GPU - PETSc for neutron transport equation
Thanks

I thank G. Haase for to help for the outcome of my project, and backing me in my work during the all semester.
Outlines

Introduction page 5
0. CEA page 6
I. Presentation of neutron transport equation page 7
  1.1 Context page 7
  1.2 Neutron transport equation page 8
  1.3 Neutron diffusion equation page 8
II. The linear system to solve page 10
  2.1 Discretization of neutron diffusion equation page 10
  2.2 Domain decomposition page 12
  2.3 The linear system page 13
  2.4 The MINOS solver page 14
  2.5 The linear system used for our project page 15
III Introduction to KSP PETSc methods page 16
  3.1 The numerical methods page 16
IV Introduction to Multigrid with PETSc page 17
  4.1 The Multigrid concept page 17
  4.2 Multigrid with PETSc page 18
V PETSc-CUDA presentation page 20
  5.1 PETSc with CUDA page 20
VI PETSc used in neutronic domain page 21
  6.1 Multigrid code page 21
  6.2 Multigrid Results page 24
  6.3 KSP code page 33
  6.4 KSP Results page 36
VII GPU used in neutronic domain page 44
  7.1 Introduction page 44
  7.2 Conjugate gradient with CUDA-CPU page 46
  7.3 Conjugate gradient with Cublas page 47
  7.4 Results page 48
Conclusion page 55
Bibliography page 56
Webography page 56
Abstract. This paper is about the optimisation for solving the neutron transport equation. We are looking for to implement the conjugate gradient method with GPU programmation, in order to speed-up the resolution of neutron transport equation. We will study different implementations of conjugate-gradient, with CUDA and CUBLAS langage, and also with PETSc-CUDA. Then, we will discuss on multigrid implementation with PETSc, and finally conclude on obtained performances.

Key words. Parallel Computing, Eigenproblem, Conjugate-gradient method, multigrid, Graphics Programmation Unit
**Introduction**

As computer power increases, scientists want to solve more and more complicated problems. They need to optimize both memory requirements and time computation, which is commonly called «High Performance Computing».

HPC is applied to sciences as well as business or data mining and holds a leading place in every domain which uses computer science. More than a concept, HPC is nowadays a standard and is used in several domains, like petrology, sismology, economy, nuclear …

Many codes developed 10 or 15 years ago need to be rewritten in order to take advantage of parallel computers. This is the case of MINOS, a CEA neutronics code. MINOS' target is to solve the neutron transport equation in order to compute the power produced by a power plant.

Indeed, the control of nuclear reaction is based on the neutron transport equation, that's why its resolution is a major purpose.

Moreover, the computation time is essential, so as to intervene fast to reduce or to improve a nuclear reaction, in terms of the needs energy.

Although MINOS is considerably optimised, CEA is still doing research to ameliorate it. The recent researchs about GPU programation prove that using graphic cards to make a part of calculations can considerably speed-up the time calculation. It exists different languages in order to program on graphic cards, such that CUDA or CUBLAS. As the performances of GPU are really satisfying, some libraries and languages such as PETSc or Python are working on improve their performance by using GPU programation. PETSc is a free library which is very adapted to factorize and solve linear systems using parallelism. Indeed, PETSc uses MPI or openMPI and offers a lot of functions for linear and non linear systems.

The subject of this project is to implement the conjugate-gradient method with several languages using GPU programation, in order to use it later for solving the neutron transport equation in one dimension, as in this case, the matrix is symmetric and positiv-definite. We will use the languages CUDA, CUBLAS, and the PETSc library with CUDA, as it extends a lot of numerical methods to solve linear systems, notably several versions of conjugate-gradient method.

The organization of this paper is as follows. First, we will present the neutron transport equation, and then explain how we obtain from this equation a linear system. We will focus next on the generated codes for conjugate-gradient method, and their structure. We will then discuss on multigrid programation with PETSc. Finally, we will study the performances we obtain, and conclude on it.
0. CEA

Based in Ile-de-France, CEA Saclay is a national and european research center. CEA gathers applied research as well as fundamental research. With 5000 researchers, CEA Saclay contributes to improve nuclear central park optimization, but also its functionment and does research to manage nuclear garbages, respecting the environment. To do so, CEA bases its research on simulation, especially with computer science.
I. Presentation of neutron transport equation

The neutron transport equation is fundamental in neutronic science. Indeed, scientists must solve it in order to understand the nuclear reaction in a core. In this part, we will present first the issue and then the neutron transport equation. We will end with the neutron diffusion equation, linked with the first mentioned equation.

1.1 Context

In a nuclear reactor, free neutrons collide with atoms. Two scenarios are possible:
- the neutron is absorbed by the atom (that's an absorption reaction),
- or the atom generates two or three other free neutrons: That's a fission reaction.

Suppose that there are $N$ fissions at time $t$. We call « $K_{eff}$ » the number of neutrons rejected in a fission. In case of a fission, we obtain $N \cdot K_{eff}$ fission at time $t+1$, $N \cdot K_{eff}^2$ fissions at time $t+2$.

We deduce that at $t+k$ time ($k$ is a natural integer), we obtain $N \cdot K_{eff}^k$ fissions: $(N \cdot K_{eff}^k)$ is a geometric suite (its reason is $K_{eff}$) which converges only if $K_{eff} < 1$.
- When $K_{eff} < 1$ (resp. $K_{eff} > 1$), we qualify the nuclear reaction as a under (resp. upper) critical reaction.
- When $K_{eff} = 1$, we qualify it as a critical reaction.

The coefficient $K_{eff}$ determines if nuclear reaction will decrease or increase in the futur. Solving the neutron transport equation gives us the value of $K_{eff}$, that's why we have to solve it the fastest as possible.

On the previous picture, we observe that $K_{eff}$ is equal to 3 for each fission.

The fission phenomena produces heat, which will create water vapor. This vapor is employed to action turbines, which then produces electricity.

As we presented the general functionment of a nuclear core, we will now study the neutron transport equation.
1.2 Neutron transport equation

The neutron transport equation has been established by L. Boltzmann in 1872, and is also called Boltzmann's equation\(^1\). We present it as follows:

\[
- \nabla \cdot \left( \vec{J}(\vec{r}, v, \vec{\Omega}, t) \right) - \Sigma_t(\vec{r}, v, \vec{\Omega}, t)\Phi(\vec{r}, v, \vec{\Omega}, t) + Q(\vec{r}, v, \vec{\Omega}, t) = \frac{1}{v} \frac{\partial}{\partial t} \Phi(\vec{r}, v, \vec{\Omega}, t) \quad \text{\(1\)}
\]

\(^1\) Reference: Précis de neutronique Paul REUSS

Where:
- \(\Phi\) is the neutron flux (mol/m\(^3\)),
- the current is \(\vec{J}\) (mol/m\(^2\)sec).

This equation cannot be solved with some particular conditions, for example, in case of stationary state of Fick's laws (established in 1855). Before introducing the Fick's law, let's see the diffusion definition: «a system tends to homogenize its chemical elements concentrations: this natural phenomena is called diffusion.»

First Fick's law: it relates the diffusive flux to the concentration field, by postulating that the flux goes from regions of high concentration to regions of low concentration. The magnitude of flux is proportional to the concentration gradient. In one dimension, we obtain this equation:

\[
J = -D \frac{\partial \phi}{\partial x}
\]

Where:
- \(D\) is the diffusion coefficient (m\(^2\)/sec).

Second Fick's law: it predicts how diffusion causes the concentration field to change with time. It is represented with this equation:

Where:
- \(x\) is the position.

\[
\frac{\partial \phi}{\partial t} = D \frac{\partial^2 \phi}{\partial x^2}
\]

The use of Fick's law in a core reactor leads to the diffusion approximation: Boltzmann's equation can be simplify and approximate by the neutron diffusion equation.

We will now focus on this equation, which depends upon 7 variables only.

1.3 Neutron diffusion equation

We saw previously the diffusion definition: To illustrate this definition with neutrons exemple, that's means that in a core, the dense neutrons area tends to populate the sparse neutron areas. In the neutron diffusion equation, we introduce a new variable:

\[
\vec{p} = -D \nabla \phi
\]

Where:
- \(p\) is the current.

The neutron diffusion equation is as follows:

Where:
- \(S\)\(\phi\) are the scattering sources,
- \(S\)\(f\) are the fission sources,
- \(R\) is the domain,
- \(\sigma_a\) is the absorption coefficient,
- \(\sigma_f\) is the fission coefficient,
- \(\lambda\) is Keff.

\(^2\) Reference: Pierre Guérin thesis
In fact, this equation doesn't consider the angular variable.

Now we have presented the neutron diffusion equation, and justify its utility, we will focus on the corresponding linear system. Indeed, it is more adapted to use the linear system because of the large panel of numerical methods that can solve the equation.
II. The linear system to solve

In this part, we will present the method used to discretize the equation diffusion, and the decomposition domain method. Then we will explain the obtained linear system and finally, we will present the solver MINOS, the CEA's program to modelize the resolution of linear system.

2.1 Discretization of neutron diffusion equation

Let's consider the previous neutron diffusion equation\(^1\):

\[
\begin{align*}
\frac{\partial \rho}{\partial t} + D \nabla \phi &= 0 \quad \text{on } R \\
\nabla \cdot \rho + \sigma_s \phi &= \frac{1}{\lambda} S_f + S_\phi \quad \text{on } R \\
\phi &= 0 \quad \text{on } \partial R \\
\rho \in H(div, R), \quad \phi \in L^2(R)
\end{align*}
\]

These coupled equations are discretized with Raviart Thomas with Finite elements method. There, we will detail how to obtain the linear system from these equations. We consider:

\[
\overrightarrow{\rho} \in H(div, R) \quad \text{and} \quad \phi \in L^2(R)
\]

satisfying the neutron diffusion equation.

Let multiply the first equation line by a vector \(q\), and the second with a function \(\phi\) (\(\phi\) is a squared-integrable function). We write this system with its variational form, in order to discretize the set of equation (1.1) and (1.3) with a finite element method. We finally obtain the following system:

\[
\begin{align*}
\int_R -\frac{1}{D} \overrightarrow{\rho} \cdot \overrightarrow{q} + \int_R \nabla \cdot \rho \psi &= 0 \quad \forall \psi \in H(div, R) \quad (2.1) \\
\int_R \nabla \cdot \overrightarrow{\rho} \psi + \int_R \sigma_s \phi \psi &= \frac{1}{\lambda} \int_R S_f \psi + \int_R S_\phi \psi \quad \forall \psi \in L^2(R) \quad (2.2) \\
\phi &= 0 \quad \text{on } \partial R \quad (2.3)
\end{align*}
\]

We now apply the Green's formula on equation (2.1):

\[
\begin{align*}
\int_R -\frac{1}{D} \overrightarrow{\rho} \cdot \overrightarrow{q} + \int_R \nabla \cdot \rho \psi &= 0 \quad \forall \psi \in H(div, R) \quad (3.1) \\
\int_R \nabla \cdot \overrightarrow{\rho} \psi + \int_R \sigma_s \phi \psi &= \frac{1}{\lambda} \int_R S_f \psi + \int_R S_\phi \psi \quad \forall \psi \in L^2(R) \quad (3.2) \\
\phi &= 0 \quad \text{on } \partial R \quad (3.3)
\end{align*}
\]
We employ the Raviart Thomas Finite element method to discretize the different functional spaces. The Raviart Thomas method is illustrated as follows:

**Raviart-Thomas: 0 order RT0**

We consider now the following written:

\[ \begin{align*}
\vec{p}_h &= \begin{pmatrix} \tilde{p}_1 & \ldots & \tilde{p}_N \end{pmatrix} \in W_h \\
\varphi_h &= \begin{pmatrix} \varphi_1 \ldots \varphi_N \end{pmatrix} \in V_h
\end{align*} \]

and so we obtain the following matricial system:
\[ B\varphi - AP = 0 \]
\[ T\varphi - \frac{1}{\lambda} BTP = S \]

with:
\[ A_{ij} = \int_{\Omega} \frac{1}{D_h} \frac{p_i}{q_j} \cdot \frac{q_i}{p_j} \quad B_{ij} = \int_{\Omega} \text{div} \frac{p_i}{q_j} \cdot \varphi_j \quad T_{ij} = \int_{\Omega} \sigma \varphi_i \psi_j \]

This system can be rewritten with P as single unknown member:
\[ \varphi = T^{-1} \left( \frac{1}{k_{eff}} S - B^T P \right) \]
and replace the new expression of \( \varphi \) in the previous matrix system:
\[ (B T^{-1} B^T + A)P = BT^{-1}S \frac{1}{k_{eff}} \]
The biggest eigen value of this linear system is \( \frac{1}{k_{eff}} \).

