Title

Performance Modeling: The Convolution Approach

ISBN

9781439815694

Authors

Bailey, David H
Snavely, Allan
Carrington, Laura

Publication Date

2010-11-23

DOI

10.1201/b10509-10

Peer reviewed

Chapter 9

The Roofline Model

Samuel W. Williams

Lawrence Berkeley National Laboratory

The Roofline model is a visually intuitive performance model constructed using bound and bottleneck analysis [362, 367, 368]. It is designed to drive programmers towards an intuitive understanding of performance on modern computer architectures. As such, it not only provides programmers with realistic performance expectations, but also enumerates the potential impediments to performance. Knowledge of these bottlenecks drives programmers to implement particular classes of optimizations. This chapter will focus on architecture-oriented roofline models as opposed to using performance counters to generate a roofline model.

This chapter is organized as follows. Section 9.1 defines the abstract architecture model used by the roofline model. Section 9.2 introduces the basic form of the roofline model, where Sections 9.3–9.6 iteratively refine the model with tighter and tighter performance bounds.

9.1 Introduction

In this section we define the abstract architectural model used for the Roofline model. Understanding of the model is critical in one's ability to apply the Roofline model to widely varying computational kernels. We then introduce the concept of arithmetic intensity to the reader and provide several diverse examples that the reader may find useful in their attempt to estimate arithmetic intensity for their kernels of interest. Finally, we define the requisite key terms in this chapter's glossary.

9.1.1 Abstract Architecture Model

The roofline model presumes a simple architectural model consisting of black boxed computational elements (e.g., CPUs, Cores, or Functional Units) and memory elements (e.g., DRAM, caches, local stores, or register files) interconnected by a network. Whenever one or more processing elements may access a memory element, that memory element is considered shared. In general, there is no restriction on the number or balance of computational and memory elements. As such, a large number of possible topologies exist, allowing the model to be applied to a large number of current and future computer architectures. At any given level of the hierarchy, processing elements may only communicate either with memory elements at that level, or with memory elements at a coarser level. That is, processors, cores, or functional units may only communicate with each other via a shared memory.

Consider Figure 9.1. We show two different dual-processor architectures. Conceptually, any processor can reference any memory location. However, Figure 9.1(a) partitions memory and creates additional arcs. This is done to convey the fact that the bandwidth to a given processor may depend on which

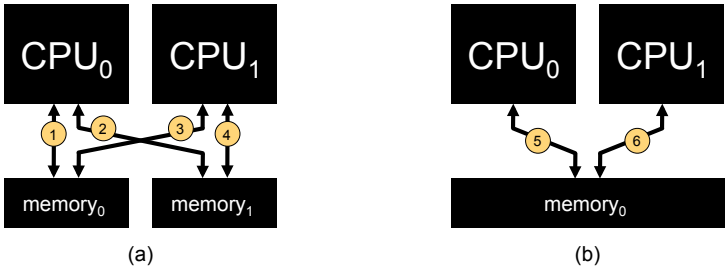

FIGURE 9.1: High-level architectural model showing two black-boxed processors either connected to separate memories (a) or to a common shared memoryi (b). The arrows denote the ISA's ability to access information and not necessarily hardware connectivity.

memory the address may lie in. As such, these figures are used to denote non-uniform memory access (NUMA) architectures.

9.1.2 Communication, Computation, and Locality

With this model, a kernel can be distilled down to the movement of data from one or more memories to a processor where it may be buffered, duplicated, and computed on. That modified data or any new data is then communicated back to those memories.

The movement of data from the memories to the processors, or communication, is bounded by the characteristics of the processor-memory interconnect. Consider Figure 9.1. There is a maximum bandwidth on any link as well as a maximum bandwidth limit on any subset of links i.e., the total bandwidth from or to memory_0 may be individually limited.

Computation, for purposes of this chapter, consists of floating-point operations including multiply, add, compare, etc. Each processor has an associated computation rate. Nominally, as processors are black boxed, one does not distinguish how performance is distributed among cores within a multicore chip. However, when using a hierarchical model for multicore (discussed at the end of this chapter), rather than only modeling memory-processor communication and processor computation, we will model memory-cache communication, cache-core communication, and core computation.

Although there is some initial locality of data in memory_i, once moved to processor_j we may assume that caches seamlessly provide for locality within the processor. That is, subsequent references to that data will not generate capacity misses in 3C's (compulsory, capacity, conflict) vernacular [165]. This technique may be extended to the cache hierarchy.

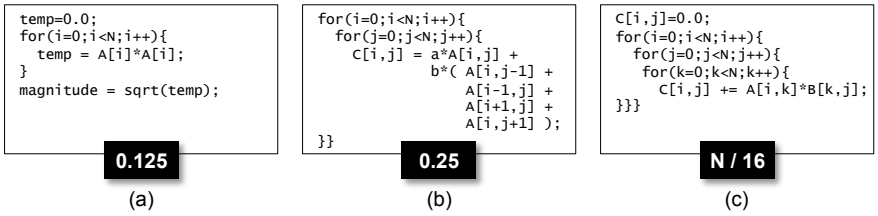

FIGURE 9.2: Arithmetic Intensities for three common HPC kernels including (a) dot products, (b) stencils, and (c) matrix multiplication.

9.1.3 Arithmetic Intensity

Arithmetic intensity is a kernel's ratio of computation to traffic and is measured in flops:bytes. Remember traffic is the volume of data to a particular memory. It is not the number of loads and stores. Processors whose caches filter most memory requests will have very high arithmetic intensities. A similar concept is machine balance [77] which represents the ratio of peak floating-point performance to peak bandwidth. A simple comparison between machine balance and arithmetic intensity may provide some insight as to potential performance bottlenecks. That is, when arithmetic intensity exceeds machine balance, it is likely the kernel will spend more time in computation than communication. As such, it is likely compute bound. Unfortunately such simple approximations gloss over many of the details of computer architecture and result in performance far below performance expectations. Such situations motivated the creation of the roofline model.

