step makes a tiny adjustment to the weights that improve the model with respect to a single (`input, result`) pair. SGD gradually transforms the random initial weights into a trained model, sometimes capable of superhuman accuracy that results in articles in newspapers.

DSA Supercomputer Network

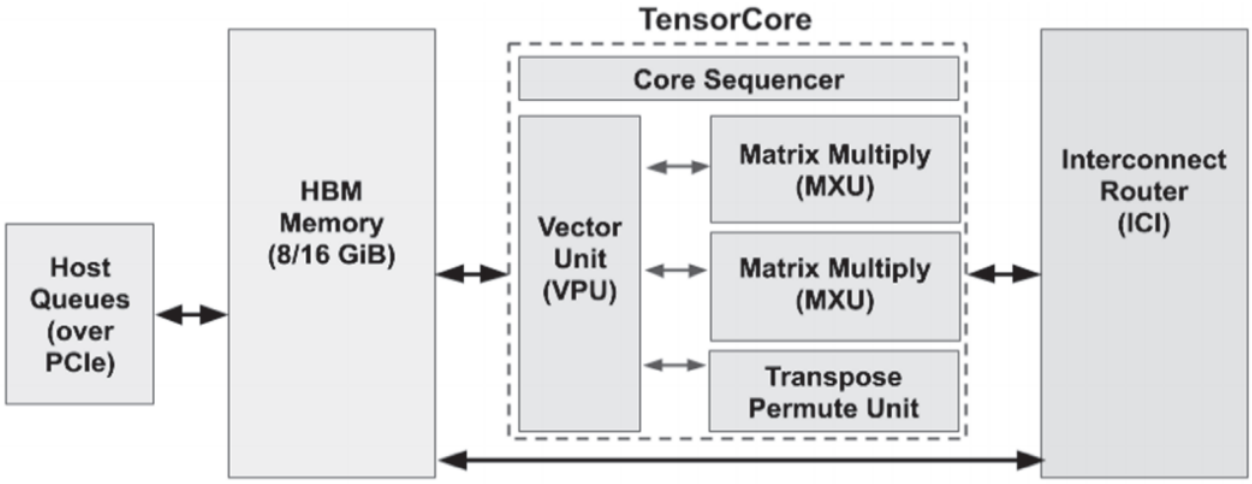
The DNN training computation appetite is essentially unlimited, thus Google chose to build a DSA supercomputer instead of clustering CPU hosts with DSA chips as was done for TPUv1. The first reason is that training time is huge. One TPUv3 chip would take *months* to train a single Google production application, so a typical application might want to use hundreds of chips. Second, DNN wisdom is that bigger data sets plus bigger machines lead to bigger breakthroughs.

The critical architectural feature of a modern supercomputer is how its chips communicate: what is the speed of a link; what is the interconnect topology; does it have centralized versus distributed switches; and so on. This choice is much easier for a DSA supercomputer, as the communication

patterns are limited and known. For training, most traffic is an all-reduce over weight updates from all nodes of the machine. It turns out that all reduce can be mapped efficiently onto a 2D torus topology. An on-chip switch routes messages. To enable a 2D torus, the TPUv3 chip has four custom *Inter-Core Interconnect* (ICI) links, each running at 656 Gbits/s each direction. ICI enables direct connections between chips to form a supercomputer using only a small fraction of each chip.

The TPUv3 supercomputer uses a 32 × 32 2D torus (1024 chips), which is 64 links × 656 Gbits/s = 42.3 Terabits/s of bisection bandwidth. As a comparison, a separate Infi niband switch (used in CPU clusters) that connected 64 hosts (each with 16 DSA chips) has 64 ports using "only" 100 Gbit/s links and a bisection bandwidth of at most 6.4 Terabits/s. The TPUv3 supercomputer provides 6.6× the bisection bandwidth over conventional cluster switches while skipping the cost of the Infi niband network cards, Infi niband switch, and the communication delays of going through the CPU hosts of clusters.

Figure 6.12.1: Block diagram of a TPUv3 TensorCore showing the six major blocks of a TensorCore (COD Figure 6.23).



DSA Supercomputer Node

The node of the TPUv3 supercomputer followed the main ideas of TPUv1: a large two-dimensional matrix multiply unit (MXU) plus large, software-controlled on-chip memories instead of caches. Unlike TPUv1, TPUv3 uses two cores per chip. Global wires on a chip do not scale with shrinking feature size, so their relative delay increases. Given that training can use many processors, two smaller TensorCores per chip prevented the excessive latencies of a single large full-chip core. Google stopped at two because it is easier to efficiently generate programs for two "brawny" cores per chip than numerous "wimpy" cores.

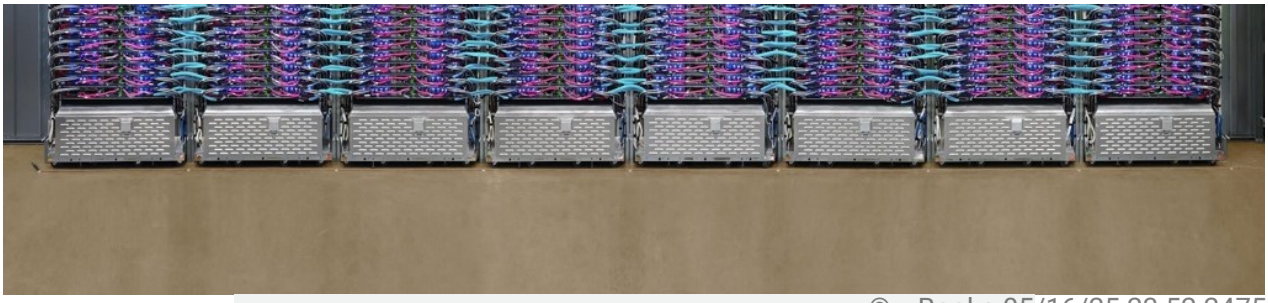
1. *Inter-Core Interconnect (ICI)*, which is explained previously.

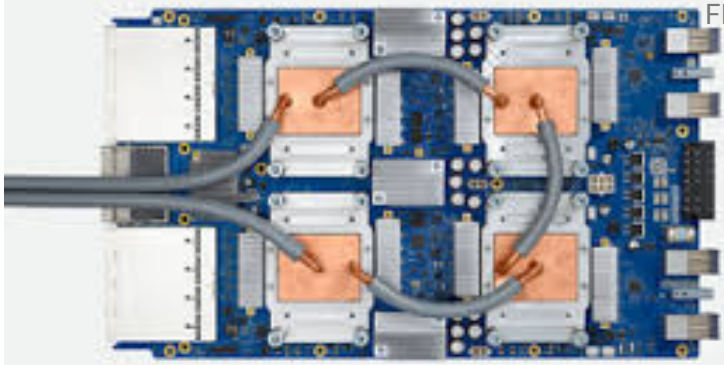
2. *High Bandwidth Memory (HBM)*. TPUv1 was memory bound for most of its applications [Jouppi, 2018]. Google solved the memory bottleneck of TPUv1 by using High Bandwidth Memory (HBM) DRAM. It offers 25 times the bandwidth of TPUv1 DRAMs by using an interposer substrate that connects the TPUv3 chip via 64 64-bit buses to four short stacks of DRAM chips. Conventional CPU servers support many more DRAM chips, but at much lower bandwidth of at most eight 64-bit busses.
3. The *Core Sequencer* executes *VLIW* instructions from the core's on-chip, software-managed Instruction Memory (*Imem*), executes scalar operations using a 4K 32-bit scalar data memory (*Smem*) and 32 32-bit scalar registers (*Sregs*), and forwards vector instructions to the VPU. The 322-bit VLIW instruction can launch eight operations: two scalar ALU, two vector ALU, vector load and store, and a pair of slots that queue data to and from the matrix multiply and transpose units.
4. The *Vector Processing Unit* (VPU) performs vector operations using a large on-chip vector memory (Vmem) with 32K 128 × 32-bit elements (16 MiB) and 32 2D vector registers (Vregs) that each contain 128 × 8 32-bit elements (4 KiB). The VPU collects and distributes data to Vmem via data-level parallelism (2D matrix and vector functional units) and instruction-level parallelism (eight operations per instruction).
5. The MXU produces 32-bit FP products from 16-bit FP inputs that accumulate in 32 bits. All other computations are in 32-bit FP except for results going directly to an MXU input, which are converted to 16-bit FP. TPUv3 has two MXUs per TensorCore.
6. The Transpose Reduction Permute Unit does 128 × 128 matrix transposes, reductions, and permutations of the VPU lanes.

The next figure shows the TPUv3 supercomputer and the TPUv3 node board, and the figure after that lists the specification of TPUv1, TPUv3, and NVIDIA Volta GPU that we will use for comparisons. The figure after both of those shows the rooflines, which are quite similar. The memory bandwidths are the same (900 Gbytes/second), the 16-bit floating point rooflines are nearly indistinguishable for TPUv3 and Volta (123 versus 125 TeraFLOPS/second), and there is a small difference in 32-bit floating point (14 versus 16 TeraFLOPS/second). Note the large performance difference between 16- and 32-bit floating point arithmetic for both chips.

Figure 6.12.2: A TPUv3 supercomputer consisting of up to 1024 chips (left) (COD Figure 6.24).

It is about 6 ft tall and 40 ft long. A TPUv3 board (right) has four chips and uses liquid cooling.





DSA Arithmetic

Peak performance is 8× higher when using 16-bit floating point instead of 32-bit floating point for matrix multiply, so it is vital to use 16-bit to get the highest performance. While Google could have built an MXU using standard IEEE half (fp16) and single (fp32) floating-point formats, they first checked the accuracy of 16-bit floating-point operations for DNNs. They found that:

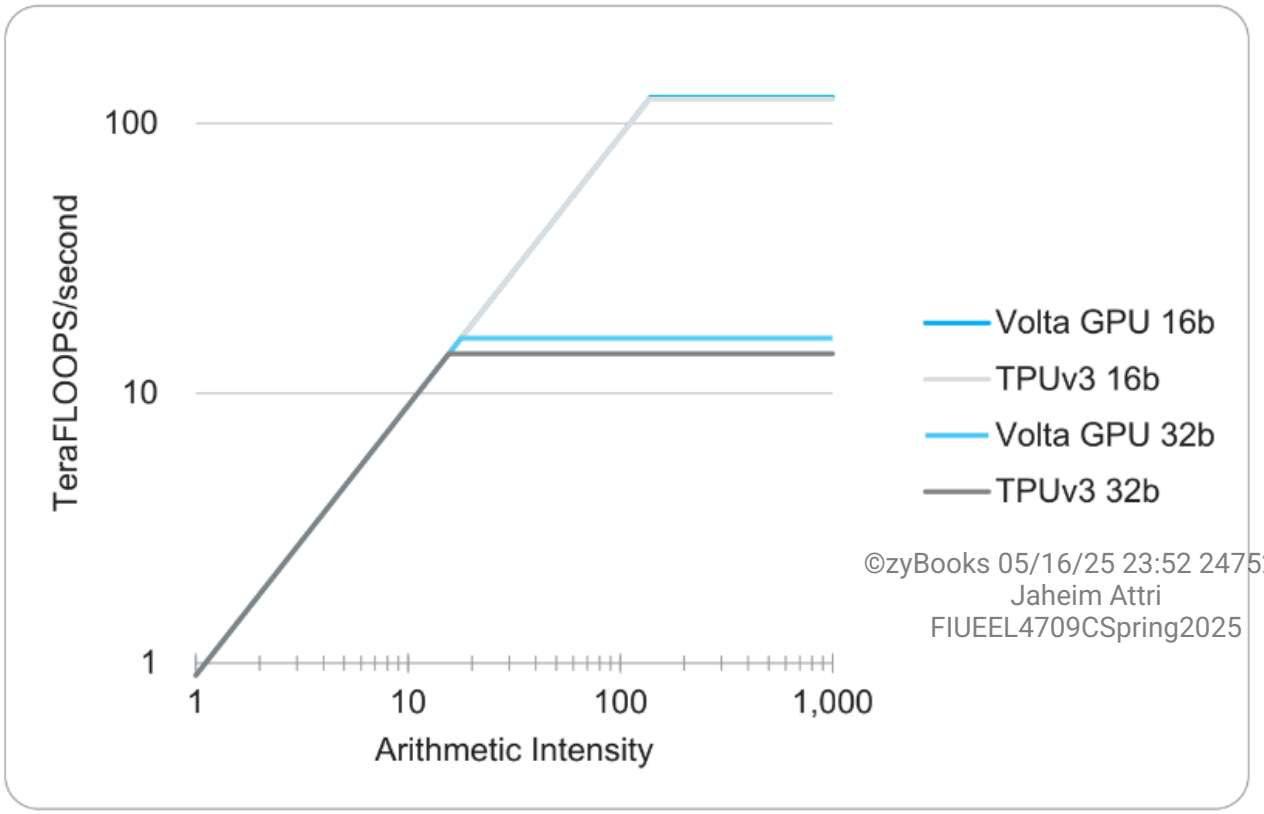
Matrix multiplication outputs and internal sums must remain in fp32.
The 5-bit exponent of fp16 matrix multiplication inputs leads to failure of computations that go outside its narrow range, which the 8-bit exponent of fp32 avoids.
Reducing the matrix multiplication input mantissa size from fp32's 23 bits to 7 bits did not hurt accuracy.
The resulting *Brain floating format* (*bf16*) keeps the same 8-bit exponent as fp32 but chops the mantissa to 7 bits. Given the same exponent size, there is no danger in losing the small update values due to floating point underflow of a smaller exponent, so all programs in this section used bf16 on TPUv3 without much difficulty. However, fp16 requires adjustments to training software to deliver convergence and efficiency. Micikevicius *et al*. used *loss scaling* on GPUs, which preserves the effect from small gradients by scaling losses to fit the smaller exponents of fp16 [Micikevicius *et al*. , 2017; Kalamkar *et al*. , 2019].