In the Raviart Thomas case, \( p_x \) and \( p_y \) are uncoupled and the system writes:
\[ B = B_x + B_y \quad A = \begin{vmatrix} A_x & -1 \\ -1 & A_y \end{vmatrix} \]
We fix:
\[ W_x = A_x + B_x T^{-1} B_x^T \quad W_y = A_y + B_y T^{-1} B_y^T \]
Finally, we obtain the following linear system:
\[ W_x \quad B_x T^{-1} B_y^T \]
\[ B_y T^{-1} B_x^T \quad W_y \]
To solve this linear system, CEA currently factorizes the previous matrix with Cholesky method, and solve it with a Gauss Seidel per blocks.

In our specific case, the domain decomposition method is used too to solve more precisely the diffusion equation.

### 2.2 Domain decomposition

On every domain, we intervene on the boundary conditions only. Note that in our case, we consider a cartesian mesh. To explain how we decompose the domain, we illustrate it with a scheme:

We consider two domains, R1 and R2 and the normal vectors associated with each domain. In neutron transport equation case, we rewrite the p vector as follows:
\[ p_1 n_1 + \alpha \varphi_1 = - p_2 n_1 \quad p_2 n_2 + \alpha \varphi_2 = - p_1 n_1 \]
and introduce this it in the following system:

\[
\begin{align*}
\int_{\mathbb{R}} & -\frac{1}{D} \vec{p} \cdot \vec{q} + \int_{\mathbb{R}} \nabla \cdot \vec{q} \varphi = 0 \quad \forall \vec{q} \in H(\text{div}, \mathbb{R}) \\
\int_{\mathbb{R}} & \nabla \cdot \vec{p} \psi + \int_{\mathbb{R}} \sigma_a \varphi \psi = \frac{1}{k_{\text{eff}}} \int_{\mathbb{R}} \sigma_f \varphi \psi \quad \forall \psi \in L^2(\mathbb{R})
\end{align*}
\]

With the same operations detailed in the previous paragraph, the linear system which is very similar as the one mentioned before. Indeed, this decomposition domain method is an iterative one: in consequence, only the second member of the system is modify.

All the previous operations contributes to give a linear system, which form is \(Ax = b\). It is easier to solve this kind of system, because we dispose of a lot of numerical methods to solve it. In the next paragraph, we will focus on the obtained linear system, which returns the coefficient \(K_{\text{eff}}\).

### 2.3 The linear system

The following linear system\(^1\) is the result of the decomposition domain and discretization method application, realised on equation in neutron diffusion. This system is a sequential linear system.

\[
\begin{align*}
A_x + & B_x T_x^{-1} B_x^T \\
& B_y T_y^{-1} B_y^T \\
& A_y + B_y T_y^{-1} B_y^T \\
\end{align*}
\]

\(\mathcal{H}'\)

\(^1\)Reference: Pierre guerin thesis

**Characteristics of matrices:**
- \(\mathcal{H}'\) is symmetric positiv defined,
- \(T\) is diagonal, but it depends on the choice of \(W_h\) and \(V_h\),
- \(B\) is bidiagonal,
- \(A\) is a dense matrix (\(A\) is also called reference of coupling of currents matrix)
- \(W\) has the same profil as \(A\),
- \(S\) is the second member.

It results that the biggest eigen value of this linear system is \(\frac{1}{k_{\text{eff}}}\): to solve this linear system is a major issue for the control of a nuclear reaction.

Currently, CEA implements a Cholesky method to factorize the matrix \(\mathcal{H}'\) and then, Gauss Seidel method to solve the linear system.

As the matrice size is huge, we use parallelization to implement the solver of this linear system. Particularly, the MINOS Solver is a parallelised code, developped in C++, which makes this resolution.
2.4 The MINOS solver

The Minos solver contains different C++ programs, which are optimised. Indeed, The C++ structure offers stability and is well adapted for parallelization. We will focus on the linear system representation in Minos only.

2.4.1 Matrix storage

The matrix $H$ is as follows:

Every row contains only three non zero elements, and every bloc is sharing one element. The matrix $H'$ is store as flat profil. We conserve the values of each bloc from off diagonal (as the matrix is symmetric) in an array called 'matCur'. The 'mut' array is an array containing the number off diagonal elements for each row, and 'matCurD' contains all the diagonal values. Note also that $H'$ is symmetric, positiv-definite.

We mentioned it before, the linear system is solved under a Gauss-Seidel method.

In every Gauss Seidel step, the linear system corresponds to a decomposition of $H'$ matrice for each direction. These systems (for each direction) are solved with a Cholesky method. MINOS iterates Cholesky resolution, until the error $\epsilon$ is inferior as a value fixed by the user. Currently, the error is fixed by a limit:

$$\frac{\|S^{n+1} - S^n\|_2}{\|S^{n+1}\|_2} < \epsilon$$

$\epsilon$ Reference pierre guérin thesis

where:

$S^{n+1}$ (resp. $S^n$) is the result obtained at the $n+1$ (resp. $n$) iteration. Right now, the error is from order $10^{-5}$.

MINOS solver uses parallelization to solve the linear system. To be more precise, we use MPI and openMPI, which accelerate computation time.

2.4.2 Parallelisation

The data distribution is parallelised with MPI functions : it presently exists one method to distribute data with parallelism.

All data of matrices and vectors are distributed on all processes, then, we decompose the data under the number of process and each process makes calculations with a part of sent data. It has been proved that the time for communication is more important than the time of calculation, as soon as we use very large matrices and an important number of process.
2.5 The linear system used for our project

As I can not access to the exact CEA matrix and vectors, I choose to use an other symmetric positive definite matrix. Moreover, the storage format of CEA matrix is not optimal, so we loose a considerable time to translate it as a PETSc storage format. In this case, we modify a little bit the matrix structure, and use a tridiagonal symmetric matrix.

That will be better to compare the time calculation, because the time to translate the matrix format will not interfer.

The aim of this project is to implement the conjugate-gradient method, in order to solve the linear system with it. We implement it by using graphic card programmation, so we can speed-up the time calculation. In what follows, we will briefly study the PETSc library, and detail all the KSP methods proposed to solve our linear system.
III Introduction to KSP PETSc methods

PETSc in an abbreviation for «Portable, Extensible Toolkit for Scientific computation». This library provides functions and many tools to implements numerical methods used to solve linear and non linear systems. PETSc is particularly adapted for sparse and dense systems as well as large-scale systems, because most of PETSc's functions are parallelized. In this paragraph, we will introduce the KSP application for solving linear systems.

3.1 The numerical methods

In this part, we will present the diverse numerical methods we chose to solve our linear system. We will first explain theoretically the different numerical methods which are developed.

- The Conjugate Gradient methods:
  This method is an effective means to solve linear systems where the coefficient matrix is symmetric and positive definite.
  To begin, we consider a linear system like $Ax = b$, where $A$ is symmetric and positive definite. We fixe a vector $x^{(0)}$, and successively calculate $x^{(1)}$, $x^{(2)}$, $x^{(3)}$, ..., $x^{(k-1)}$ to generate a sequence of $\{x^{(k)}\}$ to approximate the solution $x$ of $Ax = b$. We decompose $A$ as the followed format: $A = L+D+L^t$ where $L$ is a strictly lower triangular matrix and $D$ a diagonal matrix of the same size as $L$.

  We need next to calculate the matrix $M$, where $M = (D+L)D^{-1}(D+L)^t$.

  We apply next the following algorithm:

  $p^{(0)} = r^{(0)} = b - Ax^{(0)}$  
  $Mr^{(0)} = r^{(0)}$

  until convergence, do:

  $a^{(k)} = (r^{(k)} , r^{(k)}) / (p^{(k)} , Ap^{(k)})$
  $x^{(k+1)} = x^{(k)} + a^{(k)} p^{(k)}$
  $r^{(k+1)} = r^{(k)} - a^{(k)} Ap^{(k)}$

  $Mr^{(k+1)} = r^{(k+1)}$

  $b^{(k)} = (r^{(k+1)} , r^{(k+1)}) / (r^{(k)} , r^{(k)})$

  $p^{(k+1)} = r^{(k+1)} + b^{(k)} p^{(k)} = b - Ax^{(k+1)}$

  we converge when $p^{(k+1)} = 0$.

  This method corresponds to the KSP_TYPE « CG ».

- Conjugate Gradient Method on the Normal Equations
  The cgne solver applies too the CG iterative method, but it is applied to the normal equations without explicitly forming the matrix $A^tA$.

  This method corresponds to the KSP_TYPE « CGNE ». As we saw, the KSP object in PETSc are particularly adapted to solve linear systems.

However, PETSc also developed the multigrid concept, as it is more and more used in HPC domain. PETSc offers a DMMG object, which creates multigrid, very adapted for linear or non linear systems. In what follows, we will study the DMMG concept with PETSc.
IV Introduction to Multigrid with PETSc

PETSc in an abbreviation for «Portable, Extensible Toolkit for Scientific computation». This library provides functions and many tools to implement numerical methods used to solve linear and non-linear systems. PETSc is particularly adapted for sparse and dense systems as well as large-scale systems, because most of PETSc's functions are parallelized. In this paragraph, we will introduce the multigrid application for solving linear systems, and then explain how PETSc implements it.

4.1 The Multigrid concept

To introduce the multigrid concept, we will consider the classic example: Poisson equation in 1 dimension.

\[-\Delta u = f (1), \text{ on } [0,1]\]  \hspace{1cm} (1.1)
\[u(0) = u(1) = 0 \text{ (boundary conditions).} \hspace{1cm} (1.2)\]

We apply the finite difference method (second order) on (1.1), and then obtain the following linear system:

\[-u(i+1) + 2u(i) - u(i-1) = fi \hspace{1cm} (2.1)\]
\[h^2 \text{ (h is the grid size)} \hspace{1cm} (2.2)\]

\[h \text{ is generally } 1/(n+1).\]

With can rewrite these equation with a matrix form:

\[Au = f \hspace{1cm} (3.1)\]
\[u = (u_1, u_2, \ldots, u_{N-1})^T \hspace{1cm} (3.2)\]
\[u(0) = u(N) = 0 \hspace{1cm} (3.3)\]
\[f = (f_1, f_2, \ldots, f_{N-1})^T \hspace{1cm} (3.4)\]

The discretized form of equation (3.1) is as a linear system:

\[A_h u_h = f_h \hspace{1cm} (4.1)\]

Applying Jacobi or Gauss-Seidel algorithm on linear system like (3.1) determine the \(u\) vector solution, which has the following form:

\[u(i) = (u(i+1) + u(i-1) + h^2fi) / 2 ; i=1, \ldots, N \hspace{1cm} (5.1)\]

Note that the Gauss-Seidel method is like Jacobi one, by using for \(u(i+1)\) the previous calculated value (updated for each iteration), until we obtain the convergence. The system \(Au = f\) admit an exact solution, \(u\). We define the error as:

\[e = u - v \hspace{1cm} (6.1)\]

The best example to study the different frequency component of error is to consider the Fourier modes. They have been introduced as initial data for Gauss-Seidel iteration. By considering the Fourier's mode for several frequencies, we can establish the two following deductions:

\[\text{The error of classic iterative methods is smoothed for every iteration. By decomposing the error as a Fourier serie, we conclude that the high frequencies for a mesh with gridsize } h \text{ are better amortized than the lower one.}\]
\[\text{The error low frequencies on a fine mesh will be considered as high frequencies on a coarse mesh, and so being amortized with iterative method.}\]

We illustrate the steps for multigrid resolution of \(Au = f\):
We make several Gauss-Seidel resolutions of $A_h u_h = f_h$ so we obtain a vector $v_h$, which is an approximation of $u$ on the finest grid (pre-smooth).

