Developer Guide

Version 0.28, January 2012

Contents

Contents **2**

1 Source code compilation **4**
1.1 How to compile . . . 4
1.2 Configuring the compiler . . . 5
1.3 CUDA installation . . . 6

2 Creating a new GPU kernel **7**
2.1 Quick start . . . 8
2.2 Compilation . . . 9
2.3 Module manager and mex function initialization . . . 10
2.4 MEX function summary . . . 13
2.5 Testing . . . 14

3 Accessing GPUmat functions and variables from a MEX file **16**
3.1 The GPUtype class . . . 17
3.2 GPUmat internal functions . . . 18
3.3 MEX file example . . . 20

4 GPUmat GPU classes **26**
4.1 GPUsingle, GPUdouble constructor . . . 28
4.2 GPUsingle, GPUdouble properties . . . 30
4.3 GPUsingle, GPUdouble methods . . . 31

5 Numerics module **32**
5.1 Indexed references . . . 32
5.1.1 GPUmat *slice* and *assign* (or *mxSlice* and *mxAssign*) . . . 33
5.1.2 Matlab wrappers to GPUmat *slice* and *assign* functions . 35
5.1.3 subsref, subsasgn . . . 37
5.1.4 Performance analysis . . . 40
5.2 GPUfill . . . 41

CONTENTS

5.2.1 Implementation 43
5.2.2 Examples 44
5.3 REPMAT 45
5.3.1 Implementation 45
5.3.2 Testing 47

6 Examples module **48**
6.1 GPUtype 48
6.1.1 gputype_properties.cpp 48
6.1.2 gputype_create1.cpp 49
6.1.3 gputype_create2.cpp 50
6.1.4 gputype_clone 51
6.1.5 Testing 51

CONTENTS

Chapter 1
Source code compilation

The structure of the *GPUmat* source code is the following:

```
|
+-GPUmat
+-GPUmatModules
+-matCUDA
+-util
```

The above structure shows the main GPUmat parts:

- *GPUmat* core functions, folder *GPUmat*.
- *GPUmat User Modules*, folder *GPUmatModules*.
- *CUDA wrappers*, folder *matCUDA*.

1.1 How to compile

Compilation step-by-step:

- STEP0. Configure a compiler under *Matlab*. See Section 1.2 below.
- STEP1. Add the *util* folder to the *Matlab* path.
- STEP2. Make sure you have installed the *CUDA* toolkit. See Section 1.3 below.
- STEP3. Run the script *compile* in the *GPUmat* root folder. The script creates a folder `'release/<arch>'` and a compressed package in the *GPUmat* root directory.

1.2 Configuring the compiler

The compilation is done from Matlab. A valid compiler should be configured in Matlab using the `mex -setup` command. Under Windows the compilation scripts are configured to use *Microsoft Visual C++* (the *Express* version can be downloaded for free from *Microsoft*). Run the command `mex -setup` from *Matlab*. The following or similar output should be displayed on the command shell:

```
>> mex -setup
Please choose your compiler for building external interface
(MEX) files:

Would you like mex to locate installed compilers [y]/n? y

Select a compiler:
[1] Microsoft Visual C++ 2008 Express in C:\...

[0] None

Compiler: 1

Please verify your choices:

Compiler: Microsoft Visual C++ 2008 Express
Location: C:\Program Files (x86)\Microsoft Visual Studio 9.0

Are these correct [y]/n? y
```

The above example shows the output on Windows with *Microsoft Visual C++ 2008 Express* installed. More information about *Matlab* supported compilers can be found on *Matlab* web site *Supported Compilers*. For example, under Windows 64bit the *Microsoft Visual Express 2008* doesn't include the 64bit compilers, therefore (as indicated on the *Matlab* web site) the Windows SDK 6.1 must be installed in order to compile 64bit binaries. Check if the following commands can be executed from command line:

- Windows, assuming *Microsoft Visual C++* installed. The commands `cl` and `lib` should run properly from command shell. In *Matlab*, after adding the *util* folder to the *Matlab* path, execute `locateCL` and check the output.

- Linux, assuming *GNU Compiler* installed. The commands `cpp --help` and `ar` should run properly from command shell.

1.3 CUDA installation

NVIDIA CUDA Toolkit can be downloaded from the *NVIDIA* web site. The `CUDA_PATH` system variable should point to the folder where the *CUDA* toolkit is installed. If the *CUDA* toolkit binaries are installed in `/CUDA/v4.0/bin`, `CUDA_PATH` should be set to `/CUDA/v4.0`. Execute `nvidiasettings` from *Matlab* to check if *CUDA* is configured properly:

```
nvidiasettings
```

The output of the above command should be similar to the following:

```
>> nvidiasettings

ans =

    sdkpath: ''
       path: 'C:\CUDA\v4.0\'
    libpath: 'C:\CUDA\v4.0\lib\x64'
    incpath: 'C:\CUDA\v4.0\include'
       arch: {'10'  '11'  '12'  '13'  '20'  '21'  '22'  '23'  '30'}
```

Chapter 2
Creating a new GPU kernel

The user can add new features to *GPUmat* either by directly modifying the core source code or by adding or modyfing a *GPUmat User Module*. The latter is the easiest way because can be done by using the *GPUmat User Modules* project, originally developed on Sourceforge (`http://sourceforge.net/projects/gpumatmodules/`) as an external *GPUmat* package and available, starting from *GPUmat* version 0.28, in folder `GPUmatModules`. A *GPUmat User Module* can be distribuited separately from *GPUmat* and installed by copying it to the *GPUmat* `modules` folder. At startup, *GPUmat* searches the `modules` folder for a script called *moduleinit.m*, which takes care of starting up the installed module.

To implement *GPUmat User Module* the user has to know few concepts about *GPUmat*

- *GPUmat* implements an internal GPU type, which is accessible from Matlab using the *GPUsingle* or *GPUdouble* classes (or any other type that will be implemented in the future) and from a mex file using the *GPUtype* class. These classes are explained in Sections *GPUmat GPU classes* and *Accessing GPUmat functions and variables from a MEX file*. Memory management is performed on these classes using a *Garbage Collector* and the user doesn't have to worry about cleaning up unused GPU memory.

- *GPUmat* functions (such as FFT, numerical functions) are available from a mex file. The user can simply access and combine them to implement more complicated functions.

- User can create a new GPU kernel, load it into *GPUmat* and execute a HOST function to access the kernel. This point is explained in Section *Creating a new GPU kernel*.

This section shows how to create a new GPU kernel and how to load it in *GPUmat*. It requires a good knowledge of CUDA, GPU programming and

MEX compilation in Matlab. The example presented in the next sections can be found in *GPUmatModules/src/Examples/numerics*. To implement a new module the user has to create the following files:

- *moduleinit.m*: this is a Matlab script that loads the module.
- GPU kernels. GPU kernels are defined in a *.cu* file and compiled as *.cubin* module, which is loaded using *GPUmat* functions.
- HOST driver(s). A driver is a mex file that executes the GPU kernel.

To run a new module the user has to do the following:

- Load the module using *moduleinit.m*
- Execute the driver function.

The goal of this short tutorial is to implement the driver functions *myplus* and *mytimes* that perform the element-wise sum and multiplication of GPU variables. Any user defined function requires a GPU kernel, stored in a CUDA module that is loaded using the *GPUmat* module manager *GPUuserModuleLoad*, and a driver function (or host function) that calls the GPU kernel from Matlab. Implemented modules must define a new function called *moduleinit.m*, which initializes the module (loading for example the required *.cubin*). Find the *moduleinit.m* script in the example folder. This tutorial includes the following source code:

- *myplus.cpp*, *mytimes.cpp*: the driver functions (or host functions) that call the GPU kernel.
- *numerics.cu*: the CUDA module that contains the GPU kernels. It is compiled into different *.cubin* files.

