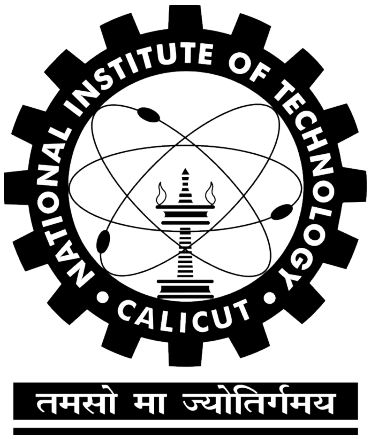
Advances in GPU based computing

A
Seminar Report

by

Harith R
B140276CS



Department of Computer Science and Engineering
National Institute of Technology, Calicut
Monsoon 2017

Contents

List of Figures

1 Introduction

A General-Purpose Graphics Processing Unit (commonly termed as GPGPU) is a graphics processing unit (GPU) used to perform computation in applications traditionally handled by the central processing unit (CPU). Over the past few decades, GPUs have developed from special-purpose graphics accelerators to general-purpose massively parallel co-processors.
Algorithms well-suited to GPGPU implementation exhibit two properties: they are data parallel and throughput intensive. They are considerably faster than CPUs in executing such algorithms.
In this seminar, I discuss the Amdahl's law of parallelization, how GPUs perform compared to CPUs when running Machine Learning Algorithms[1][2] and a Sparse Graph traversal algorithm called HBFT[3].

2 Motivation

In recent years, with the rise of big data processing, cryptocurrency mining and artificial intelligence [3], the amount of raw data available for proessing is huge. Because of this there is a rising need for parallelization of algorithms that work on data.
Since GPUs have a large number of execution units (cores), algorithms that are parallelizable will perform much better on them. Hence, GPUs have gained increased traction in high performance computing (HPC) running certain algorithms 10-100 times faster than traditional CPUs.
Today, when it comes to machine learning and artificial intelligence using large datasets and dense graphs, GPUs make the backbone of processing, with manufacturers like NVIDIA and AMD making huge leaps in GPU architecture and performance.

3 Advances in GPU based computing

3.1 GPU vs CPU

CPUs are developed and optimized for sequential serial processing, i.e. they focus more on executing one job as fast as possible. While GPUs are designed to process all of the data simultaneously, effectively juggling hundreds if not thousands of jobs all at once, even though those jobs will be completed slower than CPUs.
The main difference betwen GPUs and CPUs are that GPUs devote proportionally more transistors to arithmetic logic units and less to caches and flow control in comparison to CPUs. GPUs also typically have higher memory bandwidth and simpler instruction sets compared to CPUs.

Theoretically, due to the highly parallel nature of GPUs, they are capable of performance which is magnitudes greater than that of CPUs.
Amdahl's law[4] gives the theoretical speedup when multiple cores are used as:

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

where S_{latency} is the theoretical speedup of the execution of the whole task, s is the speedup of the part of the task that benefits from improved system resources, p is the proportion of execution time that the part benefiting from improved resources originally occupied.
In an ideal case, s can be considered equal to the number of execution units (cores) available and p as the fraction of the program that is parallelizable. If p=1 i.e. the entire program can be parallelized, then $S_{\text{latency}} = s =$ number of cores
But practically, due to the following reasons GPUs can't be used to completely replace CPUs :

1. Task-parallel computations where one executes different instructions on the same or different data cannot utilize the shared flow control hardware on a GPU and often end up running sequentially.

2. CPUs, due to their higher clock speeds, have the edge in real-time applications where low latency is a priority.

3. Programmers need to learn frameworks like OpenCL and NVIDIA CUDA to make algorithms run on GPUs. Such frameworks are still evolving and not perfect for all algorithms.

