A Comparative Study of OpenMP and CUDA parallel architecture using Rodinia Benchmark Suit

Pakruddin B[1] Adeel Ahmed Khan[2] Mohsin Khan[3]
[1,2,3]Department of Computer Science & Engineering
[1,2,3]HKBKCE, Bangalore

Abstract— From the last few decades the microprocessor industry is rapidly shifted to multi-core architectures, much of the research is going on for faster computation in conventional computer systems. Scientific application takes longer time to execute as it contains more mathematical calculation, there is need to parallelise the codes in much efficient way. Many algorithms have been proposed to parallelise the code effectively, but still there are certain limitations that prevent scalable implementations of many well known algorithms. In this research work we have presented a comparative study of CUDA, OpenMP by using two well known scientific applications of Rodinia benchmark suit, a benchmark suite for heterogeneous computing. We are comparing the execution times for serial, OpenMP and CUDA codes of LavaMD and kmeans. We found that CUDA can be utilized effectively to parallelise the application.

Key words: CUDA, OpenMP, Rodinia, LavaMD, kmeans

I. INTRODUCTION

With the advent of Multi-core processors, parallel computing has become easy for certain time consuming applications [1]. The programmers are tending to think the subsequent parallel programming models from different perspectives.

OpenMP (Open Multi Processing) comprises a set of compiler directives and a library, originally targeted at structured parallelism in loops. In OpenMP, programmers interpret sequential C, C++ or FORTRAN code by a set of compiler directives for parallel implementation. Therefore, sequential algorithms are parallelized incrementally, and without major reorganization. OpenMP operates in the fork-join model. In parallel computing, the fork–join model is a way of setting up and executing parallel programs, such that execution branches off in parallel at designated points in the program, to "join" (merge) at a subsequent point and resume sequential execution. Parallel sections may fork recursively until a certain task granularity is reached. Fork–join can be considered a parallel design pattern. Programmers start an OpenMP parallel section in which they can for example annotate a for loop. The iterations will run in parallel and join at the end of the for. This works best for independent iterations, but it is possible to have different forms of synchronization. Reduction operators are built in and programmers are able to define atomic operations and critical sections.

OpenMP parallelism granularity can be controlled manually by adjusting the number of OpenMP threads in combination with a scheduling type, such as static or dynamic, which insures a coarse-grained parallelism. OpenMP supports both task and data parallelism. In OpenMP 3.0[1], programmers can define tasks. Tasks are similar to the sections statement that defines that multiple threads should execute different tasks in parallel, but tasks are more dynamic, high-level and allow nesting. With a task construct, programmers can define a block of code of which the execution can be deferred. In contrast to sections, tasks are not bound to a specific thread, which means that a thread can execute multiple tasks. Tasks can also be untied which means that tasks can be suspended and resumed by a thread. The directive-based approach and coarse-grained parallelism make this model easy for programming existent and new applications on multi-core hardware platforms.

Rodinia benchmark suite is designed for research in heterogeneous parallel computing. In order to help computer science researchers to have an insight in emerging hardware platforms, this benchmark suite are implemented for both multi-core CPUs and GPUs using three different parallel programming models - OpenMP, CUDA, and OpenCL. To address the diversity characteristics, the applications in the benchmark suite are selected carefully to cover different types of application behaviors according to the Berkeley's dwarfs, therefore they exhibit various types of computations, data access patterns, problem partitions, and optimizations. In each application, different input sizes can be specified for various uses. In our experiments, we target the OpenMP part of the benchmark suite: We choose eleven OpenMP applications [2]. We run three datasets of different sizes for each application, and we vary the number of OpenMP threads for each test case, in order to (1) understand the OpenMP scalability; (2) find the best OpenMP performance per application, platform, and dataset.

