HYBRID PARALLELISED FRAMEWORK FOR THE SOLUTION OF PDEs ON HPC CLUSTERS

MAJOR TECHNICAL PROJECT (DP 401P)

to be submitted by

RITWIK SAHA(B16110)
NIKHIL GUPTA(B16023)

for the

END-SEMESTER
EVALUATION

under the supervision of

DR. GAURAV BHUTANI

SCHOOL OF COMPUTING AND ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY MANDI

KAMAND-175005, INDIA

NOVEMBER, 2019

Contents

Abstract

Computational Fluid Dynamics is a branch of fluid mechanics which studies the flow of liquids, gas and other fluid flows using intensive numerical analysis for solving partial differential equations. Solutions of partial differential equations include substantial amount of solving linear equations of the format $Ax = b$. Various iterative methods are used to solve these equations for computational feasibility. With the recent advancements of hardware in the field of supercomputing on HPC (High performance Computing) clusters, newer and faster algorithms and tools are required to use the resources and the improved compute power available. Similarly, the above stated linear equations are solved on such HPC clusters.

Various libraries exist to simulate and solve complex problems in the field of fluid dynamics. 'Fluidity' is one such library used globally for the same. It uses a library named 'PETSc' to solve linear equations in efficient and iterative methods to substantially reduce the solve time.

Earlier project aimed at improving the usage of accelerators(GPU cards) in PETSc to efficiently used the available hardware. In odd-semester, we were able to simulate and compare the performance in solving a linear equation using PETSc on CPU and GPU. We were also able to successfully understand the building and initialisation of PETSc built to run specifically on GPU.

The results obtained in odd semester did not show any efficiency in performance. Thus we moved to the next step to incorporate GPU enabled PETSc into Fluidity.
Keywords: Fluidity, CFD, PDE , HPC, PETSc, GPU

Overview of Previous work

This section is meant for the readers to get an idea of the work done, the motivation and the challenges faced throughout the project.

The reader may skip to chapter 2 for the work done in Even Semester.

1.1 Performance between multiple cores and GPU

Running KSP(Krylov Subspace Method) example on upto 12 cores on HPC and GPU (GTX 1050), the time taken vs. dimension graph provides an overview which aligns with the expected results that more the number of cores, the lesser would be the time for computation. The results seem to be more pronounced for higher dimensions of vector/matrix. As expected, computation on GPU easily wins over computation on 12 hyper-threaded cores of CPU even when the GPU is a modest NVIDIA-GTX 1050 running on a personal laptop.

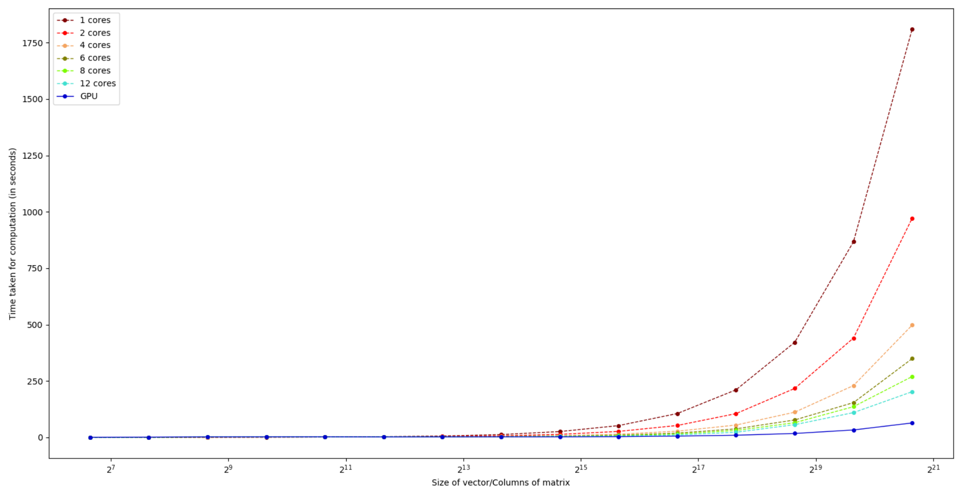


Figure 1.1: Benchmarking KSP solver for symmetric tridiagonal matrix using multiple cores and GPU

However judging the whole scenario only with this graph would be unfair as we are clearly missing out on the observations for lower dimensionality of vector/matrix. Therefore, for greater insight we would be zooming into the same graph as above upto dimensionality of 16,000

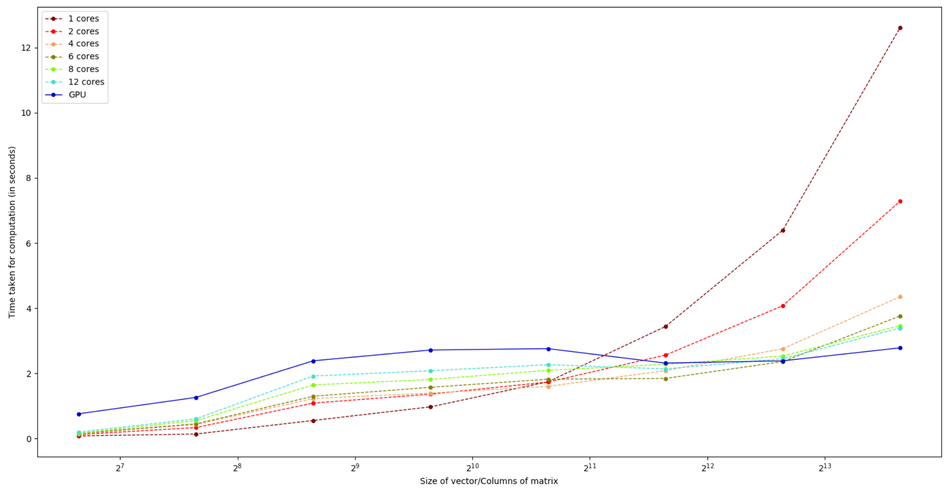


Figure 1.2: Zoomed in graph of Benchmarking KSP solver for symmetric tridiagonal matrix using multiple cores and GPU

For lower dimensions upto approximately 2^{12}, we can see that GPU takes the longest time to compute as there is data transfer overhead between the CPU memory and GPU memory caused by cudaMemCopy() function. Also, for the same range of lower dimensionality, higher number of cores result in more time taken due to race conditions and message passing overhead(even though it is multicore architecture possibly cc-NUMA architecture).

