PAMM · Proc. Appl. Math. Mech. **14**, 963 – 964 (2014) / **DOI** 10.1002/pamm.201410462

Achieving Portable High Performance for Iterative Solvers on Accelerators

Karl Rupp[1,2*], **Philippe Tillet**[1], **Ansgar Jüngel**[2], and **Tibor Grasser**[1]

[1] Institute for Microelectronics, TU Wien, Gußhausstraße 27-29/E360, A-1040 Wien
[2] Institute for Analysis and Scientific Computing, TU Wien, Wiedner Hauptstraße 8-10/E101, A-1040 Wien

We propose performance enhancements for the implementation of the conjugate gradient method and the generalized minimum residual method for accelerators such as graphics processing units. Through a minimization of memory transfers from global memory via pipelining as well as a reduction of the number of compute kernels through kernel fusion, the performance is improved by up to two-fold when compared to standard implementations based on vendor-tuned routines.

1 Introduction

With the introduction of massively parallel hardware architectures for general purpose computations such as graphics processing units (GPUs) or Intel's many integrated cores architecture, a much higher computational power expressed in the number of floating point operations per second as well as memory bandwidth became available on average workstations. In order to leverage such computational power for the numerical solution of coupled partial differential equations by means for finite difference, finite element, or finite volume methods, iterative solvers are the method of choice due to more inherent fine-grained parallelism as compared to sparse direct solvers. We consider unpreconditioned iterative solvers in this work, as they can still be competitive with preconditioned variants for medium-sized problems due to better parallel hardware utilization.

2 Improved Formulations and Implementations

As iterative solvers exhibit low arithmetic intensity, i.e. the number of floating point operations per byte, the key to fast implementations is a minimization of data transfer through the memory bus connecting the execution units with main random access memory. Following the ideas from vector machines, a direct comparison of the standard conjugate gradient (CG) method and its pipelined variant for an initial guess x_0 is as follows [1]:

Standard CG

$p_0 = r_0 = b - Ax_0$
For $i = 0$ until convergence
1. Compute and store Ap_i
2. Compute $\langle p_i, Ap_i \rangle$
3. $\alpha_i = \langle r_i, r_i \rangle / \langle p_i, Ap_i \rangle$
4. $x_{i+1} = x_i + \alpha_i p_i$
5. $r_{i+1} = r_i - \alpha_i Ap_i$
6. Compute $\langle r_{i+1}, r_{i+1} \rangle$
7. $\beta_i = \langle r_{i+1}, r_{i+1} \rangle / \langle r_i, r_i \rangle$
8. $p_{i+1} = r_{i+1} + \beta_i p_i$

Pipelined CG

$p_0 = r_0 = b - Ax_0$
For $i = 1$ until convergence
1. If $i = 1$: Compute α_0, β_0, Ap_0
2. $x_i = x_{i-1} + \alpha_{i-1} p_{i-1}$
3. $r_i = r_{i-1} - \alpha_{i-1} Ap_i$
4. $p_i = r_i + \beta_{i-1} p_{i-1}$
5. Compute and store Ap_i
6. Compute $\langle Ap_i, Ap_i \rangle$, $\langle p_i, Ap_i \rangle$, $\langle r_i, r_i \rangle$
7. $\alpha_i = \langle r_i, r_i \rangle / \langle p_i, Ap_i \rangle$
8. $\beta_i = (\alpha_i^2 \langle Ap_i, Ap_i \rangle - \langle r_i, r_i \rangle) / \langle r_i, r_i \rangle$

A conventional implementation of the standard CG method is implemented by calling vendor-tuned implementations from the basic linear algebra subprograms (BLAS) supplemented by suitable sparse matrix-vector product implementations [2]. However, by relying on BLAS routines, data reuse between lines 1 and 2 as well as lines 5 and 6 is not possible other than through caching effects. The pipelined version of the CG method, which is obtained by making use of the three-term recursion for computing β_i, allows for better data reuse: Not only can the operations in lines 2-4 be computed without loading the respective values from p_{i-1} and r_i repeatedly, but also the inner product $\langle r_i, r_i \rangle$ can be calculated in the same compute kernel. Then, the computation of Ap_i as well as $\langle Ap_i, Ap_i \rangle$ and $\langle p_i, Ap_i \rangle$ in lines 5 and 6 is possible within a single kernel. Summing up, the pipelined CG version only requires two compute kernels and one data transfer between host and compute device instead of six kernels and two transfers for the standard formulation. We note that these properties are also advantageous in the strong scaling limit on supercomputers [3].

