Using State-Of-The-Art Sparse Matrix Optimizations for Accelerating the Performance of Multiphysics Simulations*

Vasileios Karakasis[1], Georgios Goumas[1], Konstantinos Nikas[1], Nectarios Koziris[1], Juha Ruokolainen[2], and Peter Råback[2]

[1] National Technical University of Athens, Greece
[2] CSC – IT Center for Science Ltd., Finland

1 Introduction

Multiphysics simulations are at the core of modern Computer Aided Engineering (CAE) allowing the analysis of multiple, simultaneously acting physical phenomena. These simulations often rely on Finite Element Methods (FEM) and the solution of large linear systems which, in turn, end up in multiple calls of the costly Sparse Matrix-Vector Multiplication (SpM×V) kernel. The major—and mostly inherent—performance problem of the this kernel is its very low flop:byte ratio, meaning that the algorithm must retrieve a significant amount of data from the memory hierarchy in order to perform a useful operation. In modern hardware, where the processor speed has far overwhelmed that of the memory subsystem, this characteristic becomes an overkill [1]. Indeed, our preliminary experiments with the Elmer multiphysics package [3] showed that 60–90% of the total execution time of the solver was spent in the SpM×V routine. Despite being relatively compact, the widely adopted Compressed Sparse Row (CSR) storage format for sparse matrices cannot compensate for the very low flop:byte ratio of the SpM×V kernel, since it itself has a lot of redundant information. We have recently proposed the Compressed Sparse eXtended (CSX) format [2], which applies aggressive compression to the column indexing structure of CSR. Instead of storing the column index of every non-zero element of the matrix, CSX detects dense substructures of non-zero elements and stores only the initial column index of each substructure (encoded as a delta distance from the previous one) and a two-byte descriptor of the substructure. The greatest advantage of CSX over similar attempts in the past [4, 5] is that it incorporates a variety of different dense substructures (incl. horizontal, vertical, diagonal and 2-D blocks) in a single storage format representation allowing high compression ratios, while its baseline performance, i.e., when no substructure is detected, is still higher than CSR's. The considerable reduction of the sparse matrix memory footprint achieved by CSX alleviates the memory subsystem significantly, especially for shared memory architectures, where an average performance improvement of more than 40% over multithreaded CSR implementations can be observed.

In this paper, we integrate CSX into the Elmer [3] multiphysics simulation software and evaluate its impact on the total execution time of the solver. Elmer employs iterative Krylov subspace methods for treating large problems using the Bi-Conjugate Gradient Stabilized (BiCGStab) method for the solution of

* The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007–2013) under grant agreement n° RI-261557.

the resulting linear systems. To ensure a fair comparison with CSX, we also implemented and compared a multithreaded version of the CSR used by Elmer. CSX amortized its preprocessing cost within less than 300 linear system iterations and built an up to 20% performance gain in the overall solver time after 1000 linear system iterations. To our knowledge, this is one of the first attempts to evaluate the real impact of an innovative sparse-matrix storage format within a 'production' multiphysics software.

The rest of the paper is organized as follows: Section 2 describes the CSX storage format briefly, Section 3 presents our experimental evaluation process and the performance results, and Section 4 concludes the paper and designates future work directions.

2 Optimizing SpM×V for memory bandwidth

The most widely used storage format for non-special (e.g., diagonal) sparse matrices is the Compressed Sparse Row (CSR) format. CSR compresses the row indexing information needed to locate a single element inside a sparse matrix by keeping only *number-of-rows* 'pointers' to the start of each row (assuming a row-wise layout of the non-zero elements) instead of *number-of-nonzeros* indices. However, there is still a lot of redundant information lurking behind the column indices, which CSR keeps intact in favor of simplicity and straightforwardness. For example, it is very common for sparse matrices, especially those arising from physical simulations, to have sequences of continuous non-zero elements. In such cases, it would suffice to store just the column index of the first element and the size of the sequence. CSX goes even further by replacing the column indices with the delta distances between them, which can be stored with one or two bytes in most of the cases, instead of the typical four-byte integer representation of the full column indices.

