Numerical Study of Geometric Multigrid Methods on CPU–GPU Heterogeneous Computers

Chunsheng Feng, Shi Shu, Jinchao Xu, Chen-Song Zhang

August 22, 2012

Abstract

The geometric multigrid method (GMG) is one of the most efficient solving techniques for discrete algebraic systems arising from many types of partial differential equations. GMG utilizes a hierarchy of grids or discretizations and reduces the error at a number of frequencies simultaneously. Graphics processing units (GPUs) have recently burst onto the scientific computing scene as a technology that has yielded substantial performance and energy-efficiency improvements. A central challenge in implementing GMG on GPUs, though, is that computational work on coarse levels cannot fully utilize the capacity of a GPU. In this work, we perform numerical studies of GMG on CPU–GPU heterogeneous computers. Furthermore, we compare our implementation with an efficient CPU implementation of GMG and with the most popular fast Poisson solver, Fast Fourier Transform, in the cuFFT library developed by NVIDIA.
1 Introduction

Simulation-based scientific discovery and engineering design demand extreme computing power and high-efficiency algorithms (Bjørstad et al. 1992; Bjørstad, Dryja, and Rahman 2003; Hey, Tansley, and Tolle 2009; Kaushik et al. 2011; Keyes 2011). This demand is one of the main forces driving the pursuit of extreme-scale computer hardware and software during the last few decades. It has become increasingly important for algorithms to be well-suited to the emerging hardware architecture. In fact, the co-design of architectures, algorithms, and appellations is particulary important given that researchers are trying to achieve exascale (10^18 floating-point operations per second) computing. Although the question of what is the best computer architecture to achieve exascale or higher remains highly debatable, many researchers agree that hybrid architectures make increasing sense due to modern-day energy-consumption constraints, whereby we can no longer reduce voltage proportional to the density of transistors. There are already quite a few heterogeneous computing architectures available, including the Cell Broadband Engine Architecture (CBEA), Graphics Processing Units (GPUs), and Field Programmable Gate Arrays (FPGAs) (Carpenter and Symon 2009; Brodtkorb et al. 2011; Wolfe 2012).

A GPU is a symmetric multicore processor that can be accessed and controlled by CPUs. The Intel/AMD CPU accelerated by NVIDIA/AMD GPUs is probably the most commonly used heterogeneous high-performance computing (HPC) architecture at present. GPU-accelerated supercomputers feature in many of the top computing systems in the HPC Top500 (Top500 2011) and the Green500 (Green500 2011). It has been reported that many “old” supercomputers, such as Jaguar, are currently being redesigned in order to incorporate GPUs and thereby achieve better
performance. GPUs have evolved from fixed-pipeline application-specific integrated circuits into highly programmable, versatile computing devices. Under conditions often met in scientific computing, modern GPUs surpass CPUs in computational power, data throughput, and computational efficiency per watt by almost one order of magnitude (Buck 2007; Chen et al. 2009; Nickolls and Dally 2010).

Not only are GPUs the key ingredient in many current and forthcoming petaflop supercomputers, they also provide an affordable desktop supercomputing environment for everyday usage, with peak computational performance matching that of the most powerful supercomputers of only a decade ago. General-purpose graphics processing units (GPGPU), as a high-performance computational device are becoming increasingly popular. Today’s NVIDIA Fermi GPU and the upcoming (expected in December 2012) Intel Many Integrated Core (MIC) architecture are the most promising co-processors with high energy-efficiency and computation power. The Intel Knight Corner MIC (50 cores) is capable of delivering 1 Teraflop operation in double precision per second, whereas the peak performance of Tesla 2090 is 665 Gigaflop operations in double precision per second. On the other hand, as GPU has a high-volume graphics market, it is expected by many experts to have a price advantage over MIC, at least immediately after its launch.

Probably one of the most discussed features of the MIC architecture is that it shares the x86 instruction set such that users often assume that they do not need to change their existing codebase in order to migrate to MIC. However, this assumption is subject to argument as even if legacy code can easily be migrated, whether the application it is then used for is able to achieve the desired performance is questionable. Achieving scalable scientific applications in the exascale era is our ultimate goal. Hence, software, more importantly algorithms, must adapt if it is to unleash the power of the hardware. Unfortunately, none of the processors envisioned at present will relieve today’s programmers from the hard work of preparing their applications. In fact, power constraints will actually cause us to use simpler processors at lower clock rates for the majority of our work. As an inevitable consequence, improvements in terms of performance will largely arise from more parallel algorithms and implementation.

It is still too early to tell which architecture(s) will dominate the supercomputing market in the future. In all likelihood, different applications will benefit from any given architecture in specific ways. Thus a one-size-fits-all solution will almost certainly not arise. In many numerical simulation applications, the most time-consuming aspect is usually the solution of large linear systems of equations. Often, as they are generated by discretized partial differential equations (PDEs), the corresponding coefficient matrices are very sparse. The Laplace operator (or Laplacian) occurs in many PDEs that describe physical phenomena such as heat diffusion, wave propagation and electrostatics, and gravitational potential. For many efficient methods for solving discrete problems arising from PDEs, a fast Poisson solver is a key ingredient in achieving a high level of efficiency (Xu 2010). Numerical schemes based on fast Poisson solvers have been successfully applied to many practical problems among which are computer tomography, power grid analysis, and quantum-chemical simulation (Köstler et al. 2007; Sturmer, Kostler, and Rude 2008; Shi et al. 2009; Yang, Cai, and Zhou 2011).

Because of its plausible linear complexity—i.e., the low computational cost of solving a linear system with \( N \) unknowns is \( O(N) \)—the Poisson solver is one of the most popular GMG methods (Hackbusch 1985; Bramble 1993; Briggs, Henson, and McCormick 2000; Trottenberg, Oosterlee, and Schüller 2001; Brandt 2011). Although the GMG’s applicability is limited as it requires explicit information on the hierarchy of the discrete system, when it can be applied, GMG is far more
efficient than its algebraic version, the algebraic multigrid (AMG) method (Brandt, McCormick, and Ruge 1982; Brandt 1986; Ruge and Stüben 1987; Trottenberg, Oosterlee, and Schüller 2001). Another popular choice is the direct solver based on the fast Fourier transform or the FFT (Cooley and Tukey 1965) on tensor product grids. The computational cost of the FFT-based fast Poisson solver is $O(N \log N)$, and FFT can easily be called from highly optimized software libraries, such as FFTW (Frigo and Johnson 2005) and the Intel Math Kernel Library (MKL). These advantages make FFT an extremely appealing method (Sturmer, Kostler, and Rude 2008; Lord et al. 2008) when it is applicable.

It is well-known that heterogeneous architectures pose new programming difficulties compared to existing serial and parallel platforms (Chamberlain et al. 2007; Brodtkorb et al. 2010). In this paper, we investigate the performance of fast Poisson solution algorithms, more specifically, GMG and FFT, on modern massively parallel computing environments like Tesla GPUs. Considerable effort has been devoted to developing efficient solvers for linear systems arising from PDEs and other applications (Bolz, Farmer, and Grinspun 2003; Bell and Garland 2008; Bell and Garland 2009; Barrachina et al. 2009; Jeschke and Cline 2009; Cao et al. 2010; Elble, Sahinidis, and Vouzis 2010; Georgescu and Okuda 2010; Bell, Dalton, and Olson 2011; Heuveline et al. 2011; Heuveline, Lukarski, and Weiss 2011; Knibbe, Oosterlee, and Vuik 2011) and the references therein.

The main purpose of this paper is to consider the following important questions, all of which are central to understanding geometric multigrid methods on GPU architecture:

- Is it possible to achieve a satisfactory speedup on GPUs for multigrid algorithms?
- How does the performance of multigrid algorithms on GPUs compare with their performance on a single state-of-the-art CPU core?
- How much of the computational power of GPUs can be used for multigrid algorithms? How cost-effective are CPU–GPU systems?
- Compared with the optimized implementation of direct solvers based on FFT, does GMG have any advantages in addition to its generality?

