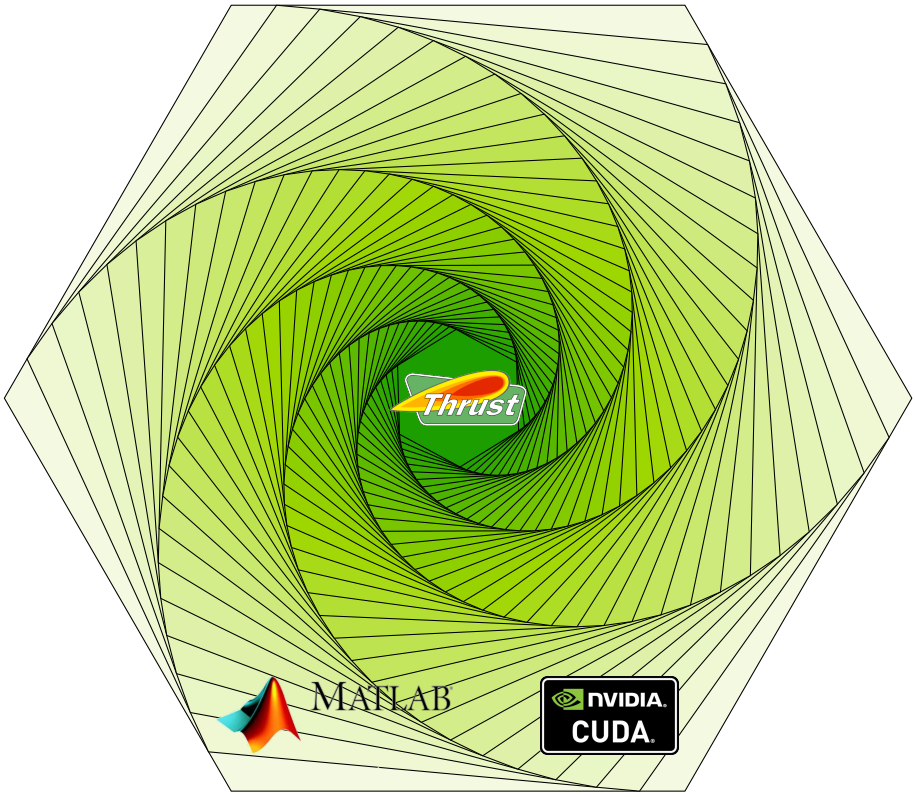
mThrust,

a Matlab Toolbox for GPU

User Guide



Francois Cuvelier
University Paris XIII / Institut Galilée
L.A.G.A./Département de Mathématiques
http://www.math.univ-paris13.fr/~cuvelier

Contents

1 Introduction

Thrust is a C++ template library for CUDA based on the Standard Template Library (STL). It's a powerful, open source library of parallel algorithms and data structures which can perform GPU-accelerated sort, scan, transform, and reductions with just a few lines of code. Since version 7.0 of CUDA, the Thrust library included manages the complex numbers.

The mThrust Matlab toolbox is an interface to the Thrust GPU-Accelerated Library included in CUDA Toolkit. It makes it possible to handle vectors directly stored on the host memory or on GPU memory.

This toolbox was tested on

- Ubuntu 14.04 LTS (updated the 10/12/2016) with CUDA version 7.0, 7.5 and 8.0 and Matlab Release 2015a to 2016b.
- Ubuntu 16.04 LTS (updated the 10/12/2016) with CUDA version 7.0, 7.5 and 8.0 and Matlab Release 2015a to 2016b.
- OpenSuse 13.2 (updated the 10/12/2016) with CUDA 8.0 and Matlab Release 2016b.
- CentOS 7 (updated the 10/20/2016) with CUDA version 7.0, 7.5 and 8.0 and Matlab Release 2015a to 2016b.

1.1 mThrust usage samples

1.1.1 Thrust sample

In Listing 1.1, the Matlab vector `H` is transfered on GPU device and stored on a `mThrust` object/vector `D`. Then this vector is directly modified.

Listing 1.1: First mThrust sample

```
H=[14,20,38,46];
D=mThrust(H,'device'); % transfert H on GPU device
D([1,2])=[99,98]
```

Here are the Matlab output of these commands :

```
D =

 thrust::device_vector<double>[4]
 99  98  38  46
```

1.1.2 Thrust sample

In Listing 1.2, three mThrust objects are used corresponding to vectors of 10 `double` stored on GPU device. All computations are done on GPU and the mThrust object `gy` is transfered to the Matlab vector `y`.

Listing 1.2: First mThrust sample

```
gx=mThrust(10,'device','double');
gx(1:5)=1:5;
gx(10)=pi;
gy=2*gx+1;
gz=cos(gy)-sin(gx);
disp(gx)
disp(gy)
disp(gz)
y=gather(gy)
```

Here are the Matlab output of these commands :

```
 thrust::device_vector<double>[10]
 1  2  3  4  5
0  0  0  0  3.14159
 thrust::device_vector<double>[10]
 3  5  7  9  11
1  1  1  1  7.28319
 thrust::device_vector<double>[10]
 -1.83146  -0.625635  0.612782  -0.154328  0.96335
0.540302  0.540302  0.540302  0.540302  0.540302

y =

    3.0000
    5.0000
    7.0000
    9.0000
   11.0000
    1.0000
    1.0000
    1.0000
    1.0000
    7.2832
```

2 Installation

We suppose here that you have the Matlab software (release greater than 2015a) correctly installed on your **linux** computer with the CUDA Toolkit (version greater than 7.0 for complex numbers). And so your computer has an Nvidia GPU device...

For macOS, it is possible that the installation is held without major change but I do not have MAC at hand to try it ...

For Windows OS, major changes in compilation process are needed.

There are two methods for install/prepare this toolbox. The first one uses the Matlab function `Thrust.compile.build` and the second one uses Makefile.

One can note that the Matlab function `Thrust.compile.build` is just an interface to the Linux `make` command and also uses Makefile...

The compilation of all the source files in the `mThrust` toolbox take twenty minutes on my laptop! So be patient...

2.1 Get the `mThrust` toolbox

To do

2.2 Installation via Matlab

Under Matlab, the function `Thrust.compile.build` try to properly configure the toolbox for CUDA/Thrust compilation. The default directory for CUDA is set to `/usr/local/cuda`. One can change this path by using the `'cudapath'` option :

```
Thrust.compile.build('cudapath','/usr/local/cuda-7.5')
```

This function automatically change the file `build/config` (see next section) to reflect the current configuration.

If using Matlab `R2016b` release with CUDA 7.5, then all the MEX-files will be created in directory `build/lib/CUDA_7.5/Matlab_R2016b`.

With `'debug'` option set to `true`, one can create a debug version of all the mex files stored in directory `build/lib/CUDA_7.5/Matlab_R2016b_debug`. In this case, the command is :

```
Thrust.compile.build('cudapath','/usr/local/cuda-7.5','debug',true)
```

2.3 Installation via Makefile

The first step is to properly (and manually) configure the `mThrust` toolbox to prepare the compilation.

- We have to edit the file `build/config` which will be included in Makefile. In this file we set the five variables `CUDA_PATH`, `CUDA_VERSION`, `SMS` (Streaming Multiprocessors, compute capability version), `MATLAB_PATH` and `MATLAB_VERSION` to accurately reflect the computer configuration. For example, on my laptop this file contains :

  ```
  CUDA_PATH = /usr/local/cuda
  ```

```
CUDA_VERSION = 8.0
SMS = 30
MATLAB_PATH = /usr/local/MATLAB/R2016b
MATLAB_VERSION = R2016b
```

- Make sure that the makefile for your CUDA version is in the `build` directory with the name `build/Makefile.cuda-$(CUDA_VERSION)` otherwise you must create it. For that one can be inspired by Makefile given with CUDA samples. Actually, makefile for CUDA 7.0, 7.5 and 8.0 are present in the `build` directory.

Thereafter, on the root folder of the toolbox, one can use the command `make` to build all the MEX-files stored in directory:
`build/lib/CUDA_$(CUDA_VERSION)/Matlab_$(MATLAB_VERSION)`

By using command `make libdebug` we create a debug version of all the mex files stored in directory:
`build/lib/CUDA_$(CUDA_VERSION)/Matlab_$(MATLAB_VERSION)_debug`

3 Usefull tools

3.1 Initialization of the mThrust toolbox

To initialize the toolbox, one can used the `Thrust.init` function.

Listing 3.1: Initialize `mThrust` toolbox

```
Thrust.init()
```

By default this function print some informations :

```
mThrust toolbox configuration
  CUDA version : 8.0
  CUDA sms     : 30
  CUDA path    : /usr/local/cuda
```

3.2 Initialization of the mThrust toolbox in debug mode

If the `mThrust` toolbox was compiled with debug option to true, then one can initialize the toolbox in debug mode by using the matlab command `Thrust.init('debug',true)`. Thereafter, when using mThrust object and functions, some debugging informations can be printed. For example, when a mThrust object is created or deleted on GPU device, its memory address is printed in green color.

