CUDACL: A Tool for CUDA and OpenCL Programmers

Ferosh Jacob*, David Whittaker†, Sagar Thapaliya†, Purushotham Bangalore†, Marjan Mernik‡ and Jeff Gray*

*Department of Computer Science
University of Alabama
Tuscaloosa, AL 30487
Email: {fjacob, gray}@ua.edu

†Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, AL 35205
Email: {dpwhitt, sagar, puri}@cis.uab.edu

‡Faculty of Electrical Engineering and Computer Science
University of Maribor
Maribor, Slovenia
Email: marjan.mernik@uni-mb.si

Abstract

Graphical Processing Unit (GPU) programming languages are used extensively for general-purpose computations. However, GPU programming languages are at a level of abstraction suitable only for use by expert parallel programmers. This paper presents a new approach through which 'C' or Java programmers can access these languages without having to focus on the technical or language-specific details. A prototype of the approach, named CUDACL, is introduced through which a programmer can specify one or more parallel blocks in a file and execute in a GPU. CUDACL also helps the programmer to make CUDA or OpenCL kernel calls inside an existing program. Two scenarios have been successfully implemented to assess the usability and potential of the tool. The tool was created based on a detailed analysis of the CUDA and OpenCL programs. Our evaluation of CUDACL compared to other similar approaches shows the efficiency and effectiveness of CUDACL.

Index Terms

CUDACL, OpenCL, CUDA, Eclipse, Java, 'C'

1. Introduction

Parallel programming can be defined as the creation of code for computations that can be executed simultaneously. Some of the benefits of parallel programming include performance, throughput, and redundancy avoidance. The dependency of the executing code usually defines the complexity of the parallel program. There have been many efforts [1], [2], [3], [4], [5], [6] that have focused on converting existing sequential code to execute in parallel. The languages, platforms and frameworks to perform such conversions have continually evolved, but the central goal has remained the same, i.e., executing code in parallel. The vast amount of technical details involved in the parallel programming domain gave rise to abstractions over parallel programming languages [7], [8], directives [9], and annotations [10] in sequential code, which can aid in generating parallel code. But, even with all of these advances, parallel programming still requires skill beyond that of an average programmer. Currently, in order to write a program that will execute a block of code in parallel, a programmer must learn a parallel programming Application Programming Interface (API) that can be used to describe the computation. Even after the execution, the programmer must use other APIs or frameworks to evaluate the performance of their parallel program. The duration of the implementation and learning phases of a parallel programmer depends on the specific framework or language that the programmer is proficient in using and the programmer's knowledge of parallel programming, in general.

With the rise in performance, the GPUs originally used for graphics cards have found an application for hosting general-purpose parallel computations, traditionally executed in CPUs. The initial GPU programming languages

suffered from portability issues and a steep learning curve [11]. NVIDIA's Computation Unified Device Architecture (CUDA) [1], Microsoft's Direct Compute[2], and Khronos Group's OpenCL[3] are the most commonly used frameworks for General-Purpose GPU (GPGPU) programming.

A survey of general-purpose computation on graphics hardware reveals that GPGPU algorithms continue to be developed for a wide range of problems [12]. The problems are not limited to the scientific or graphics community. To use these GPGPUs outside of their intended context, much work is required in the parallel programming domain to make GPGPUs more generally available. The Eclipse Parallel Tool Platform (PTP[4]), an open-source Eclipse Foundation[5] project, and NVIDIA's parallel Nsight[6] are the most popular tools for supporting GPGPU programming. The features of Eclipse PTP include a standard and portable parallel IDE, and a scalable parallel debugger (however, the debugger has not yet been extended to a GPU platform). NVIDIA parallel Nsight enables GPU programming in Microsoft's Visual Studio 2008[7]. It supports debugging, profiling, and code analysis of OpenCL, CUDA, and DirectCompute programs. In addition to being just an editor for the development of parallel programs, the intent of our current work is to assist the programmer in executing parallel blocks in a GPU with minimum intervention from the user about language-specific details.

In this paper, we introduce a tool named CUDACL as a first step toward enabling traditional C/Java programmers in writing GPU programs. With CUDACL, we address the following questions: 1) Is it possible to allow programmers to use GPUs as just another tool in the IDE? 2) A programmer often selects a block of code and specifies the device on which it is to execute in order to understand the performance concerns. Should performance tuning be at a high-level of abstraction, such as that similar to what a Graphical User Interface (GUI) developer sees in a What You See Is What You Get (WYSIWYG) editor? 3) A simple analysis for CUDA or OpenCL reveals that there are few configuration parameters, but many technical details involved in the execution. How far can default values be provided without much performance loss? 4) How much information should be shown to the user for performance tuning and how much information should be hidden to make the programming task more simplified?

CUDACL supports OpenCL and CUDA for Java and 'C' programming. We have selected Java and 'C' because of their popularity[8]. 'C' programs are modified to 'C' with CUDA or OpenCL and Java classes are modified to Java with JOCL[9] or JCUDA[10]. JOCL and JCUDA are Java libraries providing Java bindings to OpenCL and CUDA, respectively. CUDACL is hosted within Eclipse as a plug-in. To understand the level of abstraction usually used by the programmer and also the details that can be hidden from the programmer, a detailed analysis of CUDA and OpenCL was carried out and is explained in Section 2. The design of CUDACL is explained in Section 3 using a simple `ArrayAdd` program. Section 4 includes an assessment of the project using two case studies taken from existing open source programs, one for CUDA with 'C' and another using OpenCL with Java. Related works are explained in Section 5 and the paper concludes by discussing some of the possible extensions of the current work.

