Abstract of Dissertation

Krylov Solvers Accelerated by Manycore Processors
(メニーコアプロセッサによるクリロフ解法の高速化)

ジョルゼスク　セルバン

1. Introduction

1.1 Background

At the beginning of the current decade it was realized that the frequency wall has been hit. Mainly due to power consumption problems, the processor frequency could not longer be raised, literally putting an end to the serial speedup or ``free speedup" we have been getting for the last two decades or so. The only way to move forward was to increase parallelism, or in other words the number of cores. Currently, CPUs with four cores are mainstream and eighty-core prototypes have already been presented. In the near future, CPUs with hundreds of cores are expected. This has lead to a paradigm shift in numerical methods and solver design, as only methods which expose large amounts of fine-grained parallelism will be able to scale.

Krylov-type iterative solvers are widely used in the field of finite element computation or for other problems where the matrix of the system to be solved is too large for direct methods to work. Accelerated solvers will benefit fields like structural analysis or computation fluid dynamics, just to name a few. To be truly useful for real world problems, such solvers have to be large scale and not limited by the amount of memory available on one accelerator.

1.2 Motivation and purpose

GPUs are the most efficient accelerators, in terms of both performance and cost, for linear algebra applications.　Last generation GPUs from both NVIDIA and AMD provide close to two Teraflop/s of computational power. Moreover, GPUs are the only truly mainstream manycore processors available today, having hundreds of cores. They provide a glimpse into the future of CPUs and, as such, developing efficient methods, algorithms and codes for GPUs is an excellent way of ensuring future performance and scalability.

The purpose of this research is to investigate the issues related to the implementation of next-generation manycore-accelerated heterogeneous Krylov solvers. We aim to find methods and algorithms able to exploit and provide the large level of fine-grained parallelism required by such processors, ways to reduce the necessary memory bandwidth and communication size and improve convergence. This thesis concentrates on two Krylov solvers: Conjugate Gradients (CG) method, the most used method for systems of linear equations which are symmetric and positive definite and Generalized Minimal Residual (GMRES) method, a very robust method for matrices which can be applied to general matrices.

This research work is divided into four parts:

1. finding/adapting the most appropriate methods that can provide both robustness and reduce the computational, memory and communication requirements
2. writing a distributed GPU-accelerated solver package capable of running on multiple GPUs and on a GPU

cluster

3. building a test-bed composed of a GPU cluster and using it to measure the performance and behavior of various solvers and
4. use the acquired data in order to predict the behavior and issues encountered while porting Krylov solvers on large scale manycore clusters

2. Mixed precision in Krylov solvers

Krylov solvers require double precision accuracy (or larger) in order to converge. Though very recently native precision support has been made available on some GPUs, the vast majority supports single precision (32-bit) alone. Moreover, it is a hardware fact that double precision performance is at least two times slower than single precision on most architectures and an order of magnitude slower on some. In the future, when double precision will be fully supported on all accelerators, the same reasoning will apply to the difference between quadruple and double precision. As such, ways to make use of lower precision as much as possible, both due to lack of support or in order to accelerate the computation, have been devised. Ways of achieving that are *iterative refinement* and *quasi-double precision*.

2.1 Iterative refinement

Iterative refinement is an old and well understood method used to improve the accuracy of linear solvers by wrapping them in an outer correction loop. In a mixed precision formulation, one wraps a lower precision solver into a high precision correction loop. In this way one can get accurate high-precision results while doing most of the computation in lower precision. The use of iterative refinement accelerates the computation by reducing the amount of data that has to be transported between all levels of the system, starting from the memory subsystem of the GPU and up to the cluster interconnect. This is a very important aspect since the very high performance provided by GPU-accelerated nodes (and manycore-accelerated nodes in general) puts an unprecedented strain on the network interconnect.

We investigated the convergence behavior under iterative refinement, where the solver is ran in single precision on the GPU while correction iterations are run in double precision on the GPU. Iterative refinement does not work if the matrices are too ill-conditioned. By using a collection of 60 matrices whose condition numbers, after diagonal preconditioning, have been computed directly from the CG method using Lanczos, the condition number threshold was found to be around 10^8. This is as expected, since the condition number acts as an amplifier for the rounding error and, for 8 digits of accuracy, as in the case of single precision, a condition number larger than 10^8 would leave no single digit correct. The same error amplification explains another effect that we observed: the stalling of the iterative refinement after a certain threshold has been reached. We predicted the point of stalling has to be the “maximum possible” level of residual reduction, by which we understand the number of significant digits remaining after including the effect of the condition number.

In order to be able to efficiently use iterative refinement, we proposed an algorithm which can automatically stop the solver before the previously mentioned threshold has been reached. This algorithm uses condition number estimation, performed directly from the Krylov solver, and outer residual monitoring. A similar algorithm has been developed in order to automatically choose the precision of the inner solver which, if not chosen well, will lead to a large increase in the number of iterations.

2.2 Quasi-double precision

As opposed to mixed-precision methods in which a lower precision is chosen because it is faster, there are particular situations, like vast majority of GPUs, where the lower precision is all that is available. With the cost of many extra instructions, by combining two single precision numbers, it is possible to achieve a precision only slightly lower than double precision. For this reason, it was called quasi-double precision.