The compilation of the above files is presented in Section *Compilation* and doesn't require the *NVMEX* script provided by *NVIDIA*, which is usually used to compile CUDA code from Matlab.

2.1 Quick start

Before reading the next sections, you can run the pre-compiled code available on *GPUmat* installation, folder `modules/Examples/numerics` by executing from Matlab:

```
moduleinit
runme
```

The output of the above command should be:

```
** Loading ->numericsXX.cubin
* Start Test
* Test finished
```

2.2 Compilation

The compilation is done from Matlab, using scripts that are provided in the *util* folder. This folder must be added to the Matlab path. A valid compiler should be configured in Matlab using the *mex -setup* command as explained in Section *Source Code Compilation*. The file *nvidiasettings.m*, located in the *util* folder, is used by the compilation scripts. It contains the following line:

```
cuda.path     = getenv('CUDA_PATH');
```

The system variable `CUDA_PATH` should point to the folder where *CUDA* is installed. Modify the value of `CUDA_PATH` according to your system and run the following command from Matlab:

```
nvidiasettings
```

The output of the above command should be:

```
>> nvidiasettings
>> nvidiasettings

ans =

    sdkpath: ''
       path: 'C:\CUDA\v4.0\'
    libpath: 'C:\CUDA\v4.0\lib\x64'
    incpath: 'C:\CUDA\v4.0\include'
       arch: {'10'  '11'  '12'  '13'  '20'  '21'  '22'  '23'  '30'}
```

The above output shows that the variable *path* is pointing to the right place. Use *make cpp* to compile *.cpp* files or *make cuda* to compile the CUDA *.cubin* file, as follows:

```
make cpp
make cuda
```

If you get an error running *make cuda*, it is possible that *nvcc* is not able to find the `C++` compiler on your system. In this case the location of the `C++` compiler should be added to the system path (not the Matlab path). At the end of the compilation you should have the following files in the example directory (mex extension depends on system. On Windows 32bit is mexw32):

```
myplus.mexw32
mytimes.nexw32
numerics10.cubin
numerics11.cubin
numerics12.cubin
numerics13.cubin
...
```

2.3 Module manager and mex function initialization

The module manager is implemented in *GPUmat* function *GPUuserModuleLoad*. This function is used as follows:

```
GPUuserModuleLoad(module_name,cubin_file);
```

The above command loads the cubin module `cubin_file` and assigns to it the name `module_name`. Loaded modules can be displayed using the command *GPUuserModulesInfo*. The procedure to access the module from a user defined function is the following:

- The module should be loaded from Matlab using the function *GPUuserModuleLoad*. A function *moduleinit.m* should be provided with the module and executed before accessing the module. If the module is not loaded, the mex function will not be able to access it.
- The mex function should initialize some variables and load the module handler only the first time it is called from Matlab, as explained below.

An example of a *moduleinit.m* function is the following:

```
function moduleinit
disp('- Loading module EXAMPLES_NUMERICS');
ver = 0.21;
gver = GPUmatVersion;
```

```
if (str2num(gver.version)<ver)
  warning(['MODULE ...
  return;
end

[status,major,minor] = cudaGetDeviceMajorMinor(0);
cubin = ['numerics' num2str(major) num2str(minor) '.cubin'];
disp(['** Loading ->' cubin ]);
GPUuserModuleLoad('examples_numerics',['.' filesep cubin])
end
```

GPUmat version is checked at the beginning of the script. The loaded module depends on the CUDA capability of the GPU, which is retrieved using the *cudaGetDeviceMajorMinor* function.

The code to initialize the mex function is the following (from *myplus.cpp* or *mytimes.cpp*):

```
static CUfunction drvfunf; // float
static CUfunction drvfunc; // complex
static CUfunction drvfund; // double
static CUfunction drvfuncd;//double complex

static int init = 0;

static GPUmat *gm;

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

  CUresult cudastatus = CUDA_SUCCESS;

  if (nrhs != 3)
    mexErrMsgTxt("Wrong number of arguments");

  if (init == 0) {
    // Initialize function
    mexLock();

    // load GPUmat
    gm = gmGetGPUmat();
    // load module
```

```
    CUmodule *drvmod = gmGetModule("examples_numerics");

    // load float GPU function
    CUresult status =
             cuModuleGetFunction(&drvfunf, *drvmod, "PLUSF");
    if (CUDA_SUCCESS != status) {
      mexErrMsgTxt("Unable to load user function.");
    }

    // load complex GPU function
    status = cuModuleGetFunction(&drvfunc, *drvmod, "PLUSC");
    if (CUDA_SUCCESS != status) {
      mexErrMsgTxt("Unable to load user function.");
    }

    // load double GPU function
    status = cuModuleGetFunction(&drvfund, *drvmod, "PLUSD");
    if (CUDA_SUCCESS != status) {
      mexErrMsgTxt("Unable to load user function.");
    }

    // load complex GPU function
    status = cuModuleGetFunction(&drvfuncd, *drvmod, "PLUSCD");
    if (CUDA_SUCCESS != status) {
      mexErrMsgTxt("Unable to load user function.");
    }

    init = 1;
  }
```

The first time the function *myplus* (or *mytimes*) is called from Matlab, the variable *init* is equal to 0, and the mex file is locked on Matlab workspace by using *mexLock()*. Locking the mex function is not really mandatory. During the initialization, the `init` variable is set to 1, so that the second time we call the function *myplus* (or *mytimes*) the initialization procedure is skipped. The value of the different static variables do not change during the Matlab session. The function *GPUgetUserModule* returns the *CUmodule* handle as a floating point variable. In the C++ code we have to cast this value to a *CUmodule* type, as follows:

```
// load module
CUmodule *drvmod = gmGetModule("examples_numerics");
```

The variable *drvmod* is used to retrieve the handle to the different functions (GPU kernels) available in the *.cubin* module, as follows:

```
// load float GPU function
CUresult status =
    cuModuleGetFunction(&drvfunf, *drvmod, "PLUSF");
if (CUDA_SUCCESS != status) {
  mexErrMsgTxt("Unable to load user function.");
}
```

In the above code, the *PLUSF* kernel handler is loaded into the variable *drvfunf*. Please note that the variable *drvfunf* is static and defined before the *mexFunction*. By doing so, the scope of this variable is outside the *mexFunction* and *drvfunf* will be persistent during the entire Matlab session. The function *myplus* works with single/double precision real/complex types. It means that there are different functions defined in the *.cubin* modules, one for each type. We load these functions during the initialization phase into the variables *drvfunf*, *drvfunc*, *drvfund* and *drvfuncd* for real/single, complex/single, real/double and complex/double respectively.

