JCSE International Journal of Computer Sciences and Engineering Open Access
Research Paper Volume-2, Issue-12 E-ISSN: 2347-2693

Comparison of Open MP and CUDA

Fazlul Kader Murshed Nawaz [1], Arnab Chattopadhyay[1],
Kirthan G J[1], Girish D Mane[1], Rohith N Savanth[1]

[1]*School of Computer Science and Engineering, VIT University Vellore, INDIA,*
www.ijcseonline.org

Received: Dec/04/2014 Revised: Dec/10/2014 Accepted: Dec/23/2014 Published: Dec/31/ 2014

***Abstract*—** In this paper we will study the architectures of both Open Mp and CUDA architectures and make some comparisons among both in terms of basic program execution time. We will use a basic serial version of Matrix Multiplication Algorithm and modify it in terms of CUDA and Open MP and study comparatively.

***Keywords*—**OpenMP; CUDA; Matrix Multiplication; GPU; CPU;

I. Introduction

This project we will use the NVIDIA CUDA GPU programming environment to explore data parallel hardware and programming environments. The goals include exploring the space of parallel algorithms, understanding how the data-parallel hardware scales performance with more resources, and utilizing the data-parallel programming model.

At the start of multicore CPUs and GPUs the processor chips have become parallel systems. But speed of the program will be increased if software exploits parallelism provided by the underlying multiprocessor architecture. Hence there is a big need to design and develop the software so that it uses multithreading, each thread running concurrently on a processor, potentially increasing the speed of the program dramatically. To develop such a scalable parallel applications, a parallel programming model is required that supports parallel multicore programming environment. NVIDIA's graphics processing units (GPUs) are very powerful and highly parallel. GPUs have hundreds of processor cores and thousands of threads running concurrently on these cores, thus because of intensive computing power they are much faster than the CPU.

II. DATA LEVEL PARALLELISM

Data parallelism is a form of parallelization of computing across multiple processors in parallel computing environments. It mainly focuses on distributing the data across different parallel computing nodes. In a multiprocessor system executing a single set of instructions (SIMD), data parallelism is achieved when each processor performs the same task on different pieces of distributed data. In some situations, a single execution thread controls operations on all pieces of data. In others, different threads control the operation, but they execute the same code.

For instance, consider a 2-processor system (CPUs A and B) in a parallel environment, and one wish to do a task on some data d. It is possible to tell CPU A to do that task on one part of d and CPU B on another part simultaneously, thereby reducing the duration of the execution. The data can be assigned using conditional statements. As a specific example, consider adding two matrices. In a data parallel implementation, CPU A could add all elements from the top half of the matrices, while CPU B could add all elements from the bottom half of the matrices. Since the two processors work in parallel, the job of performing matrix addition would take one half the time of performing the same operation in serial using one CPU alone.Data parallelism emphasizes the distributed (parallelized) nature of the data.

Corresponding Author: *F . K . M . Nawaz, fkmnawaz@gmail.com*

III. Application

At start, they were used for graphics purposes only. But now GPUs are becoming more and more popular for a variety of general-purpose, non-graphical applications too. For example they are used in the fields of computational chemistry, sparse matrix solvers, physics models, sorting, and searching etc. For sorting algorithms there are many modifications using GPUs and ISSD (Improved Sorting considering Special Distributions). The programs designed for GPGPU (General Purpose GPU) run on the multi processors using many threads concurrently. As a result, these programs are extremely fast

IV. Cuda

CUDA stands for Compute Unified Device Architecture. It is a parallel programming paradigm released in 2007 by NVIDIA. It is used to develop software for graphics processors and is used to develop a variety of general purpose applications for GPUs that are highly parallel in nature and run on hundreds of GPU's processor cores. It aims at improving the efficiency of parallel algorithm implementation and facilitating programmers in harnessing the power of the Modern NVIDIA GPU that consists of hundreds of scalar processing elements per chip.

