Raising the level of abstraction of GPU-programming

F. Jacob[1], **R. Arora**[1], **P. Bangalore**[1], **M. Mernik**[21] **and J. Gray**[3]
[1]Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL, USA
[2]Faculty of Electrical Engineering and Computer Science, University of Maribor, Maribor, Slovenia
[3]Department of Computer Science, University of Alabama, Tuscaloosa, AL, USA

Abstract— *General-purpose computing on GPUs (graphics processing units) has received much attention lately due to the benefits of stream processing to exploit limitations of parallel processing. However, programming GPUs has several challenges with respect to the amount of effort spent in combining the kernel functional code of an application with the parallel concerns offered by APIs from various GPUs. This paper introduces our approach for raising the level of abstaction for programming GPUs. We have implemented an abstract API that can be used with the Compute Unified Device Architecture (CUDA) and the Open Compute Language (OpenCL) frameworks, so that the mechanical steps involved in writing the GPU code are abstracted in separate modules. The approach involves static code analysis and generative programming techniques for automatically generating the host code required for CUDA and OpenCL frameworks from minimal specifications provided by the programmers. The generated code resembles the hand-written code with comparable performance.*

Keywords: parallel programming, GPU programming, abstraction, CUDA, opencl

1. Introduction

Due to their highly parallel architecture, Graphics Processing Units (GPUs) are excellent computational platforms for certain classes of scientific applications. NVIDIA's Compute Unified Device Architecture (CUDA)[1], Microsoft's DirectCompute[2] and Khornos Group's OpenCL[3], are widely used frameworks for writing applications that can run on GPUs. These frameworks provide hardware-based abstraction and decrease the burden on programmers by obviating the need to learn graphics programming. The CUDA framework is used for writing programs targeting NVIDIA's graphics cards. OpenCL is an emerging framework to execute programs on heterogeneous platforms (e.g., CPUs, GPUs and other processors).

When a programmer intends to use CUDA APIs for running the code on the NVIDIA architecture, it is required that the memory space is allocated on both the host machine and the GPU. Data is copied from the main memory of the host machine to the GPU memory, then the kernel (which is a function call from the host) is invoked and the computation is done in parallel on GPUs. The results are then moved back from the GPU memory to the main memory of the host. Despite the minor differences in writing the code using CUDA or OpenCL frameworks, there is much commonality between the two that can be isolated in separate modules in order to create another level of abstraction.

In order to obtain the best performance (or to compare the performance of graphics cards), a programmer may run the application on diverse architectures and experiment with different programming models. As an example, because CUDA APIs are optimized for the NVIDIA architecture, it is quite possible that one might obtain better performance with them instead of using OpenCL APIs if the application is executed on an NVIDIA graphics chip. In such a scenario, this research intends to reduce the burden on the programmer by generating both CUDA and OpenCL code from a single set of specifications using these abstractions.

Another advantage of using abstractions is reduction in the number of code clones (i.e., repeated code blocks spread across multiple locations), which are considered harmful because of the maintenance challenges in updating each of the cloned locations whenever a change to any one of the code blocks is made [1][2]. During this research, several code samples from different sources were analyzed and it was observed that the host code forms the significant portion of the entire application. Often, the majority of the host code involves setting up the execution environment for the parallel program. An OpenCL code snippet (Figure 1) is used to create and build the program. This block of code is used very frequently across various programs. Similarly, the code for `commandqueue` and `kernel` also appears at multiple places. The code snippet shown in Figure 1 was found with minor changes (exception handling, variable name, timer information) in 9 files out of 18 host files. An analysis of the host code from OpenCL samples (NVIDIA OpenCL installation package[4]) was done using the clone detector (Simian[5]) and the results are presented in Figure 2. The Y-axis represents the number of duplicates and the X-axis represents the total number of lines in a particular duplicate. Overall, Simian found 1102 (roughly 20%) duplicated lines

