Centralized Management with Dynamical Location of Web Applications at Network Level Using Hybrid Computing

Amaresh K S[1], Naveen Kumar R[2], Vinay S K[3], Venkatesh P[4]

[1,3,4] *Assistant Professor, PESITM, Shivamogga*

[2]*Net backup Engineer, Tech Aerosoft Pvt Ltd, Bengaluru*

Abstract: **In current network system there is no centralized management model to monitor the collaboration between users and web application.The primary goal of proposed centralized model is to serve user requests by allowing the creation of cutting-edge, powerful, and complex processing that retain simple and easy-to-use interfaces and creating processes with dynamic codification, allocation, and bilateral execution andproviding a athenaeum to create processes to use combinations of CPUs and a GPU for tasks within a single host program process, a process can be created by using available GPUs. Each process can use all of the multiprocessors of each GPU and have its own separate namespace, and have its own separate CUDA context For C and CUDA kernels to process data efficiently using the available resources.**

I. INTRODUCTION

In this digital world, more than 90% of desktop and notebook computers have integrated Graphics Processing Units i.e. GPU's, for better graphics processing. Graphics Processing Unit is not only for graphics applications, even for non-graphics applications too. In the past few years, the graphics programmable processor has evolved into an increasingly convincing computational resource. But GPU sits idle if graphics job queue is empty, which decreases the GPU's efficiency.Many applications with data-parallelism can map data elements to processing threads. For each data element we can only read from the input, execute some operations on it, and write to the output. It is possible to have multiple inputs and multiple outputs. Web servers and web browsers are communicating client-server computer programs for distributing documents and information, generally called web data,the web is becoming increasingly important for businesses and it's the first place people go to when they are researching for information over the Internet.

1.1 CPU

The brain of the computer, processor, central processor or microprocessor, the CPU was first developed at Intel with the help of Ted Hoff in the early 1970's and is short for Central processing Unit. The Computer CPU is responsible for handling all instructions it receives from hardware and software running on the computer.

1.2 GPU

General-Purpose computing on Graphics Processing Units (GPU) has recently emerged as a powerful computing paradigm because of the massive parallelism pro-vided by several hundreds of processing cores originally designed as special hardware for real-time and high-definition 3D graphics, GPUs have evolved into many-core processors to accelerate highly parallel computations. The GPU is designed such that more transistors are devoted to processing cores rather than the sophisticated control hardware, and therefore able to address problems that can be represented as data-parallel computations.

II. MOTIVATION

Highlights

► Hybrid computing allows full exploitation of the power (CPU+GPU) in a computer.
► Proper orchestration of workload is managed by an on-demand strategy.
► Total number of threads running in the system should be limited to the number of CPUs.

Hybrid systems with CPU and GPU are becoming the trend in system design. Although hybrid systems with CPU and GPU are widely used, programmers may not utilize them efficiently since it is challenging for the programmer to split and balance the workload between CPU and GPU.
In current network system there is no centralized management model to monitor the collaboration between users and web application. Due to truancy of centralized management in the mainstream network system, misgivings faced are proper memory ration, data transfer, host best practices and Occupancy filling (maximal usage) of CPUs and GPUs. It can impact development to suffer privation to commit, before development even starts, to a GPU versus a CPU solution for each task step.

Task allocation reflects some of the most important issues in clusters. The goal is to execute all the arriving tasks at minimum cost. This cost could be measured, for instance, in terms of the time needed to compute the tasks, the use of resources, the energy consumed, or any combination of them.

III. ISSUES AND CHALLENGES

As we seek to develop parallel applications, we must understand that such development presents at least three major challenges. I argue that all these challenges are equally present whether one programs a many-core GPU, an MIC, or a multi-core CPU. Unfortunately, there is little compiler technology that can help programmers to meet these challenges today. These challenges are the reasons why compiler-based solutions from vendors will have limited success in creating a scalable parallel code base for many applications. The focus is to clearly understand these challenges before we discuss key techniques that have been proven effective in practice.

