Computers and Mathematics with Applications 70 (2015) 1162–1181

Contents lists available at ScienceDirect

Computers and Mathematics with Applications

journal homepage: www.elsevier.com/locate/camwa

Accelerating algebraic multigrid solvers on NVIDIA GPUs

Hui Liu [*], Bo Yang, Zhangxin Chen

Department of Chemical and Petroleum Engineering, University of Calgary, Calgary, Alberta, Canada, T2N 1N4

A R T I C L E I N F O

Article history:
Received 8 July 2014
Received in revised form 5 July 2015
Accepted 11 July 2015
Available online 30 July 2015

Keywords:
Algebraic multigrid
GPU
Parallel computing
Algorithm

A B S T R A C T

This paper presents the development of parallel algebraic multigrid solvers on NVIDIA GPUs. A classical algebraic multigrid solver and a smoothed aggregation algebraic multigrid solver are implemented. The W-cycle, F-cycle and V-cycle are investigated. Various coarsening strategies and interpolation operators are implemented. A group of smoothers are also provided. Numerical experiments show that the solution phase of our GPU-based algebraic solvers is up to 13 times faster than that of the sequential CPU-based algebraic multigrid solvers on our workstation.

1. Introduction

In many scientific computing applications, linear systems are required to be solved. The Krylov subspace solvers and the incomplete LU factorization (ILU) preconditioners [1–4] are the most commonly used methods. The ILU methods are efficient and easy to implement. However, when the matrices of the linear systems are positive definite, multigrid methods are the most effective, including algebraic multigrid methods and geometric multigrid methods. The multigrid methods utilize several coarse levels to interact with the original linear system and to exchange information. The multigrid methods are efficient and it is well-known that the convergence rate of these methods is optimal [5–8].

The AMG solvers have been studied by many researchers. Ruge and Stüben developed a classical AMG solver [5–9] and the RS (Ruge–Stüben) coarsening strategy, which were the foundation behind the development of many other AMG solvers in the early age. Cleary, Luby, Jones and Plassmann designed a parallel coarsening strategy CLJP for parallel computers [10,11]. Henson and Yang proposed parallel coarsening strategies PMIS and HMIS [12,13]. Besides, Yang and her collaborators developed a famous parallel AMG solver BoomerAMG [10,12,14,13], which is the most famous parallel AMG solver/preconditioner for parallel computers. The black oil model is a classical reservoir model in porous media, has been widely applied in the oil and gas industry, has three phases (oil, gas and water) and three components (oil, gas and water), and assumes that the gas component can exist in the oil phase. The model is difficult to solve when the problem size is large and/or heterogeneous geological conditions are applied, such as highly heterogeneous permeability and porosity. Its pressure equations are elliptic or parabolic partial differential equations (PDE) depending on the consideration of the underlying reservoir (stationary or transient), and the pressure unknowns dominate the solution errors. Many multi-stage preconditioners [15,16] were developed to accelerate the solution of linear systems from this model, such as a Constrained Pressure Residual preconditioner [17,18] and a FASP preconditioner [16]. The idea of these multi-stage preconditioners is to solve the pressure equation using multigrid solvers and then to solve the preconditioned system using ILU preconditioning methods [16–18].

Recently, GPU (Graphics Precessing Unit) computing becomes more and more popular. GPUs are special electronic devices designed to manipulate and to accelerate the creation of images in a frame buffer intended for output to display

[*] Corresponding author.
E-mail address: hui.jw.liu@gmail.com (H. Liu).

http://dx.doi.org/10.1016/j.camwa.2015.07.005
0898-1221/© 2015 Elsevier Ltd. All rights reserved.

on screens, which can calculate large amounts of graphical data simultaneously. Modern GPUs have high memory speed and floating point performance, such as the NVIDIA Tesla K40, which has a memory speed of 288 GB/s and a peak double precision floating point performance of 1.66 Tflops, while the memory speed of CPUs is relatively low, such as the DDR4-3200 memory, which has a theoretical peak bandwidth of 51.2 GB/s on a 128-bits bus [19]. The linear solvers and preconditioners, such as algebraic multigrid solvers, are IO intensive applications. The memory speed of GPUs is much higher than memory of CPUs, which makes GPUs good platforms for linear solvers and other scientific computing applications. However, they have different architectures from the traditional CPUs, which means that new implementations must be developed to utilize the power of GPUs, such as FFT [20], BLAS [21,22,20], and Krylov subspace solvers [23,3,4,24]. NVIDIA developed a hybrid matrix format HYB for general matrices [21,22]. Naumov [25] and Chen et al. [24] studied parallel triangular solvers for GPUs. Saad et al. developed a type of JAD matrix for GPU computing and its corresponding SpMV algorithm [4,3]. Chen et al. designed another hybrid matrix format HEC and SpMV algorithm [26], Krylov solvers [27,28] and classical AMG solver [29]. Haase et al. developed a parallel AMG solver using a GPU cluster [30]. Bell et al. from NVIDIA investigated fine-grained parallelism of AMG solvers using a single GPU [31]. Bolz, Buatois, Goddeke, Bell, Wang, Brannick, Stone and their collaborators also studied GPU-based parallel algebraic multigrid solvers, and details can be found in Refs. [32–37]. GPUs are also being widely used in the solution of dense linear systems, FFT [20] and image processing.

This paper presents our work on developing parallel AMG solvers on NVIDIA GPUs, including a classical AMG solver and a smoothed aggregation multigrid solver. For the smoothed aggregation AMG solver, classical methods are applied. For the classical AMG solver, two coarsening strategies, the Ruge–Stüben method and the CLJP method, are implemented. Several interpolation operators are investigated, including standard interpolation, direct interpolation, multi-pass interpolation and an interpolation operator introduced by [30]. The smoothers are unified by a simple formula, and a group of general purpose smoothers are implemented on the GPUs, including the Jacobi, damped Jacobi, weighted Jacobi, block Jacobi, Gauss–Seidel, block-Jacobi–Gauss–Seidel, and polynomial smoothers. When implementing the AMG solvers on the GPUs, the sparse matrix–vector multiplication (SpMV) and vector operations are the most important operations. The implementation of vector operations on GPUs is trivial and a lot of packages, such as cuBLAS (CUDA Basic Linear Algebra Subroutines), provide basic vector operations. The SpMV is relatively complicated, and in this paper, the matrix format HEC and its corresponding SpMV algorithm are adopted [26]. Numerical experiments are performed to analyze the cost of the communications, the IO performance, the vector and matrix–vector operations, and the performance of GPU-based AMG solvers, using the NVIDIA Tesla C2050 GPU and Intel Xeon X5570 CPUs. The GPU memory speed of the Tesla C2050 is 144 GB/s, and the CPU memory type is DDR-1333, whose data width is 64 bits and data transfer speed is 10.6 GB/s. The numerical experiments show that speed-ups up to thirteen have been obtained for the solution phase of our AMG solvers on our workstation with NVIDIA Tesla C2050/C2070 GPUs and Intel Xeon X5570 CPUs.

The framework of the paper is as follows: in Section 2, the algorithms for AMG solvers are presented. In Section 3, GPU computing, the architectures of GPUs and our implementation details are introduced. In Section 4, numerical experiments are carried to test the performance of our implementations, including communication, vector operations, matrix–vector operation and AMG solvers. In Section 5, conclusions and discussions are given.

2. Algebraic multigrid solvers

The linear system we solve is written as follows:

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

where A is a sparse positive-definite matrix ($A \in R^{n\times n}$), b is the right-hand side vector ($b \in R^n$), and x is the unknown to be calculated ($x \in R^n$). This linear system may be derived from the discretization of an elliptic equation, heat transfer equation and a pressure equation from the black oil model or the compositional model by using the finite element method, the finite difference method or the finite volume method. We implement the V-cycle, F-cycle and W-cycle [2]. The structure of the standard V-cycle AMG solver is shown in Fig. 1. This solver has $L + 1$ levels. The original linear system $Ax = b$ stands for the finest level ($l = 0$) and level L is the coarsest level. Two adjoint levels exchange information through restriction operation (fine level to coarse level) and prolongation operation (coarse level to fine level). A typical AMG solver consists of two phases, a setup phase and a solution phase. The setup phase builds the coarser grids, smoothers, restriction operators and prolongation operators. The solution phase solves the multi-level system. Different AMG solvers have different coarser grids, prolongation operators and other components. However, they share the same setup and solution structures, which can be implemented similarly. Details are discussed in the following sections.

2.1. AMG setup phase

In the setup phase, the hierarchical system shown in Fig. 1 is assembled. On each level l except level L, a coarser grid is chosen. The size of the coarser grid should be much smaller than the current grid so that the linear system on the coarser grid is small and easy to solve, and the error on it should approximate the error on the current fine grid well. This requirement can reduce the complexity of the AMG solvers and the calculations in each solution iteration. The choices of coarser grids for various types of smoothed aggregation AMG solvers and classical AMG solvers are different.

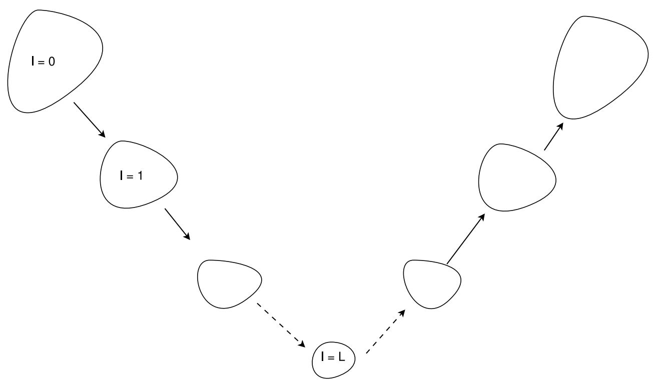

Fig. 1. Structure of AMG solver.

