High-Performance Sparse Matrix-Vector Multiplication on GPUs for Structured Grid Computations

Jeswin Godwin Justin Holewinski P. Sadayappan
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{godwin, holewins, saday}@cse.ohio-state.edu

ABSTRACT

In this paper, we address efficient sparse matrix-vector multiplication for matrices arising from structured grid problems with high degrees of freedom at each grid node. Sparse matrix-vector multiplication is a critical step in the iterative solution of sparse linear systems of equations arising in the solution of partial differential equations using uniform grids for discretization. With uniform grids, the resulting linear system $A\vec{x} = \vec{b}$ has a matrix A that is sparse with a very regular structure. The specific focus of this paper is on sparse matrices that have a block structure due to the large number of unknowns at each grid point. Sparse matrix storage formats such as Compressed Sparse Row (CSR) and Diagonal format (DIA) are not the most effective for such matrices.

In this work, we present a new sparse matrix storage format that takes advantage of the diagonal structure of matrices for stencil operations on structured grids. Unlike other formats such as the Diagonal storage format (DIA), we specifically optimize for the case of higher degrees of freedom, where formats such as DIA are forced to explicitly represent many zero elements in the sparse matrix. We develop efficient sparse matrix-vector multiplication for structured grid computations on GPU architectures using CUDA [25].

1. INTRODUCTION

Structured grid computations form the basis for many important applications in computational science, and there is great demand for accelerating these computations on modern heterogeneous architectures, such as GPU accelerators. Numeric partial differential equation solvers often employ structured grids to solve problems such as environmental modeling [20]. Numerical computation software that can solve structured grid problems such as PETsc [2] have recently begun to implement portions of their solvers on GPU accelerators, and achieving high-performance on these devices has been shown to be a hard problem.

The formulation of many structured grid problems involves the formation of a sparse matrix which models the relationship between neighboring points in the problem grid as a set of linear equations. The program then solves the linear system $A\vec{x} = \vec{b}$ by performing sparse matrix-vector multiplication repeatedly until the found $\vec{x}$ satisfies a convergence condition, typically involving an error tolerance.

For such solvers, the sparse matrix-vector multiplication is the primary bottleneck. A large body of research has investigated the optimization of sparse matrix-vector multiplication on heterogeneous architectures, including GPU architectures. This research has involved the optimization of sparse matrix-vector multiplication using architecture-specific optimizations as well as new layout formats for the sparse matrix. However, the bulk of this research has focused on sparse matrices that encode linear equations of scalar quantities. Many applications solve structured grid problems on higher-dimensional entities, such as vectors and matrices. Sparse matrices that encode linear equations on vectors and matrices provide additional opportunities for efficient data layout for GPU architectures.

In this paper, we propose a new storage format, which we call Column Diagonal Storage (CDS), for sparse matrices that result from stencil computations on structured grids. This new storage format is optimized for efficient storage and computation on GPU architectures for diagonal sparse matrices that arise from structured grid computations on higher-dimensional entities.

This paper makes the following contributions:

- Proposes a new storage format for block-diagonal sparse matrices which leads to higher performance for structured grid computations on GPU architectures
- Performs a detailed performance analysis of the proposed matrix layout extension on many problem sizes and dimensionalities

The rest of the paper is organized as follows. In Section 2, we present background material about GPU computing, structured grid computations, and sparse matrix storage formats. In Section 3, we present our sparse matrix storage format for structured grid computations on GPUs. In Section 4, we present an experimental evaluation of our storage format. Related work is then presented in Section 5. Finally, in Section 6, we conclude.

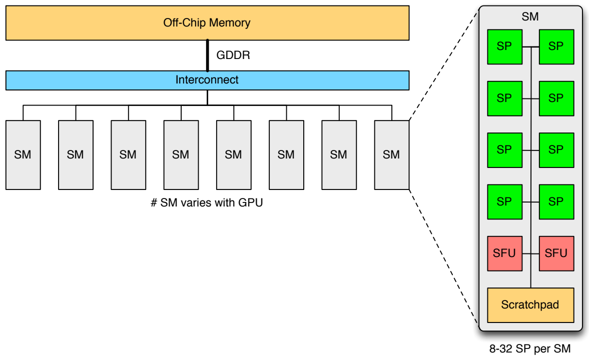

Figure 1: Basic architecture of nVidia GPU architectures

2. BACKGROUND

In this section, we provide background information on structured grid computations, sparse matrix-vector multiplications, and their implementation on GPU architectures. We first discuss GPU architectures and their programming models. We then address structured grid computations and the sparse linear algebra systems that arise n this context.

2.1 Graphics Processing Units (GPUs)

Graphics Processing Units (GPUs) are massively-threaded, many-core architectures with peak floating-point throughput over 1 TFLOP/s. In recent years, these devices have become increasingly programmable and therefore useful for general purpose workloads. The HPC community in particular has invested many resources in improving the performance of computational programs on GPU architectures. For the purposes of presentation for this paper, we will only consider nVidia GPU devices.

These devices offer interesting challenges for achieving high-performance, but also have the potential to greatly accelerate numeric programs compared to CPU implementations. In nVidia GPU architectures, there are on the order of 512 streaming processors, arranged into blocks of 8-32 per streaming multi-processor. Individual thread blocks are scheduled onto the streaming multi-processors, and cannot migrate after being scheduled. However, a single multiprocessor can concurrently handle several blocks of threads. Figure 1 depicts the basic architecture.

Each thread has access to the GPU's global, off-chip memory, as well as a shared scratch-pad memory which is shared among all threads within a block. Thus, threads within a block can communicate and exchange data through shared memory. Threads can also issue barriers that cause all threads within a block to synchronize. However, thread synchronization is, in general, not feasible across blocks.

2.1.1 Programming Model

GPU devices are commonly programmed using low-level programming models, of which the two most common are CUDA [25] and OpenCL [18, 26]. In both models, the programmer writes an imperative program (called a *kernel*) that is executed by each thread on the device. Threads are spawned in blocks, which are 1-, 2-, or 3-dimensional rectangular groups of cooperative threads. These blocks are further arranged into a 1- or 2-dimensional grid of blocks.