The most difficult challenge is that some important problems do not have massively parallel algorithms that exhibit desired behavior. To put this challenge into perspective, in order to achieve exa-FLOPS performance by the end of the decade, we will need to have billion-way parallelism assuming the current trends in clock frequency. At the chip level, developers who hope to enjoy continued performance growth in the next decade will need to use algorithms with at least ten-thousand-way parallelism. Many “scalable” algorithms today fall short when measured with these standards.

There are three levels of difficulties in the parallelism challenge.
First, some problems do not have work-efficient parallel algorithms that exhibit massive parallelism. That is, we simply do not know how to solve these problems with a large number of parallel execution units without significantly increasing the computation complexity.
Second, some problems have known parallel algorithms with ample parallelism but questionable numerical stability. Some of these parallel algorithms do not have the same level of numerical stability as well-known sequential algorithms.
Third, some applications that have parallel algorithms are plagued by catastrophic load imbalance due to highly non-uniform data distribution

The second challenge arises in applications where parallel algorithms do not have sufficient data reuse to achieve good scalability in many core processors.

The third challenge, and perhaps the least daunting one of the three, is that for parallel algorithms with high levels of parallelism and significant data locality, engineering the implementation of parallel algorithms to actually achieve good scalability is still challenging.

Today, a parallel programmer needs to determine layout arrangements of data, allocate memory and temporary storage, arrange pointers, perform index calculation, and orchestrate data movement in order to make use of the on-chip memory resources to support data re-use. The programmer also has to decompose work into tasks, organize threads to perform the tasks, perform thread index calculations to access data in different levels of the memory hierarchy, determine data sharing patterns, and check data bounds. Many parameters of thesearrangements need to be determined for each hardware platform.
The ability to meet these challenges often defines the haves and have-nots in parallel application development. Problems with good solutions to these challenges enjoy excellent scalability and efficiency. Others struggle. In the next several postings in this miniseries, I will discuss key techniques that have been proven effective in developing scalable parallel algorithms for challenging application.

We propose a novel allocation strategy, resources aware scheduler (RAS) to solve these problems.
This section presents the design of RAS and compares RAS results with the existing strategies we discussed in the previous section.

IV. DESIGN OF RAS

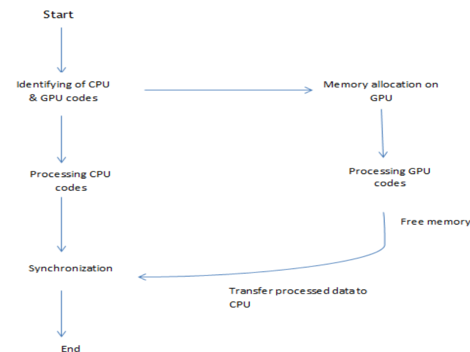


Fig. 1: control flow between CPU and GPU

Fig 1 shows the control flow between cpu and gpu .the complicated part of designing is how to I identify the type of request and resource allocation for the scheduler .we considered the concurrency level 25 as min so that the scheduler can schedule on gpu . the algorithm as follows
Step 1: if S>min goto step 2, goto step 5
Step 2: calculate resource requirement
Step 3:check for required resource are available if so goto step 4,else goto step5
Step 4: allocate req res on gpu and start lunching application
Step 5: if req res not available start processing
On cpu

V. IMPLEMENTATION

We implement RAS in the form of an external library with CUDA and pthread as our proof-of concept system. The

library glues CPU code and GPU code together. We write separate code for CPU and GPU in our implementation. Asynchronous operations and extra threads can be used to handle the devices. Asynchronous operations are more complex than extra threads. So for the ease of programming, we first generate threads using pthread to handle all the devices we can use and assign devices to these threads. We initialize the devices before entering the accelerated region because it takes more time on initialization than computation on the GPU we use (Nvidia Tesla M2075) and this overhead makes allocation inaccurate. The allocation code is a critical section and only one thread can execute this code segment while other threads waiting for the allocation thread to finish its work. We use the main thread of the program as the allocation thread in our implementation. We schedule the task using the strategy described in Section 3 and the scheduler distributes the workload to each thread then threads can do their work. If a GPU-assigned thread receives a task which only needs partial data when doing partial computation, it will just transfer the needed data and the overhead of data movement is included in profiling. Otherwise, it will load all the data into GPU memory before the first step. We use this technique to reduce the data transfer time between GPU and CPU. After all work is done, threads synchronize at the exit point and exit.

