



GPULib

Harnessing the Power of the GPU

Version 1.2.0

Tech-X Corporation
5621 Arapahoe Avenue, Suite A
Boulder, CO 80303
<http://www.txcorp.com>
info@txcorp.com



Contents

Table of Contents	1
1 Preface	2
2 Introduction to GPULib	3
2.1 Overview	3
3 Licensing	4
4 Installation	5
4.1 Unix-like Platforms (Linux, Mac OS X)	5
4.2 Windows	7
5 GPULib — Using the GPU for General Calculations	8
5.1 Using GPULib	8
5.1.1 IDL	8
5.1.2 MATLAB	9
5.2 C	10
5.3 Python and Java	10
5.4 General Considerations	10
5.5 IDL Functions Included in GPULib	11
5.6 MATLAB Functions Included in GPULib	12
6 Troubleshooting Guide	17
6.1 No speedup seen in IDL	17
6.2 Attempt to call undefined procedure/function: 'GPUINIT'.	17
6.3 Emulation mode in IDL	18
6.4 Incorrect results in IDL or MATLAB	18
6.5 GPULib produces errors after screen resolution is changed	18
6.6 Errors when using double-precision floating point variables	18
6.7 Unreasonable speedups or no speedup on 64-bit machines	19
6.8 Seeing errors in double precision calculations	19

1 Preface

Names of functions, parameters, and other code samples are denoted in a fixed font — for example, `ls -la`.

Terms requiring a specific definition will appear in *italics* on first use.

GPULib is designed to use NVIDIA hardware and the CUDA libraries on Linux, Mac OS X, and Windows.

GPULib and related documentation copyright 2007-2008, Tech-X Corporation, Boulder, CO. Tech-X ® is a registered trademark of Tech-X Corporation. GPULib is a trademark of Tech-X Corporation.

All other company, product and brand names are the property of their respective owners.

2 Introduction to GPULib

2.1 Overview

In 1965, Intel co-founder Gordon E. Moore coined what became known as Moore's Law – an observation that the number of transistors that can be inexpensively placed on an integrated circuit is increasing exponentially, doubling approximately every two years. This doubling of computer processing power every twenty-four months has allowed scientists and engineers to tackle ever larger and more complex problems in simulation and analysis. As amazing as the performance increases in general purpose processors have been, however, they pale in comparison to modern graphics processors (GPUs). Graphics hardware has been increasing in performance at a much faster rate than CPUs. Additionally, since they are designed for graphics applications, GPUs include a great deal of parallelism and are optimized for vector- and matrix-based calculations by design. Until recently, this power was used mostly for entertainment and computer-generated animations; the scientific community only took advantage of GPU performance in visualization applications.

Recently, NVIDIA has released the Compute Unified Device Architecture (CUDA), a C language environment that allows developers to access the capabilities of GPU hardware to increase performance of their applications. However, as robust and useful as the CUDA environment and tools are, it still requires users to be comfortable in writing C code, familiar with data parallel programming paradigms, and to understand work with concepts such as thread blocks, warp sizes and memory access coalescence.

The main goal of GPULib is to provide a portable, transparent interface to CUDA from within high-level (fourth-generation) languages, such as IDL or MATLAB. By providing an abstract layer on top of CUDA, GPULib allows scientists and engineers to spend more time developing algorithms and solving problems, and less time learning the implementation details of GPU and parallel programming. By providing a higher-level interface to CUDA, we can both speed development as well as bring the technology to a broader range of users.

Using GPULib, vector calculations, matrix transforms, and array manipulations can be off-loaded from the CPU to NVIDIA graphics hardware, easily leveraging the optimizations and speed of the GPU to increase performance of applications. With language bindings for both IDL and MATLAB, GPULib can be integrated quickly into a researcher's existing workflow.

3 Licensing

GPULib is licensed by Tech-X in two ways; non-commercial use is covered by GNU Affero General Public License, version 3. This license can be found in the file `COPYING` provided in the distribution. Commercial licensing of GPULib is covered by the proprietary license included with GPULib, in the file `GPULib-LicenseAgreement.pdf`.

4 Installation

IDL bindings require IDL v6.4 or higher. MATLAB bindings require MATLAB version 7.6 (2008a) or above. All bindings require version 2 or above of the CUDA SDK and Toolkit.