We will consider answers to these questions based on carefully designed numerical experiments described herein.

The rest of the paper is organized as follows: In Section 2 we introduce the preliminary features of the hardware and algorithms under investigation. In Section 3 we give details about our implementation of GMG in a CPU–GPU heterogeneous computing environment. In Section 4 we analyze the complexity of the GMG algorithm. We report our numerical tests and analysis in Section 5. We then summarize the paper with some concluding remarks in Section 6.

2 Preliminaries

2.1 A Brief Glance at GPU and CUDA

Graphics processing units (GPUs) recently burst onto the scientific computing scene as an innovative technology that has demonstrated substantial performance and energy-efficiency improvements for many scientific applications. A typical CPU–GPU heterogenous architecture contains one or more CPUs (host) and a GPU (device). GPU has its own device memory, which is connected to the host via a PCI express bus. One of the main drawbacks of using such an architecture for PDE
applications is that it is necessary to exchange data between the host and the device frequently (see Figure 1) (Brodtkorb et al. 2010).

Figure 1: Schematic of heterogenous architecture: a quad core CPU in combination with a GPU (Brodtkorb et al. 2010). Data must be moved to the GPU memory, and parallel kernels are launched asynchronously on the GPU by the host.

In regard to sparse matrix operations, the memory bandwidth is usually where the bottleneck occurs. Furthermore, the gap between the speed of floating-point operations and the speed for accessing memory grows every year (Asanovic et al. 2006). In this sense, we do not expect that the iterative linear solvers, which is usually memory-bounded, to readily derive benefits easily from increasing the number of cores. One way to address this problem is to use a high-bandwidth memory, such as Convey’s Scatter-Gather memory. Another is to add multithreading, where the execution unit saves the state of two or more threads, and can swap execution between threads in a single cycle. There are two ways to do this—either by swapping between threads at a cache miss or by alternating between threads on every cycle. While one thread is waiting for memory, the execution unit keeps busy by switching to a different thread. To be effective, though, the program must become even more parallel, which will be exploited via multithreading.

CUDA (Compute Unified Device Architecture) (NVIDIA 2012a) is a parallel computing platform and programming model invented by NVIDIA. It delivers dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU). NVIDIA provides a complete toolkit for programming on the CUDA architecture, supporting standard computing languages such as C/C++ and Fortran. CUDA C and Fortran are the most widely used programming languages for GPU programming today (Wolfe 2012). CUDA was developed simultaneously with the GeForce 8 architecture (NVIDIA’s internal code name for the latter is Tesla), and publicly announced in 2006. In addition to CUDA, other options are OpenCL, AMD Stream SDK, and OpenACC supported by CAPS (CAPS 2012), CRAY (Inc. 2012), NVIDIA, and PGI (PGI 2012).

2.2 The Poisson Equation and Its Finite Difference Discretizations

Consider the Poisson equation

\[
\begin{aligned}
\Delta u &= f \quad \text{in } \Omega \\
u &= 0 \quad \text{on } \partial \Omega,
\end{aligned}
\]  

(2.1)

where \( \Omega = (0,1)^d \subset \mathbb{R}^d \). The standard central finite difference method is applied to discretize the Poisson equation (2.1) (Morton and Mayers 2005). In other words, the Laplace operator is
discretized by the classical second-order central difference scheme. After discretization, we end up with a system of linear equations:

\[ A \vec{u} = \vec{f}. \]  \hspace{1cm} (2.2)

\section*{2D Five-Point Stencil}

We use the five-point central difference scheme in 2D. Consider a uniform square mesh of \( \Omega = [0,1]^2 \) with size \( h = \frac{1}{n} \) and in which \( x_i = ih, y_j = jh \) \((i,j = 0,1,\ldots,n)\). Let \( u_{i,j} \) be the numerical approximation of \( u(x_i, y_j) \). The five-point central difference scheme for solving (2.1) in 2D can be written as follows:

\[ -u_{i-1,j} - u_{i,j+1} + 4u_{i,j} - u_{i+1,j} - u_{i,j-1} = f(x_i, y_j) \quad i, j = 1,2,\ldots,n-1. \]

\section*{3D Seven-Point Stencil}

We use the seven-point central difference scheme in 3D. Similar to the 2D case, we consider a uniform cube mesh of \( \Omega = [0,1]^3 \) with size \( h = \frac{1}{n} \) and in which \( x_i = ih, y_j = jh \) and \( z_k = kh, i,j,k = 0,1,\ldots,n \). Let \( u_{i,j,k} \approx u(x_i, y_j, z_k) \) be the approximate solution. The seven-point central difference scheme for solving (2.1) in 3D reads

\[ -u_{i-1,j,k} - u_{i,j-1,k} - u_{i,j,k-1} + 6u_{i,j,k} - u_{i+1,j,k} - u_{i,j+1,k} - u_{i,j,k+1} = f(x_i, y_j, z_k), \]

for all \( i,j,k = 1,2,\ldots,n-1 \).

\section*{2.3 Fast Fourier Transform}

A fast Fourier transform (FFT) is an efficient algorithm for computing the discrete Fourier transform (DFT) and its inverse. DFT decomposes a sequence of values into components of different frequencies. Computing DFT directly from its definition is usually too slow to be practical. The FFT can be used to compute the same result, but much more quickly. In fact, computing a DFT of \( N \) points directly, according to its definition, takes \( O(N^2) \) arithmetical operations, whereas FFT can compute the same result in \( O(N \log N) \) operations (Walker 1996).

On tensor product grids, FFT can be used to solve the Poisson equation efficiently. We now explain the key steps for using FFT to solve the 2D Poisson equation (the 3D case is similar):

1. Apply 2D forward FFT to \( f(x, y) \) to obtain \( \hat{f}(k_x, k_y) \), where \( k_x \) and \( k_y \) are the wave numbers.

The 2D Poisson equation in the Fourier space can then be written as

\[ -\Delta u(x, y) = f(x, y) \xrightarrow{FFT} - (k_x^2 + k_y^2) \hat{u}(k_x, k_y) = \hat{f}(k_x, k_y). \]  \hspace{1cm} (2.1)

2. Apply the inverse of the Laplace operator to \( \hat{f}(k_x, k_y) \) to obtain \( \hat{u}(k_x, k_y) \), which is the element-wise division in the Fourier space

\[ \hat{u}(k_x, k_y) = \frac{\hat{f}(k_x, k_y)}{k_x^2 + k_y^2}. \]

3. Apply 2D inverse FFT to \( \hat{u}(k_x, k_y) \) to obtain \( u(x, y) \).
The NVIDIA CUDA Fast Fourier Transform (cuFFT version 4.1) library provides a simple interface for computing FFTs up to 10 times faster than MKL 10.2.3 for single precision. By using hundreds of processor cores on NVIDIA GPUs, cuFFT is able to deliver the floating point performance of a GPU without necessitating the development of custom GPU FFT implementation (NVIDIA 2012b).

3 Geometric Multigrid Method for GPU

Multigrid (MG) methods in numerical analysis comprise a group of algorithms for solving differential equations using a hierarchy of discretizations. The main idea driving multigrid methodology is that of accelerating the convergence of a simple (but usually slow) iterative method by global correction from time to time, accomplished by solving corresponding coarse-level problems. Multigrid methods are typically applied to numerically solving discretized partial differential equations (Hackbusch 1983; Trottenberg, Oosterlee, and Schüller 2001). In this section, we briefly review standard multigrid and full multigrid (V-cycle) algorithms and their respective implementations in a CPU–GPU heterogenous computing environment.

