GPU-Accelerated Scalable Solver for Banded Linear Systems

Hang Liu
George Washington University

Jung-Hee Seo Rajat Mittal
Johns Hopkins University

H. Howie Huang
George Washington University

***Abstract*—Solving a banded linear system efficiently is important to many scientific and engineering applications. Current solvers achieve good scalability only on the linear systems that can be partitioned into independent subsystems. In this paper, we present a GPU based, scalable Bi-Conjugate Gradient Stabilized solver that can be used to solve a wide range of banded linear systems. We utilize a row-oriented matrix decomposition method to divide the banded linear system into several correlated sub-linear systems and solve them on multiple GPUs collaboratively. We design a number of GPU and MPI optimizations to speedup inter-GPU and inter-machine communications. We evaluate the solver on Poisson equation and advection diffusion equation as well as several other banded linear systems. The solver achieves a speedup of more than 21 times running from 6 to 192 GPUs on the XSEDE's Keeneland supercomputer and because of small communication overhead, can scale upto 32 GPUs on Amazon EC2 with relatively slow ethernet network.**

I. Introduction

A fast solver of banded linear systems is critical for a variety of scientific simulations, e.g, quantum chromodynamics (QCD) and computational fluid dynamics (CFD), where solving large banded linear systems accounts for the majority of the runtime, e.g., 90% or more of the CFD runtime can be spent on solving two partial differential equations (PDE) [1]. Clearly, there is an urgent need for a generic solver, especially a scalable solver for a single integrated linear system, like our motivating problem of cardiac simulation that we will present shortly. In this paper, we are interested in developing a GPU based high-performance solver for such linear systems. In our case, there is inherent dependence within the linear system, thus the problem could not easily be decomposed into disjointed sub-systems, which in turn poses a challenge in achieving high scalability on large-scale GPU based supercomputers.

Recent works fall short in delivering either generality or scalability. On one hand, existing scalable solvers are only applicable for specific banded linear systems. For example, [2] employs domain decomposition to divide the QCD problem into smaller independent blocks. In this case, when applying the Dirichlet boundary [3], each block can be solved independently. In other words, matching one sub-domain with one block largely avoids the inter-node communication, which contributes to good scalability of this solver. For another example, [4] is a solver for tridiagonal matrix (simplified banded linear system) with stable feature. Using the SPIKE algorithm [5] that divides a large banded matrix into several smaller, independently solvable matrices, this solver requires the gathering of all "spikes" and solving them on a single machine. This limit explains its modest scalability of 16 GPUs. Furthermore, there are a few projects that can scale on many GPUs for certain problems, such as wave simulation [6][7], basin simulation [8] and FFT [9].

On the other hand, current generic banded system solvers cannot work with large problems. For example, NVIDIA's CUSP [10] provides a Bi-Conjugate Gradient STABilized (BiCGSTAB) solver that is applicable for banded problems, but it only works on one GPU. Similar limitation on scalability is also observed in other related works [11][12][13]. Notably, in [14], the solver decomposes the banded system into a set of the equations solved by multiple GPU threads, but again this solver runs only on one GPU.

In this paper, we design and develop a GPU based BiCGSTAB solver (GBCG) that meets both *g*enerality and *s*calability requirements. It is well suited for all types of banded linear systems. And this solver combines a new matrix decomposition method with several optimizations for inter-GPU and inter-machine communications to achieve good scalability on large-scale GPU clusters. The popular iterative solvers for linear system $A\vec{x} = \vec{b}$ are the multi-grid (MG) methods [15][16][17], and the Krylov space solvers (e.g., conjugate gradient (CG) solver [3][18], generalized minimal residual method (GMRES) [19], and BiCGSTAB [20]). General speaking, Krylov space solvers enjoy several benefits such as inherent parallelism, applicability on arbitrary grid size, and suitable for any boundary conditions. Note that we also develop a GPU based Stencil Conjugate Gradient solver (GSCG) that can be used to accelerate the symmetric positive definite matrix banded linear systems.

Our approach of matrix decomposition [21] is to factorize the large matrix into smaller sub-matrices in the row-oriented fashion, and solve the sub linear systems in parallel. For a generic linear system like Poisson equation and advection-diffusion (AD) equation that are generated from our motivating heart simulation problem, large matrices cannot be partitioned into independent sub-matrices. Since the construction of the matrix is already done row-wise in the target problem, our method requires no additional communication during matrix decomposition, for every machine solves the rows it has already constructed. However, a number of data transfers are still required in order to correctly solve the problem. For example, the correlation of the sub-matrices is maintained by a vector communication before the Sparse Matrix-Vector Multiplication (SpMV). Specifically, each machine talks to its neighboring machines to update the vectors that are needed by the boundary rows of the sub-matrices during SpMV operations. For the vector communications, we introduce two-phase communication protocol for neighboring machines to reduce the overhead, and for the scalar communications, we apply a tree based broadcast method for all the participating machines. In addition, we utilize the registered memory on MPI calls to inform the network adapter about the virtual-to-physical address of the buffer, and overlap data copying from GPU to the registered memory with GPU computation. By combining these techniques, we are able to significantly reduce the time required for both vector and scalar communications.

The contributions of our work are two-fold. First, to the best of our knowledge, this is the first work in designing and implementing a scalable GPU-based solver for a single

TABLE I: Summary for GPU based solvers

Solver	Problem Type	Decomposition Type	Subsystem type	GPU No.	Achieved/Max Speedup	Efficiency
CUSP BICG[10]	General	N.A.	N.A.	1	N.A.	N.A.
Li[14]	Banded system	SPIKE	Independent	1	N.A.	N.A.
Ament[12]	Poisson	Domain based	Dependent	1⇒6	1.9/6	32%
Babich[2]	QCD	Domain based	Independent	64⇒256	2.56/4	64%
Chang[4]	Tridiagonal system	SPIKE	Independent	1⇒16	10/16	62.5%
Our GBCG	Banded system	Matrix based	Dependent	6⇒192	21.8/32	68.1%

integrated banded $A\vec{x} = \vec{b}$ linear system. In particular, we design row-oriented matrix decomposition method to divide the banded linear system, and utilize several GPU and MPI optimization techniques to minimize the communication overheads from the tight correlation among the sub-systems. As a result, we are able to achieve 70% of the ideal network (InfiniBand) bandwidth. Table I presents a comparison of our GBCG and several existing solvers.

