Finite-Element Sparse Matrix Vector Multiplication on Graphic Processing Units

Maryam Mehri Dehnavi, David M. Fernández, and Dennis Giannacopoulos

Department of Electrical and Computer Engineering, McGill University, Montreal, QC H3A2A7, Canada

A wide class of finite-element (FE) electromagnetic applications requires computing very large sparse matrix vector multiplications (SMVM). Due to the sparsity pattern and size of the matrices, solvers can run relatively slowly. The rapid evolution of graphic processing units (GPUs) in performance, architecture, and programmability make them very attractive platforms for accelerating computationally intensive kernels such as SMVM. This work presents a new algorithm to accelerate the performance of the SMVM kernel on graphic processing units.

***Index Terms*—Computer architecture, graphic processing units (GPUs), parallel processing, sparse matrix vector multiplication (SMVM).**

I. Introduction

THE performance of finite-element (FE) electromagnetic applications can be dominated by the iterative solvers used, such as conjugate gradient (CG) based methods. As problems become larger and more complex, the computation overhead of these kernels dramatically increases the execution time of such solvers on single-core CPUs. Thus, the development of efficient methods to improve the performance of iterative solvers on parallel processors is almost inevitable.

One of the most important kernels in iterative solvers such as the CG method is the sparse matrix vector multiplication (SMVM). This operation is performed at each iteration and often consumes a majority of the computation time. The main objective of the SMVM kernel is to calculate Ax, where A is a sparse matrix and x is a dense vector. Major limitations of SMVM computation involving FE matrices are large memory storage and bandwidth requirements as well as indirect and irregular memory accesses.

Graphic processing units (GPUs) have recently evolved into very attractive commodity data-parallel coprocessors. Easy-to-learn programming interfaces such as CUDA [1] have allowed massive multithreading and increased utilization of large numbers of cores on the GPU, making them cost-efficient highly parallel platforms to solve computationally intensive scientific problems [2].

The main objective of this work is to accelerate the performance of finite-element SMVM kernels on the NVIDIA GT8800 graphic cards using a new algorithm, namely Prefetch-Compressed Row Storage (PCSR).

II. GPU Architecture

Modern GPUs are massively parallel and conform to single instruction multiple data (SIMD) architectures. Several levels of parallelism are offered by GPUs through multiple pipelines and vector processing. GPU architectures such as AMD-ATI X1k series process data in parallel using vector processors, while others such as NVIDIA G80 use multiple pipelines to perform parallel operations. With the ability to launch thousands of threads in parallel and processing trillions of operations in seconds, NVIDIA GPUs are among the best for general-purpose programming [1], [2].

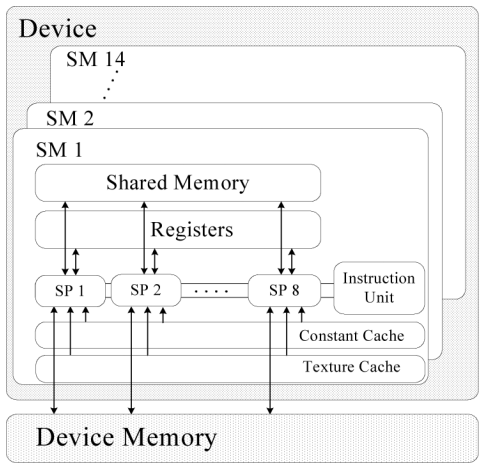

Fig. 1. GT8800 underlying architecture.

The NVIDIA GT 8800 graphic card (Fig. 1) consists of 14 *streaming multiprocessors* (SMs), each containing eight *scalar processors* (SPs), or processor cores running at 1.5 GHz. Each of the SMs accesses a separate 16 KB shared memory and a total of 8192 registers. The 14 SMs are connected via 512 MB of off-chip device memory.

