Hadoop+Aparapi: Making heterogenous MapReduce programming easier

Semih Okur, Cosmin Radoi, Yu Lin Computer Science Department
University of Illinois at Urbana Champaign
{okur2, cos, yulin2}@illinois.edu

***Abstract*—Lately, programmers have started to take advantage of GPU capabilities of cloud-based machines. Using the GPUs can decrease the number of nodes required to perform the computation by increasing the productivity per node.**

We combine Hadoop, a widely-used MapReduce framework, with Aparapi, a new Java-to-OpenCL conversion tool from AMD. We propose an easy-to-use API which allows easy implementation of MapReduce algorithms that make use of the GPU. Our API improves upon Hadoop by further hiding the complexity of GPU programming, thus allowing the programmer to concentrate on her algorithm. We also propose an accompanying refactoring that allows the programmer to specify the GPU part of their map computation by using very lightweight annotation.

I. Introduction

As the applications that perform computationally-intensive tasks on large data sets (e.g., data crawled from the web) are becoming widely used, improving their performance is becoming a concern. Traditional CPU computation can no longer satisfy the requirement for performance. GPGPU can deliver higher performance and have been used in various areas, such as scientific computing. Meanwhile, computing in cloud through MapReduce programs is also widely adopted since people can distribute their tasks and execute them in parallel on very many machines.

MapReduce [1] is a programming model or software framework that allows developers to process large data sets. There are two major functions in the MapReduce: the map function takes a set of data and seperates and converts them to another set of data tuples (usually key/value pairs); the reduce function takes the output of the map function and merges those data into a smaller data set. The MapReduce framework was originally proposed by Google to support large data analysis applications. The MapReduce applications were intended to run on a large number of CPUs in the cloud. However, the emergence of the commodity GPUs provides a chance to improve the performance of MapReduce applications by running map/reduce functions on GPUs in the cloud, since GPUs have an order of magnitude higher computation power and memory bandwidth compared with CPUs. In this paper, we propose an innovative approach to leverage the power offered by GPUs in the cloud and our initial results indicate that running MapReduce applications on GPUs can achieve remarkable performance improvement , especially for computational-intensive applications.

Though running MapReduce applications on the GPU of each node in the cloud is attractive, there are some challenges. GPU programming is hard because GPU programming languages, e.g. OpenCL or CUDA, currently lack many of the high-level programming abstractions found in non-graphics languages. Meanwhile, developers have to be familiar with the GPU architecture in order to fully exploit its power. Furthermore, integrating GPGPU with MapReduce is hard since most MapReduce frameworks do not support interfaces for GPU programming. Some previous work tried to solve these challenges by either allowing the MapReduce framework to use the map/reduce functions written in GPU programming language like CUDA [2], or developing a brand-new MapReduce framework that supporting GPUs [3]. In contrast, we propose an approach that allows programmers to to use the GPU without programming in CUDA or OpenCL and is available as a library for Hadoop, a very popular MapReduce library.

Our tool, HAPI, is developed as a library on top of Hadoop [4] and Aparapi [5]. Hadoop is an open-source implementation of MapReduce for Java. By connecting many computers together and scheduling them to work in parallel, Hadoop can process large volumes of data efficiently. Hadoop provides convenient interfaces for implementing map and reduce functions, and it internally implements some efficient, automatic algorithms for distributing the data work across machines, so the users can quickly write and deploy their MapReduce programs and run them in cloud. However, Hadoop only utilizes the underlying parallelism of the CPU cores rather than GPUs. Aparapi is an open source project released by AMD, which aims at improving Java application performance by offloading data parallel operations to an OpenCL [6] capable device such as a GPU [7]. It converts Java byte code to OpenCL at runtime and executes it on the GPU. OpenCL is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors. OpenCL provides a low-level API that allows developers write compute kernels to leverage the massive parallel computing power of GPUs. It has been adopted by Intel, AMD, Nvidia, and ARM Holdings. Developers can implement their applications by leveraging Aparapi interfaces and Aparapi is responsible to run their Java applications on GPUs. If Aparapi cannot execute on the GPU because of the limitations of the Aparapi library, it will execute in a Java thread pool. According to the study of [7], a sample Java based quantitative finance application *JQuantlb* were able to achieve up to 20x improvement in performance after refactoring to use Aparapi kernels. In HAPI, we combine Hadoop and Aparapi by providing new GPU map/reduce interfaces that encapsulate the Aparapi interfaces for Hadoop framework. Our experiment on a computational intensive application shows that by combining Hadoop and Aparapi and running the MapReduce program on GPUs, we get upto 80x speedup.