II. GPU ARCHITECTURE

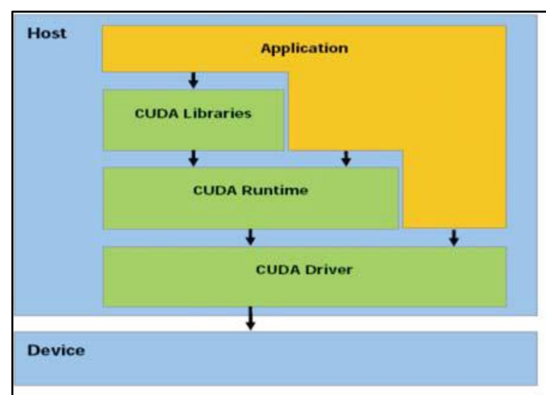

Fig. 2.1: GPU Architecture

Fig 2.1 shows GPU architecture is used for high performance parallel computing and it has special purpose functionality. GPU computing use of GPU together with CPU to parallelize engineering and scientific applications.GPU is a processor with computational resources [5].CUDA (Compute Unified Device Architecture) system specifically designed for GPU programming. GPU is a powerful many-core processor for graphics representation. It contains several Streaming Multiprocessors(SM) which supports concurrent threads. A

All rights reserved by www.ijsrd.com

SM contains multiple Streaming Processors (SP) and low capacity shared memory, which has low data access latency and speedy bandwidth. Global memory is unique memory in GPU that can be shared by both CPU and GPU. Therefore data can be exchanged between CPU and GPU through global memory [5].

III. CUDA ARCHITECTURE

Figure 3.1 shows CUDA application architecture. The applications can be written in high level languages like C, C++ or Java or in device level APIs. NVidia supports DirectX of Microsoft directly via hardware. But OpenCL is supported via CUDA driver and OpenCL drivers.

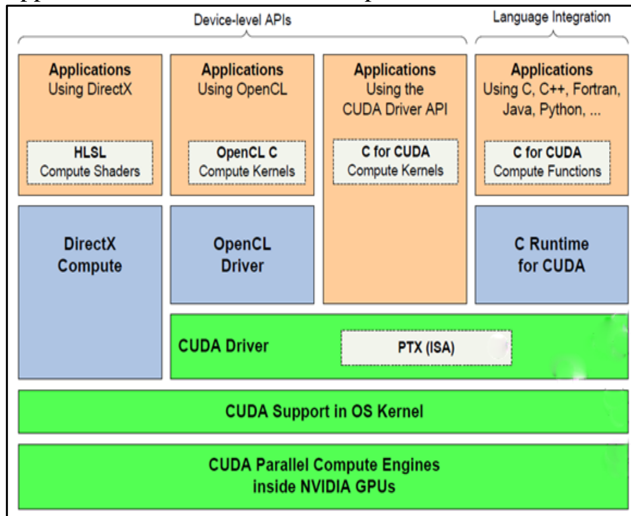

Fig. 3.1: CUDA programming architecture

A. *CUDA Programming Terminologies*

Since GPUs are Streaming Multiprocessors (SMs), they usually execute same instructions but on different data. Single Instruction Multiple Threads (SIMT) architecture followed in most GPUs allow create of large number of threads (65535 and higher in NVidia) that carry out small arithmetic or logical jobs. These threads are organized in blocks and blocks into grids. The threads can be arranged in more than one dimension i.e. x, y and z. Each group of threads is called a block. Such blocks can again be arranged in three dimensions logically. Such an arrangement is called a grid. There can be multiple grids in present GPU architectures. Each grid is mapped to a multi-processor in CUDA. Each multi-processor has many CUDA cores which are small independent execution units with their own registers, local memory and shared memory. Each block of threads in a grid are mapped to a CUDA core. Threads have their own registers and local memory stack of few KBs depending on device's compute capability. Threads in same block have access to a common shared memory through which data sharing is possible. This shared memory is an on chip memory dedicated to individual core. Figure 3.2 shows sample arrangement of threads into two dimensional blocks and grid. Apart from shared memory there are three global memories accessible to all threads across all blocks and grids; they are global heap memory, texture memory and constant memory. The size of these memories is at least 1GB. The access to global memory is slower than shared memory but faster than RAM.

