Gedae Functional Interface to Launch Package Programmers Manual

January 2008

Last revision November 2, 2009

Address:	Gedae, Inc.
	1247 N Church St, STE 5
	Moorestown, NJ 08057
Telephone:	(856) 231-4458
FAX:	(856) 231-1403
Internet:	www.gedae.com

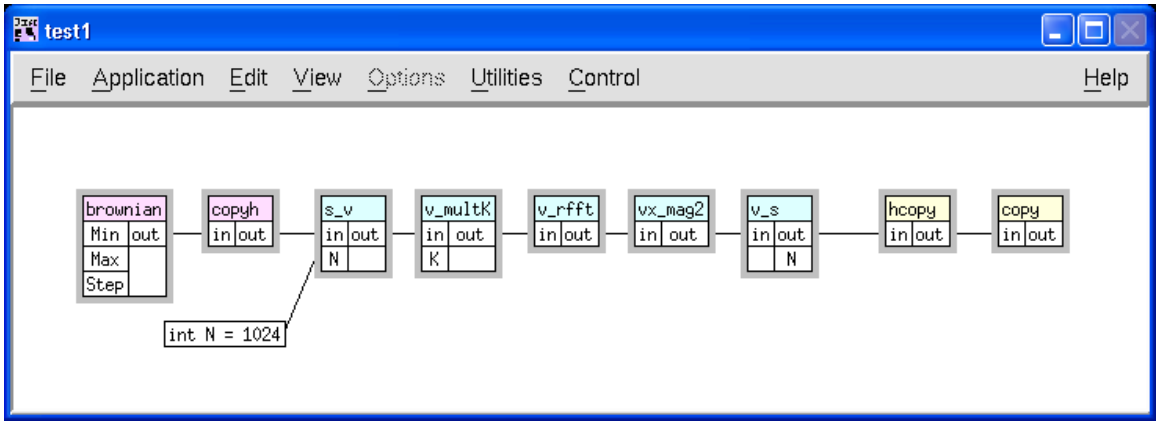
Table of Contents

Introduction

A user can convert a Gedae graph into a simple function call that can be linked with any external program. This capability allows users to replace functions in their current code with a call to a Gedae created launch package that runs on a high performance processor. Replacing functions with Gedae generated functions provides immediate acceleration of a user's application with minimal impact to their current code.

How to Create an Application Graph That Implements a Particular Function

We will illustrate how a Gedae graph can be turned into a function call with the following example. The blue Group in the graph below scales its input vector by a parameter K, takes the real fft of the scaled vector and creates an output vector that is the squared magnitude of the fft output.



The blue Group can be converted into an object file that implements the function call with prototype:

```
void test1(
```

```
float *in /* s_v<in */,
int n_in,
float *out /* v_s>out */,
int n_out,
float K /* v_multK<K */);
```

Where

- `in` is the input vector data,
- `n_in` is the number of scalar samples in the input vectors
- `out` is the output vector data
- `n_out` is the number of scalar samples in the output vectors
- `K` is the scaling parameter to the `v_multK` box.

In the above example, the values of n_in must be a multiple of 1024, and n_out must be a multiple of 512 (since the output vector size of the v_rfft is half the input vector size). If 5 vectors are to be processed in one call to the function, then n_in will be 1024*5 = 5120 and n_out will be 512*5 = 2560.

The function interface that gets generated is defined by "host" boundaries to the group and by user settable parameters. In the above graph, the host boundaries are placed in the graph by the copyh and hcopy boxes. A list of the boxes provided in embeddable/stream/host is:

Primitive	**Data Type**	**Input/Output**
embeddable/stream/host/copyh	float	input
embeddable/stream/host/i_copyh	int	input
embeddable/stream/host/x_copyh	complex	input

embeddable/stream/host/hcopy	float	output
embeddable/stream/host/i_hcopy	int	output
embeddable/stream/host/x_hcopy	complex	output

Each host boundary – input or output adds a pointer argument to the generated function call and an integer parameter saying how much data is being supplied or how much data is expected as the return value. The parameters are named after the

All host boundary boxes are passed scalar data. If a user desires the function to process vectors or matrices, then an appropriate conversion function must be added. The following conversion functions convert scalars to/from higher level data types:

Primitive	**Data Type**	**Conversion Type**
embeddable/vector/s_v	float	scalar to vector
embeddable/vector/v_s	float	vector to scalar
embeddable/vector/complex/x_vx	complex	scalar to vector
embeddable/vector/complex/vx_x	complex	vector to scalar
embeddable/matrix/s_m	float	scalar to matrix
embeddable/matrix/m_s	float	matrix to scalar
embeddable/matrix/complex/x_mx	complex	scalar to matrix
embeddable/matrix/complex/mx_x	complex	matrix to scalar

Again, while not all possible conversion functions are available, it is easy to see how to create new conversion functions from the ones provided.