As the size of an FP multiplier scales with the square of the *mantissa* width, the bf16 multiplier is half the size and energy of an fp16 multiplier. Bf16 delivers a rare combination: reducing hardware and energy while simplifying software by making loss scaling unnecessary.

Figure 6.12.3: Key processor features of TPUv1, TPUv3, and NVIDIA Volta GPU (COD Figure 6.25).

Feature	TPUv1	TPUv3	Volta
Peak TeraFLOPS / Chip	92 (8b int)	123 (16b), 14 (32b)	125 (16b),16 (32b)
Network links x Gbits/s / Chip	--	4 x 656	6 x 200
Max chips / supercomputer	--	1024	Varies
Clock Rate (MHz)	700	940	1530
TDP (Watts) / Chip	75	450	450
Die Size (mm^2)	<310	<685	815
Chip Technology	28 nm	>12 nm	12 nm
Memory size (on-/off-chip)	28 MiB / 8 GiB	32 MiB /32 GiB	36 MiB / 32 GiB
Memory GB/s/Chip	34	900	900
MXUs / Core, MXU Size	1 256x256	2 128x128	8 4x4
Cores / Chip	1	2	80
Chips / CPU Host	4	8	8 or 16

Figure 6.12.4: Rooflines of TPUv3 and Volta (COD Figure 6.26).



TPUv3 DSA versus Volta GPU

Let us compare TPUv3 and Volta GPU architectures before we compare performance.

Multichip parallelization is built into TPUv3 through ICI and supported through all-reduce operations supported by the TPUv3 compiler. Similar-sized multichip GPU systems use a tiered networking approach, with NVIDIA's NVLink inside a chassis and host-controlled InfiniBand networks and switches to tie multiple chassis together.

TPUv3 offers 16-bit brain floating point arithmetic designed for DNNs inside 128 × 128 arrays that halves the hardware and energy versus IEEE fp16 multipliers. Volta GPUs have also embraced arrays, with a finer granularity—4 × 4 or 16 × 16 depending on hardware or software descriptions—while using fp16 rather than bf16, so they may require software to perform loss scaling plus extra die area and energy.

TPUv3 is a dual-core, in-order machine, where the compiler overlaps computation, memory, and network activities. Volta GPUs are latency-tolerant 80-core machines, where each core has many threads and thus very large (20 MiB) register files. The reading hardware plus CUDA coding conventions support overlapped operations.

TPUv3 use a software-controlled 32 MiB scratchpad memory that the compiler schedules, while Volta hardware manages a 6 MiB cache and software manages a 7.5 MiB scratchpad memory. The TPUv3 compiler directs sequential DRAM accesses typical of DNNs via direct memory access (DMA) controllers on TPUv3s while GPUs use multithreading plus coalescing hardware for them.

In addition to the contrasting architectural choices, TPU and GPU chips use different technologies, die areas, clock rates, and power. Figure 6.27 gives three related cost measures of these systems: approximate die size adjusted for technology, power for a 16-chip system, and cloud price per chip. The GPU adjusted die size is almost twice that of the TPUs, which suggests the capital costs of the chips is double, because there would be twice as many TPU dies per wafer. GPU power is 1.3× higher, which suggests higher operating expenses, as the TCO is correlated with power. Finally, the hourly rental prices on Google Cloud Engine are 1.6x higher for the GPU. These three different measures consistently suggest TPUv3 is roughly half to three-fourths as expensive as the Volta GPU.

Figure 6.12.5: Adjusted comparison of GPU and TPUv3 (COD Figure 6.27).

Die sizes are adjusted by the square of the technology, as the semiconductor technology for TPUs is similar but larger and older than that of the GPU. Google picked 15 nm for TPUs based on the information in Figure 6.25. Thermal Design Power (TDP) isfor 16-chip systems.

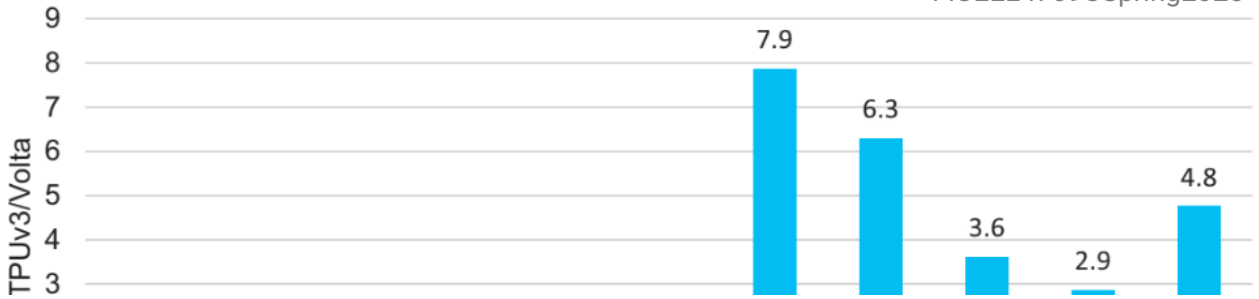
	Die size	Adjusted die size	(kw)	Cloud price
Volta	815	815	12.0	$3.24
TPUv3	<685	<438	9.3	$2.00

Performance

Before showing performance of TPUv3 supercomputers, let us establish the virtues of a single chip, for a 1024× speedup from 1024 wimpy chips is uninteresting. The figure below shows the performance of TPUv3 relative to the Volta GPU chips for two sets of programs. The first set is five programs that Google and NVIDIA both submitted to MLPerf 0.6, and both use 16-bit arithmetic with NVIDIA software performing loss scaling. The TPUv3 geometric mean of these programs versus Volta is 0.95, so they are about the same speed. Google also wanted to measure performance of its production workloads, similar to what they used for TPUv1 in Section 6.7. The TPUv3 geometric mean speedup of the production applications over Volta was 4.8 for TPUv3, primarily because they use 8× slower fp32 on GPUs instead of fp16 (COD Figure 6.26). These are large production applications that are continuously improved, and not simple benchmarks, so it is a lot of work to get them to run at all, and more to run well. Applications programmers focus on TPUv3, since they are in everyday use, so there is little enthusiasm to include loss scaling needed for fp16.

Alas, only ResNet-50 from MLPerf 0.6 can scale beyond 1000 TPUs and GPUs. The next figure below shows ResNet-50 results for MLPerf 0.6; NVIDIA ran ResNet-50 on a cluster of 96 DGX-2H each with 16 Voltas connected via Infiniband switches at 41% of linear scale-up for 1536 chips. MLPerf 0.6 benchmarks are much smaller than the production applications; the time to train them is orders-of-magnitude less than production training runs. Thus, Google included production applications largely to show substantial programs that can scale to supercomputer size. One runs at 96% and three run at 99% of perfect linear scale-up for 1024 chips!

Figure 6.12.6: Performance per chip of TPUv3 relative to Volta for five MLPerf 0.6 benchmarks and four production applications (COD Figure 6.28).



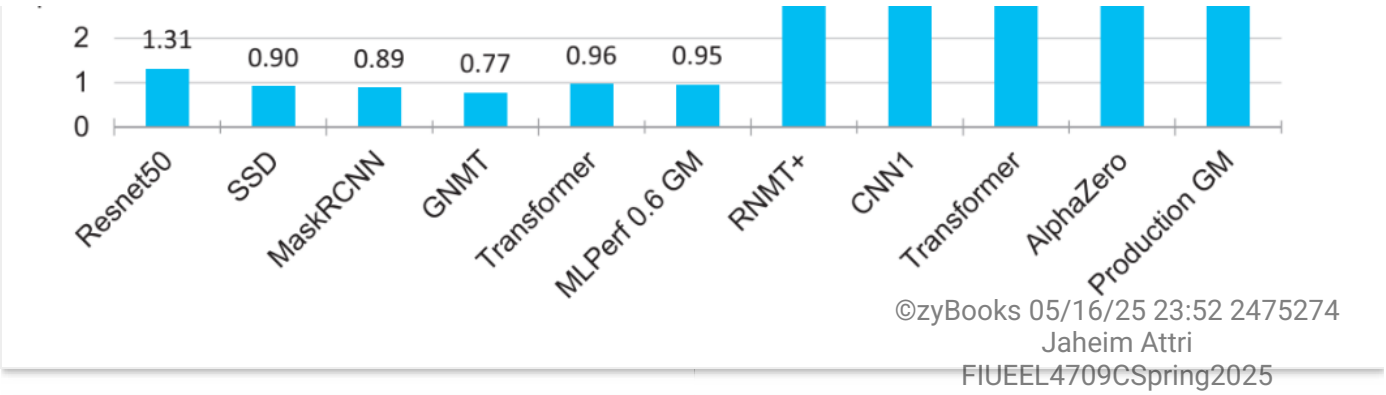
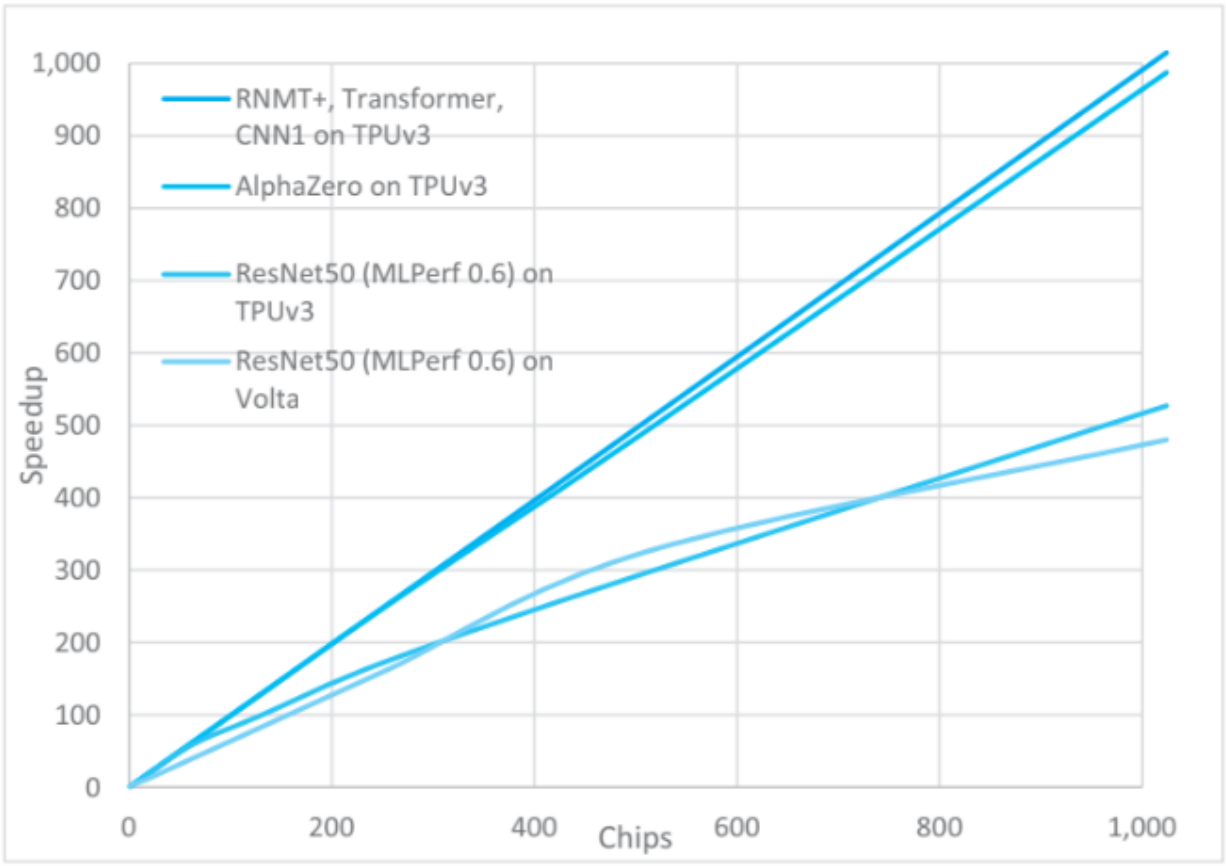


Figure 6.12.7: Supercomputer scaling: TPUv3 and Volta (COD Figure 6.29).



