

GPUmat User Modules

Version 0.15, April 2010

Contents

Contents	2
1 Creating a new GPU kernel	9
1.1 Quick start	10
1.2 Compilation	10
1.3 Module manager and mex function initialization	11
1.4 MEX function summary	14
1.5 Testing	16
2 Accessing GPUmat functions and variables from a MEX file	17
2.1 The GPUtype class	18
2.2 GPUmat internal functions	19
2.3 MEX file example	21
3 GPUmat GPU classes	27
3.1 GPUsingle, GPUdouble constructor	29
3.2 GPUsingle, GPUdouble properties	31
3.3 GPUsingle, GPUdouble methods	32
4 Numerics module	33
4.1 Release notes	33
4.2 Indexed references	35
4.2.1 GPUmat <i>slice</i> and <i>assign</i> (or <i>mxSlice</i> and <i>mxAssign</i>)	36
4.2.2 Matlab wrappers to GPUmat <i>slice</i> and <i>assign</i> functions	38
4.2.3 <i>subsref</i> , <i>subsasgn</i>	40
4.2.4 Performance analysis	43
4.3 GPUfill	44
4.3.1 Implementation	46
4.3.2 Examples	47
4.4 REPMAT	48
4.4.1 Implementation	48

4.4.2	Testing	50
5	Examples module	51
5.1	GPUtype	51
5.1.1	gputype_properties.cpp	51
5.1.2	gputype_create1.cpp	52
5.1.3	gputype_create2.cpp	53
5.1.4	gputype_clone	54
5.1.5	Testing	54
6	Function Reference	55
6.1	Functions - by module	55
6.1.1	NUMERICS module	55
6.1.2	EXAMPLES:GPATYPE module	58
6.1.3	EXAMPLES:CODEOPT module	59
6.1.4	EXAMPLES:NUMERICS module	59
6.2	Functions - alphabetical list	60
6.2.1	abs	60
6.2.2	acos	61
6.2.3	acosh	62
6.2.4	and	63
6.2.5	asin	64
6.2.6	asinh	65
6.2.7	assign	67
6.2.8	atan	68
6.2.9	atanh	69
6.2.10	ceil	70
6.2.11	clone	71
6.2.12	colon	72
6.2.13	conj	73
6.2.14	cos	74
6.2.15	cosh	75
6.2.16	ctranspose	76
6.2.17	eq	77
6.2.18	exp	78
6.2.19	eye	79
6.2.20	fft	80
6.2.21	fft2	81
6.2.22	floor	82
6.2.23	ge	83
6.2.24	getPtr	84

6.2.25	getSizeOf	85
6.2.26	getType	86
6.2.27	GPUabs	87
6.2.28	GPUacos	88
6.2.29	GPUacosh	89
6.2.30	GPUand	90
6.2.31	GPUasin	91
6.2.32	GPUasinh	92
6.2.33	GPUatan	93
6.2.34	GPUatanh	94
6.2.35	GPUceil	95
6.2.36	GPUconj	96
6.2.37	GPUcos	97
6.2.38	GPUcosh	98
6.2.39	GPUctranspose	99
6.2.40	GPUeq	100
6.2.41	GPUexp	101
6.2.42	GPUeye	102
6.2.43	GPUfill	104
6.2.44	GPUfloor	105
6.2.45	GPUge	106
6.2.46	GPUgt	107
6.2.47	GPUldivide	108
6.2.48	GPUle	109
6.2.49	GPUlog	110
6.2.50	GPUlog10	111
6.2.51	GPUlog1p	112
6.2.52	GPUlog2	113
6.2.53	GPUlt	114
6.2.54	GPUminus	115
6.2.55	GPUmtimes	116
6.2.56	GPUne	117
6.2.57	GPUnot	118
6.2.58	GPUones	119
6.2.59	GPUor	120
6.2.60	GPUplus	121
6.2.61	GPUpower	122
6.2.62	GPUrdivide	123
6.2.63	GPUround	124
6.2.64	GPUsin	125
6.2.65	GPUsinh	126

6.2.66	GPUsqrt	127
6.2.67	GPUtan	128
6.2.68	GPUtanh	129
6.2.69	GPUtimes	130
6.2.70	GPUtranspose	131
6.2.71	GPUuminus	132
6.2.72	GPUzeros	133
6.2.73	gt	134
6.2.74	ifft	135
6.2.75	ifft2	136
6.2.76	iscomplex	137
6.2.77	isempty	138
6.2.78	isreal	139
6.2.79	isscalar	140
6.2.80	ldivide	141
6.2.81	le	142
6.2.82	length	143
6.2.83	log	144
6.2.84	log10	145
6.2.85	log1p	146
6.2.86	log2	147
6.2.87	lt	148
6.2.88	memCpyDtoD	149
6.2.89	memCpyHtoD	150
6.2.90	minus	151
6.2.91	mrdivide	152
6.2.92	mtimes	153
6.2.93	ndims	154
6.2.94	ne	155
6.2.95	not	156
6.2.96	numel	157
6.2.97	ones	158
6.2.98	or	159
6.2.99	permute	160
6.2.100	plus	161
6.2.101	power	162
6.2.102	rdivide	163
6.2.103	repmat	164
6.2.104	round	165
6.2.105	sin	166
6.2.106	sinh	167

6.2.107	size	168
6.2.108	slice	170
6.2.109	sqrt	171
6.2.110	subsref	172
6.2.111	tan	173
6.2.112	tanh	174
6.2.113	times	175
6.2.114	transpose	176
6.2.115	uminus	177
6.2.116	zeros	179
6.3	Examples - alphabetical list	180
6.3.1	forloop1	180
6.3.2	gputype_create1	181
6.3.3	gputype_create2	182
6.3.4	gputype_properties	183
6.3.5	myexp	184
6.3.6	myplus	184
6.3.7	myslice1	185
6.3.8	myslice2	185
6.3.9	mytimes	186

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 or modify existing are (file *GPUmat.hh*):

Create and modify 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
<code>gm->gputype.clone</code>	Clones a GPUtype. The new GPUtype points to a different GPU memory location
<code>gm->gputype.mxToGPUtype</code>	Creates a GPUtype from a Matlab array
<code>gm->gputype.colon</code>	Fills a GPUtype with specified values. Can be used to create an array of ones, zeros or different sequence of values
<code>gm->gputype.floatToDouble</code>	Converts a single precision GPUtype to double precision.
<code>gm->gputype.doubleToFloat</code>	Converts a double precision GPUtype to single precision.
<code>gm->gputype.realToComplex</code>	Converts a real GPUtype to complex.

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

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 drvfun[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(&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, 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 drvfuncs[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.

Chapter 4

Numerics module

4.1 Release notes

NUMERICS MODULE

- Implemented functions
 - * GPUeye
 - * eye
 - * size
 - * subsref
 - * subsasgn
 - * END
 - * slice
 - * assign
 - * repmat
 - * GPUfill
 - * memCpyDtoD
 - * memCpyHtoD
 - * check manual for full reference

- Module initialization
 - * Run moduleinit from Matlab

EXAMPLES

Folder:

* Examples

Functions:

- * IndexedReference
- * SliceAssign
- * GPUfill
- * memCpy

COMPILATION

Include folder '../utils' in the MATLAB path.

- make cpp

Compiles all .cpp files

- make cuda

Compiles .cu files into .cubin modules

- make all

Compiles .cpp and .cu files and copy the compiled files to the release folder

TESTING

Run GPUtestInit before running tests

Folder

- * Tests

Functions

- * test_GPUeye
- * test_eye
- * test_slice
- * test_assign
- * test_repmat
- * test_GPUfill

```
* test_memCpyHtoD  
* test_memCpyDtoD
```

4.2 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 *mxAssign*, 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*.

4.2.1 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 *s* 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);
```

4.2.2 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;
```

4.2.3 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 = GPUsingle(rand(100));
B = GPUdouble(rand(100));
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.

4.2.4 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.

4.3 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 $((i + \text{offsetp}) \% p) == 0$ 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
```

4.3.1 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);
```

4.3.2 Examples

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

4.4 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|
```

4.4.1 Implementation

The command

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

where the matrix *A* has dimensions $M \times N$, 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 $M \times N$, 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 3×2 , 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 3×2 , 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 $M \times N$, it will become $M \times N \times 1 \times 1 \times 1 \dots \times 1$. 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);
```

4.4.2 Testing

The testing procedure can be found in `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 5

Examples module

5.1 GPUtype

The EXAMPLES:GPATYPE module (*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.