Using the CUDA programming model, the GPU is viewed as a compute device capable of executing a large number of threads in parallel. While the main core of the code is run on the CPU, parts of the applications that exhibit rich data parallelism are implemented as kernel functions on the device (GPU). Data required by the kernel is transferred to the GPU global memory, and the parallel portion of the application is then executed on the device using many different threads. The programmer divides the threads into threads blocks that are distributed among the SMs allowing each multiprocessor to run a maximum of eight blocks. Thread blocks allocated to one SM communicate via fast shared memory, but blocks from different SMs can only communicate through global memory with a memory access latency of up to 600 cycles. Every 32 threads in a block execute the same instruction and are called a warp. When threads in the same warp follow different paths of control flow, we say that these threads

Manuscript received December 21, 2009; accepted February 07, 2010. Current version published July 21, 2010. Corresponding author: M. Mehri (e-mail: maryam.mehridehnavi@mail.mcgill.ca).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TMAG.2010.2043511

```
CSR Algorithm
 for i=1 to number of rows
  Y[i]=0
   for j=PTR[i] to PTR[i+1]
     Y[i]=Y[i] +VAL[j] * X [INDX[j]]
   end for
 end for
```

Fig. 2. CSR algorithm.

diverge in their execution. Thread divergence forces the threads in a warp to execute sequentially, thus reducing the execution speed of the application and should be avoided [1].

III. Sparse Matrix Vector Multiplication (SMVM)

The SMVM kernel is one of the most popular kernels in solving sparse linear systems for large and complex finite element simulations. A variety of sparse matrix representations exist, each having a distinct form of data storage and access, manipulation of matrix entries, and calculation of the matrix vector multiplication product. Compressed Sparse Row (CSR) is one of the most commonly used data structures for SMVM solvers. The nonzero elements of the sparse matrix in this format are stored in a value vector (VAL), while the corresponding index values are held in another vector (INDX). The format also uses a pointer array (PTR), which points to the first entry of each row in VAL and INDX [3]. The sparse vector matrix product in this format is calculated using two nested loop iterations (Fig. 2).

IV. PCSR (Prefetch-Compressed Row Storage)

Many challenges exist in optimizing the performance of scientific applications such as the SMVM kernel on GPU platforms. Some are as follows: global memory access latency, limited shared memory, thread synchronizations, thread divergence, inadequate number of threads, and limited global memory bandwidth. The way the programmer addresses these issues differs depending on the application [1].

A new SMVM algorithm, namely PCSR, is proposed in this section. By combining CSR with a novel partitioning scheme and computation strategy, the execution time of the SMVM kernel is accelerated on the NVIIDA GPUs. To clarify the major advantages of our method, a survey of previous work on SMVM kernel optimization techniques for the GPU are first presented, and the details of the new implementation are then described.

A. Previous Work

Since the release of CUDA in 2007, few works have investigated the SMVM kernel optimization on the GPUs. Buatois *et al.* [4] investigated the performance of Blocked-CSR on the G80 series of NVIDIA graphic cards. To increase the performance of their method, the matrix filling ratio is decreased, adding extra nonzeros to the value vector and increasing the number of memory transactions. Sengupta *et al.* [5] proposed the use of segmented scan for calculating SMVM on GPUs. Wiggers *et al.* [6] reorders matrix rows to increase parallelism in the SMVM kernel and reduce thread divergence when a row is calculated by a single thread. Sorting matrix rows increases processing overhead considerably increasing the execution time on the host. Comparing the performance of various SMVM representations

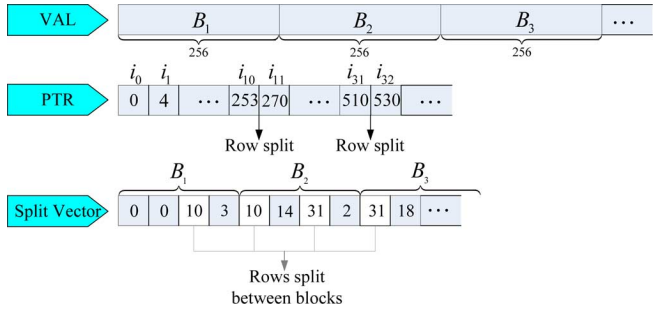

Fig. 3. PCSR partitioning scheme (e.g., row 10 is partitioned between blocks 1 and 2 (B1 and B2); the Split Vector shows that three elements of row 10 are stored in B1 and 14 in B2).

on the GPU, Bell *et al.* [9] proposed a new method to optimize the CSR format on the GPU. To decrease thread divergence, instead of calculating each row by a single thread, all threads on a single warp are responsible for computations of one row. Matrices with average nonzeros less than 32 per row do not benefit from their proposed technique, and since every element is fetched from the global memory separately and only when their value is required, a majority of memory fetches are uncoalesced when run on the GT8800.