Since the example above has a floating point vector as an input the copyh box is followed by an s_v box to convert the input scalar stream to a vector. The output vector is converted to a scalar

host boundary with a v_s box followed by an hcopy. The input vector size is hard-coded to 1024 in the above example, and the resulting output vector will be of size 512.

The only parameter in the graph is the user settable parameter value v_multK<K and this appears as the last parameter to the generated function test. Parameters that are hardwired on the graph such as the s_v<N input – or are not user settable because they affect the static data flow of the graph – are not made part of the function interface.

The Function Generated for a Graph

In order to create a graph that implements the function you desire, you need to know the rules for how the functional interface is created from the graph. The code-generation process (described below) will cause a function to be created that is named after the graph. So if the graph is found in `test/func_codegen/test1`, then the function created will be named `test1`. Two arguments are added for each input host boundary. The first argument is a pointer of the data type of the input. The second argument is how many scalar samples of that type are to be passed to the function. For the s_v<in input of the example graph, the parameters that are added are:

```
float *in /* s_v<in */,
int n_in,
```

After all of the input host boundary arguments are added to the function, the output host boundary arguments are added to the function. Output arguments are formed in the same way as input arguments. The arguments for the v_s>out host boundary are:

```
float *out /* v_s>out */,
int n_out,
```

Following the host inputs and host outputs are the parameter values to the graph. For scalar parameters these are given as a parameter of the given type. For example, the v_multK<K input argument is:

```
float K /* v_multK<K */
```

Parameters are distinguished from host boundaries in that the parameter value is set once at the beginning of function execution, while the input and output to the function can be as large as desired. All inputs and outputs should be balanced as dictated by the dataflow of the application. They must be provided in multiples of the vector or matrix sizes dictated by the token types of the application. Since the test1 graph takes input vectors of size 1024 and produces an output vector of size 512, the following are legitimate calls to the test1 function:

```
test1(in,1024,out,512,37);      /* process 1 vector with K =  37 */
test1(in,204800,out,102400,23); /* process 200 vectors with K = 23 */
```

How to Code-generate the Function from the Graph

Once the graph has been created, the user can partition and map the launch package using the Group Control dialog in the usual way. Changing the mapping and other Group control parameters will not change the behavior of the function but distributing the function to run on multiple high performance processors will make the function run faster. The user need not set any optimization parameters, which allows the user to quickly verify the correctness of the generated function; at a later time, the user can optimize the graph and produce a faster function that produces identical results.

Once all the group control parameters have been set to create an optimal implementation, the user can create the linkable object that provides the function call for the graph. To do this, the user must bring up the Launch Package Dialog, set the launch package directory and press the Make button. In addition to building all the other launch package objects, as for previous versions of Gedae, the make will also add a linkable object to the launch package directory that implements three functions.

If the make fails it may be due to a problem using the standard compilation settings. Also, the user may which to override the standard settings for optimization or debugging purposes. The compilation settings are set by files in the $GEDAE/<hostname>/<target_name>/bin directory but can be overridden by the user by creating files in the $GEDAEHOME/<hostname>/embedded/<target_name> directory. (where $GEDAEHOME/<hostname> is the directory from which the user runs the Gedae development environment).

target host type	**host description**	**<host_name>**	**<target_name>**
NT	Windows PC	`nt`	`ent`
Redhat	Linux PC	`redhat`	`eredhat`
PPU	Linux Cell BE/ PPU	`redhat`	`ppu`

.

The $GEDAE/<hostname>/<targetname>/bin/std_make_info sets most of the compilation parameters. Environment variable settings in this file provide defaults, and these defaults can be overridden by commenting out lines in the $GEDAEHOME/<hostname>/embedded/<targetname>/runtime_make_info.

The final link into an object suitable for linking with the users final program is done by $GEDAE/<hostname>/<targetname>/bin/link_func (or link_func.bat for Windows systems). The user can provide their own version of this library by create a file $GEDAEHOME/<hostname>/embedded/<targetname>/link_func for non-windows systems and $GEDAEHOME/nt/embedded/ent/link_func.bat for Windows systems.

