Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors

Nathan Bell
NVIDIA Research
nbell@nvidia.com

Michael Garland
NVIDIA Research
mgarland@nvidia.com

ABSTRACT

Sparse matrix-vector multiplication (SpMV) is of singular importance in sparse linear algebra. In contrast to the uniform regularity of dense linear algebra, sparse operations encounter a broad spectrum of matrices ranging from the regular to the highly irregular. Harnessing the tremendous potential of throughput-oriented processors for sparse operations requires that we expose substantial fine-grained parallelism and impose sufficient regularity on execution paths and memory access patterns. We explore SpMV methods that are well-suited to throughput-oriented architectures like the GPU and which exploit several common sparsity classes. The techniques we propose are efficient, successfully utilizing large percentages of peak bandwidth. Furthermore, they deliver excellent total throughput, averaging 16 GFLOP/s and 10 GFLOP/s in double precision for structured grid and unstructured mesh matrices, respectively, on a GeForce GTX 285. This is roughly 2.8 times the throughput previously achieved on Cell BE and more than 10 times that of a quad-core Intel Clovertown system.

1. INTRODUCTION

Sparse matrix structures arise in numerous computational disciplines, and as a result, methods for efficiently processing them are often critical to the performance of many applications. Sparse matrix-vector multiplication (SpMV) operations have proven to be 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. The remaining part of these iterative methods (e.g., the conjugate gradient method [19]), typically reduce to dense linear algebra operations that are readily handled by optimized BLAS [14] and LAPACK [1] implementations.

Modern NVIDIA GPUs are throughput-oriented manycore processors that offer very high peak computational throughput. Realizing this potential requires exposing large amounts of fine-grained parallelism and structuring computations to exhibit sufficient regularity of execution paths and memory access patterns. Recently, Volkov and Demmel [21] and Barrachina *et al.* [2] have demonstrated how to achieve significant percentages of peak floating point throughput and bandwidth on dense matrix operations. These achievements are due, in part, to the regular access patterns of dense matrix operations. In contrast, sparse matrix operations must cope with various forms of irregularity present in the underlying matrix representation.

In this paper, we explore the design of efficient SpMV kernels for throughput-oriented processors like the GPU. We implement these kernels in CUDA [15, 16] and analyze their performance on the GeForce GTX 285 GPU. Sparse matrices arising in different problems can exhibit a broad spectrum of regularity. We consider data representations and implementation techniques that span this spectrum, from highly regular diagonal matrices to completely unstructured matrices with highly varying row lengths.

Optimizing SpMV for throughput-oriented manycore processors is qualitatively different than SpMV on latency-oriented multicores. Whereas a multicore SpMV kernel needs to develop 4 or perhaps 8 threads of execution, a manycore implementation must distribute work among thousands or tens of thousands of threads. Manycore processors will often demand a high degree of fine-grained parallelism because, instead of using large sophisticated caches to avoid memory latency, they use hardware multithreading to hide the latency of memory accesses. This distinction implies that parallel decomposition strategies that suffice for multicore processors may fail to expose the necessary level of parallelism in a manycore setting. Furthermore, as the parallel granularity is refined, the impact of thread workload imbalances becomes a significant concern. In contrast, multicore implementations will tend to "average out" small scale irregularities by processing many elements per thread.

Despite the irregularity of the SpMV computation, we demonstrate that it can be mapped quite successfully onto the fine-grained parallel architecture employed by the GPU. For unstructured finite-element matrices, we measure performance of roughly 10 GFLOP/s in double precision and around 15 GFLOP/s in single precision. Moreover, these kernels achieve high bandwidth utilization, often in excess of 100 GByte/s without caching and 140 GByte/s with caching enabled. These figures correspond, respectively, to 63% and 88% of the theoretical peak DRAM bandwidth of 159.0 GByte/s. Harnessing a large portion of peak bandwidth demonstrates that these kernels are highly efficient on the inherently bandwidth-limited SpMV computation.

2. TARGET PLATFORM

Our SpMV kernels are designed to be run on throughput-oriented architectures in general and GPUs supporting the CUDA parallel computing architecture [15, 16] in particular. Broadly speaking, we assume that throughput-oriented processors will provide (1) mas-

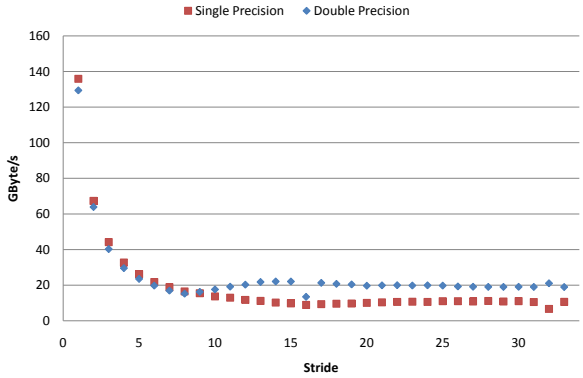


Figure 1: Effect of stride on achieved bandwidth in `saxpy` and `daxpy` kernels for computing $y \leftarrow ax + y$.

sive hardware multithreading, (2) some form of SIMD thread execution, and (3) vectorized or coalesced load/store operations.

Current NVIDIA GPUs support up to 30K co-resident parallel threads, and a blocked SPMD (Single Program Multiple Data) programming model. A single parallel program, or *kernel*, is executed across a collection of parallel threads. These threads are organized into blocks which are the granularity at which threads can share on-chip low-latency memory and synchronize with hardware barrier instructions. Thread creation, scheduling, and management is performed entirely in hardware. Since these processors rely on hardware multithreading, rather than caches, to hide the latency of memory accesses, a program must expose substantial fine-grained parallelism to fully utilize the processor.

The GPU employs a SIMT (Single Instruction Multiple Thread) architecture [15, 16] in which threads are executed in groups of 32 called *warps*. A warp executes a single instruction at a time across all its threads. The threads of a warp are free to follow their own execution path and all such *execution divergence* is handled automatically in hardware. However, it is substantially more efficient for threads of a warp to follow the same execution path for the bulk of the computation.

The threads of a warp are also free to use arbitrary addresses when accessing off-chip memory with load/store operations. Accessing scattered locations results in *memory divergence* and requires the processor to perform one memory transaction per thread. On the other hand, if the locations being accessed are sufficiently close together, the per-thread operations can be *coalesced* for greater memory efficiency. When accessing 32- or 64-bit values, global memory can be conceptually organized into a sequence of 128-byte segments. Memory requests are serviced for 16 threads (a half-warp) at a time. The number of memory transactions performed for a half-warp will be the number of segments touched by the addresses used by that half-warp[1]. If a memory request performed by a half-warp touches precisely 1 segment, we call this request *fully coalesced*, and one in which each thread touches a separate segment we call *uncoalesced*. If only the upper or lower half of a segment is accessed, the size of the transaction is reduced [17].

Non-contiguous memory accesses reduce bandwidth efficiency and therefore the performance of memory-bound kernels. Figure 1 illustrates the bandwidth achieved by a `saxpy` kernel with variable stride between elements. Any stride greater than 1 results in non-contiguous access to the x and y vectors. Since larger strides result in a single warp touching more segments, we expect an inverse relationship between stride and performance of the memory-bound `saxpy` kernel. This expectation is confirmed by our measurements, which show that the effective memory bandwidth (and hence performance) decreases as the stride increases from 1 to 33. Memory accesses with unit stride are more than ten times faster than those with greater separation. The double precision `daxpy` kernel shows similar behavior, although it achieves roughly twice the bandwidth of `saxpy` at larger strides. Ensuring proper memory alignment is also important, as unaligned accesses will deliver only approximately 60% of the aligned access rate.

