A second generation of DEFG: Declarative Framework for GPUs

Robert Senser and Tom Altman
Department of Computer Science and Engineering, University of Colorado Denver, Denver, CO
robert.senser@ucdenver.edu[1], tom.altman@ucdenver.edu

Abstract - *DEFG is our declarative language and framework for the efficient generation of OpenCL GPU applications. Using our new DEFG 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. The DEFG declarative language and corresponding OpenCL kernels provide complete OpenCL applications. The lines-of-code comparison demonstrates that the C/C++ DEFG applications require significantly less coding than hand-written CPU-side OpenCL applications. The run-time results demonstrate equivalent, or better, performance characteristics compared to the hand-written applications.*

Keywords: OpenCL, algorithms, declarative language, C++

1 Introduction

This paper is a continuation of our previous work [1], where a description of the DEFG prototype and its associated performance results were introduced. This paper describes our completed DEFG Version 2 and reports on the early promising tests results showing significant improvement in performance and functionality.

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* (DEFG) provides a domain-specific computer language (DSL) designed to assist the software developer. It mitigates the need for a deep understanding of the full CPU-side OpenCL API, therefore allowing the developer 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 DEFG. 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, image processing, and parallel sorting. Using DEFG permits us to focus on the algorithms, which are coded mainly in the GPU kernels, and to spend less time focusing on the CPU-side code. In this full implementation of DEFG, 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 [2] and from the graph theory: BFS and APSP [3].

Common GPU environments in use today, such as OpenCL [4] and NVIDIA's proprietary CUDA [5], tend to provide low-level, very specialized APIs. Their usage requires an understanding of complex, CPU-side APIs [6]. DEFG 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, DEFG uses design patterns and declarative statements to produce high performance CPU-side code, which performs the desired computations. Once the developer has produced the kernel code to be executed on the GPU, DEFG simplifies the task of executing this kernel code. Complex CPU-side operations outside the context of the DEFG design patterns can be utilized within DEFG as callable functions.

This DEFG implementation consists of a parser written in Java, using ANTLR 3 [7], a Java-based optimizer, and a 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 tree is then optimized for run-time performance and decorated with cross-reference information needed for code generation. The tree is then processed by a code generator, which uses the TinyXML2 library [8] to accept the XML-based tree. For example, the twelve lines of DEFG code expressed in Figure 1 result in approximately 460 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

[1] Contact author.

is designed to execute on any OpenCL-supported 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.

The CPU-side code obtains the kernel source code and then calls the appropriate OpenCL APIs to compile this kernel source code. In addition, the OpenCL CPU-side code acquires and manages the low-level buffers accessed by the device kernel. These required actions tend to make the CPU-side code quite verbose and often complex; additional API 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 [9] and DSP [10] hardware. This flexibility unfortunately 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. DEFG takes over much of the burden of writing the OpenCL CPU-side code, thus permitting the developer to focus on the device kernels and the actual parallel algorithms.

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

Section 2 describes related work and includes a description of the three existing OpenCL applications, which we used as reference/benchmark applications and converted to DEFG. The DEFG language is briefly described in Section 3. We then present our experimental results in terms of lines-of-code counts and run times in Section 4. 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. [11] 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). DEFG 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: *internal* DSLs and *external* DSLs [12]. DSLs of both varieties have been produced for GPU-based HPC.

Internal DSLs for GPU-based HPC include extensions to Python such as: PyGPU [13], PyCUDA [14], and PyOpenCL [15]. These DSLs tend to consist of Python wrappers placed around a particular GPU API. There are also C/C++ extensions such as Bacon [16]. Aside from DEFG, other GPU external DSLs include the SPL digital signal processing language [17] and the MATLAB Parallel Computing Toolbox (which supports CUDA and permits passing some MATLAB functions to the GPU and permits GPU kernel execution [18]). Both MATLAB and DEFG require that the GPU kernel be provided.

The BFS and APSP implementations we chose for our DEFG testing are existing implementations, easily obtained from software development kits (SDKs) and benchmarks [19-20]. Obviously, there exist other published algorithms and implementations that may provide better overall run-time performance but that is not the primary goal of this research. We have implemented a subset of these algorithms in DEFG and will present our results in a future paper. 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 [21]. For APSP, Katz and Kider provide a method for using *tiling* with the Floyd-Warshall APSP algorithm to minimize GPU global memory access times [22].

3 DEFG Framework Language

The DEFG *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 DEFG source file is shown in Figure 1. The declare statement is used to name the DEFG 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 DEFG sample, express declare statements. The syntax on line 6, enclosed in "[["and"]]" symbols, 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 DEFG.

