RC24704 (W0812-047) December 8, 2008
Computer Science

IBM Research Report

Optimizing Sparse Matrix-Vector Multiplication on GPUs Using Compile-time and Run-time Strategies

Muthu Manikandan Baskaran
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH
USA

Rajesh Bordawekar
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
USA

Optimizing Sparse Matrix-Vector Multiplication on GPUs using Compile-time and Run-time Strategies

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.

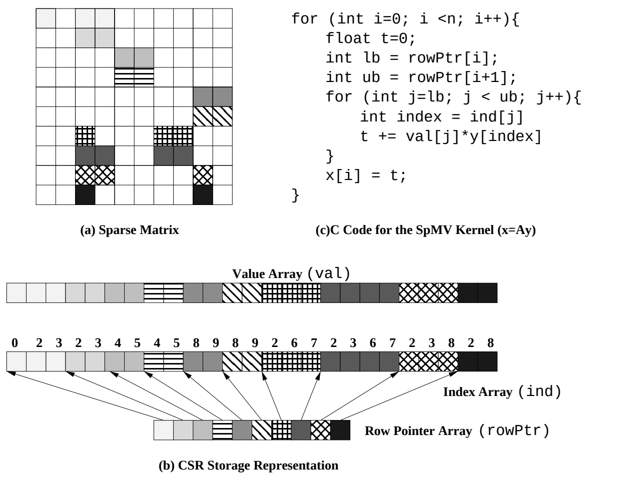
In this paper, we evaluate the various challenges in developing a high-performance SpMV kernel on NVIDIA GPUs using the CUDA programming model and propose a framework that employs both *compile-time* and *run-time* optimizations. The compile-time optimizations include: (1) exploiting synchronization-free parallelism, (2) optimized thread mapping based on the *affinity towards optimal memory access pattern*, (3) optimized off-chip memory access to tolerate the high latency, and (4) exploiting data reuse. The runtime optimizations involve a runtime inspection of the sparse matrix to determine dense non-zero sub-blocks, which facilitate the reuse of input vector elements while execution. We propose a new blocked storage format for storing and accessing elements of a sparse matrix in an optimized manner from the GPU memories. 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 the one from NVIDIA's CUDPP library and the one implemented using optimal segmented scan primitive. Our approach outperforms the other existing implementations by a factor of 2 to 4. Using our framework, we achieve a peak SpMV performance that is 70% of the performance observed for SpMV computations using dense matrices stored in sparse format.

1. Introduction

Modern computer architecture has shifted towards designs that employ multiple processor cores on a chip, so called *multicore* processors. Unfortunately, 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 architectural-specific optimization strategies involve specific optimizations targeted towards improving memory characteristics 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 ($x = Ay$).



```
for (int i=0; i <n; i++){
    float t=0;
    int lb = rowPtr[i];
    int ub = rowPtr[i+1];
    for (int j=lb; j < ub; j++){
        int index = ind[j]
        t += val[j]*y[index]
    }
    x[i] = t;
}
```

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 [20]. Sparse matrix computations involve far more memory accesses per floating point operation, due to indirect and irregular memory accesses. Higher performance in SpMV computation requires optimizations that best utilize the properties of the sparse matrix and also the underlying system architecture. If the sparse matrix structure is known only at runtime, then runtime optimizations are needed to analyze and utilize the properties of the sparse matrix to yield better performance. 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 peak performance of over 350 GFLOPS and the NVIDIA GeForce GTX 280 chip has a 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) [7]. 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) [15], Stream SDK [1], and OpenCL [16], have made GPUs powerful and attractive choice for developing high-performance numerical and scientific computations. Unfortunately, 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 (Figure 2). 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 revealed two key inter-related obstacles in improving the SpMV performance on the NVIDIA GPUs: thread mapping and data access strategies. To address these concerns, we have proposed a system that uses both compile- and runtime optimizations. We have implemented and tested this system on two state-of-the-art NVIDIA GPUs, 8800 GTX and GTX 280, using a set of sparse matrices used in a wide variety of application domains. We have also experimentally evaluated our approach against two existing SpMV CUDA implementations, namely, NVIDIA's CUDPP [4] library and the one implemented using optimal segmented scan primitives from Dotsenko et al. [6]. Our results demonstrate that in majority of cases, our compile-time and runtime optimizations *substantially* improved performance of the SpMV kernel over the naive implementation, the implementation using segmented scan, and the NVIDIA CUDPP library. In some cases, we observe a factor of 6 improvement over the naive implementation, and a factor of 2 to 4 over CUDPP and scan implementations. The experimental results conclusively demonstrate the advantages of our optimizations over existing approaches. Our framework has been implemented and it is in the process of being released as an Open Source project.