3.1 Geometric multigrid method

The key steps in the multigrid method (see Figure 2) are as follows:

- **Relaxation** or **Smoothing**: Reduce high-frequency errors using one or more smoothing steps based on a simple iterative method, like Jacobi or Gauss-Seidel.
- **Restriction**: Restrict the residual on a finer grid to a coarser grid.
- **Prolongation** or **Interpolation**: Represent the correction computed on a coarser grid to a finer grid.

![Figure 2: Pictorial representation of a multigrid method with three grid levels.](image)

One of the simplest multilevel iterative methods is the multigrid V-cycle (see Figure 3). The algorithm proceeds from left to right and from top (finest grid) to bottom (coarsest grid) and back up again. The V-cycle algorithm can be written as (shown in Figure 3)

*cuFFT 4.1 on Tesla M2090, ECC on, MKL 10.2.3, and TYAN FT72-B7015 Xeon x5680 Six-Core 3.33GHz.
Algorithm 1: Multigrid V-cycle $\vec{u} = \text{MG-V}(\mu_f, \mu_b, L, \vec{u}, \vec{f})$

1. for $l = 0$ to $L - 2$ do
2.     Relax$_{forward}(\mu_f, A_l, \vec{f}_l, \vec{u}_l)$
3.     $\vec{r}_l = \vec{f}_l - A_l \vec{u}_l; \vec{f}_{l+1} = R_{l+1}^{l+1} \vec{r}_l$
4.     Relax$_{forward}(\mu_f, A_{L-1}, \vec{f}_{L-1}, \vec{u}_{L-1})$
5. for $l = L - 1$ to $0$ do
6.     $\vec{u}_l = \vec{u}_l + P_{l+1}^{l+1} \vec{u}_{l+1}$
7.     Relax$_{backward}(\mu_b, A_1, \vec{f}_1, \vec{u}_1)$

Remark 3.1 (Coarsest-level solver) Note that, for simplicity, we assume that the coarsest level $L - 1$ contains one degree of freedom. Hence, Relax$_{forward}(\mu_f, A_{L-1}, \vec{f}_{L-1}, \vec{u}_{L-1})$ in Algorithm 1 solves the coarsest-level problem exactly. The same thing happens in Algorithm 2.

The full multigrid (FMG) usually gives the best performance in terms of computational complexity. The idea of FMG is represented in Figure 4. We start from the coarsest grid and solve the discrete problem on the coarsest grid. Then, we interpolate this solution to the second-coarsest grid and perform one V-cycle. These two steps are repeated recursively on finer and finer grids, until the finest grid possible is achieved. The details are described in Algorithm 2.

Figure 3: A schematic description of the V-cycle.

Figure 4: A schematic description of the full multigrid algorithm. The algorithm proceeds from left to right and from top (finest grid) to bottom (coarsest grid).
Algorithm 2: Full Multigrid V-cycle $\vec{u} = \text{FMG-V} (\mu_f, \mu_b, L, \vec{u}, \vec{f})$

1. init $\vec{f}_l, \vec{u}_l$, $l = 0, \ldots, L - 1$
2. RelaxGS\text{\_forward}(\mu_f, A_{L-1}, \vec{f}_{L-1}, \vec{u}_{L-1})
3. for $l = L - 2$ to 0 do
   4. $\vec{u}_l = P_{l+1}^{\perp} \vec{u}_{l+1}$
   5. $u_l = \text{V-cycle}(\mu_f, \mu_b, l, \vec{u}_l, \vec{f}_l)$

3.2 Implementation of GMG on a CPU–GPU machine

Algorithm 3: $\vec{u}, \text{iter} = \text{GMGSolve}(d, L, L_\theta, \text{tols}, \text{maxit}, \mu_f, \mu_b)$

1. for $l = 0$ to $L$ do: init $\vec{u}_l$ and $\vec{f}_l$; resinit = $\| \vec{f}_0 - A \vec{u}_0 \|
2. for $l = L_\theta$ to $L - 1$ do: init $\vec{u}_l$ and $\vec{f}_l$; res = resinit; iter = 0
3. while $(\text{res} > \text{tols} \times \text{resinit}) \text{and} (\text{iter} < \text{maxit})$ do
   4. for $l = 0$ to $L_\theta - 1$ at GPU do
      5. RelaxGS\text{\_forward}(\mu_f, A_l, \vec{f}_l, \vec{u}_l); $\vec{r}_l = \vec{f}_l - A_l \vec{u}_l$; $\vec{r}_{l+1} = R_{l+1}^{\perp} \vec{r}_l$
   6. copy $\vec{f}_{L_\theta}$ from DEVICE memory to HOST memory
   7. $\vec{u}_{L_\theta} = \text{MG-V}(\mu_f, \mu_b, L_\theta, \vec{u}_{L_\theta}, \vec{f}_{L_\theta})$
   8. copy $\vec{u}_{L_\theta}$ from HOST memory to DEVICE memory
   9. for $l = L_\theta - 1$ to 0 at GPU do
      10. $\vec{u}_l = \vec{u}_l + P_{l+1}^{\perp} \vec{u}_{l+1}$; RelaxGS\text{\_backward}(\mu_b, A_l, \vec{f}_l, \vec{u}_l)
      11. res = $\| \vec{f} - A \vec{u}_0 \|
      12. iter = iter + 1
   13. copy $\vec{u}_0$ from DEVICE memory to HOST memory $\vec{u}$

Remark 3.2 We offer some remarks about our implementation:

1. There have been many discussions on how to implement geometric multigrid methods efficiently on modern computer architectures; see, for example, Weiss [2001].
2. We use a four-color and an eight-color Gauss-Seidel smoother for RelaxGS in 2D and 3D, respectively. As we are considering structured grids and the coloring is easy to obtain, we prefer the GS smoother over the weighted Jacobi smoother. A weighted Jacobi smoother is likely to achieve a higher peak performance and higher speedup over the corresponding CPU version; however, it usually requires more iterations and wall time compared with the colored GS smoother if both methods use same multilevel iteration, like V-cycle.
3. The gray boxes represent the code segments running on GPU (kernel functions). When $L_\theta = 0$, Algorithm 3 runs on CPU completely, and when $L_\theta = L$, Algorithm 3 runs on GPU solely. However, when $0 < L_\theta < L$, these functions run in CPU–GPU hybrid mode.
4. Graphics processors provide texture memory to accelerate frequently performed operations. As optimized data access is crucial to GPU performance, the use of texture memory can sometimes provide a considerable performance boost. We band the vectors $\vec{u}_l$ as one dimension texture memory in function $\vec{r}_l = \vec{f}_l - A_l \vec{u}_l$, $l = 0, 1, \ldots, L - 2$. 

9
4 Complexity Analysis of the GMG Algorithm

For the geometry multigrid of the finite difference method on structured meshes, it is not necessary to explicitly assemble the global stiffness matrix $A$ (i.e., matrix-free). A subroutine for the matrix-vector multiplication of the corresponding finite difference operator is called whenever we need to compute $\vec{r} = \vec{f} - A\vec{u}$. This subroutine can be implemented directly from the central difference scheme.

One V-cycle of the GMG algorithm consists of computing the residual, forward relaxation, backward relaxation, restriction, prolongation, and the inner product. Take the 2D case as an example: we can analyze the time and space complexity of these operations. The time complexity of one V-cycle can be proven to be $O(N)$ for both the 2D and 3D cases. The optimal complexity of GMG has been analyzed by Griebel (Griebel 1989).

Next, we analyze the exact operation counts for a V-cycle.