Previous results were implemented on various versions of NVIDIA GPUs, each with a different memory bandwidth and processing power. To compare our method to other work, we applied the row-per-thread and row-per-warp methods using the code in [9] on our GPU and present comparison results. Our proposed algorithm introduces new techniques to hide global memory access latency via data perfecting and memory coalescing. The technique also regularizes the data access pattern on the GPU by proper partitioning and padding the matrix with zeros. Detailed description of the method and its major contributions are given in the proceeding sections.

B. PCSR Algorithm

Details of the partitioning scheme and padding method used in PCSR are proposed in this section. Methods of efficiently accessing the x vector and the algorithm steps are also presented.

- *Partitioning scheme*
  To obtain a reasonable execution time on the GPU, global memory accesses should be minimized by transferring data on to shared memory. Due to the limited storage of shared memory, vectors require to be partitioned and transferred in small segments. Different row sizes in small matrices complicate the partitioning of the vectors. We propose an efficient partitioning method that benefits from the inherent parallelism on the GPU. To maximize resource usage on an SM, 768 threads should run simultaneously on its architecture. Therefore, if three blocks are active per SM, 256 threads should be executed via one block to maximize performance. The value and index vectors in the CSR representation should also be divided into blocks of 256 elements (vectors are padded with zeros to be divisible to 256). Searching through the row pointer vector, rows split between the blocks are found, and their id as well as their spreading pattern between two blocks is stored in a new vector called the Split Vector (Fig. 3). For matrices with more than 256 average number of nonzeros per row, the

split vector will store only the id of blocks holding elements of more than one row to keep the size and transfer time of the split vector to GPU memory negligible compared to the total data transfer time.

Simultaneous loading of data from global memory to shared memory, coalesced memory accesses, and reduced memory transfer time are the major benefits of partitioning. Partitioning the vectors and loading them from global memory at the beginning of the kernel, will also reduce the effects of thread divergence. Divergent threads in the computation section of the kernel will fetch their required data from on-chip shared memory, avoiding the serialization of global memory accesses.

- *Zero padding*
  Minimizing thread divergence on GPUs is essential for achieving good performance. If each thread calculates one row, the diversity in row sizes will cause thread divergence and threads will execute sequentially. Assigning a warp to each row [9] will also cause thread divergence since the number of nonzeros per row are not necessarily multiples of 32. Since the execution is serialized in divergent threads, we reduce the number of operations per thread by padding. Padding each row to be a multiple of the padding factor (n) will allow the kernel to reduce the product vector using parallel reduction. Every n value in the product vector can be added via parallel reduction and stored in another vector called *sum*. Because of the padding, in the reduction procedure, threads will not add values of more than one row. The number of elements corresponding to a row in the sum array is less than the product vector. Thus, to calculate the results of each row, a thread will only add the elements in the *sum* vector corresponding to that row, reducing the number of operations executing sequentially (although increasing the padding factor is beneficial in reducing thread divergence, larger n decreases the vector filling ratio and increases the value vector size).
- *Texture memory*
  The x vector cannot be divided between blocks due to the irregular indirect access to its elements in the SMVM kernel. Accessing the global memory for every index increases memory latencies. To avoid such accesses, the x vector is loaded on to texture memory, and its elements are spread on the shared memory of each block simultaneously. The texture memory is an on-chip cached memory space, thus a texture fetch costs one memory read from global memory only on a cache miss; otherwise, it just costs one read from the texture cache. Loading the x vector to texture memory decreases global memory access latencies and enhances the performance of the SMVM kernel.
  In the proposed technique, threads in a block simultaneously load 256 elements of the x vector corresponding to the index vector values on to shared memory. The technique enables simultaneous spreading of the x vector on the GPU with minimum memory access latency and also minimizes the effects of thread divergence throughout the kernel.
- *Algorithm steps*
  Fig. 4 shows the seven steps in the PCSR algorithm. Partitions of the index and value vector allocated to each block (256 elements) are first loaded into shared memory simultaneously to coalesce memory accesses and reduce memory transfer time. The x vector elements are then loaded from texture memory and spread in shared memory. The 256 elements allocated to each block are multiplied with the corresponding values of the x vector in parallel by the 256 threads in a block. After determining the index and split pattern of the rows in each block using the Split Vector, required elements of the PTR array are loaded into shared memory. Depending on the padding factor, the product vector is reduced in parallel to generate the sum vector values. Using the sum vector, the final value of each row is calculated by different threads with minimum thread divergence, and the results are written into the global memory simultaneously.