Our study makes the following key contributions:

1. We have evaluated the various challenges in developing a high-performance SpMV kernel on NVIDIA GPUs using the CUDA programming model and proposed a framework that employs both *compile-time* and *run-time* optimizations.
2. We have developed a compile-time optimizer that applies the following optimizations to a SpMV code executing on a GPU: (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 latency, and (4) exploiting data reuse. We have proposed a new blocked storage format for storing and accessing elements of a sparse matrix in an optimized manner from the GPU memories. We have also implemented an optional runtime optimizer that first performs an inspection of the sparse matrix to identify dense non-zero sub-blocks that could facilitate the reuse of input vector elements while execution.
3. We have evaluated 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. Our optimization techniques result in significant performance improvements on both the GPUs over existing parallel SpMV implementations by a factor of 2 to 4. Using our framework, we are able to achieve a peak SpMV performance that is 70% of the performance observed for SpMV computations using dense matrices stored in sparse format.

The rest of the paper is organized as follows: Section 2 presents an overview of the NVIDIA GPU architectures and the CUDA programming model. The problem statement is presented in Section 3. Section 4 describes the proposed SpMV optimization framework 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, the CUDA programming interface, and the GPU execution model.

2.1 GPU Computing Architecture

The GPU parallel computing architecture comprises of a set of multiprocessor units called the *streaming multiprocessors (SMs)*, each one containing a set of processor cores (called the *streaming processors* (SPs)). The NVIDIA GeForce 8800 GTX has 16 SMs each consisting of 8 SPs and the NVIDIA GeForce GTX280 has 30 SMs with 8 SPs in each SM. The SPs within a SM communicate through a fast explicitly managed on-chip local store memory, also called the *shared memory*, while the different SMs communicate through a slower off-chip DRAM, also called the *global memory*. Each SM unit also has a fixed number of *registers*.

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 (768MB on the 8800 GTX), (2) off-chip local memory, (3) on-chip shared memory (16KB per multiprocessor in 8800 GTX), (4) off-chip constant memory with on-chip cache (64KB in 8800 GTX), and (5) off-chip texture memory with on-chip cache. Fig. 2 illustrates the memories in GPUs along with their hierarchical order and access latencies.

2.2 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 [15]. 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). The dimensions of the thread and thread block are specified at the time of launching the kernel, through the identifiers *blockDim* and *gridDim*, respectively. The dimensions may be 1, 2 or 3.

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 and variables in this space are declared using the *__shared__* identifier. The GPU DRAM is accessible by all threads in a kernel.

2.3 GPU Execution Model

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 Program Counter. The SM hardware employs a zero overhead warp scheduling through the CUDA runtime scheduler. Warps whose next instruction has its operands ready are eligible for execution and eligible warps are selected for execution on a prioritized scheduling policy. The warp scheduling is completely transparent to the CUDA programmer.

The computational resources on a multiprocessor unit, i.e., the shared memory and the register bank, are shared among the active thread blocks on that unit. For example, an application abstracted as a grid of 64 thread blocks can have 4 thread blocks mapped on each of the 16 multiprocessors of the NVIDIA GeForce 8800 GTX. The GeForce 8800 GTX GPU has a 16 KB shared memory space and 8192 registers. If the shared memory usage per thread block is 8 KB or the register usage is 4096, at most 2 thread blocks can be concurrently active on a multiprocessor. When any of the two thread blocks complete execution, another thread block can become active on the multiprocessor.

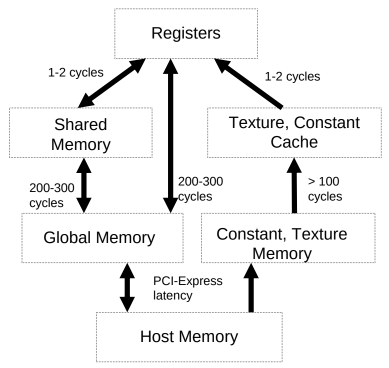


Figure 2. Memory Hierarchy in the NVIDIA GPUs.

3. Problem Statement

GPUs are massively data-parallel systems with very high per-chip parallelism. A NVIDIA 8800 GTX GPU has a theoretical peak performance of around 350 GFlops and a peak off-chip memory bandwidth of over 85 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 properties of 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 memory organization is designed to improve the memory throughput of applications executed on GPUs. However, 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 a framework for optimizing SpMV computations on GPUs that uses compile- and run-time strategies to match application requirements against the architectural constraints.

4. System Design and Implementation

In this Section, we discuss in detail the design and implementation of our framework for optimizing SpMV computations on GPUs. The main components of the framework are: (1) a module performing compile-time optimizations, (2) runtime inspector that analyzes the sparse matrix structure, and (3) a module executing the optimized kernel on GPU device.

Figure 3 sketches the design of our proposed system. The runtime inspector analyzes the sparse matrix structure to derive optimized block storage format. The inspector is invoked over the input sparse matrix at the host (CPU) side to perform the structural analysis. Since runtime preprocessing incurs some overhead, it is an optional module that can be by-passed to only apply compile-time optimizations. However, in many real applications, e.g., the Conjugate Gradient Solver, the same sparse matrix is used repeatedly and the cost of preprocessing the sparse matrix could then be amortized over the multiple iterations. The runtime strategy to extract optimal block structure is explained in detail in Section 4.2.

