Volume 3, Issue 10, October 2013 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering

Research Paper

Available online at: www.ijarcsse.com

Analysis of Sparse Matrix Data Formats and Locality on Graphical Processing Unit (GPU) Performance

Richard Haney, Ram Mohan*

North Carolina A&T State University

USA

Abstract— The Graphical Processing Unit is gaining popularity as a powerful commodity processor for computational modelling analysis in many engineering and scientific applications. Majority of scientific and engineering computational analysis codes are based on techniques such as the finite element method that results in large memory bound sparse matrix linear equation systems. The associated sparse matrices are stored in different sparse matrix compression formats in computational analysis software code developments. Solution of such large sparse matrix linear system of equations are integral and time consuming part of the computations which are solved via iterative solution approaches such as the pre-conditioned conjugate gradient method. This paper evaluates the effect of sparse matrix compression formats and the associated data locality on the performance of preconditioned conjugate gradient solvers in a Graphical Processing Unit (GPU) computing system. Computational performance variations and inferences between GPU and CPU platforms are discussed in the context of a finite element based engineering application software analysis code for composite process flow modelling.

Keywords— GPU Computing, sparse matrix formats, linear equation systems, performance, preconditioned conjugate gradient method

I. INTRODUCTION

Modern Graphical Processing Units (GPU) has been very useful in accelerating and improving the performance of many computationally intensive scientific and engineering applications. The exceptional computational power, improved and accessible programming languages for programmability, and relatively low hardware cost have helped the paradigm of GPU as a high performance commodity co-processor [1-3]. The power of a graphics processor as a co-processor however comes at the cost of flexibility, as large interrupted data streams execute optimally in the GPU environment [4].

Computational methods such as finite element techniques with unstructured mesh geometries are commonly used in many engineering applications such as structural mechanics, fluid flow, etc., employing the associated partial differential equations in physics based mathematical models. Computational solution of such problems often results in a system of linear equations for the numerical solution of the principal problem variables represented by $\underline{\underline{A}}\,\underline{x} = \underline{b}$. Here $\underline{\underline{A}}, \underline{x}$ are matrix of size $N \; X \; N$ and vector of size N respectively. In many unstructured engineering analysis applications, such matrix $\underline{\underline{A}}$ are large and sparse with most of the matrix entries being zero. Full matrix storage of these sparse matrices requires significant memory of the computer systems. Memory foot print of these large, sparse matrices is decreased using sparse matrix storage formats such as compressed sparse row (CSR) format [5, 6] that are regularly used in such computational modelling analysis code developments. Solution of these linear systems of equations is generally performed using iterative methods such as the preconditioned conjugate gradient solvers. Matrix – Vector products are the prime components of a conjugate gradient solver. The CSR matrix format however results in a downside for GPU computational platforms. This is due to the indexing data structures used increasing non-contiguous memory accesses [3, 7, 8] made during matrix-vector operations of these preconditioned conjugate gradient linear equation system solvers. The effect of this non-contiguous memory access has negligible impact on CPU based systems, which is not the case for GPU. The present paper focuses on the understanding and analysis of the computational performance differences that can be expected and potential alternative strategies that can improve such performance in a GPU system. Results and findings based on the GPU and CPU implementations in a preconditioned conjugate gradient solver in the CPU and GPU code developments for composite process flow modelling engineering applications are discussed and presented.

The present paper is organized as follows. A brief background on a typical GPU and its evolution is briefly discussed in section II, as well as the details of the CPU and GPU hardware employed in the present work. The next section describes the compression formats that are used for large sparse matrix storage of the unstructured finite element analysis problems in the present study. Compressed Sparse Row (CSR) is commonly used in many engineering applications for the sparse matrix storage and the potential memory access irregularities that can be experienced are presented. An alternative sparse matrix storage format, Blocked CSR and its potential to mitigate the adverse effects of CSR are discussed. Computational performance analysis and comparisons of these two sparse matrix storage formats used within the preconditioned conjugate gradient linear system solver routine in the CPU and GPU versions of an in-house

composite process flow modelling analysis code is presented next. Performance analysis comparisons are based on the execution time of the computational modelling application in the CPU and GPU. Computational performance results for the same candidate problem and associated finite element models are employed to illustrate the differences and effect of regular irregular access patterns between the GPU and CPU, and few potential solutions are proposed. The present analysis and research findings clearly illustrate the correlation between the computational hardware, software strategies (sparse matrix storage differences) on the overall computational performance.