2.4 MEX function summary

We perform the following operations in the mex function:

- Initialization: explained in the previous section
- Read input/output variables.
- Execute the GPU kernel

Variables are read with the following code:

```
//IN1 is the input GPU array
GPUtype IN1 = gm->gputype.getGPUtype(prhs[0]);

//IN2 is the input GPU array
GPUtype IN2 = gm->gputype.getGPUtype(prhs[1]);

//OUT is the output GPU array (result)
GPUtype OUT = gm->gputype.getGPUtype(prhs[2]);
```

The *GPUtype* type is used to access *GPUsingle* and *GPUdouble* variables from a mex function. Please check Section *Accessing GPUmat functions and variables from a MEX file* for details. The following code assigns to the variable *drvfun* the right function handler depending on operands type:

```
// The GPU kernel depends on the type of input/output
CUfunction drvfun;
if (tin1 == gpuFLOAT) {
  drvfun = drvfunf;
} else if (tin1 == gpuCFLOAT) {
  drvfun = drvfunc;
} else if (tin1 == gpuDOUBLE) {
  drvfun = drvfund;
} else if (tin1 == gpuCDOUBLE) {
  drvfun = drvfuncd;
}
```

The variable *drvfun* is passed to the function *hostGPUDRV*, which calls the GPU kernel, as follows:

```
hostdrv_pars_t gpuprhs[3];
int gpunrhs = 3;
gpuprhs[0] = hostdrv_pars(&d_IN1,sizeof(d_IN1));
gpuprhs[1] = hostdrv_pars(&d_IN2,sizeof(d_IN2));
gpuprhs[2] = hostdrv_pars(&d_OUT,sizeof(d_OUT));

int N = nin1;

hostGPUDRV(drvfun, N, gpunrhs, gpuprhs);
```

The function *hostGPUDRV* can handle an arbitrary number of input arguments, passed using the vector *gpurhs*.

2.5 Testing

Several functions are provided to help testing the implemented functions. The *util* folder contains the following procedures:

- *GPUtestInit.m*: initializes a global configuration variable called *GPUtest*.
- *GPUtestLOG.m*: writes to the log file, which is initialized with procedure *GPUtestInit.m*.

- *compareCPUGPU.m*: use this function to compare CPU and GPU variables.

The files *testmyplus.m* and *testmytimes.m* located in the example folder, are typical examples of testing using the above functions. Please check these files for more details about testing.

Chapter 3
Accessing GPUmat functions and variables from a MEX file

A *GPUsingle* or a *GPUdouble* (defined in *GPUmat*) can be accessed from a mex file, by using functions provided in the file *GPUmat.hh*, located in the *include* folder. During the compilation phase you have to link also the file *GPUmat.cpp* included in the *common* folder (check available make files for compilation examples). In order to communicate with *GPUmat* from a mex file, it is necessary to create a *GPUmat* static variable, as follows:

```
static GPUmat *gm;
void mexFunction ...
...
// load GPUmat
gm = gmGetGPUmat();
...
```

The *gm* variable can be used to access *GPUmat* functions. Please check the *Developer Reference Manual* for more details about the structure *GPUmat* $*$ *gm*. A typical behavior of a user defined function is the following:

- Read input variables (*GPUsingle*, *GPUdouble*).
- Convert them into *GPUtype* objects.
- Create the output *GPUtype* result using provided functions, such as `gm->gputype.create`.
- Execute a GPU kernel on *GPUtype* objects.
- Return to Matlab a *GPUsingle* or *GPUdouble*, created from the *GPUtype* result using `gm->gputype.createMxArray`.

The *GPUsingle* or *GPUdouble* can be converted into a *GPUtype* by using the function `gm->gputype.getGPUtype`. Functions to create new *GPUtype* objects or modify existing are (file *GPUmat.hh*):

Create and modify GPUtype objects	
`gm->gputype.create`	Creates a GPUtype
`gm->gputype.createMx`	Creates a GPUtype
`gm->gputype.createMxArray`	Creates a GPUsingle or GPUdouble to be returned to Matlab
`gm->gputype.getGPUtype`	Creates a GPUtype from a Matlab GPUmat variable
`gm->gputype.slice`	Creates a slice from a GPUtype using specified Range
`gm->gputype.assign`	Assigns a GPUtype to another. A Range can be applied either to the left or right hand side
`gm->gputype.clone`	Clones a GPUtype. The new GPUtype points to a different GPU memory location
`gm->gputype.mxToGPUtype`	Creates a GPUtype from a Matlab array
`gm->gputype.colon`	Fills a GPUtype with specified values. Can be used to create an array of ones, zeros or different sequence of values
`gm->gputype.floatToDouble`	Converts a single precision GPUtype to double precision.
`gm->gputype.doubleToFloat`	Converts a double precision GPUtype to single precision.
`gm->gputype.realToComplex`	Converts a real GPUtype to complex.

3.1 The GPUtype class

The *GPUtype* class is defined in the file *GPUmat.hh*. This class is a container for the internal *GPUmat* type used for GPU variables. It is implemented as a smart pointer, and in general should not be allocated using the *new* statement. If it is not allocated using *new*, the garbage collection is handled automatically, as follows:

- When a *GPUtype* is created using *GPUmat* functions, a GPU variable is created in *GPUmat*, and pointer is set in *GPUtype* class as well.

- When the mex function exits, all the *GPUtype* variables defined on the stack are deleted. The corresponding pointer to *GPUmat* variable is deleted only if there are no more *GPUtype* objects pointing to the same variable.
- The creation of a *GPUtype* using *new* is not recommended.

The *GPUtype* has the following properties, with corresponding functions to access them:

Accessing GPUtype properties
`gpuTYPE_t TYPE (gm->gputype.getType)` Define the type of the *GPUtyte* (float, double, real, complex, etc.). Check *GPUmat.hh* for the definition.
`const int * SIZE (gm->gputype.getSize)` The SIZE array contains the dimensions of the *GPUtype* (for example {3,4,5}).
`int NDIMS (gm->gputype.getNdims)` The number of elements of the SIZE array.
`int NUMEL (gm->gputype.getNumel)` The number of elements. It is the product of the elements of SIZE.
`const void * GPUptr (gm->gputype.getGPUptr)` The pointer to the GPU memory.
`int DATASIZE (gm->gputype.getDataSize)` The size of the GPU variable on the GPU. For example, a gpuFLOAT has DATASIZE=4. A gpuCFLOAT has DATASIZE=8.

3.2 GPUmat internal functions

The structure *gm* created with *gmGetGPUmat* has several pointers to *GPUmat* internal functions. The complete reference of these functions is available in the *Developer Reference Manual*. The following shows a summary of the structure:

```
GPUmat
+ gputype
   +- getType
   +- getSize
+- numerics
   +- Abs
   +- Exp
```

```
   +- ExpDrv
   +- ...
+- fft
   +- FFT1Drv
   +- FFT2Drv
   +- FFT3Drv
   +- IFFT1Drv
   +- IFFT2Drv
   +- IFFT3Drv
```

The *numerics* functions are used to access *GPUmat* functions such as *Exp* or *Abs*. The *fft* functions are used to access *GPUmat* FFT functions. The following code (from file *myexp.cpp* in the *GPUmatModules/src/Examples/numerics* folder) shows how to access these functions:

```
GPUtype IN  = gm->gputype.getGPUtype(prhs[0]);
GPUtype OUT = gm->gputype.getGPUtype(prhs[1]);
gm->numerics.Exp(IN,OUT);
```

In the above code we read the variables *IN* and *OUT*, and run `Exp` on them. The *Exp* function calculates the exponential of the *IN* variable and stores the result in *OUT*. In general *GPUmat* functions require the output result to be passed as an input variable. But almost every function has also an equivalent driver function which creates also the output result. Driver functions have names such as *ExpDrv* or *AbsDrv*. Please check the *Developer Reference Manual* for more details. The previous code using driver function is the following:

```
GPUtype IN1  = gm->gputype.getGPUtype(prhs[0]);
GPUtype r    = gm->numerics.ExpDrv(IN1);
plhs[0] = gm->gputype.createMxArray(r);
```

