SparkCL: A Unified Programming Framework for Accelerators on Heterogeneous Clusters

Sung-Soo Kim, Jongho Won

ETRI
218 Gajeong-ro, Yuseong-gu, Deajeon, South Korea
{sungsoo,jhwon}@etri.re.kr
http://www.etri.re.kr

Abstract. We introduce SparkCL, an open source unified programming framework based on Java, OpenCL and the Apache Spark framework. The motivation behind this work is to bring unconventional compute cores such as FPGAs/GPUs/APUs/DSPs and future core types into mainstream programming use. The framework allows equal treatment of different computing devices under the Spark framework and introduces the ability to offload computations to acceleration devices. The new framework is seamlessly integrated into the standard Spark framework via a Java-OpenCL device programming layer which is based on Aparapi and a Spark programming layer that includes new kernel function types and modified Spark transformations and actions. The framework allows a single code base to target any type of compute core that supports OpenCL and easy integration of new core types into a Spark cluster.

Keywords: heterogeneous computing, parallel computing

1 Introduction

It has been shown theoretically [13, **?**] and increasingly in practice that data centric [12] and HPC [15] applications can benefit from using different types of compute cores in tandem in order to achieve a computational intensive goal. The benefits translate into an increase in the overall power efficiency and performance of a system. Several major examples of such gains can be found in the ever growing list of heterogeneous systems on the green 500 list [6]. A heterogeneous system is typically comprised of conventional cores (CPUs) and unconventional cores (GPUs/FPGAs/DSPs). Such a system employs both conventional and unconventional cores on different types of tasks. Heterogeneous computing brings improved performance and power efficiency through specialization.

One of the main difficulties with hardware specialization is that different types of hardware devices require different types of programming languages, development tools and technical skill sets to master them. Even from a software development stand point alone the increase in the number of lines of code will lead to an increase in system instability [19].

For heterogeneous computing to become a mainstream system design choice, standards need to be adopted for programming various devices. One such standard, OpenCL [16], has started to gain momentum since its introduction in 2008. In the past several years it is becoming an industry choice, with support extended across multiple architectures, albeit not without resistance from manufacturers that supply their own specialized software development environments.

OpenCL although a good step in the right direction is still a difficult challenge for many software engineers [9]. The need to deal with the technical details of OpenCL and the fact that C/C++ is not the language of choice in many development environments such as data centers hampers its chances of success and main stream adoption.

We are now and have been in the past several years in the age of big data processing. The need for processing vast amounts of data and the prediction that we will soon enter the age of exascale computing, a thousand fold increase in the amount of data and calculations on todays standards, is a major motivator in the search for more power efficient computing solutions. As of today we are far from the exascale power efficiency goals set by the DOE for 2020 [22]. In order to achieve those goals significant changes need to be made in the way we design and program large scale systems. In addition it is predicted [14] that heterogeneity will play a major role in future exascale systems.

Todays main stream big data processing frameworks such as Spark [24] and Hadoop [4] treat computers as a collection of conventional cores. If they do utilize unconventional cores, it is through specialized libraries typically of the GPU kind [11]. They do not treat unconventional cores as equal citizens in a computing environment and they do not open the possibility of integrating new types of unconventional cores into their midst easily. SparkCL is an attempt at changing the status quo by allowing a mainstream framework (Spark) to seemingly integrate unconventional cores into its core operations. To achieve that goal we introduce two open source frameworks: aparapi-ucores [7] and spark-ucores (code name SparkCL) [8]. Together they allow a single java code base to be sent to Spark workers and accelerated through specialized hardware if available. The rest of this paper is organized as follows. Section two gives an overview of the SparkCL framework, section three discusses the execution model and the new functionality added to Spark. Section four discusses some related work, sections five and six discuss conclusions and future directions respectively.

2 SparkCL Overview

We have combined the power of two separate open source frameworks: Aparapi and Spark to create a new framework that allows accelerating execution using simple java code running on top of Aparapi and the Spark framework.