II. GRAPHICAL PROCESSING UNITS

The modern GPU consists a heavy concentration of transistors in the Arithmetic Logic Units (ALUs) and wide data bus [1, 9]. The memory architecture has remained relatively simple to facilitate quicker access to input data. Unlike the CPU [4, 8], GPU separates the configuration/instruction from the data to be operated on by the processor's large number of ALUs [1, 7, 10].

A. GPU Evolution

Gaming industry had been a key driving force for the improvement in the computing power of GPUs over the years [1, 7]. Images are rendered using matrix operations applied simultaneously to large sets of data at the same time. Graphics hardware receives the instruction/configuration prior to the influx of the data to be operated, a natural Single Instruction Data (SIMD) environment seen in many parallel systems [11, 12]. Newer and more detailed games required faster processing of matrix operations as well as a wider data bus to enable rapid execution. Transistors were concentrated for floating point operations to facilitate large number of matrix operations [1] inherent in graphical processing hardware. The CPU on the other hand needed to maintain flexibility [13]. Both CPU and GPU must deal with latency and do so in distinctly different ways based on their architecture. The CPU maintains complex memory hierarchy to execute multiple processes in quick succession to deal with latency [1, 4, 10, 13]. GPU on the other hand deals with latency by executing large data sets with the same configuration/instruction which is set up prior to any data processing with a wide data bus and transistors concentrated in the arithmetic region. Figure 1 illustrates the CPU and GPU transistor distribution. All these evolution resulted in a computational workhorse optimized for data throughput.

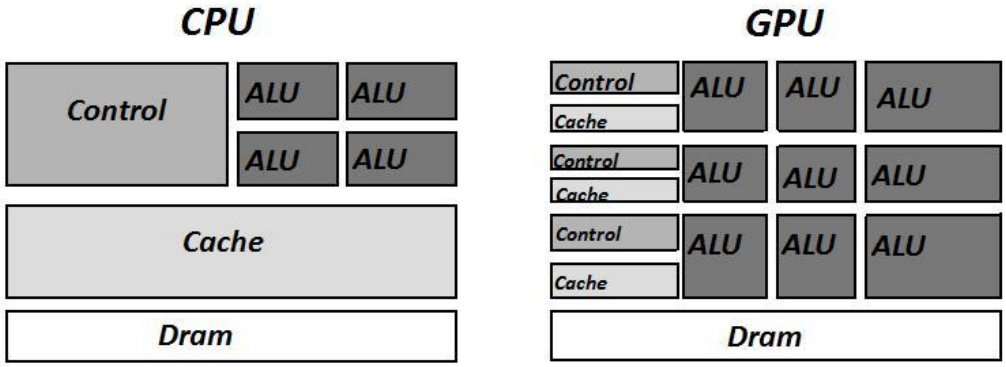

Fig. 1. Transistor distribution in CPU and GPU

Initially graphics hardware operated in a fixed pipeline and any programming was low level and very complex [1, 10]. Thus the ability to access the GPU in any realistic way to leverage its computational power was limited. The development of flexible pipeline as well as high level language constructs paved the way for the use of GPU as a co-processor [1, 7]. GPU software paradigms are further enabling their use as computational processors in computationally intensive applications such as the one employed in the present work.

B. GPU and CPU Architectures

In this work, we employ two commodity CPU and GPU processors in our investigations. CPU used in the present work is a 6-core Opteron processor 2431, with a clock frequency of 2.4 GHz, L2 and L3 cache system of 3 MB and 6 MB respectively. GPU used is a dual core Nvidia Quadro FX56000, with a frame buffer of 3GB (1.5GB/GPU), clock frequency of 400 MHz and uses Nvidia gigathread memory using GDDR3 SDRAM and a 384-bit wide bus.

III. SPARSE MATRIX FORMATS

The system of linear equations that arise from computational modelling discretization such as the finite element methods are large [14] that can become memory bound [15, 16] and are mostly sparse. Iterative solution of linear system of equations that arise from these problems via methods like preconditioned conjugate gradient involve matrix – vector multiplications. Due to the sparse nature of these matrices, storage formats that only store the non-zero entries of the matrix are commonly used in many computational applications based on discretization of governing equations through space discretization methods such as the finite element, finite difference methods.

