The Roofline model offers insight on how to improve the performance of software and hardware.

BY SAMUEL WILLIAMS, ANDREW WATERMAN, AND DAVID PATTERSON

Roofline: An Insightful Visual Performance Model for Multicore Architectures

CONVENTIONAL WISDOM IN computer architecture produced similar designs. Nearly every desktop and server computer uses caches, pipelining, superscalar instruction issue, and out-of-order execution. Although the instruction sets varied, the microprocessors were all from the same school of

design. The relatively recent switch to multicore means that microprocessors will become more diverse, since no conventional wisdom has yet emerged concerning their design. For example, some offer many simple processors vs. fewer complex processors, some depend on multithreading, and some even replace caches with explic-itly addressed local stores. Manufac-turers will likely offer multiple prod-ucts with differing numbers of cores to cover multiple price-performance points, since Moore's Law will permit the doubling of the number of cores per chip every two years.[4] While di-versity may be understandable in this time of uncertainty, it exacerbates the

already difficult jobs of programmers, compiler writers, and even architects. Hence, an easy-to-understand model that offers performance guidelines would be especially valuable.

Such a model need not be perfect, just insightful. The 3Cs (compulsory, capacity, and conflict misses) model for caches is an analogy.[19] It is not perfect, as it ignores potentially important factors like block size, block-allocation policy, and block-replacement policy. It also has quirks; for example, a miss might be labeled "capacity" in one design and "conflict" in another cache of the same size. Yet the 3Cs model has been popular for nearly 20 years precisely because it offers insight into the behavior of programs, helping programmers, compiler writers, and architects improve their respective designs.

Here, we propose one such model we call Roofline, demonstrating it on four diverse multicore computers using four key floating-point kernels.

Performance Models

Stochastic analytical models[4,24] and statistical performance models[7,25] can accurately predict program performance on multiprocessors but rarely provide insight into how to improve the performance of programs, compilers, and computers[1] and can be difficult to use by nonexperts.[25]

An alternative, simpler approach is "bound and bottleneck analysis." Rather than try to predict performance, it provides "valuable insight into the primary factors affecting the performance of computer systems. In particular, the critical influence of the system bottleneck is highlighted and quantified."[20]

The best-known example of a performance bound is surely Amdahl's Law,[3] which says the performance gain of a parallel computer is limited by the serial portion of a parallel program and was recently applied to heterogeneous multicore computers.[4,18]

Roofline Model

For the foreseeable future, off-chip memory bandwidth will often be the constraining resource in system performance.[23] Hence, we want a model that relates processor performance to off-chip memory traffic. Toward this goal, we use the term "operational in-
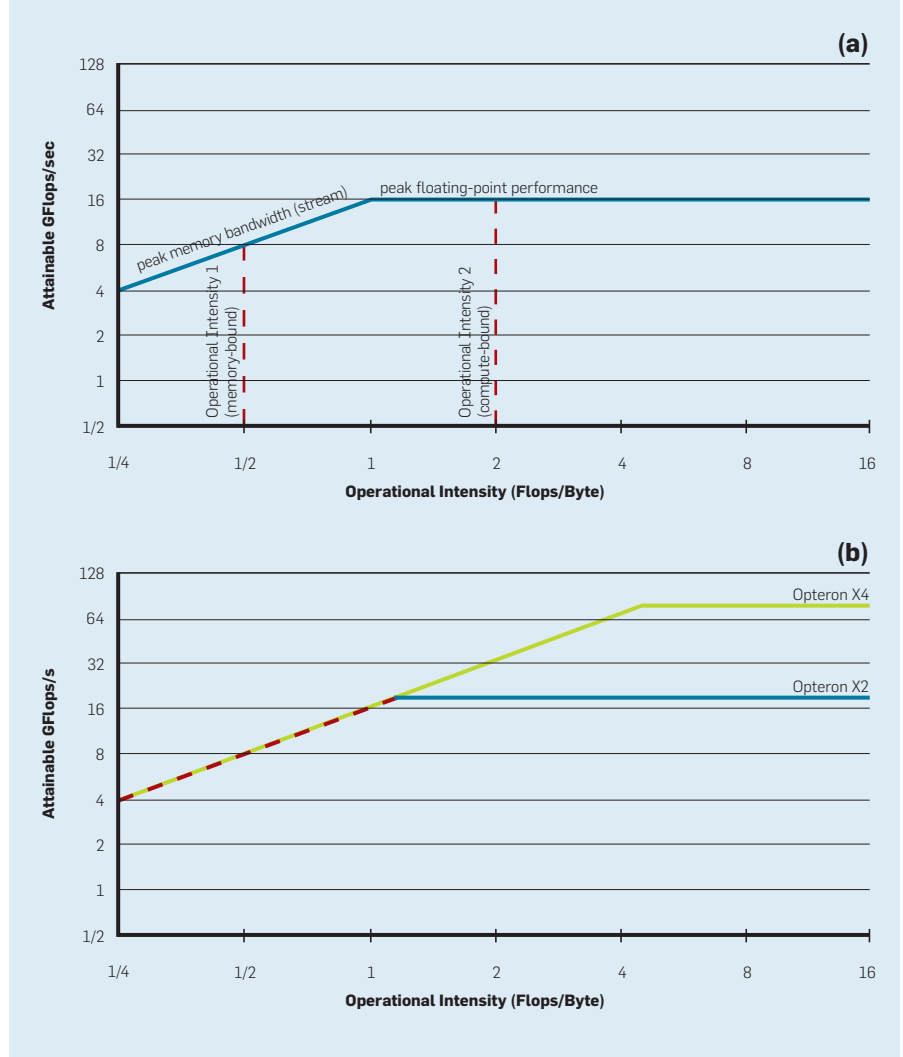
tensity" to mean operations per byte of DRAM traffic, defining total bytes accessed as those bytes that go to the main memory after they have been filtered by the cache hierarchy. That is, we measure traffic between the caches and memory rather than between the processor and the caches. Thus, operational intensity predicts the DRAM bandwidth needed by a kernel on a particular computer.

We say "operational intensity" instead of, say, "arithmetic intensity"[16] or "machine balance"[8,9] for two reasons: First, arithmetic intensity and machine balance measure traffic between the processor and the cache, whereas efficiency-level programmers want to measure traffic between the caches and DRAM. This subtle change allows them to include memory optimizations of a computer into our bound-and-bottleneck model. Second, we

think the model will work with kernels where the operations are not arithmetic, as discussed later, so we needed a more general term than "arithmetic."

The proposed Roofline model ties together floating-point performance, operational intensity, and memory performance in a 2D graph. Peak floating-point performance can be found through hardware specifications or microbenchmarks. The working sets of the kernels we consider here do not fit fully in on-chip caches, so peak memory performance is defined by the memory system behind the caches. Although one can find memory performance through the STREAM benchmark,[22] for this work we wrote a series of progressively optimized microbenchmarks designed to determine sustainable DRAM bandwidth. They include all techniques to get the best memory performance, including

Figure 1: Roofline model for (a) AMD Opteron X2 and (b) Opteron X2 vs. Opteron X4.

prefetching and data alignment. (See Section A.1 in the online Appendix[a] for more detail of how to measure processor and memory performance and operational intensity.)

Figure 1a outlines the model for a 2.2GHz AMD Opteron X2 model 2214 in a dual-socket system. The graph is on a log-log scale. The y-axis is attainable floating-point performance. The x-axis is operational intensity, varying from 0.25 Flops/DRAM byte-accessed to 16 Flops/DRAM byte-accessed. The system being modeled has peak double precision floating-point performance of 17.6 GFlops/sec and peak memory bandwidth of 15GB/sec from our benchmark. This latter measure is the steady-state bandwidth potential of the memory in a computer, not the pin bandwidth of the DRAM chips.