In the above code the output variable *OUT* is created by the driver function *ExpDrv* and returned to Matlab using `gm->gputype.createMxArray`. Many other examples are available in the *GPUmatModules/src/Examples/GPUmat* folder. The following is the source code of the file *gmFFT1.cpp*:

```
#include <stdio.h>
#include <string.h>
#include <stdarg.h>
#ifdef UNIX
#include <stdint.h>
```

```
#endif
#include "mex.h"
// CUDA
#include "cuda.h"
#include "cuda_runtime.h"
#include "GPUmat.hh"
// static paramaters
static CUfunction drvfuns[4];
static int init = 0;
static GPUmat *gm;
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[]) {
  // At least 2 arguments expected
  // Input and result
  if (nrhs!=1)
     mexErrMsgTxt("Wrong number of arguments");
  if (init == 0) {
    // Initialize function
    //mexLock();
    // load GPUmat
    gm = gmGetGPUmat();
    init = 1;
  }
  // mex parameters are:
  // IN1
  // OUT
  GPUtype IN1  = gm->gputype.getGPUtype(prhs[0]);
  GPUtype R = gm->fft.FFT1Drv(IN1);

  plhs[0] = gm->gputype.createMxArray(R);
}
```

3.3 MEX file example

The following example (function *eye.cpp* from the *numerics* module) shows how to access a *GPUtype* from a mex function.

```
#include <stdio.h>
#include <string.h>
#include <stdarg.h>
```

```
#ifdef UNIX
#include <stdint.h>
#endif

#include "mex.h"

// CUDA
#include "cuda.h"
#include "cuda_runtime.h"

#include "GPUmat.hh"
#include "numerics.hh"

// static paramaters

static CUfunction drvfuns[4];
static int init = 0;
static GPUmat *gm;

/*
 * EYE(N, GPUsingle) is the N-by-N identity matrix.
 * EYE(N, GPUdouble) is the N-by-N identity matrix
 *
 * EYE(M,N, GPUsingle) or EYE([M,N], GPUsingle) is
 * an M-by-N matrix with 1's on the diagonal and
 * zeros elsewhere.
 *
 */
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[]) {

  CUresult cudastatus = CUDA_SUCCESS;

  // At least 2 arguments expected
  // The last argument is always a GPUtype
  if (nrhs<2)
     mexErrMsgTxt("Wrong number of arguments");

  if (init == 0) {
    // Initialize function
```

```
    mexLock();

    // load GPUmat
    gm = gmGetGPUmat();

    // load module
    CUmodule *drvmod = gmGetModule("numerics");

    // load float GPU function
    CUresult status =
      cuModuleGetFunction(&drvfuns[N_EYEF], *drvmod, "EYEF");
    if (CUDA_SUCCESS != status) {
      mexErrMsgTxt("Unable to load user function.");
    }

    // load complex GPU function
    status =
      cuModuleGetFunction(&drvfuns[N_EYEC], *drvmod, "EYEC");
    if (CUDA_SUCCESS != status) {
      mexErrMsgTxt("Unable to load user function.");
    }

    // load double GPU function
    status =
      cuModuleGetFunction(&drvfuns[N_EYED], *drvmod, "EYED");
    if (CUDA_SUCCESS != status) {
      mexErrMsgTxt("Unable to load user function.");
    }

    // load complex GPU function
    status =
      cuModuleGetFunction(&drvfuns[N_EYEDC], *drvmod, "EYEDC");
    if (CUDA_SUCCESS != status) {
      mexErrMsgTxt("Unable to load user function.");
    }

    init = 1;
  }

  // This function is called such as the last argument
  // is always of type GPUtype
```

```
  // For example:
  // eye(3,4,5,GPUsingle)
  //
  // mex parameters are:
  // LAST parameter -> IN
  // 0:LAST -> dimensions

  GPUtype IN = gm->gputype.getGPUtype(prhs[nrhs-1]);
  gpuTYPE_t tin = gm->gputype.getType(IN);

  // we use an existing GPUmat that allows to create
  // a GPUtype with variable arguments, similar to the
  // Matlab syntax for eye function

  // nrhs-1 because last argument is a GPUtype
  // r is the returned output
  GPUtype r = gm->gputype.createMx(tin, nrhs-1, prhs);

  try {
    GPUeye(r, gm, drvfuns);
  } catch (GPUexception ex) {
    mexErrMsgTxt(ex.getError());
  }

  plhs[0] = gm->gputype.createMxArray(r);

}
```

The following example (function *myexp.cpp* from the *GPUmatModules/src/Examples/numerics* module) shows how to access *GPUmat* internal functions from a mex function.

```
#include <stdio.h>
#include <string.h>
#include <stdarg.h>

#ifdef UNIX
#include <stdint.h>
#endif

#include "mex.h"
```

```
// CUDA
#include "cuda.h"
#include "cuda_runtime.h"

#include "GPUmat.hh"


// static paramaters

static CUfunction drvfuns[4];
static int init = 0;
static GPUmat *gm;

/*
 */
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[]) {

  CUresult cudastatus = CUDA_SUCCESS;

  // At least 2 arguments expected
  // Input and result
  if (nrhs!=2)
     mexErrMsgTxt("Wrong number of arguments");

  if (init == 0) {
    // Initialize function
    mexLock();

    // load GPUmat
    gm = gmGetGPUmat();

    // load module
    // NOT REQUIRED

    // load float GPU function
    // NOT REQUIRED

    init = 1;
  }
```

```
  // mex parameters are:
  // IN
  // OUT

  GPUtype IN  = gm->gputype.getGPUtype(prhs[0]);
  GPUtype OUT = gm->gputype.getGPUtype(prhs[1]);
  gm->numerics.Exp(IN,OUT);

}
```

Chapter 4
GPUmat GPU classes

The *GPUsingle* or *GPUdouble* classes are used to create and initialize GPU variables in Matlab, either using the empty constructor or using and existing Matlab variable. Here is an example:

```
Ah = rand(1000);            % Matlab variable
A  = GPUsingle(Ah);         % GPU variable
B  = GPUsingle(rand(100)); % GPU variable
C  = GPUdouble(rand(100)); % GPU variable
```

These classes implement a destructor, which frees the GPU memory that is not used anymore. The life-time of a GPU variable is the same as any other Matlab variable. In the following example, the second assignment to A automatically deletes the previously created variable and frees the corresponding GPU memory occupied by an array with `size=100x100`:

```
A = GPUsingle(rand(100));
A = GPUsingle(rand(10));
```

The *GPUsingle* or *GPUdouble* classes have the same properties. In the following example we introduce some of the properties of the *GPUsingle* class with a simple example: the low level function *cublasGetVector* is used to retrieve the content of the *GPUsingle A* into the Matlab variable *Ah*.

```
A = GPUsingle([1 2; 3 4]);
% Ah should be single precision, because
% A is single precision
Ah = single(zeros(1,numel(A)));
[status Ah] = cublasGetVector (numel(A), ...
              getSizeOf(A), getPtr(A), 1, Ah, 1);
```

```
cublasCheckStatus( status, ...
          'Unable to retrieve variable values from GPU.');
Ah

ans =

     1     3     2     4
```

In the result *Ah* the data is stored using column-major storage, the same format as Matlab and Fortran. Complex numbers are stored interleaving in memory imaginary and real part values. In the above example we use the CUBLAS function *cublasGetVector* to transfer the data from the GPU to the CPU memory. The function *numel* is used to get the number of elements in A. The function *getSizeOf* returns the size of a single element of A. Finally the function *getPtr* returns the pointer to the GPU memory.

