Communication-Avoiding Krylov Techniques on Graphic Processing Units

Maryam MehriDehnavi[1][2], Yousef El-Kurdi[1], James Demmel[2], and Dennis Giannacopoulos[1]

[1]ECE Department, McGill University, Montreal, QC H3A2A7 Canada
[2]EECS and Math Department, University of California Berkeley, CA 94720 USA

Communicating data within the graphic processing unit (GPU) memory system and between the CPU and GPU are major bottlenecks in accelerating Krylov solvers on GPUs. Communication-avoiding techniques reduce the communication cost of Krylov subspace methods by computing several vectors of a Krylov subspace "at once," using a kernel called "matrix powers." The matrix powers kernel is implemented on a recent generation of NVIDIA GPUs and speedups of up to 5.7 times are reported for the communication-avoiding matrix powers kernel compared to the standards prase matrix vector multiplication (SpMV) implementation.

Index Terms—**Graphic processors, Krylov solvers, numerical algorithms, parallel algorithms.**

I. INTRODUCTION

THE sparse matrix vector multiplication (SpMV) kernel is a dominant computing kernel in standard Krylov subspace methods (KSMs). Computing a few arithmetic operations per datum, SpMV operations are classified as communication-bound. The cost of communication (moving data between levels of the memory hierarchy) is considerably higher than the cost of arithmetic computations in modern architectures and this gap is expected to further widen. Thus, in order to enhance the performance of communication bound kernels such as SpMV, new strategies should be explored to minimize communication and data movement.

A. Communication-Avoiding Krylov Techniques

Communication-avoiding (CA) algorithms [1] communicate less than the state-of-the-art algorithms at the expense of more arithmetic operations. Standard implementations of SpMV in KSMs, require reloading the sparse matrix to caches and fast memory in each iteration when they are too large to fit in fast memory; thus, overwhelming the algorithm with communication and data movement between fast and slow memory. Communication-avoiding Krylov techniques [1] minimize communication via computing k steps of the iterative solver at the same time. To take k steps at the same time, and so potentially reduce memory traffic by a factor of k, a new sparse matrix kernel is required, called the matrix powers kernel. Where p_i is a polynomial of degree i, the matrix powers kernel computes the basis $[p_1(A)x, p_2(A)x, p_3(A)x, \ldots, p_k(A)x]$. To compute the aforementioned basis for a matrix A that does not fit into fast memory, the matrix is first divided into partitions (cache-blocks) that fit into the desired memory space. The partitions are then loaded into fast memory to compute the basis. To avoid communication between fast and slow memory and between partitions, nonlocal rows might also be copied to a partition ("remote/ghost" rows) leading to redundant arithmetic operations [2]. For a well partitioned A matrix (where A has a low surface-to-volume ratio), the communication cost of the k-step matrix powers kernel will be O(1) compared to $O(k)$ for k SpMV operations in a naïve implementation [2].

B. Graphic Processing Units

GPUs have become an important resource for scientific computing in recent years. With easy to learn APIs such as CUDA [3] introduced by NVIDIA, general purpose programming for modern scientific computations on GPUs have gained considerable attention. The GPU consists of streaming multiprocessors (SMs) and each SM contains basic processing units called scalar processors (SPs). To run compute intensive parts of an application on the GPU initial data has to be transferred from CPU memory to GPU global memory and a GPU kernel is then launched. Using a single data multiple thread paradigm, GPU threads grouped into thread blocks (TBs) proceed with the computations and transfer the results back to CPU. The GPU memory hierarchy consists of an on-board global memory with long access latency, a fast access shared memory, registers, and caches. Threads inside a block communicate via shared memory and their execution can be synchronized. Every 32 threads in a block execute the same instruction and are called a warp. Referred to as thread divergence, if threads inside a warp go through different computation paths their execution is serialized; to achieve higher speedups, thread divergence should be avoided while accelerating problems on GPUs.