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

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* variable 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)
```

5.1.2 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");
}
```

5.1.3 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);
```

5.1.4 `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);
```

5.1.5 Testing

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

Chapter 6

Function Reference

6.1 Functions - by module

6.1.1 NUMERICS module

Name	Description
abs	Absolute value
acos	Inverse cosine
acosh	Inverse hyperbolic cosine
and	Logical AND
asin	Inverse sine
asinh	Inverse hyperbolic sine
assign	Indexed assignment
atan	Inverse tangent, result in radians
atanh	Inverse hyperbolic tangent
ceil	Round towards plus infinity
clone	Creates a copy of a GPUtype
colon	Colon
complex	Construct complex data from real and imaginary components
conj	CONJ(X) is the complex conjugate of X
cos	Cosine of argument in radians
cosh	Hyperbolic cosine
ctranspose	Complex conjugate transpose
eq	Equal
exp	Exponential
eye	Identity matrix
fft	Discrete Fourier transform
fft2	Two-dimensional discrete Fourier Transform

floor	Round towards minus infinity
ge	Greater than or equal
getPtr	Get pointer on GPU memory
getSizeOf	Get the size of the GPU datatype (similar to sizeof in C)
getType	Get the type of the GPU variable
GPUabs	Absolute value
GPUacos	Inverse cosine
GPUacosh	Inverse hyperbolic cosine
GPUand	Logical AND
GPUasin	Inverse sine
GPUasinh	Inverse hyperbolic sine
GPUatan	Inverse tangent, result in radians
GPUatanh	Inverse hyperbolic tangent
GPUceil	Round towards plus infinity
GPUcomplex	Construct complex data from real and imaginary components
GPUconj	$\text{GPUconj}(X, R)$ is the complex conjugate of X
GPUcos	Cosine of argument in radians
GPUcosh	Hyperbolic cosine
GPUctranspose	Complex conjugate transpose
GPUeq	Equal
GPUexp	Exponential
GPUeye	Identity matrix
GPUfill	Fill a GPU variable
GPUfloor	Round towards minus infinity
GPUge	Greater than or equal
GPUgt	Greater than
GPUimag	Imaginary part of complex number
GPUldivide	Left array divide
GPUle	Less than or equal
GPUlog	Natural logarithm
GPUlog10	Common (base 10) logarithm
GPUlog1p	Compute $\log(1+z)$ accurately
GPUlog2	Base 2 logarithm and dissect floating point number
GPUlt	Less than
GPUminus	Minus

GPUtimes	Matrix multiply
GPUne	Not equal
GPUnot	Logical NOT
GPUones	GPU ones array
GPUor	Logical OR
GPUplus	Plus
GPUpower	Array power
GPUrdivide	Right array divide
GPUreal	Real part of complex number
GPUround	Round towards nearest integer
GPUsin	Sine of argument in radians
GPUsinh	Hyperbolic sine
GPUsqrt	Square root
GPUtan	Tangent of argument in radians
GPUtanh	Hyperbolic tangent
GPUtimes	Array multiply
GPUtranspose	Transpose
GPUuminus	Unary minus
GPUzeros	GPU zeros array
gt	Greater than
ifft	Inverse discrete Fourier transform
ifft2	Two-dimensional inverse discrete Fourier transform
imag	Imaginary part of complex number
iscomplex	True for complex array
isempty	True for empty GPUsingle array
isreal	True for real array
isscalar	True if array is a scalar
ldivide	Left array divide
le	Less than or equal
length	Length of vector
log	Natural logarithm
log10	Common (base 10) logarithm
log1p	Compute $\log(1+z)$ accurately
log2	Base 2 logarithm and dissect floating point number
lt	Less than
memCpyDtoD	Device-Device memory copy

<code>memCpyHtoD</code>	Host-Device memory copy
<code>minus</code>	Minus
<code>mrdivide</code>	Slash or right matrix divide
<code>mtimes</code>	Matrix multiply
<code>ndims</code>	Number of dimensions
<code>ne</code>	Not equal
<code>not</code>	Logical NOT
<code>numel</code>	Number of elements in an array or subscripted array expression.
<code>ones</code>	GPU ones array
<code>or</code>	Logical OR
<code>permute</code>	Permute array dimensions
<code>plus</code>	Plus
<code>power</code>	Array power
<code>rdivide</code>	Right array divide
<code>real</code>	Real part of complex number
<code>repmat</code>	Replicate and tile an array
<code>round</code>	Round towards nearest integer
<code>sin</code>	Sine of argument in radians
<code>sinh</code>	Hyperbolic sine
<code>size</code>	Size of array
<code>slice</code>	Subscripted reference
<code>sqrt</code>	Square root
<code>subsref</code>	Subscripted reference
<code>tan</code>	Tangent of argument in radians
<code>tanh</code>	Hyperbolic tangent
<code>times</code>	Array multiply
<code>transpose</code>	Transpose
<code>uminus</code>	Unary minus
<code>zeros</code>	GPU zeros array

6.1.2 EXAMPLES:GPATYPE module

Name	Description
<code>gputype_create1</code>	GPUtype creation example
<code>gputype_create2</code>	GPUtype creation example
<code>gputype_properties</code>	GPUtype properties example

6.1.3 EXAMPLES:CODEOPT module

Name	Description
forloop1	Code optimization example

6.1.4 EXAMPLES:NUMERICS module

Name	Description
myexp	GPUmat internal functions example
myplus	GPUmat internal functions example
myslice1	GPUmat internal functions example
myslice2	GPUmat internal functions example
mytimes	GPUmat internal functions example

6.2 Functions - alphabetical list

6.2.1 abs

abs - Absolute value

SYNTAX

```
R = abs(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

ABS(X) is the absolute value of the elements of X. When X is complex, ABS(X) is the complex modulus (magnitude) of the elements of X.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(1,5)+i*rand(1,5));
R = abs(X)
```

6.2.2 `acos`

`acos` - Inverse cosine

SYNTAX

```
R = acos(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

`ACOS(X)` is the arccosine of the elements of `X`. NaN (Not A Number) results are obtained if `ABS(x) > 1.0` for some element.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));
R = acos(X)
```

MATLAB COMPATIBILITY

NaN returned if `ABS(x) > 1.0` . In this case Matlab returns a complex number. Not implemented for complex `X`.

6.2.3 acosh

acosh - Inverse hyperbolic cosine

SYNTAX

```
R = acosh(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

ACOSH(X) is the inverse hyperbolic cosine of the elements of X.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10)) + 1;
R = acosh(X)
```

MATLAB COMPATIBILITY

NaN is returned if $X < 1.0$. Not implemented for complex X.

6.2.4 and

and - Logical AND

SYNTAX

```
R = A & B  
R = and(A,B)  
A - GPUsingle, GPUdouble  
B - GPUsingle, GPUdouble  
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

A & B performs a logical AND of arrays A and B and returns an array containing elements set to either logical 1 (TRUE) or logical 0 (FALSE).

Compilation supported

EXAMPLE

```
A = GPUsingle([1 3 0 4]);  
B = GPUsingle([0 1 10 2]);  
R = A & B;  
single(R)
```

6.2.5 asin

asin - Inverse sine

SYNTAX

```
R = asin(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

ASIN(X) is the arcsine of the elements of X. NaN (Not A Number) results are obtained if $ABS(x) > 1.0$ for some element.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));
R = asin(X)
```

MATLAB COMPATIBILITY

NaN returned if $ABS(x) > 1.0$. In this case Matlab returns a complex number. Not implemented for complex X.

6.2.6 asinh

asinh - Inverse hyperbolic sine

SYNTAX

```
R = asinh(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

ASINH(X) is the inverse hyperbolic sine of the elements of X.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));
R = asinh(X)
```

MATLAB COMPATIBILITY

Not implemented for complex X.

CHAPTER 6. Function Reference
6.2. FUNCTIONS - ALPHABETICAL LIST

6.2.7 assign

assign - Indexed assignement

SYNTAX

`assign(dir, P, Q, R1, R2, ..., RN)`

P - GPUsingle, GPUdouble

Q - GPUsingle, GPUdouble, Matlab (scalar supported)

MODULE NAME

NUMERICS

DESCRIPTION

ASSIGN(DIR, P, Q, R1, R2, ..., RN) performs the following operations, depending on the value of the parameter DIR:

DIR = 0 -> P = Q(R1, R2, ..., RN)

DIR = 1 -> P(R1, R2, ..., RN) = Q

R1, R2, RN represents a sequence of ranges. A range can be constructed as follows:

[inf, stride, sup] - defines a range between inf and sup with specified stride. It is similar to the Matlab syntax A(inf:stride:sup). The special keyword END (please note, uppercase END) can be used.

':' - similar to the colon used in Matlab indexing.

{[i1, i2, ..., in]} -any array enclosed by brackets is considered an indexes array, similar to A([1 2 3 4 1 2]) in Matlab.

i1 - a single value is interpreted as an index. Similar to A(10) in Matlab.

Compilation supported

EXAMPLE

```
A = GPUsingle(rand(100));
B = GPUsingle(rand(10,10));
Ah = single(A);
Bh = single(B);
Ah(1:10,1:10) = Bh;
assign(1, A, B, [1,1,10],[1,1,10]);
assign(1, A, Bh, [1,1,10],[1,1,10]);
assign(1, A, single(10), [1,1,10],[1,1,10]);
```

6.2.8 atan

atan - Inverse tangent, result in radians

SYNTAX

```
R = atan(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

ATAN(X) is the arctangent of the elements of X.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));
R = atan(X)
```

MATLAB COMPATIBILITY

Not implemented for complex X.

6.2.9 atanh

atanh - Inverse hyperbolic tangent

SYNTAX

```
R = atanh(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

ATANH(X) is the inverse hyperbolic tangent of the elements of X.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));
R = atanh(X)
```

MATLAB COMPATIBILITY

Not implemented for complex X.

6.2.10 `ceil`

`ceil` - Round towards plus infinity

SYNTAX

```
R = ceil(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

`CEIL(X)` rounds the elements of `X` to the nearest integers towards infinity.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));
R = ceil(X)
```

MATLAB COMPATIBILITY

Not implemented for complex `X`.

6.2.11 clone

clone - Creates a copy of a GPUtype

SYNTAX

```
R = clone(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

CLONE(X) creates a copy of X.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));
R = clone(X)
```

6.2.12 colon

colon - Colon

SYNTAX

```
R = colon(J,K,GPUsingle)
R = colon(J,D,K,GPUsingle)
```

MODULE NAME

NUMERICS

DESCRIPTION

`COLON(J,K,GPUsingle)` is the same as `J:K` and `COLON(J,D,K,GPUsingle)` is the same as `J:D:K`. `J:K` is the same as `[J, J+1, ..., K]`. `J:K` is empty if `J > K`. `J:D:K` is the same as `[J, J+D, ..., J+m*D]` where `m = fix((K-J)/D)`. `J:D:K` is empty if `D == 0`, if `D > 0` and `J > K`, or if `D < 0` and `J < K`.

Compilation supported

EXAMPLE

```
A = colon(1,2,10,GPUsingle)
```

6.2.13 conj

conj - `CONJ(X)` is the complex conjugate of `X`

SYNTAX

```
R = conj(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

For a complex `X`, `CONJ(X) = REAL(X) - i*IMAG(X)`.

