Implementing band solvers in MAGMA

Razvan Carbunescu

Motivation

Linear algebra packages as LAPACK ubiquitous in today's computations appearing as an easy way to get performance with little programmer knowledge required of the optimizations required for different machines and systems.

When because of program and data sizes one computer wasn't enough to contain an application linear algebra packages moved to parallel machines and we got packages as scaLAPACK which try and provide a similarly great performance with little knowledge required of network topologies or other parallel programming problems.

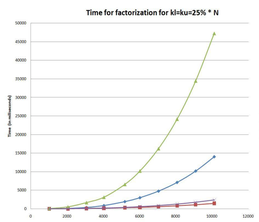
As improvement in simple-core CPU's performance has dropped off while our applications are growing larger and another type of different platform has appeared namely the multicore and hybrid cpu+gpu systems. In order to address performance on these systems linear algebra packages as PLASMA (1) and MAGMA(2) are being developed to help get better performance out of our machines.

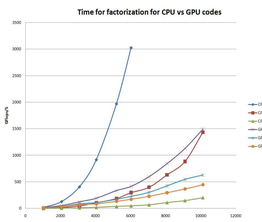
Since most matrices that appear in real-world applications can be considered (after preconditioning) as large banded matrices with relatively small bandwidth implementing a banded solver within the MAGMA could provide benefit to many applications. To that end this project deals initially with a LU factorization of a banded matrix in MAGMA.

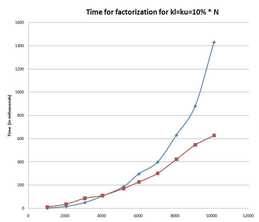
Results

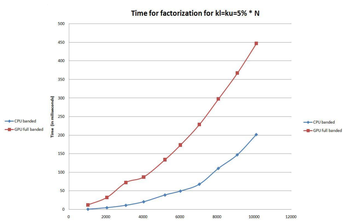
The results presented below are comparing the following codes:

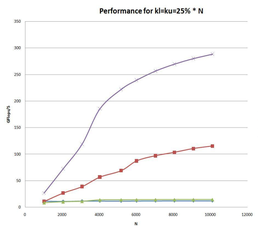
- ***CPU banded*** is the single precision general banded matrix storage LU factorization (sgbtrf) from Intel's MKL LAPACK implementation
- ***CPU full*** is the single precision general matrix LU factorization (sgetrf) from Intel's MKL LAPACK implementation
- ***GPU banded*** is the algorithm that I implemented and could be named single precision banded general matrix LU factorization in MAGMA (sgebtrf)
- ***GPU full*** is the single precision general matrix LU factorization (sgetrf) available in MAGMA 0.2 (http://icl.cs.utk.edu/magma/)

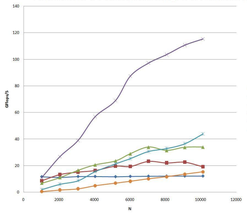


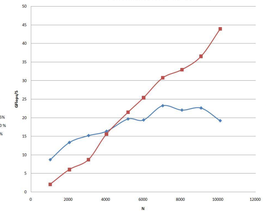


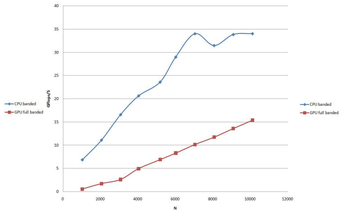












From these simulations it can be seen that the banded GPU code always does better than the non-banded GPU code and than the non-banded CPU code. Also when the bandwidth of the matrix is large the algorithm beats the banded CPU code. For a fixed bandwidth it can also be seen that as the dimension of the matrix increases the GPU version eventually catches up and then beats the CPU code which plateaus must faster.

Work in Progress and Future Work

While the current code will work well for large bandwidth matrices we want to address the smaller bandwidth matrices by implementing the Magma version of the band solver for matrices stored in band format. This code is currently in progress but there have been some issues with the correct panel factorization for the banded storage case.

In the future after this other algorithm is tested and verified the next simple extension will be to copy and adapt these 2 algorithms for all data types: double (d), complex(c), double complex(z) and also to take advantage of any symmetry in the matrix: a banded positive definite matrix (pob) and positive definite matrix in band storage (pb).

After that the next natural extension is to incorporate other factorizations as QR factorization and Hessenberg decomposition.

Acknowledgements

I would like to thank Prof Jim Demmel from UC Berkeley and Prof Stan Tomov from University of Tennessee for their help during this project.

Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227).

References

(1) PLASMA webpage: http://icl.cs.utk.edu/plasma/
(2) MAGMA webpage: http://icl.cs.utk.edu/magma/

Single Precision General Banded matrix LU factorization (sgebtrf) Algorithm in MAGMA

The overall idea of the MAGMA operations is to do as the BLAS2 and the small operations on the CPU and to leave the intensive BLAS3 operations to the GPU.

Assuming the matrix A resides on the CPU initially the entire matrix is copied over from the CPU to the GPU. This in practice might not be the case as the matrix A could already exist on the GPU therefore cancelling this cost.

For block A(i,i) we do:

Step 1)

Panel P(i,i) is moved from the GPU to the CPU

Step 2)

The CPU begins the factorization of the Panel which is a BLAS 2 set of operations so even though it operates on less of the matrix this step might take longer than the GPU part

The GPU computes the U(i-1,i+1) by a triangular solve with LL(i-1,i-1) which was computed for block A(i-1,i-1). The triangular solve is a BLAS 3 operation.

The GPU then computes an update to a part of the matrix A(i:n,i:n); the reason why the update only affects a part of the Schur complement is exactly because the matrix is banded. This update is a BLAS 3 operation.

Step 3)

The CPU sends back to the GPU the factorized panel.

Step 4)

Since the CPU factorization might have pivoted we propagate any permutation over to the matrix rows and columns

Step 5)

In order to ensure that the panel is ready for the next step the GPU computes the U(i,i+1) by a triangular solve with LL(i,i). The triangular solve is a BLAS 3 operation.

The GPU then computes an update to a part of the matrix A(i+1:n,i+1); the reason why the update only affects a part of the Schur complement is exactly because the matrix is banded. This update is a BLAS 3 operation.

After all diagonal blocks are done the Matrix A is copied back from the GPU to the CPU in it's factorized state. It should be noted that the upper bandwidth of the U matrix can grow throughout this process from ku to ku+kl as is hinted in the representation.