Organization: The rest of the paper is organized as follows. We present a simple motivating example that shows how a MapReduce application can be run on the GPU and how HAPI works in Section II. We present our design and implementation, including the architecture of our framework and the API we provide in Section III. In Section V, we evaluate HAPI on a widely-used scientific application, an N-Body simulation.

II. Motivating Example

In this section, we use a small example to show how HAPI can significantly improve performance. Suppose we have a MapReduce program shown in Figure 1, which is used for estimating the value of π through quasi-Monte Carlo method. The program tries to generate points inside an 1×1 square randomly and calculate the number of points inside and outside the circle with the diameter 1. Thus, the value of π can be calculated through dividing the number of points inside by the number of points outside the circle.

In the original MapReduce program shown in Figure 1(a), the *map* method takes individual <key, value> pairs as inputs, which are used to generate random points, and create a new list of <key, value> pairs, which are the numbers of points inside and outside the circle. Then, the reduce function (which is not shown in the example) can read all these output key/value pairs generated from each mapper and calculate the total number of points. In the original mapper, all the computation is in the `map` method, which will be further executed on one node in a cloud. Note that the `run` method in Figure 1(a) is used to divide the input data.

However, we observe that the `map` method is computational intensive and that the computation in the `for` loop can be moved into GPU kernel to achieve higher performance. To run the computation in a GPU kernel, we should make sure that the data transferred to GPU is loop-independent. Note that the statement `x*x+y*y>0.25` is loop-independent, thus it can be executed in a GPU kernel. However, since variables *numOutside* and *numInside* are not loop-independent, the statement at line 21 and 23 in Figure 1(a) should not be run on GPU. Moreover, since current version of Aparapi does not support objects, the statements for preparing `Point` objects (line 16 to 19 in Figure 1(a)) should not be executed on GPU either. Therefore, in order to run the map method on GPU, we have to separate the calculations in the map method into three parts: (a) pre-processing the input key/value pairs of map method to prepare data that can be executed on GPU; (b) the GPU kernel calculations; (c) post-processing the calculation results of GPU and output them to the collectors from which reduce method can fetch the data. Our HAPI defines the interfaces for these three operations which can be implemented by developers. The transformed MapReduce program that uses HAPI is shown in Figure 1(b). In the pre-processing phase, we introduce a list to hold the coordinates of the points (line 14-24 in Figure 1(b)), which will be further used by the GPU calculation (line 29 in Figure 1(b)). Then, we use the output of the GPU calculation which is a `boolean` value to denote whether a point is inside or outside the circle, and calculate *numOutside* and *numInside* in the post-processing phase (line 40-44 in Figure 1(b)).

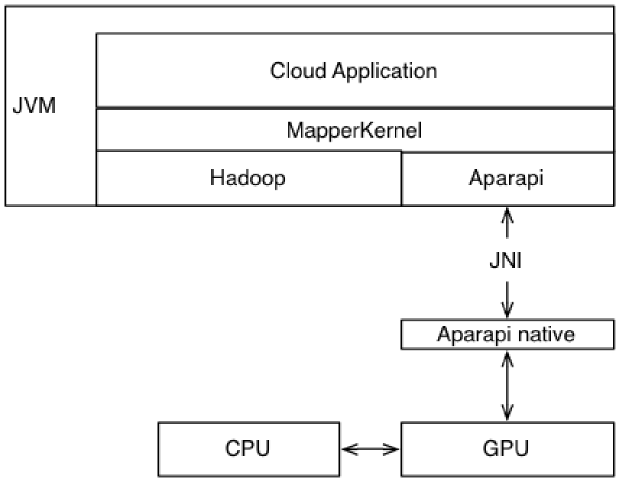

Fig. 2. The architecture of HAPI

III. Implementation of HAPI

A. HAPI *Architecture*

HAPI is designed to facilitate the programming of computational intensive MapReduce applications. Since the objective is to execute the MapReduce applications on GPUs of the nodes in the cloud, HAPI should integrate the features of both Hadoop and Aparapi, and it is built on top both Hadoop and Aparapi.