The GPU provides a small cache—referred to as a *texture cache* because of its use in the graphics pipeline—for read-only data. This cache does *not* reduce the latency of loads. Its purpose is to amplify bandwidth by aggregating load requests to the same address from many threads, effectively servicing them all with a single off-chip memory transaction. Except where otherwise noted, our SpMV kernels access the vector x via this cache while using uncached loads to access the representation of the matrix A.

[1]Earlier devices that do not support Compute Capability 1.2 have stricter coalescing requirements [17].

3. MATRIX-VECTOR MULTIPLICATION

Sparse matrix-vector multiplication (SpMV) is arguably the most important operation in sparse matrix computations. Iterative methods for solving large linear systems ($Ax = b$) and eigenvalue problems ($Ax = \lambda x$) generally require hundreds if not thousands of matrix-vector products to reach convergence. In this paper, we consider the operation $y \leftarrow Ax + y$ where A is large and sparse and x and y are column vectors. While not a true analog of the BLAS `gemv` operation (i.e., $y \leftarrow \alpha Ax + \beta y$), our routines are easily generalized. We choose this SpMV variant because it isolates the sparse component of the computation from the dense component. Specifically, the number of floating point operations in $y \leftarrow Ax + y$ is always twice the number of nonzeros in A (one multiply and one add per element), independent of the matrix dimensions. In contrast, the more general form also includes the influence of the vector operation βy which, when the matrix has few nonzeros per row, will substantially skew performance.

Because of the importance of SpMV operations, there is a substantial literature exploring numerous optimization techniques. Vuduc [22] provides a good overview of these techniques, many of which are implemented in the Sparsity framework [12]. One of the primary goals of most SpMV optimizations is to mitigate the impact of irregularity in the underlying matrix structure. This is also our central concern. Specifically, we aim to minimize execution and memory divergence caused by the potential irregularity of the underlying sparse matrix. Since an efficient SpMV kernel should be memory-bound, our measure of success will be the fraction of peak bandwidth these kernels can achieve.

We address this problem by focusing on choosing appropriate matrix formats and designing parallel kernels that can operate on these formats efficiently. We choose well-known formats—DIA, ELL, CSR, and COO—that are supported by standard sparse matrix packages such as SPARSKIT [18], and we organize our computations to minimize divergence. Not surprisingly, our techniques follow optimization techniques that were successful on vector and SIMD machines of the past.

Given a particular matrix, it is essential that we select a format that is a good fit for its sparsity pattern. We identify three basic sparsity regimes: (1) diagonal matrices, (2) matrices with roughly uniform row lengths, and (3) matrices with non-uniform row lengths. Each of these sparsity regimes is best addressed using different formats. Since we are not concerned with modifying matrices, we restrict our attention to static formats, as opposed to

$$\texttt{A} = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

DIA format:

$$\texttt{data} = \begin{bmatrix} * & 1 & 7 \\ * & 2 & 8 \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \qquad \texttt{offsets} = \begin{bmatrix} -2 & 0 & 1 \end{bmatrix}$$

ELL format:

$$\texttt{data} = \begin{bmatrix} 1 & 7 & * \\ 2 & 8 & * \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \qquad \texttt{indices} = \begin{bmatrix} 0 & 1 & * \\ 1 & 2 & * \\ 0 & 2 & 3 \\ 1 & 3 & * \end{bmatrix}$$

CSR format:

$$\begin{aligned} \texttt{ptr} &= \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix} \\ \texttt{indices} &= \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix} \\ \texttt{data} &= \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix} \end{aligned}$$

COO format:

$$\begin{aligned} \texttt{row} &= \begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 2 & 3 & 3 \end{bmatrix} \\ \texttt{indices} &= \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix} \\ \texttt{data} &= \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix} \end{aligned}$$

Figure 2: Sparse matrix representations for a simple example matrix A. Padding entries (marked $*$) are set to zero.

those suitable for rapid insertion and deletion of elements. Furthermore, we do not consider transformations such as row permutation that globally restructure the matrix.

In this paper, we summarize how data is laid out in memory and how work is assigned to parallel threads. The source code for these SpMV programs can be found in our technical report [4] and its accompanying software package.

3.1 Diagonal Format

When nonzero values are restricted to a small number of matrix diagonals, the *diagonal* format (DIA) is an appropriate representation [19]. Although not general purpose, this format efficiently encodes matrices arising from the application of stencils to regular grids, a common discretization method.

The diagonal format is formed by two arrays: `data`, which stores the nonzero values, and `offsets`, which stores the offset of each diagonal from the main diagonal. Diagonals above and below the main diagonal have positive and negative offsets, respectively. Figure 2 shows an example.

Parallelizing SpMV for the diagonal format is straightforward. We assign one thread to each row and store `data` in column-major order so that consecutive elements within each diagonal are adjacent. As each thread iterates over the diagonals crossing its row, this memory layout guarantees that contiguous threads access contiguous elements of the `data` array, as shown in Figure 3. Since

$$\begin{aligned} \texttt{data} \quad & \begin{bmatrix} * & * & 5 & 6 & 1 & 2 & 3 & 4 & 7 & 8 & 9 & * \end{bmatrix} \\ \text{Iteration 0} \quad & \begin{bmatrix} & & 2 & 3 & & & & & & & & \end{bmatrix} \\ \text{Iteration 1} \quad & \begin{bmatrix} & & & & 0 & 1 & 2 & 3 & & & & \end{bmatrix} \\ \text{Iteration 2} \quad & \begin{bmatrix} & & & & & & & & 0 & 1 & 2 & \end{bmatrix} \end{aligned}$$

Figure 3: The DIA `data` array and the memory access pattern of the DIA SpMV kernel running in 4 threads across 3 iterations.

$$\begin{aligned} \texttt{data} \quad & \begin{bmatrix} 1 & 2 & 5 & 6 & 7 & 8 & 3 & 4 & * & * & 9 & * \end{bmatrix} \\ \texttt{indices} \quad & \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 2 & 2 & 3 & * & * & 3 & * \end{bmatrix} \\ \text{Iteration 0} \quad & \begin{bmatrix} 0 & 1 & 2 & 3 & & & & & & & & \end{bmatrix} \\ \text{Iteration 1} \quad & \begin{bmatrix} & & & & 0 & 1 & 2 & 3 & & & & \end{bmatrix} \\ \text{Iteration 2} \quad & \begin{bmatrix} & & & & & & & & 0 & 1 & 2 & 3 \end{bmatrix} \end{aligned}$$

Figure 4: Memory access pattern of the ELL SpMV kernel.

the DIA format for this matrix has three diagonals the `data` array is accessed three times by each of the four executing threads. Given that consecutive rows, and hence threads, correspond to consecutive matrix columns, threads within the same warp access the `x` vector contiguously as well. We gain further efficiency by padding columns to align the `data` array properly and by reading the `offsets` array only once per thread block, rather than once per thread.

