International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 2 Issue 2, February- 2013

Performance Optimization Of Highly Computational Tasks Using CUDA

Mandar S.Karyakarte
Department of Information Technology
Vishwakarma Institute of Information Technology
Pune, INDIA

Harsh Kundnani
Department of Information Technology
Vishwakarma Institute of Information Technology
Pune, INDIA

Abstract**—***The paper analyses the features and generalized optimization methods, on establishing strategies for improving software performance when using the Compute Unified Device Architecture (CUDA) implemented in the latest generation GPUs. The performance for progressively optimizing a matrix multiplication, prime number search for a very large data in CUDA is evaluated. A particular interest was to investigate how well, does CUDA optimizes the speed of computing as compared to a Central Processing Unit(CPU).Also the time required for copying of data from host to device which is from the Central Processing Unit (CPU) to Graphics Processing Unit (GPU) and back when the input is significantly large amount of data.*

Keywords**—** *Compute Unified Device Architecture, Parallel Processing, General Purpose Graphics Processing Unit, Massively Parallel Processing.*

I. Introduction To Parallel Processing

The silicon based processor chips are reaching their physical limits in processing speed, as they are constrained by the speed of electricity, light and certain thermodynamic law. A very viable solution to overcome this limitation is to connect multiple processors working in co-ordination with each other to solve grand challenge problems. Hence high performance computing requires the use of the Massively Parallel Processing (MPP) systems containing thousands of powerful CPUs. Parallel Machines provide a wonderful opportunity for applications with large computational requirements.

II. Significance Of Using Parallel Processing

a. Execution Speed: There is an ever-increasing appetite among some types of computer users for faster and faster machines; this was epitomized in a statement by the late Steve Jobs, founder/CEO of Apple and Pixar. For 30 years, one of the important methods for the improving the performanceof consumer computing devices has been to increase the speed at which theprocessor's clock operated. Starting with the first personal computers of the early1980s, consumer CPUs ran with internal clocks operating around 1MHz. About 30 years later, most desktop processors have clock speeds between 1GHz and 4GHz, nearly 1,000 times faster than the clock on the personal computer of 1980's [1].A major source of speedup is the parallelizing of operations. Parallel operations can be either within-processor, such as pipelining or having several ALUs within a processor, or between processor, in which many processors work on different parts of a problem in parallel.

b. Memory: Parallel processing application often tend to use huge amounts of memory, and in many cases the amount of memory needed is more than the memory that could be allocated on a machine. If many machines are working together, then we can accommodate the large memory needs.

c. Distributed Processing: Apart from the two most famous issues in Computer Science –time (speed) and space (memory capacity), distributed processing demands for high availability and near real time performance to solve the queries. In a distributed database, for instance, parts of the database may be physically located at the widely dispersed sites; successful resolution of queries depends on how fast the server computes and gives response.

III. Different approaches for Parallel Processing

The different approaches for parallel processing are divided into two categories i.e. hardware and software. The talk about hardware part, it can further be sub-categorized into many different types of approaches. Some of these are discussed below.

a. Multicore Computing:

A multi-core processor is a single computing component with two or more independent with actual processing units, which are the units that read and execute program instructions.

b. Massively Parallel Processing:

A Massively Parallel Processor (MPP) is a single computer with many networked processors. MPPs are much more similar to the clusters, except they have specialized interconnect networks. MPPs also are usually larger than clusters, and they havemuch more than 100 processors. In a MPP, "each CPU contains

IJERT

its own memory and copy of the operating system and application. Each subsystem communicates with the others via a high-speed interconnect [2].

c. General Purpose Graphics Processing Unit:

It is the utilization of the Graphics Processing Unit (GPU), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the Central Processing Unit. Any GPU providing a functionally complete set of operations performed on set of arbitrary bits can compute any computable value.

Software approaches:
Concurrent programming languages, libraries, APIs, and parallel programming models have been created for programming parallel computers. Concurrent languages can be defined as one which uses the concept of simultaneously executing process or threads of execution as a means of structuring a program. These can be divided into classes based on the underlying memory architecture—shared memory, distributed memory, or shared distributed memory. Shared memory programming languages communicate by manipulating shared memory variables. Distributed memory uses message passing. POSIX Threads and OpenMP are two of most widely used shared memory APIs, whereas Message Passing Interface is the most widely used message-passing system API. Our focus will be on the GPU programming languages and especially on the Compute Unified Device Architecture (CUDA).
The different graphics processing unit programming languages are:

a. Open Computing Language (OpenCL):

