GPUmat User Guide

Version 0.28, January 2012

Contents

CONTENTS

CONTENTS

CONTENTS

CONTENTS

CONTENTS

CONTENTS

CONTENTS

Chapter 1
Introduction

GPUmat enables *Matlab* code to run on the Graphical Processing Unit (GPU). The following is a summary of *GPUmat* most important features:

- GPU computational power can be easily accessed from *Matlab* without any GPU knowledge.
- *Matlab* code is directly executed on the GPU. The execution is transparent to the user.
- *GPUmat* speeds up *Matlab* functions by using the GPU multi-processor architecture.
- Existing *Matlab* code can be ported and executed on GPUs with few modifications.
- GPU resources are accessed using *Matlab* scripting language. The fast code prototyping capability of the scripting language is combined with the fast code execution on the GPU.
- *GPUmat* can be used as a Source Development Kit to create new functions and extend the library functionality.
- GPU operations can be easily recorded into new functions using the *GPUmat* compiler.

1.1 About GPUs

Although GPUs have been traditionally used only for computer graphics, a recent technique called GPGPU (General-purpose computing on graphics processing units) allows the GPUs to perform numerical computations usually handled by CPU. The advantage of using GPUs for general purpose computation is the performance speed up that can be achieved due to the parallel architecture of these devices.

One of the most promising GPGPU technologies is called CUDA SDK [1], developed by NVIDIA. For further information about CUDA, GPGPU and related topics please check [2] [3].

1.2 About GPUmat

GPUmat started as a Freeware project developed by the GP-you Group (`http://gp-you.org`). Starting from version 0.22 part of the project was released as open source:

- *GPUmat User Modules* on Sourceforge (`http://sourceforge.net/projects/gpumatmodules`).
- *matCUDA* on Sourceforge (`http://sourceforge.net/projects/matcuda`).

Starting from version 0.280, *GPUmat* is distributed as open source and external packages *GPUmat User Modules* and *matCUDA* are combined again into a single project called *GPUmat* on Sourceforge (`http://sourceforge.net/projects/gpumat`).

The *GPUmat User Modules* project allows the user to access *GPUmat* internal functions directly from a mex file and to add to *GPUmat* a user implemented GPU kernel. Documentation for this project can be found in the *GPUmat* installation folder, on the old Sourceforge Wiki page (`http://sourceforge.net/apps/mediawiki/gpumatmodules`) and on the latest Sourceforge Wiki page (`http://sourceforge.net/apps/mediawiki/gpumat`). Some examples can be found in the *GPUmat* installation folder *modules*.

The *matCUDA* project is a collection of *Matlab* wrappers to CUDA CUBLAS and CUFFT libraries. Documentation can be found in the *GPUmat* installation folder, on the old Sourceforge Wiki page (`http://sourceforge.net/apps/mediawiki/matcuda`) and on the latest Sourceforge Wiki page (`http://sourceforge.net/apps/mediawiki/gpumat`).

The initial web site for *GPUmat* will be still available (`http://gp-you.org`), but the main project web site will be on Sourceforge (`http://sourceforge.net/apps/mediawiki/gpumat`).

1.3 System requirements

GPUmat was tested under Windows and Linux with Matlab ver. R2007a or newer installed. CUDA should be installed on the system. Follow the instructions on NVIDIA's CUDA website [2] to download and install the software.

1.4 Credits and license

Starting from version 0.280, *GPUmat* is distributed as open source software (please check license.txt for more information). Versions prior to 0.280 were developed and distributed by the GP-you Group as Freeware.

1.5 How to install

To install *GPUmat* unpack the downloaded package and follow these steps:

- **STEP0** (Windows): Microsoft Visual C++ 2008 Redistributable Package installation. This package is required only on Windows. You might have this package already installed. Try to run *GPUmat* by following steps *STEP1* to *STEP3*. If it fails, install the C++ Redistributable by running the executable (*vcredist_x86.exe*, or *vcredist_x64.exe* depending on the architecture) that you find in the *etc* folder in the *GPUmat* installation package.
- **STEP1**: start *Matlab* and change directory to the folder where the library was unpacked.
- **STEP2**: start *GPUmat* using the **GPUstart** command.
- **STEP3** (optional but suggested): add the library path to the *Matlab* path by using the `"File->Set Path"` menu. The *Matlab* documentation describes how to add a new path. This step is not mandatory if the **GPUstart** command is started from the directory where the library was unpacked.

The **GPUstart** command should generate the following output in your *Matlab* command window:

```
>> GPUstart
Starting GPU
There is 1 device supporting CUDA
CUDA Driver Version:                              2.30
CUDA Runtime Version:                             2.30

Device 0: "GeForce GTX 275"
  CUDA Capability Major revision number:          1
  CUDA Capability Minor revision number:          3
  Total amount of global memory:                  939196416 bytes
  Number of multiprocessors:                      30
  Number of cores:                                240
  - CUDA compute capability 1.3
...done
- Loading module EXAMPLES_CODEOPT
- Loading module EXAMPLES_NUMERICS
  -> numerics13.cubin
- Loading module NUMERICS
  -> numerics13.cubin
```

If you get some error, make sure that *GPUmat* is in the *Matlab* path, or run the diagnostic command

```
>> GPUmatSystemCheck
```

The above command generates a report about the system configuration:

```
*** GPUmat system diagnostics
* Running on          -> "win32"
* GPUmat version      -> 0.21
* GPUmat build        -> 23-Oct-2009
* GPUmat architecture  -> "win32"

*** ARCHITECTURE TEST
*** GPUmat architecture test -> passed.

*** CUDA TEST
*** CUDA CUBLAS -> installed.
*** CUDA CUFFT  -> installed.
*** CUDA CUDART -> installed.
```

```
...
```

On Windows it is also necessary to have the Microsoft Visual C++ 2008 Redistributable Package installed. *GPUstart* generates an error if this package is not installed.

The GPU environment will not correctly work if a CUDA compatible graphic card and CUDA toolkit are not installed on the system.

1.6 How to compile

Please check the *Developer Guide* for more information about the compilation of the software.

1.7 Terminology

The following is a summary of common terms and concepts used in this manual:

- GPU: Graphics Processing Unit. It is the graphic card. We assume that the GPU is compatible with NVIDIA's CUDA SDK.
- HOST: The computer where the GPU is installed.
- CPU: The Central Processing Unit installed on the HOST.
- GPU memory: the memory available on the GPU.
- CPU memory: the memory available on the HOST.
- CUDA capable GPU: a GPU compatible with NVIDIA CUDA SDK.

1.8 Documentation overview

This manual is organized as follows:

- Quick start: describes *GPUmat* basic concepts by using simple examples.
- Overview: describes *GPUmat* high level functions.

- *GPUmat* compiler: describes how to record new functions using the *GPUmat* compiler.

The first two chapters contain enough information to understand the basic concepts of the library and are intended for users with at least some experience with *Matlab*. The *Function reference* can be found in Chapter 5.

Chapter 2
Quick start

The most important concepts about *GPUmat* are the following:

- GPUmat defines the following GPU variables (or classes): *i*) *GPUsingle*, *ii*) *GPUdouble*. They correspond to single and double precision floating point variables respectively. We will refer to these variables as GPU variables, because although they are available from *Matlab* workspace as any other *Matlab* variable, they are allocated on the GPU memory. *Matlab* variables are allocated on CPU memory.
- *GPUmat* defines functions and operators that are called from *Matlab* and executed on the GPU. These functions work with *GPUsingle* or *GPUdouble* classes.

The next example creates two single precision *Matlab* variables *Ah* and *A*, allocated on the CPU memory and on the GPU memory respectively. *Ah* is used to initialize *A*.

```
Ah = single(rand(100,100)); % Ah in on CPU memory
A  = GPUsingle(Ah);         % A  is on GPU memory
```

In the above code the function **single** in used to create the single precision *Matlab* array *Ah*, and similarly the *GPUsingle* function is used to create a single precision GPU variable. Although is always possible to use *GPUsingle* or *GPUdouble* to create a GPU variable, these functions perform a memory transfer from CPU memory to GPU memory (they copy the content of the CPU array to the GPU memory). It is faster if the GPU array is directly created on the GPU memory. For example, it is possible to directly use the function *rand* as follows:

```
% Ah in on CPU memory
Ah = single(rand(100,100));
% A  is directly created on GPU memory
A  = rand(100,100,GPUsingle);
```

In the above code, there is no memory transfer between CPU and GPU. In a similar way we can create two double precision *Matlab* variables *Bh* and *B*, as follows:

```
if GPUisDoublePrecision
  Bh = rand(100,100); % Bh in on CPU memory
  B  = GPUdouble(Bh); % B  is on GPU memory
end
```

The optimized version of the above code without CPU to GPU memory transfer is the following:

```
if GPUisDoublePrecision
  Bh = rand(100,100);           % Bh in on CPU memory
  B  = rand(100,100,GPUdouble); % B  is on GPU memory
end
```

If a double precision *Matlab* array is used to initialize a *GPUsingle* variable, it is converted to a single precision variable resulting in a loss of precision:

```
Ah = rand(100,100); % Ah in on CPU memory, double precision
A  = GPUsingle(Ah); % A is on GPU memory, single precision
```

During the initialization of the GPU variable *A*, the data in the *Matlab* array *Ah* is copied from the CPU memory to the GPU memory. The data transfer is transparent to the user.

There are several ways to create a GPU variable, as explained in Section 3.2. The command

```
A  = colon(0,2,6,GPUsingle) % A is on GPU memory
if GPUisDoublePrecision
  B  = colon(0,2,6,GPUdouble) % B is on GPU memory
end
```

results in

```
A  =
     0 2 4 6

B  =
     0 2 4 6
```

Using the *colon* function to create a vector with arbitrary real increments between the elements,

```
A = colon(0,.1,.5,GPUsingle) % A is on GPU memory
```

results in

```
A =
    0    0.1000    0.2000    0.3000    0.4000    0.5000
```

In the following example, the function **single** is used to convert the GPU variable *A* into the *Matlab* variable *Ch*, while the function **double** is used to convert the double precision GPU variable *B* into the double precision *Matlab Dh*. Every time a GPU variable is converted into a *Matlab* variable, the data is copied from GPU memory to CPU memory.

```
Ah = single(rand(100,100));   % Ah in on CPU memory
A  = GPUsingle(Ah);           % Create GPU variable A

% The following creates the same variable A without
% CPU to GPU memory transfer
A  = rand(100,100,GPUsingle); % Create GPU variable A
Ch = single(A);               % convert A (GPU) to Ch (CPU)

if GPUisDoublePrecision
  Bh = rand(100,100);         % Bh in on CPU memory
  B  = GPUdouble(Bh);         % Create GPU variable B
% The following creates the same variable A without
% CPU to GPU memory transfer
  B = rand(100,100,GPUdouble);% Create GPU variable B
  Dh = double(B);             % convert B (GPU) to Dh (CPU)
end
```

The following example shows:

- The creation of the GPU variable A, initialized with *Matlab* array Ah.
- The calculation of `exp(A)`. The execution is on GPU and the result is stored on the GPU variable C.
- The conversion of the result C into the *Matlab* variable Ch.

```
Ah = single(rand(100,100)); % Ah in on CPU memory
A  = GPUsingle(Ah); % Create A (GPU)  initialized with  Ah (CPU)
C  = exp(A);        % exp(A) performed on GPU
Ch = single(C);     % convert C (GPU) to Ch (CPU)
```

The above example without CPU to GPU memory transfer is the following:

```
Ah = single(rand(100,100));   % Ah in on CPU memory
A  = rand(100,100,GPUsingle); % Create A (GPU)
C  = exp(A);        % exp(A) performed on GPU
Ch = single(C);     % convert C (GPU) to Ch (CPU)
```

Please note that in the above code Ah and A are different. The previous example in double precision is the following:

```
if GPUisDoublePrecision
  Ah = rand(100,100); % Ah in on CPU memory
  A  = GPUdouble(Ah); % Create A (GPU) initialized with Ah (CPU)
  C  = exp(A);        % exp(A) performed on GPU
  Ch = double(C);     % convert C (GPU) to Ch (CPU)
end
```

To visualize the contents of a GPU variable, type the name of the variable on the *Matlab* command window:

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

```
A

ans =
```

```
    0.8147    0.0975    0.1576    0.1419    0.6557
    0.9058    0.2785    0.9706    0.4218    0.0357
    0.1270    0.5469    0.9572    0.9157    0.8491
    0.9134    0.9575    0.4854    0.7922    0.9340
    0.6324    0.9649    0.8003    0.9595    0.6787

Single precision REAL GPU type.
```

Next sections show different examples: matrix addition, matrix multiplication and FFT calculation.