Second, we evaluate our solvers and new combined simulator on two different GPU clusters, i.e., XSEDE's Keeneland supercomputer and Amazon EC2 GPU cluster. With our new GBCG solver, we can run the cardiac simulation with very high resolutions which was difficult before. Furthermore, the tests show that our GSCG outperforms the CUSP CG solver by more than 22%. Also, the GBCG solver can achieve close to 22 times speedup from 6 to 192 GPUs, enjoying fast InfiniBand network on Keeneland, and thanks to small communication overhead, can scale upto 32 GPUs on Amazon EC2 with ethernet network. Note that all the experiments in this paper are studied in double precision, and it is easy to support other precisions.

The paper is organized as follows: we present the background of our simulation and discuss the problem definition in Section II. In Section III, we propose the matrix decomposition based scalable GBCG and GSCG solvers, and present several hardware based optimizations. In Section IV, we evaluate GBCG on Keeneland and Amazon GPU clusters. Finally, we conclude in Section V.

II. Background

A. Motivation

CFD is widely used in scientific and engineering fields to investigate fluid motion and its interactions with certain defined boundaries. In CFD, the Navier-Stokes equations which govern the fluid motion are discretized into linear systems of millions of equations and thus the solution for large scale problem remains computationally challenging. Most of CFD codes are however written in Fortran or C, and translation of whole code to CUDA for GPUs is non-trivial. The major time consumption part of CFD code is the linear system of equations that can be written in the form of $A\vec{x} = \vec{b}$, which is also the target problem for our GPU based banded system solver.

For the application of GPU accelerated flow solver we consider the simulation of blood flow pattern inside the human left ventricle. Features of cardiac flows that include highly complex three-dimensional geometries, relatively high (4,000) Reynolds numbers [1] that result in transition to turbulence, and finally, large-scale boundary motion induced by active (muscle contraction) as well as passive (flow-induced such as in valve leaflets) mechanisms, represent a significant challenge for modeling of cardiac hemodynamics and it demands large

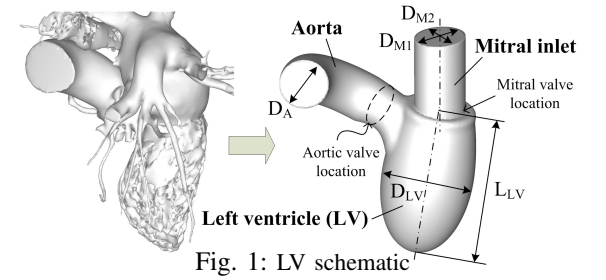


Fig. 1: LV schematic

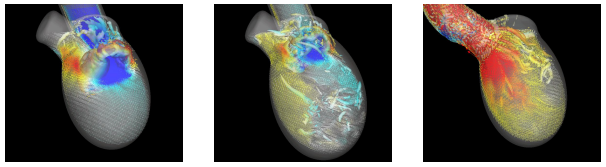


Fig. 2: GPU accelerated diastole and asystole of left ventricle

amount of computational resources. A related work [22] uses GPU to simulate the complex boundary CFD which is the same type as our work, but this work is conducted on only one GPU. In addition, muscle based cardiac simulation is also a popular research topic [23][24][25][26].

In Figure 1, we use a simplified three-dimensional geometric model of the left ventricle which is constructed based on high resolution, multi-detector contrast CT scan data of the normal human left ventricle. The 3D ventricle model is represented by triangular surface meshes and immersed into the rectangular, Cartesian volume grid for the flow simulation. The left ventricle motion (expansion and contraction) is prescribed by satisfying the blood volume flow rate. A size of physical domain for the flow simulation is $6cm \times 6cm \times 11cm$ in real dimension and the blood flow comes into the ventricle through the mitral inlet by the expansion of the ventricle. The flow field simulation is performed by solving the incompressible Navier-Stokes equations using the immersed boundary method. In this method, the boundary condition on the surface of arbitrary geometry is satisfied by using the ghost cells so that the problem can be solved on the Cartesian topology. The details about immersed boundary technique can be found in [1]. Figure 2 presents the screenshot of one heartbeat cycle at three different stages with iso-surface of vortical structure, stream lines, and velocity vectors which are generated by our new simulator. This $256^2 \times 512$ problem running for 10,000 time steps [1] is calculated using GBCG in this work, which was not possible before with the previous MG solver due to its limitations for large problem sizes. Note that we discretize a heart of $6cm \times 6cm \times 11cm$ in physical dimensions into a mathematical model with $256^2 \times 512$ cells. The evaluations of our new cardiac simulation are discussed in Section IV.

[1] In this work, a whole left-ventricle beat time interval is sampled by 10,000 times, each time is called a time step.

B. Problem Definition

The incompressible Navier-Stokes equations are written as

$$\nabla \cdot \vec{u} = 0, \quad \frac{\partial \vec{u}}{\partial t} + (\vec{u} \cdot \nabla)\vec{u} + \frac{1}{\rho}\nabla p = \nu \nabla^2 \vec{u} \tag{1}$$

where $\vec{u}$ is a velocity vector, p is a pressure, ρ and ν are the density and kinematic viscosity of the fluid. The base line flow solver used in this study is *Vicar3D* which is developed by our group at JHU. Here, the equation (1) is solved on the non-body conformal Cartesian grid and the physical boundary of any shape is treated by the sharp-interface immersed boundary method. The incompressible Navier-Stokes equations are usually solved by the fractional step method and a three-step method implemented in the current flow solver can be written as

$$\frac{\vec{u} * - \vec{u}^n}{\Delta t} = -(\vec{u} \cdot \nabla)\vec{u} + \nu \nabla^2 \vec{u}, \tag{2}$$

$$\nabla^2 p = \frac{\rho}{\Delta t}(\nabla \cdot \vec{u}*), \tag{3}$$

$$\vec{u}^{n+1} = \vec{u} * - \frac{\Delta t}{\rho}\nabla p. \tag{4}$$

The equation (2) is the advection-diffusion (AD) equation solved for the intermediate velocity, $\vec{u}*$. The pressure is obtained by solving the Poisson equation, equation (3), and the final velocity field is obtained by the velocity correction step, equation (4). It is important to note that equation (2) and (3) are differential equations and solving these equations takes most of computational time for the flow simulation. If we apply second-order central finite differencing method, equation (2) and (3) are written as

