Jack Dongarra, David A. Bader, Jakub Kurzak

Scientific Computing with Multicore and Accelerators

List of Figures

List of Tables

Contents

Chapter 1

Sparse Matrix-Vector Multiplication on Multicore and Accelerators

Samuel Williams

Lawrence Berkeley National Laboratory

Nathan Bell

NVIDIA Research

Jee Whan Choi

Georgia Institute of Technology

Michael Garland

NVIDIA Research

Leonid Oliker

Lawrence Berkeley National Laboratory

Richard Vuduc

Georgia Institute of Technology

1.1 Introduction

This chapter consolidates recent work on the development of high-performance multicore and accelerator-based implementations of *sparse matrix-vector multiplication* (SpMV). As an object of study, SpMV is an interesting computation for two key reasons. First, it appears widely in applications in scientific and engineering computing, financial and economic modeling, and information retrieval, among others, and is therefore of great practical interest. Secondly, it is both simple to describe but challenging to implement well, since its performance is limited by a variety of factors, including low computational intensity, potentially highly irregular memory access behavior, and a strong input dependence that be known only at run time. Thus, we believe SpMV is both practically important and provides important insights for understanding the algorithmic and implementation principles necessary to making effective use of state-of-the-art systems.

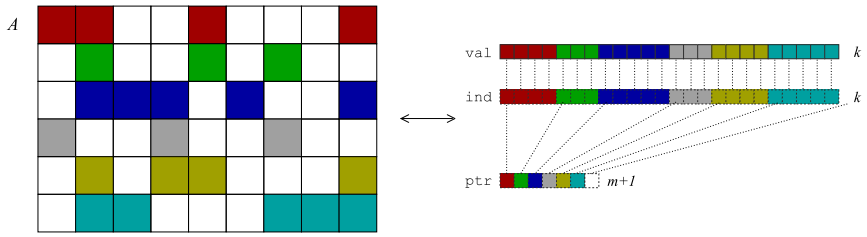
The key findings and results of this chapter are primarily the direct result of three recent publications [5,7,15]. This chapter focuses on synthesizing the main findings from across the three studies, emphasizing high-level design and implementation principles. Some of the data in this chapter are new, as they include recent hardware platforms not available in prior work (*e.g.*, Intel Nehalem, STI PowerXCell 8i, and NVIDIA GeForce GTX 285). However, we also must necessarily omit discussion of some platform-specific implementation details, as well as a more in-depth discussion of some of the research issues explored in the original studies (*e.g.*, autotuning). For such details, we recommend that the interested reader consult the original studies.

1.2 Sparse Matrix-Vector Multiplication: Overview and Intuition

Sparse matrix-vector multiplication (SpMV) operations are of particular importance in computational science. They represent the dominant cost in many *iterative methods* for solving large-scale linear systems and eigenvalue problems which arise in a wide variety of scientific and engineering applications. In the course of solving a sparse linear system $Ax = b$, such methods generally require the computation of hundreds or perhaps thousands of SpMV operations with the matrix A. Sparse matrix-vector multiplication is the foundation of a broad class of solvers, including notable examples such as the conjugate gradients method (CG), the generalized minimum residual method (GMRES), and the biconjugate gradients stabilized method (BiCGstab), among many others [11]. The remaining components of these methods reduce to dense

linear algebra operations that are readily handled by optimized BLAS implementations.

The specific SpMV operation we consider is $y \leftarrow y + Ax$, where A is an $M \times N$ sparse matrix, and x, y are dense vectors. We refer to x as the *source vector* and y as the *destination vector*. By "sparse," we mean that most of the entries of A are zero, and therefore compact representations of A can eliminate unnecessary storage and computation. However, the cost of a sparse representation is a more complex data structure since, unlike the dense case, it must explicitly track which non-zero entries are stored. As an example, Figure 1.1 illustrates the most common sparse matrix representation, called compressed sparse row (CSR) storage, and provides a base-line sequential SpMV implementation.




```
// Basic SpMV implementation,
// y <- y + A*x, where A is in CSR.
 for (i = 0; i < m; ++i) {
   double y0 = y[i];
   for (k = ptr[i]; k < ptr[i+1]; ++k)
     y0 += val[k] * x[ind[k]];
   y[i] = y0;
 }
```

FIGURE 1.1: Compressed sparse row (CSR) storage, and a basic CSR-based SpMV implementation.

To gain some intuition for SpMV performance, we make two observations. First, SpMV requires just two floating point operations (flops) per non-zero entry of A—namely one multiplication and one addition. By comparison, the code given in Figure 1.1 will execute many more instructions on secondary tasks such as integer indexing, resulting in a relatively high overhead. Second, SpMV has relatively little reuse and is memory intensive. Temporal data locality is limited to the accesses of x and y; every element of A is used exactly once. Indeed, a first order estimate on SpMV performance is simply that it will be bounded from below by the time to read A, which in effect amounts to the time required to stream the matrix data structure from memory through the processor. On a modern processor, whose peak computational throughput is

substantially higher than its peak memory bandwidth, we thus expect its computational efficiency to be low and ultimately limited by memory bandwidth. Consequently, much of the effort to optimize SpMV performance focuses on changing loop and data structures to maximize parallelism and reduce non-flop instruction overheads, to regularize memory access patterns, to minimize memory traffic, and to maximize locality. We discuss how this intuition, given current multicore and accelerator architectures described in Section 1.3, informs specific design principles and optimization techniques in Sections 1.4 and 1.5, respectively.

1.3 Architectures, Programming Models, and Matrices

Our SpMV optimizations are driven by the diversity in available hardware architectures, parallel programming models, and input matrices that arise in practice. This section summarizes the main characteristics of these platforms and inputs that influence the process of optimizing and tuning SpMV.

1.3.1 Hardware Architectures

We consider SpMV optimization in the context of three diverse multicore and accelerator platforms: a dual-socket quad-core server based on Intel's Xeon X5550 multicore processor ("Nehalem"); a dual-socket blade based on IBM's QS22 PowerXCell 8i processor; and NVIDIA's GeForce GTX 285 GPU. Table 1.1 summarizes the specifications of these systems.

These architectures differ along several dimensions, perhaps the most significant of which is the programmer's view of the memory system, as summarized by Table 1.2. In particular, we may broadly classify these architectures by the number of levels of memory (address spaces) and the nature or degree of software control. Differences in these design aspects affect how the programmer must (a) orchestrate data movement and (b) restructure the computation to achieve good spatial and temporal locality within each level of memory.

For example, our Intel Nehalem platform, typical of conventional cache-based mulitcore processors, uses an explicit two-level memory hierarchy: DRAM and registers. Programmers (often via compilers) will control the movement of data from DRAM to registers (through explicit loads and stores) and then from registers to functional units (other instructions). Cache hierarchies are often instantiated between DRAM and registers to accelerate performance, largely through automated placement of demand memory accesses within the caches. Automated caching simplifies programming, at the cost of a loss of transparency in when and how data moves through the hierarchy as well as the cost of data movement. For the most part, programmers may mod-

Core Architecture	**Intel Nehalem**	**IBM Cell SPE**	**NVIDIA GT200 SM**
Type	SMT out-of-order	dual-issue in-order	SIMT in-order
Clock (GHz)	2.66	3.20	1.47
DP Peak (GFlop/s)	10.66	12.80	2.96
Register File	16×128b	128×128b	512×1024b
Local Store	—	256 KB	16 KB
L1 Data Cache	32 KB	—	—
L2 Cache	256 KB	—	—

Socket Architecture	**Xeon X5550 Nehalem**	**PowerXCell 8i Cell Blade**	**GTX 285 GeForce**
Cores per Socket	4	8 (+PPE)	30
Last Level Cache	8 MB L3	—	—
Primary memory parallelism paradigm	HW prefetch	DMA	Multithreading

Node Architecture	**Xeon X5550 Nehalem**	**PowerXCell 8i Cell Blade**	**GTX 285 GeForce**
Sockets per SMP	2	2	1 (+CPU)
DP Peak (GFlop/s)	85.33	76.80	88.84
DRAM Pin Bandwidth (GB/s)	51.20	51.20	159.00

TABLE 1.1: Architectural summary of evaluated platforms. Note, all performance numbers are theoretical peak.

ify their programs to elicit better cache behavior, though caches and hardware prefetchers try to render such modifications unnecessary.