We calculate the associated residu $r_h = f_h - A_h v_h$.

We consider this residu on a coarser grid (with $2h$ as gridsize) by using a restriction operator $R$.

We solve the following equation: $A_{2h} (u_{2h} - v_{2h}) = A_{2h} e_{2h} = r_{2h}$.

We use a prolongation (or interpolation) operator on $e_{2h}$ on the finest grid to calculate $e_h$.

We modify in consequence the approximation $v_h$: $v_h = v_h + e_h$.

We make several iterations of Gauss-Seidel on $A_h u_h = f_h$, considering as initial solution: $v_h$.

We will now focus our attention on the multigrid with PETSc, as PETSc offers methods to handle with it.

### 4.2 Multigrid with PETSc

Before using Multigrid methods with PETSc, one should know that the multigrid in PETSc is considered as a preconditioner and not as a standalone solver. However, that can be changed by using the PETSc KSP methods.

The multigrid concept is represented by a structure in PETSc, the DMMG structure, which interacts with DA structure, KSP, PC and some others PETSc types (such as Mat, Vec, ...). More generally, DMMG represents an object oriented framework programming style. Each object are virtually created by DMMG, and DMMG runs them.

With PETSc, the multigrid methods work as follows:

Each solver (smoothers and coarse grid solve) is represented by a KSP object, but there is no good reason to explain this. Moreover, we have to use a PCType of PCKSP for the composite preconditioners if we decide to employ the KSP methods.

In these conditions, the DMMG (ie multigrid with PETSc) manages the construction of multigrid preconditioners and multigrid solvers. The biggest advantage with PETSc is that we can easily use PETSc multigrid to solve linear (using KSP methods) or non linear (using SNES methods) systems.

The PETSc Library provides also codes to generate all the parameters of multigrid such as the right hand side (only available for linear systems), but also the Jacobian matrices, the interpolation or restriction operators, and the function evaluations for a given level of discretization (only available for non linear systems). All these access are executed with PETSc functions, which are easy to handle.

What exactly does the DMMG ? That creates a KSP (in case of linear system) or SNES (non linear system) object, and set the PCType by using the DMMG functions which can
access to PC associated to DMMG. For each level, the vectors, restriction (or interpolation) functions and the matrices are created and filled up. Note that, to introduce the values in initial solution, we use the DA structure (allows logically rectangular meshes creation in 1, 2 or 3 dimensions). However, this one is not available with CUDA, it is replaced by DM one. The DM structure works as follows:
That create a KSP or SNES object, then set the preconditioner type. The vectors, and restriction (or interpolation) functions, as well as the matrices are created and filled up for each level.

We will now explain how CUDA is used with the DMMG object.
V PETSc-CUDA presentation

Nowadays PETSc is available with CUDA. However, this version of PETSc is only available in petsc-dev, but will be included in the next PETSc version (3.2). The main advantage of PETSc is that the programmer does not have to implement any CUDA’s code: the programmatnion is still with PETSc functions, and CUDA appears only with the arguments when we run the code. This is still the same idea of PETSc, one code, but several runnings. The other advantage is that anyone can code on GPU, without deep knowledge about CUDA.

5.1 PETSc with CUDA

As the manipulation of CUDA with PETSc is really simple, the installation of PETSc-dev is still the most complicated part of work! One should consider, before installing the PETSc-dev version, which versions of cusp, thrust and CUDA are already installed on computer. Actually, the thrust version 1.4.0 and cusp 0.4.0 are available with PETSc-CUDA, which is a problem as it is complicated to find the good thrust version (there is no historic-versions on official website).

The PETSc-dev version is currently installed on fermi, but there is still a problem of compatibility with the thrust versions. In consequence, I cannot run my KSP code with CUDA, as one of the necessary library to do so was n't correctly installed.

However, I could perfectly run my multigrid code: the DMMG object is particularly adapted for using CUDA. The main advantage of PETSc is that you have to implement your code, and then run it with the PETSc-CUDA options to use it: one code, but two way to run it, is the concept of PETSc-CUDA.

Some operations are directly run on the GPU:

- MatMult(...),
- KSPSolve(...),
- VecAXPY(...),
- VecWAYPX(...),

as well as other vectors operations.

Nowadays, we have to use the jacobi preconditioners if we want to run our code on GPU. Most of KSP types are also available with the GPU.

As we mentioned before, the programmer does not have to consider any CUDA implementation. We use the classic PETSc functions, and just run the code with the PETSc-CUDA options. For example, one should run a PETSc code with da_vec_type cuda if we want that the vector, associated to the DA/DMMG, runs on the GPU.

Now we studied the PETSc-CUDA, as well as CUDA behaviour with PETSc, let's see the different PETSc codes.
VI PETSc used in neutronic domain

The solver MINOS is currently parallelized to optimize the computation time. However, PETSc offers functions which are parallelized: the user doesn't have to consider the MPI communication, which are optimized as well as the CUDA implementation. Besides, the storages format for matrices are optimized and adapted for such systems as neutron transport. We will present first the PETSc multigrid apply to a similar system as neutron transport equation (briefly presented in 2.5), then the codes using only KSP functions to solve the same system.

6.1 Multigrid code

We implemented a program to solve the linear system (presented in 2.5) by using the DMMG functions. This is justify by the PETSc-CUDA implementation: as we said before, the DMMG options allow to run PETSc code on GPU.

The code is implemented as follows:

```c
void ksp_Results(KSP ksp) ; which returns the KSP results,
void ksp_Options(KSP ksp) ; which inserts the differents options to the KSP,
PetscErrorCode ComputeInitialSolution(DMMG dmmg) ; which compute the initial solution for DMMG,
PetscErrorCode ComputeRHS_VecSet(DMMG dmmg,Vec b) ; which compute the right hand side for DMMG,
PetscErrorCode ComputeMatrix_A(DMMG dmmg,Mat jac, Mat A) ; which compute the matrix associated to the DMMG.
```

We won't detail `void ksp_Results(KSP ksp)` ; and `void ksp_Options(KSP ksp)` ; as they are similar as the KSP code presented in 4.4.

This function ComputeInitialSolution(DMMG dmmg) uses the DA structure to acces to the initial solution: in term of perfomance, this is much better. We insert the values to the vector by using a DA structure.

```c
PetscErrorCode ComputeInitialSolution(DMMG dmmg) {
    DA da = (DA) dmmg->dm;
    PetscInt mx,xs,xm,i;
    PetscInt my,ys,ym,j;
    PetscScalar **array;
    DAGetInfo(da, 0, &mx, &my, 0,0,0,0,0,0,0,0,0);
    DAGetCorners(da,&xs,&ys,0,&xm,&ym,0,0,0,0,0,0,0,0,0,0,0);
    Vec x = (Vec) dmmg->x;
    DAVecGetArray(da, x, &array);
    for (j=ys; j<ys+ym; j++){
        for(i=xs; i<xs+xm; i++){
            array[j][i] = 1.0;
        }
    }
}
```
In the function ComputeRHS_VecSet(DMMG dmmg, Vec b), we use the VecSet(...) function, as it presents also good performances gain.

PetscErrorCode ComputeRHS_VecSet(DMMG dmmg, Vec b) {
    PetscPrintf(PETSC_COMM_WORLD, "in rhs VecSet\n");
    VecSet(b, 1.0);
    VecAssemblyBegin(b);
    VecAssemblyEnd(b);
    return(0);
}

the function ComputeMatrix_A(DMMG dmmg, Mat jac, Mat A) insert the values in the matrix associated to the DMMG. We use MatSetValuesStencil, to insert all the values cause this function is particularly adapted for sparse matrix, and use the grid index.

PetscErrorCode ComputeMatrix_A(DMMG dmmg, Mat jac, Mat A) {

    PetscInt m, n;    MatStencil row, col[3];
    PetscScalar v[3];    DA da = (DA) dmmg->dm;
    PetscInt i, j, mx, my, xm, ym, xs, ys;

    DAGetInfo(da, 0, &mx, &my, 0, 0, 0, 0, 0, 0, 0, 0);
    DAGetCorners(da, &xs, &ys, 0, &xm, &ym, 0);
    for (i = xs; i < xs + xm; i++) {
        row.i = i;
        for (j = ys; j < ys + ym; j++) {
            row.j = j;
            if (i == xs) {
                v[0] = 1/(abs(i - j)+1.0);
                col[0].i = i; col[0].j = j;
                v[1] = 1/(abs(i - j)+1.0);
                col[1].i = i; col[1].j = j + 1;
                MatSetValuesStencil(A, 1, &row, 2, col, v, INSERT_VALUES);
            } else {
                if (j == ys) {
                    
                
            
        
    
}
\[ v[0] = \frac{1}{(abs(i-j)+1.0)}; \]
\[ col[0].i = i; col[0].j = j; \]
\[ v[1] = \frac{1}{(abs(i-j)+1.0)}; \]
\[ col[1].i = i; col[1].j = j-1; \]
\[ MatSetValuesStencil(A,1,&row,2,col,v,INSERT_VALUES); \]

}\}\)

\[
\]

\[ MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY); \]
\[ MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY); \]
\[ return(0); \]

int main(int argc,char **argv) {

DMMG *dmmg; DA da;
PetscReal norm; PC pc;

PetscInitialize(&argc,&argv,(char *)0,help);
PetscOptionsSetFromOptions();
DMMGCreate(PETSC_COMM_WORLD,1,PETSC_NULL,&dmmg);
DMMGSetDM(dmmg,(DM)da);
ComputeInitialSolution(*dmmg);
DMMGSetKSP(dmmg,ComputeRHS_VecSet,ComputeMatrix_A);
ksp_Options(DMMGGetKSP(dmmg));
KSPGetPC(DMMGGetKSP(dmmg),&pc);
PCSetType(pc,PCJACOBI);
PCFactorSetShiftType(pc,MAT_SHIFT_POSITIVE_DEFINITE);
DMMGSetUp(dmmg);
DMMGSolve(dmmg);
VecAssemblyBegin(DMMGGetRHS(dmmg));
VecAssemblyEnd(DMMGGetRHS(dmmg));
ksp_Results(DMMGGetKSP(dmmg));
VecAssemblyBegin(DMMGGetx(dmmg));
VecAssemblyEnd(DMMGGetx(dmmg));
DMMGView(dmmg,PETSC_VIEWER_STDOUT_WORLD);
MatMult(DMMGGetJ(dmmg),DMMGGetx(dmmg),DMMGGetr(dmmg));
VecAXPY(DMMGGetr(dmmg),-1.0,DMMGGetRHS(dmmg));
VecNorm(DMMGGetr(dmmg),NORM_2,&norm);
DMMGDestroy(dmmg);
DADestroy(da);
PetscFinalize();
return 0;

We choose to run this code with three different solvers: the classic conjugate gradient (CG), the conjugate gradient squared method (CGS) and finally, the CGNE one, which corresponds to the Conjugate Gradient iterative method. However, the performances obtained with CGS solver are similar to the CG one, so we won't detail the array of results for CGS solver.