[1]http://www.nvidia.com/object/CUDA_home_new.html
[2]http://code.msdn.microsoft.com/directcomputehol
[3]http://www.khronos.org/OpenCL/
[4]http://developer.nvidia.com/object/opencl.html
[5]http://www.redhillconsulting.com.au/products/simian/

```
// create the program
cl_program cpProgram = clCreateProgramWithSource(cxGPUContext,1, (const char **)&source, &program_length,
                                                 &ciErrNum);
if(ciErrNum != CL_SUCCESS){
    shrLog(LOGBOTH, 0.0, "Error\n");
    return ciErrNum;
    }
free(header);
free(source);

// build the program
ciErrNum = clBuildProgram(cpProgram, 0, NULL, NULL, NULL, NULL);
if(ciErrNum != CL_SUCCESS){
    // write out standard error, Build Log and PTX, then return error
    shrLog(LOGBOTH | ERRORMSG, (double)ciErrNum, STDERROR);
    oclLogBuildInfo(cpProgram, oclGetFirstDev(cxGPUContext));
    oclLogPtx(cpProgram, oclGetFirstDev(cxGPUContext), "oclMatrixMul.ptx");
    return ciErrNum;
}
```

Fig. 1: OpenCL code fragment to create and build kernel program.

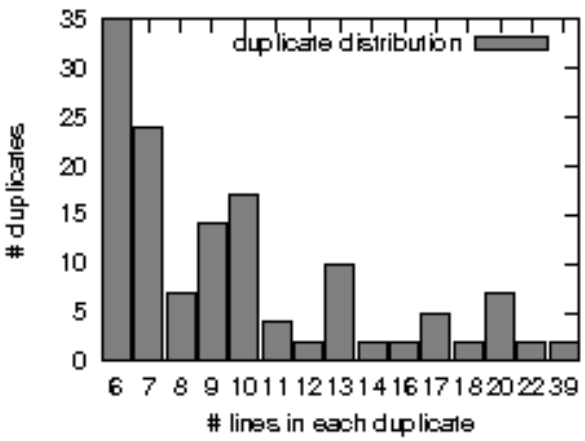

Fig. 2: Clones in OpenCL code samples.

from a total of 5200 lines of code (excluding comments) in 18 files. The main advantages of this new layer of abstraction are: 1) reduced burden on the programmer in terms of reduction in the number of lines of code that the programmer has to write, 2) reduction in code clones, and 3) increase in flexibility in developing and testing the applications for the GPU-environment (one can generate both CUDA and OpenCL code from the same specifications).

Several other research efforts in the area of high-level parallel programming models and environments have shown that abstractions can reduce the burden on programmers without compromise performance [3][4][5][6][7][8]. In the work presented in this paper, domain engineering (the domain being programming models for GPUs) [9] is used to develop the abstractions in the form of APIs. This approach is analogous to the directive-based approach provided by OpenMP[6] for writing parallel programs for shared memory architectures. The CDT (C/C++ Development Tooling[7]) parser is then used to search and replace the APIs with the actual code for CUDA/OpenCL. This type of automatic code generation for product families (CUDA/OpenCL) to reduce the burden on the programmers is called generative programming [9].

The rest of the paper presents more insight into the improved abstraction offered by our approach. An example scenario is introduced in Section 2. Code generation for the example scenario is shown in Section 3. Section 4 describes an overview of the implementation details. Section 5 discusses some experimental evaluation of the current research. A list of related works and their comparison is included in Section 6 of the paper. Section 7 concludes with a description of future work.

2. Example Scenario

Consider an example where sequential code is created to add two arrays, as shown in Figure 3. The code includes declaring three arrays, initializing two of them, passing the arrays to a method to execute the addition operation, and finally cleaning up the resources that were used. The aim of the programmer is to execute this code in parallel for a given configuration. The APIs for creating the parallel version code (right side of Figure 3) is explained is Section 3. The first step in the process is to generate the necessary host code for CUDA and OpenCL after taking minimum input from the user and then the kernel code. The calculations are done in kernel code, and the set up to call the kernel is referred to as the host code. A pictorial representation or flow of the execution of host and kernel code is shown in Figure 4. An important observation is made that the GPU and CPU (host) do not share the memory.