Compilation supported

EXAMPLE

```
A = GPUsingle(rand(1,5) + i*rand(1,5));
B = conj(A)
```

6.2.14 `cos`

`cos` - Cosine of argument in radians

SYNTAX

```
R = cos(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

`COS(X)` is the cosine of the elements of `X`.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));
R = cos(X)
```

MATLAB COMPATIBILITY

Not implemented for complex `X`.

6.2.15 `cosh`

`cosh` - Hyperbolic cosine

SYNTAX

```
R = cosh(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

`COSH(X)` is the hyperbolic cosine of the elements of `X`.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));
R = cosh(X)
```

MATLAB COMPATIBILITY

Not implemented for complex `X`.

6.2.16 `ctranspose`

`ctranspose` - Complex conjugate transpose

SYNTAX

```
R = X'  
R = ctranspose(X)  
X - GPUsingle, GPUdouble  
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

X' is the complex conjugate transpose of X .

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10)+i*rand(10));  
R = X'  
R = ctranspose(X)
```

6.2.17 eq

eq - Equal

SYNTAX

```
R = X == Y
R = eq(X,Y)
X - GPUsingle, GPUdouble
Y - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

A == B (eq(A, B)) does element by element comparisons between A and B.

Compilation supported

EXAMPLE

```
A = GPUsingle([1 2 0 4]);
B = GPUsingle([1 0 0 4]);
R = A == B;
single(R)
R = eq(A, B);
single(R)
```

6.2.18 `exp`

`exp` - Exponential

SYNTAX

R = `exp(X)`
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble

MODULE NAME

NUMERICS

DESCRIPTION

`EXP(X)` is the exponential of the elements of X, e to the X. For complex $Z=X+i*Y$, $EXP(Z) = EXP(X)*(COS(Y)+i*SIN(Y))$.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(1,5)+i*rand(1,5));  
R = exp(X)
```

6.2.19 eye

eye - Identity matrix

SYNTAX

```
eye(N, CLASSNAME)
eye(M, N, CLASSNAME)
eye([M, N], CLASSNAME)
eye(M, N, P, ...?, CLASSNAME)
eye([M N P ...], CLASSNAME)
```

CLASSNAME = GPUsingle/GPUdouble

MODULE NAME

NUMERICS

DESCRIPTION

EYE(M, N, CLASSNAME) or EYE([M, N], CLASSNAME) is an M-by-N matrix with 1's of class CLASSNAME on the diagonal and zeros elsewhere. CLASSNAME can be GPUsingle or GPUdouble

Compilation supported

EXAMPLE

```
X = eye(2,3,GPUsingle);
X = eye([4 5], GPUdouble);
```

6.2.20 `fft`

`fft` - Discrete Fourier transform

SYNTAX

```
R = fft(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

`FFT(X)` is the discrete Fourier transform (DFT) of vector `X`.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(1,5)+i*rand(1,5));
R = fft(X)
```

6.2.21 `fft2`

`fft2` - Two-dimensional discrete Fourier Transform

SYNTAX

```
R = fft2(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

FFT2(X) returns the two-dimensional Fourier transform of matrix X.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(5,5)+i*rand(5,5));
R = fft2(X)
```

6.2.22 floor

floor - Round towards minus infinity

SYNTAX

```
R = floor(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

FLOOR(X) rounds the elements of X to the nearest integers towards minus infinity.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(1,5));
R = floor(X)
```

MATLAB COMPATIBILITY

Not implemented for complex X.

6.2.23 ge

ge - Greater than or equal

SYNTAX

```
R = X >= Y
R = ge(X,Y)
X - GPUsingle, GPUdouble
Y - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

$A \geq B$ (`ge(A, B)`) does element by element comparisons between A and B.

Compilation supported

EXAMPLE

```
A = GPUsingle([1 2 0 4]);
B = GPUsingle([1 0 0 4]);
R = A >= B;
single(R)
R = ge(A, B);
single(R)
```

6.2.24 `getPtr`

getPtr - Get pointer on GPU memory

SYNTAX

`R = getPtr(X)`

`X` - GPU variable

`R` - the pointer to the GPU memory region

MODULE NAME

NUMERICS

DESCRIPTION

This is a low level function used to get the pointer value to the GPU memory of a GPU variable

Compilation not supported

EXAMPLE

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

6.2.25 `getSizeOf`

getSizeOf - Get the size of the GPU datatype (similar to `sizeof` in C)

SYNTAX

```
R = getSizeOf(X)
X - GPU variable
R - the size of the GPU variable datatype
```

MODULE NAME

NUMERICS

DESCRIPTION

This is a low level function used to get the size of the datatype of the GPU variable.

Compilation not supported

EXAMPLE

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

6.2.26 `getType`

getType - Get the type of the GPU variable

SYNTAX

```
R = getType(X)
X - GPU variable
R - the type of the GPU variable
```

MODULE NAME

NUMERICS

DESCRIPTION

This is a low level function used to get the type of the GPU variable (FLOAT = 0, COMPLEX FLOAT = 1, DOUBLE = 2, COMPLEX DOUBLE = 3)

Compilation not supported

EXAMPLE

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

6.2.27 GPUabs

GPUabs - Absolute value

SYNTAX

```
R = GPUabs(X, R)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

`GPUabs(X, R)` is equivalent to `ABS(X)`, but result is returned in the input parameter `R`.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(1,5)+i*rand(1,5));
R = zeros(size(X),GPUsingle);
GPUabs(X, R)
```

6.2.28 GPUacos

GPUacos - Inverse cosine

SYNTAX

`GPUacos(X, R)`

X - `GPUsingle`, `GPUdouble`

R - `GPUsingle`, `GPUdouble`

MODULE NAME

NUMERICS

DESCRIPTION

`GPUacos(X, R)` is equivalent to `ACOS(X)`, but result is returned in the input parameter R.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));  
R = zeros(size(X), GPUsingle);  
GPUacos(X, R)
```

6.2.29 GPUacosh

GPUacosh - Inverse hyperbolic cosine

SYNTAX

`GPUacosh(X, R)`

X - `GPUsingle`, `GPUdouble`

R - `GPUsingle`, `GPUdouble`

MODULE NAME

NUMERICS

DESCRIPTION

`GPUacosh(X, R)` is equivalent to `ACOSH(X)`, but result is returned in the input parameter R.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10)) + 1;
R = zeros(size(X), GPUsingle);
GPUacosh(X, R)
```

6.2.30 GPUand

GPUand - Logical AND

SYNTAX

GPUand(A, B, R)

A - GPUsingle, GPUdouble

B - GPUsingle, GPUdouble

R - GPUsingle, GPUdouble

MODULE NAME

NUMERICS

DESCRIPTION

GPUand(A, B, R) is equivalent to A & B, but result is returned in the input parameter R.

Compilation supported

EXAMPLE

```
A = GPUsingle([1 3 0 4]);  
B = GPUsingle([0 1 10 2]);  
R = zeros(size(A), GPUsingle);  
GPUand(A, B, R);
```

6.2.31 GPUasin

GPUasin - Inverse sine

SYNTAX

`GPUasin(X, R)`

X - `GPUsingle`, `GPUdouble`

R - `GPUsingle`, `GPUdouble`

MODULE NAME

NUMERICS

DESCRIPTION

`GPUasin(X, R)` is equivalent to `ASIN(X)`, but result is returned in input parameter R.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));  
R = zeros(size(X), GPUsingle);  
GPUasin(X, R);
```

6.2.32 GPUasinh

GPUasinh - Inverse hyperbolic sine

SYNTAX

`GPUasinh(X, R)`

X - `GPUsingle`, `GPUdouble`

R - `GPUsingle`, `GPUdouble`

MODULE NAME

NUMERICS

DESCRIPTION

`GPUasinh(X, R)` is equivalent to `ASINH(X)` , but result is returned in the input parameter R.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));  
R = zeros(size(X), GPUsingle);  
GPUasinh(X, R)
```

6.2.33 GPUatan

GPUatan - Inverse tangent, result in radians

SYNTAX

GPUatan(X, R)

X - GPUsingle, GPUdouble

R - GPUsingle, GPUdouble

MODULE NAME

NUMERICS

DESCRIPTION

GPUatan(X, R) is equivalent to `ATAN(X)`, but result is returned in the input parameter R.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));  
R = zeros(size(X), GPUsingle);  
GPUatan(X, R)
```

6.2.34 GPUatanh

GPUatanh - Inverse hyperbolic tangent

SYNTAX

GPUatanh(X, R)

X - GPUsingle, GPUdouble

R - GPUsingle, GPUdouble

MODULE NAME

NUMERICS

DESCRIPTION

GPUatanh(X, R) is equivalent to ATANH(X), but result is returned in the input parameter R.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));  
R = zeros(size(X), GPUsingle);  
GPUatanh(X, R)
```

6.2.35 GPUceil

GPUceil - Round towards plus infinity

SYNTAX

`GPUceil(X, R)`

X - `GPUsingle`, `GPUdouble`

R - `GPUsingle`, `GPUdouble`

MODULE NAME

NUMERICS

DESCRIPTION

`GPUceil(X, R)` is equivalent to `CEIL(X)`, but result is returned in the input parameter R.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));  
R = zeros(size(X), GPUsingle);  
GPUceil(X, R)
```

6.2.36 GPUconj

GPUconj - GPUconj(X, R) is the complex conjugate of X

SYNTAX

GPUconj(X, R)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble

MODULE NAME

NUMERICS

DESCRIPTION

GPUconj(X, R) is equivalent to CONJ(X), but result is returned in the input parameter R.

