

GPU PARALLEL COLLECTIONS FOR SCALA

By

KISHEN DAS KONDABAGILU RAJANNA

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2011

Copyright © by Kishen Das Kondabagilu Rajanna 2011

All Rights Reserved

ACKNOWLEDGEMENTS

Firstly, my sincere thanks to Dr. Nathaniel Nystrom; I could never have completed this work without his ideas and constant guidance. I am highly indebted to Mr. Derek White for all the co-operation and collaborations during the course of this work. I would also like to thank members of my committee Dr. David Kung and Dr. Christoph Csallner for their precious time and advice. Special thanks to Dr. Gutemberg Guerra-Filho for allowing me to work in his lab. I wish to acknowledge the support of all the faculty members at UTA.

Last but not the least; I would like to thank my wonderful parents Rajanna and Jayalakshmi and beloved wife Roopa Murthy for showing all the support and encouragement during my graduation.

March 25, 2011

ABSTRACT

GPU PARALLEL COLLECTIONS FOR SCALA

Kishen Das Kondabagilu Rajanna, M.S.

The University of Texas at Arlington, 2011

Supervising Professor: Nathaniel Nystrom

A decade ago, graphics processing units have been used specifically for high-speed graphics. Of late, they are becoming more popular as general purpose parallel processors. With the release of CUDA, ATI Stream and OpenCL, programmers can now split their program execution between CPU and GPU, whenever appropriate, resulting in huge performance gain. The cost of GPU is declining and also their performance is improving faster than CPUs. Although enormous performance gains can be achieved by parallelizing the code, identifying the right candidate for GPU execution is very tricky. Coding in OpenCL is a difficult task because of the memory management complexities of the GPU.

We are developing Firepile, a library and a compiler for GPU programming in Scala. Firepile makes it easier to port parallelizable code to GPU. As part of the library, we have been working on parallel collections that abstract the memory management and code complexities from the end-user. Constructs such as Array, Map, and Matrix have been provided as part of this library. Functions like map, reduce, scan, sort, find, and filter have been parallelized and implemented in the library that run on the GPU. Our experiments show that the library achieves performance similar to OpenCL implementation with much shorter, easier to understand code. Many of the inner details of the GPU architecture are hidden within the library. So, programmers need not understand the GPU architecture to achieve high performance.

Scala is a natural blend of functional paradigm and object orientation and is interoperable with Java. Hence it has been selected for the Firepile compiler over Java.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	iii
ABSTRACT.....	iv
LIST OF ILLUSTRATIONS.....	viii
LIST OF TABLES.....	ix
Chapter	Page
1. INTRODUCTION.....	1
2. BACKGROUND ON GPUS.....	3
2.1 GPU Architecture.....	3
2.2 OpenCL.....	5
2.3 GPGPU Library	7
3. BACKGROUND ON SCALA.....	10
4. FIREPILE COMPILER.....	12
4.1 Brief introduction	12
4.2 Data flow in Firepile.....	16
4.3 Using Firepile Parallel collections: An example	18
5. PORTING FUNCTIONS TO GPU.....	19
5.1 Reduce	19
5.2 Map	20
5.3 Scan	21
5.4 Filter.....	22
5.5 Find.....	24
5.6 Sort	25
6. RELATED WORK	27
7. BENCHMARKS.....	29
7.1 System Configuration	29
7.2 Results.....	29

8. CONCLUSIONS.....	35
9. FUTURE WORK	38
APPENDIX	
A. API	39
B. GLOSSARY.....	44
C. NVIDIA's OpenCL IMPLEMENTATION OF REDUCE KERNEL.....	51
REFERENCES	66
BIOGRAPHICAL INFORMATION.....	70

LIST OF ILLUSTRATIONS

Figure	Page
1 GPU Memory Hierarchy	3
2 GPU Hardware.....	4
3 OpenCL Memory Model	5
4 OpenCL Summary.....	6
5 Firepile Architecture	12
6 Firepile Memory Model.....	16
7 Reduce	19
8 Map.....	20
9 Scan.	21
10 Filter.....	23
11 Find.....	25
12 Time comparison for the Reduce function.....	31
13 Time comparison for the Map function.	31
14 Time comparison for the Scan function.....	33
15 Time comparison for the Filter function.	33
16 Time comparison for the Sort function.....	34
17 Lines of code comparison for various functions.....	34

LIST OF TABLES

Table	Page
1 Time comparisons for the function Reduce.....	30
2 Time comparisons for the function Map.....	30
3 Nvidia OpenCL implementation Vs Firepile, Scan.....	32
4 Nvidia OpenCL implementation Vs Firepile, Filter.....	32
5 Nvidia OpenCL implementation Vs Firepile, Sort.....	32
6 Nvidia OpenCL implementation Vs Firepile, Lines of code.....	32

CHAPTER 1

INTRODUCTION

As CPUs reach their upper limits on the clock rate, they are relying heavily on parallelism with additional processors to improve the throughput. With more number of processors, the cost also increases making it beyond the reach of layman. In recent days GPU computing is gaining momentum by leaps and bounds. A GPU has thousands of parallel units that can be utilized for parallelized code. Since the cost of GPU is dropping, it is becoming more affordable and with the release of OpenCL which removes the dependency of writing kernels for specific GPUs, GPU computing seems to hold lot of future. Still, languages like OpenCL [24, 25, 26], CUDA [28], and ATI Stream [29] are far from being simple and elegant languages to code for the developers at large. Programming models are highly restrictive without recursion, dynamic allocation and virtual method dispatch. Also, parallelizing the code is tricky and if not implemented properly can make it slower than a single core processor.

Functional languages are frequently used by the scientific community and language researchers, as it is easier to develop domain-specific languages and to experiment with compiler techniques [19] and type systems [20], respectively. Also, object orientation is a popular choice for business implementations. Since Scala offers best of both functional language and object orientation, it was a natural choice for this project.

OpenCL, although very similar to CUDA and ATI Stream in terms of coding complexity, removes the dependency of coding to specific series of GPUs.

My contributions are towards implementing the Scala reflection framework, where Scala compiler APIs have been made use of, for obtaining Scala object types and mapping that to corresponding Java types. Scala reflection framework has been used for type inference and for

class and function translations. Buffer backed array (BBArray, see section 4.2), an optimized collection for GPUs has been used for the first time in Firepile parallel collections, considerably reducing the data transfer time between GPU and CPU. A general approach for writing kernels in Scala using Firepile, defining various layers of OpenCL memory model has been designed and corresponding translation has been implemented using escape analysis. Designing the parallel collection APIs, porting them to the GPU architecture and also making them easily available to the end-user, abstracting the inner details of GPUs has been the main work of this thesis.

In this dissertation, I discuss the architecture of GPUs, reasons for selecting OpenCL over other GPU programming languages like CUDA and ATI Stream, writing kernels in OpenCL and its memory model, reasons for choosing Scala [18, 21, 22, 23] over Java, architecture and inner-working of Firepile compiler, which is mainly used for function translation and writing kernels in Scala, porting various higher-order functions to GPU architecture, relevant approaches of using GPUs for object oriented languages and how it differs from Firepile , comparing the performance of Firepile collections with ScalaCL, Scala parallel and sequential collections and OpenCL implementations, some of the approaches to further improve the performance of parallel collections and features that would be added in future.

There is a glossary at the end, for GPU related terminologies.

CHAPTER 2

BACKGROUND ON GPUS

Graphic Processing Unit (GPU) is a specialized circuit designed to rapidly process graphics data and to off load the same from the CPU. They have highly parallel structures and are much faster than micro-processors when dealing with graphics data as they execute large data blocks in parallel. Nvidia and ATI technologies are the companies that mainly produce GPUs. Hardware-accelerated 2D and 3D graphics cards with the support of graphics APIs by OpenGL and Direct3D existed until the late-90s. With the advent of the first GPU [1], GeForce 256, released in 1999 by Nvidia, GPUs have come a long way, starting as specialized graphics devices to being considered for general purpose computations. In general, GPUs are programmable, easy to install, cost-effective devices with hundreds of cores that show great performance for parallel programs.

2.1 GPU Architecture

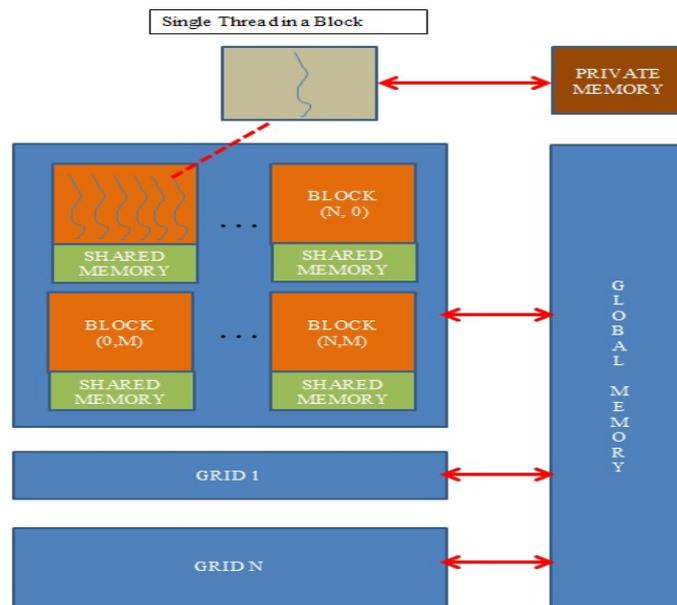


Figure 1 GPU Memory Hierarchy

A GPU grid is a collection of blocks. Block is a collection of threads. Multiple threads run in parallel in each block. Shared memory is visible to all the threads in a block. Each thread will have its own private memory. All the threads in every block have got access to the same global memory.

Apart from the aforementioned memory types, there are also constant and cache memory spaces that are read-only memory accessible to all the threads. The global, constant and texture memories are optimized for different memory usages such as communication across blocks, read-only data and texture data, respectively. Texture memory also offers different addressing modes and filtering, for some specific data formats like 2D & 3D texture data. General GPU hardware implementation is depicted in the figure 2.

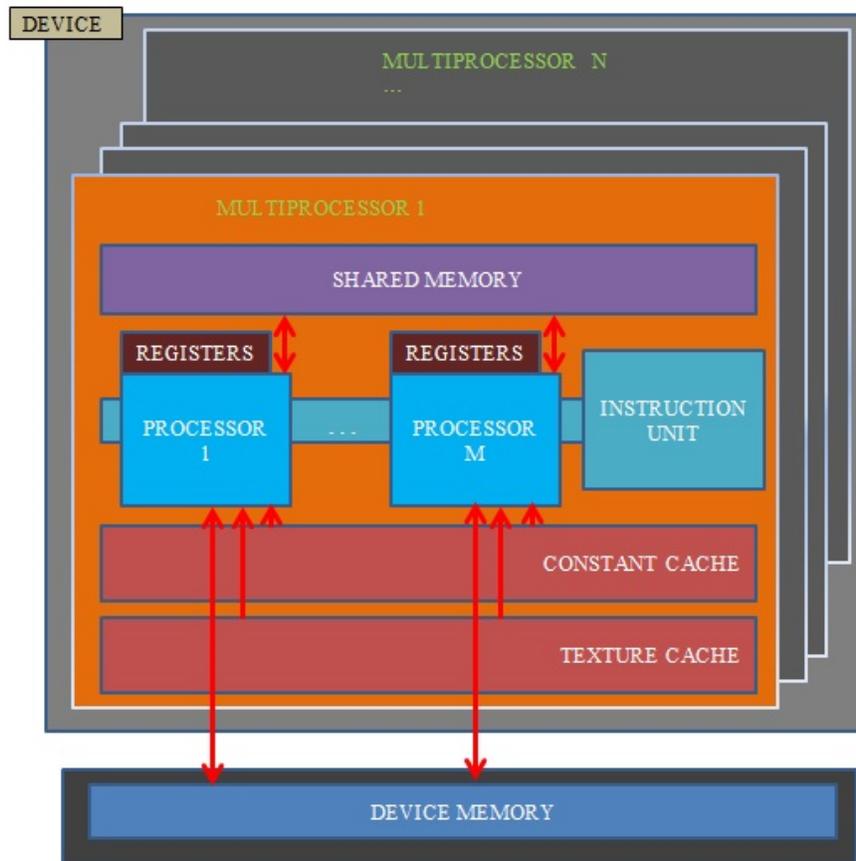


Figure 2 GPU Hardware

2.2 OpenCL

OpenGL [35], CUDA [28], ATI Stream [29] and OpenCL [24, 25, 26] are some of the languages available for GPU programming. OpenGL is only used for writing applications that deal with 2D and 3D graphics. CUDA and ATI Stream are coded for specific series of GPUs. We chose OpenCL for Firepile as it has a more generic model of programming. In OpenCL, both CPU and GPU can be used as a target device without any dependency on their hardware architecture. OpenCL lets programmers write a single portable program that uses all resources in the heterogeneous platform. Below is the memory model for the OpenCL.

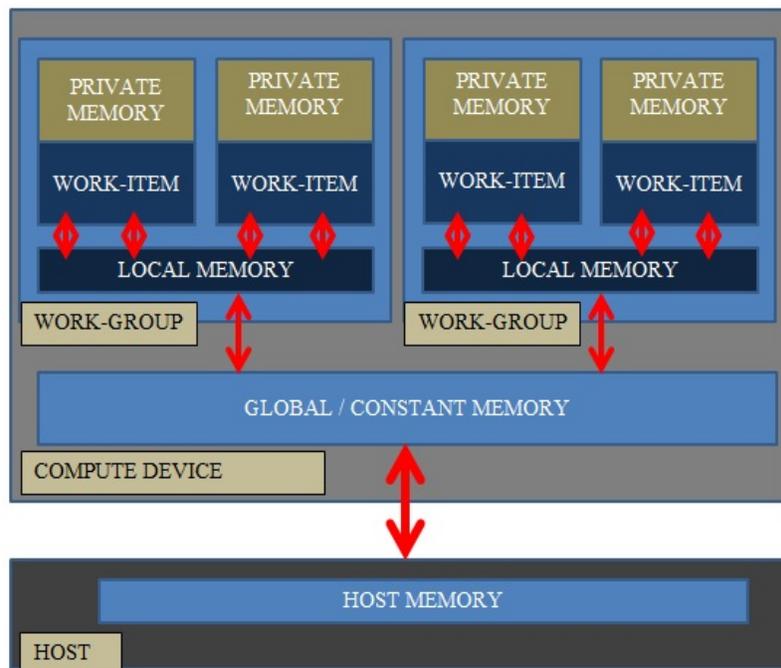


Figure 3 OpenCL Memory Model

OpenCL distinguishes five types of memories that are mapped to various memory spaces of a GPU. Private memory is per work-item or a thread in a block. Local Memory or block memory is shared within a workgroup i.e. between all the threads in a block. Global and constant memory is visible to all workgroups i.e. to all the threads of all the blocks. CPU RAM is

referred to as host memory. Movement of data is from host memory to the global or device memory of the GPU and then to the shared or local memory of block and back.

