# DEF-G: Declarative Framework for GPU Environment

**Robert Senser and Tom Altman**
Department of Computer Science and Engineering, University of Colorado Denver, Denver, CO

**Abstract -** *DEF-G is a declarative language and framework for the efficient generation of OpenCL GPU applications. Using our proof-of-concept DEF-G implementation, run-time and lines-of-code comparisons are provided for three well-known algorithms (Sobel image filtering, breadth-first search and all-pairs shortest path), each evaluated on three different platforms. The DEF-G declarative language and corresponding OpenCL kernels generated complete OpenCL applications in C/C++. Initial lines-of-code comparison demonstrates that the DEF-G applications require significantly less coding than hand-written CPU-side OpenCL applications. The run-time results demonstrate very similar performance characteristics compared to the hand-written applications. We also provide useful observations, which we found to be noteworthy for practitioners, concerning the effectiveness of certain OpenCL API options.*

**Keywords:** OpenCL, graph algorithms, declarative language

## 1 Introduction

Producing high performance computing (HPC) software for use on graphical processing units (GPUs) is often a difficult and daunting task. This type of software tends to require the use of specialized, parallel algorithms and requires the use of low-level application programming interfaces (APIs), in the context of a thorough understanding of the GPU architecture. The *Declarative Framework for GPUs* (DEF-G) provides a domain-specific computer language (DSL) to assist the software developer. It mitigates the need for a deep understanding of the full CPU-side API used with technologies such as OpenCL, while allowing the user to focus on the algorithms being used and on the most efficient usage of the overall GPU architecture.

Our research in processing large, sparse graphs on GPUs has, out of necessity, led to the direct development of DEF-G. As these large graphs tend to lack locality of reference, the parallel algorithms needed to process them efficiently tend to be complex. Sample problem domains range from graph problems such as the Breadth-First Search (BFS), Single-Source Shortest Path (SSSP), and All-Points Shortest Path (APSP) to iterative matrix inversion, parallel prefix computation, and parallel sorting. Using DEF-G permits us to focus on the algorithms, which were coded mainly in the GPU kernels, and to spend less time focusing on the CPU-side code. In this proof-of-concept implementation of DEF-G, we have implemented and measured, in terms of lines-of-code and run-time performance, three well-known algorithms: Sobel image filtering for edge detection [1] and from the graph theory: BFS and APSP [2].

Common GPU environments in use today, such as OpenCL [3] and NVIDIA's proprietary CUDA [4], tend to provide low-level, very specialized APIs. Their usage requires an understanding of complex, CPU-side APIs [5]. DEF-G provides several higher-level design patterns that abstract the CPU-side coding to a declarative level. Much as the now-ubiquitous relational databases accept database requests as declarative SQL statements and quickly return the requested data, DEF-G uses design patterns and declarative statements to produce high performance CPU-side code, which performs the desired computations. This implementation of DEF-G supports OpenCL; we expect future versions to support both OpenCL and CUDA. Once the developer has produced the kernel code to be executed on the GPU, DEF-G simplifies the task of executing the kernel code. Complex CPU-side operations outside the context of the DEF-G design patterns can be utilized by DEF-G as callable functions.

The current DEF-G implementation consists of a parser written in Java, using ANTLR 3 [6], and our code generator, which is written in C++. The parser handles syntax checking and results in an abstract syntax tree, expressed as an XML document. This abstract syntax tree is then processed by our code generator, which uses the TinyXML2 library [7] to accept the syntax tree. For example, the twelve lines of DEF-G code shown in Figure 1 result in approximately 200 lines of C/C++ code, a snippet of which is shown in Figure 2. The OpenCL kernel executed by this code is shown in Figure 3. Note that this generated OpenCL code is intended to execute on any supported OpenCL device, including the CPU.

