

*David H. Bailey and Robert F. Lucas, Editors*

---

# ***Performance Tuning of Scientific Applications***



---

## List of Figures

2.1	High-level architectural model showing two black-boxed processors either connected to separate memories or to a common shared memory. The arrows denote the ISA's ability to access information and not necessarily hardware connectivity. . . . .	13
2.2	Arithmetic Intensities for three common HPC kernels . . . . .	14
2.3	Roofline Model for an Opteron SMP. Also, performance bounds are calculated for three non-descript kernels. . . . .	16
2.4	NUMA ceiling resulting from improper data layout. The code shown is initialization-only. It is not the possible computational kernels. . . . .	17
2.5	Performance ceilings as a result of insufficient instruction-level parallelism. (a) a code snippet in which the loop is unrolled. note, (FP add) ILP remains constant. (b) code snippet in which partial sums are computed. ILP increases with unrolling. . . . .	21
2.6	Example of Ceilings associated with data-level parallelism. . . . .	23
2.7	Equivalence of Roofline models . . . . .	24
2.8	Performance interplay between Arithmetic Intensity and the Roofline for two different problem sizes for the same non-descript kernel. . . . .	25
2.9	Refinement of the previous simple bandwidth-processor model to incorporate caches or local stores. Remember, arrows denote the ability to access information and not necessarily hardware connectivity. . . . .	28
5.1	Visualization of the data structures associated with the heat equation stencil. (a) the 3D temperature grid. (b) the stencil operator performed at each point in the grid. (c) pseudocode for stencil operator. . . . .	37
5.2	Visualization of the datastructures associated with LBMHD. (a) the 3D macroscopic grid. (b) the D3Q27 momentum scalar velocities. (c) D3Q15 magnetic vector velocities. (d) C structure of arrays datastructure. Note, each pointer refers to a $N^3$ grid, and $X$ is the unit stride dimension. . . . .	39

5.3	Sparse Matrix Vector Multiplication (SpMV). (a) visualization of the algebra: $y \leftarrow Ax$ , where $A$ is a sparse matrix. (b) Standard compressed sparse row (CSR) representation of the matrix. This structure of arrays implementation is favored on most architectures. (c) The standard implementation of SpMV for a matrix stored in CSR. The outer loop is trivially parallelized without any data dependencies. . . . .	40
5.4	Benchmark matrices used as inputs to our auto-tuned SpMV library framework. . . . .	41
5.5	Generic visualization of the auto-tuning flow. . . . .	48
5.6	Using a pointer-to-function table to accelerate auto-tuning search. . . . .	49
5.7	Benefits of auto-tuning the 7-point Laplacian Stencil. . . . .	51
5.8	Benefits of auto-tuning the lattice Boltzmann Magnetohydrodynamics (LBMHD) application. . . . .	52
5.9	Auto-tuning Sparse Matrix-Vector Multiplication. Note, horizontal axis is the matrix (problem) and multicore scalability is not shown. . . . .	53

---

## *List of Tables*

5.1	Architectural summary of evaluated platforms. . . . .	35
5.2	Interplay between the bottleneck each optimization addresses (parallelism, memory traffic, memory bandwidth, in-core performance) and the impact on implementation (code-only, data structures, styles of parallelism). Obviously, changing data or parallelism structure will mandate some code changes. †Efficient SIMD requires data structures be aligned to 128-byte boundaries. . . . .	47



---

# Contents

<b>I</b>	<b>Introduction</b>	<b>3</b>
<b>II</b>	<b>Modern Computer Architectures and Performance</b>	<b>5</b>
<b>III</b>	<b>Performance Measurement and Benchmarking</b>	<b>7</b>
<b>IV</b>	<b>Performance Modeling</b>	<b>9</b>
<b>2</b>	<b>The Roofline Model</b>	<b>11</b>
	<i>Samuel W. Williams</i>	
2.1	Introduction . . . . .	12
2.1.1	Abstract Architecture Model . . . . .	12
2.1.2	Communication, Computation, and Locality . . . . .	13
2.1.3	Arithmetic Intensity . . . . .	13
2.1.4	Examples of Arithmetic Intensity . . . . .	14
2.2	The Roofline . . . . .	15
2.3	Bandwidth Ceilings . . . . .	17
2.3.1	NUMA . . . . .	18
2.3.2	Prefetching, DMA, and Little's Law . . . . .	18
2.3.3	TLB issues . . . . .	19
2.3.4	Strided Access Patterns . . . . .	19
2.4	In-Core Ceilings . . . . .	20
2.4.1	Instruction-Level Parallelism . . . . .	20
2.4.2	Functional Unit Heterogeneity . . . . .	21
2.4.3	Data-Level Parallelism . . . . .	22
2.4.4	Hardware Multithreading . . . . .	22
2.4.5	Multicore Parallelism . . . . .	23
2.4.6	Combining Ceilings . . . . .	24
2.5	Arithmetic Intensity Walls . . . . .	24
2.5.1	Compulsory Miss traffic . . . . .	25
2.5.2	Capacity Miss traffic . . . . .	26
2.5.3	Write Allocation Traffic . . . . .	26
2.5.4	Conflict Miss Traffic . . . . .	26
2.5.5	Minimum Memory Quanta . . . . .	27
2.5.6	Elimination of Superfluous Floating-point Operations . . . . .	27

2.6	Alternate Roofline Models . . . . .	28
2.6.1	Hierarchically Architectural Model . . . . .	28
2.6.2	Hierarchically Roofline Models . . . . .	29
2.7	Summary . . . . .	29
2.8	Acknowledgements . . . . .	29
2.9	Glossary . . . . .	30
<b>V</b>	<b>Automatic Performance Tuning</b>	<b>31</b>
<b>5</b>	<b>Auto-tuning Memory-Intensive Kernels for Multicore</b>	<b>33</b>
	<i>Samuel W. Williams, Kaushik Datta, Leonid Oliker, and <b>more authors???</b></i>	
5.1	Introduction . . . . .	34
5.2	Experimental Setup . . . . .	35
5.2.1	AMD Opteron 2356 (Barcelona) . . . . .	36
5.2.2	Xeon E5345 (Clovertown) . . . . .	36
5.2.3	IBM Blue Gene/P (Compute Node) . . . . .	36
5.3	Computational Kernels . . . . .	37
5.3.1	Laplacian Differential Operator (Stencil) . . . . .	37
5.3.2	Lattice Boltzmann Magnetohydrodynamics (LBMHD)	38
5.3.3	Sparse Matrix-Vector Multiplication (SpMV) . . . . .	39
5.4	Optimizing Performance . . . . .	42
5.4.1	Parallelism . . . . .	42
5.4.2	Minimizing Memory Traffic . . . . .	43
5.4.3	Maximizing Memory Bandwidth . . . . .	45
5.4.4	Maximizing In-core Performance . . . . .	46
5.4.5	Interplay between Benefit and Implementation . . . . .	46
5.5	Automatic Performance Tuning . . . . .	46
5.5.1	Code Generation . . . . .	48
5.5.2	Auto-tuning Benchmark . . . . .	48
5.5.3	Search Strategies . . . . .	49
5.6	Results . . . . .	50
5.6.1	Laplacian Stencil . . . . .	50
5.6.2	Lattice Boltzmann Magnetohydrodynamics (LBMHD)	52
5.6.3	Sparse Matrix-Vector Multiplication (SpMV) . . . . .	53
5.7	Summary . . . . .	54
5.8	Acknowledgments . . . . .	54
	<b>Bibliography</b>	<b>55</b>



**Part I**

**Introduction**



Part II

**Modern Computer  
Architectures and  
Performance**



Part III

**Performance Measurement  
and Benchmarking**



**Part IV**

**Performance Modeling**



# Chapter 2

## The Roofline Model

Samuel W. Williams

*Lawrence Berkeley National Laboratory*

2.1	Introduction .....	12
2.1.1	Abstract Architecture Model .....	12
2.1.2	Communication, Computation, and Locality .....	12
2.1.3	Arithmetic Intensity .....	13
2.1.4	Examples of Arithmetic Intensity .....	14
2.2	The Roofline .....	15
2.3	Bandwidth Ceilings .....	17
2.3.1	NUMA .....	17
2.3.2	Prefetching, DMA, and Little's Law .....	18
2.3.3	TLB issues .....	19
2.3.4	Strided Access Patterns .....	19
2.4	In-Core Ceilings .....	19
2.4.1	Instruction-Level Parallelism .....	20
2.4.2	Functional Unit Heterogeneity .....	21
2.4.3	Data-Level Parallelism .....	22
2.4.4	Hardware Multithreading .....	22
2.4.5	Multicore Parallelism .....	23
2.4.6	Combining Ceilings .....	24
2.5	Arithmetic Intensity Walls .....	24
2.5.1	Compulsory Miss traffic .....	25
2.5.2	Capacity Miss traffic .....	26
2.5.3	Write Allocation Traffic .....	26
2.5.4	Conflict Miss Traffic .....	26
2.5.5	Minimum Memory Quanta .....	27
2.5.6	Elimination of Superfluous Floating-point Operations .....	27
2.6	Alternate Roofline Models .....	27
2.6.1	Hierarchically Architectural Model .....	28
2.6.2	Hierarchically Roofline Models .....	29
2.7	Summary .....	29
2.8	Acknowledgements .....	29
2.9	Glossary .....	30

The Roofline model is a visually intuitive performance model constructed using bound and bottleneck analysis [11–13]. 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 2.1 defines the abstract architecture model used by the roofline model. Section 2.2 introduces the basic form of the roofline model, where sections Section 2.3–2.6 iteratively refine the model with tighter and tighter performance bounds.

---

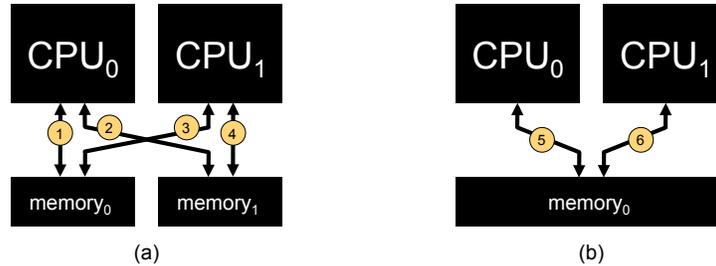
## 2.1 Introduction

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.