Use of local memory is optional but is preferred over global memory as it is much faster. Local memory cannot be shared across blocks. Ideally local memory is preferable when lot of processing within a block is done on the data which is copied from global memory. Synchronization between work-items is possible only within workgroups with the help of barriers and memory fences.

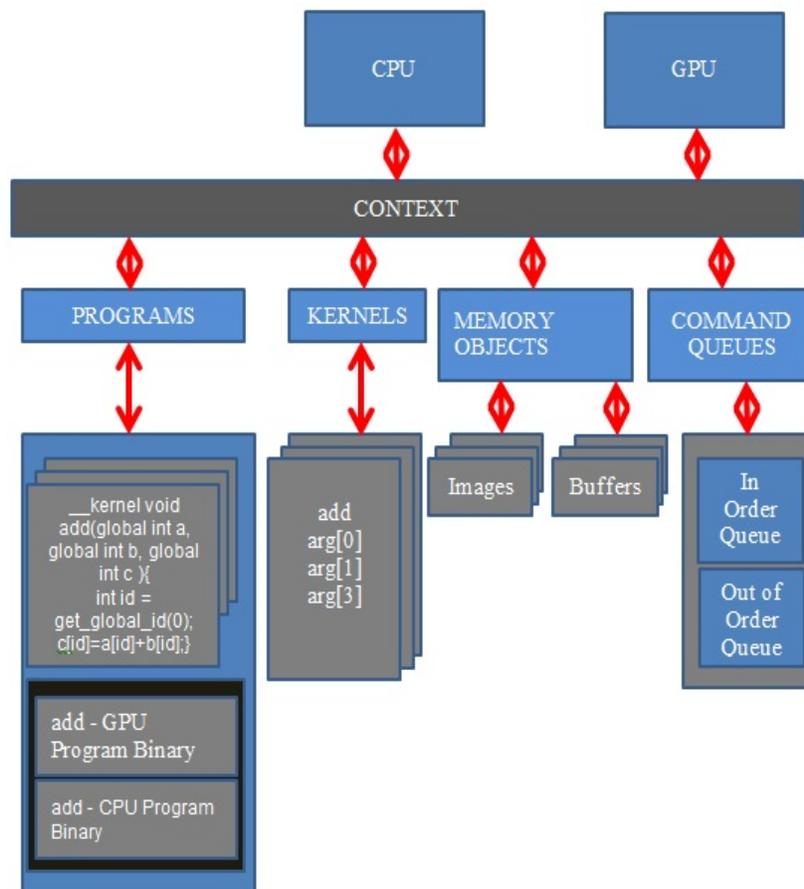


Figure 4 OpenCL Summary

Porting to GPU example: In this example, we will see how a for-loop is unrolled and executed on the GPU in parallel. Here 'add' is a function written in Scala that adds the corresponding elements of two arrays and puts the result in the third array. In the corresponding OpenCL

version, the block within the for-loop, can be realized as a kernel, where the two arrays are added in parallel for each index.

```
// Scala Source
def add ( a : Array[Float], b : Array[Float], c :
Array[Float]) : Array[Float] = {
    for ( i <- 0 until a.length)
        c(i) = a(i) + b(i)
    c
}
// Corresponding OpenCL kernel code
__kernel void add (global int a, global int b, global int c)
{
    int id = get_global_id(0);
        c[id] = a[id] + b[id];
}
```

To implement and execute a small kernel like the one described above, it will take minimum of 600 – 700 lines of code. Please refer to section 7.2 for lines of code comparison between Firepile and OpenCL for various functions. Since OpenCL has 5 different types of memories and parallelizing the code is very tricky, programmers find it difficult to adopt OpenCL for general computations. Firepile tries to address this point by providing parallel collections and a simple way writing kernels in Scala. Basic steps involved in executing a kernel are explained in the next section.

2.3 GPGPU Library

Following are the basic steps involved in executing a kernel on the GPU.

1. Creating OpenCL buffers

cl_mem **clCreateBuffer** (cl_context context, cl_mem_flags flags, size_t size, void *host_ptr, cl_int *errcode_ret)

2. Copying the data to the GPU

The following function enqueues command write to a buffer object from host memory.

cl_int **clEnqueueWriteBuffer** (cl_command_queue command_queue, cl_mem buffer, cl_bool blocking_write, size_t offset, size_t cb, const void *ptr, cl_uint num_events_in_wait_list, const cl_event *event_wait_list, cl_event *event)

3. Building the Kernel program

Function to create an OpenCL context

```
cl_context clCreateContext (const cl_context_properties *properties, cl_uint
num_devices, const cl_device_id *devices, void (*pfn_notify)(const char *errinfo,
const void *private_info, size_t cb, void *user_data), void *user_data, cl_int
*errcode_ret)
```

Below function creates a command queue for the GPU device with a valid OpenCL context.

```
cl_command_queue clCreateCommandQueue (cl_context context, cl_device_id
device, cl_command_queue_properties properties, cl_int *errcode_ret)
```

Function to build the kernel program.

```
cl_int clBuildProgram (cl_program program, cl_uint num_devices, const
cl_device_id *device_list, const char *options, void (*pfn_notify)(cl_program, void
*user_data), void *user_data)
```

Function to create the Kernel object

```
cl_kernel clCreateKernel (cl_program program, const char *kernel_name, cl_int
*errcode_ret)
```

4. Initializing Kernel arguments

Function to set a specific argument to a kernel.

```
cl_int clSetKernelArg (cl_kernel kernel, cl_uint arg_index, size_t arg_size,
const void *arg_value)
```

5. Launching the Kernel

Below function enqueues a command to execute the kernel on the GPU device.

```
cl_int clEnqueueNDRangeKernel (cl_command_queue command_queue,
cl_kernel kernel, cl_uint work_dim, const size_t *global_work_offset, const size_t
*global_work_size, const size_t *local_work_size, cl_uint num_events_in_wait_list,
const cl_event *event_wait_list, cl_event *event)
```

6. Copying the data back to the CPU

The following function enqueues command to read from a OpenCL buffer object to host memory

```
cl_int clEnqueueReadBuffer (cl_command_queue command_queue, cl_mem
buffer, cl_bool blocking_read, size_t offset, size_t cb, void *ptr, cl_uint
num_events_in_wait_list, const cl_event *event_wait_list, cl_event *event)
```

7. Freeing GPU resources

Function to release a particular OpenCL Buffer object.

`cl_int clReleaseMemObject (cl_mem memobj)`

There is good documentation available on the Nvidia and AMD websites [24, 25, 26] for OpenCL programming and hence a detailed discussion of each of the step has been left out. Also, see APPENDIX C for a complete OpenCL code of reduce.

CHAPTER 3

BACKGROUND ON SCALA

Scala [18, 22, 23, 24, 41] is an object-oriented and functional language. It is completely interoperable with Java. Hence, Firepile collections can also be used from Java. Scala also adds a uniform object model, implicits, traits, pattern matching and higher-order functions, novel ways to abstract and compose programs, extensible collections library, case classes, sealed classes, and first-class functions.

In Scala, every value is an object and every operation is a method invocation. Scala is a functional language, in the sense that every function is a value. Functions can be anonymous, curried, or nested. Familiar higher-order functions are implemented as methods of Scala classes. Firepile collections maintain the consistency of the logic for these higher-order functions.

Sealed classes are used for pattern matching. For Firepile, it is easier to implement virtual method dispatch as all the subclasses of a sealed class, as they should be in the same class.

Example:

```
sealed abstract class Insect
case class Bristletails(species : String) extends Insect
case class Butterfly(species : String, numberOfWings : Int ) extends Insect

def printSpecies( insect : Insect ): Unit = insect match {

  case Bristletails(species) => println(" The Bristletail species is:"+ species)
  case Butterfly(species,wings) => println(" The butterfly species is:"+species )

}
```

Implicits are very helpful in hiding the GPU details. In Firepile, implicits have been extensively used for implementing optimized collection called BBArry (Please see section 4.2).

Example:

```
def add( a:Int ) (implicit b:Int) = { a+b }

def func ( ) = {

    implicit val b : Int = 10
    println(" 10 + 4 is : " + add(4) )
}
// Here the variable 'b' in function 'func' is implicitly passed to the
// function 'add' and hence if you run the function 'func' it prints 14.
```

For Firepile, first-class functions are helpful in specialization and memorization of translated kernels. For example: `gpuArray.foldLeft(0)(_+_)` , here we can specialize the translated reduce kernel for the `'_+_'` function.

Scala treats algebraic data types [36, 37] as special cases of class hierarchies. Algebraic data type is a data type each of whose values is data from other data types wrapped in one of the constructors of the data type. Tuples and products belong to this category. No separate algebraic data types are needed as every type is a class. We can pattern match directly over classes. A pattern can access the constructor parameters of a case class.

With support of all the above features and many more libraries, it takes less code in Scala than Java to achieve the same thing. This increases the productivity and quality. Scala code compiles to bytecode and runs on the Java virtual machine(JVM), J2EE, servlets, all Java libraries etc.

To summarize Scala in brief: it is a structured, functional, object-oriented language with static type checking and safety. It is both compiled and interpreted language that runs on the JVM.

Logic of the programs can be quickly tested with interactive interpreters. General interest is growing for Scala among developers and many features planned for future Java releases are already there in Scala and hence it was picked up against Java.

CHAPTER 4
FIREPILE COMPILER

4.1 Brief introduction

The Firepile compiler translates relevant Java bytecode generated by the Scala compiler into OpenCL kernel and embeds that within Scala bytecode with JNA(Java Native Access) wrappers provided by NativeLibs4Java library [10]. Soot framework [15] has been used for the bytecode analysis. During this process, bytecode is translated into an abstract syntax tree which is later realized as OpenCL code. The architecture of the compiler is shown below.

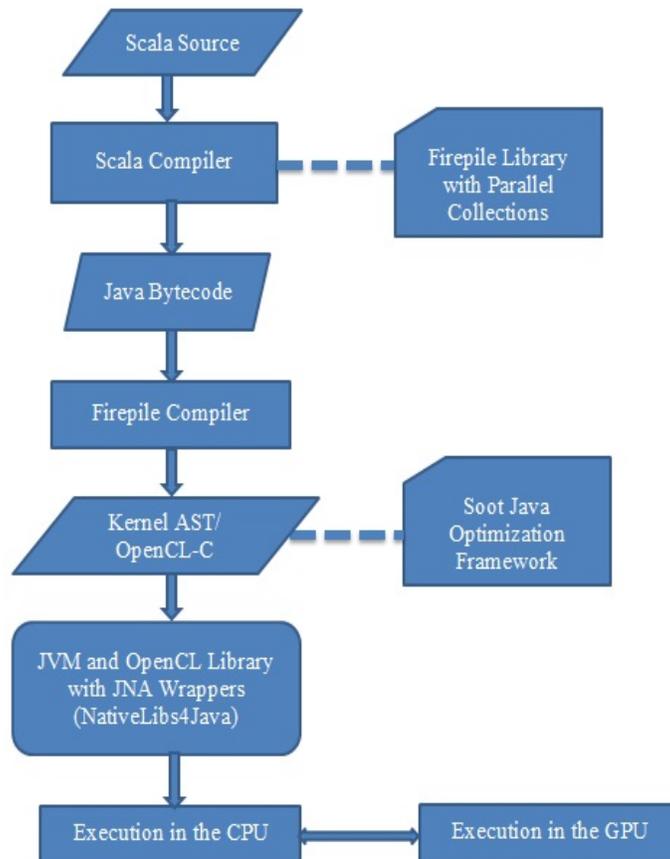


Figure 5 Firepile Architecture

Writing kernels using Firepile: Firepile makes it easier for the user to realize the GPU grid by providing a simple and unique style of writing kernels in Scala.

Below is an example of the reduce function written in Scala using Firepile –

```
def reduce(idata: Array[Float], f: (Float,Float) => Float)
(implicit dev: Device): Float = {
  // partition the problem by padding out to the next
  // power of 2, and dividing into equal-length blocks
  val space=dev.defaultPaddedPartition(idata.length)
  // Declare the output with size as number of blocks
  val odata = new Array[Float](space.blocks)
  val n = idata.length
  // spawn the computation on the GPU, returning an array
  // with one result per block
  space.spawn {
    // for each block, in parallel
    space.groups.foreach {
      g => {
        // Declare Local Array
        val sdata = Array.ofDim[Float](g.items.size)
        // for each thread, in parallel
        g.items.foreach {
          item=> {
            // Index for accessing the right thread
            val j = g.id * (g.items.size * 2) + item.id
            if (j < n)
              sdata(item.id)= idata(j)
            else
              sdata(item.id) =0f
            // do the first round of the reduction into per-block storage
            if (j + g.items.size < n)
              sdata(item.id) = f ( sdata(item.id), idata(j + g.items.size))
            // make sure subsequent rounds can see the writes to sdata
            g.barrier
            var k = g.items.size / 2
            while ( k > 0 ) {
              if (item.id < k)
                sdata(item.id) =f ( sdata(item.id) , sdata(item.id + k) )
              g.barrier
              k>>=1
            }
            //Copy the result back to global memory
            if (item.id == 0)
              odata(g.id) = sdata(0)
          } } } }
        // Do final reduction on the CPU and return the result
        odata.reduceLeft(_+_ )
      }
    }
  }
}
```

User initially creates the GPU grid and later writes the kernel which is passed onto the spawn method that does the translation. User can explicitly set the local and global work sizes. Otherwise it is pre-computed based on the problem size. There are constructs available for accessing GPU specific things like global Id, local id, group id, memory fence and barriers.

Function Translation: A function that it is to be translated as a kernel is considered as the root method and a complete call-graph is created for that function using Soot. In the above reduce example, the block that is being passed to spawn method is considered as a root method. Later Grimp body is obtained from the call-graph which gives us the list of SootUnits. SootUnits represent each of statement in the original Scala block. These SootUnits are translated to an abstract syntax tree and later to C language. Having an intermediate AST gives us the flexibility of translating them to any desired language. Below is an example of Scala function to C function translation.

```
// Scala function to add two floats
def add( a : Float , b : Float ) : Float = { a + b }

// Grimp method body
<float add (float,float)>
l1 := @parameter0: float
l2 := @parameter1: float
return l1 + l2

// Abstract Syntax Tree
FunDef(ValueType("float"),
  Id("add") ,
  List(Formal(ValueType("float"), Id("_arg0")),
    Formal(ValueType("float", Id("_arg1"))),
    Assign(Id(l1),Id(_arg0)),
    Assign(Id(l2),Id(_arg1)),
    Return(Bin(Id(l1),+,Id(l2))))))

// Translated 'C' function:
float add(float _arg0, float _arg1) {
  float l1;
  float l2;
  l1 = _arg0;
  l2 = _arg1;
  return l1 + l2;
}
```

Class Translation: Firepile creates a struct for each of the Scala class that is used in the kernel. Virtual dispatch is not supported on the GPU. So, Firepile makes use of sealed classes for accommodating the class inheritance. Since the compiler can identify all the sub-classes of a sealed class, Firepile creates a single struct which is tagged union of structs of all the subclasses of a given sealed class. Virtual dispatch is implemented by branching on the type tag and invoking the method using static dispatch.