1.2 Attempt at Hybridization

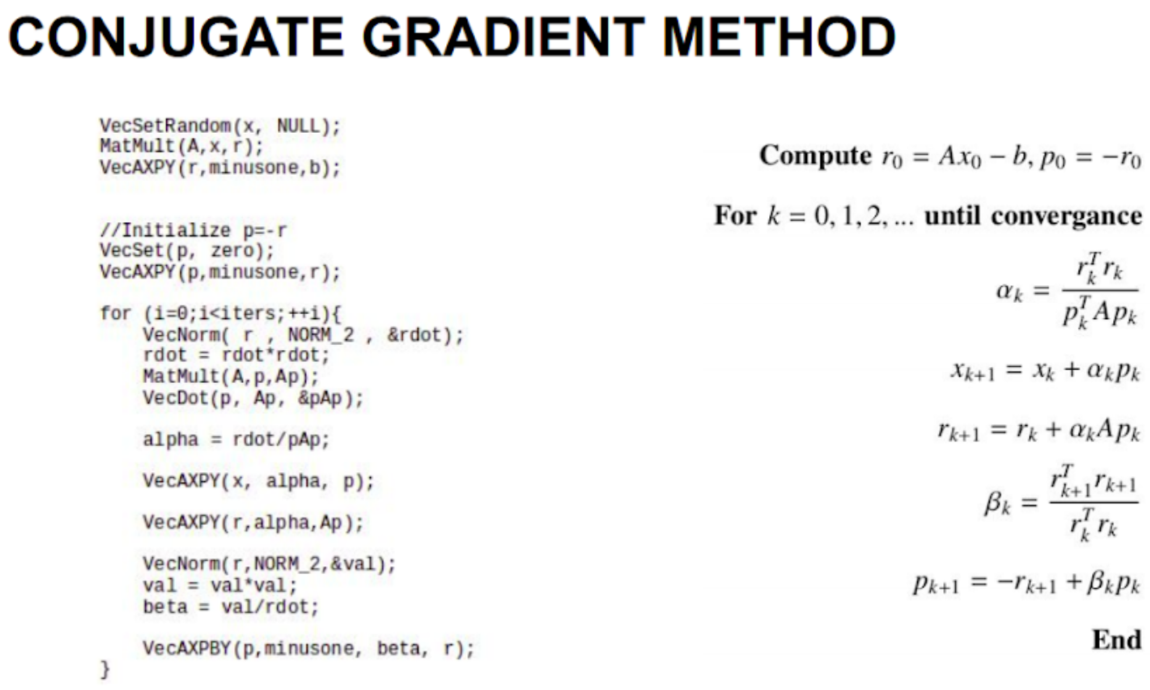
Hybridization in our context means that PETSc is able to use CPU cores as well as GPU cores simultaneously for possible marginal gain over running the same problem on exclusively GPU or exclusively CPU.

We wrote a naive implementation of conjugate gradient method and attempted to hybridize it (Figure 1.3 and 1.4).

1.3 Benchmarking Observations

The “ratio” in Figure 1.5 denotes the number of vector indices allocated to CPU to that of GPU

- We could only hybridize with GPU+one CPU core



CONJUGATE GRADIENT METHOD

```
VecSetRandom(x, NULL);
MatMult(A,x,r);
VecAXPY(r,minusone,b);


//Initialize p=-r
VecSet(p, zero);
VecAXPY(p,minusone,r);

for (i=0;i<iters;++i){
    VecNorm( r , NORM_2 , &rdot);
    rdot = rdot*rdot;
    MatMult(A,p,Ap);
    VecDot(p, Ap, &pAp);

    alpha = rdot/pAp;

    VecAXPY(x, alpha, p);

    VecAXPY(r,alpha,Ap);

    VecNorm(r,NORM_2,&val);
    val = val*val;
    beta = val/rdot;

    VecAXPBY(p,minusone, beta, r);
}
```

```
**Compute** $r_0 = Ax_0 - b, p_0 = -r_0$
**For** $k = 0, 1, 2, ...$ **until convergance**
    $\alpha_k = \frac{r_k^T r_k}{p_k^T A p_k}$
    $x_{k+1} = x_k + \alpha_k p_k$
    $r_{k+1} = r_k + \alpha_k A p_k$
    $\beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
    $p_{k+1} = -r_{k+1} + \beta_k p_k$
**End**
```