After a coarser grid is chosen, a restriction operator R_l and an interpolation (prolongation) operator P_l are determined. In general, the restriction operator R_l is the transpose of the interpolation (prolongation) operator P_l:

$$R_l = P_l^T.$$

The matrix on the coarser grid is calculated by

$$A_{l+1} = R_l A_l P_l. \tag{2}$$

For many relaxation-type iterative methods, such as the Gauss–Seidel method, the components of the residuals, which are known as the high frequency components of the errors, are damped rapidly [2], while low frequency error components are difficult to damp. However, with a coarser grid or matrix, many low frequency components are mapped into high frequency components on the coarser grid by the restriction operator. Pre-smoothers S_l and post-smoothers T_l are constructed after the A_l is obtained.

The coarsening algorithms we use are the RS algorithm introduced by Ruge and Stüben [5,6] and the CLJP algorithm introduced by Cleary, Luby, Jones and Plassmann [10,11]. The RS algorithm is a two-pass method and is also a sequential algorithm. The CLJP algorithm is relatively easy to parallelize and has been widely applied by traditional parallel computing.

The prolongation operators are the standard one proposed by Ruge and Stüben (`RSSTD`), direct interpolation (`RSD`), multiple pass interpolation (`MPRS`) and the one introduced in Haase's paper [30]. The definition of direct interpolation and multiple pass interpolation can be found in [38]. The last one, denoted in the rest of the paper by `ML`, is simpler than the other two. Let P be the prolongation operator associated with `ML`; we have

$$P_{i,j} = \begin{cases} 1 & i \text{ is a coarsening node,} \\ 0 & i \text{ is a fine node and is weakly connected with } j, \\ 1/n_i & i \text{ is a fine node and is strongly connected with } j, \end{cases} \tag{3}$$

where $P_{i,j}$ is the (i, j)th element of operator P and n_i is the number of coarsening nodes connected with fine node i.

For a given matrix A_l, the right-hand side b_l and the initial solution x_l, the pre-smoother S_l and the post-smoother T_l have the same form so that

$$x_l^{m+1} = \alpha x_l^m + \beta M_l(b_l - A_l x_l^m) = \alpha x_l^m + \beta M_l(r_l), \tag{4}$$

where α and β are related coefficients and M_l is a map from R^{n_l} to R^{n_l}, with n_l being the dimension of matrix A_l. The coefficients α and β are positive and often equal 1. Here we define $\texttt{diag}(A_l)$ as the main diagonal of matrix A_l. $\texttt{block\_diag}(A_l)$ is a diagonal block matrix, which is defined as

$$\texttt{block\_diag}(A_l) = \begin{vmatrix} A_{11} & 0 & \cdots & 0 \\ 0 & A_{22} & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & A_{kk} \end{vmatrix} = \begin{vmatrix} D_1 & 0 & \cdots & 0 \\ 0 & D_2 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & D_k \end{vmatrix}, \tag{5}$$

where each $D_i = A_{ii}$ $(1 \le i \le k)$ is a diagonal block of A_l and k is the number of diagonal blocks. Different smoothers can be defined by

- If $M_l = \texttt{diag}(A_l)^{-1}$ and α and β are both 1, then we have a Jacobi smoother.
- If $M_l = \texttt{diag}(A_l)^{-1}$, $\alpha = 1$ and β is positive but not 1, we have a damped Jacobi smoother.
- If $M_l = \texttt{diag}(A_l)^{-1}$, $\beta = 1 - \alpha$ and $0 < \alpha < 1$, we have the weighted Jacobi smoother.

- If $M_l = \texttt{block\_diag}(A_l)^{-1}$ and α and β are both 1, then we have a block Jacobi smoother. Each subproblem, $D_i x_i = r_i (x_i = D_i^{-1} r_i)$, is solved by the ILU preconditioning method, where a lower triangular system and an upper triangular system are required to be solved.
- If $M_l = L_l^{-1}$ and the two coefficients are 1, where L_l is the lower part of matrix A_l, we have the so-called Gauss–Seidel smoother. Since a lower triangular linear system is required to solve, it is well-known that the standard Gauss–Seidel smoother is hard to parallelize.
- If $M_l = \texttt{block\_diag}(L_l)^{-1}$ and the two coefficients are 1, we have the block-Jacobi–Gauss–Seidel smoother, which is a hybrid of the block Jacobi and Gauss–Seidel methods. A lower triangular system needs to be solved, for each subproblem, too.
- If the inverse of A_l can be described by the Neumann polynomial [4],

  $$A_l^{-1} = \omega(I - G)^{-1} = \omega(I + G + G^2 + G^3 + G^m + \cdots), \tag{6}$$

  where ω is a scalar and $G = I - \omega A_l$, then we can define M_l as

  $$M_l = I + G + G^2 + G^3 + G^m. \tag{7}$$

  If the two coefficients α and β are 1, then we have a Neumann polynomial smoother.

We have implemented the Jacobi, damped Jacobi, block Jacobi, weighted Jacobi, block-Jacobi–Gauss–Seidel, and polynomial smoothers. For the block Jacobi, Gauss–Seidel, and block-Jacobi–Gauss–Seidel smoothers, triangular systems are required to be solved. The standard way is to solve the system row by row. Let us take a lower triangular system, $Lx = b$, as an example. The x_i is solved by the following formula:

$$x_i = \frac{1}{L_{ii}} \left(b_i - \sum_{j=1}^{i-1} L_{ij} x_j \right). \tag{8}$$

To obtain x_i, we have to know x_j $(1 \le j < i)$ if $L_{ij} \ne 0$ $(1 \le j < i)$. The standard method is to solve the system from the first row to the last row one by one, which introduces global dependence and is serial intrinsically. In this case, the parallelism is low. The upper triangular system is similar. For the above equation, x_i depends on x_j $(1 \le j < i)$ only if $L_{ij} \ne 0$ $(1 \le j < i)$. The dependence can be reduced by setting certain L_{ij} $(1 \le j < i)$ to zero by some mathematical techniques, such as the block Jacobi method, which drops the matrix entries that do not belong to the diagonal blocks. The parallelism can be increased by using smaller block sizes. A level schedule method is applied for the parallelization of triangular solvers [27,2,39,40]. The idea is to group all unknown components x_i into different levels so that each unknown within the same level can be computed simultaneously [2,4,39]. Since the solution of a lower triangular system and the solution of an upper triangular system are similar, the lower triangular system, $Lx = b$ $(L \in R^{n\times n})$, is studied. The level of x_i $(1 \le i \le n)$ is defined as

$$l(i) = 1 + \max_j l(j), \quad \text{for all } j \text{ such that } L_{ij} \ne 0 \ (j < i), i = 1, 2, \ldots, n, \tag{9}$$

where L_{ij} is the (i, j)th entry of L and $l(i)$ is zero initially. The level schedule method is described in Algorithm 1, where `nlev` is the number of levels and we assume there is an array that records level information of each unknown. The number of levels in Algorithm 1 depends on the structure of the lower triangular matrix L. The ideal case is that there is only one level and the solution process is fully parallelized. The worst scenario is that the number of levels equals the number of rows of matrix L; in this case, the solution process is completely serial.

Algorithm 1 Level schedule method for $Lx = b$

```
1: for $i = 1$ : nlev do
2:     start = level($i$);
3:     end = level($i + 1$) − 1;
4:     for $j$ = start: end do
5:         solve the $j$th row;
6:     end for
7: end for
```

The parallelization of the setup phase is challenging. The coarsening phase of the classical AMG introduced by Ruge–Stüben is sequential, which creates difficulties for GPU implementation. The structure of the coarser matrices and interpolation operators obtained on each level is irregular, which also introduces a barrier for GPU computing. Different algorithms have different properties, such as the RS and CLJP algorithms, and they need different parallelization strategies. The early work from NVIDIA shows that the setup phase can be accelerated around two times faster [31]. At this moment, we implement the setup phase on the CPUs and transfer the data to the GPUs. The data conversion and transfer introduce overhead. The setup phase is also a bottleneck of the whole AMG process. In the future, we will consider to accelerate the setup phase on the GPUs.

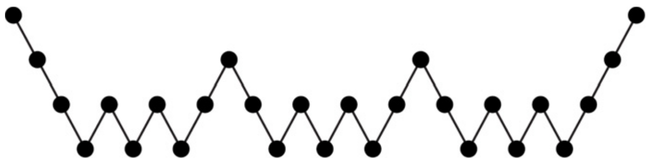

Fig. 2. W-cycle, 4 levels and $\gamma = 2$.

2.2. AMG solution phase

The solution phase of AMG is recursive and is formulated in Algorithm 2, which shows one iteration of AMG. On each level l, the algorithm requires A_l, R_l, P_l, S_l and T_l, where A_l, R_l and P_l are matrices, and the S_l and T_l are the pre-smoother and post-smoother, which are typically damped Jacobi, weighted-Jacobi, Gauss–Seidel, and block-Jacobi–Gauss–Seidel smoothers. The initial solution x is usually given by the user, and a regular choice is zero. In Algorithm 2, the initial guess for x_l is set to the zero vector.

Algorithm 2 AMG W-cycle Solution Algorithm: amg_solve(l, γ)

```
Require: $b_l, x_l, A_l, R_l, P_l, S_l, T_l, 0 \le l < L$

if ($l < L$) then
    $x_l = S_l(x_l, A_l, b_l)$                    ▷ Pre-smoothing
    $r = b_l - A_l x_l$
    $b_{l+1} = R_l r$                             ▷ Restriction
    for $i = 0; i < \gamma; i++$ do
        amg_solve($l + 1, \gamma$)
    end for
    $x_l = x_l + P_l x_{l+1}$                     ▷ Prolongation
    $x_l = T_l(x_l, A_l, b_l)$                    ▷ Post-smoothing
else
    $x_l = A_l^{-1} b_l$
end if
```

Remark. When the parameter γ is one, we have the standard V-cycle. Fig. 2 is the structure of a W-cycle with four levels and $\gamma = 2$ [2]. Fig. 1 is the structure of a V-cycle. From Fig. 2, we can see that the W-cycle spends most of the iterations on coarser grids. The W-cycle also has higher complexity than the V-cycle, and the complexity of the F-cycle is between them. During the solution process, the pre-smoothing and post-smoothing may be called more than once. In this paper, we use CPUs to setup the AMG system and use GPUs to solve it. When implementing the AMG solvers, they are designed to be used as the preconditioners of the Krylov subspace solvers, such as GMRES, BICGSTAB and ORTHOMIN.