II. PREVIOUS WORK

A brief survey of the k-step Krylov techniques is presented in this section; algorithmic details of these techniques and a complete survey of previous work on k-step Krylov solvers and available preconditioners can be found in [2]. The k-step Krylov subspace methods were initially introduced by Rosendale [4], and later studied in work such as [5], [6]. All this work used a monomial basis and reported convergence for $k < 5$ in k-step KSMs. By using a scaled monomial basis [7], a scaled and shifted Chebychev basis [8] and Newton basis [9], the coverage of the k-step Krylov subspace techniques were further improved at the expense of increased dependency in the algorithm. This problem is resolved by Hoemmen *et al.* [2] by eliminating the need for scaled basis vectors. Carson *et al.* proposed techniques to extend CA-Krylov techniques to 2-sided methods such as BiCGStab in [10] and repaired their numerical instability in [11]. A more detailed survey of available work on communication-avoiding KSMs is presented in

[2]. The dominant computing kernel in $k\text{-}step$ Krylov solvers is the matrix powers kernel which is accelerated on GPUs in this work.

A considerable number of work has been done on accelerating sparse matrix vector multiplication on GPUs [12]–[14]. None of the available implementations of SpMV on GPUs consider cache blocking for GPU global memory (device memory). If the matrix is larger than the device memory, computing k SpMVs requires reloading the matrix to the GPU for each SpMV kernel which is very costly. With only 1.5 GB of global memory in GPUs such as NVIDIA GTX480, matrices from many real problems cannot be fully stored on the device. Memory might also be allocated to store preconditioners and other data structures, leaving only a part of GPU global memory for storing A. As a result, the matrix has to be transferred to the GPU in each iteration, increasing data transfers between GPU and CPU memory in iterative solvers. The matrix powers kernel reduces data communication between CPU and GPU by partitioning the matrix and computing k SpMV operations at the same time for each partition.

In this work the matrix powers kernel is implemented on GPUs by cache blocking the matrices to fit on the GPU global memory. Our work is closely related to the work proposed by Mohiyuddin *et al.* [15], which studies the performance of the matrix powers kernel on an 8-core Intel Clovertown. The proposed implementation of the communication-avoiding matrix powers kernel on GPUs will be used in communication-avoiding KSMs in future work.

III. IMPLEMENTATION DETAILS

Implementation details of the matrix powers kernel on GPU global memory are presented in this section. The auto-tuning stage partitions the matrix to fit into GPU global memory; the partitions are then used in the matrix powers kernel.

A. Auto-Tuning Stage

The first stage of the algorithm is the partitioning stage where the matrix is either divided into equal partitions using a naïve partitioning strategy or graph and hyper-graph partitioners such as Metis [16] and Zoltan [17]. The results presented in this work are achieved via naïve row block partitioning; other partitioning methods will be studied in future work. The matrix is first divided into equal partitions of row blocks. The partitions are balanced based on the floating point operations required to compute k steps of the matrix powers for each row block and are recursively reduced to fit into GPU global memory (see Fig. 1). The size of each partition is equal to the memory required to store local and remote rows in compressed row storage (CSR) format for each partition.

B. Matrix Powers Kernel

Along with the corresponding elements of the source vector partitions generated by the auto-tuner are transferred to GPU global memory one after another. For each partition k steps of the matrix powers kernel are computed while it is in global memory (Fig. 2). Sparse matrix vector multiplications are computed in parallel on the GPU for each partition using the CUS-
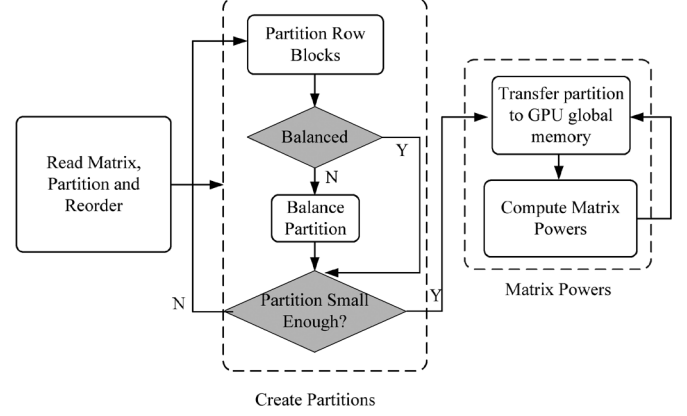