### 2.1.1 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 2.1. We show two different dual-processor architectures. Conceptually, any processor can reference any memory location. However, Figure 2.1(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 memory the address may lie in. As such, these figures are used to denote non-uniform memory access (NUMA) architectures.



**FIGURE 2.1:** High-level architectural model showing two black-boxed processors either connected to separate memories or to a common shared memory. The arrows denote the ISA’s ability to access information and not necessarily hardware connectivity.

### 2.1.2 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 2.1. 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<sub>0</sub> 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, he will model memory–cache communication, cache–core communication, and core computation.

Although there is some initial locality of data in memory<sub>*i*</sub>, once moved to processor<sub>*j*</sub>, 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 vernacular [6]. This technique may be extended to the cache hierarchy.

### 2.1.3 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 par-

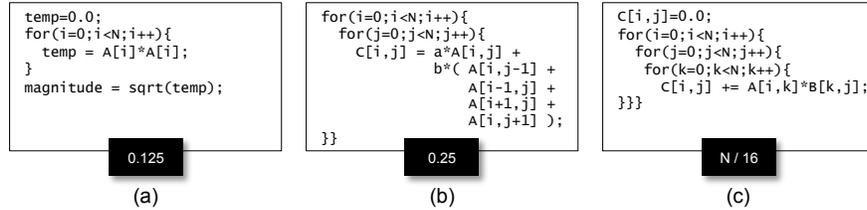


FIGURE 2.2: Arithmetic Intensities for three common HPC kernels

ticular 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 [3] 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.

### 2.1.4 Examples of Arithmetic Intensity

Figure 2.2 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 2.2(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  $8 \cdot N$  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 2.2(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 the results is that references to  $A[i, j]$  only generates  $8 \cdot N^2$  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  $\frac{6 \cdot N^2}{24 \cdot N^2} = 0.25$ .

Finally, Figure 2.2(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  $\frac{N}{16}$ .

---

## 2.2 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 [7]. As such we may consider the two principle performance bounds (computation and communication) in isolation and compare their corresponding times to determine the bottleneck and attainable performance. Consider Figure 2.1(b). Assuming we have a simple homogenous kernel that must transfer  $B$  bytes of data from memory<sub>0</sub> and perform  $\frac{F}{2}$  floating-point operations on both CPU<sub>0</sub> and CPU<sub>1</sub>, the memory can support *PeakBandwidth* bytes per second and combined, 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:

$$\text{Total Time} = \max \left\{ \begin{array}{l} F / \text{PeakPerformance} \\ B / \text{PeakBandwidth} \end{array} \right. \quad (2.1)$$

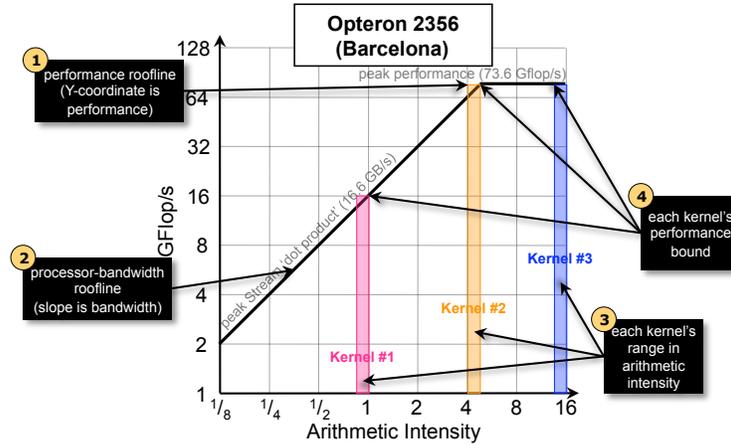
Reciprocating and multiplying by  $F$  Flops, we observe performance is bound to:

$$\text{AttainablePerformance (GFlop/s)} = \min \left\{ \begin{array}{l} \text{PeakPerformance} \\ \text{PeakBandwidth} \times \text{ArithmeticIntensity} \end{array} \right. \quad (2.2)$$

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.

Using the Stream benchmark [9], one may determine that the maximum bandwidth one can attain using a 2.3GHz dual-socket  $\times$  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.

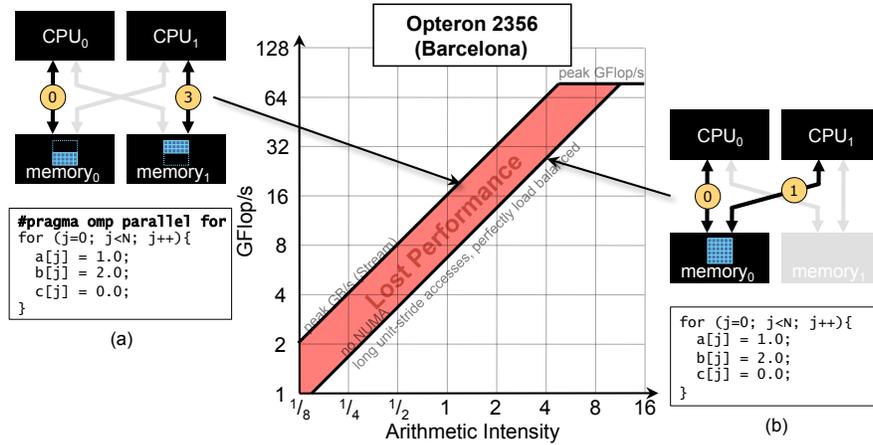


**FIGURE 2.3:** Roofline Model for an Opteron SMP. Also, performance bounds are calculated for three non-descript kernels.

Of course, as shown in Equation 2.2, it is not possible to always attain both, and in practice may not be possible to achieve either.

Figure 2.3 visualizes Equation 2.2 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 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 is 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 2.3, with Flop:DRAM byte arithmetic intensities of about 1, 4, and 16 respectively. When mapped onto Figure 2.3, 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.



**FIGURE 2.4:** NUMA ceiling resulting from improper data layout. The code shown is initialization-only. It is not the possible computational kernels.

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.

## 2.3 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.

### 2.3.1 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 an OpenMP pragma (`#pragma omp parallel for`) and the use of the first touch policy [4]. 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 2.4(a). We observe the array (blue 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 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<sub>0</sub>. As such, not only does one forgo half the system's peak bandwidth by not using the other memory, but he also loses 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  $2.5\times$  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 `free()`'d, 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 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.

### 2.3.2 Prefetching, DMA, and Little's Law

Little's Law [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 don't operate on cache lines but doubles. As such, it is difficult to

get a hundred 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 principle different between the two is granularity. Software prefetching only operates on cache lines where 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, 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 every decreasing degrees of memory-level parallelism.

### 2.3.3 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.

### 2.3.4 Strided Access Patterns

A common solution to the above problem is to lay out the data as one multicomponent array (*i.e.* an array of cartesian vectors instead of 3 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 transferred. 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).

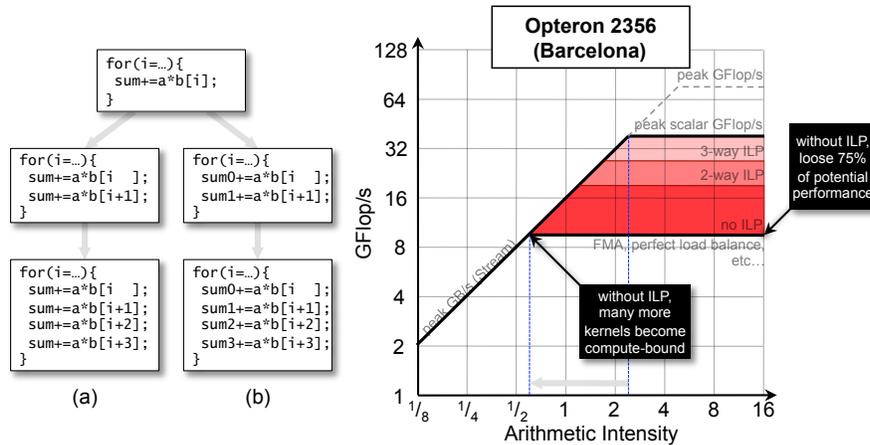
## 2.4 In-Core Ceilings

Given the complexity of modern core architectures, floating-point performance simply doesn't fall out based on arithmetic intensity. 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 enumerate some of the common ceilings. Each is examined in isolation. All code examples assume an x86 architecture.

### 2.4.1 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 [2].

As an example, consider Figure 2.5. In this case we plot scalar floating-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 ILP ceilings (in red). Figure 2.5(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 2.5(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 2.5(b) should substantially outperform (a).



**FIGURE 2.5:** Performance ceilings as a result of insufficient instruction-level parallelism. (a) a code snippet in which the loop is unrolled. note, (FP add) ILP remains constant. (b) code snippet in which partial sums are computed. ILP increases with unrolling.

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.

#### 2.4.2 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 execu-

tion 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 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.

### 2.4.3 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 2.6. The code is a simplified version of that in Figure 2.5(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 discussed in the previous subsection, we cannot achieve 73.6 due to the fact that this code does not perform any floating-point multiplies.

### 2.4.4 Hardware Multithreading

Hardware multithreading [5] 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 to cover the demanded instruction-level parallelism. Moreover, the exemplar of this architecture, Sun's Niagara [8], doesn't implement SIMD or heterogeneous functional units. However, a different set

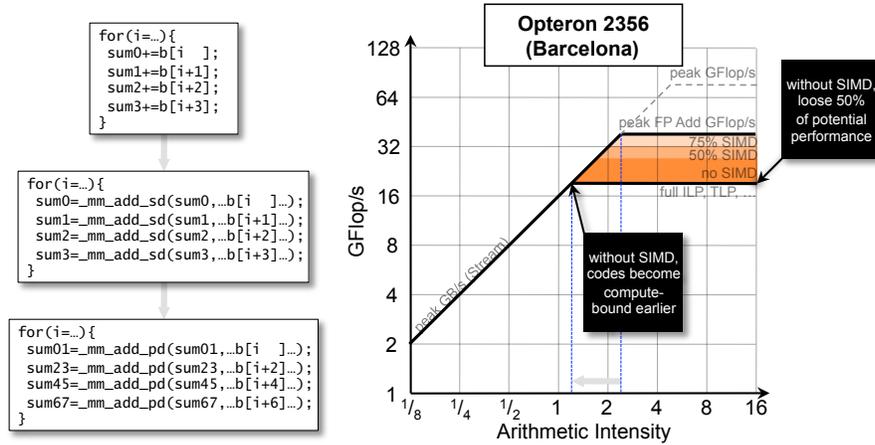


FIGURE 2.6: Example of Ceilings associated with data-level parallelism.