3. GPU computing techniques

Algorithm 2 shows that most of the operations are matrix and vector operations as follows:

$$y = \alpha A x + \beta y, \tag{10}$$
$$z = \alpha A x + \beta y, \tag{11}$$
$$y = \alpha x + \beta y, \tag{12}$$
$$z = \alpha x + \beta y, \tag{13}$$
$$\alpha = \langle x, y \rangle. \tag{14}$$

The vector has a simple data structure, which is a floating point array. The sparse matrix is slightly more complex. Its data structure and sparse matrix–vector multiplication algorithm will be discussed in the following sections.

3.1. GPU computing overview

The NVIDIA GPUs are applied for our research platform. The processors of typical NVIDIA GPUs are organized into two levels: streaming multiprocessor (SM) and streaming processor (SP or CUDA core). Each GPU may have a different number of SMs. Depending on the architectures of the GPUs, each SM may have 32, 128 and 192 SPs (CUDA cores). The Tesla C2070 has 14 SMs and a total of 448 SPs. The Tesla K20X has 14 SMs (or SMXs) and each SM has 192 SPs, which has 2688 CUDA cores in total. The Tesla K40 has 15 SMs (or SMXs) and a total of 2880 CUDA cores.

The memory architecture of NVIDIA GPUs is hierarchical, including register, L1 cache, L2 cache, shared memory and global memory. The memory size of shared memory is small, and each multiprocessor owns a small portion of the shared memory, such as 48 kB. However, its speed is faster than the global memory and is used to communicate among threads in a block. The global memory stores our input data, such as matrices and vectors, and it is also used for data transfers between the host (CPUs) and the device (GPUs).

The CUDA threads are also organized into two levels: grid and block. If a grid has n_b blocks and each block has n_t threads, then it has $n_t \times n_b$ threads in total. The n_t is also called the block size. In CUDA, threads in a block are divided into warps, which have 32 threads. The threads in a warp are executed simultaneously. Usually, the block size, denoted by `blockDim.x`, is a multiple of 32, such as 128 and 256. In this paper, a two-dimensional grid is used and the default block size is 256. Each block in the grid has a unique block id, (`blockIdx.x`, `blockIdx.y`). Thread in each block is one dimensional, which has a unique id, `threadIdx.x`. Each thread in the grid also has a unique global thread id, t, which is one dimensional and is numbered block by block, from the first block to the last block. The global thread id is from 0 to $n_t \times n_b - 1$, which is defined as

$$t = \texttt{blockDim.x} * (\texttt{blockIdx.x} + \texttt{blockIdx.y} * \texttt{gridDim.x}) + \texttt{threadIdx.x}. \tag{15}$$

The performance of the AMG solvers is influenced by many factors. Here a simple performance model for the V-cycle AMG solver, which works on CPUs and GPUs, is provided to study these factors,

$$t_{solve} = \sum_{i=0}^{L-1} (t_{pre,i} + t_{post,i} + t_{r,i} + t_{res,i} + t_{pro,i}) + t_{c,L}, \tag{16}$$

where t_{solve} is the total running time of the solution phase on the devices (CPU or GPU); $t_{c,L}$ is the running time of the solution on the coarsest level, L; $t_{pre,i}$ is the pre-smoothing time for matrix A_i; $t_{post,i}$ is the post-smoothing time for matrix A_i; $t_{r,i}$ is the time for the calculation of residual r_i, $r_i = b_i - A_i x_i$; $t_{res,i}$ is the restriction operation time and $t_{pro,i}$ is the prolongation operation time. The total running time of the CPU-version AMG solvers is defined as

$$t = t_{setup} + t_{solve}, \tag{17}$$

where t_{setup} is the setup time on the CPU, and the total running time of the GPU-version AMG solvers is given as

$$t = t_{setup} + t_{solve} + t_{comm}, \tag{18}$$

where t_{setup} is the setup time on the CPU and t_{comm} is the communication time between CPUs and GPUs. The calculation of residual r and the restriction and prolongation operations are matrix–vector and vector operations. Iterative methods are used for the solution of the coarsest level, which are also combinations of matrix–vector and vector operations. From the discussions about the smoothers, we know that the matrix–vector and vector operations and the triangular solvers are used by these smoothers. We can see that the AMG solvers are IO intense applications. The memory is accessed when performing SpMV, vector and related operations. The potential speed-up of the GPU-version AMG solvers vs CPU-version AMG solvers is bounded by the ratio of GPU memory speed and CPU memory speed. For example, if the memory speed of a GPU is 300 GB/s and the memory speed of CPU is 30 GB/s, the potential maximal speed-up is around 10. If CPU memory with higher data transfer speed is used, such as 51.2 GB/s, the potential maximal speed-up is around 6.

Besides the memory speed of the GPUs, many factors also affect the performance of GPU applications. The NVIDIA GPUs have hierarchical memory architectures, including L1 cache and L2 cache. These memories are much faster than the global memory, and if the size of these memories is large, more data can be held, and the memory access will be more efficient. For the global memory, if the memory access is coalesced, the number of transactions can be minimized. If all threads in a warp execute the same instructions, there is no branch divergence problem, and in this case, the threads in the warp do not need to wait for each other and the warp has the best efficiency. In the design of a GPU algorithm, we need to make sure all threads in a warp follow the same execution path and minimize the branch divergence. In the GPUs, the warp scheduling also introduces overhead, and it is well-known that when the problem size is small, GPU applications may be slower than CPU applications. All these factors influence the GPU performance and time terms above. In this paper, since the setup phase is performed on the CPUs, the setup of the AMG solvers, the data conversion and the data transfer introduce overhead. They reduce the potential maximal speed-up of the AMG solvers.

The sparsity of the given matrix A and the choices of algorithms also affect the potential speed-up, especially the choices of smoothers. The Jacobi, damped Jacobi, weighted Jacobi and polynomial smoothers are combinations of SpMV and vector operations, which have been well studied and are easy to parallelize. The speed-up of these smoothers is bounded by the ratio of GPU memory speed and CPU memory speed. However, the block Jacobi, Gauss–Seidel, and block-Jacobi–Gauss–Seidel are difficult to parallelize on the GPUs, since triangular systems are required to be solved. As we discussed above, the solution of triangular systems is serial intrinsically. If the level scheduling method for the solution of triangular systems has too many levels, the smoothers may not have enough parallelism, and the speed-up of the GPU-version AMG solvers may be low. The time terms, $t_{pre,i}$ and $t_{post,i}$, on the GPUs and potential speed-up for these three smoothers are affected by non-zero patterns of matrices on each level and the choice of block sizes, which are hard to predict.

3.2. Sparse matrix–vector multiplication

To speed up the sparse matrix–vector multiplication operation, we use a hybrid matrix format HEC shown in Fig. 3 [26]. An HEC matrix consists of two parts: ELL (Ellpack) matrix and CSR (Compressed Sparse Row) matrix. The ELL matrix is regular

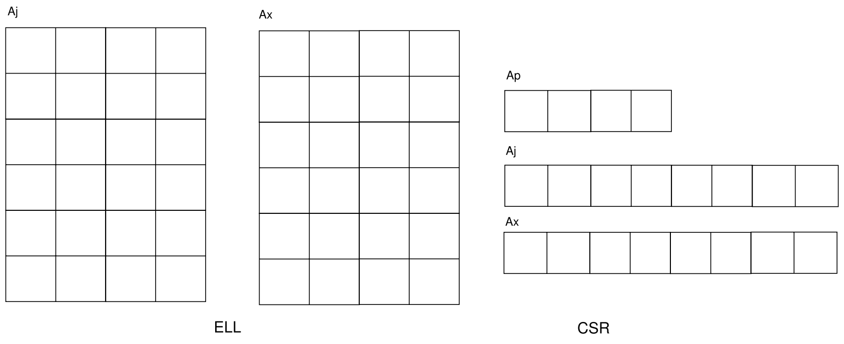

Fig. 3. HEC matrix format.

```
typedef mat_csr_t_
{
    FLOAT *Ap;
    FLOAT *Aj;
    FLOAT *Ax;
    INT num_rows;
    INT num_cols;
    INT num_nonzeros;
} mat_csr_t;
```

Fig. 4. Data structure of CSR matrix.

```
typedef mat_ell_t_
{
    FLOAT *Aj;
    FLOAT *Ax;
    INT num_rows;
    INT num_cols;
    INT num_nonzeros;
    INT stride;
} mat_ell_t;
```

Fig. 5. Data structure of ELL matrix.

and each row has the same length. The problem for the ELL matrix is that if one row is too long, the other rows have the same length and a lot of memory is wasted. The HEC matrix uses an ELL matrix to store part of each row and uses a CSR matrix to store the irregular part [26]. The HEC format is general, which is suitable for regular sparse matrices and dense matrices.

In this paper, the FLOAT is single precision or double precision floating point type, and INT is integer type. The data structures of a CSR matrix, ELL matrix and HEC matrix are represented in Figs. 4–6, respectively. The num_rows, num_cols and num_nonzeros mean number of rows, columns and non-zero values for a matrix. For a CSR matrix, the array Ap stores the starting locations of each row, the Aj stores the column indices of each non-zero entry and Ax stores the value of each entry. For an ELL matrix, only two arrays are required, Aj and Ax, which store the column indices and the values of non-zero entries. For GPU computing, the ELL matrix is stored in column-major order and each column of the ELL matrix is aligned such that the column length is a multiple of 32. The real column length is noted by stride, which is calculated by

$$s = \texttt{floor}((n + l_s - 1)/l_s) \times l_s, \tag{19}$$

where l_s ($l_s > 0$) is a multiple of 32, such as 32 and 64, and n is the number of rows of the given matrix. In this paper, l_s equals 32. In this case, the global memory access for the ELL matrix is coalesced. An HEC matrix is a mixture of the ELL matrix and the CSR matrix.

```
typedef mat_hec_t_
{
    mat_csr_t csr;
    mat_ell_t ell;
    INT num_rows;
    INT num_cols;
    INT num_nonzeros;
} mat_hec_t;
```

Fig. 6. Data structure of HEC matrix.