To better support performance-oriented programming, accelerator architectures have tried to provide more explicit control of the memory hierarchy. Architectures like Cell's SPEs take a three-level approach with the addition of a software-controlled local store memory seated between DRAM and registers. For correct execution, the programmer must explicitly regiment transfers of data from DRAM to the local store (via DMA transfers), with the ability to rely on compilers to control data movement from local store to registers.

GPU architectures have also adopted a three-level memory hierarchy. Programs access data from external DRAM (device memory) and store values in registers. They may also work with data in on-chip local store (shared memory), although unlike Cell's SPEs, this is not required. GPU's may also cache (device) DRAM to register transfers, although for this generation of GPU architecture these caches are for read-only data. For discrete GPU cards, such as those we benchmark here, the GPU DRAM is separate from the CPU DRAM

Data Movement	Xeon X5550 Nehalem	PowerXCell 8i Cell Blade	GTX 285 T10P
LS↔regs	N/A	Compiler	Compiler[2]
DRAM↔LS	N/A	User[3]	Compiler[2]
DRAM↔regs	Compiler[1]	N/A	Compiler
Host↔DRAM	N/A	N/A	User[3]

TABLE 1.2: Automation of the movement of data between address spaces. [1]Cached and prefetch accelerated. [2]Guided via language attributes. [3]functions to interface with DMA engines.

(host memory) on the motherboard. Programs may transfer data between these memory systems via explicit DMA or memory mapping. For motherboard GPUs, both host and device memory are provided by the motherboard DRAM.

Intel Xeon X5550 (Nehalem): The recently released Nehalem includes minor enhancements to the Core microarchitecture, but dramatic changes to the cache and memory architecture. Each core supports 2-way multithreading. When the per-thread instruction-level parallelism is low, running two threads per core can more efficiently utilize functional units. Cores implement 2-way SIMD and separate add and multiply functional units, yielding a peak throughput of 4 double-precision flops per cycle per core.

Nehalem implements an inclusive (content of L2 cache includes that of the L1) cache hierarchy. In particular, each core has a private 32 KB L1 and as well as a private 256 KB L2 cache. Moreover, all cores on a socket share an 8 MB L3 cache, in contrast to the previous processor generation's use of private 4 MB caches kept coherent via a snoopy (a cache coherency mechanism) frontside bus. Critically, Intel has integrated three DDR3 memory controllers on each chip and implemented a inter-chip network (QuickPath) that carries snoop and remote node access requests. Unfortunately, this non-uniform memory access (NUMA) architecture may suffer poor scalability on some challenging problems.

IBM QS22 PowerXCell 8i (Cell Blade): The IBM Cell processors used in this chapter represent the enhanced double-precision (eDP) variant of the Cell processor found in Sony's PlayStation3 game console. The Cell is based on a single-chip heterogeneous core design. In particular, each chip has one dual-threaded, dual-issue, conventional cache-based PowerPC core (the PPE) and eight, efficiency-optimized, local store-based SPEs. Each SPE executes all code from a small 256 KB, DMA-filled, local store and can execute one double-precision SIMD fused multiply add (FMA) per cycle. As such, the aggregate SPE performance greatly exceeds that of the PPE, meaning performance-critical and performance-intensive routines should be reimplemented for the SPE architecture. Like Nehalem, all memory controllers are integrated on-chip, and the servers are dual-socket NUMA SMPs.

Programming Model	Xeon X5550 Nehalem	PowerXCell 8i Cell Blade	GTX 285 GeForce
OpenMP	✓	—	—
PThreads	✓	✓†	—
CUDA	—	—	✓

TABLE 1.3: Programming models used by platforms. †Only in conjunction with libspe2.

There are two principle differences between the QS22 blades and prior generations (*e.g.*, IBM's QS20). First, each SPE's double-precision performance has been dramatically improved to be half of single-precision performance. Secondly, the 512 MB of XDR DRAM per processor has been replaced with 16 GB of DDR2-800 DRAM per socket. In principle, both DRAM types should deliver the same bandwidth (25.6 GB/s per socket), but we have observed that this is rarely true. Rather, sustained bandwidth is often less than 20 GB/s per socket. Consequently, we expect better performance for large or floating-point intensive problems on QS22 than QS20, whereas we expect worse performance on small (less than 1 GB) memory-intensive problems.

NVIDIA GeForce GTX 285: The NVIDIA GPU considered in this chapter is based on the GT200 processor architecture. It differs markedly from the other systems in its direct support for massive fine-grained multithreading as the primary mechanism for hiding memory latency. This current-generation GPU consists of 30 streaming multiprocessors (SMs), each supporting up to 1024 co-resident threads, for a total of up to 30,720 threads per chip. Each SM has a very large 64 KB register file, providing 16,384 32-bit registers for its resident threads, and a 16 KB on-chip local store that can be shared amongst them. It schedules and executes its threads in SIMD groups of 32 called "warps". The on-board GDDR3 memory is connected to the GPU by a wide data path that delivers extremely high bandwidth, with a theoretical peak of 159 GB/s on the GTX 285.

1.3.2 Parallel Programming Models

As architectures continue to diversify, they have mandated specialized programming models. Thus, as shown in Table 1.3, our SpMV experiments employ three distinct shared memory parallel programming models, depending on the platform: OpenMP (Nehalem only), POSIX Threads (Nehalem and Cell), and CUDA (GTX 285 only).

OpenMP: OpenMP [3] is a pragma controlled, fork-join, shared memory parallel programming model. Its simplicity derives from the common use-case of just identifying the key parallelizable loops and annotating them with a pragma. Although OpenMP provides some speedup for SpMV, we show

that in practice a considerable amount of additional work can and must be performed in order to achieve high performance.

POSIX Threads: POSIX threads (or pthreads) [1] is a function-driven, fork-join, shared memory parallel programming model. For our purposes, when using pthreads, threads are created at the beginning of the application and used throughout in a bulk-synchronous SPMD model. Indeed, we created and used a number of fast, low-overhead, spin barriers designed to work in such a style.

Cell's SPEs are programmed using libspe. However, in many ways, it behaves like pthreads. Essentially, for every SPE thread, one creates a PPE thread. Often, that thread immediately spawns an SPE thread and promptly yields. As such in practice it is common for each application to have one pthread and 16 SPE threads running simultaneously.

CUDA: The CUDA platform [2] provides a simple and direct model for programming the GPU. A CUDA program consists of a one or more host threads, running on the CPU, that may launch parallel "kernels" on the GPU. Each kernel is a blocked SPMD computation: it executes a single sequential program across many parallel threads, which are additionally grouped into thread blocks. Threads within a block may synchronize freely at barriers, but separate blocks may not directly synchronize with each other. The decomposition of parallel work into kernels provides the only means for bulk synchronization between separate blocks.

Matching the hierarchical organization of threads is a hierarchy of disjoint memory spaces, including: (a) thread-private memory, which is typically stored in registers, (b) per-block shared memory, which is stored in fast on-chip local store, and (c) device memory visible to all threads, which is stored in external DRAM. GPU hardware provides a number of caching mechanisms for device memory. The read-only texture cache, utilized via language attributes, provides read-only caching of data optimized for access patterns typical in graphics applications. Future generations of GPU hardware will provide full read-write caching of device memory [9].