OpenCL is an open and cross-platform standard for developing high performance applications on parallel hardware. This standard is supported by major vendors including NVIDIA, AMD, and Intel. There are two major components defined by the standard: the OpenCL C programming language used on the parallel device and the CPU-side APIs for C/C++ that provide access to the device's OpenCL kernels. The CPU manages the execution of the kernels on the OpenCL parallel device.

```
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
```

**Figure 1**:  Sample DEF-G Code

```
// *** buffers in
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 }
```

**Figure 2**:  Snippet of Generated OpenCL Code

```
 __kernel void sobel_filter(__global uchar4* inputImage, __global uchar4* outputImage) {
        uint x = get_global_id(0);  uint y = get_global_id(1);
        uint width = get_global_size(0);  uint height = get_global_size(1);
        float4 Gx = (float4)(0);  float4 Gy = Gx;
        int c = x + y * width;
        /* Read each texel component and calculate ..*/
        if( x >= 1 && x < (width-1) && y >= 1 && y < height - 1)
        {
                float4 i00 = convert_float4(inputImage[c - 1 - width]);
                // similar lines omitted
                float4 i22 = convert_float4(inputImage[c + 1 + width]);
                Gx =   i00 + (float4)(2) * i10 + i20 - i02  - (float4)(2) * i12 - i22;
                Gy =   i00 - i20  + (float4)(2)*i01 - (float4)(2)*i21 + i02  -  i22;
                /* taking root of sums of squares of Gx and Gy */
                outputImage[c] = convert_uchar4(hypot(Gx, Gy)/(float4)(2));
        }
 }
```

**Figure 3:**  Snippet of Sobel OpenCL Kernel Code (from AMD APP SDK 2.8) [18]

The OpenCL C/C++ CPU-side code is required to obtain the kernel source code and call appropriate OpenCL APIs to compile the kernel code. In addition, the OpenCL CPU-side code is required to acquire and manage the low-level buffers accessed by the device kernel. These two requirements tend to make the CPU-side code verbose and often complex; additional complexity is added by the OpenCL requirement to support many different types of parallel platforms and devices, examples being CPUs, GPUs, and even specialized FPGA [8] and DSP [9] hardware. This requirement adds numerous specialized API parameters to the OpenCL API. It can be argued that the OpenCL API is unnecessarily complex, not easily learned, and somewhat hard to use and debug. DEF-G takes over much of the burden of writing the OpenCL CPU-side code, permitting the developer more focus on the device kernels and parallel algorithms proper.

We approached our work as follows: using three existing OpenCL applications and using the existing OpenCL kernels without any changes, we replaced the existing CPU-side code with the DEF-G generated code. The DEF-G source modules needed on average about 90% fewer lines of code. We then compared the computational performance of the three applications over three different OpenCL platforms. Performance variations between the DEF-G results and the reference results were identified and analyzed.

The next sections describe related work, followed by the three existing OpenCL applications that were used as reference/benchmark applications and converted to DEF-G. We then present our experimental results in terms of lines of code and run times, and make some observations for GPU practitioners. A summary of ongoing and future work is presented in the last section.

# 2 Related Work

Numerous attempts have been made to construct languages, compilers, and tools to make the production of high performance parallel solutions easier. In 2005, Shen et al. [10] talked about the "holy grail" of parallelization, which is the automated parallelization of serial programs, being out of reach. However, progress is being made. One approach towards the efficient production of GPU-based parallel solutions is the use of domain-specific languages (DSL). DEF-G is a DSL, a language and associated tools that facilitate the production of OpenCL applications. Martin Fowler defines a DSL as a "computer programming language of limited expressiveness focused on a particular domain," and suggests that DSLs can be broken into two categories: *interna*l DSLs and *external* DSLs [11]. DSLs of both varieties have been produced for GPU-based HPC.

Internal DSLs for GPU-based HPC include extensions to Python such as: PyGPU [12], PyCUDA [13], and PyOpenCL [14]. These DSLs tend to consist of Python wrappers placed around a particular GPU API. There are also C/C++ extensions such as Bacon [15]. Aside from DEF-G, other GPU external DSLs include the SPL digital signal processing language [16] and the MATLAB Parallel Computing Toolbox (which supports CUDA and permits passing some MATLAB functions to the GPU and permits GPU kernel execution [17]). Both MATLAB and DEF-G require that the GPU kernel be provided.

The BFS and APSP implementations we chose for our DEF-G testing were existing implementations, easily obtained from software development kits (SDKs) and benchmarks [18-19]. There exists many other published algorithms and implementations that may provide better overall run-time performances. We anticipate implementing a subset of these in DEF-G. For example, Merrill, et al. suggest a much faster BFS solution which uses *prefix sum* to help distribute the work among GPU threads without locking [20]. We intend to apply the prefix sum lock-avoidance approach to graph-oriented algorithms, which were not addressed in this study. For APSP, Katz and Kider provide a method for using *tiling* with the Floyd-Warshall APSP algorithm to minimize GPU global memory access times [21].