A. Compressed Sparse Row Format

Compressed Sparse Row (CSR) is a common sparse matrix storage format in several finite element based code developments [16-19]. Besides lowering the required storage/memory space for large matrix systems, CSR can also enhance computational speed as only non-zero elements are involved in operations such as matrix-vector products. Smaller memory usage also could mean only the upper levels of CPU cache hierarchy with faster access are involved.

Figure 2 illustrates the data structure used for sparse matrix systems. The CSR format can however create spatial locality issues [20, 21] that can impact the computational performance of linear system of equations solvers such as the preconditioned conjugate gradient solver. The data structures used for indexing can increase the number of misses in the memory system as the number of non-contiguous elements are increased.

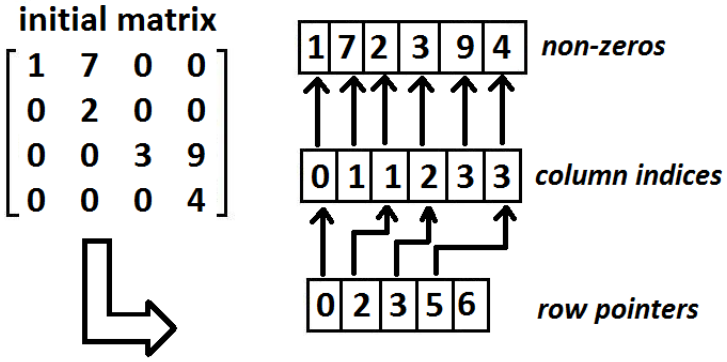

Fig. 2. CSR format structure

B. Blocked Compressed Sparse Row Format

This Blocked compressed sparse row (BCSR) storage format mitigates some of the irregular access of CSR. Block compressed storage format with a sub-block size of *2 X 2* and is represented as BCSR2X2 is used in the present work. BCSR2X2 alleviates the potential for spatial locality problem by increasing the number of non-zero elements from a single vector to a vector of sub-blocks [20-22]. This allows the GPU processor Cache to be loaded with elements that are near each other in the original matrix preserving some locality. Figure 3 illustrates the BCSR2X2 representation of a matrix.

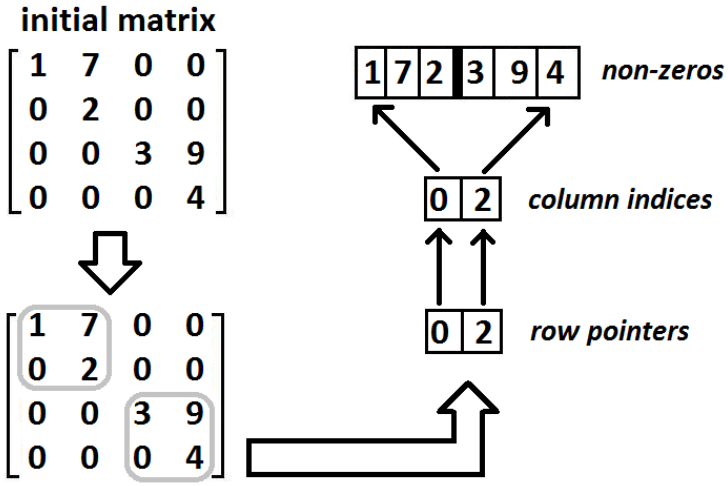

Fig. 3. BCSR2X2 format structure

The BCSR2X2 sparse matrix storage format could increase the data locality but has some potential issues. Additional computational time is added to iterate over a sub-block unlike the case of CSR with a single element. There exists increased indirection via the extra set of array pointers needed to access the new sub-blocks during the execution of matrix-vector multiplication. However, BCSR2X2 will increase spatial locality if dense sub-blocks exists within the sparse matrix being compressed. In the case of finite element discretization such as those experienced in composite process flow modelling that provided the test case sparse matrices in the present work, mesh renumbering provides sparse matrices of a smaller bandwidth with dense *2 X 2* sub-matrices.

IV. EFFECT OF MATRIX FORMATS AND DATA LOCALITY

It is clear from the above discussions that the sparse matrix formats influence the data locality and hence the performance of a GPU hardware. In the present work, we investigate the effect of these two compression formats for the same matrices that are formed during the transient mould filling analysis employing finite element discretization in a liquid composite moulding process. In particular, the performance of the iterative solver employed in this application problem based on preconditioned conjugate gradient technique is investigated. Further details of the composite process flow modelling code development and computational methods employed are found in [17]. A pseudo code of the preconditioned conjugate gradient method for the solution of linear system of equations $\underline{\underline{A}}\,\underline{x} = \underline{b}$ is presented in Algorithm 1.The linear system solver through preconditioned conjugate gradient method is the computationally intensive portion of the analysis code. As seen in Algorithm 1, preconditioned conjugate gradient solver algorithm involves several matrix-vector products whose performance is heavily influenced by the underlying matrix structure, data structure format and locality.