### 6.2 Multigrid Results

**CGNE**
- Grid size: 10 000
- Without Cuda

<table>
<thead>
<tr>
<th>level</th>
<th>Time</th>
<th>Flops</th>
<th>Flops/sec</th>
<th>DMMG iterations</th>
<th>Residual norm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.278</td>
<td>21920000</td>
<td>17150000</td>
<td>87</td>
<td>0.0150379</td>
</tr>
<tr>
<td>2</td>
<td>1.310</td>
<td>33440000</td>
<td>25520000</td>
<td>66</td>
<td>0.0185555</td>
</tr>
<tr>
<td>3</td>
<td>1.350</td>
<td>51980000</td>
<td>38510000</td>
<td>51</td>
<td>0.022592</td>
</tr>
<tr>
<td>4</td>
<td>1.403</td>
<td>80050000</td>
<td>57070000</td>
<td>39</td>
<td>0.0277488</td>
</tr>
<tr>
<td>5</td>
<td>1.595</td>
<td>124200000</td>
<td>77890000</td>
<td>30</td>
<td>0.0339612</td>
</tr>
<tr>
<td>6</td>
<td>1.730</td>
<td>192500000</td>
<td>111300000</td>
<td>23</td>
<td>0.0417193</td>
</tr>
<tr>
<td>7</td>
<td>2.148</td>
<td>305100000</td>
<td>142100000</td>
<td>18</td>
<td>0.0504998</td>
</tr>
<tr>
<td>8</td>
<td>2.886</td>
<td>482400000</td>
<td>167200000</td>
<td>14</td>
<td>0.061509</td>
</tr>
<tr>
<td>9</td>
<td>4.266</td>
<td>772900000</td>
<td>181200000</td>
<td>11</td>
<td>0.074435</td>
</tr>
<tr>
<td>10</td>
<td>6.691</td>
<td>1162000000</td>
<td>173700000</td>
<td>8</td>
<td>0.0959991</td>
</tr>
<tr>
<td>11</td>
<td>11.300</td>
<td>1812000000</td>
<td>160400000</td>
<td>6</td>
<td>0.121133</td>
</tr>
</tbody>
</table>
With Cuda

For each PETSc function, we observe that the Megaflop rates is increasing with the number of levels, which is completely normal regarding the size of matrix for each level.

With the first levels (1 to 5), we can conclude to a considerable gain of Megaflops rate for VecDot function. For first level, the difference is about 20. With the biggest level, the Megaflop rate is just 2 times better.

For VecNorm function, the gain is impressive: for the 11 level, the Megaflop rate is more than 10, compared to the rate without using Cuda.
We make the same conclusion about VecAXPY and VecAYPX, the gain is about 7 to 8 times superior.

The gain obtained on KSPSolve is disappointing, as it is only about 2 times better with Cuda. However, the MatMult function shows also good gain performances, about 3 to 10 times better with Cuda (gains are different for each level).

The gain obtained on PCApply function is also very significant, but we can't conclude on a general gain for each level, as the gain is quite different for each level (the gain is about 2 to 3 times superior).

Which is now surprising is that the Time and global Megaflops rates are not so different with or without Cuda. We had the same conclusion with a PETSc example, available in the tutorial of PETSc. However, this conclusion is not available anymore when the grid size increases.

> **grid size: 100 000**

- without Cuda

<table>
<thead>
<tr>
<th>level</th>
<th>Time</th>
<th>Flops</th>
<th>Flops/sec</th>
<th>DMMG iterations</th>
<th>Residual norm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.392</td>
<td>89200000</td>
<td>64070000</td>
<td>35</td>
<td>0.0301563</td>
</tr>
<tr>
<td>2</td>
<td>1.539</td>
<td>139400000</td>
<td>90610000</td>
<td>27</td>
<td>0.0368426</td>
</tr>
<tr>
<td>3</td>
<td>1.798</td>
<td>219800000</td>
<td>122300000</td>
<td>21</td>
<td>0.0447776</td>
</tr>
<tr>
<td>4</td>
<td>2.291</td>
<td>340600000</td>
<td>148700000</td>
<td>16</td>
<td>0.0553796</td>
</tr>
<tr>
<td>5</td>
<td>3.125</td>
<td>522200000</td>
<td>167100000</td>
<td>12</td>
<td>0.0694722</td>
</tr>
<tr>
<td>6</td>
<td>4.840</td>
<td>885400000</td>
<td>183000000</td>
<td>10</td>
<td>0.0802987</td>
</tr>
<tr>
<td>7</td>
<td>7.973</td>
<td>1452000000</td>
<td>182100000</td>
<td>8</td>
<td>0.0959991</td>
</tr>
<tr>
<td>8</td>
<td>13.770</td>
<td>2265000000</td>
<td>164400000</td>
<td>6</td>
<td>0.121133</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VecDot</th>
<th>VecNorm</th>
<th>VecAXPY</th>
<th>VecAYPX</th>
<th>KSPSolve</th>
<th>PCApply</th>
<th>MatMult</th>
<th>MatMultTranspose</th>
</tr>
</thead>
<tbody>
<tr>
<td>2074</td>
<td>1817</td>
<td>1613</td>
<td>980</td>
<td>862</td>
<td>428</td>
<td>651</td>
<td>698</td>
</tr>
<tr>
<td>1608</td>
<td>1826</td>
<td>1564</td>
<td>917</td>
<td>769</td>
<td>390</td>
<td>587</td>
<td>613</td>
</tr>
<tr>
<td>1329</td>
<td>1736</td>
<td>1366</td>
<td>895</td>
<td>720</td>
<td>322</td>
<td>599</td>
<td>607</td>
</tr>
<tr>
<td>1195</td>
<td>1601</td>
<td>1189</td>
<td>821</td>
<td>653</td>
<td>268</td>
<td>557</td>
<td>587</td>
</tr>
<tr>
<td>1134</td>
<td>1556</td>
<td>1139</td>
<td>752</td>
<td>633</td>
<td>260</td>
<td>563</td>
<td>583</td>
</tr>
<tr>
<td>1121</td>
<td>1563</td>
<td>1109</td>
<td>754</td>
<td>625</td>
<td>251</td>
<td>570</td>
<td>589</td>
</tr>
<tr>
<td>1142</td>
<td>1569</td>
<td>1116</td>
<td>756</td>
<td>611</td>
<td>242</td>
<td>560</td>
<td>584</td>
</tr>
<tr>
<td>1112</td>
<td>1570</td>
<td>1124</td>
<td>719</td>
<td>593</td>
<td>227</td>
<td>570</td>
<td>590</td>
</tr>
</tbody>
</table>

- With Cuda
For a 100,000 grid size, we can observe time gain (from less than 10 seconds to 2 seconds). However, the global Megaflops rates are still comparable, the difference is not so significant.

The gain about Megaflops rate are quite the same. For PCApply, we observe it is now about 3 to 4 times better.

### grid size: 1,000,000
- Without Cuda

<table>
<thead>
<tr>
<th>level</th>
<th>Time</th>
<th>Flops</th>
<th>Flops/sec</th>
<th>DMMG iterations</th>
<th>Residual norm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.09203</td>
<td>92800000</td>
<td>100800000</td>
<td>35</td>
<td>0.0301563</td>
</tr>
<tr>
<td>2</td>
<td>0.99900</td>
<td>145000000</td>
<td>145200000</td>
<td>27</td>
<td>0.0368426</td>
</tr>
<tr>
<td>3</td>
<td>1.19400</td>
<td>228600000</td>
<td>191400000</td>
<td>21</td>
<td>0.0447776</td>
</tr>
<tr>
<td>4</td>
<td>1.58500</td>
<td>354200000</td>
<td>223500000</td>
<td>16</td>
<td>0.0553796</td>
</tr>
<tr>
<td>5</td>
<td>2.30900</td>
<td>543000000</td>
<td>235200000</td>
<td>12</td>
<td>0.0694722</td>
</tr>
<tr>
<td>6</td>
<td>3.75400</td>
<td>920600000</td>
<td>245200000</td>
<td>10</td>
<td>0.0802987</td>
</tr>
<tr>
<td>7</td>
<td>6.58600</td>
<td>1509000000</td>
<td>229200000</td>
<td>8</td>
<td>0.0959991</td>
</tr>
<tr>
<td>8</td>
<td>11.84000</td>
<td>2354000000</td>
<td>198800000</td>
<td>6</td>
<td>0.121133</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VecDot</th>
<th>VecNorm</th>
<th>VecAXPY</th>
<th>VecAYPX</th>
<th>KSPSolve</th>
<th>PCApply</th>
<th>MatMult</th>
<th>MatMultTranspose</th>
</tr>
</thead>
<tbody>
<tr>
<td>544</td>
<td>1140</td>
<td>5443</td>
<td>5750</td>
<td>849</td>
<td>1602</td>
<td>4833</td>
<td>344</td>
</tr>
<tr>
<td>823</td>
<td>2172</td>
<td>6835</td>
<td>7805</td>
<td>1017</td>
<td>1500</td>
<td>5220</td>
<td>377</td>
</tr>
<tr>
<td>1150</td>
<td>4114</td>
<td>7962</td>
<td>9568</td>
<td>1136</td>
<td>1515</td>
<td>5375</td>
<td>397</td>
</tr>
<tr>
<td>1418</td>
<td>7063</td>
<td>7868</td>
<td>10732</td>
<td>1156</td>
<td>1273</td>
<td>5062</td>
<td>397</td>
</tr>
<tr>
<td>1636</td>
<td>10353</td>
<td>7949</td>
<td>11452</td>
<td>1142</td>
<td>1061</td>
<td>4459</td>
<td>397</td>
</tr>
<tr>
<td>1805</td>
<td>15393</td>
<td>7754</td>
<td>11821</td>
<td>1140</td>
<td>925</td>
<td>4122</td>
<td>405</td>
</tr>
<tr>
<td>1885</td>
<td>21564</td>
<td>7261</td>
<td>11995</td>
<td>1089</td>
<td>781</td>
<td>3691</td>
<td>400</td>
</tr>
<tr>
<td>1940</td>
<td>27020</td>
<td>6431</td>
<td>12060</td>
<td>1034</td>
<td>638</td>
<td>3073</td>
<td>406</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VecDot</th>
<th>VecNorm</th>
<th>VecAXPY</th>
<th>VecAYPX</th>
<th>KSPSolve</th>
<th>PCApply</th>
<th>MatMult</th>
<th>MatMultTranspose</th>
</tr>
</thead>
<tbody>
<tr>
<td>1143</td>
<td>1557</td>
<td>1160</td>
<td>763</td>
<td>640</td>
<td>261</td>
<td>559</td>
<td>583</td>
</tr>
<tr>
<td>1123</td>
<td>1556</td>
<td>1133</td>
<td>777</td>
<td>632</td>
<td>257</td>
<td>567</td>
<td>589</td>
</tr>
<tr>
<td>1137</td>
<td>1565</td>
<td>1137</td>
<td>751</td>
<td>616</td>
<td>245</td>
<td>557</td>
<td>583</td>
</tr>
<tr>
<td>1088</td>
<td>1571</td>
<td>1132</td>
<td>701</td>
<td>592</td>
<td>229</td>
<td>552</td>
<td>580</td>
</tr>
</tbody>
</table>

- With Cuda
For a 1 000 000 grid size, the time gain is about 1 to 2 seconds. The gain about the Vec operations are still very satisfiable, (about 20 times still for VecAYPX).