Figure 1.3: Algorithm for CG method

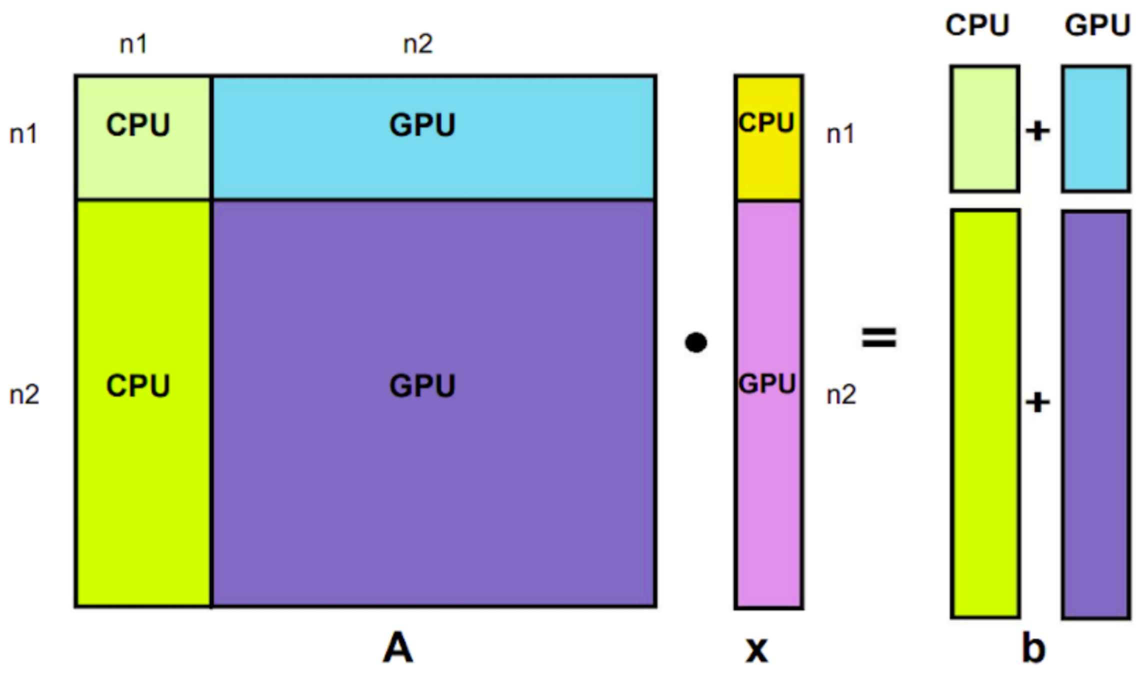


Figure 1.4: Naive methodology taken for implementation of CG method in a hybrid fashion

- We could not hybridize with GPU+multiple CPU cores as all cores are claiming the GPU leading to deadlock

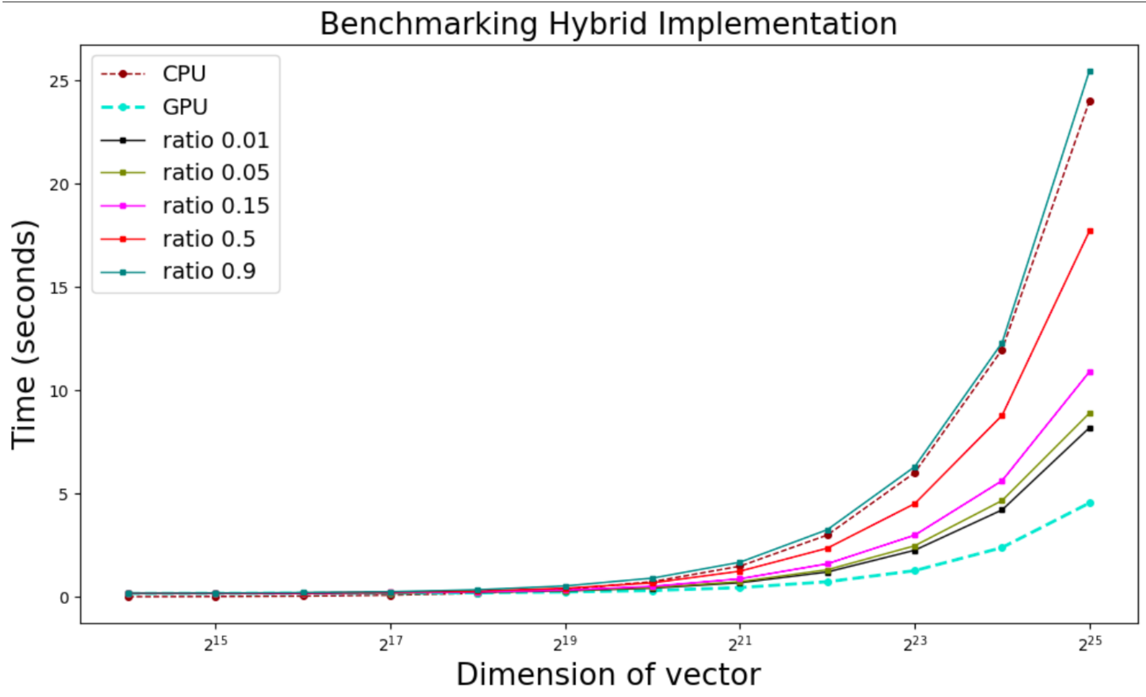


Figure 1.5: Benchmarking conjugate gradient method with different ratios

- IIT Mandi HPC Cluster is incompatible with "Multinode + GPU" PETSC implementation due to absence of InfiniBand connectivity.
- Contrary to expectations, a hybrid solver with 0.01 parts allocated to CPU and rest to GPU, performs worse than simple GPU implementation. Same is the case with all other ratios used.

Work done in Even Semester

2.1 Fluidity : Set-up and first install

2.1.1 What is fluidity?

Fluidity is an open source, general purpose, multiphase computational fluid dynamics code capable of numerically solving the Navier-Stokes equation and accompanying field equations on arbitrary unstructured finite element meshes in one, two and three dimensions. It is parallelised using MPI and is capable of scaling to many thousands of processors. Other innovative and novel features include the use of anisotropic adaptive mesh technology, and a user-friendly GUI and a Python interface which can be used to calculate diagnostic fields, set prescribed fields or set user-defined boundary conditions.[10]

2.1.2 The objective

As fluidity makes extensive calls to the PETSc library as its primary solver, and relies a lot on sparse matrices, fluidity can be made to leverage power of GPU through PETSc, at least in theory. This required understanding relevant sections of the vast fluidity codebase. Also, as a prerequisite to the attempt of building fluidity with GPU-enabled PETSc with GPU specific changes, we need to build fluidity from its source code which is not as easy as it sounds due to hugely outdated documentation, highly version specific dependencies, large number of dependencies, elaborate configuration options.

