International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 11
***Article Received:** 25 July 2023 **Revised:** 12 September 2023 **Accepted:** 30 November 2023*

Analyzing Execution Time of Jit4gpu and Jit4opencl Against CPU Benchmarks

Palyam Nata Sekhar [1]

[1] Research Scholar, Department of Computer Science, Dr. A. P. J. Abdul Kalam University, Indore, Madhya Pradesh

Dr. Arpana Bharani [2]

[2] Supervisor, Department of Computer Science, Dr. A. P. J. Abdul Kalam University, Indore, Madhya Pradesh

ABSTRACT

There is a critical need to optimize execution time due to the increasing computing needs of current applications. By optimizing code dynamically during runtime, JIT compilation provides a dynamic method to code optimization and has the ability to improve speed for activities that need parallel processing. For very parallel workloads, JIT4GPU is the way to go, and JIT4OPENCL is all about making the most of the OpenCL framework so that parallel computation may happen on many platforms. To achieve great performance in jobs demanding heavy calculation, JIT4GPU optimizes code to fully use the parallel processing capabilities of GPUs, making it particularly designed for GPU architectures. But JIT4OpenCL is all about the OpenCL platform, which means it's more versatile and can run on a wide range of hardware, including GPUs and CPUs. The experimental assessment compares the execution time of CPU-optimized benchmarks to those built using jit4GPU and jit4OpenCL.

Keywords: Compilation, Benchmark, Computing, Performance, Python

I.INTRODUCTION

With the introduction of varied hardware designs such as GPUs (Graphics Processing Units) and multi-core CPUs, the demand for efficient and high-performance code execution has grown in importance in the ever-changing computing environment. While these designs provide tremendous processing power, optimizing software to fully use them is a significant task. One effective approach to these problems is the rise of Just-In-Time (JIT) compilation, which allows programs to be optimized dynamically and executed in real-time according to the hardware they are running on. Some of the most well-known JIT compilation frameworks are JIT4GPU and JIT4OpenCL, which focus on different areas of GPU and OpenCL-based programming, respectively. For code optimization and execution on GPU architectures, there is a JIT compilation system called JIT4GPU. Graphics processing units (GPUs) are very parallel processors that were first developed for visual rendering but are now extensively used for general-purpose computing jobs because of how well they handle large-scale parallelism. By taking use of this parallelism, JIT4GPU generates optimized machine code that targets the GPU hardware directly, leading to significant improvements in speed for activities that need a lot of computation. In scientific simulations, image processing, and deep learning applications, this framework shines when the workload can be parallelized.

Conversely, JIT4OpenCL is an extensible JIT compilation system for the OpenCL (Open Computing Language) environment. For parallel programming on heterogeneous platforms, such as CPUs, GPUs, and others, OpenCL is the open standard to follow. Because it produces code that is compatible with all devices that support OpenCL, JIT4OpenCL is a very portable solution. Because it let the same code to be performed across numerous devices with few adjustments, this flexibility is especially helpful in contexts with varied hardware. In contrast to frameworks like JIT4GPU, which create code with a high degree of specificity for any given hardware architecture, this generic approach typically results in subpar performance.

With more and more industries requiring efficient, high-performance applications, comparing JIT4GPU with JIT4OpenCL becomes necessary. Although they use distinct approaches and target platforms, both frameworks strive to increase program execution efficiency. In contrast to JIT4OpenCL, which prioritizes adaptability and cross-platform compatibility to make it work with more devices, JIT4GPU is dedicated on optimizing performance on GPU hardware by taking use of its distinctive architectural characteristics. In order to choose the most appropriate tool for their unique requirements, researchers and developers must be familiar with the benefits and drawbacks of each framework. Using JIT4GPU and JIT4OpenCL as examples, we thoroughly compare their respective program performance in this research. We test important performance indicators across various hardware configurations, including execution speed, resource consumption, and scalability. We want to shed light on the compromises that exist between supporting many platforms and optimizing performance for individual hardware. In addition to assisting with performance-based framework selection, this comparison will add to our knowledge of how to best use JIT compilation to handle the demands of today's computing applications.

