Journal of Physics: Conference Series **490** (2014) 012097 doi:10.1088/1742-6596/490/1/012097

Parallel iterative solution of the Hermite Collocation equations on GPUs II

N. Vilanakis and E. Mathioudakis [1]

Department of Sciences, Technical University of Crete, University Campus, 73100 Chania, Crete, Greece

E-mail: nivilanakis@isc.tuc.gr, manolis@science.tuc.gr

Abstract. Hermite Collocation is a high order finite element method for Boundary Value Problems modelling applications in several fields of science and engineering. Application of this integration free numerical solver for the solution of linear BVPs results in a large and sparse general system of algebraic equations, suggesting the usage of an efficient iterative solver especially for realistic simulations. In part I of this work an efficient parallel algorithm of the Schur complement method coupled with Bi-Conjugate Gradient Stabilized (BiCGSTAB) iterative solver has been designed for multicore computing architectures with a Graphics Processing Unit (GPU). In the present work the proposed algorithm has been extended for high performance computing environments consisting of multiprocessor machines with multiple GPUs. Since this is a distributed GPU and shared CPU memory parallel architecture, a hybrid memory treatment is needed for the development of the parallel algorithm. The realization of the algorithm took place on a multiprocessor machine HP SL390 with Tesla M2070 GPUs using the OpenMP and OpenACC standards. Execution time measurements reveal the efficiency of the parallel implementation.

1. Introduction

The Collocation discretization method, based on Hermite bi-cubic elements, is a well-known high accurate finite element technique for solving elliptic boundary value problems (BVPs) [1, 2, 3]. Applying the method to problems on square domains, and by using uniform $n_s \times n_s$ discretization and the appropriate *red-black* numbering of unknowns and equations [4] the resulting red-black collocation linear system $A\boldsymbol{x} = \mathbf{b}$ is in 2-cyclic normal form [5], namely,

$$\begin{bmatrix} D_R & H_B \\ H_R & D_B \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_R \\ \boldsymbol{x}_B \end{bmatrix} = \begin{bmatrix} \mathbf{b}_R \\ \mathbf{b}_B \end{bmatrix} \tag{1}$$

where the matrices D_R and D_B are block diagonal and nonsingular [6]. The large size of the collocation system, especially for fine discretizations, and the demanding storage requirements refer immediately to iterative methods for its efficient solution [7, 8, 9] on parallel computing environments [10, 3, 11, 12]. The preconditioned BiCGSTAB [13] iterative method is an efficient solver for the collocation linear system (1) for high performance parallel architectures [11, 10, 12].

[1] This work was supported by EU (European Social Fund ESF) and Greek funds through the operational program *Education and Lifelong Learning* of the National Strategic Reference Framework (NSRF) - Research Funding Program: THALIS.

In [14] a new Schur Complement type iterative solver was proposed for multicore machines with a Graphic Processing Unit (GPU). The method approximates iteratively only the black coloured unknowns only, while the red ones are directly calculated from the black ones. Since the method is based on the application of a two sided precondition technique into the linear system (1), the iterative procedure of the algorithm for the Schur complement linear system includes an unpreconditioned solver. The BiCGSTAB method is chosen for this purpose evaluating all the black unknowns. Taking into account that this is the most computationally intense part of the solution process, and more specifically the two matrix-vector multiplications involving the Schur complement matrix for every iterative step of BiCGSTAB algorithm, a significant part of the parallel calculations for these matrix-vector multiplications are chosen to be performed on the GPU in order to achieve performance acceleration. The algorithm is based on the shared model for the CPU and GPU cores. But if more than one GPU are available, which means that each one stores the available data in its local memory, and due to the specific structure of matrices H_R and H_B, there is data dependency for every matrix-vector multiplication. The following section presents this extended hybrid algorithm based on the shared-distributed memory model required for the case of multiple GPUs.

