A scalable hybrid algorithm based on domain decomposition and algebraic multigrid for solving partial differential equations on a cluster of CPU/GPUs

Li Luo
Shenzhen Inst. of Adv. Tech.
Chinese Academy of Sciences
Shenzhen 518055, P. R. China
li.luo@siat.ac.cn

Chao Yang
Inst. of Software
Chinese Academy of Sciences
Beijing 100190, P. R. China
yangchao@iscas.ac.cn

Yubo Zhao
Shenzhen Inst. of Adv. Tech.
Chinese Academy of Sciences
Shenzhen 518055, P. R. China
yb.zhao@siat.ac.cn

Xiao-Chuan Cai
Dept. of Computer Science
Univ. of Colorado at Boulder
Boulder, CO 80309, USA
cai@cs.colorado.edu

ABSTRACT

Several of the top ranked supercomputers are based on the hybrid architecture consisting of a large number of CPUs and GPUs. Very high performance has been obtained for problems with special structures, such as FFT-based image processing or N-body based particle calculations. However, for the class of problems described by partial differential equations discretized by finite difference (or other mesh based methods such as finite element) methods, obtaining even reasonably good performance on a CPU/GPU cluster is challenging. In this paper, we propose and test a hybrid algorithm that matches the architecture of the cluster. The scalability of the approach is realized by a domain decomposition method, and the high performance on GPU is realized by using a smoothed aggregation based algebraic multigrid method. Incomplete factorization, which performs beautifully on CPU but poorly on GPU, is completely avoided in the approach. We report some numerical results obtained by using up to 32 CPU/GPU pairs for solving a PDE problem with up to 32 millions unknowns.

1. INTRODUCTION

Many scientific and engineering problems can be studied by solving partial differential equations (PDEs) discretized by a mesh based method such as finite element or finite difference. Mature and general purpose computational algorithms and high performance software are available for CPU-based large scale supercomputers, for example, PETSc [1]. In the past few years, several of the top ranked supercomputers have moved to hybrid architectures consisting of a large number of CPUs and GPUs. Tremendous speedup has been observed, in comparison with CPU-only calculations, for some computational problems with special structures, for example, Fast Fourier Transforms (FFT) on a single GPU card [13] and GPU clusters [3], Fast Multipole Method (FMM) based particle simulations [7, 9, 10]. Although success has been made in solving dense linear algebra problems using GPUs (see, e.g., [18]), most of the general sparse matrix based parallel solvers don't work well on GPUs, because of the unstructured and irregular nature of the problems and, in particular, the poor performance of incomplete factorization algorithms that are often in the inner-most loop of a preconditioned iterative solver. Efforts have been made in exploiting GPU for sparse matrix calculations. For example, the development version [12] of PETSc begins recently to have GPU support via the Cusp [4] and Thrust [17] libraries from NVIDIA. Rocha et. al. [14] implemented a Jacobi-preconditioned conjugate gradient method to solve sparse linear systems arising in cardiac electrophysiology, where both CSR and ELLPACK matrix formats are investigated. More advanced GPU-based preconditioning techniques such as the algebraic multigrid method is employed in [5] and about 100 times speedup is observed on an eight-GPU configuration than a typical server CPU core.

In this paper, in order to avoid the use of incomplete factorization based components in a preconditioner, we propose and test a hybrid algorithm based on a domain decomposition method and an algebraic multigrid method. The basic assumption required by the proposed algorithms is that equal number of CPUs and GPUs are used on each of the computing node in the cluster. The extension of the algorithm to the case of more GPU cards attached to a CPU is straightforward, but has not been studied in this paper. In the algorithm, the partial differential equation is first divided by a partition of the underlying mesh into a number of overlapping submeshes, each is mapped onto a pair of CPU and GPU. Within a computing node, we perform the subdomain preconditioning operation on the GPU and all the other operations on the CPU. To take architectural advantage of the GPU card, we use a smoothed aggregation (SA) based multigrid method which further partitions the submesh into several much smaller aggregates and the basis of each aggregate gives rise to a set of degrees of freedom on the coarse level. The coarsest level of the SA subdomain preconditioner is solved by a dense LU solver.

The rest of the paper is organized as follows. In Section 2, a hybrid algorithm based on an additive Schwarz preconditioner and an SA subdomain solver is introduced. Numerical results on a NVIDIA Tesla S1070 cluster are then provided in Section 3 to show the efficiency of the proposed method. The paper is concluded in Section 4.