Algorithm 1: Preconditioned conjugate gradient (solves $Ax = b$)
Input: Matrix A and load/force vector b
Output: solution vector x

1. **Set** $r_0 \Leftarrow b - Ax_0$
2. **Set** $z_0 \Leftarrow M^{-1} r_0$
3. **Set** $p_0 \Leftarrow z_0$
4. **Set** $k \Leftarrow 0$
5. **DO UNTIL CONVERGENCE**
6. $\alpha_k \Leftarrow \frac{r_k^T z_k}{p_k^T A p_k}$
7. $x_{k+1} \Leftarrow x_k + \alpha_k p_k$
8. $r_{k+1} \Leftarrow r_k - \alpha_k A p_k$
9. **IF** $\left(\|r_k - r_{k+1}\| \leq \xi\right)$ **BREAK**
10. $z_{k+1} \Leftarrow M^{-1} r_{k+1}$
11. $\beta_k \Leftarrow \frac{z_{k+1}^T r_{k+1}}{z_k^T r_k}$
12. $p_{k+1} \Leftarrow z_{k+1} + \beta_k p_k$
13. $k \Leftarrow k + 1$
14. **END DO**

Finite element discretization based flow modelling code described in [17] was migrated and implemented for execution in the GPU processor studied. The computationally intensive part of the code for the iterative solution of linear system of equations was modified for the GPU implementation with the concurrent number cruncher (CNC) [5, 23].

The CNC software is a third party implementation of the preconditioned conjugate gradient solver using CUDA CuBLAS library calls and custom definitions for sparse matrix-vector operations. The code used in this study loads computed data to matrix and vector classes defined in the CNC software. The compression formats, either CSR or BCSR, are applied via CNC functions and the result is ported to the local GPU for execution via defined kernels. Once completed, the solution vector is transferred back to the CPU and written back to the composite process flow modelling analysis main code for mass convergence testing, and further continuation of the physical problem analysis.

Two demonstration composite flow modelling physical problems consisting of a simple and complex geometry and mesh configuration resulting in the same sparse matrices for the solution of linear system were analysed. Identical material, injection and boundary conditions were employed for the analysis in the CPU and GPU hardware architectures and associated CPU and GPU code developments. Post-processing analysis of flow progression and resin infusion time indicated that both the CPU and GPU implementations produced same physically accurate numerical results that are well within the same numerical precision. Further analysis of the preconditioned conjugate gradient solver routine indicated that both CPU and GPU implementations consistently resulted in the same number of iterations for the same defined convergence tolerance level in both CPU and GPU analysis runs. As the same deterministic physical problem and matrix system is solved in both cases on two different computing hardwares, it should result in a consistent physical solution as well as the number of iterations for the iterative solver. Clearly, this demonstrates the computational effectiveness and accuracy of the GPU implementation and CNC.

Computational performance analysis based on the total execution time for all the calls to the iterative pre-conditioned conjugate gradient solver was conducted based on two model configurations consisting of a finite element mesh with 1344 nodes (Mesh A) and 26,936 nodes (Mesh B). These two finite element mesh configurations result in two different sized sparse matrix systems with different sparse structures based on the finite element node numbering. With the optimized node numbering employed during the mesh generation, sparse matrices of smaller bandwidth with 2X2 sub-matrix blocks for the case of triangular elements employed in the finite element model meshes Mesh A and B can be expected. The matrices formed and employed in the solution of linear system equations for these two problems were stored both in CSR and BCSR2X2 formats in CPU and GPU implementations. With the physical solution, number of iterations for the convergence tolerance being the same, the computational time and performance of the CPU and GPU implementation based on the pre-conditioned conjugate gradient solver routine were analysed. Computational performance comparisons based on the execution time of the several calls to the pre-conditioned conjugate gradient solver routine were performed based on the following.