II.REVIEW OF LITERATURE

Magni, Alberto et al., (2014) Enabling functional portability across multi-core systems from various suppliers is a primary goal of OpenCL's architecture. It is quite difficult to transfer OpenCL applications' performance since there isn't a single cross-target optimizing compiler. The lack of effective portability is caused by the requirement for programmers to manually adjust apps for each individual device. We demonstrate that thread-coarsening, a compiler change tailored to data-parallel languages, may enhance performance on various GPU devices. Then, we deal with the issue of choosing an optimal value for the coarsening factor parameter, which is determining the optimal number of threads to combine. Our experimental results demonstrate the difficulty of the problem: optimal configurations are hard to discover, and naïve coarsening causes significant performance degradation. We provide a model-based approach that use machine learning to forecast the optimal coarsening factor by analyzing static properties of kernel functions. The model adapts itself to each of the possible structures without human intervention. We test our method on 17 benchmarks using four devices: two graphics processing units (GPUs) from Nvidia and two GPUs from AMD, spanning two generations. We get speedups averaging 1.11X to 1.33X using our method.

Garg, Rahul & Amaral, José. (2010) A new compilation system allows Python programs with heavy numerical calculations to run in a CPU-GPU hybrid environment. By itself, this compiler determines which memory addresses should be sent to the GPU and generates an accurate mapping between the two address spaces. As a result, a common address space in the virtual realm is included into the programming paradigm. The framework is built using jit4GPU, a just-in-time compiler from C to the AMD CAL interface, and unPython, an ahead-of-time compiler from Python/NumPy to C. A number of benchmarks show that the produced GPU code is fifty times quicker than the produced OpenMP code, according to experimental assessment. Optimised CPU BLAS code and GPU performance for single-precision calculations are often comparable.

Ryoo, Shane et al., (2008) Modern many-core CPUs, like the GeForce 8800 GTX, let programmers take use of many layers of parallelism to speed up their apps. On the other hand, since the system is complicated, iterative optimization might result in a local performance maximum. We provide program optimization carving, a method that takes a whole optimization space as input and reduces it to a collection of configurations that potentially include the global maximum. After that, we may test out the other combinations to see which one works best. This method successfully finds a configuration that is close to the best while simultaneously reducing the number of variants to be assessed by up to 98%. In comparison to randomly selecting search parameters, we demonstrate that our method is far better for certain applications.

Greenfield, Perry. (2007) The calibration and interpretation of data obtained from the Hubble Space Telescope (HST) have been aided by Python, according to experts. Initial implementation of this language was as an alternate scripting tool for the Imaging Reduction and processing Facility (IRAF) astronomical processing system. In order to make IRAF tasks perform more reliably and integrate them with the many libraries and tools available in Python, it is used for scripting IRAF applications. Introducing PyRAF, a new scripting environment that streamlines Python development and enables the addition of more robust features than first planned. Other applications, including Numarray in Python, were developed as a result of PyRAF's popularity. There is also the development of PyFITS, a library that adds to matplotlib that can read and write the standard astronomical Flexible Image Transport System (FITS) data format.

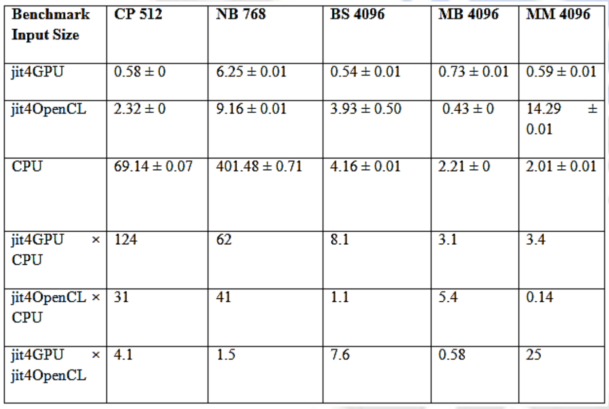
III.EXPERIMENTAL METHODOLOGY

In the studies, a Phenom II X4 925 2.8 GHz CPU was used in combination with a Radeon 5850 GPU that has 1 GB of 1 GHz GDDR5. We utilized the most recent Catalyst drivers, 10.7 with OpenCL 1.1. Python 2.6 and NumPy 1.2 made up the software platform, which ran on a 64-bit Linux 2.6.32 kernel. Using the optimization setting -O2, GCC 4.4 was the C/C++ compiler that was used. All four cores were used throughout the execution of CPU instructions. We generated OpenMP by using the -fopenmp option in gcc. This device is known as the AMD machine. It is also shown that the OpenCL code generated by jit4OpenCL is portable via testing conducted on a desktop PC equipped with an NVidia GTX260 GPU, an Intel Core 2 Duo E5200 (2.5GHz), and 6GB DDR3 RAM. This machine runs Ubuntu 9.10 32-bit with GCC 4.4 and has the NVidia Computing Software Development Kit (SDK) 3.2 Beta installed. The manufacturer of this machine is NVidia. For each tabled result, twenty runs of the benchmark were performed. We provide the mean and the 95% confidence interval here.