4.1 Unix-like Platforms (Linux, Mac OS X)

To build GPULib on Unix-like platforms, you will need gcc 3.3 or higher. To build GPULib, follow the following steps.

1. Install the CUDA driver, toolkit, and SDK, available from NVIDIA at http://www.nvidia.com/object/cuda_get.html
Detailed instructions on installing the CUDA components are provided at the NVIDIA Web site.
2. You will also need to modify your PATH and LD_LIBRARY_PATH as instructed by installer: for example, if CUDA is installed in

```
/usr/local/cuda
```

you will need to add the following to your `.bashrc` file:

```
export PATH=/usr/local/cuda/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda/lib
```

On Mac OS X, these paths are

```
export PATH=/usr/local/cuda/bin:$PATH
export DYLD_LIBRARY_PATH=/usr/local/cuda/lib
```

3. Extract the GPULib source code with a command similar to

```
tar -xvzf gpulib-1.0.0.tar.gz
```

4. Configure the build by running the configure script. This can be done with the default options with

```
./configure
```

The configure script has several options to modify the build for your particular needs; common options include the following:

- `-prefix=foo` — install GPULib in the location 'foo'. The default is `/usr/local/`.
- `-disable-IDL` — do not build IDL bindings
- `-disable-MATLAB` — do not build MATLAB bindings
- `-enable-emulator` — use emulation mode for CUDA and GPULib. This is useful if you are building on a machine without CUDA-enabled hardware.
- `-with-cuda-dir=foo` — look for CUDA in location 'foo' instead of the default locations
- `-with-rsiidl-bindir=foo` set location of the IDL binary files
- `-with-rsiidl-libdir=foo` set location of the IDL libraries
- `-with-matlab-dir=foo` — look for MATLAB in the directory 'foo' instead of the default locations

For a complete list of available options for configure, run

```
./configure -help
```

Options can be combined as needed. Also, note that at the moment GPULib will not work if configured out of place (VPATH build).

5. Build the software with

```
make
```

and the HTML documentation with

```
make html
```

After the build completes, you can install GPULib with

```
make install
```

6. After the software is installed, you will need to add the location of GPULib to the IDL environment variables. This can be done by editing your `.bashrc` file. For example, if your install of GPULib is located in `/usr/local/gpulib`, you would edit your `.bashrc` to include

```
export IDL_DLM_PATH=+/usr/local/gpulib:<IDL_DEFAULT>
export IDL_PATH=+/usr/local/gpulib:<IDL_DEFAULT>
```

7. To verify the build, you can run a series of tests for IDL, MATLAB, and C. To do this, use

```
make check
```

Tests will only be run for the bindings you enabled, and will display a variety of data. For example, the beginning of the IDL tests will look similar to

Making check in IDL

```
idl gpu_run_test2
```

```
IDL Version 7.0, Mac OS X (darwin i386 m32). (c) 2007, ITT Visual Information Solutions
```

```
Installation number: 209577.
```

```
Licensed for use by: Tech-X Corporation
```

```
% Compiled module: GPUINIT.
```

```
% Loaded DLM: GPULIB.
```

```
% Compiled module: GPUFLTARR.
```

```
% Compiled module: GPUMAKE_ARRAY.
```

```
% Compiled module: GPUGETHANDLE.
```

```
% Compiled module: GPUHANDLE__DEFINE.
```

```
    0.756607    2.33993    0.196372    0.516154    0.0442747    0.839950
```

```
    0.756607    2.33993    0.196372    0.516154    0.0442747    0.839950
```

```
CPU Time =    0.16103697
```

```
GPU Time =    0.0069429874
```

```
Speedup =    23.194190
```

4.2 Windows

Users of GPULib on Windows will need to install the CUDA driver, toolkit, and SDK, available from NVIDIA at

http://www.nvidia.com/object/cuda_get.html

Detailed instructions on installing the CUDA components are provided at the NVIDIA Web site.