1.3.3 Matrices

SpMV performance depends strongly on the properties of the input matrix. The most influential factors include the matrix dimension, non-zero density (*e.g.*, non-zeros per row), variance in the number of non-zeros per row/column, and the specific non-zero pattern (*e.g.*, small dense subblocks, diagonal substructure, randomly distributed). In our experiments, we consider a variety of matrices that arise in real applications and that also vary with respect to these attributes, taken from various sources (see Vuduc [13, App. B]). Figure 1.2 summarizes this matrix benchmark set, and includes small "spyplots" of the non-zero pattern.

	Dense	Protein	Spheres	Cantilever	Wind Tunnel	Harbor	QCD	Ship	Economics	Epidemiology	Accelerator	Circuit	webbase	LP
Spyplot														
Rows	2K	36K	83K	62K	218K	47K	49K	141K	207K	526K	121K	171K	1M	4K
Cols	2K	36K	83K	62K	218K	47K	49K	141K	207K	526K	121K	171K	1M	1M
NNZ	4.0M	4.3M	6.0M	4.0M	11.6M	2.4M	1.9M	4.0M	1.3M	2.1M	2.6M	0.9M	3.1M	11.3M
average NNZ/Row	2000	119	72	65	53	50	39	28	6	4	22	6	3	2825
Symmetric	-	✓	✓	✓	✓	-	-	✓	-	-	✓	-	-	-

FIGURE 1.2: Set of matrices used across all three platforms. Note: NNZ is the number of nonzeros. Although some matrices are symmetric, no implementation in this chapter exploits that property.

1.4 Implications of Architecture on SpMV

Given the diversity of architectural approaches and the sensitivity of SpMV performance on the input matrix (possibly known only at run time), we might reasonably conclude that there is not likely to be a single "best" SpMV implementation. In this section, we explore how some of the major architectural features will influence the design of an SpMV implementation.

1.4.1 Memory Subsystem

The reference CSR implementation of SpMV in Figure 1.1 is dominated by three memory access patterns: (a) a unit-stride read of the matrix nonzero values and column indices; (b) a matrix-dependent, and potentially random gather from the source vector; and (c) a unit-stride write of the destination vector. In principle, these are all memory demand requests, and although caches can filter many of them, in practice restructuring these accesses is essential to good performance.

Recall from Section 1.2 that in the best case of perfect temporal reuse of the source and destination vectors, a lower bound on the time to execute an SpMV is simply the time to read (stream) the matrix data structure. In double-precision CSR, this means roughly 8 bytes for the non-zero value plus 4 bytes for the integer column index, assuming a 32-bit `int`, or 12 bytes of traffic per non-zero. These requests will be compulsory misses on a cached memory hierarchy. At two flops per non-zero, performance in flops per second will be at most (DRAM bandwidth) divided by (12 bytes) times (2 flops), or $\frac{\text{bandwidth}}{6}$. The main solution is to further compress the matrix data

structure through smaller indices, exploiting symmetry, or alternative matrix formats that can reduce indices (*e.g.*, exploiting dense block substructure via register blocking [8]).

If the source and destination vector working sets exceed the cache capacity, performance will be further diminished by capacity misses. Additionally, a highly irregular distribution of non-zeros will reduce the spatial locality of vector or cache line accesses, implying wasted bandwidth. In either case, careful orchestration of source and destination vector accesses is essential. Reordering rows and columns or use of explicit cache-level blocking of the matrix data structure (akin to cache blocking or tiling in the case of dense matrix operations), can help. Moreover, these techniques are applicable regardless of whether a particular architecture is actually cache-based or instead uses a software-controlled local-store.

Analogous to the locality-enduced partitioning challenges in the distributed memory world, NUMA architectures warrant careful allocation and placement of data. Again, block-based organization of the matrix data structure, and allocation and placement of data accordingly are essential. (Even for cacheable workloads, we may observe the same effects due to non-uniform cache architectures.

Finally, there are frequently additional memory (and even cache) latency tolerance mechanisms available, many of which can be controlled in a well-designed SpMV implementation. These mechanisms include hardware and software prefetchers, hardware multithreading, out-of-order execution, and DMA.

1.4.2 Processor Core

Though we intuitively expect the memory system to be the main bottleneck, it might not be for every combination of processor and matrix. In fact, modern efficiency-oriented cores may falter when dealing with the irregular computational structure associated with SpMV. In particular, consider that the reference SpMV implementation in Figure 1.1 has little or no instruction- or data-level parallelism (ILP and DLP, respectively); worse, its instruction mix is dominated by non-floating-point instructions. Attaining peak performance on conventional processor architectures requires just the opposite: high ILP, high DLP, and floating-point intensity. For example, on a Nehalem-class processor, one must express 5-way ILP and 2-way DLP collectively among the two threads per core. Clearly, one may trade ILP for DLP and thus express 10-way DLP via 5 instructions. Achieving peak performance on a GPU typically requires thousand-way thread-level parallelism (TLP) to fully utilize each SM and 32-way DLP within a SIMD warp to avoid divergence. On either machine, reorganization of loop structure (*e.g.*, segmented scan) as well as the data structure (*e.g.*, register blocking) may express more parallelism and ensure core performance is not an impediment to SpMV performance. Moreover,

Optimization	Xeon X5550 Nehalem	PowerXCell 8i Cell Blade	GTX 285 GeForce
Partitioned Storage	✓	✓	✓
Register Blocking	✓	✓	✓
Index Compression	✓	16b only	—
Format Exploration	✓	BCOO only	✓
Cache Blocking	✓	✓	✓
TLB Blocking	✓	✓	—
SW Prefetch	✓	—	—
DMA	—	✓	—

TABLE 1.4: Programming models used by platforms. †Only in conjunction with libspe2.

techniques like register blocking can asymptotically amortize the instruction or operation overhead to one load, multiply and add per nonzero.

1.5 Optimization Principles for SpMV

Given the architectures and intuition about SpMV outlined in previous sections, this section describes specific and effective performance optimization techniques. We organize these optimizations into three broad categories, based on their expected benefit: (a) efficient parallelization, (b) reducing memory traffic, or (c) orchestrating data movement. Table 1.4 summarizes these optimizations. There are many possible techniques within each category, of which we discuss only a subset in this chapter; refer to the original reference studies for additional details [5, 7, 15].

1.5.1 Reorganization for Efficient Parallelization

Multicore and accelerators are becoming massively parallel compute platforms. Although SpMV exhibits inherent loop-level parallelism, we must rescast this parallelism as thread-, data-, or instruction-level parallelism and quite possibly, restructure the algorithm to express even more parallelism.

Naïve Parallelization: The simplest approach to parallelization is to simply allocate one or more rows to each thread. In such a scheme, there is typically far more thread-level parallelism than there is hardware support, and we might expect a thread scheduler to perform effective load balancing. This expection does indeed hold on GPU platforms, but for the Cell and Nehalem processors, thread creation and management is a relatively expensive operation. Consequently, on those platforms, we are driven to the other extreme

where we match the expressed degree of software thread-level parallelism to available hardware thread-level parallelism. However, this thread-centric approach requires programmers perform explicit load balancing, which we do.

Segmented scan: An alternative approach to parallelization is to treat each non-zero (or groups of consecutive non-zeros) as the unit of parallelism, rather than a row or rows. This so-called *segmented scan* implementation typically creates much more parallelism and obviates the load balancing problem [6]. Although this parallelism may be cast as thread-, data-, or instruction-level parallelism, they all require efficient conditional execution, which is a major challenge on the Cell and Nehalem platforms. As such, it was only implemented on the GPU platform.