One can plot a horizontal line showing peak floating-point performance of the computer. The actual floating-point performance of a floating-point kernel can be no higher than the horizontal line, since this line is the hardware limit.

How might we plot peak memory performance? Since the x-axis is Flops per Byte and the y-axis is GFlops/sec, gigabytes per second (GB/sec)—or (GFlops/sec)/(Flops/Byte)—is just a line of unit slope in Figure 1. Hence, we can plot a second line that bounds the maximum floating-point performance that the memory system of the computer can support for a given operational intensity. This formula drives the two performance limits in the graph in Figure 1a:

$$\text{Attainable GFlops/sec} = min \begin{cases} \text{Peak Floating-Point Performance} \\ \text{Peak Memory Bandwidth} \times \text{Operational Intensity} \end{cases}$$

The two lines intersect at the point of peak computational performance and peak memory bandwidth. Note that these limits are created once per multicore computer, not once per kernel.

For a given kernel, we can find a point on the x-axis based on its operational intensity. If we draw a vertical line (the pink dashed line in the figures) through that point, the performance of the kernel on that computer

a Please go to doi.acm.org/10.1145/1498765.149 8785#supp

The Roofline sets an upper bound on performance of a kernel depending on the kernel's operational intensity. If we think of operational intensity as a column that hits the roof, either it hits the flat part of the roof, meaning performance is compute-bound, or performance is ultimately memory-bound.

must lie somewhere along that line.

The horizontal and diagonal lines give this bound model its name. The Roofline sets an upper bound on performance of a kernel depending on the kernel's operational intensity. If we think of operational intensity as a column that hits the roof, either it hits the flat part of the roof, meaning performance is compute-bound, or it hits the slanted part of the roof, meaning performance is ultimately memory-bound. In Figure 1a, a kernel with operational intensity 2.0 Flops/Byte is compute-bound and a kernel with operational intensity 1.0 Flops/Byte is memory-bound. Given a Roofline, you can use it repeatedly on different kernels, since the Roofline doesn't vary.

Note that the ridge point (where the diagonal and horizontal roofs meet) offers insight into the computer's overall performance. The x-coordinate of the ridge point is the minimum operational intensity required to achieve maximum performance. If the ridge point is far to the right, then only kernels with very high operational intensity can achieve the maximum performance of that computer. If it is far to the left, then almost any kernel can potentially hit maximum performance. As we explain later, the ridge point suggests the level of difficulty for programmers and compiler writers to achieve peak performance.

To illustrate, we compare the Opteron X2 with two cores in Figure 1a to its successor, the Opteron X4 with four cores. To simplify board design, they share the same socket. Hence, they have the same DRAM channels and can thus have the same peak memory bandwidth, although prefetching is better in the X4. In addition to doubling the number of cores, the X4 also has twice the peak floating-point performance per core; X4 cores can issue two floating-point SSE2 instructions per clock cycle, whereas X2 cores can issue two instructions every other clock. As the clock rate is slightly faster—2.2GHz for X2 vs. 2.3GHz for X4—the X4 is able to achieve slightly more than four times the peak floating-point performance of the X2 with the same memory bandwidth.

Figure 1b compares the Roofline models for these two systems. As expected, the ridge point shifts right

from 1.0 Flops/Byte in the Opteron X2 to 4.4 in the Opteron X4. Hence, to realize a performance gain using the X4, kernels need an operational intensity greater than 1.0 Flops/Byte.

Adding Ceilings to the Model

The Roofline model provides an upper bound to performance. Suppose a program performs far below its Roofline. What optimizations should one implement and in what order? Another advantage of bound-and-bottleneck analysis is that "a number of alternatives can be treated together, with a single bounding analysis providing useful information about them all."[20]
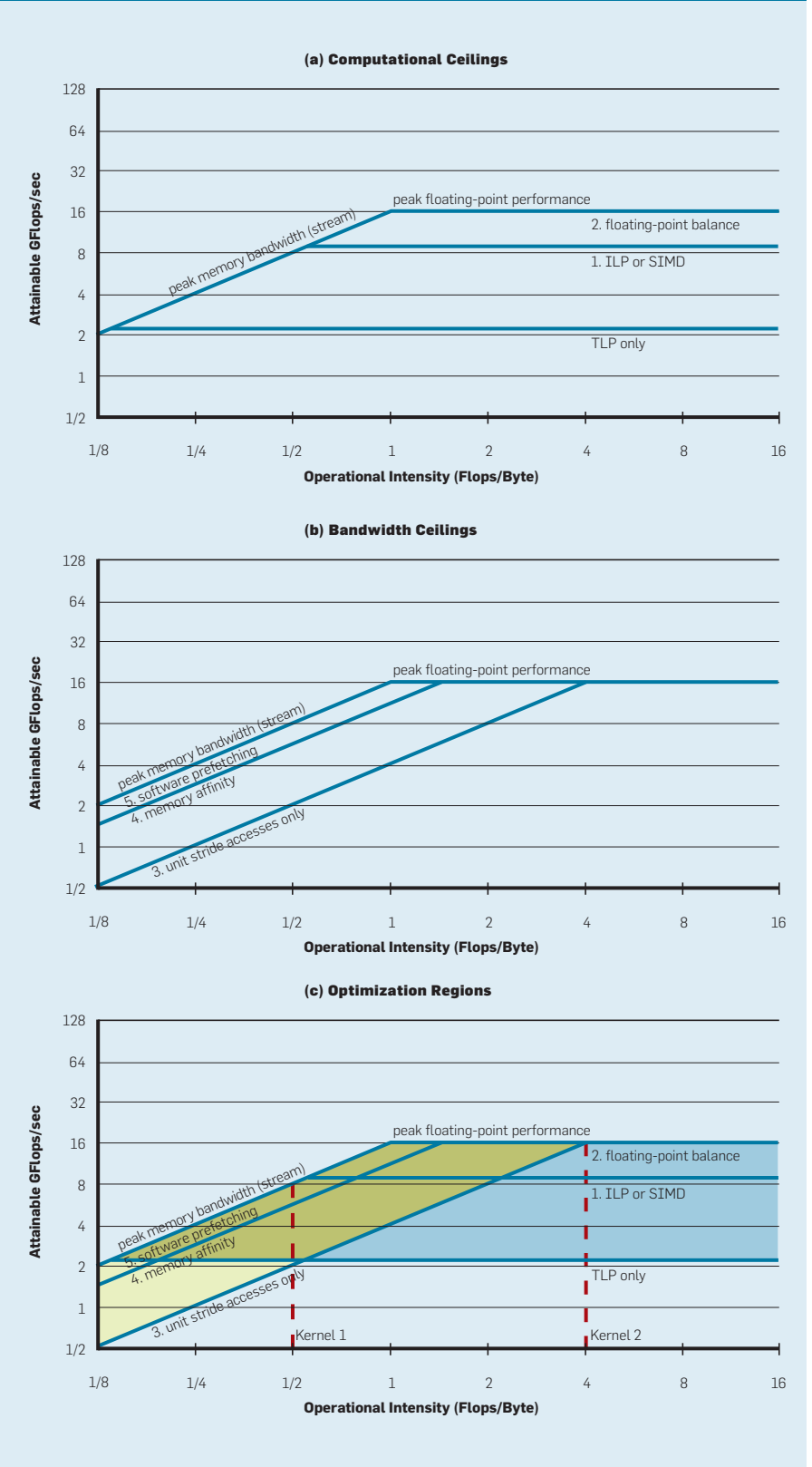
We leverage this insight to add multiple ceilings to the Roofline model to guide which optimizations to implement. It is similar to the guidelines loop balance gives the compiler. We can think of each optimization as a "performance ceiling" below the appropriate Roofline, meaning you cannot break through a ceiling without first performing the associated optimization.

