**What is DEFG?**

DEFG is a framework, based on a straightforward 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.

**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

---

**A DEFG Example:**

This:

1. declare application sobel
2. declare integer Xdim (0)
3. declare integer Ydim (0)
4. declare integer BUF_SIZE (0)
5. declare gpu gpus ( any )
6. declare kernel sobel_filter sobelfilter_Kernels ( || 2D,Xdim,Ydim || )
7. declare integer buffer image1 ( BUF_SIZE )
8. declare integer buffer image2 ( BUF_SIZE )
9. 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:

```c
if ( status != CL_SUCCESS ) { handle error }
cl_mem buffer_image2 = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, BUF_SIZE * sizeof(int), NULL, &status);
if ( status != CL_SUCCESS ) { handle error }
cl_mem buffer_image1 = clCreateBuffer(context, CL_MEM_READ_WRITE, BUF_SIZE * sizeof(int), NULL, &status);
if ( status != CL_SUCCESS ) { handle error }

clMemBufferImage2 = std::make_shared<cl_mem>(buffer_image2);
clMemBufferImage1 = std::make_shared<cl_mem>(buffer_image1);

clEnqueueNDRangeKernel(commandQueue, multi_filter, 1, NULL, (0), (0), (0), NULL, &status);
if ( status != CL_SUCCESS ) { handle error }
clEnqueueNDRangeKernel(commandQueue, multi_exec, 1, NULL, (1), (0), (0), NULL, &status);
if ( status != CL_SUCCESS ) { handle error }

clEnqueueReadBuffer(commandQueue, buffer_image2, CL_TRUE, 0, BUF_SIZE * sizeof(int), image2, 0, NULL, NULL);
if ( status != CL_SUCCESS ) { handle error }
clEnqueueReadBuffer(commandQueue, buffer_image1, CL_TRUE, 0, BUF_SIZE * sizeof(int), image1, 0, NULL, NULL);
if ( status != CL_SUCCESS ) { handle error }
```

**Lines of Code Comparison**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>DEFG</th>
<th>Declarative</th>
<th>DEFG Generated</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOBEL</td>
<td>BFS</td>
<td>BFS-4096</td>
<td>BFS-65536</td>
<td>SOBEL</td>
</tr>
<tr>
<td>BFS-4096</td>
<td>BFS-65536</td>
<td>FW</td>
<td></td>
<td>SOBEL</td>
</tr>
<tr>
<td>FW</td>
<td>BFS-65536</td>
<td>BFS-4096</td>
<td></td>
<td>SOBEL</td>
</tr>
</tbody>
</table>

**Run-Time Performance Comparison**

Platform and Version:

- BFS-4096
- BFS-65536
- FW
- SOBEL

---

**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**

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

**Execute kernel until condition**

```c
loop
execute part1 kernel1 ( graph_nodes(in) ... NODE_CNT(in) )
sel STOP (0)
exeucte part2 kernel2 ( graph_mask(inout) ... STOP (inout) )
while STOP eq 0
```

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.