2.1.3 Role of Containerization

As discussed in the previous-sem report, HPC of IIT-Mandi has singularity(a containerization software) installed so as to enable users to provide the very specific environment requirements of the software they are trying to run. Thus the scripts used to build fluidity would be purely singularity based.

2.1.4 Difficulties faced and solutions that worked

- As discussed earlier, outdated documentation and a large number of very version specific dependencies.
- We had to retrieve the building steps from the various CI(continuous integration) scripts we obtained from the fluidity github repository. Also, the maintainers of the repo were very quick to sort out issues raised by us.
- We learned that the environment for building fluidity can be set by 'sudo apt install fluidity-dev' after updating apt-package-manager with required ppa(personal package archive)
- After that we were able to install unchanged fluidity on CPU configured PETSc and most of the tests passed (39 out 1800 failed)

2.1.5 Singularity Script

```
Bootstrap: docker
From: ubuntu:bionic

%post

    export DEBIAN_FRONTEND=noninteractive

    ## update the system
    apt-get update
    apt-get -y dist-upgrade

    ## Install gpg
    apt-get -y install gnupg dirmngr

    ## add ppa
    echo "deb http://ppa.launchpad.net/fluidity-core/ppa/ubuntu bionic main" \
     > /etc/apt/sources.list.d/fluidity-core-ppa-bionic.list
    gpg --keyserver keyserver.ubuntu.com --recv 0D45605A33BAC3BE
    gpg --export --armor 33BAC3BE | apt-key add -
    apt-get update

    ## Set timezone
```

```
echo "Europe/London" > /etc/timezone

# install fluidity dependencies
apt-get -y install fluidity-dev

## set environment options
# export PATH="${PATH}:/usr/local/texlive/2019/bin/x86_64-linux"
export PETSC_DIR="/usr/lib/petscdir/3.8.3"
export LD_LIBRARY_PATH="/usr/lib/petscdir/3.8.3/linux-gnu-c-opt/lib"
export LDFLAGS="-L/usr/lib/x86_64-linux-gnu/hdf5/openmpi"
export CPPFLAGS="-I/usr/include/hdf5/openmpi"
export OMPI_MCA_btl_vader_single_copy_mechanism="none"

cd /

## obtain fluidity
git clone --depth 2 https://github.com/FluidityProject/fluidity.git
cd /fluidity

## set environment again
export MPLBACKEND="PS"
export OMPI_MCA_btl="^openib"

## build and test fluidity
./configure --enable-2d-adaptivity
make makefiles || echo "may fail, no need to worry!"
make -j
make -j fltools

## tests
make unittest
make THREADS=8 test
```

2.1.6 Building unchanged fluidity in a CUDA environment

This would be the next logical step to take after successfully building fluidity on vanilla ubuntu. This time the support of ‘apt install fluidity-dev‘ was not available fully due to our requirement of GPU configured PETSc. The singularity scripts were built in 3 parts to maintain modularity and keeping room for trial and error.

2.1.6.1 Script-I : fluidity_b0.singularity

```
Bootstrap: docker
From: nvidia/cuda:10.0-devel-ubuntu18.04

%post

    ## set some variables specific to cuda
    echo "export DEBIAN_FRONTEND=noninteractive" >>$SINGULARITY_ENVIRONMENT
    echo "export PATH=/usr/local/cuda-10.0/bin:$PATH" >>$SINGULARITY_ENVIRONMENT
    echo "export LD_LIBRARY_PATH=/usr/local/cuda-10.0/lib64:/usr/lib64/nvidia" \
     >>$SINGULARITY_ENVIRONMENT
    $(cat $SINGULARITY_ENVIRONMENT)

    mkdir -p /usr/lib64/nvidia

    ## check for nvcc
    nvcc --version

    ## change ubuntu repositories mirror to Japan (faster downloads)
    sed -i 's|http://archive.ubuntu|http://jp.archive.ubuntu|g' /etc/apt/sources.list
    sed -i 's|https://archive.ubuntu|https://jp.archive.ubuntu|g' \
     /etc/apt/sources.list

    ## Some essentials for PETSc
    apt -y update
    apt -y install git wget python g++ gcc gfortran curl vim

    ## download PETSc source
    mkdir /installs
    cd /installs
    wget http://ftp.mcs.anl.gov/pub/petsc/release-snapshots/petsc-3.10.5.tar.gz
    cd /
```

```
## Some essentials for PETSc
apt -y install build-essential valgrind
apt -y install cmake
apt -y install python3 python3-distutils

## Get pip for python3
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py || echo "curl failed"
python3 get-pip.py || echo "get-pip script failed"
rm get-pip.py || echo "get-pip removal failed"
python3 -m pip install --upgrade pip || echo "pip upgrade failed"

## petsc GPU installation

## install PETSc with GPU support
cd /installs
tar xvzf ./petsc-3.10.5.tar.gz -C ./
rm ./petsc-3.10.5.tar.gz

cd ./petsc-3.10.5
sed -i '1s/^/NVCCFLAGS += -Xcompiler -openmp\nNVCCFLAGS += -Xcompiler \
 -fopenmp\n/' makefile

./configure \
    --with-cuda=1 \
    --with-precision=double \
    --with-clanguage=c \
    --download-mpich=yes \
    --download-cusp \
    --download-zoltan=yes \
    --download-metis=yes \
    --download-parmetis=yes \
    --download-fblaslapack=yes \
    --download-blacs=yes \
    --download-scalapack=yes \
    --download-mumps=yes --download-hypre=yes \
    --download-suitesparse=yes --download-ml=yes --with-fortran-interfaces\
    || echo "petsc configure failed"
make all test || echo "*****make and tests failed*****"
cp -r ./lib/petsc /lib || echo "*****PETSc lib copy failed*****"

## final steps of singularity
cd /
apt clean
```