4.1 GPUsingle, GPUdouble constructor

GPU variable constructor

`A = GPUsingle(Ah), A = GPUdouble(Ah)`

Creates a GPU variable *A* initialized with the Matlab array *Ah*. *A* has the same properties as *Ah*, such as the size and the number of elements. Example:

```
Ah = rand(1000);           % Matlab variable
A  = GPUsingle(Ah);        % GPU variable
B  = GPUsingle(rand(100)); % GPU variable
C  = GPUdouble(rand(100)); % GPU variable
```

`A = GPUsingle(), A = GPUdouble()`

Creates an empty GPU variable. GPU memory is not automatically allocated and the following steps must be performed to allocate the memory:

- step1: initialize the size of the array by using *setSize*.
- step2: set the type of the *GPUsingle* (*GPUdouble*) by using *setComplex* or *setReal* if the stored data is complex or real respectively.
- step3: use the *GPUallocVector* function. Please note that this function should be used only after *step1* and *step2*.

There is no memory transfer between the CPU and the GPU when using the empty constructor. Example:

```
A = GPUsingle();        %empty constructor
setSize(A,[100 100]);   %set variable size
setReal(A);             %set variable as real
GPUallocVector(A);      %allocate on GPU memory

A = GPUsingle();         %empty constructor
setSize(A,[10 10]);      %set variable size
setComplex(A);           %set variable as complex
GPUallocVector(A);       %allocate on GPU memory
```

Using the *zeros* function has a similar output as the above commands, and the GPU variable is inizialized with zeros. Example:

```
A = GPUsingle();        %empty constructor
setSize(A,[100 100]);  %set variable size
setReal(A);             %set variable as real
GPUallocVector(A);      %allocate on GPU memory

% above commands are similar to
A = zeros([100 100], GPUsingle);

% If we need a complex variable:
A = complex(zeros([100 100], GPUsingle));
```

4.2 GPUsingle, GPUdouble properties

Fields summary

`GPUPTR`

GPUPTR is the pointer to the GPU memory. The pointer is indirectly set by using *GPUallocVector*. Its value can be retrieved by using the *getPtr* function. Example:

```
N = 10;
A = rand(1,N,GPUsingle);
Isamin = cublasIsamin(N, getPtr(A), 1);
```

`COMPLEX`

COMPLEX is a flag and defines a complex GPU variable. It is set using *setComplex* and reset using *setReal*. Use *iscomplex* to check its value. The flag must be set using *setComplex* before allocating the variable memory using *GPUallocVector*. The flag has no effect if set after calling *GPUallocVector*. If a real GPU variable needs to be converted to complex use the function *complex*. Example:

```
A = rand(5,GPUsingle);
iscomplex(A)
A = GPUsingle(rand(5)+i*rand(5));
iscomplex(A)
```

`SIZE`

SIZE stores the variable size. The functions to modify it and to get its value are *setSize* and *size* (or *getSize*) respectively. The *SIZE* must be defined before using *GPUallocVector*. Modifying the *SIZE* on initialized variables changes only this property, but the elements in memory remain the same. The user is responsible to make sure that the *SIZE* property is consistent with stored elements. Otherwise use high level function *reshape* that has additional logic and checks. Example:

```
A = GPUsingle();
setSize(A,[100 100]);
GPUallocVector(A);
size(A)
```

4.3 GPUsingle, GPUdouble methods

Methods summary	
`getPtr(A)`	Get `GPUPTR` of the GPU variable A.
`setSize(A,size)`	Set `SIZE` of the GPU variable A.
`size(A) (getSize(A))`	Get the `SIZE` of the GPU variable A.
`setReal(A)`	Set the GPU variable A as real. This method should be used before allocating the GPU variable with *GPUallocVector*.
`setComplex(A)`	Set the GPU variable A as complex. This method should be used before allocating the GPU variable with *GPUallocVector*.
`isreal(A)`	Returns 1 if the GPU variable A is real.
`iscomplex(A)`	Returns 1 if the GPU variable A is complex.

Chapter 5
Numerics module

5.1 Indexed references

This document explains how to access the elements of a GPU variable using GPUmat internal functions. Basically, we want to perform in *GPUmat* operations similar to the following *Matlab* commands:

```
A = B(1:10);
A(1:10) = C;
```

The concepts explained in this document will be used to implement the functions *subsref* and *subsasgn* that are used in *GPUmat* to access the elements of a GPU variable. We will also develop the functions *slice* and *assign*, that are similar to *subsref* and *subsasgn* but faster.

The following functions (see file *GPUmat.hh*) are used to access the elements of a GPU array from a MEX file:

```
GPUtype (*slice)
  (const GPUtype &p, const Range &r);
GPUtype (*mxSlice)
  (const GPUtype &p, const Range &r);
void (*assign)
  (const GPUtype &p, const GPUtype &q, const Range &r, int dir);
void (*mxAssign)
  (const GPUtype &p, const GPUtype &q, const Range &r, int dir);
```

The functions *slice* and *assign* have the same behavior of *mxSlice* and *mx-Assign*, but using indexes that start from 0 instead of 1 (like in Matlab or Fortran).

The next sections present the following topics:

- GPUmat *slice* and *assign* internal functions.

- Implementation of the *Matlab* wrappers called *slice* and *assign* to the internal functions *slice* and *assign*.
- Implementation of the *Matlab* functions *subsref* and *subsasgn*.

5.1.1 GPUmat slice and assign (or mxSlice and mxAssign)

The function *assign* (or *mxAssign*) allows the user to perform the following operations depending on the value of the parameter *dir*:

```
dir=0 -> p = q(r)
dir=1 -> p(r) = q
```

In the above code, the parameter *r* is of type *Range* (defined in the file *GPUmat.hh*). A *Range* is constructed as a list of different type of *Range*:

- TYPE1. A *TYPE1 Range* defines a sequence of indexes from *inf* to *sup* with a specified *stride* (`[inf:stride:sup]`). A single value *Range* is considered also *TYPE1*.
- TYPE2. A *TYPE2 Range* is an array of indexes (int, float, double). For example, the indexes `[1 3 2 1]` cannot be represented using *TYPE1 Range* and a *TYPE2* is used.
- TYPE3. A *TYPE3 Range* is the same as *TYPE2*, but the indexes array is defined using a *GPUtype* variable.

The function *slice* (or *mxSlice*) is basically a wrapper to the function *assign* using `dir=0`. The result of the operation is created and returned to the caller. In the next sections we will use mainly the *assign* function in the examples.