1. Speed Up: This is defined as the ratio of the computational execution time for the CPU hardware to that of GPU hardware. Wall clock time for CPU and GPU hardware studied is compared. Computational time comparisons are based on the pre-conditioned conjugate gradient solver for the two model problems (Mesh A and Mesh B) employing the CSR and BCSR2X2 matrix formats for the sparse matrix systems.
2. Total execution time for preconditioned conjugate gradient solver routine during the solution of the same application problem on CPU and GPU

Table 1 compares the speed up for the two different compression formats. While the GPU consistently showed a higher speed up in both model mesh sizes, the performance was much higher in the case of larger model Mesh B with 26,936 nodes.

Table 1: Speed up performance comparisons for the two compression formats

	Model	CPU Time x 10^{-3} (sec)	GPU Time x 10^{-3} (sec)	Speedup Factor
CSR Format	Mesh A	7,880	1,706	4.63
	Mesh B	1,901,580	54,569	34.85
BCSR2x2 Format	Mesh A	12,680	1,747	7.26
	Mesh B	7,999,280	39,968	200.14

Algorithm 2: Blocked CSR 2x2
Input: Matrix A and load/force vector b
Output: solution vector x

1. **Set** $x\_base \Leftarrow 0$
2. **Set** $BLOCK\_SIZE \Leftarrow 2*2$
3. **Set** $BM \Leftarrow 2, BN \Leftarrow 2$
4. **Set** $sum[BM]$
5. **FOR** $I \Leftarrow 0$ **TO** $M-1$
6. **FOR** $i \Leftarrow 0$ **TO** $BM-1$
7. $sum[i] \Leftarrow 0$
8. **FOR** $JJ \Leftarrow rptr[I]$ **TO** $rptr[I+1]$
9. **Set** $ofs \Leftarrow 0$
10. $J \Leftarrow colidx[JJ]$
11. $a\_base \Leftarrow JJ * BLOCK\_SIZE$
12. $b\_base \Leftarrow J * BN$
13. **FOR** $di \Leftarrow 0$ **TO** BM
14. **FOR** $dj \Leftarrow 0$ **TO** BN
15. $sum[di] \Leftarrow sum[di] + A[a\_base + ofs] * b[b\_base + dj]$
16. $ofs \Leftarrow ofs + 1$
17. **END FOR**
18. **END FOR**
19. **FOR** $i \Leftarrow 0$ **TO** BM
20. $x[x\_base + i] \Leftarrow sum[i]$
21. $x\_base \Leftarrow x\_base + BM$
22. **END FOR**
23. **END FOR**

Clearly based on the speed up comparisons, the total execution time for the preconditioned conjugate gradient solver on the GPU is lower than that of CPU. Figure 4 and Figure 5 present the total execution time for the preconditioned conjugate gradient solver during the complete candidate model problem using Mesh B for CPU and GPU based analysis runs respectively. The total execution time for GPU with BCSR2X2 was lower than CSR while it was opposite in the case of CPU. The execution of the several matrix-vector products in the preconditioned conjugate gradient solver includes extra nested loops to access the sub-matrices (see Algorithm 2, lines 13 – 18) The CPU architecture studied is clearly worse with this additional computational load. In contrast, the reduction in computational time due to data locality of BCSR2X2 in the GPU studied contributes to the overall reduction in the execution time.