Here we give an example of Matlab session with the `mThrust` toolbox in debug mode:

```
>> Thrust.init('debug',true)
mThrust toolbox configuration (debug mode)
  CUDA version : 8.0
  CUDA sms     : 30
```

```
  CUDA path    : /usr/local/cuda
>> gx=Thrust.ones(5)
 -> mex_mThrust_create.cu:75 : starting
Lock mex file : mex_mThrust_create.cu...
create : device_vector<double>[5] (memory adress: 0X7F70D7A86960)
 -> mex_mThrust_create.cu:124 : ending
 -> mex_mThrust_set.cu:33 : starting
 -> mex_mThrust_set.cu:95 : ending

gx =

 -> mex_mThrust_disp.cu:119 : starting
 thrust::device_vector<double>[5]
 1  1  1  1  1
 -> mex_mThrust_disp.cu:150 : ending
>> gx=1
 -> mex_mThrust_delete.cu:73 : starting
delete : device_vector<double>[5] (memory adress: 0X7F70D7A86960)
 -> mex_mThrust_delete.cu:103 : ending

gx =

     1
```

3.3 CUDA interfaces

Some functions of the CUDA API were interfaced.

- cuda.DeviceReset : destroy all allocations and reset all state on the current device in the current process.
  Interface to the `cudaDeviceReset` function of the CUDA Runtime API.
- cuda.DeviceSynchronize : wait for compute device to finish.
  Interface to the `cudaDeviceSynchronize` function of the CUDA Runtime API.
- cuda.GetDevice : returns which device is currently being used.
  Interface to the `cudaGetDevice` function of the CUDA Runtime API.
- cuda.GetDeviceCount : returns the number of compute-capable devices.
  Interface to the `cudaGetDeviceCount` function of the CUDA Runtime API.
- cuda.GetDeviceProperties : returns information about the compute-device number 0.
  With the optinal input parameter `NumDev`, returns information about the compute-device number `NumDev`.
  Interface to the `cudaGetDeviceCount` function of the CUDA Runtime API.

On my laptop computer, the prop=cuda.GetDeviceProperties Matlab command gives as output

```
prop =

  struct with fields:

                  Name: 'Quadro K1100M'
                 Index: 0
      ComputeCapability: '3.0'
         RuntimeVersion: 8
          DriverVersion: 8
            TotalMemory: 2.0868e+09
        AvailableMemory: 1.6210e+09
    MultiprocessorCount: 2
              CoresByMP: 192
              CudaCores: 384
     GPUMaxClockRateMhz: 705.5001
     MaxThreadsPerBlock: 1024
      SharedMemPerBlock: 49152
     MaxThreadBlockSize: [1024 1024 64]
            MaxGridSize: [2.1475e+09 65535 65535]
```

3.4 CUDA/MEX compilation

The Matlab function `Thrust.compile.mex` can build a mex file from CUDA source code using or not Thrust API. The CUDA source code must be with a `.cu` file extension and must contain a `mexFunction` as for all C/C++ MEX-file. (see Matlab documentation mexfunction.html.

The syntax of this function is :
`Thrust.compile.mex(cudafile)`
where `cudafile` is a string containing the name of the CUDA MEX-file with its relative path if not in current directory. One can specifies some options to this command :
`Thrust.compile.mex(...,Name,Value)` specifies options using one or more option name, option value pairs. Name is the option name and must appear inside single quotes ('). Value is the corresponding value.

Here are the different options :

- `Thrust.compile.mex(...,'outpath',Value)`, specifies where the MEX file will be written. Value is a string (current directory as default)
- `Thrust.compile.mex(...,'force',Value)`, specifies if compilation is forced or not. Value is a logical (false as default)
- `Thrust.compile.mex(...,'samepath',Value)`, specifies if the MEX file will be written in the same path as input file. Value is a logical (false as default)

- `Thrust.compile.mex(...,'autopath',Value)`, specifies if the MEX file will be written in a directory automaticaly selected. Value is a logical (false as default)
- `Thrust.compile.mex(...,'dep',Value)`, specifies dependencies for Makefile Value is a string (empty as default)

This function is used for every CUDA MEX-file compiled via the Matlab `Thrust.compile.build` command used in section 2.2.

As a simple example, we want a function which returns the free and total device memory in MB. So we have to interface, the CUDA Runtime API function `cudaMemGetInfo`. The complete code is given in Listing 3.2.

Listing 3.2: Interface to the function `cudaMemGetInfo` of the CUDA Runtime API

```
#include <cuda.h>
#include "mex.h"

void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[] ){
  if (nlhs !=2 || nrhs !=0)
    mexErrMsgTxt("Bad input... To do");
  mwSize dims[2]={1,1};
  size_t free_byte,tot_byte;
  cudaMemGetInfo(&free_byte,&tot_byte); // in Byte
  plhs[0]=mxCreateNumericArray(2, dims, mxDOUBLE_CLASS, mxREAL);
  double *pmem_free = mxGetPr(plhs[0]);
  *pmem_free = ((double)free_byte)/1024.0/1024.0; // in Megabyte
  plhs[1]=mxCreateNumericArray(2, dims, mxDOUBLE_CLASS, mxREAL);
  double *pmem_tot = mxGetPr(plhs[1]);
  *pmem_tot = ((double)tot_byte)/1024.0/1024.0; // in Megabyte
}
```

To compile the MEX file `cudaMemGetInfo.cu` present in the current directory we use the following command :

Listing 3.3: Compilation sample

```
Thrust.compile.mex('cudaMemGetInfo.cu')
```

Here are the Matlab output of these commands :

```
[Matlab] Build file Makefile.mThrust
[Matlab] Run system command "make -f Makefile.mThrust"
    That can take a few minutes ... be patient
********** NVCC compiler *************
NVCC compiler used to build cudaMemGetInfo.o ...
********** MEX  compiler *************
MEX compiler used to build cudaMemGetInfo.mexa64 ...
Building with 'g++'.
MEX completed successfully.
```

We give a sample of usage of this function on Listing 3.4.

Listing 3.4: Usage of `cudaMemGetInfo` MEX function

```
[freemem,totmem]=cudaMemGetInfo();
fprintf('Free memory: %.2f(Mb), Total memory:%.2f(Mb)\n', ...
    freemem,totmem);
gx=Thrust.ones(8*1024*1024);
[freemem,totmem]=cudaMemGetInfo();
fprintf('Free memory: %.2f(Mb), Total memory:%.2f(Mb)\n', ...
    freemem,totmem);
```

Here are the Matlab output of these commands :

```
Free memory: 1375.06(Mb), Total memory:1990.12(Mb)
Free memory: 1310.06(Mb), Total memory:1990.12(Mb)
```

A more complex example using CUDA kernel is given in section 6.1 page 36.

4 mThrust objects

mThrust objects are 1D-arrays (or vectors) which could be stored on two containers : 'device' for GPU device (default) or 'host' for usual RAM memory. The type of mThrust objects could be 'double', 'float', 'int', 'unsigned' or 'logical'. The numerical mThrust objects (i.e. not 'logical') can also been complex.

4.1 Constructors

```
gx=mThrust(dim,container,type,<iscomplex>)
```

Create a mThrust vector of type `type`, dimension `dim` and stored on `container`. The optional 4th parameter denoted by <iscomplex> is logical. If true, then the mThrust vector is complex (only if `type` is not logical) otherwise is not.

We give in Listing 4.1, three contructors samples. The first one create a vector of 10000 double on the GPU device memory. The second, create a vector of 10000 int on the host memory. And, the last one create a vector of 10000 complex float on the GPU device memory.

Listing 4.1: mThrust constructor

```
gx=mThrust(10000,'device','double')
y=mThrust(10000,'host','int')
gz=mThrust(10000,'device','float',true)
```

Here are the Matlab output of these two commands :

```
gx =

 thrust::device_vector<double>[10000]
 0  0  0  0  0
 . . .
 0  0  0  0  0

y =

 thrust::host_vector<int>[10000]
```

```
 0  0  0  0  0
 . . .
 0  0  0  0  0

gz =

 thrust::device_vector<complex<float>>[10000]
 (0,0)  (0,0)  (0,0)  (0,0)  (0,0)
 . . .
 (0,0)  (0,0)  (0,0)  (0,0)  (0,0)
```

```
gA=mThrust(A,container)
```

Create a mThrust vector which is a copy on `container` of Matlab vector `A` (same dimension and type).

We give in Listing 4.2, two contructors samples. The first one tranfer the Matlab vector `A` (11 double) on the GPU device memory. The second, transfer the Matlab vector `A` (10 int32) on the GPU device memory.