⇒ grid size: 10 000 000

- Without Cuda

<table>
<thead>
<tr>
<th>level</th>
<th>Time</th>
<th>Flops</th>
<th>Flops/sec</th>
<th>DMMG iterations</th>
<th>Residual norm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.38700</td>
<td>408000000</td>
<td>294200000</td>
<td>15</td>
<td>0.0582586</td>
</tr>
<tr>
<td>2</td>
<td>2.26000</td>
<td>618000000</td>
<td>273500000</td>
<td>11</td>
<td>0.074435</td>
</tr>
<tr>
<td>3</td>
<td>4.04800</td>
<td>1038000000</td>
<td>256400000</td>
<td>9</td>
<td>0.0873468</td>
</tr>
<tr>
<td>4</td>
<td>7.45700</td>
<td>1670000000</td>
<td>223900000</td>
<td>7</td>
<td>0.106906</td>
</tr>
</tbody>
</table>

For a 1 000 000 grid size, the time gain is about 1 to 2 seconds. The gain about the Vec operations are still very satisfiable, (about 20 times still for VecAYPX).

- With Cuda

<table>
<thead>
<tr>
<th>level</th>
<th>Time</th>
<th>Flops</th>
<th>Flops/sec</th>
<th>DMMG iterations</th>
<th>Residual norm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.663</td>
<td>1670000000</td>
<td>250700000</td>
<td>6</td>
<td>0.121133</td>
</tr>
</tbody>
</table>

We have the same conclusion as before for 10 000 000 grid size: the gain for each functions are still the same.
We can conclude, regarding the previous results, that Cuda with PETSc offers very good gain performances for Megaflops rates (for each detailed function). However, the global rates for time and Megaflops rates are disappointing, we were expecting more.

- **Time**
- **Mgflop rates**
- **Mgflop rates for each function**
- **Iterations**

We run the same code with a different solver, the CG to compare the results with the CGNE solver: the question was, does the solver have an influence on the results? Can we obtain different gains just because of solver?

- **CG**
  - Grid size: 10 000
    - Without Cuda

<table>
<thead>
<tr>
<th>level</th>
<th>Time</th>
<th>Flops</th>
<th>Flops/sec</th>
<th>DMMG iterations</th>
<th>Residual norm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.326</td>
<td>63820000</td>
<td>48130000</td>
<td>354</td>
<td>0.0009998738</td>
</tr>
<tr>
<td>2</td>
<td>1.361</td>
<td>90650000</td>
<td>66620000</td>
<td>251</td>
<td>0.00140858</td>
</tr>
<tr>
<td>3</td>
<td>1.404</td>
<td>12810000</td>
<td>91270000</td>
<td>177</td>
<td>0.00199748</td>
</tr>
<tr>
<td>4</td>
<td>1.486</td>
<td>18290000</td>
<td>12310000</td>
<td>126</td>
<td>0.00280598</td>
</tr>
<tr>
<td>5</td>
<td>1.620</td>
<td>25940000</td>
<td>16010000</td>
<td>89</td>
<td>0.00397251</td>
</tr>
<tr>
<td>6</td>
<td>1.885</td>
<td>36910000</td>
<td>19580000</td>
<td>63</td>
<td>0.00561196</td>
</tr>
<tr>
<td>7</td>
<td>2.355</td>
<td>53100000</td>
<td>22550000</td>
<td>45</td>
<td>0.00785674</td>
</tr>
<tr>
<td>8</td>
<td>3.155</td>
<td>76270000</td>
<td>24180000</td>
<td>32</td>
<td>0.010485</td>
</tr>
<tr>
<td>9</td>
<td>4.565</td>
<td>11110000</td>
<td>24330000</td>
<td>23</td>
<td>0.0153719</td>
</tr>
<tr>
<td>10</td>
<td>7.020</td>
<td>15770000</td>
<td>22460000</td>
<td>16</td>
<td>0.0220971</td>
</tr>
<tr>
<td>11</td>
<td>11.840</td>
<td>24160000</td>
<td>20410000</td>
<td>12</td>
<td>0.0294628</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VecDot</th>
<th>VecNorm</th>
<th>VecAXPY</th>
<th>VecAYPX</th>
<th>KSPSolve</th>
<th>PCApply</th>
<th>MatMult</th>
</tr>
</thead>
<tbody>
<tr>
<td>2123</td>
<td>1837</td>
<td>1689</td>
<td>992</td>
<td>1057</td>
<td>463</td>
<td>679</td>
</tr>
<tr>
<td>2058</td>
<td>1850</td>
<td>1703</td>
<td>983</td>
<td>1057</td>
<td>461</td>
<td>680</td>
</tr>
<tr>
<td>2117</td>
<td>1845</td>
<td>1695</td>
<td>989</td>
<td>1059</td>
<td>459</td>
<td>678</td>
</tr>
<tr>
<td>2116</td>
<td>1848</td>
<td>1689</td>
<td>985</td>
<td>1047</td>
<td>450</td>
<td>668</td>
</tr>
<tr>
<td>2044</td>
<td>1828</td>
<td>1605</td>
<td>979</td>
<td>984</td>
<td>409</td>
<td>618</td>
</tr>
<tr>
<td>1602</td>
<td>1780</td>
<td>1470</td>
<td>907</td>
<td>909</td>
<td>379</td>
<td>596</td>
</tr>
<tr>
<td>1213</td>
<td>1666</td>
<td>1210</td>
<td>816</td>
<td>804</td>
<td>302</td>
<td>587</td>
</tr>
<tr>
<td>1134</td>
<td>1554</td>
<td>1168</td>
<td>758</td>
<td>751</td>
<td>259</td>
<td>569</td>
</tr>
<tr>
<td>1078</td>
<td>1553</td>
<td>1103</td>
<td>722</td>
<td>712</td>
<td>237</td>
<td>552</td>
</tr>
<tr>
<td>1091</td>
<td>1567</td>
<td>1100</td>
<td>717</td>
<td>704</td>
<td>231</td>
<td>556</td>
</tr>
<tr>
<td>1099</td>
<td>1571</td>
<td>1114</td>
<td>737</td>
<td>696</td>
<td>224</td>
<td>554</td>
</tr>
</tbody>
</table>
Clearly, the Megaflops gains for each function are the same, but we notify a significant difference about the time, which is better than CGNE solver. The solver CG is about ten times faster as CGNE solver.

\[ \begin{array}{|c|c|c|c|c|c|}
\hline
\text{level} & \text{Time} & \text{Flops} & \text{Flops/sec} & \text{DMMG iterations} & \text{Residual norm} \\
\hline
1 & 0.1464 & 673700000 & 460000000 & 354 & 0.000998738 \\
2 & 0.1429 & 956900000 & 669800000 & 251 & 0.00140858 \\
3 & 0.1396 & 1352000000 & 968900000 & 177 & 0.00199748 \\
4 & 0.1410 & 1931000000 & 1369000000 & 126 & 0.00280598 \\
5 & 0.1447 & 2738000000 & 1892000000 & 89 & 0.00397251 \\
6 & 0.1553 & 3896000000 & 2509000000 & 63 & 0.00561196 \\
7 & 0.1793 & 5605000000 & 3125000000 & 45 & 0.00785674 \\
8 & 0.2245 & 8049000000 & 3585000000 & 32 & 0.0110485 \\
9 & 0.3199 & 1172000000 & 3665000000 & 23 & 0.0153719 \\
10 & 0.5056 & 1664000000 & 3290000000 & 16 & 0.0220971 \\
11 & 11.3200 & 2549000000 & 2252000000 & 12 & 0.0294628 \\
\hline
\end{array} \]

\[ \begin{array}{|c|c|c|c|c|c|c|}
\hline
\text{VecDot} & \text{VecNorm} & \text{VecAXPY} & \text{VecAYPX} & \text{KSPSolve} & \text{PCApply} & \text{MatMult} \\
\hline
125 & 125 & 1455 & 1272000000 & 339000000 & 720000000 & 3003000000 \\
24100000 & 24200000 & 260800000 & 236200000 & 640000000 & 1231000000 & 4232000000 \\
4800000 & 48300000 & 423700000 & 404000000 & 121300000 & 183500000 & 5542000000 \\
9200000 & 94500000 & 589000000 & 549200000 & 669400000 & 212200000 & 2129000000 \\
17520000 & 18200000 & 760900000 & 755400000 & 345000000 & 212100000 & 7458000000 \\
32030000 & 34760000 & 902700000 & 941700000 & 497400000 & 199700000 & 7738000000 \\
55410000 & 63990000 & 984000000 & 1066200000 & 619600000 & 173200000 & 7673000000 \\
85220000 & 106790000 & 1007400000 & 1144800000 & 665500000 & 143300000 & 7330000000 \\
118780000 & 167230000 & 997400000 & 1186800000 & 641000000 & 112700000 & 6744000000 \\
146570000 & 229210000 & 948000000 & 1209300000 & 559000000 & 853000000 & 5836000000 \\
166820000 & 284430000 & 885200000 & 1214300000 & 439500000 & 648000000 & 4609000000 \\
\hline
\end{array} \]

Clearly, the Megaflops gains for each function are the same, but we notify a significant difference about the time, which is better than CGNE solver. The solver CG is about ten times faster as CGNE solver.

- With Cuda

- Without Cuda
With Cuda

<table>
<thead>
<tr>
<th>VecDot</th>
<th>VecNorm</th>
<th>VecAXPY</th>
<th>VecAYPX</th>
<th>KSPSolve</th>
<th>PCApply</th>
<th>MatMult</th>
</tr>
</thead>
<tbody>
<tr>
<td>2091</td>
<td>1838</td>
<td>1675</td>
<td>984</td>
<td>1036</td>
<td>443</td>
<td>661</td>
</tr>
<tr>
<td>1947</td>
<td>1832</td>
<td>1592</td>
<td>960</td>
<td>970</td>
<td>409</td>
<td>611</td>
</tr>
<tr>
<td>1520</td>
<td>1778</td>
<td>1337</td>
<td>895</td>
<td>883</td>
<td>361</td>
<td>596</td>
</tr>
<tr>
<td>1197</td>
<td>1598</td>
<td>1200</td>
<td>810</td>
<td>769</td>
<td>273</td>
<td>557</td>
</tr>
<tr>
<td>1134</td>
<td>1548</td>
<td>1130</td>
<td>766</td>
<td>742</td>
<td>257</td>
<td>563</td>
</tr>
<tr>
<td>1123</td>
<td>1561</td>
<td>1121</td>
<td>747</td>
<td>733</td>
<td>252</td>
<td>569</td>
</tr>
<tr>
<td>1139</td>
<td>1569</td>
<td>1122</td>
<td>800</td>
<td>721</td>
<td>239</td>
<td>560</td>
</tr>
<tr>
<td>1109</td>
<td>1571</td>
<td>1117</td>
<td>725</td>
<td>690</td>
<td>213</td>
<td>571</td>
</tr>
</tbody>
</table>

Without Cuda

For 100 000 grid size, we conclude on the same gain rates for each functions. The time gain is about 2 times better with Cuda, except for last levels.