Before native precision become available, by computing the difference between the bandwidth used in single and quasi-double precision for all operations involved in a CG solver, we concluded that all involved operations are so memory bounded that even an order of magnitude increase in the number of operations, as required for implementing the quasi-double precision, comes almost for free. Of course, this is also a consequence of the huge computational power of which current GPUs are capable of. These results were confirmed when native double precision become available on the GeForce 280GTX series. Since the analysis was based not on the solver itself but on all the operations involved, this conclusion can be generalized to most Krylov-space solvers, as the operations involved are almost the same. We have also confirmed that the difference between the solutions obtained in double and quasi-double precision is very small, at the order of maximum two digits.

3. Solver design

Implementing distributed multicore accelerated solvers in general is a complex task. Implementing a solver capable of running on heterogeneous clusters, with support for mixed precision, is very difficult without the right tools. Towards this goal, a solver package intended to ease the development of GPU-accelerated Krylov solvers, by abstracting out most of the difficulties encountered when dealing with GPUs, was created. Additionally, the implementation style required for using multiple GPUs makes it very suitable for multicore/manycore. To the best of our knowledge, there is currently no such tool available.

The solver is designed as a stack. The first level provides a unified interface to various Basic Linear Algebra Subsystem (BLAS) libraries and a large set of performance counters. The second level defines vectors and matrices, which can hold data on both CPU and GPU and in various precisions (single, quasi-double, double, integer) and sparse formats. Data movement and conversion are automatically managed. The third level provides all the BLAS functions working directly on vectors and matrices, independent on the data format, location and BLAS libraries being used. The next four levels scale up this functionality to multiple GPUs/cores (levels 4-5) and multiple cluster nodes (levels 6-7). Data synchronization and any kind of communication is automatically managed. Scaling to multiple cores is a side-effect of the necessity of dividing the work into multiple threads, as required for using multiple GPUs in parallel. All solver and preconditioner related but not method specific details, including the iterative refinement loop, form level 8. The actual solvers are implemented on top of this stack. As such, a solver implementation can work unchanged on one or multiple nodes, inside which it can run on one or multiple GPUs or CPU cores or a combination of the two, using any desired combination of data formats, precisions and BLAS libraries. Selecting the run configuration can be done very easily, usually using command-line options or simple API calls.

5. Performance study

In order to get a performance assessment as realistic as possible, tests were done on a collection of matrices, coming from various types of real-world problems, taken from Univ. of Florida's Sparse Matrix Collection. A CG and

a GMRES solver, both built on top of the proposed solver package were run on the previously presented CPUs and GPUs, in single precision with iterative refinement, quasi-double precision and double precision. Tests were run on several multicore CPU and GPU models. Scalability tests were done on a 5-node GPU cluster, with three GPUs in each node. The GPU cluster has been designed to be as close as possible to what an average researched might have. We also made sure the hardware is well balanced, so that useful measurements for modeling scalability can be made.

Using the counters included in the solver, the performance for each piece of hardware was computed. Results were averaged over the entire collection minus the smaller matrices. Matrices were stored and operated on in Compressed Sparse Row (CSR) format. Great care has been taken to optimize the speed of the sparse-matrix vector multiplication routine, where most of the solver time is spent. Overall, the performance of GPU was up to an order of magnitude higher as compared to CPUs from the same generation. For three GPUs per node, up to 15x speedup has been achieved. Besides pure performance, we computed the number of Mflop/s obtained per used Watt and the cost in yen spent for each Mflop/s, on each architecture. It is worth noticing that GPUs excel in every category and that the MFlop/s in quasi-double precision, which can be had on relatively cheap GPUs, is by far the cheapest.

6. Scalability

Scalability for both solvers was analyzed both inside one node and on the cluster. Node level results show that, for matrices which involve lots of communication, even the fast PCIe interconnect can become a bottleneck. However, for the typical matrices found in finite element problems, the scalability is good. The same can be said for the entire cluster. However, by using a performance model, we found that the standard domain-decomposition approach does not scale well to many nodes. For larger systems the use of hierarchical methods might be a much better choice.

7. Conclusions

Motivated by the recent shift towards manycore architectures we studied the performance and other issues encountered while adapting Krylov solvers to multicore processors and GPUs. Iterative refinement and quasi-double precision were analyzed, and quasi-double precision on cheap GPUs was found to be by far the most cost efficient way of running Krylov solvers. As expected, GPUs scored the highest in performance, performance / Watt and cost / performance. This reinforces our belief that GPUs are a platform indeed worth exploring. Compared against CPUs released in the same period, using three GPUs in parallel provided up to 15x.

As a tool for implementing and analyzing Krylov solvers we have created a portable and scalable solver development package. Many things were learned from the implementation of the solver package. First, we found the presence of a layer that automatically manages the movement and conversion of data between host and accelerators and between accelerators to be of particular importance. Since the host and all accelerators have different address spaces, using such a layer is as close as one can get to a shared memory system. Moreover, this makes code much easier to implement when dealing with many different precisions, as one does when using mixed-precision techniques. Second, we found that, as GPUs work so much faster than regular processors, not overlapping communication with computation leads in almost every case to large performance degradation. This happens even when working with multiple GPUs inside one node and communication over the fast and low latency PCIe connection. We have also found that for larger numbers of nodes, parallelization using domain decomposition does not scale well, and that hierarchical methods are required.