Table 1 shows the comparison between CUDA and OpenCL APIs. There can be other APIs available in both

[6]http://www.openmp.org/specs/mp-documents/paper/

[7]http://www.eclipse.org/cdt

```
/*                                                    /*
 *  SEQUENTIAL CODE                                    *   PARALLEL CODE USING ABSTRACT APIS
 */                                                    */
                                                   //Set GPU execution parameters
                                                      _GPUinit(16,16,4,4);

float* h_A = (float*) malloc (mem_size_A);            _XPUmalloc(h_A,"float*",mem_size_A);
float* h_B = (float*) malloc (mem_size_B);            _XPUmalloc(h_B,"float*",mem_size_B);
float* h_C = (float*) malloc (mem_size_C);            _XPUmalloc(h_C,"float*",mem_size_C);

initarray(h_A,h_B);                                    initarray(h_A,h_B);
sequentialAdd(h_A,h_B);                                _GPUcall("arrayAdd", in(h_A,h_B),out(h_C));
printArray(h_C);                                      printArray(h_C);

free(h_A);                                         //Release Memory
free(h_B);                                            _XPUrelease();
free(h_C);
```

Fig. 3: Code setup for adding two arrays.

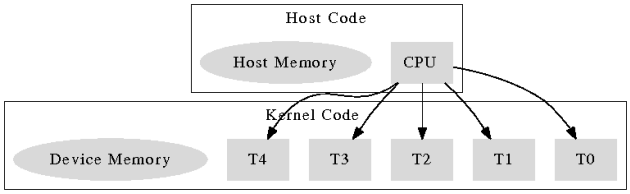

Fig. 4: Host and device execution.

Table 1: API comparison of OpenCL and CUDA.

Function	CUDA	OpenCL
Allocate Memory	cudaMalloc	clCreateBuffer
Transfer Memory	cudaMemcpy	clWriteBuffer clReadBuffer
Call Kernel	$<<< x, y >>>$	clEnqueueNDRange clSetKernelArg
Block Identifier	blockIdx	get_group_id
Thread Identifier	threadIdx	get_local_id
Release Memory	cudaFree	clReleaseMemObject

CUDA and OpenCL to achieve similar functions. The procedure inside a general host code is similar for both OpenCL and CUDA, and is explained below. The discussion illustrates the high level of similarity within the host code.

1) **Allocate Memory**
   In this step the device memory is allocated. CUDA performs a device memory allocation with a function call to `cudaMalloc`, and OpenCL with a function call to `clCreateBuffer`. There is a slight difference between these two function calls: OpenCL provides an additional bit field to specify allocation and usage information (e.g., `CL_MEM_READ_ONLY`).
2) **Copy from Host to Device Memory**
   In this step, the host variables that are required for the device calculations are copied to device variables, which are defined with qualifiers that limit the scope of the variables. CUDA uses `cudaMemCpy` for this step and OpenCL uses `clWriteBuffer`
3) **Execute the Kernel code**
   To execute code on the GPU, some of the parameters (e.g., block size in CUDA or work group in OpenCL) must be initialized. There are some differences in the way these parameters are set in CUDA and OpenCL, but both still have a very striking similarity at a higher level which makes it possible to generate the code for both.
4) **Copy from Device to Host Memory**
   After executing the kernel code, the results are copied from device variables back to the corresponding host variables. CUDA uses the `cudaMemCpy` and OpenCL uses `clReadBuffer`.
5) **Release Memory**
   To release the memory occupied by the device variables, `cudaFree` and `clReleaseMemObject` are the common APIs in CUDA and OpenCL, respectively.