2.1.6.2 Script-II : fluidity_b1.singularity (building upon script-I)

```
Bootstrap: localimage
From: fluidity_b0.simg

%post

    # Install gpg
    apt -y update
    apt-get -y install gnupg dirmngr

    # add ppa
    echo "deb http://ppa.launchpad.net/fluidity-core/ppa/ubuntu bionic main" > \
     /etc/apt/sources.list.d/fluidity-core-ppa-bionic.list
    gpg --keyserver keyserver.ubuntu.com --recv 0D45605A33BAC3BE
    gpg --export --armor 33BAC3BE | apt-key add -
    apt-get update

    export DEBIAN_FRONTEND=noninteractive

    ln -fs /usr/share/zoneinfo/Aisa/Calcutta /etc/localtime
    apt-get install -y tzdata

    ## Set timezone
    #echo "Aisa/Calcutta" > /etc/timezone

    # install fluidity dependencies
    apt-get -y install fluidity-dev
    ## set environment options
    $(cat $SINGULARITY_ENVIRONMENT)
    echo "export PETSC_DIR=/installs/petsc-3.10.5" >>$SINGULARITY_ENVIRONMENT
    echo "export PATH=/installs/petsc-3.10.5/arch-linux2-c-debug/bin:$PATH" >> \
$SINGULARITY_ENVIRONMENT
    echo "export LD_LIBRARY_PATH=/installs/petsc-3.10.5/arch-linux2-c-debug/lib \
    :/usr/lib/petscdir/3.8.3/linux-gnu-c-opt/lib:$LD_LIBRARY_PATH" \
     >>$SINGULARITY_ENVIRONMENT
    echo 'export LDFLAGS="-L/usr/lib/x86_64-linux-gnu/hdf5/openmpi"' \
     >>$SINGULARITY_ENVIRONMENT
    echo 'export CPPFLAGS="-I/usr/include/hdf5/openmpi"' >>$SINGULARITY_ENVIRONMENT
    echo 'export OMPI_MCA_btl_vader_single_copy_mechanism="none"' \
     >>$SINGULARITY_ENVIRONMENT
    echo 'export OMPI_MCA_btl="^openib"' >>$SINGULARITY_ENVIRONMENT
    $(cat $SINGULARITY_ENVIRONMENT)

    cd /
```

2.1.6.3 Script-III : fluidity_final.singularity (final image based on Script-II)

```
Bootstrap: localimage
From: fluidity_b1.simg

%post

   ## obtain fluidity
    git clone --depth 2 https://github.com/FluidityProject/fluidity.git
    cd /fluidity

    ## build and test fluidity
    ./configure --enable-2d-adaptivity
    make makefiles || echo "may fail, no need to worry!"
    make -j
    make -j fltools

    make THREADS=8 test
```

Thus, we were able to successfully build fluidity(unchanged) in a GPU environment. Most of the tests passed and this work laid further ground for testing changed fluidity(configured to make GPU specific calls)

2.2 Making GPU-Specific changes to Fluidity

2.2.1 Changes in Fluidity Source Code

Since Fluidity is an enormous and complicated library, understanding the library is out of the scope of this project. In spite of providing path of GPU enabled PETSC library in configure step as mentioned in the previous section, running standard test cases of FLuidity did not use the GPU card. This was confirmed by the command nvidia-smi in the host system outside singularity environment. Thus a need to make changes in fluidity source code was observed.

3 files were traced and modifications were done to call petsc directives which use GPU cards. All instances of VECTYPE and MATTYPE were replaced with their CUDA equivalent. All possible directives which create seq/CPU type vector/matrices were replaced with direct available CUDA versions.

1. $PETSC_DIR/femtools/Sparse_Tools_Petsc.F90

   - Line 433:
     Original :

     ```
     call MatCreateAIJ(MPI_COMM_SELF, urows, ucols, urows, ucols, &
     PETSC_NULL_INTEGER, dnnz, 0, PETSC_NULL_INTEGER, matrix%M, ierr)
     ```

     Modified :

     ```
     call MatCreateAIJCUSPARSE(MPI_COMM_SELF, urows, ucols, urows, ucols, &
     PETSC_DECIDE, dnnz, 0, PETSC_NULL_INTEGER, matrix%M, ierr)
     ```

   - Line 442 :
     Original :

     ```
     call MatCreateAIJ(MPI_COMM_FEMTOOLS, nprows*blocks(1), npcols*blocks(2), &
     urows, ucols, &
     PETSC_NULL_INTEGER, dnnz, PETSC_NULL_INTEGER, onnz, matrix%M, ierr)
     ```

     Modified :

     ```
     call MatCreateAIJCUSPARSE(MPI_COMM_FEMTOOLS, nprows*blocks(1), npcols*blocks(2),&
     urows, ucols, &
     PETSC_DECIDE, dnnz, PETSC_DECIDE, onnz, matrix%M, ierr)
     ```

2. $PETSC_DIR/femtools/Multigrid.F90

- Line 855 :

  Original :

  ```
  call MatCreateAIJ(MPI_COMM_FEMTOOLS, nrows, ncols ,PETSC_DECIDE,PETSC_DECIDE,&
  PETSC_NULL_INTEGER, dnnz, PETSC_NULL_INTEGER, onnz, P, ierr)
  ```

  Modified :

  ```
  call MatCreateAIJCUSPARSE(MPI_COMM_FEMTOOLS, nrows, ncols, PETSC_DECIDE, &
  PETSC_DECIDE, PETSC_DECIDE, dnnz, PETSC_DECIDE, onnz, P, ierr)
  ```