For Windows, pre-built DLLs are provided in the `bindist` directory of the distribution. These libraries are built for 32- and 64-bit versions of Windows. The latter is named `gpulib-x64.dll` — if you are running GPULib on a 64-bit Windows system, you should rename this file `gpulib.dll`, replacing the existing file with that name, in the `IDL` or `MATLAB` directory. To use these libraries, you will also need to modify your IDL or MATLAB environment. For example, in the IDL Workbench, you need to set the `IDL_PATH` and `IDL_DLM_PATH` variables by opening the preferences dialog box by choosing Window -> Preferences. From there, open the IDL section of the tree to the left and choose Paths. This will allow you to add the location of GPULib `bindist/IDL/` directory to the environment variables. Make sure to select the check box for each path as well, which will allow IDL to search subdirectories.

For Windows, pre-built DLLs are provided in the `bindist` directory of the distribution. These libraries are built for 32- and 64-bit versions of Windows, each in a separate directory. To use these libraries, you will also need to modify your IDL or MATLAB environment. For example, in the IDL Workbench, you need to set the `IDL_PATH` and `IDL_DLM_PATH` variables by opening the preferences dialog box by choosing Window -> Preferences. From there, open the IDL section of the tree to the left and choose Paths. This will allow you to add the location of GPULib `bindist/IDL/` directory to the environment variables. Make sure to select the check box for each path as well, which will allow IDL to search subdirectories.

The MATLAB bindings also require an additional step for Windows users; you will need to modify the file `gpuInit.m` at lines 52 and 53 to reflect the GPULib and CUDA installation locations. This file includes examples and detailed instructions.

5 GPULib — Using the GPU for General Calculations

5.1 Using GPULib

GPULib provides bindings for several high-level languages; how you access the functions it provides will depend on the language being used.

5.1.1 IDL

To use GPULib from within an IDL script, you first need to call the `gpuinit` procedure. This will load the needed libraries and give you access to GPU-related functions. Once `gpuinit` is loaded, you will have access to the GPULib functions. Full documentation for all functions in the GPU is provided in the `docs/` directory in HTML format. Open the file `index.html` to begin.

To run the sample tests provided for IDL, run

```
IDL> @test2
```

You can also run unit tests for GPULib's IDL bindings by entering

```
IDL> @gpu_run_unittests
```

The IDL GPULib interface consists of both procedure and function interfaces. The following example computes $z = \sin(xy)$ for 360 element vectors x and y using both interfaces. The procedural version is a bit longer than the function form because operations cannot be chained together; each operation, no matter how simple, takes a line:

```
IDL> gpuinit
IDL> x = findgen(360) * !dior
IDL> y = 2 * findgen(360) * !dior
IDL> gpuPutarr, x, x_gpu      ; transfer x to the GPU variable x_gpu
IDL> gpuPutarr, y, y_gpu      ; transfer y
IDL> z_gpu = gpuFltarr(360)   ; allocate GPU variable for result
IDL> gpuMult, x_gpu, y_gpu, z_gpu
IDL> gpuSin, z_gpu, z_gpu
IDL> gpuGetarr, z_gpu, z      ; transfer GPU variable z_gpu back to normal variable z
IDL> plot, z
IDL> gpuFree, [x_gpu, y_gpu, z_gpu]
```

The function form computes $z = \sin(xy)$ in a much more readable form than the procedural interface and does not leak any memory:

```
IDL> gpuinit
IDL> x = findgen(360) * !dior
IDL> y = 2 * findgen(360) * !dior
IDL> z_gpu = gpuSin(gpuMult(x, y))
IDL> plot, gpuGetarr(z_gpu)
IDL> gpuFree, z_gpu
```

The above version is not quite as efficient as the procedure form, though. In general, there are two bottlenecks to fast GPU computation: allocating memory on the GPU and transfer between CPU and GPU. Keeping results

on the GPU and doing many calculations before transferring a final result to the CPU is desirable. Reusing variables for multiple calculations also can eliminate some memory allocation. This was straight-forward to do using the procedure interface by using `z_gpu` for the results of both the multiplication and the sine calculations, but the above function interface example creates temporary GPU variables which are cleaned up automatically, but slow the calculation. The above example could be done using a pre-defined `z_gpu` using the LHS keyword:

```
IDL> gpuinit
IDL> x = findgen(360) * !dtr
IDL> y = 2 * findgen(360) * !dtr
IDL> z_gpu = gpuFltarr(360)
IDL> z_gpu = gpuSin(gpuMult(x, y, LHS=z_gpu), LHS=z_gpu)
IDL> plot, gpuGetarr(z_gpu)
IDL> gpuFree, z_gpu
```

It is possible to program as efficiently as the procedure forms with the function forms of the routines by using the LHS keyword, so make good use of them!