2. A HYBRID ALGORITHM

In many applications, the discretization of a PDE with finite element or finite difference method results in a linear system of equations

$$Ax = b, \tag{1}$$

where A is a large sparse matrix and b a given vector. In this paper, we assume that A is also symmetric positive definite, which is true if, for example, the PDE is a self-adjoint elliptic problem. There are several software packages offering efficient parallel solvers for such problems on supercomputers made of CPUs [1, 6, 8, 19], but algorithms and software that are efficient on a cluster of CPU/GPUs are still lacking. We consider the class of preconditioned iterative methods that solves the preconditioned system

$$M^{-1}Ax = M^{-1}b,$$

where the preconditioner M is an approximation of A^{-1}. We first make two observations:

- The performance of this approach depends heavily on how M^{-1} is defined and implemented, because the computation of $M^{-1}v$ is usually much more expensive than the computation of Av in terms of the compute time, the communication time, and the memory requirement.
- The accuracy of the solution of (1) has (almost) nothing to do with M^{-1}. This means that we have lots of flexibilities about how M^{-1} is computed, and sometimes, we don't have to compute it too accurately in order to obtain higher level of efficiency.

Base on the above observations, we propose to allocate all calculations related to M^{-1} to the GPUs and keep all other calculations on the CPUs. On the GPUs, we approximately compute M^{-1} using a suitable algorithm. Such an approach may not be the best in terms of the total number of floating point operations, but offers much better results in terms of the total compute time.

Because our algorithm is based on domain decomposition and multigrid methods, we further assume that associated with the matrix A, there is a computational domain Ω, with which we obtain mesh based partitions of A. This assumption can be replaced by any graph-based algorithm if the mesh information is not avaliable.

For the model problem studied in this paper, we employ an additive Schwarz preconditioned Conjugate Gradient (CG) algorithm ([15]) to solve (1). The procedure of the preconditioned CG algorithm is provided in Algorithm 1, where M^{-1} is a Schwarz preconditioner.

Algorithm 1 Preconditioned CG for $Ax = b$

```
1. $r_0 = b - Ax_0$, $z_0 = M^{-1}r_0$, $p_0 = z_0$
2. do $j = 0, 1, ...$ until convergence
   3. $\alpha_j = (r_j, z_j)/(Ap_j, p_j)$
   4. $x_{j+1} = x_j + \alpha_j p_j$
   5. $r_{j+1} = r_j - \alpha_j Ap_j$
   6. $z_{j+1} = M^{-1}r_{j+1}$
   7. $\beta_j = (r_{j+1}, z_{j+1})/(r_j, z_j)$
   8. $p_{j+1} = z_{j+1} + \beta_j p_j$
9. end do
```

Denote np as the number of CPU/GPU pairs. We partition the computational domain Ω into np non-overlapping subdomains. An overlapping decomposition can be obtained by extending each subdomain with δ mesh layers. Each overlapping subdomain Ω_k is managed by an MPI process assigned to a CPU/GPU pair. The procedure of the additive Schwarz (AS) preconditioner is provided in Algorithm 2, where R_k^T and R_k serve as a restriction operator and an interpolation operator respectively; their detailed definitions can be found in, e.g., [16].

Algorithm 2 Additive Schwarz: $z \leftarrow AS(r)$

```
1. For $k = 1, 2, ..., np$
       Restriction: $r_k = R_k r$
       Solve the subdomain problem $z_k = B_k^{-1} r_k$
   End for
2. Interpolation: $z \leftarrow \sum_{k=1}^{np} R_k^T r_k$
```

For each CPU/GPU pair, the subdomain matrix $C = B_k$ and the right-hand side vector $d = r_k$ are both copied to

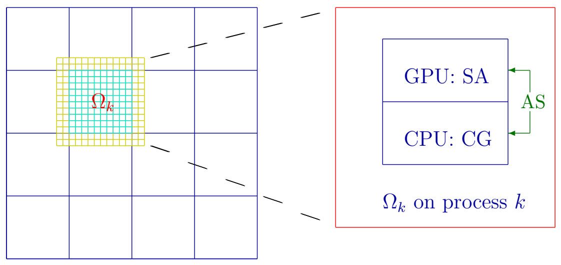

Figure 1: Sketch of the additive Schwarz preconditioned CG algorithm. Left: overlapping domain decomposition of a two-dimensional rectangular domain. Right: Each subdomain is assigned to a MPI process that is further assigned to a CPU/GPU pair.