$$\left(1 - \frac{1}{2}\Delta t \nu \cdot {\delta^2}_{i,j,k}\right)\vec{u}* =$$

$$\vec{u}^n + \Delta t \left\{-(\vec{u} \cdot \nabla)\vec{u} + \frac{1}{2}\nu \nabla^2 \vec{u}\right\} = \overrightarrow{RHS}_{AD}, \tag{5}$$

$${\delta^2}_{i,j,k} p = \frac{\rho}{\Delta t}(\nabla \cdot \vec{u}*) = \overrightarrow{RHS}_p, \tag{6}$$

where ${\delta^2}_{i,j,k}$ is a central differencing operator for the Laplacian on the Cartesian grid system which is defined by

$$\begin{aligned}{\delta^2}_{i,j,k}\phi = &\frac{1}{\Delta x_i}\left(\frac{-\phi_{i+1,j,k} + \phi_{i,j,k}}{0.5(\Delta x_{i+1} + \Delta x_i)} + \frac{\phi_{i,j,k} - \phi_{i-1,j,k}}{0.5(\Delta x_i + \Delta x_{i-1})}\right) + \\ &\frac{1}{\Delta y_i}\left(\frac{-\phi_{i,j+1,k} + \phi_{i,j,k}}{0.5(\Delta y_{j+1} + \Delta y_j)} + \frac{\phi_{i,j,k} - \phi_{i,j-1,k}}{0.5(\Delta y_j + \Delta y_{j-1})}\right) + \\ &\frac{1}{\Delta z_i}\left(\frac{-\phi_{i,j,k+1} + \phi_{i,j,k}}{0.5(\Delta z_{k+1} + \Delta z_k)} + \frac{\phi_{i,j,k} - \phi_{i,j,k-1}}{0.5(\Delta z_k + \Delta z_{k-1})}\right),\end{aligned} \tag{7}$$

where (i, j, k) are the indices in (x, y, z) dimension, respectively and ($\Delta x, \Delta y, \Delta z$) are computational grid spacing. Once $\phi_{i,j,k}$ is arranged into one dimensional array $\vec{x}$, equations (5) and (6) are linear systems of equations that can be written in the matrix form as

$$A\vec{x} = \vec{b}. \tag{8}$$

where each row of A is the stencil of corresponding entry in $\vec{x}$. In this paper, we develop GBCG and GSCG for different type of flow simulations, and they are integrated with the flow simulator Vicar3D to solve equations (5) and (6). The storage format of A is optimized for GSCG and GBCG, respectively. We will discuss them in detail in Section III.

III. Scalable GPU-based Solvers

In this section, we first discuss the matrix decomposition method. Next, our GPU based GBCG solver is presented. We further address the communication overheads in GBCG by MPI and GPU communication optimizations. Last, we introduce GSCG method.

A. Matrix Decomposition

Current decomposition methods such as domain decomposition, functional decomposition [27], or SPIKE algorithm [28] are not applicable for an integrated banded linear system. Briefly, domain decomposition aims to split the whole physical domain into several smaller sub-domains and iteratively seek the solution by coordinating the calculation of adjacent sub-domains [29][30]. Often the problems for each sub-domain should be intrinsically independent to avoid massive communication overhead. Similarly, functional decomposition calculates several sub-functions, which are independent with each other, in parallelism. Further, the SPIKE algorithm also aims to divide a large banded linear systems into several independent sub-system. In this case, the solving of the "spikes" is a global procedure done by a single thread, thereby limiting the scalability.

In this work, we design a row-oriented matrix decomposition for a generic banded linear system. In this approach, the matrix A is factorized into an equal number of smaller canonical forms, each of which contains the same number of the rows of matrix A. Similar division is also applied to $\vec{x}$ and $\vec{b}$. Figure 3 describes the matrix decomposition method for $A\vec{x} = \vec{b}$ when employing three GPUs. In this case, GPU 0 has $A_0\vec{x_0} = b_0$, GPU 1 $A_1\vec{x_1} = b_1$ and GPU 2 $A_2\vec{x_2} = b_2$. Specifically, A is divided into three sub-matrices (A_1, A_2 and A_3) that have the same number of rows. next, the sub-vectors that multiply with the sub-matrices should contain extra buffers so that they can be reached by the boundary rows of the sub-matrices. For example, the second GPU which solves $A_1\vec{x_1} = \vec{b_1}$ contains a SpMV of A_1 and $\vec{x_1}$. Assuming the first row of A_1 ($A_1[0]$) is the k-th row of A ($A[k]$), the first non-zero element of $A_1[0]$ (note A_1 contains seven elements) reaches as far as (k-plane width) element of $\vec{x}$. Therefore, $\vec{x_1}$ need to have extra buffers at the boundary. If the vectors do not participate in SpMV operation, they are divided without buffers. Due to A is a narrow-banded matrix, every row of A only needs a relatively small portion of (plane width) the vector for SpMV.

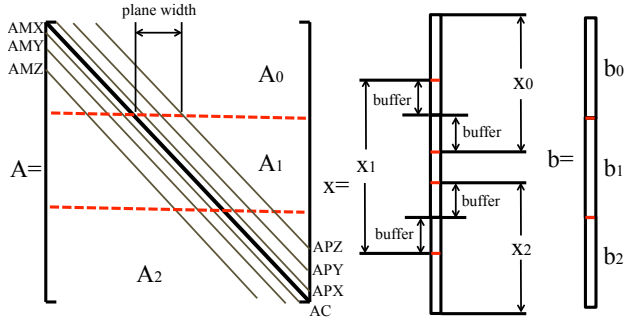


Fig. 3: Matrix decomposition

Our approach of row-oriented matrix decomposition is more appropriate for banded linear system than typical LU (lower-upper, LU factorization) or column-oriented matrix

decomposition [3]. For one, this approach avoids the inter-machine communication during the procedure of dividing $A\vec{x} = \vec{b}$ into sub-systems. In our target problem, every row of A is constructed by one machine. And after the construction of the linear system, all GPUs of the machine only solve the sub-linear equation constructed by its own CPUs. Therefore, there is no communication needed for decomposition. In contrast, both LU and column-oriented matrix decomposition need inter-machine communication for adjusting the workload during decomposition. And for the solving procedure, all of them need the same amount of communications as ours.