For example, to reduce computational bottlenecks on the Opteron X2, almost any kernel can be helped with two optimizations:

Improve instruction-level parallelism (ILP) and apply SIMD. For superscalar architectures, the highest performance comes when fetching, executing, and committing the maximum number of instructions per clock cycle. The goal is to improve the code from the compiler to increase ILP. The highest performance comes from completely covering the functional unit latency. One way to hide instruction latency is by unrolling loops. For x86-based architectures, another way is using floating-point SIMD instructions whenever possible, since a SIMD instruction operates on pairs of adjacent operands; and

Balance floating-point operation mix. The best performance requires that a significant fraction of the instruction mix be floating-point operations (discussed later). Peak floating-point performance typically also requires an equal number of simultaneous floating-point additions and multiplications, since many computers have multiply-add instructions or an equal number of adders and multipliers.

Figure 2: Roofline model with ceilings for Opteron X2.

(a) Computational Ceilings

(b) Bandwidth Ceilings

(c) Optimization Regions

Memory bottlenecks can be reduced with the help of three optimizations:

Restructure loops for unit stride accesses. Optimizing for unit-stride memory accesses engages hardware prefetching, significantly increasing memory bandwidth;

Ensure memory affinity. Most microprocessors today include a memory controller on the same chip with the

Table 1: Characteristics of four recent multicore computers.

MPU Type	Intel Xeon (Clovertown, e5345)	AMD Opteron X4 (Barcelona, 2356)	Sun UltraS-PARC T2+ (Niagara 2, 5120)	IBM Cell (QS20)
ISA	x86/64	x86/64	SPARC	Cell SPEs
Total Threads	8	8	128	16
Total Cores	8	8	16	16
Total Sockets	2	2	2	2
GHz	2.33	2.30	1.17	3.20
Peak GFlops/sec	75	74	19	29
Peak DRAM GB/sec	21.3r, 10.6w	2 × 10.6	2 × 21.3r, 2 × 10.6w	2 × 25.6
Stream GB/sec	5.9	16.6	26.0	47.0
DRAM Type	FBDIMM	DDR2	FBDIMM	XDR

processors. If the system has two multicore chips, then some addresses go to the DRAM local to one multicore chip, and the rest go over a chip interconnect to access the DRAM local to another chip. The latter lowers performance. This optimization allocates data and the threads tasked to that data to the same memory-processor pair, so the processors rarely have to access the memory attached to other chips; and

Use software prefetching. The highest performance usually requires keeping many memory operations in flight, which is easier to do via prefetching than by waiting until the data is actually requested by the program. On some computers, software prefetching delivers more bandwidth than hardware prefetching alone.

Like the computational Roofline, computational ceilings can come from an optimization manual,[2] though it's easy to imagine collecting the necessary parameters from simple microbenchmarks. The memory ceilings require running experiments on each computer to determine the gap be-

tween them (see online Appendix A.1). The good news is that like the Roofline, the ceilings must be measured only once per multicore computer.

Figure 2 adds ceilings to the Roofline model in Figure 1a; Figure 2a shows the computational ceilings and Figure 2b the memory bandwidth ceilings. Although the higher ceilings are not labeled with lower optimizations, these lower optimizations are implied; to break through a ceiling, the programmer must have already broken through all the ones below. Figure 2a shows the computational "ceilings" of 8.8 GFlops/sec if the floating-point operation mix is imbalanced and 2.2 GFlops/sec if the optimizations to increase ILP or SIMD are also missing. Figure 2b shows the memory bandwidth ceilings of 11 GB/sec without software prefetching, 4.8 GB/sec without memory affinity optimizations, and 2.7 GB/sec with only unit stride optimizations.

Figure 2c combines Figures 2a and 2b into a single graph. The operational intensity of a kernel determines the optimization region, and thus which

optimizations to try. The middle of Figure 2c shows that computational optimizations and memory bandwidth optimizations overlap; we picked the colors to highlight this overlap. For example, Kernel 2 falls in the blue trapezoid on the right, suggesting the programmer should work only on the computational optimizations. If a kernel fell in the yellow triangle on the lower left, the model would suggest trying just memory optimizations. Kernel 1 falls in the green (= yellow + blue) parallelogram in the middle, suggesting the programmer try both types of optimization. Note that the Kernel 1 vertical line falls below the floating-point imbalance optimization, so optimization 2 may be skipped.

The ceilings of the Roofline model suggest which optimizations the programmer should perform. The height of the gap between a ceiling and the next higher ceiling is the potential reward for trying this optimization. Thus, Figure 2 suggests that optimization 1, which improves ILP/SIMD, has a large potential benefit for optimizing computation on that computer, and optimization 4, which improves memory affinity, has a large potential benefit for improving memory bandwidth on that computer.

The order of the ceilings suggests the optimization order, so we rank the ceilings from bottom to top; those most likely to be realized by a compiler or with little effort by a programmer are at the bottom and those that are difficult for a programmer to implement or inherently lacking in a kernel are at the top. The one quirky ceiling is floating-point balance, since the actual mix depends on the kernel. For most kernels, achieving parity between multiplies and additions is difficult, but for a few kernels, parity is natural. One example is sparse matrix-vector multiplication; for this domain, we would place floating-point mix as the lowest ceiling, since it is inherent. Like the 3Cs model, as long as the Roofline model delivers on insight, it need not be perfect.

Tying the 3Cs to Operational Intensity
Operational intensity tells programmers which ceilings need the most attention. Thus far, we have assumed

that the operational intensity is fixed, though this is not always the case; for example, for some kernels, the operational intensity increases with problem size (such as for Dense Matrix and FFT problems).

Caches filter the number of accesses that go to memory, so optimizations that improve cache performance increase operational intensity. Thus, we may couple the 3Cs model to the Roofline model. Compulsory misses set the minimum memory traffic and hence the highest possible operational intensity. Memory traffic from conflict and capacity misses can considerably lower the operational intensity of a kernel, so we should try to eliminate such misses.

For example, we can reduce traffic from conflict misses by padding arrays to change cache line addressing. A second example is that some computers have a non-allocating store instruction, so stores go directly to memory and do not affect caches. This approach prevents loading a cache block with data to be overwritten, thereby reducing memory traffic. It also prevents displacing useful items in the cache with data that will not be read, thereby saving conflict misses.

This shift of operational intensity to the right could put a kernel in a different optimization region. Generally, we advise improving operational intensity of the kernel before implementing other optimizations.

Demonstrating the Model

To demonstrate the Roofline model's utility, we now construct Roofline models for four recent multicore computers and then optimize four floating-point kernels. We'll then show that the ceilings and rooflines bound the observed performance for all computers and kernels.

Four diverse multicore computers. Given the lack of conventional wisdom concerning multicore architecture, it's not surprising that there are as many different designs as there are chips. Table 1 lists the key characteristics of the four multicore computers, all dual-socket systems, that we discuss here.

The Intel Xeon uses relatively sophisticated processors, capable of executing two SIMD instructions per clock cycle that can each perform two

double-precision floating-point operations. It is the only one of the four machines with a front-side bus connecting to a common north bridge chip and memory controller. The other three have the memory controller on chip.

The Opteron X4 also uses sophisticated cores with high peak floating-point performance but is the only computer of the four with on-chip L3 caches. The two sockets communicate over separate, dedicated hypertransport links, making it possible to build a "glueless" multi-chip system.

