Optimizing Sparse Matrix-Vector Multiplication on GPUs

Muthu Manikandan Baskaran
Dept. of Computer Science and Engineering
The Ohio State University, Columbus, OH, USA
baskaran@cse.ohio-state.edu

Rajesh Bordawekar
IBM TJ Watson Research Center
Hawthorne, NY, USA
bordaw@us.ibm.com

Abstract

We are witnessing the emergence of Graphics Processor units (GPUs) as powerful massively parallel systems. Furthermore, the introduction of new APIs for general-purpose computations on GPUs, namely CUDA from NVIDIA, Stream SDK from AMD, and OpenCL, makes GPUs an attractive choice for high-performance numerical and scientific computing. Sparse Matrix-Vector multiplication (SpMV) is one of the most important and heavily used kernels in scientific computing. However with indirect and irregular memory accesses resulting in more memory accesses per floating point operation, optimization of SpMV kernel is a significant challenge in any architecture.

In this paper, we evaluate the various challenges in developing a high-performance SpMV kernel on NVIDIA GPUs using the CUDA programming model and propose optimizations to effectively address them. The optimizations include: (1) exploiting synchronization-free parallelism, (2) optimized thread mapping based on the affinity towards optimal memory access pattern, (3) optimized off-chip memory access to tolerate the high access latency, and (4) exploiting data locality and reuse. We evaluate our optimizations over two classes of NVIDIA GPU chips, namely, GeForce 8800 GTX and GeForce GTX 280, and we compare the performance of our approach with that of existing parallel SpMV implementations such as (1) the one from NVIDIA's SpMV library, (2) the one from NVIDIA's CUDPP library, and (3) the one implemented using optimal segmented scan primitive. Our approach outperforms the CUDPP and segmented scan implementations by a factor of 2 to 8. Our approach is either in par with NVIDIA's SpMV library in performance or achieves up to 35% improvement over NVIDIA's SpMV library.

1. Introduction

Modern computer architecture has shifted towards designs that employ multiple processor cores on a chip, so called *multicore* processors. However, the current multicore systems are so architecturally diverse that to fully exploit the potential of multiple processors, the applications have to be specialized for the underlying system using architecture-specific optimization strategies.

One of the key reasons for the architectural diversity is the need to balance memory and processor capabilities. Memory bandwidth has always been a performance bottleneck in traditional computer architectures, and it is even more pronounced in multicore systems. The trend in computer architecture shows that increasing processor cores on a chip is more cost effective than increasing memory bandwidth. Hence, memory bottleneck is going to remain as the key performance bottleneck in future multicore architectures. Traditionally, a multi-level cache hierarchy is used to alleviate the memory bottleneck. Due to various reasons concerning power efficiency and performance, many modern multicore processors, instead of caches, support fast explicitly managed on-chip memories, often referred to as *scratchpad memories* or *local stores*. The scratchpad memories are software-managed, unlike caches that are hardware-controlled, and hence the execution times of programs using scratchpad memories can be more accurately predicted and controlled.

Thus, many of the architecture-specific optimization strategies involve specific optimizations targeted towards improving memory throughput of an application. These optimizations enable parallel applications to yield higher performance by tolerating the underlying memory bottleneck while utilizing the computational power of the multi-core system. Such memory optimizations are better appreciated in applications that are inherently memory-bound. One such memory-bound application kernel that is heavily used in many scientific and engineering applications is the *Sparse Matrix-Vector Multiplication* (SpMV) kernel. The SpMV kernel computes a vector x as a result of multiplying a sparse matrix A by a vector y ($x = Ay$).

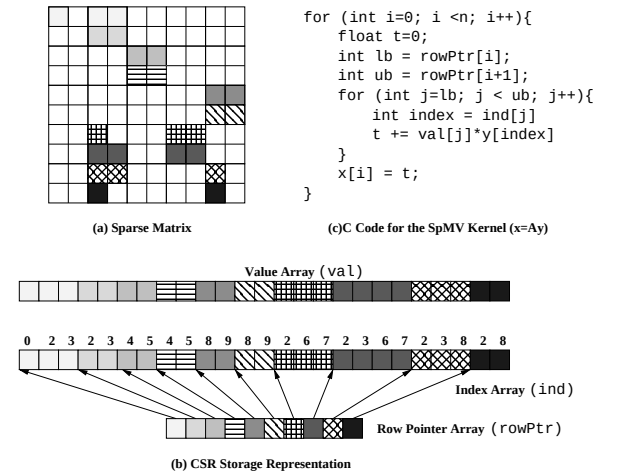

Figure 1. Sparse Matrix-Vector Multiplication and CSR Sparse Matrix Storage Format

Although SpMV is a prominent kernel used in many engineering and scientific applications, it is well known that SpMV yields only a small fraction of machine peak performance [21]. Sparse matrix computations involve far more memory accesses per floating point operation, due to indirect and irregular memory accesses. Higher performance for SpMV computation requires optimizations that best utilize the properties of the sparse matrix and also the underlying system architecture. The storage format of sparse matrix is also very important in determining the performance. The most common sparse matrix storage format is the Compressed Sparse Row (CSR) format (Figure 1). The non-zero elements of each row in the sparse matrix are stored contiguously in a dense array, `val`. A dense integer array, `ind`, stores the column index of each non-zero element. Another dense integer array, `rowPtr`, stores the starting position of each row of the sparse matrix in `val` (and `ind`). Figure 1(c) presents the SpMV kernel code in C. Some basic characteristics of the SpMV computation can be inferred from the kernel presented in Figure 1(c). They include: (1) existence of synchronization-free parallelism across the rows, (2) existence of reuse of input and output vector elements, (3) non-existence of data reuse of matrix elements, and (4) more memory accesses per floating operation involving a non-zero element.