Figure 2 shows the architecture of HAPI. The original MapReduce applications are executed and scheduled directly by Hadoop, while Hadoop is executed on JVM. However, in HAPI, we insert a layer (i.e. MapperKernel) between Hadoop and MapReduce applications. MapperKernel is a library which extends Hadoop's mapper and meanwhile encapsulates Aparapi's GPU interface. MapperKernel provides three interfaces for pre-processing, GPU calculating and post-processing the data. Thus, under this architecture, the MapReduce applications will leverage the interfaces provided by MapperKernel layer, which will then invoke Hadoop and Aparapi. Furthermore, Aparapi will compile the Java code for GPU calculation to OpenCL code. Aparapi native library will control the communication between CPU and GPU and execute the OpenCL code on GPU. Finally, the data is transferred between Java MapReduce application and OpenCL through JNI.

B. *API*

One approach for making GPU-executable MapReduce easier to program is to provide the user with a lightweight API that, when implemented, automatically makes certain parts of the computation execute on the GPU. The challenge in

```
1 public class SimplePiMapper<K1, V1, K2, V2>
2     implements MapRunnable<K1, V1, K2, V2> {
3
4     public void map(LongWritable offset,
5         LongWritable size,
6         OutputCollector<BooleanWritable,
7         LongWritable> out,
8         Reporter reporter) throws IOException {
9         long numInside = 0L;
10        long numOutside = 0L;
11
12        final Point[] points =
13            new Point[size.get()];
14
15        for (int i = 0; i < size.get(); i++) {
16            points[i] = new Point(Math.random(),
17                Math.random());
18            final float x = points[i].x - 0.5f;
19            final float y = points[i].y - 0.5f;
20            if (x * x + y * y > 0.25) {
21                numOutside++;
22            } else {
23                numInside++;
24            }
25        }
26
27        out.collect(new BooleanWritable(true),
28            new LongWritable(numInside));
29        out.collect(new BooleanWritable(false),
30            new LongWritable(numOutside));
31    }
32
33    public void run(RecordReader<K1, V1> input,
34        OutputCollector<K2, V2> output,
35        Reporter reporter) throws IOException {
36
37        // allocate key & value instances
38        // that are re-used for all entries
39        K1 key = input.createKey();
40        V1 value = input.createValue();
41
42        while (input.next(key, value)) {
43            // map pair to output
44            map((LongWritable)key,
45                (LongWritable)value,
46                (OutputCollector<BooleanWritable,
47                 LongWritable>)output, reporter);
48        }
49    }
50 }
```

(a) Original MapReduce Program.

```
1 public class GPUPiMapper extends
2     // the first two type are input key/value pair
3     // the last two type are output key/value pair
4     MapperKernel2FloatToBoolean<LongWritable, LongWritable,
5     BooleanWritable, LongWritable> {
6
7     @Override
8     public List<FloatTuple2> preprocess(
9         RecordReader<LongWritable, LongWritable> input,
10        Reporter reporter) throws IOException {
11
12        LongWritable key = input.createKey();
13        LongWritable value = input.createValue();
14        ArrayList<FloatTuple2> allGpuIn
15            = new ArrayList<FloatTuple2 >();
16
17        while (input.next(key, value)) {
18            for (int i = 0; i < value.get()- key.get();
19                i++)
20                allGpuIn.add(new FloatTuple2(
21                    (float) Math.random(),
22                    (float) Math.random()));
23        }
24        return allGpuIn;
25    }
26
27
28    public boolean gpu(float x, float y) {
29        return x * x + y * y > 0.25f;
30    }
31
32
33    @Override
34    public void postprocess(List<Boolean> gpuOut,
35        OutputCollector<BooleanWritable, LongWritable>
36        output)
37        throws IOException {
38        int numInside = 0;
39        int numOutside = 0;
40        for (boolean x : gpuOut)
41            if (x)
42                numInside++;
43            else
44                numOutside++;
45        output.collect(new BooleanWritable(true),
46            new LongWritable(numInside));
47        output.collect(new BooleanWritable(false),
48            new LongWritable(numOutside));
49    }
50 }
```

(b) MapReduce Program with HAPI.