As can be seen in Fig 1, the user execution kernel written in Java gets executed by the SparkCL framework which in turn delegates the code to Spark and eventually to Aparapi UCores on OpenCL capable devices on the cluster. To allow this programming flow, with indifferent treatment of unconventional cores,

a new programming layer is built on top of the existing Spark and Aparapi infrastructure.

3 Spark Layer Modifications

Spark, as do other big data frameworks, views computers as collections of cores. It does not take unconventional cores into consideration when it comes to performing general computing functions. To accommodate unconventional cores we had to modify the way functions and data are sent to acceleration devices. Several key issues have been identified in our previous work [20, **?**] on accelerating the Hadoop framework: 1) A task has to be computational intensive to justify the overhead of using an accelerator i.e. do to architectural limitations, data transfer overhead, and the way integration of most accelerators into compute fabric is done today, it is difficult to accelerate computationally simple tasks. Instead, the focus should be on highly compute intensive tasks. 2) Enough data must be collected in order to enable efficient acceleration. In most cases a combination of computation complexity and a large amount of data to process is required for efficient acceleration. 3) High level, big data programming frameworks tend to help users generalize their algorithms by distancing them from the concerns of efficient operation of the underlying hardware. That means that things such as memory allocation and device friendly data types are not their strong suite or focus of interest. To be able to utilize accelerators efficiently, we need to tackle the aforementioned problems before any hope of efficient integration and acceleration can be made.

3.1 Execution Model and Kernel Types

In standard Spark a function has a single call site and is replicated across the cluster. This is sufficient for CPUs but does not allow for the special requirements of accelerators, namely large amounts of primitive type data and complex computations. To allow for such needs we have created what we call a SparkKernel type. This type encapsulates a run function, an Aparapi Kernel (with functionality similar to the existing call override in standard Spark), and two additional functions used to prepare/pre-process the data before running the kernel and post processing of data after the kernel was executed.

An example of an abstract kernel type can be seen in Fig 2. The user overrides the abstract kernel type with his implementations of the three functions. 3.1.1 Kernel Function Overrides The following are the standard functions that should be overridden in the users kernel. 3.1.1.1 voidmapParameters(Tdata) Prepare the data and decide which device to use if any. Users can use the API to do things such as set the computation range, choose execution mode (CPU/Accelerator/JTP) and selectively run the kernel if conditions are not ideal to run an accelerated version, for example, in a case where not enough data exists to justify the execution of the kernel. 3.1.1.2 void run() A standard Aparapi kernel that will be executed on each capable device node on the cluster.

The framework will try to cache it by default to avoid multiple instantiation on each worker node. 3.1.1.3 R mapReturnValue(T data) Post process the data and handle any activity or cleanup needed post kernel execution. If the kernel was not selectively executed, we can call an alternative compute function here. 3.1.2 Simple User Kernel Example Fig 3. Shows a simple vector add kernel, the heart of the kernel is a simple two line code segment that gets executed on accelerated hardware if available. Map parameters and map return facilitate the pre/post processing of the kernel. Complete examples with source can be found accompanied to the open source SparkCL framework distribution [8]. 3.1.3 SparkCL Transformations and Actions Standard Spark actions such as Map and transformations such as Reduce needed to be ported to the new framework. SparkCL introduces three such new constructs: 1) MapCL - Map a SparkCL kernel using a Spark style map function. 2) MapCLPartition - Map a SparkCL kernel using a Spark style map partition function. 3) ReduceCL- Map a SparkCL kernel using a Spark style tree-reduce function. 3.1.4 AparapiUCoresIntegration Behind the scenes after the Spark-SparkCL integration has been handled by SparkCL, the kernel is sent to a standard Spark worker, together with the Aparapi UCores Java based library. The worker in-turn will execute the Kernel using the Aparapi UCores Framework [7]. Aparapi UCores is a fork of the Aparapi framework [3]. The standard Aparapi library has been expanded to support binary execution flow (needed to support FPGAs and optional for other devices), accelerator types, and platform selection support to allow for easy integration into multiple OpenCL environments and big data frameworks. 3.1.5 Worker Setup and Spark Integration A SparkCL worker is a standard Spark worker that is capable of running both accelerated and non-accelerated tasks i.e. standard Spark tasks. The key to this ability are the following factors: 1) A SparkCL worker is an unmodified Spark Worker class org.apache.spark.deploy.worker.Worker 2) If a standard Spark job is sent to the kernel, it will be executed using the standard worker class functionality. 3) If a SparkCL job is sent to the worker, the SparkCL extensions will take over and handle the acceleration. 4) A SparkCL worker is executed using start-up scripts that make sure it binds to an OpenCL specific/general (ICD capable) implementation on the node. 5) A worker can set its preferred OpenCL orientation on start-up (device/framework) and a user can modify it thorough the kernel code. To help accomplish the above, included in the distribution is a simple launch script: scripts/spark-submit-and-set-env.sh [OpenCL implementation] [Architecture] [Device Type] OpenCL implementation can be std or fpga for now. Note that std means any ICD compatible OpenCL implementation (AMD/Intel/NVidia etc.) Architecture - AMD/Intel/NVidia etc. this string gets sent to Aparapi UCores to filter the available platforms. This is where you select the OpenCL platform you want the worker to bind to. Device Type selected default acceleration device can be CPU/ACC/JTP. In addition the kernel code can choose to switch between devices programmatically inside the mapParameters function. An example of how to start different types of workers on a single node can be seen in Fig 5. The specific node has two OpenCL implementations (Altera/AMD) and three types of devices FPGA, GPU and