The diagonal format avoids the need to store row/column indices and guarantees contiguous (i.e., coalesced) accesses to the `data`, `x`, and `y` arrays. However, it can also potentially waste storage since it allocates additional memory for padding and explicitly stores zero values that occur in occupied diagonals. Within its intended application area—stencils applied to regular grids—these problems do not arise. However, it is important to note that many matrices have sparsity patterns that are inappropriate for DIA.

3.2 ELL Format

The ELLPACK/ITPACK [11] (or ELL) format is more general than DIA and is particularly well-suited to vector architectures. An M-by-N sparse matrix with at most K nonzeros per row is stored as a dense M-by-K array `data` of nonzeros and array `indices` of column indices. All rows are zero-padded to length K. ELL is more general than DIA since the nonzero columns need not follow any particular pattern. See Figure 2 for an example of this format. ELL is most efficient when the maximum number of nonzeros per row does not substantially differ from the average, which is often the case with matrices obtained from semi-structured meshes and well-behaved unstructured meshes.

The structure of the ELL computation is nearly identical to the DIA case, with the exception that column indices are explicit in ELL and implicit in DIA. The memory access pattern of the ELL kernel, shown in Figure 3, is also similar. As with DIA, we store the ELL arrays in column-major order and pad them for alignment. However, unlike DIA, the ELL kernel does not necessarily access the `x` vector contiguously.

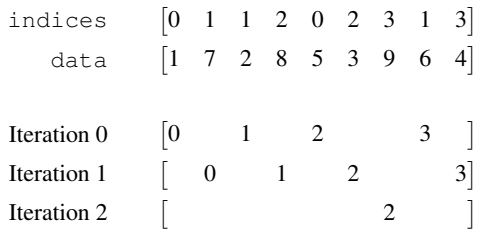


Figure 5: CSR arrays `indices` and `data` and the memory access pattern of the scalar CSR SpMV kernel.

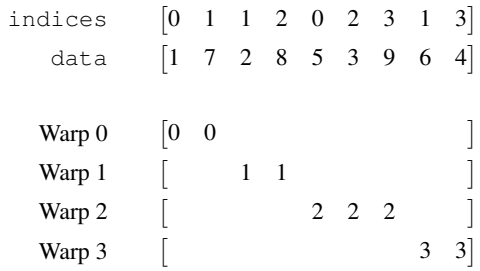


Figure 6: CSR arrays `indices` and `data` and the memory access pattern of the vector CSR SpMV kernel.

3.3 Compressed Sparse Row Format

The *compressed sparse row* (CSR) format is perhaps the most popular general-purpose sparse matrix representation. Like the ELL format, CSR explicitly stores column indices and nonzero values in arrays `indices` and `data`. A third array of row pointers, `ptr`, allows the CSR format to represent rows of varying length. Figure 2 illustrates the CSR representation of an example matrix.

Like DIA and ELL, one simple approach to parallelizing the CSR SpMV operation is to assign one thread to each matrix row. We refer to this method as the *scalar* CSR kernel. While the scalar kernel exhibits fine-grained parallelism, its performance suffers from several drawbacks. The most significant among these problems is the manner in which threads within a warp access the CSR `indices` and `data` arrays. While the column indices and nonzero values for a given row are stored contiguously in the CSR data structure, these values are not accessed simultaneously. Instead, each thread reads the elements of its row sequentially, producing the pattern shown in Figure 5. We have previously described variants of the scalar kernel that use on-chip shared memory to buffer windows of x and improve coalescing of A [16, 10]. While these variants can improve performance in some case by up to a factor of 2 on older hardware generations [10], they make several structural assumptions, notably that A is square and that its rows are all relatively small.

In an alternative to the scalar method, which we call the *vector* kernel, one *warp* processes each matrix row. The vector kernel can be viewed as an application of the vector strip mining pattern to the sparse dot product computed for each matrix row. Unlike the other kernels we have discussed so far, the vector kernel requires coordination among threads in the form of a intra-warp parallel reduction to sum per-thread results together. Note that the parallel reduction changes the order of summation from that of the scalar kernel.

The vector kernel accesses `indices` and `data` contiguously, and therefore overcomes the principal deficiency of the scalar approach. Memory access in Figure 6 are labeled by warp index (rather than iteration number) since the order in which different warps access memory is undefined and indeed, unimportant for coalescing considerations. In this example, no warp iterates more than once while reading the CSR arrays since no row has more than 32 nonzero values. Baskaran and Bordawekar [3] have independently implemented an essentially 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.

Unlike DIA and ELL, the CSR storage format permits a variable number of nonzeros per row without wasted space. While CSR efficiently represents a broader class of sparse matrices, this additional flexibility introduces *thread divergence*. For instance, when the scalar kernel is applied to a matrix with a highly variable number of nonzeros per row, it is likely that many threads within a warp will remain idle while the thread with the longest row continues iterating. Matrices whose distribution of nonzeros per row follows a *power law* distribution are particularly troublesome for the scalar kernel. Since warps execute independently, this form of thread divergence is less pronounced in the vector kernel. On the other hand, efficient execution of the vector kernel demands that matrix rows contain a number of nonzeros greater than the warp size (32). As a result, performance of the vector kernel is sensitive to matrix row size.

3.4 Coordinate Format

The *coordinate* (COO) format is a particularly simple storage scheme. As shown in Figure 2, the arrays: `row`, `indices`, and `data` store the row indices, column indices, and values, respectively, of the nonzero entries. We further assume that entries with the same row index are stored contiguously.

We perform parallel SpMV on COO matrices by assigning one thread to each nonzero. Each thread computes the appropriate $A_{ij}\ x_j$ product and we then perform a *segmented reduction* operation to sum values across threads. Our method is similar to the segmented sum formulation used by Blelloch *et al.* [7, 6] on the CM-2 and Cray C90 and the CUDA-based segmented scan implementation by Sengupta *et al.* [20]. Figure 7 illustrates the memory access pattern of the COO kernel. Here we have illustrated the case of a single executing warp on a fictitious architecture with four threads per warp. The primary advantage of the COO kernel, and segmented operations in general, is that its performance is largely insensitive to irregularity in the underlying data structure. Therefore, our COO method offers robust performance across a wide variety of sparsity patterns.

Our COO kernel is broadly similar to the segmented scan implementation described by Sengupta *et al.* [20], provided in the CUDPP [9] library, and which we have previously used ourselves [10]. However, our current implementation, being tailored specifically to SpMV rather than parallel scan in general, differs in a few concrete ways. We fuse the product formation and reduction into a single kernel. We use segmented reduction, which is moderately simpler and cheaper than segmented scan, and therefore better suited to SpMV. Furthermore, we use row indices in place of other segment descriptors (e.g., head flags), which is another application-specific optimization.

3.5 Hybrid Format

While the ELLPACK format is well-suited to vector and SIMD

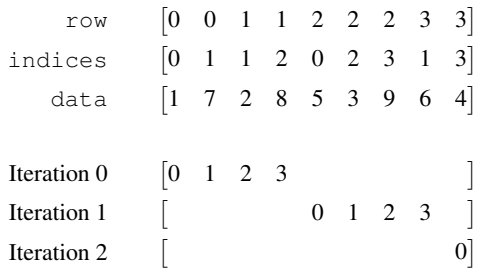


Figure 7: COO arrays `row`, `indices`, and `data` and the memory access pattern of the COO SpMV kernel.

architectures, its efficiency rapidly degrades when the number of nonzeros per matrix row varies. In contrast, the storage efficiency of the COO format is invariant to the distribution of nonzeros per row, and the use of segmented reduction makes its performance largely invariant as well. To obtain the advantages of both, we combine these into a hybrid ELL/COO format.