5.1.2 MATLAB

To begin using GPULib in MATLAB, you must initialize by calling the procedure `gpuInit()`. This will load the needed libraries and give you access to GPU-related functions. Note that MATLAB will produce the following warning when calling `gpuInit()`:

```
>> gpuInit()
Warning: Warnings messages were produced while parsing. Check the functions you
intend to use for correctness. Warning text can be viewed using:
[notfound,warnings]=loadlibrary(...)
> In loadlibrary at 374
   In gpuInit at 48
>>
```

This warning is expected, and is not the symptom of an error.

Once `gpuInit()` has been called, you can see an overview of the functionality provided by typing

```
>> help gpuArray
```

To see a complete list of functions available provided to MATLAB by GPULib, use the command

```
>> methods gpuArray -full
```

Help on specific functions can be seen using the dot notation; for example, for details on calculating sine using GPULib, issue the command

```
>> help gpuArray.sin
```

To run the sample tests provided for MATLAB, run

```
>> gpuArrayTest(1000, 0.001)
```

The first parameter of this function is the array size to be used for the tests, and the second is the precision desired for the calculations. Note that this test was designed to be run as part of a larger test suite, and will exit MATLAB upon completion on Unix platforms.

5.2 C

The header files and libraries used by GPULib to provide IDL and MATLAB bindings can also be integrated into C programs directly. An example of how to do this is contained in the `C/` subdirectory of GPULib.

5.3 Python and Java

Python and Java bindings are included in GPULib, but they are currently unsupported. Demo code and documentation for the Python bindings can be found in the `Python/` subdirectory of GPULib, and Java bindings can be found in `Java/`.

5.4 General Considerations

GPULib allows users to take advantage of GPU hardware without learning all of the low-level implementation details, but some care is still needed when designing algorithms to make the most of the hardware. Small changes to source code can often significantly improve (or hinder) performance of algorithms using GPULib.

One of the biggest concerns of which the programmer should be aware is the cost of moving data to or from the GPU. Despite ever-increasing bus speeds, moving data onto and off of the GPU card is still an expensive operation. With this in mind, users of GPULib should minimize such data transfer whenever possible. The basic workflow for algorithms using GPULib recommended by Tech-X is

1. Initial setup and declaration of data
2. Necessary data is moved to the GPU
3. All operations that can be accelerated using GPULib are performed
4. Data is moved back from the GPU to main memory
5. Post-processing of data using functions that cannot be done on the GPU

By reducing data transfers between main memory and the GPU, overhead can be minimized and the overall algorithm can receive increased performance. Similarly, you will see the most performance increase when using the GPU to work on large datasets; for example, while the GPU can calculate $\sin(x)$ faster than the CPU, the cost of moving a single variable to and from the graphics card negates any speed increase that may be obtained. However, if x is an array of 10,000 elements, the GPU would offer substantial performance gains. The inflection point denoting the size of the data at which point it is beneficial to move to the GPU will vary based on what calculations are being done on the data and the specific graphics hardware in use. Scaling studies of the part of your algorithm you wish to accelerate with GPULib may help you decide at what point the overhead of moving to the GPU will be worth the investment.

Finally, GPULib functions are vectorized in both IDL and MATLAB; the means that using operators on vectors is vastly more efficient than the equivalent operations done in a non-vectorized manner. For example, using IDL to calculate sine for a series of numbers by indexing each element in turn:

```
y = fltarr(5)
x = [1, 2, 3, 4, 5]
for i = 0, n_elements(x)-1 do begin
    y[i]=sin(x[i])
endfor
print, y
```

is much less efficient than the same operation done as a vector:

```
z = sin(x)
print, z
```

Similarly, using the vectorized operations in GPULib is much more efficient than indexing individual elements on the graphics hardware. It also has the side benefit of making source code more readable and easier to maintain.