Fig. 1. Motivation Exampe.

designing such an API lays in balancing three conflicting forces:

1) **need for flexibility:** the API should be flexible enough to allow the implementation of most, if not all, map-style computations
2) **technical limitations:** automatically converting Java code to OpenCL is challenging. APARAPI is the state of the art in this respect but even it only recognizes a very narrow subset of the Java language. This subset does not include basic features like objects or dynamic dispatch. Although APARAPI and similar tools will improve, full recognition of the Java language is hard, if not impossible, due to the SIMD model of computation inherent to GPUs.
3) **ease of use:** this is the primary goal of our tool

Our solution, presented in Fig. 4, is an API that segregates the part of the computation that is most beneficial to be run on the GPU to a single method and wraps the rest of the complex Java computation around it. The `map` operation is split in three stages:

- **preprocessing**: in this stage, the input $\langle Key, Value \rangle$ pairs are processed to prepare the input values for the GPU computation. This stage corresponds to lines 8-25 in the manually transformed example (Fig.1-b).
- **gpu**: in this stage, the expensive part of the computation is run on the GPU – lines 28-30 in the example.
- **postprocessing**: in this stage the output from the GPU is interpreted and the results are sent to the `Mapper`'s output – lines 34-49 in the example.

Figure 4 shows a small portion of the classes available to the user. The abstract `MapperKernel` makes the link between Hadoop and Aparapi. The user can extend it directly or he can choose to extend one of its easier-to-use subclasses. The subclasses are named after the signature of the gpu mapper. For instance, `MapperKernel2FloatToBoolean` is a mapper whose gpu() method takes a pair of float primitives and returns a boolean. The user just needs to override the three methods in its interface that correspond to the three aforementioned stages, i.e. `preprocess`, `gpu`, and `postprocess`.