B. GPU-based Bi-Conjugate Gradient stabilized (GBCG)

The high level idea of GBCG is similar to pre-conditioner BiCGSTAB [31], which can be divided in four components: pre-conditioner, CG, Bi (bi-direction) and stabilized. The pre-conditioner aims to improve the condition number of the sparse matrix and hence accelerate the solver's convergence speed [3]. CG means that BiCGSTAB evolves from CG solver, and Bi means the search direction contains two parts which are introduced by the BiCG algorithm. Since BiCG aims to solve the linear system without requiring matrix A to be self-adjoint, it maintains the correct search direction by combining the direction with the residual direction, where p_i and s stand for these two directions in BiCGSTAB. Last, stabilized indicates that two convergence constants are computed to repair the irregular convergence behavior of BiCG. This is done by computing ρ_i and α with the initial vector $\hat{r}_0$ which does not change. Theoretically, BiCGSTAB is likely to become stagnate when the convergence requirement is very high. But we vary precision requirement to 10^{-15} without suffering this problem. For the common cases, we only require the precision to 10^{-5} at most.

The algorithm of GBCG is presented in algorithm 1.

Algorithm 1: GBCG

$\hat{\mathbf{r}}_0 = \mathbf{r}_0 = \mathbf{b} - \mathrm{A}\mathbf{x}_0 \quad \rho_0 = \alpha = \omega_0 = 1 \quad \mathbf{v}_0 = \mathbf{p}_0 = \mathbf{0}$

While $\boxed{||\mathbf{r}_{i-1}||_{max} > \epsilon}$ and $i = 1, 2, \ldots, i_{max}$ do:

$\boxed{\rho_i} = < \hat{\mathbf{r}}_0, \mathbf{r}_{i-1} > \quad \beta = \frac{\rho_i}{\rho_{i-1}} \times \frac{\alpha}{\omega_{i-1}} \quad \mathbf{p}_i = \mathbf{r}_{i-1} + \beta(\mathbf{p}_{i-1} - \omega_{i-1}\mathbf{v}_{i-1})$

$\boxed{\mathbf{y}} = \mathrm{K}^{-1}\mathbf{p}_i \quad \mathbf{v}_i = \mathrm{A}\mathbf{y} \quad \boxed{\alpha} = \frac{\rho_i}{<\hat{\mathbf{r}}_0, \mathbf{v}_i>}$

$\mathbf{s} = \mathbf{r}_{i-1} - \alpha\mathbf{v}_i \quad \boxed{\mathbf{z}} = \mathrm{K}^{-1}\mathbf{s} \quad \mathbf{t} = \mathrm{A}\mathbf{z}$

$\omega_i = \frac{\boxed{<\mathrm{K}^{-1}\mathbf{t}, \mathrm{K}^{-1}\mathbf{s}>}}{\boxed{<\mathrm{K}^{-1}\mathbf{t}, \mathrm{K}^{-1}\mathbf{t}>}} \quad \mathbf{x}_i = \mathbf{x}_{i-1} + \alpha\mathbf{y} + \omega_i\mathbf{z} \quad \mathbf{r}_i = \mathbf{s} - \omega_i\mathbf{t}$

Matrix decomposition distributes each GPU the same size of workload and every GPU performs GBCG on its own workload. Once each GPU proceeds to the boxes in GBCG, inter-GPU communication is required. In algorithm 1, we use solid and dotted boxes to show two types of communications. The solid box indicates the scalar communication and dotted box stands for vector communication. The first solid box represents residual checking. Each GPU communicates with all other GPUs to obtain the global maximum norm of residual. If the precision requirement is not achieved and the current iteration time is below the iteration limit, GBCG continues to correct x_i; otherwise GBCG terminates and returns x_i. Within every iteration, there are three more solid boxes that stand for the global scalar sum communication, e.g., the first solid box is ρ_i communication. Each GPU only holds its own part of the inner dot product of $< r_0, r_{i-1} >$. Therefore, a communication is required to sum all ρ_is reside on different GPUs. And the last solid box stands for the sum communication for the numerator and denominator respectively. GBCG puts the local values in a single MPI message in order to save the communication time.

The dotted boxes in algorithm 1 are for vector communication to update the vector buffer before a SpMV. The reason of vector communication is explained earlier in Section III-A. In total, six vector communications is required per-iteration for GBCG.

GBCG stores the banded matrix by a format evolves from Diagonal (DIA) format [32][33]. Specifically, we utilize DIA format other than other formats such as Coordinate (COO), Compressed Sparse Row (CSR), ELLPACK (ELL), Hybrid (HYBRID) to store banded matrix for two reasons. First, DIA is space efficient for banded matrices. For example, in a cubic domain with n as the number of cells in each dimension, the generated sparse matrix A needs to store $7n^3 - 6n^2 + 12n - 8$ double precision values. DIA consumes $7n^3$ storage units, and ELL, CSR, COO and HYBRID require more storage spaces. Second, we optimize the matrix entry access time by arranging every row of DIA format contiguously into one array. In particular, DIA stores the seven diagonal parallel sequences into several arrays. We put the seven arrays into one array by row-wise fashion. Therefore, every row of the banded matrix is stored contiguously. As a result, one warp threads can access global memory in one time to fetch all the requested data for this whole warp since they are stored contiguously [34]. In the meantime, continuous data storage leads to more TLB hits that deliver the faster memory access time.

Applicability to Banded Linear Systems: As it has already shown that BiCGSTAB is applicable for all sparse linear systems [3], our solver can be applied to other linear systems, beyond Possion and AD equations in our cardiac simulation. We further evaluate our solver on other banded linear systems from the Matrix Market [35], such as SHERMAN1 (oil reservoir simulation), NOS7 (Lanczos with partial re-orthogonalization) and GR_30_30 (finite-difference Laplacians). Further explanation for these systems can be found in [35]. For these linear systems, as the largest matrix is 1,000×1,000, our solver can solve them within 200 ms per iteration on a single GPU.