Type inference: Type inference is required during both class and function translation. Some of the Scala compiler APIs have been extended to create the Scala Reflection Framework that gives us the complete structure of Scala classes. Once we analyze all the types used in the Soot units of Grimp, we map these Java types to the types obtained from Scala Reflection Framework.

Memory Analysis: Dynamic memory allocation is not available in the GPU. This restricts us to know the maximum memory usage in advance before the launch of GPU kernels, so that we can have the memory pre-allocated on the GPU. We are in the process of implementing some of the heap bound analysis techniques of Leena et. al. [38, 39, 40].

Escape Analysis: Escape analysis is used to identify the dynamic scope of all the variables to fit them into the best available GPU memory. An example is given below:

```
1    declare Array A
2    for loop1
3      declare Array B
4      for loop2
5        access Arrays A & B
6      access Array A
```

Here we can easily see that the array A has global scope and the scope of B is restricted to the body of 'for' loop1. If the above statements were to be executed on the GPU, array A can be realized as Global array and array B as Local array.

Below is a specific example of a Scala kernel written using Firepile that shows the

general structure of declaring variables used in the kernel and how they are mapped to various GPU memory spaces.

```
1 // Variables declared outside 'spawn' are mapped to global memory
2   val odata = new Array[Float](arraySize)
3
4   space.spawn {
5     // for each block, in parallel
6     //Declare variables in global memory here
7     val n = odata.length
8     space.groups.foreach {
9       g => {
10        // Declare variables in shared or local memory here
11        val sdata = Array.ofDim[Float](g.items.size)
12
13        // for each thread, in parallel
14        g.items.foreach {
15          item=> {
16            //Declare private variables exclusive to a thread here
17            val len = sdata.length
18            //Write kernel code here
19            ...
20          }
21        }
22      }
23    }
24  }
```

Here escape analysis is used to identify the scope of all the variables that are declared inside the spawn invocation. Depending on the layer at which they are declared, they will be mapped to relevant GPU memory spaces. The variables that are declared outside 'spawn' but used within the kernel code will have global space and are mapped to global memory. In the above example float array 'odata' is an example for this. Also, the variables that are declared inside 'spawn', but outside 'space.groups.foreach' block are also considered global. The integer 'n' belongs to this category. Similarly, the variables within 'space.groups.foreach' but outside 'g.items.foreach' block are considered local and variables declared inside 'g.items.foreach' block will be private variables that are exclusive to a thread. Float array 'sdata' is a local array and integer 'len' is a private variable in the above example.

BBArray: Fig. 6 depicts the Firepile memory model from the end-user perspective. Firepile provides byte buffer array called BBArray, where BB stands of ByteBuffer. The user has to only deal with BBArray that holds the data in a Byte Buffer. Since BBArray implements the trait ArrayLike, it has all the features of Scala array. Since BBArray knows about its type through

Class Manifest, implicit marshaling and un-marshaling is done when storing the data to and retrieving data from BBArrray. So, for the user, using BBArrray is no different from using a regular Scala array and its performance is lot better compared to regular Scala array when exchanging data with GPU.

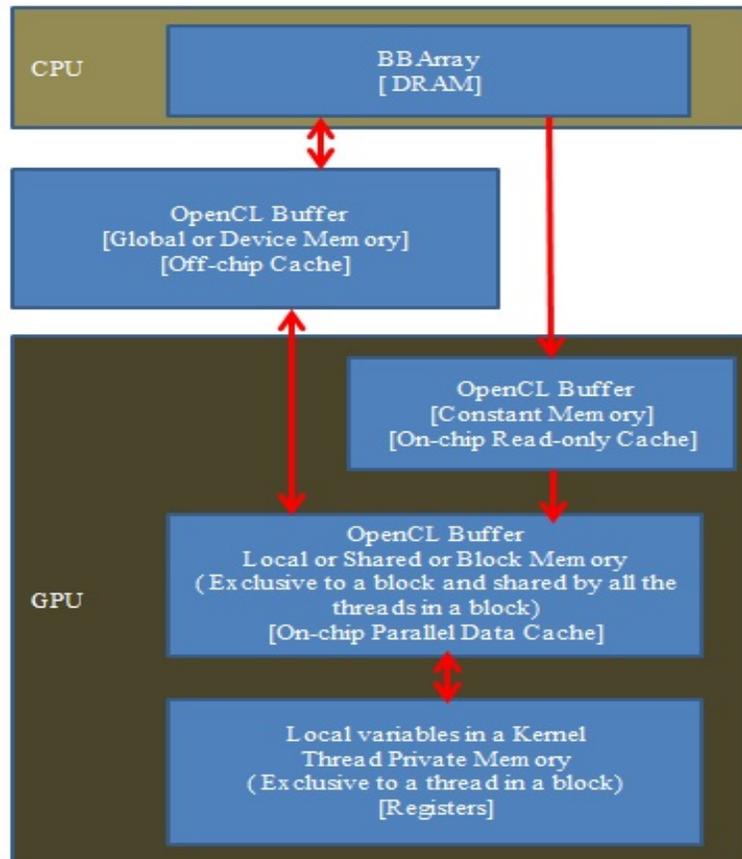


Figure 6 Firepile Memory Model

Performance: It takes 4 clock cycles to issue a read from global memory, 4 clock cycles to issue a write to local memory, but above 400 to 600 clock cycles to read a float from global memory. So, good performance is observed when local memory is used whenever possible. Accessing a register should not add any extra clock cycles, but there can be delays because of the register memory bank conflicts. Also, non-coalesced global memory access is much slower than coalesced memory access. Performance can also be improved by reducing bank conflicts

while accessing the local memory. OpenCL user guides [24, 25, 26] should be referred to better understand the performance implications of various memory usages.

4.3 Using Firepile Parallel collections: An example

In this example we will see how a collection API is used by the user.

```
Line 1  val array: Array[Float]
.....
Line 2  val arrayReduce : BBArry[Float] =
BBArry.fromArray(array)

Line 3  val gpuArray : GPUArray[Float] = new
GPUArray(arrayReduce, firepile.gpu)

// Reduce with _+_ operator
Line 4  val gpuSum: Float = gpuArray.reduce(_+_ )

// Sorting GPU Array elements
Line 5  val sortedArray = gpuArray.sort
```

In the above example, Line4 shows reduction operation performed on the GPUArray. Reduction function is also known as fold, accumulate, reject and compress. The basic idea of reduction is to iterate over a bunch of elements, repeatedly applying the same function and returning a single value. For example: Say if the Array has 5 elements [0 1 2 3 4], then the reduction with the operator “+” would return value of the expression $0 + 1 + 2 + 3 + 4$. This can also be pursued as $0 + (1 + (2 + (3 + 4)))$.

Here the function ‘_+_’ is translated to a C-function with the help of Firepile compiler. And later it is embedded into the reduction kernel code. The kernel code can be either hand-written or generated with the help of Scala Firepile library. The latter is still in the development stage, but many of the simple kernels can be written as of now. The same function implemented in C has around 500 lines of code with an additional 25 lines of kernel code. This is all abstracted in the parallel collections library and user can achieve the same result in couple of lines. Please see Appendix C for Nvidia’s OpenCL implementation of the reduce kernel. Also, see section 5.1 to understand how reduce is executed in the GPU in parallel.

CHAPTER 5

PORTING FUNCTIONS TO GPU

The basic idea of porting generic functions to GPU architecture is to convert each of the problems to smaller problems that fit in a GPU block, so that they can be executed in parallel. Many of the functionalities can be converted to Map-Reduce framework, where the steps of Map can be completely executed in parallel and even reduce can be executed in parallel at the block level.

5.1 Reduce

As explained in section 4.2, reduction [33] is to iterate over a list of elements, repeatedly applying the same function 'f' and returning a single value.

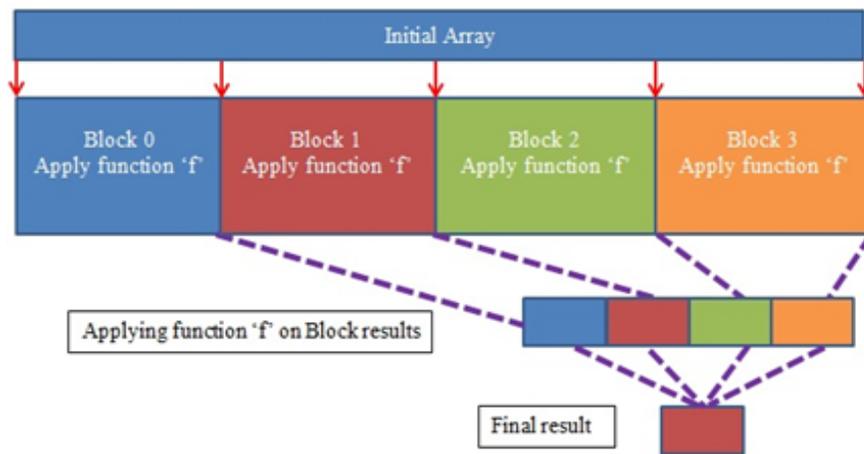


Figure 7 Reduce

In step 1, the input BBAarray is copied to OpenCL buffer in global memory. In Fig. 7, the input array in the global memory is shown as 'Initial Array'. In step 2, data is copied in parallel to the block or local memory, where the reduction happens at the block level with synchronization among the threads using memory barriers. In step 3, the result of each block is copied to an

array in the global memory and final reduction is done. Here the final reduction of block results can be done either in the GPU or CPU.

Example: Reduce for $f(a,b) \Rightarrow a + b$

Step 1. 1 2 3 4 5 6 7 8 [Input Array in Global Memory which is copied from BBArray]

Step 2. 1 2 | 3 4 | 5 6 | 7 8 | [Reduction in 4 blocks, in parallel]

Step 3. 3 | 7 | 12 | 15 | [Reduction of block results]

Step 4. 38 [Final Result]

5.2 Map

In map [33], each of the array elements is mapped to some other value determined by the function 'f'.

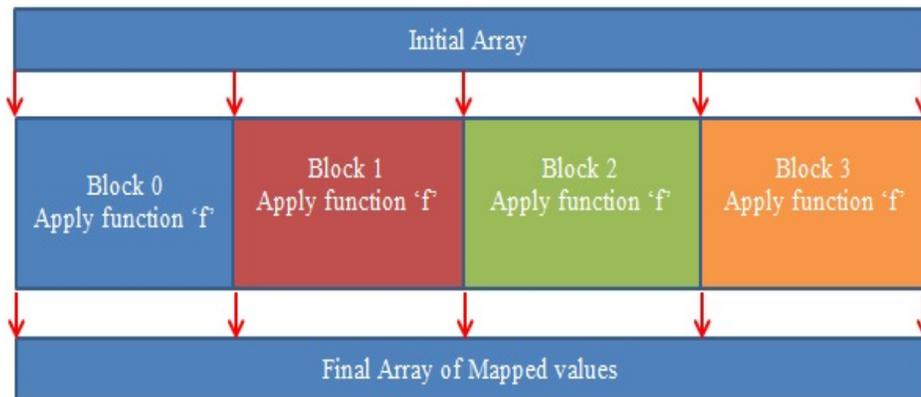


Figure 8 Map

In step 1, the input BBArray is copied to the OpenCL buffer in Global Memory. In step 2, each of the thread in a block executes the function 'f' on its corresponding global array element, in parallel. In step 3, result is copied back to the global memory.

Example: Map for $f(a) \Rightarrow a + 1$ [Function to be applied on each element]

Step 1. 1 2 3 4 5 6 7 8 [Input array in Global Memory which is copied from BBArray]

Step 2. 1 2 | 3 4 | 5 6 | 7 8 | [Apply function 'f' in parallel in each block]

Step 3. 2 3 | 4 5 | 6 7 | 8 9 | [Mapped elements in Global Memory]

Step 4. 2 3 4 5 6 7 8 9 [Final Result]

5.3 Scan

Scan [30, 31, 32, 33] is an operation on a list where each element in the result list is obtained by applying function 'f' on the elements of the original list until the index of the current element. Scan with addition as the operator is known as prefix sum or partial sum. The output of Scan is nothing but a list where each element is the result of reduction on the original list of elements till its index. Hence it's called partial reduction as well. There are two types of Scan: Inclusive and Exclusive.

Inclusive prefix sum: $[a_0, a_0 + a_1, a_0 + a_1 + a_2, \dots, a_0 + a_1 + a_2 + \dots + a_n]$

Exclusive prefix sum: $[/, a_0, a_0 + a_1, a_0 + a_1 + a_2, \dots, a_0 + a_1 + a_2 + \dots + a_{n-1}]$

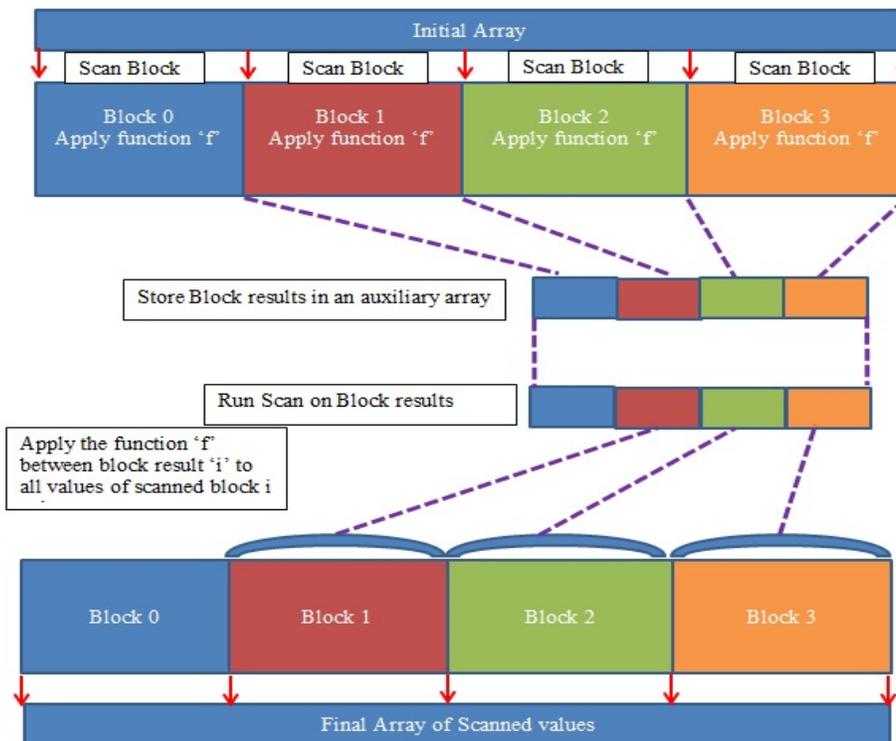


Figure 9 Scan

In step 1, input BBArry is copied to OpenCL buffer in global memory. In step 2, Scan is run for each of the block with synchronization among the threads using memory barriers. In step 3, we will have the scanned results for each block from the previous step, where each element of the block will be the result of the reduction until its relative index within the block. In step 4, last element of each of the block is copied to another array. In step 5, scan is executed within a single block for the array obtained in step 4. In step 6, we will have the scan result of the array obtained in step 4. Each element in this array represents the scan result until its own block. In step 7, each of the element of the array obtained in step 6, is added to each element of its appropriate block as shown in figure 9. In step 8, we will have scanned result for the entire input array in the global memory.

