

Raising the Level of Abstraction of GPU-programming

Ferosh Jacob¹, Ritu Arora², Purushotham Bangalore²,
Marjan Mernik^{3,2}, Jeff Gray¹
jacobf@ua.edu, {ritu, puri, mernik}@cis.uab.edu,
gray@cs.ua.edu

¹Department of Computer Science
University of Alabama, Tuscaloosa, Alabama

²Department of Computer & Info Sciences
University of Alabama at Birmingham, Birmingham, Alabama

³Faculty of Electrical Engineering and Computer Science
University of Maribor, Slovenia

Introduction

- GPU Programming
- CUDA and OpenCL
- Why Abstraction?

Example Scenario

- Sequential: Vector Addition
- CUDA and OpenCL: Vector Addition
- Parallel: Vector Addition

Implementation

Experimental Results

- Source Code comparison
- Execution time Comparison

Related Works

Conclusion and Future Work

- An editor for GPU programming

Why GPU Programming?

- Extensively parallel programs
- Excellent computational platforms for scientific calculations
- Introduction of GPGPU
- Desktops and laptops with GPUs
- OpenCL, CUDA, DirectCompute

Overview: CUDA and OpenCL

Similarities

- C language modified
- Used in GPGPU
- Follows a SPMD

Differences

- CUDA \Rightarrow NVIDIA, OpenCL \Rightarrow Khronos
- OpenCL multi-vendor support
- OpenCL is still evolving
- OpenCL \Rightarrow heterogeneous

Abstraction in GPU

- Allows programmer to focus on essence of parallel computing, rather than language-specific accidental complexities of CUDA or OpenCL
- Higher level similarity in program structure
- Duplicated code
- Need to support heterogeneous architecture
- Higher abstraction allows the programmer to focus on the important issues while technical infrastructure is transparent

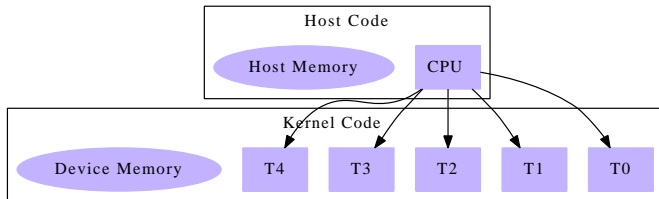
Source code for Vector Addition

```
1  /*
2  *   SEQUENTIAL CODE
3  */
4  float* h_A = (float*) malloc (mem_size_A);
5  float* h_B = (float*) malloc (mem_size_B);
6  float* h_C = (float*) malloc (mem_size_C);
7
8  initarray(h_A, h_B);
9  sequentialAdd(h_A, h_B, h_C);
10 printArray(h_C);
11
12 //Release Memory
13 free(h_A);
14 free(h_B);
15 free(h_C);
```

CUDA and OpenCL

Five Steps

1. Allocate memory in host
2. Copy memory from host to device
3. Execute the kernel
4. Copy memory from device to host
5. Release all memory



API correspondence

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

Source code comparison of parallel to sequential

```
1  /*
2   *   PARALLEL CODE
3   */
4
5  //Set GPU execution parameters
6  _GPUinit(16,16,4,4);
7  _XPUmalloc(h_A, "float*", mem_size_A);
8  _XPUmalloc(h_B, "float*", mem_size_B);
9  _XPUmalloc(h_C, "float*", mem_size_C);
10
11  initarray(h_A, h_B);
12  _GPUcall("arrayAdd", in(h_A, h_B), out(h_C));
13  printArray(h_C);
14
15  //Release Memory
16  _XPUrelease();
```

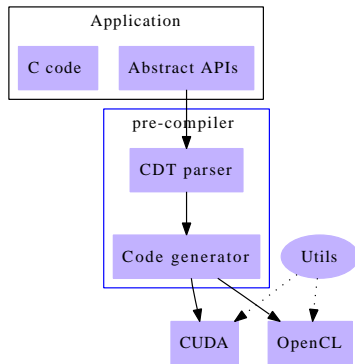
Generated code for CUDA and OpenCL

```
1  /*
2   * CUDA code _XPUmalloc(h_C,"float*",memsize_C)
3   */
4  float* h_C = (float*) malloc (mem_size_C);
5  float* _GEN_PREFIX17;
6  cutilSafeCall(cudaMalloc((void**)&_GEN_PREFIX17 ,
7                          mem_size_C));
```

```
1  /*
2   * OPenCL code _GPUcall("arrayAdd", in(h_A,h_B),out(h_C));
3   */
4  err = clSetKernelArg(kernel[0],0, sizeof(cl_mem),&h_A);
5  err |= clSetKernelArg(kernel[0],1, sizeof(cl_mem),&h_B);
6  err |= clSetKernelArg(kernel[0],2, sizeof(cl_mem),&h_C);
7
8  assert(err == CL_SUCCESS);
9  clFinish(cmd_queue);
10
11 //Read the results back to host
12 err = clEnqueueReadBuffer(cmd_queue, h_C, CL_TRUE, 0, mem_size_C,
13                          (void*)_GEN_PREFIX20 ,
```

Overview

1. C Code with additional predefined functions
2. Function calls (Abstract APIs) changed to GPU code
3. Frequently used block of code are saved as functions in Utils

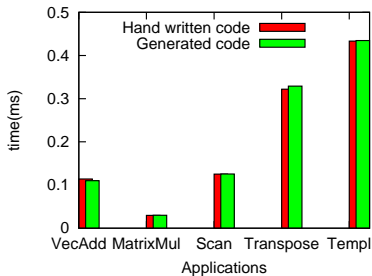


Experimentation

Application	CUDA	CPP	Abstr	# vars,lines	use
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

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

Execution time comparison



- Generated code performs as efficient as hand written for different problems

Related Works

- *hiCUDA* : Use directives in C code, tightly coupled with CUDA
- CUDA-lite : Use annotations in base code, targeted for CUDA programmers
- CGiS : A new GPU programming language with support for many devices, OpenCL not addressed
- CUPP : A tool to integrate CUDA into existing C++ applications
- Sh : Abstract layer for GPU languages but in graphics domain

Conclusion and future work

Conclusion

- An approach to improve GPU programming
- Static code analysis for host code
- No Performance loss(hand written code versus generated code)

Future work

- Need to generate kernel code for complete evaluation

CUDACL: GPU programming in Eclipse IDE

test.c CSeRFileListener.jav ParseHelper.java MainVisitor.java PartVisitor.java NodeVisitor.java CLUDA Editor

CLUDA Configuration

Parallel blocks
The list of parallel blocks from the file ArrayAdd.c

- ArrayAdd

Variables
Variables identified and classified by static code analysis

Input Variables a b c

Output Variables c

Loop Variables j

GPU Execution parameters

Thread (work item) and block (work group) size

Thread(x) 256 Thread(y) 001 Thread(z) 001

Block(x) 001 Block(y) 001

use OpenCL API based on variable

Linking sequential file

Code Generation

CUDA

OpenCL Execute on the device

In same file