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 ceilings but may invariably see substantial performance degradation due to DLP and heterogenous functional unit ceilings.

### 2.4.5 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 dividing by the total number of threads. The ra-

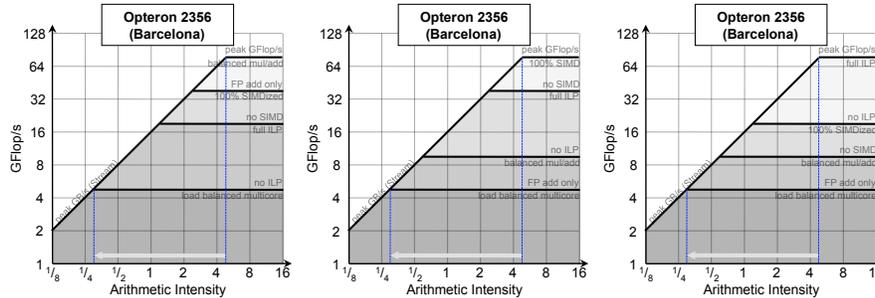


FIGURE 2.7: Equivalence of Roofline models

tio 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.

#### 2.4.6 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 2.7.

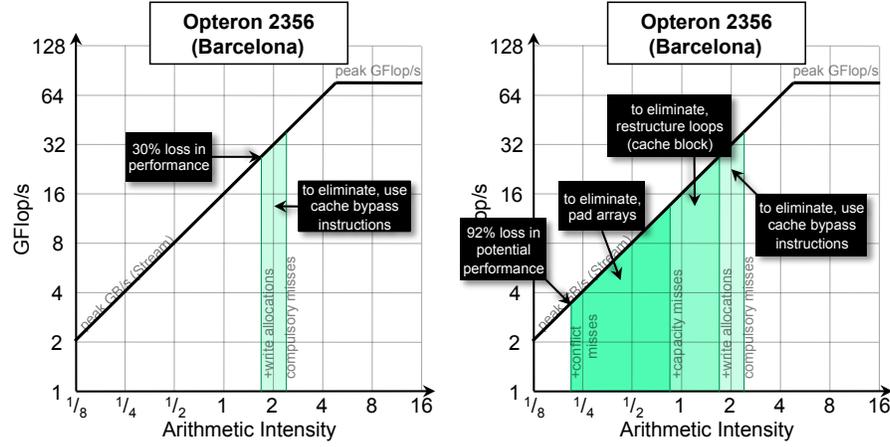
However, the question of how ceilings should be ordered arises. Often, one uses intuition to order the ceilings based on which are most likely to be implicit in the algorithm or discovered by the compiler. Ceilings placed near the roofline are those that are not present in the algorithm or unlikely to be discovered by the compiler. As such, based on this intuition, one could adopt any of the three equivalent roofline models in Figure 2.7.

## 2.5 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 [6]. Unfortunately, on real codes there are a number of other significant terms in the arithmetic intensity denominator.

$$AI = \frac{\text{Total FP Operations}}{\text{Compulsory} + \text{Allocation} + \text{Capacity} + \text{Conflict Memory Traffic} + \dots} \quad (2.3)$$

In much the same way one denotes ceilings to express a lack of instruction,



**FIGURE 2.8:** Performance interplay between Arithmetic Intensity and the Roofline for two different problem sizes for the same nondescript kernel.

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 2.3 shows that write allocation traffic, capacity cache misses, 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 2.8. As capacity and conflict misses are heavily dependent on whether the specified problem size exceed 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.

### 2.5.1 Compulsory Miss traffic

It should be noted that compulsory traffic may not be 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 [10].

### 2.5.2 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 be diminished 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.

### 2.5.3 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 an 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 advantage: 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.

### 2.5.4 Conflict Miss Traffic

Similarly, unlike local stores, caches are not fully associative. That is, depending on address, only certain locations in the cache may be 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 cognizant 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.

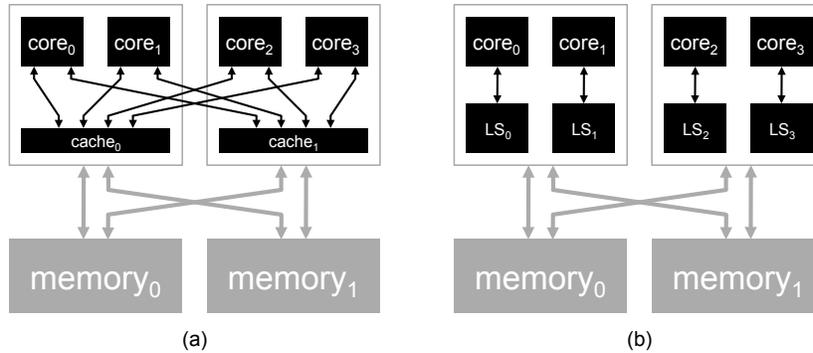
### 2.5.5 Minimum Memory Quanta

Naïvely, one could simply 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.

### 2.5.6 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 possibility 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 performance. 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.



**FIGURE 2.9:** Refinement of the previous simple bandwidth-processor model to incorporate caches or local stores. Remember, arrows denote the ability to access information and not necessarily hardware connectivity.

## 2.6 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.

### 2.6.1 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 2.1 were in fact dual-core processors, one could construct several different hierarchical models (Figure 2.9) depending on the cache/local store topology. Figure 2.9(a) shows it is possible for  $\text{core}_0$  to read from  $\text{cache}_3$  (simple cache coherency), but on the local store architecture, although any core can read from any DRAM location,  $\text{core}_0$  can only read  $\text{LocalStore}_0$ .

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).

### 2.6.2 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.

---

## 2.7 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.

---

## 2.8 Acknowledgements

I wish to express my 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).

---

## 2.9 Glossary

**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 transferred to the computational units.

**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, exponents or trigonometric functions as these are typically decomposed into the base floating-point operations.

**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.

**Traffic:** or communication is the volume of data that must be transferred 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.

Part V

**Automatic Performance  
Tuning**



# Chapter 5

---

## *Auto-tuning Memory-Intensive Kernels for Multicore*

**Samuel W. Williams**

*Lawrence Berkeley National Laboratory*

**Kaushik Datta**

*University of California at Berkeley*

**Leonid Oliker**

*Lawrence Berkeley National Laboratory*

*more authors???*

*from somewhere??*

5.1	Introduction .....	34
5.2	Experimental Setup .....	35
5.2.1	AMD Opteron 2356 (Barcelona) .....	35
5.2.2	Xeon E5345 (Clovertown) .....	36
5.2.3	IBM Blue Gene/P (Compute Node) .....	36
5.3	Computational Kernels .....	37
5.3.1	Laplacian Differential Operator (Stencil) .....	37
5.3.2	Lattice Boltzmann Magnetohydrodynamics (LBMHD) .....	38
5.3.3	Sparse Matrix-Vector Multiplication (SpMV) .....	39
5.4	Optimizing Performance .....	41
5.4.1	Parallelism .....	42
5.4.2	Minimizing Memory Traffic .....	42
5.4.3	Maximizing Memory Bandwidth .....	45
5.4.4	Maximizing In-core Performance .....	46
5.4.5	Interplay between Benefit and Implementation .....	46
5.5	Automatic Performance Tuning .....	46
5.5.1	Code Generation .....	47
5.5.2	Auto-tuning Benchmark .....	48
5.5.3	Search Strategies .....	49
5.6	Results .....	50
5.6.1	Laplacian Stencil .....	50
5.6.2	Lattice Boltzmann Magnetohydrodynamics (LBMHD) .....	51
5.6.3	Sparse Matrix-Vector Multiplication (SpMV) .....	52
5.7	Summary .....	54
5.8	Acknowledgments .....	54

In this, chapter, we discuss the optimization of three memory-intensive computational kernels (sparse matrix vector multiplication, the Laplacian differential operator applied to structured grids, and the `collision()` operator with the lattice Boltzmann magnetohydrodynamics (LBMHD) application. They are all implemented using a single process, (POSIX) threaded, SPMD model. Unlike their computationally-intense dense linear algebra cousins, performance is ultimately limited by DRAM memory bandwidth and the volume of data that must be transferred. To provide performance portability across current and future multicore architectures, we utilize automatic performance tuning, or auto-tuning.

The chapter is organized as follows. First, we define the memory-intensive regime and detail the machines throughout this chapter. Next, we discuss the three memory-intensive kernels that we auto-tuned. We then proceed with a discussion of performance optimization and automatic performance tuning. Finally, we show and discuss the benefits of applying the auto-tuning technique to three memory-intensive kernels.

---

## 5.1 Introduction

Arithmetic Intensity is a particularly valuable metric in predicting the performance of many single program multiple data (SPMD) kernels. It is defined as the ratio of requisite floating-point operations to total DRAM memory traffic. Often, on cache-based architectures, one simplifies total DRAM memory traffic to include just compulsory reads, write allocates, and compulsory writes.

Memory-intensive computational kernels are characterized by those kernels with arithmetic intensities that are constant or scale slowly with data size. For example, BLAS-1 operations like a dot product of two  $N$ -element vectors perform  $2 \cdot N$  floating-point operations, but must transfer  $2 \cdot 8N$  bytes. This results in an arithmetic intensity ( $\frac{1}{8}$ ) that does not depend on the size of the vectors. As this arithmetic intensity is substantially lower than most machines' flop:byte ratio, one generally expects such kernels to be memory-bound for any moderately-large vector size with performance, measured in floating-point operations per second (GFlop/s), being the product of memory bandwidth and arithmetic intensity. Even computational kernels whose arithmetic intensity scales slowly with problem size like out-of-place complex-complex FFT's, roughly  $0.16 \log(n)$ , may be memory-bound for any practical size of  $n$ .