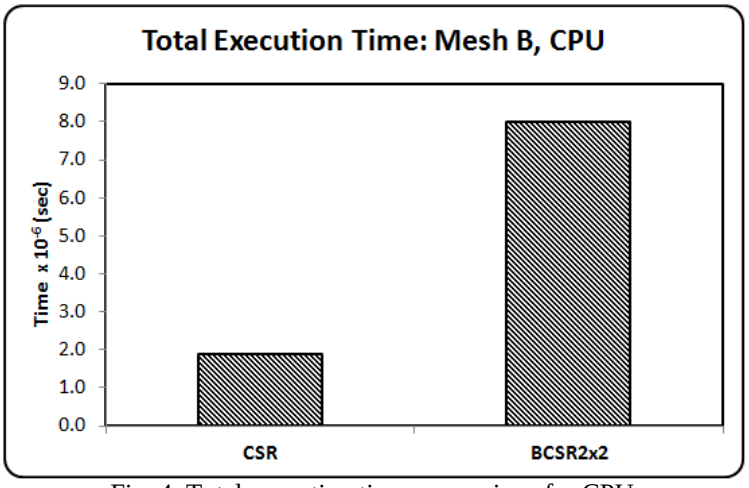

Fig. 4. Total execution time comparison for CPU

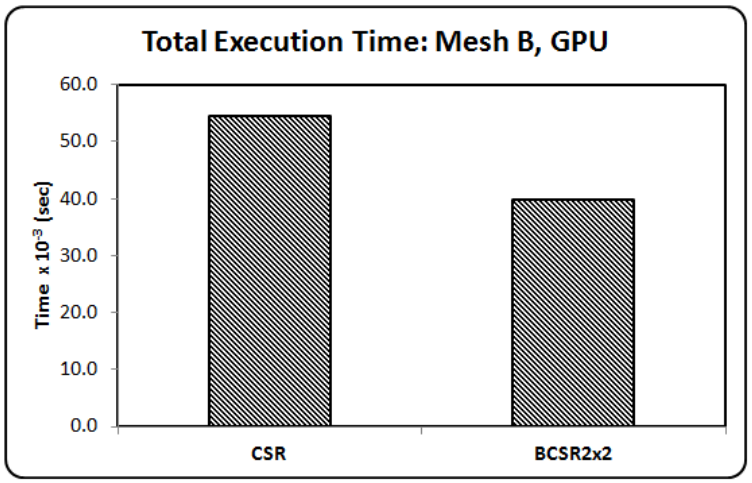

Fig. 5: Total execution time comparison for GPU

Table 2: KFLOPs comparison for mesh B

Compression	KFLOPs (GPU)	KFLOPs (CPU)
CSR	20.709	3.225
BCSR2x2	28.288	1.191

V. BENCHMARKING ON DIFFERENT HARDWARE ARCHITECTURE

The above discussions clearly illustrate the performance effectiveness of the GPU hardware compared to the CPU hardware studied. Clearly, for scientific and engineering applications where total analysis time is important, GPU hardware studied clearly shows a superior timing performance for the computational modelling analysis. However, due to inherent processor differences in terms of clock speeds, benchmarking different hardware is difficult to quantify. Total computational time employed also depend on the cost of arithmetic operations in different architectures that can vary and detailed information in commodity processors similar to that used in the present study are proprietary [24, 25]. A comparison of the floating point operation rate (FLOPs) may also be misleading in diversified architectures with different clock frequencies. In the present work, we obtain and compare a normalized floating point operations rate with the actual FLOPs normalized with their respective clock frequencies. Normalizing FLOPs with the clock frequency helps to avoid the idiosyncrasies of individual hardware, to provide arithmetic power comparisons. Execution time depends on the processor speeds and normalized FLOPs allow for a more definitive comparison over raw wall clock time or FLOPs.

Table 2 presents the normalized FLOPs for the model Mesh B for the pre-conditioned solver on the CPU and GPU studied (shown in KFLOPs). While the normalized FLOPs are higher with BCSR2X2 format compared to CSR in the case of GPU, it was lower for the BCSR2X2 compared to CSR for the CPU. The higher FLOPs for the GPU results in decreased execution time for Mesh B (see Figure 5) since the device are more fully utilized. The KFLOPs shown in

Table **2** are far lower than the theoretical peak of 388.8 GFLOPs [26, 27] for the Nvidia Quadro FX-5600 used in this study. The theoretical peak of the GPU device is calculated under the assumption of large regular data inputs to the ALUs – not exactly what occurs when employed with sparse matrices like those of Mesh B [27-29].

BCSR compression format can have a dramatic effect on the solution time depending on the distribution of the non-zero terms in the sparse matrix. The layout of non-zeros in a particular sparse matrix can be utilized to define the sub-matrix block sizes with several research studies in this direction cited in the literature [2, 20, 30].

VI. CONCLUDING REMARKS

The computational power of GPU-enhanced software for deterministic engineering and scientific applications is well documented, but that power is tempered by the specialized nature from which GPUs have developed. The CSR format, a commonly used data structure for sparse matrices that are seen in several engineering and scientific applications based on spatial discretization of governing partial differential equations, while easy and ubiquitous, lends itself to spatial locality issues that are detrimental to the GPU as demonstrated in the present paper. Blocking compression formats such as BCSR2x2 can help but the degree of enhancement depends on the data pattern of the initial sparse matrix. The effectiveness of the BCSR2X2, block compression format in the context of preconditioned conjugate gradient solver for sparse linear system of equations in composite process flow modelling analysis via finite element discretization has been demonstrated. Clearly, the computational performance of engineering and scientific applications and the execution time depends on the hardware architecture, software paradigms and data structures. The present work clearly highlights the influence of sparse matrix data structures on the computational performance with the CPU and GPU hardware studied.