**grid size: 1 000 000**

<table>
<thead>
<tr>
<th>level</th>
<th>Time</th>
<th>Flops</th>
<th>Flops/sec</th>
<th>DMMG iterations</th>
<th>Residual norm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.8964</td>
<td>2139000000</td>
<td>2368000000</td>
<td>112</td>
<td>0.00315673</td>
</tr>
<tr>
<td>2</td>
<td>0.9385</td>
<td>3072000000</td>
<td>3273000000</td>
<td>80</td>
<td>0.00441942</td>
</tr>
<tr>
<td>3</td>
<td>0.1080</td>
<td>4330000000</td>
<td>4010000000</td>
<td>56</td>
<td>0.00631345</td>
</tr>
<tr>
<td>4</td>
<td>0.1386</td>
<td>6238000000</td>
<td>4502000000</td>
<td>40</td>
<td>0.00883883</td>
</tr>
<tr>
<td>5</td>
<td>0.1974</td>
<td>8838000000</td>
<td>4477000000</td>
<td>28</td>
<td>0.0126269</td>
</tr>
<tr>
<td>6</td>
<td>0.3223</td>
<td>1282000000</td>
<td>3979000000</td>
<td>20</td>
<td>0.0176777</td>
</tr>
<tr>
<td>7</td>
<td>0.5622</td>
<td>1836000000</td>
<td>3265000000</td>
<td>14</td>
<td>0.0252538</td>
</tr>
<tr>
<td>8</td>
<td>1.047e+01</td>
<td>2700000000</td>
<td>2578000000</td>
<td>10</td>
<td>0.0353553</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>level</th>
<th>Time</th>
<th>Flops</th>
<th>Flops/sec</th>
<th>DMMG iterations</th>
<th>Residual norm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.398</td>
<td>6580000000</td>
<td>2744000000</td>
<td>36</td>
<td>0.00982093</td>
</tr>
<tr>
<td>2</td>
<td>3.507</td>
<td>9660000000</td>
<td>2755000000</td>
<td>26</td>
<td>0.0135982</td>
</tr>
<tr>
<td>3</td>
<td>5.462</td>
<td>13660000000</td>
<td>2501000000</td>
<td>18</td>
<td>0.0196419</td>
</tr>
<tr>
<td>4</td>
<td>9.228</td>
<td>20220000000</td>
<td>2191000000</td>
<td>13</td>
<td>0.0271964</td>
</tr>
</tbody>
</table>
With Cuda

<table>
<thead>
<tr>
<th>level</th>
<th>Time</th>
<th>Flops</th>
<th>Flops/sec</th>
<th>DM MG iterations</th>
<th>Residual norm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1152</td>
<td>695000000</td>
<td>603000000</td>
<td>36</td>
<td>0.00982093</td>
</tr>
<tr>
<td>2</td>
<td>0.1909</td>
<td>1020000000</td>
<td>534000000</td>
<td>26</td>
<td>0.0135982</td>
</tr>
<tr>
<td>3</td>
<td>0.3441</td>
<td>1442000000</td>
<td>419100000</td>
<td>18</td>
<td>0.0196419</td>
</tr>
<tr>
<td>4</td>
<td>0.6458</td>
<td>2134000000</td>
<td>330500000</td>
<td>13</td>
<td>0.0271964</td>
</tr>
</tbody>
</table>

With 1 000 000 grid size, we can observe that time computation is much better with Cuda: the difference is very significant.

- grid size: 10 000 000

Without Cuda

<table>
<thead>
<tr>
<th>level</th>
<th>Time</th>
<th>Flops</th>
<th>Flops/sec</th>
<th>DM MG iterations</th>
<th>Residual norm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.121</td>
<td>2260000000</td>
<td>317400000</td>
<td>12</td>
<td>0.0294628</td>
</tr>
</tbody>
</table>

With Cuda

<table>
<thead>
<tr>
<th>level</th>
<th>Time</th>
<th>Flops</th>
<th>Flops/sec</th>
<th>DM MG iterations</th>
<th>Residual norm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1141</td>
<td>1571</td>
<td>1143</td>
<td>746</td>
<td>698</td>
</tr>
</tbody>
</table>
One can conclude also, for each grid size level, about the Megaflop rate from KSPSolve: it much more than the one with CGNE, and for each grid size, the Megaflop rate is at least 2 times better with Cuda.

- Time
- Mgflpop rates
- Mgflpop rates for each function
- Iterations

PETSc presents the advantage that we can pass the KSP type as a parameter, so we have one code for three different solvers. We will study in the next part the developed code.

### 6.3 KSP code

The code is implement as follows: we established two functions: Translate Matrix H(int matrix size) which create and insert the values of matrix, and CG_Solve(Mat mat, Vec b), which solves the linear system Ax = b and insert the solution in right hand side vector because of gain memory.

Let's detailed first the function Translate Matrix H(...):

We pass as parameter the size of matrix, so we can allocate the non zero structure for PETSc storage format. Actually, by storing the good non zero structure, we can obtain a good gain performance.

The array nnz (PetscInt nnz[matrix size]) contains the number of non zero elements for each row. In our case, every row contains three non zero elements, just the first and last one contains two non zero elements. The array v (PetscScalar v[3]) contains the non zero elements to insert for each row. Note that we use the PETSc types, because of performances. We then insert the values in nnz array and create the matrix by using the following functions:

```c
MatCreateSeqAIJ(PETSC_COMM_WORLD,matrix_size,matrix_size,PETSC_DECIDE,nnz,&mat);
MatSetOption(mat,MAT_SYMMETRIC,PETSC_TRUE);
MatZeroEntries(mat);
MatSetFromOptions(mat);
```

With these functions, we create a sparse matrix, symmetric (because of options). The function MatSetFromOptions(...) is used to apply the options passed as arguments. We now insert the values by using the function MatSetValues(...), it is very performant because we insert all the non zero element for each row. We used before MatSetValue(...), which insert just on value, and observe that the time computation was very long, so we...
optimized the code by insert all the values for each row.

Mat Translate_Matrix_H(int matrix_size)
{
    Mat mat;
    PetscInt idxm[1],idxn[3];
    PetscScalar v[3];
    PetscInt nnz[matrix_size];
    PetscFunctionBegin;

    MatCreate(PETSC_COMM_WORLD,&mat);
    nnz[0] = 2;
    nnz[matrix_size-1] = 2;
    for(i=1;i<matrix_size-1;i++)
        { nnz[i] = 3; }

    MatCreateSeqAIJ(PETSC_COMM_WORLD,matrix_size,
                    matrix_size,PETSC_DECIDE,nnz,&mat);
    MatSetOption(mat,MAT_SYMMETRIC,PETSC_TRUE);
    MatZeroEntries(mat);
    MatSetFromOptions(mat);
    idxm[0] = 0;
    idxn[0] = 0;
    v[0] = 1.0;
    idxn[1] = 1;
    v[1] = 0.5;
    MatSetValues(mat,1,idxm,2,idxn,v,INSERT_VALUES);
    for(i=1;i<matrix_size-1;i++)
        { idxm[0] = i;
          idxn[0] = i-1;
          v[0] = 0.5;
          idxn[1] = i;
          v[1] = 1.0;
          idxn[2] = i+1;
          v[2] = 0.5;
          MatSetValues(mat,1,idxm,3,idxn,v,INSERT_VALUES);
        }
    idxm[0] = matrix_size-1;
    idxn[0] = matrix_size-2;
    v[0] = 0.5;
    idxn[1] = matrix_size-1;
    v[1] = 1.0;
    MatSetValues(mat,1,idxm,2,idxn,v,INSERT_VALUES);

    MatAssemblyBegin(mat,MAT_FINAL_ASSEMBLY);
    MatAssemblyEnd(mat,MAT_FINAL_ASSEMBLY);
    return mat;
}

We will now detail the function which solves the linear system. The function CG_Solve(Mat mat, Vec b) has as parameter the matrix and right hand side vector. We create the KSP context by using the function KSPCreate(...), then add a preconditioner with KSPGetPC(...). We can add also some options with KSPSetFromOptions(...): it considers the arguments passed when we run the code, such as the KSP type (CG, CGNE or CGS). The function KSPSolve(...) solve the linear system Ax = b. In our case, the final solution is stored in right hand side vector b.

KSP CG_Solve(Mat mat, Vec b)
{  
    PC pc;  
    KSP ksp;  
    PetscReal normVector;  
    KSPConvergedReason reason;  
    int its;

    PetscFunctionBegin;

    KSPCreate(MPI_COMM_WORLD,&ksp);
    KSPSetOperators(ksp,mat,mat,DIFFERENT_NONZERO_PATTERN);
    KSPSetInitialGuessNonzero(ksp,PETSC_TRUE);
    KSPGetPC(ksp,&pc);
    PCSetType(pc,PCJACOBI);
    PCFactorSetShiftType(pc,MAT_SHIFT_POSITIVE_DEFINITE);
    KSPSetFromOptions(ksp);
    KSPSetNormType(ksp,KSP_NORM_PRECONDITIONED);
    KSPSetUp(ksp);

    KSPSetTolerances(ksp,0.000010,0.000000,10000.000000,10000);
    KSPSolve(ksp,b,b);

    KSPGetConvergedReason(ksp,&reason);
    if (reason==KSP_DIVERGED_INDEFINITE_PC) {
        PetscPrintf(PETSC_COMM_WORLD, "Divergence because of
        indefinite preconditioner;\n        Run the executable again but with '-pc_factor_shift_type
        POSITIVE_DEFINITE' option.\n",PETSC_VIEWER_STDOUT_WORLD);
    } else{  
        if (reason<0) {
            PetscPrintf(PETSC_COMM_WORLD, "Other kind of
            divergence: this should not happen:
            %fn",reason,PETSC_VIEWER_STDOUT_WORLD);
        } else {  
            KSPGetIterationNumber(ksp,&its);
            printf("\nConvergence in %d iterations.\n",(int)its);
        }
    }

    VecNorm(b,NORM_2,&normVector);
    return ksp;
}

int main(int argc,char **argv){

    PetscInt n = 10;
    PetscMPIInt size;
    PetscErrorCode ierr;
    int matrix_size=524288;
    Mat A;
    Vec b;
    KSP ksp;

    PetscInitialize(&argc,&argv,PETSC_NULL,help);
MPI_Comm_size(PETSC_COMM_WORLD,&size);

A=Translate_Matrix_H(matrix_size);

ierr = VecCreate(PETSC_COMM_WORLD,&b);CHKERRQ(ierr);
ierr = VecSetSizes(b,PETSC_DECIDE,matrix_size);CHKERRQ(ierr);
ierr = VecSetFromOptions(b);CHKERRQ(ierr);
ierr = VecSet(b,1.0);CHKERRQ(ierr);

ksp=CG_Solve(A,b);
KSPDestroy(ksp);
MatDestroy(A);
VecDestroy(b);

PetscFinalize();
return 0;

} 

In our case, we decide to run this code with three different solvers (CG,CGNE and CGS).

6.4 KSP Results

We will just present some results, as this code is not run with Cuda.

CG
Matrix size : 10 000

Convergence in 354 iterations.