Unfortunately, arithmetic intensity (and thus performance) can be degraded if superfluous memory traffic exists (*e.g.* conflict misses, capacity misses, speculative traffic, or write allocations). The foremost goal in optimizing memory-intensive kernels is to eliminate as much of this superfluous

<b>Core Architecture</b>	<b>AMD Barcelona</b>	<b>Intel Core2</b>	<b>IBM PowerPC 450</b>
Type	superscalar out-of-order	superscalar out-of-order	dual issue in-order
Clock (GHz)	2.3	2.66	0.85
DP Peak (GFlop/s)	9.2	10.7	3.4
Private L1 Data Cache	64 KB	32 KB	32 KB
Private L2 Data Cache	512 KB	—	—

<b>Socket Architecture</b>	<b>Opteron 2356 Barcelona</b>	<b>Xeon E5355 Clovertown</b>	<b>Blue Gene/P Compute Chip</b>
Cores per Socket	4	4 (MCM)	4
Shared Cache	2 MB L3	2×4 MB L2	8 MB L2
memory parallelism paradigm	HW prefetch	HW prefetch	HW prefetch

<b>System Architecture</b>	<b>Opteron 2356 Barcelona</b>	<b>Xeon E5355 Clovertown</b>	<b>Blue Gene/P Compute Node</b>
Sockets per SMP	2	2	1
DP Peak (GFlop/s)	73.69	85.33	13.60
DRAM Bandwidth (GB/s)	21.33	21.33(read) 10.66(write)	13.60
DP Flop:Byte Ratio	3.45	2.66	1.00

TABLE 5.1: Architectural summary of evaluated platforms.

memory traffic as possible. To that end, we may exploit a number of strategies that either passively or actively elicit better memory subsystem performance. Ultimately, when performance is limited by compulsory memory traffic, reorganization of data structures or algorithms is necessary.

## 5.2 Experimental Setup

In this section, we discuss the three multicore SMP computers used in this chapter — AMD’s Opteron 2356 (Barcelona), Intel’s Xeon E5345 Clovertown, and IBM’s Blue Gene/P (used exclusively in SMP mode). As of 2009, these architecture’s dominate the top500 list of supercomputers. The key features of these computers are shown in Table 5.1 and detailed in the following subsections.

### 5.2.1 AMD Opteron 2356 (Barcelona)

Although superseded by the more recent Shanghai and Istanbul incarnations, the Opteron 2356 (Barcelona) effectively represents the future x86 core and system architecture. The machine used in this work is a 2.3 GHz dual-socket  $\times$  quad-core SMP. As each superscalar out-of-order core may complete both a SIMD floating-point add and a SIMD floating-point multiply per cycle, the peak double-precision floating-point performance (assuming balance between adds and multiplies) is 73.6 GFlop/s. Each core has a private 64 KB L1 data cache and a private 512 KB L2 victim cache. The four cores on a socket share a 2 MB L3 cache.

Unlike Intel's older Xeon's, the Opteron integrates the memory controllers on chip and provides an inter-socket network (via Hypertransport) to provide cache coherency as well as direct access to remote memory. This machine uses DDR2-667 DIMMs providing a DRAM pin bandwidth of 10.66 GB/s per socket.

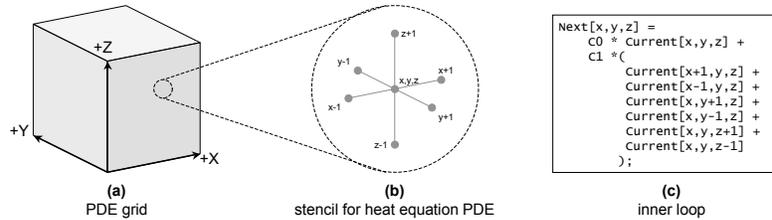
### 5.2.2 Xeon E5345 (Clovertown)

Providing an interesting comparison to Barcelona, the Xeon E5345 (Clovertown) uses a modern superscalar out-of-order core architecture coupled with an older frontside bus (FSB) architecture in which two multichip modules (MCM) are connected with an external memory controller hub (MCH) via two frontside buses. Although two FSBs allows a higher bus frequency, as these chips are regimented into a cache coherent SMP, each memory transaction on one bus requires the MCH to produce a coherency transaction on the other. In effect this eliminates the parallelism advantage in having two FSBs. To rectify this, a snoop filter was instantiated within the MCH to safely eliminate as much coherency traffic as possible. Regardless, the limited FSB bandwidth (10.66 GB/s) bottlenecks the substantial DRAM read bandwidth of 21.33 GB/s.

Each core runs at 2.4 GHz, has a private 32 KB L1 data cache, and like the Opteron may complete one SIMD floating-point add and one SIMD floating-point multiply per cycle. Unlike the Opteron, the two cores on a chip share a 4 MB L2 and may only communicate with the other two cores of this nominal quad-core MCM via the frontside bus.

### 5.2.3 IBM Blue Gene/P (Compute Node)

IBM's Blue Gene/P (BGP) takes a radically different approach to ultra-scale system performance compared to traditional superscalar processors as it relies more heavily on power efficiency to deliver strength in numbers instead of maximizing performance per node. To that end, the compute node instantiates four PowerPC 450 embedded cores in its one chip. These cores are dual-issue, in-order, SIMD enabled cores that run at a mere 850 MHz. As



**FIGURE 5.1:** Visualization of the data structures associated with the heat equation stencil. (a) the 3D temperature grid. (b) the stencil operator performed at each point in the grid. (c) pseudocode for stencil operator.

such, each node’s peak performance is only 13.6 GFlop/s — a far cry from the x86 superscalar performance. However, the order of magnitude reduction in node power results in superior power efficiency.

Each of the four cores on a BGP compute chip has a highly associative 32 KB L1 data cache, and they collectively share an 8 MB L3. As it is a single chip solution, cache-coherency is substantially simpler as all snoops and probes are on chip. The chip has two 128-bit DDR2-425 DRAM channels providing 13.6 GB/s of bandwidth to a mere 4 GB of DRAM capacity. Like Opterons and Xeons, Blue Gene/P has hardware prefetch capabilities.

## 5.3 Computational Kernels

In this section, we introduce the three memory-intensive kernels used throughout the rest of the chapter: the Laplacian stencil, the `collision()–stream()` operators extracted from Lattice Boltzmann Magnetohydrodynamics (LBMHD), and sparse matrix-vector multiplication (SpMV).

### 5.3.1 Laplacian Differential Operator (Stencil)

Partial differential equation (PDE) solvers constitute a large fraction of scientific applications in such diverse areas as heat diffusion, electromagnetics, and fluid dynamics. Solutions to these problems are often implemented using an explicit method via iterative finite-difference techniques that sweep over a spatial grid performing linear combinations of a point’s nearest neighbor in a computation called a *stencil*. As such, our first kernel is the quintessential finite difference operator found in many partial difference equations — the 7-point Laplacian stencil.

This kernel is implemented as out-of-place Laplacian stencil and is visualized by Figure 5.1. As this uses Jacobi’s method, we maintain a copy of

the grid for both the current and next time steps and thereby avoid any data hazards. Conceptually, the stencil operator in Figure 5.1(b) is simultaneously applied to every point in the  $256^3$  scalar grid shown in Figure 5.1(a). This method allows an implementation to select any traversal of the points.

This kernel is exemplified by an interesting memory access pattern with 7 reads and one write presented to the cache hierarchy. However, there is possibility of 6-fold reuse of the read data. Unfortunately this requires substantial cache capacity. Much of the auto-tuning effort for this kernel is aimed at eliciting this ideal cache utilization through the elimination of cache capacity misses. Secondary efforts are geared toward the elimination of conflict misses and write allocation traffic. Thus, with appropriate optimization, memory bandwidth and compulsory memory traffic provide the ultimate performance impediment. To that end, in-core performance must be improved through various techniques only to the point where it is not the bottleneck. For further details on the heat equation and auto-tuning approaches, we direct the reader to [?].

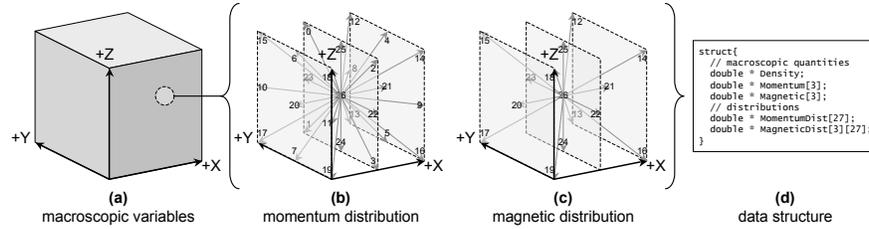
### 5.3.2 Lattice Boltzmann Magnetohydrodynamics (LBMHD)

The second kernel examined in this chapter is the inner loop from the Lattice Boltzmann Magnetohydrodynamics (LBMHD) application [?]. LBMHD was developed to study homogeneous isotropic turbulence in dissipative magnetohydrodynamics (MHD) — the theory pertaining to the macroscopic behavior of electrically conducting fluids interacting with a magnetic field. The study of MHD turbulence is important in the physics of stellar phenomena, accretion discs, interstellar and intergalactic media, and plasma instabilities in magnetic fusion devices [?].

In Lattice methods, the macroscopic quantities (like density or momentum) at each point in space are reconstructed through operations on a momentum lattice — a discretization of momentum along 27 vectors. As LBMHD couples computational fluid dynamics with Maxwell’s equations, the momentum lattice is augmented with a 15 velocity (cartesian vectors) magnetic lattice as shown in Figure 5.2. Clearly, this creates very high memory capacity requirements — over 1 KB per point in space.

LBM methods iterate through time calling two functions per time step: a `collision()` operator, where the grid is evolved one timestep, and a `stream()` operator that exchanges data with neighboring processors. In a shared memory, threaded implementation, `stream()` degenerates into a function designed to maintain periodic boundary conditions.