# 3 DEF-G Framework Language

The DEF-G *declarative language* consists of a number of declare, execute and call statements, and some *optional statements* such as sequence/times and loop/while. An example DEF-G source file is shown in Figure 1. The declare statement is used to name the DEF-G application, define and name the GPU kernels to be executed, define any required scalar variables such as a graph's node count, and define the buffers to be given to the GPU. Lines 1 to 8, in the DEF-G sample, show declare statements. The syntax on line 6 enclosed in "[["and"]]" brackets is our method for setting the global grid size. The call statement is used to invoke C/C++ functions, e.g., to obtain the input data; the sample has call statements on lines 9 and 11. The execute statement on line 10 is used to execute the kernel. The flow of control is a design pattern built into DEF-G.

The optional statements are used to provide support for more complex design patterns where the kernels may have to be executed a variable number of times. Figure 4 contains a DEF-G example which executes the kernel once for each graph node. Figure 4, line 9, shows the sequence statement application. DEF-G contains statements to process scalar values returned by kernels. This capability was used in the DEF-G BFS solution to conditionally stop the parallel device processing. DEF-G generates OpenCL 1.1 code.

# 4 Discussion of Results

To test the viability of DEF-G, we selected three existing OpenCL solutions based on well-known algorithms: Sobel image filtering and Floyd-Warshall APSP, both from the AMD APP SDK [16], and breadth-first search from the OpenDwarfs benchmark [17]. We will refer to these solutions as SOBEL, FW, and BFS, respectively. SOBEL was chosen because it represents the class of simpler GPU problems, where a single kernel is called once and because it has significant RAM locality of reference.

```
01. declare application  floydwarshall
02.   declare integer NODE_CNT (0)
03.   declare integer BUF_SIZE (0)
04.   declare gpu gpuone ( any )
05.   declare kernel  floydWarshallPass FloydWarshall_Kernels  ( [[ 2D,NODE_CNT ]] )
06.   declare integer buffer buffer1 ( $BUF_SIZE )
07.             integer buffer buffer2 ( $BUF_SIZE )
08.   call init_input (buffer1(in) buffer2(in) $NODE_CNT(out) $BUF_SIZE(out))
09.   sequence $NODE_CNT times
10.     execute run1 floydWarshallPass ( buffer1(inout) buffer2(out) $NODE_CNT(in) $CNT(in) )
11.   call disp_output (buffer1(in) buffer2(in) $NODE_CNT(in))
12. end
```

**Figure 4**: Sample DEF-G Code Showing a Sequence

In future implementations of DEF-G, we expect to support several concurrent GPU devices in a declarative manner and SOBEL provides a good test case for this added support.

FW and BFS were selected because they represent two different classes of graph-oriented GPU problems, with BFS being the more complex. The FW algorithm requires the same operation to be repeated for each graph node; in this implementation, the FW kernel is called once for each node. This call-for-each-node behavior must be managed from the CPU-side. The OpenDwarfs BFS implementation is based on the work by Harish [22] and uses a version of Dijkstra's algorithm [2]. The actual OpenDwarfs code is a port of the BFS CUDA code from the Rodinia benchmark [23]. This BFS implementation requires that a pair of kernels be repeated until success is indicated by the second kernel. This repetition is managed by the CPU-side code.

All three of these were converted to DEF-G, keeping the unmodified OpenCL kernels. The conversions to DEF-G produce exactly the same results as the corresponding reference version. Before discussing the performance results, we summarize the hardware and software used. The tests were run on three different configurations, which we call CPU, GPU-GT 430, and GPU-Tesla T20, which are listed in order of increasing power, as shown in Table 1.

In terms of module line count results, the three DEF-G versions were much smaller than their reference counterparts. Table 2 shows the line counts for SOBEL, BFS, and FW. Shown are the number of lines of DEF-G declarative code, the number of lines of generated code, and the estimated number of non-comment lines in the reference version. This data is shown graphically in Plot 1. On average, the DEF-G code is 7.7 percent of the generated code, and 4.4 percent of the reference code. It should be noted that the reference code tended to include additional functionality; therefore, the comparison with the generated code is likely to be more indicative of the DEF-G's effectiveness.