9.1.4 Examples of Arithmetic Intensity

Figure 9.2 presents pseudocode for three common kernels within scientific computing: calculation of vector magnitude, a stencil sweep for a 2D PDE, and dense matrix-matrix multiplication. Assume all arrays are double precision.

Arithmetic intensity is the ratio of total floating-point operations to total DRAM bytes. Assuming N is sufficiently large that the array of Figure 9.2(a) does not fit in cache and enough to amortize the poor performance of the square root, then we observe that it performs N flops while transferring only $8 \cdot N$ bytes (N doubles). The second access to `A[i]` exploits the cache/register file locality within the processor. The result is an arithmetic intensity of 0.125 flops per byte.

Figure 9.2(b) presents a much more interesting example. Assuming the processor's cache is substantially larger than $8 \cdot N$, but substantially smaller than $16 \cdot N^2$, we observe that the leading point in the stencil `A[i,j+1]` will eventually be reused by subsequent stencils as `A[i+1,j]`, `A[i,j]`, `A[i-1,j]`, and `A[i,j-1]`. Although references to `A[i,j]` only generate $8 \cdot N^2$ bytes of communication, accesses to `C[i,j]` generate $16 \cdot N^2$ bytes because write-allocate

cache architectures will generate both a read for the initial fill on the write miss in addition to the eventual write back. As the code performs $6 \cdot N^2$ flops, the resultant arithmetic intensity is $(6 \cdot N^2)/(24 \cdot N^2) = 0.25$.

Finally, Figure 9.2(c) shows the pseudocode for a dense matrix-matrix multiplication. Assuming the cache is substantially larger than $24 \cdot N^2$, then `A[i,j]`, `B[i,j]`, and `C[i,j]` can be kept in cache and only their initial and write back references will generate DRAM memory traffic. As such, we observe the loop nest will perform $2 \cdot N^3$ flops while only transferring $32 \cdot N^2$ bytes. The result is an arithmetic intensity of $N/16$.

9.2 The Roofline

Given the aforementioned abstract architectural model and a kernel's estimated arithmetic intensity, we create a intuitive and utilitarian model that allows programmers rather than computer architects to bound attainable performance. We call this model the "Roofline Model." The roofline model is built using Bound and Bottleneck analysis [207]. As such we may consider the two principal performance bounds (computation and communication) in isolation and compare their corresponding times to determine the bottleneck and attainable performance. Consider Figure 9.1(b). Consider a simple homogeneous kernel that must transfer B bytes of data from memory$_0$ and perform $\frac{F}{2}$ floating-point operations on both CPU$_0$ and CPU$_1$. Moreover, assume the the memory can support *PeakBandwidth* bytes per second and combined, and the processors can perform *PeakPerformance* floating-point operations per second. Simple analysis suggests it will take $\frac{B}{\text{PeakBandwidth}}$ seconds to transfer the data and $\frac{F}{\text{PeakPerformance}}$ seconds to compute on it. Assuming one may perfectly overlap communication and computation it will take:

$$\text{Total Time} = \max \begin{cases} F \ / \ \text{PeakPerformance} \\ B \ / \ \text{PeakBandwidth} \end{cases} \tag{9.1}$$

Reciprocating and multiplying by F flops, we observe performance is bound to:

$$\text{AttainablePerformance (Gflop/s)} = \min \begin{cases} \text{PeakPerformance} \\ \text{PeakBandwidth} \times \text{ArithmeticIntensity} \end{cases} \tag{9.2}$$

Where Arithmetic Intensity is F/B.

Although a given architecture has a fixed peak bandwidth and peak performance, arithmetic intensity will vary dramatically from one kernel to the next and substantially as one optimizes a given kernel. As such, we may plot attainable performance as a function of arithmetic intensity. Given the tremendous range in performance and arithmetic intensities, we will plot these figures on a log-log scale.

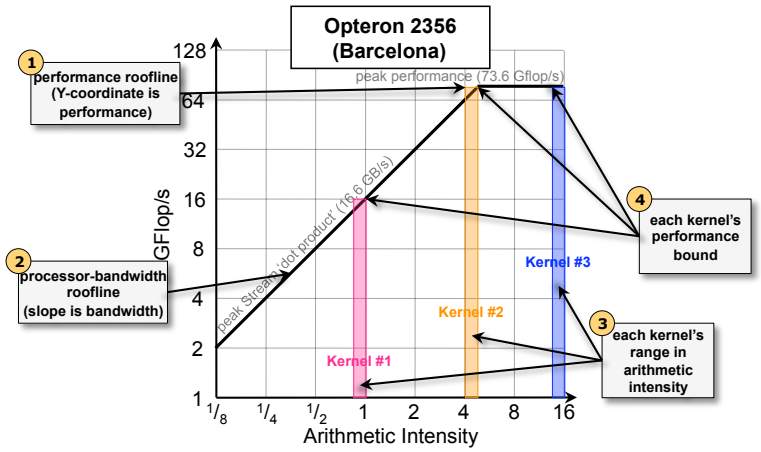

FIGURE 9.3: Roofline Model for an Opteron SMP. Also, performance bounds are calculated for three non-descript kernels.

Using the Stream benchmark [317], one may determine that the maximum bandwidth one can attain using a 2.3 GHz dual-socket × quad-core Opteron 2356 Sun 2200 M2 is 16.6 GB/s. Similarly, using a processor optimization manual it is clear that the maximum performance one can attain is 73.6 Gflop/s. Of course, as shown in Equation 9.2, it is not possible to always attain both, and in practice may not be possible to achieve either.