C. Communication Optimization Strategies

Six communications per-iteration as required by our solver could possibly become the Achilles' heel for high scalability. The native implementation shows long communication time and leads to poor GBCG scalability. To address this problem, we develop two inter-machine communication optimization strategies to improve the communication speed substantially. Specifically, we can achieve 70% of the idealized communication bandwidth. As a result, these optimizations lead GBCG scale to O(100) GPUs.

With Poisson equation expressed in terms of $A\vec{x} = \vec{b}$, 25.3 double precision Gflop per-iteration is required for GBCG to solve $256^2 \times 512$ problem. And assuming 16 double precision Gflops can be achieved for each GPU, one GBCG iteration can be completed within 8 ms of wall clock time by 192 GPUs. Here we use 16 Gflops per GPU, double the numbers from [9]

that achieves 8 Gflops double precision per GPU (including communication time) when employing 192 GPUs. As this is all computation time in one iteration, all communication operations of GBCG must be faster than 8 ms.

The Keeneland supercomputer uses Mellanox FDR InfiniBand with 56Gb/s bandwidth (54.54 Gb/s after signaling overheads). Ideally, the vector communication between neighboring machines, for 256×512 double precision array, can be as short as 154 μs. And the scalar communication is almost the time of sending protocol headers time since the data is only one double precision scalar. In total, six communications can be completed in around 300 μs which is shorter than 8 ms. Therefore, GBCG can scale to large number of GPUs even if we only achieve 5% of idealized bandwidth. But without any optimization, our first implementation cannot achieve even 1% of the idealized network bandwidth mostly due to congestion. Figure 6(a) shows that the native communication for vector is around 50 ms.

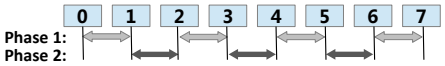


Fig. 4: Two-phase based neighboring machine vector communication

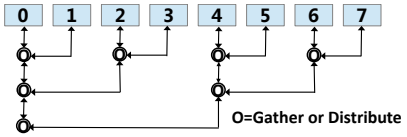
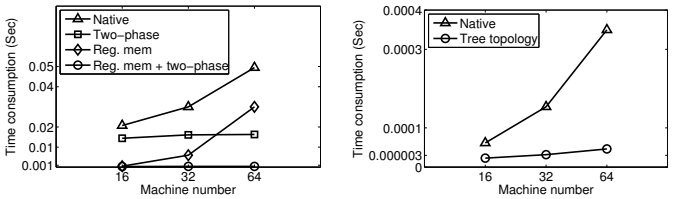


Fig. 5: Tree topology based scalar communication

Vector communication: Our optimized vector communication leverages three optimization techniques. First, we utilize registered memory for MPI data exchange. Specifically, the sending and receiving buffers are allocated as pinned memory. In the meantime, we use the mpi_leave_pinned flag to inform the network adapter the virtual-to-physical address mapping of the sending and receiving buffers. Second, we overlap the copying of sending data from GPU to CPU with GPU computation. Third, we exploit two-phase communication strategy to overcome communication congestion. Figure 4 presents the two-phase strategy with eight machines. In the first phase, we allow machine pairs $[0\ 1],\ [2\ 3], ..., [6\ 7]$ to communicate. In the second phase, the $[1\ 2],\ [3\ 4], ..., [5\ 6]$ machine pairs talk with each other. Figure 6(a) plots the performance of different vector communication techniques. Compared to native implementation, two-phase optimization alone renders 3× speedup for 64 machines. One step further, the combination of two-phase and registered memory optimizations enables our solver to achieve 70% of idealized communication bandwidth. Figure 6(a) also points out that any optimization strategy alone cannot provide such communication speed. In detail, our vector communication time for 64 machines is 448μs seconds (the average of 0.2 million tests).

Scalar communication: Here tree topology based scalar broadcast is utilized to optimize scalar communication. Specifically, our solver conducts scalar communication for exchanging the global maximum or global sum information across all the machines. In the native implementation, every machine simply conducts the broadcast and performs the calculation. This introduces both network congestion and long communication time. Figure 5 describes our tree topology based broadcast



(a) Vector (b) Scalar

Fig. 6: Communication optimizations: (a) Vector communication with registered memory and two-phase communication (b) Scalar communication with tree topology for broadcast.

method. Specifically, the scalar number from each machine is to gather to one machine in a tree based manner for processing. Afterwards, the result is distributed to all machines in the reverse order. Through this optimization, the number of scalar communication decreases from $64^2 \times 2$ to 63×2 times. Figure 6(b) demonstrates that our solver can complete 64 machines scalar communication in 46.2 μs (the average of 0.2 million tests).

D. GPU-based Stencil CG (GSCG) Algorithm

In this part, we develop the GSCG method that is dedicated for banded linear systems with positive definite symmetric matrices by applying matrix decomposition to standard CG method [18]. From the decomposition aspect, the distribution of the workload in GSCG is the same as the GBCG algorithm. We develop GSCG for several reasons. First, CG is applicable for several existing problems, e.g., heat dissipation and PDE on uniform grid. And most importantly, if our heart simulation problem is non-immersed body case discretized on uniform mesh, CG is applicable as well. Second, it solves a simpler matrix that can be further optimized. Specifically, CG has only one SpMV and three communications (one vector, two scalar) per iteration. Third, since CG requires the sparse matrix to be positive definite symmetric, we propose to use a stencil to represent the whole sparse matrix in Poisson and AD equations. In the cardiac simulation, we gather all the cells of the fluid body x[x_axis, y_axis, z_axis] into $\vec{x}$ by x_axis first, y_axis and z_axis last. Therefore, (x_axis, y_axis, z_axis) of every entry x[i] can be induced by index i easily. With the cell's position, we can decide its stencil (note the stencil of each cell is uniquely determined by its location). For example, every interior cell (x_axis×y_axis×z_axis≠0) has the stencil [-1, -1, -1, 6, -1, -1, -1], and the boundary cell is applied with Neumann or Dirichlet condition [1].

