GPULib 1.8.2 User Guide

Tech-X Corporation
12 January 2015

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

Table of Contents

1 Preface **4**

2 Introduction **4**

3 Installation **5**

4 Using GPULib **5**

4.1 Overview 5
4.2 Initialization 5
4.3 Data transfer 6
4.4 Vector Operations 6
4.5 Array subscripting 7
4.6 Functional Form 8
4.7 New operator overloading API for IDL 8.0 and later 9
4.8 Full API documentation 11

5 Troubleshooting **11**

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.

For more information about GPULib, the GPULib blog posts release and other event announcements:

```
http://gpulib.blogspot.com
```

The mailing list is a great resource for asking questions to other GPULib users:

```
https://ice.txcorp.com/mailman/listinfo/gpulib-users
```

Contact `support@txcorp.com` if you are having difficulties using this library.

2 Introduction

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. 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. GPULib can be integrated quickly into a researcher's existing workflow.

3 Installation

See the `README.txt` file in the root of the distribution for detailed installation instructions.

4 Using GPULib

4.1 Overview

GPULib is an IDL library for accelerating array operations using NVIDIA's CUDA, if appropriate hardware is installed.

The IDL bindings for the library are designed to enable developers to write software that executes both on systems with a CUDA enabled GPU and those without (though, of course, there will be no acceleration in that case).

In order to use the library efficiently, developers must be aware of the different memories used by the CPU and the GPU. Data has to be explicitly transferred to and from the GPU memory, which can be a time consuming operation. The IDL bindings for the GPULib can hide the code for these explicit data transfer calls for convenience, but one should be aware that this can severely impact the overall performance.

4.2 Initialization

GPUlib is initialized by a call to `GPUINIT`. This call detects whether a GPU or a GPU emulator is available. If none is available, all the GPU operations will be emulated in IDL. Requests to use a particular GPU device

or software emulation can be passed as parameters to `GPUINIT`.

After running `GPUINIT`, a `!gpu` system variable will be available. This structure contains the mode (`1` indicates using a physical GPU, `0` indicates using pure IDL emulation, and `-1` indicates using device emulator), so if you want to check that you are running on one of the GPUs in your system, check `!gpu.mode` to make sure it is `1`:

```
IDL> print, !gpu.mode
1
```

If you cannot get `GPUINIT` to recognize your GPU, make sure that your `IDL_DLM_PATH` points to the location of the `gpulib.dlm` file.

4.3 Data transfer

Users can explicitly transfer data between GPU and host memory by using `GPUPUTARR` and `GPUGETARR`. GPU variables can either be allocated explicitly or they are allocated on the fly during transfer, e.g., if `dx` is undefined and the user issues the following command:

```
gpuPutArr, findgen(100), dx
```

then a correctly sized `dx` variable will be allocated. It is the user's responsibility to clean up the allocated space via `gpuFree, dx` (but since GPU variables are objects, they are freed by the automatic garbage collection in IDL 8.0 or later).

4.4 Vector Operations

In order to get optimal performance using GPUlib, one should perform multiple operations on long vectors and only rarely transfer data between the GPU and CPU.

There are multiple syntactical forms for vector operations on the GPU: short and long forms, procedure and function forms. For example, the procedure short form for `GPUADD` is:

```
gpuAdd, dx, dy, dresult
```

which adds the elements of the vector under `dx` and `dy` and stores the result in `dresult`.

If `dresult` is an undefined variable at the time of the call, it will be allocated on the GPU. Likewise, if `dx` or `dy` are not GPU variables, then they will be transferred to temporary variables on the GPU.

For example:

```
gpuAdd, findgen(10), findgen(10) + 5, dresult
```

will transfer `findgen(10)` and `findgen(10) + 5` to the GPU, compute their sum on the GPU, and store the result in the `dresult` variable.