Graphics Processing Units (GPUs) are one of the most powerful multi-core systems currently in use. For example, the NVIDIA GeForce 8800 GTX GPU chip has a single-precision peak performance of over 350 GFLOPS and the NVIDIA GeForce GTX 280 chip has a single-precision peak performance of over 900 GFLOPS. In addition to the primary use of GPUs in accelerating graphics rendering operations, there has been considerable interest in exploiting GPUs for General Purpose computation (GPGPU) [9]. Until very recently, GPGPU computations were performed by transforming matrix operations into specialized graphics processing such as texture operations. The introduction of new parallel programming interfaces for general purpose computations, such as Compute Unified Device Architecture (CUDA) [16], Stream SDK [1], and OpenCL [17], have made GPUs powerful and attractive choice for developing high-performance numerical and scientific computations. Many modern GPUs exhibit a complex memory organization with multiple low latency on-chip memories in addition to the off-chip DRAM. In addition, they also exhibit a hybrid cache and local-store hierarchy. The access latencies and the optimal access patterns of each of the memories vary significantly, posing a significant challenge to devise techniques that optimally utilize the various memories to tolerate the latency and improve the memory throughput. The memory hierarchy along with the highly parallel execution model make application optimizations difficult. The challenges increase many-fold when the application to be optimized is a memory-intensive kernel like SpMV.

In this work, we investigate the problem of optimizing SpMV kernels on a modern GPU, specifically, on the NVIDIA GTX series using the CUDA parallel programming model. First, we evaluate the NVIDIA GPU architecture and the CUDA execution model using a naive non-optimized implementation of the SpMV kernel. Our experiments reveal two key inter-related obstacles in improving the SpMV performance on the NVIDIA GPUs: thread mapping and data access strategies. We address these concerns by proposing various optimizations that take into account both the application and the architectural characteristics. The optimizations include: (1) exploiting synchronization-free parallelism, (2) optimized thread mapping based on the *affinity towards optimal memory access pattern*, (3) optimized off-chip memory access to tolerate the high access latency, and (4) exploiting data locality and reuse. We evaluate our optimizations using two different NVIDIA GPUs, namely, GeForce 8800 GTX, and GeForce GTX 280, using a large set of sparse matrices derived from real applications. We compare our approach against three existing SpMV CUDA implementations, namely, NVIDIA's SpMV library [2], NVIDIA's CUDPP [6] library and the one implemented using optimal segmented scan primitives from Dotsenko et al. [8].

Our work makes the following key contributions:

- We provide solutions to address two key inter-related concerns in improving the performance of memory-bound applications like SpMV on NVIDIA GPUs, namely, thread mapping and data access strategies.
- We implement an effective and optimized SpMV kernel on NVIDIA GPUs considering the architectural characteristics. Our implementation (which optimizes over the CSR storage format) achieves a comparable or better performance than the implementation using the new sparse matrix storage format ("Hybrid" format) proposed in NVIDIA's SpMV library. Our implementation outper-

forms the NVIDIA CUDPP library and the implementation using optimal segmented scan by a factor of 2 to 8.

- We provide an effective solution that does not change the storage format of sparse matrices and retains the more general CSR format. Furthermore, unlike NVIDIA's SpMV library there is no preprocessing performed to change the data layout of matrix elements or input/output vector elements.

The rest of the paper is organized as follows: Section 2 presents an overview of the NVIDIA GPU architecture and the CUDA programming model. The problem statement is presented in Section 3. Section 4 describes the proposed SpMV optimizations in detail. Experimental results are presented in Section 5. Section 6 discusses related work. Finally, we conclude in Section 7.

2. GPU Architecture and the CUDA Programming Model

In this Section, we discuss about the GPU parallel computing architecture and the CUDA programming interface.

GPU Computing Architecture: The GPU parallel computing architecture consists of a set of multiprocessor units called the *streaming multiprocessors (SMs)*, each one containing a set of processor cores (called the *streaming processors* (SPs)). There are various memories available in GPUs for a programmer. The memories are organized in a hybrid cache and local-store hierarchy. The memories are as follows: (1) off-chip global memory, (2) off-chip local memory, (3) on-chip shared memory, (4) off-chip constant memory with on-chip cache, (5) off-chip texture memory with on-chip cache, and (6) on-chip registers.

The global memory is a large memory and has a very high latency. The shared memory is present in each SM and is organized into banks. When multiple addresses belonging to the same bank are accessed at the same time, it results in bank conflict. Each SM has a set of registers. The constant and texture memories are read-only regions in the global memory space and they have on-chip read-only caches. Accessing constant cache is faster, but it has only a single port and hence it is beneficial when multiple processor cores load the same value from the cache. Texture cache has higher latency than constant cache, but it does not suffer greatly when memory read accesses are irregular and it is also beneficial for accessing data with 2D spatial locality.

CUDA Programming Model: Programming GPUs for general-purpose applications is enabled through an easy-to-use C/C++ language interface exposed by the NVIDIA Compute Unified Device Architecture (CUDA) technology [16]. The CUDA programming model provides an abstraction of the GPU parallel architecture using a minimal set of programming constructs such as hierarchy of threads, hierarchy of memories, and synchronization primitives. A CUDA program comprises of a host program which is run on the CPU or host and a set of CUDA kernels that are launched from the host program on the GPU device. The CUDA kernel is a parallel kernel that is executed on a set of threads. The threads are organized into groups called *thread blocks*. The threads within a thread block synchronize among themselves through barrier synchronization primitives in CUDA and they communicate through a shared memory space that is available to the thread block. A kernel comprises of a *grid* of one or more thread blocks. Each thread in a thread block is uniquely identified by its thread id (threadIdx) within its block and each thread block is uniquely identified by its block id (blockIdx). Each CUDA thread has access to various memories at different levels in the hierarchy. The threads have a private local memory space and register space. The threads in a thread block share a shared memory space. The GPU DRAM is accessible by all threads in a kernel.

The GPU computing architecture employs a Single Instruction Multiple Threads (SIMT) model of execution. The threads in a kernel are executed in groups called *warps*, where a warp is an unit of execution. The scalar SPs within a SM share a single instruction unit and the threads of a warp are executed on the SPs. All the threads of a warp execute the same instruction and each warp has its own Program Counter.