The run-time performance comparison turned out to be very interesting. The raw run times, in milliseconds, are presented in Table 3. Plot 2 shows this data presented in 3D form. The results shown are the average of ten runs done for each case. Where we encountered unexpected results, we often reran the tests with manual code changes to isolate the underlying technical causes. We made these code changes to both the DEF-G and reference OpenCL code. However, the numbers shown here are only the original times, i.e., those prior to any manual code modifications.

SOBEL is the simplest application and the run-time performance results between DEF-G and the reference cases are comparable. The SOBEL results are shown on the graph in purple. The DEF-G performance was slightly faster on the CPU and GPU-GT 430 runs, and was slightly slower on the GPU-Tesla T20. This similarity of results is not surprising as the CPU-side support needed for SOBEL is not complex.

The run-time results of the FW tests, which are shown in green, were a surprise to us. We saw no obvious explanation for why DEF-G should be consistently faster. We reviewed the OpenCL code for both DEF-G and the AMD SDK-supplied reference case, and did not find any significant differences in buffer usage or the OpenCL API functions used. We did notice that the reference case was using asynchronous events when not required and we temporarily disabled them and reran the reference case. The FW reference case run

**Table 1:** Test Configurations

| Name | Configuration Data |
|---|---|
| CPU | Windows 7, Intel I3 Processor, 1.33 GHz, 4 GB RAM, using AMD OpenCL SDK 2.8 (no GPU) |
| GPU-GT 430 | Windows 7, Intel Pentium 4 Processor, 3.2 GHz, 1.5 GB RAM, using NVIDIA OpenCL SDK 4.2, NVIDIA GT 430 GPU with 2 Compute Units, 1400 MHz and 1024M RAM |
| GPU-Tesla T20 | Penguin Computing Cluster, Linux Cent OS 5.3, AMD Opteron 2427 Processor, 2.2 GHz, 24 GB RAM, using NVIDIA OpenCL SDK 4.0, NVIDIA Tesla T20 with 14 Compute Units, 1147 MHz and 2687M RAM |

**Table 2:** Lines of Code

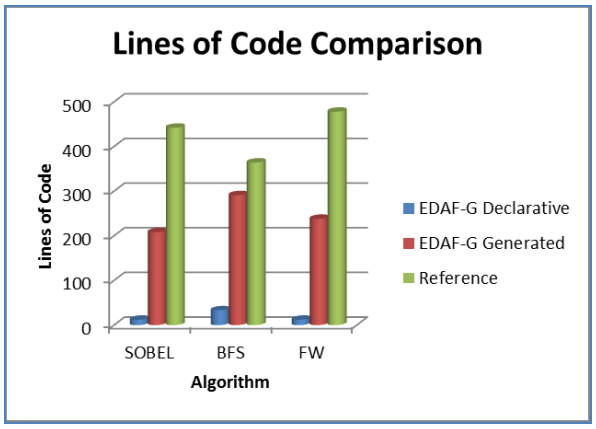
| | DEF-G | DEF-G | |
|---|---|---|---|
| | **Declarative** | **Generated** | **Reference** |
| **BFS** | 33 | 291 | 364 |
| **FW** | 12 | 238 | 478 |
| **SOBEL** | 12 | 208 | 442 |

times dropped three-fold from an average 51.2 ms to 17 ms. This difference was later traced to what we identified as an error in the OpenCL event handling. We feel the DEF-G Tesla time of 11.3 ms and the reference case time of 17 ms are reasonably close and this test tends to show that, for this implementation of the Floyd-Warshall algorithm, both implementations' run times were comparable.