Compilation supported

EXAMPLE

```
A = GPUsingle(rand(1,5) + i*rand(1,5));  
R = complex(zeros(size(A), GPUsingle));  
GPUconj(A, R)
```

6.2.37 GPUcos

GPUcos - Cosine of argument in radians

SYNTAX

`GPUcos(X, R)`

X - `GPUsingle`, `GPUdouble`

R - `GPUsingle`, `GPUdouble`

MODULE NAME

NUMERICS

DESCRIPTION

`GPUcos(X, R)` is equivalent to `COS(X)`, but result is returned in the input parameter R.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));  
R = zeros(size(X), GPUsingle);  
GPUcos(X, R)
```

6.2.38 GPUcosh

GPUcosh - Hyperbolic cosine

SYNTAX

`GPUcosh(X, R)`

X - `GPUsingle`, `GPUdouble`

R - `GPUsingle`, `GPUdouble`

MODULE NAME

NUMERICS

DESCRIPTION

`GPUcosh(X, R)` is equivalent to `COSH(X)` , but result is returned in the input parameter R.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));  
R = zeros(size(X), GPUsingle);  
GPUcosh(X, R)
```

6.2.39 GPUctranspose

GPUctranspose - Complex conjugate transpose

SYNTAX

```
GPUctranspose(X, R)  
X - GPUsingle, GPUdouble  
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

`GPUctranspose(X, R)` is equivalent to `ctranspose(X)`, but result is returned in the input parameter `R`.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10)+i*rand(10));  
R = complex(zeros(size(X), GPUsingle));  
GPUctranspose(X, R)
```

6.2.40 GPUeq

GPUeq - Equal

SYNTAX

`GPUeq(X,Y,R)`

X - `GPUsingle`, `GPUdouble`

Y - `GPUsingle`, `GPUdouble`

R - `GPUsingle`, `GPUdouble`

MODULE NAME

NUMERICS

DESCRIPTION

`GPUeq(A, B, R)` is equivalent to `eq(A, B)`, but result is returned in the input parameter R.

Compilation supported

EXAMPLE

```
A = GPUsingle([1 2 0 4]);
B = GPUsingle([1 0 0 4]);
R = zeros(size(A), GPUsingle);
GPUeq(A, B, R);
```

6.2.41 GPUexp

GPUexp - Exponential

SYNTAX

GPUexp(X, R)

X - GPUsingle, GPUdouble

R - GPUsingle, GPUdouble

MODULE NAME

NUMERICS

DESCRIPTION

GPUexp(X, R) is equivalent to EXP(X), but result is returned in the input parameter R.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(1,5)+i*rand(1,5));  
R = complex(zeros(size(X), GPUsingle));  
GPUexp(X, R)
```

6.2.42 GPUeye

GPUeye - Identity matrix

SYNTAX

GPUeye(R)
R - GPUsingle, GPUdouble

MODULE NAME

NUMERICS

DESCRIPTION

GPUeye(R) fills the matrix R with 1's on the diagonal and zeros elsewhere.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));  
GPUeye(X)
```

CHAPTER 6. Function Reference
6.2. FUNCTIONS - ALPHABETICAL LIST

6.2.43 GPUfill

GPUfill - Fill a GPU variable

SYNTAX

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

A - GPUsingle, GPUdouble

offset, incr, m, p, offsetp, type - Matlab

MODULE NAME

NUMERICS

DESCRIPTION

GPUfill(A, offset, incr, m, p, offsetp, type) fills an existing array with specific values.

Compilation supported

EXAMPLE

```
%% Fill with ones
A = zeros(5,GPUsingle);
GPUfill(A, 1, 0, 0, 0, 0, 0);
%% Fill with ones, and element every 2
A = zeros(5,GPUsingle);
GPUfill(A, 1, 0, 0, 2, 0, 0);
%% Fill with ones, and element every 2
% starting from the 2nd element
A = zeros(5,GPUsingle);
GPUfill(A, 1, 0, 0, 2, 1, 0);
%% Fill with a sequence of numbers from 1 to numel(A)
A = zeros(5,GPUsingle);
GPUfill(A, 1, 1, numel(A), 0, 0, 0);
%% Fill with a sequence of numbers from 1 to numel(A)
% An element every 2 is modified
A = zeros(5,GPUsingle);
GPUfill(A, 1, 1, numel(A), 2, 0, 0);
%% type=2 to modify both real and complex part
A = zeros(2,complex(GPUsingle));
GPUfill(A, 1, 1, numel(A), 0, 0, 2);
%% Modify only the complex part
A = zeros(2,complex(GPUsingle));
GPUfill(A, 1, 1, numel(A), 0, 0, 1);
```

6.2.44 GPUfloor

GPUfloor - Round towards minus infinity

SYNTAX

GPUfloor(X, R)

X - GPUsingle, GPUdouble

R - GPUsingle, GPUdouble

MODULE NAME

NUMERICS

DESCRIPTION

GPUfloor(X, R) is equivalent to FLOOR(X), but result is returned in the input parameter R.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(1,5));  
R = zeros(size(X), GPUsingle);  
GPUfloor(X, R)
```

6.2.45 GPUge

GPUge - Greater than or equal

SYNTAX

GPUge(X,Y,R)

X - GPUsingle, GPUdouble

Y - GPUsingle, GPUdouble

R - GPUsingle, GPUdouble

MODULE NAME

NUMERICS

DESCRIPTION

GPUge(A, B, R) is equivalent to ge(A, B), but result is returned in the input parameter R.

Compilation supported

EXAMPLE

```
A = GPUsingle([1 2 0 4]);  
B = GPUsingle([1 0 0 4]);  
R = zeros(size(B),GPUsingle);  
GPUge(A, B, R);
```

6.2.46 GPUgt

GPUgt - Greater than

SYNTAX

GPUgt(X,Y, R)
X - GPUsingle, GPUdouble
Y - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble

MODULE NAME

NUMERICS

DESCRIPTION

GPUgt(A, B, R) is equivalent to gt(A, B), but result is returned in the input parameter R.

Compilation supported

EXAMPLE

```
A = GPUsingle([1 2 0 4]);  
B = GPUsingle([1 0 0 4]);  
R = zeros(size(B), GPUsingle);  
GPUgt(A, B, R);
```

6.2.47 GPUldivide

GPUldivide - Left array divide

SYNTAX

GPUldivide(X,Y,R)

X - GPUsingle, GPUdouble

Y - GPUsingle, GPUdouble

R - GPUsingle, GPUdouble

MODULE NAME

NUMERICS

DESCRIPTION

GPUldivide(A, B, R) is equivalent to ldivide(A, B), but result is returned in the input parameter R.

Compilation supported

EXAMPLE

```
A = GPUsingle(rand(10));
B = GPUsingle(rand(10));
R = zeros(size(B), GPUsingle);
GPUldivide(A, B, R);
```

6.2.48 GPUle

GPUle - Less than or equal

SYNTAX

`GPUle(X,Y,R)`

X - `GPUsingle`, `GPUdouble`

Y - `GPUsingle`, `GPUdouble`

R - `GPUsingle`, `GPUdouble`

MODULE NAME

NUMERICS

DESCRIPTION

`GPUle(A, B, R)` is equivalent to `le(A, B)`, but result is returned in the input parameter R.

Compilation supported

EXAMPLE

```
A = GPUsingle([1 2 0 4]);
B = GPUsingle([1 0 0 4]);
R = zeros(size(A), GPUsingle);
GPUle(A, B, R);
```

6.2.49 GPUlog

GPUlog - Natural logarithm

SYNTAX

GPUlog(X, R)

X - GPUsingle, GPUdouble

R - GPUsingle, GPUdouble

MODULE NAME

NUMERICS

DESCRIPTION

GPUlog(X,R) is equivalent to LOG(X), but the result is returned in input parameter R.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));  
R = zeros(size(X), GPUsingle);  
GPUlog(X,R)
```

6.2.50 GPUlog10

GPUlog10 - Common (base 10) logarithm

SYNTAX

GPUlog10(X, R)