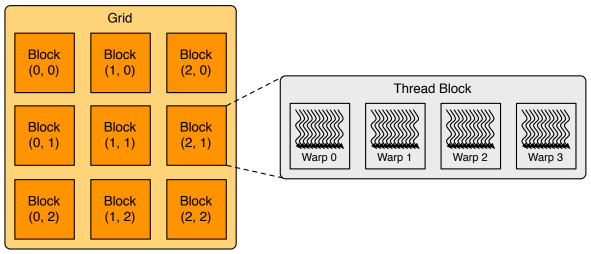

Figure 2: Distribution of threads in a GPU kernel

Both the size and number of thread blocks is fixed when launching a GPU kernel and cannot be changed after the threads have launched. This distribution of threads is depicted in Figure 2.

To hide memory latencies, it is common to schedule hundreds of threads per block. The threads are scheduled on the streaming processors at the granularity of warps, which are 32 threads. The streaming processors are time-shared between all warps currently active on the streaming multiprocessor, and thread context switches incur no overhead.

2.1.2 Challenges

When programming GPU devices, there are several sources of inefficiencies that are somewhat unique to GPU architectures. GPU devices provide a very high off-chip memory bandwidth (144 GB/sec for Tesla C2050), but this bandwidth is only achievable with *coalesced* access. If threads executing concurrently on a streaming multi-processor issue requests to contiguous memory locations, the requests can be fulfilled concurrently by the memory system. If the requests are not to contiguous memory, the memory system will be forced to serialize the requests, leading to wasted bandwidth.

Additionally, the shared scratchpad memory is a banked memory architecture. If concurrently executing threads in a block make requests to shared memory locations in the same bank, a bank conflict will occur and the requests will be serialized. Therefore, to achieve efficient access to shared memory, concurrently executing threads should access memory belonging to different banks. Finally, all concurrently executing threads in a block must issue the same instruction to the streaming processors. If threads exhibit divergent control flow, then their execution will be serialized on the streaming multi-processor, leading to wasted compute resources and inefficiency.

2.2 Structured Grid Computations

Structured grid problems occur in many scientific domains, including electromagnetics [22, 24], computational fluid dynamics (CFD) [1], environmental modeling [20] and astrophysics [14]. Such problems are often formulated as discretizations of partial differential equations on structured grids that represent surfaces or spaces. Generally, a structured grid problem involves the application of a stencil operator to every point in an N-dimensional space.

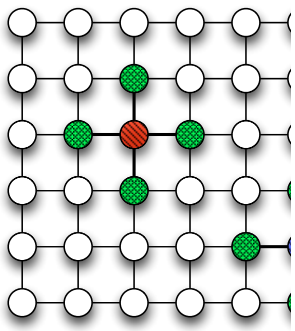

Figure 3: Application of the 2-D Lapla
2-D structured grid.

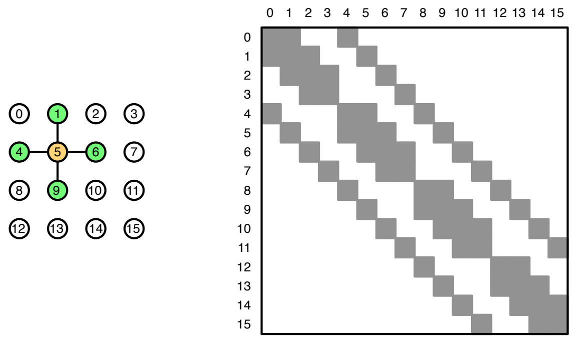

Figure 4: Example of a structured grid computation.

2.2.1 *Stencil Operators*

A stencil operator is defined as a functio
value for a point in the structured grid bas
value of the grid point and its neighboring
2-dimensional Laplacian stencil operation

$$g_{i,j} = c_{i,j}^{i,j} \cdot f_{i,j} + c_{i,j}^{i-1,j} \cdot f_{i-1,j} + c_{i,j}^{i+}$$
$$c_{i,j}^{i,j-1} \cdot f_{i,j-1} + c_{i,j}^{i,j+1} \cdot f_{i,j+1}$$

In this formulation, f is a 2-D array of v
The notation $f_{i,j}$ denotes the value at i
coefficient $c_{l,m}^{i,j}$ corresponds to a value tha
points (i, j) and (l, m) in the grid, and n
each pair of points in the grid. We comp
each point in the grid in g from the previ
f.

Figure 3 shows a visual depiction of t
stencil operator on a small 2-D grid. Ap
to the red point computes a new value bas
value at the red point and the surround
Care must be taken around the grid bou
The blue point shows the stencil operator
point on the boundary. In this case, the
point is ignored and does not contribute to
However, different types of structured gr
impose different handling of boundary cel

- Wrap around to the other side of th
- Clamp to the nearest grid point
- Assign a constant to boundary cells

2.2.2 *Solving Stencil Problems*

Depending on the type of problem being solved, there are two broad approaches for solving structured grid problems. The first explicitly applies the stencil operator to every point in the grid, producing a new grid of values from the original grid. In imperative languages like C, this would be implemented as a series of `for` loops that scan every point in the grid and applies the stencil operator. However, not all types of structured grid problems can be effetively solved using this model.

The second approach involves the formation of a linear system of equations from the application of the stencil operator on each grid point. The linear equations are encoded into a linear system $A\vec{x} = \vec{b}$. In many cases, $\vec{b}$ is known, and the goal is to find $\vec{x}$ within some acceptable tolerance. An initial guess $\vec{x}_0$ is used, and the system is repeatedly solved using the result from the previous iteration until the tolerance is met ($\vec{b} - \vec{x}_i \leq \vec{\epsilon}$). Such systems often require many iterations to convergence, making the matrix-vector multiply a very performance-critical component of the solver.