The purpose of our hybrid (HYB) format is to store the *typical* number of nonzeros per row in the ELL data structure and the remaining entries of *exceptional* rows in the COO format. The typical number of nonzeros per row is often known a priori, as in the case of manifold meshes, and the ELL portion of the matrix is readily extracted. However, in the general case this number must be determined directly from the input matrix. Our implementation computes a histogram of the row sizes and determines the the largest number K such that using K columns per row in the ELL portion of the HYB matrix meets a certain objective measure. Based on empirical results, we assume that the fully-occupied ELL format is roughly three times faster than COO, except when the number of rows is less than (approximately) 4K. Under these modeling assumptions, it is profitable to add a K-th column to the ELL structure when the number of matrix rows with K (or more) nonzeros is at least $\max(4096, M/3)$, where M is the total number of matrix rows. When the population of rows with K or more nonzero values falls below this threshold, the remaining nonzero values are placed into COO.

3.6 Format Summary

Table 1 summarizes the salient features of our SpMV kernels. For the HYB entries, we have assumed that only a small number of nonzero values is stored in the COO portion. Recall that full utilization of the GPU requires many thousands of active threads. Therefore, the finer granularity of the CSR (vector) and COO kernels is advantageous when applied to matrices with a limited number of rows. Note that such matrices are not necessarily small, as the number of nonzero entries per row is still arbitrarily large.

Except for the CSR kernels, all methods benefit from full coalescing when accessing the sparse matrix format. As Figure 5 illustrates, the memory access pattern of the scalar CSR kernel seldom benefits from coalescing. Warps of the vector kernel access the CSR structure in a contiguous but not generally aligned fashion, which implies partial coalescing.

The rightmost columns of Table 1 reflect the computational intensity of the various kernels. In single precision arithmetic, the DIA kernel generally has the lowest ratio of bytes per FLOP, and therefore the highest computational intensity. Meanwhile, the COO format, which explicitly stores (uncompressed) row and column entries, has the lowest intensity.

Note that these figures are only rough approximations to the true computational intensities, which are matrix-dependent. Specifically, these estimates ignore accesses to the `y` vector, the `ptr` array of the CSR format, and the `offset` array of the DIA format. Our matrix-specific bandwidth results in Section 4 provide a more accurate empirical measurement of actual bandwidth usage.

			Bytes/FLOP	
Kernel	Granularity	Coalescing	32-bit	64-bit
DIA	thread : row	full	4	8
ELL	thread : row	full	6	10
CSR (scalar)	thread : row	rare	6	10
CSR (vector)	warp : row	partial	6	10
COO	thread : nonzero	full	8	12
HYB	thread : row	full	6	10

Table 1: Summary of SpMV kernel properties.

3.7 Implementation Notes

Our SpMV kernel implementations use standard CUDA idioms such parallelizing the outermost `for` loop over many threads and computing parallel reductions or segmented scans in shared memory. Vector formats like DIA and ELL are well-suited to this programming model and have straightforward implementations. Similarly, our CSR (scalar) kernel is a direct translation [4] of the standard serial CSR SpMV implementation to CUDA.

However, like Volkov and Demmel [21], we sometimes adopt a *persistent* warp-oriented programming style as opposed to the standard block-oriented approach [17]. Here, "persistent" means that a fixed number of warps are launched and proceed to iterate many times as they process a specific interval of the data set. Letting W denote the number of active warps, a warp of the CSR (vector) kernel processes $O(M/W)$ rows, where M is the total number of matrix rows. Similarly, each warp of the COO kernel processes $O(NNZ/W)$ matrix elements, where NNZ is the total number of nonzero values. In either case the warps persist over the duration of the computation, as opposed to processing a fixed number of results before being retired (as a block) and replaced with another block. This strategy benefits the CSR (vector) kernel as it tends to even out moderate imbalances in per-row workload. In the COO kernel, persistence permits the "carry out" value of one iteration (i.e., a set of 32 elements) to pass into the "carry in" of the next iteration without being written to DRAM, as a non-persistent approach would necessitate.

To facilitate reproducibility, our SpMV implementations are available as open source software. These kernels also provide the basis of CUSP [5], a library of generic parallel algorithms for sparse matrix and graph computations, which is currently in development.

4. PERFORMANCE ANALYSIS

In order to assess the efficiency of these sparse formats and their associated kernels, we have collected SpMV performance data on a broad collection of matrices. All of our experiments are run on a system comprised of an NVIDIA GTX 285 GPU paired with an Intel Core i7 965 CPU. Each of our SpMV performance measurements is an average (arithmetic mean) over 500 trials or 3.0 seconds of execution time, whichever is less. We report computation rates in terms of GFLOP/s, which we determine by dividing the required arithmetic operations—precisely twice the number of nonzeros in the matrix—by the average running time. We characterize mem-

ory throughput using the *effective* bandwidth of the computation: the total number of bytes read/written by all threads divided by average running time. This may differ somewhat from the amount of data being transacted with DRAM when caching is used for the x vector; however, this remains a reasonable model of bandwidth utilization since the cache amplifies bandwidth without reducing latency.

Our measurements do not include time spent transferring data between host (CPU) and device (GPU) memory, since we are trying to measure the performance of the kernels. In many applications, the relevant data structures are created on the device, or reside in device memory for the duration of a series of computations, which renders the cost of such transfers negligible. When the data is not resident in device memory, such as when the device is used to accelerate a single, self-contained component of the overall computation, then transfer overhead is a potential concern. For example, when the device is used to offload an iterative solver (e.g., the conjugate gradient method [19]) then the transfer overhead is generally amortized over 100s or 1000s of solver iterations. In this case, the available host/device bandwidth is not a bottleneck.

4.1 Synthetic Tests

We begin our performance study with a collection of synthetic examples which highlight the tradeoffs discussed in Section 3.6. Efficient SpMV execution on the GPU requires the computation to be divided into thousands of threads of execution. Our target platform, the GTX 285 processor, accommodates up to 30K co-resident threads and typically requires several thousand threads to reach full utilization.

Our SpMV kernels operate on various granularities, either (1) one thread per row, (2) one warp per row, or (3) one thread per nonzero element. Consequently, these kernels expose different levels of parallelism on different matrices. For example, assigning one thread per row, as in ELL, implicitly assumes that the number of rows is large enough to generate sufficient parallel work. In contrast, assigning one thread per nonzero, as in COO, only requires that the total number of nonzero entries is sufficiently large.