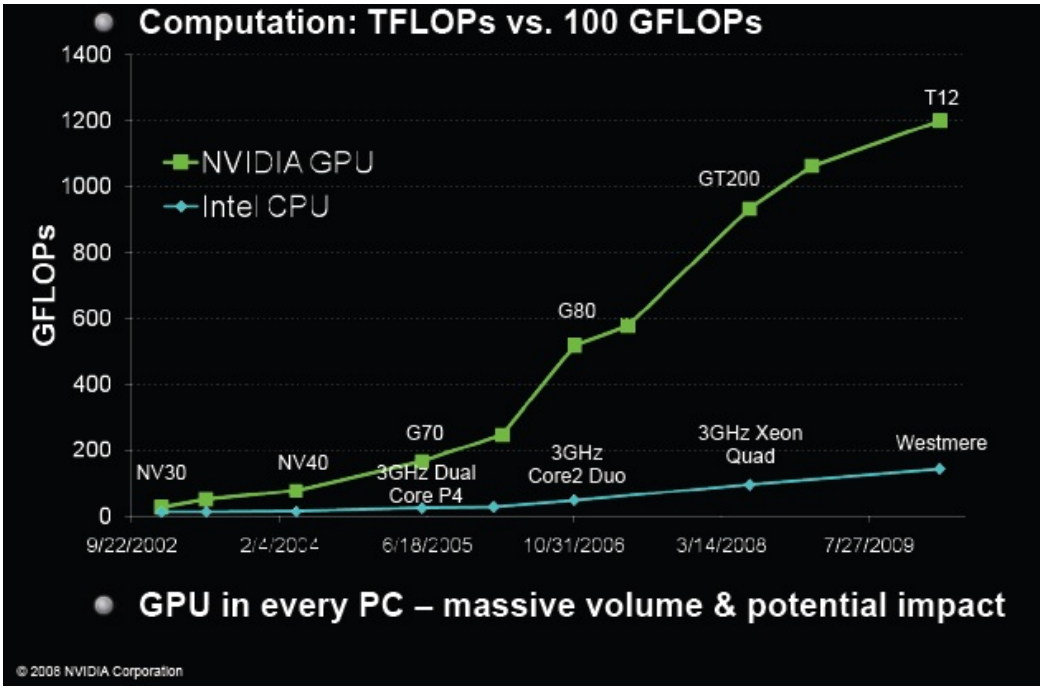


Figure 1: Performance Comparison in terms of floating point operations per second ($FLOPS$)
[5]

The challenge in executing algorithms on GPUs is to map the serial well-known solution to an image space involving vertices, textures, shaders etc. There are a variety of computational resources available on the GPU:

- Programmable processors – vertex, primitive, fragment and mainly compute pipelines allow programmer to perform kernel on streams of data

- Rasterizer – creates fragments and interpolates per-vertex constants such as texture coordinates and color
- Texture unit – read-only memory interface
- Framebuffer – write-only memory interface

The most common form for a stream to take in GPGPU is a 2D grid because this fits naturally with the rendering model built into GPUs. Many computations naturally map into grids: matrix algebra, image processing, physically based simulation, and so on. Since textures are used as memory, texture lookups are then used as memory reads. Certain operations can be done automatically by the GPU because of this.
As of 2016, OpenCL is the dominant open general-purpose GPU computing language, and is an open standard defined by the Khronos Group. It's based on C++. OpenCL provides a cross-platform GPGPU platform that additionally supports data parallel compute on CPUs. OpenCL is actively supported on Intel, AMD, Nvidia, and ARM platforms.
Nvidia CUDA on the other hand is a dominant proprietary framework built by Nvidia for Nvidia GPUs. It is also based on C++, but has additional capabilities that OpenCL doesn't, such as multi-GPU usage.

Most of today's high performance computing systems work on huge datasets and rely on parallelism for the best performance. Hence, GPU implementations of well known algorithms are rising to popularity.

3.2 Machine Learning Algorithms

Machine Learning (ML) is a very powerful domain in the field of computer science. Training ML models using huge datasets may take from weeks to months even on supercomputers.
GPUs helps in reducing training time from days or even weeks to just hours. Parallel implementations of two popular machine learning algorithms , kernel k-means, k-nearest neighbours [1] and Backpropogation[2] using GPUs have been made.