SIMDization: All of the platforms provide some hardware mechanism to support short-vector oriented data parallelism. Although compilers can in principle extract and generate the code to exploit such SIMD units, we find that the state-of-practice in what the compiler provides lags what is possible. Therefore, we consider explicit SIMDization on the Cell and Nehalem platforms. Explicit SIMDization replaces pairs of memory or arithmetic operations on consecutive elements with a compiler intrinsic (*e.g.*`_mm_mul_pd()`). Clearly this manual process is extremely intrusive and should be used sparingly. On GPU platforms, in contrast, SIMDization is provided implicitly by the hardware, which packs consecutive threads of a block into 32-thread SIMD warps. Instead of using explicit vector operations, the programmer organizes the execution of threads to minimize execution and memory access divergence within warps.

Equivalent Representations: Rather than simply reorganizing loop structures to efficiently parallelize the kernel, we may also reorganize the matrix data structure itself. One approach is the ELLPACK/ITPACK [10] (ELL) storage format. This format organizes an $M \times N$ matrix having at most K nonzeros per row as two dense $M \times K$ arrays stored in column-major order, padding rows with fewer than K nonzeros with zeros. Parallelizing across rows is free of load imbalance, but at the cost of wasting work on the zero values inserted for padding. Clearly, matrices for which the maximum number of nonzeros per row is significantly larger than the average will perform poorly. For each architecture and matrix combination, one must analyze the potential benefit. Due to the massive parallelism demanded by the GPUs, ELLPACK (including our customized variants) was only considered for that platform.

Partitioned Storage: Nominally, in CSR one implements the sparse matrix on a shared memory architecture as three large arrays (one for values, one for column indices, one for row pointers). However, on NUMA and shared cache architectures such storage may be inefficient. Allocating the matrix all at once can lead to it being pinned to one set of memory controllers. Idle memory controllers can impair performance. Even applying the appropriate OpenMP pragma to the initialization of the matrix (for NUMA issues) can still result in suboptimal performance given bank and cache conflicts. An alternate solution, implemented on the Cell and Nehalem platforms is to partition the matrix into

submatrices and store each contiguously with the appropriate array padding to mitigate conflicts arising from thread contention in the cache and memory subsystem [16].

1.5.2 Orchestrating Data Movement

Across architectures, there are a variety of hardware mechanisms for orchestrating data movement, by which we mean placing or moving data in order to satisfy current or future demand accesses (loads and stores). The data reorganization techniques related to parallelism, such as partitioned storage (Section 1.5.1), can also influence these hardware-based data movement mechanisms. Beyond these, we can use additional hardware-specific operations or methods, namely prefetch and vector instructions, to explicity orchestrate data movement.

Hardware Stream Prefetching: Hardware stream prefetchers are an architectural component designed to hide memory latency. Typically, upon detection of a stream of cache misses (*i.e.* misses to consecutive cache lines), the prefetcher will engage and speculatively load the next lines. In doing so, the prefetcher can hide the true latency of a cache miss. Unfortunately, to minimize overhead, prefetchers use very simple heuristics and detect relatively simple unit-stride and strided patterns. Moreover, as they are outside of the core, they typically don't have access to a TLB and thus cannot cross a TLB page boundary. Eliciting good prefetcher behavior is essential to high performance on memory-intensive kernels. To that end, we often restructure access patterns into a limited (few per thread) number of unit-stride accesses. This is relatively easy for matrix and destination vector accesses when using CSR.

Software Prefetching: Occasional discontinuities in the address stream can halt a hardware stream prefetcher. Although such patterns are uncommon in purely streaming applications, SpMV contains a mix of streaming and random access behavior (*e.g.*, to the source vector) that can lead to cache misses or disruptions to the hardware prefetcher. To minimize these effects, we may insert a software prefetch (an instruction) for each cache line of the nonzero arrays. We may then tune to find the optimal prefetch distance — far enough ahead to hide memory latency but not so far as to evict useful data from the cache.

Direct Memory Access (DMA): Analogous to software prefetch, we may generate a DMA command to load a number of nonzeros or vector elements. Typically, we double-buffer these operations as to hide (rather than simply amortize) and latency or overhead. Just as software prefetch required tuning so as not to pollute the cache, on Cell, we were forced to balance nonzero, source vector, and destination vector buffer sizes to maximize performance.

Vector: Vector instruction sets permit bulk loads and stores with a single instruction. Similarly, multiple loads to consecutive memory addresses may be coalesced into a single memory transaction. For SpMV, reorganizing memory

accesses so that they may be coalesced will substantially improve performance. For example, utilizing a column-major layout for the 2D arrays in the ELLPACK format ensures that consecutive threads, which process consecutive matrix rows, access memory in a coalesced manner.

1.5.3 Reducing Memory Traffic

When bandwidth-constrained, one may improve SpMV performance by reducing memory traffic through additional data and loop structure reorganization. This section summarizes these techniques.

Cache Blocking: As noted in Section 1.4, the source or destination vectors could exhibit poor spatial and temporal locality. We can improve this behavior by translating cache blocking or tiling techniques commonly used for dense matrix computations to SpMV. However, where this technique amounts to loop restructuring in the dense case, the sparse case requires both loop restructuring and a change in data structure. In particular, we may partition the matrix into submatrices and store these submatrices individually. The most naïve solution would be to ensure that each submatrix spans a fixed number of columns — the cache block size. However, such technique can be very inefficient and underutilize the cache (or local store) as not all source vector elements within that span may be used. Rather, we individually tune the size of each submatrix so that it touches a fixed number of source vector cache lines. That is, only non-empty columns count towards cache capacity. On a cache-based architecture, we may now perform SpMV on the submatrix and can ensure the working set size never exceeds cache capacity.

On Cell, a small change is performed. Rather than storing each column index in its entirety, we may separate out the cache block's column offset as well as high and low bits of the remainder. Within a cache block, there will be many duplicates of the high bits. As such, we may eliminate the duplicates from the volume of memory traffic and encode a list of unique cache lines that must be loaded. Prior to performing the submatrix SpMV operation, we use a DMA get list (`mfc_getl`) to gather these cache lines and pack them contiguously in the local store. Unlike the traditional DMA which operates only on contiguous data, a list DMA defines a list of disjoint addresses and stanza lengths that should be read from DRAM and packed contiguously in the local store. In essence they perform gather/scatter on arbitrary sized elements. As the offsets required to access the local store copy of the relevant source vector elements index the now packed elements, the offsets (column indices) are dramatically different, and guaranteed to be less than 256KB. As such, when storing the matrix, not only may we encode the local store offsets instead of the DRAM offsets, but we may always use a 16-bit column index (the high 16 bits have been encapsulated into the DMA list). Thus, on Cell we maintain two copies of the matrix: the generic DRAM representation, and the local store optimized representation.

Cache blocking can also be applied on a GPU, either explicitly for each

SM's local store, or implicitly for the texture cache available in current generation NVIDIA GPUs. The texture cache mechanism permits tagging of arbitrary regions of device memory as read-only cacheable data. Accesses to texture locations by multiple threads may be satisfied by a single memory transaction, thus reducing external bandwidth demand, although it may not reduce memory fetch latency. Unlike local store, the texture cache can aggregate requests from threads even when they are not in the same thread block.

TLB Blocking: In many ways the performance impacts of cache misses translate to page cache (TLB) misses. As such, we may apply our cache blocking techniques to the TLB in which we mandate that each submatrix may touch not only a finite number of cache lines, but also a finite number of TLB pages. This technique often provides a performance boost on matrices for which cache blocking already benefited performance.