TABLE II: GSCG vs. CUSP-CG time consumption

Problem size (# of grid points)	GSCG (seconds)	CUSP-CG (seconds)	Speedup
128^3	1.97	2.20	1.12
256^3	30.6	35.9	1.17
300^3	59.0	72.2	1.22

The stencil representation for banded matrix helps save the precious on-device memory space and reduce the data access time. Specifically, as we discuss in Section III-B, the generated sparse matrix for Poisson or AD equations consumes $7n^3 - 6n^2 + 12n - 8$ double precision storage, where n is the number of cells in each dimension (cube case). Here

the stencil representation for sparse matrix needs only *one* integer to indicate the dimension of the physical domain, as it is discretized on a uniform grid mesh. As we discussed before, GSCG does not have to access on-device memory for matrix entries any more. And [36] points out memory data access is the bottleneck for solving sparse $A\vec{x} = \vec{b}$ on GPU. At meantime, our evaluation shows SpMV consumes around 80% of per-iteration CG time. And the results show that our GSCG is 22% faster than CUSP CG. The detail test results are presented in Table II.

IV. Experiments

In this work, we mainly use two GPU clusters: Keeneland, a supercomputer belong to NSF Extreme Science and Engineering Discovery Environment (XSEDE) program; and Amazon Elastic Compute Cloud (EC2), which provides GPU instances on demand for high-performance computing applications. In Amazon EC2, the users need to pay for other resources, e.g., storage and network, besides the GPU instances. Keeneland and EC2 have NVIDIA Telsa M-class GPU codenamed Fermi M2090 and M2050, respectively. Keeneland installs three GPUs on each node and EC2 two GPUs. Specifically, M2050 has one Telsa GPU with 448 CUDA cores with 515 Gflops double precision floating point peak performance, and M2090 has one GPU Tesla GPU with 512 CUDA cores with 665 Gflops double precision floating point peak performance. M2090 has 6GB GDDR5 memory, twice as much as M2050. ECC is on for both M2050 and M2090. By default, the results present in this section are run on Keeneland, which has Linux 2.6.32 kernel with libraries such as gcc 4.4.6, ifort 12.1.5, Open MPI 1.6.1, and CUDA 5.0. Note that GPUDirect v3 of CUDA 5.0 on Keeneland is not supported. The compilation optimization flag is set as -O0. We set up the same environment on Amazon instances as Keeneland.

We have implemented both GSCG and GBCG in C and CUDA with Jacobi pre-conditioner [3], and have also integrated the solvers with our cardiac simulation code. On each machine, we utilize one host thread to issue all GPU kernels with GPU stream. During scalar communication, host thread works out the local result first. Next, a MPI based communication is used to obtain the global result. For vector communication, the inter-GPU communication can happen on the same machine or two different machines. We overlap these two communications. First, GBCG issues GPUDirect P2P inter-GPU communication to stream for two GPUs on the same machine. Next, GBCG conducts two-phase, registered memory based inter-GPU vector communication for two GPUs on different machines.

In this work, we study GBCG scalability by testing our new cardiac simulator. Since FEM leads to the same type of $A\vec{x} = \vec{b}$ for AD and Poisson equations, here we report the evaluation of Poisson equation unless otherwise specified. We conduct the evaluation of GBCG with different optimization approaches, problem sizes, precision requirements and platform environments. In this section, we use small, medium, and large problems to stand for 128^3, 256^3 and $256^2 \times 512$ problem sizes, respectively. For different precision criteria, we use low, medium, and high precisions for 5E-2, 5E-3 and 5E-4 precision requirements, respectively. By default, we present GBCG results with large problem and high precision requirement.

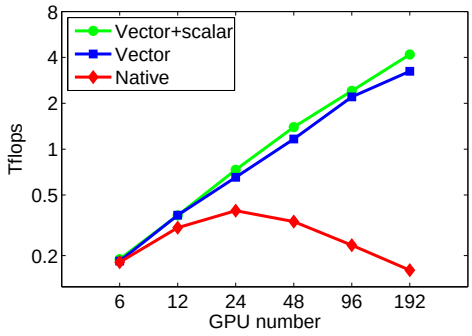


Fig. 7: Gflops of different optimization techniques for GBCG.

Every test is run for 1,000 time steps of the cardiac simulation. Within each time step, GBCG requires from 67 to 216 iterations to converge. Therefore, our performance measurement is an average (arithmetic mean) of 67,000 to 216,000 iterations. The computation rates are reported in terms of Gflops that is determined by dividing the total arithmetic operations—treating all types of double precision floating point operations as 1 flop—per iteration by the average per iteration time (including computation and communication time). To clearly characterize the reason of strong or weak scalability trends in our solver, we define communication overhead as the ratio of communication time and the total runtime. Note this communication ratio only measures the MPI communication and does not include the communication between CPU and GPU, as MPI communication is the major overhead for our solver.

A. Scalability Analysis

Different Optimizations: Overcoming communication limitation is vital for our solver to achieve strong scaling. Figure 7 plots the strong scaling of GBCG corresponding to different communication optimization methods as mentioned in Section III-B. We see significant Gflops increase after the two optimized GBCG due to native GBCG having higher communication ratio. Specifically, native GBCG consumes 9 ms, 9.8 μs and 33 ms for vector, scalar communication and one iteration when using 24 GPUs. As a result, the communication overhead is 54.5%. For 48 GPUs, native GBCG spends 47.79 ms per iteration. But it needs 20.72 ms for vector communication (scalar communication is 10 μs) which leads to communication ratio of 86.3%. Therefore, native GBCG can only scale to 24 GPUs but instantly feel a Gflops drop when GPU number increases to 48. For the vector communication optimized GBCG, it needs 7.63 ms per-iteration with 448 μs and 350 μs for vector and scalar communication on 192 GPUs, respectively. The corresponding communication ratio is 30.1%. This result shows that scalar communication becomes important as vector communication decreases. With the tree-based scalar communication, the total time consumption, vector and scalar communication times are 5.91 ms, 448 μs and 46.2 μs, respectively. The communication ratio becomes 18.3%. This evaluation highlights that efficient MPI communication is essential for scaling our solver.

Different Problem Sizes: Figure 8 shows the scalability trends of different problem sizes. As expected, GBCG cannot scale well for small problem size with 192 GPUs since the total time consumption, vector communication and scalar

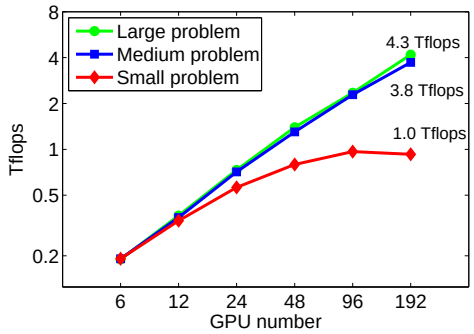


Fig. 8: Scalability of different Poisson equation problem sizes: small, medium and large.