TYPE1 Range

A *TYPE1 Range* is represented by 3 values (can degenerate to 1 value, representing only 1 element): *inf*,*stride*,*sup*. For example, the following command in Matlab:

```
A = B(1:10)
```

is equivalent to the following command:

```
gm->gputype.mxAssign(A, B, Range(1,1,10), 0);
```

We use *mxAssign* in the above command because the index starts from 1. In a similar way we can use *assign*, as follows:

```
gm->gputype.assign(A, B, Range(0,1,9), 0);
```

The statement `Range(0,1,9)` defines a sequence of indexes from 0 to 9. The *Range* type allows also to use the keywords *BEGIN* and *END*, in a similar way as they are used in Matlab. For example:

```
A = B(1:end)
```

is equivalent to the following command:

```
gm->gputype.mxAssign(A, B, Range(1,1,END), 0);
```

or

```
gm->gputype.mxAssign(A, B, Range(BEGIN,1,END), 0);
```

The *Range* type can be combined to define a multi dimensional *Range*. For example:

```
A = B(1:10,1:end)
```

is equivalent to the following command:

```
gm->gputype.mxAssign(A, B, Range(1,1,10,Range(1,1,END)), 0);
```

TYPE2 Range

A *TYPE2 Range* is a an array of indexes. The array can be of type **int**, **float** or **double**. A *TYPE2 Range* is constructed as follows:

```
Range(int s, int* c)
Range(int s, int* c, const Range &r)
```

The parameter *s* passed to the constructor represents the last index of the array *c*. The last index is the same as the number of elements minus 1. For example:

```
A = B([3 4 6 1])
```

is equivalent to the following command:

```
int r[] = {3,4,6,1};
gm->gputype.mxAssign(A, B, Range(3,r), 0);
```

Please note that the parameters *3* and *r* in `Range(3,r)` are the index of the last element in the array *r* and the pointer to the array *r*. A *TYPE2 Range* can be combined with a *TYPE1 Range*, as follows:

```
A = B([3 4 6 1],1:end)
```

is equivalent to the following command:

```
int r[] = {3,4,6,1};
gm->gputype.mxAssign(A, B, Range(3,r, Range(1,1,END)), 0);
```

TYPE3 Range

A *TYPE3 Range* is very similar to a *TYPE2 Range*, but the indexes array is a *GPUtype*. For example:

```
IDX = GPUsingle([3 4 6 1]);
A = B(IDX)
```

is equivalent to the following command:

```
gm->gputype.mxAssign(A, B, Range(IDX), 0);
```

More examples

```
A(1:10) = B
```

is equivalent to the following command:

```
gm->gputype.mxAssign(A, B, Range(1,1,10), 1);
```

```
A = B(1:10);
```

is equivalent to the following command:

```
GPUtype A = gm->gputype.mxSlice(B, Range(1,1,10));
plhs[0] = gm->gputype.createMxArray(A);
```

5.1.2 Matlab wrappers to GPUmat slice and assign functions

In this section we explain the implementation of the Matlab wrappers to the functions *slice* and *assign* explained in Section GPUmat slice and assign (mxSlice, mxAssign). The implemented *Matlab* functions in **NUMERICS MODULE** are:

```
B = slice(A, varargin)
assign(dir, A, B, varargin)
```

The above functions are wrappers to the *GPUmat slice* and *assign* functions. The parameter *varargin* represents a *Matlab* cell array of variable length, and it is used to store the *Range* definition. The *Range* should be defined with a syntax that is similar to the *Matlab* syntax used to access array elements. In particular, we manage the following cases:

- `A(1:2:10)`. This is similar to a *TYPE1 Range* explained in Section TYPE1 Range.
- `A([1 3 5 2 1])`. This is similar to a *TYPE2 Range* explained in Section TYPE2 Range.
- `A(:)`. The index ':' is equivalent to a *TYPE1 Range* (`Range(1,1,END)`).
- Keyword `end`, for example `A(1:2:10,end)`.

The structure of the MEX functions *slice.cpp* and *assign.cpp* is very simple:

- Read input parameters.
- Parse the *Range*.
- Call the *GPUmat* function (either *slice* or *assign*).
- Return a value (this point applies only to *slice*).

The *slice* code that performs the above operations is the following:

```
GPUtype RHS = gm->gputype.getGPUtype(prhs[0]);
Range *rg;
parseRange(nrhs-1,&prhs[1],&rg, mygc1);
GPUtype OUT = gm->gputype.mxSlice(RHS,*rg);
```

The core of the *slice* function is the *parseRange* function, defined in *numerics.cpp*. The *parseRange* function performs a loop through the elements of the *Matlab* cell array (*varargin*) that contains the range, and fills the variable *rg* passed as a reference. The parsing strategy used in *parseRange* is the following:

- The element found is of type *mxCHAR_CLASS*. This is interpreted as ':' and the generated *Range* is `Range(1,1,END)`.
- The element found is of type *mxDOUBLE_CLASS*. This is interpreted as a *TYPE1 Range*.
- The element found is of type *mxCELL_CLASS*. This is interpreted as a *TYPE2 Range*.

The *Range* is constructed creating new objects of type *Range*. A simple *Garbage Collector mygc1* is used to automatically delete created pointers. Please check the definition of the template class `MyGCObj<T>` in file *GPUmat.hh*.

Examples

```
Ah = Bh(1:end);
A = slice(B,[1,1,END]);

Ah = Bh(1:10,:);
A = slice(B,[1,1,10],':');

Ah = Bh([2 3 1],:);
A = slice(B,{[2 3 1]},':');

Ah = Bh([2 3 1],1);
A = slice(B,{[2 3 1]},1);

Ah = Bh(:,:);
A = slice(B,':',':');

assign(1, A, B, [1,1,10],[1,1,10]);
Ah(1:10,1:10) = Bh;

assign(1, A, B, {[2 3 1 5]},[1,1,10]);
Ah([2 3 1 5],1:10) = Bh;
```

5.1.3 subsref, subsasgn

Matlab functions *subsref* and *subsasgn* are called for commands similar to the following:

```
A = B(1:10)
C(1:10,:) = D;
```

For further information about *subsref* and *subsasgn* please check the *Matlab* manual. In particular, the following command:

```
A = B(1:10)
```

is translated into the following *Matlab* function call:

```
A = subsref(B,S)
```

where

```
S.type = '()'
S.subs = [1:10]
```

In a similar way, the following command:

```
B(1:10) = A
```

is translated into the following *Matlab* function call:

```
B = subsasgn(B,S,A)
```

where

```
S.type = '()'
S.subs = [1:10]
```

The implemented *subsref* and *subsasgn* MEX functions are very similar to the functions *slice* and *assign* explained in Matlab wrappers to GPUmat slice and assign functions. The structure is the following:

- Read input parameters.
- Parse the *Range*.
- Call the *GPUmat* function (either *slice* or *assign*).
- Return a value (this point applies only to *slice*).

The *Range* parsing is done by the function *parseMxRange*, which is similar to the function *parseRange* explained in previous sections.

In the function *subsasgn* we manage also automatic *GPUtype* variable casting. The internal *GPUmat* function *assign* requires the input *GPUtype* variables to be of the same type, but in function *subsasgn* we want to manage also the following condition:

```
A = rand(100,GPUsingle);
B = rand(100,GPUsingle);
A(1:10) = B(1:10);
```

The above commands require that the variable B is converted to single precision before assigning its values to A. This is done with the following code:

```
int lhsf =  gm->gputype.isFloat(LHS);
int rhsf =  gm->gputype.isFloat(RHS);
int lhsd =  gm->gputype.isDouble(LHS);
int rhsd =  gm->gputype.isDouble(RHS);
if (lhsf && rhsd) {
  // cast RHS to FLOAT
  RHS = gm->gputype.doubleToFloat(RHS);
```

```
}
if (lhsd && rhsf) {
  // cast RHS to DOUBLE
  RHS = gm->gputype.floatToDouble(RHS);
}
```

The same is done with *COMPLEX* and *REAL* variables, as follows:

```
int lhscpx =  gm->gputype.isComplex(LHS);
int rhscpx =  gm->gputype.isComplex(RHS);
if (lhscpx && !rhscpx) {
  // convert RHS to complex
  RHS = gm->gputype.realToComplex(RHS);
} else if (!lhscpx && rhscpx) {
  // convert LHS to complex
  LHS = gm->gputype.realToComplex(LHS);
}
```

Performance issues in *subsref* and *subsasgn*

The following expression:

```
A(1:end)
```

generates in *Matlab* a call to *subsref* passing an array with all the indexes. It means that if the array *A* has `1e6` elements, an array with `1e6` indexes will be passed to the function *subsref*. This is of course a huge waste of memory, because we don't need actually all the indexes to be stored, but just the first, the last and the stride. To avoid the creation of such huge array also on GPU memory, the function *parseMxRange* scans the indexes array and simplifies it if possible. **This operation is time consuming.** This situation is managed in a better way using the function *slice* or *assign*. Next section shows some performance tests.