Fig. 1. The data structure used by CSX to encode the column indices of a sparse matrix.

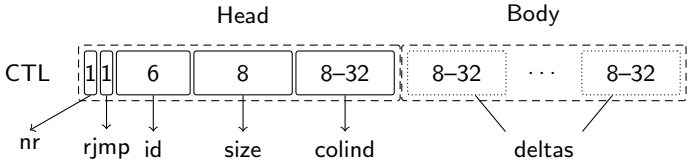


Figure 1 shows in detail the data structure (`ctl`) used by CSX to store the column indices of the sparse matrix. The main component of the `ctl` structure is the *unit*, which encodes either a dense substructure or a sequence of delta distances of the same type. The unit is made up of two parts: the *head* and the *body*. The head is a multiple byte sequence that stores basic information about the encoded unit. The first byte of the head stores a unique 6-bit ID of the substructure being encoded (e.g., 2×2 block) plus some metadata information for changing and/or jumping rows, the second byte stores the size of the substructure (e.g., 4 in our case), while the rest store the the initial column index of the encoded substructure as a delta distance from the previous one in a variable-length field. The body can be either empty, if the type ID refers to a dense substructure, or store the delta distances, if a unit of delta sequences is being encoded.

Table 1. The test problems used for the experimental evaluation.

Problem name	*Equations involved*	*SpM×V exec. time (%)*
fluxsolver	Heat + Flux	57.4
HeatControl	Heat	57.5
PoissonDG	Poisson + Discontinuous Galerkin	62.0
shell	Reissner-Mindlin	83.0
vortex3d	Navier-Stokes + Vorticity	92.3

CSX supports all the major dense substructures that can be encountered in a sparse matrix (horizontal, vertical, diagonal, anti-diagonal and row- or column-oriented blocks) and can easily be expanded to support more. For each encoded unit, we use LLVM to generate substructure-specific optimized code in the runtime. This adds significantly to the flexibility of CSX, which can support indefinitely many substructures, provided that only 64 are encountered simultaneously in the same matrix. The selection of substructures to be encoded by CSX is made by a heuristic favoring those encodings that lead to higher compression ratios.

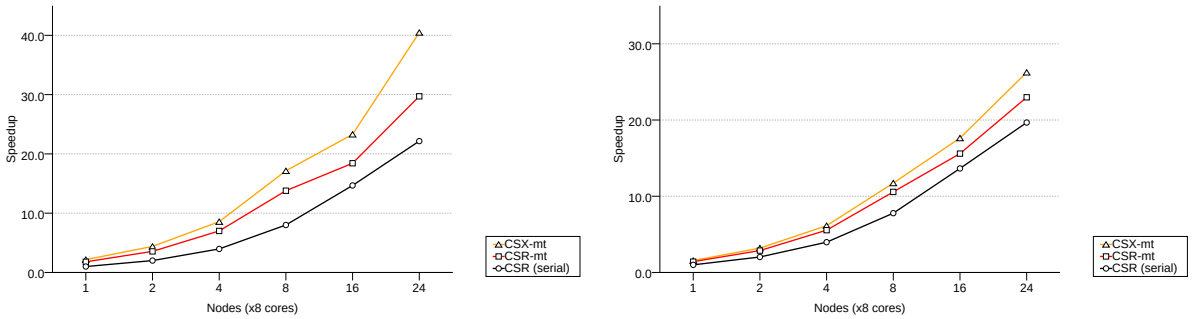
Detecting so many substructures inside a sparse matrix though, can be costly and this is not strange to CSX. Nonetheless, we have managed to considerably reduce the preprocessing cost without losing in performance by examining a mere 1% of the total non-zero elements using samples uniformly distributed all over the matrix.