Fig. 1. Steps in the auto-tuner to generate cache blocks for global memory.

for each partition (cache block)
   transfer the partition to GPU Global memory
   for $i = 1$ to k do
      call a GPU kernel to compute $x_j^{(i)}$ (for all j belonging to the current partition)
      copy $x_j^{(i)}$ to the CPU (for all j belonging to the current partition)
   remove the current partition from global memory

Fig. 2. Matrix powers implementation on GPU global memory, x_j^i is the jth component of $x^i = A^i x^{(0)}$.

for $i = 1$ *to* k do
   for each partition (cache block)
   transfer the partition to GPU Global memory
   call a GPU kernel to compute $x_j^{(i)}$ (for all j belonging to the current partition)
   transfer $x_j^{(i)}$ to CPU (for all j belonging to the current partition)
   remove the current partition from global memory

Fig. 3. Standard computation of k SpMVs on the GPU, x_j^i is the jth component of $x^i = A^i x^{(0)}$.

PARSE SpMV kernel [14]. The generated vectors for each partition can then be used in the communication-avoiding Krylov solvers.

Performance results for the matrix powers kernel are tested on the NVIDIA GTX480. The GTX480 graphic card contains 480 CUDA cores and operates at 1.4 GHz, the size of global memory is 1.5 GB with a bandwidth of 177 GB/s. The shared memory is configured to 48 KB. All speedups are calculated using the following formula:

$$\frac{\text{time(matrix powers kernel for } (Ax, A^2x, \ldots, A^k x))}{\text{time}(k \; SpMV \text{ strandard operations})}. \quad (1)$$

The k SpMV standard operations in (1) are computed using the implementation in Fig. 3. Similar to the matrix powers kernel implementation, the SpMV operations in the standard

Fig. 4.   Each matrix is described by its name, description, number of rows, number of nonzeros, average number of nonzeros per row and its nonzero pattern. The matrices are stored in compressed row storage format.

Fig. 5.   Performance of the matrix powers kernel cache blocking for global memory on NVIDIA GTX480. The "AkX" indicates the best performanc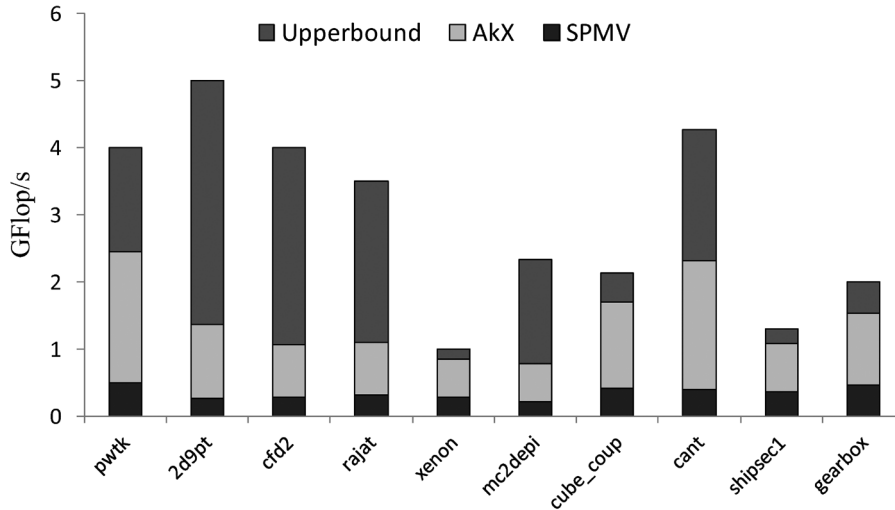e obtained for all $k < 40$. The label "upper bound" shows the performance achievable via scaling the standard k SpMV.

(also referred to as naïve) algorithm are accelerated on the GPU using the CUSPARSE SpMV kernel.