X - GPUsingle, GPUdouble

R - GPUsingle, GPUdouble

MODULE NAME

NUMERICS

DESCRIPTION

GPUlog10(X, R) is equivalent to LOG10(X), but the result is returned in input parameter R.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));  
R = zeros(size(X), GPUsingle);  
GPUlog10(X, R)
```

6.2.51 GPUlog1p

GPUlog1p - Compute $\log(1+z)$ accurately

SYNTAX

GPUlog1p(X, R)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble

MODULE NAME

NUMERICS

DESCRIPTION

GPUlog1p(X, R) is equivalent to LOG1P(X), but the result is returned in input parameter R.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));  
R = zeros(size(X), GPUsingle);  
GPUlog1p(X, R)
```

6.2.52 GPUlog2

GPUlog2 - Base 2 logarithm and dissect floating point number

SYNTAX

GPUlog2(X, R)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble

MODULE NAME

NUMERICS

DESCRIPTION

GPUlog2(X, R) is equivalent to LOG2(X), but the result is returned in input parameter R.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));  
R = zeros(size(X), GPUsingle);  
GPUlog2(X, R)
```

6.2.53 GPUlt

GPUlt - Less than

SYNTAX

`GPUlt(X,Y,R)`

X - `GPUsingle`, `GPUdouble`

Y - `GPUsingle`, `GPUdouble`

R - `GPUsingle`, `GPUdouble`

MODULE NAME

NUMERICS

DESCRIPTION

`GPUlt(X, Y, R)` is equivalent to `lt(X, Y)`, but the result is returned in input parameter R.

Compilation supported

EXAMPLE

```
A = GPUsingle([1 2 0 4]);
B = GPUsingle([1 0 0 4]);
R = zeros(size(B), GPUsingle);
GPUlt(A, B, R);
```

6.2.54 GPUminus

GPUminus - Minus

SYNTAX

`GPUminus(X,Y,R)`

X - `GPUsingle`, `GPUdouble`

Y - `GPUsingle`, `GPUdouble`

R - `GPUsingle`, `GPUdouble`

MODULE NAME

NUMERICS

DESCRIPTION

`GPUminus(X, Y, R)` is equivalent to `minus(X, Y)`, but the result is returned in input parameter R.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));
Y = GPUsingle(rand(10));
R = zeros(size(X), GPUsingle);
GPUminus(Y, X, R);
```

6.2.55 GPUmtimes

GPUmtimes - Matrix multiply

SYNTAX

`GPUmtimes(X,Y,R)`

X - `GPUsingle`, `GPUdouble`

R - `GPUsingle`, `GPUdouble`

Y - `GPUsingle`, `GPUdouble`

MODULE NAME

NUMERICS

DESCRIPTION

`GPUmtimes(X, Y, R)` is equivalent to `mtimes(X, Y)`, but the result is returned in input parameter R.

Compilation supported

EXAMPLE

```
A = GPUsingle(rand(10));
B = GPUsingle(rand(10));
R = zeros(size(A), GPUsingle);
GPUmtimes(A, B, R);
```

6.2.56 GPUne

GPUne - Not equal

SYNTAX

`GPUne(X,Y,R)`

X - `GPUsingle`, `GPUdouble`

Y - `GPUsingle`, `GPUdouble`

R - `GPUsingle`, `GPUdouble`

MODULE NAME

NUMERICS

DESCRIPTION

`GPUne(X, Y, R)` is equivalent to `ne(X, Y)`, but the result is returned in input parameter R.

Compilation supported

EXAMPLE

```
A = GPUsingle([1 2 0 4]);
B = GPUsingle([1 0 0 4]);
R = zeros(size(B), GPUsingle);
GPUne(A, B, R);
```

6.2.57 GPUnot

GPUnot - Logical NOT

SYNTAX

`GPUnot(X, R)`

X - `GPUsingle`, `GPUdouble`

R - `GPUsingle`, `GPUdouble`

MODULE NAME

NUMERICS

DESCRIPTION

`GPUnot(X, R)` is equivalent to `not(X)`, but the result is returned in input parameter R.

Compilation supported

EXAMPLE

```
A = GPUsingle([1 2 0 4]);  
R = zeros(size(A), GPUsingle);  
GPUnot(A, R);
```

6.2.58 GPUones

GPUones - GPU ones array

SYNTAX

`GPUones(R)`

R - `GPUsingle`, `GPUdouble`

MODULE NAME

NUMERICS

DESCRIPTION

`GPUones(R)` sets to one all the elements of R.

Compilation supported

EXAMPLE

```
A = GPUsingle(rand(5));  
GPUones(A)
```

6.2.59 GPUor

GPUor - Logical OR

SYNTAX

`GPUor(X,Y, R)`

X - `GPUsingle`, `GPUdouble`

Y - `GPUsingle`, `GPUdouble`

R - `GPUsingle`, `GPUdouble`

MODULE NAME

NUMERICS

DESCRIPTION

`GPUor(X, Y, R)` is equivalent to `or(X, Y)`, but the result is returned in input parameter R.

Compilation supported

EXAMPLE

```
A = GPUsingle([1 2 0 4]);
B = GPUsingle([1 0 0 4]);
R = zeros(size(B), GPUsingle);
GPUor(A, B, R);
```

6.2.60 GPUplus

GPUplus - Plus

SYNTAX

GPUplus(X,Y,R)

X - GPUsingle, GPUdouble

Y - GPUsingle, GPUdouble

R - GPUsingle, GPUdouble

MODULE NAME

NUMERICS

DESCRIPTION

GPUplus(X, Y, R) is equivalent to plus(X, Y), but the result is returned in input parameter R.

Compilation supported

EXAMPLE

```
A = GPUsingle(rand(10));  
B = GPUsingle(rand(10));  
R = zeros(size(B), GPUsingle);  
GPUplus(A, B, R);
```

6.2.61 GPUpower

GPUpower - Array power

SYNTAX

GPUpower(X,Y,R)

X - GPUsingle, GPUdouble

Y - GPUsingle, GPUdouble

R - GPUsingle, GPUdouble

MODULE NAME

NUMERICS

DESCRIPTION

GPUpower(X, Y, R) is equivalent to power(X, Y), but the result is returned in input parameter R.

Compilation supported

6.2.62 GPUrdivide

GPUrdivide - Right array divide

SYNTAX

`GPUrdivide(X,Y)`

X - `GPUsingle`, `GPUdouble`

Y - `GPUsingle`, `GPUdouble`

R - `GPUsingle`, `GPUdouble`

MODULE NAME

NUMERICS

DESCRIPTION

`GPUrdivide(X, Y, R)` is equivalent to `rdivide(X, Y)`, but the result is returned in input parameter R.

Compilation supported

EXAMPLE

```
A = GPUsingle(rand(10));  
B = GPUsingle(rand(10));  
R = zeros(size(A), GPUsingle);  
GPUrdivide(A, B, R);
```

6.2.63 GPUround

GPUround - Round towards nearest integer

SYNTAX

GPUround(X, R)

X - GPUsingle, GPUdouble

R - GPUsingle, GPUdouble

MODULE NAME

NUMERICS

DESCRIPTION

GPUround(X, R) is equivalent to round(X), but the result is returned in input parameter R.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));  
R = zeros(size(X), GPUsingle);  
GPUround(X,R);
```

6.2.64 GPU`sin`

GPU`sin` - Sine of argument in radians

SYNTAX

`GPUsin(X, R)`

X - `GPUsingle`, `GPUdouble`

R - `GPUsingle`, `GPUdouble`

MODULE NAME

NUMERICS

DESCRIPTION

`GPUsin(X, R)` is equivalent to `sin(X)`, but the result is returned in input parameter R.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));  
R = zeros(size(X), GPUsingle);  
GPUsin(X,R)
```

6.2.65 GPUsinh

GPUsinh - Hyperbolic sine

SYNTAX

`GPUsinh(X, R)`

X - `GPUsingle`, `GPUdouble`

R - `GPUsingle`, `GPUdouble`

MODULE NAME

NUMERICS

DESCRIPTION

`GPUsinh(X, R)` is equivalent to `sinh(X)`, but the result is returned in input parameter R.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));  
R = zeros(size(X), GPUsingle);  
GPUsinh(X,R)
```

6.2.66 GPUsqrt

GPUsqrt - Square root

SYNTAX

`GPUsqrt(X,R)`

X - `GPUsingle`, `GPUdouble`

R - `GPUsingle`, `GPUdouble`

MODULE NAME

NUMERICS

DESCRIPTION

`GPUsqrt(X, R)` is equivalent to `sqrt(X)`, but the result is returned in input parameter R.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));  
R = zeros(size(X), GPUsingle);  
GPUsqrt(X,R)
```

6.2.67 GPUtan

GPUtan - Tangent of argument in radians

SYNTAX

`GPUtan(X,R)`

X - `GPUsingle`, `GPUdouble`

R - `GPUsingle`, `GPUdouble`

MODULE NAME

NUMERICS

DESCRIPTION

`GPUtan(X, R)` is equivalent to `tan(X)`, but the result is returned in input parameter R.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));  
R = zeros(size(X), GPUsingle);  
GPUtan(X,R)
```

6.2.68 GPUtanh

GPUtanh - Hyperbolic tangent

SYNTAX

`GPUtanh(X)`

X - `GPUsingle`, `GPUdouble`

R - `GPUsingle`, `GPUdouble`

MODULE NAME

NUMERICS

DESCRIPTION

`GPUtanh(X, R)` is equivalent to `tanh(X)`, but the result is returned in input parameter R.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));  
R = zeros(size(X), GPUsingle);  
GPUtanh(X, R)
```

6.2.69 GPUtimes

GPUtimes - Array multiply

SYNTAX

`GPUtimes(X,Y,R)`

X - `GPUsingle`, `GPUdouble`

Y - `GPUsingle`, `GPUdouble`

R - `GPUsingle`, `GPUdouble`

MODULE NAME

NUMERICS

DESCRIPTION

`GPUtimes(X, Y, R)` is equivalent to `times(X, Y)`, but the result is returned in input parameter R.