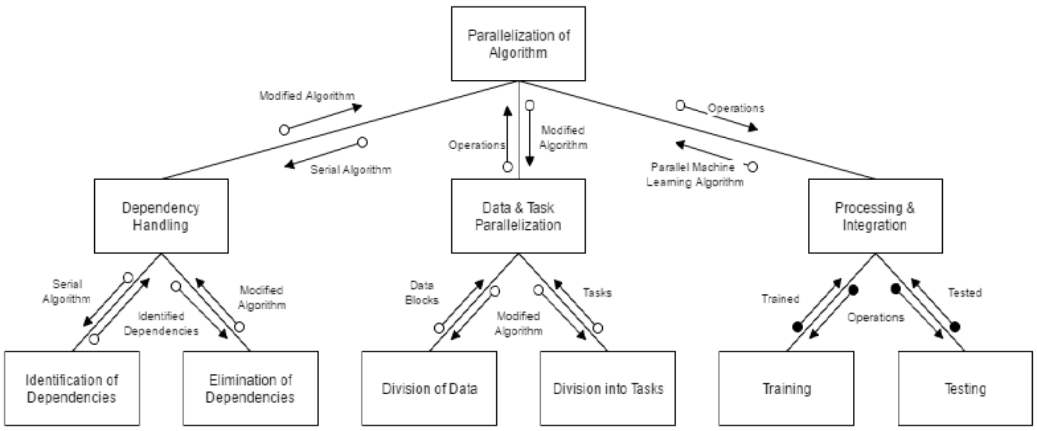


Figure 2: Structure of Parallelizing model
[1]

Dependency Handling Module: Input is the serial machine learning algorithm. Output is the modified algorithm after identification and elimination of dependencies.
Data and Task Parallelization Module : Input is the output of Dependency Handling Module. Output is a set of operations that are identified as data intensive and task intensive.
Processing and Integration module: Input is the output of Data and Task Parallelization Module. Output is the final CUDA executable machine learning program
The following points describe the software and hardware used for implementing the algorithms, and for running those serial and parallel implementations, respectively: • User machine: Dell Inspiron 5521 with an Intel i7 core processor, 8GB RAM
• Mainframe GPU Server: Connoi 2X4GPU Workstation with 2 x Intel Xenon Processor, 32GB RAM having an NVIDIA Tesla K20 GPU with 2496 CUDA cores.

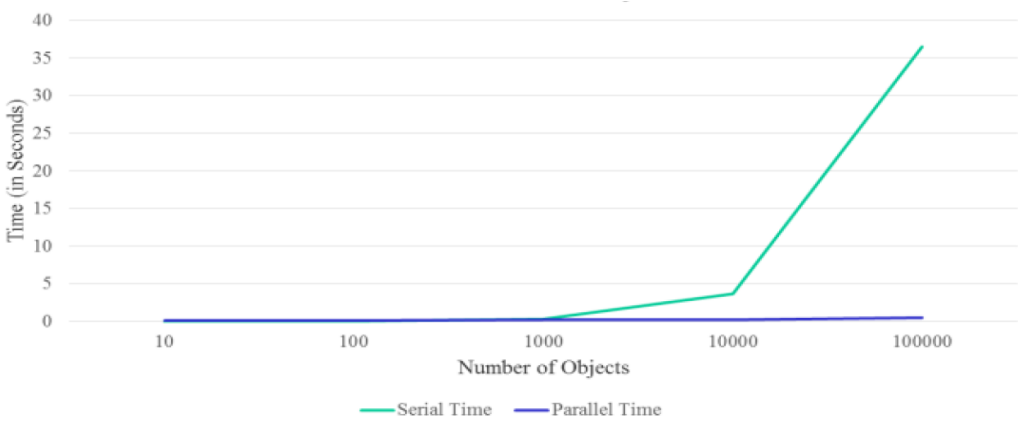


Figure 3: Performance of K-Means Clustering
[1]

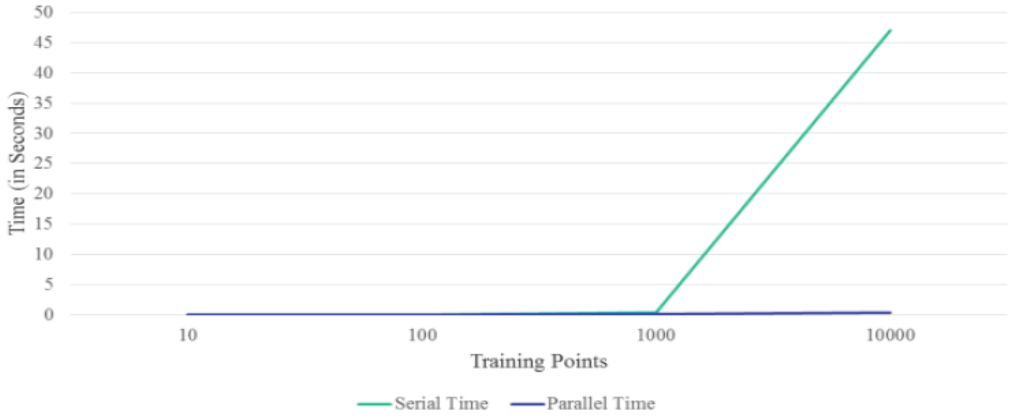


Figure 4: Performance of K-Nearest Neighbours Regression
[1]