1: Load VAL and INDX vectors to shared memory
2: Load and spread the x vector
3: Calculate the product vector in parallel
4: Load PTR array values related to the block
5: Reduce the product vector via padding and store in sum
6: Calculate each row by one thread
7: Load results in to global memory

Fig. 4. Prefetch-CSR algorithm.

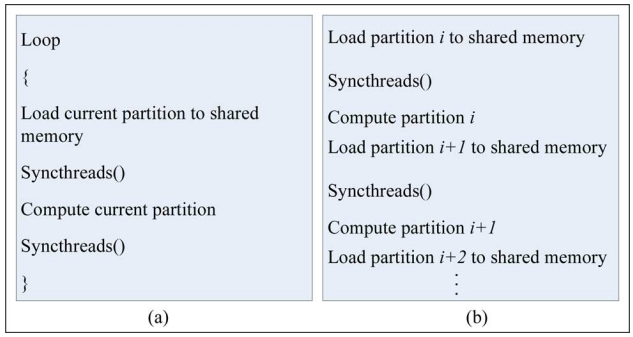

Fig. 5. Prefetching data in PCSR. (a) Without prefetching. (b) With prefetching.

C. Prefetching

The time required to load data from global memory is high due to the 300 cycle global memory access latency. Perfecting the required data for the next iteration in each thread block hides much of the global memory access delay. While many threads are waiting on global memory accesses, others process with the necessary calculations for the current data in shared memory. Details of the prefetching methods are shown in Fig. 5; the prefetching loop is also unrolled to maximize performance.

V. Results

We have investigated the performance of our technique on various sparse matrices from [7] with different average nonzeros per row (Table I). The performance of the algorithm is tested on GT8800 NVIDIA graphic cards using CUDA 2.3, and the execution speed of the kernel is represented in GFLOPS (billion floating operations per second). The SMVM kernel is a part of iterative solvers, thus data transfers between host and device memory occur at most twice (at the beginning and the end of iteration) and are neglected over a large number of SMVM operations [9].

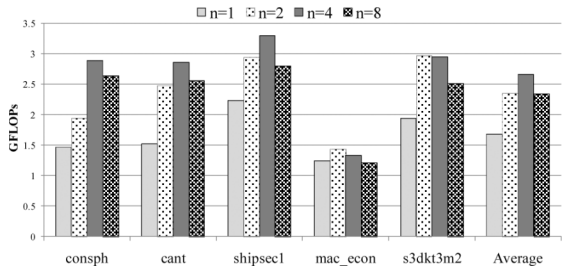

Fig. 6. Effect of the padding factor (n) in PCSR.

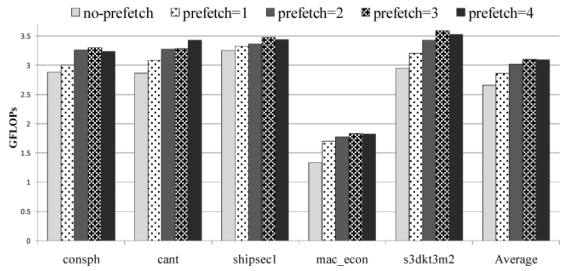

Fig. 7. Varying the number of prefetches in PCSR.

TABLE I
NONZEROS (NNZ) AND FILLING RATIO PERCENTAGE FOR DIFFERENT PADDING FACTORS (n) IN MATRICES

Matrix Name	consph	cant	shipsec	mac_econ	s3dkt3m2
nnz	6010480	4007383	7813404	1273389	3843910
nnz/row	72.1	64.1	55.4	21.24	6.1
n=2	98.8	99.2	98	93.8	97.8
n=4	96.4	98	97.3	80	97.7
n=8	92.2	93.3	96.1	44	68.69

In Fig. 6, the performance of the proposed technique has been shown. The execution time of the kernel is tested for padding factors of 1, 2, 4, and 8 (the filling ratio of the padded matrices are shown in Table I).

Padding the matrix rows to be multiples of four increases the performance to 60% compared to no padding (padding factor 1). For padding factors larger than four, the number of zeros added due to padding are increased, decreasing the filling ratio and the SMVM kernel performance. Setting the padding factor to its optimum value (four), Fig. 7 shows the effects of prefetching data to hide global memory latency. The results show an average 16% increase in performance if each block prefetches and operates on four partitions of 256 value vector elements (Section IV-B).