2. The parallel algorithm

The parallel algorithm for shared memory architectures in [14] is based on a specific mapping of unknowns into the CPU/GPU computational threads permitting uniform load balancing for the computation among cores, minimizing the communication cost between threads and shared memory and also eliminating the idle core threads during the parallel procedures. Following the same mapping of unknowns in case of N number of GPUs available and for even $k = \frac{n_s}{N}$ the parallel procedures of $\boldsymbol{t} = H_R\boldsymbol{z}$ and $\boldsymbol{q} = H_B\boldsymbol{s}$ GPU matrix vector multiplications have to use GPU and CPU core threads in order to avoid transferring multiple data copies from CPU memory to each GPU local memory. The core threads for every GPU $j = 1, \ldots, N$ calculate the vector parts $\boldsymbol{t_l}$ and $\boldsymbol{q_l}$ with $l = (j-1)k+1, \ldots, jk$ being the number of subvectors of length $2n_s$. For the evaluation on the jth GPU of subvectors $\boldsymbol{t_{(j-1)k+1}}$,$\boldsymbol{t_{(j-1)k+2}}$,$\boldsymbol{t_{jk-1}}$, $\boldsymbol{t_{jk}}$, $\boldsymbol{q_{(j-1)k+1}}$ and $\boldsymbol{q_{jk}}$ the subvectors $\boldsymbol{z_{(j-1)k}}$,$\boldsymbol{z_{jk+1}}$,$\boldsymbol{s_{(j-1)k-1}}$,$\boldsymbol{s_{(j-1)k}}$,$\boldsymbol{s_{jk+1}}$ and $\boldsymbol{s_{jk+2}}$ are needed, which have been stored on the $j-1$ and $j+1$ GPU's local memories. For this reason these operations are performed by CPU core threads, since the required data vectors are also stored on the CPU's shared memory.

The above are implemented in the parallel algorithm that follows and describes the $\boldsymbol{t} = H_R\boldsymbol{z}$ GPU/CPU matrix vector multiplication.

```
!$OMP PARALLEL
        k = n_s/N
        call acc_device_num(j,acc_device_nvidia)
!$ACC KERNELS COPYIN(z((j-1)k+1:jk)) COPYOUT(t((j-1)k+1:jk))
!$ACC LOOP INDEPENDENT
        do  l = (j-1)k+1  to  jk
            The j^th GPU computes  t_l
        enddo
!$ACC END KERNELS
!$OMP SECTION
            The j^th CPU core computes  t_{(j-1)k+1} , t_{(j-1)k+2} , t_{jk-1} , t_{jk}
!$OMP END SECTION
!$OMP END PARALLEL
```

The GPU/CPU matrix vector multiplication $\boldsymbol{q} = H_B \boldsymbol{s}$ can be described with the following analogous parallel algorithm

!$OMP PARALLEL
  $k = \frac{n_s}{N}$
  call acc_device_num(j,acc_device_nvidia)
!$ACC KERNELS COPYIN($\boldsymbol{s((j-1)k+1:jk)}$) COPYOUT($\boldsymbol{q((j-1)k+1:jk)}$)
!$ACC LOOP INDEPENDENT
  do $l = (j-1)k+1$ *to* jk
    The j^{th} GPU *computes* $\boldsymbol{q_l}$
  enddo
!$ACC END KERNELS
!$OMP SECTION
    The j^{th} CPU core *computes* $\boldsymbol{q_{(j-1)k+1}}$, $\boldsymbol{q_{jk}}$
!$OMP END SECTION
!$OMP END PARALLEL

We point out that, for the efficient implementation of the above algorithm parts at least N CPU core threads have to be available. The CPU threads for $j = 1, \ldots, N$ manage the GPU processes. More specifically, each one loads the required data from the CPU shared memory and transfers it to the corresponding GPU local memory. When all GPU calculations are performed the same CPU thread transfers the data from GPU's memory back to the CPU's memory. This is expressed in the above algorithms by the outer OMP PARALLEL procedure. The OpenACC subroutine acc_device_num assigns each GPU card to a CPU core thread created by the OMP parallel region.

3. Realization on a Shared-memory parallel computer with GPUs

The shared memory machine HP SL390s G7 consists of a 6-core Xeon X5660@2.8GHz type processor with 12 MB Level 3 cache memory. The total memory is 24 GB and the operating system is Oracle Linux version 6.2. This machine is also equipped with two Fermi edition Tesla M2070 GPUs [15] connected via PCI-e gen2 slots. Each GPU has 6GB of memory and 448 cores on 14 multiprocessors. The application is developed in double precision Fortran code using OpenMP [16, 17] and OpenACC [18] standards with PGI's compilers [19] version 12.9. The basic linear algebra operations subroutines from scientific libraries BLAS and LAPACK [20] are considered.