Example: Inclusive Prefix Scan $a_1 + a_2 + \dots + a_N$, Blocks of 2 elements

Step 1. 1 2 3 4 5 6 7 8 [Input array in Global Memory which is copied from BBArry]
 Step 2. 1 2 | 3 4 | 5 6 | 7 8 | [Execute Scan for each block]
 Step 3. 1 3 | 3 7 | 5 11 | 7 15 | [Scan Results for each block]
 Step 4. 3 | 7 | 11 | 15 | [Last element of each block from previous step]
 Step 5. 3 7 11 15 | [Again run Scan on block results within a single block]
 Step 6. 3 10 21 36 | [Scan result from previous step]
 Step 7. 1 3 | 3 7 | 5 11 | 7 15 | [Add the block results to the array obtained in the third step]
 Step 8. 1 3 | 6 10 | 15 21 | 28 36 | [Result in the global memory]
 Step 9. 1 3 6 10 15 21 28 36 [Final Output]

5.4 Filter

Output of a filter [31, 33] operation will be a list of elements of the original list that have passed the condition 'f'. Please look at the figure and the example to understand it's working.

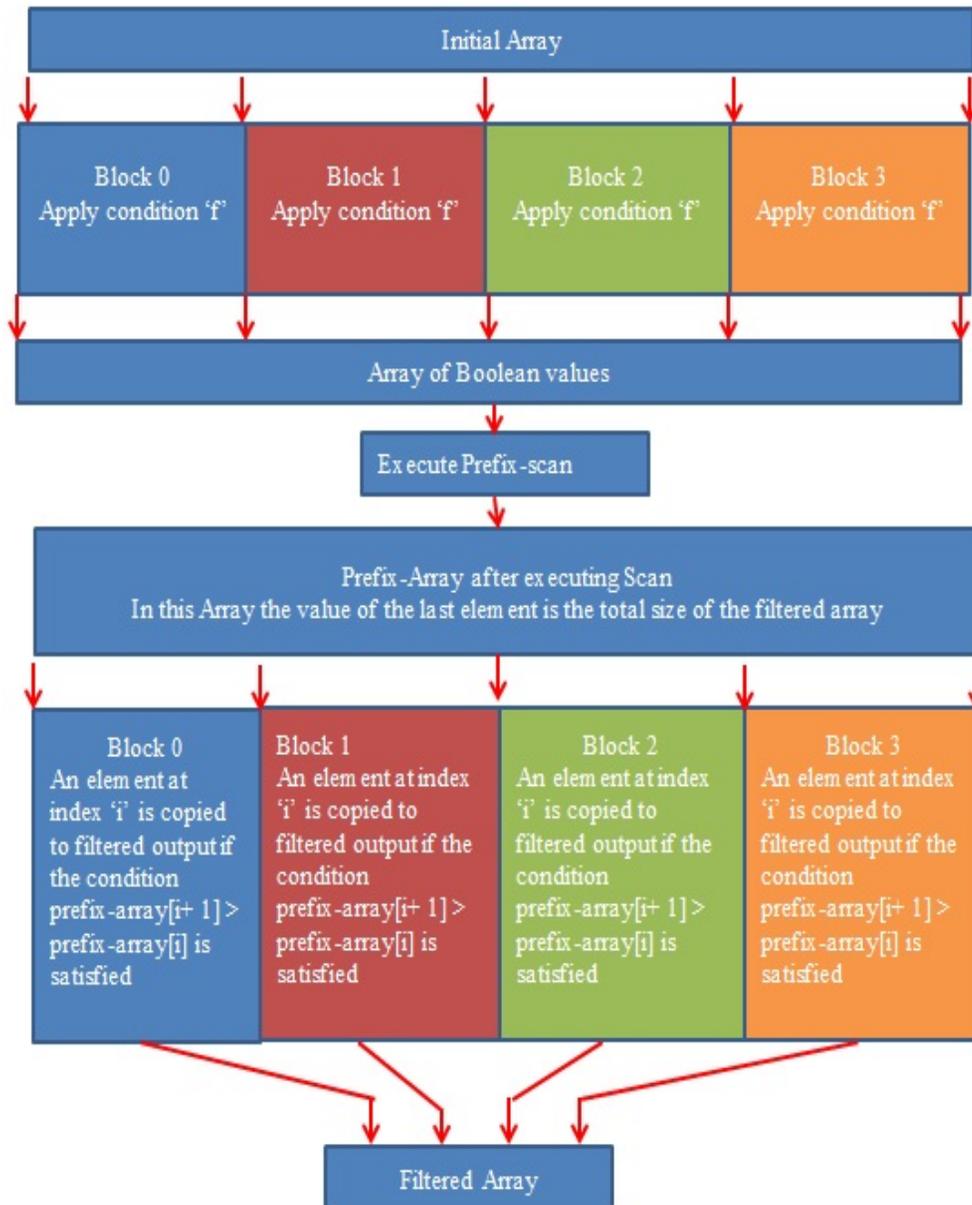


Figure 10 Filter

In the step 1, input BBArry is copied to OpenCL buffer in global memory. In step 2, condition 'f' is executed and the boolean result is stored in an array in global memory. In step 3, we will have an array of Boolean values obtained from step 2. In step 4, Scan is run in parallel for this boolean array. Please refer to section 5.3 to understand the parallel execution of Scan. In step

5, for each element in the array, the condition of whether Prefix-sum Array [i + 1] is greater than Prefix-sum Array [i] is checked. This will tell us whether the index 'i+1' of the original array holds the filtered element or not. If yes, then it will be one greater than the value of its previous index. If the condition satisfies, then the element at original-input-array [i + 1] is copied to the array of filtered elements. The final output is obtained in step 6.

Example: Filter with condition if (array[i] %2 ==0)

Step 1. 1 2 3 4 5 6 7 8 [Input array in Global Memory which is copied from BBArry]

Step 2. 1 2 | 3 4 | 5 6 | 7 8 | [Execute condition 'f' on the single element in each thread of various blocks]

Step 3. 1 0 | 1 0 | 1 0 | 1 0 | [Boolean results from previous step]

Step 4. 1 1 | 2 2 | 3 3 | 4 4 | [Prefix-sum of Boolean array]

Step 5. 1 | 3 | 5 | 7 | [if Prefix-sum Array[i + 1] > Prefix-sum Array[i],
then copy the element at original-input- array[i + 1] to the array of filtered elements]

Step 6. 1 3 5 7 [Filtered elements]

5.5 Find

Find [31, 33] is very similar to Filter, except that the condition 'f' will always be to check whether the element at index 'i' is same as the search element.

Example: Filter with condition if(array[i] == searchElement). In the below example 9 is the search element.

Step 1. 9 2 3 4 9 6 9 8 [Initial Array]

Step 2. 9 2 | 3 4 | 9 6 | 9 8 | [Execute filtering condition on each element of each block]

Step 3. 1 0 | 0 0 | 1 0 | 1 0 | [Boolean results from previous step]

Step 4. 1 1 | 1 1 | 2 2 | 3 3 | [Prefix-sum of Boolean array]

Step 5. 9 | 9 | 9 | 9 | [if Prefix-sum Array[i + 1] > Prefix-sum Array[i],
then copy the element at original-input- array[i + 1] to the array of filtered elements]

Step 6. 9 9 9 [Array of search elements]

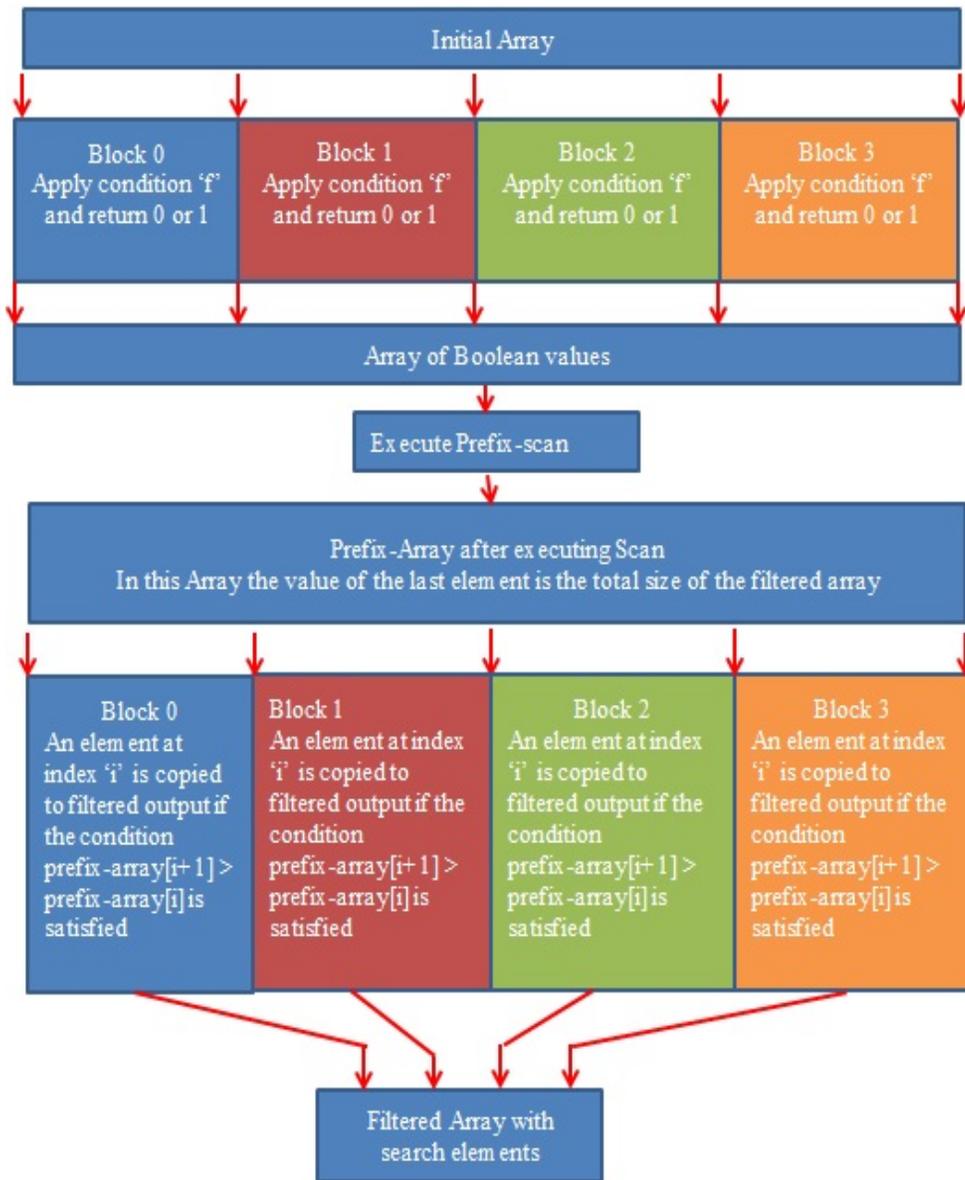


Figure 11 Find

5.6 Sort

Well known parallel sorting mechanism, Bitonic merge-sort [33, 34] has been used for this. Bitonic merge-sort takes $O((\log n)^2)$ time to sort with 'n' parallel processors.

This sorting technique makes use of bitonic sequence, which is nothing but a sequence in which first half of the elements are in ascending order and the other half in descending order. Bitonic sequence undergoes repetitive binary split. Binary split splits a list into two equal sized lists and the corresponding elements between these two lists are compared and the values are exchanged if necessary. This process is continued, till the entire list is sorted.

Bitonic merge-sort fits well into GPU as the comparisons and merging can be done in parallel

Example: 24 20 15 9 4 2 5 8 10 11 12 13 22 30 32 45

After Binary-split

10 11 12 9 4 2 5 8 | 24 20 15 13 22 30 32 45

Here

a) Each element in the first half is smaller than each element in the second half

b) Each half is a bitonic list of length $n/2$.

Keep applying the Binary-split to each half repeatedly till they are sorted

10 11 12 9 | 4 2 5 8 || 24 20 15 13 | 22 30 32 45

4 2 5 8 | 10 11 12 9 || 22 20 15 13 | 24 30 32 45

4 2 | 5 8 || 10 9 | 12 11 || 15 13 | 22 20 || 24 30 | 32 45

Sorted Array

2 4 5 8 9 10 11 12 13 15 20 22 24 30 32 45

Quicksort is not suitable for the GPU architecture [27] and is much slower than bitonic sort.

CHAPTER 6

RELATED WORK

Although there have been many efforts [7, 8, 12, 13 , 14 , 16 , 17] of using heterogeneous environment for object oriented languages, only the relevant ones that offer parallel collections are discussed below.

PyOpenCL [6] and PyCUDA [5] provide scripting for GPUs by having OpenCL and CUDA libraries for Python. Unlike Firepile, hand written kernel code is fed to compiler.SourceModule, which will be later compiled into a kernel binary that can run in the GPU. Since kernels are memoized, they are generated only if required. PyOpenCL and PyCUDA include tight integration of its data structures for host-to-device memory transfer and common operations.

PyCUDA also has GPUArray whose functions get executed on the GPU. It is still not very generic enough to be consumed by the users and at times users themselves have to write the kernels.

JavaCL [2] is a Java wrapper for OpenCL APIs. JNA has been used to facilitate native interactions. JavaCL includes a low level API that resembles the OpenCL API as well as an object oriented API. Firepile uses the JavaCL library as an OpenCL API. Unlike Firepile, kernel code and device management code must be written by hand. JavaCL does not have parallel collections.