In serial implementations, `collision()` typically dominates the run time. To ensure that an auto-tuned `collision()` continues to dominate runtime in a threaded environment, we also thread-parallelize `stream()`. We restrict our exploration to a  $128^3$  problem on the x86 architectures, but only  $64^3$  on BlueGene as it lacks sufficient DRAM. For further details on LBMHD



**FIGURE 5.2:** Visualization of the datastructures associated with LBMHD. (a) the 3D macroscopic grid. (b) the D3Q27 momentum scalar velocities. (c) D3Q15 magnetic vector velocities. (d) C structure of arrays datastructure. Note, each pointer refers to a  $N^3$  grid, and  $X$  is the unit stride dimension.

and previous auto-tuning approaches, we direct the reader to the following papers [?, ?].

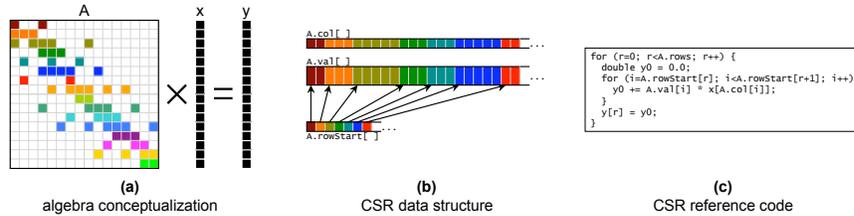
The `collision()` code is far too complex to duplicate here. Superficially, the `collision()` operator must read the lattice velocities from the current time step, reconstruct the macroscopic quantities of momentum, magnetic field, and density, and create the lattice velocities for the next time step. When distilled, this involves reading 73 doubles, performing 1300 floating point operations, and writing 79 doubles per lattice update. This results in a compulsory-limited arithmetic intensity of about 0.7 on write allocate architectures, but may be improved to about 1.07 through the use of cache bypass instructions.

Conceptually, the `collision()` operator within LBMHD is comprised both a 15 and a 27 point stencil similar to the previously discussed Laplacian Stencil. However, as lattice methods utilize an auxiliary grid which store a distribution of velocities at each point, these stencil operators are different in that they reference a different velocity from each neighbor. As such, there is no inter-lattice update reuse. Proper selection of data layout (structure-of-arrays) transformed the principal problem challenge from cache blocking to TLB blocking. When coupled with a code transformation, one may reap the benefits of good cache and TLB locality simultaneously with effective SIMDization.

As this application was designed for a weak-scaled MPI SPMD environment, we may simply tune to optimize single node performance, then integrate the resultant optimized implementation back into the MPI version.

### 5.3.3 Sparse Matrix-Vector Multiplication (SpMV)

Sparse Matrix Vector Multiplication (SpMV) dominates the performance of diverse applications in scientific and engineering computing, economic modeling and information retrieval; yet, conventional implementations have historically performed poorly on single-core cache-based microprocessor sys-



**FIGURE 5.3:** Sparse Matrix Vector Multiplication (SpMV). (a) visualization of the algebra:  $y \leftarrow Ax$ , where  $A$  is a sparse matrix. (b) Standard compressed sparse row (CSR) representation of the matrix. This structure of arrays implementation is favored on most architectures. (c) The standard implementation of SpMV for a matrix stored in CSR. The outer loop is trivially parallelized without any data dependencies.

tems [10]. Compared to dense linear algebra kernels, sparse kernels like SpMV suffer from high instruction and storage overhead per floating-point operations, and a lack of instruction- and data-level parallelism in the reference implementations. Even worse, unlike the implicit (arithmetic) addressing possible in dense linear algebra and structured grid calculations (stencils and lattice methods), indexing neighboring points in a sparse matrix requires an indirect access. This can result in potentially irregular memory access patterns (jumps and discontinuities). As such, achieving good performance on these kernels often requires selection of a compact data structure, reordering of the computations to favor regular memory access patterns, and code transformations based on runtime knowledge of the sparse matrix. This need for run-time optimization and tuning is a major distinction from most other computational methods.

In this chapter, we consider the Sparse Matrix-Vector Multiplication (SpMV) operation  $y \leftarrow Ax$ , where  $A$  is a sparse matrix, and  $x, y$  are dense vectors. A sparse matrix is a special case of the matrices found in linear algebra in which most of the matrix entries are zero. In a matrix-vector multiplication, computation on zeros does not change the result. As such, they may be eliminated from both the representation and the computation leaving only the *nonzeros*. Although the most common data structure used to store a sparse matrix for SpMV-heavy computations is compressed sparse row (CSR) format, illustrated with the corresponding kernel in Figure 5.3, we will explore alternate representations of the compute kernel. CSR requires a minimum overhead of 4 bytes (column index) per 8 byte nonzero. As microprocessors only have sufficient cache capacity to cache the vectors in their entirety, we may define the compulsory memory traffic as 12 bytes per nonzero. SpMV will perform 2 flops per nonzero. As such, the ideal CSR arithmetic is only 0.166 flops per byte; making SpMV heavily memory-bound. Capacity misses and sub-optimal bandwidth will substantially impair performance.

	Dense	Protein	Spheres	Cantilever	Wind Tunnel	Harbor	QCD
Spyplot (Sparsity)							
Rows	2K	36K	83K	62K	218K	47K	49K
Columns	2K	36K	83K	62K	218K	47K	49K
nonzeros (NNZ)	4.0M	4.3M	6.0M	4.0M	11.6M	2.4M	1.9M

	Ship	Economics	Epidemiology	Accelerator	Circuit	webbase	LP
Spyplot (Sparsity)							
Rows	141K	207K	526K	121K	171K	1M	4K
Columns	141K	207K	526K	121K	171K	1M	1M
nonzeros (NNZ)	4.0M	1.3M	2.1M	2.6M	0.9M	3.1M	11.3M

**FIGURE 5.4:** Benchmark matrices used as inputs to our auto-tuned SpMV library framework.

Unlike most of dense linear algebra, stencils on structured grids, and Fourier transforms, where the problems are often characterized by a few integers representing the dimensions of the input, matrices used in sparse linear algebra are not only characterized by their dimensions but also by their *sparsity* pattern — a scatter plot of nonzeros. Figure 5.4 presents the spyplot and the key characteristics associated with each matrix used in this chapter. Observe that for the most part, the vectors are small, but the matrices (in terms of nonzeros) are large. Remember, 12 bytes are required per nonzero. As such, a matrix with four million nonzeros requires at least 32 MB of storage — far larger than most caches. We selected a set of matrices that would exhibit several classes of sparsity: dense, low bandwidth (principally FEM), unstructured, and extreme aspect ratio. Such matrices will see differing cache capacity issues on multicore SMPs. In addition, we ensured the matrices would run the gambit of nonzeros per row — a key component in CSR performance. Finally, although some matrices are symmetric, we convert all of them to non-symmetric format and do not exploit this characteristic.

For further details on the sparse matrix-vector multiplication and previous auto-tuning efforts, we direct the reader to [?, ?].

## 5.4 Optimizing Performance

Broadly speaking, we may either classify optimizations by their impact on implementation and usage, or by the performance bottleneck they attempt to eliminate. For example, an implementation-based categorization may delineate optimizations into four groups based on what changes are required: only code structure, data structures, the style of parallelism, or algorithms. On the other hand, if we categorize optimizations by bottleneck, we may create groups that: more efficiently exploit parallelism, minimize memory traffic, maximize memory bandwidth, or maximize in-core performance. That being said, in this section, we describe the optimizations employed by our three auto-tuners grouped by the bottleneck-oriented taxonomy. Moreover, as we're focused on memory-intensive kernels, we will prioritize the optimizations accordingly.

### 5.4.1 Parallelism

Broadly speaking parallelism encompasses approaches to synchronization, communication, use of threads or processes, and problem decomposition.

**Synchronization and Communication:** Although a number of alternate strategies are possible (including DAG-based schedulers *??*), we adopted a POSIX thread-based, SPMD, bulk-synchronous approach to exploiting multicore parallelism. Unlike process-based, shared memory-optimized message passing approaches, we exploit the ever-present cache coherency mechanisms for both efficient communication as well as to eliminate system calls. We enforce bulk synchronous semantics via a shared memory spin barrier.

**Problem Decomposition:** We utilize two different approaches to problem decomposition. First, the structured grid codes spatially decompose the stencil sweep into subdomains by partitioning the problem in two dimensions (not the unit stride). We ensure there are at least as many subdomains as there are threads. Subdomains may then be assigned to threads in chunks in a round-robin ordering. For LBMHD, the subdomains are not perfect rectangular volumes. Rather, within each plane the boundaries are aligned to cache lines. In effect this performs only loop parallelization through blocking. No part of the data structure is changed.

Conversely, we apply a very different technique when parallelizing sparse matrix-vector multiplication. To ensure there are no data dependencies, we only parallelized by rows, creating submatrices each of which contain roughly the same number of nonzeros, but may span wildly different numbers of rows. Each of these submatrices is stored separately as if it were its own matrix. We may now individually optimize each submatrix including uniquely register blocking each. Clearly, SpMV went well beyond simple loop parallelization and subsumes data structure transformations as well.

### 5.4.2 Minimizing Memory Traffic

When a kernel is memory-bound, there are two principal optimizations: reduce the volume of memory traffic or increase the attained memory bandwidth. For simple memory access patterns, modern superscalar processors often achieve a high fraction of memory bandwidth. As such, our primary focus should be on techniques that minimize the volume of memory traffic. Broadly speaking, we may classify memory traffic using the Three C's *??* cache model's compulsory, conflict, and capacity misses augmented with speculative (prefetch) and write allocate traffic. We implemented a set of optimizations that attempt to minimize each class of traffic. Not all optimizations are applicable to all kernels.