2. Analysis

An analysis was done for both CUDA and OpenCL to explore the abstraction possibilities observed from their common capabilities. In the CUDA analysis, priority was given to the data flow of a GPU program, while OpenCL was analyzed to identify the templates used in an OpenCL program. Code for the analysis was collected from the code samples from the NVIDIA installation package, assumed to be written by expert GPU programmers.

2.1. Data flow analysis using CUDA

Data flow in the current context can be defined as the flow of data from GPU to CPU or vice versa, the flow of data between multiple threads, and the flow of data within the GPU (e.g., shared to global or constant). The flow of data within the GPU should be handled inside the kernel and is left to the programmer's preference; hence, our analysis was focused on the other forms of data flow. If thread synchronization is not used in the kernel code, it is assumed that the program has no data flow between the threads. There is a case where this is untrue: warps of 32 threads are synchronized at the instruction issue level by the warp scheduler, and some kernels take advantage of this fact to keep from having to do an expensive barrier while still sharing data within the warp. The use of local memory is a better indication of data sharing or communication between threads in that case.

An example code segment is shown in Figure 1. While executing the program in the GPU, the variables `h_A` and `h_B` are used as input variables and `h_C` is used as an output variable. For the analysis, only pointer variables are

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://www.eclipse.org/ptp/
5. http://eclipse.org
6. http://developer.nvidia.com/object/nsight.html
7. http://www.microsoft.com/visualstudio/en-us/products
8. http://langpop.com/
9. http://jocl.org
10. http://jcuda.org

```
// Copy vectors from host memory to device memory
cudaMemcpy(d_A,h_A,size,cudaMemcpyHostToDevice);
cudaMemcpy(d_B,h_B,size,cudaMemcpyHostToDevice);

// Invoke kernel
VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A,d_B,d_C,N);

// Copy result from device memory to host memory
// h_C contains the result in host memory
cudaMemcpy(h_C,d_C,size,cudaMemcpyDeviceToHost);
```

Figure 1. Level A: host code to invoke VectorAdd

```
if(doMultiBlock){

  MonteCarloKernel<<<gridMain, THREAD_N>>>(
      (__TOptionValue *)plan->d_Buffer,
      plan->d_Samples,plan->pathN);

  MonteCarloReduce<<<plan->optionCount,THREAD_N>>>(
      (__TOptionValue *)plan->d_Buffer, accumN);
}else{
  MonteCarloOneBlockPerOption<<<plan->optionCount,
      THREAD_N>>>
      (plan->d_Samples,plan->pathN);
}
```

Figure 2. Level C: host code from MonteCarlo program in the NVIDIA CUDA installation package

considered because other variables are available in the GPU without explicitly copying (`N` in the Figure 1) the variables. The code segment is from the host program in the code samples of the NVIDIA CUDA installation package. As a general rule, for a GPU call from a CPU, the input variables should be copied from CPU to GPU before the GPU execution and output variables should be copied back to the CPU after execution. There can be exceptions as revealed by the following analysis. Instead of explicit copy operations, memory can be mapped between device and host variables. Because this is an implementation detail, we consider variable mapping also as two copy operations.

Based on the data flow, programs are grouped into three levels. If the kernel code does not communicate between threads, input variables are copied to the GPU before execution, and output variables are copied back to main memory after execution of a single kernel. We classify this form of data flow as level A. Missing thread synchronization or the use of local memory in kernel code is an explicit sign of level A. As an example, the code segment shown in Figure 1 belongs to level A. Thread synchronization in level A programs are classified as level B. In this form of data flow, the input variables are copied to the GPU and the output variables are copied back to the CPU before and after execution, but there is synchronization between multiple threads. The distinction is made because GPU code for level A programs can be generated directly from the sequential code, as explained in Section 3. In the case of level C programs, the variables are not copied back and forth. This is mainly done to improve the performance of the GPU programs. In level C, the variables are reused by other kernels, thereby creating a dependency between different kernels. Using Figure 2 as an example, if the `doMultiBlock` is set, it executes the `MonteCarloKernel` kernel followed by `MonteCarloReduce`, without making any memory transfer to GPU or from GPU. In this case, `MonteCarloReduce` is clearly level C. In general, for level C programs, all variables inside the kernel code will not be copied either to the GPU or back to the CPU.

For the analysis, 42 kernels were selected from 25 randomly selected programs that are provided as code samples from the installation package of NVIDIA CUDA. A summary of the results is shown in Table 1. The column IOS shows whether or not the input variables are copied before the call, the output variables are copied after the call, and thread synchronization appears in the parallel code, respectively. Conclusions drawn from the analysis are listed below:

- **Automatic Code Conversion**: There are a fair amount of programs; i.e., 48% (20) , whose code can be automatically generated if the sequential code is available.
- **Copy mismatch**: There exist programs; i.e., 12% (5) , where all the variables in the kernel are not copied. Even though the variables are not copied, memory should be allocated in the host code before the first access.

2.2. Program analysis of OpenCL

OpenCL supports the execution of programs in heterogeneous platforms (e.g., both GPUs and CPUs). Every OpenCL program includes a considerable amount of code that is used to initialize a program. As an example, in a program `oclMatrVecMul` from the OpenCL installation package of NVIDIA, three steps – 1) creating the OpenCL context, 2) creating a command queue for device 0, and 3) setting up the program – are achieved with 34 lines of code. This amount of verbose coding is necessary even though the above steps are common to most OpenCL programs. The situation is similar to the early stages of GUI programming where several dozens of lines were needed to simply create a new window. Our analysis was done with the intention of finding the frequently used steps of an OpenCL GPU program. If the steps for a general OpenCL program can be found, templates can be provided that can free the programmer from writing much of the common code manually. Furthermore, one or more steps can be abstracted to standard functions to simplify the development process.

