REVIEWS OF LITERATURE
ISSN:2347-2723
IMPACT FACTOR : 3.3754(UIF)
VOLUME - 5 | ISSUE - 5 | DECEMBER - 2017

BALANCING THE LOAD IN HYBRID HIGH PERFORMANCE COMPUTING (HPC) SYSTEMS

Shabnaz fathima
Assistant Professor, International Institute of Business Studies.

ABSTRACT: -

Hybrid computing is emerging as a requirement for power efficient system design. There is no fixed value to denote that the machine is the fastest machine and also there is no standard model to predict the performance. In hybrid architectures, more speed up is obtained by overlapping the computations of available computing elements. Therefore there is a need to make these computing elements work together by balancing the workload. In this paper we are going to study the behavior of load balancing ratio (LDR) for varying workload on CPU+GPU architecture and verify the hypothesis that the performance is increased when the time taken by CPU is approximately equal to time taken by GPU. Methodology will be tested against Rodinia 3.0 benchmarks.

KEYWORDS: *Hybrid computing, load balancing ratio.*

I. INTRODUCTION :

High Performance Computing is a process of aggregating computing power in a way that delivers much higher performance than one could get out of a typical desktop computer or workstation in order to solve large problems in science, engineering or business [1]. Due to the increase in demand for power and speed, HPC will likely interest business of all sizes, particularly for transaction processing and data warehouses [3]. The requirement of hybrid architectures is increasing due to the growing trend of using the multiple computing resources as the sole computing resource. In such cases, the performance is strongly affected by the dependence that exists between the parallel code and architecture. The process of allocating the tasks to the processors is often a problem that requires considerable programmer's effort [2]. This is also a very time consuming process and it also requires the programmer to have in depth knowledge of the target architecture and also the programming knowledge. Therefore there is a tremendous need for the efficient automatic parallelization tools. In hybrid architectures, the CPU is idle most of the time. In the computations performed by the CPU and GPU, CPU performs the serial portion of the task and the GPU performs the parallel portion of the task. The CPU time and the communication time is more. This had a direct impact on the total execution time. It is possible to make the both (CPU and GPU) work together simultaneously by balancing the workload.

This paper is structured as follows: in section 2 introduces some issues that motivated this research and the main goals to achieve. Section 3 shows some of the existing automatic parallelization tools and there drawbacks. Section 4 discusses about NAS and Rodinia

Benchmarks and observations on NAS Benchmarks obtained through manual approach. Related work is shown in section 5 .Section 6 closes with some conclusions and future research to be done.

II. BACKGROUND AND MOTIVATION

Researchers and programmers have been attracted by parallel programming for many years. Parallel programming is a difficult task that may lead to many run time errors. So some tools or methods facilitating the process are required. Parallel programming can be achieved by a programmer manually or it can be achieved automatically by a compiler. Both these approaches has some advantages and some disadvantages. A high level of optimization can be achieved with the help of automatic parallelization compiler .Careful analysis and fine tuning can take this to another level to produce more optimized code. Programming on hybrid systems is obviously dependent on the type of architecture used and the performance obtained is strongly conditioned by the set of machines performing the computations. This means, in most of the cases, the techniques used on homogeneous architectures must be modified to be applied to the systems that have hybrid architectures [5]. In hybrid architectures, since the task is shared is between the available computing elements, there is a need to make these computing elements work together by balancing the workload so that the performance can be increased. The load can be distributed in many ways. One such way is to divide the data into several chunks and obtain performance measures. Based on the measures obtained assign number of chunks to each type of core. The other way to distribute the load is to run the benchmarks in different compute resources using different balancing configurations and use suitable techniques to predict chunk distribution for newer application.

The motivation of this research is, the load distribution behavior varies for different workload. There is no standard model to predict the performance. More speedup is obtained by overlapping the computations of CPU and GPU, was this the best speedup? The load distribution behavior varies for different workload, what is the impact of this on performance? Our objective is to study the behavior of load distribution for varying workload on CPU+GPU architecture. To verify the hypothesis that the total execution time is minimized when time taken by CPU is approximately equal to the time taken by GPU.