the local memory of the GPU card before solving the subdomain problem on GPU using a SA algorithm described later. Then the solution vector $x = z_k$ on the GPU side is copied back to the local memory of the CPU, which requires synchronization to make sure the global vector is completely assembled. Note that the subdomain matrix is copied to GPU only once and does not need to be copied back. The data between CPU and GPU within each MPI process is typically transfered through the PCI-Express path between the host CPU memory and the GPU memory. A sketch of the additive Schwarz preconditioned CG algorithm is illustrated in Figure 1. We implement the additive Schwarz preconditioned CG algorithm on a cluster of CPU/GPUs, where SA is allocated and executed on the GPUs and all other operations are performed on the CPUs.

We employ a smoothed aggregation (SA) based algebraic multigrid method to solve the subdomain problems B_k^{-1} in the AS preconditioner. The SA algorithm [2] is defined recursively by using several operators. Let P be the prolongation operator which is a full rank matrix whose range contains the algebraically smoothed components of the residual corresponding to an approximate solution of $Cx = d$, where C and d are the subdomain matrix and the right-hand side for a subdomain problem. With the prolongator, we can define a coarse version of C, as $C_c = P^T CP$, and an iterative method is defined as

$$x \leftarrow x - Py,$$

where y is obtained by solving a coarse grid problem

$$C_c y = P^T(Cx - d).$$

Let $n = n_1$ be the dimension of C, and denote the fine level linear system $Cx = d$ as $C_1 x = d_1$. We introduce a sequence of coarse matrices as

$$C_{l+1} = (I^l_{l+1})^T C_l I^l_{l+1},$$

where the prolongator I^l_{l+1} is defined as the product of a given prolongation smoother, S_l, and a tentative prolongator, P^l_{l+1}

$$I^l_{l+1} = S_l P^l_{l+1}$$

for $l = 1, ..., L - 1$. One popular choice for the prolongation smoother is Richardson's method:

$$S_l = I - \frac{4}{3\lambda_l} C_l$$

where λ_l is an upper bound on the spectral radius of the matrix on level l. At each level, for the system $C_l x = d_l$, we need a smoother

$$x \leftarrow (IT_l C_l)x + T_l d_l,$$

where T_l is an approximate inverse of C_l for $l = 1, ..., L - 1$. Then, SA can be defined as in Algorithm 3.

Algorithm 3 Smoothed Aggregation: $x_l = AMG_l(x_l, d_l)$

```
0. If on the coarsest level, then:
   Solve $C_l x_l = d_l$ by direct LU, else:
1. Apply $\mu$ steps of smoothing to $C_l x = d_l$
2. Coarse grid correction:
   (a). Set $d_{l+1} = (I^l_{l+1})^T(d_l - Cx_l)$ and $x_{l+1} = 0$
   (b). Solve the coarse problem $B_{l+1} x_{l+1} = d_{l+1}$
        by $\gamma$ applications of $x_{l+1} = AMG_{l+1}(x_{l+1}, d_{l+1})$
   (c). Then correct the solution on the level $l$
        by $x_l \leftarrow x_l + I^l_{l+1} x_{l+1}$
3. Apply $\mu$ steps of smoothing to $C_l x = d_l$.
```

Since SA is only used as part of a preconditioner, as observed earlier in the paper, the convergence of SA is not necessary. In our implementation, we only apply the smoother for a small number (μ) of sweeps for the best performance in terms of the total compute time. Increasing the number of smoothing steps helps in reducing the total number of outer iterations, but may increase the overall compute time according to our experiments. Between levels, we use either a Jacobi or a polynomial smoother. When the later is used, the basis functions we choose are the Chebyshev polynomials of the first kind. The polynomial of matrices can be computed by a sequence of sparse matrix-vector multiplication (SpMV) that can be applied in a very efficient way.

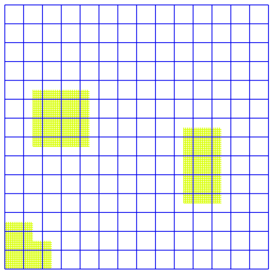

Figure 2: Some possible aggregate candidates on a regular rectangular mesh.