It is obvious that the device variables should be of the same type and also will be allocated the same size as the corresponding host variables. Another inference that can be made from the above similarities, is that input variables should be copied to the device from the host and output variables should be copied back to the host from the device after computation. The ultimate goal of the project is to hide the device details completely from the programmer, so that the programmer only specifies the block of code or method that needs to execute faster. All of the other details should be handled automatically by the programming environment. The relevant question in this context is, to what extent we can achieve this? To automate the steps above, how much information do we need from the programmer? Do programmers manually define the device variables and ensure that they are of the same size and type?

```
          /*
           * CUDA code before kernel call
           */

//Variable intialization
float* h_A = (float*) malloc (mem_size_A);
float* _GEN_PREFIX15;
cutilSafeCall(cudaMalloc((void**)&_GEN_PREFIX15,
              mem_size_A));

float* h_B = (float*) malloc (mem_size_B);
float* _GEN_PREFIX16;
cutilSafeCall(cudaMalloc((void**)&_GEN_PREFIX16,
              mem_size_B));

float* h_C = (float*) malloc (mem_size_C);
float* _GEN_PREFIX17;
cutilSafeCall(cudaMalloc((void**)&_GEN_PREFIX17,
              mem_size_C));

//Sequential function call
initarray(h_A,h_B);

//GPU parameters intialiazation
dim3 threads(16,16);
dim3 grid(4,4);

//Copy host to device
cutilSafeCall(cudaMemcpy(_GEN_PREFIX15,h_A,
              mem_size_A,cudaMemcpyHostToDevice));

cutilSafeCall(cudaMemcpy(_GEN_PREFIX16,h_B,
              mem_size_B,cudaMemcpyHostToDevice));
```

Fig. 5: Snapshot of code generated for CUDA.

```
          /*
           * OpenCL with kernel call
           */

//Execution of kernel
err = clSetKernelArg(kernel[0],0,sizeof(cl_mem),
                     &h_A);
err |= clSetKernelArg(kernel[0],1,sizeof(cl_mem),
                     &h_B);
err |= clSetKernelArg(kernel[0],2,sizeof(cl_mem),
                     &h_C);

assert(err == CL_SUCCESS);
clFinish(cmd_queue);

//Read the results back to host
err = clEnqueueReadBuffer(cmd_queue,h_C,CL_TRUE,0,
                          mem_size_C,
                          (void*)_GEN_PREFIX20,
                          0,NULL,NULL);

assert(err == CL_SUCCESS);
clFinish(cmd_queue);

//Sequential function call
printArray(h_C);

//Release device memory
clReleaseMemObject(_GEN_PREFIX18);
clReleaseMemObject(_GEN_PREFIX19);
clReleaseMemObject(_GEN_PREFIX20);

// Release host memory
free(h_A);free(h_B);free(h_C);
```

Fig. 6: Snapshot of code generated for OpenCL.

3. Generalizing CUDA and OpenCL

This section discusses generalizing the CUDA and OpenCL APIs. Abstract APIs can generate code for both CUDA and OpenCL as it can be considered as generalized APIs for GPU programming. Sample code generated using CUDA and OpenCL APIs are shown in Figure 5 and Figure 6: Figure 5 shows the CUDA code before calling the kernel and Figure 6 shows the OpenCL code after calling the kernel. The code fragments are taken from the files generated (CUDA and OpenCL) for the abstract code shown in Figure 3. The APIs used for the sample program are explained below. As a naming convention, the prefix 'GPU' (as used in `GPUcall`) refers to GPU-code generating operations, and XPU (as used in `XPUmalloc`) refers to both ordinary 'C' and GPU-code operations.

1) **XPUmalloc**
   `XPUmalloc` links a host variable and a device variable, which is a generated variable (in Figure 5 for h_A it is _GEN_PREFIX15). This API generates both 'C' and GPU code. First, it allocates memory in the host for the given variable and generates a new variable of the same type and size, then it allocates memory in the device for the generated variable.
2) **GPUinit**
   `GPUinit` provides the initialization parameters for the GPU execution. The first two arguments refer to the global size and the last two parameters refer to the local size. The global size in CUDA is called the block size, and in OpenCL it is called the work group. Similarly, the local size in CUDA is called the thread size and in OpenCL it is called the work item.
3) **GPUcall**
   `GPUcall` generates the code for the execution of the GPU with its parameters. From Figure 3, it can be seen that this method takes two sets of parameters; first, set 'in(..)' which allows the user to specify the input parameters, and the second set 'out()' to specify the output parameters. Before passing the control to the GPU, the input parameters are copied to the GPU and after execution the output parameters are copied back to the host variables. Because the host variables are already linked with the device variables in `XPUmalloc`, the programmer is relieved from the task of managing device variables in the host code.
4) **XPUrelease**
   `XPUrelease` manages the release of memory in both the device and host variables. This does not mean that it takes care of all the variables used in the program. It just monitors the variables that have been initialized using the `XPUmalloc` function call.