Listing 4.2: mThrust constructor by transfer

```
A=0:0.1:1;
gA=mThrust(A,'device')
I=int32(1:10);
gI=mThrust(I,'device')
```

Here are the Matlab output of these two commands :

```
gA =

 thrust::device_vector<double>[11]
 0  0.1  0.2  0.3  0.4
 . . .
 0.6  0.7  0.8  0.9  1

gI =

 thrust::device_vector<int>[10]
 1  2  3  4  5
6  7  8  9  10
```

```
gB=mThrust(gA)
```

Create a new mThrust vector which is a copy of a mThrust vector `gA` (same dimension, type, ...).

We give in Listing 4.2, two contructors samples. The first one transfer the Matlab vector `A` (11 double) on the GPU device memory. The second, make a raw copy the mThrust vector `gA` on a new mThrust vector `gB`. The last line copy the mThrust object `gA` in the mThrust object `gC`: these two objects point to the same Thrust object on GPU memory.

Listing 4.3: mThrust constructor by copy

```
A=0:0.1:1;
gA=mThrust(A,'device')
gB=mThrust(gA)
gC=gA
```

Here are the Matlab output of these two commands :

```
gA =

 thrust::device_vector<double>[11]
 0  0.1  0.2  0.3  0.4
 . . .
 0.6  0.7  0.8  0.9  1

gB =

 thrust::device_vector<double>[11]
 0  0.1  0.2  0.3  0.4
 . . .
 0.6  0.7  0.8  0.9  1

gC =

 thrust::device_vector<double>[11]
 0  0.1  0.2  0.3  0.4
 . . .
 0.6  0.7  0.8  0.9  1
```

The `mThrust` vector `gB` is linked to a new Thrust vector on the GPU device memory and the `mThrust` vector `gC` is linked to the same Thrust vector pointed by `gA`. One can see it by using `debug` method of mThrust object:

```
>> debug(gA)
  - Thrust device_vector<double>[11]
      memory address: 0X7F7D53F05F30
                size: 88 octets
>> debug(gB)
  - Thrust device_vector<double>[11]
      memory address: 0X7F7D53ECC170
                size: 88 octets
>> debug(gC)
  - Thrust device_vector<double>[11]
      memory address: 0X7F7D53F05F30
                size: 88 octets
```

So if we modify `gB` then `gA` is unchanged and if we modify `gC`, `gA` is also modified :

```
>> gB(1:3)=1
gB =
 thrust::device_vector<double>[11]
```

```
 1  1  1  0.3  0.4
 . . .
 0.6  0.7  0.8  0.9  1
>> gA
gA =
 thrust::device_vector<double>[11]
 0  0.1  0.2  0.3  0.4
 . . .
 0.6  0.7  0.8  0.9  1
>> gC(1:3)=2
gC =
 thrust::device_vector<double>[11]
 2  2  2  0.3  0.4
 . . .
 0.6  0.7  0.8  0.9  1
>> gA
gA =
 thrust::device_vector<double>[11]
 2  2  2  0.3  0.4
 . . .
 0.6  0.7  0.8  0.9  1
```

5 Implemented functions

5.1 Particular Constructors

5.1.1 Thrust.linspace

Generate linearly spaced `mThrust` vector.

```
gx=mThrust.linspace(a,b,n);
```

generates n points. The spacing between the points is (b-a)/(n-1). Returns the n points stored in a `mThrust` vector of double on GPU device.

```
gx=Thrust.linspace(a,b,n,Name,Value);
```

Use Name,Value pairs to specify the container and type of datas.

- With Name as 'container', Value could be 'device' for GPU device (default) or 'host' for usual RAM memory.
- With Name as 'type', Value could be 'double' or 'float'.

Listing 5.1: mThrust linspace constructor

```
gx=Thrust.linspace(0,1,11)
gy=Thrust.linspace(0,1,11,'container','host')
```

```
gx =

 thrust::device_vector<double>[11]
 0  0.1  0.2  0.3  0.4
 . . .
 0.6  0.7  0.8  0.9  1

gy =

 thrust::host_vector<double>[11]
 0  0.1  0.2  0.3  0.4
 . . .
 0.6  0.7  0.8  0.9  1
```

5.1.2 Thrust.ones

Generate `mThrust` vector of ones.

```
gx=Thrust.ones(n);
```

generates a `mThrust` vector with size n of ones on GPU device (as double).

```
gx=Thrust.ones(n,Name,Value);
```

Use Name,Value pairs to specify the container and type of datas.

- With Name as 'container', Value could be 'device' for GPU device (default) or 'host' for usual RAM memory.
- With Name as 'type', Value could be 'double', 'float', 'int', 'unsigned' or 'logical'.

Listing 5.2: mThrust ones constructor

```
gx=Thrust.ones(100)
gy=Thrust.ones(100,'type','int')
gz=Thrust.ones(100,'container','host','type','float')
```

```
gx =

 thrust::device_vector<double>[100]
 1  1  1  1  1
 . . .
 1  1  1  1  1

gy =

 thrust::device_vector<int>[100]
 1  1  1  1  1
 . . .
 1  1  1  1  1
```

```
gz =

 thrust::host_vector<float>[100]
 1  1  1  1  1
 . . .
 1  1  1  1  1
```

5.1.3 Thrust.rand

Returns a `mThrust` vector with uniformly distributed pseudorandom numbers.

```
gx=Thrust.rand(n);
```

generates a `mThrust` vector of dimension n with uniformly distributed pseudo-random numbers on GPU device (as double).

```
gx=Thrust.rand(n,Name,Value);
```

Use Name,Value pairs to specify optional parameters

- With Name as 'container', Value could be 'device' for GPU device (default) or 'host' for usual RAM memory.
- With Name as 'type', Value could be 'double' or 'float'.
- With Name as 'interval', Value is the interval [a, b] to generate values from the uniform distribution on the interval [a, b]

Listing 5.3: mThrust rand constructor

```
gx=Thrust.rand(100)
gy=Thrust.rand(1000,'interval',[-2,2])
gz=Thrust.rand(10^6,'container','host','type','float')
```

```
gx =

 thrust::device_vector<double>[100]
 0.664163  0.79109  0.51133  0.0455485  0.185651
 . . .
 0.0773698  0.0324065  0.923051  0.697965  0.438272

gy =

 thrust::device_vector<double>[1000]
 0.846738  -1.38529  1.81422  0.901886  1.01901
 . . .
 -1.60307  1.51494  -0.75851  -1.19596  -0.520456

gz =
```

```
thrust::host_vector<float>[1000000]
0.985151  0.0358211  0.85322  0.707773  0.694495
. . .
0.480608  0.373391  0.656636  0.327292  0.712194
```

5.1.4 Thrust.randn

Return a `mThrust` vector with normally distributed pseudorandom numbers.

```
gx=Thrust.randn(n);
```

generates a `mThrust` vector of dimension n containing pseudorandom values drawn from the standard normal distribution on GPU device (as double).

```
gx=Thrust.randn(n,Name,Value);
```

Use Name,Value pairs to specify optional parameters

- With Name as 'container', Value could be 'device' for GPU device (default) or 'host' for usual RAM memory.
- With Name as 'type', Value could be 'double' or 'float'.
- With Name as 'mu', Value is the mean of the normal distribution.
- With Name as 'sigma', Value is the standard deviation of the normal distribution.

Listing 5.4: mThrust randn constructor

```
gx=Thrust.randn(100)
gy=Thrust.randn(10^5,'mu',2,'sigma',0.5)
gz=Thrust.randn(100,'container','host','type','float')
```

```
gx =

 thrust::device_vector<double>[100]
 0.423852  0.81021  0.0284027  -1.68964  -0.894038
 . . .
 -1.42299  -1.84655  1.4259  0.518557  -0.155353

gy =

 thrust::device_vector<double>[100000]
 2.27916  1.48961  2.84017  2.29959  2.34476
 . . .
 1.83797  2.26261  1.68908  2.48642  1.71502

gz =

 thrust::host_vector<float>[100]
 2.1741  -1.80139  1.05034  0.546891  0.508632
```

```
  . . .
  1.95347   -0.0606639   -0.26426   -0.645634   0.568501
```

5.1.5 Thrust.sequence

Generate `mThrust` vector which contains a sequence of numbers.

```
gx=Thrust.sequence(n);
```

generates the sequence $0 : n-1$ in a `mThrust` vector with size n on GPU device (as double).

```
gx=Thrust.zeros(n,Name,Value);
```

Use Name,Value pairs to specify the container and type of datas.

- With Name as 'container', Value could be 'device' for GPU device (default) or 'host' for usual RAM memory.
- With Name as 'type', Value could be 'double', 'float', 'int', 'unsigned' or 'logical'.
- With Name as 'init', Value is the initial number (default 0).
- With Name as 'step', Value is the step number (default 1).

