Advances in Matrix Functions and Matrix Equations, Manchester UK, 2013

Work in progress

Efficient Computation of the Matrix Square Root in Heterogeneous Platforms

Pedro Costa pfac@di.uminho.pt
Alberto Proença aproenca@di.uminho.pt
Rui Ralha* r_ralha@math.uminho.pt

Context

The square root of a matrix is a common operation in problems of several fields, including Markov models of finance, the solution to differential equations, computation of the polar decomposition and the matrix sign function. Efficient coding of numerical algorithms that takes advantage of recent diversity of computing resources allows the study of more complex problems [1].

Previous work on this algorithm coding has been mainly focused on multi-core shared memory environments; heterogeneous distributed memory environments are still unexplored. The available resources in the recent hardware accelerators hold great potential to improve performance and efficiency.

The Numerical Algorithm Group (NAG)[2] delivers a highly reliable commercial numerical library containing several specialized multi-core functions for matrix operations. While the NAG library includes some implementations for CUDA-enabled Graphics Processing Unit (GPU) accelerators in heterogeneous platforms, it has yet no matrix square root function optimized for these devices [3].

The same applies for GPU libraries listed by NVidia [4, 5, 6, 7, 8]. In particular, the Matrix Algebra on GPU and Multi-core Architectures (MAGMA) project, which aims to develop a LAPACK competitor package for heterogeneous platforms, did not address yet the matrix square root computation [9].

In a previous work, E. Deadman (NAG) and others devised a blocked approach to the Schur method to compute the square root of a matrix in a multi-core environment [10]. While blocked approaches aim a more efficient use of the memory hierarchy, they are also very well suited for vector computing devices, such as GPUs and the recent Intel Many Integrated Core (MIC) architecture. This work addresses efficient computation of the matrix square root in heterogeneous platforms.

Matrix Square Roots

The square root of a matrix A is any matrix X which satisfies the equation $A = X^2$. When it exists, it is not unique. When A has no real negative eigenvalues, it has a unique square root whose eigenvalues all lie in the open right half-plane (i.e. have non-negative real parts) [11, p. 20]. This is the so-called principal square root $A^{1/2}$ and plays a major role in applications. Therefore, we will be interested in computing such square root whenever it exists.

The Schur method of Björck and Hammarling [12] computes the square root of the matrix by reducing A to the upper triangular form T and solving

$$U_{ii}^2 = T_{ii} \ , \tag{1}$$

$$U_{ii}U_{ij} + U_{ij}U_{jj} = T_{ij} - \sum_{k=i+1}^{j-1} U_{ik}U_{kj} \ , \tag{2}$$

where $U^2 = T$, being U also upper triangular. This is the most numerically stable method and it is implemented in MATLAB as the `sqrtm` and `sqrtm_real` functions [13].

Equations (1) and (2) describe an algorithm which can be computed in three distinct ways, due to the dependency each element has on those on its left and below. In [10], the first implementation, in Fortran, iterates over the columns of a matrix, from left to right, and follows the column from the bottom up. At any given point, only one element of the matrix is ready to be computed. This implementation, while impossible to parallelize, allows for a more efficient usage of the cache memory by taking advantage of unit-stride accesses (which directly improve both spatial and temporal locality).

Column accesses is optimal in Fortran since arrays are stored as column-major; on the other hand, when using C/C++ it is more intuitive to think of arrays as row-major, which implies iterating first over the rows of the matrix in order to maximize cache efficiency. The implementation remains similar: rows are accessed from the bottom up, in each row the elements are accessed from left to right, and at any given moment only one element is ready to be computed.

One other way of implementing this algorithm is iterating over the super-diagonals of the matrix. Starting with the main diagonal and going up, all the elements in the same diagonal can be computed in parallel, although it worsens locality.

As for the blocked implementations, the sub-matrices in the main diagonal are also upper triangular, so the point method is applied. The remaining blocks are computed by solving the Sylvester equation.

Heterogeneous Platforms

In this work, heterogeneous platforms contain one or more computational nodes interconnected through a low latency communication topology. Each node contains one or more multi-core conventional computing units, and hardware accelerator device(s). These platforms are said to be heterogeneous since they collaboratively use different types of computing units, such as GPUs and Intel MIC devices.

This work intends to efficiently implement the diagonal approach of the matrix square root algorithm on heterogeneous platforms, where it may take advantage of the massive parallelism available in these systems. Attaining high performance in heterogeneous platforms is not trivial, as it requires understanding the programming model (or even the

*Dep. Math and Applications, University of Minho, Portugal

paradigm) for a particular device and the underlying hardware architecture.

In a computational node, the multi-core Central Processing Unit (CPU) devices share a single memory space, despite the existence of multiple memory slots. This programming model is known as shared memory. Current heterogeneous nodes implement the distributed memory model: each accelerator device has its own distinct memory space, separated from the one used by the multi-core and other accelerator devices; and each computational node has memory spaces distinct from those used by the other nodes. Communication between distinct spaces is expensive and must be avoided.