In the SA algorithm, a hierarchy of coarse problems is constructed based on the linear system itself and on certain assumptions about the smooth components of the error. At each level, the prolongation matrix is defined by a decomposition of the set of degrees of freedom associated with the matrix C_l into an aggregate partition, $\{C^1_l, ...C^{N_l}_l\}$ where each aggregate C^i_l is formed based on the connectivity and strength of connection between the elements of C_l, without the need for explicit knowledge of the problem geometry. Figure 2 shows a few possible aggregate candidates on a regular rectangular mesh. The level hierarchy in the SA algorithm is extended until the number of rows in the matrix of the coarsest level is less than 500, which usually results in $4 \sim 5$ levels. Then the matrix on the coarsest level is factorized using a dense LU factorization and is solved by an triangular solver. We employ the MAGMA library[11] which is a dense linear algebra library similar to LAPACK but for heterogeneous/hybrid architectures.

The performance of SA on a GPU depends mainly on three operations: BLAS-axpy (in $2.(a), 2.(c)$), SpMV (in $1., 3.$), and the dense triangular solver $(0.)$. On a GPU, thread blocks are assigned to handle rows of the vector associated with some unknowns and the corresponding rows of matrix.

3. NUMERICAL EXPERIMENTS

The experiments were carried out on a NVIDIA S1070 GPU cluster with 14 nodes. Each node is equipped with two quad-core 2.26 GHz Intel Xeon E5520 CPU processors and four 1.3 GHz NVIDIA Tesla C1060 GPU cards. Nodes are interconnected by a 20Gb InfiniBand DDR network. CUDA Toolkit 3.2 are used for programming and the CUDA kernels in the code are compiled by NVIDIA CUDA Compiler with flag `-arch_sm 13` in order to enable double precision. The CPU code is compiled by Intel MPI compiler using `-O3` optimization level.

In this paper, we study the numerical solution of the Poisson equation with homogeneous Dirichlet boundary conditions on the computational domain $\Omega = [0,1]^2$. A 5-point finite difference scheme is employed to discretize the problem on a uniform $N \times N$ rectangular mesh. The resulting sparse matrix is symmetric positive definite. The stopping condition for the iterative solver is when the relative residual is smaller than 10^{-6}. Even though not tested, we expect the code to work for other second or forth order elliptic problems with variable coefficients.

In the rest of the section, “Iter” is the number of CG iterations, “TSolve” is the total compute time, “TData” is the data transfer time between CPU and GPU, “Eff” is the parallel efficiency as compared with the run using the smallest number of processors in the same table.

First we discuss the parameters of this algorithm and distinguish the optimal. Both the Jacobi and the Chebyshev polynomial smoothers are tested in the experiments. Table 1 shows the impact on the number of iterations and total compute time by using different numbers of sweeps in Jacobi or different degrees of the Chebyshev polynomial. The number of MPI processes is fixed to 32 and the mesh is 8193×4097. The overlap of the additive Schwarz preconditioner is fixed to 1 here. We observe that the number of iterations can be reduced by increasing the number of sweeps of Jacobi or the degree of polynomial of the Chebyshev smoother, however, if the interest is the compute time, 1 sweep of Jacobi is the clear winner.

Table 1: Performance comparison between the Jacobi smoother and the Chebyshev polynomial smoother, with mesh size 8193×4097 and $np = 32$, time is shown in seconds.

Jacobi smoother			Polynomial smoother		
Sweeps	Iter	TSolve	Degree	Iter	TSolve
1	249	26.687	-	-	-
2	242	28.175	2	306	36.719
3	241	29.579	3	316	39.557
4	240	31.631	4	304	39.489
5	236	33.267	5	285	39.517
6	230	34.093	6	283	40.399

Multiple cycles of SA solves the subdomain problem more accurately, thus results in less iteration count of the outer CG. Table 2 reveals how the iteration count is influenced by the cycles of SA. All cycles are executed on GPU so as to avoid multiple data copies between CPU and GPU. In this test, the Jacobi smoother is used. From the table we see that, as expected, the number of iterations is reduced significantly, and the solving time does not increase until cycles= 3.

Table 2: Influence by the number of cycles of SA, with mesh size 8193×4097 and $np = 32$, time is shown in seconds.