5.5 IDL Functions Included in GPULib

The wrapped GPULib functions in IDL are:

- gpuGetHandle()
- gpuFree, x_gpu [, ERROR=integer]
- gpuPutArr, x [, x_gpu] [, ERROR=integer]
- gpuGetArr [, x_gpu], x [, ERROR=integer]
- gpuAdd, p1, p2, p3 [, p4] [, p5] [, p6] [, /NONBLOCKING] [, ERROR=integer]
- gpuSub, p1, p2, p3 [, p4] [, p5] [, p6] [, /NONBLOCKING] [, ERROR=integer]
- gpuMult, p1, p2, p3 [, p4] [, p5] [, p6] [, /NONBLOCKING] [, ERROR=integer]
- gpuDiv, p1, p2, p3 [, p4] [, p5] [, p6] [, /NONBLOCKING] [, ERROR=integer]
- gpuSqrt, p1, p2 [, p3] [, p4] [, p5] [, p6] [, /NONBLOCKING] [, ERROR=integer]
- gpuExp, p1, p2 [, p3] [, p4] [, p5] [, p6] [, /NONBLOCKING] [, ERROR=integer]
- gpuExp2, p1, p2 [, p3] [, p4] [, p5] [, p6] [, /NONBLOCKING] [, ERROR=integer]
- gpuExp10, p1, p2 [, p3] [, p4] [, p5] [, p6] [, /NONBLOCKING] [, ERROR=integer]
- gpuLog, p1, p2 [, p3] [, p4] [, p5] [, p6] [, /NONBLOCKING] [, ERROR=integer]
- gpuLog2, p1, p2 [, p3] [, p4] [, p5] [, p6] [, /NONBLOCKING] [, ERROR=integer]
- gpuLog10, p1, p2 [, p3] [, p4] [, p5] [, p6] [, /NONBLOCKING] [, ERROR=integer]
- gpuLog1p, p1, p2 [, p3] [, p4] [, p5] [, p6] [, /NONBLOCKING] [, ERROR=integer]
- gpuSin, p1, p2 [, p3] [, p4] [, p5] [, p6] [, /NONBLOCKING] [, ERROR=integer]
- gpuCos, p1, p2 [, p3] [, p4] [, p5] [, p6] [, /NONBLOCKING] [, ERROR=integer]
- gpuTan, p1, p2 [, p3] [, p4] [, p5] [, p6] [, /NONBLOCKING] [, ERROR=integer]
- gpuAsin, p1, p2 [, p3] [, p4] [, p5] [, p6] [, /NONBLOCKING] [, ERROR=integer]
- gpuAcos, p1, p2 [, p3] [, p4] [, p5] [, p6] [, /NONBLOCKING] [, ERROR=integer]
- gpuAtan, p1, p2 [, p3] [, p4] [, p5] [, p6] [, /NONBLOCKING] [, ERROR=integer]
- gpuErf, p1, p2 [, p3] [, p4] [, p5] [, p6] [, /NONBLOCKING] [, ERROR=integer]
- gpuLgamma, p1, p2 [, p3] [, p4] [, p5] [, p6] [, /NONBLOCKING] [, ERROR=integer]
- gpuTgamma, p1, p2 [, p3] [, p4] [, p5] [, p6] [, /NONBLOCKING] [, ERROR=integer]
- gpuLogb, p1, p2 [, p3] [, p4] [, p5] [, p6] [, /NONBLOCKING] [, ERROR=integer]