IV. RESULTS

In this section, the performance of the proposed implementation of the matrix powers kernel on GPU global memory is studied using ten matrices (Fig. 4) from the University of Florida matrix repository [18]. All matrices are cache blocked assuming only one fourth of the matrix can be stored in global memory at one time.

Fig. 5 shows the performance of the matrix powers kernel for global memory cache blocking (the best performance obtained for all $k < 40$). Speedups of up to 5.7 and 4.98 are achieved for well structured matrices, such as 'Cant' and "2d-9pt." The naïve SpMV performance is lower for matrices with smaller numbers of nonzeros per row such as "2d9pt" and "mc2depi."

The CUSPARSE SpMV implementation performs poorly for such problems due to an increase in thread divergence. The extra flops performed in the matrix powers kernel (for the best k) compared to k steps of the standard SpMV is shown in Table I. For an unstructured matrix such as "Xenon" that achieves the least speedup from the matrix powers kernel, in only 5 steps of the matrix powers kernel up to 23% more flops are computed (see Table I). The upper bound in Fig. 5 is computed for the best performing k using

$$\frac{(\text{arithmetic\_intensity (matrix powers)})}{(\text{arithmetic\_intensity}(SpMV)).\text{performance}(SpMV)}$$

where the arithmetic intensity is the effective flops to bytes transferred ratio [15]. The generated x^i vectors (where $x^i = A^i x^{(0)}$) are transferred to the CPU for both the naïve SpMV and matrix powers kernels at each step. The aforementioned transfers are also included in computing the upperbound. Table I

TABLE I
THE BEST SPEEDUP OF THE MATRIX POWERS KERNEL COMPARED TO NAÏVE SpMV, FRACTION OF TOTAL TIME SPENT IN COMMUNICATING DATA IN THE NAÏVE
SpMV IMPLEMENTATION AND EXTRA COMPUTED FLOPS IN THE MATRIX POWERS KERNEL PERFORMING k STEPS

Matrix	pwtk	2d9pt	cfd2	rajat	xenon	mc2depi	Cube coup	cant	shipsec1	gearbox
k	15	34	7	15	5	11	8	14	6	7
Speedup	4.92	4.98	3.79	3.49	2.85	3.53	3.98	5.7	2.88	3.21
Communication vs. Total time	91%	84%	90%	87%	87%	78%	88%	93%	93%	96%
AkXflops/naïveflops	1.3	1.1	1.2	1.03	1.23	1.02	1.22	1.16	1.24	1.26

shows the fraction of total time spent in communicating data between GPU and CPU memory for all the tested problems (for the best performing k). The table shows on average 90 percent of the SpMV kernel execution time is spent in transferring data between CPU and GPU global memory which further justifies the importance of avoiding communications using the communication-avoiding matrix powers kernel. For matrices such as "2d-9pt" and "mc2depi," which have the least number of nonzeros per row, a smaller percentage of total time is spent in communicating data. Also, compared to other matrices, the performance gap between the matrix powers kernel and the upperbound is larger for the aforementioned matrices. This is because the time spent in computing operations such as spreading the initial and source vectors at each step of the matrix powers kernel are no longer negligible for these problems. Increased thread divergence on the GPU for matrices with fewer nonzeros per row also increases the execution time of arithmetic computations for "2d9pt" and "mc2depi." As shown in Table I for some matrices the best speedup for the matrix powers kernel is achieved for k parameters as high as 34 and 15, which indicates the importance of using better polynomial bases and residual replacement to achieve both stability and convergence in CA Krylov techniques[11].

V. CONCLUSION AND FUTURE WORK

The matrix powers kernel in communication-avoiding Krylov techniques is accelerated and speedups of up to 5.7 are obtained for global memory cache blocking compared to the standard implementation of k SpMV operations; in future work, we intend to enhance the performance of this kernel by implementing other matrix partitioning schemes and enhancing the auto-tuning phase. The performance of the matrix powers kernel in Krylov subspace methods will be studied and preconditioners such as the sparse approximate inverse [19] will be used to enhance the convergence of communication-avoiding KSMs.