The sparse matrices formed for these problems have a very regular structure. Since grid points are only related to their immediate neighbors, the sparse matrix will have diagonals of non-zero entries at regular intervals. The distance between these diagonals is dependant on the shape of the stencil operation, but in general there are m diagonals, where m is the number of "points" in the stencil. The number of "points" in a stencil is the number of grid elements touched by each application of the stencil function.

As an example, consider the computation in Figure 4. A 2-D Laplacian stencil operation is applied to a 4×4 structured grid of points. The computation is performed by scanning the entire grid, and applying the shown stencil. The tan point is an arbitrarily selected point in the grid, and the green points are the additional points needed to compute the new value for the tan point. We see that each point needs a total of five grid points in the computation, except for the boundary points where points lying outside of the grid are not considered.

This problem is formulated as a linear system of equations by generating a 16×16 matrix of coefficients. Generally, for an $N \times M$ grid, the matrix will be of size $(N \cdot M) \times (N \cdot M)$. Each row and column corresponds to one grid point, and the element at index (i, j) is the coefficient that relates grid point i to grid point j (e.g. Equation 1). Note that this naturally extends to grids of any dimensionality. The grid points are numbered, so an N-dimensional grid with bounds $(n_0, n_1, ..., n_N)$ will require a matrix of size $(n_0 \cdot n_1 \cdot ... \cdot n_i) \times (n_0 \cdot n_1 \cdot ... \cdot n_i)$.

The resulting matrix for the sample problem is shown in Figure 4. The gray elements represent non-zero values in the sparse matrix. While the matrix is sparse, it does have a very regular structure. In particular, there is one diagonal for each stencil point.

$$\begin{pmatrix} 3 & 0 & 5 & 0 & 0 \\ 0 & 2 & 0 & 1 & 0 \\ 1 & 0 & 6 & 0 & 9 \\ 0 & 8 & 0 & 2 & 0 \\ 0 & 0 & 3 & 0 & 7 \end{pmatrix}$$

(a) Original

$$\begin{aligned} arr &: [3,5,2,1,1,6,9,8,2,3,7] \\ col &: [0,2,1,3,0,2,4,1,3,2,4] \\ row &: [0,2,4,7,9,12] \end{aligned}$$

(b) CSR

$$\begin{aligned} data &: \begin{pmatrix} x & 3 & 5 \\ x & 2 & 1 \\ 1 & 6 & 9 \\ 8 & 2 & x \\ 3 & 7 & x \end{pmatrix} \\ diag &: [-2,0,2] \end{aligned}$$

(c) DIA

$$\begin{aligned} data &: \begin{pmatrix} 3 & 5 & x \\ 2 & 1 & x \\ 1 & 6 & 9 \\ 8 & 2 & x \\ 3 & 7 & x \end{pmatrix} \\ col &: \begin{pmatrix} 0 & 2 & x \\ 1 & 3 & x \\ 0 & 2 & 4 \\ 1 & 3 & x \\ 2 & 4 & x \end{pmatrix} \end{aligned}$$

(d) ELLPACK

Figure 5: Examples of sparse matrix layouts

2.2.3 Degrees of Freedom

An interesting case arises when we consider structured grids containing higher-dimensional entities. For example, many applications of structured grids use vector quantities at grid points and matrices as the coefficients that relate the grid points. The linear equations formed for such problems are thus linear equations of vectors and matrices, requiring matrix-vector operations at each stencil application instead of simple scalar arithmetic. For such problems, if the vectors are of size n and the matrices are of size $n \times n$, we say that the problem has n *degrees of freedom*.

A large body of research has focused on optimizing sparse matrix-vector multiplication for different matrix storage formats on various architectures, including SMP systems and GPU accelerators. However, much of this research has focused on the scalar case (one degree of freedom in the problem). As we will show in Section 3, the storage format for sparse matrices representing structured grid problems with more than one degree of freedom has a significant impact on both the memory efficiency of the storage format, as well as the performance of sparse matrix-vector multiplication code.

2.3 Sparse Matrix Storage Formats

Prior work has proposed several sparse matrix storage formats, each optimized for different use-cases. The performance of these storage formats has been analyzed for various problem domains by many researchers. In this section, we explore three of the previously proposed formats: Compressed Sparse Row (CSR), Diagonal (DIA), and ELLPACK.

2.3.1 CSR

Compressed Sparse Row (CSR) is a matrix storage technique that aims to minimize the storage requirement for general sparse matrices. In this format, the non-zero elements of the sparse matrix are collapsed into a dense array (arr), and two additional vectors are used to track the column index of each non-zero element (col) and the offset into arr of the start of each row (row).

If N_{nz} is the number of non-zero elements in the sparse matrix and N_r is the number of rows, $|arr| = |col| = N_{nz}$ and $|row| = N_r + 1$. For every element in arr, the col vector stores the column number of the element in the sparse matrix. The row vector stores the offset into arr for the start of each row, with the convention that $row[N_r] = N_{nz} + 1$. Otherwise, we would not know the value of N_{nz}.

As an example, consider the CSR format of the sparse matrix in Figure 5. The original sparse matrix is shown in (a) and the CSR representation is shown in (b).

2.3.2 DIA

The Diagonal (DIA) [27] matrix storage format is specially optimized for sparse matrices composed of non-zero elements along diagonals. In this format, the diagonals are laid out as columns in a dense matrix structure ($data$), starting with the farthest sub-diagonal and ending with the largest super-diagonal. An additional vector ($diag$) is kept which maintains the offset of the diagonal represented by column i in $data$ from the central diagonal. Since the structure of the matrix is known, this storage format does not require column offset or row pointer vectors like CSR, leading to a lower storage overhead.

Consider the sparse diagonal matrix in Figure 5(a) and its DIA representation in (c). The x elements in the $data$ matrix represent unused elements that are needed to pad the diagonals into full columns. Unlike CSR, the DIA matrix layout is more efficient for diagonal matrices and allows contiguous memory access when reading matrix elements along diagonals.