IV.RESULTS AND DISCUSSION

Execution timings are shown in seconds in the first rows of Table 1. The lower half of the table displays the relative speedups between each compiler and the CPU, as well as the speedup of jit4GPU over jit4OpenCL. We compare MM to the BLAS version of matrix multiplication, which is greatly optimized. On multi-core CPUs, both JIT compilers produce superior OpenMP code, but jit4GPU code generates far higher performance (with the exception of MB, where jit4OpenCL performance is about twice as good as jit4GPU performance). Compared to jit4GPU, jit4OpenCL has a lot more overhead due to the intermediate translation to OpenCL and the extra compilation step to turn the resulting OpenCL code into native code. The performance comparison is affected by this cost in benchmarks that run for short periods of time. As an example, jit4OpenCL yields a sixfold faster kernel (GPU execution alone) for MB than jit4GPU.

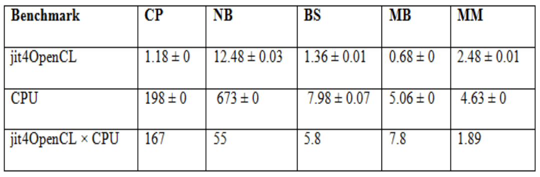
Table 1 Comparison of JIT4GPU, JIT4OpenCL, and CPU Performance on an AMD Machine



Benchmark Input Size	CP 512	NB 768	BS 4096	MB 4096	MM 4096
jit4GPU	0.58 ± 0	6.25 ± 0.01	0.54 ± 0.01	0.73 ± 0.01	0.59 ± 0.01
jit4OpenCL	2.32 ± 0	9.16 ± 0.01	3.93 ± 0.50	0.43 ± 0	14.29 ± 0.01
CPU	69.14 ± 0.07	401.48 ± 0.71	4.16 ± 0.01	2.21 ± 0	2.01 ± 0.01
jit4GPU × CPU	124	62	8.1	3.1	3.4
jit4OpenCL × CPU	31	41	1.1	5.4	0.14
jit4GPU × jit4OpenCL	4.1	1.5	7.6	0.58	25

The primary motivation for re-targeting the compiler to produce OpenCL code is to provide a compilation infrastructure capable of producing code for several platforms. Table 2 displays the results of running the jit4OpenCL function in the aforementioned NVidia GPU. The larger speedups are not unexpected, considering that this is a 2-core CPU (as opposed to the quad-core CPU in the AMD system). Notwithstanding, these results suggest that the produced OpenCL code performs well on many systems.

Table 2 Comparisons Between JIT4OpenCL and CPU Code Performance on an NVIDIA Machine



Benchmark	CP	NB	BS	MB	MM
jit4OpenCL	1.18 ± 0	12.48 ± 0.03	1.36 ± 0.01	0.68 ± 0	2.48 ± 0.01
CPU	198 ± 0	673 ± 0	7.98 ± 0.07	5.06 ± 0	4.63 ± 0
jit4OpenCL × CPU	167	55	5.8	7.8	1.89

CONCLUSION

By comparing JIT4GPU with JIT4OpenCL, we can see how each JIT compilation system addresses current computing demands with its own set of benefits and drawbacks. When it comes to optimizing code for GPU-specific architectures, JIT4GPU really shines. It provides top-notch performance for jobs that need a lot of computational intensity and parallelism. Applications like as deep learning, scientific simulations, and real-time image processing are well-suited to its emphasis on using the distinct characteristics of GPUs. On the other hand, JIT4OpenCL provides a flexible and portable solution that can operate on many types of hardware. In GPU-centric jobs, it may not be as fast as JIT4GPU, but its versatility lets developers use the same code on numerous devices with few changes, which is its strongest suit. Because of this, JIT4OpenCL is very useful in heterogeneous computing settings where interoperability across platforms is paramount.

In the end, the application's needs and the hardware environment dictate which of JIT4GPU and JIT4OpenCL is better. In cases where portability and flexibility are of utmost importance, JIT4OpenCL is the preferable alternative, while JIT4GPU is the obvious choice for developers focusing on GPU performance. The findings of this research will be invaluable in making these judgments, which will improve the efficacy and efficiency of software development for various computer systems.