Index Compression: As cache blocking restricts the range of column indices for each submatrix, we only need to encode the offset from the first column for all indices and shift the pointer to the source vector. This results in column indices encoded with fewer bits and a reduction in memory traffic. This technique is implemented when possible on CPUs, but is always implemented on Cell due to fact that it is the local store offset that is being encoded.

Register Blocking: Register blocking exploits 2D similarity among column indices (geometric proximity in the sparsity pattern of nonzers) to aggregate nonzeros into small dense matrices in which some elements are zero. Almost invariably, these matrices (blocks) are less than 16×16 and in this chapter they are always powers of two less than 8×8. The principal advantage of register blocking when memory-bound is that the matrix structure requires only one index per block rather than one per nonzero. Asymptotically, this can reduce memory traffic by 33%, and thus boost performance by 50%.

Matrix Formats: In addition to the ubiquitous CSR and ELLPACK matrix storage formats, we also explored the simpler coordinate (COO) format. In coordinate, each nonzero is accompanied with both a column index and a row index. Nominally, nonzeros can appear in any order. As such, the resultant read/increment/write data dependency limits loop parallelization or software pipelining. However, we may optimize the format by sorting nonzeros by row. This allows for elimination of the data dependency in favor of a simple loop terminated when consecutive nonzeros have different row indices. The result resembles CSR, but is more amenable to segmented scan and conditional execution.

1.5.4 Putting It All Together: Implementations

In this chapter we present 8 different SpMV kernels spanning 5 basic matrix formats (Table 1.5). We did not implement all 15 (5 formats × 3 architectures) because some combinations were inappropriate for certain architectures. For each kernel, we implemented a different subset of the optimizations

described in Sections 1.5.1–1.5.3. When we apply cache blocking-like optimizations (CPU, Cell, and the GPU's hybrid format), each submatrix may be individually optimized. In this section, we describe the architecture-specific peculiarities of each implementation grouped by format.

BCOO (CPU/Cell/GPU): Block coordinate (BCOO) was the most widely used matrix format. We implemented an SpMV kernel for each architecture. However, there were some differences. The CPU and Cell implementations were most similar. Both explored register blocks in powers of two from 1×1 to 8×8. The CPU implementation included exploration of prefetching as well as cache and TLB blocking. The Cell implementation uses DMA instead of prefetching, and always cache blocks for the maximum available local store capacity. When parallelized, one thread was created per hardware thread context or SPE. Cache and TLB blocking occur after parallelization based on nonzeros. Cell implements a degenerate form of segmented scan in which conditional stores are emulated in software via SIMD muxing.

The GPU implementation was somewhat different. It only implements 1×1 COO, has no need for prefetching, uses the texture cache for implicit exploitation of source vector temporal locality, and maximizes parallelism by individually assigning one CUDA thread to each nonzero. It then uses a segmented reduction reminiscent of Blelloch, *et al.* [6] on the CM-2 and Cray C90 and the CUDA-based segmented scan implementation by Sengupta *et al.* [12].

BCSR (CPU/GPU): CSR flavors were implemented both on CPUs and GPUs. They were not implemented on Cell as the benefit would have been small and challenging to implement a fast 1×1 variant. The CPU implementation incorporates the same optimizations as its BCOO brethren, but the GPU implementations (there were two) are quite different. The first GPU CSR implementation, hereafter referred to as CSR(scalar), uses the standard loop and data structures. It parallelizes the computation by assigning one thread to process each row. Unfortunately, this simple approach is generally inefficient as threads within a warp do not access contiguous memory locations, thus preventing memory coalescing and degrading memory bandwidth efficiency. The second implementation, CSR(vector), assigns one 32-thread warp to each row, effectively strip mining the inner loop of the sequential SpMV computation. This method allows for memory coalescing at the cost of an intra-warp reduction at the end of each row. Independently, Baskaran and Bordawekar [4] implemented a similar approach, although they assign one half-warp to each row and pad each row to be a multiple of 16 in length. Their padding guarantees alignment, and hence slightly higher degrees of coalescing, albeit at the cost of potentially significant additional storage. This may incrementally improve performance in some cases, but shares the same fundamental performance characteristics.

GCSR (CPU): GCSR is a variant on BCSR and an alternative to BCOO. It is only beneficial when cache blocking produces submatrices with many empty rows distributed at random (*i.e.* hypersparse). GCSR augments the BCSR data structure with a row coordinate associated with each row

Implementation	Xeon X5550 Nehalem	PowerXCell 8i Cell Blade	GTX 285 GeForce
BCOO	✓	✓	✓
BCSR	✓	—	✓†
GCSR	✓	—	—
Hybrid	—	—	✓
BELLPACK	—	—	✓

TABLE 1.5: Implementations as a function of machine. †Only 1×1 CSR.

pointer. This coordinate is conceptually similar to column indices but as nonzeros are sorted by rows, far fewer are required.

Hybrid (GPU): Unlike the CPU/Cell implementations where hybrid implementations arise from local specialization based on individually optimized cache blocks, the "Hybrid GPU" implementation merges ELLPACK and COO to attain ELLPACK's high performance potential with the performance invariability of COO. Given a parameter K, the matrix is partitioned into two portions. The first submatrix is stored in ELLPACK form with K nonzeros per row, padding rows with fewer than K nonzeros. The second submatrix is stored in COO form and holds the excess entries from rows with greater than K nonzeros. The splitting parameter K can be specified directly, or chosen automatically using an empirical heuristic which is currently $K = \max(4096,\ M/3)$.

BELLPACK (GPU): BELLPACK is a GPU-only implementation that applies one-dimensional row blocking, row permutation, and register blocking to ELLPACK. In CUDA terms, one-dimensional row blocking means that we assign one thread block per block row of the matrix, where the block row size is tunable. Combining 1-D row blocking and row permutation effectively reduces the padding required relative to conventional ELLPACK storage. Within a row block, warps are assigned to consecutive register-block rows. To achieve coalesced accesses, elements from different blocks within the same warp are interleaved. The register block size is tunable. There are no restrictions on the register block size, though a "poor" choice will lead to poor performance.

1.6 Results and Analysis

In this section, we present SpMV performance for our three platforms as a function of input (matrix), optimization (*e.g.*, register blocking), and approach to parallelization (*e.g.*, OpenMP vs. Pthreads). We then examine performance by architecture, to give both implementation and architectural insight.

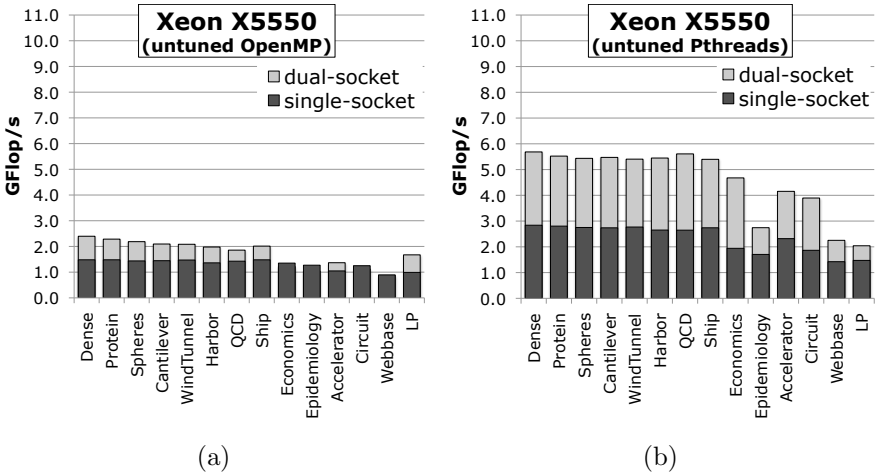


FIGURE 1.3: Untuned SpMV performance as a function of threading model, hardware concurrency, and matrix.