2D case

- Residual: $\vec{r}_l = \vec{f}_l - A_l\vec{u}_l$

$$r_{i,j}^l = f_{i,j}^l - 4u_{i,j}^l + u_{i\pm 1,j}^l + u_{i,j\pm 1}^l, \quad i, j = 1, 2, \ldots, n_l - 1, \quad (4.1)$$

where $u_{i\pm 1,j}^l = u_{i-1,j}^l + u_{i+1,j}^l$ and in this case when $i = 0$ or $i = n_l + 1$, then $u_{i\pm 1,j}^l = 0$. From now on, we will use the notation $\pm$ in this section. The equation (4.1) requires 6 floating-point operations or work units (W) per unknown in the 2D case. Furthermore, we obtain the total number of floating-point operations for the residual in one V-cycle as

$$W_{Residual} = 6 \sum_{l=0}^{L-2} (n_l)^2. \quad (4.2)$$

- Gauss-Seidel Relaxation:

$$u_{i,j}^l = \frac{1}{4}(f_{i,j}^l + u_{i\pm 1,j}^l + u_{i,j\pm 1}^l), \quad i, j = 1, 2, \ldots, n_l - 1. \quad (4.3)$$

Equation (4.3) shows that there are 5 floating-point operations per unknown in the 2D case. Hence, the total number of floating-point operations in the forward and backward Gauss-Seidel relaxation in one V-cycle is

$$W_{GS\text{forward}} = 5 \sum_{l=0}^{L-1} (n_l)^2, \quad W_{GS\text{backward}} = 5 \sum_{l=0}^{L-2} (n_l)^2. \quad (4.4)$$

- Restriction: $\vec{r}_{l+1} = R_{l+1}^l \vec{r}_l$

$$r_{i,j}^{l+1} = \frac{1}{8}(2r_{2i,2j}^l + r_{2i+1,2j}^l + r_{2i,2j+1}^l + r_{2i-1,2j-1}^l + r_{2i+1,2j+1}^l), \quad i, j = 1, 2, \ldots, n_l - 1. \quad (4.5)$$
Equation (4.5) requires 8 floating-point operations per unknown in the 2D case. Furthermore, we obtain the total floating-point operations of restriction for one V-cycle as

$$W_{\text{restriction}} = 8 \sum_{l=1}^{L-1} (n_l)^2.$$  

(4.6)

- Prolongation: $\vec{c}_l = \vec{c}_l + P_l^{l+1} \vec{c}_{l+1}$

$$
\begin{align*}
\epsilon_{2i,2j}^l &= \epsilon_{2i,2j}^l + \epsilon_{i,j}^{l+1}, \\
\epsilon_{2i,2j+1}^l &= \epsilon_{2i,2j+1}^l + \frac{1}{2} (\epsilon_{i,j}^{l+1} + \epsilon_{i,j+1}^{l+1}), \\
\epsilon_{2i+1,2j}^l &= \epsilon_{2i+1,2j}^l + \frac{1}{2} (\epsilon_{i,j}^{l+1} + \epsilon_{i+1,j}^{l+1}), \\
\epsilon_{2i+1,2j+1}^l &= \epsilon_{2i+1,2j+1}^l + \frac{1}{2} (\epsilon_{i,j}^{l+1} + \epsilon_{i+1,j+1}^{l+1}).
\end{align*}
\tag{4.7}
$$

Equation (4.7) shows that there are $\frac{3 \times 3^{l+1}}{4}$ floating-point operations per unknown for the 2D case. Furthermore, we can obtain the total number of floating-point operations of prolongation for one V-cycle as

$$W_{\text{prolongation}} = 2.5 \sum_{l=0}^{L-2} (n_l)^2.$$  

(4.8)

- Compute the norm of the residual: $\| \cdot \|$

$$\| \bar{r}_0 \|_{L^2} = \sum_{j=1}^{n_0} r_{0,j}^2.$$  

(4.9)

From equation (4.9), we can see that the total number of floating-point operations for computing $\ell^2$-norm is $2(n_0)^2$.

By combining equations (4.2), (4.4), (4.6), (4.8), and (4.9), we can obtain the total number of floating-point operations per unknown for the 2D case in one V-cycle as

$$6 \sum_{l=0}^{L-2} (n_l)^2 + 5 \sum_{l=0}^{L-1} (n_l)^2 + 5 \sum_{l=0}^{L-2} (n_l)^2 + 8 \sum_{l=1}^{L-1} (n_l)^2 + \frac{10}{4} \sum_{l=0}^{L-2} (n_l)^2 + (2 + 6)(n_0)^2$$

$$\approx 36.$$  

This means the total number of floating-point operations per unknown required by one V-cycle in the 2D case is about 36.
3D case

Similarly, we can count the complexity of a V-cycle in 3D:

- Residual: $\vec{r}_{l+1} = \vec{f}_l - A_l \vec{u}_l$
  
  $$r_{i,j,k}^l = f_{i,j,k}^l - 6u_{i,j,k}^l + u_{i\pm1,j,k}^l + u_{i,j\pm1,k}^l + u_{i,j,k\pm1}^l, \quad i, j, k = 1, \ldots, n_l - 1. \quad (4.10)$$

  $$W_{Residual} = 8 \sum_{l=0}^{L-2} (n_l)^3 + 8(n_0)^3. \quad (4.11)$$

- Gauss-Seidel Relaxation:
  
  $$u_{i,j,k}^l = \frac{1}{6}(f_{i,j,k}^l + u_{i\pm1,j,k}^l + u_{i,j\pm1,k}^l + u_{i,j,k\pm1}^l), \quad i, j, k = 1, \ldots, n_l - 1. \quad (4.12)$$

  $$W_{GSforward} = 7 \sum_{l=0}^{L-1} (n_l)^3, \quad W_{GSbackward} = 7 \sum_{l=0}^{L-2} (n_l)^3. \quad (4.13)$$

- Restriction: $\vec{r}_{l+1} = R_{l+1}^l \vec{r}_l$
  
  $$r_{i,j,k}^{l+1} = \frac{1}{16}\left(2r_{2i,2j,2k}^l + r_{2i\pm1,2j,2k}^l + r_{2i,2j\pm1,2k}^l + r_{2i,2j,2k\pm1}^l + r_{2i,2j-1,2k-1}^l + r_{2i,2j+1,2k+1}^l + r_{2i-1,2j,2k-1}^l + r_{2i+1,2j,2k+1}^l + r_{2i-1,2j-1,2k-1}^l + r_{2i+1,2j-1,2k+1}^l + r_{2i+1,2j+1,2k+1}^l\right), \quad (4.14)$$

  for $i, j, k = 1, \ldots, n_l - 1$.

  Then we obtain that

  $$W_{Restriiction} = 16 \sum_{l=1}^{L-1} (n_l)^3. \quad (4.15)$$
• Prolongation: \( \mathbf{e}_l = \mathbf{e}_l + \mathbf{P}_l^{l+1} \mathbf{e}_{l+1} \)

\[
W_{\text{Prolongation}} = \frac{23}{8} \sum_{l=0}^{L-2} (n_l)^3. \tag{4.16}
\]

• Compute the norm of the residual: \( \| \cdot \| \)

\[
\| \mathbf{r}_0 \|_{L^2} = \sum_{j=1}^{(n_0)^3} r_0^j r_0^j. \tag{4.17}
\]

By combining (4.11), (4.13), (4.15), (4.16), and (4.17), we obtain the total number of floating-point operations per unknown for the 3D case in one V-cycle as

\[
8 \sum_{l=0}^{L-2} (n_l)^3 + 7 \sum_{l=0}^{L-1} (n_l)^3 + 7 \sum_{l=0}^{L-2} (n_l)^3 + 16 \sum_{l=1}^{L-1} (n_l)^3 + \frac{23}{8} \sum_{l=0}^{L-2} (n_l)^3 + (2 + 8)(n_0)^3 \approx 41.
\]

Hence, the total number of floating-point operations per unknown for one V-cycle is 36 and 41 for the 2D and 3D cases, respectively.