Listing 5.5: mThrust sequence constructor

```
gx=Thrust.sequence(100)
gy=Thrust.sequence(100,'type','int','init',1,'step',2)
gz=Thrust.sequence(100,'container','host','step',0.5)
```

```
gx =

 thrust::device_vector<double>[100]
 0  1  2  3  4
 . . .
 95  96  97  98  99

gy =

 thrust::device_vector<int>[100]
 1  3  5  7  9
 . . .
 191  193  195  197  199

gz =

 thrust::host_vector<double>[100]
 0  0.5  1  1.5  2
 . . .
```

```
  47.5  48  48.5  49  49.5
```

5.1.6 Thrust.zeros

Generate `mThrust` vector of zeros.

```
gx=Thrust.zeros(n);
```

generates a `mThrust` vector with size n of zeros on GPU device (as double).

```
gx=Thrust.zeros(n,Name,Value);
```

Use Name,Value pairs to specify the container and type of datas.

- With Name as 'container', Value could be 'device' for GPU device (default) or 'host' for usual RAM memory.
- With Name as 'type', Value could be 'double', 'float', 'int', 'unsigned' or 'logical'.

Listing 5.6: mThrust zeros constructor

```
gx=Thrust.zeros(100)
gy=Thrust.zeros(100,'type','int')
gz=Thrust.zeros(10^6,'container','host','type','float')
```

```
gx =

 thrust::device_vector<double>[100]
 0  0  0  0  0
 . . .
 0  0  0  0  0

gy =

 thrust::device_vector<int>[100]
 0  0  0  0  0
 . . .
 0  0  0  0  0

gz =

 thrust::host_vector<float>[1000000]
 0  0  0  0  0
 . . .
 0  0  0  0  0
```

5.2 Arithmetic operators

5.2.1 Binary and unary + operator

Listing 5.7: mThrust + operator

```
gx=Thrust.ones(5);
gy=Thrust.linspace(0,1,5);
gz=gx+gy
gx+1
10+gy
gw = +gx
```

```
gz =

 thrust::device_vector<double>[5]
 1  1.25  1.5  1.75  2

ans =

 thrust::device_vector<double>[5]
 2  2  2  2  2

ans =

 thrust::device_vector<double>[5]
 10  10.25  10.5  10.75  11

gw =

 thrust::device_vector<double>[5]
 1  1  1  1  1
```

5.2.2 Binary and unary − operator

Listing 5.8: mThrust − operator

```
gx=Thrust.ones(5);
gy=Thrust.linspace(0,1,5);
gz=gx-gy
gx-1
10-gy
gw=-gx
```

```
gz =

 thrust::device_vector<double>[5]
 1  0.75  0.5  0.25  0

ans =
```

```
 thrust::device_vector<double>[5]
 0  0  0  0  0

ans =

 thrust::device_vector<double>[5]
 10  9.75  9.5  9.25  9

gw =

 thrust::device_vector<double>[5]
 -1  -1  -1  -1  -1
```

5.2.3 Binary $*$ operator

Listing 5.9: mThrust $*$ operator

```
gx=Thrust.ones(5);
gy=Thrust.linspace(0,1,5);
gz=2*gy
gw=gx*2
```

```
gz =

 thrust::device_vector<double>[5]
 0  0.5  1  1.5  2

gw =

 thrust::device_vector<double>[5]
 2  2  2  2  2
```

5.2.4 Binary / operator

Listing 5.10: mThrust / operator

```
gx=Thrust.ones(5);
gy= gx/2
```

```
gy =

 thrust::device_vector<double>[5]
 0.5  0.5  0.5  0.5  0.5
```

5.2.5 Binary .* element-by-element multiplication operator

Listing 5.11: mThrust .* operator

```
gx=2*Thrust.ones(5);
gy=Thrust.linspace(0,1,5);
gz=gx.*gy
gw=gx.*2
gt=2.*gx
```

```
gz =

 thrust::device_vector<double>[5]
 0  0.5  1  1.5  2

gw =

 thrust::device_vector<double>[5]
 4  4  4  4  4

gt =

 thrust::device_vector<double>[5]
 4  4  4  4  4
```

5.2.6 Binary ./ element-by-element division operator

Listing 5.12: mThrust ./ operator

```
gx=Thrust.linspace(0,1,5);
gy=Thrust.linspace(1,2,5);
gz=gx./gy
gx./2
2./gx
```

```
gz =

 thrust::device_vector<double>[5]
 0  0.2  0.333333  0.428571  0.5

ans =

 thrust::device_vector<double>[5]
 0  0.125  0.25  0.375  0.5

ans =

 thrust::device_vector<double>[5]
 0  0.125  0.25  0.375  0.5
```

5.3 Set Operations

Union, intersection, difference of two `mThrust` vectors

5.3.1 Thrust.union

Set union of two `mThrust` vectors

Listing 5.13: Thrust.union function

```
gx=Thrust.sequence(11)
gy=Thrust.sequence(11,'init',-5)
gz=Thrust.union(gx,gy)
```

```
gx =

 thrust::device_vector<double>[11]
 0  1  2  3  4
 . . .
 6  7  8  9  10

gy =

 thrust::device_vector<double>[11]
 -5  -4  -3  -2  -1
 . . .
 1  2  3  4  5

gz =

 thrust::device_vector<double>[16]
 -5  -4  -3  -2  -1
 . . .
 6  7  8  9  10
```

5.3.2 Thrust.intersect

Set intersection of two `mThrust` vectors

Listing 5.14: Thrust.intersect function

```
gx=Thrust.sequence(11)
gy=Thrust.sequence(11,'init',-5)
gz=Thrust.intersect(gx,gy)
```

```
gx =

 thrust::device_vector<double>[11]
 0  1  2  3  4
 . . .
 6  7  8  9  10
```

```
gy =

 thrust::device_vector<double>[11]
 -5  -4  -3  -2  -1
 . . .
 1  2  3  4  5

gz =

 thrust::device_vector<double>[6]
 0  1  2  3  4
5
```

5.3.3 Thrust.setdiff

Set difference of two `mThrust` vectors

Listing 5.15: `Thrust.setdiff` function

```
gx=Thrust.sequence(11)
gy=Thrust.sequence(11,'init',-5)
gz=Thrust.setdiff(gx,gy)
```

```
gx =

 thrust::device_vector<double>[11]
 0  1  2  3  4
 . . .
 6  7  8  9  10

gy =

 thrust::device_vector<double>[11]
 -5  -4  -3  -2  -1
 . . .
 1  2  3  4  5

gz =

 thrust::device_vector<double>[5]
 6  7  8  9  10
```

5.4 Subscripted reference and assignment

5.4.1 Indexed Reference

Listing 5.16: Subscripted reference

```
gx=Thrust.sequence(11)
gy=gx([1:4,8])
gx([end-3:end])
```

```
gx =

 thrust::device_vector<double>[11]
 0  1  2  3  4
 . . .
 6  7  8  9  10

gy =

 thrust::device_vector<double>[5]
 0  1  2  3  7

ans =

 thrust::device_vector<double>[4]
 7  8  9  10
```

5.4.2 Subscripted assignment

Listing 5.17: Subscripted assignment

```
gx=Thrust.sequence(11)
gy=Thrust.set(20,-1)
gx([1:4,8])=1
gx(9:11)=1:3
gy(1:4)=gx([end-3:end])
```

```
gx =

 thrust::device_vector<double>[11]
 0  1  2  3  4
 . . .
 6  7  8  9  10

gy =

 thrust::device_vector<double>[20]
 -1  -1  -1  -1  -1
 . . .
 -1  -1  -1  -1  -1

gx =

 thrust::device_vector<double>[11]
 1  1  1  1  4
```

```
 . . .
 6  1  8  9  10

gx =

 thrust::device_vector<double>[11]
 1  1  1  1  4
 . . .
 6  1  1  2  3

gy =

 thrust::device_vector<double>[20]
 1  1  2  3  -1
 . . .
 -1  -1  -1  -1  -1
```

5.5 Arithmetic functions

5.5.1 Function sum

Sum of mThrust elements.

Listing 5.18: **sum** function

```
gy=Thrust.ones(10)
sum(gy)
```

```
gy =

 thrust::device_vector<double>[10]
 1  1  1  1  1
1  1  1  1  1

ans =

    10
```

5.5.2 Function prod

Product of mThrust elements.