4.5 Array subscripting

GPULib provides mechanisms to subscript vectors and arrays on the GPU via the `GPUSUBARR` and `SPUSUBSCRIPT` routines. Depending on the dimensionality of the object, `GPUSUBARR` either takes one or two indices for both the source and destination arrays. These indices can either be scalars or two element arrays specifying the lower and upper bound of a segment (no stride is allowed). An index or upper bound of `-1` corresponds to IDL's `*` indicating all the elements of a dimension.

For example, the following operation:

```
gpuSubArr, dx, [3, 5], -1, dy, -1, [3, 5]
```

is equivalent to the following indexing for CPU variables `x` and `y`:

```
x[*, 3:5] = y[3:5, *]
```

(This assumes that the first dimension of `x` has the same number of elements as the second dimension of `y`.)

The `GPUSUBSCRIPT` routine is for indexing an array with an index array. For example, to do the equivalent of `dsub = dx[[0, 4, 7]]`, do:

```
IDL> dx = gpuputarr(2. * findgen(10))
IDL> dind = gpuputarr([0., 4., 7.])
IDL> dsub = gpusubscript(dind, dx)
IDL> sub = gpugetarr(dsub)
IDL> print, sub
```

```
      0.00000      8.00000      14.0000
```

You can also do the subscripting on the left hand side of the assignment, with the LEFTHANDSIDE keyword:

```
IDL> dsub = gpuputarr([-1., -1., -1.], lhs=dsub)
IDL> dx = gpusubscript(dind, dsub, lhs=dx, /leftsubscript)
IDL> x = gpugetarr(dx)
IDL> print, x
     -1.00000      2.00000      4.00000      6.00000     -1.00000
      10.0000      12.0000     -1.00000      16.0000      18.0000
```

This is equivalent to:

```
dx[[0, 4, 7]] = [-1., -1., -1.]
```

4.6 Functional Form

In addition to the procedural form described here, GPULib also provides a functional form for most operations. So instead of writing:

```
gpuAdd, dx, dy, dz
```

one can more intuitively write:

```
dz = gpuAdd(dx, dy)
```

The main disadvantage of the functional form is that in the above form, GPULib needs to allocate temporary storage on the GPU, as the location of the result is not know at the time of invocation of GPUADD. This can be a time consuming step (compared to the extremely fast execution of the addition).

In order to avoid this extra allocation you can pre-allocate variables to use as temporary variables and provide them via the LHS keyword:

```
IDL> dx = gpufindgen(10)
IDL> dy = gpufindgen(10)
IDL> dz = gpufltarr(10)
IDL> dz = gpuAdd(dx, dy, LHS=dz)
```

Without the LHS keyword, dz would have been allocated anyway, but often a single temporary variable can be used in many calculations,

4.7 New operator overloading API for IDL 8.0 and later

IDL 8.0 introduced operator overloading to IDL objects. While the old GPULib API still works, IDL 8.0 users can make use of the new API with operator overloading.

The gpulib_new_api.pro example in the demos/new_api directory shows some simple examples of how to use operator overloading with GPULib. We'll provide an explanation of the example.

First, we initialize GPULib with GPUINIT. This is required before calling any GPULib routines. Arguments to use a particular graphics card (if multiple are present) or emulation mode (hardware, software, or pure IDL) are available:

```
IDL> gpuinit
```

Next, we create two simple arrays on the GPU:

```
IDL> dx = gpuFindgen(10)
IDL> dy = gpuFindgen(10)
```

Instead of calling GPUADD to add these arrays, the + operator can now be used. Most common arithmetic operators are available now: +, -, *, /, ^, #, ##, <, >, gt, ge, lt, le, ne, eq, and mod:

```
IDL> dz = dx + dy
```

The standard IDL HELP routine will give more useful help on GPULib variables now as well:

```
IDL> help, dz
DZ              GPUFLOAT  = Array[10]
```