The figure below shows where PetaFLOPs/second and FLOPs/Watt of AlphaZero on TPUv3 would rank in the Top500 and Green500 lists. This comparison is imperfect: conventional supercomputers crunch 32- and 64-bit data rather than the 16- and 32-bit data of TPUs. However, TPUs are running a real application on real data versus weakly scaled Linpack benchmark on synthetic data. Remarkably, a TPUv3 supercomputer runs a production application using real world data at 70% of peak performance, higher than general-purpose supercomputers run the Linpack benchmark. Moreover, TPUv3 supercomputers with chips running a production application have 10× performance/Watt of the #1 traditional supercomputer on the Green500 list running Linpack

and 44× of the #4 supercomputer on the Top500 list.

Reasons for TPUv3's success include the built-in ICI network, large multiplier arrays, and bf16 arithmetic. TPUv3 has a smaller die in an older semiconductor process and lower cloud prices despite being less mature at many levels of hardware/software system stack than CPUs and GPUs. These good results despite technological disadvantages suggest the TPU DSA approach is cost-effective and can deliver high architectural efficiency into the future.

Now that we have seen a wide range of results of benchmarking different architectures, let us return to our DGEMM example to see in detail how much we have to change the C code to exploit multiple processors.

Figure 6.12.8: Traditional versus TPUv3 supercomputer Top500 and Green500 rank (June 2019) for Linpack and AlphaZero (COD Figure 6.30).

Name	Cores	Benchmark	Peta Flop/s	% of Peak	Megawatts	GFlops/ Watt	Top500	Green500
Tianhe	4865k	Linpack	61.4	61%	18.48	3.3	4	57
SaturnV	22k	Linpack	1.1	59%	0.97	15.1	469	1
TPUv3	2k	AlphaZero	86.9	70%	0.59	146.3	4	1

Elaboration

The original TPUv3 paper included two more production applications, MLP0 and MLP1. They rely on embeddings. An embedding at the start of a DNN model transforms from sparse representations into a dense representation suitable for linear algebra; embeddings also contain weights. Embeddings might use vectors where features can be represented by notions of distance between vectors. Embeddings involve table lookups, link traversal, and variable-length data fields, so they are irregular and memory intensive. TensorFlow kernels for embeddings had not been developed for GPUs, so Google excluded MLPs. On TPUv3, their speedups at 1024 chips are 14% and 40%, limited by the embeddings.

PARTICIPATION ACTIVITY 6.12.1: TPUv3 Volta.

1) What are the two phases of Deep Neural Networks, or DNNs?

- Training and learning
- Training and computation
- Training and inference

2) The TPUv3 uses a VLIW (Very Long Instruction Word) so that _____.

- a long sequence of operations can be pipelined
- data can be loaded along with the instruction to be executed
- several operations can be launched simultaneously

3) The peak performance for matrix multiply on the TPUv3 with 16-bit floating point instructions is _____ that of 32-bit floating point instructions.

- slower than
- the same as
- faster than

4) What interconnect topology exists in the TPUv3?

- 32 x 32 2D matrix
- 32 x 32 2D torus
- 32 x 32 2D crossbar

6.13 Going faster: Multiple processors and matrix multiply

(Original section[1])

This section is the final and largest step in our incremental performance journey of adapting DGEMM to the underlying hardware of the Intel Core i7 (Skylake). Each Core i7 has 24 cores, and the computer we have been using has 2 Core i7s. Thus, we have 48 cores on which to run DGEMM.

The figure below shows the OpenMP version of DGEMM that utilizes those cores. Note that line 27 is the *single* line added to COD Figure 5.48 (Optimized C version of DGEMM using cache blocking) to make this code run on multiple processors: an OpenMP pragma that tells the compiler to use multiple threads in the outermost for loop. It tells the computer to spread the work of the outermost loop across all the threads.

Figure 6.13.1: OpenMP version of DGEMM (COD Figure 6.31)

OpenMP version of DGEMM from COD Figure 5.48 (Optimized C version of DGEMM using cache b

Line 27 is the only OpenMP code, making the outermost for loop operate in parallel. This line is the difference from COD Figure 5.48 (Optimized C version of DGEMM using cache blocking).

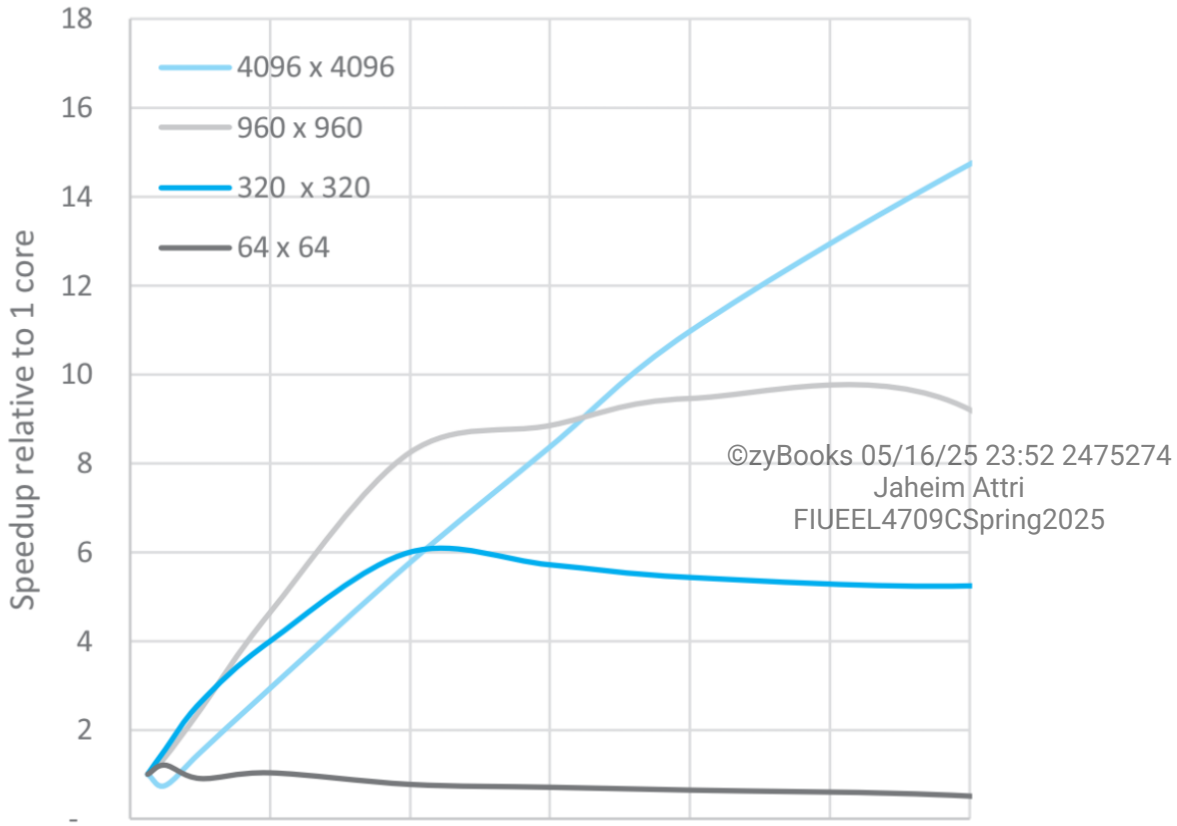
```
1  #include <x86intrin.h>
2  #define UNROLL (4)
3  #define BLOCKSIZE 32
4  void do_block (int n, int si, int sj, int sk,
5                 double *A, double *B, double *C)
6  {
7     for ( int i = si; i < si + BLOCKSIZE; i += UNROLL * 8 )
8        for ( int j = sj; j < sj + BLOCKSIZE; j++ ) {
9           __m512d c[UNROLL];
10          for ( int r = 0; r < UNROLL; r++ )
11             c[r] = _mm512_load_pd(C + i + r * 8 + j * n);   //[ UNRC
12
13          for( int k = sk; k < sk + BLOCKSIZE; k++ )
14          {
15             __m512d bb = _mm512_broadcastd_pd(_mm_load_sd(B + k + j
16             for (int r = 0; r < UNROLL; r++)
17                c[r] = _mm512_fmadd_pd(_mm512_load_pd(A+n*k+r*8+i), k
18          }
19
20          for ( int r = 0; r < UNROLL; r++ )
21             _mm512_store_pd(C + i + r * 4 + j * n, c[r]);
22       }
23 }
24
25 void dgemm (int n, double* A, double* B, double* C)
26 {
27 #pragma omp parallel for
```

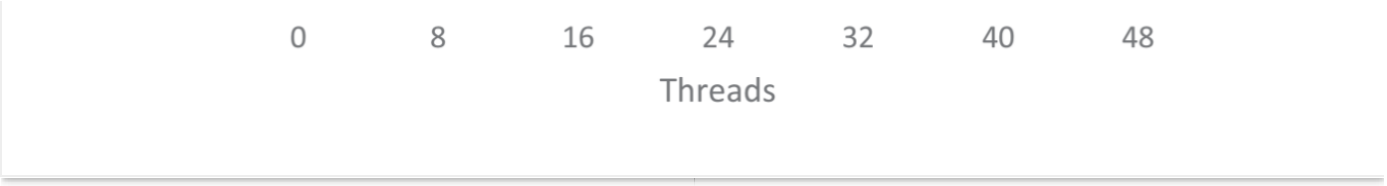
```
28    for ( int sj = 0; sj < n; sj += BLOCKSIZE )
29       for ( int si = 0; si < n; si += BLOCKSIZE )
30          for ( int sk = 0; sk < n; sk += BLOCKSIZE )
31             do_block(n, si, sj, sk, A, B, C);
32 }
```


The figure below plots a classic multiprocessor speedup graph, showing the performance improvement versus a single thread as the number of threads increase. This graph makes it easy to see the challenges of strong scaling versus weak scaling. When everything fits in the first level data cache, as is the case for 64 × 64 matrices, adding threads actually hurts performance. The 48-threaded version of DGEMM is almost half as fast as the single-threaded version in this case. In contrast, the two largest matrices get a 17 × speedup from 48 threads, and hence the classic two "up and to the right" lines in the figure below.

Figure 6.13.2: Performance improvements relative to a single thread as the number of threads increase (COD Figure 6.32).

The most honest way to present such graphs is to make performance relative to the best version of a single processor program, which we did. This plot is relative to the performance of the code in COD Figure 5.48 (Optimized C version of DGEMM using cache blocking) *without* including OpenMP pragmas.

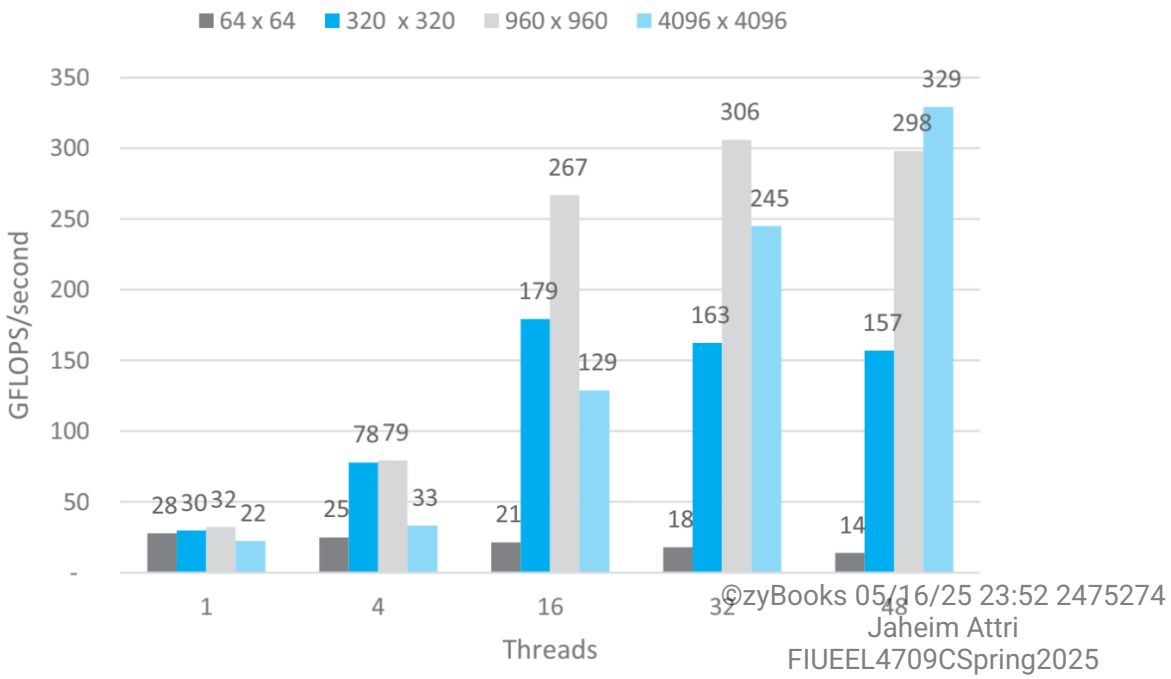




The figure below shows the absolute performance increase as we increase the number of threads from 1 to 48. DGEMM now operates at 308 GLOPS for 960 × 960 matrices. As our original C version of DGEMM in COD Figure 3.21 (Unoptimized C version of a double precision matrix multiply …) ran this code at just 2 GFLOPS, the optimizations in COD Chapter 3 (Arithmetic for Computers) to 6 (Parallel Processor from Client to Cloud) that tailor the code to the underlying hardware result in a speedup of almost 250 times! If we start with the Python version, the speedup is nearly 50,000 for a C version of DGEMM optimized for data-level parallelism, instruction-level parallelism, the memory hierarchy, and thread level parallelism.