OpenCL is a structure for writing programs that executes across a promiscuous platforms consisting of Central Processing Unit, Graphics Processing Unit and other processors. OpenCL has a language for writing kernels which arefunctions that execute on OpenCL devices, plus application programming interfaces (APIs) that are used to define and then control the platforms.

b. Open Hybrid Multicore Parallel Programming (OpenHMPP):

The main idea was and is still to let developers handle hardware accelerators without the complexity associated with GPU programming. This approach was based on the decisions to enable relationship between an application code and the use of hardware accelerators which is not very strong i.e. hardware accelerators could accelerate the application without use of coding. The OpenHMPP directive-based programming model offers a powerful syntax to efficiently offload computations on hardware accelerators and to optimize data movement.

c. Compute Unified Device Architecture (CUDA):

CUDA is a parallel computing platform and programming model created by NVIDIA and implemented by the graphics processing units (GPUs) that they produce [3]. CUDA gives developers access to the virtual instruction set and memory of the parallel computationalelements in CUDA GPUs. Using CUDA, the NVIDIA GPUs can be used for computation like CPUs. GPUs have a parallel architecture which increases the throughput and it executes many threads simultaneously and unlike CPU it doesn’t execute a single thread very quickly.

IV. What is CUDA and why CUDA?

Graphics Processing Units have been used for a long time solely to accelerate graphics rendering on computers. In order to satisfy the increasing need for improved three-dimensional rendering at a high resolution and a large number of frames per second, the GPU has evolved from a one-purpose specialized architecture to multiple purposes complex architectures, able to do much more than just provide video rendering. The acceleration of a broad class of applications became possible once with the introduction of the NVIDIA Compute Unified Device.

CUDA is NVIDIA’s parallel computing architecture that enables dramatic increases in computing performance by harnessing the power of the GPU.

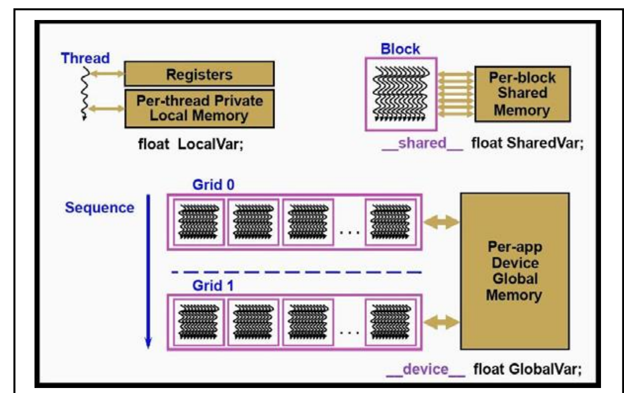

Figure 1

CUDA is a software and hardwarearchitecture that enables the NVIDIA graphics processor to execute programswritten in C, C++, FORTRAN, OpenCL, Direct Compute and other languages. A CUDA program invokes

IJERT

more parallel program kernels. The kernel can process in each set of parallel threads in a parallel manner. These threads can be grouped into thread blocks which consist of more threads and also grids of thread blocks which consist of more thread blocks. The GPU processor launches a kernelprogram on a grid containing parallel thread blocks. Each thread from the block executes a function assigned to it by the kernel and each thread has an uniqueID associated with it, to its private memory within the thread block.

The different threads generated by Compute Unified Device Architecture are mapped to the different graphics processing units' hardware processor; a GPU does the execution of one or more kernel grids whereas a multiprocessor does the execution of one or more thread blocks. The CUDA cores which are contained in the multiprocessor run the threads within blocks [4].

The memory hierarchy of each multiprocessor is dived in such a way that itcontains a set of 32-bit registry with a zone of shared memory, which can be easily accessby each core of the multiprocessor but at the same timehidden from other multi-processors. The number of registry and the size of shared memory greatly dependon the generation of a GPU. Besides shared memory, a multiprocessor further contains two read – only memory caches, one for texture and another one for constants [5]. In order to improve software performance byprogramming in CUDA, programmersneed to optimize the number of naturallyactive threads and balance each thread's resources which are the number of registers and threads per multiprocessor and also the global memorybandwidth and the amount of on-chip memory assigned per thread. Using these techniques,many applications improved their execution time up to 500X in kernel codes

In the NVIDIA CUDA programming model a system is comprised of a traditional CPU (representing the host) and one or more massively data-parallel coprocessors(representing the devices). The CUDA runtime has library functions for managing both the device memory and transfers from the host to the compute devices.