5.1.4 Performance analysis

SUBSASGN performance (CPU = Dual_E6600@2.4GHZ,GPU = GTX275)					
N.	Operation	CPU	GPU (ver. 0.23)	GPU (ver. 0.22)	GPU assign
1	A(1:end) = B	0.007636	0.0126	0.01822	0.000382
2	A(1:10,:)= B	0.00006	0.000638	0.000333	0.000327
3	A(:,:)= B	0.003462	0.000706	0.000338	0.000371
4	A(1:2:end)= B	0.004054	0.006677	0.030853	0.000364
5	A(end:-5:1)= B	0.002161	0.003077	0.018304	0.000318
6	A(end:-5:1,:)= B	0.001726	0.000756	0.000904	0.000318
7	A(:) = B	0.000291	0.000658	0.003723	0.000356

The table shows in general that the performance of the *assign* function is better. As already mentioned in previous sections, the command:

```
A(1:end) = B
```

generates a call to *subsasgn* function passing the array of all indexes. In order to optimize the memory used on GPU, we simplify the indexes array when possible. This operation is time consuming and reduces the performance of the *subsasgn* function compared to *assign*. We have also the following remarks:

- The performance of the function *subsasgn* was improved in *GPUmat* version 0.23 compared to version 0.22.
- We do not expect the GPU to be faster than the CPU in memory operations.
- It is better to use the function *assign* if possible.

5.2 GPUfill

This document explains the usage and implementation of the function *GPUfill*. The *GPUfill* function is used to fill an existing array with specific value. The usage is:

```
GPUfill(A, offset, incr, m, p, offsetp, type)
```

The generic element *A(i)* of the variable *A* is modified as follows:

```
c    = incr*(i % m) + offset
A(i) = c
```

In the above expression `i % m` is the *mod(i,m)* (modulus). With `offset=1` and `incr=0` the result is to fill *A* with ones. For example:

```
A = zeros(5,GPUsingle);
GPUfill(A, 1, 0, 0, 0, 0, 0);
A

ans =

     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
```

The parameter type is used to modify the real or imaginary part of *A* as follows:

- type=0. Only the real part of *A* is modified.
- type=1. Only the imaginary part of *A* is modified.
- type=2. Both real and imaginary parts of *A* are modified.

The parameter *p* is used to select the elements of *A* to be modified. More specifically, only elements with index *i* such as `((i + offsep) % p)==0` are modified. If `offsetp=0` and `p=2`, then an element every 2 is modified, starting from the first element of the variable. For example:

```
A = zeros(5,GPUsingle);
GPUfill(A, 1, 0, 0, 2, 0, 0);
A
ans =

     1     0     1     0     1
     0     1     0     1     0
     1     0     1     0     1
     0     1     0     1     0
     1     0     1     0     1
```

Using `offsetp=1`, then an element every 2 is modified, starting from the second element of the variable. For example:

```
A = zeros(5,GPUsingle);
GPUfill(A, 1, 0, 0, 2, 1, 0);
A
ans =

     0     1     0     1     0
     1     0     1     0     1
     0     1     0     1     0
     1     0     1     0     1
     0     1     0     1     0
```

A sequence of numbers from 1 to `numel(A)` is generated as follows:

```
A = zeros(5,GPUsingle);
GPUfill(A, 1, 1, numel(A), 0, 0, 0);
A
ans =

     1     6    11    16    21
     2     7    12    17    22
     3     8    13    18    23
     4     9    14    19    24
     5    10    15    20    25
```

Same as above, but an element every 2 is modified:

```
A = zeros(5,GPUsingle);
GPUfill(A, 1, 1, numel(A), 2, 0, 0);
A
```

```
ans =

     1     0    11     0    21
     0     7     0    17     0
     3     0    13     0    23
     0     9     0    19     0
     5     0    15     0    25
```

The following examples show how to modify only the real or complex part (or both) using the *type* parameter.

```
A = zeros(2,complex(GPUsingle));
GPUfill(A, 1, 1, numel(A), 0, 0, 2);
A
ans =

   1.0000 + 1.0000i   3.0000 + 3.0000i
   2.0000 + 2.0000i   4.0000 + 4.0000i
```

```
A = zeros(2,complex(GPUsingle));
GPUfill(A, 1, 1, numel(A), 0, 0, 1);
A
ans =

        0 + 1.0000i        0 + 3.0000i
        0 + 2.0000i        0 + 4.0000i
```

5.2.1 Implementation

The function *GPUfill* is implemented using the GPUmat function *colon*. The *colon* interface has the following input parameters:

```
void (*colon) (const GPUtype &q, double offset,
               double incr, int m, int p,
               int offsetp, int type);
```

The meaning of the input parameters is the same as the parameters used in *GPUfill*. The mex function parses the input arguments as follows:

```
GPUtype DST = gm->gputype.getGPUtype(prhs[0]);
double offset = mxGetScalar(prhs[1]);
double incr   = mxGetScalar(prhs[2]);
int m         = (int) mxGetScalar(prhs[3]);
```

```
int p         = (int) mxGetScalar(prhs[4]);
int offsetp   = (int) mxGetScalar(prhs[5]);
int type      = (int) mxGetScalar(prhs[6]);
```

The function performs also some check on the input parameters and applies the default if necessary, as follows:

```
if ((type!=0)&&(type!=1)&&(type!=2))
  mexErrMsgTxt("Wrong type. Allowed values are 0,1,2");

if (m<=0)
  m = dst_numel;
if (p<=0)
  p = 1;
```

Finally, the *colon* function is called:

```
gm->gputype.colon(DST, offset, incr, m, p, offsetp, type);
```

5.2.2 Examples

Please refer to the file *GPUfill.m* in the *GPUmatModules/src/numerics/Examples* folder for more *GPUfill* examples.

5.3 REPMAT

This document explains the implementation of the Matlab function *repmat*. The *repmat* function is used to replicate an array, for example:

```
A = rand(5);
B = repmat(A,2,2);
```

The result is to replicate A as follows:

```
B= |A A|
   |A A|
```

The result of the following code

```
A = rand(5);
B = repmat(A,3,2);
```

is

```
B= |A A|
   |A A|
   |A A|
```

5.3.1 Implementation

The command

```
repmat(A, 2, 3)
```

where the matrix A has dimensions MxN, is equivalent to the following command:

```
A([1:M 1:M], [1:N 1:N 1:N]);
```

The command

```
repmat(A, 1, 1, 3)
```

where the matrix A has dimensions MxN, is equivalent to the following command:

```
A([1:M], [1:N], [1 1 1]);
```

The above examples can be generalized for a matrix A with arbitrary dimensions. The *repmat* function is implemented in the *NUMERICS* module (*repmat.cpp*). The code performs the following operations:

- Read and parse input parameters
- Create the output result R based on the concepts explained at the beginning of this section
- Return the result to Matlab