Figure 8 explores the relationship between work granularity and efficiency on a sequence of dense matrices stored in sparse format. We vary the dimension of these matrices while holding the number of nonzeros fixed at (approximately) 4M. As expected, COO performance is essentially independent of the matrix dimensions; there is only a slight increase as the number of rows approaches the total number of matrix entries due to coalescing of writes to the y vector. The CSR (vector) kernel performance peaks once the number of rows is comparable to the maximum number of warps that can be co-resident (960), and degrades as the row length drops below 32 since those rows underutilize the 32-thread warps assigned to process them. Similarly, performance of the ELL kernel peaks when the number of matrix rows exceeds the maximum number of threads (30K). Note that the threshold used in the HYB format modeling assumptions (cf. Section 3.5) is related to the crossover point between ELL and COO performance. Indeed, this parameter choice ensures that HYB performance (not shown) is equal to the maximum of the ELL or COO rates across this set of examples. The ELL, COO, and CSR (scalar) kernels converge to roughly the same level when there is only 1 nonzero per row. While in most practical usage scenarios, the decomposition strategies employed by CSR (vector) and ELL expose sufficient parallelism to fully utilize the processor, this example highlights the value of robust parallel strategies in atypical scenarios.

In addition to exposing a high degree of parallelism, efficient SpMV kernels must also effectively balance the per-thread workload. This is particularly true for threads within a warp, where inter-thread imbalance guarantees underutilization of the processor cores. Figure 9 demonstrates the effect of varying row length distribution on kernel performance. We construct a set of matrices with 30K rows of $32+\delta$ nonzeros each, where δ is a random variable drawn from a Pareto distribution. As the Pareto parameter k is increased from 1, the variance in row lengths decreases rapidly. The Pareto distribution is a power law probability distribution that models event frequencies in various scientific phenomena. Therefore, there can be a drastic difference in size between the largest rows and the average. Within certain application areas it is not uncommon to find sparse matrices with such power-law row length distributions.

Figure 9 shows the results of this experiment. Again as expected, COO performance is almost completely insensitive to row length distribution. Both CSR kernels suffer significantly at high variance ($k = 1$). The effect on the vector kernel is less severe because (1) CSR (vector) performance improves with row size and (2) separate warps are independent, whereas threads within a warp must wait until all threads have finished their respective rows. Since there is no implicit synchronization between different warps, we allow warps of the vector kernel to iterate over several rows, thus smoothing out load imbalance. Although warps within the same block are retired together, this looser form of (unnecessary) synchronization

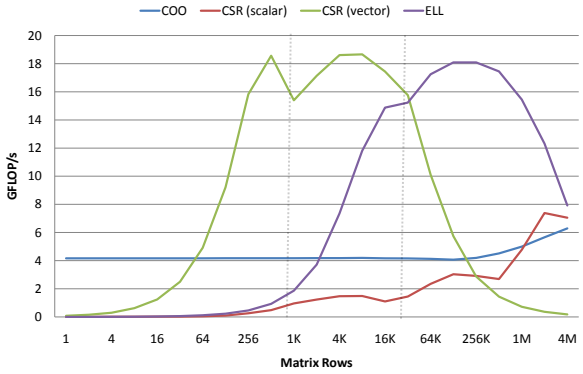


Figure 8: SpMV performance on dense matrices with a fixed number of nonzeros and variable matrix dimensions. The vertical lines mark the maximum number of warps (960) and threads (30K) that may be simultaneously co-resident.

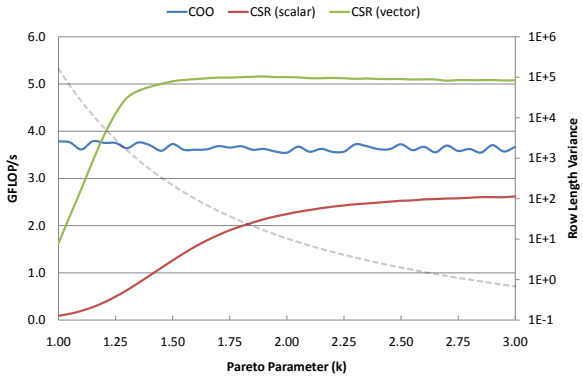


Figure 9: SpMV performance with Pareto-distributed row lengths. Variance in row length (dashed curve) is substantial.

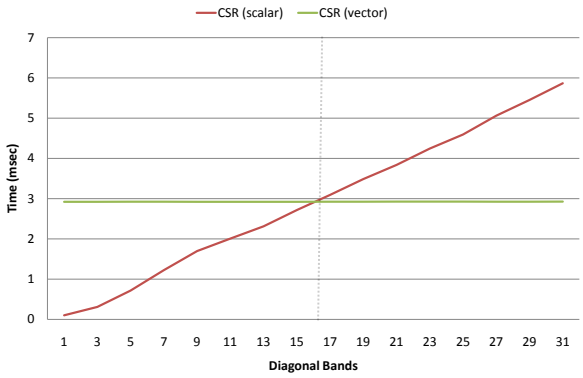


Figure 10: Time per SpMV operation on banded matrices with variable bandwidth.

is less detrimental to performance than the finer-grained implicit synchronization imposed on threads within a warp.

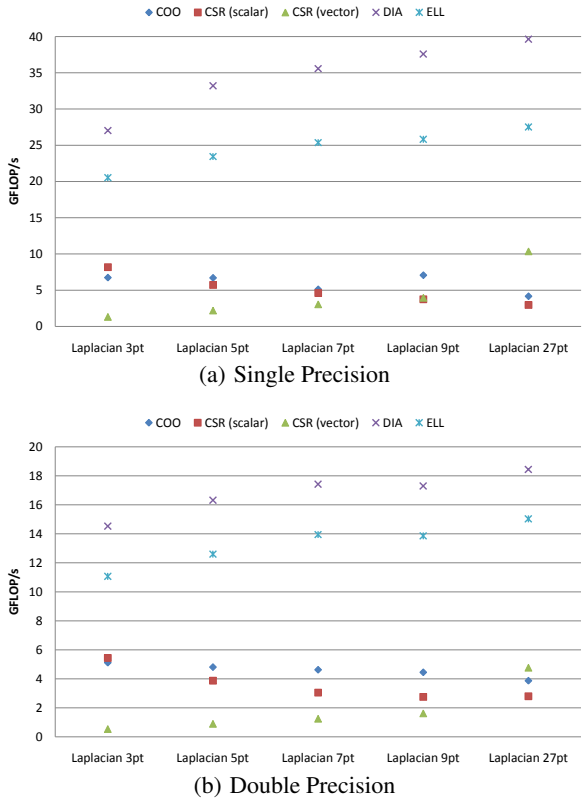
To isolate the effect of row length on CSR kernel performance we present Figure 10, which shows the running time of the CSR kernels on 256K-by-256K matrices with varying numbers of diagonal bands. With the exception of the first and last few rows, the number of nonzeros per row is directly proportional to the number of bands. The time required by the CSR (vector) kernel is virtually constant and does not depend on the number of bands. In contrast, the time required by the CSR (scalar) slowly increases and then enters a regime of linear growth. In both kernels, coalescing of memory operations is responsible for the observed performance. Since CSR (vector) accesses contiguous sections of memory it benefits from coalescing, which allows large segments of memory to be accessed with little or no additional cost. When processing a matrix with bandwidth K, the consecutive threads of the CSR (scalar) kernel access memory with stride K. Therefore, when K is small, CSR (scalar) receives some benefit from memory coalescing. This fact accounts for the relative efficiency of $K = 1, 3, 5, 7$ as compared to the linear scaling regime for $K > 16$. The banded matrices serve to illustrate a case where the available parallelism is high, variance in row length is low, but yet efficiency of the SpMV kernels suffers due to poor utilization of computational resources and the memory subsystem.