3. Problem Statement

GPUs are massively data-parallel systems with very high parallelism per-chip. A NVIDIA GTX 280 GPU has a theoretical single-precision peak performance of around 933 GFlops and a peak off-chip memory bandwidth of over 141 GBps. However, the off-chip memory latency is as high as 200 clock cycles. To fully exploit the massive computing resources of the GPUs, the off-chip memory latency needs to be efficiently hidden. Thus, optimizations for enhancing the memory performance are critical to GPU systems for utilizing their raw computing power. Furthermore, in future systems, where there will be even more processor cores on chip, memory bottleneck will increasingly become a very critical issue. Hence, reducing the memory footprint and tolerating the memory access latency are very important for high performance, especially for memory bound applications.

Matrix vector multiplication is a memory-bound application kernel in which each matrix element that is brought from memory is used *only once* in the computation. Hence, the kernel is characterized by a high memory overhead per floating point operation. When the matrix is sparse, it incurs further complexity in terms of memory overhead because of the indirect and irregular memory accesses. Sparse matrix vector (SpMV) multiplication involves, on an average, more than two memory operations for accessing a single non-zero matrix element and is heavily memory-bound. In addition, the SpMV-specific optimizations depend heavily on the structural properties of the sparse matrix, many of which might be known only at run-time.

As discussed in Section 2, the GPU architecture has multiple low latency memories in addition to the off-chip DRAM, and has a hybrid cache and local-store hierarchy. The characteristics of the various memories available in the GPU are diverse in terms of latency, optimal memory access pattern, and control (either hardware-controlled or software-controlled). This imposes several challenges to effectively reduce memory footprint and hide latency. The optimal access pattern is also dependent on the manner in which threads are mapped for computation and also on the number of threads involved in global memory access as involving more threads would assist in hiding the global memory access latency. Hence, there has to be an optimal thread mapping to ensure optimized memory access.

In summary, enhancing memory performance is key for utilizing the high computation power of GPU systems, especially for memory-bound applications such as the SpMV kernel. However, there are significant challenges to be addressed, both in the context of the underlying architecture and the application. In this work, we develop techniques for optimizing SpMV computations on GPUs that match application requirements against the architectural constraints.

4. Implementation of the Optimizations

In this Section, we discuss the implementation of our techniques for optimizing SpMV computations on GPUs. There are various storage formats (as explained in [2]) for sparse matrices, some of which may be well suited for specific patterns of sparse matrices. However, we base our optimizations on the more general CSR format and discuss ways to adapt CSR storage format to suit the GPU architecture. We first explain how we devise the architecture-specific optimizations for SpMV kernel and also then validate the applicability of these optimizations in attaining good performance by illustrating with a few performance results.

Exploiting Synchronization-free Parallelism: The CUDA programming model provides an API to synchronize across all threads belonging to a thread block. However, there is no API in CUDA to synchronize between thread blocks. To synchronize between thread blocks, the CUDA programmer has to explicitly implement synchronization primitives using atomic reads/writes in the global memory space, which incurs a high overhead. Hence, it is critical to utilize synchronization-free parallelism across thread blocks. In SpMV computation, the parallelism available across rows makes it a natural choice to distribute computations corresponding to a row or a set of rows to a thread block. The naive way of parallelizing SpMV (in CSR format) is to allocate one thread to perform the computation corresponding to one row and a thread block to handle a set of rows.

Optimized Thread Mapping: In GPUs, thread mapping for computation should ensure that sufficient threads are involved to hide global memory access latency and also ensure that the global memory access is optimized, as it is very critical for performance. The most optimal pattern of access for global memory is the hardware optimized *coalesced* access pattern that would be enabled when consecutive threads of a half-warp (i.e. group of 16 threads) access consecutive elements. It is, therefore, necessary to involve multiple threads for the computation corresponding to each row, and also arrive at a thread mapping based on the *affinity towards optimal memory access pattern*. Our thread mapping strategy maps multiple threads (16 threads) per row such that consecutive threads access consecutive non-zero elements of the row in a cyclic fashion to compute partial products corresponding to the non-zero elements. The threads mapped to a row then compute the output vector element corresponding to the row from the partial products through parallel sum reduction. The partial products are stored in shared memory as they are accessed only by threads within a thread block.

Optimized (Aligned) Global Memory Access: Before we proceed to explain our optimization to enable hardware optimized global memory coalesced accesses, we discuss about global memory access coalescing in NVIDIA GPUs. Global memory access coalescing is applicable to memory requests issued by threads belonging to the same half-warp. The constraints for global memory accesses of a half-warp to get coalesced are slightly different for NVIDIA GeForce 8800 GTX and NVIDIA GeForce GTX 280. The global memory can be assumed to be consisting of aligned memory segments. We further base our discussion to memory requests for 32-bit words. In 8800 GTX device, when all 16 words requested by the threads of a half-warp lie within the same 64 byte memory segment and if consecutive threads access consecutive words, then all the memory requests of the half-warp are coalesced into one memory transaction. But if that access pattern is not followed among the threads of a half-warp, then it results in 16 separate memory requests. However, in GTX 280 device, the access pattern need not be so strict for coalescing to happen. In GTX 280, the hardware detects the number of 128 byte memory segments that hold the 16 words requested by the threads of a half-warp and issues as many memory transactions. There is no restriction on the sequence of access within the threads of a half-warp.

In both GPU devices, when the base address of global memory access requests issued by the threads of a half-warp is aligned to memory segment boundary and the threads access words in sequence, it results in fewer memory transactions. It is a strict requirement for coalescing in GeForce 8800 GTX, however it is beneficial even in GeForce GTX 280. Hence we need to adjust the computation to force the access pattern to be aligned in the above-mentioned manner.