Listing 5.19: **prod** function

```
gy=2*Thrust.ones(10)
prod(gy)
```

```
gy =

 thrust::device_vector<double>[10]
 2  2  2  2  2
2  2  2  2  2

ans =

        1024
```

5.6 Equality, Relational, and Conditional Operators

5.6.1 Binary equality operators

The binary equality operators compare their operands for strict equality or inequality. The equality operators, equal to == and not equal to ~=

Listing 5.20: mThrust == operator

```
gx=Thrust.linspace(0,1,5)
gy=Thrust.linspace(-1,2,5)
gz=(gx==gy)
```

```
gx =

 thrust::device_vector<double>[5]
 0  0.25  0.5  0.75  1

gy =

 thrust::device_vector<double>[5]
 -1  -0.25  0.5  1.25  2

gz =

 thrust::device_vector<logical>[5]
 0  0  1  0  0
```

5.6.2 Binary relational operators

The relational operators determine if one operand is greater than > , less than <, greater than or equal to >=, less than or equal to <= to another operand

Listing 5.21: mThrust < operator

```
gx=Thrust.linspace(0,1,5)
gy=Thrust.linspace(-1,2,5)
gz=(gx<gy)
```

```
gx =

 thrust::device_vector<double>[5]
 0  0.25  0.5  0.75  1

gy =

 thrust::device_vector<double>[5]
 -1  -0.25  0.5  1.25  2

gz =

 thrust::device_vector<logical>[5]
 0  0  0  1  1
```

5.6.3 Logical operators

The logical operators & (and), | (or)

Listing 5.22: mThrust & operator

```
gx=Thrust.linspace(0,1,5)
gy=Thrust.linspace(-1,2,5)
gz=(gx<gy)&(gy>=1.5)
```

```
gx =

 thrust::device_vector<double>[5]
 0  0.25  0.5  0.75  1

gy =

 thrust::device_vector<double>[5]
 -1  -0.25  0.5  1.25  2

gz =

 thrust::device_vector<logical>[5]
 0  0  0  0  1
```

5.7 Mathematical functions

5.7.1 double fun(double) or float fun(float)

There is the list of the mathematical functions which takes as input value a `mThrust` vector of type double or float and returns a `mThrust` vector with same size an same type.

`acos :` Calculate the arc cosine of the input argument.

`acosh` : Calculate the nonnegative arc hyperbolic cosine of the input argument.

`asin` : Calculate the arc sine of the input argument.

`asinh` : Calculate the arc hyperbolic sine of the input argument.

`atan` : Calculate the arc tangent of the input argument.

`atan2` : Calculate the arc tangent of the ratio of first and second input arguments.

`atanh` : Calculate the arc hyperbolic tangent of the input argument.

`cbrt` : Calculate the cube root of the input argument.

`ceil` : Calculate ceiling of the input argument.

`cos` : Calculate the cosine of the input argument.

`cosh` : Calculate the hyperbolic cosine of the input argument.

`cospi` : Calculate the cosine of the input argument x*pi

`cyl_bessel_i0` : Calculate the value of the regular modified cylindrical Bessel function of order 0 for the input argument.

`cyl_bessel_i1` : Calculate the value of the regular modified cylindrical Bessel function of order 1 for the input argument.

`erf` : Calculate the error function of the input argument.

`erfc` : Calculate the complementary error function of the input argument.

`erfcinv` : Calculate the inverse complementary error function of the input argument.

`erfcx` : Calculate the scaled complementary error function of the input argument.

`erfinv` : Calculate the inverse error function of the input argument.

`exp` : Calculate the base e exponential of the input argument.

`exp10` : Calculate the base 10 exponential of the input argument.

`exp2` : Calculate the base 2 exponential of the input argument.

`expm1` : Calculate the base e exponential of the input argument, minus 1.

`fabs` : Calculate the absolute value of the input argument.

`floor` : Calculate the largest integer less than or equal to the input argument.

`j0` : Calculate the value of the Bessel function of the first kind of order 0 for the input argument.

`j1` : Calculate the value of the Bessel function of the first kind of order 1 for the input argument.

`jn` : Calculate the value of the Bessel function of the first kind of order n for the input argument.

`lgamma` : Calculate the natural logarithm of the absolute value of the gamma function of the input argument.

`log` : Calculate the base e logarithm of the input argument.

`log10` : Calculate the base 10 logarithm of the input argument.

`log1p` : Calculate the value of loge(1 + x).

`log2` : Calculate the base 2 logarithm of the input argument.

`nearbyint` : Round the input argument to the nearest integer.

`normcdf` : Calculate the standard normal cumulative distribution function.

`normcdfinv` : Calculate the inverse of the standard normal cumulative distribution function.

`rcbrt` : Calculate reciprocal cube root function.

`rint` : Round to nearest integer value in floating-point.

`round` : Round to nearest integer value in floating-point.

`rsqrt` : Calculate the reciprocal of the square root of the input argument.

`sin` : Calculate the sine of the input argument.

`sinh` : Calculate the hyperbolic sine of the input argument.

`sinpi` : Calculate the sine of the input argument x*pi.

`sqrt` : Calculate the square root of the input argument.

`tan` : Calculate the tangent of the input argument.

`tanh` : Calculate the hyperbolic tangent of the input argument.

`tgamma` : Calculate the gamma function of the input argument.

`trunc` : Truncate input argument to the integral part.

`y0` : Calculate the value of the Bessel function of the second kind of order 0 for the input argument.

`y1` : Calculate the value of the Bessel function of the second kind of order 1 for the input argument.

As sample, we compute the cosine of an `mThrust` vector

Listing 5.23: Cosine function on `mThrust` vector

```
gx=Thrust.linspace(0,2*pi,1000)
gy=cos(gx)
gfx=Thrust.linspace(0,2*pi,1000,'type','float')
gfy=cos(gfx)
```

```
gx =

 thrust::device_vector<double>[1000]
 0  0.00628947  0.0125789  0.0188684  0.0251579
 . . .
 6.25803  6.26432  6.27061  6.2769  6.28319

gy =

 thrust::device_vector<double>[1000]
 1  0.99998  0.999921  0.999822  0.999684
 . . .
 0.999684  0.999822  0.999921  0.99998  1

gfx =

 thrust::device_vector<float>[1000]
 0  0.00628947  0.0125789  0.0188684  0.0251579
 . . .
 6.25803  6.26432  6.27061  6.2769  6.28319

gfy =

 thrust::device_vector<float>[1000]
 1  0.99998  0.999921  0.999822  0.999684
 . . .
 0.999684  0.999822  0.999921  0.99998  1
```

5.7.2 fun(complex<double>) or fun(complex<float>)

There is the list of the mathematical functions which takes as input value a `mThrust` complex vector of type double or float and returns a `mThrust` complex or real vector with same size an same type.

`abs` : Returns the real magnitude (also known as absolute value) of a complex.

`acos` : Returns the complex arc cosine of a complex number. The range of the real part of the result is $[0, \pi]$ and the range of the imaginary part is $[-\infty, +\infty]$.

`acosh` : Returns the complex inverse hyperbolic cosine of a complex number. The range of the real part of the result is $[0, +\infty]$ and the range of the imaginary part is $[-\pi, \pi]$.

`arg` : Returns the real phase angle (also known as argument) in radians of a complex.

`asin` : Returns the complex arc sine of a complex number. The range of the real part of the result is $[-\pi/2, \pi/2]$ and the range of the imaginary part is $[-\infty, +\infty]$.

`asinh` : Returns the complex inverse hyperbolic sine of a complex number. The range of the real part of the result is $[-\infty, +\infty]$ and the range of the imaginary part is $[-\pi/2, \pi/2]$.

`atan` : Returns the complex arc tangent of a complex number. The range of the real part of the result is $[-\pi/2, \pi/2]$ and the range of the imaginary part is $[-\infty, +\infty]$.

`atanh` : Returns the complex inverse hyperbolic tangent of a complex number. The range of the real part of the result is $[-\infty, +\infty]$ and the range of the imaginary part is $[-\pi/2, \pi/2]$.

`conj` : Returns the complex conjugate of a complex.

`cos` : Returns the complex cosine of a complex number.

`cosh` : Returns the complex hyperbolic cosine of a complex number.

`exp` : Returns the complex exponential of a complex number.

`log` : Returns the complex natural logarithm of a complex number.

`log10` : Returns the complex base 10 logarithm of a complex number.

`norm` : Returns the real square of the magnitude of a complex.

`sin` : Returns the complex sine of a complex number.

`sinh` : Returns the complex hyperbolic sine of a complex number.

`sqrt` : Returns the complex square root of a complex number.

`tan` : Returns the complex tangent of a complex number.

`tanh` : Returns the complex hyperbolic tangent of a complex number.

As sample, we compute the absolute value of an `mThrust` complex vector

Listing 5.24: Absolute value function on a `mThrust` complex vector

```
n=10;
gx=Thrust.ones(n)+i*Thrust.sequence(n)
gy=abs(gx)
```

```
gx =

 thrust::device_vector<complex<double>>[10]
 (1,0)  (1,1)  (1,2)  (1,3)  (1,4)
(1,5)  (1,6)  (1,7)  (1,8)  (1,9)

gy =

 thrust::device_vector<double>[10]
 1  1.41421  2.23607  3.16228  4.12311
```

```
 5.09902   6.08276   7.07107   8.06226   9.05539
```

5.8 Cumulative mathematical functions

There is the list of the mathematical functions which takes as input value a `mThrust` vector of type double, float, int or unsigned and returns a `mThrust` vector with same size an same type.

`cumsum :` Computes the cumulative sum of the input argument.

`cumprod :` Computes the cumulative prod of the input argument.

`cummin :` Computes the cumulative minimum of the input argument.

`cummax :` Computes the cumulative maximum of the input argument.