ScalaCL [11] is an experimental collections library and Scala compiler plugin that translates Scala code into corresponding kernel code, employing OpenCL and JavaCL wrappers for execution. Starting out as a DSL for parallel expressions on GPUs, ScalaCL has expanded to perform optimizations on some Scala loops and now includes a parallel collections

library that can translate user functions and embed them into OpenCL kernels. Similar to Firepile, ScalaCL aims to translate Scala functions to OpenCL functions. Unlike Firepile, ScalaCL has chosen to take a Scala compiler plugin approach for compile time translation of Scala code to OpenCL code where Firepile performs translation on Java bytecode at runtime. The ScalaCL feature that performs translation is restricted to be used only with map and filter operations of the parallel collection library while Firepile attempts to translate as much of Scala as possible and allows users to write "root level" kernel functions using Scala. Another experimental feature of the ScalaCL compiler plugin are "general optimizations" that attempt to rewrite common `Array[T]`, `List[T]`, and inline range methods such as `foreach`, `map`, `filter`, etc as while loops.

ScalaCL parallel collections has map and filter functions and is still in its early developmental stage.

Scala Parallel Collections have been introduced in the 2.9.0 RC1 release of Scala. These collections are supposed to utilize the multi-core processors. The test environment for this project had just four cores and parallel collections performed worse than its sequential counterpart. Although it might perform better in the system with more cores, having more cores is considerably costlier than having a single GPU.

CHAPTER 7

BENCHMARKS

We have compared the performance of Firepile with Nvidia's OpenCL implementations, ScalaCL & Scala parallel collections and Scala sequential collections for the same data values and range of problem sizes. An average of 10 runs has been considered for the benchmarking.

7.1 System Configuration

Experiments were performed using a system with 3.0 GHz Intel Core 2 Quad Q9650 CPU, 8GB RAM, an NVIDIA GeForce 9800GT graphics card with 512MB of video memory running Windows 7 Professional 64-bit. Firepile was compiled and run with Scala 2.8.1, OpenCL v1.1, Soot 2.4.0, NativeLibs4Java 1.0-beta-5 and Java 1.6.0 24b07 using the HotSpot VM. Scala 2.9.0RC1 has been used for Scala parallel collections.

7.2 Results

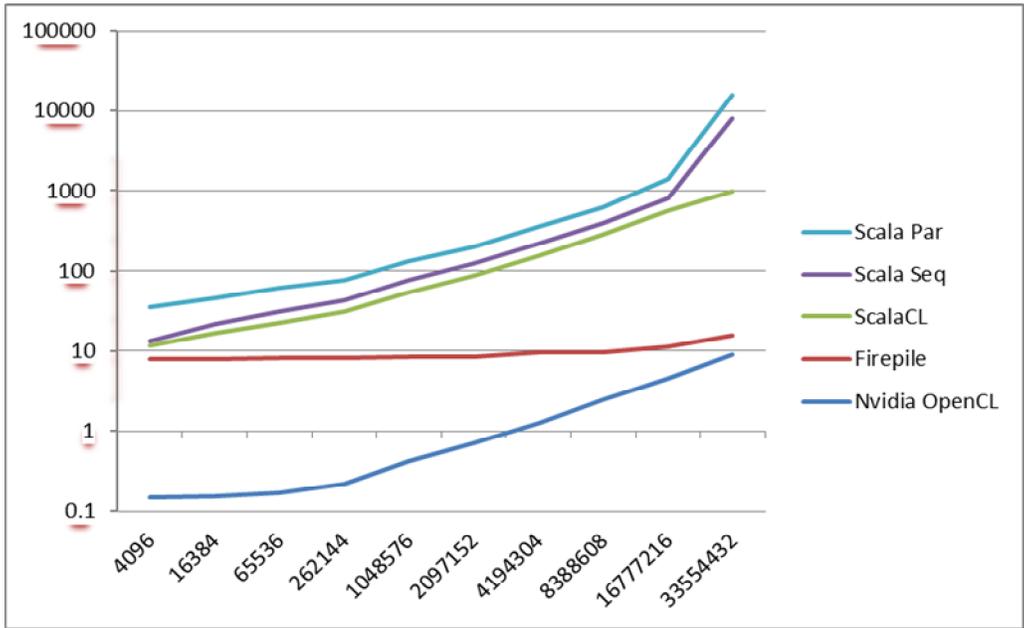
- Nvidia OpenCL – Nvidia's OpenCL implementations.
- Firepile – Parallel collections of Firepile
- ScalaCL – Parallel collections of ScalaCL
- Scala Seq – Scala Sequential collections of Scala 2.8.1
- Scala Par – Scala Parallel collections that are available with Scala 2.9.0RC1

Table 1 Time comparison for the function Reduce (Time in milliseconds)

Size	Nvidia OpenCL	Firepile	ScalaCL	Scala Seq	Scala Par
4096	0.14740	7.81694	3.87961	1.53824	21.4758
16384	0.15460	7.72820	8.95029	5.08035	24.0533
65536	0.16780	7.89446	14.4335	8.09857	29.6063
262144	0.22140	8.02442	22.8166	11.0351	34.6427
1048576	0.42430	7.92170	45.4027	22.9916	55.1370
2097152	0.69300	7.88978	78.0851	35.5271	80.5316
4194304	1.25660	8.51072	146.803	62.2379	146.238
8388608	2.45620	7.24321	279.737	114.105	213.413
16777216	4.62270	6.87295	548.048	252.369	584.200
33554432	9.05570	6.55644	983.514	7101.22	7723.53

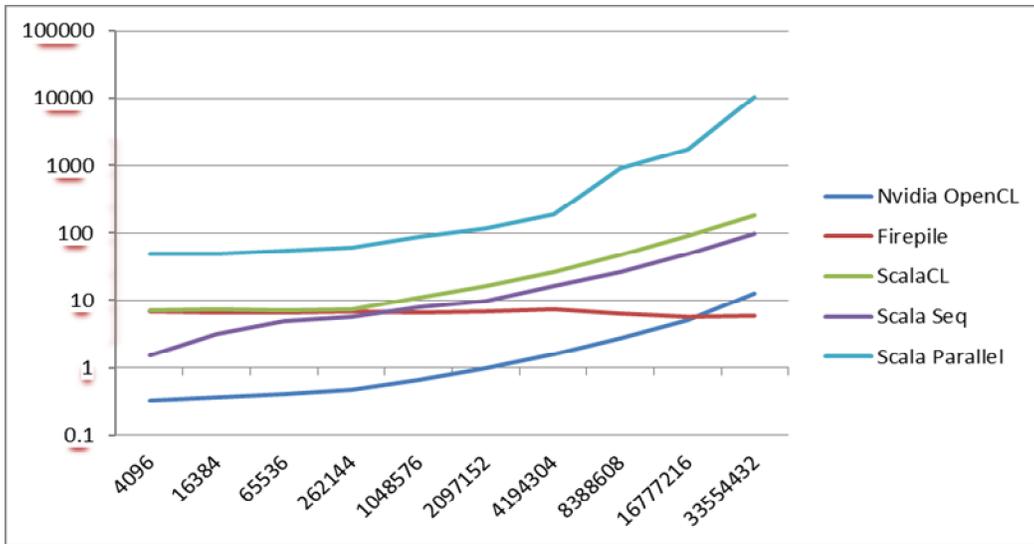
Table 2 Time comparison for the function Map (Time In milliseconds)

Size	Nvidia OpenCL	Firepile	ScalaCL	Scala Seq	Scala Par
4096	0.33340	6.91757	7.10160	1.52680	48.5987
16384	0.36811	6.51659	7.46320	3.19068	49.2613
65536	0.41460	6.68278	7.14440	4.93703	55.0455
262144	0.47270	6.75511	7.34251	5.65183	60.2186
1048576	0.66330	6.69810	10.8539	7.98327	86.5625
2097152	0.98220	6.79423	16.1261	9.99568	116.180
4194304	1.59740	7.29901	26.5463	16.4692	189.620
8388608	2.74630	6.26742	48.0836	26.7348	914.011
16777216	5.14860	5.75922	90.8670	49.4152	1747.97
33554432	12.6957	5.83417	179.011	98.1851	10599.1



X-axis: Number of elements, Y-axis: Time in Milliseconds

Figure 12 Time comparisons for the Reduce function



X-axis: Number of elements, Y-axis: Time in Milliseconds

Figure 13 Time comparisons for the Map function

Table 3 Nvidia OpenCL implementation vs Firepile, Filter (Time in milliseconds)

Size	Nvidia OpenCL	Firepile
4096	4.65300	7.38857
16384	4.66800	9.51642
65536	4.53700	8.32100
262144	4.51200	7.65093
1048576	Failed	8.86231

Table 4 Nvidia OpenCL implementation vs Firepile, Scan (Time in milliseconds)

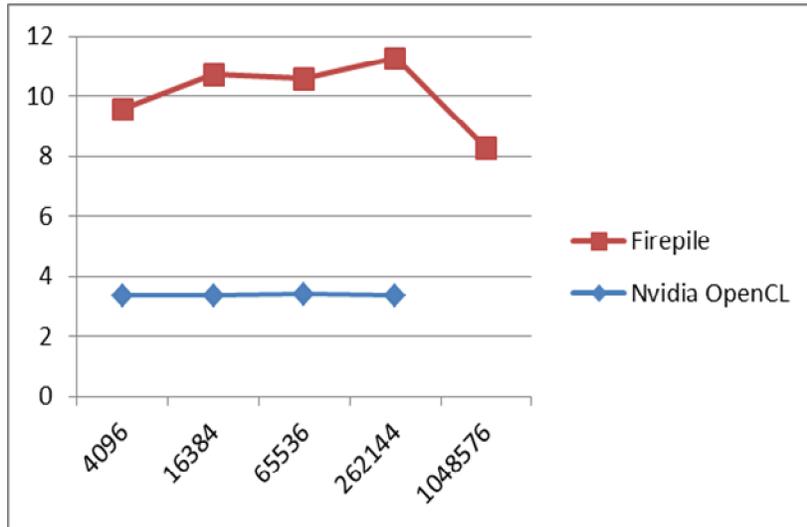
Size	Nvidia OpenCL	Firepile
4096	3.36190	6.19574
16384	3.34560	7.35181
65536	3.39150	7.18279
262144	3.35070	7.90528
1048576	Failed	8.26279

Table 5 Nvidia OpenCL implementation vs Firepile, Sort (Time in milliseconds)

Size	Nvidia OpenCL	Firepile
4096	0.11990	4.95005
16384	0.22320	5.47398
65536	0.35290	5.51283
262144	0.57800	5.54935
1048576	0.76640	5.70373

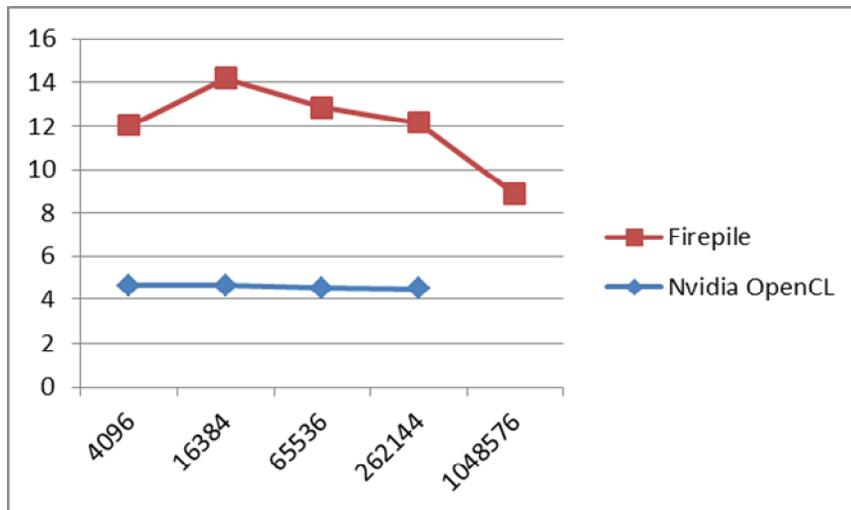
Table 6 Nvidia OpenCL implementation vs Firepile, Lines of code

Functions	Nvidia OpenCL	Firepile
Reduce	525	5
Map	510	5
Scan	560	5
Filter	570	5
Sort	700	5



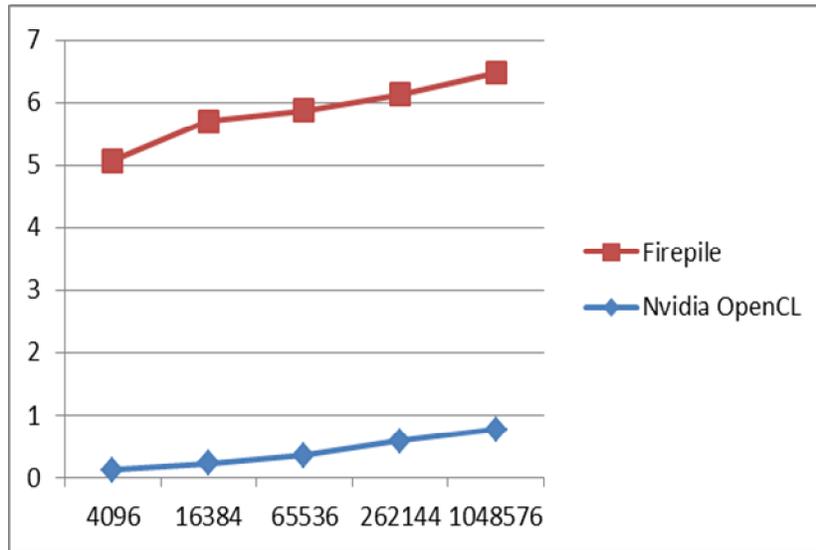
X-axis: Number of elements, Y-axis: Time in Milliseconds

Figure 14 Time comparison for the Scan function



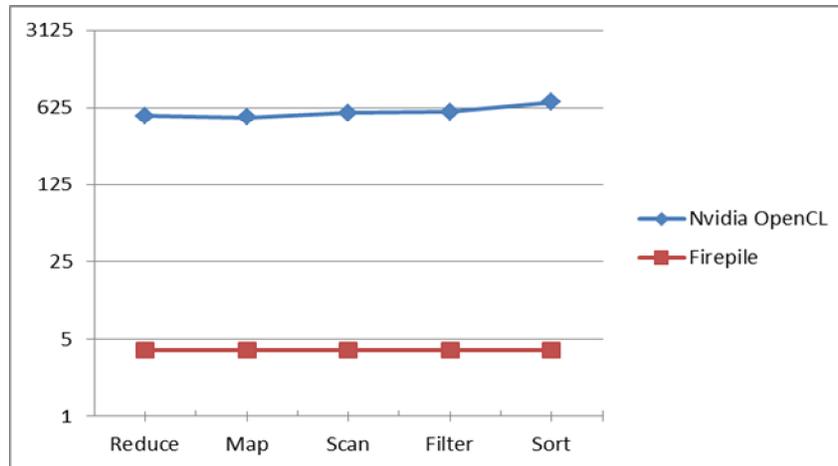
X-axis: Number of elements, Y-axis: Time in Milliseconds

Figure 15 Time comparison for the Filter function



X-axis: Number of elements, Y-axis: Time in Milliseconds

Figure 16 Time comparison for the Sort function



X-axis: Function Names, Y-axis: Lines of Code

Figure 17 Lines of Code comparison for various functions

CHAPTER 8

CONCLUSIONS

As discussed in the introduction, my main contributions are towards implementing Scala reflection framework which is used for type inference, function and class translations, adopting BBArrray for storing data, implementing escape analysis for identifying various layers of a Scala kernel and mapping them to corresponding OpenCL memory spaces, designing and implementating GPU parallel collections for the Firepile library.