2.1 Matrix addition example

The following code can be found in the *QuickStart.m* file located in the examples folder, and it shows how to port existing Matlab code and run it on the GPU. The example creates two variables A and B, add them and store the result into the variable C. The original Matlab code is the following:

```
A = single(rand(100)); % A is on CPU memory
B = single(rand(100)); % B is on CPU memory
C = A+B; % executed on CPU. C is on CPU memory
```

The above code in double precision is the following:

```
A = rand(100); % A is on CPU memory
B = rand(100); % B is on CPU memory
C = A+B; % executed on CPU. C is on CPU memory
```

The ported *GPUmat* code (single and double precision) is the following:

```
%% single precision
A = rand(100,GPUsingle); % A is on GPU memory
B = rand(100,GPUsingle); % B is on GPU memory
C = A+B; % executed on GPU. C is on GPU memory

%% double precision
if GPUisDoublePrecision
  A = rand(100,GPUdouble); % A is on GPU memory
  B = rand(100,GPUdouble); % B is on GPU memory
```

```
  C = A+B; % executed on GPU. C is on GPU memory
end
```

Please note the difference between the original code and the modified code. Every Matlab variable has been converted to the *GPUsingle* or *GPUdouble* class: "A = rand(100)" becomes "A = rand(100,GPUsingle)".

Any operation on *GPUsingle* variables generates a *GPUsingle*, i.e. *C* (in the modified code) is also a *GPUsingle*. Functions involving *GPUsingle* variables, like A + B in the above example, are executed on the GPU. To convert the GPU variables *A*, *B* and *C* into the *Matlab* variables *Ah*, *Bh* and *Ch* use the functions **single** and **double**, as follows:

```
%% single precision
A = rand(100,GPUsingle); % A is on GPU memory
B = rand(100,GPUsingle); % B is on GPU memory
C = A+B; % executed on GPU. C is on GPU memory

Ah = single(A); %Ah is on HOST, A is on GPU
Bh = single(B); %Bh is on HOST, B is on GPU
Ch = single(C); %Ch is on HOST, C is on GPU

%% double precision
if GPUisDoublePrecision
  A = rand(100,GPUdouble); % A is on GPU memory
  B = rand(100,GPUdouble); % B is on GPU memory
  C = A+B; % executed on GPU. C is on GPU memory

  Ah = double(A); %Ah is on HOST, A is on GPU
  Bh = double(B); %Bh is on HOST, B is on GPU
  Ch = double(C); %Ch is on HOST, C is on GPU
end
```

The following code shows a different way to initialize the arrays *A* and *B* by using the **colon** function. The original Matlab code is the following:

```
A = single(colon(0,1,1000)); % A is on CPU memory
B = single(colon(0,1,1000)); % B is on CPU memory
C = A+B; % executed on CPU. C is on CPU memory
```

The ported *GPUmat* code is the following:

```
A = colon(0,1,1000,GPUsingle); % A is on GPU memory
B = colon(0,1,1000,GPUsingle); % B is on GPU memory
C = A+B; % executed on GPU. C is on GPU memory
```

The *Matlab* expression

```
A = single(colon(0,1,1000));
```

is equivalent to

```
A = single([0:1:1000]);
```

and creates a vector with single precision elements having values from 0 to 1000. Scalars are automatically converted to GPU variables, as follows:

```
A = rand(100,GPUsingle); % A is on GPU memory
C = A+1; % executed on GPU. C is on GPU memory

% equivalent to
C = A+GPUsingle(1);
```

In the above example, the *Matlab* scalar can be converted to a GPU variable using *GPUsingle*, but this is not necessary because the conversion is automatically done in *GPUmat*. Automatic casting between GPU and *Matlab* for non scalar variables is not done automatically. The following code generates an error:

```
A = colon(0,1,1000,GPUsingle); % A is on GPU memory
B = colon(0,1,1000);           % B is on CPU memory
C = A+B; % ERROR
```

Element-by-element operations, such as the the matrix addition `A + B`, are highly optimized for the GPU. It is suggested to use this kind of operations as explained in Section 3.11.

2.2 Matrix multiplication example

This section describes the code to perform the following tasks:

- Create A and B on the GPU memory.

- Multiply A and B and store the results in C.
- Convert the result C into the *Matlab* variable Ch.

```
A = rand(100,100,GPUsingle); % A is on GPU memory
B = rand(100,100,GPUsingle); % B is on GPU memory
C  = A*B;         % executed  on GPU, C is on GPU memory
Ch  = single(C);  % Ch is on CPU memory
```

The equivalent code on the CPU is the following:

```
A = single(rand(100,100)); % A is on CPU memory
B = single(rand(100,100)); % B is on CPU memory
C  = A*B;         % executed  on CPU, C is on CPU memory
```

2.3 FFT calculation example

This section describes the code to perform the following tasks:

- Create two arrays A and B on the GPU.
- Calculate 1D FFT of A.
- Calculate 2D FFT of B.
- Transfer results from GPU into *Matlab* variables Ah and Bh.

```
A = rand(1,100,GPUsingle);   % GPU
B = rand(100,100,GPUsingle); % GPU

%% 1D FFT
FFT_A = fft(A);  % executed on GPU

%% 2D FFT
FFT_B = fft2(B); % executed on GPU

%% Convert GPU into Matlab variables
Ah = single(A); % Ah is on HOST
Bh = single(B); % Bh is on HOST
```

```
FFT_Ah = single(FFT_A); % FFT_Ah is on HOST
FFT_Bh = single(FFT_B); % FFT_Bh is on HOST
```

The equivalent code that executes above operations entirely on the CPU is the following:

```
A = single(rand(1,100));   % CPU
B = single(rand(100,100)); % CPU

%% 1D FFT
FFT_A = fft(A);  % executed on CPU

%% 2D FFT
FFT_B = fft2(B); % executed on CPU
```

2.4 GPUmat compiler

The *GPUmat* compiler is used to record GPU operations into a new function. The compiled function is optimized and faster than the non compiled code. Moreover, the *GPUmat* compiler can be used to optimize for-loops, as shown in the *GPUmatCompiler.m* file located in the *GPUmat example* folder.

2.5 Variable assignment

Variable assignment in *GPUmat* is different from *Matlab*. For example, the following commands create in *Matlab* two arrays A and B, and B is assigned to A:

```
A = rand(3); % CPU
B = rand(3); % CPU
A = B;
```

In the above example, A and B have the same values but are distinct variables. It means that the following statement has effect only on A:

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

```
B
```

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

A =

   10.0000    0.7379    0.7817
    0.4959    0.3107    0.1115
    0.9885    0.6004    0.5793


B =

    0.0068    0.7379    0.7817
    0.4959    0.3107    0.1115
    0.9885    0.6004    0.5793
```

The above commands have a different behavior in *GPUmat*. If a *GPUmat* variable B is assigned to a *GPUmat* variable A, then the two objects are exactly the same. It means that the following command has effects on both A and B:

```
A = rand(3,GPUsingle); % GPU
B = rand(3,GPUsingle); % GPU
A = B;
```

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

ans =

   10.0000    0.0946    0.3821
    0.3778    0.9091    0.6603
```

```
    0.5180    0.2076    0.7584

Single precision REAL GPU type.

ans =

   10.0000    0.0946    0.3821
    0.3778    0.9091    0.6603
    0.5180    0.2076    0.7584

Single precision REAL GPU type.
```

To assign to A the *GPUmat* variable B, the *clone* command must be used, as follows:

```
A = rand(3,GPUsingle); % GPU
B = rand(3,GPUsingle); % GPU
A = clone(B);
```

2.6 Performance analysis

The easiest way to evaluate the performance in *Matlab* are the **tic** and **toc** commands, as follows:

```
A = rand(1000,1000); % A is on CPU
B = rand(1000,1000); % B is on CPU
tic;A.*B;toc;  % executed on CPU
```

The GPU code performance can be evaluated in a similar way by using **tic**, **toc** and the **GPUsync** command, as follows:

```
A = rand(1000,1000,GPUsingle);
B = rand(1000,1000,GPUsingle);
tic;A.*B;GPUsync;toc;
```

The following example shows a simple *Matlab* script to compare the execution time of the element-by-element multiplication between two matrices A and B on the GPU and on the CPU.

```
N = 100:100:2000;
timecpu = zeros(1,length(N));
timegpu = zeros(1,length(N));

index=1;
for i=N
Ah = single(rand(i));   % CPU
A  = rand(i,GPUsingle); % GPU

%% Execution on GPU
tic;
A.*A;
GPUsync;
timegpu(index) = toc;

%% Execution on CPU
tic;
Ah.*Ah;
timecpu(index) = toc;

% increase index
index = index +1;
end
```

The above code calculates the two vectors *timecpu* and *timegpu* that can be used to evaluate the speed-up between the GPU and the CPU as follows:

```
speedup = timecpu./timegpu
```

Chapter 3
GPUmat overview

GPUmat functions are grouped into high level and low level functions. High level functions can be used in a similar way as existing *Matlab* functions, while to use low level functions the user needs some experience in GPU programming. For example, low level functions can directly manage GPU memory, which is automatically handled with a Garbage Collector on high level functions. Low level functions can also directly access CUDA libraries such as CUBLAS and CUFFT. The detailed list of high level and low level functions can be found in Chapter 5. *GPUmat* can be used in the following ways:

- As any other *Matlab* toolbox by using high level functions. This is the easiest way to use *GPUmat*.
- As a GPU Source Development Kit, in order to integrate functions that are not available in the library, by using both low and high level functions. The *GPUmat* compiler can also be used to record GPU operations into new functions.

This chapter describes how to use the *GPUmat* high level functions. Users can find further information about low level functions in Chapter **??**. The full function reference is in Chapter 5. This chapter describes the following topics:

- Starting the GPU environment
- Creating a GPU variable
- Performing calculations on the GPU
- Converting a GPU variable into a *Matlab* variable
- Indexed references

- GPUmat functions
- GPU memory management
- Complex numbers
- Compatibility between *Matlab* and *GPUmat*
- *GPUmat* code performance

3.1 Starting the GPU environment

Name	Description
`GPUstart`	Starts GPU environment and loads the required library components
`GPUstop`	Stops the GPU environment
`GPUinfo`	Prints information about available CUDA capable GPUs

Table 3.1: GPU management functions.

Table 3.1 shows functions used to start *GPUmat* and to manage the GPU. The **GPUstart** and **GPUstop** commands are used to start and to stop *GPUmat* respectively. If more than a GPU is installed in the system, the user will be prompted to select the GPU device to use. The command **GPUinfo** prints information about installed GPUs:

```
GPUinfo
There is 1 device supporting CUDA
CUDA Driver Version:                              2.30
CUDA Runtime Version:                             2.30

Device 0: "GeForce GTX 275"
  CUDA Capability Major revision number:            1
  CUDA Capability Minor revision number:            3
  Total amount of global memory:                    939196416 bytes
  Number of multiprocessors:                        30
  Number of cores:                                  240
```

3.2 Creating a GPU variable

A GPU variable is a *Matlab* variable that is allocated on GPU memory and is created using the *Matlab* classes **GPUsingle** or **GPUdouble**. The *GPUsingle* and *GPUdouble* classes are equivalent to the single and double precision real/complex types in *Matlab*.

Functions to create a GPU variable are shown in table 3.2, and explained with more details in the next paragraphs. It is important to know that a memory transfer between GPU and CPU is required if the GPU variable is initialized with a *Matlab* array. A memory transfer is a time consuming task and might reduce the performance of the code.

Function	**Description**
`A = GPUsingle(Ah)` `A = GPUdouble(Ah)`	Creates a GPU array A initialized with the *Matlab* array Ah. **Requires GPU-CPU memory transfer**.
`A = rand(size, GPUsingle)` `A = rand(size, GPUdouble)`	Creates a GPU array initialized with random numbers (uniform distribution).
`A = randn(size, GPUsingle)` `A = randn(size, GPUdouble)`	Creates a GPU array initialized with random numbers (normal distribution).
`A = zeros(size, GPUsingle)` `A = zeros(size, GPUdouble)`	Creates a GPU array initialized with zeros.
`A = ones(size, GPUsingle)` `A = ones(size, GPUdouble)`	Creates a GPU array initialized with ones.
`A = colon(begin, stride, end, GPUsingle)` `A = colon(begin, stride, end, GPUdouble)`	`A = colon(begin, stride, end, GPUsingle)` creates a regularly spaced GPU vector A with values in the range `[begin:end]`.
`C = vertcat(A,B)` or `C = [A;B]`	Vertical concatenation. Can be applied to more than 2 GPU vectors.