The optional DEFG statements can be 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 DEFG example which executes the kernel once for each graph node. Figure 4, line 9, shows the sequence statement application. DEFG contains statements to process scalar values returned by kernels. This capability was used in the DEFG BFS solution to conditionally stop the parallel device processing. DEFG Version 2 generates OpenCL 1.1 code in keeping within the limits of NVIDIA's current OpenCL support [23].

4 Discussion of Results

To test the viability of DEFG, we selected three existing OpenCL solutions based on well-known algorithms: Sobel image filtering and Floyd-Warshall APSP, both from the AMD APP SDK [17], and breadth-first search from the OpenDwarfs benchmark [18]. 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. DEFG can support several concurrent GPU devices, in a declarative manner, and SOBEL provides a good test case for this added capability. This capability will be more fully covered in a future paper.

FW and BFS were selected because they represent two different classes of graph-oriented GPU problems, with the BFS solution requiring multiple GPU kernels. The FW algorithm simply requires that a common operation be repeated for each graph node. In this FW implementation, the OpenCL 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 [24] and uses a version of Dijkstra's algorithm [3]. The actual OpenDwarfs code is an OpenCL port of the BFS CUDA code from the Rodinia benchmark [25]. 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 DEFG, keeping the unmodified OpenCL kernels. The conversions to DEFG 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 two configurations, which we call CPU and GPU-Tesla T20, which are listed in Table 1.

In terms of developer-written module line count results, the three DEFG 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 DEFG 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 DEFG code is 4.2 percent of the generated code, and 5.1 percent of the reference code. It should be noted that the reference code tended to include additional functionality and that the DEFG generated-code counts include an additional 150 lines of template code used to identify and select the requested GPU devices.

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 DEFG 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 obtained are comparable, as expected. The SOBEL results are shown on the graph in purple. The DEFG performance was slightly faster on the CPU and GPU-Tesla T20. DEFG needed 23.0 ms and 3.7 ms, respectively, while the reference case needed 24.8 ms and 4.1 ms.

The run-time results of the FW tests, which are shown in green, surprised us. We saw no obvious explanation for why DEFG should be substantially faster. We reviewed the OpenCL code for both DEFG 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 T20 reference case run 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 reference case's OpenCL event handling.

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. Our earlier prototype version of DEFG was substantially slower than the reference BFS; prototype DEFG needed 59.4 ms to perform what the reference case BFS did in 11.3 ms. The DEFG Version 2 buffer-use optimization reduced the average BFS T20 run time from 59.4 ms to 8 ms! This drastic improvement in performance is due to the optimizer's removal of unneeded buffer transfer operations between the CPU and GPU.

We cannot leave the BFS performance topic without noting that the OpenCL CPU configuration's performance was better than the GPU performance for the Tesla 4,096 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 [24], which does not compensate for the lack of RAM cache found in many GPU designs. The CPU version most likely fared so well due to the multiple levels of memory caching provided by the Intel I3; it is likely that the 4,096 node test case fit entirely into the Intel I3's cache.

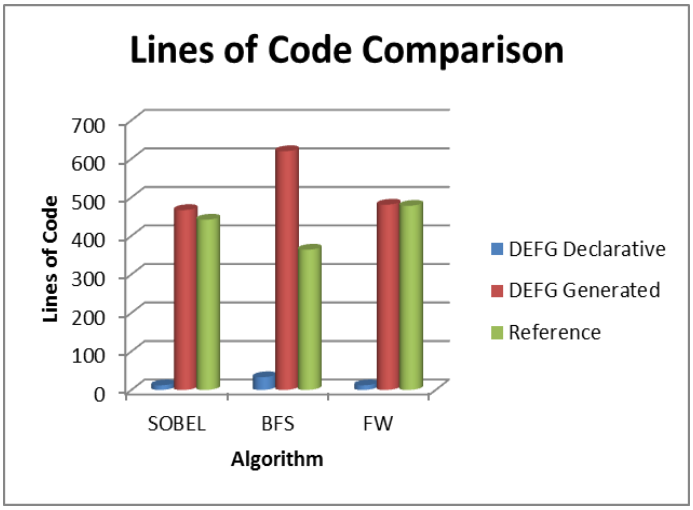
In summary, these four comparison tests have shown that, at least in these cases, the declarative approach used in DEFG can be used to produce OpenCL applications with fewer lines of code and comparable, or better, performance levels.