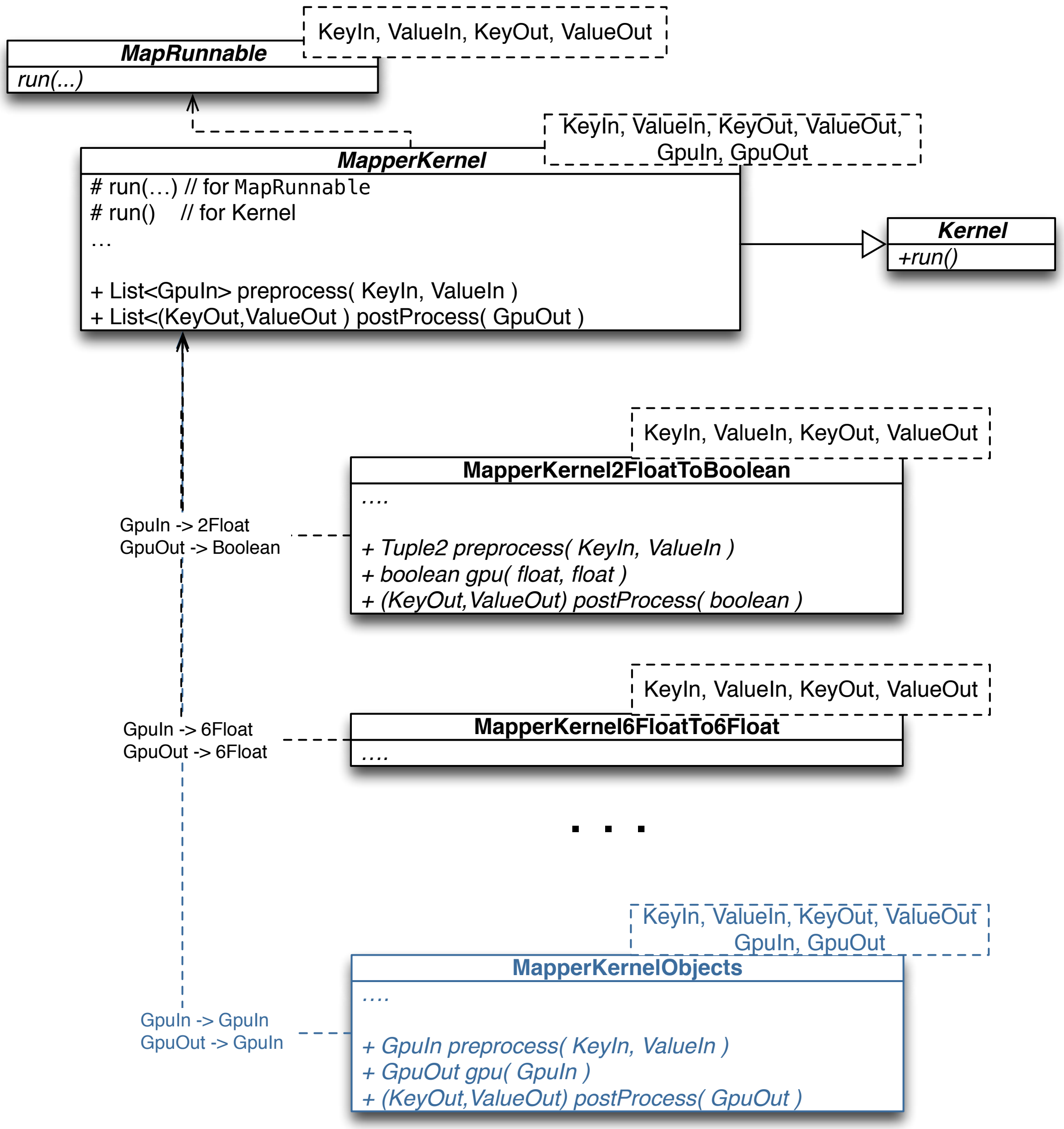
KeyIn, ValueIn, KeyOut, ValueOut
MapRunnable
run(...)
KeyIn, ValueIn, KeyOut, ValueOut, GpuIn, GpuOut
MapperKernel
# run(...) // for MapRunnable
# run() // for Kernel
...
+ List<GpuIn> preprocess(KeyIn, ValueIn)
+ List<(KeyOut,ValueOut) postProcess(GpuOut)
Kernel
+run()
KeyIn, ValueIn, KeyOut, ValueOut
MapperKernel2FloatToBoolean
....
+ Tuple2 preprocess(KeyIn, ValueIn)
+ boolean gpu(float, float)
+ (KeyOut,ValueOut) postProcess(boolean)
GpuIn -> 2Float
GpuOut -> Boolean
KeyIn, ValueIn, KeyOut, ValueOut
MapperKernel6FloatTo6Float
....
GpuIn -> 6Float
GpuOut -> 6Float
KeyIn, ValueIn, KeyOut, ValueOut GpuIn, GpuOut
MapperKernelObjects
....
+ GpuIn preprocess(KeyIn, ValueIn)
+ GpuOut gpu(GpuIn)
+ (KeyOut,ValueOut) postProcess(GpuOut)
GpuIn -> GpuIn
GpuOut -> GpuIn

Fig. 4. The KernelMapper API

Figure 1.b shows how the KernelMapper class can be extended to implement Pi estimation computation in the motivating example. The KernelMapper requires six type parameters (generic parameters): the first two types are for the input $\langle key, value \rangle$ pair, the middle two are the input and output types for the GPU mapper, and the last two are the for the output $\langle key, value \rangle$ pair. In the case of the `GPUPiEstimator` the type parameters are `<LongWritable, LongWritable, Point, Boolean, BooleanWritable, LongWritable>`.

The `preprocess` method takes an input $\langle key, value \rangle$ pair and outputs a collection of values that will be fed to the GPU mapper. In our example, the preprocess takes an offset and size and outputs a collection of random points.