- Line 864 :

  Original :

  ```
  call MatCreateAIJ(MPI_COMM_SELF, nrows, ncols, nrows, ncols, &
  PETSC_NULL_INTEGER, dnnz, 0, PETSC_NULL_INTEGER, P, ierr)
  ```

  Modified :

  ```
  call MatCreateAIJCUSPARSE(MPI_COMM_SELF, nrows, ncols, nrows, ncols, &
  PETSC_DECIDE, dnnz, 0, PETSC_NULL_INTEGER, P, ierr)
  ```

3. $PETSC_DIR/femtools/Petsc_Tools.F90

- Line 588 :

  Original :

  ```
  if (parallel) then
          call VecCreateMPI(MPI_COMM_FEMTOOLS, plength, ulength, vec, ierr)
  else
          call VecCreateSeq(MPI_COMM_SELF, ulength, vec, ierr)
  end if
  ```

  Modified :

  ```
  if (parallel) then
          call VecCreate(MPI_COMM_FEMTOOLS, vec, ierr)
          call VecSetType(vec, VECMPICUDA, ierr)
          call VecSetSizes(vec, plength, ulength, ierr)
  else
          call VecCreateSeqCUDA(MPI_COMM_SELF, ulength, vec, ierr)
  end if
  ```

- Line 1128 :

  Original :

```
call MatCreateAIJ(MPI_COMM_SELF, nprows, npcols, nprows, npcols, &
PETSC_NULL_INTEGER, nnz, 0, PETSC_NULL_INTEGER, M, ierr)
```

Modified :

```
call MatCreateAIJCUSPARSE(MPI_COMM_SELF, nprows, npcols, nprows, npcols, &
PETSC_DECIDE, nnz, 0, PETSC_NULL_INTEGER, M, ierr)
```

- Line 1215 :
  Original :

  ```
  call MatSetType(M, MATAIJ, ierr)
  ```

  Modified :

  ```
  call MatSetType(M, MATAIJCUSPARSE, ierr)
  ```

- Line 1304 :
  Original :

  ```
  call MatCreateAIJ(MPI_COMM_FEMTOOLS, nrowsp, ncolsp, nrows, ncols, &
  PETSC_NULL_INTEGER, d_nnz, PETSC_NULL_INTEGER, o_nnz, M, ierr)
  ```

  Modified :

  ```
  call MatCreateAIJCUSPARSE(MPI_COMM_FEMTOOLS, nrowsp, ncolsp, nrows, ncols, &
  PETSC_DECIDE, d_nnz, PETSC_DECIDE, o_nnz, M, ierr)
  ```

However upon re-configuring and compiling the changes, we encountered some errors, which upon analyzing we saw that CUDA version of vec-types were not defined in PETSc fortran libraries. Being a bug on the PETSc side, we found out the specific macros need to be defined at the source code level of PETSc and reported the same as an issue on the PETSc official repository. The bug is now ironed out in future versions of PETSc.

Link to the issue : https://gitlab.com/petsc/petsc/-/issues/660

2.2.2 Changes required in PETSc

In File : $PETSC_DIR/include/petsc/finclude/petscvec.h

The following lines were added :

```
#define VECCUDA 'cuda'
#define VECSEQCUDA 'seqcuda'
#define VECMPICUDA 'mpicuda'
```

2.2.3 Changes in the singularity script(fluidity_b0.singulairty)

The above changes were written to be executed by the singularity script in an automated fashion. The following lines were added :

```
## include VECCUDA VecType(s) for fortran
    cd ./include/petsc/finclude/
    head -n 29 petscvec.h > tempfile
    printf \
        "\n#define VECCUDA 'cuda'\n#define VECSEQCUDA 'seqcuda'\n#define \
         VECMPICUDA  mpicuda'\n" >> tempfile
    tail -n +30 petscvec.h >> tempfile
    mv tempfile petscvec.h
    cd ../../../
```

2.2.4 FORTRAN simulation

(Checking for GPU-enabled FORTRAN using pure-petsc codes)

As directed by our mentor, we checked whether fortran codes are actually able to leverage GPU or not for calculation. We used the same KSP-based solver (written in C) used for benchmarking in the previous sem and hand-converted it to FORTRAN and timed them for different matrix sizes ranging from 100 to 1638400 in 15 steps each multiplying the dimension by 2. The results were plotted on a logarithmic scale on x-axis (Figure 2.6 and 2.7).