CPU. We can potentially launch three types of different workers each using a different device type. Note that for the purpose of Spark Scheduling, in the above example, we tell the worker to use one core. This way it will be sent acceleration tasks sequentially and they will not compete on the same hardware acceleration resources. SparkCL has provisions to prevent contention, but this method avoids the potential conflict altogether.

3.2 Demo Applications

To demonstrate the operation of the framework we have implemented three algorithms in SparkCL. They can be found in the distributed open source project [8].

SparkCLPi A SparkCL distributed Pi calculation based on the MapCL kernel variants. There are several implementations including an optimized standard Spark one which is significantly faster than the one supplied by the standard Spark distribution on our performance tests. All Pi tests can be launched by name from the command line to allow for easy performance comparison testing.

SparkCLVectorAdd A SparkCL distributed vector addition based on the ReduceCL kernel type. Given sufficient work it executes on the potentially accelerated workers instead of on the single driver (Uses Spark tree reduce instead of reduce).

SparkCLWordCount A SparkCL distributed word count application based on the MapCL kernel type. It is a functional replicate of the Spark demo but implemented in SparkCL. It demonstrates a somewhat more complex kernel with local data and selective execution.

4 Related Work

Several high level OpenCL based programming frameworks have been developed in recent years [3, **?**,**?**]. Although these libraries support OpenCL their focus is on GPU and CPU development. Several attempts at accelerating the Hadoop framework involved using a modified MapReduce framework to accommodate for the limitations and strengths of accelerators [17, **?**]. A previous attempt at accelerating a Hadoop K-Means algorithm on FPGAs showed good promise but involved significant development efforts [14, 15]. Several other attempts were made at creating accelerator frameworks, but all in all these attempts concentrated on GPUs leaving out other types of accelerators hence they did not fully adopt the broader premise of unconventional cores, that each accelerator will have strengths and weaknesses and different types of accelerators should be used to tackle different computation tasks.

5 Conclusion

We introduce SparkCL an open source high level Java- OpenCL framework based on Apache Spark and Aparapi, capable of running high performance and data

centric application across different platforms and devices with the hope that it will be helpful to heterogeneous system users and the research community. We offer the potential to use a single high level Java-OpenCL code base across different architectures in a heterogeneous cluster in order to try to maximize code reuse while attempting to harness the power and efficiency of heterogeneous computing.