A more detailed description of the APIs is shown in Table 2. Three out of the five methods are written specifically for GPU programming and the other two generate 'C' code. In addition to the code shown in Figure 5, a few additional

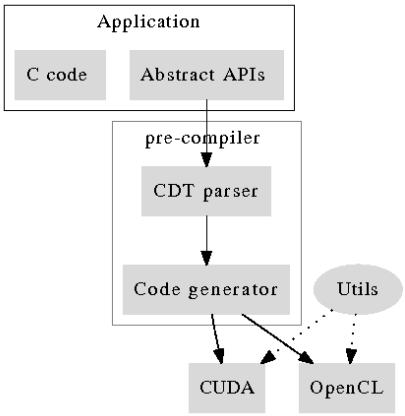

Fig. 7: Components in AbstractAPI implementation.

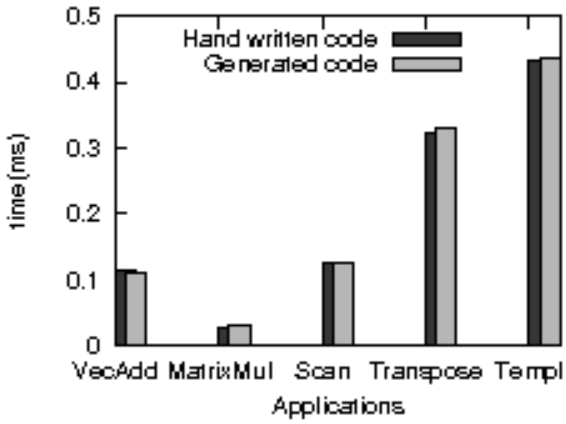

Fig. 8: Comparison of execution time.

lines of code are needed. For OpenCL, the additional code includes finding the devices, creating and building the program and assigning to kernels.

4. Implementation

A pictorial representation of implementation is shown in Figure 7. The abstract APIs are parsed using a "C" parser and passed to the code generator, which generates the code required for the frameworks. Some of the standard functions are available in the "Utils" library, and those functions are included automatically based on usage. For example, while using OpenCL, finding devices and creating kernels are included as standard functions and those function calls will be included in the generated code. This reduces code redundancy, which has the potential to ease code maintenance. The programmer can choose between CUDA, OpenCL or both while translating the Abstract APIs.

The implementation involves an Abstract Syntax Tree (AST) parser and a 'C' code generator. The CDT has been used to address these tasks. The CDT project exposes APIs for traversing through the AST of C/C++ and modifying them. The in-line source transformation is achieved by using necessary visitors to the AST nodes of interest. Because the APIs use in-line transformation, a programmer can include CUDA-specific code if interested only in CUDA. The abstract APIs will generate code at the point where they are referenced in the program and integrate with other CUDA-specific code.

In the data structures used by our API, there are two lookup tables, one for linking the variables of the host and the device, and the other for linking kernels with their names. These data structures assist in the support of multiple devices, which often occur when using OpenCL. A difference between CUDA and OpenCL is the command queue; all the executions in OpenCL are performed with a command queue, but CUDA does not have such a requirement. As a general API, the 'finish' command is issued after inserting a new command into the queue.