Cycles	Iter	TSolve
1	249	26.687
2	209	26.277
3	188	26.919
4	182	29.497
5	178	31.586

We next investigate the optimal size of overlap in the additive Schwarz preconditioner. Generally speaking, a larger overlap usually results in fewer number of iterations due to more communications between subdomains. But the overall compute time may not decrease since the size of the subdomain system, as well as the communication time between the CPUs grow relatively. Table 3 shows the impact of the overlapping size, where the sweeps of Jacobi is 1 and the

SA cycles is 3. As shown, it is a little strange that both the number of iterations and the compute time first grow to some extent, then quickly reduce as the overlap increases.

Table 3: Impact of overlapping size in the additive Schwarz preconditioner, with mesh size 8193×4097 and $np = 32$, time is shown in seconds.

Overlap	Iter	TSolve
0	185	25.85
1	188	26.92
2	191	27.71
3	177	26.53
4	161	24.46
5	153	23.78

We then examine the mesh scalability of the hybrid solver by fixing the number of MPI processes to 32 and increasing the mesh size. The results are provided in Table 4, where results using a CPU-based sparse LU factorization (instead of the GPU-based SA method) as subdomain solvers are also included for comparison. The optimal parameters discussed above are used for all tests from now on. The overlap is fixed to 1 since the CPU-based LU approach requires too much memory when the mesh is very fine. It can be seen from Table 4 that when the mesh is small, the hybrid version costs more time than the pure CPU version, but this situation quickly changes when the mesh size grows up to 2049×2049. It is also observed that the number of iterations of the hybrid solver is greater than that of the pure CPU version, due to the fact that SA is unable to solve the subdomain problems as exactly as direct LU. Table 4 also indicates that the time spent on the data transfer (TData) between the CPU and GPU within the same MPI process is almost negligible compared to the total solution time.

In the strong scaling test, we use a fixed 2049×2049 mesh and increase the number of MPI processes. In the ideal situation, the compute time should be reduced proportionally as more MPI processes are deployed. Strong scaling results using both the CPU-based and the hybrid one-level approaches are provided in Table 5, from which we see that the hybrid approach is always faster than the CPU-based approach.

Table 4: Performance comparison on the mesh scalabilities between the CPU-based and the hybrid approaches, $np = 32$, time is shown in seconds.

	CPU approach		Hybrid approach		
Mesh	Iter	TSolve	Iter	TData	TSolve
513×513	63	0.29	73	0.01	0.67
1025×1025	77	0.77	94	0.04	2.54
2049×2049	98	4.52	126	0.16	3.64
4097×4097	101	24.78	159	0.66	12.68

Superlinear speedup is observed for the CPU-based implementation, but the speedup for the hybrid implementation is not as good.

Table 5: Performance comparison on the strong scalabilities between the CPU-based and the hybrid approaches, mesh size is 2049×2049, time is shown in seconds.

	CPU approach			Hybrid approach		
np	Iter	TSolve	Eff	Iter	TSolve	Eff
2	12	87.37	n/a	62	16.28	n/a
4	55	51.96	84.1%	75	11.77	69.2%
8	79	21.74	100.4%	96	7.88	51.7%
16	75	9.73	112.2%	104	6.79	30.0%
32	98	4.49	121.7%	126	3.57	28.5%

In the weak scaling test, starting from a relatively small 2049×2049 mesh with 4 processes, we increase the number of MPI processes and the mesh size at the same time, so that the mesh size per MPI process is fixed. In the ideal situation, the compute time should remain unchanged which is in fact hard to achieve due to the increasing cost of communication between MPI processes. Table 6 again indicates that the hybrid approach is superior to the CPU-based approach although the parallel efficiency is poor.

4. CONCLUDING REMARKS

In this paper, we proposed and tested a hybrid algorithm based on domain decomposition and smooth aggregation multigrid method for solving elliptic partial differential equations on a cluster of CPUs and GPUs. In the preconditioned Krylov subspace framework, we allocate and execute all pre-

Table 6: Performance comparison on the weak scalabilities between the CPU-based and the hybrid approaches, starting from a 2049×2049 mesh with $np = 4$, time is shown in seconds.

	CPU approach			Hybrid approach		
np	Iter	TSolve	Eff	Iter	TSolve	Eff
4	55	51.97	n/a	98	12.09	n/a
8	85	60.09	86%	159	18.40	66%
16	84	60.24	86%	177	20.56	59%
32	114	68.66	76%	188	26.92	45%

conditioner related operations on the GPUs and all other operations are performed of the CPUs. We carefully investigated the impact of several important parameters that determine the performance of the algorithms. In terms of the number of iterations, the CPU-only approach is clearly better, but for large meshes the hybrid CPU/GPU approach is better in terms of the overall compute time. On the GPU, the mathematically simple Jacobi based smoother performs much better than the more sophisticated Chebyshev polynomial smoother. Our numerical experiments were obtained on a CPU/GPU cluster using up to 32 CPU/GPU pairs, and for a problem with up to 32 millions unknowns.