6 Future Directions

We plan to continue work on SparkCL framework and hope that others will find it useful and join the effort of developing and extending the SparkCL framework. The project is in its infancy and leaves a lot to be desired in optimization, setup and usability, we envision a framework that would be easy to use and install so that any programmer/researcher would be able to use this system without having intimate knowledge of the underlying architectures and hardware. concerned with power efficiency

References

1. Ruby-OpenCL: A Ruby binding of OpenCL (Oct 2009), `http://ruby-opencl.rubyforge.org`
2. JavaCL (Aug 2011), `https://code.google.com/p/javacl/`
3. Aparapi API for data parallel Java (Feb 2012), `https://code.google.com/p/aparapi/`
4. Apache Hadoop project (2014), `http://hadoop.apache.org`
5. Apache Spark project (2015), `http://spark.apache.org`
6. The Green500 List - November 2014 (Nov 2014), `http://www.green500.org/news/green500-list-november-2014`
7. Aparapi for Unconventional Cores. (Feb 2015), `https://gitlab.com/mora/aparapi-ucores`
8. Spark for Unconventional Cores (SparkCL) (2015), `https://gitlab.com/mora/spark-ucores`
9. TIOBE Software (2015), `http://www.tiobe.com`
10. Basaran, C., Kang, K.D.: Grex: An Efficient MapReduce Framework for Graphics Processing Units. Journal of Parallel and Distributed Computing 73(4), 522–533 (Apr 2013)
11. Canny, J.: Interactive Machine Learning. University of California, Berkeley (2014)
12. Chalamalasetti, S., Margala, M., Vanderbauwhede, W., Wright, M., Ranganathan, P.: Evaluating FPGA-acceleration for Real-time Unstructured Search. In: Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software. pp. 200–209. ISPASS '12, IEEE Computer Society (2012)
13. Chung, E.S., Milder, P.A., Hoe, J.C., Mai, K.: Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs? In: Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture. pp. 225–236. MICRO '43, IEEE Computer Society (2010)
14. Dongarra, J., et. al: The International Exascale Software Project Roadmap. International Journal of High Performance Computing Applications 25(1), 3–60 (Feb 2011)

15. Gan, L., Fu, H., Luk, W., Yang, C., Xue, W., Huang, X., Zhang, Y., Yang, G.: Accelerating solvers for global atmospheric equations through mixed-precision data flow engine. In: Proceedings of the 2013 23rd International Conference on Field Programmable Logic and Applications (FPL). pp. 1–6. FPL '13, IEEE Computer Society (2013)
16. Group, K.O.W.: OpenCL-The open standard for parallel programming of heterogeneous systems (2011), http://www.khronos.org/opencl
17. He, B., Fang, W., Luo, Q., Govindaraju, N.K., Wang, T.: Mars: A MapReduce Framework on Graphics Processors. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques. pp. 260–269. PACT '08, ACM (2008)
18. Huang, S., Xiao, S., Feng, W.: On the Energy Efficiency of Graphics Processing Units for Scientific Computing. In: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing. pp. 1–8. IPDPS '09, IEEE Computer Society (2009)
19. McConnell, S.: Code Complete: A Practical Handbook of Software Construction. Microsoft Press (1993)
20. Segal, O., Margala, M., Chalamalasetti, S., Wright, M.: High level programming framework for FPGAs in the data center. In: Proceedings of the 2014 24th International Conference on Field Programmable Logic and Applications (FPL). pp. 1–4. FPL '14, IEEE Computer Society (2014)
21. Segal, O., Margala, M., Chalamalasetti, S.R., Wright, M.: High Level Programming for Heterogeneous Architectures. CoRR abs/1408.4964 (2014)
22. Shalf, J., Dosanjh, S., Morrison, J.: Exascale Computing Technology Challenges. In: Proceedings of the 9th International Conference on High Performance Computing for Computational Science. pp. 1–25. VECPAR'10, Springer-Verlag (2011)
23. Xin, R.S., Rosen, J., Zaharia, M., Franklin, M.J., Shenker, S., Stoica, I.: Shark: SQL and Rich Analytics at Scale. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. pp. 1324. SIGMOD 13, ACM (2013)
24. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster Computing with Working Sets. In: Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing. pp. 10–10. HotCloud'10, USENIX Association (2010)