As sample, we compute the cumulative sum of a `mThrust` vector

Listing 5.25: Cumulative sum function on `mThrust` vector

```
gx=Thrust.zeros(100,'type','int')
gx.sequence(1,100,'init',1,'step',2);gx
gy=cumsum(gx)
```

```
gx =

 thrust::device_vector<int>[100]
 0  0  0  0  0
 . . .
 0  0  0  0  0

gx =

 thrust::device_vector<int>[100]
 1  3  5  7  9
 . . .
 191  193  195  197  199

gy =

 thrust::device_vector<int>[100]
 1  4  9  16  25
 . . .
 9216  9409  9604  9801  10000
```

5.9 Sort function

Let gx be a `mThrust` vector

```
sort(gx)
```

Modify gx such that its elements are sorted into ascending order.

Listing 5.26: sort function on mThrust vector

```
gx=mThrust([1,5,2,4,8,1],'device')
sort(gx)
gx
```

```
gx =

 thrust::device_vector<double>[6]
 1  5  2  4  8
1

gx =

 thrust::device_vector<double>[6]
 1  1  2  4  5
8
```

```
gy=sort(gx)
```

Create gy mThrust vector with same container and type as gx which contains the sorted into ascending order of the mThrust vector gx. gx is unchanged.

Listing 5.27: sort function on mThrust vector

```
gx=mThrust([1,5,2,4,8,1],'device')
gy=sort(gx)
gx
```

```
gx =

 thrust::device_vector<double>[6]
 1  5  2  4  8
1

gy =

 thrust::device_vector<double>[6]
 1  1  2  4  5
8

gx =

 thrust::device_vector<double>[6]
```

```
  1   5   2   4   8
 1
```

```
[gy,gI]=sort(gx)
```

Create the `mThrust` vector gy with same container and type as gx and the `mThrust` vector gI of integer type with same container as gx. The vector gy contains the sorted into ascending order of the `mThrust` vector gx. gI is the index vector such that gy=gx(gI) gx is unchanged.

Listing 5.28: sort function on `mThrust` vector

```
gx=mThrust([1,5,2,4,8,1],'device')
[gy,gI]=sort(gx)
gx(gI)
```

```
gx =

 thrust::device_vector<double>[6]
 1   5   2   4   8
1

gy =

 thrust::device_vector<double>[6]
 1   1   2   4   5
8

gI =

 thrust::device_vector<int>[6]
 0   5   2   3   1
4

ans =

 thrust::device_vector<double>[6]
 1   1   2   4   5
8
```

In all previous sort command, we can sort in descending order by using Name,Value pair with 'mode' as Name and 'descend' as Value.

Listing 5.29: sort function on `mThrust` vector

```
gx=mThrust([1,5,2,4,8,1],'device')
[gy,gI]=sort(gx,'mode','descend')
gx(gI)
```

```
gx =

 thrust::device_vector<double>[6]
 1  5  2  4  8
1

gy =

 thrust::device_vector<double>[6]
 8  5  4  2  1
1

gI =

 thrust::device_vector<int>[6]
 4  1  3  2  0
5

ans =

 thrust::device_vector<double>[6]
 8  5  4  2  1
1
```

5.10 Find function

Find indices and values of nonzero elements of a `mThrust` vector. Let gx be a `mThrust` vector

```
gI=find(gx)
```

returns a `mThrust` vector containing the indices of each nonzero element in gx.

Listing 5.30: **find** function on `mThrust` vector

```
gx=mThrust([1,5,0,2,4,8,0,1],'device')
gI=find(gx)
```

```
gx =

 thrust::device_vector<double>[8]
 1  5  0  2  4
8  0  1

gI =

 thrust::device_vector<int>[6]
 0  1  3  4  5
```

```
7
```

5.11 unique function

Return the unique values of a `mThrust` vector. These values are in sorted order. Let gx be a `mThrust` vector

```
gy=unique(gx)
```

returns a `mThrust` vector containing the unique values of gx in sorted order.

Listing 5.31: unique function on `mThrust` vector

```
gx=mThrust([1,5,0,2,4,8,0,1],'device')
gy=unique(gx)
```

```
gx =

 thrust::device_vector<double>[8]
 1  5  0  2  4
8  0  1

gy =

 thrust::device_vector<double>[6]
 0  1  2  4  5
8
```

6 Using CUDA kernels

6.1 Newton Fractals

A generalization of Newton's iteration is

$$z_{n+1} = z_n - a\frac{f(z_n)}{f'(z_n)} \tag{1}$$

and a non optimized Matlab function based on this formula is given in Listing 6.1. A more complete description can be found in [].

```
function varargout=GeneralizedNewton(f,fp,x0,varargin);
% GeneralizedNewton solves a nonlinear system of equations
%   x=GeneralizedNewton(f,fp,x0); solves the nonlinear
%   system of equations f(x)=0 using Newtons methods with
%   x real or complex, starting with the initial guess x0
%   up to a tolerance tol, doing at most maxit iterations.
%   The analytical derivative of f is given by the
%   parameter fp. The initial guess x0 is a scalar.
%   Returns last point of the generalized Newton sequence
```

```
%      x(n+1)=x(n)-a*f(x(n))/fp(x(n))
%    (a is one by default)
%    If this sequence converge then the return value x is an
%    approximation of a root of function f.
%
% x=GeneralizedNewton(f,fp,x0,Name,Value);
%    uses Name,Value pairs to specify the tolerence with 'tol'
%    as Name (default Value is 1e-8), the maximum number of
%    iterations with 'itermax' as Name (default Value is 200)
%    and the complex parameter a with 'a' as Name
%    corresponding to the generalization of Newton's iteration
%    (default 1).
%
% [x,S]=GeneralizedNewton(...);
%    returns a struct S which contains some informations ...
%
  p = inputParser;
  p.addParamValue('tol', 1e-8, @isscalar );
  p.addParamValue('itermax', 200, @isscalar );
  p.addParamValue('a', 1, @isscalar );
  p.parse(varargin{:});
  a=p.Results.a;
  tol=p.Results.tol;
  itermax=p.Results.itermax;
  k=1;err=tol+1;alpha=[];
  x(k)=x0;
  bconv=false; % logical true if convergence
  while ( (~bconv) & (k<=itermax) )
    dx=a*f(x(k))/fp(x(k));
    x(k+1)=x(k)-dx;
    bconv = (abs(dx)<=tol);
    k=k+1;
  end
  if nargout>=1, varargout{1}=x(end);end
  if nargout==2
    varargout{2}=struct('iter',k,'x',x,'converge',bconv, ...
                        'itermax',itermax,'tol',tol,'a',a);
  end
end
```

Listing 6.1: GeneralizedNewton Matlab function

For each $z_0 \in \mathbb{C}$, we can obtain from previous function the number of iterations really made and, if the sequence converge, an approximation of a root of the function f. The Newton fractal can be coloured by number of iterations required or by root reached (see Figure 1).

We want to represent the Newton fractals on the rectangular domain of the complex plane $\mathcal{P}$ defined with $x \in [a, b]$ and $y \in [c, d]$ by $z = x + \imath y$.

We can construct a complex grid of this domain with Matlab by using :

```
x=linspace(a,b,N(1));
y=linspace(c,d,N(2));
[xGrid,yGrid] = meshgrid( x, y );
z0 = xGrid + 1i*yGrid;
```

So we have `z0(r,s)=x(s)+1i*y(r)`, $\forall s \in [\![1, N(1)]\!]$, $r \in [\![1, N(2)]\!]$. On each point `z0(r,s)` of the complex grid `z0`, we apply `GeneralizedNewton` function and we store the number of iteration in `count(r,s)` and the last iterate in `z(r,s)`. The Matlab code which computes the `N(2)`-by-`N(1)` arrays `count` and `z` follows :

```
  for s=1:N(1)
    for r=1:N(2)
```

```
        [zc,S]=GeneralizedNewton(f,fp,z0(r,s),'tol',R.tol, ...
            'itermax',R.itermax,'a',R.a);
        z(r,s)=zc;
        count(r,s)=S.iter;
    end
end
```

The complete function which computes the two arrays count and z is given in Listing 6.6, page 43.

To represent the Newton fractal coloured by number of iterations required we used the following code :

```
image(x,y,count,'CDataMapping','scaled')
colorbar
title('Number of iterations')
```

We give on Figure 1 (left) the graphical result.