As with the data flow analysis, 15 programs were randomly selected from the code samples that are shipped with the NVIDIA OpenCL installation package. Because this analysis was intended to extract the possible abstractions from an OpenCL program, other details like data flow

Table 1. CUDA program analysis

Program name	Kernel name	IOS	L
simpleMultiGPU	reduceKernel	$\surd\surd X$	A
3DFD	stencil_3D_16x16_order8	$\surd\surd\surd$	B
Marching cubes	classifyVoxel	$X\surd X$	C
	compactVoxels	$X\surd X$	C
	generateTriangles	$\surd\surd\surd$	B
	generateTriangles2	$\surd\surd\surd$	B
AsynchAPI	increment_kernel	$\surd\surd X$	A
cudastreams	init_array	$\surd\surd X$	A
matrixMulDrv	matrixMul	$\surd\surd X$	A
BicubicTexture	d_render	$\surd\surd X$	A
	d_renderBicubic	$\surd\surd X$	A
	d_renderFastBicubic	$\surd\surd X$	A
simpleTexture	transformKernel	$\surd\surd X$	A
MersenneTwister	BoxMullerGPU	$X\surd X$	C
	RandomGPU	$X\surd X$	C
Blackscholes	BlackScholesGPU	$\surd\surd X$	A
simpleTextureDrv	transform	$\surd\surd X$	A
MonteCarloMultiGPU	MonteCarloKernel	$\surd XX$	C
	MonteCarloReduce	$\surd\surd X$	A
	MonteCarloOneBlockPerOption	$\surd\surd X$	A
clock	timedReduction	$\surd\surd\surd$	B
simpleZeroCopy	vectorAddGPU	$\surd\surd\surd$	B
oceanFFT	calculateSlopeKernel	$\surd\surd X$	A
	generateSpectrumKernel	$\surd\surd X$	A
convolutionSeparable	convolutionColumnsKernel	$\surd\surd\surd$	B
	convolutionRowsKernel	$\surd\surd\surd$	B
SobelFilter	SobelShared	$\surd\surd\surd$	B
	SobelCopyImage	$\surd\surd\surd$	B
	SobelTex	$\surd\surd\surd$	B
postProcessGL	cudaProcess	$\surd\surd\surd$	B
cppintegration	kernel	$\surd\surd X$	A
	kernel2	$\surd\surd X$	A
sortingNetworks	bitonicSortShared	$\surd\surd\surd$	B
	oddEvenMergeGlobal	$\surd\surd\surd$	B
quasirandomgenerator	quasirandomGeneratorKernel	$\surd\surd X$	A
	inverseCNDKernel	$\surd\surd X$	A
template	testKernel	$\surd\surd\surd$	B
recursiveGaussian	d_transpose	$\surd\surd\surd$	B
	d_simpleRecursive_rgba	$\surd\surd X$	A
	d_recursiveGaussian_rgba	$\surd\surd X$	A
dwtHaar1D	dwtHaar1D	$\surd\surd\surd$	B
scalarProd	scalarProdGPU	$\surd\surd\surd$	B

and kernel code were avoided. The steps for the OpenCL programs are shown in Table 2. As can be observed from the table, all of the programs execute steps 1, 4, and 5. Step 2 provides an option to the user of the program to specify the devices on which he or she intends to execute the program. Step 3 is for executing the kernel on multiple devices.

The following conclusions were made from the analysis:

- **Default template**: Every OpenCL program consists of creating a context, setting up the program, and cleaning up the OpenCL resources.
- **Device specification**: Each kernel could be specified with a device or multiple devices in which the kernel is meant to be executed.

Table 2. OpenCL program steps

No	Step Description
1	create the OpenCL context on available GPU devices
2	if user specified GPU/GPUs create command queue for each
3	else find how many available GPUs create command queues for all create a command queue for device0 or a given device
4	program set up load program from file create the program build the program create kernel run calculation on the queues or queue
5	clean OpenCL resources

2.3. Discussion

From the CUDA examples, any GPU call could be considered as a three-step process: copy or map the variables before the execution, execute on the GPU and copy back or unmap the variables after the execution. An abstract representation of the three steps consists of: 1) `copyin(in_paramlist)`, 2) `callkernel(original_paramlist)`, and 3) `copyout(out_paramlist)`. In the code, `in_paramlist` refers to the list of variables that have to be copied to the GPU before execution, and `out_paramlist` refers to the list of variables that have to be copied back to the CPU, and `original_paramlist` refers to the list of original parameters required for the call.

From the OpenCL examples, to make OpenCL programming easier and faster, the steps identified could be written as functions and included as libraries with the newly written code. If the user is interested only in single device execution, a single method call `initOpenCL(devices, kernel_name)` which returns the `commandQueues` (used for memory transfer and kernel execution) can abstract all the details required for the execution of the kernel. `devices` is the list of devices on which the user intends to execute the kernel `kernel_name`. In the OpenCL programs analyzed, 33% (5) of the programs used multiple devices while 67% (10) used a single device for execution.