**Remark 4.1 (Space complexity of V-cycle GMG)** In GMG Algorithm 3, we need only keep \( \mathbf{u}_l, \mathbf{f}_l \) (\( l = 0, 1, \ldots, L - 1 \)) and \( \mathbf{r}_l \) (\( l = 0, 1, \ldots, L - 2 \)) in the host or device memory. Therefore, we obtain the memory space complexity of GMG Algorithm 3 as follows:

\[
\frac{2 \sum_{l=0}^{L-1} (n_l)^d + \sum_{l=0}^{L-2} (n_l)^d}{(n_0)^d} \approx 4. \tag{4.18}
\]

Equation (4.18) shows that the memory space complexity of GMG Algorithm 3 has about 4 times as many unknowns for both 2D and 3D. Equation (4.18) also shows that the space complexity of the GMG V-cycle is \( O(N) \).

## 5 Numerical Experiment

In this section, we perform several numerical experiments and analyze the performance of GMG as proposed in Algorithm 3.

### 5.1 Environment for Comparisons

Our focal environment is a low-cost commodity-level NVIDIA GPU together with a HP computing workstation. Details in regard to the machine are set out in Table 1. From this table, we notice that the initial and energy-consumption costs of the particular GPU we use is roughly 2 times of the CPU costs.
Table 1: Experiment Environment

| CPU Type     | AMD FX-8150 8-core |
|--------------|---------------------|
| CPU Clock    | 3.6 GHz × 8 cores   |
| CPU Energy Consumption | 85 Watts (idle) ∼ 262 Watts (peak) |
| CPU Price    | 300 US Dollars      |
| Host Memory Size | 16GB                |
| GPU Type     | NVIDIA GeForce GTX 480sp |
| GPU Clock    | 1.4 GHz × 15 × 32 cores |
| GPU Energy Consumption | 141 Watts (idle) ∼ 440 Watts (peak) |
| GPU Price    | 485 US Dollars      |
| Device Memory Size | 1.5GB              |
| Operating System | CentOS 6.2        |
| CUDA Driver  | CUDA 4.1            |
| Host Compiler| gcc 4.4.6           |
| Device Compiler | nvcc 4.1         |

For our numerical experiments, we chose to use AMD FX(tm)-8150 Eight-Core 3.6GHz CPU (its peak performance in double precision is 1.78GFLOPs†) and the NVIDIA GeForce GTX 480 GPU. GTX 480 supports CUDA and it is composed of 15 multiprocessors, each of which has 32 cores (480 cores in total). Each multiprocessor is equipped with 48KB of very fast shared memory, which stores both data and instructions. All the multiprocessors are connected to the global memory, which is understood as an SMP architecture. The global memory is limited to a maximum size of 1.5GB. However, there is also a read-only cache memory called a texture cache, which is bound to a part of the global memory when a code is initiated by the multiprocessors. The main performance parameters of GeForce GTX 480 is described in Table 2‡.

Table 2: Theoretical peak performance of NVIDIA GeForce GTX 480

| Performance Type                  | GFLOPs  |
|-----------------------------------|---------|
| Single-precision performance      | 1300.00 |
| Double-precision performance      | 177.00  |
| Theoretical memory bandwidth      | 177.00  |
| Device to device memory bandwidth | 148.39  |
| Device to host memory bandwidth   | 4.46    |
| Host to host memory bandwidth     | 9.44    |
| Host to device memory bandwidth   | 3.92    |

Remark 5.1 (Multicore effect) We note that, in all our numerical tests on CPUs, we use only a single CPU core in order to provide an unbiased benchmark on CPUs. The reason is that we want to eliminate the effect of different multi-threaded implementations of GMG on multicore CPUs. It is fair to say that on AMD FX8150 (8-core) the speedup of eight-thread GMG (over one thread version) is, in general, 2.0 to 4.0, depending on the algorithm and implementation.

For test purposes, our focus is the simple model problem (2.1) in the 2D and 3D cases. To be more specific, we consider the following two cases:

†Obtained experimentally using LINPACK (http://www.netlib.org/benchmark/linpackc).
‡Numbers in the last four rows are obtained experimentally using the bandwidthtest of CUDA 4.1 SDK.
Example 5.1 For the model problem 2.1, let
\[ \Omega = (0, 1)^2 \subset \mathbb{R}^2, \]
\[ f(x, y) = \sin(\pi x) \sin(\pi y), \quad (x, y) \in \Omega. \]

Example 5.2 For the model problem 2.1, let
\[ \Omega = (0, 1)^3 \subset \mathbb{R}^3, \]
\[ f(x, y, z) = \sin(\pi x) \sin(\pi y) \sin(\pi z), \quad (x, y, z) \in \Omega. \]

The tolerance for the convergence is tol = 10^{-6} and \( \mu_f = \mu_b = 1. \)

5.2 CPU v.s. GPU

First, we test the number of iterations and the discretization error of the finite difference schemes on CPUs and GPUs. Tables 3 and 4 show that both the GPU version and the CPU version of GMG achieve the optimal discretization error, \( O(h^2) \), in \( \mathbb{R}^2 \) and \( \mathbb{R}^3 \). Furthermore, the GPU and the CPU versions take the same number of iterations as each other to reach the given convergence tolerance, i.e., the GPU version is equivalent to the corresponding serial version.

### Table 3: The iteration numbers, discretization errors, and convergence rates of V-cycle in 2D

| \( N \)          | \( \# \) | \( \| u - u_h \| \)  | \( \frac{\| u - u^1_h \|}{\| u - u^1_h \|} \) | \( \# \) | \( \| u - u_h \| \)  | \( \frac{\| u - u^1_h \|}{\| u - u^1_h \|} \) |
|------------------|-------|----------------|-----------------|-------|----------------|-----------------|
| (2^5 + 1)^2      | 11    | 6.250e-6       |                 | 11    | 6.250e-6       |                 |
| (2^6 + 1)^2      | 11    | 1.565e-6       | 3.99            | 11    | 1.565e-6       | 3.99            |
| (2^10 + 1)^2     | 11    | 3.910e-7       | 4.00            | 11    | 3.910e-7       | 4.00            |
| (2^11 + 1)^2     | 11    | 9.719e-8       | 4.02            | 11    | 9.719e-8       | 4.02            |
| (2^12 + 1)^2     | 11    | 2.370e-8       | 4.10            | 11    | 2.370e-8       | 4.10            |

### Table 4: The iteration numbers, discretization errors, and convergence rates of V-cycle in 3D

| \( N \)          | \( \# \) | \( \| u - u_h \| \)  | \( \frac{\| u - u^1_h \|}{\| u - u^1_h \|} \) | \( \# \) | \( \| u - u_h \| \)  | \( \frac{\| u - u^1_h \|}{\| u - u^1_h \|} \) |
|------------------|-------|----------------|-----------------|-------|----------------|-----------------|
| (2^5 + 1)^3      | 15    | 2.713e-4       |                 | 15    | 2.713e-4       |                 |
| (2^6 + 1)^3      | 15    | 6.936e-5       | 3.91            | 15    | 6.936e-5       | 3.91            |
| (2^7 + 1)^3      | 15    | 1.753e-5       | 3.97            | 15    | 1.753e-5       | 3.97            |
| (2^8 + 1)^3      | 15    | 4.404e-6       | 3.98            | 15    | 4.404e-6       | 3.98            |
Second, we compare the computing time for the GPU version and the CPU version of GMG. For the 2D test problem, in the best-case scenario, the GPU version of GMG can achieve 18.49 times speedup, which is 40% of the SpMV operation on the finest grid (see Table 5 and Figure 6). The extent of speedup indicates that when \( L \leq 10 \), the maximum speedup is archived at \( L_\theta = 3 \) or 4; however, when \( L > 10 \), the maximum speedup is archived at \( L_\theta = L \) (all run on GPU). Moreover, the speedup increases as \( L \) increases. In this case, we can obtain 15.17 GFLOPs in double precision, which is 8.6% of the theoretical peak performance of GTX 480 (see Figure 6).

Similarly, for the 3D test problem, in the best-case scenario, we can achieve 15.99 times speedup, which is 61% of the SpMV operation on the finest grid (see Figure 7 and Table 6). Similar to the 2D case above, if \( L \leq 8 \), then \( L_\theta = 2 \) or 3 gives the best speedup. However, if the size of the problem is large enough, then we should run it completely on GPU. In this case, 13.59 GFLOPs double-precision operations are performed every second, which is approximately 7.6% of the peak performance of GTX 480 (see Figure 7).