**Array Padding:** Caches have limited associativity. When too many memory references map to the same set in the cache, a conflict miss will occur and useful data will be evicted. These conflicts may arise from intra-thread conflicts or, when shared caches are in play, from inter-thread conflicts. Due to the limited number of memory streams and reuse, inter-thread conflict misses predominate on SpMV and the Laplacian stencil. On LBMHD, where the `collision()` operator attempts to keep elements from 150 different arrays in the cache, eliminating intra-thread conflict misses is key. As such, we implemented two different strategies to mitigate cache conflict misses. For SpMV and stencils, we pad each array so that the address of the first element maps to a unique set in the cache. Moreover, the padding is selected to ensure that the set addresses of the threads' first elements are equally spaced (by set address) in the last level cache. The ideal padding may be either calculated arithmetically or obtained experimentally. For SpMV, we simply `malloc` each thread block independently with enough space to pad by the cache size. We then align to a 4 MB boundary and pad by the thread's fraction of the cache. Array padding for LBMHD is somewhat more complex. We pad each velocity's array so that when referenced with the corresponding stencil offset (and corresponding address offset) the resultant physical address maps to a unique, equally-spaced cache set. Although this sounds complicated in practice, it's relatively easy to implement. For the details of how these kernels exploit array padding, we direct the reader to *???*

**Cache Blocking:** The reference implementations of many kernels maintain substantial cache working sets. In practice, processor architects cannot implement caches that are large enough to avoid capacity misses adding to the volume of memory traffic. LBMHD does not exhibit any inter-stencil reuse. That is, there is no two stencils reuse the same values. As such, cache capacity misses are nonexistent. However, the Laplacian stencil shows substantial reuse. Like dense matrix-vector multiplication, SpMV will also show reuse on vector accesses. In either case, we must restructure code, and possibly data, to eliminate capacity miss traffic. Like cache blocking in dense linear algebra, we may apply a simple loop blocking technique to the Laplacian stencil to ensure an implementation generates relatively few capacity misses. In practice, this is

implemented the same as problem decomposition for parallelization. However, defining dense blocks (of source vectors) for SpMV often yields a dramatically suboptimal solution. As such, we employ a novel sparse blocking technique that ensures that each cache block touches the same number of cache lines regardless of how many rows or columns the block spans. In practice, for a given thread's block of the matrix, we cache block it by simply adding columns of the sparse matrix until the number of unique cache lines touched reaches a preset number. Clearly, this requires substantial data structure changes.

**Cache Bypass:** Based on consumer applications, most cache architectures implement a *write allocate* protocol. That is, if a store (write) misses in the cache, an existing line will be selected for eviction, the target line will be loaded into the cache, and the target word will be written to the line in the cache. Such an approach is based on the implicit assumption that if data is written, it will be promptly read and modified many times. Note, usage of such a policy is orthogonal to the *write-back* or *write-through* choice. Unfortunately, many computational kernels found in HPC read and write to separate arrays or data structures. As such, most writes allocate a line in the cache, completely obliterate its previous contents, and eventually write it back to DRAM. This makes write allocate not only superfluous, but expensive as it will generate twice the memory traffic as a read — an obvious target for optimization when memory-bound.

Modern write-allocate cache architectures provide a means of eliminating this superfluous memory traffic via a special store instruction that bypasses the cache hierarchy. In the x86 ISA, this is implemented with the `movntpd` instruction. Unfortunately, most compilers cannot resolve the complex decision as to when to use this instruction; improper usage can reduce performance by an order of magnitude, where correct usage can improve performance by 50%. As such, in practice, we may only exploit this functionality through the use of SIMD *intrinsics* — a language construct with the interface of a function that the compiler will map directly to one instruction.

Often, the vectors used in SpMV are small enough to fit in cache. As such, the totality of DRAM memory traffic is reads and there is no need to use cache bypass. However, Jacobi stencils and lattice methods read and write to separate arrays. For other finite-difference operators like gradient or divergence, cache bypass may only improve performance by 75% or 25% respectively. Given this variability in benefit and the human effort required to implement this optimization, one should analyze the code before proceeding with this optimization. Nevertheless, usage of cache bypass on the Laplacian stencil or LBMHD can reduce the total memory traffic by 33% and improve performance by 50%.

**Register blocking for SpMV:** For most matrices, SpMV is dominated by compulsory misses. As such, neither cache blocking nor cache bypass will provide substantial benefits. That is not to say nothing can be done. Rather, a radical solution has emerged that eliminates compulsory miss traffic. Broadly speaking, sparse matrices require substantial meta data per nonzero — per-

haps a 50% overhead. However, we observe that many nonzeros are clustered in relatively small regions. As such, the optimization known as *register blocking* reorganizes the sparse matrix of nonzeros into a sparse matrix of small  $R \times C$  dense matrices. Meta data is now needed only for each register block rather than each nonzero. If the zero fill required to make those  $R \times C$  register blocks dense is less than the reduction in meta data, then the total memory traffic has been reduced — a clear win for a memory-bound kernel. Similarly, we may note that a range of column or row indices can be represented by a 16-bit integer instead of a 32-bit integer. This can save 2 bytes (or 17%) per nonzero. Please note, the term register blocking, when applied to sparse linear algebra refers to a hierarchical restructuring of the data, but when applied to dense linear algebra refers to a unroll and jam technique. In this chapter, our SpMV code heuristically explores these matrix compression techniques to find the combination that minimizes the matrix footprint.

### 5.4.3 Maximizing Memory Bandwidth

Now that we've discussed optimizations designed to minimize the volume of memory traffic, we may examine optimizations that maximize the rate at which said volume of data can be streamed into the processor. Basically, these optimizations aim to either avoiding memory latency or hide memory latency..

**TLB Blocking:** All modern microprocessors use virtual memory. To translate the virtual address produced by the program's execution into the physical address required to access the cache or DRAM, the processor must inspect the page table to determine the mapping. As this is a slow process and page table entries rarely change, page table entries may be placed in a very fast, specialized cache on chip — the *translation lookaside buffer* or TLB. Unfortunately, TLBs are small and thus may not be able to cache all the pages referenced by an application (regardless of page size). As such, it is possible to generate TLB capacity misses. These typically don't generate superfluous DRAM traffic like normal cache capacity misses because evicted page table entries may land in the L2 or L3 cache. The performance difference (resulting from an increase in average memory latency) between translations that hit in the TLB and those that hit in the L3 is substantial. We may recast the cache blocking technique (which eliminated cache capacity misses) to eliminate TLB capacity misses and avoid memory latency. In LBMHD, we used a loop interchange technique coupled with an auxiliary data structure. This allowed us to trade cache capacity for increased page locality (and reduced TLB capacity misses). This technique is detailed in ???.

**Prefetching:** Memory latency is high. To satisfy Little's Law ??? and maximize memory bandwidth, the processor must express substantial memory-level parallelism. Unfortunately, superscalar execution may be insufficient. As such, hardware designers have incorporated both hardware and software prefetching techniques into their processors. The goal for either is to hide memory latency. A software prefetch is an instruction like a load without

a target address. As such, the processor will not stall waiting for it to complete. The user simply prefetches one element from each cache line to initiate the entire line's load. Unfortunately, such a practice requires the programmer to tune for the optimal "prefetch distance" — how far ahead prefetch addresses should be from load addresses. If he aims too low, latency will not be completely hidden. If he aims too high, cache capacity will be exhausted. More recently, hardware prefetchers have begun to supplant software prefetching. Typically, they detect a series of cache misses, speculate as to future addresses, and prefetch them into the cache without requiring any user interaction.

In this chapter, we structure our auto-tuned codes to synergize with hardware prefetchers (long unit-stride accesses) but supplement this with software prefetching. This general approach provides performance portability as we make no assumptions as to whether a processor implements software prefetching, hardware prefetching, or both.

#### **5.4.4 Maximizing In-core Performance**

For memory-intensive computations our primary focus should be on minimizing memory traffic and maximizing memory bandwidth. However, it is important not to overlook in-core(cache) performance. Code written without thought to the forms of parallelism required to attain good in-core performance may actually be compute-bound rather than memory-bound. The most common techniques are unroll and jam, permuting or reordering the computation given an unrolling, and SIMDization. We explored all of these via auto-tuning on all three kernels.

#### **5.4.5 Interplay between Benefit and Implementation**

The bottleneck that an optimization attempts to alleviate is orthogonal to the scope of the software implementation effort that is required to achieve it. For example, Table 5.2 lists the optimizations used when auto-tuning our three memory-intensive kernels. Loop or code structure transformations have perennially been the only changes allowed by an auto-tuner as they preserve the input and output semantics. Nevertheless, we see many optimizations require an abrogation of this convention as changes to data structures are required for ideal performance.

---

### **5.5 Automatic Performance Tuning**

Given this diversity of computer architectures, performance optimization has become a challenge as optimizing an application for one microarchitecture

Optimization	Loop/Code Structure	Data Structure	Style of Parallelism
BS SPMD (pthreads)			✓
Decomposition (loop-based)	✓		
		✓	
Array Padding		✓	
Cache Blocking (loop-based)	✓		
		✓	
Cache Bypass (movntpd)	✓		
Reg. Blocking (sparse)		✓	
TLB Blocking (loop-based)	✓		
		✓	
Prefetching (software)	✓		
Unroll and Jam	✓		
Reordering	✓		
SIMDization	✓	†	

**TABLE 5.2:** Interplay between the bottleneck each optimization addresses (parallelism, memory traffic, memory bandwidth, in-core performance) and the impact on implementation (code-only, data structures, styles of parallelism). Obviously, changing data or parallelism structure will mandate some code changes. †Efficient SIMD requires data structures be aligned to 128-byte boundaries.

may result in a substantial performance loss on another. When coupled with the demands to optimize performance in a shorter timeframe than architectural evolution (several new variants of the x86 processor lines appear every year), hand optimizing for each is not practical. To that end, automatic performance tuning, or *auto-tuning* has emerged as a productive approach to tune key computational kernels and even full applications in minutes instead of months **ATLAS, FFTW, OSKI???**. In essence, auto-tuning is built on the premise that if one can enumerate all possible implementations of a kernel, the performance of modern computers allows for the exploration of these variants in less time than a human would require to optimize for one. Moreover, once this auto-tuner has been constructed it can be reused on any evolution of these architectures. The best choice or parameterization for the optimizations may be either architecture-dependent, input-dependent, or both. If it is neither, simple optimization will suffice, and auto-tuning is not needed.