3. Design details of CUDACL

The overall design of CUDACL is shown as a block-diagram in Figure 3. In the figure, the inputs and outputs are shown as an oval shape, internal details are shown as boxes and the configuration file is shown as a double circle. From the discussion in Section 2, the general form for a GPU call supporting both CUDA and OpenCL would be `_GPUCall(kernel_name, in(in_paramlist), out(out_paramlist))`, given that device information has already been associated with the `kernel_name`, and

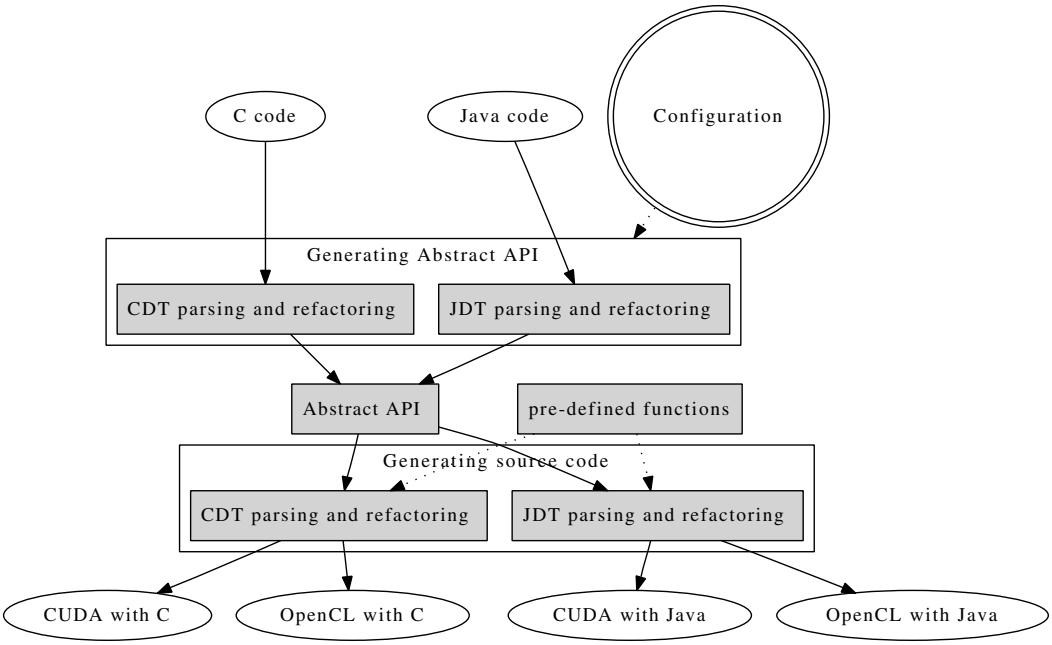

Figure 3. An overview of CUDACL

`original_paramlist` has been extracted from the program.

In our earlier work [13], we introduced abstract APIs that provide a general representation of GPU programs. The abstract APIs can be implemented for OpenCL, CUDA, or both. The abstract APIs also provide automatic management of GPU to CPU variable mapping. The programmer is freed from the task of creating the GPU variable, which is mapped to a CPU variable. Access to GPU variables can be achieved using the CPU variable and the necessary code for the mapping would be generated. One of the limitations of the earlier work was that to create the mapping, a programmer had to specify the size of the variable, even though the information could be retrieved from the CPU variable mapped to the new variable. The focus of the current project is to generate abstract APIs using static code analysis and a configuration file.

3.1. Configuring CUDACL for code generation

To execute a code block from sequential program code in a GPU, the user must provide several configuration settings. More informed and customized configuration settings can yield a faster program. For execution in a GPU, the work item (thread) size and work group (block) size has to be specified. In the existing architectures there can be three dimensions for the threads and two dimensions for the block. OpenCL even provides an API (`clGetKernelWorkGroupInfo`) to retrieve the best value of the work group for the given register usage, local memory usage, and device. CUDA provides an Excel™ sheet to calculate the best value for block size[11]. The user is provided an option to use the OpenCL API to find the best size of work items and then work group for a given problem size (variable), relieving the programmer from explicitly specifying the details. CUDACL also has the option to generate new parallel blocks with newly defined input and output variables. For each parallel block defined, a file is created with a method declaration (empty body for levels B and C, modified code for level A) for the kernel code. A device can be selected from the list of devices as the default execution device, but executing in multiple devices is not supported in the current design. A general scenario for creating a parallel block is explained below.

While the programmer wants to create a parallel block, he or she clicks the starting and ending line in the editor (Eclipse). CUDACL responds by highlighting the line numbers (Figure 8) and prompts for a name of the parallel block. On entering the name of the block, the tool creates a ".gpl" file and opening that file will show a screen as in Figure 4. In the form there are multiple sections. A short description of each of the sections is given below in the following.

3.1.1. Parallel blocks. Any file can have one or more parallel blocks. All the other sections are dependent on the parallel block.