2.3.3 ELLPACK

The ELLPACK [15] matrix format is another space-efficient technique for storing sparse matrices. If K is the largest number of non-zero elements per row of an $N \times M$ matrix, the ELLPACK format stores the matrix as an $N \times K$ dense matrix ($data$), along with a column index matrix (col) that stores the column index of each element. For rows that contain less than K non-zero elements, the matrices $data$ and col are padded with unused elements.

Consider the example in Figure 5. The ELLPACK representation of the sparse matrix is shown in (d).

2.3.4 Comparisons

The CSR, DIA, and ELLPACK sparse matrix layouts are all commonly used to efficiently store sparse matrices. For the sparse matrix in Figure 5, the equivalent matrix in the other matrix layouts is shown. In addition to the difference in storage size, these three layouts offer various trade-offs. The CSR representation can efficiently represent any sparse matrix structure, but requires indirection through the col and row vectors for each matrix access. In contrast, the DIA representation is more efficient and does not require indirection through an addition array for all accesses, but is only efficient for sparse matrices with non-zeros along diagonals. Finally, the ELLPACK representation offers an efficient storage format if the maximum number of non-zero elements in all rows is significantly less than the number of columns in the sparse matrix.

3. COLUMN DIAGONAL STORAGE (CDS)

In this work, we introduce a new sparse matrix storage format that takes into account the block-diagonal structure found in structured grid computations involving problems with degrees of freedom greater than one. Consider the block-diagonal sparse matrix in Figure 6. Colors are used to differentiate between the block diagonals, and empty spaces represent zero values. This matrix contains block diagonals that represent 2×2 dense matrices. There are three diagonals of dense blocks in the sparse matrix, but if we consider the matrix as a general diagonal-sparse matrix then we have nine diagonals that contain non-zero elements.

Therefore, the standard DIA storage format will require us to represent nine diagonals, including the zero values that occur within "sparse" diagonals. Figure 6 shows the DIA representation for this sparse matrix. Clearly, there is a large fraction of wasted space used to store zero and unused elements. When performing sparse matrix-vector multiplication, there will also be a significant amount of wasted work multiplying by zero using this representation.

If we instead try to directly modify the DIA layout to consider the $N \times N$ sub-matrices as atomic blocks by linearizing the sub-matrices into columns in the formatted matrix, we can reduce the amount of wasted space in the DIA format for higher degrees of freedom. However, this leads to non-contiguous memory access across cooperating threads in thread blocks. As we will show, our proposed CDS storage format minimizes wasted space as well as maximizing memory coalescing across threads.

3.1 Block-Diagonal Layout

Instead of considering just the diagonals of the sparse matrix which contain non-zero elements, we propose to store the matrix according to its block-diagonal structure. For the sparse matrix in Figure 6, we consider three diagonals composed of 2×2 blocks instead of nine diagonals of scalar entities. We then layout each diagonal by linearizing the columns in each block. We start with the first column of the first block, then the first column of the second block, and so forth for each block. We then proceed to layout the second column of each block. This continues until all columns have been laid out. This procedure is then repeated for each diagonal.

For a block-diagonal sparse matrix with k diagonals and blocks having b columns, this storage format will require $k \cdot b$ columns. If n is the length of the main diagonal of the sparse matrix, then $k \cdot b \cdot n$ is then the storage requirement for our compact representation. Like the DIA storage format, we also maintain a vector containing the diagonal number for each column. However, since each group of b columns correspond to the same diagonal, we only need to store k diagonal indices. Therefore, the total storage requirement is $k \cdot b \cdot n + k$. Figure 6 shows an example of our proposed layout format.

3.2 Overhead and Performance

Compared to the DIA storage format, we incur less storage overhead for block-diagonal sparse matrices. If the block size is 1×1, then our format reduces to the DIA format. However, as we increase the block size we see that the amount of wasted space decreases compared to the DIA format. The benefits of this are two-fold. First, we require less total memory to store the sparse matrix. This is particularly important on accelerator devices with limited memory. Modern GPU accelerators have up to 3GB of off-chip storage, while high-performance CPUs typically have 32-64GB. Second, eliminating the zero values from the representation allows us to also eliminate the useless computation that results from using the zero values. The contribution of these elements to the entire matrix-vector multiplication will be zero, so we do not need to compute the result.

3.3 CUDA Implementation

For evaluation of our proposed sparse matrix layout, we implemented a sparse matrix-vector kernel in CUDA [25] using our storage format. The kernel implements sparse matrix-vector multiplication for any number of degrees of freedom. For efficiency, we store the vector $\vec{x}$ as a texture and access it through texture fetches. This allows us to make use of the texture cache available on the GPU.

We linearize the columns of each diagonal to ensure we have contiguous access along the output vector $\vec{b}$. We schedule one thread for each element of $\vec{b}$, and neighboring threads will access columns of the sparse matrix. Therefore, the column-major ordering ensures memory coalescing. As a result of this, the access to $\vec{x}$ is not contiguous for neighboring threads. However, most accesses to $\vec{x}$ will be to the same element for neighboring threads so broadcast reads and caches are effective. In particular, storing $\vec{x}$ as a texture and making use of the texture cache leads to increased performance.

4. EVALUATION

To evaluate the performance of our proposed sparse matrix storage format, we implemented sparse matrix-vector multiplication kernels for our sparse matrix storage format using CUDA 4.0 [25] and evaluated performance on various data sets for different nVidia GPU architectures, including GTX 280, Quadro Plex S2200 S4[1], Tesla C2050, and GTX 580. The characteristics of these GPUs are described in Table 1.

For comparison, we use the CSR-based structured grid solver found in the PETsc [2] library. Additionally, we used DIA-based and ELLPACK- based sparse matrix-vector kernels from Cusp [6] to compare performance with other storage formats. We compare the performance of our storage format by looking at device floating-point throughput for different problem sizes and degrees of freedom.

4.1 DoF Scaling

As our proposed sparse matrix storage format is made to efficiently handle matrices that arise from structured grid problems with high degrees of freedom, we evaluate the performance of the kernels as we vary the degrees of freedom in the problem. We selected four representative problem

[1]The Quadro Plex S2200 S4 unit has four GPU chips exposed as four separate CUDA devices. For this paper, we only consider one GPU.