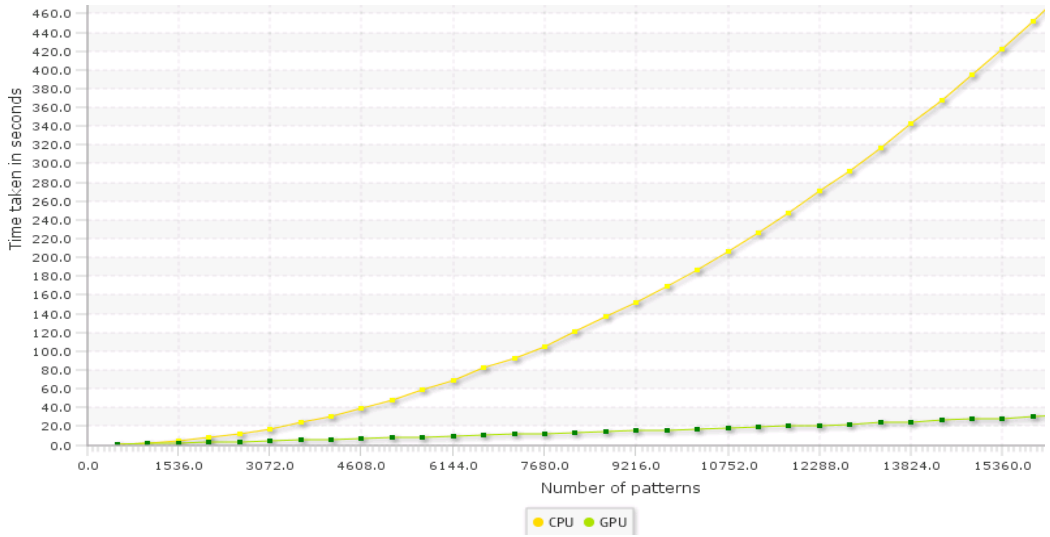


Figure 5: Performance of Backpropogation Algorithm
[2]

In the above results, parallel implementations of k-means and k-nearest neighbours algorithms ran faster when the number of training points were increased beyond a limit. But the serial implementation of Backpropogation algorithm ran much better compared to the parallel, when the number of input patterns were increased.

3.3 HBFT Algorithm

Breadth-first traversal is widely used in many applications today like electronic design automation (EDA).A GPU implementation of BFT called HBFT was made by Dabarera Et Al. in their 2016 paper, for breadth-first traversal in sparse graphs[3].
All the N edges of the graph were stored in an array called *GraphList*. Initially, all vertices, except for the root, are assumed to be at level(*depth*) -1. All the levels are store in *level* array.
The CPU implementation of HBFT ran on a single thread,which checked the edges one by one and updated the levels of the vertices. The GPU implementation spawned N threads where each thread is assigned to check one edge. While checking, if all the vertices at a particular level (*depth*) in the graph

has been checked, the algorithm moves on to the next level. The GPU is able to check all edges at a particular level faster due to the parallel nature of the code and hence performs much better than the CPU.

```
struct Edge element;
for(k=0;k<*edges;k++){
        element = adjacencyList[k];
        if ((level[element.from]>=0)&&
          (level[element.to]==-1)){
                level[element.to] =
                 level[element.from]+1;
      }
}
```

Figure 6: Serial Implementation
[3]

```
__global__ void BreadthFirstSearch(
   struct Edge * adjacencyList, int * vertices,
   int * level, int * lev, int * edges ){

   int tid = (blockDim.x * blockIdx.x ) +
                    threadIdx.x;
   *lev = 0;
   if(tid<*edges){

     struct Edge element =
               adjacencyList[tid];
  if (level[element.from]>=0 and
          level[element.to]==-1){
     level[element.to] =
               level[element.from]+1;
  } }}
```