Compilation supported

EXAMPLE

```
A = GPUsingle(rand(10));
B = GPUsingle(rand(10));
R = zeros(size(A), GPUsingle);
GPUtimes(A, B, R);
```

6.2.70 GPUtranspose

GPUtranspose - Transpose

SYNTAX

```
GPUtranspose(X, R)  
X - GPUsingle, GPUdouble  
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

`GPUtranspose(X, R)` is equivalent to `transpose(X)`, but the result is returned in input parameter `R`.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));  
R = zeros(size(X), GPUsingle);  
GPUtranspose(X, R)
```

6.2.71 GPUuminus

GPUuminus - Unary minus

SYNTAX

`GPUuminus(X, R)`

X - `GPUsingle`, `GPUdouble`

R - `GPUsingle`, `GPUdouble`

MODULE NAME

NUMERICS

DESCRIPTION

`GPUuminus(X, R)` is equivalent to `uminus(X)`, but the result is returned in input parameter R.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));  
R = zeros(size(X), GPUsingle);  
GPUuminus(X, R)
```

6.2.72 GPUzeros

GPUzeros - GPU zeros array

SYNTAX

GPUzeros(R)

R - GPUsingle, GPUdouble

MODULE NAME

NUMERICS

DESCRIPTION

GPUzeros(R) sets to zero all the elements of R.

Compilation supported

EXAMPLE

```
A = GPUsingle(rand(5));  
GPUzeros(A)
```

6.2.73 **gt**

gt - Greater than

SYNTAX

```
R = X > Y
R = gt(X,Y)
X - GPUsingle, GPUdouble
Y - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

$A > B$ (`gt(A, B)`) does element by element comparisons between A and B.

Compilation supported

EXAMPLE

```
A = GPUsingle([1 2 0 4]);
B = GPUsingle([1 0 0 4]);
R = A > B;
single(R)
R = gt(A, B);
single(R)
```

6.2.74 `ifft`

`ifft` - Inverse discrete Fourier transform

SYNTAX

```
R = ifft(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

`IFFT(X)` is the inverse discrete Fourier transform of `X`.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(1,5)+i*rand(1,5));
R = fft(X);
X = ifft(R);
```

6.2.75 `ifft2`

ifft2 - Two-dimensional inverse discrete Fourier transform

SYNTAX

```
R = ifft2(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

IFFT2(F) returns the two-dimensional inverse Fourier transform of matrix F.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(5,5)+i*rand(5,5));
R = fft2(X);
X = ifft2(R);
```

6.2.76 **iscomplex**

iscomplex - True for complex array

SYNTAX

R = `iscomplex(X)`
X - GPU variable
R - logical (0 or 1)

MODULE NAME

NUMERICS

DESCRIPTION

`ISCOMPLEX(X)` returns 1 if X does have an imaginary part and 0 otherwise.

Compilation not supported

EXAMPLE

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

6.2.77 isempty

isempty - True for empty GPUsingle array

SYNTAX

R = isempty(X)
X - GPU variable
R - logical (0 or 1)

MODULE NAME

NUMERICS

DESCRIPTION

ISEMPTY(X) returns 1 if X is an empty GPUsingle array and 0 otherwise. An empty GPUsingle array has no elements, that is `prod(size(X))==0`.

Compilation not supported

EXAMPLE

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

6.2.78 **isreal**

isreal - True for real array

SYNTAX

R = `isreal(X)`
X - GPU variable
R - logical (0 or 1)

MODULE NAME

NUMERICS

DESCRIPTION

ISREAL(X) returns 1 if X does not have an imaginary part and 0 otherwise.

Compilation not supported

EXAMPLE

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

6.2.79 isscalar

isscalar - True if array is a scalar

SYNTAX

R = isscalar(X)
X - GPU variable
R - logical (0 or 1)

MODULE NAME

NUMERICS

DESCRIPTION

ISSCALAR(S) returns 1 if S is a 1x1 matrix and 0 otherwise.

Compilation not supported

EXAMPLE

```
A = GPUsingle(rand(5));  
isscalar(A)  
A = GPUsingle(1);  
isscalar(A)  
A = GPUdouble(1);  
isscalar(A)
```

6.2.80 ldivide

ldivide - Left array divide

SYNTAX

```
R = X .\ Y
R = ldivide(X,Y)
X - GPUsingle, GPUdouble
Y - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

$A.\backslash B$ denotes element-by-element division. A and B must have the same dimensions unless one is a scalar. A scalar can be divided with anything.

Compilation supported

EXAMPLE

```
A = GPUsingle(rand(10));
B = GPUsingle(rand(10));
R = A .\ B
A = GPUsingle(rand(10)+i*rand(10));
B = GPUsingle(rand(10)+i*rand(10));
R = A .\ B
```

6.2.81 le

le - Less than or equal

SYNTAX

```
R = X <= Y
R = le(X,Y)
X - GPUsingle, GPUdouble
Y - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

A <= B (le(A, B)) does element by element comparisons between A and B.

Compilation supported

EXAMPLE

```
A = GPUsingle([1 2 0 4]);
B = GPUsingle([1 0 0 4]);
R = A <= B;
single(R)
R = le(A, B);
single(R)
```

6.2.82 length

length - Length of vector

SYNTAX

R = length(X)
X - GPU variable

MODULE NAME

NUMERICS

DESCRIPTION

LENGTH(X) returns the length of vector X. It is equivalent to MAX(SIZE(X)) for non-empty arrays and 0 for empty ones.

Compilation not supported

EXAMPLE

```
A = GPUsingle(rand(5));  
length(A)
```

6.2.83 **log**

log - Natural logarithm

SYNTAX

```
R = log(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

LOG(X) is the natural logarithm of the elements of X. NaN results are produced if X is not positive.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));
R = log(X)
```

MATLAB COMPATIBILITY

Not implemented for complex X.

6.2.84 `log10`

`log10` - Common (base 10) logarithm

SYNTAX

```
R = log10(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

`LOG10(X)` is the base 10 logarithm of the elements of `X`. NaN results are produced if `X` is not positive.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));
R = log10(X)
```

MATLAB COMPATIBILITY

Not implemented for complex `X`.

6.2.85 `log1p`

log1p - Compute $\log(1+z)$ accurately

SYNTAX

```
R = log1p(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

LOG1P(Z) computes $\log(1+z)$. Only REAL values are accepted.
Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));
R = log1p(X)
```

MATLAB COMPATIBILITY

Not implemented for complex X.

6.2.86 `log2`

`log2` - Base 2 logarithm and dissect floating point number

SYNTAX

```
R = log2(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

`Y = LOG2(X)` is the base 2 logarithm of the elements of `X`.
Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));
R = log2(X)
```

MATLAB COMPATIBILITY

Not implemented for complex `X`.

6.2.87 It

It - Less than

SYNTAX

```
R = X < Y
R = lt(X,Y)
X - GPUsingle, GPUdouble
Y - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

$A < B$ (`lt(A, B)`) does element by element comparisons between A and B.

Compilation supported

EXAMPLE

```
A = GPUsingle([1 2 0 4]);
B = GPUsingle([1 0 0 4]);
R = A < B;
single(R)
R = lt(A, B);
single(R)
```

6.2.88 memCpyDtoD

memCpyDtoD - Device-Device memory copy

SYNTAX

```
memCpyDtoD(R, X, index, count)
```

R - GPUsingle, GPUdouble

X - GPUsingle, GPUdouble

MODULE NAME

NUMERICS

DESCRIPTION

`memCpyDtoD(R, X, index, count)` copies *count* elements from X to R(index)

Compilation supported

EXAMPLE

```
R = GPUsingle(rand(100,100));  
X = GPUsingle(rand(100,100));  
memCpyDtoD(R, X, 100, 20)
```

6.2.89 memCpyHtoD

memCpyHtoD - Host-Device memory copy

SYNTAX

`memCpyHtoD(R, X, index, count)`

R - GPUsingle, GPUdouble

X - Matlab array

MODULE NAME

NUMERICS

DESCRIPTION

`memCpyHtoD(R, X, index, count)` copies *count* elements from the Matlab variable X (CPU) to R(index)

Compilation supported

EXAMPLE

```
R = GPUsingle(rand(100,100));  
X = single(rand(100,100));  
memCpyHtoD(R, X, 100, 20)
```

6.2.90 minus

minus - Minus

SYNTAX

```
R = X - Y
R = minus(X,Y)
X - GPUsingle, GPUdouble
Y - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

$X - Y$ subtracts matrix Y from X . X and Y must have the same dimensions unless one is a scalar. A scalar can be subtracted from anything.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));
Y = GPUsingle(rand(10));
R = Y - X
X = GPUdouble(rand(10));
Y = GPUdouble(rand(10));
R = Y - X
```

6.2.91 **mrdivide**

mrdivide - Slash or right matrix divide

SYNTAX

$R = X / Y$

X - GPUsingle, GPUdouble

Y - GPUsingle, GPUdouble

R - GPUsingle, GPUdouble

MODULE NAME

NUMERICS

DESCRIPTION

Slash or right matrix divide.

Compilation supported

EXAMPLE

```
A = GPUsingle(rand(10));  
B = A / 5  
A = GPUdouble(rand(10));  
B = A / 5
```

MATLAB COMPATIBILITY

Supported only A / n where n is scalar.

6.2.92 **mtimes**

mtimes - Matrix multiply

SYNTAX

```
R = X * Y
R = mtimes(X,Y)
X - GPUsingle, GPUdouble
Y - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

* (mtimes(X, Y)) is the matrix product of X and Y.