Figure 9.3 visualizes Equation 9.2 for this SMP via the black line. Observe that as arithmetic intensity increases, so to does the performance bound. However, at the machine's flop:byte ratio, the performance bound saturates at the machine's peak performance. Beyond this point, although performance is at its maximum, used bandwidth decreases. Note, the slope of the roofline in the bandwidth-limited regions is actually the machine's Stream bandwidth. However, on a log-log scale the line always appears at a 45-degree angle. On this scale, doubling the bandwidth will shift the line up instead of changing its perceived slope.

This Roofline model may be used to bound the Opteron's attainable performance for a variety of computational kernels. Consider three generic kernels, labeled 1, 2, and 3 in Figure 9.3, with flop:DRAM byte arithmetic intensities of about 1, 4, and 16 respectively. When mapped onto Figure 9.3, we observe that the Roofline at Kernel #1's arithmetic intensity is in the bandwidth-limited region (i.e., performance is still increasing with arithmetic intensity). Scanning upward from its X-coordinate along the Y-axis, we may derive a performance bound based on the Roofline at said X-coordinate. Thus, it would be unreasonable to expect Kernel #1 to ever attain better than 16 Gflop/s. With an arithmetic intensity of 16, Kernel #3 is clearly ultimately compute-bound. Kernel #2 is a more interesting case as its performance is heavily dependent

on exactly calculating arithmetic intensity as well as both the kernel's and machine's ability to perfectly overlap communication (loads and stores from DRAM) and computation. Failure on any of these three fronts will diminish performance.

In terms of the Roofline model, performance is no longer a scalar, but a coordinate in arithmetic intensity–Gflop/s space. As the roofline itself is only a performance bound, it is common the actual performance will be below the roofline (it can never be above). As programmers interested in architectural analysis and program optimization, we are motivated to understand why performance is below the roofline (instead of on it) and how we may optimize a program to rectify this. The following sections refine the roofline model to enhance its utility in this field.

9.3 Bandwidth Ceilings

Eliciting good performance from modern SMP memory subsystems can be elusive. Architectures exploit a number of techniques to hide memory latency (HW, SW prefetching, TLB misses) and increase memory bandwidth (multiple controllers, burst accesses, NUMA). For each of these architectural paradigms, there is a commensurate set of optimizations that must be implemented to extract peak memory subsystem performance. This section enumerates these potential performance impediments and visualizes them using the concept of *bandwidth performance ceilings*. Essentially a ceiling is structure internal to the roofline denoting a complete failure to exploit an architectural paradigm. In essence, just as the roofline acted to constrain performance to be beneath it, so too do ceilings constrain performance to be beneath them. Software optimization removes these ceilings as impediments to performance.

9.3.1 NUMA

We begin by considering the NUMA issues in the Stream benchmark as it will likely be illustrative of the solution to many common optimization mistakes made when programming multisocket SMPs. As written, there is a loop designed to initialize the values of the arrays to be streamed. Subtly, this loop is also used to distribute data among the processor sockets through the combination of a OpenMP pragma (`#pragma omp parallel for`) and the use of the first touch policy [134]. Although the virtual addresses of the elements appear contiguous, their physical addresses are mapped to the memory controllers on different sockets. This optimized case is well visualized in Figure 9.4(a). We observe the array (grid) has been partitioned with half placed in each of the two memories. When the processors compute on this data they find that the pieces of the array they're tasked with using are in the memory to which

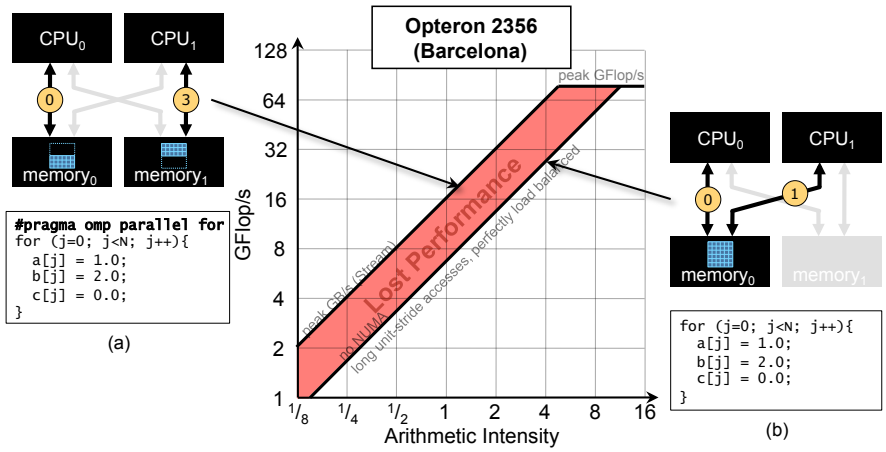

FIGURE 9.4: NUMA ceiling resulting from improper data layout. The codes shown are initialization-only (a) with and (b) without proper exploitation of a first-touch policy. Initialization is completely orthogonal to the possible computational kernels.

they have the highest bandwidth. If, on the other hand, the pragma were omitted, then the array would likely be placed in its entirety within memory$_0$. As such, not only does one forgo half the system's peak bandwidth by not using the other memory, but he also looses additional performance as link 1 likely has substantially lower bandwidth than 0 or 3, but must transfer just as much data. We may plot the resultant bandwidth on the Roofline figure to the right. We observe a 2.5× degradation in performance. Not only will this depress the performance of any memory-bound kernels, but it expands the range of memory-bound arithmetic intensities to about 10 flops per DRAM byte.