The compile-time optimizer module is invoked on the host for determining the set of compile-time optimizations that would be

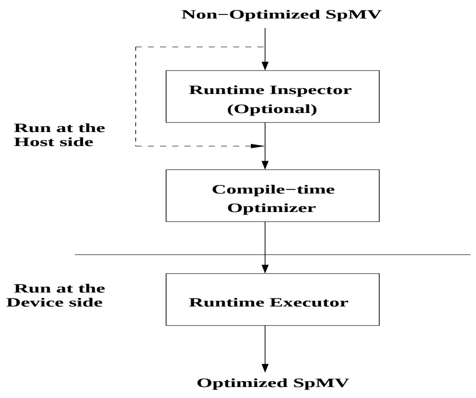


Figure 3. System Design and Modules

applied to the SpMV kernel. The various compile-time optimizations that are devised for efficient execution of SpMV kernel on GPU architecture are described in detail in Section 4.1. The final optimized SpMV kernel CUDA code depends on the runtime and compile-time optimizations that are applied. The optimized kernel is executed on the GPU device by the runtime executor. It should be noted that the storage format for the sparse matrices in our framework is our block storage format when our runtime preprocessing is performed, and it is the CSR format when only our compile-time optimizations are performed.

4.1 Compile-time Optimizations

```
int tid = threadIdx.y;
int bid = blockIdx.y;
int myi = bid * BLOCKSIZE + tid;

if (myi < n) {
    float t=0;
    int lb = rowPtr[myi];
    int ub = rowPtr[myi+1];
        for (int j=lb; j<ub; j++) {
            int index = ind[j];
            t += val[j] * y[index];
        }
    x[myi] = t;
}
```

Figure 4. Naive CUDA SpMV code

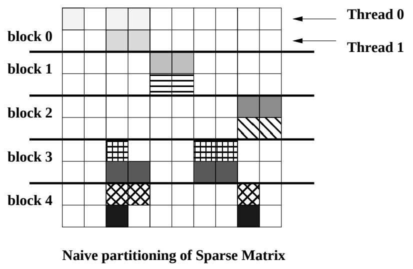


Figure 5. Naive Thread Mapping in a CUDA execution

In this Section, we present some of the key performance optimizations that can be performed at compile-time to improve performance on GPUs. 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.

The compile-time optimizations targeted to improve the performance of SpMV on GPUs are as follows:

1. **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.

2. **Optimized Thread Mapping**: The naive way of parallelizing SpMV is to allocate one thread to perform the computation corresponding to one row and a thread block to handle a set of rows, as shown in Figure 5. Figure 4 shows the CUDA code corresponding to such a naive mapping in which a one-dimensional grid of thread blocks and a one-dimensional block of threads computing are used to perform SpMV. However 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 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 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 the 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.

3. **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 (i.e. group of 16 threads). 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 non-zeros of the sparse matrix. Hence the computation involving a row should be adjusted to first access the non-aligned portion of the row, before proceeding to access the aligned portion. If this adjustment is not made, the entire row would be accessed non-optimally and the memory access cost would increase.

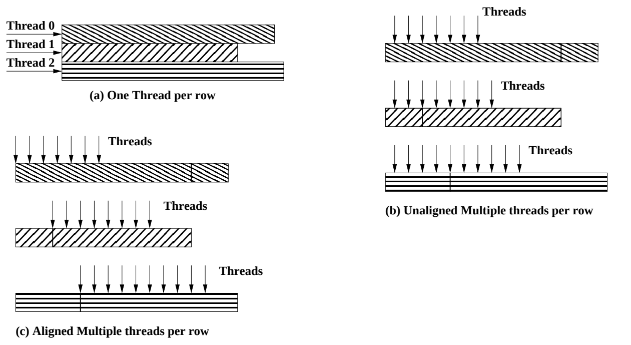

Thread 0
Thread 1
Thread 2
(a) One Thread per row
Threads
(b) Unaligned Multiple threads per row
(c) Aligned Multiple threads per row

Figure 6. Illustration of our Compile-time Optimizations such as Optimized Thread Mapping and Optimized (Aligned) Global Memory Access

Figure 6 illustrates the afore-mentioned optimizations. Figure 7 displays the measured performance numbers (in GFLOPS) and clearly depicts the performance improvements achieved for few sample sparse matrices due to optimal thread mapping and optimal global memory access as opposed to just exploiting synchronization-free parallelism as illustrated in Figures 4 and 5. The performance improvement in GTX 280 due to optimized global memory accesses, after the alignment adjustment, is less compared to that in 8800 GTX due to the less constrained coalescing requirement in the GTX 280 architecture.

4. **Exploiting Data Reuse**: The input and output vectors are the ones that exhibit data reuse in SpMV computation. Exploiting the reuse of input vector elements depends on the non-zero access pattern of the sparse matrix. Hence it requires run-time analysis of the sparse matrix. Our runtime analysis to create an optimal block storage of the sparse matrix is explained in Section 4.2. The reuse of output vector elements is performed at compile-time and exploiting synchronization-free parallelism with optimized thread mapping 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.

5. **Other Optimizations**: The other optimizations that are performed are: (1) optimizing shared memory access by minimizing bank conflicts through effective padding of the array(s) in shared memory, (2) avoiding divergence among threads of a thread block (to be specific, warp), and (3) reducing loop overhead.