The Sun UltraSPARC T2+ uses relatively simple processors at a modest clock rate compared to the other three, allowing it to have twice as many cores per chip. It is also highly multithreaded, with eight hardware-supported threads per core. It has the highest memory bandwidth of the four, as each chip has two dual-channel memory controllers that can drive four sets of DDR2/FBDIMMs.

The clock rate of the IBM Cell QS20 is the highest of the four multicores at 3.2GHz. It is also the most unusual of the four, with a heterogeneous design, a relatively simple PowerPC core, and eight synergistic processing elements (SPEs) with their own unique SIMD-style instruction set. Each SPE also has its own local memory, instead of a cache. An SPE must transfer data from main
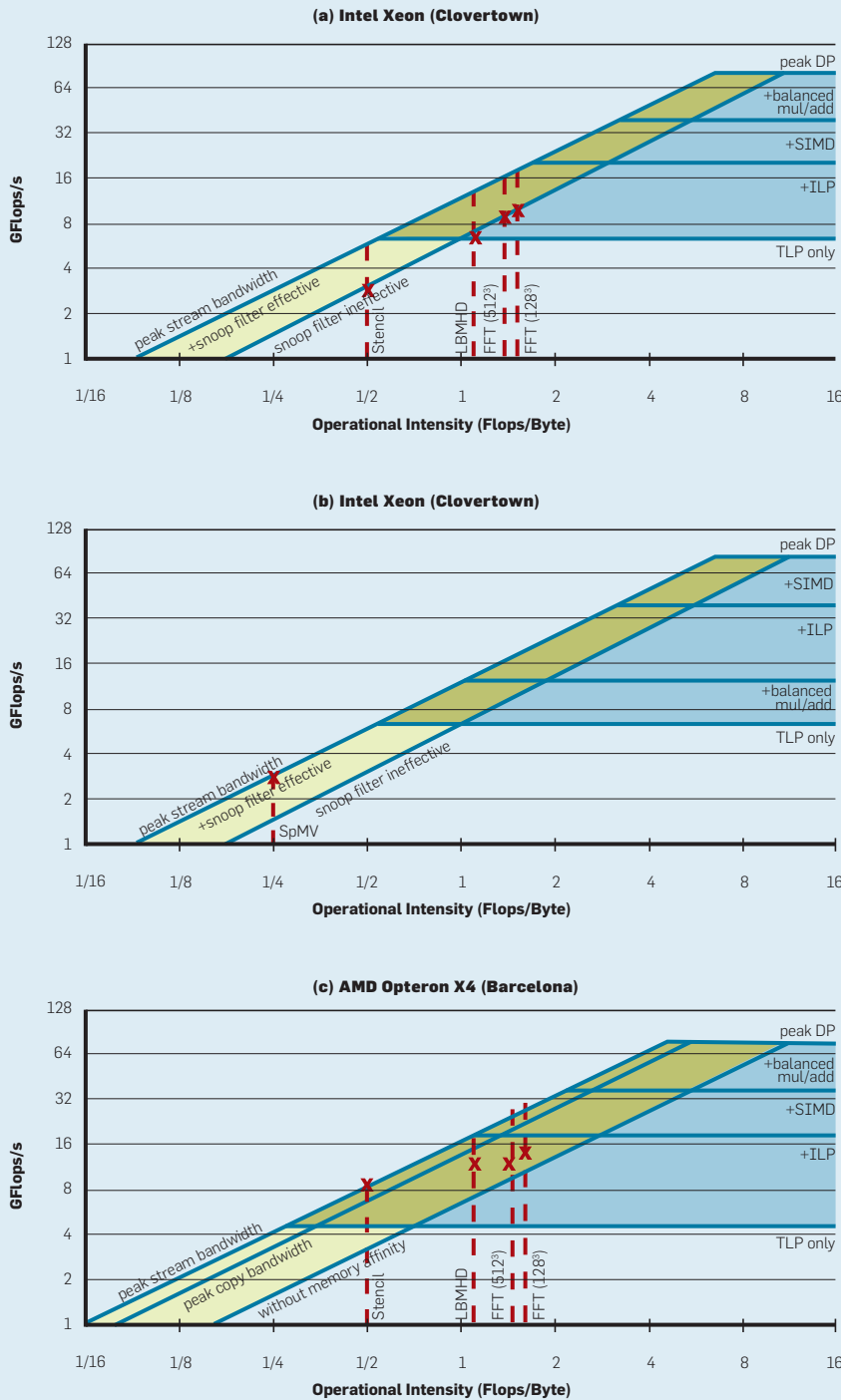
memory into the local memory to operate on it and then back to main memory when the computation is completed. It uses Direct Memory Access, which has some similarity to software prefetching. The lack of caches means porting programs to Cell is more challenging.

Four diverse floating-point kernels. Rather than pick programs from a standard parallel benchmark suite (such as Parsec[5] and Splash-2[30]), we were inspired by the work of Phil Colella,[11] an expert in scientific computing at Lawrence Berkeley National Laboratory, who identified seven numerical methods he believes will be important for computational science and engineering for at least the next decade. Because he identified seven, they are called the Seven Dwarfs and are specified at a high level of abstraction to allow reasoning about their behavior across a broad range of implementations. The widely read "Berkeley View" report[4] found that if the data types were changed from floating point to integer, the same Seven Dwarfs would also be found in many other programs. Note that the claim is not that the Dwarfs are easy to parallelize but that they will be important to computing in most current and future applications; designers are thus advised to make sure they run well on the systems they create, whether or

Table 2: Characteristics of four floating-point kernels.

Name	Operational Intensity	Description
SpMV[29]	0.17 to 0.25	Sparse Matrix-Vector multiply: y = A*x where A is a sparse matrix and x, y are dense vectors; multiplies and adds equal.
LBMHD[28]	0.70 to 1.07	Lattice-Boltzmann Magnetohydro-dynamics is a structured grid code with a series of time steps.
Stencil[12]	0.33 to 0.50	A multigrid kernel that updates seven nearby points in a 3D stencil for a 256^3 problem.
3D FFT	1.09 to 1.64	3D Fast Fourier Transform (2 sizes: 128^3 and 512^3).

Figure 3a–3c: Roofline model for Intel Xeon, AMD Opteron X4, and IBM Cell.

(a) Intel Xeon (Clovertown)

(b) Intel Xeon (Clovertown)

(c) AMD Opteron X4 (Barcelona)

tion are from three sources:[12, 28, 29]

For these kernels, there is sufficient parallelism to utilize all the cores and threads and keep them load balanced; see online Appendix A.2 for how to handle cases when load is not balanced.

Roofline models and results. Figure 3 shows the Roofline models for Xeon, X4, and Cell. The pink vertical dashed lines indicate the operational intensity and the red X marks performance achieved for that particular kernel. However, achieving balance is difficult for the others. Hence, each computer in Figure 3 has two graphs: the left one has multiply-add balance as the top ceiling and is used for Lattice-Boltzmann Magnetohydrodynamics (LBMHD), Stencil, and 3D FFT; the right one has multiply-add as the bottom ceiling and is used for SpMV. Since the T2+ lacks a fused multiply-add instruction nor can it simultaneously issue multiplies and adds, Figure 4 shows a single roofline for the four kernels on the T2+ without the multiply-add balance ceiling.

The Intel Xeon has the highest peak double-precision performance of the four multicores. However, the Roofline model in Figure 3a shows this level of performance can be achieved only with operational intensities of at least 6.7 Flops/Byte; in other words Clovertown requires 55 floating-point operations for every double-precision operand (8B) going to DRAM to achieve peak performance. This high ratio is due in part to the limitation of the front-side bus, which also carries the coherency traffic that can consume up to half the bus bandwidth. Intel includes a snoop filter to prevent unnecessary coherency traffic on the bus. If the working set is small enough for the hardware to filter, the snoop filter nearly doubles the delivered memory bandwidth.