When converting a given matrix A to its HEC matrix, the row length of the ELL matrix is calculated by solving a minimum problem [22]:

$$\text{find } l \ (l \geq 0) \text{ such that } w(i) = i * n + p_r * nz(i) \text{ is minimized,} \tag{20}$$

where p_r is the relative performance of the ELL and CSR matrices and $nz(i)$ is the number of non-zeros in the CSR part when the row length of the ELL part is i. A typical value for p_r is 20 [22,26].

Algorithm 3 Sparse Matrix–Vector Multiplication, $y = Ax$

```
for $i = 1 : n$ do                                      ▷ ELL, Use one GPU kernel to deal with this loop
    the $i$th thread calculates the $i$th row of ELL matrix;                     ▷ Use one thread
end for

for $i = 1 : n$ do                                      ▷ CSR, Use one GPU kernel to deal with this loop
    the $i$th thread calculates the $i$th row of CSR matrix;                     ▷ Use one thread
end for
```

A sparse matrix–vector multiplication kernel [26] is developed as shown in Algorithm 3. The ith thread calculates the ith row. The ELL part is calculated first and the CSR part is then processed. Since the columns of the ELL matrix are aligned, when the ELL part is calculated, the global memory access for the matrix is coalesced.

3.3. Data structures of the AMG solvers

The AMG solvers have similar data structures on each level. Each level has a right-hand side b_l and a solution x_l. On the CPU, each prolongation operator and restriction operator are CSR matrices. On the GPU, each prolongation operator and restriction operator are HEC matrices, which are converted from the CSR matrices. In this case, the data structures are independent of the coarser grid and prolongation algorithms. However, the size of each matrix depends on the original matrix and the choices of AMG algorithms, such as coarsening algorithms and interpolation algorithms.

The data structure of the CPU-version smoother is shown in Fig. 7, which has three matrices, a vector and several parameters. `A` is a reference to the coarser matrix on the level. The matrices `S` and `N` and vector `D` are used to describe smoothers. The Jacobi smoother can be described by a diagonal matrix or a vector `D`, and the Gauss–Seidel smoother can be described by a lower triangular matrix. The LU factorization family smoothers can be described by a lower triangular matrix and an upper triangular matrix. `alpha`, `beta` and `damping` are related parameters, such as a damping coefficient, and `sm_type` is the smoother type. The data structure of the GPU-version smoother is presented in Fig. 8. The data structure `amg_sm_gpu` is converted from the structure `amg_sm_cpu`.

4. Numerical experiments

In this section, numerical experiments are carried out to test AMG solvers. The experiments are performed on our workstation with Intel Xeon X5570 CPU and NVIDIA Tesla C2050/C2070 GPUs. The operating system is CentOS X86_64 with CUDA Toolkit 5.1 and GCC 4.4. All CPU codes are compiled with -O3 option. All calculations use double precision. The CPU memory in our system is DDR3-10600, which runs at a frequency 1333 MHz and has a peak transfer rate at 10.6 GB/s. The communications between CPU and GPU are through a PCIe interface, which has a peak transfer rate of 8 GB/s. The memory speed of the GPUs is 144 GB/s.

In this paper, we are interested in the speed of the GPU-version AMG solvers and the experiments are designed to test the performance of GPU-version AMG solvers vs that of CPU-version AMG solvers. The number of maximal iterations is set as 100. The matrices used in this section are listed in Table 1. The `af_shell8`, `parabolic_fem` and `atmosmodd` are from Matrix UF Sparse Matrix Collection [41]. The `af_shell8` is from a structural problem and it is positive definite.

```
typedef amg_sm_cpu_
{
    mat_csr_t A;
    mat_csr_t S;
    mat_csr_t N;
    FLOAT *D;

    FLOAT alpha;
    FLOAT beta;
    FLOAT damping;

    INT sm_type;
} amg_sm_cpu;
```

Fig. 7. Data structure of the smoother on the CPU.

```
typedef amg_sm_gpu_
{
    mat_hec_t A;
    mat_hec_t S;
    mat_hec_t N;
    FLOAT *D;

    FLOAT alpha;
    FLOAT beta;
    FLOAT damping;

    INT sm_type;
} amg_sm_gpu;
```

Fig. 8. Data structure of the smoother on the GPU.

Table 1
Matrices for numerical experiments.

Matrix	# of Rows	Non-zeros
af_shell8	504,855	9,046,865
parabolic_fem	525,825	2,100,225
atmosmodd	1,270,432	8,814,880
SPE10	2,188,851	29,915,573
P2D5P_1000	1,000,000	4,996,000
P3D7P_100	1,000,000	6,940,000
P3D7P_110	1,331,000	9,244,400
P3D7P_120	1,728,000	12,009,600
P3D7P_130	2,197,000	15,277,600

The parabolic_fem is from a diffusion–convection reaction problem and the matrix is symmetric positive definite. The atmosmodd is an atmospheric model. The SPE10 is from a black oil problem [15] and the grid is structured. The grid dimension is $60 \times 220 \times 85$ and no grid refinement is applied. The P2D5P_1000 is from a two-dimensional Poisson equation. Its grid is structured and its dimension is 1000×1000. The P3D7P_100, P3D7P_110 P3D7P_120 and P3D7P_130 are from three-dimensional Poisson equations, and their grid dimensions are $100 \times 100 \times 100$, $110 \times 110 \times 110$, $120 \times 120 \times 120$, and $130 \times 130 \times 130$, respectively.

Table 2
$y = \alpha x + \beta y$.

Size	CPU (s)	GPU speed-up	IO (GB/s)
10,000	8.05e−6	1.99	55.19
100,000	9.87e−5	4.61	104.44
1,000,000	2.01e−3	10.73	119.24
2,000,000	4.28e−3	11.53	120.60
10,000,000	1.99e−2	10.80	121.52

Table 3
Dot product.

Size	CPU (s)	GPU speed-up	IO (GB/s)
10,000	9.21e−6	1.13	18.34
100,000	9.18e−5	2.68	59.67
1,000,000	1.69e−3	12.47	110.17
2,000,000	3.74e−3	14.01	111.50
10,000,000	1.60e−2	12.65	117.90

Table 4
Vector transfer, GPU to CPU.

Size	Time (s)	IO (GB/s)
10,000	6.59e−5	1.21
100,000	5.72e−4	1.40
1,000,000	3.64e−3	2.19
2,000,000	7.27e−3	2.20
10,000,000	3.35e−2	2.39

Table 5
Vector transfer, CPU to GPU.

Vector	Size (MB)	Time (s)	Speed (GB/s)
100	7.63e−4	2.38e−6	0.31
10,000	7.63e−2	2.15e−5	3.46
250,000	1.91	4.90e−4	3.79
1,000,000	7.63	2.08e−3	3.58
2,250,000	17.17	4.32e−3	3.87
4,000,000	30.52	8.28e−3	3.60
9,000,000	68.66	1.87e−2	3.58

4.1. *Vector operation performance*

The vector operation performance is tested, including $y = \alpha x + \beta y$ and dot product, where α and β are 1.1 and 0.2. The calculation of each component of y includes two multiplications, one addition and one integer operation to calculate the thread id, and it also has two memory reading and one memory writing. The results are presented in Tables 2 and 3, where CPU time, GPU speed-up, and global memory performance are recorded. For GPU, the time that is used to calculate the memory speed and speed-up includes memory access time and calculating time. The CPU execution is sequential.

From Tables 2 to 3, we can see that when the size of vectors is large enough, a maximal speed-up of 14 can be obtained. The vector operations are IO intense algorithms, and their performance is bounded by IO bandwidth. The results show that the speed-up of GPU-version vs CPU-version is bounded by the ratio of GPU memory speed and CPU memory speed, which is around 14 for our workstation. However, if a newer workstation is applied, such as a system with NVIDIA Tesla K40 (memory speed 288 GB/s) and DDR4-3200 CPU memory (memory speed 51.2 GB/s), a maximal speed-up of six can be expected.

The communications between CPU and GPU are through a PCIe interface, which has a peak transfer rate of 8 GB/s. Pageable memory that is allocated by `cudaMalloc` has a peak transfer rate of 4+ GB/s using PCIe x16 Gen2 [42]. Tables 4 and 5 show the data transfer performance for different vector sizes between CPU and GPU, which indicate the IO speed is high. Since we are using PCIe 2.0, if a new interface is applied, such as PCIe 3.0, the bandwidth will be much higher.

4.2. *SpMV performance*

Sparse matrix–vector multiplication (SpMV) is one of the most important operations for AMG methods. This section compares the performance of different SpMV kernels, including CSR-S, CSR-V, ELL, HYB, and HEC kernels.

The CSR-S is a scalar version of SpMV for a CSR format matrix. For the sparse matrix–vector multiplication $y = Ax$, each component, $y[i]$, of the y is calculated by one thread, which loops one row of the CSR matrix and calculates the result.

Table 6
SpMV performance.

Matrix	CPU (s)	CSR-S	CSR-V	ELL	HYB	HEC
af_shell8	2.04e−2	1.72	4.08	10.05	11.23	12.64
parabolic_fem	5.75e−3	5.34	1.12	9.53	9.49	9.41
P2D5P_1000	1.37e−2	7.71	1.36	19.18	18.99	18.96
P3D7P_130	3.91e−2	4.60	1.67	19.10	19.08	19.12
atmosmodd	1.86e−2	3.85	1.39	14.41	14.39	14.43
SPE10	7.20e−2	2.15	2.51	1.83	14.73	14.81

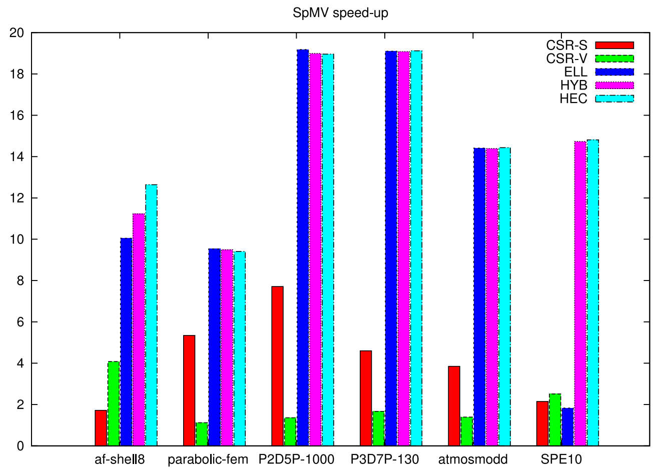

Fig. 9. SpMV performance.