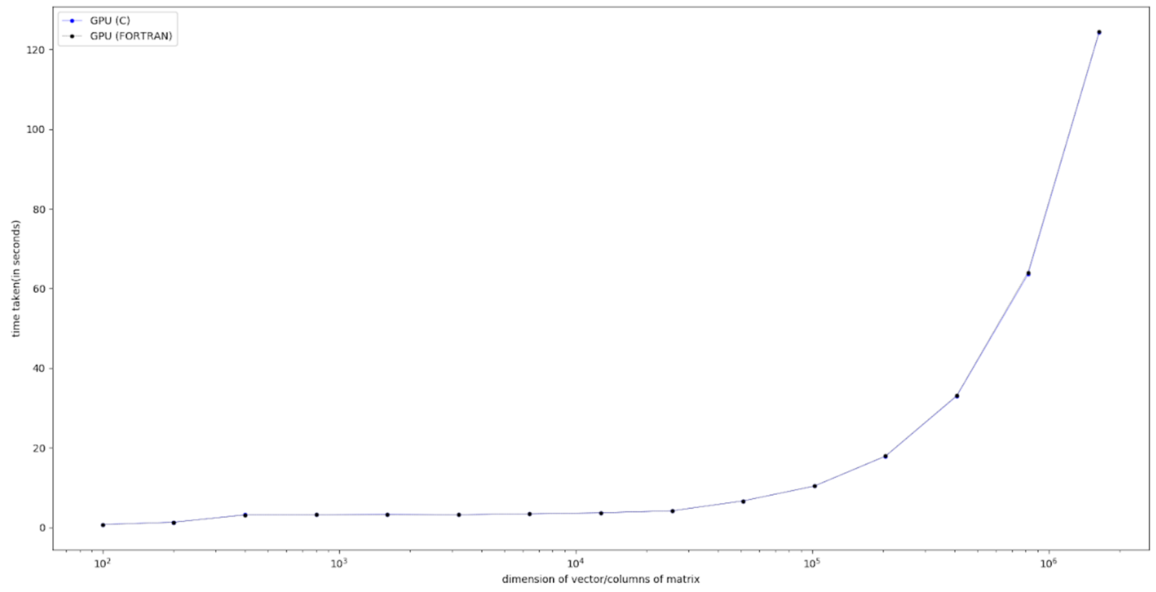


Figure 2.6: Benchmarking KSP based solver written in FORTRAN

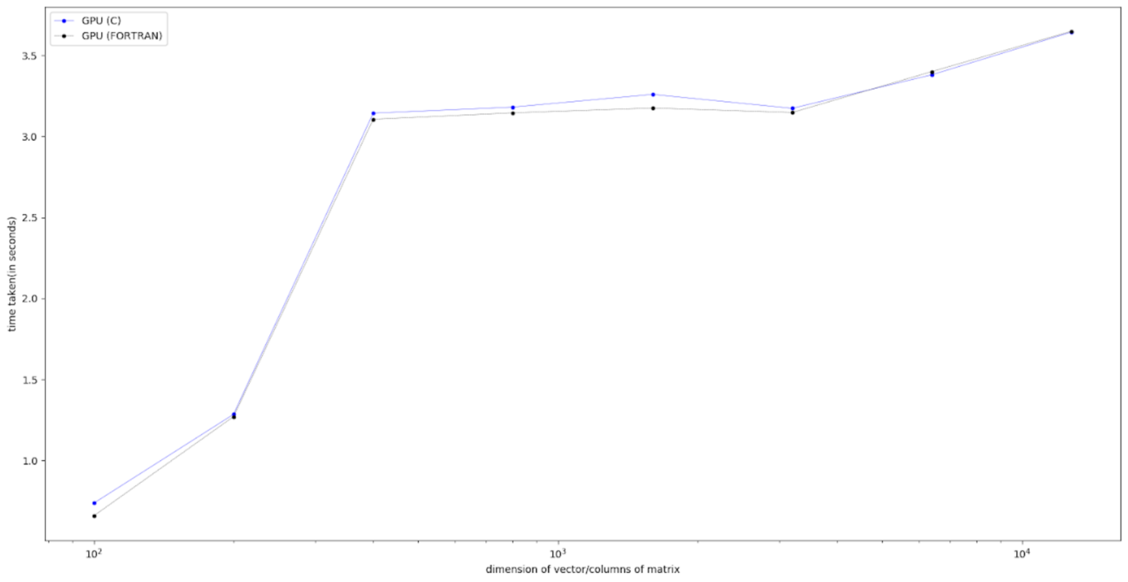


Figure 2.7: Benchmarking KSP based solver written in FORTRAN-Zoomed in

As we can see, that there is no appreciable difference between the runtimes of the same code written in C and FORTRAN when running on GPU, we can conclude that FORTRAN is able to leverage GPU at par with C.

2.2.5 Installing GPU-changed fluidity on changed PETSC

Running Singularity Script-III over the simg(singulairty image) output of the Script-II, which itself is built on the changed Script-I (changes discussed in sub-section 2.2.3) was a success with 54 failed tests and rest 1800 tests got passed.