REFERENCES

[1] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick, Avoiding Communication in Computing Krylov Subspaces Univ. California Berkeley EECS, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2007-123, 2007.

[2] M. Hoemmen, "Communication-Avoiding Krylov Subspace Methods," Thesis, Dept. Comput. Sci., UC Berkeley, , 2010.

[3] NVIDIA CUDA [Online]. Available: http://developer.nvidia.com/object/cuda.html

[4] J. V. Rosendale, *Minimizing Inner Product Data Dependencies in Conjugate Gradient Iteration.* Silver Spring, MD, USA: IEEE Computer Society Press, 1983.

[5] A. Chronopoulos and C. W. Gear, "S-step iterative methods for symmetric linear systems," *J. Comput. Appl. Math.*, vol. 25, no. 2, pp. 153–156, 1989.

[6] H. F. Walker, "Implementation of the GMRES method using Householder transformations," *SIAM J. Sci. Statist. Comput.*, pp. 9–152, 1988.

[7] A. C. Hindmarsh and H. F. Walker, Note on a Householder Implementation of the GMRES Method Lawrence Livermore National Laboratory, USA, Tech. Rep. UCID-20899, 1986.

[8] W. D. Joubert and G. F. Carey, "Parallelizable restarted iterative methods for nonsymmetric linear systems," *Part I: Theory. Int. J. Comput. Math.*, vol. 44, no. 1, pp. 243–267, 1992.

[9] Z. Bai, D. Hu, and L. Reichel, "A Newton basis GMRES implementation," *IMA J. Numer. Anal.*, vol. 14, no. 4, pp. 563–581, 1994.

[10] E. Carson, N. Knight, and J. Demmel, Avoiding Communication in Two-Sided Krylov Subspace Methods U.C. Berkeley, Berkeley, CA, USA, Tech. Rep. EECS-2011-93, 2011.

[11] E. Carson and J. Demmel, A Residual Replacement Strategy for Improving the Maximum Attainable Accuracy of s-step Krylov Subspace Methods U.C. Berkeley, Berkeley, CA, USA, Tech. Rep. EECS-2012-197, 2012.

[12] N. Bell and M. Garland, Efficient Sparse Matrix-Vector Multiplication on CUDA NVIDIA Tech. Rep., 2008.

[13] M. Mehri Dehnavi, D. Fernandez, and D. Giannacopoulos, "Finite element sparse matrix vector multiplication on GPUs," *IEEE Trans. Magn.*, vol. 46, no. 8, pp. 2982–2985, Aug. 2010.

[14] NVIDIA CUSPARSE Library [Online]. Available: http://developer.download.nvidia.com/compute/cuda/40rc2/toolkit/docs/CUSPARSE_Library.pdf

[15] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick, "Minimizing.communication in sparse matrix solvers," presented at the ACM/IEEE Conf. Supercomput., New York, NY, USA, Nov 2009.

[16] G. Karypis and V. Kumar, "METIS 3.0: Unstructured graph partitioning and sparse matrix ordering system Dept. Computer Sci., Univ. Minnesota, Minneapolis, MN, USA, Tech. Rep. 97-061, 1997.

[17] K. D. Devine, E. G. Boman, and R. HeaphyR. H. Bisseling and U.K. Catalyurek, "Parallel hypergraph partitioning for scientific computing," in *Proc. of IPDPS'06*, Rhodos, Greece, 2006.

[18] T. A. Davis and Y. Hu, "The university of Florida sparse matrix collection," *ACM Trans. Math. Software* Jan. 2009 [Online]. Available: http://www.cise.ufl.edu/research/sparse/matrices, (to appear)

[19] M. Mehri Dehnavi, D. Fernandez, J. Gaudiot, and D. Giannacopoulos, "Parallel sparse approximate inverse preconditioners on graphic processing units," *IEEE Trans. Parallel Distrib. Syst.*, 2012, doi:10.1109/TPDS.2012.286, to be published.