Table 3.2: Functions used to create GPU variables.

A = GPUsingle(Ah)
A = GPUdouble(Ah)

Creates a GPU single or double precision variable *A* initialized with the *Matlab* array *Ah*. *A* has the same properties as *Ah*, such as the size and the number of elements. **Requires GPU-CPU memory transfer**.

Example:

```
Ah = single(rand(1000));% Ah is a Matlab variable
A  = GPUsingle(Ah);     % GPU variable
```

```
if GPUisDoublePrecision
  Ah = rand(1000);     % Ah is a Matlab variable
  A  = GPUdouble(Ah);  % GPU variable
end
```

There is a loss of precision in the conversion between double and single precision if the GPU variable is initialized with a double precision *Matlab* array *Ah*, as follows:

```
Ah = rand(1000);   % Ah is a double precision Matlab variable
A  = GPUsingle(Ah);% GPU variable
```

Conversion between double and single precision is possible using the functions *GPUsingle* and *GPUdouble* as follows:

```
if GPUisDoublePrecision
  Ah = rand(1000);          % Ah is a Matlab variable
  A  = GPUdouble(Ah);       % GPU variable, double prec.
end

Bh = single(rand(1000));  % Bh is a Matlab variable
B  = GPUsingle(Bh);       % GPU variable, single prec.

% convert GPU single to double
```

```
if GPUisDoublePrecision
  C = GPUdouble(B);
end

% convert GPU double to single
D = GPUsingle(A);
```

A = colon(begin, stride, end, GPUsingle)
A = colon(begin, stride, end, GPUdouble)
Creates a GPU variable A with values in the range `[begin:end]`. The increment between elements is `stride`. This command is similar to the *Matlab* **colon** command.

Example:

```
A = colon(0,2,1000,GPUsingle); % A is a GPU variable
```

The syntax to create a *Matlab* variable is very similar to the above code:

```
Ah = colon(0,2,1000); % A is a CPU variable
```

Existing variables can be efficiently used also to create others. The following example shows how to create a complex GPU variable using the *colon* function:

```
A = colon(0,2,6,GPUsingle); % A is a real GPU variable
B = sqrt(-1)*A;  % B is a complex GPU variable
C = 1 + B        % All real elements of B are set to 1
```

The previous commands result in

```
>> A
ans =
     0     2     4     6

Single precision REAL GPU type.
>> B
ans =
     0  0 + 2.0000i  0 + 4.0000i  0 + 6.0000i

Single precision COMPLEX GPU type.
>> C
ans =
   1.0000  1.0000 + 2.0000i  1.0000 + 4.0000i  1.0000 + 6.0000i
```

The function *colon* is very efficient to create a GPU variable because array values are directly created on the GPU memory without any data transfer between CPU and GPU.

A = rand(size, GPUsingle)
A = rand(size, GPUdouble)
A = randn(size, GPUsingle)
A = randn(size, GPUdouble)

Have the same behavior as *Matlab* **rand** and **randn** function. Create a GPU array with random numbers (single or double precision).

Example:

```
A = rand(1,1000,GPUsingle); % A is a GPU variable
if GPUisDoublePrecision
  B = rand(1,1000,GPUdouble); % B is a GPU variable
end
A = randn(1,1000,GPUsingle); % A is a GPU variable
if GPUisDoublePrecision
  B = randn(1,1000,GPUdouble); % B is a GPU variable
end
```

A = zeros(size, GPUsingle)
A = zeros(size, GPUdouble)

Has the same behavior as *Matlab* **zeros** function. Creates a GPU array with zeros (single or double precision).

Example:

```
A = zeros(1,1000,GPUsingle); % A is a GPU variable
if GPUisDoublePrecision
  B = zeros(1,1000,GPUdouble); % B is a GPU variable
end
```

A = ones(size, GPUsingle)

A = ones(size, GPUdouble)
Has the same behavior as *Matlab* **ones** function. Creates a GPU array with ones (single or double precision).

Example:

```
A = ones(1,1000,GPUsingle); % A is a GPU variable
if GPUisDoublePrecision
  B = ones(1,1000,GPUdouble); % B is a GPU variable
end
```

Find some examples of GPU variables creation in the file *CreateGPUVariables.m* located in the *example* folder. GPU variables can be converted into different types as follows:

```
if GPUisDoublePrecision
  A = ones(1,1000,GPUdouble); % A is a GPU variable, double prec.
  B = ones(1,1000);           % B is a CPU variable, double prec.
  % A + B gives an error. The CPU variable B is not automatically
  % conveterted to a GPU variable.
  C = A+B;
  % A + 1 is OK. The scalar 1 is automatically converted to a GPU
  % variable.
  C=A+1;
end
```

If *Matlab* types and GPU types are combined together, the conversion of one type to the other is not automatic, except for scalars.

3.3 Performing calculations on the GPU

The following example explains the mechanism that allows *Matlab* functions to be executed on the GPU.

```
A = rand(10,GPUsingle); % A is on GPU
B = exp(A)              % exp calculated on GPU
```

The *exp* function in the above code is the one implemented in *GPUmat* and not the built-in function. *Matlab* uses the *GPUmat* function because the argument of the *exp* is a *GPUsingle* type. The following example shows similar code executed on CPU:

```
A = single(rand(10)); % A is on CPU
B = exp(A)            % exp calculated on CPU
```

The mechanism to execute a function on the GPU is the following:

- Functions involving the GPU variables are executed on GPU by using *GPUmat* functions.
- Not every *Matlab* function is defined in *GPUmat*. This means that not every *Matlab* code is executed on the GPU, but only the *Matlab* code that uses functions defined in *GPUmat* (The complete function reference can be found in Chapter 5).

GPUmat implements also *Matlab* operators, such as `+, -, .*`. It means that algebraic expressions such as `A + B` are also defined in *GPUmat* and executed on the GPU. GPU operators are shown on table 3.9. Here is an example:

```
A  = rand(100,100,GPUsingle);   %GPU variable
B  = A/5 + A.*A*2 + 1;          %run on GPU
C  = A < B;                     %run on GPU

% Same operation performed on CPU
A  = single(A);                 %CPU variable
B  = A/5 + A.*A*2 + 1;          %run on CPU
C  = A < B;                     %run on CPU
```

3.4 Porting existing Matlab code

To port existing *Matlab* code, *Matlab* variables have to be converted to a GPU variable, **except scalars**. The easiest way to do it is to use the *GPUsingle* or *GPUdouble* initialized with the existing *Matlab* variable, but this is not the most efficient approach because it involves a memory transfer between CPU and GPU. Here is an example:

Name	Description
`a + b`	Binary addition
`a - b`	Binary subtraction
`-a`	Unary minus
`a.*b`	Element-wise multiplication
`a*b`	Matrix multiplication
`a./b`	Right element-wise division
`a./ b`	Left element-wise division
`a.^b`	Element-wise power
`a < b`	Less than
`a > b`	Greater than
`a <= b`	Less than or equal to
`a >= b`	Greater than or equal to
`a ~= b`	Not equal to
`a == b`	Equality
`a & b`	Logical AND
`a | b`	Logical OR
`~a`	Logical NOT
`a'`	Complex conjugate transpose
`a.'`	Matrix transpose

Table 3.9: Operators defined for GPU variables

```
Ah = [0:10:1000];   % Ah is on CPU
A  = GPUsingle(Ah); % A is on GPU, single precision
if GPUisDoublePrecision
  B  = GPUdouble(Ah); % B is on GPU, double precision
end
```

The above code can be written more efficiently using the **colon** function, as follows:

```
A  = colon(0,10,1000,GPUsingle); % A is on GPU
if GPUisDoublePrecision
  B  = colon(0,10,1000,GPUdouble); % B is on GPU
```

```
end
```

Matlab scalars are automatically converted into GPU variables, as described in previous sections.

3.5 Converting a GPU variable into a Matlab variable

Although a GPU variable is available from *Matlab*, its content is stored on the GPU memory. Converting a GPU variable into a *Matlab* variable means transferring the content of the variable from the GPU to the CPU memory. The following example describes how to convert a GPU variable *A* into a *Matlab* array *Ah*, by using the functions **single** and **double**:

```
Ah = rand(10);
A  = GPUsingle(Ah);  %A is on GPU memory
Ch = single(A);      %Ch is on CPU memory
if GPUisDoublePrecision
  B  = GPUdouble(Ah);  %B is on GPU memory
  Dh = double(B);      %Dh is on CPU memory
end
```

To visualize the content of a GPU variable on the *Matlab* command window, just type its name as any other *Matlab* array:

```
A = rand(5,GPUsingle); % A is on GPU
```

```
A

ans =

    0.8147    0.0975    0.1576    0.1419    0.6557
    0.9058    0.2785    0.9706    0.4218    0.0357
    0.1270    0.5469    0.9572    0.9157    0.8491
    0.9134    0.9575    0.4854    0.7922    0.9340
    0.6324    0.9649    0.8003    0.9595    0.6787
```

```
Single precision REAL GPU type.
```

Every time the content of a *GPUsingle* is read in *Matlab*, the system performs a memory transfer from the GPU to the CPU. The same happens when a *GPUsingle* is created and initialized using a *Matlab* array. Because of the limited memory bandwidth between the HOST and the GPU, the data transfer between CPU and GPU may be time consuming and therefore its usage should be limited.

3.6 Indexed references

The elements of a GPU array can be accessed as any other *Matlab* array, for example:

```
A = rand(50,GPUsingle); % A is on GPU
B = A(1:end);
B = A(1,1:10);
B = A(:);
A(1:10) = A(21:30);
```

Above commands are translated in *Matlab* to calls to the functions *subsref* and *subsasgn*. The implementation and the source code of these functions is documented in the *GPUmat User Modules* Wiki on Sourceforge (see Chapter **??** for further details).

The functions *slice* and *assign* can also be used to access the elements of a GPU array. They have a syntax very similar to the standard *Matlab* indexing but are faster than *subsref* and *subsasgn*. Table 3.10 shows the performance analysis of the *subsasgn* function for different *GPUmat* versions compared to the function *assign* and the CPU time. More details about the above tests are presented on the *GPUmat User Modules* Wiki. The following are some *slice* and *assign* examples (also available in the *Examples* folder, file *SliceAssign.m*).

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

% Matlab syntax
Ah = Bh(1:end);
% Equivalent slice syntax
```

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

Table 3.10: *subsasgn* performance analysis.

```
A = slice(B,[1,1,END]);

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

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

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

% Matlab syntax
Ah = Bh(:,:);
% Equivalent slice syntax
A = slice(B,':',':');

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

```
Ah = single(A);
Bh = single(B);

% Matlab syntax
Ah(1:10,1:10) = Bh;
% Equivalent assign syntax
assign(1, A, B, [1,1,10],[1,1,10]);

A = rand(100,GPUsingle);
B = rand(4,10,GPUsingle);
Ah = single(A);
Bh = single(B);

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

3.7 GPUmat functions

GPUmat currently implements only a subset of *Matlab* functions. The most important operators and numerical functions are implemented and users with programming experience can extend the library by using low level and high level functions that are available and documented in the library. Table 3.11 shows a short summary of implemented functions and operators.

Implemented functions	**Example**

`Matlab operators (A*B, A-B, A.*B, A+B, etc.)`	`A = rand(1000,GPUsingle);`
	`B = rand(1000,GPUsingle);`
	`C = A + B;`

`Numerical functions (exp, sqrt, log, etc.)`	`A = rand(1000,GPUsingle);`
	`B = rand(1000,GPUsingle);`
	`C = exp(A);`
	`D = sqrt(C) + B;`

`Fast Fourier Transform`	`RE = rand(1000,GPUsingle);`
	`IM = i*rand(1000,GPUsingle);`
	`C = fft(RE + IM);`

Table 3.11: Some *GPUmat* functions.

3.8 GPU memory management

The memory is managed automatically by *GPUmat*. Any GPU variable is automatically destroyed following exactly the same life-cycle as any other *Matlab* variable. Nevertheless, the GPU memory is limited and eventually the user can manually remove GPU variables by using the *Matlab* built-in command **clear**. Table 3.12 shows functions to manage the GPU memory.

Name	**Description**
clear	*Matlab* built-in command, removes the specified variables
GPUmem	Returns available GPU memory in bytes

Table 3.12: Functions used to manage the GPU memory

The following code shows a typical situation where the GPU memory is not enough, and some variables must be manually removed:

```
A = rand(6000,3000,GPUsingle); % A is on GPU
B = rand(6000,3000,GPUsingle); % B is on GPU
C = rand(6000,3000,GPUsingle); % C is on GPU
Device memory allocation error.
Available memory is 65274 KB, required 70312 KB
```

In the above example, it is not possible to allocate the variable C because the GPU memory is not enough (see the error message). In this case we must delete other variable, such as A or B. If we need also A and B, then our GPU card has not enough memory to manage all the variables. To delete a variable (for example A), use the **clear** command, as follows:

```
clear A
```

Check the file *MemoryExample.m*, located in the *example* folder, to understand how to use functions for memory management. The file performs the following actions:

- Displays the GPU available memory.
- Creates a *GPUsingle* variable on the GPU workspace and displays the available free memory.
- Cleans up the GPU variable and displays once more the available GPU memory.

A very useful *Matlab* command is the **whos**, which can be used to check how many GPU variables are on the *Matlab* workspace. The following *Matlab* output shows the result of the **whos** command and the presence of a *GPUsingle A* on the *Matlab* workspace:

```
>> whos
Name      Size                    Bytes  Class        Attributes

A         1x1000000                 924  GPUsingle
ans       1x1                         4  uint32
```

3.9 Low level GPU memory management

Memory management using high level functions is explained in section 3.8.

Memory management methods summary	
`GPUallocVector`	Allocates a variable on GPU memory.

GPU variables are managed in the following way:

- The *GPUsingle* (*GPUdouble*) implements a destructor which takes care of clearing unused memory regions. There is no need to explicitly clean up the GPU memory. If necessary it can be done using the *Matlab clear* command.
- If the user creates a *Matlab* pointer to the GPU memory using low level functions, the memory is not automatically cleaned when the variable is not used anymore. In this case the user must manually clean the GPU memory.

Above concepts are explained in next sections.

3.9.1 Memory management using the GPU classes

The following code shows how to allocate and delete a *GPUsingle* or *GPUdouble*.

```
A = rand(100,100,GPUsingle);
clear A;

B = GPUsingle();          % creates empty GPUsingle
setReal(B);               % REAL type
setSize(B,[100 100]);     % must set GPUsingle size
GPUallocVector(B);        % allocate GPU memory
clear B;
```

```
if GPUisDoublePrecision
  A = rand(100,100,GPUdouble);
  clear A;

  B = GPUdouble();          % creates empty GPUdouble
  setReal(B);               % REAL type
  setSize(B,[100 100]);     % must set GPUdouble size
  GPUallocVector(B);        % allocate GPU memory
  clear B;
end
```

3.9.2 Memory management using low level functions

The following code shows how to allocate a variable with 100 single precision floating point elements by using *CUBLAS* functions:

```
% create a new pointer
GPUptr = 0;

% allocate using cublasAlloc
SIZE_OF_FLOAT = 4;
NUMEL = 100;
[status GPUptr]= cublasAlloc(NUMEL,SIZE_OF_FLOAT,GPUptr);
cublasCheckStatus( status, 'Device memory allocation error');
```

The function *cublasFree* is used to free the memory:

```
status = cublasFree(GPUptr);
cublasCheckStatus( status, '!!!! memory free error (GPUptr)');
```

3.10 Complex numbers

A complex number is represented as a sequence of two values, the real and imaginary part respectively. A complex vector is a sequence of complex numbers, i.e. a sequence of interleaved real and imaginary values. There are different methods to create a complex GPU variable:

- Initializing a GPU variable with a *Matlab* complex number
- Multiply a real number by the imaginary unit
- Use *GPUreal* and *GPUimag* functions (or the corresponding high level functions *real* and *imag*)

Above points are explained in the following example:

```
% 1) Initialize a GPUsingle with a Matlab complex array

  Gh = rand(10) + sqrt(-1)*rand(10); %Matlab complex variable
  G  = GPUsingle(Gh);                %GPU single complex
```

```
% 2) Using real, imag, complex, GPUreal, GPUimag, GPUcomplex
  A = GPUsingle([1 2 3 4 5] + sqrt(-1)*[6 7 8 9 10]);
  RE = real(A);
  IM = imag(A);
  % same as above code, with low level functions
  RE = zeros(size(A), GPUsingle);
  IM = zeros(size(A), GPUsingle);
  GPUreal(A, RE);
  GPUimag(A, IM);
  % convert to complex
  D = complex(RE, IM);
  % same as above code, with low level functions
  E = complex(zeros(size(RE), GPUsingle));
  GPUcomplex(RE, IM, E);


% 3) Multiply a real array by the imaginary unit

  Gh = rand(10);               % Matlab real variable
  G  = GPUsingle(Gh)*sqrt(-1); % sqrt(-1) gives imaginary unit
```

3.11 Coding guidelines

To maximize the execution performance keep in mind the following points:

- Memory Transfers. Avoid excessive memory transfers between GPU/CPU memory.
- Vectorized operations and for-loops. The best performance in both *Matlab* and *GPUmat* can be achieved by using vectorized operations and avoiding for-loops. More information can be found at the following link: ***Matlab* Code Vectorization Guide**
- Use low level functions to avoid the creation of too many intermediate and temporary variables. This can speed up the code or help solving out of GPU memory errors.
- Compile the function using the *GPUmat* compiler. The compiler can be used to record GPU functions into a new *Matlab* function. Please check Chapter 4 for more details.

Next section explains previous points with more details.

3.11.1 Memory transfers

The most time consuming task is the memory transfer from/to GPU, such as initializing a GPU variable with a *Matlab* array. Here is an example:

```
Ah = rand(1000);   % Ah is on CPU memory
A = GPUsingle(Ah); % A is on GPU memory
```

In the above code, the variable *Ah* is used to initialize the GPU variable *A*, which means that data is transferred from the CPU to the GPU memory. Vice versa, when a GPU variable is converted into a *Matlab* variable there is a memory transfer from the GPU to the CPU:

```
A = rand(1000,GPUsingle);  % A is on GPU memory
Ah = single(A);            % Ah is on CPU memory
```

The fastest way to initialize or create a GPU variable is to use existing variables on the GPU memory to create other GPU variables, or to use functions such as *zeros*, *colon* or *rand* which directly create values on the GPU without transferring data from *Matlab*. Please check Section 3.2 for more information about creating new GPU variables with *GPUmat*.

3.11.2 Vectorized code and for-loops

Another way to improve the code performance is to avoid *for loops* by using vectorized operations. For example:

```
for i=1:1e6
  A = rand(3,3);
  B = rand(3,3);
  C = A.*B;
  %% do something with C
end
```

The above code can be executed as-is on the GPU by converting **A** and **B** to *GPUsingle*, as follows:

```
for i=1:1e6
```

```
  A = rand(3,3,GPUsingle);
  B = rand(3,3,GPUsingle);
  C = A.*B;
  %% do something with C
end
```

Nevertheless, matrix operations can be used instead of the `for-loop` by creating two arrays with `3 x 3e6` elements and multiplying them element-by-element:

```
A = rand(3,3e6,GPUsingle); % A is on GPU
B = rand(3,3e6,GPUsingle); % B is on GPU
C = A.*B;                   % C is on GPU
```

The following *Matlab* code perform the matrix addition `C = A + B` using a `for-loop` statement.

```
A = rand(100);
B = rand(100);
C = zeros(100);
for i=1:size(A,1)
  for j=1:size(B,2)
    C(i,j) = A(i,j) + B(i,j);
  end
end
```

To port the code to the GPU, it is suggested to use the element-by-element addition instead of using the `for-loop`:

```
A = rand(100,GPUsingle);  % A is on GPU
B = rand(100,GPUsingle);  % B is on GPU
C = A + B;                % C is on GPU
```

3.11.3 Reduce intermediate variables creation

Consider the following code:

```
A = rand(100,GPUsingle);  % A is on GPU
B = rand(100,GPUsingle);  % B is on GPU
```

```
C = exp(A + B)*2.0;       % C is on GPU
```

In the above code, the calculation of C is done internally by *Matlab* with the following steps:

```
A = rand(100,GPUsingle);  % A is on GPU
B = rand(100,GPUsingle);  % B is on GPU
%C = exp(A + B)*2.0;      % C is on GPU
tmp1 = A+B;
tmp2 = exp(tmp1);
clear tmp1;
C = tmp2*2.0;
clear tmp2;
```

The creation of the intermediate variables *tmp1* and *tmp2* can be avoided using low level functions. Some high level functions have a corresponding low level function that performs exactly the same function without returning any value. The output vector should be passed as input argument, as follows:

```
A = rand(100,GPUsingle);  % A is on GPU
B = rand(100,GPUsingle);  % B is on GPU
%C = exp(A + B)*2.0;      % C is on GPU
% create output vector C
C = zeros(size(A), GPUsingle);
GPUplus(A,B,C);
GPUexp(C,C);
GPUtimes(C,2.0,C);
```

In the above code the result C is created using the *zeros* function. C is then updated with the sum between A and B, the *exp(C)* and finally it is multiplied by 2.0. At the end of the calculations C contains the result of `exp(A + B)*2.0`, and no intermediate temporary variable has been created. By using low level functions it is possible to avoid out of memory errors. In fact, temporary variables might not be deleted immediately by the *Matlab* garbage collector, but in the above example we are sure that only one variable (C) for the result has been created.

3.11.4 Matlab and GPU variables

Operations and functions involving *Matlab* and GPU variables at the same time are not defined, except operations involving *GPU* variables and *Matlab* scalars. The following is an example:

```
Ah = rand(5);           % Ah is on CPU
A  = rand(5,GPUsingle); % A is on GPU
Bh = 1;  % Bh is on CPU
Ah + A
Unknown operation + between 'double' and 'GPUsingle'
A + Bh
ans =

    1.8147    1.0975    1.1576    1.1419    1.6557
    1.9058    1.2785    1.9706    1.4218    1.0357
    1.1270    1.5469    1.9572    1.9157    1.8491
    1.9134    1.9575    1.4854    1.7922    1.9340
    1.6324    1.9649    1.8003    1.9595    1.6787

Single precision REAL GPU type.
```

Adding Ah and A generates an error, whereas adding A and Bh is possible because Bh is a scalar. A can be converted into a *Matlab* variable and added to Ah or in a similar way Ah can be converted into a GPU variable and added to A, as follows:

```
Ah = rand(5);
A  = rand(5,GPUsingle);

Ah + single(A); % A converted into Matlab

Ch = single(A); % A converted into Matlab Ch
Ah + Ch;        % adding Ah and Ch

D  = GPUsingle(Ah); % Ah converted into the GPUsingle D
A + D;              % adding A and D

A + GPUsingle(Ah);  % A added directly to GPUsingle(Ah)
```

3.12 Performance analysis

The easiest way to evaluate the performance in *Matlab* are the **tic** and **toc** commands, as follows:

```
A = rand(1000,1000); % A is on CPU
B = rand(1000,1000); % B is on CPU
tic;A.*B;toc;  % executed on CPU
```

The GPU code performance can be evaluated in a similar way by using **tic**, **toc** and the **GPUsync** command, as follows:

```
A = rand(1000,1000,GPUsingle);
B = rand(1000,1000,GPUsingle);
tic;A.*B;GPUsync;toc;
```

The **GPUsync** command is used to synchronize the GPU code. It means that *Matlab* waits until the GPU execution is completed. The execution of the GPU code is asynchronous, i.e. the control is returned to *Matlab* after calling the *GPUmat* function. But this does not necessarily mean that the GPU has finished its task. To force *Matlab* to wait until the GPU has finished his task, the *GPUsync* command must be used. Here is an example:

```
A = rand(1000,1000,GPUsingle);
B = rand(1000,1000,GPUsingle);
tic;A.*B;GPUsync;toc;
Elapsed time is 0.010231 seconds.
tic;A.*B;toc;
Elapsed time is 0.003808 seconds.
```

Asynchronous execution is entirely managed by *GPUmat* and is transparent to the user. The **GPUsync** should be used only when checking the GPU execution time.

Chapter 4
GPUmat compiler

4.1 Overview

The *GPUmat* compiler allows the user to record several GPU operations into a single *Matlab* function (see Table 4.1 for a summary of available *GPUmat* compiler functions). Please check Section 4.3 for the system requirements. By using the compiler it is possible to generate optimized code that is executed faster than the native *GPUmat* code. Nevertheless, there are some limitations. (see Section 4.4).

Name	**Description**
`GPUcompileStart`	Starts the compilation
`GPUcompileStop`	Stops the compilation
`GPUcompileAbort`	Aborts the compilation
`GPUfor`	Starts a for-loop
`GPUend`	Ends a for-loop
`GPUcompileMEX`	Compiles a .cpp file

Table 4.1: *GPUmat* compiler functions.

The compilation is performed as follows:

- Start the compilation. Define the input arguments of the generated function.
- Execute operations on the GPU by running *GPUmat* code. Every GPU operation is recorded into the generated function.
- Stop the compilation. Define the output arguments of the generated function.

The following code generates a function `[r1,...,rn]=name(p1,p2,...,pn)`, where *p1* to *pn* are input parameters and *r1* to *rn* are output parameters.