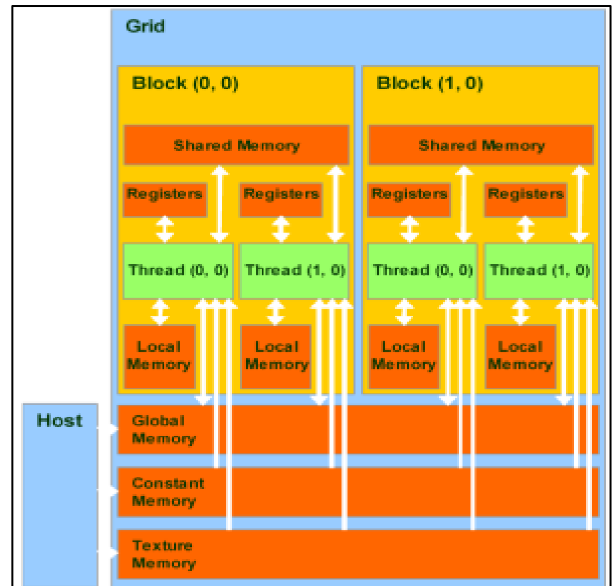

Fig. 3.2: Two dimensional logical arrangement of threads in CUDA

Figure 3.3 explains the CUDA Software Development stack [6]. The device specific code can also be written in CUDA C/C++ which is converted to native code specific to GPU architecture by tools provided in NVidia CUDA SDK. CUDA kernel calls are compiled to Parallel Thread Execution (PTX) intermediaries. PTX is similar to Java byte code or pseudo assembly code. The nvcc compiler provided with CUDA SDK converts CUDA C/C++ code to PTX which are run on device. C runtime handles code that has to run on native machine.

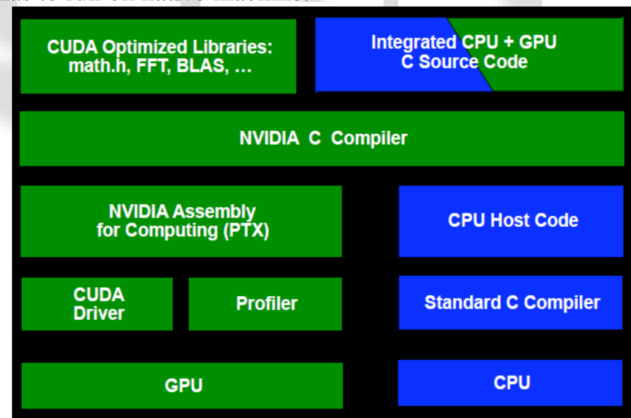

Fig. 3.3: CUDA Software Development Stack

Figure 3.4 shows how CUDA C code is converted into device code [7]. The CUDA application code is compiled using the NVCC compiler. The NVCC compiler generates two types of code namely CPU code and PTX code. The CPU code is compiled by the Visual Studio C++ compiler which is to be run on the host processor. The PTX code is architecture neutral format. It can be either run on GPU or CPU. The above processes happen on the development machine; hence constitute a virtual compilation phase. The second level of compilation happens at the device level. The CUDA device drivers convert these PTX code into device architecture specific code. Thus a simple CUDA program can run on any NVidia GPU with CUDA drivers and interface support.

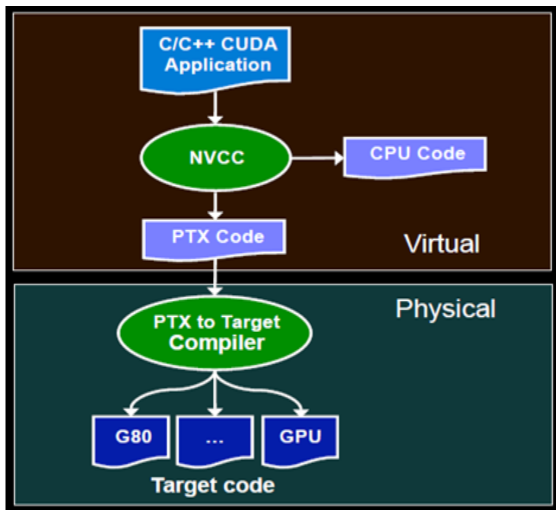

Fig. 3.4: Intermediaries Generated during Compilation of CUDA C Code to target Code

IV. RODINIA BENCHMARK SUIT

The code calculates particle potential and relocation due to mutual forces between particles within a large 3D space. This space is divided into cubes, or large boxes, that are allocated to individual cluster nodes. The large box at each node is further divided into cubes, called boxes. 26 neighbour boxes surround each box (the home box) [2]. Home boxes at the boundaries of the particle space have fewer neighbours. Particles only interact with those other particles that are within a cut-off radius since ones at larger distances exert negligible forces. Thus the box size s chosen so that cut-off radius does not span beyond any neighbour box for any particle in a home box, thus limiting the reference space to a finite number of boxes.