4.2 Run-time Inspection of the Sparse Matrix

Higher performance in SpMV computation needs optimizations that best utilize the properties of the sparse matrix and also the target architecture. There are several sparse matrices corresponding to real applications which possess dense block sub-structures. Exploiting the presence of dense blocks is very critical for obtaining high performance in SpMV computation, as it will help in reducing the number of memory loads/stores by enhancing data reuse, especially, of the input vector elements. However, extracting dense blocks in a given sparse matrix requires an inspection of the matrix. If the sparse matrix is available only at runtime, then the inspection has to be performed at runtime. 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 [21]. The Block CSR (BCSR) [8] and Unaligned Block CSR (UBCSR) sparse storage formats are proposed to improve sparse matrix computations by effectively handling dense sub-blocks in sparse matrices.

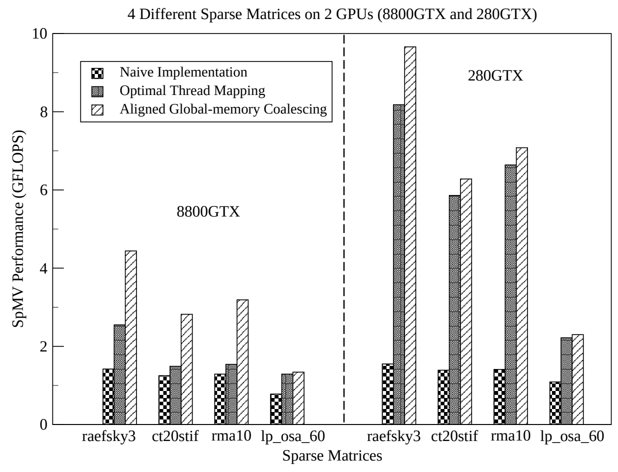


Figure 7. Comparative Evaluation of our Compile-time Optimizations such as Optimized Thread Mapping and Optimized Global Memory Access

The BCSR format is more constrained with respect to the alignment of blocks along rows and columns, and enforces the starting row (and column) of the block to be a multiple of the block size along row (and column). BCSR format reduces index storage overhead, but might lead to filling of more zero entries. The UBCSR format reduces the number of fill-in zeros, but it has more index storage. The UBCSR format is more generalized to handle both uniformly aligned and non-uniformly aligned blocks, but, in most cases, extracts *small* dense blocks of varied sizes. In the case of sequential optimization alone, the dense blocks are exploited for data reuse. In the case of parallel optimization, different blocks are optimized across different processes. Hence extracting small dense blocks is more suited for coarse-grained parallelism. However, for those GPUs that exhibit two-level parallelism with preferable fine-grained thread-level parallelism, exploiting smaller dense blocks would result in inefficient global memory accesses. Another disadvantage of implementing UBCSR format in GPUs is that the UBCSR format has additional storage overhead in the form of indices pointing to the starting row and column of each block and that of block size. The additional storage overhead would cause heavy memory access penalties in GPUs.

We propose a new block storage format that suits to GPU architecture. Our block storage format is a hybrid one between constant and variable block formats that tries to exploit the benefits of both the formats. The features of our format are:

1. To reduce the memory access penalty in reading block size and block index, our block storage format sticks to constant block sizes that enable fine-grained thread-level parallelism.

2. Our format tries to minimize the number of zero entries that are filled. To incorporate the filling of less zero entries, we relax the constraint that starting column of a block should be a multiple of the block size along column. However, we enforce that starting column should adhere to the alignment constraints of global memory coalescing. Hence, we store the starting column index of each block and also the number of column blocks for each row block.
3. Also, 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. This would help in enabling coalesced accesses in a block with less number of fill-in zeros.

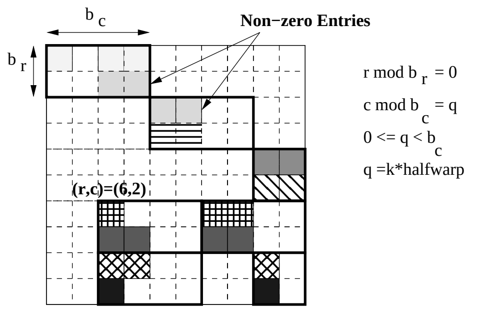


Figure 8. GPU-specific Block Storage Format

Let b_r be the block size along row, b_c the block size along column, and (r,c) be the starting row and column numbers of a block. Our block storage format has the following constraints:

$$r \ mod \ b_r = 0 \quad \wedge \quad c \ mod \ b_c = q$$

where q is either zero or a multiple of half warp size that is less than b_c. We enforce b_c also to be a multiple of half warp size. Figure 8 illustrates our block storage approach. As shown in the Figure, the blocks are of constant size and aligned according to the afore-mentioned constraints. For illustration purpose, the value of half warp in Figure 8 is assumed to be 2.

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. Hence, in our approach, we enable reuse of input vector elements at the level of shared memory and not at the level of registers, as enabling register reuse would lead to coarse-grained parallelism and disable global memory access coalescing. 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.

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 1. 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.