*****************Results : KSP******************

Let's see the ksp context
KSP Object:
  type: cg
  maximum iterations=10000
  tolerances: relative=1e-05, absolute=0, divergence=10000
  left preconditioning
  using nonzero initial guess
  using PRECONDITIONED norm type for convergence test
PC Object:
  type: jacobi
  linear system matrix = precond matrix:
Matrix Object:
  type: seqaij
  rows=10000, cols=10000
  total: nonzeros=29998, allocated nonzeros=29998
  total number of mallocs used during MatSetValues calls =0
  not using I-node routines

******************************************************************************
### PETSc Performance Summary:

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Max/Min</th>
<th>Avg</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (sec)</td>
<td>$1.329e+00$</td>
<td>1.00000</td>
<td>$1.329e+00$</td>
<td></td>
</tr>
<tr>
<td>Objects</td>
<td>$1.300e+01$</td>
<td>1.00000</td>
<td>$1.300e+01$</td>
<td></td>
</tr>
<tr>
<td>Flops</td>
<td>$6.382e+07$</td>
<td>1.00000</td>
<td>$6.382e+07$</td>
<td>$6.382e+07$</td>
</tr>
<tr>
<td>Flops/sec</td>
<td>$4.803e+07$</td>
<td>1.00000</td>
<td>$4.803e+07$</td>
<td>$4.803e+07$</td>
</tr>
<tr>
<td>MPI Messages:</td>
<td>$0.000e+00$</td>
<td>0.00000</td>
<td>$0.000e+00$</td>
<td>$0.000e+00$</td>
</tr>
<tr>
<td>MPI Message Lengths:</td>
<td>$0.000e+00$</td>
<td>0.00000</td>
<td>$0.000e+00$</td>
<td>$0.000e+00$</td>
</tr>
<tr>
<td>MPI Reductions:</td>
<td>$1.300e+01$</td>
<td>1.00000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Matrix size: 100,000

Convergence in 112 iterations.

Let's see the ksp context

**KSP Object:**
- type: cg
- maximum iterations=10000
- tolerances: relative=1e-05, absolute=0, divergence=10000
- left preconditioning
- using nonzero initial guess
- using PRECONDITIONED norm type for convergence test

**PC Object:**
- type: jacobi

Linear system matrix = precond matrix:
- type: seqaij
- rows=100000, cols=100000
- total: nonzeros=299998, allocated nonzeros=299998
- total number of mallocs used during MatSetValues calls =0
- not using I-node routines

---

### PETSc Performance Summary:

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Max/Min</th>
<th>Avg</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (sec)</td>
<td>$1.472e+00$</td>
<td>1.00000</td>
<td>$1.472e+00$</td>
<td></td>
</tr>
<tr>
<td>Objects</td>
<td>$1.300e+01$</td>
<td>1.00000</td>
<td>$1.300e+01$</td>
<td></td>
</tr>
<tr>
<td>Flops</td>
<td>$2.026e+08$</td>
<td>1.00000</td>
<td>$2.026e+08$</td>
<td>$2.026e+08$</td>
</tr>
<tr>
<td>Flops/sec</td>
<td>$1.376e+08$</td>
<td>1.00000</td>
<td>$1.376e+08$</td>
<td>$1.376e+08$</td>
</tr>
<tr>
<td>MPI Messages:</td>
<td>$0.000e+00$</td>
<td>0.00000</td>
<td>$0.000e+00$</td>
<td>$0.000e+00$</td>
</tr>
<tr>
<td>MPI Message Lengths:</td>
<td>$0.000e+00$</td>
<td>0.00000</td>
<td>$0.000e+00$</td>
<td>$0.000e+00$</td>
</tr>
<tr>
<td>MPI Reductions:</td>
<td>$1.300e+01$</td>
<td>1.00000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Matrix size: 1000,000

Convergence in 36 iterations.

Let's see the ksp context

**KSP Object:**
- type: cg
- maximum iterations=10000
- tolerances: relative=1e-05, absolute=0, divergence=10000
- left preconditioning
- using nonzero initial guess
- using PRECONDITIONED norm type for convergence test
type: cg
maximum iterations=10000
tolerances: relative=1e-05, absolute=0, divergence=10000
left preconditioning
using nonzero initial guess
using PRECONDITIONED norm type for convergence test
PC Object:
type: jacobi
linear system matrix = precond matrix:
Matrix Object:
type: seqaij
rows=1000000, cols=1000000
total: nonzeros=2999998, allocated nonzeros=2999998
total number of mallocs used during MatSetValues calls =0
not using I-node routines

**************************************
Max       Max/Min        Avg      Total
Time (sec): 2.306e+00      1.00000   2.306e+00
Objects: 1.300e+01      1.00000   1.300e+01
Flops: 6.580e+08      1.00000   6.580e+08  6.580e+08
Flops/sec: 2.853e+08      1.00000   2.853e+08  2.853e+08
MPI Messages: 0.000e+00      0.00000   0.000e+00  0.000e+00
MPI Message Lengths: 0.000e+00      0.00000   0.000e+00  0.000e+00
MPI Reductions: 1.300e+01      1.00000

CGNE
Matrix size : 10 000

Convergence in 150 iterations.

Let's see the ksp context
KSP Object:
type: cgne
maximum iterations=10000
tolerances: relative=1e-05, absolute=0, divergence=10000
left preconditioning
using nonzero initial guess
using PRECONDITIONED norm type for convergence test
PC Object:
type: jacobi
linear system matrix = precond matrix:
Matrix Object:
type: seqaij
rows=1000000, cols=1000000
total: nonzeros=2999998, allocated nonzeros=2999998
not using I-node routines

**************************************
Convergence in 61 iterations.

Let's see the ksp context
KSP Object:
  type: cgne
  maximum iterations=10000
  tolerances: relative=1e-05, absolute=0, divergence=10000
  left preconditioning
  using nonzero initial guess
  using PRECONDITIONED norm type for convergence test
PC Object:
  type: jacobi
  linear system matrix = precond matrix:
Matrix Object:
  type: seqaij
  rows=100000, cols=100000
  total: nonzeros=299998, allocated nonzeros=299998
  total number of mallocs used during MatSetValues calls =0
  not using I-node routines

Convergence in 25 iterations.

Matrix size : 100000

Convergence in 25 iterations.

Matrix size : 1000000
Let's see the ksp context
KSP Object:
  type: cgne
  maximum iterations=10000
  tolerances: relative=1e-05, absolute=0, divergence=10000
  left preconditioning
  using nonzero initial guess
  using PRECONDITIONED norm type for convergence test
PC Object:
  type: jacobi
  linear system matrix = precond matrix:
Matrix Object:
  type: seqaij
  rows=1000000, cols=1000000
  total: nonzeros=2999998, allocated nonzeros=2999998
  total number of mallocs used during MatSetValues calls =0
  not using I-node routines
 ******************************
****************************************************************************************************************
WIDEN YOUR WINDOW TO 120 CHARACTERS. Use 'enscript -r -fCourier9' to print this document
****************************************************************************************************************

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Max/Min</th>
<th>Avg</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (sec):</td>
<td>2.409e+00</td>
<td>1.000000</td>
<td>2.409e+00</td>
<td></td>
</tr>
<tr>
<td>Objects:</td>
<td>1.400e+01</td>
<td>1.000000</td>
<td>1.400e+01</td>
<td></td>
</tr>
<tr>
<td>Flops:</td>
<td>6.480e+08</td>
<td>1.000000</td>
<td>6.480e+08</td>
<td>6.480e+08</td>
</tr>
<tr>
<td>Flops/sec:</td>
<td>2.690e+08</td>
<td>1.000000</td>
<td>2.690e+08</td>
<td>2.690e+08</td>
</tr>
<tr>
<td>MPI Messages:</td>
<td>0.000e+00</td>
<td>0.000000</td>
<td>0.000e+00</td>
<td>0.000e+00</td>
</tr>
<tr>
<td>MPI Message Lengths:</td>
<td>0.000e+00</td>
<td>0.000000</td>
<td>0.000e+00</td>
<td>0.000e+00</td>
</tr>
<tr>
<td>MPI Reductions:</td>
<td>1.400e+01</td>
<td>1.000000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CGS
Matrix size : 10 000

Convergence in 40 iterations.

********************Results : KSP*******************
WIDEN YOUR WINDOW TO 120 CHARACTERS. Use 'enscript -r -fCourier9' to print this document

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Max/Min</th>
<th>Avg</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (sec):</td>
<td>1.285e+00</td>
<td>1.00000</td>
<td>1.285e+00</td>
<td></td>
</tr>
<tr>
<td>Objects:</td>
<td>1.700e+01</td>
<td>1.00000</td>
<td>1.700e+01</td>
<td></td>
</tr>
<tr>
<td>Flops:</td>
<td>1.249e+07</td>
<td>1.00000</td>
<td>1.249e+07</td>
<td>1.249e+07</td>
</tr>
<tr>
<td>Flops/sec:</td>
<td>9.720e+06</td>
<td>1.00000</td>
<td>9.720e+06</td>
<td>9.720e+06</td>
</tr>
<tr>
<td>MPI Messages:</td>
<td>0.000e+00</td>
<td>0.00000</td>
<td>0.000e+00</td>
<td>0.000e+00</td>
</tr>
<tr>
<td>MPI Message Lengths:</td>
<td>0.000e+00</td>
<td>0.00000</td>
<td>0.000e+00</td>
<td>0.000e+00</td>
</tr>
<tr>
<td>MPI Reductions:</td>
<td>1.700e+01</td>
<td>1.00000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Matrix size : 100 000

Convergence in 19 iterations.

****************Results : KSP**************

Let's see the ksp context

KSP Object:
- type: cgs
- maximum iterations=10000
- tolerances: relative=1e-05, absolute=0, divergence=10000
- left preconditioning
- using nonzero initial guess
- using PRECONDITIONED norm type for convergence test

PC Object:
- type: jacobi
- linear system matrix = precond matrix:

Matrix Object:
- type: seqaij
- rows=100000, cols=100000
- total: nonzeros=299998, allocated nonzeros=299998
- total number of mallocs used during MatSetValues calls =0
- not using I-node routines

Convergence in 9 iterations.

****************Results : KSP**************

Let's see the ksp context

Matrix size : 1000 000

Convergence in 9 iterations.

****************Results : KSP**************

Let's see the ksp context
KSP Object:
  type: cgs
  maximum iterations=10000
  tolerances: relative=1e-05, absolute=0, divergence=10000
  left preconditioning
  using nonzero initial guess
  using PRECONDITIONED norm type for convergence test
PC Object:
  type: jacobi
  linear system matrix = precond matrix:
Matrix Object:
  type: seqaij
  rows=1000000, cols=1000000
  total: nonzeros=2999998, allocated nonzeros=2999998
  not using I-node routines

WIDEN YOUR WINDOW TO 120 CHARACTERS. Use 'enscript -r -f Courier9' to print this document

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Max/Min</th>
<th>Avg</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (sec):</td>
<td>1.909e+00</td>
<td>1.00000</td>
<td>1.909e+00</td>
<td></td>
</tr>
<tr>
<td>Objects:</td>
<td>1.700e+01</td>
<td>1.00000</td>
<td>1.700e+01</td>
<td></td>
</tr>
<tr>
<td>Flops:</td>
<td>2.880e+08</td>
<td>1.00000</td>
<td>2.880e+08</td>
<td></td>
</tr>
<tr>
<td>Flops/sec:</td>
<td>1.508e+08</td>
<td>1.00000</td>
<td>1.508e+08</td>
<td></td>
</tr>
<tr>
<td>MPI Messages:</td>
<td>0.000e+00</td>
<td>0.00000</td>
<td>0.000e+00</td>
<td></td>
</tr>
<tr>
<td>MPI Message Lengths:</td>
<td>0.000e+00</td>
<td>0.00000</td>
<td>0.000e+00</td>
<td></td>
</tr>
<tr>
<td>MPI Reductions:</td>
<td>1.700e+01</td>
<td>1.00000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Conclusion

Comparaison of CG and CGNE:
CGNE converges better than CG solver (it is like twice faster).