Three different problem size ranges have been taken into consideration while discussing the performance of Firepile collections with other parallel collections. They are:

- Smaller Size: < 100 000 elements
- Large Size: > 100 000 and < 1 000 000 elements
- Very Large Size: > 1 000 000 elements

Firepile is about 10X slower than Nvidia's OpenCL implementations for smaller sizes, about 2X slower for large sizes and is as fast as OpenCL for very large data sizes. Firepile and OpenCL tend to converge on the execution time with the increase in problem size. The overhead of having many layers in Firepile becomes negligible when the data is retained in OpenCL buffers and data transfer between CPU and GPU is kept minimal.

Also, redundant synchronization points have been identified in the Nvidia's OpenCL implementations and are optimized in corresponding Firepile versions. Overhead for synchronization increases with problem size. Hence for a simple kernel like Map, Nvidia's OpenCL performance is less than Firepile for very large sizes. Also, Nvidia's implementations for Scan and Filter are not generic enough to be run for large problem sizes and hence they fail for higher problem sizes.

Firepile is as fast as its closest competitor ScalaCL collections for very smaller sizes, but easily outperforms it for large and very large data sizes. Also, ScalaCL collections have very few functions compared to Firepile. As of now there is no facility of retaining OpenCL buffers in ScalaCL and hence it's much slower than Firepile.

Scala sequential collections perform well for smaller sizes in the absence of extra overhead of executing in the GPU, but they become very slow with the increase in data size. Scala Parallel collections are generally much slower in a quad core environment than their sequential counterparts and Firepile. It might perform better in the presence of more cores, but it is outside the scope of this project. Also with more cores the cost will also significantly increase. Since other functions like Scan, Sort and Filter are much more time consuming and complex than Map and Reduce, Firepile's performance has only been compared with that of Nvidia's OpenCL implementations.

For more complex kernels, Firepile is about 2 – 5 times slower than OpenCL. This is understandable as each of these function implementations have multiple kernels and especially for 'sort' the same kernel is executed multiple times. Overhead is added in Firepile with each kernel execution and hence the performance is slower compared to simple functions.

Overall the performance of Firepile is only comparable with that of OpenCL, but considering the brevity of code, memory complexity, and difficulty in parallelizing the code, Firepile parallel collections library seems to be a clear winner.

Although Firepile does well in most cases, care should be taken about the below points:

- ❖ Data should be retained in OpenCL buffers
- ❖ Data transfer between CPU and GPU should be kept minimal
- ❖ Performance is bad for very small sizes. However this can be easily fixed by making Firepile perform computations in the CPU itself for smaller sizes.
- ❖ Currently the Scala to C function translations take considerable amount of time.

This will be fixed in the future in Firepile library by having a compiler plugin that will

make use of ASTs constructed by Scala compiler and memorizing the output for some of the known functions.

CHAPTER 9

FUTURE WORK

Composition of kernels will be accommodated in the future. With this feature the user will be able to concatenate various GPU functionalities in the same statement.

For example: `gpuArray.map(_ * 23).reduce(_+_)` . Here map is executed first and later reduce is applied on the result of the Map.

There is performance bottlenecks involved while using NativeLibs4Java. Nativelibs4java library has to be carefully diagnosed for redundant copying of data and native-call overheads. JNI and BridJ can be tried to see whether there is any considerable improvement in the performance.

Instead of storing the data in the BBArrray, if the user can directly operate on OpenCL buffers with implicit marshaling and unmarshaling, that would further save lot of time for the data transfer.

Some of the current kernel implementations should be diagnosed more carefully to identify the redundant synchronization points and branch statements. Making use of local memory whenever possible and removing conditional statements can save lot of execution time.

Going forward, a compiler plugin that facilitates the access of ASTs of Scala functions at run-time will be developed. This can replace Soot framework, considerably reducing the compilation time.

APPENDIX A
API

GPUArray

```
//Copy back the GPUArray to CPU from GPU
def cpu( ) : Unit

// Returns Array representation of GPUArray
def array( ) : Array[A]

// Returns OpenCL buffer of GPUArray
def clbuffer( ) : CLByteBuffer

//Size of GPUArray
def size( ) : Int

// Mapping from A to B with function 'f'
def map[B](f: A => B) : BArray[A]

// Map where the output is retained in the OpenCL buffer. Each function has this facility.
def map[B](f: A => B)

// blockReduce with function 'f'
def blockReduce(f: (A,A) => A) : BArray[A]

// Reduction with function 'f'
def reduce(f: (A,A) => A): A

// Scan with function 'f'
def scan(f: (A,A) => A): BArray[A]

// Finding the indices for element 'e'
def find(e: A): GpuArray[Int]

// Returns the index of first occurrence of 'e'
def indexOf(e: A ) : Int
```

```

//Filter
def filter(f: A => Boolean): BArray[A]

// Sorts the GPUArray def sort( ): GpuArray[T]

// Returns two arrays separated by the condition f
def span(f: A => Boolean): (GpuArray[A],GpuArray[A])

// Apply function 'f' on each element and return the result
def foreach[U](f: A => U): BArray[U]

                                GPUMap

//Copy back the GPUMap to CPU from GPU
def cpu( ) : Unit

// Returns HashMap representation of GPUMap
def map( ) : HashMap[A, B]

// Returns two OpenCL Byte buffers, one for keys, one for values
def clbuffer( ) : (CLByteBuffer , CLByteBuffer)

// Returns the number of values in the Map
def size( ) : Int

//Apply function 'f' on each key-> value pair
def foreach(f: (K, V) => V)

//Apply function 'f' on each key
def foreachkey(f: K => V)

// Returns all the values
def getValues: BArray[V]

// Returns all the keys
def getKeys( ): BArray[K]

// Merging two maps
def merge(B: GpuHashMap) : GpuMap[K,V]

```

```

// Search for a key
def containsKey(key: K): Boolean

// Search for a value
def containsValue(value: V): Boolean

// Array of sorted keys
def sortedKeys( ): BBArry[K]

// Sort the keys in place
def sortKeys( ) : Unit

// Returns filtered Map with the condition
def filter[K](f: K => V): GpuMap[K,V]

                                GPUMatrix

//Copy back the GPUMatrix to CPU from GPU
def cpu( ) : Unit

// Returns an array representation of Matrix
def array( ) : Array[A]

// OpenCL Byte Buffer
def clbuffer( ) : CLByteBuffer

//Size of the Matrix
def size( ) : Int

// Dot product
def dot (b: GpuMatrix): GPUMatrix[A]

// Cross product
def x (b: GpuMatrix): GPUMatrix[A]

//Transpose
def ' ( ):GPUMatrix[A]

```

```

// Flip the matrix along the main diagonal and replace each element with its complex conjugate
def c'():GPUMatrix[A]

//Inverse
def ^():GPUMatrix[A]

//Apply function f on lower Triangle elements of the Matrix
def tril(f: A => A) : GPUMatrix[A]

//Apply function f on Upper Triangle elements of the Matrix
def triu (f: A => A): GPUMatrix[A]

//Apply function f along the left diagonal
def diagl(f: A => A): GPUMatrix[A]

//Apply function f along right diagonal
def diagr([U](f: A => A) : GPUMatrix[A]

//Flip along the vertical axis
def flipv():GPUMatrix[A]

//Flip along the Horizontal axis
def fliph():GPUMatrix[A]

```

APPENDIX B
GLOSSARY

Glossary

Application: The combination of the program running on the host and OpenCL devices.

BBArray: Byte buffer array provided by Firepile library, optimized for CPU-GPU data transfer.

Blocking and Non-Blocking Enqueue API calls: A non-blocking enqueue API call places a command on a command-queue and returns immediately to the host. The blocking-mode enqueue API calls do not return to the host until the command has completed.

Barrier: There are two types of barriers – a command-queue barrier and a work-group barrier. The OpenCL API provides a function to enqueue a command-queue barrier command. This barrier command ensures that all previously enqueued commands to a command-queue have finished execution before any following commands enqueued in the command-queue can begin execution. The OpenCL C programming language provides a built-in work-group barrier function. This barrier built-in function can be used by a kernel executing on a device to perform synchronization between work-items in a work-group executing the kernel. All the work-items of a work-group must execute the barrier construct before any are allowed to continue execution beyond the barrier.

Buffer Object: A memory object that stores a linear collection of bytes. Buffer objects are accessible using a pointer in a kernel executing on a device. Buffer objects can be manipulated by the host using OpenCL API calls. A buffer object encapsulates the following information: Size in bytes, Properties that describe usage information and which region to allocate from. Buffer data.

Command: The OpenCL operations that are submitted to a command-queue for execution. For example, OpenCL commands issue kernels for execution on a compute device, manipulate memory objects, etc.

Command-queue: An object that holds commands that will be executed on a specific device. The command-queue is created on a specific device in a context. Commands to a commandqueue are queued in-order but may be executed in-order or out-of-order. Refer to In-order Execution and Out-of-order Execution.

Command-queue Barrier. See Barrier.

Compute Unit: An OpenCL device has one or more compute units. A work-group executes on a single compute unit. A compute unit is composed of one or more processing elements. A compute unit may also include dedicated texture filtering units that can be accessed by its processing elements.

Concurrency: A property of a system in which a set of tasks in a system can remain active and make progress at the same time. To utilize concurrent execution when running a program, a programmer must identify the concurrency in their problem, expose it within the source code, and then exploit it using a notation that supports concurrency.

Constant Memory: A region of global memory that remains constant during the execution of a kernel. The host allocates and initializes memory objects placed into constant memory.

Context: The environment within which the kernels execute and the domain in which synchronization and memory management is defined. The context includes a set of devices, the memory accessible to those devices, the corresponding memory properties and one or more command-queues used to schedule execution of a kernel(s) or operations on memory objects.

Device: A device is a collection of compute units. A command-queue is used to queue commands to a device. Examples of commands include executing kernels, or reading and writing memory objects. OpenCL devices typically correspond to a GPU, a multi-core CPU, and other processors such as DSPs and the Cell/B.E. processor.

Event Object: An event object encapsulates the status of an operation such as a command. It can be used to synchronize operations in a context.

Event Wait List: An event wait list is a list of event objects that can be used to control when a particular command begins execution.

Framework: A software system that contains the set of components to support software development and execution. A framework typically includes libraries, APIs, runtime systems, compilers, etc.

Global ID: A global ID is used to uniquely identify a work-item and is derived from the number of global work-items specified when executing a kernel. The global ID is a N-dimensional value that starts at (0, 0, ... 0). See also Local ID.

Global Memory: A memory region accessible to all work-items executing in a context. It is accessible to the host using commands such as read, write and map.

Handle: An opaque type that references an object allocated by OpenCL. Any operation on an object occurs by reference to that object's handle.

Host: The host interacts with the context using the OpenCL API.

Host pointer: A pointer to memory that is in the virtual address space on the host.

Illegal: Behavior of a system that is explicitly not allowed and will be reported as an error when encountered by OpenCL.

Image Object: A memory object that stores a two- or three- dimensional structured array. Image data can only be accessed with read and write functions. The read functions use a sampler. The image object encapsulates the following information: Dimensions of the image. Description of each element in the image. Properties that describe usage information and which region to allocate from. Image data. The elements of an image are selected from a list of predefined image formats.

In-order Execution: A model of execution in OpenCL where the commands in a commandqueue are executed in order of submission with each command running to completion before the next one begins. See Out-of-order Execution.

Kernel: A kernel is a function declared in a program and executed on an OpenCL device. A kernel is identified by the `__kernel` qualifier applied to any function defined in a program.

Kernel Object: A kernel object encapsulates a specific `__kernel` function declared in a program and the argument values to be used when executing this `__kernel` function.

Local ID: A local ID specifies a unique work-item ID within a given work-group that is executing a kernel. The local ID is a N-dimensional value that starts at (0, 0, ... 0). See also Global ID.

Local Memory: A memory region associated with a work-group and accessible only by workitems in that work-group. This is called shared memory in CUDA.

Memory Objects: A memory object is a handle to a reference counted region of global memory. Also see Buffer Objects and Image Objects.

Memory Regions (or Pools): A distinct address space in OpenCL. Memory regions may overlap in physical memory though OpenCL will treat them as logically distinct. The memory regions are denoted as private, local, constant and global.

Object: Objects are abstract representation of the resources that can be manipulated by the OpenCL API. Examples include program objects, kernel objects, and memory objects.

OpenCL Buffer: See Buffer Object

Out-of-Order Execution: A model of execution in which commands placed in the work queue may begin and complete execution in any order consistent with constraints imposed by event wait lists and command-queue barrier. See In-order Execution.

Platform: The host plus a collection of devices managed by the OpenCL framework that allow an application to share resources and execute kernels on devices in the platform.

Private Memory: A region of memory private to a work-item. Variables defined in one workitem's private memory are not visible to another work-item. This is called local memory in CUDA.

Processing Element: A virtual scalar processor. A work-item may execute on one or more processing elements.

Program: An OpenCL program consists of a set of kernels. Programs may also contain auxiliary functions called by the __kernel functions and constant data.

Program Object: A program object encapsulates the following information:

A reference to an associated context. A program source or binary. The latest successfully built program executable, the list of devices for which the program executable is built, the build options used and a build log. The number of kernel objects currently attached.

Resource: A class of objects defined by OpenCL. An instance of a resource is an object. The most common resources are the context, command-queue, program objects, kernel objects, and memory objects. Computational resources are hardware elements that participate in the action of advancing a program counter. Examples include the host, devices, compute units and processing elements.

Task Parallel Programming Model: A programming model in which computations are expressed in terms of multiple concurrent tasks where a task is a kernel executing in a single work-group of size one. The concurrent tasks can be running different kernels.

Thread-safe: An OpenCL API call is considered to be thread-safe if the internal state as managed by OpenCL remains consistent when called simultaneously by multiple host threads. OpenCL API calls that are thread-safe allow an application to call these functions in multiple host threads without having to implement mutual exclusion across these host threads.

Work-group: A collection of related work-items that execute on a single compute unit. The work-items in the group execute the same kernel and share local memory and work-group barriers.