The CSR-V is a vector version of the SpMV kernel for a CSR format matrix. For SpMV $y = Ax$, each component, $y[i]$, of y is calculated by a warp of 32 threads, and a warp processes each row of the matrix A. The ELL kernel is the SpMV kernel for an ELL format matrix. The HYB is the SpMV kernel for an HYB format matrix, which is similar to HEC format and contains two matrices: an ELL matrix for the regular part of a given matrix and a COO matrix for the irregular part of a given matrix. The HYB format is also a general format for arbitrary matrices. The SpMV algorithm designed by Bell et al. calculates the ELL matrix at first and then the COO matrix is calculated. The HEC is the SpMV kernel for HEC format matrix. The CSR-S, CSR-V, ELL and HYB kernels are included in the cuSPARSE library from NVIDIA. These kernels are applied to compare with our SpMV kernel. The CPU version is implemented and optimized in our implementations. The performance summaries are in Table 6, which includes CPU running time and the speed-ups of different SpMV kernels, and the speed-up is also shown by Fig. 9. The CPU execution of the SpMV is sequential.

From Table 6 and Fig. 9, we can see that the ELL, HYB and HEC kernels have the best performance and GPUs can speed the SpMV operation up to 19 times faster. The ELL is suitable for a matrix whose row lengths are similar. When one row is much longer than others, the ELL format will require too much memory and calculations, such as in the case of the SPE10 matrix. The HYB matrix and the HEC matrix are general and they are suitable for arbitrary sparse matrices. The table also shows that the HEC kernel has slightly better performance than the HYB kernel. The results also show that the maximal speed-up of GPU-version SpMV vs CPU-version SpMV is bounded by the ratio of memory speed and depends on the hardware.

4.3. *Performance of AMG components*

The AMG solvers have different phases, such as the setup phase and the solution phase. The GPU AMG solvers also have data conversion and data transfer. The AMG solvers have several components, such as coarsening, restriction, interpolation and smoothing. This section tests the performance of these phases and components.

Example 4.1. The matrix af_shell8 is tested. The stopping criteria is 1e−6 and the maximal iterations are 100. The maximal level is 8. The number of pre-smoothing and post-smoothing is 3. The standard RS interpolation and the V-cycle are applied. The RS coarsening strategy and the CLJP coarsening strategy are tested. The numerical results are shown in Tables 7–16. The CPU execution is sequential.

Table 7 shows that the setup phase (Setup) and the solution phase (Solve-C) on CPU have similar running time, the data conversion (CSR matrix to HEC matrix) and transfer time (Data-G), including all coarser matrices, interpolation and restriction matrices, and smoothers, is 0.43 s, and the solution time on GPU (Solve-G) is 0.641 s. The solution phase has a speed-up of 5.57 on our workstation. However, the results indicate that the setup phase occupies around 50% of the total

Table 7
Performance of different phases of AMG solver, CLJP strategy.

Setup (s)	Solve-C (s)	Data-G (s)	Solve-G (s)	# Itr	Speed-up
3.515	3.572	0.43	0.641	4	5.57

Table 8
Performance of different components of AMG setup phase, CLJP strategy.

Grid (s)	Interpolation (s)	Matrix (s)
0.443	0.431	2.267

Table 9
Performance of components for AMG solution phase 4.1, CLJP/.

	# CPU time (s)	# GPU time (s)	# Speed-up
Smoothers	7.275e−1	1.1868e−1	6.13
Prolongation	1.082e−2	1.488e−3	7.27
Restriction	1.239e−2	1.228e−3	10.09
Coarsest level solution	2.928e−1	6.892e−2	4.25

Table 10
Size of coarser matrices at different levels, CLJP strategy.

Level	# Row	# Non-zero	# Avg.	% (Non-zero)
0	504,855	9,042,005	17.91	3.548e−3%
1	337,102	14,542,444	43.14	1.28e−2%
2	223,629	11,899,811	53.21	2.38e−2%
3	156,184	10,008,436	64.08	4.10e−2%
4	113,827	8,173,078	71.80	6.31e−2%
5	87,121	6,550,974	75.19	8.63e−2%
6	69,197	5,203,771	75.20	1.09e−1%
7	56,881	4,176,282	73.42	1.29e−1%

Table 11
Size of interpolators at different levels, CLJP strategy.

Level	# Row	# Col	# Non-zero	# Avg.	% (Non-zero)
1	504,855	337,102	1,131,670	2.24	6.65e−04%
2	337,102	223,629	581,515	1.725	7.713855e−04%
3	223,629	156,184	354,715	1.586	1.015582e−03%
4	156,184	113,827	222,304	1.423	1.250447e−03%
5	113,827	87,121	149,495	1.313	1.507504e−03%
6	87,121	69,197	107,882	1.238	1.789530e−03%
7	69,197	56,881	81,684	1.180	2.075308e−03%

running time on CPU, the data conversion and transfer is also a bottleneck of the AMG process, which is clearly shown in Fig. 10, and the whole setup phase should be performed on the GPUs to utilize the power of GPUs and to avoid data transfer. Since the parallelization of GPU is different from the traditional CPU-based parallel computing, such as OpenMP and MPI based parallel applications, the AMG algorithms may be required to be modified such that GPU threads can work simultaneously and have high parallelism. Besides the coarsening algorithms and interpolations, the incomplete LU factorization and parallel triangular solvers, which are required by the Gauss–Seidel smoother, are required to be parallelized. If the setup phase is on the GPU, the intermediate communications and conversions can be avoided. In this case, communications only occur at the beginning and end of the AMG process to send a matrix and a few vectors. From the previous communication tests, we can see that the communications between CPU and GPU are efficient, which only have a small influence on the performance of the AMG solvers.

Table 8 shows the performance of the CLJP coarsening algorithm, the standard RS interpolation and the creation of a coarser matrix by matrix–matrix multiplication, $A_c = RA_f P$. The results show that the times for the coarsening process and generation of interpolation are similar, but the sparse matrix–matrix multiplication dominates the setup phase.

The results from Table 9 and Fig. 11 show time for one iteration of the V-cycle, which indicate that the smoother dominates the whole running time, and the prolongation and the restriction operations occupy only a small portion of the total running time. The damped Jacobi preconditioner is a vector operation, and the interpolation and the restriction operations are SpMV operations. All components are well accelerated in this example.

Table 12
Performance of different phases of AMG solver, RS strategy.

Setup (s)	Solve-C (s)	Data-G (s)	Solve-G (s)	# Itr	Speed-up
2.094	1.579	0.169	0.273	4	5.79

Table 13
Performance of different components of AMG setup phase, RS strategy.

Grid (s)	Interpolation (s)	Matrix (s)
0.134	0.36	1.402

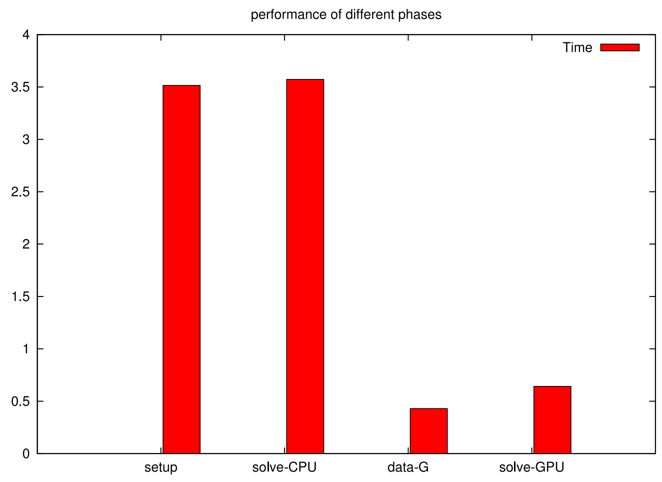

Fig. 10. Performance of different phases.

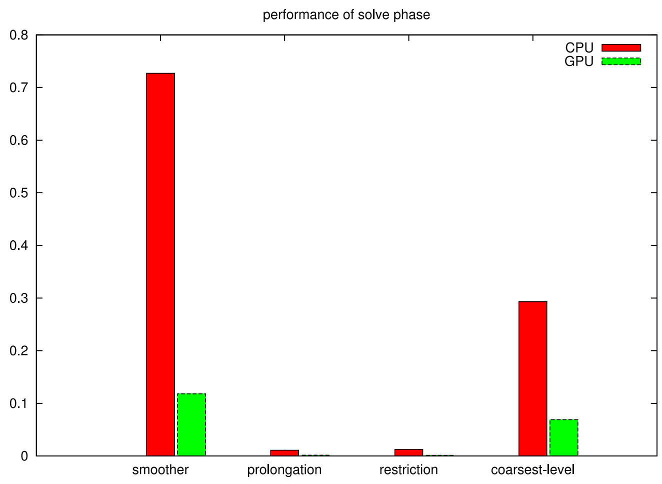

Fig. 11. Performance of different components of V-cycle.

Tables 10 and 11 show the matrix sizes of the coarser matrices and interpolation matrices on each level. For now, they are required to convert to HEC format and transfer to the GPUs. Since the restriction matrix is the transpose of an interpolation matrix, it has the same size as the interpolation matrix. The numbers of rows, columns, non-zeros and average non-zeros per row are kept as well as the percentage of the non-zeros. The interpolation matrices have very few non-zero entries per row, fewer than three entries on average. However, the coarser matrices become denser and denser.