In the SpMV kernel, the number of non-zeros in a row varies across rows, and hence the starting non-zero of a row might be in an non-aligned position in the value array that stores the non-zeros of the sparse matrix. If the computation proceeds without taking care of the alignment issue, all

rows whose starting non-zero is located in an non-aligned position will be entirely accessed in an non-optimal manner and eventually lead to increased memory access cost. We propose two solutions to resolve the issue and achieve optimized aligned accesses. In the first solution, we view a row as having an initial (possible) non-aligned portion and then an aligned portion. The execution proceeds by first computing the partial products for the non-zeros in the non-aligned portion of the row, if it exists, before proceeding to compute the partial products for the aligned portion. In the second solution, zeros are padded to ensure that the number of entries in each row is a multiple of 16. Both the solutions proved to be good experimentally. However, since the second solution requires a change in the storage format because of padding zeros, we stick to the first solution and use that for performance evaluation (that is discussed later in the paper).

Exploiting Data Locality and Reuse: The input and output vectors are the ones that exhibit data reuse in SpMV computation. The reuse of output vector elements is achieved by exploiting synchronization-free parallelism with optimized thread mapping, which ensures that partial contributions to each output vector element are computed only by a certain set of threads and the final value is written only once. The reuse pattern of input vector elements depends on the non-zero access pattern of the sparse matrix. Exploiting data reuse of the input vector elements within a thread or among threads within a thread block can be technically achieved by caching the elements in on-chip memories. The on-chip memory may be (1) texture (hardware) cache, (2) registers or (3) shared memory (software) cache. Utilizing registers or shared memory to cache input vector elements requires the programmer to identify the portions of vector that are reused, which in turn, requires the identification of dense sub-blocks in the sparse matrix. This requires an analysis of the sparse matrix (possibly at runtime). However using the hardware texture cache does not necessarily require analysis of the sparse matrix pattern. Using texture cache, we can reduce global memory traffic, especially reduce non-coalesced accesses, and hence increase global memory bandwidth. We can also exploit 1D spatial locality using texture cache. Hence, we use texture memory to store the input vector and utilize the read-only texture cache to achieve the afore-mentioned performance gains.

We also perform an optional runtime preprocessing of the sparse matrix to identify and extract dense sub-blocks. We implement a block storage format that suits the GPU architecture. The features of our format are: (1) We stick to constant block sizes that enable fine-grained thread-level parallelism, to avoid the memory access penalty in reading block size and block index (which is needed if the block size is allowed to vary), (2) We enforce that starting column of a block should adhere to the alignment constraints of global memory coalescing, and (3) We do not make the entire block dense by filling up zeros, instead, we allow each row in a block to have variable number of entries, and fill up minimal zeros that are just enough to make the number of entries in each row of a block to be a multiple of half warp size.

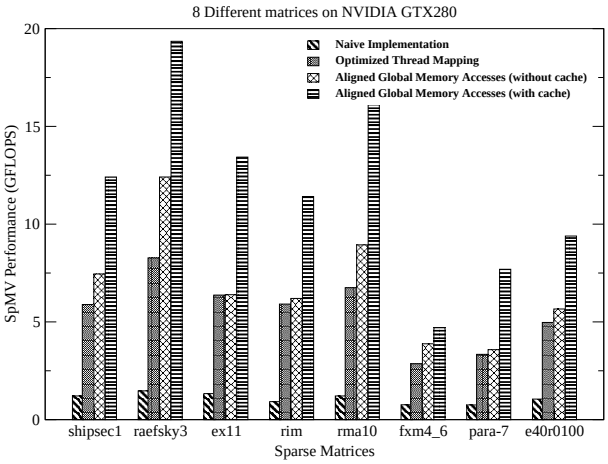

Figure 2. Evaluation of our Optimizations on GeForce GTX 280

For every block, the required input vector elements are loaded from global memory to shared memory, and they are reused across the rows of a block. The number of input vector elements loaded for every block is equal to the block size along column, and since the size is fixed, there is no additional memory access involved to read the block size. By enforcing the constraint that starting column index must be a multiple of half warp size and that number of entries in each row of a block must be a multiple of half warp size, our block storage along with optimized thread mapping ensures that the input vector elements and the sparse matrix elements are accessed in a coalesced manner.

The loads from global memory to shared memory are optimal if they involve 16 coalesced accesses. However in many practical sparse matrices, the dense sub-blocks present are very small. Hence our run-time preprocessing to identify and extract dense sub-blocks, in its current state of implementation, does not yield better performance for most of the matrices. The approach of using texture cache for data reuse outperforms the approach of performing run-time preprocessing and using shared memory cache for data reuse. This is substantiated from the performance numbers in Figure 3.

Matrix	# Coalesced Accesses			# Non-Coalesced Accesses		
	Naive	Thread Mapping	Aligned Access	Naive	Thread Mapping	Aligned Access
raefsky3	172	18898	30457	582784	292676	114466
rma10	414	3693	48289	894942	847676	295968
lp_osa_60	273	413	19181	1085155	189438	134673

Table 1. Profiling Coalesced and Non-coalesced Accesses on 8800 GTX. Number of coalesced accesses increases as the optimizations are applied.

4.1 Performance Evaluation of the Optimizations

We exemplify the effectiveness of the afore-mentioned optimizations through performance measures taken over a set

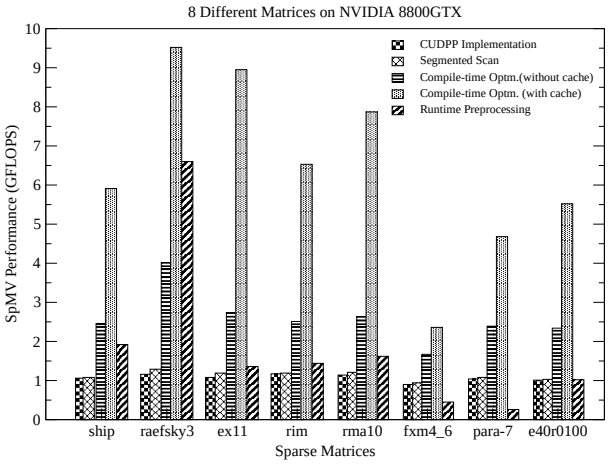

Figure 3. Comparison with Existing Approaches on GeForce 8800 GTX

of sparse matrices. Figure 2 shows the SpMV performance (in GFLOPS) for a set of sample sparse matrices on GTX 280 GPU after applying the optimizations. The performance consistently increases after applying each optimization (in the order mentioned above).