```
GPUcompileStart(name, p1, p2, ..., pn)
...
GPUcompileStop(r1, r2, ..., rn)
```

For example, the following code shows how to compile a function *myexp*, having one input and one output argument and the same behavior as the native *GPUmat exp* function:

```
A = randn(5,GPUsingle);
% A is a dummy variable

GPUcompileStart('myexp','-f',A)
R = exp(A);
GPUcompileStop(R)
```

The *GPUcompileStart* function is used to start the compilation, and has the following interface:

```
GPUcompileStart(name, p1, p2, ..., pn)
```

The parameter *name* is the name of the compiled function. Parameters *p1* to *pn* are the input arguments of the compiled function. They can be a *GPUtype* (*GPUsingle*, *GPUdouble*, etc.) or a *Matlab* variable. The variable *A* in the above example is a dummy variable. It is used to define the first input argument of the function *myexp*. After calling the *GPUcompileStart* function, we run the *GPUmat* code that should be recorded in the compiled function, as follows:

```
R = exp(A)
```

The function *GPUcompileStop*, used to stop the compilation, has the following interface:

```
GPUcompileStop(r1, r2, ..., rn)
```

Parameters *r1* to *rn* are the output arguments of the compiled function. They can be only *GPUtype* (*GPUsingle*, *GPUdouble*, etc.). The following example creates the function `[R1, R2] = myfun(A1,A2)` (two input and two output arguments):

```
A = randn(5,GPUsingle);
B = randn(5,GPUsingle);
% A and B are dummy variables

GPUcompileStart('myfun','-f', A, B)
R1 = exp(A);
R2 = floor(B);
GPUcompileStop(R1,R2)
```

The following is another example:

```
A = randn(5,GPUsingle);
% A is a dummy variable

GPUcompileStart('myfun1','-f', A)
R1 = floor(exp(A));
GPUcompileStop(R1)
```

Find more examples in the *GPUmat* folder *examples*, file *GPUmatCompiler.m*.

4.2 For loops

It is possible to generate for-loops in the compiled code by using *GPUfor* and *GPUend*. The following is an example:

```
A = randn(5,5,5,GPUsingle);
B = randn(5,GPUsingle);
GPUcompileStart('myfor1', '-f', A, B)
GPUfor it=1:5
  assign(1,A,B,':',':',it)
GPUend
GPUcompileStop
```

The following is another example with nested loops:

```
A = randn(5,5,5,GPUsingle);
```

```
B = randn(1,5,GPUsingle);
GPUcompileStart('myfor2', '-f', A, B)
GPUfor it=1:5
  GPUfor jt=1:5
    assign(1,A,B,':',jt,it)
  GPUend
GPUend
GPUcompileStop
```

4.3 System requirements

Your system must be configured to compile *Matlab* mex functions. Please check the *Matlab* manual for more details about *Building MEX-Files*. A valid compiler must be installed in order to compile. Under Windows we suggest *Microsoft Visual C++ Express Edition*, a free product from Microsoft. Under Linux we suggest the free *GPU GCC* compiler.

To configure the compiler under *Matlab* run the following command:

```
mex -setup
```

To check from *GPUmat* if the system is properly configured, run the following script after starting *GPUmat*:

```
GPUcompileCheck
```

4.4 Limitations

The *GPUmat* compilers records GPU functions only. *Matlab* functions are not included in the compilation. The following are some examples:

```
A = randn(5,5,5,GPUsingle);
a = 1;
GPUcompileStart('code_ex1', '-f', A)

if a==1
  R = exp(A);
```

```
else
  R = floor(A)
end

GPUcompileStop(R)
```

In the above code, only one *if* statement is evaluated. Therefore, only one command is executed on GPU and recorded to the compiled function. The above code is equivalent to the following:

```
A = randn(5,5,5,GPUsingle);
GPUcompileStart('code_ex1', '-f', A)
R = exp(A);
GPUcompileStop(R)
```

Not every *GPUmat* function is supported in compilation mode. Check the function reference for more details.

A *Matlab* variable passed to a *GPUmat* function is hard-coded if not defined in *GPUcompileStart* as an input parameter. For example:

```
A = randn(5,5,GPUsingle);
GPUcompileStart('code_ex2', '-f', A)
assign(1,A,single(1),':',':')
GPUcompileStop
```

In the above code, all the arguments of the function *assign* are hard-coded except *A*. The function *code_ex2* performs always the same operation on the input argument. For example:

```
>> A = randn(3,3,GPUsingle)
code_ex2(A)
A

ans =

    0.3848    1.0992   -0.4760
    0.3257    0.6532   -2.0516
    1.2963   -0.5051   -0.4483
```

```
Single precision REAL GPU type.

ans =

     1     1     1
     1     1     1
     1     1     1

Single precision REAL GPU type.
```

The following code is similar, but allows the user to define the arguments of the *assign* function:

```
A = randn(5,5,GPUsingle);
a = 1; % dummy
b = 1; % dummy
c = 1; % dummy
GPUcompileStart('code_ex3', '-f', A, a, b, c)
assign(1,A,a,b,c)
GPUcompileStop
```

The following command

```
A = randn(3,3,GPUsingle)
code_ex3(A,single(2),':',':')
A
```

generates the following output:

```
>> A = randn(3,3,GPUsingle)
code_ex3(A,single(2),':',':')
A

ans =

    0.8776    0.6011   -0.2676
    1.0336   -0.6740    0.1866
    0.4198   -1.0952    0.9509
```

```
Single precision REAL GPU type.

ans =

     2     2     2
     2     2     2
     2     2     2

Single precision REAL GPU type.
```

Indexed assignment are not implemented. For example, the following code generates an error:

```
A = randn(5,5,5,GPUsingle);
GPUcompileStart('code_ex1', '-f', A)
R = A(1:3,:,:);
GPUcompileStop(R)

A = randn(5,5,5,GPUsingle);
GPUcompileStart('code_ex1', '-f', A)
A(1:3,:,:)=1;
GPUcompileStop
```

The above code can be replaced with the following:

```
A = randn(5,5,5,GPUsingle);
GPUcompileStart('code_ex1', '-f', A)
R = slice(A,[1,1,3],':',':');
GPUcompileStop(R)

A = randn(5,5,5,GPUsingle);
GPUcompileStart('code_ex1', '-f', A)
assign(1,A,single(1),[1,1,3],':',':');
GPUcompileStop
```

Above example shows that native *Matlab* indexed assignment statements have to be replaced with functions *slice* or *assign*.

4.5 Compilation errors

4.5.1 GPUfor.1 - Unable to parse iterator

GPUmat was not able to parse the iterator. The following code contains an error:

```
GPUfor jt - 1:M
GPUend
```

The above code generates an error of type *GPUfor.1*.

4.5.2 GPUfor.2 - Iterator name cannot be i or j

The variables i and j cannot be used as iterator names. The following code generates an error:

```
GPUfor j=1:10
GPUend
```

Above code can be modified as follows:

```
GPUfor jt=1:10
GPUend
```

4.5.3 GPUfor.3 - GPUfor iterator must be a Matlab double precision variable

A valid iterator must be a *Matlab* double precision variable

4.5.4 NUMERICS.1 - Function compilation is not implemented

Some functions cannot be used during the compilation. Please check Section 4.6 for a list of not implemented functions.

4.5.5 GPUMANAGER.13 - GPUtype variable not available in compilation context

When accessing a variable during the compilation, the variable should be defined in the compilation context. A new variable is automatically added to the compilation context, whereas an existing variable should be declared when calling the function *GPUcompileStart*. For example:

```
A = randn(5,5,GPUsingle);
GPUcompileStart('code_ex4', '-f', A)
R = exp(A);
GPUcompileStop
```

In the above code, the variable R is created during the compilation and it is automatically added to the compilation context. The variable A must be passed to the function *GPUcompilerStart*, otherwise an error is generated.

4.5.6 GPUMANAGER.15 - Compilation stack overflow

The compiler stack is limited. This error can occur in the following cases:

- The script being compiled is too long. The compiled function should not be too long. Try to split your code into different parts.
- *Matlab* for-loop. If you compile a for-loop (not a GPUfor-loop), the *GPUmat* compiler generates code for each iteration of the loop (the loop is unrolled). By doing this way, it is possible that the generated codes fills the compiler stack. It is suggested to replace the native *Matlab* for-loop statements with the *GPUmat GPUfor-loop* commands.

4.6 Not implemented functions

Not every *GPUmat* function can be used during the compilation. In general, every function that retrieves a *GPUtype* property, such as *size* or *numel*, is not implemented. Find more information for each function in Chapter 5.

4.7 Additional compilation options

The *GPUcompileStart* can be executed with the additional options in Table 4.7.

Name	**Description**
`-f`	Force compilation. Overwrites target file.
`-verbose0`	Verbosity level 0
`-verbose1`	Verbosity level 1
`-verbose2`	Verbosity level 2
`-verbose4`	Verbosity level 4

Table 4.2: *GPUcompileStart* options

For example:

```
A = randn(5,5,GPUsingle);
GPUcompileStart('code_ex5', '-f', '-verbose4', A)
R = exp(A);
GPUcompileStop
```

Chapter 5
Function Reference

5.1 Functions - by category

5.1.1 GPU startup and management

Name	Description
`GPUinfo`	Prints information about the GPU device
`GPUstart`	Starts the GPU environment and loads required components
`GPUstop`	Stops the GPU environment

5.1.2 GPU variables management

Name	Description
`colon`	Colon
`double`	Converts a GPU variable into a Matlab double precision variable
`eye`	Identity matrix
`GPUdouble`	GPUdouble constructor
`GPUeye`	Identity matrix
`GPUfill`	Fill a GPU variable
`GPUones`	GPU ones array
`GPUsingle`	GPUsingle constructor
`GPUsync`	Wait until all GPU operations are completed
`GPUzeros`	GPU zeros array
`memCpyDtoD`	Device-Device memory copy

memCpyHtoD	Host-Device memory copy
ones	GPU ones array
repmat	Replicate and tile an array
setComplex	Set a GPU variable as complex
setReal	Set a GPU variable as real
setSize	Set GPU variable size
single	Converts a GPU variable into a Matlab single precision variable
zeros	GPU zeros array

5.1.3 GPU memory management

Name	**Description**
GPUallocVector	Variable allocation on GPU memory
GPUmem	Returns the free memory (bytes) on selected GPU device

5.1.4 Random numbers generator

Name	**Description**
GPUrand	GPU pseudorandom generator
GPUrandn	GPU pseudorandom generator
rand	GPU pseudorandom generator
randn	GPU pseudorandom generator

5.1.5 Numerical functions

Name	**Description**
abs	Absolute value
acos	Inverse cosine
acosh	Inverse hyperbolic cosine
and	Logical AND

`asin`	Inverse sine
`asinh`	Inverse hyperbolic sine
`assign`	Indexed assignement
`atan`	Inverse tangent, result in radians
`atanh`	Inverse hyperbolic tangent
`ceil`	Round towards plus infinity
`clone`	Creates a copy of a GPUtype
`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
`fft`	Discrete Fourier transform
`fft2`	Two-dimensional discrete Fourier Transform
`floor`	Round towards minus infinity
`ge`	Greater than or equal
`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
`GPUconj`	`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
`GPUfloor`	Round towards minus infinity
`GPUge`	Greater than or equal
`GPUgt`	Greater than
`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
GPUmtimes	Matrix multiply
GPUne	Not equal
GPUnot	Logical NOT
GPUor	Logical OR
GPUplus	Plus
GPUpower	Array power
GPUrdivide	Right array divide
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
gt	Greater than
ifft	Inverse discrete Fourier transform
ifft2	Two-dimensional inverse discrete Fourier transform
ldivide	Left array divide
le	Less than or equal
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
minus	Minus
mrdivide	Slash or right matrix divide
mtimes	Matrix multiply
ne	Not equal

not	Logical NOT
or	Logical OR
permute	Permute array dimensions
plus	Plus
power	Array power
rdivide	Right array divide
reshape	Reshape array
round	Round towards nearest integer
sin	Sine of argument in radians
sinh	Hyperbolic sine
slice	Subscripted reference
sqrt	Square root
subsref	Subscripted reference
sum	Sum of elements
tan	Tangent of argument in radians
tanh	Hyperbolic tangent
times	Array multiply
transpose	Transpose
uminus	Unary minus
vertcat	Vertical concatenation

5.1.6 General information

Name	Description
display	Display GPU variable
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
GPUisDoublePrecision	Check if GPU is double precision
iscomplex	True for complex array
isempty	True for empty GPUsingle array
isreal	True for real array
isscalar	True if array is a scalar
length	Length of vector
ndims	Number of dimensions

numel	Number of elements in an array or subscripted array expression.
size	Size of array

5.1.7 User defined modules

Name	Description
GPUgetUserModule	Returns CUDA (.cubin) module handler
GPUuserModuleLoad	Loads CUDA .cubin module
GPUuserModulesInfo	Prints loaded CUDA .cubin modules
GPUuserModuleUnload	Unloads CUDA (.cubin) module

5.1.8 GPUmat compiler

Name	Description
GPUcompileAbort	Aborts the GPUmat compilation.
GPUcompileStart	Starts the GPUmat compiler.
GPUcompileStop	Stops the GPUmat compiler.

5.1.9 Complex numbers

Name	Description
complex	Construct complex data from real and imaginary components
GPUcomplex	Construct complex data from real and imaginary components
GPUimag	Imaginary part of complex number
GPUreal	Real part of complex number
imag	Imaginary part of complex number
real	Real part of complex number

5.1.10 CUDA Driver functions

Name	Description
cuCheckStatus	Check the CUDA DRV status.
cuInit	Wrapper to CUDA driver function cuInit
cuMemGetInfo	Wrapper to CUDA driver function cuMemGetInfo

5.1.11 CUDA run-time functions

Name	Description
cudaCheckStatus	Check the CUDA run-time status
cudaGetDeviceCount	Wrapper to CUDA cudaGetDeviceCount function.
cudaGetDeviceMajorMinor	Returns CUDA compute capability major and minor numbers.
cudaGetDeviceMemory	Returns device total memory
cudaGetDeviceMultProcCount	Returns device multi-processors count
cudaGetLastError	Wrapper to CUDA cudaGetLastError function
cudaSetDevice	Wrapper to CUDA cudaSetDevice function
cudaThreadSynchronize	Wrapper to CUDA cudaThreadSynchronize function.

5.2 Functions - by module

5.2.1 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 assignement
`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`	`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
GPUmtimes	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
memCpyHtoD	Host-Device memory copy
minus	Minus
mrdivide	Slash or right matrix divide
mtimes	Matrix multiply
ndims	Number of dimensions
ne	Not equal
not	Logical NOT
numel	Number of elements in an array or subscripted array expression.
ones	GPU ones array
or	Logical OR
permute	Permute array dimensions
plus	Plus
power	Array power
rdivide	Right array divide
real	Real part of complex number
repmat	Replicate and tile an array
round	Round towards nearest integer
sin	Sine of argument in radians
sinh	Hyperbolic sine
size	Size of array

slice	Subscripted reference
sqrt	Square root
subsref	Subscripted reference
tan	Tangent of argument in radians
tanh	Hyperbolic tangent
times	Array multiply
transpose	Transpose
uminus	Unary minus
zeros	GPU zeros array

5.2.2 GPUMAT module

Name	Description
cuCheckStatus	Check the CUDA DRV status.
cudaCheckStatus	Check the CUDA run-time status
cudaGetDeviceCount	Wrapper to CUDA cudaGetDeviceCount function.
cudaGetDeviceMajorMinor	Returns CUDA compute capability major and minor numbers.
cudaGetDeviceMemory	Returns device total memory
cudaGetDeviceMultProcCount	Returns device multi-processors count
cudaGetLastError	Wrapper to CUDA cudaGetLastError function
cudaSetDevice	Wrapper to CUDA cudaSetDevice function
cudaThreadSynchronize	Wrapper to CUDA cudaThreadSynchronize function.
cufftPlan3d	Wrapper to CUFFT cufftPlan3d function
cuInit	Wrapper to CUDA driver function cuInit
cuMemGetInfo	Wrapper to CUDA driver function cuMemGetInfo
display	Display GPU variable
double	Converts a GPU variable into a Matlab double precision variable
GPUallocVector	Variable allocation on GPU memory
GPUcompileAbort	Aborts the GPUmat compilation.
GPUcompileStart	Starts the GPUmat compiler.

GPUcompileStop	Stops the GPUmat compiler.
GPUdeviceInit	Initializes a CUDA capable GPU device
GPUdouble	GPUdouble constructor
GPUgetUserModule	Returns CUDA (.cubin) module handler
GPUinfo	Prints information about the GPU device
GPUisDoublePrecision	Check if GPU is double precision
GPUmem	Returns the free memory (bytes) on selected GPU device
GPUsingle	GPUsingle constructor
GPUstart	Starts the GPU environment and loads required components
GPUstop	Stops the GPU environment
GPUsync	Wait until all GPU operations are completed
GPUuserModuleLoad	Loads CUDA .cubin module
GPUuserModulesInfo	Prints loaded CUDA .cubin modules
GPUuserModuleUnload	Unloads CUDA (.cubin) module
packfC2C	Pack two arrays into an interleaved complex array
packfR2C	Transforms a real array into a complex array with zero complex elements.
reshape	Reshape array
setComplex	Set a GPU variable as complex
setReal	Set a GPU variable as real
setSize	Set GPU variable size
single	Converts a GPU variable into a Matlab single precision variable
sum	Sum of elements
unpackfC2C	Unpack one complex array into two single precision arrays
unpackfC2R	Transforms a complex array into a real array discarding the complex part
vertcat	Vertical concatenation

5.2.3 EXAMPLES:CODEOPT module

Name	Description
forloop1	Code optimization example

5.2.4 RAND module

Name	Description
`GPUrand`	GPU pseudorandom generator
`GPUrandn`	GPU pseudorandom generator
`rand`	GPU pseudorandom generator
`randn`	GPU pseudorandom generator

5.2.5 EXAMPLES:GPUTYPE module

Name	Description
`gputype_create1`	GPUtype creation example
`gputype_create2`	GPUtype creation example
`gputype_properties`	GPUtype properties example

5.2.6 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

5.3 Operators

Operators are used in mathematical expression such as `A + B`. *GPUmat* overloads *Matlab* operators for the *GPUsingle* class.

Name	Description
a + b	Binary addition
a - b	Binary subtraction
-a	Unary minus
a.*b	Element-wise multiplication
a*b	Matrix multiplication
a./b	Right element-wise division
a./ b	Left element-wise division
a.^b	Element-wise power
a < b	Less than
a > b	Greater than
a <= b	Less than or equal to
a >= b	Greater than or equal to
a ~= b	Not equal to
a == b	Equality
a & b	Logical AND
a | b	Logical OR
~a	Logical NOT
a'	Complex conjugate transpose
a.'	Matrix transpose

5.3.1 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)
```

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

5.3.3 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)
```

5.3.4 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 >= 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)
```

5.3.5 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)
```

5.3.6 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)
```

5.3.7 lt

lt - 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)
```

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

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

MATLAB COMPATIBILITY

Supported only `A / n` where `n` is scalar.

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

5.3.11 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)
```

5.3.12 not

not - Logical NOT

SYNTAX

```
R = ~X
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

`~A (not(A))` performs a logical NOT of input array A.
Compilation supported

EXAMPLE

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

5.3.13 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)
```

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

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

MATLAB COMPATIBILITY

Implemented for REAL exponents only.

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

5.3.17 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,':',':');
```

5.3.18 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)
```

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

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

5.3.21 vertcat

vertcat - Vertical concatenation

SYNTAX

```
R = [X;Y]
X - GPUsingle, GPUdouble
Y - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

GPUMAT

DESCRIPTION

`[A;B]` is the vertical concatenation of matrices A and B. A and B must have the same number of columns. Any number of matrices can be concatenated within one pair of brackets.
Compilation not supported

EXAMPLE

```
A = [zeros(10,1,GPUsingle);colon(0,1,10,GPUsingle)'];
```

5.4 Functions - alphabetical list

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

5.4.2 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 = rand(10,GPUsingle);
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.

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

MATLAB COMPATIBILITY

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

5.4.4 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)
```

5.4.5 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 = rand(10,GPUsingle);
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.

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

MATLAB COMPATIBILITY

Not implemented for complex X.

5.4.7 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 = rand(100,GPUsingle);
B = rand(10,10,GPUsingle);
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]);
```

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

MATLAB COMPATIBILITY

Not implemented for complex X.

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

MATLAB COMPATIBILITY

Not implemented for complex X.

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

MATLAB COMPATIBILITY
Not implemented for complex X.

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

5.4.12 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)
```

5.4.13 complex

complex - Construct complex data from real and imaginary components

SYNTAX

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

MODULE NAME

NUMERICS

DESCRIPTION

`R = complex(X, Y)` creates a complex output R from the two real inputs X and Y. `R = complex(X)` creates a complex output R from the real input X. Imaginary part is set to 0.
Compilation supported

EXAMPLE

```
RE = rand(10,GPUsingle);
IM = rand(10,GPUsingle);
R = complex(RE);
R = complex(RE, IM);
```

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

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

MATLAB COMPATIBILITY

Not implemented for complex X.

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

MATLAB COMPATIBILITY

Not implemented for complex X.

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

5.4.18 cuCheckStatus

cuCheckStatus - Check the CUDA DRV status.

MODULE NAME
GPUMAT

DESCRIPTION
`cuCheckStatus(STATUS,MSG)` returns `EXIT_FAILURE(1)` or `EXIT_SUCCESS(0)` depending on STATUS value, and throws an error with message 'MSG'.
Compilation not supported

EXAMPLE

```
[status]=cuInit();
cuCheckStatus( status, 'Error initialize CUDA driver.');
```

5.4.19 cudaCheckStatus

cudaCheckStatus - Check the CUDA run-time status

MODULE NAME
GPUMAT

DESCRIPTION
`RET = cudaCheckStatus(STATUS,MSG)` returns `EXIT_FAILURE(1)` or `EXIT_SUCCESS(0)` depending on STATUS value, and throws an error with message 'MSG'.
Compilation not supported

EXAMPLE

```
status = cudaGetLastError();
cudaCheckStatus( status, 'Kernel execution error.');
```

5.4.20 cudaGetDeviceCount

cudaGetDeviceCount - Wrapper to CUDA cudaGetDeviceCount function.

MODULE NAME
GPUMAT

DESCRIPTION
Wrapper to CUDA cudaGetDeviceCount function.
Compilation not supported

EXAMPLE

```
count = 0;
[status,count] = cudaGetDeviceCount(count);
if (status ~=0)
  error('Unable to get the number of devices');
end
```

5.4.21 cudaGetDeviceMajorMinor

cudaGetDeviceMajorMinor - Returns CUDA compute capability major and minor numbers.

MODULE NAME
GPUMAT

DESCRIPTION
Returns CUDA compute capability major and minor numbers. `[STATUS, MAJOR, MINOR] = cudaGetDeviceMajorMinor(DEV)` returns the compute capability number (major, minor) of the `device=DEV`. STATUS is the result of the operation.
Compilation not supported

EXAMPLE

```
dev = 0;
[status,major,minor] = cudaGetDeviceMajorMinor(dev);
if (status ~=0)
  error(['Unable to get the compute capability']);
end

major
minor
```

5.4.22 cudaGetDeviceMemory

cudaGetDeviceMemory - Returns device total memory

MODULE NAME
GPUMAT

DESCRIPTION
`[STATUS, TOTMEM] = cudaGetDeviceMemory(DEV)` returns the total memory of the device=DEV. STATUS is the result of the operation.
Compilation not supported

EXAMPLE

```
dev = 0;
[status,totmem] = cudaGetDeviceMemory(dev);
if (status ~=0)
  error('Error getting total memory');
end
totmem = totmem/1024/1024;
disp(['Total memory=' num2str(totmem) 'MB']);
```

5.4.23 cudaGetDeviceMultProcCount

cudaGetDeviceMultProcCount - Returns device multi-processors count

MODULE NAME
GPUMAT

DESCRIPTION
`[STATUS, COUNT] = cudaGetDeviceMultProcCount(DEV)` returns the number of multi-processors of the `device=DEV`. STATUS is the result of the operation.
Compilation not supported

EXAMPLE

```
dev = 0;
[status,count] = cudaGetDeviceMultProcCount(dev);
if (status ~=0)
  error('Error getting numer of multi proc');
end
disp(['    Mult. processors = ' num2str(count) ]);
```

5.4.24 cudaGetLastError

cudaGetLastError - Wrapper to CUDA cudaGetLastError function

MODULE NAME
GPUMAT

DESCRIPTION
`[STATUS] = cudaGetLastError()` returns the last error from the run-time call. STATUS is the result of the operation.
Original function declaration:

```
cudaError_t
cudaGetLastError(void)
```

Compilation not supported

5.4.25 cudaSetDevice

cudaSetDevice - Wrapper to CUDA cudaSetDevice function

MODULE NAME

GPUMAT

DESCRIPTION

`[STATUS] = cudaSetDevice(DEV)` sets the device to DEV and returns the result of the operation in STATUS.
Original function declaration:

```
cudaError_t
cudaSetDevice( int dev )
```

Compilation not supported

5.4.26 cudaThreadSynchronize

cudaThreadSynchronize - Wrapper to CUDA cudaThreadSynchronize function.