For the implementation of the above parallel algorithm the test Dirichlet modified Helmholtz problem, which accepts the following exact solution

$$u(x,y) = 10\ \phi(x)\ \phi(y)\ ,\quad \phi(x) = e^{-100(x-0.1)^2}(x^2 - x),$$

with parameter $\lambda = 1$ was solved.

For the algorithm's performance investigation a single CPU core thread is used for the CPU implementation, while for the CPU/GPU implementations the CPU core threads where as many as the number of GPU cards. Several problem sizes are solved for $n_s = 256$ up to 2048 finite elements in each spatial direction. As every Hermite collocation finite element has 16 degrees of freedom the total degrees of freedom for every problem size is $16n_s^2$. For example in the case of the finest problem the total degrees of freedom are more than 67 millions.

Table 1 presents the total computation time in seconds and the speedup measurements using only the CPU , the CPU and one GPU card and finally cores from CPU and the two GPUs for all problem sizes.

Table 1. Speedup and time execution measurements in seconds.

n_s	CPU time	CPU + GPU time	GPU speedup	CPU + 2GPUs time	2GPUs speedup
256	12.24	11.18	1.09	8.58	1.42
512	88.83	71.25	1.25	54.61	1.63
1024	750.35	549.82	1.37	399.42	1.88
2048	9176.02	6770.11	1.36	5209.01	1.76

As expected the size of the problem and the GPU number affect the algorithm performance. An acceleration performance of almost 50% is observed using all available GPU cores for fine discretization problems.

Acknowledgment
The present research work has been co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program *Education and Lifelong Learning* of the National Strategic Reference Framework (NSRF) - Research Funding Program: THALIS. Investing in knowledge society through the European Social Fund.

References

[1] Christara C C 1996 *Advances in Engineering Software* **27** 71–89
[2] Houstis C E, Houstis E N and Rice J 1997 *Par. Comp.* **5** 141163
[3] Mathioudakis E, Papadopoulou E and Saridakis Y 1996 *Parallel Algorithms and Applications* **8** 141–154
[4] Mathioudakis E, Papadopoulou E and Saridakis Y 2004 *Computers and Maths with Appl.* **48** 951–970
[5] Varga R 2000 *Matrix Iterative Analysis* (New York: Springer Verlag)
[6] Papatheodorou T 1983 *Math. Comp.* **(41),164** 511–525
[7] Saad Y 2003 *Iterative methods for sparse linear systems* (SIAM)
[8] Dongarra J, Duff I, Sorensen D and van der Vorst H 1998 *Numerical Linear Algebra for high-performance computers* (Phil.: SIAM)
[9] Mathioudakis E, Papadopoulou E and Saridakis Y 2006 *WSEAS Trans. on Mathematics* **(5),7** 811–816
[10] Brill S H and Pinder G F 2002 *Parallel Computing* **28** 399–414
[11] Mathioudakis E, Papadopoulou E and Saridakis Y 2003 *Numerical Mathematics and advanced applications - ENUMATH 2001, Springer* 957–966
[12] Mathioudakis E and Papadopoulou E 2007 *Int. J. App. Maths and comp. sciences* **(4),3** 179–184
[13] van der Vorst H A 1992 *SIAM J. Sci.Stat.Comp.* **13** 631–644
[14] Mathioudakis E, Vilanakis N, Papadopoulou E and Saridakis Y Parallel iterative solution of the hermite collocation equations on gpus *Proc. of the World Congress on Engeneering 2013 (WCE2013, Imperial College - London, U.K.), Best Paper Award of The 2013 International Conference of Parallel and Distributed Computing* vol 2 pp 1281–1286 URL `http://www.iaeng.org/publication/WCE2013/WCE2013_pp1281-1286.pdf`
[15] *http://www.nvidia.com/object/tesla-servers.html*
[16] Rohit C 2001 *Parallel programming with OpenMP* (M. K.)
[17] *http://www.openmp.org*
[18] *http://www.openacc.org*
[19] *http://www.pgroup.com*
[20] *http://www.netlib.org*