Matrix	Grid	Diagonals	Nonzeros
Laplace 3pt	$(1{,}000{,}000)$	3	2,999,998
Laplace 5pt	$(1{,}000)^2$	5	4,996,000
Laplace 7pt	$(100)^3$	7	6,940,000
Laplace 9pt	$(1{,}000)^2$	9	8,988,004
Laplace 27pt	$(100)^3$	27	26,463,592

Table 2: Laplacian operators discretized as k-point finite difference stencils on regular grids.

4.2 Structured Matrices

We continue our performance study with a set of structured matrices that represent common stencil operations on regular 1-, 2-, and 3-dimensional grids. Our test set, listed in Table 2, consists of standard discretizations of the Laplace operator in these dimensions. The number of points in the stencil is precisely the number of occupied matrix diagonals, and thus the number of nonzeros per



(a) Single Precision

(b) Double Precision

Figure 11: SpMV throughput on structured matrices.

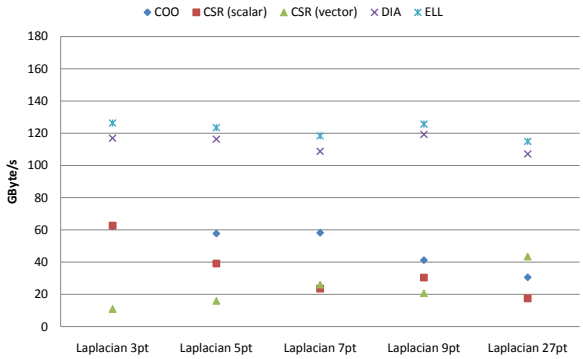
row, excepting rows corresponding to grid boundaries. Figure 11 shows SpMV performance for these matrices.

The DIA kernel, which is explicitly intended for this case, offers the best performance in all examples. On the 27-point example, the DIA kernel reaches 39.6 GFLOP/s and 18.4 GFLOP/s in single and double precision respectively. ELL performance is similar to DIA, although uniformly slower, because of the overhead of using explicit column indices. HYB performance (not shown) is identical to ELL performance on structured matrices, since all nonzeros would be stored in its ELL portion.

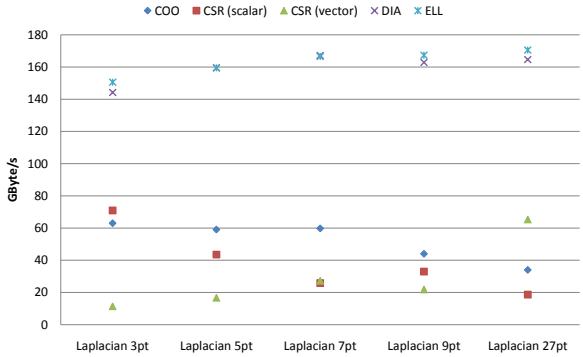
Although not competitive with DIA and ELL, the COO kernel exhibits stable performance across the structured matrices. The CSR (vector) kernel is also uncompetitive because it uses one 32-thread warp per row, and all matrices here have fewer than 32 nonzeros per row. It therefore suffers from underutilization.

Observe that CSR (scalar) performance decreases as the number of diagonals increases. The number of nonzeros across all rows in each matrix is essentially constant, so execution divergence is minimal. Memory divergence is the cause of this decline, since the increasing number of nonzeros per row gradually eliminates the possibility for coalesced loads, just as shown in Figure 10.

Although GFLOP/s is our primary metric for computational throughput, memory bandwidth utilization tells us how efficiently we are using the processor. Figure 12 reports measured memory bandwidth for single precision SpMV with and without caching of x. The maximum theoretical memory bandwidth of the GTX 285 is 159.0 GBytes/s, and without the aid of caching, the DIA and ELL kernels deliver as much as 73.5% and 79.5% of this figure respectively. With caching of x enabled, the maximum effective bandwidth of DIA is 167.1 GByte/s and 170.5 GByte/s for ELL, which



(a) Without Cache



(b) With Cache

Figure 12: Bandwidth results for structured matrices with single precision values.

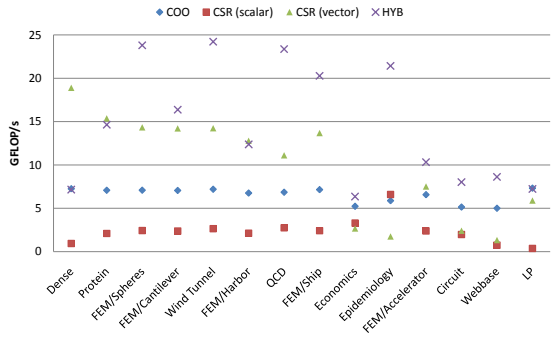
are 105.1% and 107.2% of peak, respectively. In other words, there is enough re-use of x through the cache to exceed the physical peak bandwidth.

While the peak rate of double precision arithmetic on the GTX 285 is an order of magnitude less than the peak single precision rate, SpMV performance is generally bandwidth-limited, and therefore does not approach the limits of floating point throughput. Indeed, DIA double precision (uncached) performance in the 27-point example is 45.9% that of the single precision result, which nearly matches the ratio of bytes per FLOP of the two kernels (50%) as listed in Table 1. For the ELL kernel, a similar analysis suggests a relative performance of 60.0% (double precision to single precision) which again agrees with the 59.2% observed.

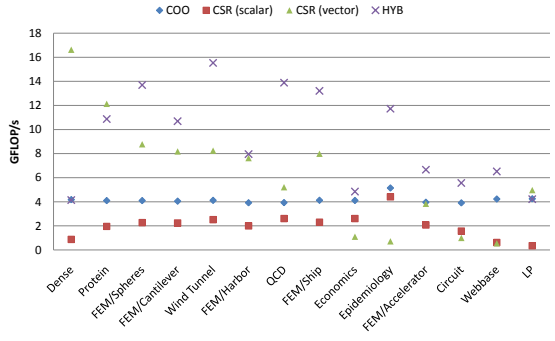
Since the remaining kernels are not immediately bandwidth-limited, we cannot expect a direct correspondence between relative performance and computational intensity. In particular, the CSR (scalar) kernel retains 92.5% of its single precision performance, while computational intensity would suggest a 60.0% figure. This anomaly is explained by the fact that *uncoalesced* double-word memory accesses are inherently more efficient than uncoalesced single-word accesses on a memory bandwidth basis (cf. Figure 1).