The Opteron X4 has a memory controller on chip, its own path to 667MHz DDR2 DRAM, and separate paths for coherency. Figure 3 shows that the ridge point in the Roofline model is to the left of the Xeon, at an operational intensity of 4.4 Flops/Byte. The Sun T2+ has the highest memory bandwidth so the ridge point is an exceptionally low operational intensity of just 0.33 Flops/Byte. It keeps multiple memory transfers in flight by using many threads. The IBM Cell ridge

not those systems are parallel.

One advantage of using these higher-level descriptions of programs is that we are not tied to code that might have been originally written to optimize an old computer to evaluate future systems. Another advantage of the restricted number is that efficiency-level programmers can create autotuners

for each kernel that would search the alternatives to produce the best code for that multicore computer, including extensive cache optimizations.[13]

Table 2 lists the four kernels from among the Seven Dwarfs we use to demonstrate the Roofline model on the four multicore computers listed in Table 1; the autotuners discussed in this sec-

point of operational intensity is 0.65 Flops/Byte.

Here, we demonstrate the Roofline model on four diverse mutlicore architectures running four kernels representative of some of the Seven Dwarfs:
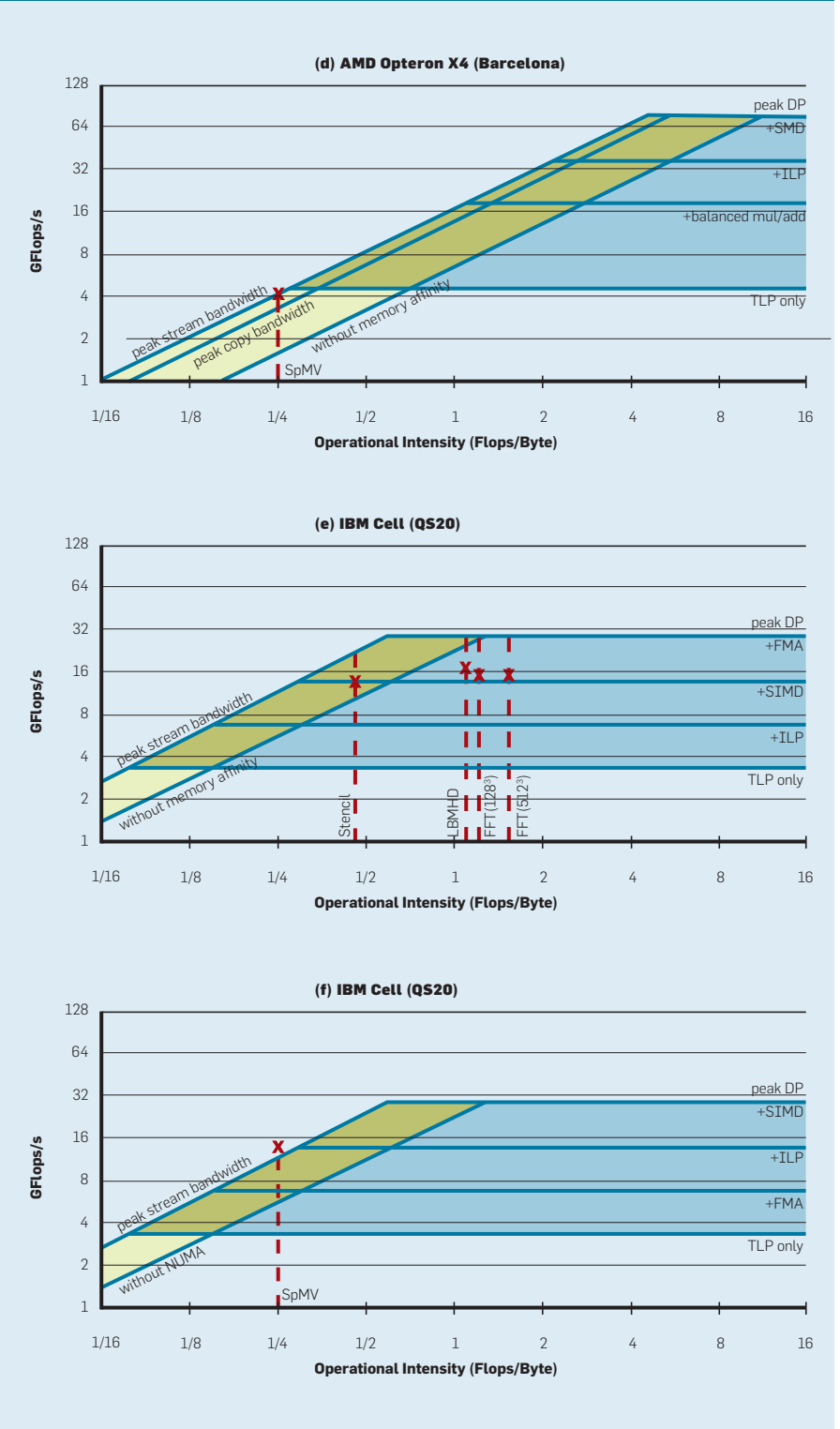
Sparse matrix-vector multiplication. The first example kernel of the sparse matrix computational dwarf is Sparse Matrix-Vector multiply (SpMV); the computation is $y = A*x$, where A is a sparse matrix and x and y are dense vectors. SpMV is popular in scientific computing, economic modeling, and information retrieval. Alas, conventional implementations often run at less than 10% of peak floating-point performance in uniprocessors. One reason is the irregular accesses to memory, which might be expected from sparse matrices. The operational intensity varies from 0.17 Flops/Byte before a register blocking optimization to 0.25 Flops/Byte afterward[26] (see online Appendix A.1).

Given that the operational intensity of SpMV was below the ridge point of all four multicores in Figure 3, most optimizations involve the memory system. Table 3 summarizes the optimizations used by SpMV and the rest of the kernels. Many are associated with the ceilings in Figure 3, and the height of the ceilings suggests the potential benefit of these optimizations.

Lattice-Boltzmann Magnetohydrodynamics. Like SpMV, LBMHD tends to achieve a small fraction of peak performance on uniprocessors due to the complexity of the data structures and the irregularity of memory access patterns. The Flop-to-Byte ratio is 0.70 vs. 0.25 or less in SpMV. By using the no-allocate store optimization, a programmer can improve the operational intensity of LBMHD to 1.07 Flops/Byte. Both x86 multicores offer this cache optimization, but Cell does not have this problem since it uses DMA. Hence, T2+ is the only one of the four computers with the lower intensity of 0.70 Flops/Byte.