From Table 12, we can see that the results for the RS coarsening strategy are similar to those for the CLJP coarsening strategy, in which the setup phase uses over 50% of the total running time. Since it is performed on CPU, it dominates the whole AMG process. The GPU-based setup phase should be developed to accelerate the AMG solvers. In this example, the solution phase is well accelerated and a speed-up of 5.79 is obtained on our workstation. Again, for the setup phase on CPU, the creation of coarser matrices dominates the whole setup, which is shown in Table 13.

Table 14
Performance of components for AMG solution phase 4.1, RS strategy.

	# CPU time (s)	# GPU time (s)	# Speed-up
Smoothers	3.045e−1	4.991e−2	6.10
Prolongation	8.743e−3	1.155e−3	7.57
Restriction	7.348e−3	1.28e−3	5.74
Coarsest level solution	6.01e−4	4.276e−3	0.14

Table 15
Size of coarser matrices at different levels, RS strategy.

Level	# Row	# Non-zero	# Avg.	% (Non-zero)
0	504,855	9,042,005	17.91	3.547574e−03%
1	216,461	10,677,187	49.33	2.278754e−02%
2	100,146	7,194,365	71.84	7.173403e−02%
3	35,160	2,271,442	64.60	1.837401e−01%
4	12,671	640,408	50.54	3.988733e−01%
5	4,478	165,296	36.91	8.243168e−01%
6	1,587	39,954	25.176	1.586377%
7	559	9,625	17.22	3.080187%

Table 16
Size of interpolators at different levels, RS strategy.

Level	# Row	# Col	# Non-zero	# Avg.	% (Non-zero)
1	504,855	216,461	1,763,938	3.49	1.614124e−03%
2	216,461	100,146	647,688	2.99	2.987807e−03%
3	100,146	35,160	229,343	2.29	6.513329e−03%
4	35,160	12,671	56,667	1.61	1.271951e−02%
5	12,671	4,478	17,206	1.35	3.032389e−02%
6	4,478	1,587	5,390	1.20	7.584514e−02%
7	1,587	559	1,833	1.155	2.066207e−01%

Table 15 shows that the coarser matrices become denser and denser. Table 16 shows that the interpolations are sparse, which have fewer than four non-zero entries per row.

4.4. *Performance of multi-threaded AMG solver*

This case enables the OpenMP support of our CPU codes. All the matrix–vector operations and vector operations are parallelized using OpenMP. In addition, up to 16 threads are employed to test the performance of a CPU-based AMG solver. The case and the settings are the same as in Example 4.1. The numerical summaries are in Table 17.

Table 17 shows when the number of threads increases from one to four, the performance of the CPU-version AMG solver increases, 57%. If more threads are employed, the performance decreases. However, in all cases, the GPU-version AMG solver shows better performance than the CPU-version AMG solver.

4.5. *Smoothed aggregation AMG solver*

This section studies the smoothed aggregation AMG solver using a different number of levels, cycles and smoothers. The default damping parameter α of smoothers is 0.8.

4.5.1. *Levels*

Example 4.2. The matrix is P2D5P_1000. The damped Jacobi smoother is applied. The number of pre-smoothing and post-smoothing is 6. The termination criterion is 1e−6. V-cycle is applied. Numerical results are shown in Table 18, where the CPU execution is sequential.

This example tests the smoothed aggregation AMG solver performance with a different number of levels. The number of levels, the number of AMG iterations, GPU time, CPU time and speed-up are collected. From Table 18, we can see that when 4 levels are applied, the GPU-version AMG solver is 9.52 times faster than the CPU-version AMG solver. When increasing the number of levels to 5, a speed-up of 9.48 is obtained. When using 6 levels, the speed-up of GPU-version AMG solver is 9.45 and when using 7 levels, the speed-up is 9.34. We also observe that by increasing the number of levels, the solver uses fewer iterations and less computation time. It means that the number of levels of an AMG solver should not be too small. The reason that the number of levels is not greater than 7 is that the size of the coarsest grid is too small and there is no need to increase the number of levels. The example shows that the solution phase of our GPU-version AMG solver is over 9 times faster than the CPU-version AMG solver on our workstation.

Table 17
Summary of Example 4.1, with different threads.

# Threads	# CPU time (s)	# GPU time (s)	# Speed-up
1	1.673	0.1671	9.69
2	1.131	0.1670	6.55
3	1.094	0.1672	6.31
4	1.065	0.1673	6.16
8	1.101	0.1689	6.31
16	1.141	0.1676	6.59

Table 18
Summary of Example 4.2, with different levels.

# Level	# Iter	# CPU time (s)	# GPU time (s)	# Speed-up
4	100	25.17	2.643	9.52
5	25	6.369	0.671	9.48
6	26	6.229	0.659	9.45
7	26	6.257	0.670	9.34

Table 19
Summary of Example 4.3, with different cycles.

Cycles	# Iter	# CPU time (s)	# GPU time (s)	# Speed-up
W	20	11.03	1.61	6.85
F	25	13.38	1.60	8.36
V	61	24.71	2.50	9.88

Table 20
Summary of Example 4.4, with different smoothers.

Smoothers	# Iter	# CPU (s)	# GPU (s)	# Speed-up
dJacobi	71	58.29	5.34	10.92
wJacobi	67	62.60	5.31	11.78
bJacobi (8)	62	69.33	7.59	9.13
bJacobi (32)	59	74.14	9.29	7.98
bJacobi (128)	55	62.98	13.70	4.60
bJacobi (1024)	52	62.76	27.54	2.28
bJGS (8)	63	59.87	6.22	9.62
bJGS (32)	62	51.16	7.22	7.08
bJGS (128)	60	57.12	9.69	5.89
bJGS (1024)	59	57.40	17.90	3.21
GS	45	52.17	51.51	1.01
Chev	58	42.36	3.78	11.2

4.5.2. *Multigrid cycles*

Two matrices are applied here to study the performance of different solution cycles. The F-cycle, the W-cycle and the V-cycle are studied.

Example 4.3. The matrix is P3D7P_110. The damped Jacobi smoother is applied. The number of pre-smoothing and post-smoothing is 6. The termination criterion is 1e−6. The maximal level is 7. The γ for W-cycle is two. Numerical results are shown in Table 19 and single core CPU execution is employed.

From Table 19 we can see that the V-cycle has the best speed-up and the W-cycle has the worst speed-up. The V-cycle is 9.88 times faster on GPUs than that on CPUs. The W-cycle is 6.85 times faster and the F-cycle AMG solver is 8.36 times faster on GPUs than those on CPUs on our workstation. As mentioned above, the W-cycle spends most of iterations on coarser grids, whose sizes are smaller than those of finer grids, while GPUs tend to have higher speed-up when they have a higher volume of workloads. Since the W-cycle has more iterations than the V-cycle and F-cycle for the same number of levels, the W-cycle has the best convergence and the V-cycle has the worst convergence in this example.

4.5.3. *Smoothers*

Example 4.4. The matrix is P3D7P_130. The number of pre-smoothing and post-smoothing is 6. The termination criterion is 1e−6. The maximal level is 7. V-cycle is applied. Numerical results are shown in Table 20 and speed-up of different smoothers are demonstrated by Fig. 12. The CPU execution is sequential.

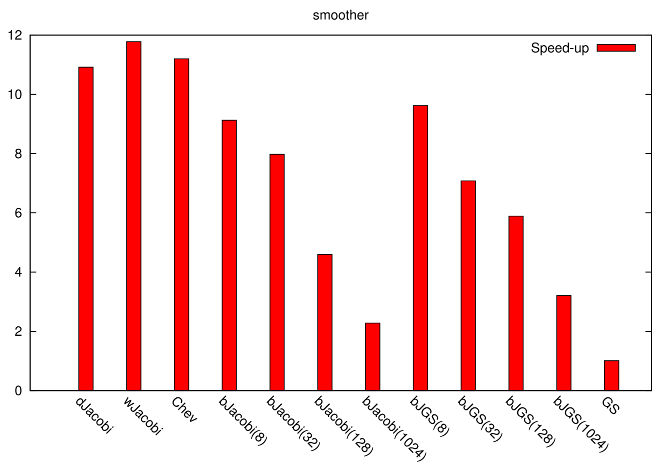

Fig. 12. Speed-ups of different smoothers.

Six smoothers are studied, including the damped Jacobi (`dJacobi`), weighted Jacobi (`wJacobi`), block Jacobi (`bJacobi`), block-Jacobi–Gauss–Seidel (`bJGS`), Gauss–Seidel (`GS`) and polynomial (`Chev`) smoothers. The damped Jacobi, weighted Jacobi and polynomial smoothers are naturally parallel and they show good speed-ups in this example, which are around 11. For the block Jacobi, block-Jacobi–Gauss–Seidel and standard Gauss–Seidel smoothers, a lower triangular linear system and/or an upper triangular linear system are required to be solved. As discussed above, data dependence exists and the parallelism of triangular solvers is limited by their matrix structures. For these smoothers, if their block sizes are small, they have high parallelism; in this case, the GPU-version triangular solver intends to have high speed-up. In this example, block sizes with 8, 32, 128 and 1024 are tested for the block Jacobi and block-Jacobi–Gauss–Seidel smoothers. The Gauss–Seidel smoother is a special case of the block-Jacobi–Gauss–Seidel smoother when the number of blocks is one. For the block Jacobi and block-Jacobi–Gauss–Seidel smoothers, we can see from Table 20 that their speed-ups decrease by increasing the block sizes, which are also clearly shown in Fig. 12. The speed-up of the block Jacobi smoother decreases from 9.13 to 2.28, and the speed-up of the block-Jacobi–Gauss–Seidel smoother decreases from 9.62 to 3.21. The Gauss–Seidel has the worst speed-up, which is only 1.01. The results coincide with the discussions about performance and potential speed-up. The results also show that the AMG solvers have better convergence if the block size is increased.

4.6. Classical AMG solver

In this section, we study the classical AMG solver with different components, such as coarsening algorithms, interpolation operators and the number of grid levels.