Figure 6.13.3: COD Figure 6.33.

DGEMM performance versus the number of threads for four matrix sizes. The performance improvement compared to the original C code in COD Figure 2.43 for the 960 × 960 matrix with 32 threads is an astounding 150 times faster!



Next up is our warnings of the fallacies and pitfalls of multiprocessing. The computer architecture graveyard is filled with parallel processing projects that have ignored them.

PARTICIPATION 6.13.1: Multiple processors and matrix multiply.

ACTIVITY

1) As compared to the original Python version, the final performance improvement of the C version of DGEMM optimized for data-level parallelism, instruction-level parallelism, the memory hierarchy, and thread level parallelism is nearly _____.

   - ○ 50,000
   - ○ 500

2) Only 1 line of code makes the single threaded version work on multiple processors.

   - ○ True
   - ○ False

3) For a 64x64 matrix the performance of the multithreaded versions is _____ than the single threaded version.

   - ○ better
   - ○ poorer

Elaboration

Although the Skylake supports two hardware threads per core, we do not get more performance from 96 threads only for the 4096 × 4096: the peak is 364 GFLOPS at 64 threads, which drops to 344 at 96 threads. The reason is that a single AVX hardware is shared between the two threads multiplexed onto one core, so assigning two threads per core can hurt performance due to the multiplexing overhead if there is not enough data to keep all the threads busy.

(*1) This section is in original form.

6.14 Fallacies and pitfalls

(Original section[1])

> "What I was really frustrated about was the fact, with Iliac IV, programming the machine was very difficult and the architecture probably was not very well suited to some of the applications we were trying to run.
> *David Kuck, the sole software architect of the Illiac IV SIMD computer, circa 1975.*

The many assaults on parallel processing have uncovered numerous fallacies and pitfalls. We cover four here.

Fallacy: Amdahl's law doesn't apply to parallel computers.

In 1987, the head of a research organization claimed that a multiprocessor machine had broken Amdahl's Law. To try to understand the basis of the media reports, let's see the quote that gave us Amdahl's Law [1967, p. 483]:

> "A fairly obvious conclusion which can be drawn at this point is that the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude.

This statement must still be true; the neglected portion of the program must limit performance. One interpretation of the law leads to the following lemma: portions of every program must be sequential, so there must be an economic upper bound to the number of processors—say, 100. By showing linear speed-up with 1000 processors, this lemma is disproved; hence the claim that Amdahl's Law was broken.

The approach of the researchers was just to use weak scaling: rather than going 1000 times faster on the same data set, they computed 1000 times more work in comparable time. For their algorithm, the sequential portion of the program was constant, independent of the size of the input, and the rest was fully parallel—hence, linear speed-up with 1000 processors.

Amdahl's Law obviously applies to parallel processors. What this research does point out is that one of the main uses of faster computers is to run larger problems. Just be sure that users really care about those problems versus being a justification to buying an expensive computer by finding a problem that just keeps lots of processors busy.

Fallacy: Peak performance tracks observed performance.

The supercomputer industry once used this metric in marketing, and the fallacy is exacerbated with parallel machines. Not only are marketers using the nearly unattainable peak performance of a uniprocessor node, but also they are then multiplying it by the total number of processors, assuming perfect speed-up! Amdahl's Law suggests how difficult it is to reach either peak; multiplying the two together multiplies the sins. The roofline model helps put peak performance in perspective.

Pitfall: Not developing the software to take advantage of, or optimize for, a novel architecture.

There is a long history of parallel software lagging behind on parallel hardware, possibly because the software problems are much harder. There are many examples we could choose!

One frequently encountered problem occurs when software designed for a uniprocessor is adapted to a multiprocessor environment. For example, the Silicon Graphics operating system originally protected the page table with a single lock, assuming that page allocation is infrequent. In a uniprocessor, this does not represent a performance problem. In a multiprocessor, it can become a major performance bottleneck for some programs. Consider a program that uses a large number of pages that are initialized at start-up, which UNIX does for statically allocated pages. Suppose the program is parallelized so that multiple processes allocate the pages. Because page allocation requires the use of the page table, which is locked whenever it is in use, even an OS kernel that allows multiple threads in the OS will be serialized if the processes all try to allocate their pages at once (which is exactly what we might expect at initialization time!).

This page table serialization eliminates parallelism in initialization and has significant impact on overall parallel performance. This performance bottleneck persists even for task-level parallelism. For example, suppose we split the parallel processing program apart into separate jobs and run them, one job per processor, so that there is no sharing between the jobs. (This is exactly what one user did, since he reasonably believed that the performance problem was due to unintended sharing or interference in his application.) Unfortunately, the lock still serializes all the jobs—so even the independent job performance is poor.

This pitfall indicates the kind of subtle but significant performance bugs that can arise when software runs on multiprocessors. Like many other key software components, the OS algorithms and data structures must be rethought in a multiprocessor context. Placing locks on smaller portions of the page table effectively eliminated the problem.

A more recent example of this pitfall comes from DSAs for DNNs. More than 100 companies are developing them in 2020, and the MLPerf benchmark is determining their relative success. A common failure mode has been to develop novel hardware without a software stack that shows that hardware in its best light, which has already led to start-up companies going out of business a few years after they were founded.

Fallacy: You can get good vector performance without providing memory

bandwidth.

As we saw with the Roofline model, memory bandwidth is quite important to all 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 DAXPY performance of the vector sequence used, since it was 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, which is just what the Roofline model would predict.

Pitfall: Assuming the instruction set architecture (ISA) completely hides all physical implementation properties.

Timing channels have been known as a vulnerability since at least the 1980s, but most architects incorrectly considered them practically unimportant. However, implementation properties, such as timing, can affect function. This pitfall was prominently exhibited by the 2018 exposure of Spectre that used microarchitecture speculation to leak private information to user-level attacker code from user-level sandboxes, the kernel, or the hypervisor. Spectre exploited these three micro-architecture techniques:

1. **Instruction Speculation:** A processor core seeks to execute dozens of instructions concurrently by speculating past branches, committing ISA changes if speculation is correct, and rolling them back when speculation is wrong. Perversely, Spectre speculatively executes instructions whose ISA changes it knows will be rolled back. Its subtle goal is to leave microarchitectural "breadcrumbs" of what the programmer thinks are hidden secrets.

2. **Caching:** Caches are invisible to an ISA. In particular, according to conventional computer architecture wisdom, which block is least recently used in a set associative cache did not matter for proper execution, so its status need not be restored on mispeculation. Spectre exploits this surprising vulnerability to place and later find "breadcrumbs" that reveal a secret. It thus uses the contents of a cache as a "side channel" to transmit a (secret) data value.

3. **Hardware multithreading:** It is much easier to notice such subtle timing changes if the attacking program can run in close proximity to the target program. Hardware multithreading, where instructions from one program can intermix with others, simplifies this task. Hardware attacks are worrisome enough that cloud providers now offer the option to prevent sharing your server with programs of other customers. For example, AWS offers "Dedicated Instances," which cost about 5% more than the traditional shared instances.

Source: This pitfall is derived from Mark Hill's perspective in Communications of the ACM , 2020, "Why 'Correct' Computers Can Leak Your Information," and was written with his help.

(*1) This section is in original form.

6.15 Concluding remarks

(Original section[1])

> "We are dedicating all of our future product development to multicore designs. We believe this is a key in inflection point for the industry. … This is not a race. This is a sea change in computing …"
> *Paul Otellini, Intel President, Intel Developers Forum, 2004*

The dream of building computers by simply aggregating processors has been around since the earliest days of computing. Progress in building and using effective and efficient parallel processors, however, has been slow. This rate of progress has been limited by difficult software problems as well as by a long process of evolving the architecture of multiprocessors to enhance usability and improve efficiency. We have discussed many of the software challenges in this chapter, including the difficulty of writing programs that obtain good speed-up due to Amdahl's Law. The wide variety of different architectural approaches and the limited success and short life of many of the parallel architectures of the past have compounded the software difficulties. We discuss the history of the development of these multiprocessors in online COD Section 6.15 (Historical perspective and further reading). To go into even greater depth on topics in this chapter, see Chapter 4 (Data-Level Parallelism in Vector, SIMD, and GPU Architectures) of *Computer Architecture: A Quantitative Approach, Sixth Edition* for more on GPUs and comparisons between GPUs and CPUs and Chapter 6 (Parallel Processor from Client to Cloud) for more on WSCs and Chapter 7 for more DSAs.

As we said in COD Chapter 1 (Computer Abstractions and Technology), despite this long and checkered past, the information technology industry has now tied its future to parallel computing. Here are some reasons it's different now than in the past:

- Clearly, *software as a service* (SaaS) is growing in importance, and clusters have proven to be a very successful way to deliver such services. By providing redundancy at a higher-level, including geographically distributed datacenters, such services have delivered 24 × 7 × 365 availability for customers around the world.
- Warehouse-Scale Computers are changing the goals and principles of server design, just as the needs of mobile clients are changing the goals and principles of microprocessor design. Both are revolutionizing the software industry as well. Performance per dollar and performance per joule drive both mobile client hardware and the WSC hardware, and parallelism is the key to delivering on those sets of goals.
- SIMD and vector operations are a good match to multimedia applications, which are playing a larger role in the postPC Era. They share the advantage of being easier for the programmer than classic parallel MIMD programming and being more energy efficient than MIMD.
- The rapidly rising popularity of machine learning is changing the nature of applications, and the neural network models that drive machine learning are naturally parallel. Moreover, domain specific software platforms like PyTorch and TensorFlow operate on arrays, making it much easier to express and exploit data level parallelism than programs written in C++.
- All desktop and server microprocessor manufacturers are building multiprocessors to achieve higher performance, so, unlike in the past, there is no easy path to higher performance for sequential applications.
- In the past, microprocessors and multiprocessors were subject to different definitions of success. When scaling uniprocessor performance, microprocessor architects were happy if single thread performance went up by the square root of the increased silicon area. Thus, they were happy with sublinear performance in terms of resources. Multiprocessor success used to be defined as *linear* speed-up as a function of the number of processors, assuming that the cost of purchase or cost of administration of *n* processors was *n* times as much as one processor. Now that parallelism is happening on-chip via multicore, we can use the traditional microprocessor metric of being successful with sublinear performance improvement.
- Unlike in the past, the open source movement has become a critical portion of the software industry. This movement is a meritocracy, where better engineering solutions can win the mind share of the developers over legacy concerns. It also embraces innovation, inviting change to old software and welcoming new languages and software products. Such an open culture could be extremely helpful in this time of rapid change.

To motivate readers to embrace this revolution, we demonstrated the potential of parallelism concretely for matrix multiply on the Intel Core i7 (Skyelake) in the Going Faster sections of COD Chapters 3 (Arithmetic for Computers) to 6 (Parallel Processor from Client to Cloud):

- Data-level parallelism in COD Chapter 3 (Arithmetic for Computers) improved performance by a factor of 7.8 by executing eight 64-bit floating-point operations in parallel using the 512-bit operands of the AVX instructions, demonstrating the value of SIMD.
- Instruction-level parallelism in COD Chapter 4 (The Processor) pushed performance up by another factor of 1.8 by unrolling loops four times to give the out-of-order execution hardware

more instructions to schedule.
- Cache optimizations in COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy) improved performance of matrices that didn't fit into the L1 data cache by another factor of 1.5 by using cache blocking to reduce cache misses.
- Thread-level parallelism in this chapter improved performance of matrices that don't fit into a single L1 data cache by another factor of 12 to 17 by utilizing all 48 cores of our multicore chips, demonstrating the value of MIMD. We did this by adding a single line using an OpenMP pragma.

Using the ideas in this book and tailoring the software to this computer added 24 lines of code to DGEMM. The overall performance speedup from these ideas realized in those two-dozen lines of code is over a factor of 150!

In an era without Dennard scaling, a reduced Moore's Law, and Amdahl's Law in full effect will see improvements in performance of general-purpose cores of only a few percent per year. Just as the industry spent a decade starting in about 2005 to try to exploit the opportunity of parallel processing, we project the challenge of the next decade will be to develop and program DSAs.