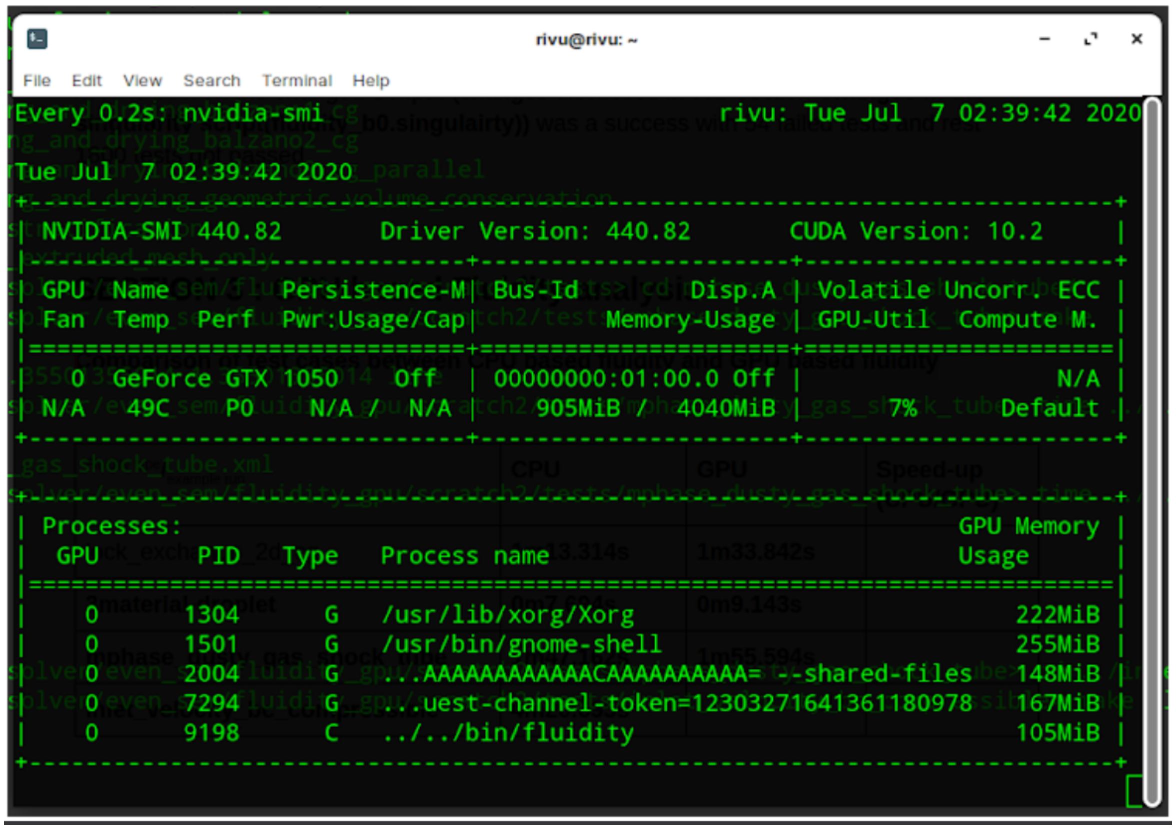
2.2.6 GPU based Fluidity analysis

Comparison of test cases between CPU based fluidity and GPU based fluidity

Fluidity type/example run	CPU	GPU	Speed-up (CPU/GPU)
lock_exchange_2d_cg	1m13.314s	1m33.842s	0.78
3material-droplet	0m7.694s	0m9.143s	0.84
mphase_dusty_gas_shock_tube	1m47.162s	1m55.594s	0.92
inlet_velocity_bc_compressible	4m20.695s	5m36.105s	0.78

2.2.7 Observations and deductions

Figure 2.8 shows the output of nvidia-smi on the system terminal outside the singularity shell. Presence of a fluidity process represents that the fluidity simulation is invoking using the GPU card. But



```
Every 0.2s: nvidia-smi                              rivu: Tue Jul  7 02:39:42 2020

Tue Jul  7 02:39:42 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.82       Driver Version: 440.82       CUDA Version: 10.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 1050     Off  | 00000000:01:00.0 Off |                  N/A |
| N/A   49C    P0    N/A /  N/A |    905MiB /  4040MiB |      7%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0      1304      G   /usr/lib/xorg/Xorg                           222MiB |
|    0      1501      G   /usr/bin/gnome-shell                         255MiB |
|    0      2004      G   ...AAAAAAAAAAAACAAAAAAAAAA= --shared-files   148MiB |
|    0      7294      G   ...uest-channel-token=12303271641361180978   167MiB |
|    0      9198      C   ../../bin/fluidity                           105MiB |
+-----------------------------------------------------------------------------+
```

Figure 2.8: NVIDIA-SMI output on terminal

the results in the previous subsection clearly indicate that there is degradation in performance. Tests examples run on standard Fluidity take less time on single core as compared to GPU compatible Fluidity. The speed up achieved is less than one. This is strictly against the intuition as well as the experimental results of Chapter 1 (work done in odd semester) where significant efficiency in speed up is observed on using GPU accelerator.

Possible Reasons :

- Hypothesis 1(more likely): It is possible that all petsc directives were not converted to their respective CUDA versions in the source code of Fluidity. As a result, few or all operations are still carried out in CPU and the excess time is due to the data transfer between the CPU and GPU to keep the data segments (Vectors and Matrices) synchronous in CPU and GPU.

- Hypothesis 2 (less likely): It is possible that PETSC is efficiently running on GPU but fluidity demands the values of the vectors and matrices time and again. Thus there is poor speedup due to excess data transfer overhead.

- Hypothesis 3 (less likely): It is possible that Fluidity is not as dependent on PETSc as assumed and thus incorporation of GPU compatible PETSc in Fluidity will not make huge difference.

2.2.8 Conclusion and Future Scope

The target to run the Fluidity library on GPU was achieved but there is no performance gain. A significant amount of work needs to be done to first understand the Fluidity in depth and then change its source code for better performance gains. All PETSC directives should be correctly changed with their CUDA versions in all the files of Fluidity source code. It will also be essential to trace these said files in the build process.

Bibliography

[1] Jacobsen, Dana, Thibault, Julien, Senocak, Inanc. (2010). An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters. Inanc Senocak.

[2] S. Cuomo, A. Galletti, G. Giunta, L. Marcellino. Toward a Multi-level Parallel Framework on GPU Cluster with PetSC-CUDA for PDE-based Optical Flow Computation, Procedia Computer Science, Volume 51, 2015.

[3] Portable, Extensible Toolkit for Scientific Computation, PETSc/Tao, www.mcs.anl.gov/petsc/

[4] Computational fluid dynamics (CFD), whatis.techtarget.com/definition/computational-fluid-dynamics-CFD

[5] Computational Science and Engineering, Gilbert Strang.

[6] PETSc Users Manual, Revision 3.12, Mathematics and Computer Science Division, Argonne National Laboratory, https://www.mcs.anl.gov/petsc/petsc-current/docs/manual.pdf

[7] PETSc Tutorial - Profiling, Nonlinear Solvers, Unstructured Grids, Threads and GPUs, https://www.mcs.anl.gov/petsc/meetings/2016/slides/tutorial2.pdf

[8] What are containers and why do you need them?, https://www.cio.com/article/2924995/what-are-containers-and-why-do-you-need-them.html

[9] The Portable Extensible Toolkit for Scientific Computing, https://www.mcs.anl.gov/petsc/documentation/tutorials/E Intro-Solvers.pdf

[10] Fluidity, https://fluidityproject.github.io/

[11] Fluidity, The Fluidity Project, https://github.com/FluidityProject/fluidity