Typically, auto-tuning a kernel is divided into three phases: enumeration of potentially valuable optimizations, implementation of a code generator to produce functionally equivalent implementations of said kernel using different combinations of the enumerated optimization space, and implementation of a search component that will benchmark these variants (perhaps using real problem data) in an attempt to find the fastest possible implementation. We may visualize the auto-tuning flow in Figure 5.5, and will discuss the principal components in the following sections.

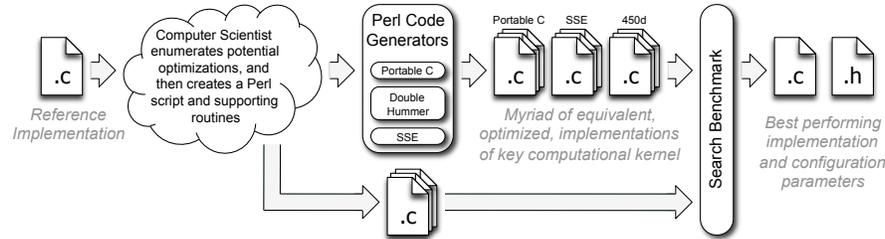


FIGURE 5.5: Generic visualization of the auto-tuning flow.

### 5.5.1 Code Generation

For purposes of this chapter, we use a simple auto-tuning methodology in which we use a Perl script to generate a few hundred potentially viable parameterizable implementations of a particular kernel. An implementation is a unique code representation that may be parameterized with a run time configuration. For example, cache blocking transforms a naïve three nested loop implementation of matrix-matrix multiplication into a six nested loop implementation that is parameterized at runtime with the sizes of the cache blocks (the range of the inner loops). This is still just one variant. However, when one register blocks matrix multiplication, the inner 6 nested loops are so small (less than 16) it is common to simply fully unroll all loops and create perhaps a few thousand different code variants. When combined with cache blocking, we may have hundreds of individually parameterizable code variants.

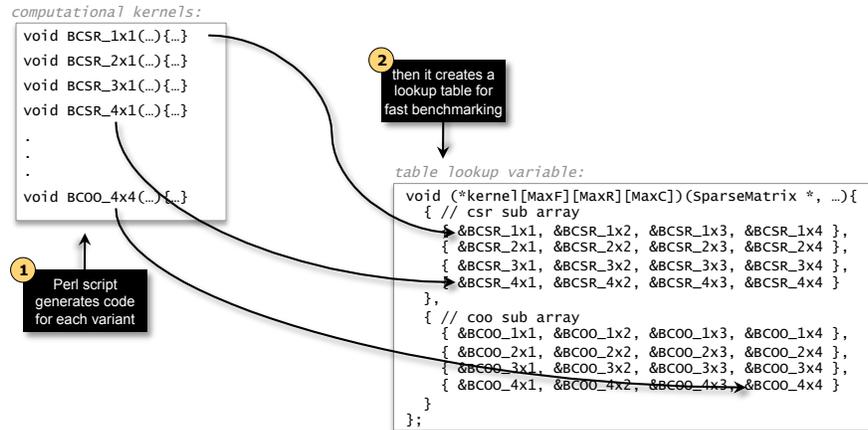
Code variants are also needed when dealing with different data structures (*i.e.* hierarchical instead of flat), styles of parallelism (dataflow instead of bulk synchronous), or even algorithms. Each of these may in turn be parameterized.

As the differences between clusters of code variants may easily be expressed algorithmically, the Perl scripting language provides a pragmatic and productive means of tackling the intellectually-uninspiring task of producing the tens of thousands of lines of code. In essence, the simplest Perl code generation techniques are nothing more than a for loop over a series of `printf`'s. Every line in the resultant C code (function declarations, variables, statements, etc...) maps to a corresponding `printf` in the Perl script.

### 5.5.2 Auto-tuning Benchmark

A Perl script may generate thousands of code variants. Rather than trying to compile an auto-tuning benchmark for each, we integrate and compile all of them into one auto-tuning benchmark. This creates the challenge of selecting the appropriate variant without substantial overhead. To that end, we create an N-dimensional pointer-to-function table indexed by the code variants and possible parameters.

For example, in SpMV we use a 3-Dimensional table indexed by kernel



**FIGURE 5.6:** Using a pointer-to-function table to accelerate auto-tuning search.

type (BCSR, BCOO, etc...) and the register block sizes as measured in rows and columns. As shown in Figure 5.6, the Perl script first generates the code for each kernel variant. In addition, it creates a pointer-to-function table that provides a very fast means of executing any kernel. During execution, one simply calls `kernel[BCSR][4-1][3-1](...)` to execute the  $4 \times 3$  BCSR kernel. The auto-tuning benchmark can be constructed to sweep through all possible formats and register blockings (nested for loops). For each combination, the matrix is reformatted and blocked, and the SpMV is benchmarked through a simple function call via the table lookup. This provides a substantial tuning time advantage over the naïve approach of compiling and executing one benchmark for every possible combination. Moreover, it provides a fast runtime solution as well as easy library integration. We’ve demonstrated this technique when auto-tuning dense linear algebra, sparse linear algebra, stencils, and lattice methods.

### 5.5.3 Search Strategies

Given an auto-tuning benchmark, we must select a traversal of the optimization-parameter space that finds good performance quickly. Over the years, a number of strategies have emerged. In this chapter, we employ three different auto-tuning strategies: exhaustive, greedy, and heuristic. When the optimization-parameter space was small, an exhaustive search implemented as a series of nested loops was acceptably fast. However, in recent years we’ve observed a combinatoric explosion in the size of the space. As a result, exhaustive search is no longer time- and resource-efficient. As a result, a number of new strategies have emerged designed to efficiently search the space **??**. Greedy

algorithms assume *???*. As such, they may transform a  $N^D$  optimization-parameter space of  $D$  optimizations each of  $N$  possible parameters into a sequential search through  $D$  optimizations each of  $N$  parameters ( $N \times D$  points). Often, with substantial architectural intuition, we may express the best (or very close to best) combination in  $O(1)$  time through an arithmetic approach that combines machine parameters and kernel characteristics.

Due to the size of the search space for the Laplacian Stencil, we were forced to perform a greedy search algorithm after ordering the optimizations with some architectural intuition. This reduced the predicted tuning time from three months to 10 minutes. Conversely, LBMHD almost essentially uses an exhaustive strategy across seven code variants each of which could accept over one hundred different parameter combinations. Typical tuning time was less than 30 minutes. SpMV used a combination of heuristics and exhaustive search. The none/cache/TLB blocking variant space was search exhaustively. That is, we benchmarked performance not blocking for either the cache or TLB, blocking for just the cache, and blocking for both the cache and TLB. Unlike the typical dense approach, the parameterization for cache and TLB blocking was obtained heuristically. Similarly, unlike the OSKI *??* approach, the register blocking was obtained heuristically by examining the resultant memory footprint size for each power-of-two register blocking. However, like LBMHD, the prefetch distance was obtained through an exhaustive search.

## 5.6 Results

In this section, we present and discuss the results from the application of three different custom auto-tuners to the three benchmarks used in this chapter. Previous papers have performed a detailed performance analysis for these three kernels *SC08, IPDPS08, SC07*

### 5.6.1 Laplacian Stencil

Figure 5.7 shows the benefits of auto-tuning the 7-point Laplacian stencil on a  $256^3$  grid on our three computers as a function of thread concurrency and increasing optimization. Threads are ordered to fully exploit all the cores within a socket before utilizing the second socket. We have condensed all optimizations into two categories: those that may be expressed in a portable C manner, and those that are ISA-specific. The former is a common code base that may be used on any cache-based architecture, not just these three. The latter includes optimizations like explicit SIMDization, and cache bypass. Thus Barcelona and Clovertown use the same x86 ISA-specific auto-tuner, and BlueGene/P uses a different one. The auto-tuning search strategy uses a

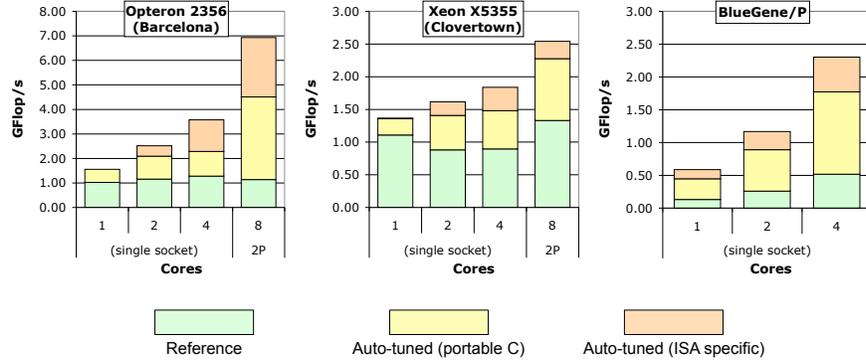


FIGURE 5.7: Benefits of auto-tuning the 7-point Laplacian Stencil.

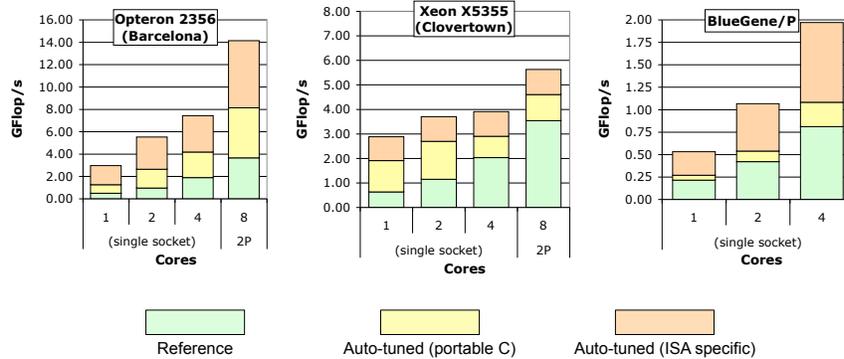
problem size-aware, greedy search algorithm in which the optimizations are searched one a time for the best parameterizations.

Clearly, the reference implementation delivers substantially suboptimal performance. As expected on the bandwidth-starved x86 processors, we see the reference implementation shows no scalability as one core may come close to fully saturating the available memory bandwidth.