Work-group Barrier: See Barrier.

Work-item: One of a collection of parallel executions of a kernel invoked on a device by a command. A work-item is executed by one or more processing elements as part of a work-

group executing on a compute unit. A work-item is distinguished from other executions within the collection by its global ID and local ID.

APPENDIX C

NVIDIA's OpenCL IMPLEMENTATION OF REDUCE KERNEL

```

/*
 * Copyright 1993-2010 NVIDIA Corporation. All rights reserved.
 *
 * NVIDIA Corporation and its licensors retain all intellectual property and
 * proprietary rights in and to this software and related documentation.
 * Any use, reproduction, disclosure, or distribution of this software
 * and related documentation without an express license agreement from
 * NVIDIA Corporation is strictly prohibited.
 *
 * Please refer to the applicable NVIDIA end user license agreement (EULA)
 * associated with this source code for terms and conditions that govern
 * your use of this NVIDIA software.
 */

/*
Parallel reduction

This sample shows how to perform a reduction operation on an array of values
to produce a single value.

Reductions are a very common computation in parallel algorithms. Any time
an array of values needs to be reduced to a single value using a binary
associative operator, a reduction can be used. Example applications include
statistics computations such as mean and standard deviation, and image
processing applications such as finding the total luminance of an
image.

This code performs sum reductions, but any associative operator such as
min() or max() could also be used.

It assumes the input size is a power of 2.

COMMAND LINE ARGUMENTS

"--shmoo":      Test performance for 1 to 32M elements with each of the 7 different kernels
"--n=<N>":      Specify the number of elements to reduce (default 1048576)
"--threads=<N>": Specify the number of threads per block (default 128)
"--kernel=<N>": Specify which kernel to run (0-6, default 6)
"--maxblocks=<N>": Specify the maximum number of thread blocks to launch (kernel 6 only,
default 64)
"--cpufinal":   Read back the per-block results and do final sum of block sums on CPU
(default false)
"--cputhresh=<N>": The threshold of number of blocks sums below which to perform a CPU
final reduction (default 1)
*/

```

```

// Common system and utility includes
#include <oclUtils.h>

// additional includes
#include <sstream>
#include <oclReduction.h>

// Forward declarations and sample-specific defines
// *****
enum ReduceType
{
    REDUCE_INT,
    REDUCE_FLOAT,
    REDUCE_DOUBLE
};

template <class T>
void runTest( int argc, const char** argv, ReduceType datatype);

#define MAX_BLOCK_DIM_SIZE 65535

extern "C"
bool isPow2(unsigned int x)
{
    return ((x&(x-1))==0);
}

cl_kernel getReductionKernel(ReduceType datatype, int whichKernel, int blockSize, int
isPowOf2);

// Main function
// *****
int main( int argc, const char** argv)
{
    // start logs
    shrSetLogFileName ("oclReduction.txt");
    shrLog("%s Starting...\n\n", argv[0]);

    char *typeChoice;
    shrGetCmdLineArgumentstr(argc, argv, "type", &typeChoice);

    // determine type of array from command line args
    if (0 == typeChoice)
    {
        typeChoice = (char*)malloc(7 * sizeof(char));
        #ifdef WIN32
            strcpy_s(typeChoice, 7 * sizeof(char) + 1, "int");
        #else
            strcpy(typeChoice, "int");
        #endif
    }
    ReduceType datatype = REDUCE_INT;

```

```

#ifdef WIN32
    if (!_strcmpi(typeChoice, "float"))
        datatype = REDUCE_FLOAT;
    else if (!_strcmpi(typeChoice, "double"))
        datatype = REDUCE_DOUBLE;
    else
        datatype = REDUCE_INT;
#else
    if (!strcmp(typeChoice, "float"))
        datatype = REDUCE_FLOAT;
    else if (!strcmp(typeChoice, "double"))
        datatype = REDUCE_DOUBLE;
    else
        datatype = REDUCE_INT;
#endif

shrLog("Reducing array of type %s.\n", typeChoice);

//Get the NVIDIA platform
ciErrNum = oclGetPlatformID(&cpPlatform);
oclCheckError(ciErrNum, CL_SUCCESS);

//Get the devices
ciErrNum = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 0, NULL,
&uiNumDevices);
oclCheckError(ciErrNum, CL_SUCCESS);
cl_device_id *cdDevices = (cl_device_id *)malloc(uiNumDevices * sizeof(cl_device_id) );
ciErrNum = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, uiNumDevices,
cdDevices, NULL);
oclCheckError(ciErrNum, CL_SUCCESS);

//Create the context
cxGPUContext = clCreateContext(0, uiNumDevices, cdDevices, NULL, NULL, &ciErrNum);
oclCheckError(ciErrNum, CL_SUCCESS);

// get and log the device info
if( shrCheckCmdLineFlag(argc, (const char**)argv, "device") ) {
    int device_nr = 0;
    shrGetCmdLineArgumenti(argc, (const char**)argv, "device", &device_nr);
    device = oclGetDev(cxGPUContext, device_nr);
} else {
    device = oclGetMaxFlopsDev(cxGPUContext);
}
oclPrintDevName(LOGBOTH, device);
shrLog("\n");

// create a command-queue
cqCommandQueue = clCreateCommandQueue(cxGPUContext, device, 0, &ciErrNum);
oclCheckError(ciErrNum, CL_SUCCESS);

source_path = shrFindFilePath("oclReduction_kernel.cl", argv[0]);

```

```

switch (datatype)
{
default:
case REDUCE_INT:
    runTest<int>( argc, argv, datatype);
    break;
case REDUCE_FLOAT:
    runTest<float>( argc, argv, datatype);
    break;
}

// finish
shrEXIT(argc, argv);
}

////////////////////////////////////////////////////////////////
// Compute sum reduction on CPU
// We use Kahan summation for an accurate sum of large arrays.
// http://en.wikipedia.org/wiki/Kahan_summation_algorithm
//
// @param data    pointer to input data
// @param size    number of input data elements
////////////////////////////////////////////////////////////////
template<class T>
T reduceCPU(T *data, int size)
{
    T sum = data[0];
    T c = (T)0.0;
    for (int i = 1; i < size; i++)
    {
        T y = data[i] - c;
        T t = sum + y;
        c = (t - sum) - y;
        sum = t;
    }
    return sum;
}

unsigned int nextPow2( unsigned int x ) {
    --x;
    x |= x >> 1;
    x |= x >> 2;
    x |= x >> 4;
    x |= x >> 8;
    x |= x >> 16;
    return ++x;
}

////////////////////////////////////////////////////////////////
// Compute the number of threads and blocks to use for the given reduction kernel
// For the kernels >= 3, we set threads / block to the minimum of maxThreads and

```

```

// n/2. For kernels < 3, we set to the minimum of maxThreads and n. For kernel
// 6, we observe the maximum specified number of blocks, because each thread in
// that kernel can process a variable number of elements.
////////////////////////////////////
void getNumBlocksAndThreads(int whichKernel, int n, int maxBlocks, int maxThreads, int
&blocks, int &threads)
{
    if (whichKernel < 3)
    {
        threads = (n < maxThreads) ? nextPow2(n) : maxThreads;
        blocks = (n + threads - 1) / threads;
    }
    else
    {
        threads = (n < maxThreads*2) ? nextPow2((n + 1)/ 2) : maxThreads;
        blocks = (n + (threads * 2 - 1)) / (threads * 2);
    }

    if (whichKernel == 6)
        blocks = MIN(maxBlocks, blocks);
}

////////////////////////////////////
// This function performs a reduction of the input data multiple times and
// measures the average reduction time.
////////////////////////////////////
template <class T>
T profileReduce(ReduceType datatype,
                cl_int n,
                int numThreads,
                int numBlocks,
                int maxThreads,
                int maxBlocks,
                int whichKernel,
                int testIterations,
                bool cpuFinalReduction,
                int cpuFinalThreshold,
                double* dTotalTime,
                T* h_odata,
                cl_mem d_idata,
                cl_mem d_odata)
{

    T gpu_result = 0;
    bool needReadBack = true;
    cl_kernel finalReductionKernel[10];
    int finalReductionIterations=0;

    //shrLog("Profile Kernel %d\n", whichKernel);

```

```

    cl_kernel reductionKernel = getReductionKernel(datatype, whichKernel, numThreads,
isPow2(n) );
    clSetKernelArg(reductionKernel, 0, sizeof(cl_mem), (void *) &d_idata);
    clSetKernelArg(reductionKernel, 1, sizeof(cl_mem), (void *) &d_odata);
    clSetKernelArg(reductionKernel, 2, sizeof(cl_int), &n);
    clSetKernelArg(reductionKernel, 3, sizeof(T) * numThreads, NULL);

if( !cpuFinalReduction ) {
    int s=numBlocks;
    int threads = 0, blocks = 0;
    int kernel = (whichKernel == 6) ? 5 : whichKernel;

    while(s > cpuFinalThreshold)
    {
        getNumBlocksAndThreads(kernel, s, maxBlocks, maxThreads, blocks, threads);

        finalReductionKernel[finalReductionIterations] = getReductionKernel(datatype, kernel,
threads, isPow2(s) );
        clSetKernelArg(finalReductionKernel[finalReductionIterations], 0, sizeof(cl_mem), (void
*) &d_odata);
        clSetKernelArg(finalReductionKernel[finalReductionIterations], 1, sizeof(cl_mem), (void
*) &d_odata);
        clSetKernelArg(finalReductionKernel[finalReductionIterations], 2, sizeof(cl_int), &n);
        clSetKernelArg(finalReductionKernel[finalReductionIterations], 3, sizeof(T) *
numThreads, NULL);

        if (kernel < 3)
            s = (s + threads - 1) / threads;
        else
            s = (s + (threads*2-1)) / (threads*2);

        finalReductionIterations++;
    }
}

size_t globalWorkSize[1];
size_t localWorkSize[1];

for (int i = 0; i < testIterations; ++i)
{
    gpu_result = 0;

    clFinish(cqCommandQueue);
    if(i>0) shrDeltaT(1);

    // execute the kernel
    globalWorkSize[0] = numBlocks * numThreads;
    localWorkSize[0] = numThreads;

```

```

        ciErrNum = clEnqueueNDRangeKernel(cqCommandQueue,reductionKernel, 1, 0,
globalWorkSize, localWorkSize,
        0, NULL, NULL);

// check if kernel execution generated an error
oclCheckError(ciErrNum, CL_SUCCESS);

if (cpuFinalReduction)
{
    // sum partial sums from each block on CPU
    // copy result from device to host
    clEnqueueReadBuffer(cqCommandQueue, d_odata, CL_FALSE, 0,
numBlocks*sizeof(T),
        h_odata, 0, NULL, NULL);

    for(int i=0; i<numBlocks; i++)
    {
        gpu_result += h_odata[i];
    }

    needReadBack = false;
}
else
{
    // sum partial block sums on GPU
    int s=numBlocks;
    int kernel = (whichKernel == 6) ? 5 : whichKernel;
    int it = 0;

    while(s > cpuFinalThreshold)
    {
        int threads = 0, blocks = 0;
        getNumBlocksAndThreads(kernel, s, maxBlocks, maxThreads, blocks, threads);

        globalWorkSize[0] = threads * blocks;
        localWorkSize[0] = threads;

        ciErrNum = clEnqueueNDRangeKernel(cqCommandQueue,finalReductionKernel[it],
1, 0, globalWorkSize, localWorkSize, 0, NULL, NULL);
        oclCheckError(ciErrNum, CL_SUCCESS);

        if (kernel < 3)
            s = (s + threads - 1) / threads;
        else
            s = (s + (threads*2-1)) / (threads*2);

        it++;
    }

    if (s > 1)
    {
        // copy result from device to host

```

```

        clEnqueueReadBuffer(cqCommandQueue, d_odata, CL_FALSE, 0, s*sizeof(T),
                           h_odata, 0, NULL, NULL);

        for(int i=0; i < s; i++)
        {
            gpu_result += h_odata[i];
        }

        needReadBack = false;
    }
}

clFinish(cqCommandQueue);
if(i>0) *dTTotalTime += shrDeltaT(1);
}

if (needReadBack)
{
    // copy final sum from device to host
    clEnqueueReadBuffer(cqCommandQueue, d_odata, CL_FALSE, 0, sizeof(T),
                        &gpu_result, 0, NULL, NULL);
}

// Release the kernels
clReleaseKernel(reductionKernel);
if( !cpuFinalReduction ) {
    for(int it=0; it<finalReductionIterations; ++it) {
        clReleaseKernel(finalReductionKernel[it]);
    }
}

return gpu_result;
}

```

```

////////////////////////////////////
// This function calls profileReduce multiple times for a range of array sizes
// and prints a report in CSV (comma-separated value) format that can be used for
// generating a "shmoo" plot showing the performance for each kernel variation
// over a wide range of input sizes.
////////////////////////////////////
template <class T>
void shmoo(int minN, int maxN, int maxThreads, int maxBlocks, ReduceType datatype)
{
    // create random input data on CPU
    unsigned int bytes = maxN * sizeof(T);

    T *h_idata = (T*) malloc(bytes);

    for(int i = 0; i < maxN; i++) {
        // Keep the numbers small so we don't get truncation error in the sum
        if (datatype == REDUCE_INT)

```

```

        h_idata[i] = (T)(rand() & 0xFF);
    else
        h_idata[i] = (rand() & 0xFF) / (T)RAND_MAX;
    }

    int maxNumBlocks = MIN( maxN / maxThreads, MAX_BLOCK_DIM_SIZE);

    // allocate mem for the result on host side
    T* h_odata = (T*) malloc(maxNumBlocks*sizeof(T));

    // allocate device memory and data
    cl_mem d_idata = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, bytes, h_idata, NULL);
    cl_mem d_odata = clCreateBuffer(cxGPUContext, CL_MEM_READ_WRITE,
    maxNumBlocks*sizeof(T), NULL, NULL);

    int testIterations = 100;
    double dTotalTime = 0.0;

    // print headers
    shrLog("Time in seconds for various numbers of elements for each kernel\n");
    shrLog("\n\n");
    shrLog("Kernel");
    for (int i = minN; i <= maxN; i *= 2)
    {
        shrLog("  %d", i);
    }

    for (int kernel = 0; kernel < 7; kernel++)
    {
        shrLog("\n");
        shrLog("%d", kernel);
        for (int i = minN; i <= maxN; i *= 2)
        {
            int numBlocks = 0;
            int numThreads = 0;
            getNumBlocksAndThreads(kernel, i, maxBlocks, maxThreads, numBlocks,
numThreads);

            double reduceTime;
            if( numBlocks <= MAX_BLOCK_DIM_SIZE ) {
                profileReduce(datatype, i, numThreads, numBlocks, maxThreads, maxBlocks, kernel,
                testIterations, false, 1, &dTotalTime, h_odata, d_idata, d_odata);
                reduceTime = dTotalTime/(double)testIterations;
            } else {
                reduceTime = -1.0;
            }
            shrLog("  %.4f m", reduceTime);
        }
    }

    // cleanup