Such performance bugs can be extremely difficult to find regardless of whether one uses OpenMP, POSIX threads, or some other threading library. Under very common conditions, it can also occur even when binding threads to cores under pthreads because data is bound to a controller by the OS, not by a `malloc()` call. For example, an initial `malloc()` call followed by an initialization routine may peg certain virtual addresses to one controller or the other. However, if that data is freed, it is returned to the heap, not the OS. As such, a subsequent call to `malloc()` will use data already on the heap, and already pinned to a controller other than the one that might be desired. Unless cognizant of these pitfalls, one should strongly consider only threading applications within a socket instead of across an entire SMP node.

9.3.2 Prefetching, DMA, and Little's Law

Little's Law [41] (see also Chapter 1) states that the concurrency (independent memory operations) that must be injected into the memory subsystem to attain peak performance is the product of memory latency and peak memory bandwidth. For processors like Opterons, this translates into more than 800 bytes of data (perhaps 13 cache lines). Hardware vendors have created a number of techniques to generate this concurrency. Unfortunately, methods like out-of-order execution operate on doubles, not on cache lines. As such, it is difficult to get 100 loads in flight. The more modern methods include software prefetching, hardware prefetching, and DMA. Software prefetching and DMA are similar in that they are both asynchronous software methods of expressing more memory-level parallelism than one could normally achieve via a scalar ISA. The principal difference between the two is granularity. Software prefetching only operates on cache lines, whereas DMA operates on arbitrary numbers of cache lines. Hardware prefetchers attempt to infer a streaming access pattern given a series of cache misses. As such they don't require software modifications, and express substantial memory-level parallelism, but are restricted to particular memory access patterns.

It is conceivable that one could create a version of Stream that mimics the memory access pattern observed in certain applications. For example, a few pseudorandom access pattern streams may individually trip up any hardware or software prefetcher, but collectively allow expression of memory-level parallelism through DMA or software prefetch. As such, one could draw a series of ceilings below the roofline that denote an every decreasing degree of memory-level parallelism.

9.3.3 TLB Issues

Modern microprocessors use virtual memory and accelerate the translation to physical addresses via small highly-associative translation lookaside buffers (TLBs). Unfortunately, these act like caches on the page table (caching page table entries). If a kernel's working set, as measured in page table entries, exceeds the TLB capacity (or associativity), then one generates TLB capacity (or conflict) misses. Such situations arise more often than one might think. Simple cache blocking for matrix multiplication can result in enough disjoint address streams, which although they may fit in cache, do not fit in the TLB.

One could implement a version of Stream that scales the number of streams for operations like TRIAD. Doing so would often result in a about the same bandwidth for low numbers of streams, but would suddenly dip for an additional array. This dip could be plotted on the roofline model as a bandwidth ceiling, and labeled with the number of arrays required to trigger it.

9.3.4 Strided Access Patterns

A common solution to the above problems is to lay out the data as one multicomponent array (i.e., an array of cartesian vectors instead of three arrays one for each component). However, the computational kernels may not use all of these components at a time. Nevertheless, the data must still be transfered. Generally, small strides (less than the cache line size) should be interpreted as a decrease in arithmetic intensity, where large strides can represent a lack of spatial locality and memory-level parallelism. One may plot a ceiling for each stride with the roofline being stride-1 (unit-stride).

9.4 In-Core Ceilings

Given the complexity of modern core architectures, floating-point performance is not simply a function of arithmetic intensity alone. Rather architectures exploit a number of paradigms to improve peak performance including pipelining, superscalar out-of-order execution, SIMD, hardware multithreading, multicore, heterogeneity, etc. Unfortunately, a commensurate set of optimizations (either generated by the compiler or explicitly expressed by the user) are required to fully exploit these paradigms. This section enumerates these potential performance impediments and visualizes them using the concept of *in-core performance ceilings*. Like bandwidth ceilings, these ceilings act to constrain performance coordinates to lie beneath them. The following section enumerates some of the common ceilings. Each is examined in isolation. All code examples assume an x86 architecture.

9.4.1 Instruction-Level Parallelism

Every instruction on every architecture has an associated latency representing the time from when the instruction's operands are available to the time where the results are made available to other instructions (assuming no other resource stalls). For floating-point computational instructions like multiply or add, these latencies are small (typically less than 8 cycles). Moreover, every microprocessor has an associated dispatch rate (a bandwidth) that represents how many independent instructions can be executed per cycle. Just as Little's Law can be used to derive the concurrency demanded by the memory subsystem using the bandwidth–latency product, so too can it be used to estimate the concurrency each core demands to keep its functional units busy. We define this to be the *instruction-level parallelism*. When a thread of execution falls short of expressing this degree of parallelism, functional units will go idle, and performance will suffer [73].

As an example, consider Figure 9.5. In this case we plot scalar floating-

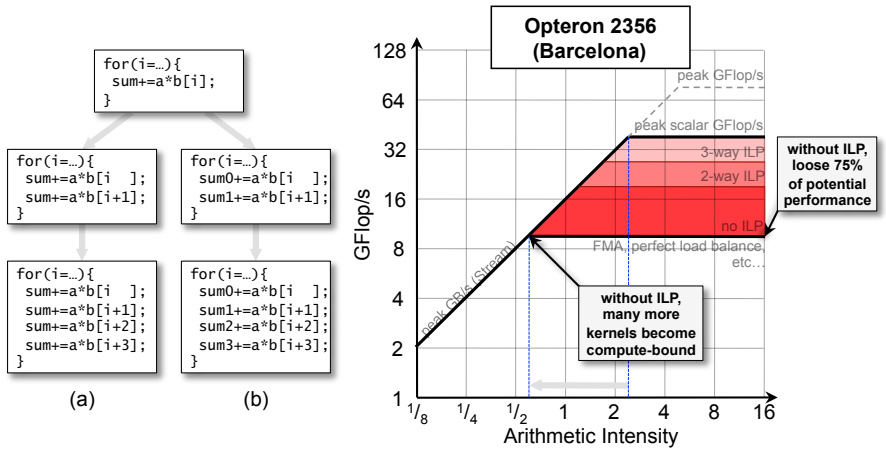

FIGURE 9.5: Performance ceilings as a result of insufficient instruction-level parallelism. (a) Code snippet in which the loop is unrolled. Note (FP add) instruction-level parallelism (ILP) remains constant. (b) Code snippet in which partial sums are computed. ILP increases with unrolling.