CG, matrix size 1 000 000

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Max/Min</th>
<th>Avg</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (sec):</td>
<td>2.306e+00</td>
<td>1.00000</td>
<td>2.306e+00</td>
<td></td>
</tr>
<tr>
<td>Objects:</td>
<td>1.300e+01</td>
<td>1.00000</td>
<td>1.300e+01</td>
<td></td>
</tr>
<tr>
<td>Flops:</td>
<td>6.580e+08</td>
<td>1.00000</td>
<td>6.580e+08</td>
<td></td>
</tr>
<tr>
<td>Flops/sec:</td>
<td>2.853e+08</td>
<td>1.00000</td>
<td>2.853e+08</td>
<td></td>
</tr>
</tbody>
</table>

CGNE, matrix size 1 000 000

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Max/Min</th>
<th>Avg</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (sec):</td>
<td>1.447e+00</td>
<td>1.00000</td>
<td>1.447e+00</td>
<td></td>
</tr>
<tr>
<td>Objects:</td>
<td>1.400e+01</td>
<td>1.00000</td>
<td>1.400e+01</td>
<td></td>
</tr>
<tr>
<td>Flops:</td>
<td>1.548e+08</td>
<td>1.00000</td>
<td>1.548e+08</td>
<td></td>
</tr>
<tr>
<td>Flops/sec:</td>
<td>1.070e+08</td>
<td>1.00000</td>
<td>1.070e+08</td>
<td></td>
</tr>
</tbody>
</table>

We can just conclude on a significant difference regarding the flops rate. We illustrated it with matrix size of 1 000 000, but it is also available for 10 000 (CG has twice grower rates than CGNE).

Comparaison of CG and CGS
The CGS solver converges better than CG (more than three times better).
### Comparaison of CGS and CGNE

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Max/Min</th>
<th>Avg</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time (sec):</strong></td>
<td>2.306e+00</td>
<td>1.00000</td>
<td>2.306e+00</td>
<td></td>
</tr>
<tr>
<td><strong>Objects:</strong></td>
<td>1.300e+01</td>
<td>1.00000</td>
<td>1.300e+01</td>
<td></td>
</tr>
<tr>
<td><strong>Flops:</strong></td>
<td>6.580e+08</td>
<td>1.00000</td>
<td><strong>6.580e+08</strong></td>
<td><strong>6.580e+08</strong></td>
</tr>
<tr>
<td><strong>Flops/sec:</strong></td>
<td>2.853e+08</td>
<td>1.00000</td>
<td><strong>2.853e+08</strong></td>
<td><strong>2.853e+08</strong></td>
</tr>
</tbody>
</table>

**CGS, matrix size 1 000 000**

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Max/Min</th>
<th>Avg</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time (sec):</strong></td>
<td>1.909e+00</td>
<td>1.00000</td>
<td>1.909e+00</td>
<td></td>
</tr>
<tr>
<td><strong>Objects:</strong></td>
<td>1.700e+01</td>
<td>1.00000</td>
<td>1.700e+01</td>
<td></td>
</tr>
<tr>
<td><strong>Flops:</strong></td>
<td>2.880e+08</td>
<td>1.00000</td>
<td><strong>2.880e+08</strong></td>
<td><strong>2.880e+08</strong></td>
</tr>
<tr>
<td><strong>Flops/sec:</strong></td>
<td>1.508e+08</td>
<td>1.00000</td>
<td><strong>1.508e+08</strong></td>
<td><strong>1.508e+08</strong></td>
</tr>
</tbody>
</table>

About times performances, we can’t conclude on a difference for solvers. However, the Flops rates are really much better for CGS solver.

- Comparaison of CGS and CGNE
  - Regarding the number of iteration for convergence, CGS is better than CGNE (not exactly two times better).
  - Considering the flops rates, we see that they are similar.

### CGNE, matrix size 1 000 000

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Max/Min</th>
<th>Avg</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time (sec):</strong></td>
<td>1.447e+00</td>
<td>1.00000</td>
<td>1.447e+00</td>
<td></td>
</tr>
<tr>
<td><strong>Objects:</strong></td>
<td>1.400e+01</td>
<td>1.00000</td>
<td>1.400e+01</td>
<td></td>
</tr>
<tr>
<td><strong>Flops:</strong></td>
<td>1.548e+08</td>
<td>1.00000</td>
<td><strong>1.548e+08</strong></td>
<td><strong>1.548e+08</strong></td>
</tr>
<tr>
<td><strong>Flops/sec:</strong></td>
<td>1.070e+08</td>
<td>1.00000</td>
<td><strong>1.070e+08</strong></td>
<td><strong>1.070e+08</strong></td>
</tr>
</tbody>
</table>

### CGS, matrix size 1 000 000

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Max/Min</th>
<th>Avg</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time (sec):</strong></td>
<td>1.909e+00</td>
<td>1.00000</td>
<td>1.909e+00</td>
<td></td>
</tr>
<tr>
<td><strong>Objects:</strong></td>
<td>1.700e+01</td>
<td>1.00000</td>
<td>1.700e+01</td>
<td></td>
</tr>
<tr>
<td><strong>Flops:</strong></td>
<td>2.880e+08</td>
<td>1.00000</td>
<td><strong>2.880e+08</strong></td>
<td><strong>2.880e+08</strong></td>
</tr>
<tr>
<td><strong>Flops/sec:</strong></td>
<td>1.508e+08</td>
<td>1.00000</td>
<td><strong>1.508e+08</strong></td>
<td><strong>1.508e+08</strong></td>
</tr>
</tbody>
</table>
VII GPU used in neutronic domain

The aim of this project was to implement the Conjugate Gradient method on GPU, by using CUDA, but also Cublas. We studied before the performances of Cublas and CUDA on scalar product examples, and it appears that CUDA offers a good gain performance, but still less than Cublas. In this context, we decide to apply it in neutronic domain, as the time and Megaflops rates gain must be more and more bigger.

In what follows, we will study the architecture of codes, and explain our implementation choices.

Then, we will briefly explain the CUDA's code, and Cublas one. We will finally present the obtained results.

7.1 Introduction

In order to implement a professional code, we decide to use the following architecture:

We have six folders:

```
./
    Makefile
    src/  Clock/  CPU/  Cublas/  Cuda/  Matrix_Market/
```

The Makefile contains in the main folder is doing the compilation of all codes in subsidiary folders. This Makefile contains the variables necessary to export the compiler path, and the command to execute the compilations in all subsidiary folders.

In each subsidiary folders, except the src/ one, we have statics libraries: all Makefiles in subsidiary folders are compiling the codes such that we obtain statics libraries. It will be latter modify, to compile it as a dynamics libraries, as the codes in Clock and Matrix_Market must not be modified anymore.

```
Clock/
libclock.h libclock.c Makefile

Cublas/
libcgculas.c libcgculas.h Makefile

CPU/
libcpu.c libcpu.h Makefile
```

The src/ folder contains the main.cu code, which includes all the generated libraries.
This structure is justified by the fact that we use a lot of different functions, and so the programmer and user can handle the code easily. Currently, the user just have to modify the following variable in the principal Makefile:

```
export FOLDER_HOME = /home/phd_hpc/fboillod/double_prec
```

You have to indicate the path to the main folder of project. Note that you also have to modify the path to compiler, and Cuda installation folder.

For compilation, we have the following scheme:

![Diagram of Makefile structure with directories and files]

The main difficulties for this architecture were about the languages: for example, the main.cu Makefile had to mix several languages (C++ and Cuda).

We represented it by the following scheme:
This project is available yet on gpu4u and fermi computers.
We will now study the diverse functions used to implement the Conjugate gradient.

### 7.2 Conjugate gradient with CUDA-CPU

First, we wanted to implement the conjugate gradient exclusively on the GPU. Because of time, I couldn’t finish this code, as it was not stable enough. I decide, in consequence, to implement CG by using both CPU and GPU.

All the functions we will present in what follows are in the libcgcuda.cu file.

To realise the CG function, we used the matrix/vector product, which is realised on GPU. We also implemented two versions of scalar product, one using exclusively the GPU, and the other one doing the product of vectors on GPU, but the addition of elements is realised on CPU. We implemented also a function __global__ void GPU_addvec(double *a, double alpha, double *b, double beta, const int N) which is doing the following operation:

\[ a[i] = alpha \times a[i] + beta \times b[i] \]

This function is realised only on GPU.

The main difficulties to implement this code was to debug it, as GPU does not allow host functions.
7.3 Conjugate gradient with Cublas

The Cublas version of Conjugate Gradient was more easier to implement, as we can debug it quicklier. Actually, it exists a function CublasGetVector(...) so we can obtain the copy of device vector on a host vector.

Besides, the Cublas functions are easy to handle. We use, to implement the Conjugate Gradient function the following Cublas functions:

- cublasDcopy (n, b_d, 1, h_d, 1);
  This function copy the vector b in h one.

- cublasDgemv ('N', n, n, 1.0, A_d, n, h_d, 1, 0.0, temp_Ah, 1);
  This function realises the following function:
  \[ \text{temp}_\text{Ah} = 1.0 \times \text{A} \times \text{h} + 0.0 \times \text{temp}_\text{Ah} \]

- cublasDscal (n, v, h_d, 1);
  This function is doing the scalar product of v and h.

- cublasDaxpy (n, -1.0, g_d, 1, h_d, 1);
  This function realises the following operation:
  \[ h = -1.0 \times g \]

- norm_cub = cublasDdot (n, x_d, 1, x_d, 1);
  This function calculates the norm of x vector.

The function we created is the following:

```c
void GPU_CG_CUBLAS (double *A_d, double *x_d, double *b_d, double *vecnul_d, int n, int LOOP, double epsilon, const int nBytes, int *its_cublas, dim3 dimGrid)
```

A is the matrix (on device), b the right hand side, and x the initial solution. Vecnul is a vector on device which contains only nul values, as the cublas vectors are not initialized when you create them. N is the size of vector (and matrix, we used a squared one). LOOP is just the maximum iterations we allow, and epsilon, the error we fixed to stop the iteration (0.0001). its_cublas is the number of iterations at convergence, and other variables are used for memory allocation.
7.4 Results

We run the previous code on two computers: on fermi (Gtx GeForce 480) and gpu4u (Gtx GeForce 280).

**Fermi:**

![Gflops rates chart for Fermi]

![Gflops/sec rates chart for Fermi]
Comparaisons:
The first scheme shows the Gflops Cublas rates, by considering the used graphic card.

The second scheme shows the Gflops/sec rate for Cuda's function, by considering the graphic card we used.
We can conclude that our Cuda's code is like two times better with the fermi computer. About the Cublas code, the performance is not so satisfiable.

We studied all codes we implemented to solve our linear system with Conjugate Gradient. We will now conclude on this project.
During my last placement in CEA, we already studied the PETSc library, and its impact on MINOS solver. We concluded that we obtained better convergences performances with PETSc, but in term of computation time, PETSc was much slower.

The issue was now to study the Conjugate Gradient by using the Graphic card: to do so, we wanted to use PETSc, as we worked with it before, but also to program the Conjugate gradient with Cuda and Cublas.

We concluded that Cublas offers better performances than Cuda. However, the Cuda's code is much more efficient with fermi graphic cards, than Cublas' code.

Regarding PETSc, we conclude on a very good gain performance for Multigrid objects, by using GPU. The computation's time for PETSc 's code is however not so significantly different as the Megaflops rates obtained (with and without GPU).

The next step is now to study our KSP code, with GPU, but to do so, it is necessary to reinstall PETSc development version. We also developed a matrix interface, which allows to load every matrix available on Matrix market website. In the future, this program will be ameliorated, and this interface will be used.
Bibliography

Thesis of M. Pierre Guérin,
Thesis of M. Berlier,
Petsc User Manual
Cuda user manual
Cublas user manual

Webography

Site de l'Idris : http://www.idris.fr/data/cours/parallel/mpi/choix_doc.html
Site de Petsc : http://www.mcs.anl.gov/petsc/petsc-as/