Interestingly altering the compression format from CSR to BCSR2X2 for mesh B decreased the execution time but not by a significant magnitude (see Figure 5) – illustrating lessened impact of spatial locality. Advances in GPU hardware such as faster context switching and Nvidia parallel data-cache have helped to mitigate locality [31]. However, as with the CPU, algorithmic design will still be important to the GPU to ensure optimal execution. The present findings are based on performance comparisons on a single GPU and CPU. As GPU platforms are increasingly becoming common, future investigations into multi-GPU clusters in different hardware platforms are currently in progress. The current trend in multiple core CPUs and the potential contention for finite GPU resources should also be examined. The present and such systematic studies are required to understand the computational performance – architecture relationships for scientific and engineering applications. This would facilitate the understanding of the optimal architecture for a given problem needs to provide best throughput in complex engineering applications.

ACKNOWLEDGMENT

The support in part by contracts/grants from Office of Naval Research and Clarkson Aerospace through the AFRL Minority Leaders Program is acknowledged.

REFERENCES

[1] Boggan, S., and Pressel, D.: 'GPUs: An Emerging Platform for General-Purpose Computation', in Editor (Ed.)^(Eds.): 'Book GPUs: An Emerging Platform for General-Purpose Computation' (Army Research Laboratory, 2007, edn.), pp.

[2] Jespersen, D.: 'Acceleration of a CFD code with a GPU', Scientific Programming - Exploring Languages for Expressing Medium to Massive On-Chip Parallelism, 2010, 18, (3 - 4), pp. 193-201

[3] Luebke, D., and Humphreys, G.: 'How GPUs Work', Computer, 2007, 40, (2), pp. 96-100

[4] Dokken, T., Hagen, T.R., and Hjelmervik, J.M.: 'An Introduction to General-Purpose Computing on Programmable Graphics Hardware': 'Geometrical Modelling, Numerical Simulation, and Optimization: Industrial Mathematics at SINTEF' (Springer, 2007), pp. 123-161

[5] Bell, N., and Garland, M.: 'Efficient Sparse Matrix-Vector Multiplication on CUDA', in Editor (Ed.)^(Eds.): 'Book Efficient Sparse Matrix-Vector Multiplication on CUDA' (NVIDIA Corp., 2008, edn.), pp.

[6] Smailbegovic, F.S., Gaydadjiev, G.N., and Vassiliadis, S.: 'Sparse Matrix Storage Format', in Editor (Ed.)^(Eds.): 'Book Sparse Matrix Storage Format' (2005, edn.), pp. 445-448

[7] Fatahalian, K., and Houston, M.: 'GPUs: A Closer Look', Queue - GPU Computing, 2008, 6, (2), pp. 18-28

[8] Rumpf, M., and Strzodka, R.: 'Graphics Processing Units: New Prospects for Parallel Computing': 'Numerical Solution of Partial Differential Equations on Parallel Computers' (2006), pp. 89-132

[9] Fan, Z., Qiu, F., Kaufman, A., and Yoakum-Stover, S.: 'GPU Cluster for High Performance Computing', SC '04 Proceedings of the 2004 ACM/IEEE conference on Supercomputing, 2004

[10] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P.: 'Brook for GPUs: stream computing on graphics hardware', ACM SIGGRAPH 2004 Papers, 2004, 23, (3), pp. 777-786

[11] Nickolls, J., Buck, I., Garland, M., and Skadron, K.: 'Scalable Parallel Programming', ACM Queue, 2008, 6, (2), pp. 40-53

[12] Wilkinson, B., and Allen, M.: 'Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers' (Pearson Prentice Hall, 2005, 2 edn. 2005)

[13] Silberschatz, A., Galvin, P., and Gagne, G.: 'Applied Operating System Concepts' (John Wiley & Sons, Inc., 2000, 1 edn. 2000)

[14] Chandrupatla, T.R., and Belegundu, A.D.: 'Introduction To Finite Elements In Engineering' (Prentice-Hall, Inc., 2002, 3 edn. 2002)

[15] Rao, S.S.: 'Applied Numerical Methods for Engineers And Scientists' (Prentice-Hall, Inc., 2002. 2002)

[16] Filipovic, J., Peterlfk, I., and Fousek, J.: 'GPU Acceleration of Equations Assembly in Finite Element Method - Preliminary Results', SAAHPC: Symposium on Application Accelerators in HPC, 2009