### Table 5: Wall time, GFLOPs (double precision), and speedup for computing the residual in 2D

| \( L \) | CPU  | GPU   | Speedup |
|--------|------|-------|---------|
|        | time (s) | GFLOPs | time (s) | GFLOPs |   |
| 8      | 1.5440e-4 | 2.57   | 1.6148e-5 | 24.54  | 9.56 |
| 9      | 1.7865e-3 | 0.88   | 4.9640e-5 | 31.81  | 35.99 |
| 10     | 7.8907e-3 | 0.80   | 1.8124e-4 | 34.78  | 43.54 |
| 11     | 3.2138e-2 | 0.78   | 7.0802e-4 | 35.58  | 45.39 |
| 12     | 1.3021e-1 | 0.80   | 2.8167e-3 | 35.75  | 46.23 |
In this subsection, we would like to understand more about which part(s) of the GPU implementation are the bottleneck(s). Tables 7 and 8 show the kernel time and communication time in 2D and 3D, respectively. (In these two tables, Total = Kernel time + Communication time.) The numerical results show that the kernel computation takes 75% to 90% of the total wall time. Furthermore, the portion of communication time decreases as problem size increases if we consider problems with a large degree of freedom.

### Table 7: Kernel time (seconds) and communication time (seconds) in 2D

| \(L(L_\theta = L)\) | 8     | 9     | 10    | 11     | 12  |
|------------------|-------|-------|-------|--------|-----|
| Kernel time      | 5.934e-3 | 1.064e-2 | 3.134e-2 | 1.124e-1 | 4.339e-1 |
| Communication time| 9.070e-4 | 3.491e-3 | 5.884e-3 | 1.522e-2 | 5.169e-2 |
| Communication/Total | 13.26% | 24.70% | 15.81% | 11.93% | 10.64% |
Table 8: Kernel time (seconds) and communication time (seconds) in 3D

| $L(L_\theta = L)$ | 5        | 6        | 7        | 8        |
|-------------------|----------|----------|----------|----------|
| Kernel time       | 8.544e-3 | 2.404e-2 | 9.997e-2 | 7.287e-1 |
| Communication time| 4.938e-4 | 3.716e-3 | 9.574e-3 | 5.141e-2 |
| Communication/Total| 5.46%    | 13.39%   | 8.74%    | 6.59%    |

Tables 9 and 10 show the wall time (ratio to the total kernel time) for each function of one V-cycle in 2D and 3D, respectively. Tables 11 and 12 show the GFLOPs for each function of one V-cycle in 2D and 3D, respectively. The numerical results show that the multicolored Gauss-Seidel smoother (GSforward and GSbackward) counts for more than 50% of the total kernel time. Furthermore, because we are using the multicolored GS smoother, we need to launch the GS kernel several times (equal to the number of colors)—this introduces larger overhead. On the other hand, as we noted, using the weighted Jacobi method does not help, although it could result in better parallelism. Another observation is that computing the Euclidean norm of the residual gets very low efficiency due to its high memory-access/computation rate.

Table 9: Wall time ratio in the kernel time of each subroutine in one V-cycle in 2D

| $L(L_\theta = L)$ | 8        | 9        | 10       | 11       | 12       |
|-------------------|----------|----------|----------|----------|----------|
| Residual          | 16.41%   | 16.35%   | 19.64%   | 21.64%   | 22.11%   |
| GSforward         | 29.42%   | 29.39%   | 25.74%   | 25.57%   | 25.55%   |
| GSbackward        | 23.29%   | 24.55%   | 24.97%   | 24.61%   | 24.85%   |
| Restriction       | 14.82%   | 14.25%   | 8.90%    | 6.53%    | 5.99%    |
| Prolongation      | 11.01%   | 10.47%   | 10.97%   | 10.30%   | 9.92%    |
| Norm              | 5.06%    | 4.99%    | 9.78%    | 11.35%   | 11.59%   |

Table 10: Wall time ratio in the kernel time of each subroutine in one V-cycle in 3D

| $L(L_\theta = L)$ | 5        | 6        | 7        | 8        |
|-------------------|----------|----------|----------|----------|
| Residual          | 12.76%   | 13.90%   | 19.07%   | 25.34%   |
| GSforward         | 33.00%   | 30.59%   | 27.75%   | 27.21%   |
| GSbackward        | 26.71%   | 27.76%   | 26.75%   | 26.78%   |
| Restriction       | 10.03%   | 7.82%    | 5.69%    | 4.38%    |
| Prolongation      | 13.67%   | 16.96%   | 16.57%   | 15.69%   |
| Norm              | 3.82%    | 2.97%    | 4.16%    | 2.79%    |
Table 11: Performance (GFLOPs) of each subroutine in one V-cycle in 2D

| $L(L_\theta = L)$ | 8  | 9  | 10 | 11 | 12 |
|-------------------|----|----|----|----|----|
| Residual          | 6.46 | 8.75 | 21.23 | 26.01 | 27.12 |
| GSforward         | 1.71 | 2.32 | 7.70 | 10.46 | 11.14 |
| GSbackward        | 2.15 | 2.77 | 7.94 | 10.91 | 11.53 |
| Restriction       | 1.37 | 1.92 | 8.95 | 16.44 | 19.09 |
| Prolongation      | 2.57 | 3.62 | 10.01 | 14.33 | 15.87 |
| Norm              | 3.04 | 4.14 | 6.11 | 7.13 | 7.39 |

Table 12: Performance (GFLOPs) of each subroutine in one V-cycle in 3D

| $L(L_\theta = L)$ | 5  | 6  | 7  | 8  |
|-------------------|----|----|----|----|
| Residual          | 6.64 | 19.24 | 28.96 | 24.81 |
| GSforward         | 1.20 | 4.09 | 9.31 | 10.81 |
| GSbackward        | 1.49 | 4.51 | 9.64 | 10.98 |
| Restriction       | 2.58 | 10.49 | 15.46 | 25.78 |
| Prolongation      | 1.25 | 4.82 | 13.32 | 19.43 |
| Norm              | 1.20 | 3.04 | 6.41 | 7.66 |

Tables 13 and 14 show that the memory complexity of Algorithm 3 is $O(N)$ in both the 2D and 3D cases. Furthermore, the constant in $O(N)$ is also very small—less than 4.

Table 13: GPU device memory usage (double precision floating-point numbers) for 2D GMG

| $L$ | 8  | 9  | 10 | 11 | 12 |
|-----|----|----|----|----|----|
| $N$ | 66049 | 263169 | 1050625 | 4198401 | 16785409 |
| Memory usage | 242865 | 966323 | 3855029 | 15399607 | 61557433 |
| Memory usage/$N$ | 3.6770 | 3.6719 | 3.6693 | 3.6680 | 3.6673 |

Table 14: GPU device memory usage (double precision floating-point numbers) for 3D GMG

| $L$ | 5  | 6  | 7  | 8  |
|-----|----|----|----|----|
| $N$ | 35937 | 274625 | 2146689 | 16974593 |
| Memory usage | 119399 | 907337 | 7072779 | 55849869 |
| Memory usage/$N$ | 3.3225 | 3.3039 | 3.2947 | 3.2902 |

5.4 FMG vs. FFT

Now we compare the FFT method with the geometric multigrid method as fast Poisson solution methods. FFT is a direct solver, and multigrid is an iterative solver. Therefore, making a fair comparison between the two is not an easy task. We set up our comparison in the following way: we tested a sequence of FMG methods, each of which had a different number of pre- and post-relaxation sweeps. Then we compared FFT with the most efficient FMG scheme in order
to determine which gives the optimal approximation error. As FFT and FMG require the same amount of data to be transmitted, we only compare the respective kernel times here.