4.3 Unstructured Matrices

Our unstructured matrix performance study considers the same 14 matrix corpus (cf. Table 3) used by Williams *et al.* [23] in their multicore benchmarking study. Figure 13 reports single and double precision performance. In contrast to the structured case, the unstructured performance results are varied, with no single kernel outperforming all others. The HYB format achieves the highest

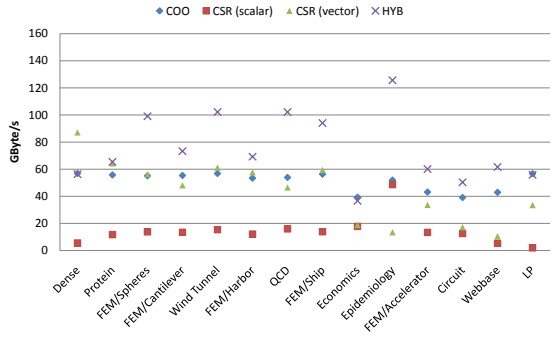


(a) Single Precision

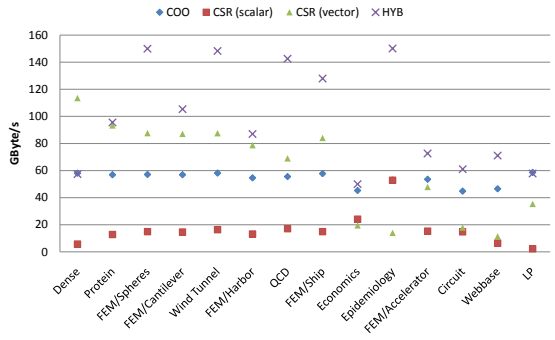


(b) Double Precision

Figure 13: SpMV throughput on unstructured matrices.



(a) Without Cache



(b) With Cache

Figure 14: Bandwidth results for unstructured matrices with single precision values.

absolute performance, reaching 24.2 GFLOP/s in the Wind Tunnel example and over 20 GFLOP/s in five of fourteen examples using single precision arithmetic. In this case, the ELL portion of the HYB format stores 99.4% of the nonzero entries. The HYB format also performs relatively well on highly irregular matrices such as LP, where the COO portion of the HYB format stores the vast majority of nonzero entries (99.1%).

Although the dense 2,000-by-2,000 matrix is efficiently stored in ELL (and therefore HYB) format, it does not perform well in this case. This is because the 2000 threads used (one per row) provide insufficient parallelism to fully utilize the GPU.

The CSR (vector) kernel is significantly faster than HYB on the 2,000 row dense matrix. Here the finer granularity of the vector kernel (one warp per row), decomposes the SpMV operation into 64,000 distinct threads of execution, which is more than sufficient to fill the device. As with the structured matrices, the vector kernel is sensitive to the number of nonzeros per matrix row. On seven examples with an average of 50 or more nonzeros per row, the vector kernel performs no worse than 12.8 GFLOP/s. Conversely, the matrices with fewer than four nonzeros per row, Epidemiology and Webbase, contribute the worst results, at 1.8 and 1.3 GFLOP/s respectively.

Compared to the other kernels, COO performance is relatively stable across the test cases. The COO kernel performs particularly well on the LP matrix, which proves especially challenging for the other methods. Although LP is the only instance where COO exhibits the best performance, it is clearly a robust fallback for matrices with pathological row length distributions.

With caching disabled, the memory bandwidth utilization of the HYB kernel, shown in Figure 14, exceeds 100 GByte/s, or 63% of the theoretical maximum, on several unstructured matrices. The bandwidth disparity between structured and unstructured cases is primarily attributable to the lack of regular access to the x vector. Caching mitigates this problem to a degree, improving computational throughput by an average of 30% and 25.0% in single and double precision respectively. Among the fastest five examples, caching improves (effective) bandwidth utilization from an average of 109.8 GBytes/s to 145.0 GBytes/s, or from 69.1% to 91.2% of the theoretical peak.

Together, our CSR and HYB kernels surpass the 10.0 GFLOP/s mark in eight of the fourteen unstructured test cases using double precision arithmetic. As shown in Figure 13, the CSR (vector) kernel achieves the highest absolute performance at 16.6 GFLOP/s on the Dense matrix. Wind Tunnel represents best-case HYB performance at 15.5 GFLOP/s. Again, COO performance is stable,

Matrix	Rows	Columns	NNZ	NNZ/Row
Dense	2,000	2,000	4,000,000	2000.0
Protein	36,417	36,417	4,344,765	119.3
FEM/Spheres	83,334	83,334	6,010,480	72.1
FEM/Cantilever	62,451	62,451	4,007,383	64.1
Wind Tunnel	217,918	217,918	11,634,424	53.3
FEM/Harbor	46,835	46,835	2,374,001	50.6
QCD	49,152	49,152	1,916,928	39.0
FEM/Ship	140,874	140,874	7,813,404	55.4
Economics	206,500	206,500	1,273,389	6.1
Epidemiology	525,825	525,825	2,100,225	3.9
FEM/Accelerator	121,192	121,192	2,624,331	21.6
Circuit	170,998	170,998	958,936	5.6
Webbase	1,000,005	1,000,005	3,105,536	3.1
LP	4,284	1,092,610	11,279,748	2632.9

Table 3: Unstructured matrices used for performance testing.

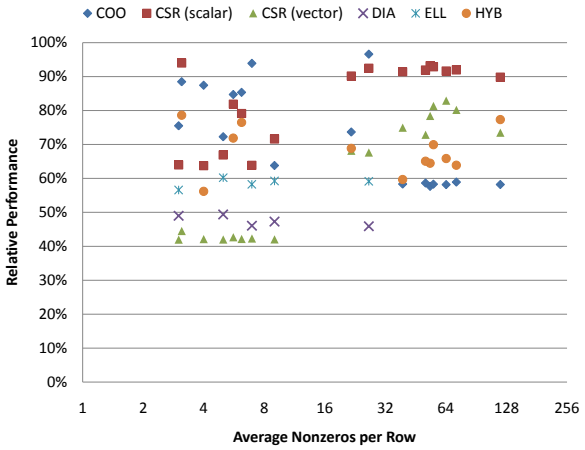


Figure 15: Double precision performance relative to single precision performance.

varying from a minimum of 3.9 to a maximum 5.1 GFLOP/s, with most results close to the 4.1 GFLOP/s mark.

The relative performance between double and single precision performance is shown in Figure 15, and follows the same pattern apparent in the structured case. The median double precision HYB performance is 65.5% of the corresponding single precision result. For CSR (vector), the median is 73.7%. The CSR (scalar) kernel retains 91.8% of its single precision performance, again owing the relative bandwidth efficiency of divergent double-word memory access to single-word accesses.

4.4 Performance Comparison

We have already demonstrated that our SpMV kernels are efficient, achieving significant fractions of theoretical peak memory bandwidth. In terms of absolute performance, our implementations are also very competitive with optimized SpMV kernels for other architectures. Figure 16 compares double precision SpMV performance of our CSR (vector) and HYB kernels to the results obtained by Williams *et al.* [23] on several single and dual socket multicore systems.

Examining the single socket platforms we find that the GPU-based methods offer the best performance in all 14 test cases, often by significant margins. This is fundamentally due to the much higher bandwidth capacity of the GPU. The (single) GPU also outperforms all dual socket systems in all considered cases.

Comparing the best GPU kernel (either CSR (vector) or HYB) to these multicore kernels, we find that the median GPU performance advantages over the single socket systems are: 2.83× for Cell, 11.03× for Opteron, 12.30× for Xeon, and 3.13× for Niagara. For dual-processor platforms, the median GPU performance factors are: 1.56× for Cell, 5.38× for Opteron, and 6.05× for Xeon.

5. FURTHER OPTIMIZATIONS

The SpMV kernels we have investigated are designed to deliver reliably good performance on the broadest possible range of matrices. They make only minimal assumptions about the matrices they encounter. Consequently, they can potentially be extended with a number of complementary optimizations for restricted subclasses of the general SpMV problem.