3.1.2. Variables. In this section, the programmer specifies the variables that should be copied into the GPU before

11. http://cs.ua.edu/graduate/fjacob/CUDA_Occupancy_calculator.xls

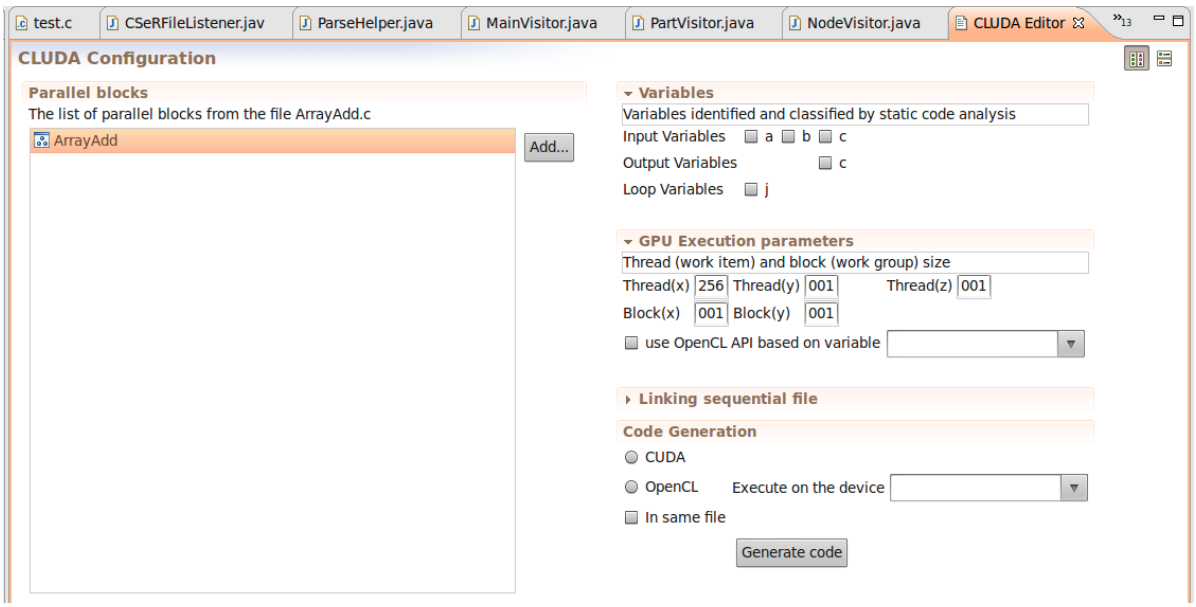

Figure 4. Configuring Execution of `ArrayAdd` using CUDACL

execution of the kernel, the variables that should be copied into the CPU after the execution, and also the loop variables. Loop variables are defined as any variable that the programmer is not interested in during the GPU computation. Initially, CUDACL shows the input, output, and loop variables based on the calculation explained in Section 2 and the programmer is free to manually configure these. The variable copy can be implemented in CUDA either using mapping or explicit copy (the current approach uses explicit copy).

3.1.3. GPU execution parameters. GPU execution parameters can be expressed in terms of integer numbers or in terms of any variable available in the context. There is another option to use the OpenCL APIs to find the work group or work item size.

3.1.4. Linking sequential file. This section links the parallel block defined by the programmer with the file. The fields of the section include starting line number, ending line number and name of the parallel block. The name of the parallel block would be the same as the generated GPU method. This helps to modify the scope of the parallel block from its initial value. Here, scope is defined in terms of line numbers in the source code.

3.1.5. Code generation. Code generation has options to generate CUDA or OpenCL code. For OpenCL, a specific target device can be specified. Even if the device is not specified, it finds out all of the devices and executes on the first one available. Another option in this section specifies how to separate the GPU code. If the option is not selected, the tool makes a copy of the main file and adds a "_GPU" to the name of the original file and generates GPU code in that file.

3.2. Level A programs

In level A programs, all the input variables are copied to the GPU before execution and all the output variables are copied back to the CPU after execution. These programs are assumed to be very similar to the sequential version; hence, a template is proposed to the programmer as a starting point for the kernel code. Figure 5 and Figure 6 show the abstract APIs generated for the host code and the corresponding kernel code, respectively. The configuration file for the program consists of the starting line number and ending line number in the host code, the position at which the code should be inserted, and the name of the GPU method. Configuration files are written for each file and for an entire source file there can be many parallel blocks defined. On activating CUDACL, it replaces the for loop (between start line number and end line number) with the newly generated abstract APIs. These abstract APIs can later be converted to CUDA or OpenCL by using the code generation option. The important steps involved in the code generation for level A programs are explained in the following:

3.2.1. Calculating arguments for the GPU call. The calculation of the arguments to be passed to the GPU is very similar to the extract method (JDT[12]) (extract function in

12. http://eclipse.org/jdt

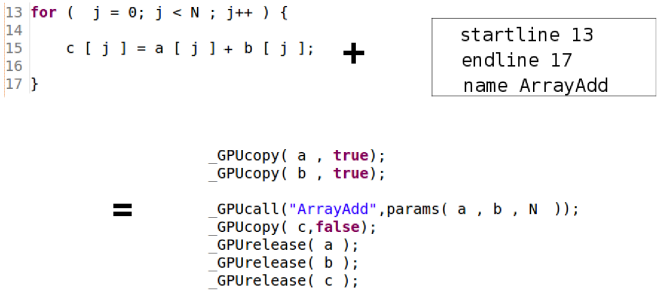

Figure 5. Host code of `ArrayAdd`

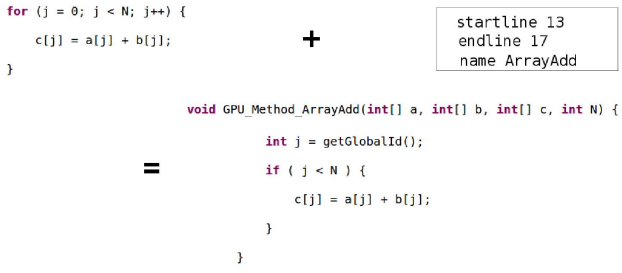

Figure 6. Kernel code of `ArrayAdd`

CDT[13]) refactoring [14]. The line numbers are read from the configuration file, the source code is parsed within the line numbers, and visitors can find the type and size of variables. Variables that appear in the left side of an infix expression (`expression operator expression`) are treated as output variables, and in the right side, as input variables. The variables created in the initialization expression of the `for` loop are treated as loop variables. From Figure 5, `a` and `b` are input variables, `c` is an output variable, and `j` is a loop variable. The arguments for the GPU call are `input_variables + output_variables - loop_variables`

3.2.2. Generating a template for the kernel. CUDACL can also be used to provide a template for the programmer as a starting point. From the lines specified in the configuration file, the loop (assuming that there is a loop) is unrolled and the body of the loop is moved to a new method with the name from the configuration file. The body of the loop statement is placed inside the `if` statement, and the conditional expression for the `if` statement is the conditional expression from the `for` loop as shown in Figure 6. The loop variable is maintained with the same declaration as in the host code, but initialized with a function call `getGlobalId()` for OpenCL and `blockDim.x * blockIdx.x + threadIdx.x` for CUDA. If the loop variable is initialized to anything other than 0, or is incremented by more than one, then the formula will not work.