5. Experimental Results

To understand the impact and benefits of our abstract API approach, we performed a series of experiments that were conducted on five files, written by expert CUDA programmers. The files were collected from the sample CUDA source code from an NVIDIA installation package. All of the files used are host files that initialize a program that performs the real calculation. The results of the experiments are summarized in Table 3. While calculating the Lines of Code (LOC), lines of code that were not pertinent to the current context (e.g., exception handling and timer information) were ignored. The column "# variables reduced" shows the number of variables that the programmer could avoid by using the abstract APIs. Column "# lines reduced" shows the number of lines reduced in the new code. The last column "API usage", indicates how many times the APIs were used in the modified code. As an overview, the LOC for the program that used the abstract API has comparable values with the LOC of the sequential code.

The execution time plot for the five programs discussed above is shown in Figure 8. The figure shows a comparison between the execution times of the original programs to the generated code using the Abstract APIs. From the figure it is evident that there is no performance loss in using the Abstract APIs. Since the current implementation only generates the host files, the kernel code from the original programs were used for the experiment.

Currently, the framework supports the code generation for the host files only. We are working on automating the code genaration for the kernel part, too. The implementation approach for generating the kernel part will be the same as for the host files. Implementing a real-time application using the abstract APIs developed in this research would suggest

Table 2: API descriptions.

API usage	Description
`_CPUmalloc(A1, A2..., 'float*', size)`	allocate in host A_1, A_2... each of $sizeof(float) * size$
`_GPUmalloc(A1, A2..., 'float*', size)`	allocate in device A_1, A_2... each of $sizeof(float) * size$
`_XPUmalloc(A1, A2...,'float*', size)`	allocate in device and host A_1, A_2... of $sizeof(float) * size$, device variable name is generated
`_GPUinit(BLOCK_SIZE, BLOCK_SIZE, WC/xdim, HC/ydim`	setting up the running configurations, first two denote working groups and rest for the working items
`_GPUcall('function_name',in(A1,A2...), out(B1, B2..)`	call a given function from the kernel with the given input and output should be as provided
`_XPU_release()`	release all the device and host memory

Table 3: Source code analysis of CUDA, CPP and Abstract API.

Application	CUDA LOC	CPP LOC	Abstract LOC	# variables reduced	# lines reduced	API usage
Vector Addition	29	15	13	3	16	6
Matrix Multiplication	28	14	12	3	14	6
Scan Test Cuda	82	NA	72	1	10	12
Transpose	39	17	26	2	13	8
Template	25	13	13	2	12	6

more enhancements to the current implementation. Providing abstraction for all the APIs of CUDA and OpenCL would be another challenge. We are developing the framework incrementally and are providing the support for abstraction as required. It should be noted that the framework is extensible and flexible. Therefore, additional features can be added as the need be without making any changes to the existing code.

6. Related works

Han et al. have developed a high-level abstraction of CUDA that is known as *hi*CUDA [5] for generating applications for NVIDIA GPUs. *hi*CUDA involves a source-to-source transformation process to generate the CUDA program from sequential programs written in C. It provides a front-end for supporting *hi*CUDA directive syntax. These directives are handled by the *hi*CUDA compiler that passes its output to a CUDA code generator. *hi*CUDA demonstrates that high-level abstraction can be attained without compromising heavily on the performance. As the name implies, *hi*CUDA is tightly coupled with CUDA, such that if a GPU other than NVIDIA is used, then the CUDA program has to be rewritten.

Sh[8] is a high-level metaprogramming language for GPUs. It is built on top of C++ and the GPU programs are written with C++ syntax. The Sh code is compiled by the C++ compiler along with the Sh library. The compiled program is converted into an intermediate representation which can then be transformed into the code for any GPU. The intermediate code is run through the backend to obtain the GPU-specific code. The intent behind the research presented in this paper is similar to that of Sh, but differs in the scope. Unlike Sh, which is specific to the graphics domain, the research presented in this paper can be applied to GPU-enable the programs from several different domains.