1.6.1 Xeon X5550 (Nehalem)

Figure 1.3(a) presents CSR SpMV performance as a function of threading model (OpenMP vs. Pthreads), matrix, and hardware (using one or both sockets of the dual-socket SMP). We observe that using one socket (via the OpenMP affinity environment variable) always delivers less than 1.5 GFlop/s. Intuitively, we expect the matrices with few nonzeros per row or large dimensions to deliver lower performance as a larger fraction of memory bandwidth is tasked with reading in row pointers and source vector elements. Nevertheless, the performance variability is quite low. Given this kernel should be bandwidth-limited (ignore the 42.66 GFlop/s peak) with a per socket STREAM bandwidth less than 18 GB/s, one would naïvely expect performance less than 3 GFlop/s (2 flops per 12 byte nonzero). The delivered performance is substantially less than this bound. Moreover, when the second socket is employed, performance improves by just 50% on some matrices and none on others.

This lack of socket scalability can well be explained by two facets of SpMV on this NUMA SMP. When the matrix was read from disk, it was done so by one thread. The underlying first-touch policy placed data on the DIMMs with affinity to the core on which that thread was running. Unfortunately, this means that when SpMV is parallelized via OpenMP, only the memory controllers attached to those DIMMs were used. Thus half the SMP's bandwidth was thrown away. The lack of scalability for the simplest matrices is thus an artifact of limited bandwidth and high inter-socket latency. For the more challenging matrices (Epidemiology through Linear Programming), the fact that the vectors were allocated via the same first-touch policy similarly

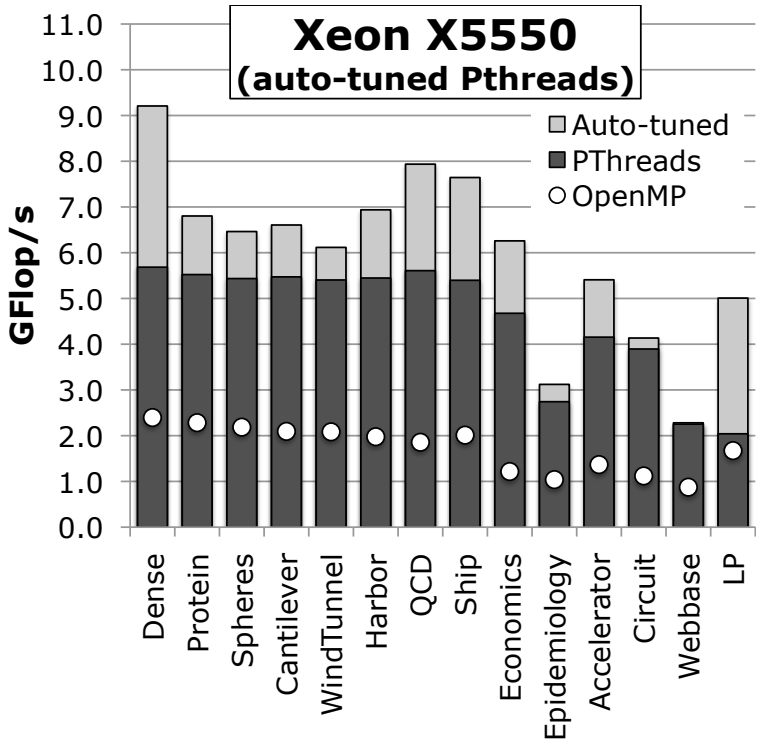


FIGURE 1.4: Nehalem pthread performance before and after auto-tuning. OpenMP included as reference.

limits performance. Finally, some matrices (*e.g.*, webbase) are particularly poorly suited to iterative sparse methods because a naïve parallelization will induce all-to-all communication. That is, each socket updates its respective half of the vector, but on the next iteration each socket will need both halves. This forces an implicit data broadcast and exposes the limited inter-socket bandwidth.

In Figure 1.3(b) We observe that the pthreads implementation (with the NUMA-aware library matrix creation routines) delivers not only substantially better performance (better than 2.5×), but also better multi-socket scalability (typically 2×). The latter is well explained by proper NUMA allocation. However, even in the single-socket configuration, pthreads usually delivers twice the performance of OpenMP. This was quite surprising given we use the correct OpenMP affinity variables, load balancing shouldn't be an issue on at least some of the matrices, and OpenMP usually performs well for structured grid computations. Although the pthreads implementation did provide close

to 3 GFlop/s per socket, we show that substantially better performance can be attained.

Figure 1.4 shows the performance benefits attained for the full SMP when auto-tuning is applied to the pthread implementation. We include OpenMP data as dots for comparison. As previously discussed, a vast number of optimizations were explored including register blocking, index compression, alternate matrix formats (BCOO, GCSR), prefetching, and cache/TLB blocking. We observe that auto-tuning often accelerated performance by 30% (typically from register blocking) and as much as 2.5× (the extreme case requiring TLB and cache blocking). Moreover, the conjunction of pthreads and auto-tuning consistently exceeded OpenMP performance by between 2.6× and 5.1×. Unfortunately, there are some matrices (*e.g.*, Epidemiology, Circuit, Webbase) for which our current tuning regimen seems ill-equiped. These are the problems that exhibit poor cache locality (both temporal and spatial) at any scale and demand substantial inter-core and inter-socket communication.

1.6.2 QS22 PowerXCell 8i

The Cell SpMV auto-tuner is built on the multicore auto-tuner we used for Nehalem. The principal differences are the threading model (superficial change from pthreads to pthreads+libspe) and the fact that due to the complexity of Cell programming, only a subset of the optimization space was implemented. Thus, on Cell the auto-tuner only implements BCOO (omitting BCSR/GCSR), always cache blocks for the available local store capacity, always compresses indices, and always uses DMA. However, the Cell auto-tuner does implement a variant on segmented scan in which each SPE runs a software pipelined, vector length of 1, BCOO segmented scan on its cache block. In essence, this transforms the doubly-nested BCSR or sequential BCOO implementations into a single very-fast pipelined loop. Note that the PPE performs no computation in the SpMV kernel, but rather is used for coordination.

Figure 1.5 presents baseline Cell performance (1×1 BCOO) using either one or both of the Cell chips on the QS22 SMP. It also shows auto-tuned performance using both chips. When examining single-socket performance, we see dramatically different behavior compared to Nehalem. Cell's performance is remarkably constant — a testament to the elimination of CSR's short loops (via segmented scan) coupled with a potentially compute-bound SPE. Although we expected the QS22's DDR2-based stream performance to be substantially lower than the QS20's XDR-based performance, the expected (bandwidth-only) performance bound of 3 GFlop/s per socket is somewhat higher than observed performance (black bar). When the second socket is used, the matrices amenable to NUMA-parallelized attain a speedup of 2×. However, we observe a very similar behavior to that of Nehalem on the challenging matrices.