4.6.1. Levels

Example 4.5. The matrix is `P2D5P_1000`. The damped Jacobi smoother is applied. The number of pre-smoothing and post-smoothing is 6. The termination criterion is 1e−6. The interpolation operator is the standard RS operator and the coarsening algorithm is the RS method. The V-cycle is applied. Numerical results are shown in Table 21. The CPU execution is sequential.

The results in Table 21 show that the solution phase of GPU-version classical AMG solver is over 9.5 times faster than the CPU-version AMG solver. When using 4 levels, the speed-up is 10.32. The GPU performance is good; however, the AMG fails to solve. When using 5 levels, the GPU-version AMG solver is 9.66 times faster than the CPU-version AMG. The results of 6 levels and 7 levels are similar to the results of 5 levels. We can see that the solution phase of the classical AMG solver on GPUs is much faster than that on CPUs on our workstation. For the convergence, this example also suggests that higher levels should be considered in real applications.

4.6.2. Multigrid cycles

Example 4.6. The matrix is `af_shell8`. The damped Jacobi smoother is applied. The number of pre-smoothing and post-smoothing is 6. The termination criterion is 1e−6. The interpolation operator is the standard RS operator and the coarsening algorithm is the RS strategy. The maximal level is 8. The γ for the W-cycle is two. Numerical results are shown in Table 22. The CPU execution is sequential.

The matrix `af_shell8` is a small matrix compared with other matrices. The speed-ups are relatively low compared with Example 4.3. The V-cycle has the best speed-up, 7.09. The W-cycle has the least speed-up 2.09. The F-cycle is in the middle, and its speed-up of 5.31 is observed. Again, since the W-cycle spends most of its iterations on coarser grids, it shows the worst speed-up in this example.

Table 21
Summary of Example 4.5, with different levels.

# Level	# Iter	# CPU time (s)	# GPU time (s)	# Speed-up
4	100	54.52	5.283	10.32
5	13	6.07	0.628	9.66
6	5	2.256	0.236	9.54
7	5	2.245	0.236	9.51

Table 22
Summary of Example 4.6, with different cycles.

Cycles	# Iter	# CPU time (s)	# GPU time (s)	# Speed-up
W	2	3.965	1.897	2.09
F	2	2.423	0.456	5.31
V	2	1.245	0.176	7.09

Table 23
Summary of Example 4.7, with different coarsening strategies.

Coarsening	# Iter	# CPU time (s)	# GPU time (s)	# Speed-up
RS	6	2.92	0.25	11.68
CLJP	13	8.48	1.15	7.39

Table 24
Summary of Example 4.8, with different interpolators.

Interpolation	# Iter	# CPU time (s)	# GPU time (s)	# Speed-up
Standard RS	5	3.35	0.364	9.18
Direct RS	6	3.42	0.342	10.0
Haase	15	8.58	0.87	9.85
Multiple Pass	6	3.56	0.34	10.43

4.6.3. Coarsening strategies

Example 4.7. The matrix is P2D5P_1000. The damped Jacobi smoother is applied. The number of pre-smoothing and post-smoothing is 6. The termination criterion is 1e−6. The interpolation operator is the standard RS operator. The maximal level is 12. V-cycle is applied. Numerical results are shown in Table 23. The CPU execution is sequential.

The RS strategy and the CLJP strategy are studied and from Table 23, we can see that the RS strategy shows better convergence and performance than the CLJP strategy for the matrix P2D5P_1000. The RS strategy uses less CPU time and GPU time than the CLJP strategy. On our workstation, the AMG solver with the RS strategy has a speed-up of 11.68 and a speed-up of 7.39 for the CLJP strategy. In this example, the RS strategy shows better convergence than the CLJP strategy.

4.6.4. Interpolation operators

Example 4.8. The matrix P3D7P_100. The damped Jacobi smoother is applied. The number of pre-smoothing and post-smoothing is 6. The termination criterion is 1e−6. The RS coarsening method is employed. The maximal level is 8. V-cycle is applied. Numerical results are shown in Table 24. The CPU execution is sequential.

Four interpolation operators and their corresponding results are shown in Table 24. For this matrix, the standard RS interpolation operator has the best convergence, which converges in 5 iterations. The direct RS interpolation and the multiple pass interpolation have similar performance. The interpolation operator in Hasse's paper is the easiest to set up, among the operations considered in this paper; however, its convergence is the worst in this example, which uses 15 iterations. All cases have good GPU performance and the speed-up for them is more than 9 on our workstation.

4.7. Combinations

This section tests various combinations of different AMG components. The matrix af_shell8 is tested. The stopping criteria is 1e−6 and the maximal iterations are 100. The maximal level is 8. The number of pre-smoothing and post-smoothing is 3. The numerical results are shown in Table 25, where CS, Interp and SM mean coarsening strategy, interpolation and smoother, respectively. The CPU time, GPU time and speed-up are reported. The CPU execution is sequential.

Two coarsening strategies, the Ruge–Stüben (RS)and CLJP coarsening strategies, four interpolations, including the standard RS (RSSTD), direct (RSD), multiple pass (MPRS), and ML interpolations, and several smoothers, including the damped

Table 25
Numerical summaries of different combinations for `af_shell8`.

CS	Interp	SM	# Iter	# CPU (s)	# GPU (s)	# Speed
CLJP	RSSTD	dJacobi	3	3.572	0.641	5.57
CLJP	RSSTD	wJacobi	7	8.096	1.507	5.37
CLJP	RSSTD	bJacobi(8)	3	3.97	0.754	5.27
CLJP	RSSTD	bJacobi(32)	3	4.606	0.961	4.79
CLJP	RSSTD	bJacobi(256)	3	4.911	2.75	1.79
CLJP	RSSTD	bJGS(8)	3	3.718	0.717	5.18
CLJP	RSSTD	bJGS(32)	3	4.174	0.846	4.94
CLJP	RSSTD	bJGS(256)	3	4.396	1.813	2.42
CLJP	RSSTD	GS	1	1.387	10.903	0.127
CLJP	RSSTD	Chev	6	7.466	1.285	5.81
CLJP	RSD	dJacobi	4	4.398	0.814	5.40
CLJP	RSD	bJacobi(32)	3	4.407	0.919	4.79
CLJP	RSD	bJGS(32)	3	3.803	0.810	4.70
CLJP	RSD	GS	1	1.652	10.653	0.155
CLJP	RSD	Chev	6	7.254	1.222	5.94
CLJP	MPRS	dJacobi	4	4.822	0.813	5.93
CLJP	ML	dJacobi	3	4.601	0.896	5.14
RS	RSSTD	dJacobi	4	1.579	0.273	5.79
RS	RSSTD	wJacobi	8	2.956	0.534	5.50
RS	RSSTD	bJacobi(8)	4	2.193	0.365	6.01
RS	RSSTD	bJacobi(32)	3	1.799	0.427	4.21
RS	RSSTD	bJacobi(256)	3	1.919	1.618	1.18
RS	RSSTD	bJGS(8)	4	2.046	0.338	6.05
RS	RSSTD	bJGS(32)	3	1.608	0.347	4.63
RS	RSSTD	bJGS(256)	3	1.707	0.987	1.73
RS	RSSTD	GS	1	0.584	4.483	0.121
RS	RSSTD	Chev	7	2.575	0.463	5.53
RS	RSD	dJacobi	4	1.197	0.189	6.23
RS	RSD	wJacobi	7	2.118	0.337	6.23
RS	RSD	bJacobi(32)	3	1.516	0.341	4.44
RS	RSD	bJGS(32)	3	1.220	0.276	4.43
RS	RSD	GS	1	0.484	3.159	0.153
RS	RSD	Chev	7	2.117	0.333	6.31
RS	MPRS	dJacobi	4	1.211	0.191	6.24

Jacobi (`dJacobi`), weighted Jacobi (`wJacobi`), block Jacobi (`bJacobi`), Gauss–Seidel (`GS`), block-Jacobi–Gauss–Seidel (`bJGS`) and Chebyshev polynomial smoothers (`Chev`), are tested.

For the solution phase of the AMG solvers, the interpolation, restriction and vector operations are naturally parallel, and they have good parallel performance. As shown by the previous example, the smoothers dominate the overall running time and the speed-ups. The damped Jacobi, weighted Jacobi and polynomial smoothers are combinations of SpMV and vector operations, which are easy to parallelize. From Table 25, we can see when they are applied, all cases are over 5 times faster than the CPU AMG solvers, such as the `CLJP` coarsening strategy with any of the four interpolations and `RS` coarsening strategy with any interpolations.

As discussed above, when the block Jacobi smoother is applied, an ILU problem needs to be solved, which has limited parallelism. The parallelism can be increased by using small blocks. For the cases with the `CLJP` coarsening strategy and `RSSTD` and `RSD` interpolations, the case with a smaller block size has a higher speed-up. When the block size is 8, the case of `CLJP` with the `RSSTD` interpolation, has a speed-up of 5.27, and when the block size is 256, the speed-up is only 1.79. The running time for CPU increases slightly, but the running time for GPU increases dramatically. The results for cases with the `RS` coarsening strategy with `RSSTD` interpolation are similar, where speed-ups decrease from 6.01 to 1.18 when the block sizes increase from 8 to 256.

For the Gauss–Seidel and block-Jacobi–Gauss–Seidel smoothers, a lower triangular problem needs to be solved. The Gauss–Seidel smoother is a special case when the block size equals the number of rows of the given matrix. Again, the lower triangular solver has limited parallelism, which can be increased by using small block sizes. From the table, we can see that for the cases with the `CLJP` coarsening strategy and `RSSTD` interpolation, the speed-ups decrease from 5.18 to 2.42 when the block sizes increase from 8 to 256. The Gauss–Seidel smoother has a speed-up of 0.127, where the GPU AMG solver is much slower than the CPU AMG solver. However, for the convergence, the Gauss–Seidel has the best convergence. The results indicate that when implementing the AMG solvers on CPUs, the Gauss–Seidel smoother is a good choice, and when implementing AMG solvers on GPUs, small block sizes are preferred. The results for cases with the `RS` coarsening strategy and `RSSTD` interpolation are similar, where the speed-ups of the GPU AMG solver decrease from 6.05 to 1.73 when the block sizes increase. The Gauss–Seidel smoother has a speed-up of 0.121. When the block size is 32, the case with the `RS` coarsening strategy and `RSD` interpolation has a speed-up of 4.43. In addition, the Gauss–Seidel smoother (or the block Gauss–Seidel smoother with a block size of 504,855) has a speed-up of 0.153 only. The results in this example also show that the performance and the speed-up of GPU-version AMG solvers are affected by the choice of smoothers and block sizes, which coincide with the discussion about triangular solvers and potential speed-up.