CUDA 2.x supports a profiling infrastructure to instrument architectural metrics such as number of coalesced accesses, number of non-coalesced accesses, number of instructions executed, number of branch instructions executed, etc. We instrumented some of the matrices to check for the number of non-coalesced accesses before and after the application of our optimizations. Table 1 provides the summary of coalesced and non-coalesced accesses for GeForce 8800 GTX. These results clearly indicate substantial improvement in coalesced accesses (and corresponding reduction in non-coalesced accesses) through our optimizations.

5. Experimental Results

We experimentally evaluated our system using two GPU processors - NVIDIA GeForce 8800 GTX and NVIDIA GeForce GTX 280, connected to a host x86/Linux system. The architectural configurations of the two NVIDIA processors are presented in Table 3. The CUDA kernels were compiled using the NVIDIA CUDA Compiler (nvcc) to generate the device code that was then launched from the CPU (host). The GPU device was connected to the CPU through a 16-x PCI Express bus. The host programs were compiled using the gcc compiler at -O3 optimization level. We used CUDA version 2.1 for our experiments. It should be noted that all performance measures we report are for single-precision data.

For our evaluation, we used 19 sparse matrices from the sparse matrix collection described in [7]. The selected sparse matrices represent a wide variety of real applications including finite element method (FEM) based modeling, structural engineering, vibroacoustics, and linear programming. The selected matrices also cover a spectrum of properties with

Feature	8800 GTX	GTX 280
Multiprocessors (SMs)	16	30
Processor cores (SPs)	8	8
Processor Clock	1.35 GHz	1.296 GHz
Off-chip Memory Size	768 MB	1 GB
Off-chip Memory BW	86.4 GBps	141.7 GBps
Peak Performance	388.8 GFLOPS	933.12 GFLOPS

Table 3. Architectural configurations of NVIDIA GeForce 8800 GTX and GeForce GTX 280

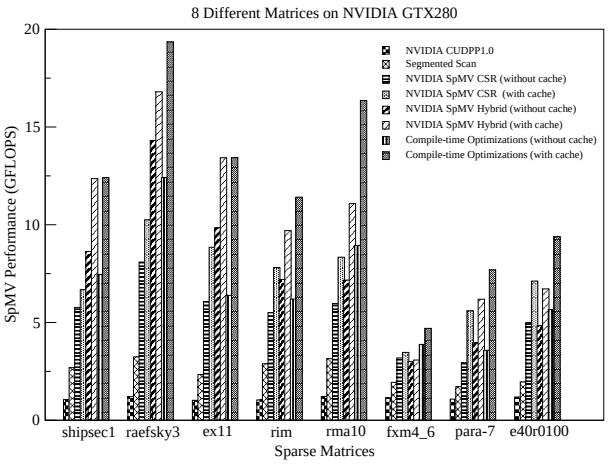

Figure 4. Comparison with Existing Approaches on GeForce GTX 280

respect to number of rows/columns of matrix, number of non-zeros in matrix, presence of uniformly or non-uniformly aligned dense sub-blocks of single block size, presence of dense sub-blocks of varied size, presence of irregularity in the structure, etc.

5.1 Existing Implementations

Parallel SpMV Implementations using Scan Primitives: NVIDIA has released a library called CUDPP [6] for data-parallel algorithm primitives, which has an implementation for SpMV for NVIDIA GPUs. The CUDPP library implements the SpMV kernel using *segmented scan* approach as proposed by Sengupta et al. [18]. Their algorithm [18] is extended from the scan algorithms proposed by Blelloch et al. [4].

The SpMV implementation ($x = Ay$) using segmented scan can be performed in three steps – (1) Compute the product $A_{ij}y_j$ for each non zero element A_{ij} and the result would be an array of products, (2) Perform a segmented scan using addition operator on the array of products (Each row in the sparse matrix corresponds to a segment), and (3) Gather the sum accumulated in the first (or last) element of each segment in the output vector.

The implementation of segmented scan in CUDPP library uses a tree-based technique. This has several performance limitations as pointed out by Dotsenko et al. [8]. The CUDPP implementation has inefficient global memory accesses, shared memory accesses with bank conflicts in some stages of their algorithm, and higher synchronization across threads. Dotsenko et al. [8] have implemented fast scan algo-

Matrix	Matrix Properties			Without Cache				With Cache		
				NVIDIA		Ours		NVIDIA		Ours
	#Rows	#Columns	#NZs	CSR	HYB	Thread Mapping	Aligned Access	CSR	HYB	Aligned Access
pwtk	217,918	217,918	11,634,424	6.23	11.92	9.47	12.06	8.11	20.72	12.43
shipsec1	140,874	140,874	7,813,404	5.77	8.63	5.89	7.45	6.68	12.36	12.41
pdb1HYS	36,417	36,417	4,344,765	6.82	7.29	7.17	9.29	9.42	13.07	17.46
rail4284	4,284	1,092,610	11,279,748	3.98	2.10	4.52	4.68	4.16	2.53	4.87
mc2depi	525,825	525,825	2,100,225	1.44	7.1	8.09	8.17	1.84	8.56	8.40
raefsky3	21,200	21,200	1,488,768	8.09	14.31	8.27	12.14	10.25	16.80	19.35
olafu	16,146	16,146	515,651	6.34	10.09	5.90	7.30	8.81	12.98	11.17
bcsstk35	30,237	30,237	740,200	6.02	7.24	5.19	6.17	8.43	9.99	9.54
venkat01	62,424	62,424	1,717,792	5.73	9.61	6.40	8.29	8.41	15.26	12.27
nasasrb	54,870	54,870	1,366,097	5.69	8.64	5.68	7.20	8.16	12.98	10.71
ex11	16,614	16,614	1,096,948	6.06	9.85	6.37	6.39	8.85	13.54	13.43
rdist1	4,134	4,134	94,408	4.12	2.19	3.16	3.65	4.16	1.91	4.79
orani678	2,529	2,529	90, 158	2.40	0.21	2.77	2.81	2.98	0.19	3.41
rim	22,560	22,560	1,014,951	5.50	7.20	5.91	6.20	7.81	9.70	11.41
rma10	46,835	46,835	2,374,001	5.97	7.17	6.75	8.94	8.34	11.08	16.35
lp_osa_60	10,280	243,246	1,408,073	0.58	0.08	0.60	0.60	0.63	0.08	0.64
fxm4_6	22,400	47,185	265,442	3.19	3.00	2.86	3.88	3.47	3.08	4.70
para-7	155,924	155,924	5,416,358	2.94	3.96	3.34	3.58	5.60	6.19	7.69
e40r0100	17,281	17,281	553,562	5.00	4.84	4.97	5.66	7.12	6.72	9.39