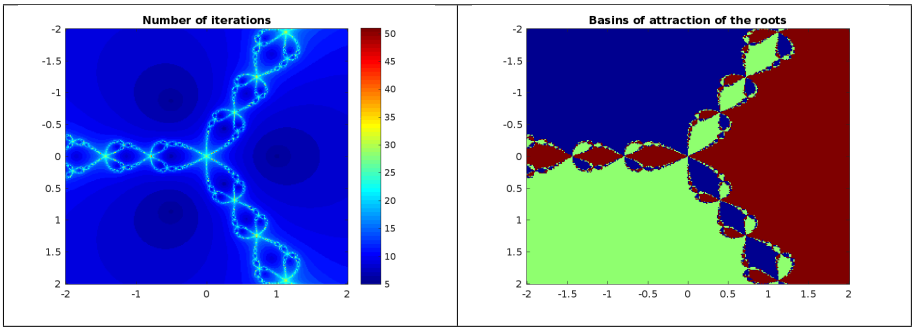


Figure 1: Newton fractal for $f(z) = z^3 - 1$ coloured by number of iterations required (left) and coloured by root reached (right)

To obtain the basins of attractions of the roots (i.e. the Newton fractal coloured by root reached) we compute the two arrays Alpha and Num. The array Alpha contains all the *unique* values of the array z except values where the sequence with initial value z0(r,s) did not converge. The array Num is N(2)-by-N(1) and Num(r,s) is 0 if sequence with initial value z0(r,s) did not converge otherwise Num(r,s) is the index of array Alpha where the sequence converge. The Matlab function findZeros compute these two arrays and it's given by:

```
function [Alpha,Num]=findZeros(z,count,itermax,tol)
% Alpha contains all the unique values with a
% of tolerance tol of the complex array z
% except those corresponding to z(r,s) where
% count(r,s)==itermax. In this case
% Num(r,s)=0, otherwize Num(r,s)=idx
% with Alpha(idx) == z(r,s) (with tolerance tol).
count=count(:);
zz=z(:);
Num=zeros(length(zz),1);
I=find((count~=itermax) & (~isnan(zz))); % index of points of ...
    convergence
[~, ia, ic] = unique(round(zz(I)/(10*tol)));% ...
    (100*tol*complex(1,1)));
Alpha=zz(I(ia));
Num(I)=ic;
Num=reshape(Num,size(z));
```

To represent the Newton fractal coloured by root reached given in Figure 1 (right) we used the following code :

```
image(x,y,Num,'CDataMapping','scaled')
title('Basins of attraction of the roots.')
```

On my laptop[1] , the computation of the arrays count and z on a 300-by-300 grid is done in 15 seconds. It's a very slow code. In a first step we will write a more efficient code in Matlab by using vectorization techniques. In a second step, by using mThrust toolbox with CUDA kernel we will write a mixed Matlab/CUDA code.

6.1.1 Vectorized Matlab code

To improve performance, we vectorize the two for loops and write a new vectorized Matlab function called GeneralizedNewtonFractalVec which compute all the Newton sequences and the number of iterations for initial values given by a 2D complex grid . This function is given in Listing 6.7, page 45.

Now, the computation time of the arrays count and z on a 300-by-300 grid is 0.14 seconds. We compare in Table 1 the performance of these two functions for differents grid sizes.

Grid	Matlab not vectorized	Matlab vectorized
100×100	1.865(s)	0.040(s) $\times$ 46.74
200×200	7.236(s)	0.064(s) $\times$ 113.69
300×300	16.299(s)	0.175(s) $\times$ 92.88
400×400	30.344(s)	0.292(s) $\times$ 103.78

Table 1: Comparison of GeneralizedNewtonFractal function (not vectorized) and GeneralizedNewtonFractalVec function (vectorized)

For larger grids, we only use the vectorized function and as sample for a 4000-by-4000 grid the computation time is $36(s)$.

6.1.2 `mThrust` toolbox using CUDA kernel

There are four steps for using CUDA kernel with `mThrust`:

1. writing CUDA kernel,
2. writing mex file interface using mThrust API,
3. compiling the mex file,
4. using the compiled mex file

[1] `Intel(R) Core(TM) i7-4800MQ CPU @ 2.70GHz` with 16Go RAM and the `Quadro K1100M` NVidia GPU

The most simple CUDA kernel use an implementation of the Newton fixed point for an initial data $z0$ as a device function. For our purpose, this function named `GeneralizedNewtonFractal` compute the *last* iterate `z` of the generalized Newton sequence and the number of iterations named `count`. This function is given in Listing 6.2.

Listing 6.2: `GeneralizedNewtonFractal` CUDA device function, a generalized Newton sequence implementation

```
__device__ void GeneralizedNewtonFractal(
                          const thrust::complex<double> &a,
                          const thrust::complex<double> &z0,
                          const unsigned int maxIters,
                          const double tol,
                          thrust::complex<double> &z,
                          unsigned int &count
                              ) {
  bool bconv=false;
  thrust::complex<double> dz;
  count=0;
  z=z0;
  double tol2=tol*tol;
  while ( ( count <= maxIters ) && (!bconv) ){
      count++;
      dz=a*f(z)/df(z);
      z=z-dz;
      bconv= dz.real()*dz.real()+dz.imag()*dz.imag() <= tol2;
      //more efficient than bconv= (abs(dz) <= tol);
  }
}
```

So, with this function a simple kernel acting on a 1D grid with `gridDim.x` blocks where each block contains `blockDim.x` threads is given in Listing 6.3.

Listing 6.3: `GeneralizedNewtonFractal_kernel` CUDA kernel function, compute ...

```
__global__ void GeneralizedNewtonFractal_kernel(
                       const thrust::complex<double> a,
                       const thrust::complex<double> *z0,
                       const unsigned int numel,
                       const unsigned int maxIters,
                       const double tol,
                       thrust::complex<double> *z,
                       unsigned int *count)
{
  const unsigned int      tid = blockDim.x * blockIdx.x + ↩
      ↪ threadIdx.x;
  const unsigned int THREAD_N = blockDim.x * gridDim.x;
  unsigned int globalThreadIdx=tid;
  while (globalThreadIdx < numel){
    GeneralizedNewtonFractal(a, z0[globalThreadIdx], ↩
        ↪ maxIters,tol,z[globalThreadIdx],count[globalThreadIdx]);
    globalThreadIdx+=THREAD_N;
  }
}
```

Classicaly, the Matlab mex interface with a C/C++ code uses the gateway routine, `mexFunction`, as the entry point to the function. The prototype of this function is :

```
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[]);
```

where arguments are

`nlhs` Number of output (left-side) arguments, or the size of the plhs array.

`plhs` Array of output arguments.

`nrhs` Number of input (right-side) arguments, or the size of the prhs array.

`prhs` Array of input arguments.

We also use the C/C++ header file, `mex.h`, containing the MATLAB API function declarations.

For our purpose, we want the function `mexFunction` to *work* on GPU device by using only arrays defined on device. So the input array data `z0` (i.e. the complex grid) must be transfered under Matlab to the GPU device by using the `mThrust` toolbox and more precisely the command :

```
gz0=mThrust(z0(:),'device');
```

As `mThrust` toolbox is an interface to Thrust vector, we have transformed the 2D complex grid z0 on an 1D-array with `z0(:)` matlab command. The other input datas are `itermax`, `tol` and the complex number `a`.

The functions f and its derivative f' are directly written as CUDA device functions. For example, if $f : z \mapsto z^3 - 1$ then we write

```
#include <thrust/complex.h>

__device__ thrust::complex<double> f(thrust::complex<double> z){
  return  z*z*z-1.0;
}
__device__ thrust::complex<double> df(thrust::complex<double> z){
  return  3.0*z*z;
}
```

By using `mThrust` toolbox one can create the *output* device arrays `gz` and `gcount`

```
gz=mThrust(numel(z0),'device','double',true);
gcount=mThrust(numel(z0),'device','unsigned',false);
```

Then these two arrays will be passed as input arguments to the mex function `mexFunction`.

We choose as order of input arguments for `mexFunction`:

`prhs[0]` , the `a` value (complex double)

`prhs[1]` , the `gz0` `mThrust` vecteur on device (complex double)

`prhs[2]` , the `itermax` value (unsigned int)

`prhs[3]` , the `tol` value (double)

`prhs[4]` , the `gz` `mThrust` vector on device (complex double)

`prhs[5]` , the `gcount` `mThrust` vector on device (unsigned int)

To simplify the writing of the code we define the two types `Cplx` and `VecCplx` as follows

```
typedef thrust::complex<double>       Cplx;
typedef thrust::device_vector<Cplx>   VecCplx;
```

Then by using the `mThrust` API, we can retrieved the Thrust vector associed with an `mThrust` object create under Matlab. For example, the Thrust device vector associated with the `gz0` `mThrust` vecteur is obtained from the `get_object` function and from the `handle` property of `gz0` as follows

```
VecCplx &z0 = get_object<VecCplx>(mxGetProperty(prhs[1], 0, "handle"));
```

Threreafter to interact with the `GeneralizedNewtonFractal_kernel` CUDA kernel function given in Listing 6.3 we must recover the pointer to the device array contained in the Thrust device vector. For that we use the function `thrust::raw_pointer_cast` of the Thrust API

```
Cplx *pz0=(Cplx *)thrust::raw_pointer_cast(&z0[0]);
```