We consider cases with 16 million unknowns in 2D \( (L = 12) \) and 3D \( (L = 8) \). For the 2D case, from Tables 15 and 16 we notice that FMG(1,2) is enough to guarantee the optimal convergence of the approximation error in \( L^2(\Omega) \). On the other hand, for the 3D case, we need to use at least FMG(3,3) in order to obtain the optimal convergence rate (see Tables 17 and 18). Moreover, the optimal FMG is 33% and 23% faster than FFT in 2D and 3D, respectively.

### Table 15: Approximation error \( \|u - u_h\| \) in 2D

| \( L \) | FFT   | FMG(1,1) | FMG(1,2) | FMG(2,2) | FMG(2,3) | FMG(3,3) |
|-------|-------|----------|----------|----------|----------|----------|
| 9     | 1.563e-6 | 1.001e-5 | 1.242e-6 | 1.004e-6 | 7.028e-7 | 7.145e-7 |
| 10    | 3.914e-7 | 2.618e-6 | 3.113e-7 | 2.518e-7 | 1.762e-7 | 1.790e-7 |
| 11    | 9.797e-8 | 6.766e-7 | 7.791e-8 | 6.304e-8 | 4.411e-8 | 4.479e-8 |
| 12    | 2.450e-8 | 1.735e-7 | 1.948e-8 | 1.577e-8 | 1.103e-8 | 1.120e-8 |

### Table 16: Kernel time (seconds) of FFT and FMG in 2D

| \( L \) | FFT   | FMG(1,1) | FMG(1,2) | FMG(2,2) | FMG(2,3) | FMG(3,3) |
|-------|-------|----------|----------|----------|----------|----------|
| 9     | 3.739e-3 | 3.611e-3 | 4.260e-3 | 4.980e-3 | 5.617e-3 | 6.348e-3 |
| 10    | 1.102e-2 | 7.434e-3 | 8.770e-3 | 1.008e-2 | 1.144e-2 | 1.282e-2 |
| 11    | 4.077e-2 | 2.203e-2 | 2.671e-2 | 2.945e-2 | 3.317e-2 | 3.701e-2 |
| 12    | 1.364e-1 | 7.860e-2 | 9.167e-2 | 1.049e-1 | 1.180e-1 | 1.310e-1 |

### Table 17: Approximation error \( \|u - u_h\|_2 \) in 3D

| \( L \) | FFT   | FMG(1,1) | FMG(1,2) | FMG(2,2) | FMG(2,3) | FMG(3,3) |
|-------|-------|----------|----------|----------|----------|----------|
| 5     | 2.841e-4 | 6.509e-3 | 2.733e-3 | 1.246e-3 | 7.873e-4 | 5.296e-4 |
| 6     | 7.100e-5 | 2.685e-3 | 9.469e-4 | 3.930e-4 | 2.426e-4 | 1.608e-4 |
| 7     | 1.774e-5 | 1.032e-3 | 2.988e-4 | 1.125e-4 | 6.751e-5 | 4.394e-5 |
| 8     | 4.437e-6 | 3.803e-4 | 8.880e-5 | 3.049e-5 | 1.784e-5 | 1.145e-5 |

### Table 18: Kernel time (seconds) of FFT and FMG in 3D

| \( L \) | FFT   | FMG(1,1) | FMG(1,2) | FMG(2,2) | FMG(2,3) | FMG(3,3) |
|-------|-------|----------|----------|----------|----------|----------|
| 5     | 5.102e-4 | 1.611e-3 | 1.932e-3 | 2.382e-3 | 2.738e-3 | 3.186e-3 |
| 6     | 1.890e-3 | 3.711e-3 | 4.474e-3 | 5.335e-3 | 6.098e-3 | 6.986e-3 |
| 7     | 5.884e-2 | 1.342e-2 | 1.586e-2 | 1.846e-2 | 2.094e-2 | 2.352e-2 |
| 8     | 1.893e-1 | 8.566e-2 | 1.007e-1 | 1.155e-1 | 1.302e-1 | 1.456e-1 |

### 6 Conclusion

In this work, we studied the performance of GMG on CPU–GPU heterogenous computers. Our numerical results suggest that in the best-case scenario the GPU version of GMG can achieve 18.5
times speed-up in 2D and 16.0 times speed-up in 3D compared with an efficient implementation of multigrid methods on CPUs. When the problem is relatively small we found that the heterogenous algorithm \((0 < L_θ < L)\) usually gives the best computational performance. On the other hand, when the problem size is large enough, then we concluded that it is generally preferable to do the computation on GPUs. We observed the smoothing and computing norm of the residual account for the low floating-point performance and low efficiency of GMG on GPUs. Furthermore, we compared our method with the Fast Fourier Transform in the state-of-the-art cuFFT library. For the test cases with 16 million unknowns \((L = 12 \text{ in 2D and } L = 8 \text{ in 3D})\), we showed that the optimal FMG method is 33\% and 23\% faster than FFT in 2D and 3D, respectively. Of at least equal importance is that GPU is more cost-effective (in terms of initial cost and daily energy consumption) than modern multicore CPUs for geometric multigrid methods.

**Acknowledgements**

The authors would like to thank Dr. Yunrong Zhu from Idaho State University and Dr. Xiaozhe Hu from Penn State University for their helpful comments and suggestions. They are also grateful for the assistance provided by Mr. Xiaqiang Yue and Mr. Zheng Li from Xiangtan University in regard in our numerical experiments.

**Nomenclature**

| Name | type | size | Brief Description |
|------|------|------|-------------------|
| \(d\) | int | scalar | Spatial dimension: 2 for 2D and 3 for 3D |
| \(L\) | int | scalar | Total number of grid levels |
| \(L_θ\) | int | scalar | Critical grid level between CPU and GPU computing |
| \(h_l\) | double | scalar | Grid size of the \(l\)-th level \(h_l = \frac{1}{2^l}\) |
| \(n_l\) | int | scalar | Grid size of each direction of the \(l\)-th level \(n_l = \frac{1}{h_l} + 1\) |
| \(N\) | int | scalar | Number of unknowns \(N = (n_0)^d\) |
| \(μ_f\) | double | scalar | Number of forward relaxation sweeps |
| \(μ_b\) | double | scalar | Number of backward relaxation sweeps |
| \(A_l\) | null | | Grid operator of No. \(l\) level |
| \(R_{l+1}^l\) | null | | Restriction operator from the \(l\)-th level to the \((l+1)\)-th level |
| \(P_{l+1}^l\) | null | | Prolongation operator from the \((l+1)\)-th level to the \(l\)-th level |
| \(\vec{u}_l\) | double | * \((2^{L-l}+1)^d\) | Solution vector of the \(l\)-th level |
| \(\vec{f}_l\) | double | * \((2^{L-l}+1)^d\) | Right-side vector of the \(l\)-th level |
| \(\vec{r}_l\) | double | * \((2^{L-l}+1)^d\) | Residual of the \(l\)-th level \(\vec{r}_l = \vec{f}_l - A_l\vec{u}_l\) |
| \(tols\) | double | scalar | Stopping criterion \(\|\vec{f} - A\vec{u}\| / \|\vec{f}\| < tols\) |
| \(\|\cdot\|\) | | | Shortened symbol for the 2-norm or \(\|\cdot\|_{l2}\) |

**References**

Asanovic, K. et al. (2006). *The Landscape of Parallel Computing Research: A View from Berkeley*. Tech. rep. UCB/EECS-2006-183. EECS Department, University of California, Berkeley.

Barrachina, S. et al. (2009). “Exploiting the capabilities of modern GPUs for dense matrix computations”. In: *Concurrency and Computation Practice and Experience* 21.18, pp. 2457–2477.
Bell, N., S. Dalton, and L. N. Olson (2011). Exposing fine-grained parallelism in algebraic multigrid methods. Tech. rep. NVIDIA Technical Report NVR-2011-002.