CUDA uses a language that is very similar to C language and has a high learning curve. It has some extensions to that language to use the GPU-specific features that include new API calls, and some new type qualifiers that apply to functions and variables. CUDA has some specific functions, called kernels. A kernel can be a function or a full program invoked by the CPU. It is executed N number of times in parallel on GPU by using N number of threads. CUDA also provides shared memory and synchronization among threads.

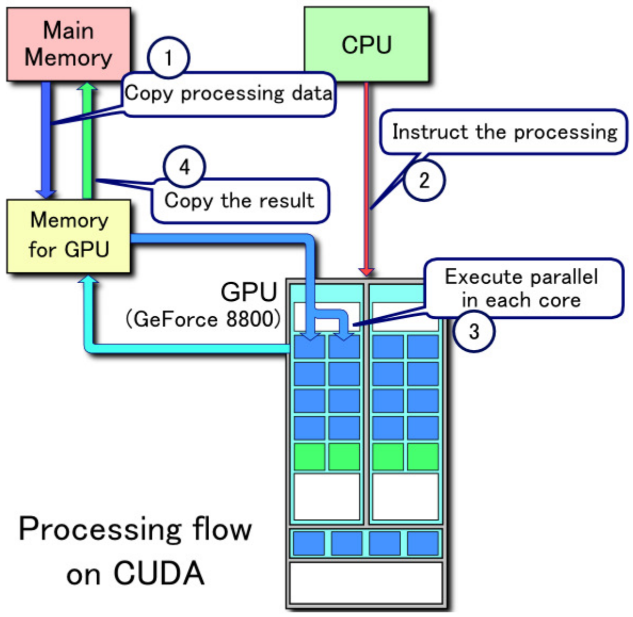

Figure 1

CUDA has several advantages over traditional general-purpose computation on GPUs (GPGPU) using graphics APIs:

- Scattered reads – code can read from arbitrary addresses in memory
- Unified virtual memory (CUDA 4.0 and above)
- Unified memory (CUDA 6.0 and above)
- Shared memory – CUDA exposes a fast shared memory region (up to 48 KB per multi-processor) that can be shared amongst threads. This can be used as a user-managed cache, enabling higher bandwidth than is possible using texture lookups.[13]
- Faster downloads and readbacks to and from the GPU
- Full support for integer and bitwise operations, including integer texture lookups

V. GPU VS CPU

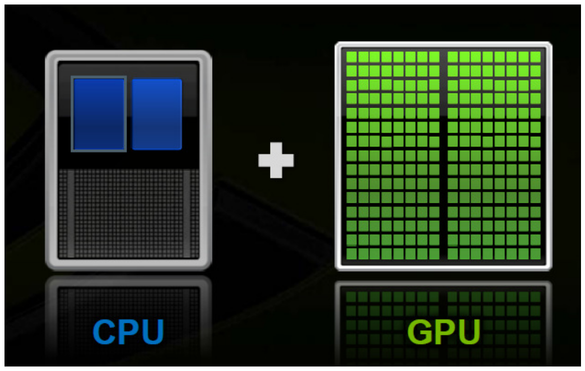

Figure 2

CPU

- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution
- CPU architecture must minimize latency within each thread

GPU

- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation
- GPU architecture hides latency with computation from other thread warps GPU

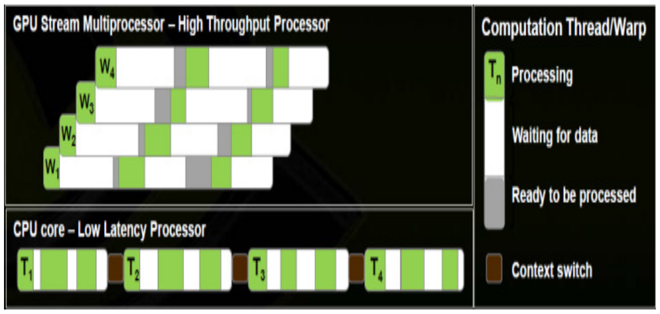

Figure 3

VI. PROCESSING FLOW

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory PCI Bus

VII. GPU ARCHITECTURE

There are two Main Components:

❖ **Global memory**

- Analogous to RAM in a CPU server
- Accessible by both GPU and CPU

- Currently up to 6 GB Bandwidth currently up to 150 GB/s for Quadro and Tesla products
- ECC on/off option for Quadro and Tesla products

❖ **Streaming Multiprocessors (SMs)**

- Perform the actual computations
- Each SM has its own Control units, registers, execution pipelines, caches

VIII. CUDA CORE

Floating point & Integer unit IEEE 754-2008 floating-point standard Fused multiply-add (FMA) instruction for both single and double precision Logic unit Move, compare unit Branch unit Register.

IX. OPEN MP

ARCHITECTURE

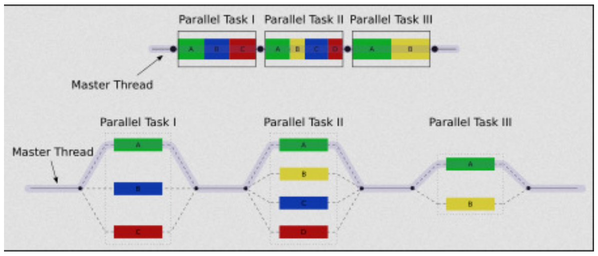

Figure 4

OpenMP is an implementation of multithreading, a method of parallelizing whereby a master thread (a series of instructions executed consecutively) forks a specified number of slave threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

The section of code that is meant to run in parallel is marked accordingly, with a preprocessor directive that will cause the threads to form before the section is executed.[4] Each thread has an id attached to it which can be obtained using a function (called omp_get_thread_num()). The thread id is an integer, and the master thread has an id of 0. After the execution of the parallelized code, the threads join back into the master thread, which continues onward to the end of the program.

By default, each thread executes the parallelized section of code independently. Work-sharing constructs can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both task parallelism and data parallelism can be achieved using OpenMP in this way.

The runtime environment allocates threads to processors depending on usage, machine load and other factors. The runtime environment can assign the number of threads based on environment variables, or the code can do so using functions. The OpenMP functions are included in a header file labeled omp.h in C/C++.

X. PERFORMANCE EXPECTATIONS

One might expect to get an N times speedup when running a program parallelized using OpenMP on a N processor platform. However, this seldom occurs for these reasons:

- ➢ When a dependency exists, a process must wait until the data it depends on is computed.
- ➢ When multiple processes share a non-parallel proof resource (like a file to write in), their requests are executed sequentially. Therefore each thread must wait until the other thread releases the resource.
- ➢ A large part of the program may not be parallelized by OpenMP, which means that the theoretical upper limit of speedup is limited according to Amdahl's law.
- ➢ N processors in a symmetric multiprocessing (SMP) may have N times the computation power, but the memory bandwidth usually does not scale up N times. Quite often, the original memory path is shared by multiple processors and performance degradation may be observed when they compete for the shared memory bandwidth.

Many other common problems affecting the final speedup in parallel computing also apply to OpenMP, like load balancing and synchronization overhead.

XI. MATRIX MULTIPLICATION

Input Matrix: a[1..n] b[1..n]

❖ **Serial Version**

```
for (i = 0; i <n; i++)
{
for (j = 0; j < n; j++)
{
sum = 0;
for (k = 0; k < n; k++)
{
sum = sum + a[i][k] * b[k][j];
}
c[i][j] = sum;
}
```

❖ **Open MP Version :**

```
#pragma omp parallel for schedule(dynamic,chunk)
private(i,j) shared(a,b,c)
for (i = 0; i <lines; i++)
{
for (j = 0; j <columns; j++)
{
        int sum = 0;
        #pragma omp parallel for reduction(+:sum) private(k)
        for (k = 0; k <N; k++)
        {
```