- `gpuTrunc`, `p1`, `p2` [, `p3`] [, `p4`] [, `p5`] [, `p6`] [, `/NONBLOCKING`] [, `ERROR=integer`]
- `gpuRound`, `p1`, `p2` [, `p3`] [, `p4`] [, `p5`] [, `p6`] [, `/NONBLOCKING`] [, `ERROR=integer`]
- `gpuRint`, `p1`, `p2` [, `p3`] [, `p4`] [, `p5`] [, `p6`] [, `/NONBLOCKING`] [, `ERROR=integer`]
- `gpuNearbyint`, `p1`, `p2` [, `p3`] [, `p4`] [, `p5`] [, `p6`] [, `/NONBLOCKING`] [, `ERROR=integer`]
- `gpuCeil`, `p1`, `p2` [, `p3`] [, `p4`] [, `p5`] [, `p6`] [, `/NONBLOCKING`] [, `ERROR=integer`]
- `gpuFloor`, `p1`, `p2` [, `p3`] [, `p4`] [, `p5`] [, `p6`] [, `/NONBLOCKING`] [, `ERROR=integer`]
- `gpuLrint`, `p1`, `p2` [, `p3`] [, `p4`] [, `p5`] [, `p6`] [, `/NONBLOCKING`] [, `ERROR=integer`]
- `gpuLround`, `p1`, `p2` [, `p3`] [, `p4`] [, `p5`] [, `p6`] [, `/NONBLOCKING`] [, `ERROR=integer`]
- `gpuSignbit`, `p1`, `p2` [, `p3`] [, `p4`] [, `p5`] [, `p6`] [, `/NONBLOCKING`] [, `ERROR=integer`]
- `gpuIsinf`, `p1`, `p2` [, `p3`] [, `p4`] [, `p5`] [, `p6`] [, `/NONBLOCKING`] [, `ERROR=integer`]
- `gpuIsnan`, `p1`, `p2` [, `p3`] [, `p4`] [, `p5`] [, `p6`] [, `/NONBLOCKING`] [, `ERROR=integer`]
- `gpuIsfinite`, `p1`, `p2` [, `p3`] [, `p4`] [, `p5`] [, `p6`] [, `/NONBLOCKING`] [, `ERROR=integer`]
- `gpuFabs`, `p1`, `p2` [, `p3`] [, `p4`] [, `p5`] [, `p6`] [, `/NONBLOCKING`] [, `ERROR=integer`]
- `gpuInterpolate`, `p_gpu`, `x_gpu`, `arg3_gpu` [, `arg4_gpu`] [, `ERROR=integer`] Calculates
- `gpuTotal(x_gpu` [, `ERROR=integer`])
- `gpuCongrid`, `x_gpu`, `nx`, `ny`, `res_gpu` [, `INTERP=integer`] [, `ERROR=integer`]
- `gpuArrRange`, `x_gpu`, `range`, `offset`, `res_gpu` [, `ERROR=integer`]
- `gpunit` [, `/HARDWARE`] [, `/EMULATOR`] [, `/IDL`] [, `ERROR=integer`]

5.6 MATLAB Functions Included in GPULib

The GPULib functions provided in MATLAB are:

- `varargout abs(varargin)`
- `varargout absAT(varargin)`
- `varargout acos(varargin)`
- `varargout acosAT(varargin)`
- `varargout acosh(varargin)`
- `varargout acoshAT(varargin)`
- `varargout asin(varargin)`
- `varargout asinAT(varargin)`
- `varargout asinh(varargin)`
- `varargout asinhAT(varargin)`
- `varargout atan(varargin)`
- `varargout atan2(varargin)`

- `varargout atanAT(varargin)`
- `varargout atanh(varargin)`
- `varargout atanhAT(varargin)`
- `varargout cbrt(varargin)`
- `varargout cbrtAT(varargin)`
- `varargout ceil(varargin)`
- `varargout ceilAT(varargin)`
- `checkStatus(self, identifier, message)`
- `res complex(varargin)`
- `varargout cos(varargin)`
- `varargout cosAT(varargin)`
- `varargout cosh(varargin)`
- `varargout coshAT(varargin)`
- `delete(self)`
- `varargout double(varargin)`
- `varargout eq(varargin)`
- `varargout eqAT(varargin)`
- `varargout erf(varargin)`
- `varargout erfAT(varargin)`
- `varargout erfc(varargin)`
- `varargout exp(varargin)`
- `varargout exp10(varargin)`
- `varargout exp10AT(varargin)`
- `varargout exp2(varargin)`
- `varargout exp2AT(varargin)`
- `varargout expAT(varargin)`
- `varargout expm1(varargin)`
- `varargout expm1AT(varargin)`
- `res find(self)`
- `HM findobj(varargin)`
- `prop findprop(object, propname)`
- `varargout fix(varargin)`
- `varargout fixAT(varargin)`