3 Experimental Evaluation

The integration of the CSX storage format into the rest of the Elmer code was straightforward; Elmer can delegate the SpM×V computation to a user-specific shared library loaded at runtime, so implementing the required library interface was enough to achieve a seamless integration. The default implementation of CSR inside Elmer is single-threaded, but we also implemented a multithreaded version to perform a fair comparison with the multithreaded CSX. Our experimental platform consisted of 192 cores (24 nodes of two-way quad-core Intel Xeon E5405 [Harpertown] processors interconnected with 1 Gbps Ethernet) running Linux 2.6.38. We used GCC 4.5 for compiling both Elmer (latest version from the SVN repository) and the CSX library along with LLVM 2.9 for the runtime code generation for CSX. Table 1 shows the 5 problems we selected from the Elmer test suite for the evaluation of our integration. We have appropriately increased the size of each problem to be adequately large for our system. Specifically, we opted for problem sizes leading to matrices with sizes larger than 576 MiB, which is the aggregate cache of the 24 nodes we used. Finally, we have used a simple Jacobi (diagonal) preconditioner for all the tested problems.

Figure 2 shows the average speedups achieved by simply the SpM×V code (Fig. 2(a)) and the total solver time (Fig. 2(b)) using the original Elmer CSR, our multithreaded CSR version and the CSX (incl. the preprocessing cost), respectively. In the course of 1000 linear system iterations, CSX was able to achieve a significant performance improvement of 37% over the multithreaded CSR implementation, which translates to a noticeable 14.8% average performance improvement of the total execution time of the solver. Nevertheless, we believe that this improvement could be even higher if other parts of the solver exploited parallelism within a single node as well, since the SpM×V component would become

Fig. 2. Average speedup of the Elmer code up to 192 cores using the CSX library (1000 linear system iterations).



(a) Speedup of the total execution time spent inside the SpM×V library, including the preprocessing cost in the case of CSX.

(b) Speedup of the total solver time.

then even more prominent, allowing a higher performance benefit from the CSX optimization. Concerning the preprocessing cost of CSX, we used the typical case of exploring all the candidate substructures using matrix sampling, and yet it was able to fully amortize its cost within 224–300 linear system iterations.

4 Conclusions & Future Work

In this paper, we presented and evaluated the integration of the recently proposed Compressed Sparse eXtended (CSX) sparse matrix storage format into the Elmer multiphysics software package, being one of the first approaches of evaluating the impact of an innovative sparse matrix storage format on a 'real-life' production multiphysics software. CSX was able to improve the performance of the SpM×V component nearly 40% compared to the multithreaded CSR and offered a 15% overall performance improvement of the solver in a 24-node, 192-core SMP cluster. In the near future, we plan to expand our evaluation to NUMA architectures and even larger systems. Additionally, we are investigating ways for minimizing the initial preprocessing cost of CSX and also extensions to the CSX's interface to support efficiently problem cases where the non-zero values of the sparse matrix change during the simulation. Finally, we plan to investigate sparse matrix reordering techniques and how these affect the overall execution time of the solver using the CSX format.

References

1. Goumas, G., Kourtis, K., Anastopoulos, N., Karakasis, V., Koziris, N.: Performance evaluation of the sparse matrix-vector multiplication on modern architectures. The Journal of Supercomputing 50(1), 36–77 (2009)
2. Kourtis, K., Karakasis, V., Goumas, G., Koziris, N.: CSX: An Extended Compression Format for SpMV on Shared Memory Systems. In: Proceedings of the 16th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP'11). pp. 247–256. ACM, San Antonio, Texas, USA (2011)
3. Lyly, M., Ruokolainen, J., Järvinen, E.: ELMER – a finite element solver for multiphysics. In: CSC Report on Scientific Computing (1999–2000)
4. Pinar, A., Heath, M.T.: Improving performance of sparse matrix-vector multiplication. In: Proceedings of the 1999 ACM/IEEE conference on Supercomputing. ACM, Portland, OR, USA (1999)
5. Vuduc, R., Demmel, J.W., Yelick, K.A.: OSKI: A library of automatically tuned sparse matrix kernels. Journal of Physics: Conference Series 16(521) (2005)