We give in Listing 6.4 the `mexFunction` code

Listing 6.4: mex function `mexFunction` using a CUDA kernel

```
void mexFunction( int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[] )
{
  MEXPRINTF("starting");
  if (nlhs !=0 || nrhs != 6)
    mexErrMsgTxt("Bad input.");
  typedef thrust::complex<double>               Cplx;
  typedef thrust::device_vector<Cplx>           VecCplx;
  typedef thrust::device_vector<unsigned int>   VecInt;

  double *pa_r = (double *) mxGetPr(prhs[0]);
  double *pa_i = (double *) mxGetPi(prhs[0]);
  Cplx a=Cplx(*pa_r,*pa_i);
  VecCplx &z0    = get_object< VecCplx >(mxGetProperty(prhs[1],0,"handle"));
  unsigned int *pmaxIter = (unsigned int *) mxGetPr(prhs[2]);
  double *ptol = (double *) mxGetPr(prhs[3]);
  VecCplx &z     = get_object< VecCplx >(mxGetProperty(prhs[4],0,"handle"));
  VecInt &count = get_object< VecInt >(mxGetProperty(prhs[5],0,"handle"));

  Cplx *pz0=(Cplx *)thrust::raw_pointer_cast(&z0[0]);
  Cplx *pz =(Cplx *)thrust::raw_pointer_cast(&z[0]);
  unsigned int *pcount=(unsigned int *)thrust::raw_pointer_cast(&count[0]);

  int blockSize; // The launch configurator returned block size
  int gridSize;  // The actual grid size needed, based on input size
  getKernelSizes(gridSize,blockSize, z0.size());

  GeneralizedNewtonFractal_kernel<<<gridSize,blockSize>>>(a, pz0, z0.size() ,*pmaxIter, *ptol, pz, pcount);
  MEXPRINTF("ending");
}
```

The complete code is given in `mThrust` toolbox by file `mex_GeneralizedNewtonFractal_simple.cu` in directory `+Thrust/+samples/+NewtonFractal`. To compile this file and obtain the binary mex file usable directly in Matlab, one can use the `mThrust` function `Thrust.compile.mex` under matlab as follows

```
P='+Thrust/+samples/+NewtonFractal/';
mexFile=[P,'mex_GeneralizedNewtonFractal_simple.cu'];
Thrust.compile.mex(mexFile,'samepath',true)
```

Finally, we give in Listing 6.5 a Matlab interface function to this mex file.

Listing 6.5: Matlab function interface to the `mexFunction`

```
function varargout=GeneralizedNewtonFractalGPUsimple(z0,varargin);
% GeneralizedNewtonFractalGPU solves the nonlinear equation z^3-1=0
%   Z=GeneralizedNewtonFractalGPUsimple(z0)
%   solves the nonlinear equations z^3-1=0 using
%   Newtons methods with z real or complex, starting with
%   the initial guess z0 up to a default tolerance 1e-8,
%   doing at most 200 iterations.
  import Thrust.samples.NewtonFractal.*;
  p = inputParser;
  p.addParamValue('tol', 1e-8, @isscalar );
  p.addParamValue('itermax', 200, @isscalar );
  p.addParamValue('verbose', 0, @(x) ismember(x,0:1) );
  p.addParamValue('a', 1, @isscalar );
  p.parse(varargin{:});
  R=p.Results;

  if R.verbose,fprintf('1) Transfert/init datas on GPU\n');end
  tstart=tic();
  gz0=mThrust(z0(:),'device'); % Only vectors are supported on ...
      mThrust
  gz=mThrust(numel(z0),'device','double',true);
  gcount=mThrust(numel(z0),'device','unsigned',false);
  cuda.DeviceSynchronize();
  T(1)=toc(tstart);
  if R.verbose,fprintf('2) Computation on GPU\n');end
  tstart=tic();
  mex_GeneralizedNewtonFractal_simple(complex(R.a), gz0, ...
      uint32(R.itermax), R.tol, gz, gcount);
  cuda.DeviceSynchronize();
  T(2)=toc(tstart);
  if R.verbose,fprintf('3) Transfert datas on Matlab\n');end
  tstart = tic();
  z=reshape(gather(gz),size(z0));
  count=reshape(gather(gcount),size(z0));
  cuda.DeviceSynchronize();
  T(3) = toc( tstart ); % transfert time to CPU
  if R.verbose
    fprintf('   to GPU : %.3f(s), computation : %.3f(s),',T(1:2))
    fprintf(' to CPU: %.3f(s)\n',T(3))
    fprintf('  -> total: %.3f(s)\n',sum(T));
  end
  if nargout>=1,varargout{1}=z;end
  if nargout>=2,varargout{2}=count;end
  if nargout==3,varargout{3}=T;end
end
```

In table 2, we compare the performance of the Matlab vectorized function and

6.1.3 Pure Matlab codes

Listing 6.6: GeneralizedNewtonFractal function, not vectorized Matlab function

```
function varargout=GeneralizedNewtonFractal(f,fp,z0,varargin);
% NEWTON solves a nonlinear system of equations
%   z=Newton(f,fp,z0,tol,maxiter,fp); solves the nonlinear
%   system of equations f(z)=0 using Newtons methods with
%   z real or complex, starting with the initial guess x0
```

Grid	Matlab Vectorized	CUDA/Thrust
1000×1000	1.512(s)	0.082(s) $\times$ 18.46
1500×1500	4.665(s)	0.165(s) $\times$ 28.28
2000×2000	9.543(s)	0.274(s) $\times$ 34.88
2500×2500	16.245(s)	0.421(s) $\times$ 38.62
3000×3000	24.698(s)	0.571(s) $\times$ 43.26
3500×3500	27.797(s)	0.769(s) $\times$ 36.16
4000×4000	45.920(s)	0.990(s) $\times$ 46.40
4500×4500	54.430(s)	1.233(s) $\times$ 44.14
5000×5000	68.173(s)	1.538(s) $\times$ 44.33

Table 2: Comparison of GeneralizedNewtonFractalVec function and GeneralizedNewtonFractalGPUsimple function

```
%   up to a tolerance tol, doing at most maxit iterations.
%   The analytical derivative of f is given by the
%   parameter fp. The initial guess z0 could be an array.
%   (vectorized Matlab version).
  import Thrust.samples.NewtonFractal.GeneralizedNewton
  p = inputParser;
  p.addParamValue('tol', 1e-8, @isscalar );
  p.addParamValue('itermax', 200, @isscalar );
  p.addParamValue('verbose', 0, @(x) ismember(x,0:1) );
  p.addParamValue('a', 1, @isscalar );
  p.parse(varargin{:});
  R=p.Results;
  tstart=tic();
  N=size(z0);
  z=zeros(N);
  count=zeros(N);
  if R.verbose>0, upd = textprogressbar(N(1));end
  for s=1:N(1)
    for r=1:N(2)
      [zc,S]=GeneralizedNewton(f,fp,z0(r,s),'tol',R.tol, ...
          'itermax',R.itermax,'a',R.a);
      z(r,s)=zc;
      count(r,s)=S.iter;
    end
   if R.verbose>0, upd(s);end
  end
  T=toc(tstart);
  if nargout>=1,varargout{1}=z;end
  if nargout>=2,varargout{2}=count;end
  if nargout==3,varargout{3}=T;end
end
```

Listing 6.7: GeneralizedNewtonFractalVec function, vectorized Matlab function

```
function varargout=GeneralizedNewtonFractalVec(f,fp,z0,varargin);
% GeneralizedNewtonFractalVec solves a nonlinear system of ...
    equations
%   Z=GeneralizedNewtonFractalVec(f,fp,z0)
%   solves the nonlinear system of equations f(z)=0 using
%   Newtons methods with z real or complex, starting with
%   the initial guess z0 up to a default tolerance 1e-8,
%   doing at most 200 iterations.
%   The analytical derivative of f is given by the
%   parameter fp. The initial guess z0 could be an array.
%   (vectorized Matlab version).
  p = inputParser;
  p.addParamValue('tol', 1e-8, @isscalar );
  p.addParamValue('itermax', 200, @isscalar );
  p.addParamValue('verbose', 0, @(x) ismember(x,0:1) );
  p.addParamValue('a', 1, @isscalar );
  p.parse(varargin{:});
  a=p.Results.a;
  verbose=p.Results.verbose;
  tol=p.Results.tol;
  itermax=p.Results.itermax;
  tstart=tic();
  k=1;err=tol+1;alpha=[];
  z=z0;
  count=zeros(size(z0));
  bconv=false; % logical true if convergence
  while ( (~bconv) & (k<=itermax) )
    dz=a*fp(z).\f(z);
    z=z-dz;
    notconv=(abs(dz)>tol);
    count = count + notconv;
    bconv = (max(notconv(:))==0);
    k=k+1;
    if verbose>0,fprintf('k=%d\n',k);end
  end
  T=toc(tstart);
  if nargout>=1,varargout{1}=z;end
  if nargout>=2,varargout{2}=count;end
  if nargout==3,varargout{3}=T;end
end
```