Figures 3 and 4 show that the operational intensity of LBMHD is high enough that both computational and memory bandwidth optimizations make sense on all multicores, except the T2+ where the Roofline ridge point is below that of LBMHD. The T2+ reaches its performance ceiling using
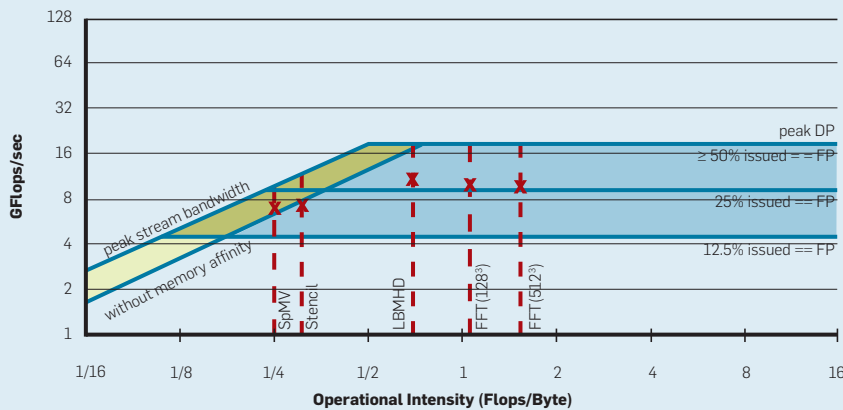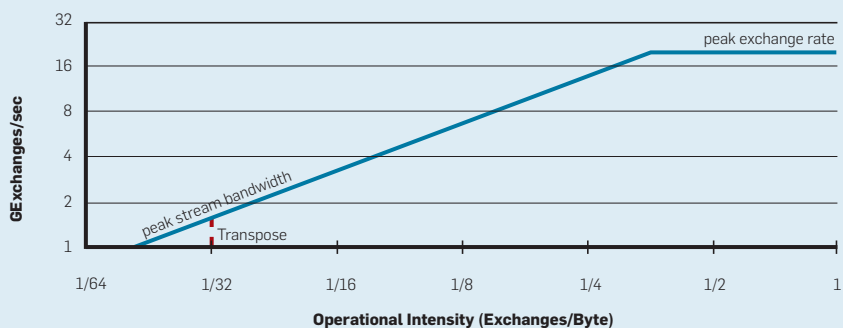

Figure 3d–3f: Roofline model for Intel Xeon, AMD Opteron X4, and IBM Cell.

only the computational optimizations.

Stencil. In general, a stencil on a structured grid is defined as a function that updates a point based on the values of its neighbors. The stencil structure remains constant as it moves from one point in space to the next. For this work, we use the stencil derived from

the explicit heat equation, a partial differential equation on a uniform 2563 3D grid.[12] The stencil's neighbors are the nearest six points along each axis, as well as the center point itself. This stencil performs eight floating-point operations for every 24B of compulsory memory traffic on write-allocate

Figure 4: Roofline model for Sun UltraSPARC T2+.

Figure 5: Roofline for transpose phase of 3D FFT for the Cell.

architectures, yielding an operational intensity of 0.33 Flops/Byte.

3D FFT. This fast Fourier transform is the classic divide-and-conquer algorithm that recursively breaks down a discrete Fourier transform into many smaller ones. The FFT is ubiquitous in many domains, including image processing and data compression. An efficient approach for 3D FFT is to perform 1D transforms along each dimension to maintain unit-stride accesses. We computed the 1D FFTs on Xeon, X4, and T2+ using an autotuned library (FFTW).[15] For Cell, we implemented a radix-2 FFT.

FFT differs from SpMV, LBMHD, and Stencil in that its operational intensity is a function of problem size. For the 128^3- and 512^3-point transforms we examine, the operational intensities are 1.09 and 1.41 Flops/Byte, respectively; Cell's 1GB main memory is too small to hold 512^3 points, so we estimate this result. On Xeon and X4, an entire 128×128 plane fits in cache,

increasing temporal locality and improving the intensity to 1.64 for the 128^3-point transform.

Productivity vs. performance. In addition to performance, productivity (or the programming difficulty of achieving good performance) is another important issue for the parallel computing revolution.[4] One question is whether a low ridge point gives insight into productivity.

The Sun T2+ (with the lowest ridge point of the four computers) was the easiest to program due to its large memory bandwidth and easy-to-understand cores. The advice for these kernels on T2+ is simply to try to get good-performing code from the compiler, then use as many threads as possible. The downside is that the L2 cache is only 16-way set associative, which can lead to conflict misses when 64 threads access the cache, as it did for the Stencil kernel.

In contrast, the computer with the highest ridge point had the lowest

unoptimized performance. The Intel Xeon was difficult to program because it was difficult to understand the memory behavior of the dual front-side buses, how hardware prefetching worked, and the difficulty of getting good SIMD code from the compiler. The C code for both it and the Opteron X4 are liberally sprinkled with intrinsic statements involving SIMD instructions to get good performance.

With a ridge point close to the Xeon, the Opteron X4 required about as much effort, since it benefited from the most types of optimization. However, its memory behavior was easier to understand than the memory behavior of the Xeon.

The IBM Cell (with a ridge point almost as low as the Sun T2+) involved two types of challenges. First, it was difficult for the compiler to exploit the SIMD instructions of Cell's SPE, so at times we needed to help the compiler by inserting intrinsic statements with assembly language instructions into the C code. This comment reflects the immaturity of the IBM compiler, as well as the difficulty of compiling for these SIMD instructions. Second, the memory system is more challenging. Since each SPE has local memory in a separate address space, we could not simply port the code and start running on the SPE. We needed to change the program to issue DMA commands to transfer data back and forth between local store and memory. The good news is that DMA played the role of software prefetch in caches. DMA for a local store is easier to program, achieve good memory performance, and overlap with computation than scheduling prefetches for caches.

To demonstrate the utility of the Roofline Model, Table 4 lists the upper and lower bounding ceilings and the GFlops/sec and GB/sec per kernel-computer pair; recall that operational intensity is the ratio between the two rates. The ceilings listed are the ceilings sandwiching actual performance. All 16 combinations of kernel and computer validate this bound-and-bottleneck model since Roofline's upper and lower ceilings bound performance and the kernels were optimized, as the lower ceilings suggest. The metric that limits performance is in bold; 15 of 16 ceilings are memory-bound for Xeon

Table 3: Kernel optimizations.[12, 28, 29]

Memory affinity. Reduce accesses to DRAM memory attached to the other socket.

Long unit-stride accesses. Change loop structures to generate long unit-stride accesses to engage the prefetchers; also reduces TLB misses.

Software prefetching. Software and hardware prefetching both used to get the most from memory systems.

Reduce conflict misses. Pad arrays to improve cache-hit rates.

Unroll and reorder loops. To expose sufficient parallelism and improve cache utilization, unroll and reorder loops to group statements with similar addresses; improves code quality, reduces register pressure, facilitates SIMD.

"SIMD-ize" code. The x86 compilers didn't generate good SSE code, so we made a code generator to produce SSE intrinsics.

Compress data structures (SpMV only). Since bandwidth limits performance, we used smaller data structures: 16b vs. 32b index and smaller representations of non-zero subblocks.[27]

and X4, while the bottleneck is almost evenly split for T2+ and Cell. For FFT, the surrounding ceilings are memory-bound for Xeon and X4, but compute-bound for T2+ and Cell.

Fallacies About Roofline
We have presented this material in several venues, prompting a number of misconceptions we address here:

Fallacy: The model does not account for all features of modern processors (such as caches and prefetching). The definition of operational intensity we use here does indeed factor-in caches; memory accesses are measured between the caches and memory, not between the processor and caches. In our discussion of performance models, we showed that the memory bandwidth measures of the computer include prefetching and any other optimization (such as blocking) that can im-

prove memory performance. Similarly, some of the optimizations in Table 3 explicitly involve memory. Moreover, in our discussion on tying the 3Cs to operational intensity, we demonstrated the optimizations' effect on increasing operational intensity by reducing capacity and conflict misses.

Fallacy: Doubling cache size increases operational intensity. Autotuning three of the four kernels gets very close to the compulsory memory traffic; the resultant working set is sometimes only a small fraction of the cache. Increasing cache size helps only with capacity misses and possibly conflict misses, so a larger cache has no effect on the operational intensity for the three kernels. However, for 128^3 3D FFT, a larger cache could capture a whole plane of a 3D cube, improving operational intensity by reducing capacity and conflict misses.

Fallacy: The model doesn't account for the long memory latency. The ceilings for no software prefetching in Figures 3 and 4 are at lower memory bandwidth precisely because they cannot hide the long memory latency.