This sea change will provide many new research and business prospects inside and outside the IT field, and the companies that dominate the DSA era may not be the same ones that dominate it today. After understanding the underlying hardware trends and how to adapt software to them that you gained from this book, perhaps you will be one of the innovators who will seize the opportunities that are certain to appear in the uncertain times ahead. We look forward to benefiting from your inventions!

(*1) This section is in original form.

6.16 Historical perspective and further reading

There is a tremendous amount of history in multiprocessors; in this section we divide our discussion by both time period and architecture. We start with the SIMD approach and the Illiac IV. We then turn to a short discussion of some other early experimental multiprocessors and progress to a discussion of some of the great debates in parallel processing. Next we discuss the historical roots of the present **multiprocessors** and conclude by discussing recent advances.

SIMD computers: Attractive idea, many attempts, no lasting successes

> "The cost of a general multiprocessor is, however, very high and further design options were considered which would decrease the cost without seriously degrading the power or effciency of the system. The options consist of recentralizing one of the three major components. … Centralizing the [control unit] gives rise to the basic organization of [an] …

> array processor such as the Illiac IV.
> *Bouknight et al. [1972]*

The SIMD model was one of the earliest models of parallel computing, dating back to the first large-scale multiprocessor, the Illiac IV. The key idea in that multiprocessor, as in more recent SIMD multiprocessors, is to have a single instruction that operates on many data items at once, using many functional units (see the figure below).

Figure 6.16.1: The Illiac IV control unit followed by its 64 processing elements (COD Figure e6.16.1).

It was perhaps the most infamous of supercomputers. The project started in 1965 and ran its first real application in 1976. The 64 processors used a 13-MHz clock, and their combined main memory size was 1 MB: 64 × 16 KB. The Illiac IV was the first machine to teach us that software for parallel machines dominates hardware issues. Photo courtesy of NASA Ames Research Center.

Although successful in pushing several technologies that proved useful in later projects, it failed as

a computer. Costs escalated from the $8 million estimate in 1966 to $31 million by 1972, despite construction of only a quarter of the planned multiprocessor. Actual performance was at best 15 MFLOPS, versus initial predictions of 1000 MFLOPS for the full system [Hord, 1982]. Delivered to NASA Ames Research in 1972, the computer required three more years of engineering before it was usable.

These events slowed investigation of SIMD, with Danny Hillis [1985] resuscitating this style in the Connection Machine, which had 65,536 1-bit processors.

Real SIMD computers need to have a mixture of SISD and SIMD instructions. There is an SISD host computer to perform operations such as branches and address calculations that do not need parallel operation. The SIMD instructions are broadcast to all the execution units, each of which has its own set of registers. For flexibility, individual execution units can be disabled during an SIMD instruction. In addition, massively parallel SIMD multiprocessors rely on interconnection or communication networks to exchange data between processing elements.

The basic tradeoff in SIMD multiprocessors is performance of a processor versus number of processors. More recent SIMDs emphasize a large degree of parallelism over performance of the individual processors. The Connection Multiprocessor 2, for example, offered 65,536 single-bit-wide processors, while the Illiac IV had 64 64-bit processors.

After being resurrected in the 1980s, first by Thinking Machines and then by MasPar, the SIMD model has once again been put to bed as a general-purpose multiprocessor architecture, for two main reasons. First, it is too inflexible. A number of important problems cannot use such a style of multiprocessor, and the architecture did not scale down in a competitive fashion; that is, small-scale SIMD multiprocessors often had worse cost performance than that of the alternatives. Second, SIMD did not take advantage of the tremendous performance and cost advantages of microprocessor technology. Instead of leveraging this low-cost technology that was improving rapidly during the height of Moore's Law and Dennard Scaling, designers of SIMD multiprocessors built custom processors for their multiprocessors.

Although SIMD computers have departed from the scene as general-purpose alternatives, this style of architecture play an important role in special-purpose designs. Many special-purpose tasks are highly data parallel and require a limited set of functional units. Thus, designers can build in support for certain operations, as well as hardwired interconnection paths among functional units. Such organizations are often called array processors, and they are useful for tasks like image processing, signal processing, and machine learning, as we see with TPUs.

Multimedia extensions as SIMD extensions to instruction sets

Many recent architectures have laid claim to being the first to offer multimedia extensions, in which a set of new instructions takes advantage of a single wide ALU that can be partitioned so that it will act as several narrower ALUs operating in parallel. It's unlikely that any appeared before 1957, however, when the Lincoln Lab's TX-2 computer offered instructions that operated on the ALU as either one 36-bit operation, two 18-bit operations, or four 9-bit operations. Ivan Sutherland,

considered the Father of Computer Graphics, built his historic Sketchpad system on the TX-2. Sketchpad did, in fact, take advantage of these SIMD instructions, despite TX-2 appearing before invention of the term SIMD in 1972.

Other early experiments

It is difficult to distinguish the first MIMD multiprocessor. Surprisingly, the first computer from the Eckert-Mauchly Corporation, for example, had duplicate units to improve **availability**.

Two of the best-documented multiprocessor projects were undertaken in the 1970s at Carnegie Mellon University. The first of these was C.mmp, which consisted of 16 PDP-11s connected by a crossbar switch to 16 memory units. It was among the first multiprocessors with more than a few processors, and it had a shared memory programming model. Much of the focus of the research in the C.mmp project was on software, especially in the OS area. A later multiprocessor, Cm*, was a cluster-based multiprocessor with a distributed memory and a nonuniform access time. The absence of caches and a long remote access latency made data placement critical. Many of the ideas in these multiprocessors would be reused in the 1980s, when the microprocessor made it much cheaper to build multiprocessors.

Great debates in parallel processing

> "The turning away from the conventional organization came in the middle 1960s, when the law of diminishing returns began to take effect in the effort to increase the operational speed of a computer. … Electronic circuits are ultimately limited in their speed of operation by the speed of light … and many of the circuits were already operating in the nanosecond range.
> *W. Jack Bouknight, et al. The Illiac IV System [1972]*

> "… sequential computers are approaching a fundamental physical limit on their potential computational power. Such a limit is the speed of light …
> *Angel L. DeCegama, The Technology of Parallel Processing, Volume I [1989]*

> "… today's multiprocessors … are nearing an impasse as technologies approach the speed of light. Even if the components of a sequential processor could be made to work this fast, the best that could be expected is no more than a few million instructions per second.
> *David Mitchell, The Transputer: The Time Is Now [1989]*

The quotes above give the classic arguments for abandoning the current form of computing, and

Amdahl [1967] gave the classic reply in support of continued focus on the IBM 360 architecture. Arguments for the advantages of parallel execution can be traced back to the 19th century [Menabrea, 1842]! Aside from these debates about the advantages and limitations of parallelism, several hot debates have focused on how to build multiprocessors.

From today's perspective, it is clear that the speed of light was not the brick wall; the brick wall was, instead, the power consumption of CMOS as the clock rates increased.

It's hard to predict the future, yet in 1989 Gordon Bell made two predictions for 1995. We included these predictions in the first edition of Computer Architecture: A Quantitative Approach, when the outcome was completely unclear. We discuss them in this section, together with an assessment of the accuracy of the prediction.

The first was that a computer capable of sustaining a tera FLOPS—one million MFLOPS—would be constructed by 1995, using either a multicomputer with 4K to 32K nodes or a Connection Multiprocessor with several million processing elements. To put this prediction in perspective, each year the Gordon Bell Prize acknowledges advances in parallelism, including the fastest real program (highest MFLOPS). In 1989 the winner used an eight-processor Cray Y-MP to run at 1680 MFLOPS. On the basis of these numbers, multiprocessors and programs would had to have improved by a factor of 3.6 each year for the fastest program to achieve 1 TFLOPS in 1995. In 1999, the first Gordon Bell prize winner crossed the 1 TFLOPS bar. Using a 5832-processor IBM RS/6000 SST system designed specially for Livermore Laboratories, they achieved 1.18 TFLOPS on a shock wave simulation. This ratio represents a year-to-year improvement of 1.93, which is still quite impressive.

What has been recognized since the 1990s is that although we may have the technology to build a TFLOPS multiprocessor, it was not clear that the machine was cost effective, except perhaps for a few very specialized and critically important applications related to national security. We estimated in 1990 that achieving 1 TFLOPS would require a machine with about 5000 processors and would cost about $100 million. The 5832-processor IBM system at Livermore cost $110 million. As might be expected, improvements in the performance of individual microprocessors both in cost and performance directly affect the cost and performance of large-scale multiprocessors, but a 5000-processor system would cost more than 5000 times the price of a desktop system using the same processor. Since that time, much faster multiprocessors have been built, but the major improvements have increasingly come from the recent processors, rather than fundamental breakthroughs in parallel architecture.

The second Bell prediction concerned the number of data streams in supercomputers shipped in 1995. Danny Hillis believed that although supercomputers with a small number of data streams might be the best sellers, the biggest multiprocessors would be multiprocessors with many data streams, and these would perform the bulk of the computations. Bell bet Hillis that in the last quarter of calendar year 1995, more sustained MFLOPS would be shipped in multiprocessors using few data streams (<100) rather than many data streams (>1000). This bet concerned only supercomputers, defined as multiprocessors costing more than $1 million and used for scientific applications. Sustained MFLOPS was defined for this bet as the number of floating-point

operations per month, so availability of multiprocessors affects their rating.

In 1989, when this bet was made, it was totally unclear who would win. In 1995, a survey of the current publicly known supercomputers showed only six multiprocessors in existence in the world with more than 1000 data streams, so Bell's prediction was a clear winner. In fact, in 1995, much smaller microprocessor-based multiprocessors (<20 processors) were becoming dominant.

In 1995, the Top500 showed that the largest number of multiprocessors were bus-based shared memory multiprocessors! By 2005, various clusters or multicomputers played a large role. For example, in the top 25 systems, 11 were custom clusters, such as the IBM Blue Gene system or the Cray XT3, 10 were clusters of shared memory multiprocessors (both using distributed and centralized memory), and the remaining 4 were clusters built using PCs with an off-the-shelf interconnect.

More recent advances and developments

With the primary exception of the parallel vector multiprocessors and more recently of the IBM Blue Gene design, all other recent MIMD computers have been built from off-the-shelf microprocessors using a bus and logically central memory or an interconnection network and a distributed memory. A number of experimental multiprocessors built in the 1980s further refined and enhanced the concepts that form the basis for many of today's multiprocessors.

The development of bus-based coherent multiprocessors

Although very large mainframes were built with multiple processors in the 1960s and 1970s, multiprocessors did not become highly successful until the 1980s. Bell [1985] suggests the key was that the smaller size of the microprocessor allowed the memory bus to replace the interconnection network hardware and that portable operating systems meant that multiprocessor projects no longer required the invention of a new operating system. In this paper, Bell defined the terms multiprocessor and multicomputer and set the stage for two different approaches to building larger-scale multiprocessors. The first bus-based multiprocessor with snooping caches was the Synapse N + 1 in 1984.

The early 1990s saw the beginning of an expansion of such systems with the use of very wide, high-speed buses (the SGI Challenge system used a 256-bit, packet-oriented bus supporting up to 8 processor boards and 32 processors) and later the use of multiple buses and crossbar interconnects, for example, in the Sun SPARCCenter and Enterprise systems. In 2001, the Sun Enterprise servers represented the primary example of large-scale (>16 processors), symmetric multiprocessors in active use.

Toward large-scale multiprocessors

In the effort to build large-scale multiprocessors, two different directions were explored: message-passing multicomputers and scalable shared memory multiprocessors. Although there had been

many attempts to build mesh and hypercube-connected multiprocessors, one of the first multiprocessors to successfully bring together all the pieces was the Cosmic Cube built at Caltech [Seitz, 1985]. It introduced important advances in routing and interconnect technology and substantially reduced the cost of the interconnect, which helped make the multicomputer viable. The Intel iPSC 860, a hypercube-connected collection of i860s, was based on these ideas. More recent multiprocessors, such as the Intel Paragon, have used networks with lower dimensionality and higher individual links. The Paragon also employed a separate i860 as a communications controller in each node, although a number of users have found it better to use both i860 processors for computation as well as communication. The Thinking Machines CM-5 made use of off-the-shelf microprocessors. It provided user-level access to the communication channel, significantly improving communication latency. In 1995, these two multiprocessors represented the state of the art in message-passing multicomputers.

Clusters

Clusters were probably "invented" in the 1960s by customers who could not fit all their work on one computer, or who needed a backup machine in case of failure of the primary machine [Pfister, 1998]. Tandem introduced a 16-node cluster in 1975. Digital followed with VAX clusters, introduced in 1984. They were originally independent computers that shared I/O devices, requiring a distributed operating system to coordinate activity. Soon they had communication links between computers, in part so that the computers could be geographically distributed to increase availability in case of a disaster at a single site. Users logged onto to the cluster and were unaware of which machine they are using. DEC (now HP) sold more than 25,000 clusters by 1993. Other early companies were Tandem (now HP) and IBM (still IBM). Virtually every company has cluster products. Most of these products are aimed at availability, with performance scaling as a secondary benefit.