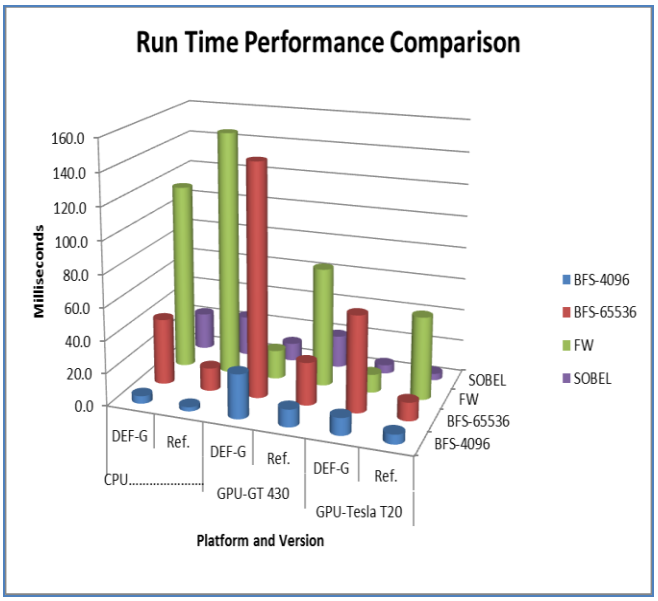
The BFS run-time comparisons used two different graphs. The first graph has 4,096 nodes, shown in blue on the graph, and the second has 65,536 nodes, shown in red. It is clear that the reference case runs significantly faster than DEF-G. For example, on the Tesla, the reference case ran in an average of 11.3 ms and DEF-G in an average of 59.4 ms. As we had done with the FW tests, we analyzed the performance difference. We found that DEF-G was moving buffers to the OpenCL device when not required. After manually adjusting the code to eliminate the movement of these buffers in the generated OpenCL code, the 59.4 ms run time dropped to an average of 28.6 ms. This performance can be improved even more by enhancing the DEF-G language to distinguish between buffers that are moved on each kernel execution and those that are initialized only once, and by the addition of buffer-use optimization to the DEF-G OpenCL code generator. The current code generator contains very little optimization functionality, but we are optimistic DEF-G can come close to the reference-case performance with these, and perhaps additional, enhancements.

We cannot leave the BFS performance topic without noting that the OpenCL CPU configuration's performance was better than either of the GPU performances, except for the Tesla 65,536 node case. We postulate that this is explained by the BFS implementation being used. This graph algorithm implementation is based on the work by Harish [22], which does not compensate for the lack of memory caching on many GPUs. The CPU version most likely fared so well due to the multiple levels of memory caching provided by the Intel I3; it is also likely that the 65,536 node case did not fit entirely in the Intel I3's cache.

In summary, these four comparison tests have shown that, at least in these three cases, the declarative approach used in DEF-G can be used to produce OpenCL applications with fewer lines of code and comparable performance levels.




**Plot 1:** Size Comparison of Module Sizes




**Plot 2:** Performance Comparison of Run Times

**Table 3:** Run-time Performance, in milliseconds

| | CPU | | GPU-GT 430 | | GPU-Tesla T20 | |
|---|---|---|---|---|---|---|
| | **DEF-G** | **Ref.** | **DEF-G** | **Ref.** | **DEF-G** | **Ref.** |
| **BFS-4096** | 4.7 | 2.6 | 27.2 | 10.7 | 10.6 | 5.8 |
| **BFS-65536** | 40.9 | 14.2 | 143.9 | 26.5 | 59.4 | 11.3 |
| **FW** | 115.6 | 152.0 | 17.9 | 73.5 | 11.3 | 51.2 |
| **SOBEL** | 23.0 | 24.8 | 11.1 | 20.0 | 5.3 | 4.1 |

# 5 Observations for Practitioners

Although our performance tests were limited to three platforms and four tests cases, we have two important observations for OpenCL HPC developers.

*Observation One*: The OpenCL "implicit model" worked well. The OpenCL clEnqueueNDRangeKernel() API call is used to execute kernels and its sixth parameter describes the number of work items that make up a work group. This can be hard to calculate and optimize. There is an option to set this parameter to NULL and allow OpenCL to set this internally. This is referred to as the "implicit model," by Munshi, et al. [24]. The proof-of-concept version of DEF-G uses this implicit model; much to our surprise the implicit model performance was equal in many cases to the tuned setting. We suggest that practitioners may want to try using the implicit mode as part of their performance testing to help verify that their explicitly-set values are superior.

*Observation Two*: The clCreateBuffer() CL_MEM_COPY_HOST_PTR option gave inconsistent performance. This option permits the clCreateBuffer() call to provide the address of the host buffer and avoid later calls to clEnqueueReadBuffer()/clEngueueWriteBuffer(). Use of this option appeared to introduce performance issues in a limited number of our tests; we encountered cases where using this option and avoiding the associated clEnqueueReadBuffer()/clEngueueWriteBuffer() calls did add significantly to the run time. We suspect the performance of this option could vary by GPU vendor and device; we suggest trying both approaches with your specific OpenCL device.