Table 2. Performance measures (in GFLOPS) on GeForce GTX 280

rithms on GPUs using a matrix-based technique, which outperforms the scan primitives in CUDPP. The matrix-based segmented scan algorithm significantly reduces the shared memory bank conflicts, improves global memory accesses, and reduces synchronization. The algorithm is explained in detail in [8]. We implemented the segmented scan algorithm from [8] and implemented SpMV using the matrix-based segmented scan algorithm, following the afore-mentioned steps. We refer to this implementation of SpMV in further discussion as the *Segmented Scan* implementation. We use CUDPP version 1.0 alpha for our comparative evaluation.

NVIDIA SpMV Library: NVIDIA has recently released a library specifically for SPMV computation. They discuss various standard storage formats for sparse matrices and their applicability on NVIDIA GPUs. The formats with which sparse matrices are represented in the library are (1) Diagonal (DIA) format suited for matrices restricted to a small number of matrix diagonals, (2) ELL format where non-zeros are stored as a dense matrix, (3) Coordinate (COO) format, (4) CSR format, (5) Hybrid (HYB) format combining ELL and COO formats, and (6) Packet (PKT) format which is tailored for symmetric mesh-based matrices. The details of the implementation are discussed in [2].

5.2 Performance Evaluation

We first compare the performance of our implementation with that of CUDPP and Segmented Scan implementations. Fig. 3 and Fig. 4 show the performance comparison measures (in GFLOPS) on 8800 GTX and GTX 280, respectively, on eight representative diverse matrices (out of the 19 matrices) belonging to different classes in terms of sparse matrix structure. “Compile-time Optimizations” in Figures 3 and 4 refer to the optimizations such as Optimized Thread Mapping and Aligned Global Memory Access discussed in Section 4. It can be clearly observed that in all cases, our optimizations out-perform both the CUDPP and Segmented Scan implementations. The CUDPP implementation, as discussed earlier, results in non-optimal global and shared memory accesses, leading to poor overall performance. The Segmented Scan implementation has an optimized segmented scan primitive. However, as discussed above, SpMV implementation using segmented scan requires three steps, and at least the step involving the product computation and that involving the segmented scan operation have to be launched as separate kernels. This results in additional kernel invocation overhead and additional copy overhead as values have to be written on to global memory in the first kernel to be used in the second kernel. Also, segmented scan has unwanted memory accesses and computation as the segmented scan primitive computes the prefix sum for each element of the segment whereas for SpMV it is enough to find the prefix sum of the first (or last) element of the segment. Another major setback with Segmented Scan implementation is that the segmented scan primitive works on a block of array and the entire block is copied on to

shared memory. Hence it can work only on a block that can fit in shared memory, at a time. So if elements belonging to a segment (in this case, row of a sparse matrix) span across blocks, then it involves unnecessary movement of partial results to and from global memory resulting in high memory access overhead. Hence it is always optimal to maintain synchronization-free parallelism by maintaining the computations of a row within a thread block. The non-existence of such a partition of computation is a cause for poor performance of the implementation. Our SpMV implementation yields about 2× to 8× higher performance compared to that of CUDPP and Segmented Scan implementations on 8800 GTX. The performance gap is even higher (about 2× to 10×) on GTX 280.

We focus the rest of our evaluation on comparison of our approach against NVIDIA SpMV library. The NVIDIA SpMV library failed to execute on 8800 GTX and hence those performance measures are not available. The HYB format of the NVIDIA SpMV library being the hybrid of ELL and COO formats, exhibits a better performance than ELL or COO formats in most cases, as discussed in [2]. The DIA and PKT formats are special formats tailored for certain specific patterns of sparse matrices. Hence, we compare our implementation only with the implementations using CSR and HYB formats of the SpMV library. Table 2 shows the performance measures (in GFLOPS) on NVIDIA GeForce GTX 280, for all 19 sparse matrices taken for study, for all versions of our implementation and SPMV library. The columns *ThreadMapping* and *AlignedAccess* refer to our compile-time optimizations such as optimized thread mapping and aligned global memory access, as explained in Section 4. It can be noted from Table 2 that we compare versions that use texture cache and those that don't use texture cache.

As explained in Section 4, we base our optimizations on the CSR format. Hence we first compare our optimized version (with and without cache) with NVIDIA SpMV library's version using optimized CSR format (with and without cache). The optimized CSR implementation from NVIDIA SpMV differs from our optimized version in two key aspects - (1) they map 32 threads per row, as opposed to 16 in our version and (2) they do not make "alignment adjustment" in computation unlike our version that does "aligned access" optimization (and hence their implementation will result in many non-coalesced accesses). Our approach outperforms their version using optimized CSR format (with and without texture cache) for all matrices under study. Our approach is less efficient than the version using HYB format when the number of non-zeros per row is less than 16, as some threads may remain idle. The version using HYB format exhibits poor performance when the number of non-zeros per row varies widely across the matrix (for e.g. "rail4284" matrix [2]). In general, our approach achieves better or equivalent performance when compared to NVIDIA SpMV library, *with no change in the sparse matrix storage format*.