All concurrent threads are based on the samecode even if they may follow different paths of execution because each CUDA device processor supports the Single-Program Multiple Data (SPMD) model and each thread resides in the same global addressspace. The parallel functions called kernels and other data structures which corresponds to the compute devices, are programmed using the standard C language with some extended keywords.A kernel is coded in a way such that it invokes thousands of threads at a time but they synchronize in such a way that they describe the work of a single thread. Threads synchronize themselves through use of built in primitives and share data among each other. The CUDA programming model is designed in such a way that it enables the components of a program, which are to be performed using data parallelism are separated and executed on a specialized massive data parallelism coprocessor. The programmer has to very efficiently divide the resources among threads so that every CUDA core can process a large number of threads and if performed correctly then this flexibility offers a very high degree of control over an application performance and it also has a great impact on optimizing the performance of applications.

IJERT

V. Optimizing Performance Using CUDA:

To optimize the performance of anapplication using CUDA, the program has been written in C language using CUDA libraries which uses the GPU for the maximum calculation purpose and uses CPU for initialization and other purposes. Analgorithm is developed in such a way in CUDA programming model that the work is dividedin small fragments which can be processed by any number of thread blocks, each containing n threads. For optimum performance, it is recommended that the number of thread blocks match the number of processors, although the threads within a block will be executed by more cores within a streaming multiprocessor. Two programs have been developed one using multithreading in C and second using CUDA. Then both the programs are tested for the time complexity i.e. the time taken for complete execution of the program by passing large amount of data to it. To perform these tests CUDA toolkit has been installed on a system with specifications as mentioned below:

Processor: Intel® Core™ i3-3110 CPU multi core with Processor Speed of 2.40GHz, Installed Memory (RAM) of 4 GB, Hard Disk space of 500GB.,Graphics Processing Unit GeFORCE GT 630M with 96 CUDA cores Graphics Clock 800 MHz's.

a. Matrix Multiplication:

Two n x n matrix has been initialized with random values and with the use of multithreading its value has been calculated. The value of n which is considered by me is large and takes considerate amount of time by a CPU to calculate its value. The same data is been tested with the CUDA and the results are compared. I have just considered the time complexity i.e. the time taken to move the data from CPU to GPU then time taken to process the data and then time taken to copy the data back to the CPU from GPU.

In the CUDA program the kernel part separates the functions performed by CPU and functions performed by GPU. A large number of threads are created depending on number of cores of the GPU, but in my program we have limited the program to around 200 threads for each block and we have considered 32 blocks. Each thread created has a unique identity and it performs an individual number multiplication.

The results generated are following as shown in the table below and a graphical representation is also shown along with it to give a clear idea about the

time complexity of the two codes performing the same computation on two different platforms.

Matrix	CPU (8 threads)	GPU
1000 x 1000	4.7022925s	0.2649s
2000 x 2000	40.101659s	2.0442s
4000 x 4000	367.137450s	16.0987s
6000 x 6000	1406.453598s	55.7108s
8000 x 8000	4432.138453s	129.0358s

Table 1: Matrix Multiplication

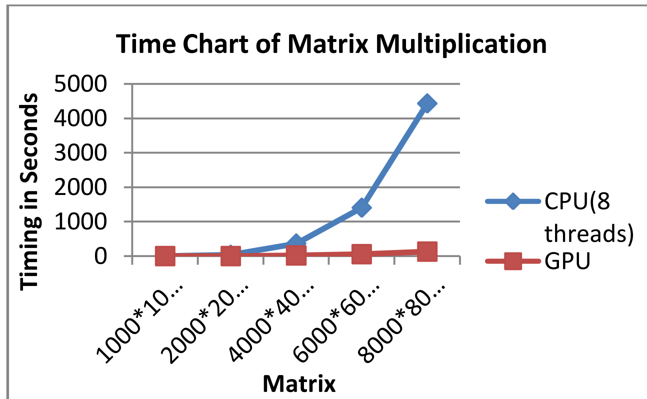

Figure 2: Graph representing the time variations

The graph shows clearly the difference in time taken to compute the matrix multiplication on CPU and GPU and how fast CUDA can perform computation.
Another table shows the time taken to copy the contents from Host to Device i.e. CPU to GPU for the 1000 x 1000 Matrix. The things which can be noticed from the given table are that most time is spent in copying the data from the CPU memory to the GPU memory for computation and again copying the results back from GPU memory to CPU memory for printing it. The important strategy that should be applied while computing using CUDA is too copy less data between host and device and perform more computation on the data that is copied. The important thing while working with CUDA is not the latency but the throughput which is the amount of transactions per unit time.