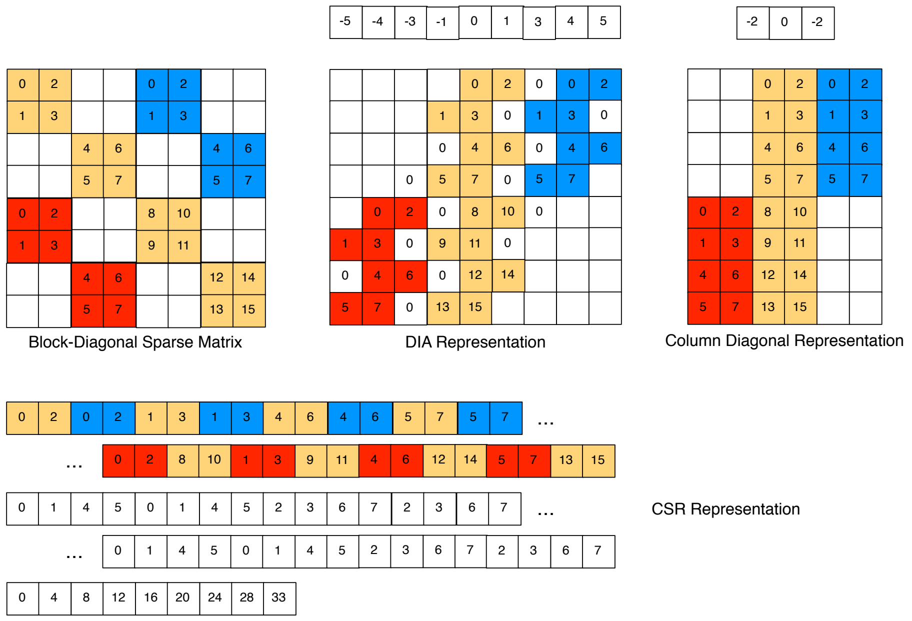

Figure 6: Layout of a sparse matrix for CSR, DIA, and our proposed Column Diagonal formats. The sparse matrix is block-diagonal with a block size of 2×2.

sizes (two 2-D and two 3-D), and measured the achieved GFLOP/s for our kernels, the CSR-based kernel from PETsc, and the DIA- and ELLPACK-based kernels from Cusp. Figure 7 shows the results of this experiment. For the shown problem size N, the sparse matrix is of size $N \times N$, so a problem size of 64^3 involves a $64^3 \times 64^3$ sparse matrix. We present results using both single- and double-precision floating-point values.

From these results, we see that our kernel equals the performance of the DIA- based kernel for one degree of freedom, across all GPUs and all problem sizes. However, as we increase the degrees of freedom in the structured grid problem, the performance of the DIA-based kernel deteriorates, while the performance of the kernel using our proposed storage format improves. We also generally perform better than the ELLPACK-based implementation, due to better use of the texture cache when accessing the vector $\vec{x}$. Both our format and the Cusp implementation of ELLPACK make use of the texture cache, but we achieve a higher cache hit ratio. Especially on the newer Fermi-based GTX 580 and Tesla C2050 cards, we significantly out-perform all other storage formats that we tested.

We see that the GTX 280 generally performs better than the Quadro Plex S2200 for these kernels. Even though both are based on the same architecture, the GTX 280 has higher off-chip bandwidth because the Quadro Plex S2200 incurs overhead from using ECC memory. Since no reuse is possible in the sparse matrix and each element is only used in a single multiply-add pair of instructions, such kernels are memory bandwidth bound. Clearly, the GTX 580 performs the best across the board due to its higher off-chip bandwidth and floating-point throughput. The Tesla C2050 results are similar to the GTX 580 results, but lower off-chip memory bandwidth leads to lower performance.

	GTX 280	**Quadro S2200 S4**	**GTX 580**	**Tesla C2050**
# SMs	240	240 (x4)	512	448
Memory	1 GB	4 GB	1.5 GB	3 GB
Compute	1.3	1.3	2.1	2.0
Peak B/W	141.7 GB/sec	102 GB/sec	192 GB/sec	144 GB/sec

Table 1: GPUs used for testing sparse matrix-vector implementations

Across all problems, the CSR representation performs the worst of all four. The CSR representation of sparse matrices is the most general of the four storage formats we tested, but the matrices from structured grids that we are considering give us a lot of opportunity for layout-specific optimizations. The DIA representation is a good choice for structured grid matrices, and we can see that the performance is significantly improved over the CSR representation. However, in the majority of cases, our proposed column-diagonal storage format leads to better performance than the DIA representation.

The benefit of the new proposed storage format is particularly significant at higher degrees of freedom. For matrices that result from structured grid computations with high degrees of freedom, the DIA storage format has to explicitly store many zero elements in the diagonals. Our storage format, in contrast, does not suffer from this and results in fewer computations that do not contribute to the final result. Since our storage format does not lose performance with only one degree of freedom, it can be used for all problems instead of forcing the user or library to choose a storage format based on the problem.