5 Ongoing and Future Work

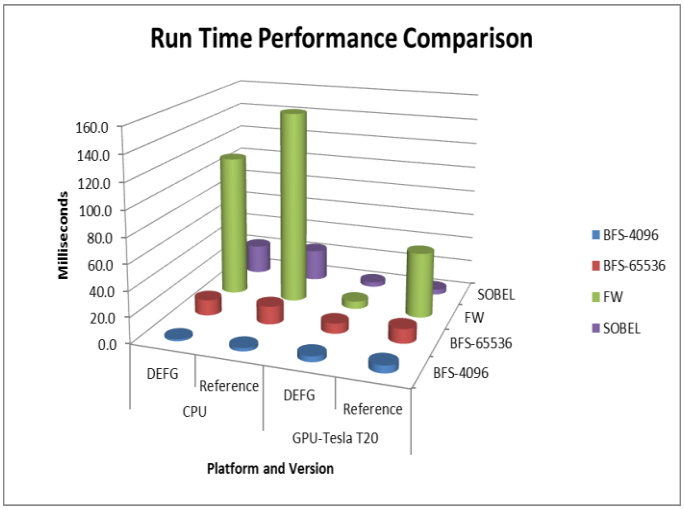
This full DEFG implementation has shown that our declarative approach is able to produce good results with less code written while maintaining similar run-time performance, at least for this family of test cases. The addition of buffer optimization has greatly benefited its buffer management performance and, hence, the overall run times. DEFG also will significantly benefit from the addition of high-performance data loaders and result displays, as well as

simple debugging aids such as logging and formatted buffer dumps. We have already enhanced the DEFG toolkit to support the use of multiple GPUs and to generate callable C/C++ modules. We have implemented the generation of human-readable OpenCL C/C++ code, which is a starting point for the creation of customized GPU applications.

DEFG 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 DEFG results continue to be very promising. DEFG provides a tool to achieve the quick utilization of new OpenCL kernels and algorithms. Given this success, we anticipate enhancing DEFG further and eventually making it publicly available. The DEFG toolkit should be a useful asset in future GPU high-performance algorithms research.



Plot 1: Size Comparison of Module Sizes



Plot 2: Performance Comparison of Run Times

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-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

	DEFG Declarative	DEFG Generated	Reference
BFS	42	620	364
FW	12	481	478
SOBEL	12	467	442

Table 3: Run-time Performance, in milliseconds

	CPU		GPU-Tesla T20	
	DEFG	**Reference**	**DEFG**	**Reference**
BFS-4096	1.5	2.6	4.3	5.8
BFS-65536	12.3	14.2	8.0	11.3
FW	111.8	152.0	6.0	51.2
SOBEL	23.0	24.8	3.7	4.1

6 Sample Code Figures

```
01. declare application  sobel
02.  declare integer Xdim (0)
03.            integer Ydim (0)
04.            integer BUF_SIZE (0)
05.  declare gpu gpuone ( * )
06.  declare kernel  sobel_filter SobelFilter_Kernels  ([[2D,Xdim,Ydim ]] )
07.  declare integer buffer image1 ( Xdim Ydim ) halo (1)
08.            integer buffer image2 ( Xdim Ydim ) halo (1)
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 DEFG 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]

```
01. declare application  floydwarshall
02.   declare integer NODE_CNT (0)
03.             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(inout) NODE_CNT(in) DEFG_CNT(in) )
11.   call disp_output (buffer1(in) buffer2(in) NODE_CNT(in))
12. end
```

Figure 4: Sample DEFG Code Showing a Sequence

7 References

[1] Senser, R. and Altman, T. "DEF-G: Declarative Framework for GPU Environment." *Proceedings of The 2013 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'13), vol II*, pp. 490-496, World-comp.org, 2013.

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

[3] Cormen, T. et al. Introduction to Algorithms (3rd ed.). MIT Press and McGraw-Hill, 2009.

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

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

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

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

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

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

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

[11] Shen, J. and Lipasti, M. Modern Processor Design: Fundamentals of Superscalar Processors. Boston: McGraw-Hill Higher Education, 2005.

[12] Fowler, M. Domain-specific Languages, Addison-Wesley Professional, 2010.

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

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

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

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

[17] Xiong, J. et al. "SPL: a language and compiler for DSP algorithms." *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation (PLDI '01)*, pp. 298-308, ACM, New York, NY, USA, 2001.

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

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

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

[21] Merrill D. et al. "Scalable GPU graph traversal." *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP '12)*, pp. 117-128, ACM, New York, NY, USA, 2012.

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

[23] NVIDIA Cuda Zone – OpenCL, [Online] Available: https://developer.nvidia.com/opencl

[24] Harish, P. and Narayanan, P. "Accelerating large graph algorithms on the GPU using CUDA." *High Performance Computing–HiPC 2007*, pp. 197-208, 2007.

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