The code [2] was derived from the ddcMD application [5] by rewriting the front end and structuring it for parallelization. The code represents MPI task that runs on a single cluster node. While the details of the code are somewhat different than the original, the code retains the structure of the MPI task in the original code. Since the rest of MPI code is not included here, the application first emulates MPI partitioning of the particle space into boxes. Then, for every particle in the home box, the nested loop processes interactions first with other particles in the home box and then with particles in all neighbour boxes. The processing of each particle consists of a single stage of calculation that is enclosed in the innermost loop. The nested loops in the application were parallelized in such a way that at any point of time GPU warp/wave front accesses adjacent memory locations. The speedup depends on the number of boxes, particles (fixed) and the actual calculation for each particle (fixed). The application is memory bound, and GPU speedup seems to saturate at about 16x when compared to single-core CPU.

V. EXPERIMENTAL SETUP AND RESULTS

We have used LavaMD and LUD codes of Rodinia benchmark suit for comparisons on Dell Server (configuration). The experimental setup is summarised in the table given below, we tracked the timings of LavaMD code with the boxes1D of 10, 20 and 30 respectively for serial, OpenMP and CUDA and tracked the timings. We ran the code using 16 cores of the system and recorded the reading with the boxes1d values as mentioned above for OpenMP. In the similar way with the 16 cores we have recorded the readings for the LUD.

In CUDA we ran the code by utilizing all GPU cores with the same boxes1d respectively. We recorded the timings for 5 times.

VI. RESULTS

The Figure 5.1 is based on the execution time of LavaMD Benchmark suit for Serial, OpenMP and CUDA respectively. The values obtained are recorded by taking the average of 5 readings of each Serial, OpenMP and CUDA for the boxes1D of 10, 20 and 30.

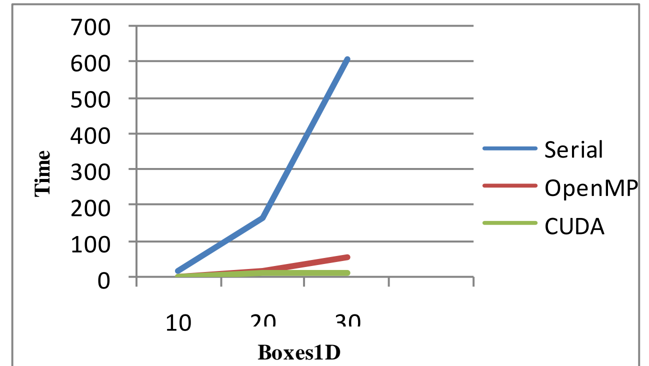

Fig. 5.1: Execution time of LavaMD Benchmark suit

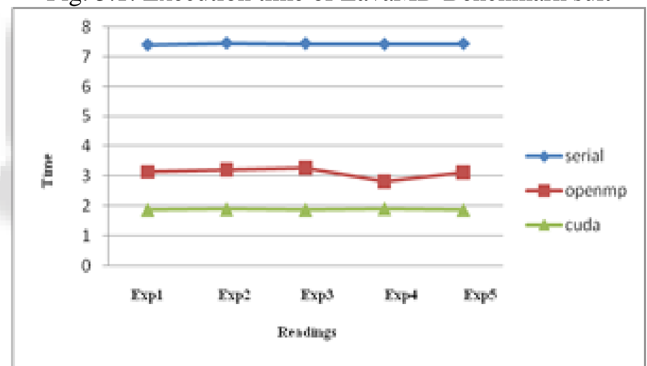

Fig. 5.2: Exection time of Kmeans Algorithm

The Figure 5.2 is based on the execution time of Kmeans algorithm of Rodinia Benchmark suit for Serial, OpenMP and CUDA respectively. The values obtained are recorded by taking the average of 5 readings of each Serial, OpenMP and CUDA. From the graph shown in fig 5.2 it has been observed from the graph that CUDA is more efficient and fastest in comparison with OpenMP even though all the CPU cores utilized effectively.

VII. CONCLUSION AND FUTURE WORK

The Rodinia benchmark suite is designed to provide parallel programs for the study of heterogeneous systems. It provides publicly available implementations of each application for both GPUs and multi-core CPUs, including data sets.In this research work we have observed that the time consumption can be reduced for parallel applications with the CUDA effectively. CUDA architecture provides better way of parallelizing the code comparing with the OpenMP. From our comparison we observed that applications run using CUDA is almost 50% efficient than OpenMP.