The `gpu` method describes how each `GpuIn` element is processed by the GPU. This is the part of the computation that Aparapi will automatically convert to OpenCL. This means that the Java code needs to conform to the Aparapi framework limitations. The code reachable from the `gpu` method is limited to:

- primitive data types
- one-dimensional array
- no object creation of array initialization
- no recursion
- no passing of the array reference
- no method calls to objects other than `this`

As the framework doesn't allow method calls, it provides a set of commonly-used mathematical functions as methods of the `Kernel` class. As Kernel is an ancestor of KernelMapper, the mathematical functions are available to the user-extended class.

The Aparapi framework is continuously improved. The complete and up-to-date list of features and limitations can be found on Aparapi's website [5].

In the example in Fig. 1.b, the `gpu` method verifies whether each point is inside the circle of radius one. The computation is very simple and conforms to Aparapi's limitations.

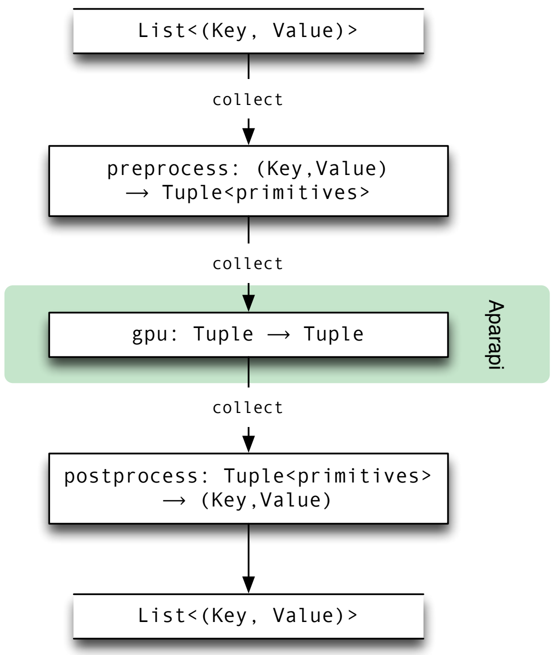
List<(Key, Value)>
collect
preprocess: (Key,Value) ⟶ Tuple<primitives>
collect
gpu: Tuple ⟶ Tuple
Aparapi
collect
postprocess: Tuple<primitives> ⟶ (Key,Value)
List<(Key, Value)>

Fig. 3. Map operation

Going further, the `postprocess` method takes as an input a `List` or `KernelOut` values and computes the final output of the Mapper. For our example, `postprocess` takes a list of booleans saying whether the points are inside the circle and counts the points inside/outside. It then writes the results to the `Context` object.

IV. Related work

Dig et al. [8] propose an automated refactoring for transforming Java loops to use the ParallelArray library. ParallelArray is capable of executing map, reduce and scan operation in parallel on the CPU.

Leung et al.[9] present an extension to JikesRVM, a research Java virtual machine, that enables it to speed up computationally intensive loops by executing them on the GPU. The parallelization is done by transforming suitable loops from Java bytecode to a special IR used by RapidMind, a higher-level framework for GPGPU programming. RapidMind, in turn, generates GPU code and executes it.

Baskaran et al. [10] and Yang et al. [11] propose separate solutions for optimizing GPU kernels. They transform the code to improve memory access by vectorization and memory coalescing, do loop unrolling and tiling for data reuse, and remap thread blocks to avoid partition camping. The best optimization parameters are found by empirical search, i.e., running the program under different parameters.

Farivar et al. [2] present a heterogeneous architecture, called MITHRA, which is a set of GPU-enabled nodes running on a MapReduce cluster. They demonstrate that GPU accelerated MapReduce clusters can achieve significant speedup (254x). The MITHRA architecture runs MapReduce programs that have GPU-executable map functions written in CUDA. However, our project provides an automatic way to execute MapReduce programs written in Java on heterogeneous MapReduce clusters, without requiring the user to manually translate the map and reduce functions to CUDA.

He et al. [3] developed a GPU based MapReduce framework, called Mars. Their vision is to hide the programming complexity of the GPU, while programmers take advantage of ease-of-use of the MapReduce. Although they indicate the integration of Mars and Hadoop as a future work, their work is similar with ours in terms of their vision that aims to hide the GPGPU complexity.