[17] Mohan, R., Tamma, K., and Ngo, N.D.: 'On a Pure Finite-Element-Based Methodology for Resin Transfer Mold Filling Simulations', Polymer Engineering and Science, 1999, 39, (1), pp. 26-43

[18] Shinn, A.F., and Vanka, S.P.: 'Implementation of a Semi-Implicit Pressure-Based Multigrid Fluid Flow Algorithm on a Graphics Processing Unit', in Editor (Ed.)^(Eds.): 'Book Implementation of a Semi-Implicit Pressure-Based Multigrid Fluid Flow Algorithm on a Graphics Processing Unit' (ASME, 2009, edn.), pp. 125-133

[19] Hoekstra, A.G., Sloot, P.M.A., Hoffmann, W., and Hertzberger, L.O.: 'Time Complexity of a Parallel Conjugate Gradient Solver for Light Scattering Simulations: Theory and SPMD Implementation', in Editor (Ed.)^(Eds.): 'Book Time Complexity of a Parallel Conjugate Gradient Solver for Light Scattering Simulations: Theory and SPMD Implementation' (Department of Computer System, University of Amsterdam, 1992, edn.), pp.

[20] Choi, J.W., Singh, A., and Vudue, R.W.: 'Model-driven Autotuning of Sparse Matrix-Vector Multiply on GPUs', in Editor (Ed.)^(Eds.): 'Book Model-driven Autotuning of Sparse Matrix-Vector Multiply on GPUs' (ACM New York, NY, USA, 2010, edn.), pp. 115-126

[21] Hugues, M.R., and Petiton, S.G.: 'Sparse Matrix Formats Evaluation and Optimization on a GPU', in Editor (Ed.)^(Eds.): 'Book Sparse Matrix Formats Evaluation and Optimization on a GPU' (2010, edn.), pp. 122-129

[22] Verschoor, M., and Jalba, A.C.: 'Analysis and performance estimation of the Conjugate Gradient method on multiple GPUs', Elsevier Journal of Parallel Computing), 2012, 38, (10-11), pp. 552-575

[23] Buatois, L., Caumon, G., and Bruno, L.: 'Concurrent number cruncher: a GPU implementation of a general sparse linear solver', International Journal of Parallel, Emergent and Distributed Systems, 2009, 24, (3), pp. 205-223

[24] Guo, P., and Wang, L.: 'Accurate CUDA performance modeling for sparse matrix-vector multiplication', in Editor (Ed.)^(Eds.): 'Book Accurate CUDA performance modeling for sparse matrix-vector multiplication' (IEEE, 2012, edn.), pp. 496-502

[25] Kothapalli, K., Mukherjee, R., and Rehman, S.: 'A performance prediction model for the CUDA GPGPU platform', in Editor (Ed.)^(Eds.): 'Book A performance prediction model for the CUDA GPGPU platform' (IEEE, 2010, edn.), pp. 463-472

[26] Nvidia, C.: 'Quadro FX 5600 Datasheet', in Editor (Ed.)^(Eds.): 'Book Quadro FX 5600 Datasheet' (NVIDIA Corp., 2007, edn.), pp.

[27] Corp, Nvidia.: 'CUDA C BEST PRACTICES GUIDE', in Editor (Ed.)^(Eds.): 'Book CUDA C BEST PRACTICES GUIDE' (NVIDIA Corp., 2011, edn.), pp.

[28] Gou, C., and Gaydadjiev, G.N.: 'Elastic Pipeline: Addressing GPU On-chip Shared Memory Bank Conflicts', in Editor (Ed.)^(Eds.): 'Book Elastic Pipeline: Addressing GPU On-chip Shared Memory Bank Conflicts' (ACM New York, NY, USA, 2011, edn.), pp.

[29] Rehman, S.: 'Exploring Irregular Memory Access Applications on the GPU'. Maters Thesis, International Institute of Information Technology, 2010

[30] Krüger, J., and Westermann, R.: 'Linear algebra operators for GPU implementation of numerical algorithms', SIGGRAPH '03 ACM SIGGRAPH 2003, 2003, 22, (3), pp. 908-916

[31] Nvidia, C.: 'NVIDIA's Next Generation Compute Architecture: Fermi', in Editor (Ed.)^(Eds.): 'Book NVIDIA's Next Generation Compute Architecture: Fermi' (NVIDIA Corp., 2009, edn.), pp.