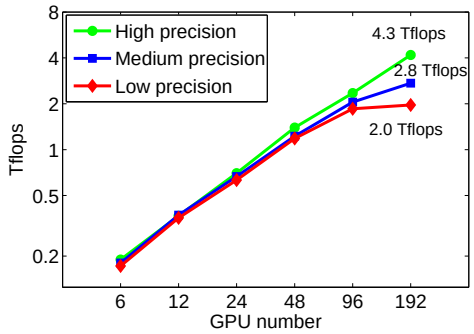


Fig. 9: The scalability of different precision criteria.

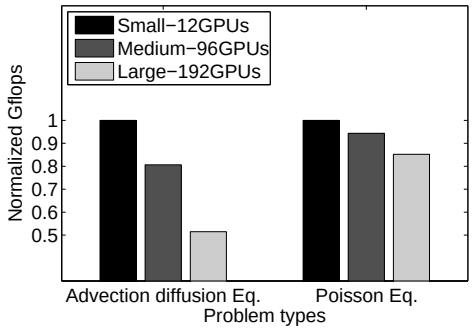


Fig. 10: Weak Scalability: small problem on 12 GPUs, medium problem on 96 GPUs and large problem on 192 GPUs.

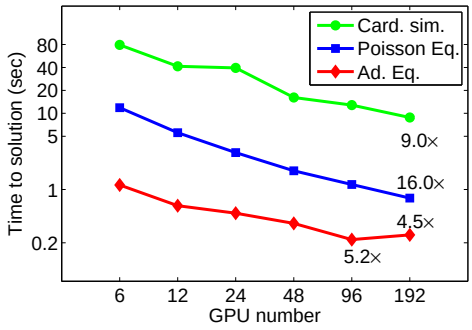


Fig. 11: Runtime of Poisson, AD equation and the simulation.

communication time consumption are 1.7 ms, 428 μs and 46.2 μs, respectively. The communication ratio is 61.2%. But for medium problem, the total time consumption, vector communication, scalar communication takes 3.3 ms, 438 μs and 46.2 μs, respectively. Therefore, the communication ratio is 32.1%. For large problem, the communication ratio is very small at 18.3%. One can see that both medium and large problems can scale nicely to O(100) GPUs.

Different Precision Criteria: We also observe that the per-iteration time consumption varies when the precision criterion fluctuates. Figure 9 plots GBCG scalability trends corresponding to different precision requirements. Clearly, the simulation with high precision scales better than medium precision. And medium precision obtains better Gflops than low precision. According to our test, low, medium and high precision problems need 57, 67 and 145 iterations on average to converge, respectively. These iteration difference leads to the Gflops difference for two reasons. First, every time step, host memory has to talk to device memory for initializing $A\vec{x} = \vec{b}$. More iterations cause host and device communication to occupy less percentage of per-iteration time. Second, [34] points out the L3 cache can hide the global memory access time. As L3 cache needs warm-up, it also helps when we increase the iteration count.

Weak Scalability: To show our solver can balance workload well when problem size increases, we also evaluate the weak scalability of GBCG in cardiac simulation. Figure 10 presents weak scalability of solving AD and Poisson equations. To keep every processor the same workload, as problem size increases, the GPU number increases accordingly. In this test, small problem employs 12 GPUs, medium problem 96 GPUs and large problem 192 GPUs. The runtime of Poisson equation shows that our workload distribution approach can maintain a good weak scalability across different problem size. Specifically, per GPU Gflops of large problem on 192 GPUs can maintain 80% that of small problem on 12 GPUs. But for AD equation, since it can converge in 10 iterations every time step, data copy from host memory to on-device memory takes large portion of the runtime. Therefore, it has a worse weak scalability. Note that GBCG can converge quickly for AD equation due to its good matrix condition number. We use 10^{-12} as the convergence criteria for AD equation.

Time to Solution: With both AD and Poisson equations ported to GPU clusters, we further study the performance of our new cardiac simulation. Figure 11 plots the per time step time consumption for cardiac simulation, Poisson equation and AD equations. As one may recall, before these two equations account for around 90% of the cardiac simulation time. With the new solver, the runtime of two equations is only 11% of the cardiac simulation. Figure 11 also shows that our new integrated cardiac simulation can maintain good scalability from 6 to 192 GPUs. We integrate our solver to cardiac simulator while giving other work full CPU resources. For Poisson equation, the time to solution speedup is 16 $\times$, but for Gflops is 21.8 $\times$. The difference results from 1) for different number of GPUs, it takes different numbers of iterations to converge, and 2) more GPUs mean more buffers, leading to more operations per-iteration. Therefore, the speedup of Gflops is higher than time to solution.

B. Amazon

We study our solver scalability on Amazon GPU high performance instance in this part. According to [37], the

interconnection of Amazon high performance instance is much slower than on supercomputers such as Keeneland in this work. In our test, we put all instances in the same placement group that shall have good inter-node communication. Figure 12 shows the strong scalability of our solver for large problem size from 4 GPUs (on 2 instances) to 32 GPUs (on 16 instances) according to different communication optimizations. The runtimes for vector and scalar communication improve from 70 ms and 40 ms, to 8 ms and 6 ms, respectively. And per-iteration time consumption with 32 GPUs are 320 ms, 210 ms and 73 ms for native, vector optimized and vector and scalar optimized tests. Therefore, the communication ratio of native, vector communication optimized and vector and scalar optimized tests are 93.8%, 80% and 54.8%, respectively. These ratios explain why our optimized solver can scale to 32 GPUs. Note that all the tests are the average report of 100 time steps with total around 17,000 iterations.

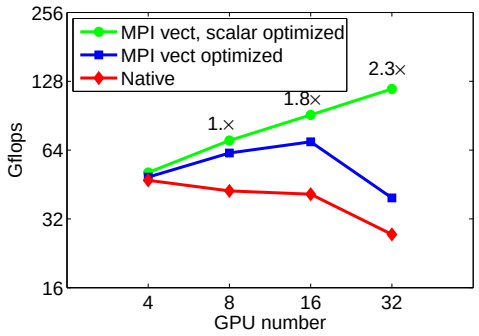


Fig. 12: Gflops on Amazon with different optimization techniques.