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	384 bits @ 1.8 GHz	512 bit @ 2.2 GHz
Peak Performance	388.8 GFLOPS	933.12 GFLOPS

Table 1. Architectural configurations of NVIDIA GeForce 8800 GTX and GeForce GTX 280

For our evaluation, we used 19 sparse matrices from the sparse matrix collection described in [5]. 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 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, etc. The first eight matrices (in the order of their appearance in Table 2 and Table 3) have dense sub-blocks of single block size that are uniformly aligned, and hence have very regular pattern. The next eight matrices have mixed block structure and the dominant blocks are smaller (in the range of 2-4) in size. The remaining matrices have quite a bit of irregularity in their structure.

5.1 Overview of Existing Parallel SpMV Implementations

NVIDIA has recently released a library called CUDPP [4] 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. [17]. Their algorithm [17] is extended from the scan algorithms proposed by Blelloch et al. [3].

A *forward inclusive scan* operation using a binary associative operator $\oplus$ is defined as follows:

$$input = [a_0 \ a_1 \ \dots \ a_{n-1}]$$

$$output = [a_0 \ a_0 \oplus a_1 \ \dots \ a_0 \oplus \cdots \oplus a_{n-1}]$$

A segmented scan operator operates on a sequence that has multiple segments and performs a scan operation on each segment separately. An *inclusive* segmented scan with addition operator can be illustrated as follows:

$$input = [[1 \ 4 \ 8 \ 9] \ [5 \ 6 \ 2 \ 4]]$$

$$output = [[1 \ 5 \ 13 \ 22] \ [5 \ 11 \ 13 \ 17]]$$

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}. 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.
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. [6]. 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. [6] have implemented fast scan algorithms on GPUs using a matrix-based technique, which outperforms the scan primitives in CUDPP. The

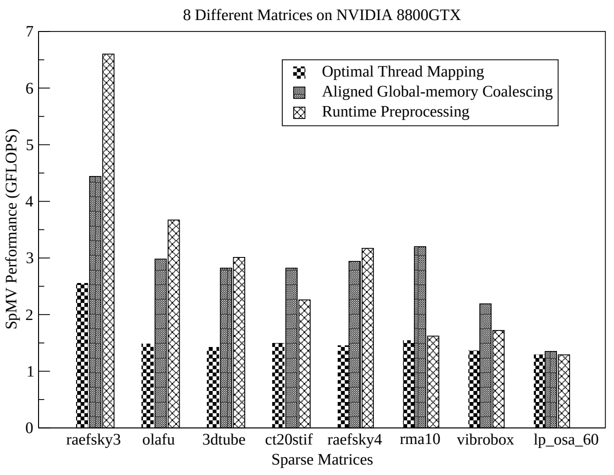


Figure 9. Evaluation of our Optimizations on GeForce 8800 GTX

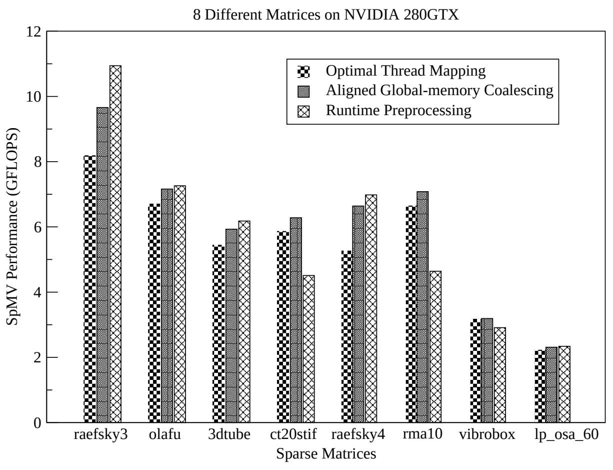


Figure 10. Evaluation of our Optimizations on GeForce GTX 280

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 [6]. We implemented the segmented scan algorithm from [6] 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.

5.2 Performance Evaluation

Table 2 and Table 3 illustrate the performance measures (in GFLOPS) on NVIDIA GeForce 8800 GTX and GeForce GTX 280, respectively, for all the 19 sparse matrices under consideration, over all implementation schemes. 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 4.1. The column *Runtime* refers to the runtime optimization involving inspection of the sparse matrix to create optimized block storage, as explained in Section 4.2. It is important to note that *aligned access* optimization encompasses *optimized thread mapping* optimization and the runtime optimization encompasses both the compile-time optimizations. The column *Naive* refers to the naive way of parallelizing SpMV by allocating one row per thread and set of rows to a thread block. The columns *CUDPP* and *SegmentedScan* refer to the CUDPP and Segmented Scan implementations as explained above. The numbers in bold identify the peak performance obtained using our optimizations. The column *RelativeGain* corresponding to each of *Naive*, *CUDPP*, and *SegmentedScan* implementations indicates the maximum performance gain achieved using our optimizations relative to that implementation. We base the rest of our explanation to eight representative diverse matrices belonging to four different classes in terms of sparse matrix structure.