When the portable C auto-tuner is applied to this kernel, we see that optimizations like cache blocking dramatically reduce superfluous memory traffic allowing substantially better performance. In general, on the x86 processors, we see a saturation of performance at around 4 cores (one socket), but a jump in performance at 8 cores as using the second socket doubles the useable memory bandwidth. However, on NUMA architectures, like Barcelona, this boost is only possible if data is allocated in a NUMA-aware manner.

Rather than hoping the compiler, the non-portable, ISA-specific auto-tuner explicitly SIMDizes the kernel via intrinsics. Unfortunately, this is only useful on compute-bound platforms like Blue Gene. Unfortunately, despite the simplicity of this kernel, the lack of unaligned SIMD loads in the ISA results in less than perfect ( $2\times$ ) scaling. Although explicit SIMDization was not beneficial on the x86 architectures, a different ISA-specific optimization, cache bypass, was useful as it reduces the memory-traffic on a memory-bound kernel. Doing so can substantially improve performance. Unfortunately, compilers will likely never be able to determine when this instruction is useful as it requires run-time knowledge.

In the end, auto-tuning improved the performance at full concurrency by  $6.1\times$ ,  $1.9\times$ , and  $4.5\times$ , for Barcelona, Clovertown, and Blue Gene/P respectively. Moreover, thread-parallelizing and auto-tuning the 7-point stencil improved performance by  $6.8\times$ ,  $2.3\times$ , and  $17.8\times$ , for Barcelona, Clovertown, and Blue Gene/P respectively. Although substantial effort was required in implementing a x86-specific auto-tuner, it may be reused on all subsequent x86 architectures thus amortizing the up front productivity cost.



**FIGURE 5.8:** Benefits of auto-tuning the lattice Boltzmann Magnetohydrodynamics (LBMHD) application.

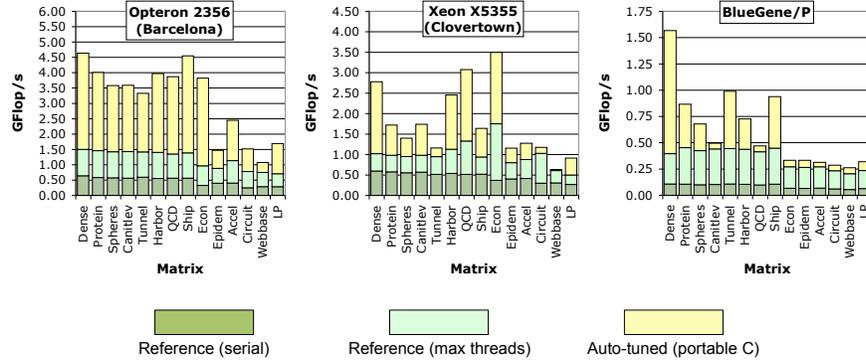
### 5.6.2 Lattice Boltzmann Magnetohydrodynamics (LBMHD)

Figure 5.8 shows the benefits of auto-tuning LBMHD on our three computers as a function of thread concurrency and increasing optimization. Unfortunately, Blue Gene/P does not have enough DRAM to simulate the desired  $128^3$  problem. As such, it only simulates a  $64^3$  problem. Once again, we have condensed all optimizations into two categories: those that may be expressed in a portable C manner, and those that are ISA-specific. The auto-tuning search strategy is exhaustive although vectorization is quantized into cache lines.

Although the reference implementation delivers good scalability, simple performance modeling using the Roofline Model ?? suggests it was delivering substantially suboptimal performance. Such a model also explains why even after auto-tuning the bandwidth-starved Clovertown shows poor scalability despite the performance boosts.

The biggest boosts derived from the portable C auto-tuner are NUMA-aware allocation, lattice-aware array padding, and vectorization (to eliminate TLB capacity misses). The Opteron and Blue Gene/P, with moderate machine balance (flops:bandwidth), see good scaling on this kernel. Conversely, Clovertown, with a high machine balance, sees poor multicore scalability. Whether compute-bound or memory-bound, the non-portable, ISA-specific auto-tuner provided tremendous performance boosts either through explicit SIMDization or cache bypass.

In the end, auto-tuning improved the full concurrency performance by  $3.9\times$ ,  $1.6\times$ , and  $2.4\times$ , for Barcelona, Clovertown, and Blue Gene/P respectively. Moreover, the coupling of thread-parallelization and auto-tuning improved LBMHD performance by an impressive  $28.3\times$ ,  $8.9\times$ , and  $9.2\times$ , for Barcelona, Clovertown, and Blue Gene/P respectively. Clearly, the ISA-specific auto-tuners were critical in achieving these speedups.



**FIGURE 5.9:** Auto-tuning Sparse Matrix-Vector Multiplication. Note, horizontal axis is the matrix (problem) and multicore scalability is not shown.

### 5.6.3 Sparse Matrix-Vector Multiplication (SpMV)

Figure 5.9 shows the benefits of auto-tuning SpMV on our three computers. Unlike the previous two figures, where optimization and benefit is basically independent of problem size, the horizontal axis in Figure 5.9 represents different problems (matrices). The ordering preserves that in Figure 5.4. The lowest bar is the untuned serial performance, while the middle bar represents untuned performance using the maximum number of cores. The top bar is the tuned (portable C) performance using the maximum number of cores. Unlike the other kernels, a non-portable ISA-specific auto-tuner was not implemented for SpMV.

The auto-tuning search strategy is somewhat more complex. Register, cache and TLB blocking use a footprint minimization heuristic based on cache and TLB topology, where prefetching is based on an exhaustive search quantized into cache lines.

We observe a trimodal performance classification: problems where both the vectors and matrix fit in cache, problems where only the vectors can be kept in cache, and problems where neither the matrix nor the vectors can be kept in cache. Clearly, on Barcelona, no matrix ever fits in the relatively small cache, but the performance differences between the problems where the vectors fit can be clearly seen. On Clovertown, where the cache grows from 4 MB to 16 MB using all 8 cores, we can see the three matrices that get a substantial performance boost through utilization of all the cache and compression of the matrices. Blue Gene/P, like Barcelona can never fit any matrix in cache, but we can see the problems where the vectors don't fit — *Economics* through *Linear Programming*.

As it turns out, on Barcelona, NUMA-aware allocation was an essential optimization across all matrices. Across all architectures, matrix compression delivered substantial performance boosts on certain matrices. Interestingly, on

Blue Gene/P, matrix compression improved performance by a degree greater than the reduction in memory traffic — an effect attributable to an initially compute-bound reference implementation. Interestingly, TLB blocking delivered substantial performance boosts on only one matrix, the extreme aspect ratio linear programming problem.

We observe that threading alone provided median speedups of 1.9 $\times$ , 2.5 $\times$ , and 4.2 $\times$  on Barcelona, Clovertown, and Blue Gene/P. Clearly, only Blue Gene/P showed reasonable scalability. However, when coupled with auto-tuning, we observe median speedups of 3.1 $\times$ , 6.3 $\times$ , and 5.1 $\times$  and maximum speedups of 9.5 $\times$ , 11.9 $\times$ , and 14.6 $\times$ . Ultimately, performance is hampered by memory bandwidth on both Barcelona and Clovertown, leading to sublinear scaling. As a result, auto-tuning strategies targeted at reducing memory traffic are critical.

---

## 5.7 Summary

Naïvely, one might expect that nothing can be done to improve the performance of memory-intensive or memory-bound kernels like stencils, LBMHD, or SpMV. However, in this chapter, we discussed a breadth of useful optimizations applicable not only to our three example kernels but to many others domains. Unfortunately, no human could explore all the parameterizations of these optimizations by hand. To that end, we showed how automatic performance tuning, or *auto-tuning*, can productively tune code and thereby dramatically improve performance across the breadth of architectures that currently dominate the top500 list. Unfortunately, to achieve the best performance, non-portable ISA-specific auto-tuners that generate explicitly SIMDized code are required.

---

## 5.8 Acknowledgments

We would like to express our gratitude to Forschungszentrum Jülich for access to their BlueGene machine. This work was supported by the ASCR Office in the DOE Office of Science under contract number DE-AC02-05CH11231, by NSF contract CNS-0325873, and by Microsoft and Intel Funding under award #20080469.

---

## Bibliography

- [1] D. Bailey. Little's Law and High Performance Computing. In *RNR Technical Report*, 1997.
- [2] David Callahan, John Cocke, and Ken Kennedy. Estimating interlock and improving balance for pipelined architectures. *J. Parallel Distrib. Comput.*, 5(4):334–358, 1988.
- [3] Steve Carr and Ken Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Trans. Program. Lang. Syst.*, 16(6):1768–1810, 1994.
- [4] Solaris Memory Placement Optimization and Sun FireServers. <http://www.sun.com/software/solaris/performance.jsp>, March 2003.
- [5] J. L. Hennessy and D. A. Patterson. *Computer Architecture : A Quantitative Approach; fourth edition*. Morgan Kaufmann, San Francisco, 2007.
- [6] M. D. Hill and A. J. Smith. Evaluating Associativity in CPU Caches. *IEEE Trans. Comput.*, 38(12):1612–1630, 1989.
- [7] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance: Computer System Analysis using Queueing Network Models*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.
- [8] S. Phillips. Victoriafalls: Scaling highly-threaded processor cores. In *HotChips 19*, 2007.
- [9] STREAM: Sustainable Memory Bandwidth in High Performance Computers. <http://www.cs.virginia.edu/stream>.
- [10] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A Library of Automatically Tuned Sparse Matrix Kernels. In *Proc. of SciDAC 2005, J. of Physics: Conference Series*. Institute of Physics Publishing, June 2005.
- [11] S. Williams. *Auto-tuning Performance on Multicore Computers*. PhD thesis, EECS Department, University of California, Berkeley, December 2008.
- [12] S. Williams, D. Patterson, L. Oliker, J. Shalf, and K. Yelick. The roofline model: A pedagogical tool for auto-tuning kernels on multicore architectures. In *IEEE HotChips Symposium on High-Performance Chips (HotChips 2008)*, August 2008.
- [13] S. Williams, A. Watterman, and D. Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. *Communications of the ACM*, April 2009.