```

```

    free(h_idata);
    free(h_odata);
    clReleaseMemObject(d_idata);
    clReleaseMemObject(d_odata);
}

/////////////////////////////////////////////////////////////////
// The main function which runs the reduction test.
/////////////////////////////////////////////////////////////////
template <class T>
void
runTest( int argc, const char** argv, ReduceType datatype)
{
    int size = 1<<24; // number of elements to reduce
    int maxThreads;

    cl_kernel reductionKernel = getReductionKernel(datatype, 0, 64, 1);
    clReleaseKernel(reductionKernel);

    if (smallBlock)
        maxThreads = 64; // number of threads per block
    else
        maxThreads = 128;

    int whichKernel = 6;
    int maxBlocks = 64;
    bool cpuFinalReduction = false;
    int cpuFinalThreshold = 1;

    shrGetCmdLineArgumenti( argc, (const char**) argv, "n", &size);
    shrGetCmdLineArgumenti( argc, (const char**) argv, "threads", &maxThreads);
    shrGetCmdLineArgumenti( argc, (const char**) argv, "kernel", &whichKernel);
    shrGetCmdLineArgumenti( argc, (const char**) argv, "maxblocks", &maxBlocks);

    shrLog(" %d elements\n", size);
    shrLog(" %d threads (max)\n", maxThreads);

    cpuFinalReduction = (shrCheckCmdLineFlag( argc, (const char**) argv, "cpufinal") ==
shrTRUE);
    shrGetCmdLineArgumenti( argc, (const char**) argv, "cputhresh", &cpuFinalThreshold);

    bool runShmoo = (shrCheckCmdLineFlag(argc, (const char**) argv, "shmoo") == shrTRUE);

#ifdef GPU_PROFILING
    if (runShmoo)
    {
        shmoo<T>(1, 33554432, maxThreads, maxBlocks, datatype);
    }
    else
#endif
{

```

```

// create random input data on CPU
unsigned int bytes = size * sizeof(T);

T *h_idata = (T *) malloc(bytes);

for(int i=0; i<size; i++)
{
    // Keep the numbers small so we don't get truncation error in the sum
    if (datatype == REDUCE_INT)
        h_idata[i] = (T)(rand() & 0xFF);
    else
        h_idata[i] = (rand() & 0xFF) / (T)RAND_MAX;
}

int numBlocks = 0;
int numThreads = 0;
getNumBlocksAndThreads(whichKernel, size, maxBlocks, maxThreads, numBlocks,
numThreads);
if (numBlocks == 1) cpuFinalThreshold = 1;

// allocate mem for the result on host side
T* h_odata = (T*) malloc(numBlocks*sizeof(T));

shrLog(" %d blocks\n\n", numBlocks);

// allocate device memory and data
cl_mem d_idata = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, bytes, h_idata, NULL);
cl_mem d_odata = clCreateBuffer(cxGPUContext, CL_MEM_READ_WRITE,
numBlocks*sizeof(T), NULL, NULL);

// warm-up
//reduce_sm10<T>(size, numThreads, numBlocks, whichKernel, d_idata, d_odata);

int testIterations = 100;
double dTotalTime = 0.0;
T gpu_result = 0;
gpu_result = profileReduce<T>(datatype, size, numThreads, numBlocks, maxThreads,
maxBlocks, whichKernel, testIterations, cpuFinalReduction,
cpuFinalThreshold, &dTotalTime,
h_odata, d_idata, d_odata);

#ifdef GPU_PROFILING
double reduceTime = dTotalTime/(double)testIterations;
shrLogEx(LOGBOTH | MASTER, 0, "oclReduction, Throughput = %.4f GB/s, Time = %.5f
s, Size = %u, NumDevsUsed = %d, Workgroup = %u\n",
1.0e-9 * (size * sizeof(int))/reduceTime, reduceTime, size, 1, numThreads);
#endif

// compute reference solution
shrLog("\nComparing against Host/C++ computation...\n");
T cpu_result = reduceCPU<T>(h_idata, size);

```

```

if (datatype == REDUCE_INT)
{
    shrLog(" GPU result = %d\n", gpu_result);
    shrLog(" CPU result = %d\n\n", cpu_result);

    shrLog("%s\n\n", (gpu_result == cpu_result) ? "PASSED" : "FAILED");
}
else
{
    shrLog(" GPU result = %.9f\n", gpu_result);
    shrLog(" CPU result = %.9f\n\n", cpu_result);

    double threshold = (datatype == REDUCE_FLOAT) ? 1e-8 * size : 1e-12;
    double diff = abs((double)gpu_result - (double)cpu_result);
    shrLog("%s\n\n", (diff < threshold) ? "PASSED" : "FAILED");
}

// cleanup
free(h_idata);
free(h_odata);
clReleaseMemObject(d_idata);
clReleaseMemObject(d_odata);
}
}

// Helper function to create and build program and kernel
// *****
cl_kernel getReductionKernel(ReduceType datatype, int whichKernel, int blockSize, int
isPowOf2)
{
    // compile cl program
    size_t program_length;
    char *source;

    std::ostringstream preamble;

    // create the program
    // with type specification depending on datatype argument
    switch (datatype)
    {
    default:
    case REDUCE_INT:
        preamble << "#define T int" << std::endl;
        break;
    case REDUCE_FLOAT:
        preamble << "#define T float" << std::endl;
        break;
    }

    // set blockSize at compile time
    preamble << "#define blockSize " << blockSize << std::endl;

```

```

// set isPow2 at compile time
preamble << "#define nlsPow2 " << isPowOf2 << std::endl;

// Load the source code and prepend the preamble
source = oclLoadProgSource(source_path, preamble.str().c_str(), &program_length);
oclCheckError(source != NULL, shrTRUE);

    cl_program cpProgram = clCreateProgramWithSource(cxGPUContext, 1,(const char **)
&source, &program_length, &ciErrNum);
    oclCheckError(ciErrNum, CL_SUCCESS);
    free(source);

// build the program
ciErrNum = clBuildProgram(cpProgram, 0, NULL, "-cl-fast-relaxed-math", NULL, NULL);
if (ciErrNum != CL_SUCCESS)
{
    // write out standard error, Build Log and PTX, then cleanup and exit
    shrLogEx(LOGBOTH | ERRORMSG, ciErrNum, STDERR);
    oclLogBuildInfo(cpProgram, oclGetFirstDev(cxGPUContext));
    oclLogPtx(cpProgram, oclGetFirstDev(cxGPUContext), "oclReduction.ptx");
    oclCheckError(ciErrNum, CL_SUCCESS);
}

// create Kernel
std::ostringstream kernelName;
kernelName << "reduce" << whichKernel;
cl_kernel ckKernel = clCreateKernel(cpProgram, kernelName.str().c_str(), &ciErrNum);
oclCheckError(ciErrNum, CL_SUCCESS);

size_t wgSize;
ciErrNum = clGetKernelWorkGroupInfo(ckKernel, device,
CL_KERNEL_WORK_GROUP_SIZE, sizeof(size_t), &wgSize, NULL);
if (wgSize == 64)
    smallBlock = true;
else smallBlock = false;

// NOTE: the program will get deleted when the kernel is also released
clReleaseProgram(cpProgram);

return ckKernel;
}

// Reduce Kernel code

/*
* Copyright 1993-2009 NVIDIA Corporation. All rights reserved.
*
* NVIDIA Corporation and its licensors retain all intellectual property and
* proprietary rights in and to this software and related documentation.
* Any use, reproduction, disclosure, or distribution of this software
* and related documentation without an express license agreement from
* NVIDIA Corporation is strictly prohibited.
*
*/

```

```

* Please refer to the applicable NVIDIA end user license agreement (EULA)
* associated with this source code for terms and conditions that govern
* your use of this NVIDIA software.
*
*/

/*
Parallel reduction kernels
*/

// The following defines are set during runtime compilation, see reduction.cpp
// #define T float
// #define blockSize 128
// #define nlsPow2 1

#ifndef _REDUCE_KERNEL_H_
#define _REDUCE_KERNEL_H_
/*
This version uses n/2 threads --
it performs the first level of reduction when reading from global memory
*/
__kernel void reduce3(__global T *g_idata, __global T *g_odata, unsigned int n, __local T*
sdata)
{
// perform first level of reduction,
// reading from global memory, writing to shared memory
unsigned int tid = get_local_id(0);
unsigned int i = get_group_id(0)*(get_local_size(0)*2) + get_local_id(0);

sdata[tid] = (i < n) ? g_idata[i] : 0;
if (i + get_local_size(0) < n)
sdata[tid] += g_idata[i+get_local_size(0)];

barrier(CLK_LOCAL_MEM_FENCE);

// do reduction in shared mem
for(unsigned int s=get_local_size(0)/2; s>0; s>>=1)
{
if (tid < s)
{
sdata[tid] += sdata[tid + s];
}
barrier(CLK_LOCAL_MEM_FENCE);
}

// write result for this block to global mem
if (tid == 0) g_odata[get_group_id(0)] = sdata[0];
}

```

REFERENCES

- [1] First GPU, <http://www.nvidia.com/page/geforce256.html>
- [2] Javacl. <http://code.google.com/p/javacl/>.
- [3] Jcuda. <http://www.jcuda.org/>.
- [4] Jocl. <http://www.jocl.org/>.
- [5] Pycuda. <http://mathematician.de/software/pycuda>.
- [6] Pyopencl. <http://mathematician.de/software/pyopencl>.
- [7] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: a java-compatible and synthesizable language for heterogeneous architectures. In Proceedings of the 25th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2010), pages 89–108, 2010.
- [8] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. In ACM SIGGRAPH 2004 Papers (SIGGRAPH '04), pages 777–786, 2004.
- [9] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language virtualization for heterogeneous parallel computing. In Onward! '10: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, October 2010.
- [10] NativeLibs4Java, <http://code.google.com/p/nativelibs4java/>
- [11] Olivier Chafik. ScalaCL: Faster Scala: optimizing compiler plugin + GPU-based collections (OpenCL). <http://code.google.com/p/scalacl/>, 2011.
- [12] Philippe Charles, Christian Grothoff, Christopher Donawa, Kemal Ebcioglu, Allan Kielstra, Christoph von Praun, Vijay Saraswat, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In Proc. OOPSLA '05, October 2005.

- [13] Shan Shan Huang, Amir Hormati, David F. Bacon, and Rodric Rabbah. Liquid metal: Object-oriented programming across the hardware/software boundary.
- [14] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization, 2009.
- [15] Soot Framework, <http://www.sable.mcgill.ca/soot/>
- [16] Calle Lejdfors and Lennart Ohlsson. Implementing an embedded gpu language by combining translation and generation. In Proceedings of the 2006 ACM symposium on Applied computing (SAC '06), pages 1610–1614, 2006.
- [17] Geoffrey Mainland and Greg Morrisett. Nikola: embedding compiled GPU functions in Haskell. In Proceedings of the third ACM symposium on Haskell (Haskell '10), pages 67–78, 2010.
- [18] Scala, <http://www.scala-lang.org/>
- [19] Aho, Alfred V.; Sethi, Ravi; & Ullman, Jeffrey D. Compilers: Principles, Techniques, and Tools. 2/e Addison Wesley. ISBN 0-201-10088-6 , 1986.
- [20] Types, Benjamin C. Pierce, Advanced Topics in Types And Programming Languages. The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts 02142 <http://mitpress.mit.edu> ISBN 0-262-16228-8, 2004.
- [21] Scala, http://www.devoxx.com/download/attachments/1705916/D8_U_08_06_01.pdf
- [22] Scala, <http://www.slideshare.net/michael.galpin/introduction-to-scala-for-java-developers-presentation>
- [23] Scala, <http://www.slideshare.net/jboner/pragmatic-real-world-scala-45-min-presentation>
- [24] OpenCL, http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf
- [25] OpenCL, The Khronos OpenCL Working Group. The OpenCL Specification version 1.0, 2008. <http://www.khronos.org/opencl/>

- [26] OpenCL, http://developer.amd.com/gpu/amdappsdk/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf
- [27] GPU Sort, http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter46.html
- [28] CUDA, http://www.nvidia.com/object/what_is_cuda_new.html
- [29] ATI Stream, <http://www.amd.com/US/PRODUCTS/TECHNOLOGIES/STREAM-TECHNOLOGY/Pages/stream-technology.aspx>
- [30] Scan, Iverson, Kenneth E., A Programming Language. Wiley, 1962.
- [31] Filter, Blleloch, Guy E. 1990. "Prefix Sums and Their Applications." Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University. , 1990.
- [32] Scan, Blleloch, Guy E. "Scans as Primitive Parallel Operations." IEEE Transactions on Computers 38(11), pp. 1526–1538. , 1989.
- [33] GPU Parallel Algorithms, Matt Pharr, Randima Fernando GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, 2005.
- [34] Bitonic Sort, Batcher, Kenneth E. "Sorting Networks and Their Applications." Proceedings of AFIPS Spring Joint Computer Conference 32, pp. 307–314. , 1968.
- [35] OpenGL, <http://www.opengl.org/>
- [36] J R Hindley, The principal type-scheme of an object in combinatory logic, Transactions of the American Mathematical Society 146, 29--60, 1969.
- [37] Algebraic data type, http://en.wikipedia.org/wiki/Algebraic_data_type
- [38] Leena Unnikrishnan, Scott D. Stoller, and Yanhong A. Liu. Optimized Live Heap Bound Analysis. In Proceedings of the Fourth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI), volume 2575 of Lecture Notes in Computer Science. © Springer-Verlag, 2003.
- [39] Leena Unnikrishnan and Scott D. Stoller. Parametric Heap Usage Analysis for Functional Programs. In Proceedings of the 8th International Symposium on Memory Management (ISMM). © ACM Press, 2009.

[40] Leena Unnikrishnan, Scott D. Stoller, and Yanhong A. Liu. Automatic Accurate Live Memory Analysis for Garbage-Collected Languages. In Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES), 2001.

[41] Scala, Martin Odersky, Bill Venners, Lex Spoon, Programming in Scala: A Comprehensive Step-by-step Guide , Artima Inc; 1st Edition , 2008.

BIOGRAPHICAL INFORMATION

Kishen Das has done his under graduation from NIE, Mysore, India and Masters in Computer Science from University of Texas at Arlington, USA. He has 7 years of experience as a developer in the IT industry. His general interests include compilers, parallel algorithms, programming languages and image processing. He is interested in developing an automated tool for insect identification. His research is broadly related to GPUs, functional and object-oriented languages and parallel programming. He is deeply interested in the study of insect migration and butterfly diversity.