First, we discuss in detail the performance improvements obtained using our compile-time and runtime optimizations on the two NVIDIA GPUs. As it can be inferred from Table 2 and Table 3, the optimized thread mapping results in significant performance improvement over naive parallelization for both GPUs (up to 1.5 times on 8800 GTX and up to 5.5 times on GTX 280). The optimization helps to tolerate global memory access latency and also assists in obtaining coalesced accesses. Figure 9 and Figure 10 present the performance gains achieved using our compile-time and run-time optimization strategies (optimizations corresponding to columns 2-4 in Table 2 and Table 3) for the eight representative sparse matrices on NVIDIA GeForce 8800 GTX and GeForce GTX 280, respectively. As discussed in Section 4.1, the alignment of global memory accesses is more critical in 8800 GTX for high performance. It is also important in GTX 280, but unaligned accesses in 8800 GTX might result in more memory transactions than that in GTX 280. This can be clearly seen in Figure 9 and Figure 10. The improvement achieved after aligned global memory access optimization over optimal thread mapping optimization is very minimal for GTX 280 whereas it is up to *1.7 times* for 8800 GTX. The performance is generally poor for matrices that are more irregular as there would be more memory transactions even after the optimizations are applied.

The runtime preprocessing optimization involves the inspection overhead and the reported performance numbers do not include them. The performance numbers indicate just the performance improvement achieved (if it is the case) due to block storage that enables data reuse of input vector elements. The block storage resulting from runtime inspection would result in filling of extra zeros and also the final computation involves additional loop overhead to iterate over blocks. In some cases, when these overheads become high, it degrades performance. As seen from Figure 9 and Figure 10, for some irregular matrices like *rma10*, the runtime optimization results in degradation of performance due to (1) additional loop overhead and (2) increased memory accesses because of the irregularity of the non-zero pattern that leads to poor data reuse of input vector. However, for more regular matrices like *raefsky3*, runtime optimization can increase the performance up to *1.7 times* over that achieved via only compile-time optimizations. We did not perform a sophisticated tuning of block sizes for our runtime block storage optimization. However we performed few empirical runs and fixed the block size along row to be 4 and block size along column to be 64 for our experiments. We also measured the runtime inspection overhead for various matrices and we found that, on an average, the inspection overhead time was around 15 times the time taken to execute the SpMV kernel on the GPU device.

Figure 11 and Figure 12 compare the performance achieved using our compile-time optimizations with that of CUDPP and Segmented Scan implementations. Since the CUDPP and Segmented Scan implementations do not involve runtime optimizations, for fairness, we have compared them with our compile-time-only and runtime optimizations. It can be clearly observed that in all cases, our both approaches 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 re-

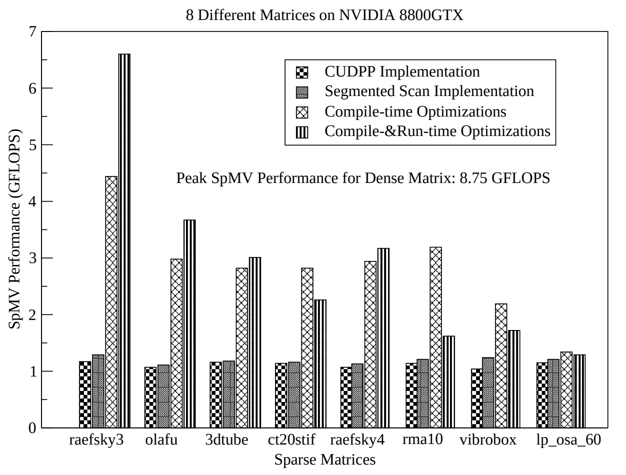


Figure 11. Comparison with Existing Approaches on GeForce 8800 GTX

quires 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 the 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 compile-time-only approach yields up to *4 times* and *1.5 times* improvement over Segmented Scan for regular and irregular matrices, respectively. For the compile-time-only approach, the performance improvement is up to 8 times and 2 times over CUDPP for regular and irregular matrices, respectively.

5.3 Profiling Architectural Metrics

The CUDA 2.0 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 compile-time optimizations. Table 4 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.

5.4 Estimation of Peak SpMV Performance

The peak performance of 8800 GTX is more than 350 GFLOPS and GTX 280 is around 933 GFLOPS. However the performance achieved for the SpMV kernel is only around 7 GFLOPS on 8800 GTX and around 10 GFLOPS on GTX 280, even for very regular matrices like *raefsky3*. Hence we estimated the peak performance that can be achieved for SpMV kernel on these GPUs. For this

Matrix	# Coalesced Accesses			# Non-Coalesced Accesses		
	Naive	Thread Mapping	Aligned Access	Naive	Thread Mapping	Aligned Access
raefsky3	172	18898	30457	582784	292676	114466
ct20stif	940	2423	34237	546081	496638	173269
rma10	414	3693	48289	894942	847676	295968
lp_osa_60	273	413	19181	1085155	189438	134673

Table 4. Profiling Coalesced and Non-coalesced Accesses on 8800 GTX. Number of coalesced accesses increases as the two compile-