Bell, N. and M. Garland (2008). “Efficient Sparse Matrix-Vector Multiplication on CUDA”. In: Memory NVR-2008-004, pp. 1–32.
— (2009). “Implementing sparse matrix-vector multiplication on throughput-oriented processors”. In: Proceedings of the Conference on High Performance Computing Networking Storage and Analysis SC 09. SC ’09 1, p. 1.

Bjørstad, P. E., M. Dryja, and T. Rahman (2003). “Additive Schwarz Methods for Elliptic Mortar Finite Element Problems”. In: Numer. Math. 2003.2, pp. 427–457.

Bjørstad, P. E. et al. (1992). “Efficient Matrix Multiplication on SIMD Computers”. In: SIAM J. Matrix Anal. Appl. 13.1, pp. 386–401.

Bolz, J., I Farmer, and E Grinspun (2003). “Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid”. In: ACM Transactions on Graphics 22, pp. 917–924.

Bramble, J. (1993). Multigrid methods. Chapman & Hall/CRC.

Brandt, A. (1986). “Algebraic multigrid theory: The symmetric case”. In: Applied Mathematics and Computation 19.1-4, pp. 23–56.

Brandt, A., S. McCormick, and J. Ruge (1982). “Algebraic multigrid (AMG) for automatic multigrid solution with application to geodetic computations, Report”. In: Inst. Comp. Studies Colorado State Univ 109, p. 110.

Brandt, A. (2011). Multigrid guide. Tech. rep.

Briggs, W. L., V. E. Henson, and S. F. McCormick (2000). A multigrid tutorial. Second. Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM), pp. xii+193.

Brodtkorb, A. R. et al. (2010). “State-of-the-art in heterogeneous computing”. In: Scientific Programming 18, pp. 1–33.

Buck, I. (2007). “GPU Computing: Programming a Massively Parallel Processor”. In: International Symposium on Code Generation and Optimization (CGO’07), pp. 17–17.

Cao, W. et al. (2010). “Implementing Sparse Matrix-Vector multiplication using CUDA based on a hybrid sparse matrix format”. In: Computer Application and System Modeling ICCASM 2010 International Conference on. Vol. 11. Iccasm. IEEE, pp. V11–161.

CAPS (2012). CAPS available at http://www.caps-entreprise.com.

Carpenter, P. and W. Symon (2009). Issues in Heterogenous GPU Clusters A Historical and Usage Analysis. Tech. rep.

Chamberlain, R. D. et al. (2007). “Application development on hybrid systems”. In: Proceedings of the 2007 ACM/IEEE conference on Supercomputing. SC ’07. Reno, Nevada: ACM, 50:1–50:10.

Chen, G. et al. (2009). “High Performance Computing Via a GPU”. In: International Conference on Information Science and Engineering, pp. 238–241.

Cooley, J. W. and J. W. Tukey (1965). “An Algorithm for the Machine Calculation Complex Fourier Series”. In: Math. Comput 19, pp. 297–301.

Ellble, J. M., N. V. Sahinidis, and P. Vouzis (June 2010). “GPU computing with Kaczmarz’s and other iterative algorithms for linear systems.” In: Parallel computing 36.5-6, pp. 215–231.

Frigo, M. and S. G. Johnson (2005). “The Design and Implementation of FFTW3”. In: Proceedings of the IEEE 93.2. Special issue on “Program Generation, Optimization, and Platform Adaptation”, pp. 216–231.

Georgescu, S. and H. Okuda (2010). “Conjugate Gradients on multiple GPUs”. In: October, pp. 1254–1273.
Green500 (2011). Green500 List, available at http://top500.org/lists/2011/11.
Griebel, M. (1989). “Zur Lösung von Finite-Differenzen- und Finite-Element-Gleichungen mittels der Hierarchischen-Transformations-Mehrgitter-Methode”. PhD thesis. Technische Universität München.
Hackbusch, W. (1985). Multi-grid methods and applications. Springer Verlag.
Heuveline, V., D. Lukarski, and J.-P. Weiss (2011). Enhanced Parallel ILU (p)-based Preconditioners for Multi-core CPUs and GPUs – The Power (q)-pattern Method. Tech. rep.
Heuveline, V. et al. (2011). Parallel Smoothers for Matrix-based Multigrid Methods on Unstructured Meshes Using Multicore CPUs and GPUs. Tech. rep.
Hey, T., S. Tansley, and K. Tolle, eds. (2009). The Fourth paradigm: Data-Intensive Scientific Discovery. Microsoft Research, p. 284.
Inc., C. (2012). Cray Inc. available at http://www.cray.com.
Jeschke, S. and D Cline (2009). “A GPU Laplacian solver for diffusion curves and Poisson image editing”. In: ACM Transactions on Graphics (TOG) 28.5.
Kaushik, D. et al. (2011). “Hybrid Programming Model for Implicit PDE Simulations on Multicore Architectures”. In: pp. 12–21.
Keyes, D. E. (Feb. 2011). “Exaflop/s: The why and the how”. In: Comptes Rendus Mécanique 339.2-3, pp. 70–77.
Knibbe, H., C. Oosterlee, and C. Vuik (July 2011). “GPU implementation of a Helmholtz Krylov solver preconditioned by a shifted Laplace multigrid method”. In: Journal of Computational and Applied Mathematics 236, pp. 281–293.
Köstler, H. et al. (Feb. 2007). “A parallel multigrid accelerated Poisson solver for ab initio molecular dynamics applications”. In: Computing and Visualization in Science 11.2, pp. 115–122.
Lord, R et al. (2008). “A fast and accurate FFT-based method for pricing early-exercise options under Levy processes”, In: SIAM J. Sci. Comput. 30.4, pp. 1678–1705.
Morton, K. W. and D. F. Mayers (2005). Numerical solution of partial differential equations. Second. An introduction. Cambridge: Cambridge University Press, pp. xiv+278.
Nickolls, J. and W. J. Dally (2010). “The GPU Computing Era”. In: Micro IEEE 30.2, pp. 56–69.
NVIDIA (2012a). CUDA 4.1, available at http://www.nvidia.com/object/cuda.
— (2012b). cuFFT Fast Fourier Transform library, available at http://developer.nvidia.com/cufft.
PGI (2012). The Portland Group http://www.pgroup.com/index.htm.
Ruge, J. and K. Stüben (1987). “Algebraic multigrid”. In: Multigrid methods 3, pp. 73–130.
Shi, J. et al. (2009). “GPU friendly fast Poisson solver for structured power grid network analysis”.
In: Proceedings of the 46th Annual Design Automation Conference - DAC ’09. New York, New York, USA: ACM Press, p. 178.
Sturmer, M., H. Köstler, and U. Rude (2008). “A fast full multigrid solver for applications in image processing”. In: Numerical Linear Algebra with Applications 15, pp. 187–200.
Top500, H. (2011). HPC Top500, available at http://top500.org/lists/2011/11.
Trottenberg, U., C. Oosterlee, and A. Schüller (2001). Multigrid. Academic Pr.
Walker, J. S. (1996). Fast Fourier transforms. Second. Studies in Advanced Mathematics. With 1 IBM-PC floppy disk (3.5 inch; HD). Boca Raton, FL: CRC Press, pp. xvi+447.
Weiss, C. (2001). “Data Locality Optimizations for Multigrid Methods on Structured Grids”. PhD thesis.
Wolfe, M. (2012). “The Heterogeneous Programming Jungle”. In: HPC Wire.
Xu, J. (2010). “Fast Poisson-Based Solvers for Linear and Nonlinear PDEs Jinchao Xu”. In: Proceedings Of The International Congress Of Mathematicians 2010. 2000, pp. 2886–2912.

Yang, J, Y Cai, and Q Zhou (2011). “Fast Poisson Solver Preconditioned Method for Robust Power Grid Analysis”. In: Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on, pp. 531–536.