The generalized minimum residual (GMRES) method does not use the same three-term recurrance as the CG method, but instead computes projections onto the full Krylov space $\mathrm{span}\{r, Ar, \ldots, A^{N-1}r\}$ for some value N typically chosen in

* Corresponding author: e-mail rupp@iue.tuwien.ac.at, phone +43 1 58801 36027, fax +43 1 58801 36099

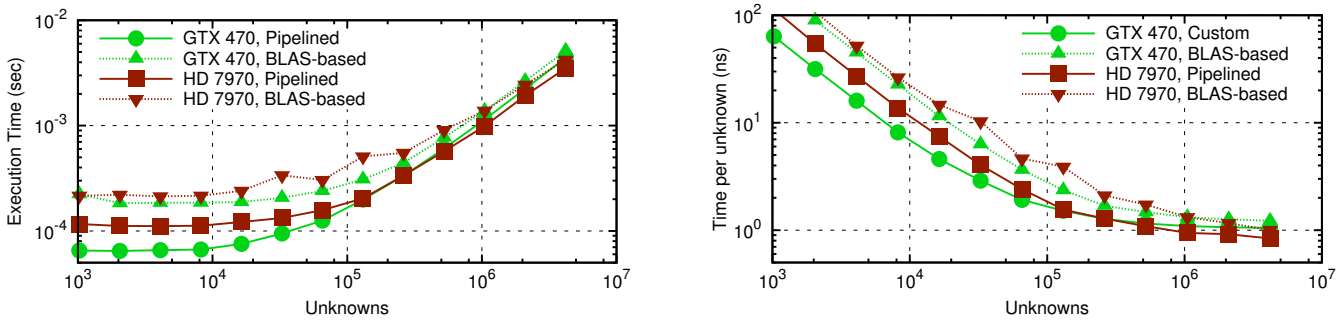

Fig. 1: Comparison of the performance of the standard CG method and the pipelined version. Left: Execution time per CG iteration. Right: Execution time per unknown in each CG iteration. The performance gain of using the pipelined version is particularly pronounced below 10^5 unknowns.

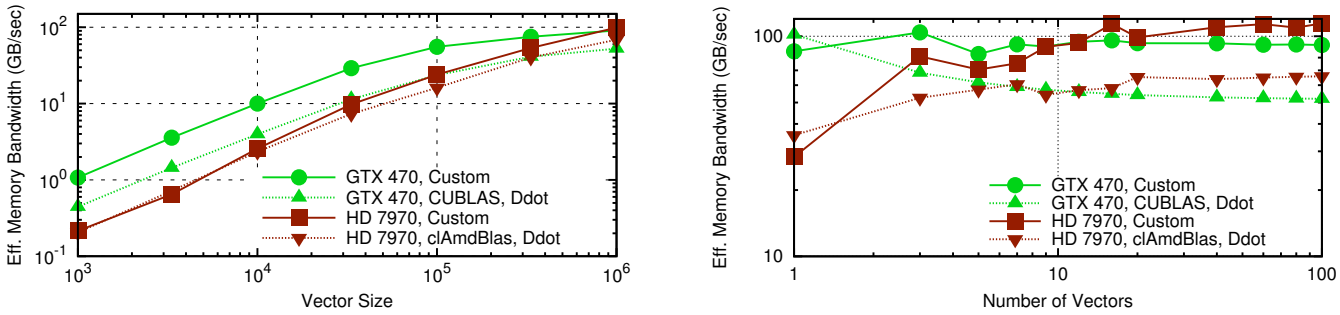

Fig. 2: Comparison of effective memory bandwidth (number of required bytes read per second, not counting redundant bytes) for the Gram-Schmidt orthogonalization $w \leftarrow w - \sum_{i=1}^{N} \langle w, v_i \rangle v_i$ within the GMRES method. Left: Fixed number of $N = 50$ vectors in the basis. Right: Fixed vector size of 10^6 elements.

the range of 20 to 50. Due to the formation of an orthonormal basis by using the Gram-Schmidt method, the computational expense primarily stems from the computation of multiple inner products with the same vector w rather than the sparse matrix-vector product. More precisely, a given orthonormal basis $v_1, \ldots, v_N$ is augmented by a linearly independent vector w by computing $w \leftarrow w - \sum_{i=1}^{N} \langle w, v_i \rangle v_i$ followed by a normalization. Using standard BLAS routines, the vector w is loaded repeatedly in each dot product, reducing the bandwidth available for loading v_i by about a factor of two. Fusing multiple inner products into a single kernel, most unnecessary reads of w can be avoided.

3 Benchmark Results and Conclusion

Our experiments were run on Linux machines equipped with an NVIDIA GeForce GTX 470 using CUDA 5 platform libraries and an AMD Radeon HD7970 using clAmdBlas 1.10.321. All results presented are obtained using OpenCL and double precision arithmetics. Our evaluations using CUDA have not shown any significant performance differences on the NVIDIA GPU and are consequently omitted for the sake of clarity.

Execution times for the CG method applied to a finite difference discretization of the Poisson equation in two spatial dimensions in Fig. 1 show that pipelined methods are able to provide a two-fold performance improvement over the standard formulation. Similarly, an almost two-fold performance gain is observed for the Gram-Schmidt orthogonalization for the GMRES method as depicted in Fig. 2.

Summing up, we have shown that the ideas employed for performance improvements in iterative solvers on large-scale clusters can also be successfully employed on GPUs. This is achieved through a minimization of data transfer and a reduction of synchronization points, resulting in up to a two-fold performance gain.

Acknowledgements K. Rupp, A. Jüngel and T. Grasser acknowledge support from the Austrian Science Fund (FWF), grant P23598.

References

[1] A. T. Chronopoulos and C. W. Gear, J. Comput. Appl. Math. **25**(2), 153–168 (1989).
[2] N. Bell and M. Garland, Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, Portland, Oregon (ACM, New York, 2009), pp. 18:1–18:11.
[3] P. Ghysels and W. Vanroose, ExaScience Lab Technical Report (2012), pp. 1–22.