point performance as a function of DRAM arithmetic intensity. However, on the roofline figure we note the performance impact from a lack of instruction-level parallelism through instruction-level parallelism (ILP) ceilings. Figure 9.5(a) presents a code snippet in which the loop is naïvely unrolled either by the user or the compiler. Although this has the desired benefit of amortizing an loop overhead, it does not increase the floating-point add instruction-level parallelism — the adds to `sum` will be serialized. Even a superscalar processor must serialize these operations. Conversely, Figure 9.5(b) shows an alternate unrolling method in which partial sums are maintained within the loop and reduced (not shown) upon loop completion. If one achieves sufficient cache locality for `b[i]` then arithmetic intensity will be sufficiently great that Figure 9.5(b) should substantially outperform (a).

Subtly, without instruction-level parallelism, the arithmetic intensity at which a processor becomes compute-bound is much lower. In a seemingly paradoxical result, it is possible that many kernels may show the signs of being compute-bound (parallel efficiency), yet deliver substantially suboptimal performance.

9.4.2 Functional Unit Heterogeneity

Processors like AMD's Opteron's and Intel's Nehalem have floating-point execution units optimized for certain instructions. Specifically, although they both have two pipelines capable of simultaneously executing two floating-point instructions, one pipeline may only perform floating-point additions,

while the other may only perform floating-point multiplies. This creates a potential performance impediment. For codes that are dominated by one or the other, attainable performance will be half that of a code that has a perfect balance between multiplies and adds. For example, codes that solve PDEs on structured grids often perform stencil operations which are dominated by adds with very few multiplies, where codes that perform dense linear algebra often see a near perfect balance between multiplies and adds. As such, we may create a series of ceilings based on the ratio of adds to multiplies. As the ratio gets further and further from 1, the resultant ceiling will approach one half of peak.

Processors like Cell, GPUs, POWER, and Itanium exploit what is known as fused-multiply add (FMA). These instructions are implemented on execution units where instead of performing multiplies and adds in parallel, they are performed in sequence (multiply two numbers and add a third to the result). Obviously the primary advantage of such an implementation is to execute the same number of floating-point operations as a machine of twice the instruction issue width. Nevertheless, such an architecture creates a similar performance issue to the case of separate multipliers and adders in that unless the code is entirely dominated by FMA's, performance may drop by a factor of two.

9.4.3 Data-Level Parallelism

Modern microprocessor vendors have attempted to boost their peak performance through the addition of Single Instruction Multiple Data (SIMD) operations. In effect, with a single instruction, a program may express two or four-way data-level parallelism. For example, the x86 instruction `addps` performs four single-precision floating-point add operations in parallel. Ideally the compiler should recognize this form of parallelism and generate these instructions. However, due to the implementation's rigid nature, compilers often fail to generate these instructions. Moreover, even programmers may not be able to exploit them due to rigid program and data structure specifications. Failure to exploit these instructions can substantially depress kernel performance.

Consider Figure 9.6. The code is a simplified version of that in Figure 9.5(b). We observe there is substantial ILP, but only floating-point adds are performed. As such, there is no data-level parallelism, and performance is bounded to less than 18.4 Gflop/s. Most x86 compilers allow the user to SIMDize their code via *intrinsics* — small functions mapped directly to one or two instructions. We observe that the first step in this process is to replace the conventional C assignments with the scalar form of these intrinsics. Of course doing so will not improve our performance bound because it has not increased the degree of data-level parallelism. However, when using the `_pd` form of the intrinsics we should unroll the loop 8 times so that we may simultaneously express both 2-way data level parallelism and 4-way instruction-level parallelism. Doing so improves our performance bound to 36.8 Gflop/s. As

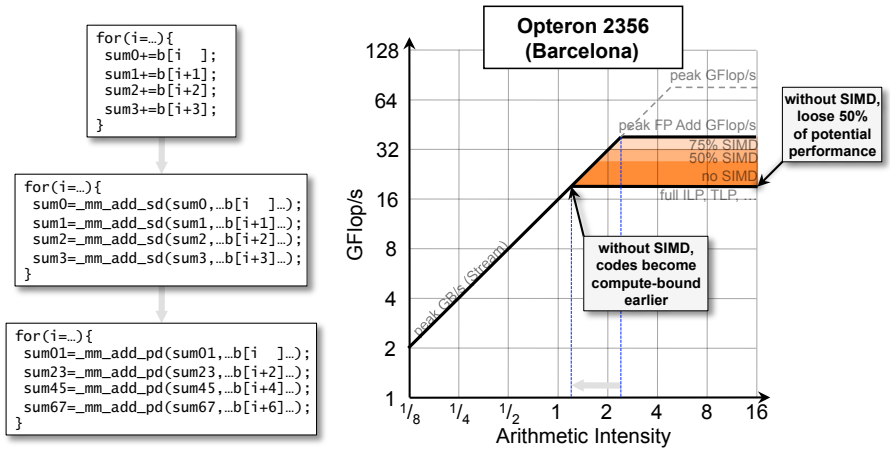

FIGURE 9.6: Example of Ceilings associated with data-level parallelism.

discussed in the previous subsection, we cannot achieve 73.6 due to the fact that this code does not perform any floating-point multiplies.