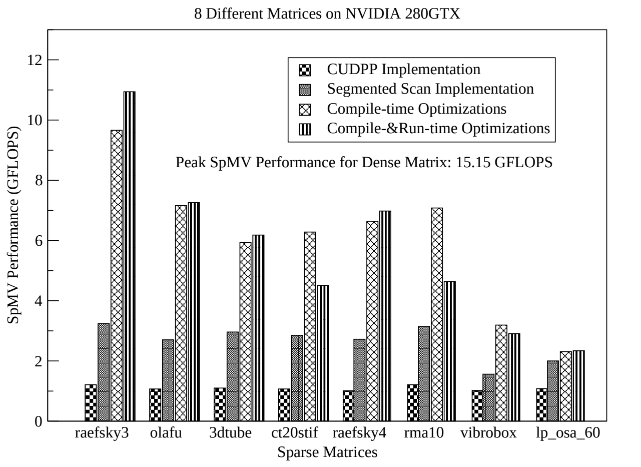


Figure 12. Comparison with Existing Approaches on GeForce GTX 280

purpose, we stored a dense matrix using our block compressed sparse row storage format and performed SpMV computation using the matrix [23]. By doing so, we could completely exploit data reuse, and at the same time, also measure the overhead due to indirect accesses. The maximum performance achieved using the dense matrix stored in sparse format was around 8.75 GFLOPS on 8800 GTX and around 15.15 GFLOPS on GTX 280. Using our compile-time and runtime optimizations, we were able to achieve 75% and 70% of the "practical" peak performance on 8800 GTX and GTX 280, respectively (for matrix *raefsky3*, our optimizations achieved 6.6 GFLOPS on 8800 GTX and 10.9 GFLOPS on GTX 280).

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 [2], exploiting block structures in sparse matrix [21, 22, 8], blocking for reuse at the level of cache [14, 19], TLB [14], and registers [12, 9], and locality-enhancing reordering [10]. OSKI [20] 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 [9] 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 comple-

Matrix	Thread Mapping GFLOPS	Aligned Access GFLOPS	Runtime GFLOPS	Naive GFLOPS	Relative Gain	CUDPP GFLOPS	Relative Gain	Segmented Scan GFLOPS	Relative Gain
raefsky3	2.54	4.44	**6.60**	1.41	**4.86**	1.16	**5.68**	1.29	**5.11**
olafu	1.47	2.98	**3.67**	1.38	**2.65**	1.07	**3.42**	1.11	**3.30**
bcsstk35	1.49	**2.80**	2.76	1.34	**2.08**	1.12	**2.5**	1.14	**2.45**
venkat01	1.78	**3.36**	2.04	1.55	**2.16**	1.23	**2.73**	1.18	**2.84**
crystk02	1.47	**3.04**	2.85	1.28	**2.37**	1.03	**2.95**	1.13	**2.69**
crystk03	1.47	**3.03**	2.88	1.51	**2.0**	1.10	**2.75**	1.18	**2.56**
nasasrb	1.44	3.15	**3.16**	1.44	**2.19**	1.17	**2.7**	1.17	**2.7**
3dtube	1.42	2.82	**3.01**	0.86	**3.5**	1.15	**2.61**	1.17	**2.57**
ct20stif	1.49	**2.82**	2.26	1.25	**2.25**	1.13	**2.49**	1.16	**2.43**
bai	1.52	**2.96**	1.45	1.49	**1.98**	1.14	**2.59**	1.12	**2.64**
raefsky4	1.45	**2.94**	3.17	1.38	**2.13**	1.07	**2.74**	1.13	**2.6**
ex11	1.51	**3.18**	1.36	1.25	**2.54**	1.08	**2.94**	1.19	**2.67**
rdist1	1.25	**2.09**	1.01	1.11	**1.88**	0.77	**2.71**	0.78	**2.69**
vavasis3	1.27	**1.84**	0.41	0.72	**2.55**	1.23	**1.49**	1.18	**1.55**
orani678	0.83	**1.15**	0.81	0.20	**5.75**	0.67	**1.71**	0.70	**1.64**
rim	1.52	**3.05**	1.44	1.29	**2.36**	1.17	**2.6**	1.19	**2.56**
vibrobox	1.36	**2.19**	1.72	1.12	**1.95**	1.04	**2.10**	1.24	**1.76**
rma10	1.54	**3.19**	1.62	1.29	**2.47**	1.14	**2.79**	1.21	**2.63**
lp_osa_60	1.29	**1.34**	1.28	0.79	**1.69**	1.15	**1.16**	1.21	**1.10**