Scientific computing on clusters emerged as a competitor to MPPs. In 1993, the Beowulf project started with the goal of fulfilling NASA's desire for a 1-GFLOPS computer for less than $50,000. In 1994, a 16-node cluster built from off-the-shelf PCs using 80486s achieved that goal. This emphasis led to a variety of software interfaces to make it easier to submit, coordinate, and debug large programs or a large number of independent programs.

Efforts were made to reduce latency of communication in clusters as well as to increase bandwidth, and several research projects worked on that problem. (One commercial result of the low-latency research was the VI interface standard, which has been embraced by Infiniband, discussed below.) Low latency then proved useful in other applications. For example, in 1997 a cluster of 100 UltraSPARC desktop computers at U.C. Berkeley, connected by 160MB/sec per link Myrinet switches, was used to set world records in database sort (sorting 8.6 GB of data originally on disk in 1 minute) and in cracking an encrypted message (taking just 3.5 hours to decipher a 40-bit DES key).

This research project, called Network of Workstations, also developed the Inktomi search engine, which led to a start-up company with the same name. Google followed the example of Inktomi to

build search engines from clusters of desktop computers rather than large-scale SMPs, which was the strategy of the leading search engine, Alta Vista, that Google took over. In 2020, all Internet services rely on clusters to serve their millions of customers.

Clusters are also very popular with scientists. One reason is their low cost, which enables individual scientists or small groups to own a cluster dedicated to their programs. Such clusters can get results faster than waiting in the long job queues of the shared MPPs at supercomputer centers, which can stretch to weeks.

For those interested in learning more, Pfister [1998] has written an entertaining book on clusters.

Recent trends in large-scale multiprocessors

In the mid-to-late 1990s, it became obvious that the hoped-for growth in the market for ultralarge-scale parallel computing was unlikely to occur. Without this market growth, it became increasingly clear that the high-end parallel computing market was too small to support the costs of highly customized hardware and software designed for a small market. Perhaps the most important trend to come out of this observation was that clustering would be used to reach the highest levels of performance. There are now three general classes of large-scale multiprocessors:

1. Clusters that integrate standard desktop motherboards using interconnection technology, such as Ethernetor Infiniband
2. Multicomputers built from standard microprocessors configured into processing elements and connected with a custom interconnect, such as the IBM Blue Gene
3. Clusters of small-scale shared memory computers, possibly with vector support, including the Earth Simulator

Blue Gene is constructed using a custom chip that includes an embedded PowerPC microprocessor offering half the performance of a high-end PowerPC, but at a much smaller fraction of the area and the power. This allowed more system functions, including the global interconnect, to be integrated onto the same die.

Looking further

There is an almost unbounded amount of information on multiprocessors and multicomputers: conferences, journal papers, and even books seem to appear faster than any single person can absorb the ideas. No doubt many of these papers will go unnoticed—not unlike the past. Most of the major architecture conferences contain papers on multiprocessors. An annual conference, SC XY (where X and Y are the last two digits of the year), brings together users, architects, software developers, and vendors and publishes the proceedings in book, CD-ROM, and online (see www.scXY.org) form. Two major journals, *Journal of Parallel and Distributed Computing* and the *IEEE Transactions on Parallel and Distributed Systems*, contain papers on all aspects of parallel processing. Several books focusing on parallel processing are included in the following references.

Asanovic, et al. [2006] surveyed the wide-ranging challenges for the industry in this multicore

challenge. That report may be a helpful in understanding the depth of the various challenges.

In addition to documenting the discovery of concepts now used in practice, these references also provide descriptions of many ideas that have been explored and found wanting, as well as ideas whose time has just not yet come. Given the move toward multicore and multiprocessors as the future of high-performance computer architecture, we expect that many new approaches will be explored in the years ahead. A few of them will manage to solve the hardware and software problems that have been the key to using multiprocessing for the past 40 years!

In an era without Dennard scaling, a reduced Moore's Law, and Amdahl's Law in full effect will see improvements in performance of general-purpose cores of only a few percent per year. Just as the industry spent a decade starting in about 2005 to try to exploit the opportunity of parallel processing, we project the challenge of the next decade will be to develop and program domain specific architectures (DSAs). This sea change will provide many new research and business prospects inside and outside the IT field, and the companies that dominate the DSA era may not be the same ones that dominate it today. After understanding the underlying hardware trends and how to adapt software to them that you gained from this book, perhaps you will be one of the innovators who will seize the opportunities that are certain to appear in the uncertain times ahead. We look forward to benefiting from your inventions!

Further reading

Almasi, G. S. and A. Gottlieb [1989]. *Highly Parallel Computing*, Benjamin/Cummings, Redwood City, CA.
A textbook covering parallel computers.

Amdahl, G. M. [1967]. "Validity of the single processor approach to achieving large scale computing capabilities," *Proc. AFIPS Spring Joint Computer Conf.*, Atlantic City, NJ (April), 483—85.
Written in response to the claims of the Illiac IV, this three-page article describes Amdahl's law and gives the classic reply to arguments for abandoning the current form of computing.

Andrews, G. R. [1991]. *Concurrent Programming: Principles and Practice*, Benjamin/Cummings, Redwood City, CA.
A text that gives the principles of parallel programming.

Archibald, J. and J.-L. Baer [1986]. "Cache coherence protocols: Evaluation using a multiprocessor simulation model," ACM Trans. on Computer Systems 4:4 (November), 273—98.
Classic survey paper of shared-bus cache coherence protocols.

Arpaci-Dusseau, A., R. Arpaci-Dusseau, D. Culler, J. Hellerstein, and D. Patterson [1997]. "High-performance sorting on networks of workstations," Proc. ACM SIG MOD/PODS Conference on Management of Data, Tucson, AZ (May), 12—15.
How a world record sort was performed on a cluster, including architecture critique of the workstation and network interface. By April 1, 1997, they pushed the record to 8.6 GB in 1 minute and 2.2 seconds to sort 100 MB.

Asanovic, K., R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. [2006]. "The landscape of parallel computing research: A view from Berkeley." Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley (December 18).
Nicknamed the "Berkeley View," this report lays out the landscape of the multicore challenge.

Bailey, D. H., E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. [1991]. "The NAS parallel benchmarks—summary and preliminary results," *Proceedings of the 1991 ACM/IEEE conference on Super-computing* (August).
Describes the NAS parallel benchmarks.

Bell, C. G. [1985]. "Multis: A new class of multiprocessor computers," *Science* 228 (April 26), 462—467.
Distinguishes shared address and nonshared address multiprocessors based on micro processors.

Bienia, C., S. Kumar, J. P. Singh, and K. Li [2008]. "The PARSEC benchmark suite: characterization and architectural implications," Princeton University Technical Report TR-81 1-008 (January).
Describes the PARSEC parallel benchmarks. Also see http://parsec.cs.princeton.edu/.

Bouknight, W. J., Denenberg, S. A., McIntyre, D. E., Randall, J. M., Sameh, A. H., & Slotnick, D. L. [1972]. The Illiac IV system. Proceedings of the IEEE, 60(4), 369—388.
It describes the most infamous SIMD supercomputer.

Culler, D. E. and J. P. Singh, with A. Gupta [1998]. *Parallel Computer Architecture*, Morgan Kaufmann, San Francisco.
A textbook on parallel computers.

Dongarra J. J., J. R. Bunch, G. B. Moler, G. W. Stewart [1979]. *LINPACK Users' Guide*, Society for Industrial Mathematics.
The original document describing Linpack, which became a widely used parallel benchmark.

Falk, H. [1976]. "Reaching for the gigaflop," *IEEE Spectrum* 13:10 (October), 65—70.
Chronicles the sad story of the Illiac IV: four times the cost and less than one-tenth the performance of original goals.

Flynn, M. J. [1966]. "Very high-speed computing systems," *Proc. IEEE* 54:12 (December), 1901—09.
Classic article showing SISD/SIMD/MISD/MIMD classifications.

Hennessy, J. and D. Patterson [2003]. Chapters 6 and 8 in *Computer Architecture: A Quantitative Approach*, third edition, Morgan Kaufmann Publishers, San Francisco.
A more in-depth coverage of a variety of multiprocessor and cluster topics, including programs and measurements.

Henning, J. L. [2007]. "SPEC CPU suite growth: an historical perspective," *Computer Architecture News*, Vol. 35, no. 1 (March).
Gives the history of SPEC, including the use of SPECrate to measure performance on independent jobs, which is being used as a parallel benchmark.

Hillis, W. D. [1989]. The connection machine. The MIT Press.
PhD Dissertation that makes case for 1-bit SIMD computer

Hord, R. M. [1982]. *The Illiac-IV, the First Supercomputer*, Computer Science Press, Rockville, MD.
A historical accounting of the Illiac IV project.

Hwang, K. [1993]. *Advanced Computer Architecture with Parallel Programming*, McGraw-Hill, New York.
Another textbook covering parallel computers.

Kozyrakis, C. and D. Patterson [2003]. "Scalable vector processors for embedded systems," *IEEE Micro* 23:6 (November-December), 36—45.

Examination of a vector architecture for the MIPS instruction set in media and signal processing.

Menabrea, L. F. [1842]. "Sketch of the analytical engine invented by Charles Babbage," *Bibliothèque Universelle de Genève* (October).
Certainly the earliest reference on multiprocessors, this mathematician made this comment while translating papers on Babbage's mechanical computer.

Pfister, G. F. [1998]. *In Search of Clusters: The Coming Battle in Lowly Parallel Computing*, second edition, Prentice Hall, Upper Saddle River, NJ.
An entertaining book that advocates clusters and is critical of NUMA multiprocessors.

Regnier, G.,S. Makineni, I. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, and A. Foong [2004]. TCP onloading for data center servers. Computer, 37(11), 48-58.
A paper describing benefits of doing TCP/IP inside servers vs. external hardware.

Seitz, C. [1985]. "The Cosmic Cube," *Comm. ACM* 28:1 (January), 22—31.
A tutorial article on a parallel processor connected via a hypertree. The Cosmic Cube is the ancestor of the Intel supercomputers.

Slotnick, D. L. [1982]. "The conception and development of parallel processors—a personal memoir," *Annals of the History of Computing* 4:1 (January), 20—30.
Recollections of the beginnings of parallel processing by the architect of the Illiac IV.

Williams, S., J. Carter, L. Oliker, J. Shalf, and K. Yelick [2008]. "Lattice Boltzmann simulation optimization on leading multicore platforms," *International Parallel & Distributed Processing Symposium (IPDPS)*.
Paper containing the results of the four multicores for LBMHD.

Williams, S., L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel [2007]. "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," *Supercomputing* (SC).
Paper containing the results of the four multicores for SPmV.

Williams, S. [2008]. *Autotuning Performance of Multicore Computers*, Ph.D. Dissertation, U.C. Berkeley.
Dissertation containing the roofline model.

Woo, S. C., M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. "The SPLASH-2 programs: characterization and methodological considerations," *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*, May, 24—36.
Paper describing the second version of the Stanford parallel benchmarks.

6.17 Self study

Domain Specific Architectures (DSAs) are leading to more computing options and a greater need to compare the costs of alternatives. For example, how do we compare the cost of running a program on a general-purpose CPU, a GPU, or an FPGA? Costs are traditionally difficult to measure, as the list price may not be what customers really have to pay, especially if they are buying a large number of computers.

Cloudy Prices. One marketplace where the prices are fixed and public for everyone is the cloud. Go to a favorite cloud provider and find the current hourly cost to rent CPU, an FPGA, and a GPU. For example, at AWS in 2020 example instances are

- CPU: r5.2xlarge
- FPGA: f1.2xlarge
- GPU: p3.2xlarge

What are the rental prices of the FPGA and GPU relative to CPU?

Enhanced Genomes. One estimate is that the total number of people whose genomes have been sequenced is about 1 million as of 2020. The dropping cost of genome sequencing could lead to large demand to analyze the raw sequencing data. A paper by Lisa Wu et al [Wu, 2019] used a DSA implemented in an FPGA to accelerate a critical piece of genomic analysis from 42 hours on a CPU to 31 minutes on an FPGA. Although Wu et al were skeptical that the program would run faster on GPUs due to load imbalances between thread, for the sake of argument, let's suppose it runs three times as fast on a GPU as a CPU. Using your answer to Cloudy Prices, what are the costs to sequence a genome on each platform? What are the costs of the FPGA and GPU relative to CPU?

Really Enhanced Genomes. A rough rule of thumb is that a custom chip is at least ten times as fast as the equivalent design in an FPGA. The issue is that a custom chip has a much higher development cost ("Non Recurring Costs" or NRE) than an FPGA. Michael Taylor and his students did some novel investigations to establish these costs [Magaki *et al*., 2016; Khazraee *et al*., 2017]. The ASIC NRE must include the cost of making the masks, and they are a substantial part of the overall costs, as this table shows for some example designs as of 2017 [Khazraee *et al*., 2017]. The authors point out that ASICs are so much faster than the alternative that the main question is how to pay for the NRE.