Compilation supported

EXAMPLE

```
A = GPUsingle(rand(10));
B = GPUsingle(rand(10));
R = A * B
A = GPUdouble(rand(10));
B = GPUdouble(rand(10));
R = A * B
A = GPUsingle(rand(10)+i*rand(10));
B = GPUsingle(rand(10)+i*rand(10));
R = A * B
```

6.2.93 ndims

ndims - Number of dimensions

SYNTAX

R = ndims(X)
X - GPU variable

MODULE NAME

NUMERICS

DESCRIPTION

N = NDIMS(X) returns the number of dimensions in the array X. The number of dimensions in an array is always greater than or equal to 2. Trailing singleton dimensions are ignored. Put simply, it is LENGTH(SIZE(X)).

Compilation not supported

EXAMPLE

```
X = GPUsingle(rand(10));  
ndims(X)
```

6.2.94 ne

ne - Not equal

SYNTAX

```
R = X ~= Y
R = ne(X,Y)
X - GPUsingle, GPUdouble
Y - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

A ~= B (ne(A, B)) does element by element comparisons between A and B.

Compilation supported

EXAMPLE

```
A = GPUsingle([1 2 0 4]);
B = GPUsingle([1 0 0 4]);
R = A ~= B;
single(R)
R = ne(A, B);
single(R)
```

6.2.95 not

not - Logical NOT

SYNTAX

$R = \sim X$

X - GPUsingle, GPUdouble

R - GPUsingle, GPUdouble

MODULE NAME

NUMERICS

DESCRIPTION

$\sim A$ (`not(A)`) performs a logical NOT of input array A.

Compilation supported

EXAMPLE

```
A = GPUsingle([1 2 0 4]);  
R = ~A;  
single(R)
```

6.2.96 numel

numel - Number of elements in an array or subscripted array expression.

SYNTAX

R = numel(X)
X - GPU variable
R - number of elements

MODULE NAME

NUMERICS

DESCRIPTION

N = NUMEL(A) returns the number of elements N in array A.

Compilation not supported

EXAMPLE

```
X = GPUsingle(rand(10));  
numel(X)  
X = GPUdouble(rand(10));  
numel(X)
```

6.2.97 ones

ones - GPU ones array

SYNTAX

```
ones(N,GPUsingle)
ones(M,N,GPUsingle)
ones([M,N],GPUsingle)
ones(M,N,P,...?,GPUsingle)
ones([M N P ...],GPUsingle)
```

```
ones(N,GPUdouble)
ones(M,N,GPUdouble)
ones([M,N],GPUdouble)
ones(M,N,P,...,GPUdouble)
ones([M N P ...],GPUdouble)
```

MODULE NAME

NUMERICS

DESCRIPTION

`ones(N,GPUsingle)` is an N-by-N GPU matrix of ones.

`ones(M,N,GPUsingle)` or `ones([M,N],GPUsingle)` is an M-by-N GPU matrix of ones.

`ones(M,N,P,...,GPUsingle)` or `ones([M N P ...],GPUsingle)` is an M-by-N-by-P-by-... GPU array of ones.

`ones(M,N,P,...,GPUdouble)` or `ones([M N P ...],GPUdouble)` is an M-by-N-by-P-by-... GPU array of ones.

Compilation supported

EXAMPLE

```
A = ones(10,GPUsingle)
B = ones(10, 10,GPUsingle)
C = ones([10 10],GPUsingle)
A = ones(10,GPUdouble)
B = ones(10, 10,GPUdouble)
C = ones([10 10],GPUdouble)
```

6.2.98 or

or - Logical OR

SYNTAX

```
R = X | Y
R = or(X,Y)
X - GPUsingle, GPUdouble
Y - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

A | B (or(A, B)) performs a logical OR of arrays A and B.
Compilation supported

EXAMPLE

```
A = GPUsingle([1 2 0 4]);
B = GPUsingle([1 0 0 4]);
R = A | B;
single(R)
R = or(A, B);
single(R)
```

6.2.99 permute

permute - Permute array dimensions

SYNTAX

```
R = permute(X, ORDER)
X - GPUsingle, GPUdouble
Y - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

R = PERMUTE(X,ORDER) rearranges the dimensions of X so that they are in the order specified by the vector ORDER.

Compilation supported

EXAMPLE

```
A = GPUsingle(rand(3,4,5));
B = permute(A,[3 2 1]);
```

6.2.100 plus

plus - Plus

SYNTAX

```
R = X + Y
R = plus(X,Y)
X - GPUsingle, GPUdouble
Y - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

$X + Y$ (`plus(X, Y)`) adds matrices X and Y . X and Y must have the same dimensions unless one is a scalar (a 1-by-1 matrix). A scalar can be added to anything.

Compilation supported

EXAMPLE

```
A = GPUsingle(rand(10));
B = GPUsingle(rand(10));
R = A + B
A = GPUsingle(rand(10)+i*rand(10));
B = GPUsingle(rand(10)+i*rand(10));
R = A + B
```

6.2.101 power

power - Array power

SYNTAX

```
R = X .^ Y
R = power(X,Y)
X - GPUsingle, GPUdouble
Y - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

$Z = X.^Y$ denotes element-by-element powers.

Compilation supported

EXAMPLE

```
A = GPUsingle(rand(10));
B = 2;
R = A .^ B
A = GPUsingle(rand(10)+i*rand(10));
R = A .^ B
```

MATLAB COMPATIBILITY

Implemented for REAL exponents only.

6.2.102 **rdivide**

rdivide - Right array divide

SYNTAX

```
R = X ./ Y
R = rdivide(X,Y)
X - GPUsingle, GPUdouble
Y - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

A./B denotes element-by-element division. A and B must have the same dimensions unless one is a scalar. A scalar can be divided with anything.

Compilation supported

EXAMPLE

```
A = GPUsingle(rand(10));
B = GPUsingle(rand(10));
R = A ./ B
A = GPUsingle(rand(10)+i*rand(10));
B = GPUsingle(rand(10)+i*rand(10));
R = A ./ B
```

6.2.103 repmat

repmat - Replicate and tile an array

SYNTAX

```
R = repmat(X,M,N)
R = REPMAT(X,[M N])
R = REPMAT(X,[M N P ...])
R - GPUsingle, GPUdouble
X - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

`R = repmat(X,M,N)` creates a large matrix `R` consisting of an `M`-by-`N` tiling of copies of `X`. The statement `repmat(X,N)` creates an `N`-by-`N` tiling.

Compilation supported

EXAMPLE

```
A = GPUsingle(rand(10));
repmat(A,3,4,5)
```

6.2.104 round

round - Round towards nearest integer

SYNTAX

```
R = round(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

ROUND(X) rounds the elements of X to the nearest integers.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));
R = round(X)
X = GPUdouble(rand(10));
R = round(X)
```

MATLAB COMPATIBILITY

Not implemented for complex X.

6.2.105 `sin`

`sin` - Sine of argument in radians

SYNTAX

`R = sin(X)`

`X` - `GPUsingle`, `GPUdouble`

`R` - `GPUsingle`, `GPUdouble`

MODULE NAME

NUMERICS

DESCRIPTION

`SIN(X)` is the sine of the elements of `X`.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));  
R = sin(X)  
X = GPUdouble(rand(10));  
R = sin(X)
```

MATLAB COMPATIBILITY

Not implemented for complex `X`.

6.2.106 `sinh`

`sinh` - Hyperbolic sine

SYNTAX

```
R = sinh(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

`SINH(X)` is the hyperbolic sine of the elements of `X`.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));
R = sinh(X)
X = GPUdouble(rand(10));
R = sinh(X)
```

MATLAB COMPATIBILITY

Not implemented for complex `X`.

6.2.107 size

size - Size of array

SYNTAX

```
R = size(X)
[M,N] = SIZE(X)
[M1,M2,...,MN] = SIZE(X)
X - GPU variable
```

MODULE NAME

NUMERICS

DESCRIPTION

$D = \text{SIZE}(X)$, for M-by-N matrix X, returns the two-element row vector $D = [M,N]$ containing the number of rows and columns in the matrix.