Table 2. Detailed Performance Measures on GeForce 8800 GTX

Matrix	Thread Mapping GFLOPS	Aligned Access GFLOPS	Runtime GFLOPS	Naive GFLOPS	Relative Gain	CUDPP GFLOPS	Relative Gain	Segmented Scan GFLOPS	Relative Gain
raefsky3	8.18	9.66	**10.9**	1.55	**7.03**	1.21	**9.0**	3.24	**3.47**
olafu	6.71	7.16	**7.26**	1.39	**5.22**	1.07	**6.78**	2.70	**2.68**
bcsstk35	**6.27**	6.22	6.26	1.27	**4.93**	0.98	**6.39**	2.78	**2.25**
venkat01	8.37	8.17	**8.64**	1.52	**5.68**	1.04	**8.30**	3.00	**2.88**
crystk02	6.77	6.29	**6.86**	1.50	**4.57**	0.89	**7.07**	2.57	**2.67**
crystk03	6.93	**7.33**	7.02	1.52	**4.82**	1.03	**7.11**	2.84	**2.58**
nasasrb	**7.26**	7.15	7.21	1.48	**4.90**	1.04	**6.98**	2.82	**2.57**
3dtube	5.48	5.93	**6.18**	0.82	**7.53**	1.10	**5.61**	2.96	**2.08**
ct20stif	5.86	**6.28**	4.51	1.39	**4.51**	1.07	**5.86**	2.85	**2.20**
bai	**6.12**	5.44	4.08	1.59	**3.84**	0.84	**7.28**	2.42	**2.52**
raefsky4	5.26	6.64	**6.98**	1.41	**4.95**	1.01	**6.91**	2.71	**2.57**
ex11	6.37	**6.96**	4.46	1.40	**4.97**	1.02	**6.82**	2.34	**2.97**
rdist1	**3.70**	3.63	2.45	1.44	**2.56**	0.66	**5.60**	2.11	**1.75**
vavasis3	2.48	**2.71**	2.41	0.64	**4.23**	1.01	**2.68**	2.62	**1.03**
orani678	1.73	**1.85**	1.31	0.18	**10.27**	0.62	**2.98**	1.14	**1.62**
rim	6.32	**6.48**	5.93	1.06	**6.11**	1.04	**6.23**	2.89	**2.24**
vibrobox	3.17	**3.19**	2.90	1.22	**2.61**	1.02	**3.12**	1.55	**2.05**
rma10	6.64	**7.08**	4.64	1.42	**4.98**	1.21	**5.85**	3.15	**2.25**
lp_osa_60	2.22	2.30	**2.33**	1.09	**2.13**	1.08	**2.14**	2.00	**1.16**

Table 3. Detailed Performance Measures on GeForce GTX 280

tion 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. As an example, consider the CSR format in SpMV. The non-zero elements are stored in a single array and when accessed in row-major order, there is spatial locality among the non-zero elements when successive elements are used for computation by the same thread. However in GPUs, such an access results in non-coalesced hardware accesses and it is preferable to make successive threads access successive elements, as it would be enable hardware access coalescing (explained in Section 4.1).

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 [21]. The BCSR [8] and UBCSR [21] 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. [11] 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. We propose a block storage format that enables fine-grained thread-level parallelism, optimal global memory access, and date reuse at the level of shared memory, instead of registers.

There has been several works that perform a runtime processing to reorder computation and data for locality enhancement for cache-based architectures (e.g. [13]). Strout et al. [18] 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, but perform only a simple processing to determine non-zero blocks of fixed block size that is aligned as per the GPU architectural constraints.

NVIDIA's CUDPP [4] library for data-parallel algorithm primitives and the implementation of optimized scan primitives by Dotsenko et al. [6] are the most prominent relevant works on sparse matrix computations on NVIDIA GPUs. We have discussed these works in detail in Section 5.

7. Conclusions and Future Work

In this work, we have presented the key architectural optimizations that have to be addressed in GPUs for high performance 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. We have developed techniques, that involve *compile-time* and *run-time* optimizations, to build a high performance SpMV kernel for efficient execution on GPUs. We have proposed a new blocked storage format for storing and accessing elements of a sparse matrix in an optimized manner from the GPU memories. 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 (factor of 2 to 4) 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 framework 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 framework to determine optimal block sizes for arbitrary irregular sparse matrices.

References

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

[2] 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.

[3] G. E. Blelloch. Prefix sums and their applications. Technical report, 1990.

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

[5] T. Davis. The university of florida sparse matrix collection. *ACM Trans. on Mathematical Software*. http://www.cise.ufl.edu/research/sparse/matrices.

[6] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli. Fast scan algorithms on graphics processors. In *ICS '08: Proceedings of the 22nd annual International Conference on Supercomputing*, pages 205–213, 2008.

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

[8] 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.

[9] 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.

[10] P. M. W. Knijenburg and H. A. G. Wijshoff. On improving data locality in sparse matrix computations. In *Technical Report 94-15, Department of Computer Science, Leiden Univ.*, 1994.

[11] Luc Buatois and Guillaume Caumon and Bruno Lvy. Concurrent Number Cruncher: An Efficient Sparse Linear Solver on the GPU. In *High Performance Computation Conference (HPCC), Springer Lecture Notes in Computer Sciences*, 2007.

[12] 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.

[13] 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.

[14] 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.

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

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

[17] 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.

[18] M. M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, 2003.

[19] 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.

[20] 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.

[21] R. Vuduc and H.-J. Moon. Fast sparse matrix vector multiplication by exploiting variable block structure. In *Proceedings of the International Conference on High-Performance Computing and Communications*, LNCS 3726, Sorrento, Italy, September 2005.

[22] R. W. Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, University of California, Berkeley, December 2003.

[23] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, 2007.