III. EXISTING SYSTEMS

Automatic parallelization is the process of converting sequential code into multi-threaded or vectorized (or even both) code in order to utilize multiple processors simultaneously in shared memory multiprocessor machine. The goal of automatic parallelization is to relieve programmers from tedious and error-prone manual parallelization process. Though the quality of automatic parallelization has improved in the past several decades, fully automatic parallelization of sequential programs by compilers remains a grand challenge due to its need for complex program analysis and unknown factors(such as input data range) during compilation. The compiler usually conducts two passes of analysis before actual parallelization. The first pass of the compiler performs a data dependence analysis of the loop to determine whether each iteration of the loop can be executed independently. The second pass attempts to justify the parallelization effort by comparing the theoretical execution time of the code after parallelization to the code's sequential execution time. There are number of automatic parallelization tools available which supports different programming languages like FORTRAN and C such as SUIF compiler, Polaris compiler, Rose compiler, Bones compiler, Cetus compiler etc. In this paper we are going to discuss source to source compilers called Cetus, Bones and GCC.

A. Cetus

Cetus is the source to source compiler infrastructure for transformation of programs. Cetus has the ability to represent the given program in symbolic terms. It was created out of the need for the compiler research environment to support the development of interprocedural analysis and some of the parallelization techniques for C, C++ and java programs. The internal representation (IR) in Cetus is visible to the pass writer through an interface called IR-API. In Cetus, it is easy to write source to source transformations and optimization modification. Portability of the infrastructure to a wide variety of platforms will make Cetus useful to larger community.

Cetus drawbacks: Cetus performs poorly, the deficit in number of parallel loops ranges from 10 to 40 percent. In LU's (NAS benchmark) case, Cetus generates fewer parallel loops because it exploits more outer-level parallelism. Cetus detects important parallel loops or their inner-loop parallelism in CG, IS, SP, and art (NAS benchmarks), but fails to parallelize such loops in EP, equake, and ammp.

CETUS tools are able to generate parallel codes, still more efforts are required to make those codes optimum in terms of performance. These tools should try to skip the loops that have smaller execution time. The performance of parallel code will increase when all the threads are mapped to physical cores. For task level parallelization, the task should have optimal size and less dependencies.

B. GCC compiler

GCC stands for "GNU Compiler Collection". GCC is an integrated distribution of the compilers for several major programming languages. These languages includes C, C++, java, Fortran etc. when GCC is invoked, it normally does preprocessing, compilation, assembly and linking. In GCC the process can be stopped at an intermediate state. The output consists of object files output by the assembler. The operands to GCC program are options and file names. GCC is one of the most robust compiler. It generates highly optimized code for variety of architectures. Its open source distribution and continuous updates make it more attractive. GCC was not designed for source to source transformations. Most of its passes operate on lower-level RTL representation.
Disadvantages: GCC does not provide friendly API for the pass writers. GCC's IR uses an ad-hoc type system, which is not reflected in its implementation language. This makes it difficult to the debuggers to provide meaningful information to the user. It requires extensive modifications to support interprocedural analysis across multiple files.

Some of the transformations were checked in GCC compiler. Loop optimization is the process of increasing execution speed and reducing the overheads associated of loops. It plays an important role in improving the cache performance and making effective use of parallel processing capabilities. Loop optimizations offer a good opportunity to improve the performance of data parallel applications. These optimizations are usually targeted at improving the granularity, load balance and data locality, while minimizing synchronization and other parallel overheads. Loop transformations found in GCC compiler is tabulated below.

TABLE 1: Loop Transformations in GCC

Loop Transformations	Unrolling	Loop Fission	Loop Fusion	Loop Interchange
GCC compiler	Yes	No	No	Yes

C. Bones compiler

Bones is the source to source compiler. It is written in Ruby programming language. It is based on Algorithmic Skeleton Technique. Algorithmic Skeleton Technique is a technique which revolves around a set of parameterisable skeleton implementation. Each skeleton implementation can be seen as template code for specific algorithm class on target architecture. Programmers are able to generate efficient target code by first identifying a number of lines of the code of certain class, followed by invoking the corresponding skeleton implementation. If no skeleton implementation is available for specific class-architecture combination, it can be manually added. Future algorithms of the same class can then be benefit from reuse of skeleton code.
Drawbacks: performance can still be improved. Code readability can be improved. Programmer's effort is required.