V. Conclusion

In conclusion, we propose matrix decomposition for banded linear system and develop a highly scalable banded system solver. In particular, our GSCG solver is 22% faster than CUSP CG solver, and the GBCG solver that is applicable for any banded system can achieve close to 22 speedup when running from 6 to 192 GPUs. Additionally, GBCG is scalable on Amazon EC2 instances.

For future work, we intend to explore scalability further, and will focus on various optimizations, e.g., memory coalescing and memory locality as in [38], improving GPU bandwidth [39]. Furthermore, we will work on better pre-conditioners for the GSCG and GBCG solvers for the problems that need a large number of iterations.

VI. Acknowledgments

This project is in part supported by National Science Foundation grants IOS-1124813 and 1124804, and an NVIDIA Academic Partnership Award.

References

[1] R. Mittal *et al.*, "A versatile sharp interface immersed boundary method for incompressible flows with complex boundaries," *Journal of Computational Physics*, 2008.

[2] R. Babich *et al.*, "Scaling lattice qcd beyond 100 gpus," in *SC*, 2011.

[3] H. William *et al.*, *Numerical Recipes in C: The art of scientific computing*. Cambridge university press, 1988.

[4] L.-W. Chang *et al.*, "A scalable, numerically stable, high-performance tridiagonal solver using gpus," in *SC*, 2012.

[5] E. Polizzi *et al.*, "Spike: A parallel environment for solving banded linear systems," *Computers & Fluids*, 2007.

[6] D. Komatitsch *et al.*, "Modeling the propagation of elastic waves using spectral elements on a cluster of 192 gpus," *Computer Science-Research and Development*, 2010.

[7] M. Rietmann *et al.*, "Forward and adjoint simulations of seismic wave propagation on emerging large-scale gpu architectures," in *SC*, 2012.

[8] M. Wen *et al.*, "Using 1000+ gpus and 10000+ cpus for sedimentary basin simulations," in *CLUSTER*, 2012.

[9] A. Nukada *et al.*, "Scalable multi-gpu 3-d fft for tsubame 2.0 supercomputer," in *SC*, 2012.

[10] NVIDIA CUSP, "http://developer.nvidia.com/cuda/cusp."

[11] J. Krüger *et al.*, "Linear algebra operators for gpu implementation of numerical algorithms," in *TOG*, 2003.

[12] M. Ament *et al.*, "A parallel preconditioned conjugate gradient solver for the poisson problem on a multi-gpu platform," in *PDP*, 2010.

[13] G. Knittel, "A cg-based poisson solver on a gpu-cluster," in *HiPC*, 2010.

[14] A. Li *et al.*, "Spike:gpu - a gpu-based baned linear system solver," 2012.

[15] J. Bolz *et al.*, "Sparse matrix solvers on the gpu: conjugate gradients and multigrid," in *TOG*, 2003.

[16] S. Fulton *et al.*, "Multigrid methods for elliptic problems: A review," *Mon. Wea. Rev*, 1986.

[17] H. Anzt *et al.*, "Block-asynchronous multigrid smoothers for gpu-accelerated systems," Technical report, Tech. Rep., 2011.

[18] J. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain," 1994.

[19] Y. Saad *et al.*, "Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems." *SIAM J. Sci. Stat. Comput.*, 1986.

[20] G. Sleijpen *et al.*, "Bicgstab (l) for linear equations involving unsymmetric matrices with complex spectrum," *Electronic Transactions on Numerical Analysis*, 1993.

[21] H. Liu *et al.*, "Matrix decomposition based conjugate gradient solver for poisson equation," in *SC*, 2012.

[22] W. Li *et al.*, "Flow simulation with complex boundaries," *GPU Gems*, 2005.

[23] A. A. Mirin *et al.*, "Toward real-time modeling of human heart ventricles at cellular resolution: simulation of drug-induced arrhythmias," in *SC*, 2012.

[24] B. J. Pope *et al.*, "Performance of hybrid programming models for multiscale cardiac simulations: Preparing for petascale computation," *Biomedical Engineering, IEEE Transactions on*, 2011.

[25] A. Neic *et al.*, "Accelerating cardiac bidomain simulations using graphics processing units," *Biomedical Engineering*, 2012.

[26] S. Niederer *et al.*, "Simulating human cardiac electrophysiology on clinical time-scales," *Frontiers in Physiology*, 2011.

[27] I. Foster, *Designing and building parallel programs*. Addison-Wesley Reading, MA, 1995.

[28] E. Polizzi *et al.*, "A parallel hybrid banded system solver: the spike algorithm," *Parallel computing*, 2006.

[29] A. Quarteroni *et al.*, *Domain decomposition methods for partial differential equations*. Clarendon Press, 1999.

[30] A. Meyer, "A parallel preconditioned conjugate gradient method using domain decomposition and inexact solvers on each subdomain," *Computing*, 1990.

[31] H. Van der Vorst, "Bi-cgstab: A fast and smoothly converging variant of bicg in the presence of rounding errors," *J. Sci. Statist. Comput*, 1992.

[32] M. Wafai *et al.*, "Sparse matrix vector multiplications on graphic processors," 2010.

[33] N. Bell *et al.*, "Efficient sparse matrix-vector multiplication on cuda," *NVIDIA Technical Report*, 2008.

[34] H. Wong *et al.*, "Demystifying gpu microarchitecture through microbenchmarking," in *ISPASS*, 2010.

[35] M. Market, "http://math.nist.gov/matrixmarket/."

[36] J. Dongarra *et al.*, "An iterative solver benchmark," *Scientific Programming*, 2001.

[37] K. R. Jackson *et al.*, "Performance analysis of high performance computing applications on the amazon web services cloud," in *CloudCom*, 2010.

[38] J. Wu *et al.*, "Optimized strategies for mapping three-dimensional ffts onto cuda gpus," in *InPar*, 2012.

[39] D. Merrill, "Allocation-oriented algorithm design with application to gpu computing," Ph.D. dissertation, University of Virginia, 2012.