Because of the variety in the memory bandwidth and computation capabilities of different NVIDIA cards, comparisons with other work are done via running their methods on the GT8800. Fig. 8 and Table II provide a comparison of our method to the row-per-warp and row-per-thread methods on GT8800 [9]. The performance of PCSR is also compared to the execution of the SMVM kernel on a quad-core CPU and the Cell-PPE. The cell results were obtained using the Cell SDK 3.0 and the PMS method [8]. The CPU platform used was Intel core2 Quad 2.4 GHz architecture with 4 MB of L2 cache per core-pair and 4 GB of global DRAM. As shown in Table II, on average our algorithm outperforms the row-per-warp and row-per-thread techniques presented in previous work by 2.45 and 3.37 times respectively. Speedups of up to 18.8 times were achieved

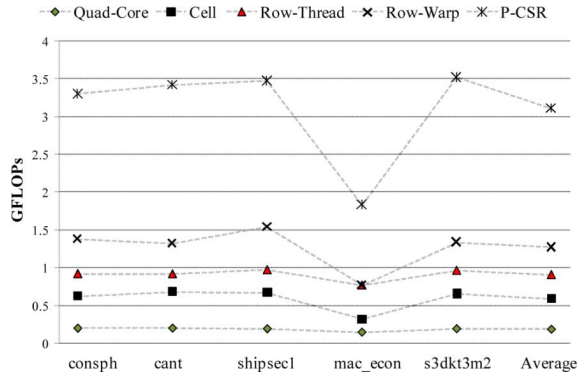

Fig. 8. PCSR performance compared to the row-per-thread and row-per-warp methods on GT8800 as well as the QUAD-Core CPU and Cell architectures.

TABLE II
SPEED UP OF PCSR COMPARED TO THE ROW-PER-THREAD AND ROW-PER-WARP METHODS ON GT 8800, THE CPU AND THE CELL

Matrix Name	consph	cant	shipsec1	mac_econ	s3dkt3m2	Average
Row thread	3.57	3.71	3.56	2.37	3.64	3.37
Row warp	2.39	2.60	2.26	2.38	2.64	2.45
Cell	5.27	5.04	5.18	5.77	5.41	5.34
CPU	17.03	17.52	18.7	13	18.8	17

compared to the quad-core CPU, and the execution time was less than what is achieved through optimized SMVM kernel on the Cell.

VI. CONCLUSION AND FUTURE WORK

We have introduced several efficient techniques to accelerate the execution of the sparse matrix vector multiplication (SMVM) on NVIDIA graphic processing units. The proposed methods increased the performance of the SMVM kernel on GT 8800 up to 18.8 times compared to the quad-core CPU and three times compared to previous work on accelerating SMVM for GPUs. Reducing the execution time of finite-element solvers such as the conjugate gradient method using the proposed optimizations will be investigated in future work.

REFERENCES

[1] NVIDIA CUDA [Online]. Available: http://developer.nvidia.com/object/cuda.html
[2] J. D. Owens *et al.*, "A survey of general-purpose computation on graphics hardware.," *Comput. Graphics Forum*, pp. 80–113, 2007.
[3] S. Yousef, *Iterative Methods for Sparse Linear Systems*. Philadelphia, PA: SIAM, 2003, p. 528.
[4] L. Buatois, G. Caumon, and B. Lévy, "Concurrent number cruncher: An efficient sparse linear solver on the GPU," in *High Performance Computing and Communications*. Berlin, Germany: Springer-Verlag, 2007, vol. 4782, Lecture Notes in Computer Science, pp. 358–371.
[5] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in *Proc. Graphics Hardw.*, 2007, pp. 97–106.
[6] W. A. Wiggers, V. Bakker, A. B. J. Kokkeler, A. B. J. , and G. J. M. Smit, "Implementing the conjugate gradient algorithm on multi-core systems," in *Proc. Int. Symp. System-on-Chip*, 2007, pp. 11–14.
[7] S. Williams *et al.*, "Optimization of sparse matrix vector multiplication on emerging multicore platforms," in *Proc. ACM/IEEE Conf. Supercomput.*, 2007, Article no. 38.
[8] D. Fernandez, D. Giannacopoulos, and W. Gross, "Efficient multicore sparse matrix-vector multiplication for FE electromagnetics," *IEEE Trans. Magn.*, vol. 45, no. 3, pp. 1392–1395, Mar. 2009.
[9] N. Bell and M. G. Fernandez, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Tech. Rep., 2008.