The mex function has 2 or more input arguments. The first argument is always the *GPUtype* that should be replicated. This is parsed using:

```
GPUtype IN = gm->gputype.getGPUtype(prhs[0]);
```

The remaining parameters (the number of parameters is variable), are parsed as follows. We use the GPUmat function *createMx*, which has the following interface:

```
GPUtype createMx (gpuTYPE_t type, int nrhs, const mxArray *prhs[]);
```

Please check the reference guide for more information about the above function. The function *createMx* creates a dummy *GPUtype*. The information we need from this variable is the size, which defines the way we have to repeat the input array. For example, the following input:

```
repmat(A,[2 3]);
```

is parsed in the mex file and a dummy *GPUtype* with *size=2x3* is created. The code is the following:

```
GPUtype DIM = gm->gputype.createMx(tin, nrhs-1, &prhs[1]);
int dim_ndims = gm->gputype.getNdims(DIM);
const int *dim_size = gm->gputype.getSize(DIM);
int nrep  = gm->gputype.getNumel(DIM);
```

The rest of the code generates the sequence of indexes to be applied to each dimension. For example, for a matrix A with dimensions 3x2, the command

```
repmat(A,2,3)
```

should generate in the mex something equivalent to the following:

```
A([1 2 3 1 2 3],[1 2 1 2 1 2])
```

A particular case is when `dim_ndims > in_ndims`. In this case we have to temporarily increase the dimensions of *IN* and restore them back at the end of the mex file. For example, for a matrix A with dimensions 3x2, the command

```
repmat(A,2,2,2)
```

should generate in the mex something equivalent to the following:

```
A([1 2 3 1 2 3],[1 2 1 2],[1 1])
```

The indexes *[1 1]* will generate an error because *IN* has only 2 dimensions, and we are actually accessing the 3rd dimension. The solution is to augment with ones the dimensions of *IN*. For example, if the size of *IN* is MxN, it will become MxNx1x1x1...x1. This operation is also possible in Matlab by using the GPUmat function *setSize*. For example:

```
A = GPUsingle(100);         % size(A) is [100 100]
setSize(A,[100 100 1 1 1]); % size(A) is [100 100 1 1 1]
```

The result R is obtained using the GPUmat *slice* function:

```
GPUtype R = gm->gputype.slice(IN, *(ind_range->next));
```

As already mentioned we have to restore the size of *IN* if it was modified:

```
if (inbackup_size!=NULL) {
  gm->gputype.setSize(IN, inbackup_ndims, inbackup_size);
}
```

Finally we return R to Matlab, as follows:

```
plhs[0] = gm->gputype.createMxArray(R);
```

5.3.2 Testing

The testing procedure can be found in *GPUmatModules/src/numerics/Tests/test_repmat.m*. To perform the tests the user has to initialize the environment variables as follows:

```
GPUtestInit
```

and then run the test:

```
test_repmat
```

The *test_repmat* procedure is configured to execute single/double and real/complex tests, depending on the configuration that has been set using the *GPUtestInit* function. For example, to run a real single precision test the user has to do the following:

```
GPUtestInit 'single' 'real'
test_repmat
```

Chapter 6
Examples module

6.1 GPUtype

The EXAMPLES:GPUTYPE module (*GPUmatModules/src/Examples/GPUtype*) contains the following examples:

- gputype_properties.cpp: shows how to access the properties of a GPUtype.
- gputype_create1.cpp: shows how to create a GPUtype and return it to Matlab.
- gputype_create2.cpp: shows how to create a GPUtype from a Matlab array.
- gputype_clone.cpp: clones a GPUtype.

6.1.1 gputype_properties.cpp

The properties of a GPUtype are described in the manual (The GPUtype class). The *gputype_properties* function takes a GPUtype argument as input and prints out information about it. The number of dimensions and the size vector are obtained as follows:

```
int ndims = gm->gputype.getNdims(IN1);
const int *s = gm->gputype.getSize(IN1);
```

The number of dimensions is the number of elements of the vector s. For example, the following code creates a 3x2x4 GPUtype:

```
A = rand(3,2,4,GPUsingle);
```

The variable A has the following *ndims* and s:

```
ndims = 3
s = {3,2,4}
```

The type of a GPUtype is defined in the *GPUmat.hh* file with the following enumeration:

```
enum gpuTYPE {
  gpuFLOAT = 0, gpuCFLOAT = 1, gpuDOUBLE = 2,
  gpuCDOUBLE = 3, gpuINT32 = 4, gpuNOTDEF = 20
};
```

Types are:

- gpuFLOAT: real/single precision type.
- gpuCFLOAT: complex/single precision type.
- gpuDOUBLE: real/double precision type.
- gpuCDOUBLE: complex/double precision type.
- gpuINT32: integer type is defined but not yet implemented.

Each element of a GPUtype has a size, depending on the type. This size in bytes can be obtained using the following code:

```
gm->gputype.getDataSize(IN1)
```

For example, the data size of a *gpuFLOAT* variable is 4. A *gpuCFLOAT* variables contains elements with size 8. To calculate the occupation in memory (in bytes) of a GPUtype variable, we have to multiply the number of elements by the data size. The number of elements is obtained as follows:

```
gm->gputype.getNumel(IN1)
```

6.1.2 gputype_create1.cpp

The *gputype_create1* function takes a value as input and generates a GPUtype. The input value is mapped into one of the possible GPUtype types, as follows:

```
gpuTYPE_t type = (gpuTYPE_t)( (int) mxGetScalar(prhs[0]));
```

The possible type values are limited, therefore we perform a check and eventually generate an error:

```
if ((type!=gpuFLOAT) && (type!=gpuCFLOAT) &&
    (type!=gpuDOUBLE) && (type!=gpuCDOUBLE)) {
  mexErrMsgTxt("Wrong TYPE");
}
```

We need the *type* and the *size* of the GPUtype to create it using the function *create*. This is done as follows:

```
int mysize[] = {100,100};
GPUtype R = gm->gputype.create(type,2,mysize, NULL);
```

The above code creates a 100x100 GPUtype. To initialize it with zeros, we use the CUDA function *cudaMemset*, which requires as one of the inputs the number of bytes to be written. Another information required by *cudaMemset* is the pointer to the GPU memory, which is obtained using the GPUmat function `gm->gputype.getGPUptr(R)`. The code for the initialization is the following:

```
// pointer to GPU memory
const void *gpuptr = gm->gputype.getGPUptr(R);
// number of elements
int numel = gm->gputype.getNumel(R);
// bytes for each element
int datasize = gm->gputype.getDataSize(R);

cudaError_t cudastatus = cudaSuccess;
cudastatus = cudaMemset((void *) gpuptr, 0, numel*datasize);
if (cudastatus != cudaSuccess) {
  mexErrMsgTxt("Error in cudaMemset");
}
```

6.1.3 gputype_create2.cpp

The *gputype_create2* function creates a GPUtype variable from the input Matlab array by using the *mxToGPUtype* GPUmat function. The code is the following:

```
GPUtype IN = gm->gputype.mxToGPUtype(prhs[0]);
plhs[0] = gm->gputype.createMxArray(IN);
```

6.1.4 gputype_clone

The function *gputype_clone* clones the input GPUtype. The returned variable points to a different GPU memory location. The code is the following:

```
GPUtype IN = gm->gputype.getGPUtype(prhs[0]);
GPUtype OUT = gm->gputype.clone(IN);
plhs[0] = gm->gputype.createMxArray(OUT);
```

6.1.5 Testing

Run the file *runme.m* in the modules folder to execute the examples for this module.