Technology	40nm	28nm	16nm
Mask Cost	$1,250,000	$2,250,000	$5,700,000
% of overall NRE	38%	52%	66%
Total NRE	$3,259,000	$4,301,000	$8,616,000

How many genomes do you need to sequence to recover the NRE for each ASIC design? The wet lab cost of genome sequencing in 2020 is about $700 per genome. Would you use FPGAs or custom ASICs for data processing?

Answers to Self-Study

Cloudy Prices for AWS US East in 2020.

- CPU r5.2xlarge: $0.504 per Hour
- CPU p3.2xlarge: $3.06 per Hour. It costs 6.1 times as much as the CPU.
- CPU f1.2xlarge: $1.65 per Hour. It costs 3.3 times as much as the CPU.

Enhanced Genomes.

- 42 hours * $0.504 per Hour = $21.17 to sequence one genome.
- 31 minutes/60 minutes per hour * $1.65 per Hour = $0.85. FPGAs cost 0.04 times as much as CPUs (1/25th)
- 42/3d hours * $3.06 per Hour = $21.17 = $42.84. GPUs cost 2.0 times as much as CPUs

Really Enhanced Genomes.

Technology	40nm	28nm	16nm
Total NRE	$3,259,000	$4,301,000	$8,616,000
Cost per genome on FPGA	$0.85	$0.85	$0.85
Number Genomes to recover NRE	3,834,118	5,060,000	10,136,471

Given these assumptions, the data processing cost per genome is already so cheap compared to the wet lab cost is that it will be hard to justify ASICs until the demand for sequencing per year per site is for tens of millions of genomes.

6.18 Exercises

Because this interactive zyBook version may have been re-ordered and hence sections renumbered, section numbers below labeled with COD refer to the original hardcopy book's section numbers.

EXERCISE 6.18.1: [5] <COD §6.2>.

First, write down a list of your daily activities that you typically do on a weekday. For instance, you might get out of bed, take a shower, get dressed, eat breakfast, dry your hair, brush your teeth. Make sure to break down your list so you have a minimum of 10 activities.

There is no single right answer for this question. The purpose is to get students to think about parallelism present in their daily lives. The answer should have at least 10 activities identified.

(a) Now consider which of these activities is already exploiting some form of parallelism (e.g., brushing multiple teeth at the same time, versus one at a time, carrying one book at a time to school, versus loading them all into your backpack and then carry them "in parallel"). For each of your activities, discuss if they are already working in parallel, but if not, why they are not.

Solution

(b) Next, consider which of the activities could be carried out concurrently (e.g., eating breakfast and listening to the news). For each of your activities, describe which other activity could be paired with this activity.

Solution

(c) For 6.1.2, what could we change about current systems (e.g., showers, clothes, TVs, cars) so that we could perform more tasks in parallel?

Solution

(d) Estimate how much shorter time it would take to carry out these activities if you tried to carry out as many tasks in parallel as possible.

Solution

EXERCISE 6.18.2: [5] <COD §6.2>.

You are trying to bake 3 blueberry pound cakes. Cake ingredients are as follows:

1 cup butter, softened
1 cup sugar
4 large eggs
1 teaspoon vanilla extract
1/2 teaspoon salt
1/4 teaspoon nutmeg
1 1/2 cups flour
1 cup blueberries

The recipe for a single cake is as follows:

Step 1: Preheat oven to 325°F (160°C). Grease and flour your cake pan.
Step 2: In large bowl, beat together with a mixer butter and sugar at medium speed until light and fluffy. Add eggs, vanilla, salt and nutmeg. Beat until thoroughly blended. Reduce mixer speed to low and add flour, 1/2 cup at a time, beating just until blended.
Step 3: Gently fold in blueberries. Spread evenly in prepared baking pan. Bake for 60

minutes.

(a) Your job is to cook 3 cakes as efficiently as possible. Assuming that you only have one oven large enough to hold one cake, one large bowl, one cake pan, and one mixer, come up with a schedule to make three cakes as quickly as possible. Identify the bottlenecks in completing this task.

Solution

(b) Assume now that you have three bowls, 3 cake pans and 3 mixers. How much faster is the process now that you have additional resources?

Solution

(c) Assume now that you have two friends that will help you cook, and that you have a large oven that can accommodate all three cakes. How will this change the schedule you arrived at in Exercise 6.2.1 above?

Solution

(d) Compare the cake-making task to computing 3 iterations of a loop on a parallel computer. Identify data-level parallelism and task-level parallelism in the cake-making loop.

Solution

EXERCISE 6.18.3: [10] <COD §6.2>.

Many computer applications involve searching through a set of data and sorting the data. A number of efficient searching and sorting algorithms have been devised in order to reduce the runtime of these tedious tasks. In this problem we will consider how best to parallelize these tasks.

(a) Consider the following binary search algorithm (a classic divide and conquer algorithm) that searches for a value *X* in a sorted N-element array A and returns the index of matched entry:


```
BinarySearch(A[0..N-1], X) {
   low = 0
   high = N - 1

   while (low <= high) {
      mid = (low + high) / 2
```

```
        if (A[mid] > X)
            high = mid - 1
        else if (A[mid] < X)
            low = mid + 1
        else
            return mid          // found
    }

    return -1                   // not found
}
```


Assume that you have Y cores on a multi-core processor to run BinarySearch. Assuming that Y is much smaller than N, express the speedup factor you might expect to obtain for values of Y and N. Plot these on a graph.

Solution

(b) Next, assume that Y is equal to N. How would this affect your conclusions in your previous answer? If you were tasked with obtaining the best speedup factor possible (i.e., strong scaling), explain how you might change this code to obtain it.

Solution

EXERCISE 6.18.4: [10] <COD §6.2>.

Consider the following piece of C code:

```
for (j = 2; j < 1000; j++)
   D[j] = D[j - 1] + D[j - 2];
```

The MIPS code corresponding to the above fragment is:

```
        li    $s0, 8000
        add   $s1, $a0, $s0
        addi  $s2, $a0, 16
loop:   l.d   $f0, -16($s2)
        l.d   $f2, -8($s2)
        add.d $f4, $f0, $f2
        s.d   $f4, 0($s2)
        addi  $s2, $s2, 8
```

```
bne   $s2, $s1, loop
```

Instructions have the following associated latencies (in cycles):

add.d	l.d	s.d	addiu
4	6	1	2

(a) How many cycles does it take to execute this code?

(b) Reorder the code to reduce stalls. Now, how many cycles does it take to execute this code? (Hint: You can remove additional stalls by changing the offset on the fsd instruction.)

(c) When an instruction in a later iteration of a loop depends upon a data value produced in an earlier iteration of the same loop, we say that there is a *loop carried dependence* between iterations of the loop. Identify the loop-carried dependences in the above code. Identify the dependent program variable and assembly-level registers. You can ignore the loop induction variable `j`.

(d) Rewrite the code by using registers to carry the data between iterations of the loop (as opposed to storing and reloading the data from main memory). Show where this code stalls, and calculate the number of cycles required to execute. Note that for this problem you will need to use the assembler pseudoinstruction "`move.d rd, rs`", which writes the value of floating-point register `rs1` into floating-point register `rd`. Assume that `mov, d` executes in a single cycle.

(e) loop unrolling was described in Chapter 4. Unroll and optimize the loop above so that each unrolled loop handles three iterations of the original loop. Show where this code stalls and calculate the number of cycles required to execute.

(f) The unrolling from Exercise 6.4.5. Works nicely because we happen to want a multiple of three iterations. What happens if the number of iterations is not known at compile time? How can we efficiently handle a number of iterations that is not a multiple of the number of iterations per unrolled loop?

(g) Consider running this code on a two-node distributed memory message passing system. Assume that we are going to use message passing as described in Section 6.7, where we introduce a new operation send `(x , y)` that sends to node x the value y, and an operation `receive( )` that waits for the value being sent to it. Assume that send operations take one cycle to issue (i.e., later instructions on the same node can proceed on the next cycle), but take several cycles to be received on the receiving node. Receive instructions stall execution on the node where they are executed until they receive a message. Can you use such a system to speed up the code for this exercise? If so, what is the maximum latency for receiving information

that can be tolerated? If not, why not?

EXERCISE 6.18.5: [10] <COD §6.2>.

Consider the following recursive mergesort algorithm (another classic divide and conquer algorithm). Mergesort was first described by John Von Neumann in 1945. The basic idea is to divide an unsorted list *x* of *m* elements into two sublists of about half the size of the original list. Repeat this operation on each sublist, and continue until we have lists of size 1 in length. Then starting with sublists of length 1, "merge" the two sublists into a single sorted list.


```
Mergesort(m)
   var list left, right, result

   if length(m) ≤ 1
      return m
   else
      var middle = length(m) / 2

      for each x in m up to middle
         add x to left

      for each x in m after middle
         add x to right

      left = Mergesort(left)
      right = Mergesort(right)
      result = Merge(left, right)

      return result
```

The merge step is carried out by the following code:


```
Merge(left,right)
   var list result

   while length(left) > 0 and length(right) > 0
      if first(left) ≤ first(right)
         append first(left) to result
```

```
        left = rest(left)
     else
        append first(right) to result
        right = rest(right)

  if length(left) > 0
     append rest(left) to result

  if length(right) > 0
     append rest(right) to result

  return result
```


(a) Assume that you have Y cores on a multicore processor to run Mergesort. Assuming that Y is much smaller than length(m), express the speedup factor you might expect to obtain for values of Y and length(m). Plot these on a graph

(b) Next, assume that Y is equal to length (m). How would this affect your conclusions your previous answer? If you were tasked with obtaining the best speedup factor possible (i.e., strong scaling), explain how you might change this code to obtain it.

EXERCISE 6.18.6: [10] <COD §6.5>.

Matrix multiplication plays an important role in a number of applications. Two matrices can only be multiplied if the number of columns of the first matrix is equal to the number of rows in the second.

Let's assume we have an $m \times n$ matrix A and we want to multiply it by an $n \times p$ matrix B. We can express their product as an $m \times p$ matrix denoted by AB (or $A \cdot B$). If we assign $C = AB$, and $C_{i,j}$ denotes the entry in C at position (i, j), then for each element i and j with $1 \le i \le m$ and $1 \le j \le p$ $c_{i,j} = \sum_{k=1}^{n} a_{i,k} \text{ X } b_{k,j}$. Now we want to see if we can parallelize the computation of C. Assume that matrices are laid out in memory sequentially as follows: $a_{1,1}$, $a_{2,1}$, $a_{3,1}$, $a_{4,1}$, …, etc.

(a) Assume that we are going to compute C on both a single core shared memory machine and a 4-core shared-memory machine. Compute the speedup we would expect to obtain on the 4-core machine, ignoring any memory issues.

(b) Repeat Exercise 6.6.1, assuming that updates to C incur a cache miss due to false sharing when consecutive elements are in a row (i.e., index i) are updated.

(c) How would you fix the false sharing issue that can occur?

EXERCISE 6.18.7: [10] <COD §6.5>.

Consider the following portions of two different programs running at the same time on four processors in a *symmetric multicore processor* (SMP). Assume that before this code is run, both x and y are 0.

Core 1: `x = 2;`
Core 2: `y = 2;`
Core 3: `w = x + y + 1;`
Core 4: `z = x + y;`

(a) What are all the possible resulting values of w, x, y, and z? For each possible outcome, explain how we might arrive at those values. You will need to examine all possible interleavings of instructions.

(b) How could you make the execution more deterministic so that only one set of values is possible?

EXERCISE 6.18.8: [10] <COD §6.7>.

The dining philosopher's problem is a classic problem of synchronization and concurrency. The general problem is stated as philosophers sitting at a round table doing one of two things: eating or thinking. When they are eating, they are not thinking, and when they are thinking, they are not eating. There is a bowl of pasta in the center. A fork is placed in between each philosopher. The result is that each philosopher has one fork to her left and one fork to her right. Given the nature of eating pasta, the philosopher needs two forks to eat, and can only use the forks on her immediate left and right. The philosophers do not speak to one another.

(a) Describe the scenario where none of philosophers ever eats (i.e., starvation). What is the sequence of events that happen that lead up to this problem?

(b) Describe how we can solve this problem by introducing the concept of a priority? But can we guarantee that we will treat all the philosophers fairly? Explain.

(c) Now assume we hire a waiter who is in charge of assigning forks to philosophers. Nobody can pick up a fork until the waiter says they can. The waiter has global knowledge of all forks. Further, if we impose the policy that philosophers will always request to pick up their left fork before requesting to pick up their right fork, then we can guarantee to avoid deadlock.

We can implement requests to the waiter as either a queue of requests or as a periodic retry of a request. With a queue, requests are handled in the order they are received. The problem with using the queue is that we may not always be able to service the philosopher whose request is at the head of the queue (due to the unavailability of resources). Describe a scenario with 5 philosophers where a queue is provided, but service is not granted even though there are forks available for another philosopher (whose request is deeper in the queue) to eat.