IV. BENCHMARKS

I worked on some of the NAS 2.3 Parallel Benchmarks and some Rodinia 3.0 Benchmarks.

A. NAS Parallel Benchmarks

NAS Parallel Benchmarks are the set of benchmarks used to evaluate the performance of supercomputers. They are developed and maintained by NASA Advanced Supercomputing (NAS) division. Some of the NAS benchmarks along with their description is listed below.

• MG(MultiGrid) : Approximate the solution to a three-dimensional discrete Poisson equation using the V-cycle multigrid method

• CG(Congugate Gradient) : Estimate the smallest eigen value of a large sparse symmetric positive-definite matrix using the inverse iteration with the conjugate gradient method as a subroutine for solving systems of linear equations

• FT(Fast Fourier Transform) : Solve a three-dimensional partial differential equation (PDE) using the fast Fourier transform (FFT)

• IS(Integer Sort) : Sort small integers using the bucket sort

• EP(Embarrisingly Parallel) : Generate independent Gaussian random variates using the Marsaglia polar method

• LU(Lower Upper Symmetric Gausseide) : Solve a synthetic system of nonlinear PDEs using three different algorithms involving block tridiagonal, scalar pentadiagonal and symmetric successive over-relaxation (SSOR) solver kernels, respectively

• BT(Block Tridiagonal) : Tests different parallel I/O techniques

• SP(Scalar Pentadiagonal) : Multiple independent systems of non-diagonally dominant, scalar pentadiagonal equations are solved

The graph shows the observation of different bench marks BT, LU, CG, EP, FT, MG and SP for 3 set 1, 4 and 16 threads. The performance is observed in the time taken by each benchmark in seconds. One can clearly see the optimization in performance for each bench marks as the number of threads increases.

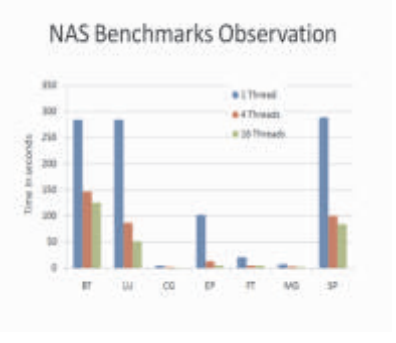

Figure 1 NAS Benchmarks

B. Rodinia Benchmark

The suite consists of four applications and five kernels. They have been parallelized with Open MP for multicore CPUs and with the CUDA API for GPUs. We use various optimization techniques in the applications and take advantage of various on-chip compute resources.

Rodinia Benchmark uses OPEN MP + CUDA architectures. We are trying to mix both the architectures such that more optimizations are obtained and suitable LDR (Load Distribution Ratio) is identified. The generalized algorithm used is given below

Algorithm

[1] Read input sequential code
[2]Analyze the input and identify CPU and GPU variables
[3] Identify parallelizable portions
[4] Calculate block grid and dimension

[5] Allocate memory to GPU and copy the code on GPU
[6] Place suitable parallel code on the kernel and GPU code on main function
[7] Make a call to kernel at appropriate place in main
[8] Return the result to CPU
[9] Free memory on GPU and CPU

The process flow chart is shown below:

The proposed system works as follows: the input variables and for loops present in the input program are identified. For loops identified are divided into two parts. Pragmas are inserted for the first part of for loop and kernels are called for the second part of for loop. The open MP + CUDA program generated is tested for different LDR.

VI. CONCLUSION AND FUTURE WORK

In this paper, we have presented how to distribute the load between CPU-GPU. We also showed benchmarking criteria for evaluation of automatic parallelization tools and compilers. Though the automatic parallelization tools mentioned in this paper are able to generate parallel codes, more efforts are required to make those codes optimum in terms of performance. These tools should try to skip the loops that have smaller execution time. Although there has been a lot of research over the past few decades on automating the task, manual parallelization continues to outperform automatic parallelization tools. This trend can be expected to continue as the performance gap in terms of efficiency and computing resource utilization between automatic and manual parallelization. So hybrid architectures are introduced into the market. The methodology is tested against Rodinia Benchmarks 3.0 for the particular application where we are trying to merge OPEN MP+CUDA architecture so that the load can be distributed to CPU and GPU to reduce the execution time such that performance will be increased.