We have restricted our attention to optimizations that perform no

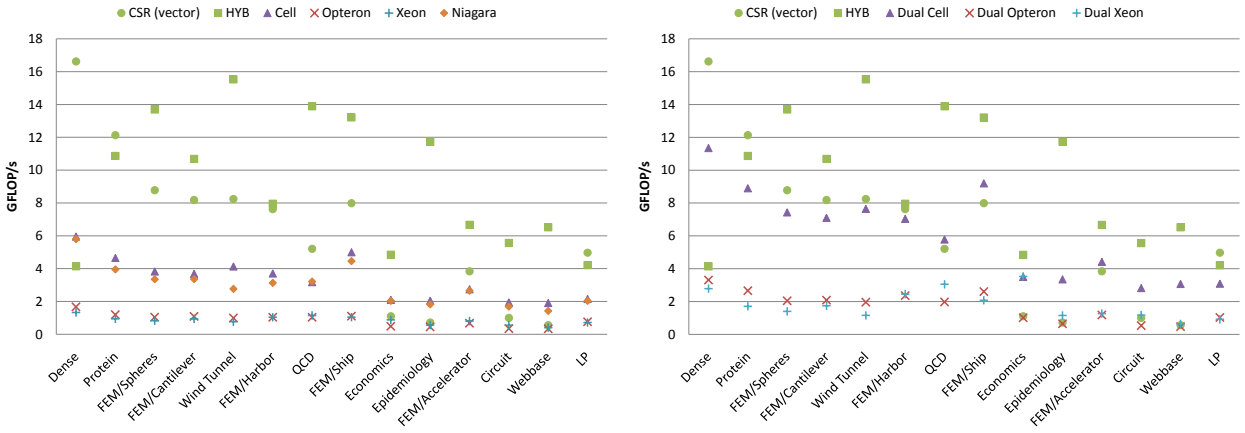


Figure 16: CSR and HYB kernels compared to several single and dual socket multicore platforms.

global restructuring of the matrix. Overall performance on a given matrix could potentially be improved by careful reordering of the rows and columns. For matrices arising from manifold meshes, as in finite element discretizations, partitioning the matrix into spatially compact pieces, as in our packet format [4], can be potentially beneficial. Of course any such global restructuring incurs a potentially non-trivial cost for restructuring, and so such techniques are most attractive when a single matrix will be reused many times.

Block formats such as Block CSR or Variable-Block CSR [12] can deliver higher performance on both CPU [23] and GPU [8] architectures, particularly for matrices arising in vector-valued problems. Blocking can be valuable because it (1) decreases indexing overhead, (2) increases the locality of memory accesses to the source vector, and (3) facilitates re-use of the source vector elements. The techniques we have applied to scalar formats are largely compatible with block format extensions.

Special handling for the case of multiple source vectors, often called *block vectors*, is another standard SpMV optimization. Like blocked matrix formats, block vectors improve memory locality and provide additional opportunities for data re-use. In the context of iterative methods for linear systems, this situation occurs when solving for several right-hand-sides simultaneously (i.e., $AX = B$ where B has multiple columns). Furthermore, in the case of eigensolvers such as the LOBPCG [13], it is not uncommon to utilize block vectors with ten or more columns.

6. CONCLUSIONS

We have explored several efficient implementation techniques for sparse matrix-vector multiplication (SpMV) in CUDA. Our kernels exploit fine-grained parallelism to effectively utilize the computational resources of the GPU. The efficiency of our kernels is demonstrated by the fact that they achieve high bandwidth utilization on both structured and unstructured matrices. We have also demonstrated that they achieve excellent absolute performance, outperforming highly optimized kernels on several multicore platforms. Our kernels are hand-written and relatively straightforward yet still deliver good bandwidth utilization. This reflects the design of GPU architectures, which are specifically designed to deliver high performance on straightforward kernels that exhibit high degrees of data parallelism.

The DIA and ELL techniques are well-suited to matrices obtained from structured grids and semi-structured meshes, while the COO approach based on segmented reduction performs consistently across a broad spectrum of matrices. We parallelize computation on the popular CSR format using both scalar and vector approaches. The scalar approach using one thread per matrix row does not benefit from memory coalescing, which consequently results in low bandwidth utilization and poor performance. On the other hand, the vector approach ensures contiguous memory access, but leads to a large proportion of idle threads when the number of nonzeros per row is smaller than the warp size. Performance of the scalar method is rarely competitive with alternative choices, while the vector kernel excels on matrices with large row sizes. We have designed our HYB format, representing the matrix in ELL and COO portions, to combine the speed of ELL and the flexibility of COO. As a result, HYB is generally the fastest format for a broad class of unstructured matrices.

Acknowledgements

We thank Sam Williams and Richard Vuduc for providing the matrices and multicore performance results used in our performance study.

7. REFERENCES

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[2] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. Solving dense linear systems on graphics processors. In *Proc. 14th Int'l Euro-Par Conference*, volume 5168 of *Lecture Notes in Computer Science*, pages 739–748. Springer, Aug. 2008.

[3] M. M. Baskaran and R. Bordawekar. Optimizing sparse matrix-vector multiplication on GPUs. IBM Research Report RC24704, IBM, Apr. 2009.

[4] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[5] N. Bell and M. Garland. CUSP : Generic parallel algorithms for sparse matrix and graph computations. `http://code.google.com/p/cusp-library/`, 2009-.

[6] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, Apr. 1994.

[7] G. E. Blelloch, M. A. Heroux, and M. Zagha. Segmented operations for sparse matrix computation on vector multiprocessors. Technical Report CMU-CS-93-173, School of Computer Science, Carnegie Mellon University, Aug. 1993.

[8] L. Buatois, G. Caumon, and B. Lévy. Concurrent number cruncher - a GPU implementation of a general sparse linear solver. *International Journal of Parallel, Emergent and Distributed Systems*, to appear.

[9] CUDPP: CUDA data parallel primitives library. `http://www.gpgpu.org/developer/cudpp/`, 2008.

[10] M. Garland. Sparse matrix computations on manycore GPU's. In *DAC '08: Proc. 45th Annual Design Automation Conference*, pages 2–6, New York, NY, USA, 2008. ACM.

[11] R. Grimes, D. Kincaid, and D. Young. ITPACK 2.0 User's Guide. Technical Report CNA-150, Center for Numerical Analysis, University of Texas, Aug. 1979.

[12] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, 2004.

[13] A. V. Knyazev. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM Journal on Scientific Computing*, 23(2):517–541, 2002.

[14] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[15] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, Mar/Apr 2008.

[16] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, Mar/Apr 2008.

[17] NVIDIA Corporation. *NVIDIA CUDA Programming Guide*, June 2008. Version 2.0.

[18] Y. Saad. SPARSKIT: A basic tool kit for sparse computations; Version 2, June 1994.

[19] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial Mathematics, 2003.

[20] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *Graphics Hardware 2007*, pages 97–106. ACM, Aug. 2007.

[21] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proc. 2008 ACM/IEEE Conference on Supercomputing*, pages 1–11, Nov. 2008.

[22] R. W. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California, Berkeley, 2003.

[23] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proc. 2007 ACM/IEEE Conference on Supercomputing*, 2007.