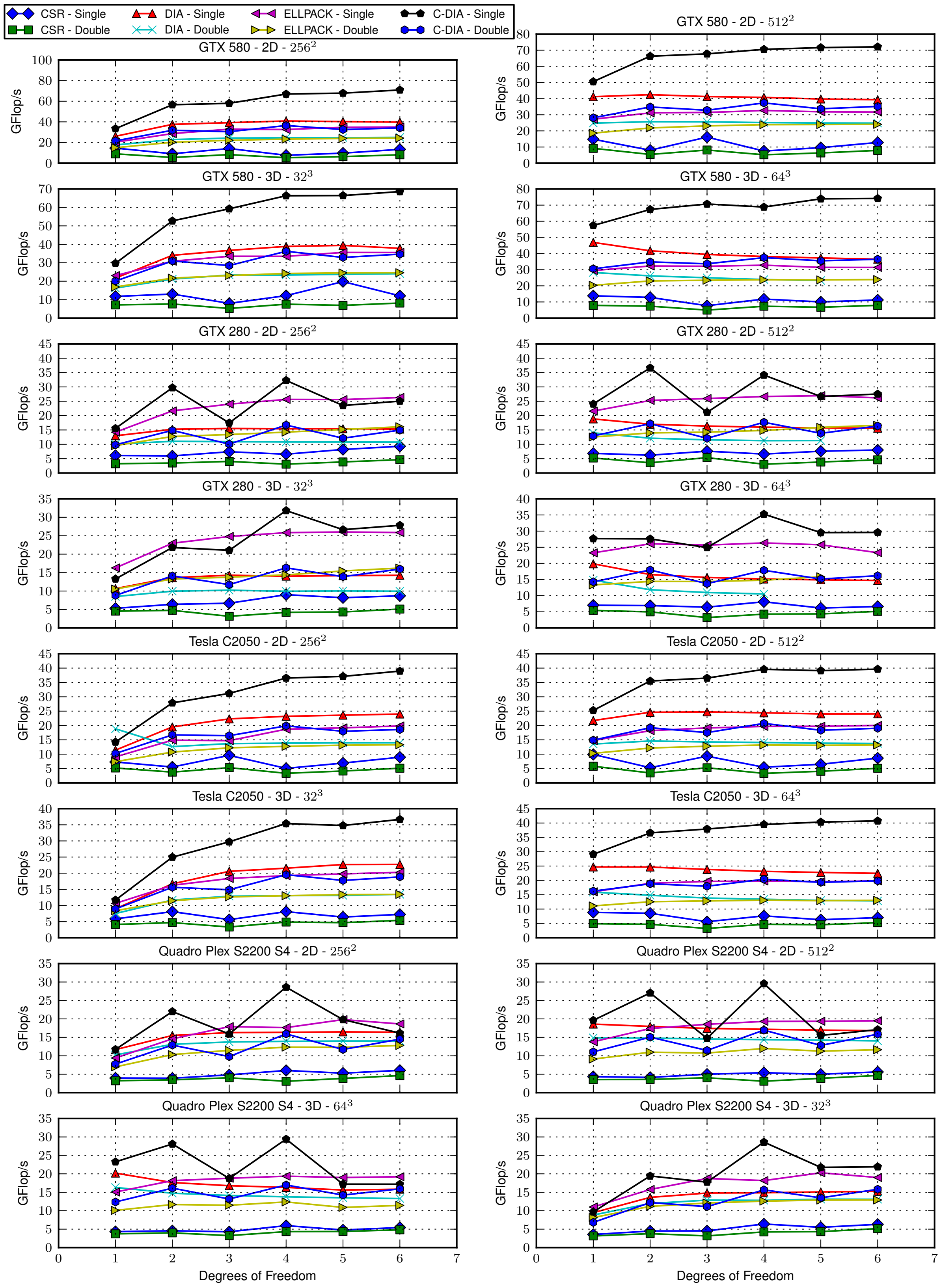

Figure 7: Performance scaling across degrees of freedom

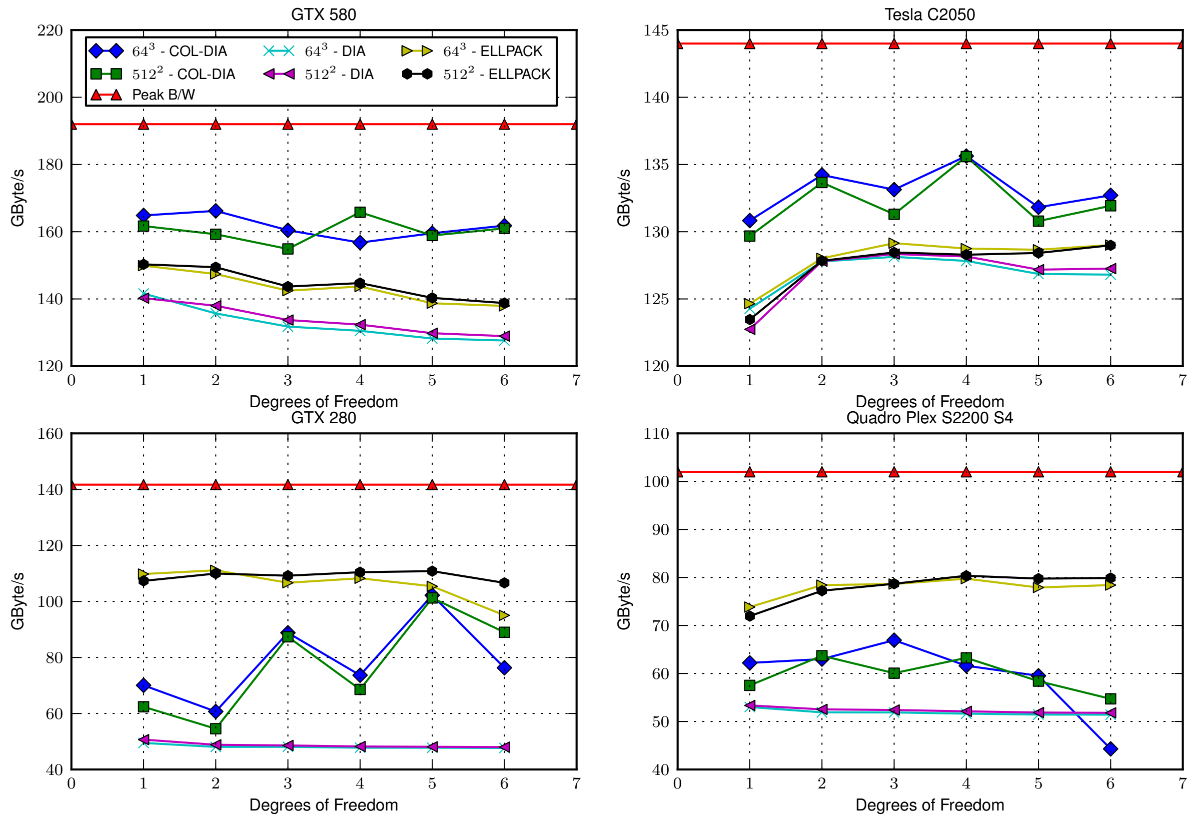

Figure 8: Bandwidth utilization for our tested kernels.