MODULE NAME

GPUMAT

DESCRIPTION

`[STATUS] = cudaThreadSynchronize()`. STATUS is the result of the operation.
Original function declaration:

```
cudaError_t cudaThreadSynchronize(void)
```

Compilation not supported

5.4.27 cufftPlan3d

cufftPlan3d - Wrapper to CUFFT cufftPlan3d function

MODULE NAME
GPUMAT

DESCRIPTION
Wrapper to CUFFT cufftPlan3d function. Original function declaration:

```
cufftResult
cufftPlan2d(cufftHandle *plan,
             int nx, int ny, int nz,
             cufftType type);
```

Original function returns only a cufftResult, whereas wrapper returns also the plan.
Compilation not supported

5.4.28 cuInit

cuInit - Wrapper to CUDA driver function cuInit

MODULE NAME
GPUMAT

DESCRIPTION
Wrapper to CUDA driver function cuInit.
Compilation not supported

5.4.29 cuMemGetInfo

cuMemGetInfo - Wrapper to CUDA driver function cuMemGetInfo

MODULE NAME
GPUMAT

DESCRIPTION
Wrapper to CUDA driver function cuMemGetInfo.
Compilation not supported

EXAMPLE

```
freemem = 0;
c = 0;
[status, freemem, c] = cuMemGetInfo(freemem,c);
```

5.4.30 display

display - Display GPU variable

SYNTAX

```
display(X)
X - GPUsingle, GPUdouble
```

MODULE NAME
GPUMAT

DESCRIPTION
Prints GPU single information. DISPLAY(X) is called for the object X when the semicolon is not used to terminate a statement.
Compilation supported

EXAMPLE

```
A = rand(10,GPUsingle);
display(A)
A
```

5.4.31 double

double - Converts a GPU variable into a Matlab double precision variable

SYNTAX

```
R = single(X)
X - GPUsingle, GPUdouble
X - Matlab variable
R - single precision Matlab variable
```

MODULE NAME

GPUMAT

DESCRIPTION

`B = SINGLE(X)` converts the content of the GPU variable X into a double precision Matlab array.
Compilation not supported

EXAMPLE

```
A  = rand(100,GPUsingle);
Ah = double(A);
```

5.4.32 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)
```

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

5.4.34 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);
```

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

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

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

MATLAB COMPATIBILITY

Not implemented for complex X.

5.4.38 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

5.4.39 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 >= 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)
```

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

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

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

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

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

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

5.4.46 GPUallocVector

GPUallocVector - Variable allocation on GPU memory

SYNTAX

```
GPUallocVector(P)
P - GPU variable
```

MODULE NAME

GPUMAT

DESCRIPTION

`P = GPUallocVector(P)` allocates the required GPU memory for P. The size of the allocated variable depends on the size of P.
A complex variable is allocated as an interleaved sequence of real and imaginary values. It means that the memory size for a complex on the GPU is `numel(P)*2*SIZE_OF_FLOAT`. It is mandatory to set the size of the variable before calling `GPUallocVector`.
Compilation not supported

EXAMPLE

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

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

5.4.47 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);
```

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

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

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

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

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

5.4.53 GPUcompileAbort

GPUcompileAbort - Aborts the GPUmat compilation.

SYNTAX

```
GPUcompileAbort
```

MODULE NAME

GPUMAT

DESCRIPTION

Aborts the GPUmat compilation. Check the manual for more information.
Compilation not supported

EXAMPLE

```
A = randn(5,GPUsingle); % A is a dummy variable
% Compile function C=myexp(B)
GPUcompileStart('myexp','-f',A)
R = exp(A);
GPUcompileAbort
```

5.4.54 GPUcompileStart

GPUcompileStart - Starts the GPUmat compiler.

SYNTAX

```
GPUcompileStart(NAME, OPTIONS, X1, X2, ..., XN)
NAME - Function name
OPTIONS - Compilation options
X1, X2, ..., XN - GPUsingle, GPUdouble, Matlab variables
```

MODULE NAME

GPUMAT

DESCRIPTION

Starts the GPUmat compiler. Check the manual for more information.
Compilation not supported

EXAMPLE

```
A = randn(5,GPUsingle); % A is a dummy variable
% Compile function C=myexp(B)
GPUcompileStart('myexp','-f',A)
R = exp(A);
GPUcompileStop(R)
```

5.4.55 GPUcompileStop

GPUcompileStop - Stops the GPUmat compiler.

SYNTAX

```
GPUcompileStop(X1, X2, ..., XN)
X1, X2, ..., XN - GPUsingle, GPUdouble, Matlab variables
```

MODULE NAME

GPUMAT

DESCRIPTION

Stops the GPUmat compiler. Check the manual for more information.
Compilation not supported

EXAMPLE

```
A = randn(5,GPUsingle); % A is a dummy variable
% Compile function C=myexp(B)
GPUcompileStart('myexp','-f',A)
R = exp(A);
GPUcompileStop(R)
```

5.4.56 GPUcomplex

GPUcomplex - Construct complex data from real and imaginary components

SYNTAX

```
GPUcomplex(X, R)
GPUcomplex(X,Y,R)
X - GPUsingle, GPUdouble
Y - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

`GPUcomplex(X, R)` is equivalent to `complex(X)`, but result is returned in the input parameter R.
Compilation supported

EXAMPLE

```
RE = rand(10,GPUsingle);
IM = rand(10,GPUsingle);
R = complex(zeros(size(RE), GPUsingle));
GPUcomplex(RE, R);
R = complex(RE, IM);
```

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

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

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

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

5.4.61 GPUdeviceInit

GPUdeviceInit - Initializes a CUDA capable GPU device

SYNTAX

```
GPUdeviceInit(dev)
dev - device number
```

MODULE NAME

GPUMAT

DESCRIPTION

`GPUdeviceInit(dev)` initializes the GPU device `dev`, where `dev` is an integer corresponding to the device number. By using GPUinfo it is possible to see the available devices and the corresponding number

Compilation not supported

EXAMPLE

```
GPUinfo
GPUdeviceInit(0)
```

5.4.62 GPUdouble

GPUdouble - GPUdouble constructor

SYNTAX

```
R = GPUdouble()
R = GPUdouble(A)
A - Either a GPU variable or a Matlab array
R - GPUsingle variable
```

MODULE NAME

GPUMAT

DESCRIPTION

`GPUdouble` is used to create a Matlab variable allocated on the GPU memory. Operations on GPUdouble objects are executed on GPU.

Compilation supported

EXAMPLE

```
GPUdouble(rand(100,100))
Ah = rand(100);
A  = GPUdouble(Ah);
Bh = rand(100) + i*rand(100);
B  = GPUdouble(Bh);
```

5.4.63 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);
```

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

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

5.4.66 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);
```

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

5.4.68 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);
```

5.4.69 GPUgetUserModule

GPUgetUserModule - Returns CUDA (.cubin) module handler

SYNTAX

```
GPUgetUserModule(module_name)
module_name - string
```

MODULE NAME

GPUMAT

DESCRIPTION

`GPUgetUserModule(module_name)` returns the handler of the loaded module `module_name`
Compilation not supported

EXAMPLE

```
%GPUgetUserModule('numerics')
```

5.4.70 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);
```

5.4.71 GPUimag

GPUimag - Imaginary part of complex number

SYNTAX

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

MODULE NAME
NUMERICS

DESCRIPTION
`GPUimag(X, R)` is equivalent to `imag(X)`, but result is returned in the input parameter R.
Compilation supported

EXAMPLE

```
A = rand(10,GPUsingle) + sqrt(-1)*rand(10,GPUsingle);
R = zeros(size(A), GPUsingle);
GPUimag(A, R);
```

5.4.72 GPUinfo

GPUinfo - Prints information about the GPU device

SYNTAX

```
GPUinfo
```

MODULE NAME
GPUMAT

DESCRIPTION
`GPUinfo` displays information about each CUDA capable device installed on the system. Printed information includes total memory and number of processors. `GPUinfo(N)` displays information about the specific device with `index= N`.
Compilation supported

EXAMPLE

```
GPUinfo(0)
```

5.4.73 GPUisDoublePrecision

GPUisDoublePrecision - Check if GPU is double precision

SYNTAX

```
GPUisDoublePrecision
```

MODULE NAME
GPUMAT

DESCRIPTION
`GPUisDoublePrecision` returns 1 if the GPU supports double precision.
Compilation supported

EXAMPLE

```
GPUisDoublePrecision
```

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

5.4.75 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);
```

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

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

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

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

5.4.80 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);
```

5.4.81 GPUmem

GPUmem - Returns the free memory (bytes) on selected GPU device

SYNTAX

```
GPUmem
```

MODULE NAME

GPUMAT

DESCRIPTION

Returns the free memory (bytes) on selected GPU device.
Compilation supported

EXAMPLE

```
GPUmem
GPUmem/1024/1024
```

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

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

5.4.84 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);
```

5.4.85 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);
```

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

5.4.87 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);
```

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

5.4.89 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

5.4.90 GPUrand

GPUrand - GPU pseudorandom generator

SYNTAX

```
GPUrand(R)
R - GPUsingle, GPUdouble
```

MODULE NAME
RAND

DESCRIPTION
`GPUrand(R)` returns in R a matrix containing pseudorandom values drawn from the standard uniform distribution
Compilation supported

EXAMPLE

```
A = ones(5,GPUsingle);
GPUrand(A)
```

5.4.91 GPUrandn

GPUrandn - GPU pseudorandom generator

SYNTAX

```
GPUrandn(R)
R - GPUsingle, GPUdouble
```

MODULE NAME

RAND

DESCRIPTION

`GPUrandn(R)` returns in R a matrix containing pseudorandom values drawn from the normal uniform distribution
Compilation supported

EXAMPLE

```
A = ones(5,GPUsingle);
GPUrandn(A)
```

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

5.4.93 GPUreal

GPUreal - Real part of complex number

SYNTAX

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

MODULE NAME

NUMERICS

DESCRIPTION

`GPUreal(X, R)` is equivalent to `real(X)`, but result is returned in the input parameter R.
Compilation supported

EXAMPLE

```
A = rand(10,GPUsingle) + sqrt(-1)*rand(10,GPUsingle);
R = zeros(size(A), GPUsingle);
GPUreal(A, R);
```

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

5.4.95 GPUsin

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

5.4.96 GPUsingle

GPUsingle - GPUsingle constructor

SYNTAX

```
R = GPUsingle()
R = GPUsingle(A)
A - Either a GPU variable or a Matlab array
R - GPUsingle variable
```

MODULE NAME

GPUMAT

DESCRIPTION

`GPUsingle` is used to create a Matlab variable allocated on the GPU memory. Operations on GPUsingle objects are executed on GPU.
Compilation supported

EXAMPLE

```
Ah = rand(100);
A  = GPUsingle(Ah);
Bh = rand(100) + i*rand(100);
B  = GPUsingle(Bh);
```

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

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

5.4.99 GPUstart

GPUstart - Starts the GPU environment and loads required components

SYNTAX

```
GPUstart
```

MODULE NAME

GPUMAT

DESCRIPTION

Start GPU environment and load required components.
Compilation not supported

EXAMPLE

```
GPUstart
```

5.4.100 GPUstop

GPUstop - Stops the GPU environment

SYNTAX

```
GPUstop
```

MODULE NAME

GPUMAT

DESCRIPTION

Stops GPU environment.
Compilation not supported

5.4.101 GPUsync

GPUsync - Wait until all GPU operations are completed

SYNTAX

```
GPUsync
```

MODULE NAME

GPUMAT

DESCRIPTION

Wait until all GPU operations are completed.
Compilation supported

EXAMPLE

```
A = rand(10,GPUsingle);
B = rand(10,GPUsingle);
tic;A + B;GPUsync;toc;
```