13. http://eclipse.org/cdt

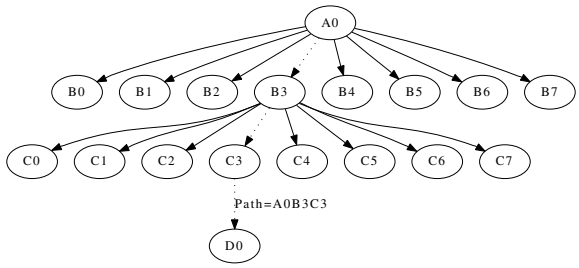

Figure 7. Octree path calculation

The formula will also not work in the case of problems with data in multi-dimensional arrays.

3.3. Level B and C programs

For level B and C programs, it is assumed that there is a different parallel implementation, so the programmer is expected to write the kernel code. CUDACL assists in calling the kernel functions from the sequential code, which is written in either 'C' or Java, with minimal input. As opposed to level A programs, the input variables and output variables that are required to copy are specified in the configuration.

4. Case Studies using CUDACL

This section explains the steps involved in developing two GPU programs using CUDACL. One program is written in 'C' and the second one is written in Java. The goals of these case studies are to illustrate and analyze how much of the details of the GPU specification can be hidden from the programmer. For the first program, which is a level A program, code generation was completely automatic. For the second one, which has parallel blocks of type B and C, code was generated after obtaining configuration parameters. The first program targets CUDA and the second targets OpenCL.

4.1. Creation of Octree with CUDA

Octree is a popular tree data structure that is often used to represent volumetric data. It performs recursive partitioning of 3D space, resulting in a hierarchical data structure that efficiently stores sparse volumetric data. Volumetric 3D data is widely used in computer graphics (e.g., to store object scenes and point clouds).

This implementation was motivated by the problem of creating a 3D shape from a point cloud [15], entirely on the GPU. A fixed depth Octree is created for a point cloud in a bottom-up approach, similar to the method by Zhou et. al [16]. Each node and point is assigned a path that is the sequence of nodes to be followed from the root node to reach the node or point. For a tree of fixed depth d, multiple points can have the same paths of length d, whereas the

```
9       /*compute path for each point*/
10      for(i = 0; i<pSize; i++){
11        ComputePath(points[i], depth, lBound,
12                    rBound, &path[i]);
13      }
14
```

Figure 8. Defining parallel block in the octree program

leaf tree nodes have unique paths. The creation of Octree involves finding paths for all the points, then creating leaf nodes for unique paths among them. Then, unique paths of the leaf nodes of length d-1 are found, which are used to find the parent nodes of the leaves. This is continued for each level of nodes created to find parent nodes, until the root node is created. The computation path for each point and node is independent, as shown in Figure 7. This process was parallelized to run on a GPU.

As shown in Figure 8, the programmer clicks on the starting and ending line numbers inside the editor (Eclipse) to define the parallel block. The lines are highlighted as shown in the figure, and upon clicking the second line number the tool prompts for the name of the method in the GPU (set to `ComputePath`). After the name has been entered, a file with extension ".gpl" is generated. The Octree program was executed with the default configurations (i.e., 256 threads per block), and the number of blocks determined by the amount of data. The execution time comparison of the sequential and parallel programs is shown in Figure 9. In the Figure, the X-axis shows the logarithmic value of the data size and the Y-axis shows the time to finish the execution in seconds. As seen in the figure, for small data size there is no remarkable advantage in using the GPU, which can be attributed to memory copy overhead, but there is a definite advantage when using a larger dataset. To show the performance of the current approach with hand-written CUDA code, a graph includes the plot of both hand-written code and code generated by CUDACL. From the figure, both work with almost the same performance increase. This is a reasonable and expected increase because in the generated code, other than the variable names and order of assignments, it is almost the same as the hand-written code.

4.2. Edge detection with OpenCL

Land type classification is an image processing problem which takes multiple images of the earth's surface as input, and attempts to classify each pixel as one of a handful of different land types. Farmland is found by using a pattern matching algorithm to search for large, flat, contiguous squares and circles on the edge map of the land, which is created by running a Sobel edge detector [17] over the daytime satellite imagery. In this case study, the integration of OpenCL kernel code into a Java program is explained.