9.4.4 Hardware Multithreading

Hardware multithreading [164] has emerged as an effective solution to the memory- and instruction-level parallelism problems with a single architectural paradigm. Threads whose current instruction's operands are ready are execute while the others wait in a queue. As all of this is performed in hardware, there is no apparent context switching. There are no ILP ceilings, as typically there is enough thread-level parallelism (TLP) to cover the demanded instruction-level parallelism. Moreover, the exemplar of this architecture, Sun's Niagara [273], doesn't implement SIMD or heterogeneous functional units. However, a different set of ceilings normally not seen on superscalar processors appear: the floating-point fraction of the dynamics instruction mix. All processors have a finite instruction fetch and decode bandwidth (the number of instructions that can be fetched per cycle). On superscalar processors, this bandwidth is far greater than the instruction bandwidth required under ideal conditions to saturate the floating-point units. However, on processors like Niagara, as the floating-point fraction dips below 50%, the non-floating-point instructions begin to sap instruction bandwidth away from the floating-point pipeline. The result: performance drops. The only effective solution here is improving the quality of code generation.

More recently, superscalar manufactures have begun to introduce hardware multithreading into their processor lines including Nehalem, Larrabee, and POWER7. In such situations, SPMD programs may not suffer from ILP

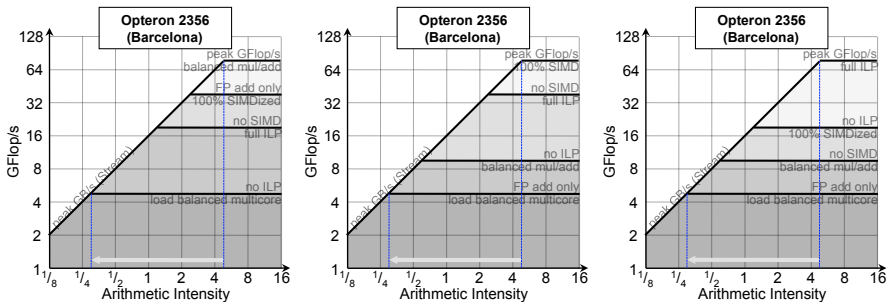

FIGURE 9.7: Equivalence of Roofline models.

ceilings but may invariably see substantial performance degradation due to data-level parallelism (DLP) and heterogenous functional unit ceilings.

9.4.5 Multicore Parallelism

Multicore has introduced yet another form of parallelism within a socket. When programs regiment cores (and threads) into bulk synchronous computations (compute/barrier), load imbalance can severely impair performance. Such an imbalance can be plotted using the roofline model. To do this, one may count the total number of floating-point operations performed across all threads and the time between the start of the computation and when the last thread enters the barrier. The ratio of these two numbers is the (load imbalanced) attained performance. Similarly, one can sum the times each thread spends in computation and divide by the total number of threads. The ratio of total flops to this number is the performance that could be attained if properly load balanced. As such, one can visualize the resultant performance loss as a load balance ceiling.

9.4.6 Combining Ceilings

All these ceilings are independent and thus may be combined as needed. For example, a lack of instruction-level parallelism can be combined with a lack of data-level parallelism to severely depress performance. As such, one may draw multiple ceilings (representing the lack of different forms of parallelism) on a single roofline figure as visualized in Figure 9.7.

However, the question of how ceilings should be ordered arises. Often, one uses intuition to order the ceilings based on which are most likely to be implicit in the algorithm or discovered by the compiler. Ceilings placed near the roofline are those that are not present in the algorithm or unlikely to be discovered by the compiler. As such, based on this intuition, one could adopt any of the three equivalent roofline models in Figure 9.7.

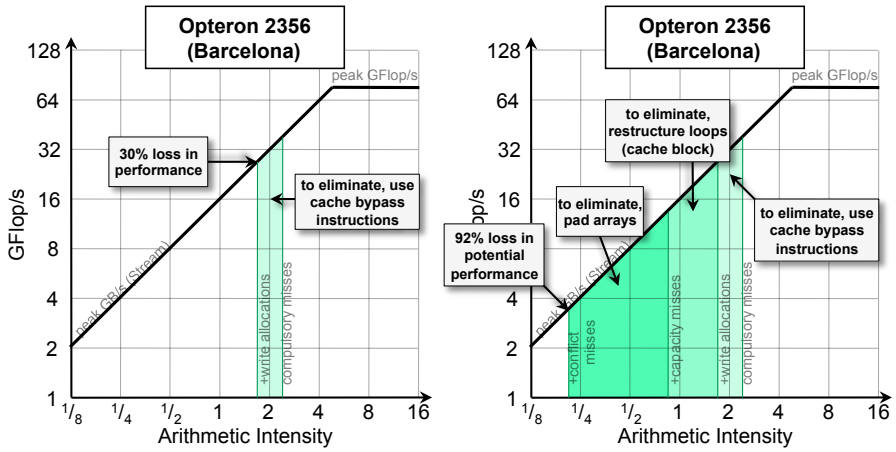

FIGURE 9.8: Performance interplay between Arithmetic Intensity and the Roofline for two different problem sizes for the same nondescript kernel.