4.2 Bandwidth Utilization

GPU devices offer high off-chip memory bandwidth, but maximizing the achievable bandwidth is often a difficult optimization problem. Sparse matrix-vector multiplication kernels in particular are sensitive to memory bandwidth since there is very little reuse across the data arrays. In general, there will only be a single multiply-add pair for each matrix element. Therefore, it is very important to optimize the kernel to maximize the off-chip bandwidth while reading matrix elements.

In Figure 8, we show the off-chip bandwidth achieved for our kernels. For each GPU, we show the off-chip bandwidth achieved for our column-diagonal kernel on two problem sizes, as well as for the Cusp diagonal kernel. On the same graph, we show the peak bandwidth available on each device. From these graphs, we see that our implementation of the column-diagonal kernel is able to achieve higher off-chip bandwidth than the Cusp diagonal kernel in most cases.

An interesting trend in these results is the achieved bandwidth as the degrees of freedom are varied. On the GTX 280, the bandwidth is clearly higher for odd degrees of freedom. However, on the GTX 580, the achieved bandwidth does not vary much. An increase in the achieved bandwidth, however, is not necessarily an indicator of increased performance. As shown in Figure 7, the GTX 280 achieves high performance on even degrees of freedom. This leads us to the conclusion that the problems with an odd degree of freedom are pulling more data across the memory bus, but this extra data is actually not used since there are unused threads. On the GTX 580, the effect is not as pronounced since the per-SM L1 cache is able to service these extraneous requests.

5. RELATED WORK

Sparse matrix-vector multiplication is a core computation in many numeric applications, and there is a large body of related work on the subject. In recent years, many researchers have looked at sparse matrix-vector multiplication on off-chip accelerators such as GPUs and other parallel architectures.

Azevedo et al. [9] proposed a technique for vectorizing sparse matrix-vector multiplication on vector architectures such as the Cray X-1 using a variation of the CSR storage format.

Some of the first work on sparse matrix-vector multiplication on GPU architectures was by Bolz et al. [7]. They implemented conjugate gradient and multigrid solvers on a GPU by using the graphics pipeline. Kruger et al. [19] explored the use of the graphics pipeline to implement sparse matrix-vector multiplication and grid- based solvers. Geveler et al. [13, 12] presented a framework for solving multi-grid problems on GPUs by decomposing the solver into a sequence of sparse matrix-vector multiplications.

Baskaran et al. [3] performed optimizations on sparse matrix-vector multiplication kernels using the CSR matrix format. They performed device-specific optimizations for both the nVidia 8800 GTX and GTX 280 GPUs. They considered general, unstructured matrices. On the GTX 280, they achieved up to 14.02 GFLOP/s on the *protein* benchmark. Garland [11] presented an approach to perform sparse matrix-vector multiplication using data parallel primitives such as *map*, *scan*, and *reduce*.

Prior work has also focused on the effects of sparse matrix storage formats. Vázquez et al. [29] proposed a new storage format, ELLPACK-R, based on the ELLPACK [15] storage format. This new format reduces the computation and data access compared to the ELLPACK format by using an additional row vector to store the number of non-zeroes per row. They also proposed a new format called the Padded Jagged Diagonals Storage (PJDS) format based on the ELLPACK-R and Jagged Diagonals Storage (JDS) format. Their PJDS implementation achieved up to 27.6 GFLOP/s on an nVidia Tesla C2070 in single-precision mode. PJDS is less efficient than our proposed format for structured grid computations since it has twice the number of global memory accesses due to the column index storage. Monakov et al. [21] proposed a storage format based on ELLPACK, called sliced ELLPACK that performs ELLPACK layout on adjacent rows of the sparse matrix. They achieve up to 21.48 GFLOP/s on an nVidia GTX 280 GPU in single-precision. Bell et al. [4, 5] examined a range of matrix layout formats for sparse matrices using CUDA on the nVidia GTX 280 GPU. They found that the DIA [28] and ELLPACK formats achieve the best performance on structured matrices that result from stencil computations. Our experimental results demonstrate that the CDS format proposed here achieves higher performance for such matrices than the DIA or ELLPACK formats.

The Blocked CSR matrix format was introduced by Im et al. [16, 17]. This format is a variant of CSR that allows for register blocking on CPU architectures. This format performs better than the CSR format but because of the parallel reduction step, it performs worse than the CDS format. Montagne [23] and Ekambaram [10] proposed the Transpose Jagged Diagonal Storage (TJDS) format, which builds upon the Jagged Diagonal Storage (JDS) format by collapsing the non- zero elements along columns instead of rows. The resulting storage requirement is $O(N_{nz} + N_{tjd})$ instead of $O(N_{nz} + N + N_{jd})$ for JDS, where N_{nz} is the number of non-zeros in the sparse matrix, and N_{tjd} and N_{jd} are the number of transpose jagged diagonals and jagged diagonals, respectively.

Research has also been conducted on auto-tuning sparse matrix-vector implementations for GPUs. Choi et al. [8] proposed an auto- tuning framework for sparse matrix-vector kernels using blocked CSR and blocked ELLPACK matrix storage formats. The Blocked ELLPACK format rearranges the rows of the sparse matrix in decreasing order of the number of non-zero elements. The rows are then separated into blocks, where each block is stored in the ELLPACK format. This formats achieves a significant increase in performance compared to ELLPACK in unstructured grid problems where the number of non-zeroes per row varies greatly. However, it does not provide much performance improvement over ELLPACK in structured grid applications , where the number of non- zeroes per row is nearly constant. Choi et al. [8] achieved up to 29.0 GFLOP/s in single-precision on an nVidia Tesla C1060 GPU.