Speed Comparison	Execution Time			Speed UP	
Size of the Matrix	CUDA execution time (in ms)	OpenMP execution time (in ms)	Serial execution time (in ms)	Speedup of CUDA (w.r.t. serial)	Speedup of OpenMP (w.r.t. serial)
50x50	1.0078	2.238	0.701	0.696	0.313
100x100	1.7397	3.738	5.727	3.291	1.5321
150x150	1.6106	8.490	19.056	11.83	2.2445
200x200	1.8689	21.193	43.455	23.26	2.05
250x250	2.0156	36.040	84.915	42.24	2.356
300x300	2.6751	59.035	130.61	48.91	2.2124
350x350	2.6807	102.170	186.890	69.73	1.82
400x400	2.6903	132.550	267.891	99.58	2.0215
450x450	2.7096	195.707	345.521	127.91	1.7655
500x500	3.0462	308.454	488.740	160.45	1.58
550x550	3.1274	430.357	645.376	206.38	1.4996

```
        sum = sum + a[i][k] * b[k][j];
        }
        c[i][j] = sum;
        }
        }
```

❖ **CUDA Version:**

```
        __global__ void gpuMM(float *a, float *b, float *c, int n)

        {
        int row = blockIdx.y*blockDim.y + threadIdx.y;
        int col = blockIdx.x*blockDim.x + threadIdx.x;

        float sum = 0.f;
        for (int j = 0; j < n; ++j)
        sum += a[row*n+j]*b[j*n+col];

        c[row*n+col] = sum;

        }
```

XII. Results and Analysis

We implemented the serial and parallel version of matrix multiplication in OpenMP and CUDA architecture, measured the execution times, speed ups and studied it detail the graphs.

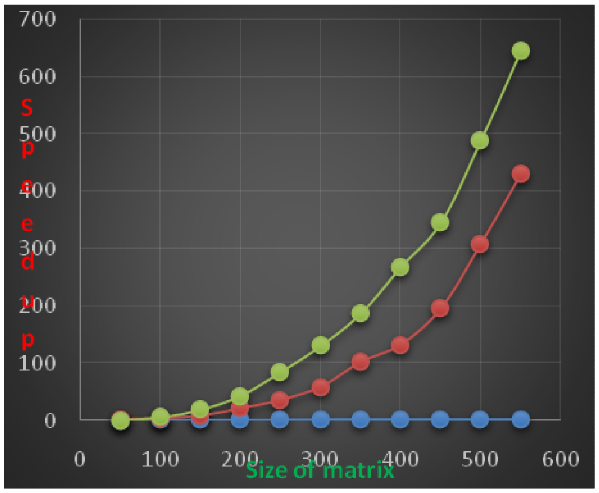

Figure 5
Blue - CUDA, Red - Open MP, Yellow: Serial

XIII. Conclusion

After the comparative study of open MP, CUDA and serial version of matrix multiplication algorithm. It is clear that CUDA execution time is lot faster than the open MP system and irrespective of the size of the matrix CUDA performs a lot better than its counterparts.

Acknowledgment

Many thanks go to Prof. Harsh Arora (VIT UNIVERSITY, Vellore, SCSE) for his sincere guidance and many joint discussions of CPU and GPU and its associated speed related issues. Also we want to thank him for his valuable lectures on Advanced Computer Architecture by help of which our project got succeeded. .

References

[1] Kirk David B, Hwu Wen Mei. W, "programming massively parallel processors", NVIDIA Elsevier, ISBN: 978-0-12-381472-2, Vol.10, **2010**, pp.**59-120**.
[2] Sanders Jason, Kandrot Edward, "Cuda By Example", Addison – Wesley, ISBN-13: 978-0-13-138768-3, **2011,** pp**.37-57.**
[3] Niraj R Chauhan and Mayur S. Burange, "Multicore Heterogeneous Computing with OpenACC", International Journal of Computer Science and Engineering E-ISSN: 2347-2693, Vol.2, Issue-3, **2014**, pp**.92-97**.