9.5 Arithmetic Intensity Walls

Thus far, we've assumed the total DRAM bytes within the arithmetic intensity ratio is dominated by "compulsory" memory traffic in 3C's parlance [165]. Unfortunately, on real codes there are a number of other significant terms in the arithmetic intensity denominator.

$$\text{AI} = \frac{\text{Total FP Operations}}{\text{Compulsory} + \text{Allocation} + \text{Capacity} + \text{Conflict Memory Traffic} + ...} \quad (9.3)$$

In much the same way one denotes ceilings to express a lack of instruction, data, or memory parallelism, one can denote *arithmetic intensity walls* to denote reduced arithmetic intensity as a result of different types superfluous memory traffic above and beyond the compulsory memory traffic — essentially, additional terms in the denominator. As such, Equation 9.3 shows that write allocation traffic, capacity cache misses, and conflict cache misses, among others, contribute to reduced arithmetic intensity. These arithmetic intensity walls act to constrain arithmetic intensity and, when bandwidth-limited, constrain performance and is visualized in Figure 9.8. As capacity and conflict misses are heavily dependent on whether the specified problem size exceeds the cache's capacity and associativity, the walls are execution-dependent rather than simply architecture-dependent. That is, a small, non-power-of-two problem may not see any performance degradation due to capacity or conflict miss traffic, but for the same code, a large, near power-of-two problem size may

result in substantial performance loss, as arithmetic intensity is constrained to be less than 0.2.

The following subsections discuss each term in the denominator and possible solutions to their impact on performance.

9.5.1 Compulsory Miss Traffic

It should be noted that compulsory traffic is not necessarily the minimum memory traffic for an algorithm. Rather compulsory traffic is only the minimum memory traffic required for a particular implementation. The most obvious example of elimination of compulsory traffic is changing data types. e.g., `double` to `single` or `int` to `short`. For memory-bound kernels, this transformation may improve performance by a factor of two, but should only be performed if one can guarantee correctness always or through the creation of special cases. More complex solutions involve in-place calculations or register blocking sparse matrix codes [346].

9.5.2 Capacity Miss Traffic

Both caches and local stores have a finite capacity. In the case of the former, when a kernel's working set exceeds the cache capacity, the cache hardware will detect that data must be swapped out, and capacity misses will occur. The result is an increase in DRAM memory traffic, and a reduced arithmetic intensity. When performance is limited by memory bandwidth, it will diminish by a commensurate amount. In the case of local stores, a program whose working size exceeds the local store size will not function correctly.

Interestingly, the most common solution to eliminating capacity misses on cache-based architectures is the same as to obtaining correct behavior on local store machines: cache blocking. In this case loops are restructured to reduce the working set size and maximize arithmetic intensity.

9.5.3 Write Allocation Traffic

Most caches today are *write-allocate*. That is, upon a write miss, the cache will first evict the selected line, then load the target line from main memory. The result is that writes generate twice the memory traffic as reads: cache line fill plus a write back vs. one fill. Unfortunately, this approach is often wasteful on scientific codes where large blocks of arrays are immediately written without being read. There is no benefit in having loaded the cache line when the next memory operations will obliterate the existing data. As such, the write fill was superfluous and should be denoted as a arithmetic intensity wall.

Modern architectures often provide a solution to this quandry either in the form of SSE's cache bypass instruction `movntpd` or PowerPC's block init instruction `dcbz`. The use of the `movntpd` instruction allows programs to bypass the cache in its entirety and write to the write combining buffers. The advan-

tage: elimination of write allocation traffic and cache pressure is reduced. The `dcbz` instruction allocates a line in the cache and zeros its contents. The advantage is that write allocation traffic has been eliminated, but cache pressure has not been reduced.

9.5.4 Conflict Miss Traffic

Similarly, unlike local stores, caches are not fully associative. That is, depending on address, only certain locations in the cache maybe used to store the requested cache line — a *set*. When one exhausts this associativity of the set, one element from that set must be selected for eviction. The result: a conflict miss and superfluous memory traffic.

Conflict misses are particularly prevalent on power-of-two problem sizes as this is a multiple of the number of sets in a cache, but can be notoriously difficult to track down due to the complexities of certain memory access patterns. Nevertheless for many well structured codes, one may pad arrays or data structures cogniziant of the memory access pattern to ensure that different sets are accessed and conflict misses are avoided. Conceptually, 1D array padding transforms an array from `Gird[Z][Y][X])` to `Gird[Z][Y][X+pad])` regardless of whether the array was statically or dynamically allocated.

9.5.5 Minimum Memory Quanta

Naïvely, one could simple count the number of doubles a program references and estimate arithmetic intensity. However, one should be mindful that both cache- and local store-based architectures operate on some minimum memory quanta hereafter referred to as *cache lines*. Typically these lines are either 64 or 128 bytes. All loads and stores after being filtered by the cache are aggregated into these lines. When this data is not subsequently used in its entirety, superfluous memory traffic has been consumed without a performance benefit. As such another term is added to the denominator and arithmetic intensity is depressed.

9.5.6 Elimination of Superfluous Floating-Point Operations

Normally, when discussing arithmetic intensity walls, we think of adding terms to the denominator of arithmetic intensity. However, one should consider the possiblity that the specified number of floating-point operations may not be a minimum, but just a compulsory number set forth by a particular implementation. For instance, one might calculate the number of flops within a loop and scale by the number of loop iterations to calculate a kernel's flop count. However, the possibility of common subexpression elimination (CSE) exists when one or more loop iterations are inspected in conjunction. The result is that the flop count may be reduced. This has the seemingly paradoxical results of decreased floating-point performance, but improved application per-

formance. The floating-point performance may decrease because arithmetic intensity was reduced while bandwidth-limited. However, because the total requisite work (as measured in floating-point operations) was reduced, the time to solution may have also been reduced.

Although this problem may seem academic, it has real world implications as a compiler may discover CSE optimizations the user didn't. When coupled with performance counter measured flop counts, the user may find himself in a predicament rectifying his performance estimations and calculations and the empirical performance observations.

9.6 Alternate Roofline Models

Thus far, we've only discussed a one-level processor-memory abstraction. However, there are certain computational kernel–architecture combinations for which increased optimization creates a new bandwidth bottleneck — cache bandwidth. One may construct separate roofline models for each level of the hierarchy and then determine the overall bottleneck. In this section we discuss this approach and analyze example codes.