VI. EVALUATION

This section evaluates our strategy. The evaluation environment is listed in Table 1. We use four stages, which to measure performance

Table 1.Evaluation environment.

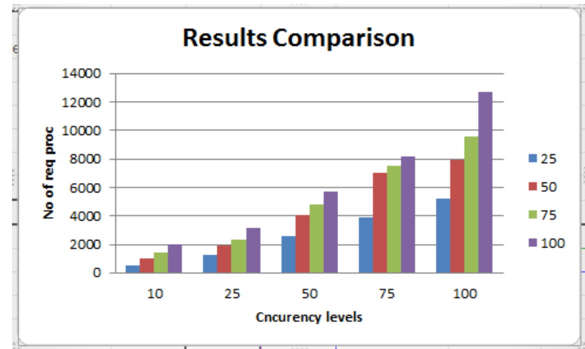
Evaluation environment.

Name	Description
CPU:	Intel(R) Xeon(R) CPU E5 26200@ 2.00GHz
GPU:	Nvidia tesla M2075 @ 1.3 GHz
CPU code compiler :	GCC 4.6.3
GPU code compiler:	NVCC 6.5
Operating system :	CentOS 5.7

We measure the load-balance of execution by calculating the difference in execution time with the following expression:

$$\frac{|time_{CPU} - time_{GPU}|}{\max\{time_{CPU}, time_{GPU}\}}$$

Good performance of Allocation strategy comes from load-balance. As Fig. 4 shows, the differences in no of requests served in certain time with different levels of concurrent and parallel processing. This means that the slower devices waits for 1=3 of the execution time in queue on CPU and the processing power of GPU is much fast compared to CPU.



time	concurrency			
	25	50	75	100
10	510	1004	1409	1976
25	1282	1937	2301	3149
50	2583	4085	4760	5666
75	3855	7008	7506	8203
100	5197	7926	9562	12695

VII. CONCLUSION AND FUTURE WORK

Heterogeneous systems with CPU and GPU are becoming popular. It is beneficial to use all the processors to solve a single task by taking advantages of data-parallelism. Existing data-parallelism allocation strategies do not take advantages of GPU's performance characteristics. It either introduces too much overhead or is not accurate enough.

We propose Allocation strategy, a novel allocation strategy to solve the problem by . Our evaluation result shows that compared with the existing strategies, can achieve up to 42.7% performance improvement on average by accurately estimating the performance of GPU.

REFERENCES

[1]. V.W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A.D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, P. Dubey, Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU, in: Proceedings of the 37th Annual InternationalSymposium on Computer Architecture, ISCA '10, ACM, New York, NY, USA, 2010, pp. 451–460.

[2]. C. Nvidia, CUDA C programming guide 5.0, 2012.

[3]. I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan, Brook for GPUs: stream computing on graphics hardware, in: ACMSIGGRAPH 2004 Papers, SIGGRAPH '04, ACM, New York, NY, USA, 2004, pp. 777–786,

[4]. A. Munshi, TheOpenCL specification version: 1.2, 2011.

[5]. T. Scogland, B. Rountree, W. chunFeng, B. de Supinski, Heterogeneous task scheduling for Accelerated OpenMP, in: 2012 IEEE 26th InternationalParallel Distributed Processing Symposium (IPDPS), 2012, pp. 144–155.

[6]. S. Hong, H. Kim, An integrated GPU power and performance model, in: Proceedings of the 37th Annual International Symposium on Computer

[7]. Architecture, ISCA '10, ACM, New York, NY, USA, 2010, pp. 280–289,