Principally, the Cell auto-tuner explores alternate register blockings while using only the BCOO format with index compression. Nevertheless, we ob-

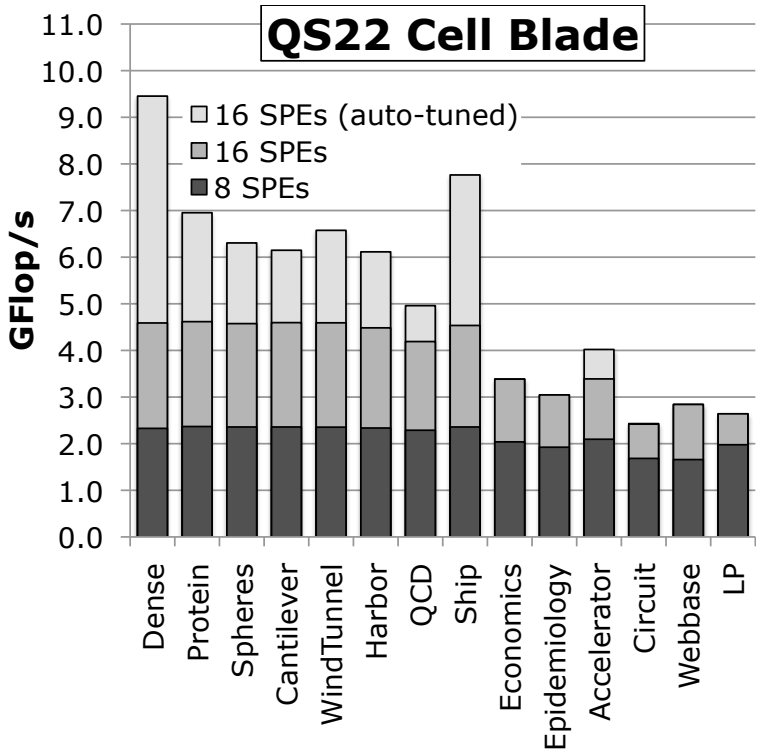


FIGURE 1.5: SpMV performance as a function of matrix, hardware concurrency, and optimization. Note, the untuned baseline is actually a DMA-enabled 1×1 COO implementation, and is thus by no means naïve.

serve that register blocking can dramatically improve performance for some matrices (over 2×). Nominally, register blocking should only improve performance by a factor of 1.5× (if memory-bound). We believe the discrepancy arises from a transition from compute-bound to memory-bound. This may be confirmed as the attained performance (over 9 GFlop/s) aligns well with the bandwidth–arithmetic intensity product. Once again, we observe that our breadth of optimizations is insufficient for more than a third of the matrices. Clearly, there is still ample research material for SpMV.

1.6.3 GTX 285

Unlike the Nehalem and Cell implementations, there is not one single GPU auto-tuner. As such, in this section, we present the performance results of 5 different GPU SpMV implementations. Moreover, in all cases, we assume data (both matrices and vectors) remain resident in GPU *device* memory obviating

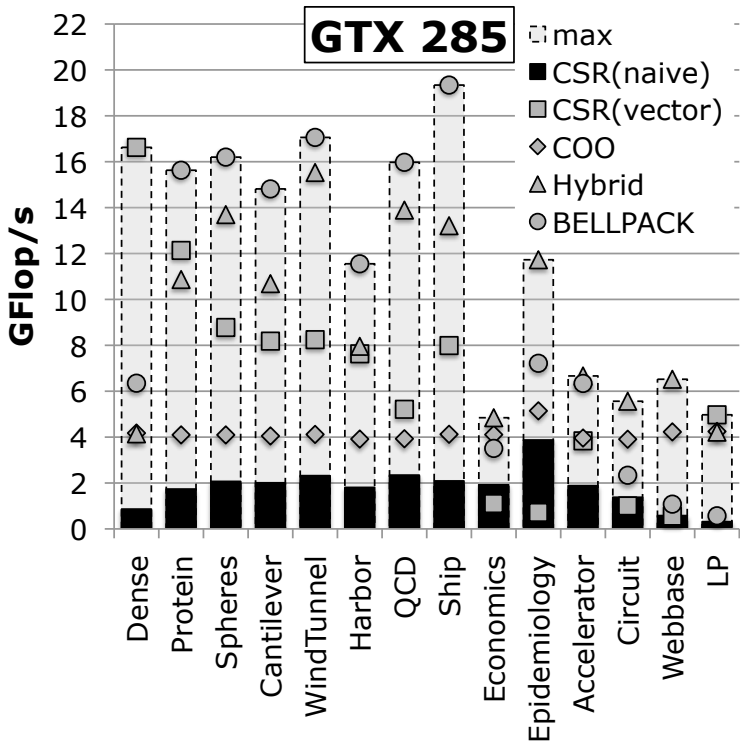


FIGURE 1.6: GPU performance as a function of matrix and implementation (dots). Dotted bars are used simply to visualize the best possible performance.

the need for any PCIe transfers. This is a reasonable assumption for any local iterative sparse solver. Finally, all GPU computations are performed in double precision and access the source vector is accelerated via the on-chip, read-only texture cache.

Figure 1.6 presents each implementation's performance on each matrix. The black bars (naïve CSR) represent a straightforward loop parallelization of the standard 1×1 CSR implementation that assigns one thread per row. GPUs may require several thousand parallel threads in order to fully utilize the processor. This need for abundant parallelism is a challenge for a simple thread-per-row decompsition strategy when matrices have relatively few rows, as is the case with our Dense and LP examples which have 2,000 and 4,284 rows, respectively. Despite the appearance of substantial parallelism on the remainder, this simple implementation exhibits substantial memory divergence when accessing the CSR data structure, since as threads access different rows, they access disjoint rather than contiguous nonzeros. Thus it does not effectively utilize the available 159 GB/s of memory bandwidth since there is an

order of magnitude difference between fully coalesced and uncoalesced memory access. All remaining implementations attempt to attain memory coalescing. They are overlaid in Figure 1.6 via dots with light grey bars highlighting the maximum performance attained via any implementation.

The vectorized CSR implementation, which exhibits much less memory divergence, delivers substantially better performance than the scalar CSR implementation. In addition, it assigns one 32-thread warp per row and thus requires far fewer rows to develop sufficient find-grained parallelism. In the vectorized approach, the degree of memory divergence, and to some extent execution divergence, is determined by the distribution of nonzeros per row. When applied to the six finite-element matrices with an average 50 or more nonzeros per row the vectorized kernel achieves no less than 7.5 GFlop/s. On the other hand, matrices such as Webbase with approximately 3.1 nonzeros per row expose a weakness of the vectorized implementation—several threads of a warp will be idle when a row is much shorter than the warp width.

The segmented reduction-based COO implementation delivers very consistent, albeit lower, performance across the matrix suite. Although robust with respect to the number of rows and distribution of nonzeros per row, the COO implementation suffers from low arithmetic intensity (2 flops for 4+4+8 bytes) and the overhead of many inter-thread operations.

The Hybrid implementation, which combines the easy vectorization of ELLPACK with the robustness of COO, achieves good performance on most every matrix. For matrices with a naturally dense substructure, which is typical of matrices arising from finite element analysis, the register blocked BELLPACK implementation delivers better performance. For such matrices, it delivers 1.1–1.5× higher throughput, which is the range expected given the bandwidth it conserves by reducing the number of indices it must read.

The Nehalem and Cell results demonstrated that there is no one "best" implementation, but rather produce a family of implementations auto-tuned to fit specific matrices. These GPU performance results demonstrate the same trend: the best choice of SpMV kernel depends on the structure of the matrix. BELLPACK delivers superior performance for matrices with small dense blocks, the Hybrid format provides better results on matrices with more challenging sparsity patterns, and CSR (vector) worked best for the matrices with the most nonzeros per row.