Figure 7: Parallel CUDA implementation
[3]

```
__global__ void parentKernel(struct Edge *
adjacencyList, int * vertices, int * level,
                int * lev, int * edges){
*lev=1; int kb=0;

while(*lev){
    kb = kb+1;
    BreadthFirstSearch<<<ceil(*edges/256.0),256>>>
         (adjacencyList,vertices,level,lev,edges);
    cudaDeviceSynchronize();
    isLevelFilled<<<ceil(*vertices/256.0),256>>>
                        (level,vertices,lev);
    cudaDeviceSynchronize();
}}
```

Figure 8: Parent kernel of CUDA implementation
[3]

In the CUDA implementation, the job of *parentKernel* is to spawn child threads of *BreadthFirstSearch* and *isLevelFilled*, until the whole level array is filled with non-negative values.

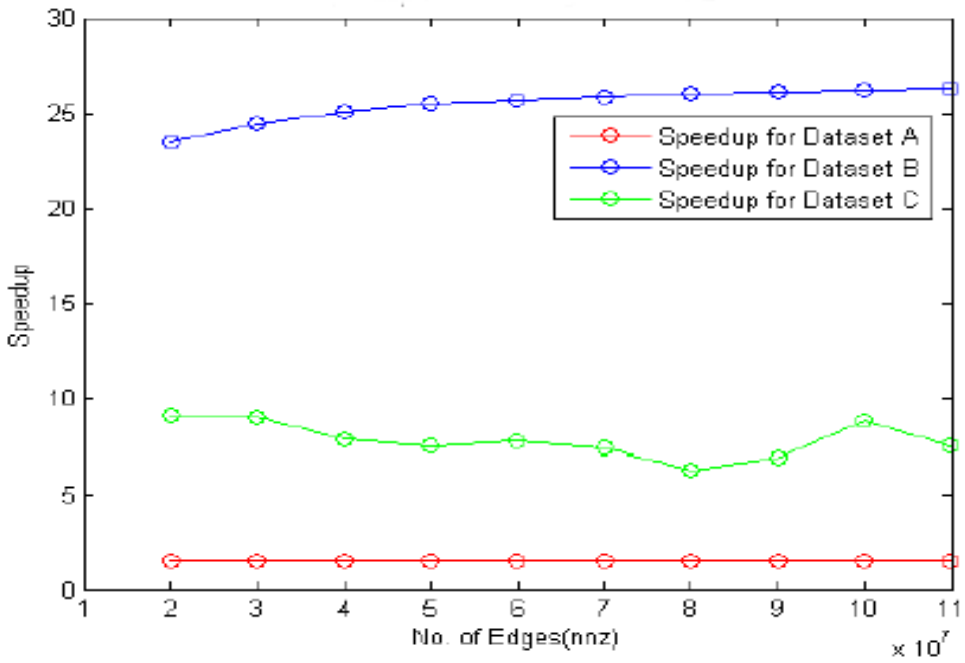


Figure 9: Speedup of HBFT on GPU
[3]

Each dataset represents a graph with around 10^7 edges. In dataset A, the edges are stored in the order of their levels. In dataset B, the edges are stored in the reverse order of their levels. While in dataset C, the edges are stored in a random order.
In dataset B, the CPU had to traverse the entire array of edges more than once to cover the entire graph, while the GPU only had to do it once. Hence in this case the speedup obtained on GPU is the highest among all datasets.

4 Conclusions

The results show that GPUs are much faster than CPUs in running algorithms that rely on parallelism and involves processing of huge datasets. Along with the algorithms depicted in this report, developers working in the fields on cryptocurrency, autonomous cars and other high perfo conserving time, energy and money.

References

[1] D. M. M. Ritvik Sharma, Vinutha M, "Revolutionizing machine learning algorithms using gpus," *IEEE Xplore:7779378*, Oct 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7779378/

[2] P. B. Kulkarni, "Projects - ann on gpu," Oct 2017. [Online]. Available: http://www.spikingneurons.com/projects_ann-on-gpu

[3] Dabarera, Karunarathna, Harshani, and Regel, "H-bft: A fast breadth-first traversal algorithm for sparse graphs and its gpu implementation," *IEEE Xplore : 7946532*, Jun 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7946532/

[4] Wikipedia, "Amdahl's law." [Online]. Available: https://en.wikipedia.org/wiki/Amdahl%27s_law

[5] Nvidia, "Introduction to cuda and gpgpu computation," *NVIDIA Presentation at Stanford*, 2010.