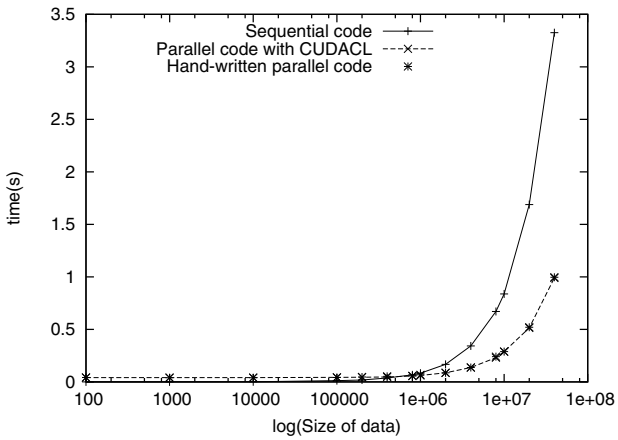

Figure 9. Octree in sequential and parallel

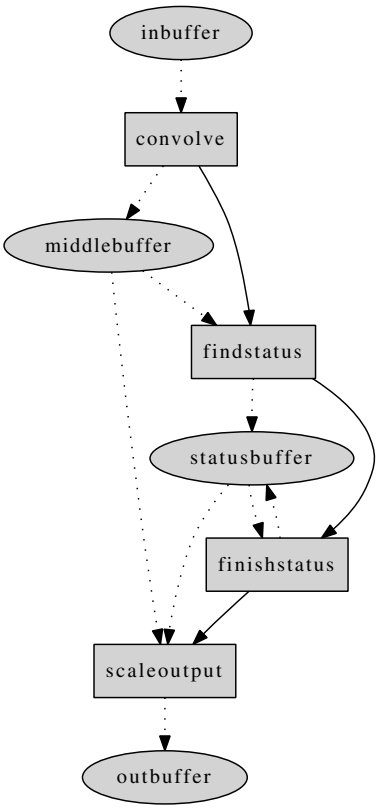

Figure 10. Four stages in the implementation of Sobel filter

An overview of the implementation of the Sobel filter is shown in Figure 10. First, the image is copied to the GPU and each pixel is convolved by a Sobel filter. The Sobel filter is a Gaussian function that is positive on the left, negative on the right, and zero down the middle. This has the effect of smoothing the image in the y direction, then finding the amount of change in the x direction. The filter is then rotated

by 90 degrees, and a second convolution is performed. The `convolve` kernel completes the operation by taking the length of the vector formed by the output of each filter, and stores the result in the `middlebuffer` image, which is kept on the GPU in float format to preserve accuracy. This image is an edge map of the original image, but its scale is unknown. The next step is to find the maximum value in the image. `findstatus` and `finishstatus` work together to complete this task. First, each block in `findstatus` finds it's local maximum value and stores it in the `statusbuffer`. Then, `finishstatus` is run in a single block, and reduces the `statusbuffer` to a single maximum value, stored in the first element of the array. Finally, `scaleoutput` reads the maximum value from the `statusbuffer`, and scales each pixel by the value (255 / max), so it can store the scaled value in the byte array `outbufffer`. An example image and its output from the Sobel operation is shown in Figure 11.

As an observation, all the parallel blocks work together with a varying number of inputs and outputs. CUDACL provides an option to upload the kernel file, which parses the input file and finds all the device methods. To call the kernel from the given context, a programmer selects the beginning and ending line numbers of the required context in the file and a ".gpl" file is created. On clicking the file, a screen shown in Figure 4 is presented to the user (with the parallel blocks from the file uploaded) and the user selects the input and output variables. In this case, there is one input to `convolve`, `findstatus`, and `finishstatus`, two inputs to `scaleoutput`, and one output from each stage. The code templates are generated to execute the parallel blocks with the specified configuration and the user has to fill the template variables with context variables. The host code generated would be similar to Figure 5, but the variables in the arguments of the GPU calls would be empty variables.

The equivalent CPU and GPU algorithms were run on two different images, a small 768x512 image and a larger 3616x3616 image. For the small image, the total CPU runtime was 263 ms, of which 257 ms was actual computation time. The equivalent OpenCL program on the same image ran in 251 ms total, of which only 5 ms was processing. The bulk of the time was spent in setting up the GPU, allocating memory, and copying buffers. In fact, 235 ms were used for one-time system setup and compile time. If this algorithm were used in batch mode for processing many files, then even small files benefit greatly from OpenCL, since each can be completely processed, including buffer copies, in 16 ms. For the large image, the CPU took a total of 8394 ms, spending 8267 ms on processing. On the other hand, the GPU spent a total of 396 ms, spending 235 ms in initialization and 119 ms in buffer copies, but only 42 ms in actual processing. The CPU used was an AMD Phenom II x4 clocked at 3.4 GHz and GPU used was an NVIDIA GeForce GTX 260.

(a) Input

(b) Output

Figure 11. Sobel operator on a test image

5. Related Works

Support for 'C' and Java programs to generate both CUDA and OpenCL makes CUDACL unique among related work in the GPGPU area. The related works for the project are represented by two categories:

1. Executing sequential code in parallel: There has been much effort in converting sequential code to parallel code even before the introduction of the GPU. Automatic parallelization of sequential code may not be the perfect solution for some cases, as revealed by our analysis. Parallel Fortran Converter (PFC) [1], [2] is a program to convert sequential programs written in Fortran to Fortran 8x, which is a version of Fortran compatible with vector computers. This program replaces the loops with array operations wherever possible by studying the data dependency in the program. PAT [3] is another tool supporting interactive conversion of sequential code to parallel code in Fortran. Several approaches [4], [5], [18] using program comprehension have been tried for automatic parallelization of code, but most of them are targeted towards comprehension of the numerical codes. Zhou et al. introduced a technique to enable the Java compiler to do loop transformations and parallelization

[6]. CONCURRENCER [19] is a tool that can refactor Java sequential code to Java parallel code. All these works were done before the introduction of GPUs, but our method is very similar in the way that it uses data dependency graphs and refactoring. OpenMP to GPGPU [20], which converts OpenMP programs to CUDA programs, is one of the few efforts in automatic code generation in the domain of GPU computing. This paper includes converting a sequential program to parallel with a simple algorithm that is applicable to a limited number of cases, but more focus has been given toward integrating GPU programming concepts into the IDE in a manner that is independent of language and framework.

2. Abstractions in parallel programs: Cg [11] is considered to be the first real high-level GPU programming language, but its usage requires the programmer to know the hardware details in order to generate efficient code. Similar to Sh [21], Cg's application domain is graphics. Brook [22], another GPU language, hides the hardware details from the programmer but it suffers from lightweight communication. CGIS [23] provides abstraction to support multiple target devices, where the high-level program specification provided by the user is converted to code based on the target device. Some of the key features of CGIS include parallel control structures, special vector operators, powerful data structures and user-specified hints. In the work presented in this paper, the focus is on bringing GPU programming to an IDE (e.g., Eclipse).