CUDA-lite [8] is an enhancement to CUDA that reduces the effort involved in writing optimized CUDA code. The programmer annotates the base code which is then transformed to generate the CUDA code. The programmer needs to provide the implementation of the kernel code utilizing the global memory. The tool then transforms the code (like kernel functions) using the annotations provided by the user into an optimized code for memory access. CUDA-lite thus reduces the coding requirement and provides a layer of abstraction over CUDA. The scope of CUDA-lite is limited to optimizing the memory access patterns and it does not handle the parallelization aspects of GPU programming. The focus of the research presented in this paper is to reduce the effort involved in parallelizing an application to run on GPUs.

CGiS [7] is a high-level, data parallel programming language for GPUs. It enables the programmer to specify the program-specifications (algorithm) at a very high-level. The CGiS compiler then converts the specifications into the actual code by choosing a mechanism that is appropriate for the target device (the GPU). This method therefore provides better performance without tying down the specifications to any particular GPU. The current scope of CGiS and the research presented in this paper are similar. However, the approach used in CGiS is different from our approach. We are generating CUDA and OpenCL code from high-level specifications whereas CGiS generates the code specifically

[8]http://libsh.org

for NV30, NV40 and G80 GPUs, CUDA, AltiVec and SSE CPUs. In our approach the heterogeneity in devices is handled directly by the OpenCL framework and hence it is easy to extend the framework to support more devices than what CGiS supports. We are also working on addressing the fault-tolerance issues in the heterogeneous environments [3] and will be supporting OpenMP in the next step of the research.

CuPP [6] is a framework for integrating CUDA into existing C++ applications. This framework provides interfaces for tasks that are reoccurring in a GPU application development context (e.g., memory management) and is built upon CUDA libraries. As compared to CuPP, our approach is targeted for applications written in C. We are providing support for generating code for the OpenCL framework as well.

7. Conclusion and Future Work

This paper presented our approach for improving GPU programming. The key ideas of the approach are based on static code analysis for generating the host code for CUDA and OpenCL frameworks from minimal specifications. The approach is explained with a sample program that includes some of the basic steps involved in GPU programming. We believe that our abstract API will help a GPU programmer by automatically generating several code fragments, which has the potential to make the host code development faster and easier. This would allow the programmer to focus more of their attention on the development of the kernel code. The generated code looks very similar to the hand-written code and the performance of the two is comparable.

The most obvious future work will involve providing similar APIs for kernel code. After generating the kernel, performance evaluation could be done. Some of the other software techniques could be used to calculate the input and output parameters, instead of requiring the user to provide them explicitly. More effort should be directed in converting a 'C' program to a running parallel program with minimum changes from the user. Supporting other frameworks like OpenMP is yet another direction of the same work.

References

[1] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proceedings of the Second Working Conference on Reverse Engineering*, Toronto, Canada, July 1995, pp. 86–95.

[2] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Evaluating the harmfulness of cloning: A change based experiment," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, Minneapolis, MN, May 2007, p. 18.

[3] R. Arora, M. Mernik, P. Bangalore, S. Roychoudhury, and S. Mukkai, "A domain-specific language for application-level checkpointing," in *Proceedings of the 5th International Conference on Distributed Computing and Internet Technology*, New Delhi, India, December 2009, pp. 26–38.

[4] R. Arora and P. Bangalore, "A framework for raising the level of abstraction of explicit parallelization," in *Proceedings of the 31st International Conference on Software Engineering*, Vancouver, Canada, May 2009, pp. 339–342.

[5] T. D. Han and T. S. Abdelrahman, "hiCUDA: A high-level directive-based language for GPU programming," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, Washington, D.C., March 2009, pp. 52–61.

[6] J. Breitbart, "CuPP - a framework for easy CUDA integration." in *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009, pp. 1–8.

[7] N. Fritz, P. Lucas, and P. Slusallek, "CGiS, a new language for data-parallel GPU programming," in *Proceedings of the 9th International Workshop Vision, Modeling, and Visualization*, Stanford, CA, November 2004, pp. 241–248.

[8] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W.-M. W. Hwu, "CUDA-lite: Reducing GPU programming complexity," in *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, Edmonton, Canada, July 2008, pp. 1–15.

[9] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Boston, MA: Addison-Wesley, 2000.