Jacob et al. [12] provides an abstract APIs that allow programmers to implement GPGPU programs by hiding GPGPU complexity. They also provide a IDE tool that makes easy to use their abstract APIs. Their APIs and tool supports both OpenCL and CUDA. While this work is not related with MapReduce framework, it is important to note that there are recent projects aiming to hide GPGPU complexity.

Lee et al. [13] present a compiler framework for automatic source to source translation of OpenMP programs to CUDA based programs. They implemented the translation steps for Cetus compiler infrastructure that allows source-to-source transformations for C programs. While this project automatically allows the programmer to take advantage of GPU power in their OpenMP programs, our project will allow the programmer to use GPU power in their MapReduce programs in the same manner.

V. Evaluation

In this section, we evaluate our GPU-based MapReduce framework in comparison with its CPU-based counterpart, which has a typical `Mapper` class.

Our experiments were performed on a server machine with Nvidia GeForce GTX275 GPU and Intel Core i7 processor running Ubuntu 10.10. The GPU consists of 240 stream processors and the frequency of the core clock is 633MHz. In contrast, the CPU has four cores running at 2.67GHz. The main memory is 3GB, and the device memory of the GPU is 896MB.

To integrate HaPi with Hadoop, the developer just needs to add two parameters to Hadoop's configuration file:

- `-Djava.library.path= location of aparapi native library`
- `mapred.map.tasks=1`.

The first parameter tells Hadoop where the Aparapi native library resides and the second parameter limits the number of mapper tasks to one per node. If Hadoop executes several GPU mapper tasks in parallel, the tasks concurrently initiate GPU calls and this can cause device unavailability errors.

To evaluate the efficiency of HaPi, we ask two research questions:

1) Does using HaPi decrease complexity?
2) Does using HaPi improve speed?

To answer these questions, we evaluate HaPi on a simple implementation of an N-Body simulation. The implementation is straightforward, leading to $O(n^2)$ complexity.

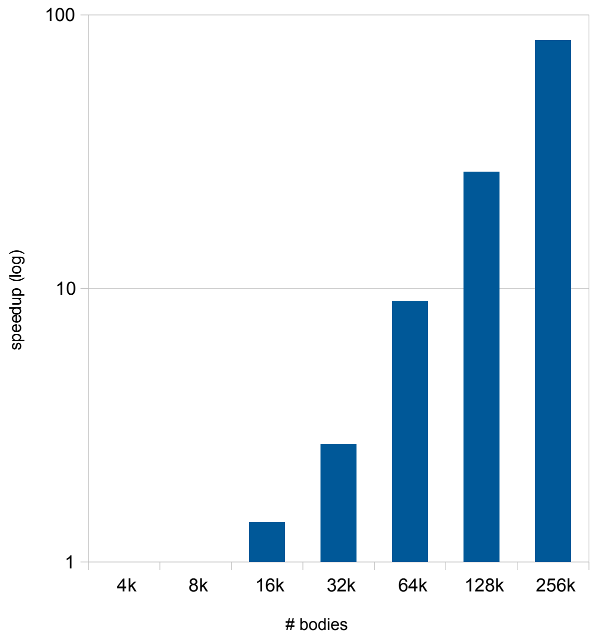

Fig. 5. Speed up - HAPI over CPU

A. *Does using* HAPI *decrease complexity?*

We have implemented N-Body in two ways: one is with a typical CPU-based Mapper and one is with a HAPI-based Mapper. Both implementations use the same code conventions. The CPU-based Mapper has 84 lines of source code while theHAPI-based Mapper has 62 lines of source code. HAPI decreases the code complexity by hiding the low-level GPGPU constructs. At least in this case, HAPI makes the code as least as compact as the CPU implementation.

B. *Does using* HAPI *improve speed?*

We executed both the CPU and the GPU-based implementations with exponentially increasing data sizes ranging from 4k to 256k. Figure 5 shows the speed-up that we get from these inputs. As the number of bodies increases, the speed-up also increases at nearly exponential rate because the N-Body has $O(n^2)$ complexity.

Even though our machine uses a typical graphics card, we get 80x speed-up with 256k bodies. Table I shows the execution time and speedup from the experiments. For the small inputs, between 4k and 16k, the execution time for GPU-based Mapper is roughly same as for the CPU-based one. The reason is that the overhead of Hadoop MapReduce framework is around 7 seconds.