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

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

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

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

5.4.106 gputype_create1

gputype_create1 - GPUtype creation example

SYNTAX

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

MODULE NAME

EXAMPLES:GPUTYPE

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
```

5.4.107 gputype_create2

gputype_create2 - GPUtype creation example

SYNTAX

```
gputype_create2(Ah)
Ah - Matlab array
```

MODULE NAME

EXAMPLES:GPUTYPE

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);
```

5.4.108 gputype_properties

gputype_properties - GPUtype properties example

SYNTAX

```
gputype_properties(X)
X - GPU variable
```

MODULE NAME

EXAMPLES:GPUTYPE

DESCRIPTION

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

EXAMPLE

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

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

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

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

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

5.4.110 GPUuserModuleLoad

GPUuserModuleLoad - Loads CUDA .cubin module

SYNTAX

```
GPUuserModuleLoad(module_name, filename)
module_name - string
filename - string
```

MODULE NAME

GPUMAT

DESCRIPTION

`GPUuserModuleLoad(module_name, filename)` loads the CUDA .cubin module (filename) and assigns to it the name `module_name`. Module handler can be retrieved using `GPUgetUserModule`.
Compilation not supported

EXAMPLE

```
%GPUuserModuleLoad('numerics','.\numerics.cubin')
```

5.4.111 GPUuserModulesInfo

GPUuserModulesInfo - Prints loaded CUDA .cubin modules

SYNTAX

```
GPUuserModulesInfo
```

MODULE NAME

GPUMAT

DESCRIPTION

`GPUuserModulesInfo` displays modules loaded using `GPUuserModuleLoad()`
Compilation not supported

EXAMPLE

```
%GPUuserModulesInfo
```

5.4.112 GPUuserModuleUnload

GPUuserModuleUnload - Unloads CUDA (.cubin) module

SYNTAX

```
GPUuserModuleUnload(module_name)
module_name - string
```

MODULE NAME

GPUMAT

DESCRIPTION

`GPUuserModuleUnload(module_name)` unload the module `module_name`
Compilation not supported

EXAMPLE

```
%GPUuserModuleUnload('numerics')
```

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

5.4.114 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)
```

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

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

5.4.117 imag

imag - Imaginary part of complex number

SYNTAX

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

R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

`R = imag(X)` returns the imaginary part of the elements of X.
Compilation supported

EXAMPLE

```
A = rand(10,GPUsingle) + sqrt(-1)*rand(10,GPUsingle);
R = imag(A);
```

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

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

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

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

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

5.4.123 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)
```

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

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

MATLAB COMPATIBILITY

Not implemented for complex X.

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

MATLAB COMPATIBILITY

Not implemented for complex X.

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

MATLAB COMPATIBILITY

Not implemented for complex X.

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

MATLAB COMPATIBILITY

Not implemented for complex X.

5.4.129 lt

lt - 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)
```

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

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

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

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

MATLAB COMPATIBILITY

Supported only `A / n` where `n` is scalar.

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

5.4.135 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

5.4.136 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

5.4.137 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

5.4.138 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

5.4.139 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

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

5.4.141 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)
```

5.4.142 not

not - Logical NOT

SYNTAX

```
R = ~X
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

`~A (not(A))` performs a logical NOT of input array A.
Compilation supported

EXAMPLE

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

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

5.4.144 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)
```

5.4.145 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)
```

5.4.146 packfC2C

packfC2C - Pack two arrays into an interleaved complex array

SYNTAX

```
PACKFC2C(RE_IDATA, IM_IDATA, ODATA)
RE_IDATA - GPUsingle, real part
IM_IDATA - GPUsingle, imaginary part
ODATA - GPUsingle, complex
```

MODULE NAME
GPUMAT

DESCRIPTION
`PACKFC2C(RE_IDATA, IM_IDATA, ODATA)` pack the values of `RE_IDATA` and `IM_IDATA` into `ODATA` as shown in the example. The type of elements of `ODATA` is complex.
Compilation not supported

5.4.147 packfR2C

packfR2C - Transforms a real array into a complex array with zero complex elements.

SYNTAX

```
PACKFR2C(RE_IDATA, ODATA)
RE_IDATA - GPUsingle, real part
ODATA - GPUsingle, complex
```

MODULE NAME
GPUMAT

DESCRIPTION
`PACKFR2C(RE_IDATA, ODATA)` transforms `RE_IDATA` into a the complex array `ODATA`. The type of elements of `ODATA` is complex. The complex part of `ODATA` is set to zero.
Compilation not supported

5.4.148 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 theyare in the order specified by the vector ORDER.
Compilation supported

EXAMPLE

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

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

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

MATLAB COMPATIBILITY

Implemented for REAL exponents only.

5.4.151 rand

rand - GPU pseudorandom generator

SYNTAX

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

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

MODULE NAME

RAND

DESCRIPTION

`rand(N,GPUsingle)` is an `N-by-N` GPU matrix of values generated with a pseudorandom generator (uniform distribution).
`rand(M,N,GPUsingle)` or `rand([M,N],GPUsingle)` is an `M-by-N` GPU matrix.
`rand(M,N,P,...,GPUsingle)` or `rand([M N P ...,GPUsingle])` is an `M-by-N-by-P-by-...` GPU array of single precision values.
`rand(M,N,P,...,GPUdouble)` or `rand([M N P ...,GPUdouble])` is an `M-by-N-by-P-by-...` GPU array of double precision values.
Compilation supported

EXAMPLE

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

5.4.152 randn

randn - GPU pseudorandom generator

SYNTAX

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

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

MODULE NAME

RAND

DESCRIPTION

`randn(N,GPUsingle)` is an `N-by-N` GPU matrix of values generated with a pseudorandom generator (normal distribution).
`randn(M,N,GPUsingle)` or `randn([M,N],GPUsingle)` is an `M-by-N` GPU matrix.
`randn(M,N,P,...,GPUsingle)` or `randn([M N P ...,GPUsingle])` is an `M-by-N-by-P-by-...` GPU array of single precision values.
`randn(M,N,P,...,GPUdouble)` or `randn([M N P ...,GPUdouble])` is an `M-by-N-by-P-by-...` GPU array of double precision values.
Compilation supported

EXAMPLE

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

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

5.4.154 real

real - Real part of complex number

SYNTAX

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

R - GPUsingle, GPUdouble
```

MODULE NAME

NUMERICS

DESCRIPTION

`R = real(X)` returns the real part of the elements of X.
Compilation supported

EXAMPLE

```
A = rand(10,GPUsingle) + sqrt(-1)*rand(10,GPUsingle);
R = real(A);
```

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

5.4.156 reshape

reshape - Reshape array

SYNTAX

```
R = reshape(X,m,n)
R = reshape(X,m,n,p,...)
R = reshape(X,[m n p ...])
X - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

GPUMAT

DESCRIPTION

`R = reshape(X,m,n)` returns the `m-by-n` matrix R whose elements are taken column-wise from X.
`R = reshape(X,m,n,p,...)` or `B = reshape(A,[m n p ...])` returns an n-dimensional array with the same elements as X but reshaped to have the size `m-by-n-by-p-by-...`.
Compilation not supported

EXAMPLE

```
X = rand(30,1,GPUsingle);
R = reshape(X, 6, 5);
R = reshape(X, [6  5]);
```

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

MATLAB COMPATIBILITY

Not implemented for complex X.

5.4.158 setComplex

setComplex - Set a GPU variable as complex

SYNTAX

```
setComplex(A)
A - GPU variable
```

MODULE NAME

GPUMAT

DESCRIPTION

`setComplex(P)` set the GPU variable P as complex. Should be called before using `GPUallocVector`.
Compilation not supported

EXAMPLE

```
A = GPUsingle();
setSize(A,[10 10]);
setComplex(A);
GPUallocVector(A);
```

5.4.159 setReal

setReal - Set a GPU variable as real

SYNTAX

```
setReal(A)
A - GPU variable
```

MODULE NAME

GPUMAT

DESCRIPTION

`setReal(P)` sets the GPU variable P as real. Should be called before using `GPUallocVector`.
Compilation not supported

EXAMPLE

```
A = GPUsingle();
setSize(A,[10 10]);
setReal(A);
GPUallocVector(A);
```

5.4.160 setSize

setSize - Set GPU variable size

SYNTAX

```
setSize(A,SIZE)
A - GPU variable
```

MODULE NAME

GPUMAT

DESCRIPTION

`setSize(R, SIZE)` set the size of R to SIZE
Compilation not supported

EXAMPLE

```
A = GPUsingle();
setSize(A,[10 10]);
A = GPUdouble();
setSize(A,[10 10]);
```

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

MATLAB COMPATIBILITY

Not implemented for complex X.

5.4.162 single

single - Converts a GPU variable into a Matlab single precision variable

SYNTAX

```
R = single(X)
X - GPU or Matlab variable
R - Matlab variable
```

MODULE NAME

GPUMAT

DESCRIPTION

`B = SINGLE(A)` returns the contents of the GPU variable A into a single precision Matlab array.
Compilation not supported

EXAMPLE

```
A  = rand(100,GPUsingle)
Ah = single(A);
A  = rand(100,GPUdouble)
Ah = single(A);
```

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

MATLAB COMPATIBILITY

Not implemented for complex X.

5.4.164 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 = 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 = rand(10,GPUsingle);
size(X)
X = rand(10,GPUdouble);
size(X)
```

5.4.165 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,':',':');
```

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

MATLAB COMPATIBILITY

Not implemented for complex X.

5.4.167 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)
```

5.4.168 sum

sum - Sum of elements

SYNTAX

```
R = sum(X)
R = sum(X, DIM)
X - GPUsingle, GPUdouble
DIM - integer
R - GPUsingle, GPUdouble
```

MODULE NAME

GPUMAT

DESCRIPTION

`S = SUM(X)` is the sum of the elements of the vector X. `S = SUM(X,DIM)` sums along the dimension DIM.
Note: currently the performance of the `sum(X,DIM)` with `DIM>1` is 3x or 4x better than the `sum(X,DIM)` with `DIM=1`.
Compilation not supported

EXAMPLE

```
X = rand(5,5,GPUsingle)+i*rand(5,5,GPUsingle);
R = sum(X);
E = sum(X,2);
```

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

MATLAB COMPATIBILITY

Not implemented for complex X.

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

MATLAB COMPATIBILITY

Not implemented for complex X.

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

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

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

5.4.174 unpackfC2C

unpackfC2C - Unpack one complex array into two single precision arrays

SYNTAX

```
UNPACKFC2C(IDATA, RE_ODATA, IM_ODATA)
```

MODULE NAME
GPUMAT

DESCRIPTION
`UNPACKFC2C(IDATA, RE_ODATA, IM_ODATA)` unpack the values of `IDATA` into two arrays `RE_ODATA` and `IM_ODATA` as shown in the example. The type of elements of `IDATA` is complex.
Compilation not supported

5.4.175 unpackfC2R

unpackfC2R - Transforms a complex array into a real array discarding the complex part

SYNTAX

```
UNPACKFC2C(IDATA, RE_ODATA)
```

MODULE NAME
GPUMAT

DESCRIPTION
`UNPACKFC2C(IDATA, RE_ODATA)` transforms the complex array `IDATA` into the array `RE_ODATA` discarding the imaginary part. The type of elements of `IDATA` is complex.
Compilation not supported

5.4.176 vertcat

vertcat - Vertical concatenation

SYNTAX

```
R = [X;Y]
X - GPUsingle, GPUdouble
Y - GPUsingle, GPUdouble
R - GPUsingle, GPUdouble
```

MODULE NAME

GPUMAT

DESCRIPTION

`[A;B]` is the vertical concatenation of matrices A and B. A and B must have the same number of columns. Any number of matrices can be concatenated within one pair of brackets.
Compilation not supported

EXAMPLE

```
A = [zeros(10,1,GPUsingle);colon(0,1,10,GPUsingle)'];
```

5.4.177 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)
```

Bibliography

[1] *NVIDIA Cuda Programming Guide*. NVIDIA Corporation.

[2] Cuda. `http://www.nvidia.com/object/cuda_home.html#`.

[3] Gpgpu. `http://www.gpgpu.org`.