All rights reserved by *www.ijsrd.com*

REFERENCES

[1] A. A. Khan, M. Khan and W. Ahmed, "Improved scheduling of virtual machines on cloud with multi-tenancy and resource heterogeneity," 2016 International Conference on Automatic Control and Dynamic Optimization Techniques (ICACDOT), Pune, India, 2016, pp. 815-819.doi:10.1109/ICACDOT.2016.7877700

[2] G. Xie and Y. H. Xiao, "How to Benchmark Supercomputers," 2015 14th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES), Guiyang, 2015, pp. 364-367. doi: 10.1109/DCABES.2015.98.

[3] T. Agarwal and M. Becchi, "Design of a hybrid MPI-CUDA benchmark suite for CPU-GPU clusters," 2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT), Edmonton, AB, 2014, pp. 505-506. doi: 10.1145/2628071.2671423

[4] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee and Kevin Skadron, “Rodinia: A Benchmark Suite for Heterogeneous Computing”, Prsoceedings of the IEEE International symposium on workload characterization(IISWC), 2009.

[5] [2] Abhishek Gupta, Osman Sarood, Laxmikant V Kale, Dejan Milojicic, “Improving HPC Application Performance in Cloud through Dynamic Load Balancing”.

[6] D. Li, W. Ai, Y. Ye and J. Liang, "A efficient algorithm for molecular dynamics simulation on hybrid CPU-GPU computing platforms," 2016 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD), Changsha, 2016, pp. 1357-1363. doi: 10.1109/FSKD.2016.7603376.

[7] Jie Shen, Ana Lucia Varbanescu, “A Detailed Performance Analysis of the OpenMP Rodinia Benchmark”, 2011

[8] Dattatraya Londhe1, Praveen Barapatre2, Nisha Gholap3, Soumitra Das, “A Survey on GPU system considering its performance on different applications”, Proceedings of the Computer Science & Engineering: An International Journal (CSEIJ), Vol. 3, No. 4, August 2013.

[9] P. Zaspel, M. Griebel, “Solving Incompressible Two-Phase Flows on Multi-GPU Clusters”, Preprint submitted to Computers & Fluids January 23, 2012.

[10] Jayshree Ghorpade1, Jitendra Parande2, Madhura Kulkarni3, Amit Bawaskar4, “GPGPU PROCESSING IN CUDA ARCHITECTURE”, Proceedings of the Advanced Computing: An International Journal (ACIJ), Vol.3, No.1,PP 105 – 120, January 2012.

[11] M. Boyer, D. Tarjan, S. T. Acton, and K. Skadron. Accelerating Leukocyte tracking using CUDA: A case study in leveraging manycore

[12] coprocessors. In Proceedings of the 23rd International Parallel and Distributed Processing Symposium, May 2009.

[13] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general purpose applications on graphics processors using CUDA. Journal of Parallel and Distributed Computing, 68(10):1370–1380, 2008.

[14] S. Che, J. Li, J. Lach, and K. Skadron. Accelerating compute intensive applications with GPUs and FPGAs. In Proceedings of the 6th IEEE Symposium on Application Specific Processors, June 2008.

[15] Fan Wu et al., “High Performance Matrix Multiplication on General Purpose Graphics Processing

[16] Units” 2010 International Conference on Computational Intelligence and Software Engineering (CiSE), 978-1-4244-5392- 4/10@2010 IEEE

[17] Study on GPU based Password Recovery for MS office 2003 Document by Xiaojing Zhan, Jingxin Hong. The 7th International Conference on Computer Science and Education (ICCSE 2012) July 14- 17, 2012. 978-1-4673-242-5/12@2012 IEEE

[18] Deguang Le et al., “Parallel AES Algorithm for Fast Data Encryption on GPU” 2010 2nd International Conference on Computer Engineering and Technology (ICCET), 978-1-4244-6349-7/10@ 2010 IEEE

[19] S. Hu, A. Dong and H. Ma, "Implement Improved Loop Level OpenMP Program Based on Parallel Region Reconstruction," 2012 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, Kyoto, 2012, pp. 225-230. doi:10.1109/SNPD.2012.24

All rights reserved by *www.ijsrd.com*