5. Conclusions

The experimental performance on GPUs of the smoothed aggregation AMG solver and the classical AMG solver is studied in this paper and various approaches for these two AMG solvers are investigated. The RS coarsening strategy and the CLJP coarsening strategy are both employed. Four interpolation operators are implemented, including the standard RS interpolator, the direct interpolator, the multiple pass interpolator and the ML interpolator. Several commonly used smoothers are implemented on GPUs and CPUs, including the damped Jacobi, weighted Jacobi, block Jacobi, Chebyshev polynomial, Gauss–Seidel, and block-Jacobi–Gauss–Seidel smoothers. Numerical experiments show that the solution phase of the GPU-version AMG solvers is accelerated up to 13 times faster using the NVIDIA Tesla C2050/C2070 GPUs as compared to the sequential execution on Intel Xeon X5570 CPUs. These results also demonstrate that the GPUs constitute a high performance computing platform for scientific computing. As discussed in this paper, the setup phase in the AMG solvers is performed on CPUs due to a difficulty. In the future, we will attempt to implement the setup phase on the GPUs and to design algorithms to accelerate this phase.

Acknowledgments

The support of Department of Chemical and Petroleum Engineering and Reservoir Simulation Group, University of Calgary is gratefully acknowledged. The research is partly supported by NSERC (IRCPJ365863)/AIEES (1741)/Foundation CMG and AITF Chairs.

References

[1] R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. Van der Vorst, Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, second ed., SIAM, 1994.
[2] Y. Saad, Iterative Methods for Sparse Linear Systems, second ed., SIAM, 2003.
[3] H. Klie, H. Sudan, R. Li, Y. Saad, Exploiting capabilities of many core platforms in reservoir simulation, in: SPE RSS Reservoir Simulation Symposium, 21–23 February 2011.
[4] R. Li, Y. Saad, GPU-accelerated Preconditioned Iterative Linear Solvers, Technical Report umsi-2010-112, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, 2010.
[5] K. Stüben, A review of algebraic multigrid, J. Comput. Appl. Math. 128 (1–2) (2001) 281–309.
[6] J.W. Ruge, K. Stüben, Algebraic multigrid (AMG), in: S.F. McCormick (Ed.), Multigrid Methods, in: Frontiers in Applied Mathematics, vol. 5, SIAM, Philadelphia, 1986.
[7] A. Brandt, S.F. McCormick, J. Ruge, Algebraic multigrid (AMG) for sparse matrix equations, in: D.J. Evans (Ed.), Sparsity and its Applications, Cambridge University Press, Cambridge, 1984, pp. 257–284.
[8] C. Wagner, Introduction to Algebraic Multigrid, Course notes of an algebraic multigrid course at the University of Heidelberg in the Winter semester, 1999.
[9] P.S. Vassilevski, Lecture Notes on Multigrid Methods, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, 2010.
[10] R. Falgout, A. Cleary, J. Jones, E. Chow, V. Henson, C. Baldwin, P. Brown, P. Vassilevski, U.M. Yang, Hypre home page, 2011. http://acts.nersc.gov/hypre.
[11] A.J. Cleary, R.D. Falgout, V.E. Henson, J.E. Jones, T.A. Manteuffel, S.F. McCormick, G.N. Miranda, J.W. Ruge, Robustness and scalability of algebraic multigrid, SIAM J. Sci. Comput. 21 (2000) 1886–1908.
[12] V.E. Henson, U.M. Yang, BoomerAMG: a parallel algebraic multigrid solver and preconditioner, Appl. Numer. Math. 41 (2000) 155–177.
[13] U.M. Yang, Parallel algebraic multigrid methods — high performance preconditioners, in: A.M. Bruaset, A. Tveito (Eds.), Numerical Solution of Partial Differential Equations on Parallel Computers, Vol. 51, Springer-Verlag, 2006, pp. 209–236. chapter.
[14] R.D. Falgout, An introduction to algebraic multigrid, Comput. Sci. Eng. 8 (2006) 24–33. Special issue on Multigrid Computing.
[15] Z. Chen, G. Huan, Y. Ma, Computational Methods for Multiphase Flows in Porous Media, in: The Computational Science and Engineering Series, vol. 2, SIAM, Philadelphia, 2006.
[16] X. Hu, W. Liu, G. Qin, J. Xu, Y. Yan, C. Zhang, Development of a fast auxiliary subspace pre-conditioner for numerical reservoir simulators, in: SPE Reservoir Characterisation and Simulation Conference and Exhibition, 9–11 October 2011, Abu Dhabi, UAE, SPE-148388-MS.
[17] J.R. Wallis, R.P. Kendall, T.E. Little, Constrained residual acceleration of conjugate residual methods, in: SPE Reservoir Simulation Symposium. 1985.
[18] H. Cao, H.A. Tchelepi, J.R. Wallis, H.E. Yardumian, Parallel scalable unstructured CPR-type linear solver for reservoir simulation, in: SPE Annual Technical Conference and Exhibition. 2005.
[19] Samsung Electronics Co, Ltd., Samsung DDR4 SDRAM Brochure, Jun 2013.
[20] NVIDIA Corporation, Nvidia CUDA Programming Guide (version 3.2), 2010.
[21] N. Bell, M. Garland, Efficient sparse matrix–vector multiplication on CUDA, NVIDIA Technical Report, NVR-2008-004, NVIDIA Corporation, 2008.
[22] N. Bell, M. Garland, Implementing sparse matrix–vector multiplication on throughput-oriented processors, Proc. Supercomput. (2009) 1–11.
[23] NVIDIA Corporation, CUSP: Generic Parallel Algorithms for Sparse Matrix and Graph, http://code.google.com/p/cusp-library/.
[24] H. Liu, S. Yu, Z. Chen, B. Hsieh, L. Shao, Parallel preconditioners for reservoir simulation on GPU, in: SPE Latin American and Caribbean Petroleum Engineering Conference held in Mexico City, Mexico, 16–18 April 2012, SPE 152811-PP.
[25] M. Naumov, Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU, NVIDIA Technical Report, 2011, June.
[26] H. Liu, S. Yu, Z. Chen, B. Hsieh, L. Shao, Sparse matrix–vector multiplication on NVIDIA GPU, Int. J. Numer. Anal. Model. Ser. B 3 (2) (2012) 185–191.
[27] Z. Chen, H. Liu, S. Yu, B. Hsieh, L. Shao, GPU-based parallel reservoir simulators, in: Proc. of 21st International Conference on Domain Decomposition Methods, France, 2012.
[28] S. Yu, H. Liu, Z. Chen, B. Hsieh, L. Shao, GPU-based parallel reservoir simulation for large-scale simulation problems, in: SPE Europec/EAGE Annual Conference, Copenhagen, Denmark, 2012.
[29] Z. Chen, H. Liu, S. Yu, Development of algebraic multigrid solvers using GPUs, in: SPE-163661-MS, SPE Reservoir Simulation Symposium, 18–20 February, The Woodlands, Texas, USA, 2013.
[30] G. Haase, M. Liebmann, C.C. Douglas, G. Plank, A parallel algebraic multigrid solver on graphics processing units, High Perform. Comput. Appl. (2010) 38–47.

[31] N. Bell, S. Dalton, L. Olson, Exposing Fine-Grained Parallelism in Algebraic Multigrid Methods, NVIDIA Technical Report NVR-2011-002, 2011, June.
[32] J. Bolz, I. Farmer, E. Grinspun, P. Schröder, Sparse matrix solvers on the GPU: conjugate gradients and multigrid, Symposium 22 (3) (2007) 917–924.
[33] L. Buatois, G. Caumon, B. Lévy, Concurrent number cruncher: an efficient sparse linear solver on the GPU, High Perform. Comput. Commun. 4782 (2007) 358–371.
[34] D. Goddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, H. Wobker, C. Becker, S. Turek, Using GPUs to improve multigrid solver performance on a cluster, Int. J. Comput. Sci. Eng. 4 (1) (2008) 36–55.
[35] N. Bell, S. Dalton, L. Olson, Exposing fine-grained parallelism in algebraic multigrid methods, SIAM J. Sci. Comput. 34 (4) (2012) 123–152.
[36] L. Wang, X. Hu, J. Cohen, J. Xu, A parallel auxiliary grid algebraic multigrid method for graphic processing unit, SIAM J. Sci. Comput. 35 (3) (2013) 263–283.
[37] J. Brannick, Y. Chen, X. Hu, L. Zikatanov, Parallel unsmoothed aggregation algebraic multigrid algorithms on GPUs, Springer Process. Math. Stat. 45 (2013) 81–102.
[38] K. Stüben, Algebraic Multigrid (AMG): An Introduction with Applications, Institute for Algorithms and Scientific Computing, Germany, 1999.
[39] Z. Chen, H. Liu, B. Yang, Parallel triangular solvers on GPU, in: Proceedings of International Workshop on Data-Intensive Scientific Discovery (DISD), 2013, Shanghai University, Shanghai, China, August 1–4, 2013.
[40] Z. Chen, H. Liu, B. Yang, Accelerating iterative linear solvers using multiple graphical processing units, Int. J. Comput. Math. 92 (7) (2015) 1422–1438.
[41] T.A. Davis, University of Florida sparse matrix collection, NA digest, 1994.
[42] G. Ruetsch, M. Fatica, CUDA Fortran for Scientists and Engineers, NVIDIA Corporation, Santa Clara, USA, 2011.