# 6 Ongoing and Future Work

This proof-of-concept DEF-G implementation has shown that our declarative approach is able to produce results with less code written and still maintain similar run-time performance, at least for this family of test cases. The addition of buffer optimization to DEF-G would greatly benefit its buffer management performance and, hence, the overall run times. DEF-G also needs the addition of high-performance data loaders and result displays, as well as simple debugging aids such as logging and formatted buffer dumps. We anticipate expanding the DEF-G toolkit to support the use of multiple GPUs, to have optional automatic tuning of various GPU parameters, and to have callable modules generated by DEF-G. Once we have automatic tuning capabilities, we will consider producing a code generator for NVIDIA's CUDA. We also expect to implement the generation of human-readable OpenCL C/C++ code that is a starting point for customized GPU applications and to implement other higher-performance approaches to BFS and APSP.

DEF-G was developed as a result of a specific need; that need being the rapid and efficient production of CPU-side code for use in GPU-based parallel algorithms research. Our DEF-G results look very promising. DEF-G provides a tool to achieve the quick performance analysis of new OpenCL kernels and algorithms. Given this success, we anticipate enhancing DEF-G and making this tool publicly available. The DEF-G toolkit should be a useful asset in future GPU high-performance algorithms research.

# 7 References


[1] Vincent, O. R., and O. Folorunso. "A descriptive algorithm for sobel image edge detection." In Proceedings of Informing Science & IT Education Conference (InSITE), pp. 97-107. 2009.

[2] Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford. Introduction to Algorithms (3rd ed.). MIT Press and McGraw-Hill. 2009.

[3] "The OpenCL Specification 1.1," [Online]. Available: http://www.khronos.org/opencl/

[4] "CUDA 5 Programing Guide," [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide

[5] OpenCL Reference Pages, [Online]. Available: http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/

[6] ANTLR 3, [Online]. Available: http://www.antlr3.org/

[7] Tiny XML2, [Online]. Available: http://www.grinninglizard.com/tinyxml2/index.html

[8] Altera Corporation OpenCL, [Online]. Available: http: www.altera.com/opencl

[9] Texas Instruments OpenCL, [Online]. Available: http://e2e.ti.com/support/dsp/omap_applications_processors/f/447/t/132798.aspx

[10] Shen, John Paul, and Mikko H Lipasti. Modern Processor Design : Fundamentals of Superscalar Processors. Boston: McGraw-Hill Higher Education, 2005.

[11] Fowler, Martin. Domain-specific languages, Addison-Wesley Professional, 2010.

[12] PyGPU, [Online]. Available: http://fileadmin.cs.lth.se/cs/Personal/Calle_Lejdfors/pygpu/

[13] PyCUDA, [Online]. Available: http://mathema.tician.de/software/pycuda

[14] PyOpenCL, [Online]. Available: http://mathema.tician.de/software/pyopencl

[15] Tuck, Nat. "Bacon: A GPU Programming Language With Just in Time Specialization (Draft)." University of Massachusetts Lowel, Lowel MA 01854.

[16] Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David Padua.. SPL: a language and compiler for DSP algorithms. In Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation (PLDI '01). ACM, New York, NY, USA, 298-308, 2001.

[17] Matlab Parallel Computing Toolbox, [Online]. Available: http://www.mathworks.com/products/parallel-computing/

[18] AMD APP SDK 2.8, [Online]. Available: http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/

[19] OpenDwarfs Benchmark, [Online]. Available: https://github.com/opendwarfs/OpenDwarfs

[20] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012.”Scalable GPU graph traversal.” In Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP '12). ACM, New York, NY, USA, 117-128.

[21] Katz, Gary J., and Joseph T. Kider Jr. "All-pairs shortest-paths for large graphs on the GPU." In Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, pp. 47-55. Eurographics Association, 2008.

[22] Harish, Pawan, and P. Narayanan. "Accelerating large graph algorithms on the GPU using CUDA." High Performance Computing–HiPC 2007 (2007): 197-208.

[23] Rodinia GPU Benchmark, [Online]. Available: http://lava.cs.virginia.edu/Rodinia/

[24] Munshi, Aaftab, Benedict Gaster, and Timothy G. Mattson. OpenCL programming guide. Addison-Wesley Professional, 2011.