REFERENCES

J. S. Vetter, R. Glassbrook, J. Dongarra, K. Schwan, B. Loftis, S. Mcnally, J. Meredith,J. Rogers,P. Roth, K. Spafford, And S. Yalamanchili, “Keeneland: Bringing Heterogeneous GPU Computing To The Computational Science Community,” Computing In Science & Engineering, 2011

C. Vömel, S. Tomov, and J. Dongarra, “Divide and Conquer On Hybrid GPU-Accelerated Multicore Systems,” SIAM Journal on Scientific Computing, Vol. 34, No. 2, Pp. C70–C82, 2012

S. Tomov, J. Dongarra, And M. Baboulin, “Towards Dense Linear Algebra For Hybrid GPU Accelerated Manycore Systems,” Parallel Computing, Vol. 36, No. 5, Pp. 232–240, 2010.

D. G. Merrill And A. S. Grimshaw, “Revisiting Sorting For GPGPU Stream Architectures,” In Proceedings Of The 19th International Conference On Parallel Architectures And Compilation Techniques, Pp. 545–546, ACM, 2010.

J. Agulleiro, F. Vazquez, E. Garzon, And J. Fernandez, “Hybrid Computing: CPU+ GPU Coprocessing And Its Application To Tomographic Reconstruction,” Ultramicroscopy, Vol. 115, Pp. 109– 114, 2012.

J. Kurzak, P. Luszczek, M. Faverge, And J. Dongarra, “LU Factorization With Partial Pivoting For A Multicore System With Accelerators”, IEEE Transactions On Parallel And Distributed Systems, Vol. 24, No. 8, August 2013

P. Guo, L. Wang, And P. Chen, “A Performance Modeling And Optimization Analysis Tool For Sparse Matrix-Vector Multiplication On GPUs” IEEE Transactions On Parallel And Distributed Systems, Vol. 25, No. 5, May 2014

A. K. Datta And R. Patel, “ CPU Scheduling For Power/Energy Management On Multicore Processors Using Cache Miss And Context Switch Data” IEEE Transactions On Parallel And Distributed Systems, Vol. 25, No. 5, May 2014

X. Liu, M. Li, S. Li, S. Peng, X. Liao, And X. Lu, “ IMGPU: GPU-Accelerated Influence Maximization In Large-Scale Social Networks”, IEEE Transactions On Parallel And Distributed Systems, Vol. 25, No. 1, January 2014

J. Zhong And B. He, “Medusa: Simplified Graph Processing On GPUs”, IEEE Transactions On Parallel And Distributed Systems, Vol. 25, No. 6, June 2014

H. Chen, J. Sun, L. He, K. Li, And H. Tan, "Bag: Managing GPU As Buffer Cache In Operating Systems", IEEE Transactions On Parallel And Distributed Systems, Vol. 25, No. 6, June 2014

H. Huynh, A. Hagiescu, O. Liang, W. Wong, And R. S. M. Goh, " Mapping Streaming Applications Onto GPU Systems", IEEE Transactions On Parallel And Distributed Systems, Vol. 25, No. 9, September 2014

M. Huang, And D. Andrews, "Modular Design Of Fully Pipelined Reduction Circuits On FPGAs", IEEE Transactions On Parallel And Distributed Systems, Vol. 24, No. 9, September 2013

V. Karakasis, T. Gkountouvas, K. Kourtis, G. Goumas And N. Koziris, "An Extended Compression Format For The Optimization Of Sparse Matrix-Vector Multiplication", IEEE Transactions On Parallel And Distributed Systems, Vol. 24, No. 10, October 2013

H. Zhibin, Z. Mingfa, And X. Limin, "Lvtppp: Live-Time Protected Pseudo partitioning Of Multicore Shared Caches", IEEE Transactions On Parallel Computing.