- `varargout floor(varargin)`
- `varargout floorAT(varargin)`
- `varargout gamma(varargin)`
- `varargout gammaAT(varargin)`
- `varargout gammaln(varargin)`
- `varargout gammalnAT(varargin)`
- `varargout ge(varargin)`
- `varargout geAT(varargin)`
- `x get(self)`
- `self gpuArray(arraySize, gpuMode, allocate, typeID)`
- `varargout gt(varargin)`
- `varargout gtAT(varargin)`
- `varargout hypot(varargin)`
- `varargout imag(varargin)`
- `varargout isfinite(varargin)`
- `varargout isfiniteAT(varargin)`
- `varargout isinf(varargin)`
- `varargout isinfAT(varargin)`
- `varargout isnan(varargin)`
- `varargout isnanAT(varargin)`
- `gpuArrayIsReal isreal(self)`
- `validity isvalid(obj)`
- `varargout ldivide(varargin)`
- `varargout ldivideAT(varargin)`
- `varargout le(varargin)`
- `varargout leAT(varargin)`
- `varargout log(varargin)`
- `varargout log10(varargin)`
- `varargout log10AT(varargin)`
- `varargout log1p(varargin)`
- `varargout log1pAT(varargin)`
- `varargout log2(varargin)`
- `varargout log2AT(varargin)`

- varargout logAT(varargin)
- varargout logb(varargin)
- varargout logbAT(varargin)
- varargout lt(varargin)
- varargout ltAT(varargin)
- result max(self)
- result min(self)
- varargout minus(varargin)
- varargout minusAT(varargin)
- res mldivide(self, other)
- varargout mod(varargin)
- res mpower(self, other)
- res mrdivide(self, other)
- varargout mtimes(varargin)
- varargout ne(varargin)
- varargout neAT(varargin)
- notify(sources, eventname)
- notify(sources, eventname, eventdata)
- varargout plus(varargin)
- varargout plusAT(varargin)
- varargout power(varargin)
- varargout rdivide(varargin)
- varargout rdivideAT(varargin)
- varargout real(varargin)
- varargout rem(varargin)
- varargout round(varargin)
- varargout roundAT(varargin)
- self set(varargin)
- varargout signbit(varargin)
- varargout signbitAT(varargin)
- varargout sin(varargin)
- varargout sinAT(varargin)
- varargout single(varargin)

- `varargout sinh(varargin)`
- `varargout sinhAT(varargin)`
- `varargout sqrt(varargin)`
- `varargout sqrtAT(varargin)`
- `self subsasgn(self, index, other)`
- `res subsref(self, index)`
- `result sum(self)`
- `varargout tan(varargin)`
- `varargout tanAT(varargin)`
- `varargout tanh(varargin)`
- `varargout tanhAT(varargin)`
- `varargout times(varargin)`
- `varargout timesAT(varargin)`
- `res transpose(self)`
- `varargout uminus(varargin)`
- `varargout uplus(varargin)`

6 Troubleshooting Guide

This section contains solutions to common problems seen by users. If you are having issues with GPULib and do not find the solution provided here, please contact Tech-X at support@txcorp.com or call 720-974-1846.

6.1 No speedup seen in IDL

“I’ve installed GPULib successfully, but I see no performance increase when using GPU functions.”

When initializing GPULib in IDL, you should see

```
IDL> gpuinit
% Compiled module: GPUINIT.
% Loaded DLM: GPULIB.
```

If you do not see the notice that the GPULIB DLM has been loaded, the most likely cause is that the environment variables are not correct. If you see this symptom, make sure the IDL_PATH and IDL_DLM_PATH variables set. Make sure these variables point to your GPULib installation, either in your shell initialization files such as .bashrc on Unix platforms, or in the IDL Workbench on Windows.

Another possible cause is that IDL is finding the GPULib libraries, but not the CUDA libraries. To verify that the CUDA libraries are seen, run the following two commands from the IDL command line:

```
IDL> gpuinit
IDL> print, !gpu
```

If successful, these should return

```
IDL> gpuinit
% Compiled module: GPUINIT.
% Loaded DLM: GPULIB.
IDL> help, !gpu, /structures
** Structure !GPU, 2 tags, length=8, data length=6:
  MODE          INT          1
  DEVICE        LONG         0
```

The first number provided by the system variable “gpu”, MODE, shows what mode GPULib is in. This variable will be set to 1 if GPULib is in hardware mode, -1 for CUDA emulation, and 0 for “pure IDL” mode – GPULib will fall back on running commands directly in IDL if no CUDA libraries are found. The second number provided is the hardware identifier, informing the user what GPU hardware is in use by GPULib. If you have CUDA-enabled hardware but the results of print, !gpu indicate you are in emulation mode, ensure that the environment variable LD_LIBRARY_PATH (or DYLIB_LIBRARY_PATH in Mac OS X) includes the location of the CUDA libraries — by default, this is /usr/local/cuda/lib on Unix platforms.