Many efforts have investigated high-level abstractions for CUDA. Using PGI [14], programmers can add the directives inside Fortran or 'C' code to mark the parallel region and the computation intensive part is executed in an accelerator-like GPU. To the best of our knowledge, PGI is currently supported only by GPUs from NVIDIA. With CUDACL, a kernel code written in CUDA or OpenCL can be called from 'C' or Java code, without changing kernel code. *hi*CUDA [9] generates CUDA code from directives inside the sequential 'C' code. CUDA-lite [10] works on annotations included in the sequential code to generate the host code and execute the kernel code as specified by the programmer, in a manner similar to CUDACL. The CuPP [24] framework assists a programmer in integrating CUDA with C++ programs. It includes automatic device/host memory management, parallel data structures and kernel call semantics. OpenCL is an emerging standard that was first released as a stable version in 2008. OpenCL is missing core features that programmers expect to incorporate in the latest releases [25]. To the best of our knowledge, CUDACL is the first work on abstraction of the OpenCL language. Hence, support for both CUDA and OpenCL in Java and 'C' programs makes CUDACL unique.

14. http://www.pgroup.com

6. Conclusion and Future Work

With the advent of the GPGPU and advances in the performance and availability of GPUs, there has been an effort to port more algorithms into the GPU context. This demands more programmer friendly APIs and tools that are focused on the concepts of GPU programming. Abstract APIs can provide a general interface for creating GPU programs. By generating abstract APIs using static code analysis, programmers can define parallel blocks within the sequential code. A graphical interface is available in CUDACL with default values to configure a GPU parallel block in 'C' and Java programs to generate CUDA and OpenCL programs. Two case studies were presented to show that CUDACL, and its associated tool support within Eclipse, can be useful for a wide range of applications. The tool was successfully used in the development of two case studies and a performance comparison revealed that CUDACL does not introduce any performance concerns. A comparative discussion with related works argued for how CUDACL is different from other works in the GPGPU area.

The major limitations of CUDACL are in template generation and finding the size of variables. The templates are only useful in the case of level A programs. The variable calculation can yield incorrect results if there is a conditional memory allocation. In the 40 programs we analyzed, none of the programs had conditional memory allocation. Supporting multiple devices will be the next goal for the project. As an extension of the work, a Domain-Specific Language (DSL) is being designed to represent the interface for the configuration. The DSL is intended to show the data dependency of the different GPU calls. More examples should be tried to evaluate the tool. Extending the work for other frameworks like OpenMP or MPI is another direction of this work.

Acknowledgments

This work was supported in part by the National Science Foundation awards CNS-0821497 and CCF-0643725.

References

[1] J. R. Allen and K. Kennedy, "Automatic loop interchange," in *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, Montreal, Canada, June 1984, pp. 233–246.

[2] R. Allen and K. Kennedy, "PFC: A program to convert Fortran to parallel form," *In Supercomputers: Design and Applications*, pp. 186–203, 1984.

[3] W. F. Appelbe, K. Smith, and C. E. McDowell, "Start/pat: A parallel-programming toolkit," *IEEE Software*, vol. 6, no. 4, pp. 29–38, 1989.

[4] B. D. Martino and G. Iannello, "PAP recognizer: A tool for automatic recognition of parallelizable patterns," in *Proceedings of the 4th International Workshop on Program Comprehension (WPC '96)*, Berlin, Germany, March 1996, p. 164.

[5] C. W. Kessler, "Pattern-driven automatic program transformation and parallelization," in *Proceedings of the 3rd Euromicro Workshop on Parallel and Distributed Processing*, San Remo, Italy, January 1995, p. 76.

[6] P. V. Artigas, M. Gupta, S. P. Midkiff, and J. E. Moreira, "Automatic loop transformations and parallelization for Java," in *Proceedings of the 14th International Conference on Supercomputing*, Santa Fe, NM, May 2000, pp. 1–10.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] W. R. Mark, R. Steven, R. S. Glanville, K. Akeley, and M. J. Kilgard, "Cg: A system for programming graphics hardware in a C-like language," *ACM Transactions on Graphics*, vol. 22, pp. 896–907, 2003.

[12] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, pp. 80–113, 2007.

[13] F. Jacob, R. Arora, P. Bangalore, M. Mernik, and J. Gray, "Raising the level of abstraction of gpu-programming," in *Proceedings of the 16th International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, July 2010, pp. 339–345.

[14] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[15] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle, "Surface reconstruction from unorganized points," in *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, Chicago, IL, July 1992, pp. 71–78.

[16] K. Zhou, M. Gong, X. Huang, and G. Baining, "Data-Parallel Octrees for surface reconstruction," *IEEE Transactions on Visualization and Computer Graphics*, vol. PP, no. 99, pp. 1 –1, May 2010.

[17] N. Kanopoulos, N. Vasanthvada, and R. L. Baker, "Design of an Image Edge Detection Filter Using the Sobel operator," *IEEE Journal of Solid-State Circuits*, vol. 23, no. 2, pp. 358–367, 1988.

[18] B. Di Martino and C. W. Keβler, "Two program comprehension tools for automatic parallelization," *IEEE Concurrency*, vol. 8, no. 1, pp. 37–47, 2000.

[19] D. Dig, J. Marrero, and M. D. Ernst, "Refactoring sequential Java code for concurrency via concurrent libraries," in *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, Vancouver, Canada, May 2009, pp. 397–407.

[20] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: a compiler framework for automatic translation and optimization," in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, February 2009, pp. 101–110.

[21] M. McCool and S. D. Toit, *Metaprogramming GPUs with Sh*. AK Peters Ltd, 2004.

[22] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for gpus: stream computing on graphics hardware," *ACM Transactions on Graphics*, vol. 23, pp. 777–786, 2004.

[23] 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.

[24] 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.

[25] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, 2010.