DEFG:

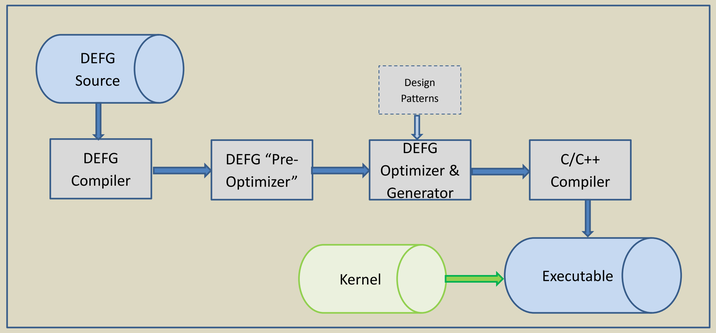
Declarative Framework for GPUs

Robert Senser and Tom Altman
Department of Computer Science and Engineering
University of Colorado Denver
Email: robert.senser@ucdenver.edu

University of Colorado Denver

What is DEFG?

DEFG is a framework, based on a straight-forward domain-specific computer language and its design patterns, used to generate the CPU side of high-performance GPU applications.



Writing the CPU portion of a GPU application can be time consuming, tedious, and error prone. Through the use of declarative statements and design patterns, DEFG instead generates the CPU code needed to manage the GPU code.

The current version of DEFG produces OpenCL C++ code and we are anticipating enhancing DEFG to produce CUDA code.

The DEFG design patterns give DEFG the unique ability to generate the complex code needed to "link" the CPU and GPU by using known patterns of interaction between the hardware units.

Our research goal is to identify these GPU design patterns and then to design tools to utilize these patterns in a declarative manner.

A DEFG Example:

This:

```
01. declare application  sobel
02.   declare integer Xdim (0)
03.   declare integer Ydim (0)
04.   declare integer BUF_SIZE (0)
05.   declare gpu gpuone ( any )
06.   declare kernel  sobel_filter SobelFilter_Kernels  ( [[ 2D,Xdim,Ydim ]] )
07.   declare integer buffer image1 ( BUF_SIZE )
08.               integer buffer image2 ( BUF_SIZE )
09.   call init_input (image1(in) Xdim (out) Ydim (out) BUF_SIZE(out))
10.   execute run1 sobel_filter ( image1(in) image2(out) )
11.   call disp_output (image2(in) Xdim (in) Ydim (in) )
12. end
```

Produces code like this:

```
cl_mem buffer_image1 = clCreateBuffer(context, CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR, (BUF_SIZE *
sizeof(int)),(void *) image1, &status);
if (status != CL_SUCCESS) { handle error }
status = clSetKernelArg(sobel_filter, 0, sizeof(cl_mem), (void *)&buffer_image1);
if (status != CL_SUCCESS) { handle error }
cl_mem buffer_image2 = clCreateBuffer(context, CL_MEM_WRITE_ONLY, (BUF_SIZE * sizeof(int)),(void *) NULL,
&status);
if (status != CL_SUCCESS) { handle error }
status = clSetKernelArg(sobel_filter, 1, sizeof(cl_mem), (void *)&buffer_image2);
if (status != CL_SUCCESS) { handle error }
// *** execution
size_t global_work_size[2]; global_work_size[0] = Xdim ; global_work_size[1] = Ydim ;
status = clEnqueueNDRangeKernel(commandQueue, sobel_filter, 2, NULL, global_work_size, NULL, 0, NULL, NULL);
if (status != CL_SUCCESS) { handle error }
// *** result buffers
status = clEnqueueReadBuffer(commandQueue, buffer_image2, CL_TRUE, 0, BUF_SIZE * sizeof(int), image2, 0,
NULL, NULL);
if (status != CL_SUCCESS) { handle error }
```

DEFG Design Patterns

The design patterns are groupings of DEFG language constructs and predefined code generation patterns. These groupings permit the GPU developer to quickly generate the CPU code for many GPU applications. Different design patterns may be combined to produce complex results.

Execute kernel 'N' times:

```
sequence NODE_CNT times
  execute run1  FloydWarshallPass  ( buffer1(inout) buffer2(inout) …  )
```

Execute kernel until condition:

```
loop
  execute part1  kernel1  ( graph_nodes(in) …  NODE_CNT(in)  )
  set STOP (0)
  execute part2  kernel2  ( graph_mask(inout) …  STOP (inout) )
while STOP eq 0
```

Execute kernel on multiple GPU cards:

```
multi_exec run1 multi_filter ( image1(in) image2(out) )
/* of course kernel multi_filter is designed for multiple GPU execution */
```

The DEFG constructs such as the 'sequence' and 'loop' operations are well understood by the optimizer and generator such that minimal and efficient code is produced.

The DEFG optimizer and generator also produce efficient code that has the movement of buffers between the devices minimized. For example with the 'loop' example above, any buffers used exclusively on the GPU between 'kernel1' and 'kernel2' are kept entirely on the GPU.

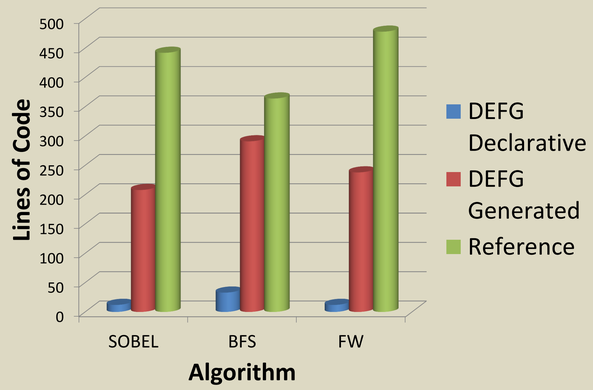
Our DEFG Findings

- Compared to hand-written applications
  - Declarative approach requires less coding
  - Similar performance
  - Design patterns facilitate automated optimizations
- Also, readable C++ programs are generated
  - Embeddable into larger systems
  - Starting point for learning or enhancement

Proposed Next Steps

- Produce more interesting use cases
- Extended support for multiple GPU card processing
- CUDA support
- Automated generation of certain kernel types
- Long term: extendable design patterns

Lines of Code Comparison



Run-Time Performance Comparison