(d) If we implement requests to the waiter by periodically repeating our request until the resources become available, will this solve the problem described in Exercise 6.8.3? Explain.

EXERCISE 6.18.9: [10] <COD §6.4>.

Consider the following three CPU organizations:

CPU SS: A 2-core superscalar microprocessor that provides out-of-order issue capabilities on 2 function units (FUs). Only a single thread can run on each core at a time.

CPU MT: A fine-grained multithreaded processor that allows instructions from 2 threads to be run concurrently (i.e., there are two functional units), though only instructions from a single thread can be issued on any cycle.

CPU SMT: An SMT processor that allows instructions from 2 threads to be run concurrently (i.e., there are two functional units), and instructions from either or both threads can be issued to run on any cycle.

Assume we have two threads X and Y to run on these CPUs that include the following operations:

Thread X	Thread Y
A1 – takes three cycles to execute	B1 – take two cycles to execute
A2 – no dependences	B2 – conflicts for a functional unit with B1
A3 – conflicts for a functional unit with A1	B3 – depends on the result of B2
A4 – depends on the result of A3	B4 – no dependences and takes two cycles to execute

Assume all instructions take a single cycle to execute unless noted otherwise or they encounter a hazard.

(a) Assume that you have 1 SS CPU. How many cycles will it take to execute these two threads? How many issue slots are wasted due to hazards?

(b) Now assume you have 2 SS CPUs. How many cycles will it take to execute these two threads? How many issue slots are wasted due to hazards?

(c) Assume that you have 1 MT CPU. How many cycles will it take to execute these two threads? How many issue slots are wasted due to hazards?

(d) Assume you have one SMT CPU. How many cycles will it take to execute the two threads? How many issue slots are wasted due to hazards?

EXERCISE 6.18.10: [30] <COD §6.4>.

Virtualization software is being aggressively deployed to reduce the costs of managing today's high performance servers. Companies like VMWare, Microsoft and IBM have all developed a range of virtualization products. The general concept, described in COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy), is that a hypervisor layer can be introduced between the hardware and the operating system to allow multiple operating systems to share the same physical hardware. The hypervisor layer is then responsible for allocating CPU and memory resources, as well as handling services typically handled by the operating system (e.g., I/O).

Virtualization provides an abstract view of the underlying hardware to the hosted operating system and application software. This will require us to rethink how multi-core and multiprocessor systems will be designed in the future to support the sharing of CPUs and memories by a number of operating systems concurrently.

(a) Select two hypervisors on the market today, and compare and contrast how they virtualize and manage the underlying hardware (CPUs and memory).

(b) Discuss what changes may be necessary in future multi-core CPU platforms in order to better match the resource demands placed on these systems. For instance, can multithreading play an effective role in alleviating the competition for computing resources?

EXERCISE 6.18.11: [10] <COD §6.3>.

We would like to execute the loop below as efficiently as possible. We have two different machines, a MIMD machine and a SIMD machine.


```
for (i = 0; i < 2000; i++)
   for (j = 0; j < 3000; j++)
      X_array[i][j] = Y_array[j][i] + 200;
```

(a) For a 4 CPU MIMD machine, show the sequence of MIPS instructions that you would execute on each CPU. What is the speedup for this MIMD machine?

(b) For an 8-wide SIMD machine (i.e., 8 parallel SIMD functional units), write an assembly program in using your own SIMD extensions to MIPS to execute the loop. Compare

the number of instructions executed on the SIMD machine to the MIMD machine.

EXERCISE 6.18.12: [10] <COD §6.3>.

A systolic array is an example of an MISD machine. A systolic array is a pipeline network or "wavefront" of data processing elements. Each of these elements does not need a program counter since execution is triggered by the arrival of data. Clocked systolic arrays compute in "lock-step" with each processor undertaking alternate compute and communication phases.

(a) Consider proposed implementations of a systolic array (you can find these on the Internet or in technical publications). Then attempt to program the loop provided in Exercise 6.11 using this MISD model. Discuss any difficulties you encounter.

(b) Discuss the similarities and differences between an MISD and SIMD machine. Answer this question in terms of data-level parallelism.

EXERCISE 6.18.13: [20] <COD §6.6>.

Assume we want to execute the DAXPY loop (shown below) in MIPS assembly on the NVIDIA 8800 GTX GPU. In this problem, we will assume that all math operations are performed on single-precision floating-point numbers (we will rename the loop SAXPY).

```
        l.d     $f0, a($sp)        : load scalar a
        addiu   $t0, $s0, #512     : upper bound of what to load
loop:   l.d     $f2, 0($s0)        : load x(i)
        mul.d   $f2, $f2, $f0      : a x x(i)
        l.d     $f4, 0($s1)        : load y(i)
        add.d   $f4, $f4, $f2      : a x x(i) + y(i)
        s.d     $f4, 0($s1)        : store into y(i)
        addiu   $s0, $s0, #8       : increment index to x
        addiu   $s1, $s1, #8       : increment index to y
        subu    $t1, $t0, $s0      : compute bound
        bne     $t1, $zero, loop   : check if done
```


Assume that instructions take the following number of cycles to execute.

Loads	Stores	Add.S	Mult.S
5	2	3	4

(a) Describe how you will construct warps for the SAXPY loop to exploit the 8 cores

provided in a single multiprocessor.

EXERCISE 6.18.14: [90] <COD §6.6>.

Download the CUDA Toolkit and SDK from https://developer.nvidia.com/cuda-downloads. Make sure to use the "emurelease" (Emulation Mode) version of the code (you will not need actual NVIDIA hardware for this assignment). Build the example programs provided in the SDK, and confirm that they run on the emulator.

(a) Using the "template" SDK sample as a starting point, write a CUDA program to perform the following vector operations:
   1. *a* - *b* (vector-vector subtraction)
   2. a · b (vector dot product)

   The dot product of two vectors a = [a_1, a_2, ..., a_n] and b = [b_1, b_2, ..., b_n] is defined as:

$$a \cdot b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

   Submit code for each program that demonstrates each operation and verifies the correctness of the results.

(b) If you have GPU hardware available, complete a performance analysis your program, examining the computation time for the GPU and a CPU version of your program for a range of vector sizes. Explain any results you see.

EXERCISE 6.18.15: [25] <COD §6.6>.

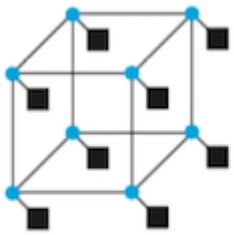
AMD has recently announced integrating a graphics processing unit with their x86 cores into a single package (though with different clocks for each of the cores). This is an example of a heterogeneous multiprocessor system. One of the key design points is to allow for fast data communication between the CPU and the GPU. Before AMD's Fusion architecture, communications were needed between discrete CPU and GPU chips. Presently, the plan is to use multiple (at least 16) PCI express channels to facilitate intercommunication.

This question will require the students to research on the Internet both the AMD Fusion architecture and the Intel QuickPath technology. The key is that students become aware of these technologies. The actual bandwidth and latency values should be available right off the company websites, and will change as the technology evolves.

(a) Compare the bandwidth and latency associated with these two interconnect technologies.

EXERCISE 6.18.16: [10] <COD §6.8>.

Refer to COD Figure 6.14b (Network topologies that have appeared in commercial parallel processors) shown below, which shows an n-cube interconnect topology of order 3 that interconnects 8 nodes. One attractive feature of an n-cube interconnection network topology is its ability to sustain broken links and still provide connectivity.



b. *n*-cube tree of 8 nodes ($8 = 2^3$ so $n = 3$)

(a) Develop an equation that computes how many links in the n-cube (where n is the order of the cube) can fail and we can still guarantee an unbroken link will exist to connect any node in the n-cube.

(b) Compare the resiliency to failure of n-cube to a fully connected interconnection network. Plot a comparison of reliability as a function of the added number of links for the two topologies.

EXERCISE 6.18.17: [60] <COD §6.10>.

Benchmarking is field of study that involves identifying representative workloads to run on specific computing platforms in order to be able to objectively compare performance of one system to another. In this exercise we will compare two classes of benchmarks: the Whetstone CPU benchmark and the PARSEC Benchmark suite. Select one program from PARSEC. All programs should be freely available on the Internet. Consider running multiple copies of Whetstone versus running the PARSEC Benchmark on any of systems described in COD Section 6.11 (Real stuff: Benchmarking Intel Core i7 versus NVIDIA Tesla GPU).

(a) What is inherently different between these two classes of workload when run on these multi-core systems?

(b) In terms of the Roofline Model, how dependent will the results you obtain when running these benchmarks be on the amount of sharing and synchronization present in the workload used?

EXERCISE 6.18.18: [15] <COD §6.10>.

When performing computations on sparse matrices, latency in the memory hierarchy becomes much more of a factor. Sparse matrices lack the spatial locality in the data stream typically found in matrix operations. As a result, new matrix representations have been proposed.

One the earliest sparse matrix representations is the Yale Sparse Matrix Format. It stores an initial sparse $m \times n$ matrix, M in row form using three one-dimensional arrays. Let R be the number of nonzero entries in M. We construct an array A of length R that contains all nonzero entries of M (in left-to-right top-to-bottom order). We also construct a second array IA of length $m + 1$ (i.e., one entry per row, plus one). $IA(i)$ contains the index in A of the first nonzero element of row i. Row i of the original matrix extends from $A(IA(i))$ to $A(IA(i + 1) - 1)$. The third array, JA, contains the column index of each element of A, so it also is of length R.

(a) Consider the sparse matrix X below and write C code that would store this code in Yale Sparse Matrix Format.

```
Row 1 [1, 2, 0, 0, 0, 0]
Row 2 [0, 0, 1, 1, 0, 0]
Row 3 [0, 0, 0, 0, 9, 0]
Row 4 [2, 0, 0, 0, 0, 2]
Row 5 [0, 0, 3, 3, 0, 7]
Row 6 [1, 3, 0, 0, 0, 1]
```

(b) In terms of storage space, assuming that each element in matrix X is single precision floating point, compute the amount of storage used to store the Matrix above in Yale Sparse Matrix Format.

(c) Perform matrix multiplication of Matrix X by Matrix Y shown below.

```
[2, 4, 1, 99, 7, 2]
```


Put this computation in a loop, and time its execution. Make sure to increase the number of times this loop is executed to get good resolution in your timing measurement. Compare the runtime of using a naïve representation of the matrix, and the Yale Sparse Matrix Format.

(d) Can you find a more efficient sparse matrix representation (in terms of space and computational overhead)?

EXERCISE 6.18.19: [10] <COD §6.11>.

In future systems, we expect to see heterogeneous computing platforms constructed out of heterogeneous CPUs. We have begun to see some appear in the embedded processing market in systems that contain both floating point DSPs and a microcontroller CPUs in a multichip module package.

Assume that you have three classes of CPU:
CPU A—A moderate speed multi-core CPU (with a floating point unit) that can execute multiple instructions per cycle.
CPU B—A fast single-core integer CPU (i.e., no floating point unit) that can execute a single instruction per cycle.
CPU C—A slow vector CPU (with floating point capability) that can execute multiple copies of the same instruction per cycle.

Assume that our processors run at the following frequencies:

CPU A	CPU B	CPU C
1 GHz	3 GHz	250 MHz

CPU A can execute 2 instructions per cycle, CPU B can execute 1 instruction per cycle, and CPU C can execute 8 instructions (through the same instruction) per cycle. Assume all operations can complete execution in a single cycle of latency without any hazards.

All three CPUs have the ability to perform integer arithmetic, though CPU B cannot perform floating point arithmetic. CPU A and B have an instruction set similar to a MIPS processor. CPU C can only perform floating point add and subtract operations, as well as memory loads and stores. Assume all CPUs have access to shared memory and that synchronization has zero cost.

The task at hand is to compare two matrices X and Y that each contain 1024 × 1024 floating point elements. The output should be a count of the number indices where the value in X was larger or equal to the value in Y.

(a) Describe how you would partition the problem on the 3 different CPUs to obtain the best performance.

(b) What kind of instruction would you add to the vector CPU C to obtain better performance?

EXERCISE 6.18.20: [10] <COD §6.11>.

Assume a quad-core computer system can process database queries at a steady state rate of requests per second. Also assume that each transaction takes, on average, a fixed amount of time to process. The following table shows pairs of transaction latency and processing rate.

Average Transaction Latency	Maximum transaction processing rate
1 ms	5000/sec
2 ms	5000/sec
1 ms	10,000/sec
2 ms	10,000/sec

For each of the pairs in the table, answer the following questions:

(a) On average, how many requests are being processed at any given instant?

(b) If we move to an eight-core system, ideally, what will happen to the system throughput (i.e., how many queries/second will the computer process)?

(c) Discuss why we rarely obtain this kind of speed-up by simply increasing the number of cores.

©zyBooks 05/16/25 23:52 2475274
Jaheim Attri
FIUEEL4709CSpring2025