	Number of threads		
Matrix	16	32	64
shipsec1	12.41	7.31	1.38
pdb1HYS	17.46	11.45	2.88
raefsky3	19.35	10.89	3.28
venkat01	12.27	9.28	1.49
rma10	16.35	10.04	2.45

Table 5. Performance measures (in GFLOPS) for varying number of threads per row on GeForce GTX 280

Bell et al. [2] compare the GPU SpMV results obtained from NVIDIA SpMV library and SpMV results on various different multi-core platforms obtained by Williams et al. [23] and illustrate that the GPU results offer the best performance. Hence comparing our approach with the NVIDIA SpMV library will give a clear picture of how our GPU results would compare against the results on different multi-core platforms.

5.3 Tuning Configuration Parameters

Varying number of threads and thread blocks: We now discuss about the variation in performance of our techniques for different GPU execution configurations, i.e. for varying numbers of threads and thread blocks. The number of threads per thread block has a direct implication on the effective utilization of processors to hide the latency of global memory accesses. Fewer threads (resulting in fewer warps) might fail to hide the latency of global memory access when the number of active thread blocks in a multiprocessor is low. We vary the number of threads per thread block used for execution (as 64, 128, 256, and 512) and measure the performance for five representative matrices for the "Aligned Access" version. The number of thread blocks in each case varies according to the number of rows in the matrix. The results are presented in Table 4. From the results, we see the variation in performance for varying number of threads is not much indicating that the optimizations that are realized are very effective in tolerating the global memory latency.

The number of thread blocks in the above experiment, as mentioned, depends on the number of rows in the matrix and is equal to $\frac{number\ of\ rows \times number\ of\ threads\ handling\ a\ row}{number\ of\ threads\ in\ a\ thread\ block}$. However if the number of thread blocks is kept fixed at a value irrespective of the number of rows, then depending on the number of rows, some thread blocks may have to handle multiple blocks of rows. This requires additional loop overhead in the code and would result in poor performance. Our experiments confirmed the same and hence we did not keep the number of thread blocks at a fixed value for our experiments.

Varying number of threads handling a row: We conducted experiments to study how effective is the choice of using 16 threads to handle the computation pertaining to a row. The choice, as explained earlier, is primarily based on the architectural feature that memory coalescing rules are

Matrix	Without Cache Number of threads 64	128	256	512	With Cache Number of threads 64	128	256	512
shipsec1	7.25	7.45	7.38	7.24	10.1	12.41	12.04	11.65
pdb1HYS	9.15	9.29	9.27	9.05	17.01	17.46	17.00	16.38
raefsky3	12.14	11.32	11.32	11.44	16.69	19.35	18.2	17.13
venkat01	8.05	8.29	8.01	8.06	11.16	12.27	11.90	11.45
rma10	8.65	8.94	8.74	8.71	13.00	16.35	15.98	15.40

Table 4. Performance measures (in GFLOPS) for varying number of threads per thread block on GeForce GTX 280

defined for a half warp, i.e. 16 threads. Hence it is the minimum number of threads that can guarantee coalescing. The number of non-zeros can be very low, even less than 16. When 16 is chosen as a general default choice, the penalty incurred for such cases (when there are less non-zeros per row) would be less compared to that while choosing 32 or 64. Also, there is a reduction involved per row for reducing the partial products that are computed in parallel by the threads. 16 threads lead to lower reduction cost while utilizing good amount of parallelism.

We measure the performance for five representative matrices for the "Aligned Access" version (using texture cache). The results of the experiment are presented in Table 5. For the reasons explained above, the performance when the choice is 16 threads per row, is higher than the other choices of 32 and 64. This also explains why our implementation always outperforms NVIDIA's implementation using CSR format which uses 32 threads per row.

6. Related Work

Over the last two decades, there has been significant amount of work on optimizing sparse matrix computations (SpMV). Most of the work have concentrated on optimizing sparse matrix kernels on general-purpose architectures. SpMV being a memory-bound kernel, most of the optimizations target performance improvements at various memory levels in memory hierarchy. The optimizations broadly include optimal data structure for storing the sparse matrix [3], exploiting block structures in sparse matrix [22, 11], and blocking for reuse at the level of cache [15, 20], TLB [15], and registers [13, 12]. OSKI [21] is a state-of-the-art library collection providing low-level primitives for automatically tuned kernels on sparse matrices. OSKI uses techniques extensively from the SPARSITY sparse-kernel automatic tuning framework [12] for arriving at optimizations for sparse kernels. Unfortunately, the optimization techniques proposed for cache-based general-purpose architectures cannot be directly applied for GPU architecture. GPUs are massively parallel systems in which having more concurrently active threads are critical for performance, especially for hiding high latency memory accesses by effective thread scheduling. This is because when there are more active threads, when some threads are busy waiting for the completion of memory access request, the thread scheduler can switch control over to other threads, thereby keeping the system busy without stalling as far as possible. Therefore, fine-grained thread-level parallelism is beneficial for GPUs, and hence, in most cases data reuse across threads is better rather than reuse within a thread. While spatial locality and temporal locality are very important to exploit at the level of cache or registers in general-purpose architectures, mapping of computation among threads that result in optimal memory access pattern has to be considered in GPU architectures which, in some cases, can negate locality, but yet turn out to be beneficial.

Recently, Williams et al. [23] emphasize and substantiate the need for *multicore specific* optimization strategies for various emerging multicore platforms including AMD dual-core, Intel quad core, STI Cell, and Sun Niagara2 systems. They clearly quantify the extent of significance of memory bandwidth bottleneck for increasing number of cores and motivate memory bandwidth reduction for SpMV computations. Our work also, on the same lines, emphasizes optimization strategies that are specific to the GPU architecture taking into consideration the complex GPU memory organization and the non-trivial optimal mapping of computation among threads.