6. CONCLUSION

In this paper, we have introduced a new sparse matrix storage format that is specially optimized for block-diagonal sparse matrices that result from structured grid computations with more than one degree of freedom. We have shown that the performance of CUDA kernels using this format exceeds that of previously proposed storage formats for more than one degree of freedom, including the often used DIA and CSR storage formats.

Acknowledments

We thank the reviewers for their feedback and suggestions to improve the paper. This work was supported in part by the National Science Foundation through award 0926688 and by the Department of Energy (subcontract to The Ohio State University from RNET Technologies; DOE award DE-SC0002434)

7. REFERENCES

[1] Badcock, K., Richards, B., and Woodgate, M. Elements of computational fluid dynamics on block structured grids using implicit solvers. *Progress in Aerospace Sciences 36*, 5-6 (2000), 351 – 392.

[2] Balay, S., Brown, J., , Buschelman, K., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H. PETSc users manual. Tech. Rep. ANL-95/11 - Revision 3.2, Argonne National Laboratory, 2011.

[3] Baskaran, M. M., and Bordawekar, R. Optimizing sparse matrix-vector multiplication on GPUs. Tech. Rep. RC24704 (W0812-047), IBM T. J. Watson Research Center, April 2009.

[4] Bell, N., and Garland, M. Efficient sparse matrix-vector multiplication on CUDA. Tech. Rep. NVR-2008-004, NVIDIA Corporation, December 2008.

[5] Bell, N., and Garland, M. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009), SC '09, ACM, pp. 18:1–18:11.

[6] Bell, N., and Garland, M. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2010. Version 0.1.0.

[7] Bolz, J., Farmer, I., Grinspun, E., and Schröoder, P. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. In *ACM SIGGRAPH 2003 Papers* (New York, NY, USA, 2003), SIGGRAPH '03, ACM, pp. 917–924.

[8] Choi, J. W., Singh, A., and Vuduc, R. W. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2010), PPoPP '10, ACM, pp. 115–126.

[9] D'Azevedo, E., Fahey, M., and Mills, R. Vectorized sparse matrix multiply for compressed row storage format. In *Computational Science - ICCS 2005*, V. Sunderam, G. van Albada, P. Sloot, and J. Dongarra, Eds., vol. 3514 of *Lecture Notes in*

Computer Science. Springer Berlin / Heidelberg, 2005, pp. 785–789. 10.1007/11428831_13.

[10] Ekambaram, A., and Montagne, E. An alternative compressed storage format for sparse matrices. In *Computer and Information Sciences - ISCIS 2003*, A. Yazici and C. Sener, Eds., vol. 2869 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2003, pp. 196–203. 10.1007/978-3-540-39737-3_25.

[11] Garland, M. Sparse matrix computations on manycore gpu's. In *Proceedings of the 45th annual Design Automation Conference* (New York, NY, USA, 2008), DAC '08, ACM, pp. 2–6.

[12] Geveler, M., Ribbrock, D., Göddeke, D., Peter, Z., and Stefan, T. Efficient finite element geometric multigrid solvers for unstructured grids on GPUs. *PARENG* (April 2011).

[13] Geveler, M., Ribbrock, D., Göddeke, D., Peter, Z., and Stefan, T. Towards a complete FEM-based simulation toolkit on GPUs: Geometric multigrid solvers. *ParCFD* (May 2011).

[14] Goodale, T., Allen, G., Lanfermann, G., Massó, J., Radke, T., Seidel, E., and Shalf, J. The Cactus framework and toolkit: Design and applications. In *Vector and Parallel Processing – VECPAR'2002, 5th International Conference, Lecture Notes in Computer Science* (Berlin, 2003), Springer.

[15] Grimes, R., Kincaid, D., and Young, D. ITPACK 2.0 user's guide, August 1979.

[16] Im, E.-J., and Yelick, K. Optimizing sparse matrix computations for register reuse in sparsity. In *Computational Science - ICCS 2001*, V. Alexandrov, J. Dongarra, B. Juliano, R. Renner, and C. Tan, Eds., vol. 2073 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2001, pp. 127–136. 10.1007/3-540-45545-0_22.

[17] Im, E.-J., Yelick, K., and Vuduc, R. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications 18*, 1 (2004), 135–158.

[18] Khronos OpenCL Working Group. The OpenCL specification - version 1.2.

[19] Krüger, J., and Westermann, R. Linear algebra operators for gpu implementation of numerical algorithms. In *ACM SIGGRAPH 2005 Courses* (New York, NY, USA, 2005), SIGGRAPH '05, ACM.

[20] Los Alamos National Laboratory. PFLOTRAN. http://ees.lanl.gov/source/orgs/ees/pflotran/index.shtml.

[21] Monakov, A., Lokhmotov, A., and Avetisyan, A. Automatically tuning sparse matrix-vector multiplication for gpu architectures. In *High Performance Embedded Architectures and Compilers*, Y. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, Eds., vol. 5952 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2010, pp. 111–125. 10.1007/978-3-642-11515-8_10.

[22] Monorchio, A., and Mittra, R. Time-domain (fe/fdtd) technique for solving complex electromagnetic problems. *Microwave and Guided Wave Letters, IEEE 8*, 2 (feb 1998), 93 –95.

[23] Montagne, E., and Ekambaram, A. An optimal storage format for sparse matrices. *Information Processing Letters 90*, 2 (2004), 87 – 92.

[24] Nikolova, N., Tam, H., and Bakr, M. Sensitivity analysis with the fdtd method on structured grids. *Microwave Theory and Techniques, IEEE Transactions on 52*, 4 (april 2004), 1207 – 1216.

[25] NVIDIA Corporation. CUDA C programming guide - version 4.0.

[26] NVIDIA Corporation. OpenCL programming guide for the CUDA architecture.

[27] Saad, Y. SPARSKIT: A basic toolkit for sparse matrix computations - version 2.

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

[29] Vázquez, F., Fernández, J. J., and Garzón, E. M. A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurrency and Computation: Practice and Experience 23*, 8 (2011), 815–826.