Fallacy: The model ignores integer units in floating-point programs, possibly limiting performance. For the example kernels we've outlined here, the amount of integer code and integer performance can affect performance. For example, the Sun UltraSPARC T2+ fetches two instructions per core per clock cycle and doesn't implement the SIMD instructions of the x86 that can operate on two double-precision floating-point operands at a time. Relative to other processors, the T2+ expends a larger fraction of its instruction issue bandwidth on integer instructions and executes them at a lower rate, hurting overall performance.

Fallacy: The model has nothing to do with multicore. Little's Law[17, 20, 21] dictates that considerable concurrency is necessary to really push the limits of the memory system. This concurrency is more easily satisfied in a multicore than in a uniprocessor. While the bandwidth orientation of the Roofline model certainly works for uniprocessors, it is even more helpful for multicores.

Fallacy: You need to recalculate the Roofline model for every kernel. The Roofline needs to be calculated for given performance metrics and comput-

ers just once; it then guides the implementation for any program for which that metric is the critical performance metric. The kernels we've explored here use floating-point operations and main memory traffic. The ceilings are measured once but can be reordered depending on whether or not multiplies and adds are naturally balanced in the kernel (see the earlier discussion on adding ceilings to the model).

Note that the heights of the ceilings we discuss here document the maximum potential gain of a code performing this optimization. An interesting future direction is to use performance counters to adjust the height of the ceilings and the order of the ceilings for a particular kernel to show the actual benefits of each optimization and the recommended order to try them (see online Appendix A.3).

Fallacy: The model is limited to easily optimized kernels that never hit in the cache. These kernels do indeed hit in the cache; for example, the cache-hit rates of our three multicores with on-chip caches are at least 94% for Stencil and 98% for FFT. Moreover, if the Seven Dwarfs were easy to optimize, it would bode well for the future of multicores. However, our experience is that it is not easy to create the fastest version of these numerical methods on the divergent multicore architectures discussed here. Indeed, three of the results were judged significant enough to be accepted for publication at major conferences.[12, 28, 29]

Fallacy: The model is limited to floating-point programs. Our focus here has been on floating-point programs, so the two axes of the model are floating-point operations per second and the floating-point operational intensity of accesses to main memory. However, the Roofline model can work for other kernels where performance is a function of different performance metrics. A concrete example is the transpose phase of 3D FFT, which performs no floating-point operations at all. Figure 5 shows a Roofline model for just this phase on Cell, with exchanges replacing Flops in the model. One exchange involves reading and writing 16B, so its operational intensity is 1/32 pairwise Exchanges/Byte. Despite the computational metric being memory exchanges, there is still a computational

	Kernel	Upper Ceiling			Achieved Performance			Lower Ceiling		
		Type	Name	Value	Compute	Memory	O.I.	Type	Name	Value
Intel Xeon	SpMV	Memory	Stream BW	11.2GB/sec	2.8GFlops/sec	**11.1GB/sec**	0.25	Memory	Snoop filter	5.9GByte/sec
	LBMHD	Memory	Snoop filter	5.9GB/sec	5.6GFlops/sec	**5.3GB/sec**	1.07	Memory	(none)	0.0GByte/sec
	Stencil	Memory	Snoop filter	5.9GB/sec	2.5GFlops/sec	**5.1GB/sec**	0.50	Memory	(none)	0.0GByte/sec
	3D FFT	Memory	Snoop filter	5.9GB/sec	9.7GFlops/sec	**5.9GB/sec**	1.64	Compute	TLP only	6.2GFlops/sec
AMD X4	SpMV	Memory	Stream BW	17.6GB/sec	4.2GFlops/sec	**16.8GB/sec**	0.25	Memory	Copy BW	13.9GByte/sec
	LBMHD	Memory	Copy BW	13.9GB/sec	11.4GFlops/sec	**10.7GB/sec**	1.07	Memory	No Affinity	7.0GByte/sec
	Stencil	Memory	Stream BW	17.6GB/sec	8.0GFlops/sec	**16.0GB/sec**	0.50	Memory	Copy BW	13.9GByte/sec
	3D FFT	Memory	Copy BW	13.9GB/sec	14.0GFlops/sec	**8.6GB/sec**	1.64	Memory	No Affinity	7.0GByte/sec
Sun T2+	SpMV	Memory	Stream BW	36.7GB/sec	7.3GFlops/sec	**29.1GB/sec**	0.25	Memory	No Affinity	19.8GByte/sec
	LBMHD	Memory	No Affinity	19.8GB/sec	10.5GFlops/sec	**15.0GB/sec**	0.70	Compute	25% issued FP	9.3GFlops/sec
	Stencil	Compute	25% issued FP	9.3GFlopss/sec	**6.8GFlops/sec**	20.3GB/sec	0.33	Memory	No Affinity	19.8GByte/sec
	3D FFT	Compute	Peak DP	19.8GFlops/sec	**9.2GFlops/sec**	10.0GB/sec	1.09	Compute	25% issued FP	9.3GFlops/sec
IBM Cell	SpMV	Memory	Stream BW	47.6GB/sec	11.8GFlops/sec	**47.1GB/sec**	0.25	Memory	FMA	7.3GFlops/sec
	LBMHD	Memory	No Affinity	23.8GB/sec	16.7GFlops/sec	**15.6GB/sec**	1.07	Memory	Without FMA	14.6GFlops/sec
	Stencil	Compute	Without FMA	14.6GFlopss/sec	**14.2GFlops/sec**	30.2GB/sec	0.47	Memory	No Affinity	23.8GByte/sec
	3D FFT	Compute	Peak DP	29.3GFlops/sec	**15.7GFlops/sec**	14.4GB/sec	1.09	Compute	SIMD	14.6GFlops/sec

Table 4: Achieved performance and nearest Roofline ceilings, with metric limiting performance in bold (3D FFT is 128³).

horizontal Roofline, since local stores and caches could affect the number of exchanges that go to DRAM.

Fallacy: The Roofline model must use DRAM bandwidth. If the working set fits in the L2 cache, the diagonal Roofline could be L2 cache bandwidth instead of DRAM bandwidth, and the operational intensity on the *x*-axis would be based on Flops per L2 cache Byte accessed. The diagonal memory performance line would move up, and the ridge point would surely move to the left. For example, Jike Chong of the University of California, Berkeley, ported two financial partial differential equation (PDE) solvers to four other multicore computers: the Intel Penryn and Larrabee and NVIDIA G80 and GTX280.[10] He used the Roofline model to keep track of all four of their peak arithmetic throughput and L1, L2, and DRAM bandwidths. By analyzing an algorithm's working set and operational intensity, he was able to use the Roofline model to quickly estimate the needs for algorithmic improvement. Specifically, for the option-pricing problem with an implicit PDE solver, the working set is small enough to fit into L1, and the L1 bandwidth is sufficient to support peak arithmetic throughput; the Roofline model thus indicates that no optimization is necessary. For option pricing with an explicit PDE formulation, the working set is too large to fit into cache, and the Roofline model helps indicate the ex-

tent cache blocking is necessary to produce peak arithmetic performance.

Conclusion

The sea change from sequential computing to parallel computing is increasing the diversity of computers that programmers must confront when building correct, efficient, scalable, portable software.[4] Here, we've described a simple, visual computational model we call the Roofline Model to help identify which systems would be a good match for important kernels or conversely to determine how to change kernel code or hardware to run desired kernels well. For floating-point kernels that do not fit completely in caches, we've shown how operational

intensity—the number of floating-point operations per byte transferred from DRAM—is an important parameter for both the kernels and the multi-core computers.