9.6.1 Hierarchically Architectural Model

One may refine the original processor–memory architectural model by hierarchically refining the processors into cores and cache (which essentially look like another level of processors and memories). Thus, if the CPUs of Figure 9.1 were in fact dual-core processors, one could construct several different hierarchical models (Figure 9.9) depending on the cache/local store topology. Figure 9.9(a) shows it is possible for core_0 to read from cache_3 (simple cache coherency), but on the local store architecture, although any core can read from any DRAM location, core_0 can only read LocalStore_0.

Just as there were limits on both individual and aggregate processor-memory bandwidths, so too are there limits on both individual and aggregate core-cache bandwidths. As a result, what were NUMA ceilings (arising when data crossed low bandwidth/high load links) when transferring data from memory to processor, become NUCA (non-uniform cache access) ceilings when data resident in one or more caches must cross low bandwidth/high load links to particular cores

Ultimately, this approach may be used to refine cores down to the register file–functional unit level. However, when constructing a model to analyze a particular kernel, the user may have some intuition as to where the bottleneck lies — i.e., L2 cache–core bandwidth with good locality in the L2. In such a situations, there is no need to construct a model with coarser granularities (L3, DRAM, etc...) or finer granularities (register files).

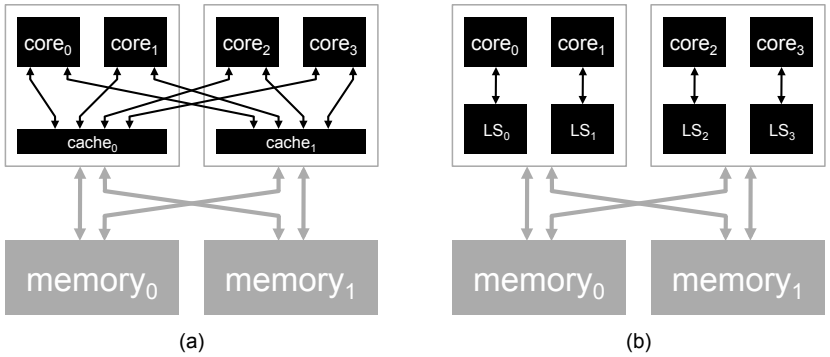

FIGURE 9.9: Refinement of the previous simple bandwidth-processor model to incorporate (a) caches or (b) local stores. Remember, arrows denote the ability to access information and not necessarily hardware connectivity.

9.6.2 Hierarchically Roofline Models

Given this memory hierarchy, we may model performance using two roofline models. First, we model the performance involved in transferring the data from DRAM to the caches or local stores. This of course means we must calculate an arithmetic intensity based on how data will be disseminated among the caches and the total number of floating-point operations. Using this arithmetic intensity and the characteristics of the processor–DRAM interconnect, we may bound attainable performance. Second, we calculate core-cache arithmetic intensity involved in transferring data to/from caches or local stores. We may also plot this using the roofline model. This bound may be a tighter or looser bound depending on architecture and kernel.

Such hierarchical models are especially useful when arithmetic intensity scales with cache capacity as it does for dense matrix-matrix multiplication. For such cases we must select a block size that is sufficiently large that the code will be limited by core performance rather than cache-core or DRAM-processor bandwidth.

9.7 Summary

The roofline model is a readily accessible performance model intended to provide performance intuition to computer scientists and computational scientists alike. Although the roofline proper is a rather loose upper bound to performance, it may be refined through the use of bandwidth ceilings, in-core

ceilings, arithmetic intensity walls, and hierarchical memory architectures to provide much tighter performance bounds.

9.8 Acknowledgments

The author wishes to express his gratitude to Professor David A. Patterson and Andrew Waterman for their help in creation of this model. This work was supported by the ASCR Office in the DOE Office of Science under contract number DE-AC02-05CH11231, Microsoft (Award #024263), Intel (Award #024894), and by matching funding through U.C. Discovery (Award #DIG07-10227).

9.9 Glossary

3C's: is a methodology of categorizing cache misses into one of three types: **c**ompulsory, **c**apacity, and **c**onflict. Identification of miss type leads programmers to software solutions and architects to hardware solutions.

Arithmetic Intensity: is a measure of locality. It is calculated as the ratio of floating-point operations to DRAM traffic in bytes.

Bandwidth: is the average rate at which traffic may be communicated. As such it is measured as the ratio of total traffic to total time and today is measured in 10^9 bytes per second (GB/s).

Ceiling: a performance bound based on the lack of exploitation of an architectural paradigm.

Communication: is the movement of traffic from a particular memory to a particular computational element.

Computation: represents local FLOPs performed on the data transfered to the computational units.

DLP: data-level parallelism represents the number of independent data items for which the same operation can be concurrently applied. DLP can be recast as ILP.

FLOP: a floating-point operation including adds, subtracts, and multiplies,

but often includes divides. It is generally not appropriate to include operations like square roots, logarithms, exponentials, or trigonometric functions as these are typically decomposed into the base floating-point operations.

ILP: instruction-level parallelism represents the number of independent instructions than can be executed concurrently.

Kernel: a deterministic computational loop nest that performs floating-point operations.

Performance: Conceptually similar to bandwidth, performance is a measure of the average rate computation is performed. As such it is calculated as the ratio of total computation to total time and today is measured in 10^9 floating-point operations per second (Gflop/s) on multicore SMPs.

Roofline: The ultimate performance bound based on peak bandwidth, peak performance, and arithmetic intensity.

TLP: instruction-level parallelism represents the number of independent threads (instruction streams) than can be executed concurrently.

Traffic: or communication is the volume of data that must be transfered to or from a computational element. It is measured in bytes. Often, we assume each computational element has some cache or internal storage capacity so that memory references are efficiently filtered to compulsory traffic.