Efficient development for these platforms can be complex: it requires efficient workload and data distribution among available resources, transparent managing of the communications among memory spaces and adapting the implementation to the particular characteristics of each device. Tools are available to help developers to manage some of these issues, and this work addresses the following:

- tools for automatic workload distribution of parallel code, among shared memory computing units: OpenMP and Threading Building Blocks (TBB) [14];
- tools to extend scheduling to accelerator devices: OpenACC[15] and TBB (for Intel devices)[16];
- tools to transparently integrate distributed memory devices in a single computing node, using a single address space: GPU And Multi-core Aware (GAMA), a framework under development at University of Minho and University of Texas at Austin [17].

Results

To evaluate the scalability of the algorithm in a multi-core environment, tests were performed varying the number of threads and the dimension of the matrix for each approach and method: rows vs diagonals, block method vs point method. The column approach is deliberately left out, since it is similar to the row approach and is less intuitive to implement in C/C++ obtaining good cache efficiency.

Results show for each combination approach-method the execution time for each matrix dimension: one fitting in L2 cache, another for L3 and two large enough to force main memory accesses. Results are presented in a normalized way for the sake of readability.

Tests were performed using a computational node in the SeARCH[1] cluster at University of Minho (UMinho), with two Intel Xeon E5-2650 devices at 2.0 GHz and 64 GB RAM. Having hardware support for 32 threads (with Hyper-Threading), runs were measured using powers of 2 as the number of threads, increasing until performance drops.

So far, all described implementations were first implemented in a multi-core environment, using point and block methods. The next step focused on the scalability study of the algorithm: in [10], the authors performed tests using the full parallelism supported by the hardware and the sequential version; no tests were performed to verify how the execution time changes as the number of threads increases. These results help to plan efficient implementations with hardware accelerators. In particular, it is relevant to find bottlenecks as this will condition further algorithm optimizations to explore massive parallelism.

[1] `http://search.di.uminho.pt`

References

[1] Mark D. Hill and Michael R. Marty. "Amdahl's Law in the Multicore Era". In: *Computer* 41.7 (July 2008), pp. 33–38. DOI: 10.1109/MC.2008.209.

[2] The Numerical Algorithms Group. *NAG Numerical Components*. URL: http://www.nag.co.uk.

[3] The Numerical Algorithms Group. *NAG Numerical Routines for GPUs Manual*. Apr. 2012. URL: http://www.nag.com/numeric/GPUs/naggpu_doc_0.6.pdf.

[4] AccelerEyes. *SQRTM - Jacket Wiki*. URL: http://wiki.accelereyes.com/wiki/index.php/SQRTM.

[5] EM Photonics. *CULAPACK Function List*. URL: http://www.culatools.com/dense/lapack/.

[6] NVIDIA. *cuBLAS Library - User Guide*. Version 5.0. 2012-10. URL: http://docs.nvidia.com/cuda/pdf/CUDA_CUBLAS_Users_Guide.pdf.

[7] NVIDIA. *cuSPARSE Library - User Guide*. Version 5.0. Oct. 2012. URL: http://docs.nvidia.com/cuda/pdf/CUDA_CUSPARSE_Users_Guide.pdf.

[8] N. Bell and M. Garland. *Cusp Library Features*. Version 0.3.0. Mar. 2012. URL: http://code.google.com/p/cusp-library/wiki/Features.

[9] E. Agullo et al. "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects". In: *Journal of Physics: Conference Series* 180.1 (2009).

[10] Edvin Deadman, Nicholas J. Higham, and Rui Ralha. "Blocked Schur Algorithms for Computing the Matrix Square Root". In: *Applied Parallel and Scientific Computing: 11th International Conference, PARA 2012, Helsinki, Finland*. Ed. by P. Manninen and P. Öster. Vol. 7782. Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2013, pp. 171–182. DOI: 10.1007/978-3-642-36803-5_12.

[11] Nicholas J. Higham. *Functions of Matrices: Theory and Computation*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2008. ISBN: 978-0-898716-46-7.

[12] Åke Björck and Sven Hammarling. "A Schur method for the square root of a matrix". In: *Linear Algebra and its Applications* 52–53 (1983), pp. 127–140.

[13] Nicholas J. Higham. *The Matrix Function Toolbox*. URL: http://www.ma.man.ac.uk/~higham/mftoolbox.

[14] Intel. *Intel Threading Building Blocks Documentation*. URL: http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/index.htm.

[15] OpenACC. *The OpenACC Application Programming Interface*. Version 1.0. Nov. 2011. URL: http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf.

[16] Intel. *Intel Xeon Phi Coprocessor Developer's Quick Start Guide*. URL: http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-developers-quick-start-guide.

[17] Artur Mariano et al. "A (ir)regularity-aware task scheduler for heterogeneous platforms". In: *Proceedings of the 2nd International Conference on High Performance Computing*. Kiev, Oct. 2012, pp. 45–56.