Compilation not supported

EXAMPLE

```
X = GPUsingle(rand(10));
size(X)
X = GPUDouble(rand(10));
size(X)
```

CHAPTER 6. Function Reference
6.2. FUNCTIONS - ALPHABETICAL LIST

6.2.108 slice

slice - Subscripted reference

SYNTAX

```
R = slice(X, R1, R2, ..., RN)
X - GPUsingle, GPUdouble
R1, R2, ..., RN - Range
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

`slice(X, R1, ..., RN)` is an array formed from the elements of `X` specified by the ranges `R1, R2, RN`. A range can be constructed as follows:

`[inf, stride, sup]` - defines a range between `inf` and `sup` with specified stride. It is similar to the Matlab syntax `A(inf:stride:sup)`. The special keyword `END` (please note, uppercase `END`) can be used.

`':'` - similar to the colon used in Matlab indexing.

`{[i1, i2, ..., in]}` -any array enclosed by brackets is considered an indexes array, similar to `A([1 2 3 4 1 2])` in Matlab.

`i1` - a single value is interpreted as an index. Similar to `A(10)` in Matlab.

Compilation supported

EXAMPLE

```
Bh = single(rand(100));
B = GPUsingle(Bh);
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, ':'', ':'');
```

6.2.109 `sqrt`

`sqrt` - Square root

SYNTAX

```
R = sqrt(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

`SQRT(X)` is the square root of the elements of `X`. NaN results are produced if `X` is not positive.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));
R = sqrt(X)
```

MATLAB COMPATIBILITY

Not implemented for complex `X`.

6.2.110 subsref

subsref - Subscripted reference

SYNTAX

```
R = X(I)
X - GPUsingle, GPUdouble
I - GPUsingle, GPUdouble, Matlab range
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

`A(I)` (`subsref`) is an array formed from the elements of `A` specified by the subscript vector `I`. The resulting array is the same size as `I` except for the special case where `A` and `I` are both vectors. In this case, `A(I)` has the same number of elements as `I` but has the orientation of `A`.

Compilation not supported

EXAMPLE

```
A = GPUsingle([1 2 3 4 5]);
A = GPUdouble([1 2 3 4 5]);
idx = GPUsingle([1 2]);
B = A(idx)
```

6.2.111 tan

tan - Tangent of argument in radians

SYNTAX

```
R = tan(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

TAN(X) is the tangent of the elements of X.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));
R = tan(X)
X = GPUdouble(rand(10));
R = tan(X)
```

MATLAB COMPATIBILITY

Not implemented for complex X.

6.2.112 tanh

tanh - Hyperbolic tangent

SYNTAX

```
R = tanh(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

TANH(X) is the hyperbolic tangent of the elements of X.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));
R = tanh(X)
X = GPUdouble(rand(10));
R = tanh(X)
```

MATLAB COMPATIBILITY

Not implemented for complex X.

6.2.113 times

times - Array multiply

SYNTAX

```
R = X .* Y
R = times(X,Y)
X - GPUsingle, GPUdouble
Y - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

`X.*Y` denotes element-by-element multiplication. `X` and `Y` must have the same dimensions unless one is a scalar. A scalar can be multiplied into anything.

Compilation supported

EXAMPLE

```
A = GPUsingle(rand(10));
B = GPUsingle(rand(10));
R = A .* B
A = GPUsingle(rand(10)+i*rand(10));
B = GPUsingle(rand(10)+i*rand(10));
R = A .* B
A = GPUdouble(rand(10)+i*rand(10));
B = GPUdouble(rand(10)+i*rand(10));
R = A .* B
```

6.2.114 transpose

transpose - Transpose

SYNTAX

```
R = X.'
R = transpose(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

`X.'` or `transpose(X)` is the non-conjugate transpose.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));
X = GPUdouble(rand(10));
R = X.'
R = transpose(X)
```

6.2.115 `uminus`

`uminus` - Unary minus

SYNTAX

```
R = -X
R = uminus(X)
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

-A negates the elements of A.

Compilation supported

EXAMPLE

```
X = GPUsingle(rand(10));
R = -X
R = uminus(X)
X = GPUdouble(rand(10));
R = -X
R = uminus(X)
```

CHAPTER 6. Function Reference
6.2. FUNCTIONS - ALPHABETICAL LIST

6.2.116 zeros

zeros - GPU zeros array

SYNTAX

```
zeros(N,GPUsingle)
zeros(M,N,GPUsingle)
zeros([M,N],GPUsingle)
zeros(M,N,P,...?,GPUsingle)
zeros([M N P ...],GPUsingle)
```

```
zeros(N,GPUdouble)
zeros(M,N,GPUdouble)
zeros([M,N],GPUdouble)
zeros(M,N,P,...?,GPUdouble)
zeros([M N P ...],GPUdouble)
```

MODULE NAME

NUMERICS

DESCRIPTION

`zeros(N,GPUsingle)` is an N-by-N GPU matrix of zeros.

`zeros(M,N,GPUsingle)` or `zeros([M,N],GPUsingle)` is an M-by-N GPU matrix of single precision zeros.

`zeros(M,N,P,...,GPUsingle)` or `zeros([M N P ...],GPUsingle)` is an M-by-N-by-P-by-... GPU array of single precision zeros.

`zeros(M,N,P,...,GPUdouble)` or `zeros([M N P ...],GPUdouble)` is an M-by-N-by-P-by-... GPU array of double precision zeros.

Compilation supported

EXAMPLE

```
A = zeros(10,GPUsingle)
B = zeros(10, 10,GPUsingle)
C = zeros([10 10],GPUsingle)
```

```
A = zeros(10,GPUdouble)
B = zeros(10, 10,GPUdouble)
C = zeros([10 10],GPUdouble)
```

6.3 Examples - alphabetical list

6.3.1 forloop1

forloop1 - Code optimization example

SYNTAX

```
forloop1(A, B, C)
```

A, B, C - GPU variable

MODULE NAME

EXAMPLES:CODEOPT

DESCRIPTION

forloop1(A, B, C) is an example of code optimization using low level GPUmat functions

Compilation not supported

6.3.2 `gputype_create1`

`gputype_create1` - GPUtype creation example

SYNTAX

```
gputype_create1(T)  
T - Matlab value
```

MODULE NAME

EXAMPLES:GPATYPE

DESCRIPTION

`gputype_create1(T)` creates a GPUtype variable of type T. Depending on the value of T, a single, double, real or complex GPUtype is created.

Compilation not supported

EXAMPLE

```
%% Create GPUtype  
  
% single/real  
R = gputype_create1(0);  
  
% single/complex  
R = gputype_create1(1);  
  
if (GPUisDoublePrecision)  
    % double/real  
    R = gputype_create1(2);  
  
    % double/complex  
    R = gputype_create1(3);  
end
```

6.3.3 `gputype_create2`

`gputype_create2` - GPUtype creation example

SYNTAX

`gputype_create2(Ah)`
Ah - Matlab array

MODULE NAME

EXAMPLES:GPATYPE

DESCRIPTION

`gputype_create2(Ah)` creates a GPUtype variable from the Matlab array Ah.

Compilation not supported

EXAMPLE

```
% Create a GPUtype from a Matlab array
if (GPUisDoublePrecision)
    Ah = rand(100);
    A = gputype_create2(Ah);
end

Ah = single(rand(100));
A = gputype_create2(Ah);
```

6.3.4 gputype_properties

gputype_properties - GPUtype properties example

SYNTAX

gputype_properties(X)
X - GPU variable

MODULE NAME

EXAMPLES:GPATYPE

DESCRIPTION

gputype_properties(X) displays properties of the GPUtype X
Compilation not supported

EXAMPLE

```
% single/real
A = GPUsingle(rand(2,2,2,2));
gputype_properties(A);

% single/complex
A = complex(A);
gputype_properties(A);

% double/real
if (GPUisDoublePrecision)
    A = GPUdouble(rand(2,2,2,2));
    gputype_properties(A);

% double/complex
A = complex(A);
gputype_properties(A);
end
```

6.3.5 myexp

myexp - GPUmat internal functions example

SYNTAX

myexp(P, R)
P, R - GPU variable

MODULE NAME

EXAMPLES:NUMERICS

DESCRIPTION

myexp(P, R) calculates the exponential of P and stores the result in R.

Compilation not supported

6.3.6 myplus

myplus - GPUmat internal functions example

SYNTAX

myplus(P, Q, R)
P, Q, R - GPU variable

MODULE NAME

EXAMPLES:NUMERICS

DESCRIPTION

myplus(P, Q, R) adds P and Q and stores the result in R

Compilation not supported

6.3.7 `myslice1`

`myslice1` - GPUmat internal functions example

SYNTAX

```
myslice1(P, R, i0, i1, i2, i3, i4, i5, i6, i7, i8)
```

P, R - GPU variable

i0,...,i8 - Matlab value (index)

MODULE NAME

EXAMPLES:NUMERICS

DESCRIPTION

`myslice1(P, R, i0, i1, i2, i3, i4, i5, i6, i7, i8)` is an example of the `slice` GPUmat internal function

Compilation not supported

6.3.8 `myslice2`

`myslice2` - GPUmat internal functions example

SYNTAX

```
R = myslice2(P, i0, i1, i2, i3, i4, i5, i6, i7, i8)
```

P, R - GPU variable

i0,...,i8 - Matlab value (index)

MODULE NAME

EXAMPLES:NUMERICS

DESCRIPTION

`R = myslice2(P, i0, i1, i2, i3, i4, i5, i6, i7, i8)` is an example of the `slice` GPUmat internal function

Compilation not supported

6.3.9 mytimes

mytimes - GPUmat internal functions example

SYNTAX

`mytimes(P, Q, R)`

P, Q, R - GPU variable

MODULE NAME

EXAMPLES:NUMERICS

DESCRIPTION

`mytimes(P, Q, R)` is the element-wise multiplication between P and Q. Result is stored in R.

Compilation not supported