

GPUmat User Modules

Version 0.12, November 2009

Contents

Contents	2
1 Creating a new GPU kernel	5
1.1 Quick start	6
1.2 Compilation	6
1.3 Module manager and mex function initialization	7
1.4 MEX function summary	10
1.5 Testing	12
2 Accessing GPUmat functions and variables from a MEX file	13
2.1 The GPUtype class	14
2.2 GPUmat internal functions	15
2.3 MEX file example	17
3 GPUmat GPU classes	22
3.1 GPUsingle, GPUdouble constructor	24
3.2 GPUsingle, GPUdouble properties	26
3.3 GPUsingle, GPUdouble methods	27

The *GPUmat User Modules* project on Sourceforge (<http://sourceforge.net/projects/gpummatmodules/>) was started by the *GP-you Group* to allow users to add new functions to the Matlab toolbox GPUmat (<http://gp-you.org>) or to use existing GPUmat functions directly from C/C++ mex files. Check also the project *matCUDA* on Sourceforge (<http://sourceforge.net/projects/matcuda/>) if you are interested in Matlab wrappers to CUDA CUFFT and CUBLAS libraries. To run any module, the user has to do the following:

- Download and install GPUmat (<http://gp-you.org>).
- Download the module binary from Sourceforge and extract it to the folder where GPUmat was installed. The binary release includes stable modules. Source code is available for stable and unstable modules.
- Start GPUmat. Modules are automatically loaded if placed in the GPUmat installation folder. To manually load a module run the *moduleinit.m* script that must be provided with every module.

To extend GPUmat the user has the following options:

- Create new Matlab functions (*.m* Matlab files) using low and high level GPUmat functions. These functions are documented in the GPUmat User Guide.
- Create new compiled mex Matlab functions by accessing from the mex file existing GPUmat functions or new GPU kernels created by the user.

To implement either *.m* or mex functions 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.

Chapter 1

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 *modules/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 *GPU-userModuleLoad*, 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 provides pre-compiled binaries for different architectures, as explained in Section Quick Start. The binaries are the result of the compilation of the following files:

- *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.

1.1 Quick start

Before reading the next sections, you can run the pre-compiled code by executing from Matlab:

```
moduleinit  
runme
```

The output of the above command should be:

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

1.2 Compilation

The compilation is done from Matlab, using scripts that are provided in the *utils* folder. This folder must be added to the Matlab path. A valid compiler should be configured in Matlab using the *mex -setup* command. Please check Matlab documentation for more details about configuring a compiler. The file *nvidiasettings.m*, located in the *utils* folder, is used by the compilation scripts. It contains the following line:

```
CUDA_ROOT = 'I:\CUDA';
```

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

```
nvidiasettings
```

The output of the above command should be:

```
>> nvidiasettings  
NVIDIA settings OK
```

If you get an error, please check again that the variable `CUDA_ROOT` is pointing to the right place. The *make all* script is used to compile the CUDA module and the C++ files:

```
make all
```

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). A pre-compiled *numericsXX.cubin* can be found in the example folder. 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
```

1.3 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.

CHAPTER 1. Creating a new GPU kernel
1.3. MODULE MANAGER AND MEX FUNCTION INITIALIZATION

- 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.

1.4 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*.

1.5 Testing

Several functions are provided to help testing implemented functions. The *utils* 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 2

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 *GPUmat User Modules 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 are (file *GPUmat.hh*):

Creating GPUtype objects	
<code>gm->gputype.create</code>	Creates a GPUtype
<code>gm->gputype.createMx</code>	Creates a GPUtype
<code>gm->gputype.createMxArray</code>	Creates a GPUsingle or GPUdouble to be returned to Matlab
<code>gm->gputype.getGPUtype</code>	Creates a GPUtype from a Matlab GPUmat variable
<code>gm->gputype.slice</code>	Creates a slice from a GPUtype using specified Range
<code>gm->gputype.assign</code>	Assigns a GPUtype to another. A Range can be applied either to the left or right hand side

2.1 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 *GPUtype* (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`.

2.2 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 *GPUmat User Modules 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 *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 *GPUmat User Modules 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 *Examples/GPUmat* folder. The following is the source code of the file *gm.FFT1.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 drvfuncs[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);
}
```

2.3 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 drvfuncs[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(&drvfun[N_EYEF], *drvmod, "EYEF");
if (CUDA_SUCCESS != status) {
    mexErrMsgTxt("Unable to load user function.");
}

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

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

// load complex GPU function
status =
    cuModuleGetFunction(&drvfun[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, drvfuncs);
} catch (GPUexception ex) {
    mexErrMsgTxt(ex.getError());
}

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

}
```

The following example (function *myexp.cpp* from the *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 drvfun[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 3

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 an 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.

3.1 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 initialized with zeros. Example:

CHAPTER 3. GPUmat GPU classes
3.1. GPUSINGLE, GPUDOUBLE CONSTRUCTOR

```
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));
```

3.2 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 = GPUsingle(rand(1,N));
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 = GPUsingle(rand(5));
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)
```

3.3 GPUsingle, GPUdouble methods

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