The PRINT routine will print a GPULib variable, handy for debugging:

```
IDL> print, dz
      0.00000      2.00000      4.00000      6.00000      8.00000
      10.0000      12.0000      14.0000      16.0000      18.0000
```

The SIZE routine will give the same output as for a regular IDL array:

```
IDL> print, size(dz)
           1          10          11          10
```

A FOREACH loop can be used to create a view into each row of a 2-dimensional GPU array or each element of a 1-dimensional GPU array:

```
IDL> darr = gpuFindgen(5, 5)
IDL> foreach drow, darr, r do print, r, drow
   0      0.00000      1.00000      2.00000      3.00000      4.00000
   1      5.00000      6.00000      7.00000      8.00000      9.00000
   2      10.0000      11.0000      12.0000      13.0000      14.0000
   3      15.0000      16.0000      17.0000      18.0000      19.0000
   4      20.0000      21.0000      22.0000      23.0000      24.0000
```

Use brackets like regular IDL arrays:

```
IDL> print, darr[*, 1]
      5.00000      6.00000      7.00000      8.00000      9.00000
```

Brackets can be used to assign to GPU arrays also (where the right hand side of the assignment is either a regular IDL array or another GPU variable):

```
IDL> darr[*, 1] = findgen(5)
IDL> print, darr
      0.00000      1.00000      2.00000      3.00000      4.00000
      0.00000      1.00000      2.00000      3.00000      4.00000
      10.0000      11.0000      12.0000      13.0000      14.0000
      15.0000      16.0000      17.0000      18.0000      19.0000
      20.0000      21.0000      22.0000      23.0000      24.0000
```

Variables can be released with GPUFREE, but automatic garbage collection in IDL 8.0 will also take care of them:

```
IDL> gpuFree, [dx, dy, dz, darr]
```

4.8 Full API documentation

See the generated documentation at `index.html` in the `docs/IDL/html-docs` directory of the GPULib distribution.

5 Troubleshooting

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

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 pointing to the GPULib installation are not correctly set. If you see this symptom, make sure the `IDL_PATH` and `IDL_DLM_PATH` variables point to your GPULib installation, either in your shell initialization files such as `.bashrc` on Unix platforms or in the IDL Workbench.

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 `mode` field of the `!gpu` system variable indicates whether GPULib is using hardware or emulation mode. 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, indicating what GPU hardware is in use by GPULib. If you have CUDA-enabled hardware but `!gpu` shows 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.

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.

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` (`gpulib.dll` on Windows) 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`.

It can be useful to examine `!error_state.msg` immediately after running `GPUINIT` to help diagnose this problem:

```
IDL> gpuinit
% Compiled module: GPUINIT.
Welcome to GPULib 1.8.2 (Revision: 3815)
% GPUINIT: using pure IDL emulation
IDL> print, !error_state.msg
CUDASETDEVICE: Error loading sharable executable.
                 Symbol: IDL_Load, File = gpulib.so
                 gpulib.so: cannot open shared object
                  file: No such file or directory
```

For example, if the above message was printed, check to make sure `gpulib.so` was in your `IDL_DLM_PATH`.

If this still does not resolve the problem, run the following in a new IDL session:

```
IDL> dlm_load, 'gpulib'
% Loaded DLM: GPULIB.
```

If the module is not loaded, the reported error often helps to pinpoint the origin of the problem.

When running GPULib code in IDL, 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.

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.

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

Full support for double precision requires CUDA driver version 180.52 and a graphics card capable of compute capability 1.3. See NVIDIA's listing of graphics cards for the compute capabilities of all of their cards:

```
http://www.nvidia.com/content/cuda/cuda-gpus.html
```

When running on a 64-bit Windows, I see error messages indicating that IDL can't load a 64-bit DLL.

64-bit Windows is capable of running the 32-bit version of IDL. Make sure to start 64-bit IDL when launching from the Windows Start menu, not 'IDL (32-bit)'.

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.