There are several sparse matrices corresponding to real applications which possess dense block substructures. Exploiting the presence of dense blocks will help in enhancing data reuse, especially, of the input vector elements. The dense block structure may either contain same size blocks that are uniformly aligned or same size blocks that are non-uniformly aligned or varied size blocks that are irregularly aligned [22]. The Block CSR (BCSR) [11] and Unaligned Block CSR (UBCSR) sparse storage formats are proposed to improve sparse matrix computations by effectively handling dense sub-blocks in sparse matrices. These approaches identify small dense blocks which are more suited for register blocking in traditional architectures and in short-vector processors. Buatois et al. [5] have developed a sparse linear solver on GPUs and have implemented SpMV, the primary kernel in the solver, using the BCSR format for register blocking. They have implemented using AMD's (then ATI's) Close-To-Metal (CTM) API for general-purpose computation on ATI GPUs. The GPUs they have targeted are the ATI

X1k series which have multiple pipelines and each pipeline has a 4-element vector processors. However in modern massively parallel SIMD architecture of NVIDIA GPUs which has scalar processors executing in SIMD fashion in a multiprocessor, the BCSR format with small dense blocks leads to coarse-grained parallelism that enhances register level data reuse, but results in non-optimal global memory accesses.

There has been several works that perform a runtime processing to reorder computation and data for locality enhancement for cache-based architectures (e.g. [14]). Strout et al. [19] developed a compile-time framework for composing run-time data and computation reordering for data locality. However in our work, we neither perform any such heavy runtime processing nor use a compiler framework to facilitate such a runtime reordering. There has been work on optimizing SpMV [10] on NVIDIA chips that are pre-CUDA. However, the memory access constraints and characteristics pertaining to the off-chip DRAM (global memory) in those chips are completely different from that of the current NVIDIA chips.

7. Conclusions and Future Work

In this work, we have presented the key architectural optimizations that have to be addressed in GPUs for efficient execution. We have analyzed the various challenges in extracting high-performance from a prominent memory-bound scientific kernel like SpMV on NVIDIA GPUs using CUDA and have developed optimizations that take into account both the application and the architectural characteristics. We have evaluated our techniques over two classes of NVIDIA GPU chips, namely, GeForce 8800 GTX (having 128 cores per chip) and GeForce GTX 280 (having 240 cores per chip). We have obtained significant performance improvements over existing parallel SpMV implementations, on both the GPU chips, clearly indicating the effectiveness of our approach to scale the performance of SpMV for increasing number of cores per chip.

We plan to extend our approach to include a more sophisticated runtime inspection module that can effectively reorder data and computation to further exploit data reuse and optimize memory access. We also plan to integrate auto tuning infrastructure into our approach to determine optimal block sizes for arbitrary irregular sparse matrices.

References

[1] AMD Stream SDK. http://ati.amd.com/technology/streamcomputing/.

[2] N. Bell and M. Garland. Efficient Sparse Matrix-Vector Multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[3] A. J. C. Bik and H. A. G. Wijshoff. Automatic Data Structure Selection and Transformation for Sparse Matrix Computations. *IEEE Trans. Parallel Distrib. Syst.*, 7(2):109–126, 1996.

[4] G. E. Blelloch. Prefix Sums and Their Applications. Technical report, 1990.

[5] L. Buatois, G. Caumon, and B. Levy. Concurrent Number Cruncher: An Efficient Sparse Linear Solver on the GPU. In *High Performance Computation Conference (HPCC), Springer Lecture Notes in Computer Sciences*, 2007.

[6] CUDPP: CUDA Data Parallel Primitives Library. http://www.gpgpu.org/developer/cudpp/.

[7] T. Davis. The University of Florida Sparse Matrix Collection. *ACM Trans. on Mathematical Software*. http://www.cise.ufl.edu/research/sparse/matrices.

[8] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli. Fast Scan Algorithms on Graphics Processors. In *ACM ICS'08*, pages 205–213, 2008.

[9] General-Purpose Computation Using Graphics Hardware. http://www.gpgpu.org/.

[10] J. B. Ian, I. Farmer, E. Grinspun, and P. Schroder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM Transactions on Graphics*, 22:917–924, 2003.

[11] E.-J. Im and K. A. Yelick. Optimizing Sparse Matrix Computations for Register Reuse in SPARSITY. In *Proceedings of the International Conference on Computational Science*, volume 2073 of *LNCS*, pages 127–136, San Francisco, CA, May 2001. Springer.

[12] E.-J. Im, K. A. Yelick, and R. Vuduc. SPARSITY: Framework for Optimizing Sparse Matrix-Vector Multiply. *International Journal of High Performance Computing Applications*, 18(1):135–158, February 2004.

[13] J. Mellor-Crummey and J. Garvin. Optimizing Sparse Matrix-Vector Product Computations Using Unroll and Jam. *Int. J. High Perform. Comput. Appl.*, 18(2):225–236, 2004.

[14] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings. *Int. J. Parallel Program.*, 29(3), 2001.

[15] R. Nishtala, R. Vuduc, J. Demmel, and K. Yelick. When Cache Blocking Sparse Matrix Vector Multiply Works and Why. In *Proceedings of the PARA'04 Workshop on the State-of-the-art in Scientific Computing*, Copenhagen, Denmark, June 2004.

[16] NVIDIA CUDA. http://developer.nvidia.com/object/cuda.html.

[17] Open Computing Language (OpenCL). http://www.khronos.org/news/press/releases/khronos_launches_heterogeneous_computing_initiative/.

[18] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan Primitives for GPU Computing. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 97–106, 2007.

[19] M. M. Strout, L. Carter, and J. Ferrante. Compile-time Composition of Run-time Data and Iteration Reorderings. In *ACM PLDI '03*, 2003.

[20] O. Temam and W. Jalby. Characterizing the Behavior of Sparse Algorithms on Caches. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 578–587, 1992.

[21] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A Library of Automatically Tuned Sparse Matrix Kernels. In *Proceedings of SciDAC 2005*, Journal of Physics: Conference Series, San Francisco, CA, USA, June 2005. Institute of Physics Publishing.

[22] R. Vuduc and H.-J. Moon. Fast Sparse Matrix Vector Multiplication by Exploiting Variable Block Structure. In *Proceed-*

ings of the International Conference on High-Performance Computing and Communications, LNCS 3726, Sorrento, Italy, September 2005.

[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 *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, 2007.