6.2 Attempt to call undefined procedure/function: 'GPUINIT'.

“When initializing GPULib in IDL, I get an error about GPUINIT being undefined.”

If you see an error such as:

```
IDL> gpuinit
% Attempt to call undefined procedure/function: 'GPUINIT'.
% Execution halted at: $MAIN$
```

this indicates that the `IDL_PATH` variable does not include the location of your GPULib installation. Ensure that this variable is set in your environment; it should be similar to

```
IDL_PATH=+/usr/local/gpulib:<IDL_DEFAULT>
```

The “+” prefacing the path to GPULib instructs IDL to search all subdirectories of the provided path.

6.3 Emulation mode in IDL

“I have the CUDA toolchain and GPULib installed, but when I call GPUINIT in IDL, I get the message that GPULib is running in pure IDL emulation.”

The most common cause for this behavior is that IDL couldn’t find the `gpulib.so` library, or couldn’t find the NVIDIA libraries. Make sure that your `IDL_DLM_PATH` points to the location of the `gpulib.so` library. Also, make sure that your dynamic library path (`LD_LIBRARY_PATH` on Linux, `DYLIB_LIBRARY_PATH` on OS X) includes the location of the CUDA libraries (e.g. `/usr/local/cuda/lib`).

If this still does not resolve the problem, run:

```
dml_open, 'gpulib'
```

in a new IDL session. The reported error often helps to pinpoint the origin of the problem.

6.4 Incorrect results in IDL or MATLAB

“When running GPULib code in IDL or MATLAB, I see results that differ from the same calculations made using just the CPU.”

GPU memory is linear, without any memory protection. If a CUDA application contains errors, unexpected results may be seen. (In extreme cases, CUDA code can be made to write directly to the screen, by writing to the buffer of your primary surface.) Memory corruption such as this is usually cured by rebooting the machine, which will re-initialize the hardware. In general, however, if the source code is well-behaved and only use memory that you’ve allocated through one of the CUDA memory allocation routines, this should not happen.

6.5 GPULib produces errors after screen resolution is changed

“I’ve changed my screen resolution while running a CUDA application, and now I’m seeing all kinds of errors in GPULib (or the video display).”

Due to the way memory is allocated by the GPU, changing your screen resolution while having a CUDA application may cause the primary surface to require memory that was owned by CUDA, or give memory to CUDA that contains data left over from the display. If your graphics settings are modified while a GPULib application is running, the results are undetermined.

6.6 Errors when using double-precision floating point variables

“I see errors when using double-precision floating point variables in GPULib.”

Full support for double precision requires CUDA driver version 180.52. If double-precision support is vital for your application, please contact support@txcorp.com and we can provide a workaround.

6.7 Unreasonable speedups or no speedup on 64-bit machines

“When running on a 64-bit platform, I see unexpected results for speedup when running the tests included in GPULib.”

IDL and GPULib must match; if you wish to run IDL in 64-bit mode, the GPULib libraries must also have been built with a 64-bit compiler. If you have 32-bit libraries built, you can run IDL in 32-bit mode on Unix platforms by giving the argument “-32” when starting up IDL.

Pre-built DLLs for IDL and MATLAB are provided in the `bindist/IDL` and `bindist/MATLAB` directories of GPULib. These libraries are built for 32- and 64-bit versions of Windows. The latter is named `gpulib-x64.dll` — if you are running GPULib on a 64-bit Windows system, you should rename this file `gpulib.dll`, replacing the existing file with that name, in the IDL or MATLAB directory.

6.8 Seeing errors in double precision calculations

“When using double precision calculations in GPULib, I’m seeing errors. I also see errors in the double precision unit tests in IDL.”

First, ensure your GPU hardware is capable of running CUDA with double precision. Not all cards can do this; check the NVIDIA Web site for more information. Second, if your hardware is double-enabled, make sure you have the most recent CUDA drivers from NVIDIA. This is also a good idea in general, as NVIDIA is constantly improving their software.