TABLE I
EXPERIMENT RESULTS

# Bodies	CPU runtime (s)	HAPI runtime (s)	Speed-up
4k	7.1	7.1	1
8k	7.1	7.1	1
16k	10.2	7.1	1.4
32k	19.1	7.1	2.7
64k	64.2	7.1	9.0
128k	271.9	10.2	26.7
256k	1047.4	13	80.6

VI. DISCUSSION

Hybrid CPU-GPU mapper. In addition to powerful GPU's, cloud nodes also have many CPU cores. In the current implementation, HAPI uses only one CPU core for preparing the data for the GPU and leaves the CPU cores idle. An idea worth exploring is using both the CPUs and GPU on the machine in parallel. The challenge in this approach would be balancing the work to achieve good utilization of all processing units.

Evolving Aparapi. In the current implementation, HAPI tries to provide the programmer with an elegant API while working within the bounds of Aparapi's subset of Java. Forking or evolving Aparapi can provide many opportunities for improving the API. For example, Aparapi does not currently support boxed primitives. Adding this capability would allow HAPI to significantly reduce the number of `MapperKernel` subclasses while retaining the same ease of use and flexibility.

Automated hybrid execution Another ambitious goal would be to execute MapReduce programs on heterogeneous architectures without user-assistance. In this approach, the GPGPU complexity would be completely hidden from the programmer and he could write the `Mapper` class in the classical Hadoop fashion.

VII. CONCLUSION

Using GPGPUs in the cloud comes with the promise of great performance improvements, but it is also very hard to program. In this paper we present HAPI, a tool that allows programmers to write MapReduce mappers that delegate computationally-intensive parts to the GPU by using Java, without programming in OpenCL or CUDA.

Our preliminary evaluation suggests that using HAPI can provide very good speedup, up to 80x, while keeping the program complexity low.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251264

[2] R. Farivar, A. Verma, E. Chan, and R. Campbell, "Mithra: Multiple data independent tasks on a heterogeneous resource architecture," in *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, 31 2009-sept. 4 2009, pp. 1 –10.

[3] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 260–269. [Online]. Available: http://doi.acm.org/10.1145/1454115.1454152

[4] "Hadoop," http://hadoop.apache.org/.

[5] "Aparapi," http://developer.amd.com/zones/java/aparapi/Pages/default.aspx.

[6] "OpenCL," http://www.khronos.org/opencl/.

[7] "Leveraging aparapi to help improve financial java application performance."

[8] D. Dig, M. Tarce, C. Radoi, M. Minea, and R. Johnson, "Relooper: refactoring for loop parallelism in java," in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, ser. OOPSLA '09. New York, NY, USA: ACM, 2009, pp. 793–794. [Online]. Available: http://doi.acm.org/10.1145/1639950.1640018

[9] A. Leung, O. Lhoták, and G. Lashari, "Automatic parallelization for graphics processing units," in *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, ser. PPPJ '09. New York, NY, USA: ACM, 2009, pp. 91–100. [Online]. Available: http://doi.acm.org/10.1145/1596655.1596670

[10] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "A compiler framework for optimization of affine loop nests for gpgpus," in *Proceedings of the 22nd annual international conference on Supercomputing*, ser. ICS '08. New York, NY, USA: ACM, 2008, pp. 225–234. [Online]. Available: http://doi.acm.org/10.1145/1375527.1375562

[11] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A gpgpu compiler for memory optimization and parallelism management," in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 86–97. [Online]. Available: http://doi.acm.org/10.1145/1806596.1806606

[12] F. Jacob, D. Whittaker, S. Thapaliya, P. Bangalore, M. Mernik, and J. Gray, "Cudacl: A tool for cuda and opencl programmers," in *High Performance Computing (HiPC), 2010 International Conference on*, dec. 2010, pp. 1 –11.

[13] S. Lee, S.-J. Min, and R. Eigenmann, "Openmp to gpgpu: a compiler framework for automatic translation and optimization," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '09. New York, NY, USA: ACM, 2009, pp. 101–110. [Online]. Available: http://doi.acm.org/10.1145/1504176.1504194