Name	Start Time	Duration	Size	Throughput
Memcpy HtoD	68.655ms	985.603 micro seconds	3.815MB	3.78GB/s
Memcpy DtoH	500.109 ms	1.62 ms	3.815MB	2.29GB/s

Table 2: Time Variation in copying Data from Host to Device and Device to Host for Matrix Multiplication Problem

b. Prime Number Search

Searching for prime numbers between a given ranges has been performed as the next program to test that CUDA optimizes the speed of computation as compared to the CPU. A large range of 8000 numbers have been considered as input for the program and the same has been written with multithreading in C to run on CPU and using CUDA libraries to run on GPU. In this program the time complexity i.e. the time taken to move the data from CPU to GPU then time taken to process the data and then time taken to copy the data back to the CPU from GPU is considered.
In the CPU version of this program the range is divided according to the slices and this program also uses 8 threads. Each thread is assigned a task to compute the different slices of numbers depending on the range entered. When each thread has finished searching for prime numbers within its slices it stops working.
In the CUDA version of the program also the range is divided and accordingly the number of threads are passed which checks for itself if they are prime or not and if they are prime the numbers are stored in the output array.
Tables below and figure 3 shows the results.

Range of Numbers	CPU (8 threads)	GPU
1000	63.005960s	0.0009s
2000	125.011815s	0.002s
4000	250.022927s	0.0067s
6000	375.031079s	0.0144s
8000	500.043820s	0.0259s

Table 3: Prime Number Search

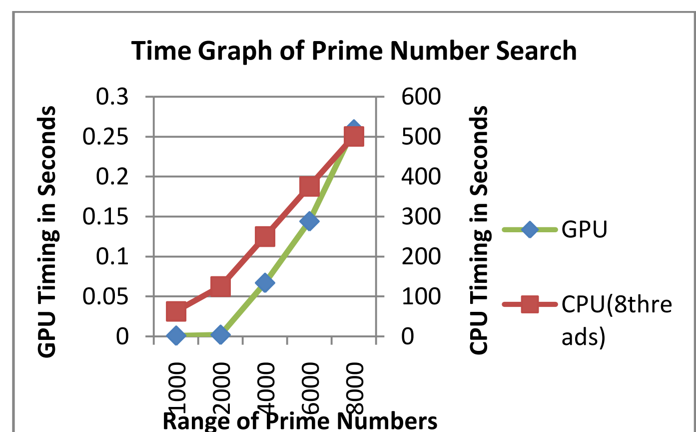

Figure 3: Graph representing the time variations

IJERT

The graph here shows the time variation of the GPU which maximum reaches till 0.25s whereas the time variation of the CPU which reaches maximum till 500s. It clearly depicts the difference in time taken to compute the Prime number search on CPU and GPU and how fast CUDA can perform computation.
Another table shows the time taken to copy the 1000 numbers from Host to Device. The things which can be noticed from the given table are same as that from the above Matrix Multiplication Memory table.

Name	Start Time	Duration	Size	Throughput
Memcpy HtoD	65.398ms	8.288 micro seconds	31.26KB	3.6GB/s
Memcpy D to H	91.28ms	7.073 micro seconds	31.25KB	4.21GB/s

Table 4: Time Variation in copying Data from Host to Device and Device to Host for Prime Number Search Problem

VI. Conclusion

In this paper we have analysed several aspects regarding the improvement of performance of applications written using CUDA. We addressed two problems Matrix Multiplication and Prime Number Search using the both multithreading in C and using CUDA and analysed various time complexities based on the amount of data given to each problem at a given time. We also addressed the problem of time consumption during the copying of data from host to device and device to host. The time taken by the GPU for computation can be optimized much more by defining the correct number of threads and maintain the ratio of creation of thread to work done by each thread.

The computation results can vary to a great deal if a much higher core CPU processor and GPU processor is used for computation of the same problem and can result in a much more higher results. The possibilities of CUDA are endless and with more and more research many new applications are developed for scientific and domestic uses.

IJERT

VII. References

1. CUDA by Example: An Introduction to General-Purpose GPU Programming. http://books.google.co.in/books?id=49OmnOmTEtQC

2. NVIDIA CUDA http://www.nvidia.com/object/cuda_home_new.html

3. Programming on Parallel Machines written by Norm Matloff heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBook.pdf

4. Improving Software Performance in the Compute Unified Device Architecture by Alexander PIRJAN

5. http://www.readperiodicals.com/201010/2281802401.html