In this paper, we only considered the case when the number of CPU/GPU pairs in the cluster is relatively small. Multilevel domain decomposition will be necessary for larger clusters. To deal with the additional communication among GPU cards, in multilevel methods, the new feature, GPUDirect, offered in CUDA 4.0 that supports peer-to-peer communication between GPUs over PCIe in the same system, will be very useful. This feature benefits the communication between subdomains in the overlapping Schwarz method, where the input right-hand side and the solution can be sent to the neighboring processors without the aid of CPU memory.

5. ACKNOWLEDGMENTS

We sincerely thank the support of the PETSc group of the Argonne National Laboratory. The work of LL and YBZ was partially supported by the Knowledge Innovation Program of the Chinese Academy of Sciences under grant KJCX2-EW-L01, the work of CY was partially supported by the NSF China grant 61170075, and the work of XCC was partially supported by the NSF grant DMS 0913089.

6. REFERENCES

[1] S. Balay, K. Buschelman, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, L. C. McInnes, B. Smith, and H. Zhang. *PETSc Users Manual.* Argonne National Laboratory, 2010.

[2] M. Brezina, R. Falgout, S. MacLachlan, T. Manteuffel, S. McCormick, and J. Ruge. Adaptive smoothed aggregation (α-SA) multigrid. *SIAM Rev.*, 47:317–346, 2005.

[3] Y. Chen, X. Cui, and H. Mei. Large-scale FFT on GPU clusters. In *ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing*, pages 315–324, New York, NY, USA, 2010. ACM.

[4] The Cusp library. `http://code.google.com/p/cusp-library/`.

[5] C. C. Douglas, H. Lee, G. Haase, M. Liebmann, V. Calo, and N. Collier. Parallel algebraic multigrid method with GP-GPU hardware acceleration. *J. Comput. Appl. Math.* to appear.

[6] R. D. Falgout and U. M. Yang. hypre: A library of high performance preconditioners. In *ICCS '02: Proceedings of the International Conference on Computational Science-Part III*, pages 632–641, London, UK, 2002. Springer-Verlag.

[7] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji. 42 TFlops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence. In *SC'09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 62:1–62:12, New York, NY, USA, 2009. ACM.

[8] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the trilinos project. *ACM Trans. Math. Software*, 31:397–423, 2005.

[9] P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kalé, and T. R. Quinn. Scaling hierarchical N-body simulations on GPU clusters. In *SC'10: Proceedings of the Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[10] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros. A massively parallel adaptive fast-multipole method on heterogeneous architectures. In *SC'09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 58:1–58:12, New York, NY, USA, 2009. ACM.

[11] Matrix algebra on GPU and multicore architectures. `http://icl.cs.utk.edu/magma/`.

[12] V. Minden, B. Smith, and M. G. Knepley. Preliminary implementation of PETSc using GPUs. In *Proceedings of the 2010 International Workshop of GPU Solutions to Multiscale Problems in Science and Engineering*, 2010.

[13] A. Nukada and S. Matsuoka. Auto-tuning 3-D FFT library for CUDA GPUs. In *SC'09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 30:1–30:10, New York, NY, USA, 2009. ACM.

[14] B. M. Rocha, F. O. Campos, R. M. Amorim, G. Plank, R. W. d. Santos, M. Liebmann, and G. Haase. Accelerating cardiac excitation spread simulations using graphics processing units. *Concurr. Comput.: Pract. Exper.*, 23:708–720, 2011.

[15] Y. Saad. *Iterative Methods for Sparse Linear Systems.* PWS, 1996.

[16] B. Smith, P. Bjørstad, and W. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, New York, 1996.

[17] The Thrust library. `http://code.google.com/p/thrust/`.

[18] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Comput.*, 36:232–240, 2010.

[19] A. M. Wissink, R. D. Hornung, S. R. Kohn, and S. S. Smith. Parallel multi-physics AMR applications using the SAMRAI library. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, page 184, New York, NY, USA, 2001. ACM.