REFERENCES

1. Çelik, Ahmet & Nie, Pengyu & Rossbach, Christopher & Gligoric, Milos. (2019). Design, implementation, and application of GPU-based Java bytecode interpreters. Proceedings of the ACM on Programming Languages. 3. 1-28. 10.1145/3360603.

2. Hill, N. & Mooney, Scott & Ryklin, Edward & Prusky, Glen. (2019). Shady: a Software Engine for Real-Time Visual Stimulus Manipulation. Journal of Neuroscience Methods. 320. 10.1016/j.jneumeth.2019.03.020.

3. Kim, Gloria & Hayashi, Akihiro & Sarkar, Vivek. (2018). Exploration of Supervised Machine Learning

Techniques for Runtime Selection of CPU vs. GPU Execution in Java Programs. 10.1007/978-3-319-74896-2_7.

4. Fumero, Juan & Steuwer, Michel & Stadler, Lukas & Dubach, Christophe. (2017). Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation. ACM SIGPLAN Notices. 52. 60-73. 10.1145/3140607.3050761.
5. Fumero, Juan & Steuwer, Michel & Stadler, Lukas & Dubach, Christophe. (2017). Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation. 10.1145/3050748.3050761.
6. Zohouri, Hamid Reza & Maruyamay, Naoya & Smith, Aaron & Matsuda, Motohiko & Matsuoka, Satoshi. (2016). Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs. 409-420. 10.1109/SC.2016.34.
7. Alvanos, Michail & Tiotto, Ettore & Amaral, José & Farreras, Montse & Martorell, Xavier. (2016). Using Shared-Data Localization to Reduce the Cost of Inspector-Execution in Unified-Parallel-C Programs. Parallel Computing. 54. 10.1016/j.parco.2016.03.002.
8. Ishizaki, Kazuaki & Hayashi, Akihiro & Koblents, Gita & Sarkar, Vivek. (2015). Compiling and Optimizing Java 8 Programs for GPU Execution. 10.1109/PACT.2015.46.
9. Labschutz, Matthias & Bruckner, Stefan & Gröller, Eduard & Hadwiger, Markus & Rautek, Peter. (2015). JiTTree: A Just-in-Time Compiled Sparse GPU Volume Data Structure. IEEE Transactions on Visualization and Computer Graphics. 22. 1-1. 10.1109/TVCG.2015.2467331.
10. Magni, Alberto & Dubach, Christophe & O'Boyle, Michael. (2014). Automatic optimization of thread-coarsening for graphics processors. Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT. 10.1145/2628071.2628087.
11. Yan, Wanglong & Shi, Xiaohua & Yan, Xin & Wang, Lina. (2013). Computing OpenSURF on OpenCL and general purpose GPU. International Journal of Advanced Robotic Systems. 10. 1. 10.5772/57057.
12. Yan, Wanglong & Shi, Xiaohua & Yan, Xin & Wang, Lina. (2013). Computing OpenSURF on OpenCL and general purpose GPU. International Journal of Advanced Robotic Systems. 10. 1. 10.5772/57057.
13. Zhang, Yao & Sinclair, Mark & Chien, Andrew. (2013). Improving Performance Portability in OpenCL Programs. 136-150. 10.1007/978-3-642-38750-0_11.
14. Ali, Akhtar & Dastgeer, Usman & Kessler, Christoph. (2012). OpenCL for programming shared memory multicore CPUs.
15. Demidov, Denis & Ahnert, Karsten & Rupp, Karl & Gottschling, Peter. (2012). Programming CUDA and OpenCL: A Case Study Using Modern C++ Libraries. SIAM Journal on Scientific Computing. 35. 10.1137/120903683.
16. Garg, Rahul & Amaral, José. (2010). Compiling Python to a hybrid execution environment. International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS. 19-30. 10.1145/1735688.1735695.
17. Ryoo, Shane & Rodrigues, Christopher & Stone, Sam & Stratton, John & Ueng, Sain-Zee & Baghsorkhi, Sara & Hwu, Wen-mei. (2008). Program optimization carving for GPU computing. Journal of Parallel and Distributed Computing. 68(10). 1389-1401. 10.1016/j.jpdc.2008.05.011.
18. Greenfield, Perry. (2007). Reaching for the Stars with Python. Computing in Science & Engineering. 9(3). 38-40. 10.1109/MCSE.2007.62.