We applied Roofline to four kernels from among the Seven Dwarfs[4,11] to four recent multicore designs: AMD Opteron X4, Intel Xeon, IBM Cell, and Sun T2+. The ridge point—the minimum operational intensity to achieve maximum performance—proved to be a better predictor of performance than clock rate or peak performance. Cell offered the highest attained performance (GFlops/sec) on these kernels, but T2+ was the easiest computer on which to achieve its highest performance. One reason is because the ridge point of the Roofline Model for T2+ was the lowest.

Just as the graphical Roofline Model offers insights into the difficulty of achieving the peak performance of a computer, it also makes obvious when a computer is imbalanced. The operational ridge points for the two x86 computers were 4.4 and 6.7—meaning a 35 to 55 Flops/Byte operand that accesses DRAM—yet the operational intensities for the 16 combinations of kernels and computers in Table 4 ranged from 0.25 to just 1.64, with a median of 0.60 Flops/Byte. Architects should keep the ridge point in mind if they want programs to reach peak performance on their new designs.

We measured the roofline and ceilings using microbenchmarks but could have used performance counters (see online Appendix A.1 and A.3). There may indeed be a synergistic relationship between performance counters and the Roofline Model. The requirements for automatic creation of a Roofline model could guide the designer as to which metrics should be collected when faced with literally hundreds of candidates but only a limited hardware budget.[6]

Roofline offers insights into other types of multicore systems (such as vector processors and graphical processing units); other kernels (such as sort and ray tracing); other computational metrics (such as pair-wise sorts per second and frames per second); and other traffic metrics (such as L3 cache bandwidth and I/O bandwidth). Alas, there are many more opportunities for Roofline-oriented research than we can pursue. We thus invite others to join us in the exploration of the effectiveness of the Roofline Model.

Acknowledgments

References
1. Adve, V. *Analyzing the Behavior and Performance of Parallel Programs.* Ph.D. thesis, University of Wisconsin, 1993; www.cs.wisc.edu/techreports/1993/TR1201.pdf.
2. AMD. *Software Optimization Guide for AMD Family 10h Processors, Publication 40546,* Apr. 2008; www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/40546.pdf.
3. Amdahl, G. Validity of the single processor approach to achieving large-scale computing capabilities. In *Proceedings of the AFIPS Conference,* 1967, 483–485.
4. Asanovic, K., Bodik, R., Catanzaro, B., Gebis, J., Keutzer, K., Patterson, D., Plishker, W., Shalf, J., Williams, S., and Yelick, K. *The Landscape of Parallel Computing Research: A View from Berkeley.* Technical Report UCB/EECS-2006-183. EECS, University of California, Berkeley, Dec. 2006.
5. Bienia, C., Kumar, S., Singh, J., and Li, K. *The PARSEC Benchmark Suite: Characterization and Architectural Implications.* Technical Report TR-81 1-008. Princeton University, Jan. 2008.
6. Bird, S., Waterman, A., Klues, K., Datta, K., Liu, R., Nishtala, R., Williams, S., Asanovi, K., Demmel, J., Patterson, D., and Yelick, K. A case for sensible performance counters. Submitted to the First USENIX Workshop on Hot Topics in Parallelism (Berkeley CA, Mar. 30–31, 2009); www.usenix.org/events/hotpar09/.
7. Boyd, E., Azeem, W., Lee, H., Shih, T., Hung, S., and Davidson, E. A hierarchical approach to modeling and improving the performance of scientific applications on the KSR1. In *Proceedings of the 1994 International Conference on Parallel Processing,* 1994, 188–192.
8. Callahan, D., Cocke, J., and Kennedy, K. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel Distributed Computing 5* (1988), 334–358.
9. Carr, S. and Kennedy, K. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems 16,* 4 (Nov. 1994).
10. Chong, J. Private communication on financial PDE solvers, 2008.
11. Colella, P. *Defining Software Requirements for Scientific Computing.* Presentation, 2004.
12. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., and Yelick, K. Stencil computation optimization and autotuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE SC08 Conference* (Austin, TX, Nov. 15–21). IEEE Press, Piscataway, NJ, 2008, 1-12.
13. Demmel, J., Dongarra, J., Eijkhout, V., Fuentes, E., Petitet, A., Vuduc, R., Whaley, R., and Yelick, K. Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE: Special Issue on Program Generation, Optimization, and Adaptation 93,* 2 (2005).
14. Dubois, M. and Briggs, F.A. Performance of synchronized iterative processes in multiprocessor systems. *IEEE Transactions on Software Engineering SE-8,* 4 (July 1982), 419–431.
15. Frigo, M. and Johnson, S. The design and implementation of FFTW3. *Proceedings of the IEEE: Special Issue on Program Generation, Optimization, and Platform Adaptation 93,* 2 (2005).
16. Harris, M. Mapping computational concepts to GPUs. In *ACM SIGGRAPH Courses,* Chapter 31 (Los Angeles, July 31-Aug. 4). ACM Press, New York, 2005.
17. Hennessy, J. and Patterson, D. *Computer Architecture: A Quantitative Approach, Fourth Edition.* Morgan Kaufmann Publishers, Boston, MA, 2007.
18. Hill, M. and Marty, M. Amdahl's Law in the multicore era. *IEEE Computer* (July 2008), 33–38.
19. Hill, M. and Smith, A. Evaluating associativity in CPU caches. *IEEE Transactions on Computers 38,* 12 (Dec. 1989), 1612–1630.
20. Lazowska, E., Zahorjan, J., Graham, S., and Sevcik, K. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models.* Prentice Hall, Upper Saddle River, NJ, 1984.
21. Little, J.D.C. A proof of the queueing formula L = λ W. *Operations Research 9,* 3 (1961), 383–387.
22. McCalpin, J. *STREAM: Sustainable Memory Bandwidth in High-Performance Computers,* 1995; www.cs.virginia.edu/stream.
23. Patterson, D. Latency lags bandwidth. *Commun. ACM 47,* 10 (Oct. 2004).
24. Thomasian, A. and Bay, P. Analytic queueing network models for parallel processing of task systems. *IEEE Transactions on Computers C-35,* 12 (Dec. 1986), 1045–1054.
25. Tikir, M., Carrington, L., Strohmaier, E., and Snavely, A. A genetic algorithms approach to modeling the performance of memory-bound computations. In *Proceedings of the SC07 Conference* (Reno, NV, Nov. 10–16). ACM Press, New York, 2007.
26. Vuduc, R., Demmel, J., Yelick, K., Kamil, S., Nishtala, R., and Lee, B. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of the ACM/IEEE SC02 Conference* (Baltimore, MD, Nov. 16–22). IEEE Computer Society Press, Los Alamitos, CA, 2002.
27. Williams, S. *Autotuning Performance on Multicore Computers.* Ph.D. Thesis. University of California, Berkeley, Dec. 2008; www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-164.html.
28. Williams, S., Carter, J., Oliker, L., Shalf, J., and Yelick, K. Lattice Boltzmann simulation optimization on leading multicore platforms. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing Symposium* (Miami, FL, Apr. 14–18, 2008), 1–14.
29. Williams, S., Oliker, L., Vuduc, F., Shalf, J., Yelick, K., and Demmel, J. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the ACM/IEEE SC07 Conference* (Reno, NV, Nov. 10–16). ACM Press, New York, 2007.
30. Woo, S., Ohara, M., Torrie, E., Singh, J.-P., and Gupta, A.The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture.* ACM Press, New York, 1995, 24–37.

Samuel Williams (SWWilliams@lbl.gov) is a research scientist at Lawrence Berkeley National Laboratory, Berkeley, CA.

Andrew Waterman (waterman@eecs.berkeley.edu) is a graduate student researcher in the Parallel Computing Laboratory of the University of California, Berkeley.

David Patterson (pattrsn@eecs.berkeley.edu) is Director of the Parallel Computing Laboratory of the University of California, Berkeley, and a past president of ACM.