Code-generation Products

The file <graphname>.h contains the prototypes for functions code-generated for the application graph, where <graphname> is the name of the application graph. For example, for the test/func_codegen/test1 graph, the file test1.h is placed in the launch package directory. This file contains the prototypes for the initialization, startup, execution, kill and free function which are:

```
void init_test1(void);

void start_test1(void);

void kill_test1(void);

void free_test1(void);

void test1(

  float *in /* s_v<in */,

  int n_in,

  float *out /* v_s>out */,

  int n_out,

  float K /* v_multK<K */);
```

The initialization and start functions are called before execution. The kill and free functions are called after execution. The execution function (for example, test1) can be called multiple times, and the state of the execution is retained between invocations. Startup and tear down are separated into four functions to allow multiple launch packages to be used in the same program. This feature is illustrated in the examples below.

This file should get included in the user's application code that will call these functions. The actual object file or library that gets linked with the user's application is found in different places for different host BSPs. The following table shows where the linkable objects are found relative to the launch package directory for the three supported host types:

host type	**host description**	**linkable object**
NT	Windows PC	`ent/nt/<graphname>.lib`
Redhat	Linux PC	`eredhat/redhat/<graphname>.o`
PPU	Linux Cell/BE	`ppu/ppu/<graphname>.o`

Using the Generated Functions

To use the generated functions in a C program application, the user must call the start and init functions once at the beginning of the application, call the execution function as many times as desired during the application execution and then call the kill and free functions prior to application exit. For example, the test1 graph function can be called as:

```
#include <stdio.h>
#include <math.h>
#include <test1.h>
main() {
  float z[1024];
  float x[512];
  for (i=0; i<1024; i++) { z[i] = cos(0.1*i); }
  init_test1();
  start_test1();
  test1(z,1024,x,512,10);
…
  test1(z,1024,x,512,20);
…
  test1(z,1024,x,512,40);
…
  kill_test1();
```

```
  free_test1();
}
```

If the file using the `test1.h` include file is a C++ source file rather than a C source file, then instead of

```
#include <test1.h>
```

The file should contain:

```
extern "C" {
#include <test1.h>
}
```

If multiple launch packages are being used, the init and free functions are still called only once at the beginning and end, but the start and kill functions are called between launch packages to perform the context switch. This procedure can also be used if the launch package ties up system resources that must be used by other parts of the application, such as large memory buffers or coprocessors like the SPEs. The following example illustrates switching between two launch packages:

```
#include <stdio.h>
#include <math.h>
#include <test1.h>
#include <test2.h>
main() {
  float z[1024];
```

```
  float x[512];
  for (i=0; i<1024; i++) { z[i] = cos(0.1*i); }
  init_test1();
  start_test1();
  test1(z,1024,x,512,10);
  test1(z,1024,x,512,20);
  test1(z,1024,x,512,40);
  kill_test1();
  start_test2();
  test2(z,1024,x,512,10);
  test2(z,1024,x,512,20);
  test2(z,1024,x,512,40);
  kill_test2();
  free_test1();
  free_test2();
}
```

Linking the Final Application

The method of linking the function code-generated for the application differs depending on the host type of the application. Here are the link lines for NT, Redhat and PPU hosts.

Host type NT (command-line):

```
link /out:<userapp>.exe <users_objects> -LIBPATH
<launchdir>/ent/nt <graphname>.lib –nodefaultlib:libc      –
nodefaultlib:libcmt
```

Host type NT (Visual Studio):

In the Linker's configuration properties, put `<launchdir>/ent/nt` in Additional Library Directories in the "General" frame, put `<graphname>.lib` in Additional Dependencies in the "Input" frame and put `libc;libcmt` in Ignore Specific Library in the "Input" frame.

Host type Redhat:

```
gcc -o <userapp> <users_objects>
<launchdir>/eredhat/redhat/<graphname>.o -lpthread –lrt
<other_libs>
```

Host type PPU:

```
ppu-gcc -o <userapp> <users_objects>
<launchdir>/ppu/ppu/<graphname>.o -m32 –lpthread –lrt
<other_libs>
```

Where:

<userapp> is the name of the application executable the user is generating

<users_objects> is the list of object files and libraries that the user links to create the application. One of the objects includes the call to the startup, execution and kill functions.

<launchdir> is the launch package directory name in which the user's application has been created.

<graphname> is the name of the user's Gedae graph

<other_libs> is a list of libraries that may be needed to link the primitives included in the Gedae graph. For example, a primitive that calls the math.h function cos will require –lm to be linked with it for the Redhat and PPU hosts.