1.7 Summary: Cross-Study Comparison

The implementations of SpMV described in this chapter represent the state-of-the-art in performance on the three target hardware platforms. Though SpMV is itself simple to state and to analyze, the diversity of architectural designs and input matrix characteristics means a complex combination

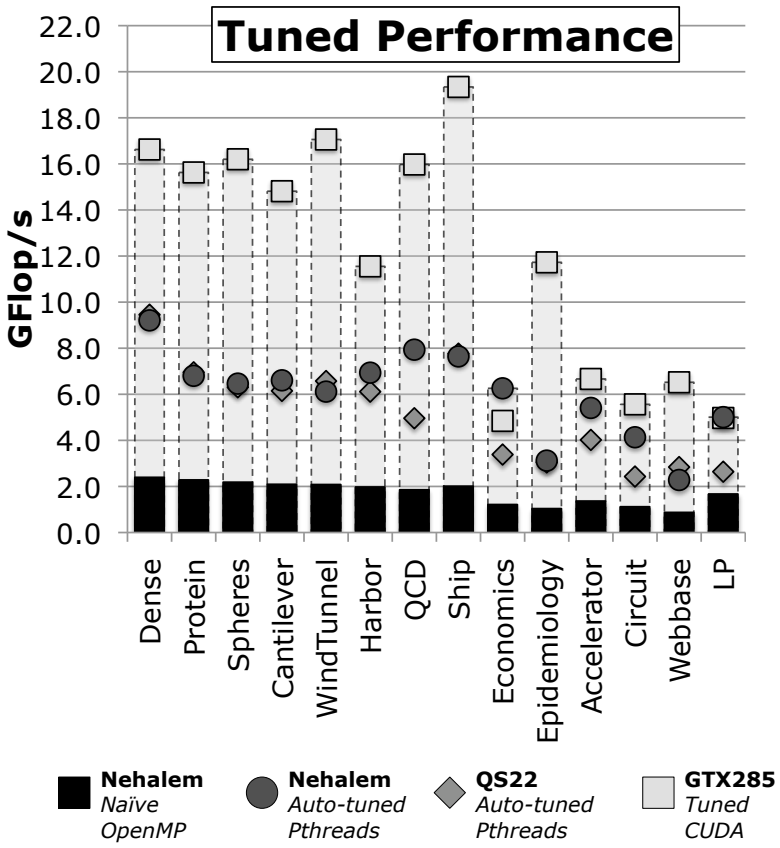


FIGURE 1.7: Performance comparison across architectures. Note: The naïve baseline (Nehalem/OpenMP) is shown as a black bar. The best implementation for each architecture — Nehalem (circle), Cell (diamond), GTX285 (square) — is shown as a color-coded dot with a light gray bar denoting the global best.

of architecture- and matrix-specific techniques is essential to achieving this level of performance.

To summarize the main findings of this chapter, Figure 1.7 compares the best performance we attain on each platform across the suite of matrices. The common baseline performance is the naïve OpenMP parallelization running on Nehalem (the bottom black bars).

Clearly, all optimized implementations deliver substantially better performance. In addition, we observe that the GPU usually delivers better than twice Nehalem's performance, although this should come as no surprise given the GPU has more than triple the Nehalem SMP's bandwidth. As our in-

tuition expected, bandwidth is the determining performance factor; however, note that achieving high levels of sustainable bandwidth requires a significant tuning effort.

Figure 1.7 also shows that the gap between untuned and tuned performance, comparing the baseline OpenMP Nehalem implementation to its tuned Nehalem counterpart (circles), we see a 2–4× difference in performance. This observation suggests just how important selection of the appropriate programming model and performance tuning can be. Moreover, it implies the need for improved tools and techniques that simplify and automate the tuning process.

We observe several cases where Nehalem performance is comparable (Accelerator, Circuit) or exceeds (Economics, LP) GPU performance. Two of these matrices (Circuit and Economics) are the smallest in our test corpus, and may fit in Nehalem's aggregate SMP cache. However, the SMP nature impedes communication of vector elements between successive SpMV's thus limiting the benefit. The other two have sparsity patterns that appear to pose problems for our GPU implementations, partially due to the fact that the vector working set exceeds the texture cache's capacity.

One seeming change relative to the earlier literature shown in Figure 1.7 is Cell's relatively lackluster performance. However, we note that at the time of its initial release, the tuned Cell implementation delivered far better performance than any commodity CPU on the market at that time [14, 15]. That is, due to the absence of any substantive Cell hardware development in the last 4 years, commodity multicore now delivers comparable performance.

1.8 Acknowledgments

The authors acknowledge Georgia Institute of Technology (Georgia Tech), its Sony-Toshiba-IBM Center of Competence, and the National Science Foundation (NSF) for the use of Cell Broadband Engine resources that have contributed to this research. The authors from Lawrence Berkeley National Laboratory (LBNL) are supported by the Director, Office of Science, of the U.S. Department of Energy (DOE) under contract number DE-AC02-05CH11231 and by NSF contract CNS-0325873, along with generous funding and equipment from Microsoft, Intel, and U.C. Discovery (under Awards #024263, #024894, and #DIG07-10227, respectively). The authors from Georgia Tech were supported in part by NSF award number 0833136, NSF CAREER award number 0953100, NSF TeraGrid allocation CCR-090024, joint NSF 0903447 and Semiconductor Research Corporation (SRC) Award 1981, a Raytheon Faculty Fellowship, and grants from the Defense Advanced Research Projects Agency (DARPA) and Intel Corporation. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of DOE, NSF, SRC, DARPA, Microsoft, or Intel.

Bibliography

[1] The Open Group Base Specifications, Issue 6: POSIX Threads (`pthread.h`). IEEE Std 1003.1, 2004. `http://www.opengroup.org/onlinepubs/009695399/basedefs/pthread.h.html`.

[2] NVIDIA CUDA (Compute Unified Device Architecture): Programming Guide, Version 2.1. `http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.1.pdf`, December 2008.

[3] OpenMP: Application Program Interface, version 3.0, May 2008. `http://www.openmp.org/mp-documents/spec30.pdf`.

[4] Muthu Manikandan Baskaran and Rajesh Bordawekar. Optimizing sparse matrix-vector multiplication on GPUs using compile-time and run-time strategies. Technical Report RC24704 (W0812-047), IBM T.J. Watson Research Center, Yorktown Heights, NY, USA, December 2008.

[5] Nathan Bell and Michael Garland. Implementing a sparse matrix-vector multiplication on throughput-oriented processors. In *Proc. ACM/IEEE Conf. Supercomputing (SC)*, Portland, OR, USA, November 2009.

[6] Guy E. Blelloch, Michael A. Heroux, and Marco Zagha. Segmented operations for sparse matrix computations on vector multiprocessors. Technical report, Carnegie Mellon University, Department of Computer Science, Pittsburgh, PA, USA, August 1993.

[7] Jee Whan Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP)*, Bangalore, India, January 2010.

[8] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int'l J. of High Performance Computing Applications (IJHPCA)*, 18(1):135–158, February 2004.

[9] NVIDIA. NVIDIA's next generation CUDA compute architecture: Fermi^{TM}, v1.1. Whitepaper (electronic), September 2009. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

[10] John R. Rice and Ronald F. Boisvert. *Solving elliptic problems using ELLPACK*. Springer Verlag, 1984.

[11] Yousef Saad. *Iterative Methods for Sparse Linear Systems, Second Edition*. Society for Industrial and Applied Mathematics, April 2003.

[12] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In *Proc. ACM SIGGRAPH/EUROGRAPHICS Symp. Graphics Hardware*, San Diego, CA, USA, 2007.

[13] Richard W. Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, University of California, Berkeley, CA, USA, January 2004.

[14] Sam Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proc. ACM/IEEE Conf. Supercomputing (SC)*, 2007.

[15] Sam Williams, Richard Vuduc, Leonid Oliker, John Shalf, Katherine Yelick, and James Demmel. Optimizing sparse matrix-vector multiply on emerging multicore platforms. *Parallel Computing (ParCo)*, 35(3):178–194, March 2009. Extends conference version: `http://dx.doi.org/10.1145/1362622.1362674`. .

[16] Samuel Webb Williams. *Auto-tuning performance on multicore computers*. UCB/EECS-2008-164, University of California, Berkeley, CA, USA, December 2008.