A Brief High Level Introduction to Parallel Computing

Fyroz S Dadapeer

Abstract

The report below attempts to provide a brief overview of the different parallel computing programming models and platforms such as OpenMP, MPI, Python Concurrency, CUDA Programming, PyCUDA and TensorFlow across CPUs, GPUs and TPUs and tries to perform a simple benchmark test to understand the performance between them.

Introduction

Any person who uses a computing device nowadays expects low latency and high bandwidth. Low latency refers to the user expecting a snappy response to any action performed on the computing device and high bandwidth refers to the user expecting large amounts of computing being done at the same time. This is seen by the example of users needing a fast and very small load times on a website, fast load times on performing any operation on an application like video processing or editing or getting fast inference/training from a ML/DL model. Applications nowadays necessarily require fast computation to generate the desired output.

The problem of low latency and high bandwidth can be solved using 2 fundamental computer processing systems : CPU and GPU. CPU basically processes 1 instruction at a time and cycles through the list of instructions that needs to be executed continuously switching between the different data that it needs for execution. This it does by cycling through millions of instructions continuously. To get low latency and high bandwidth a CPU can process more instructions (higher frequency). However, there exists limitations to this type of incremental increase in the frequency of the cycles in the CPU. The first is the increase in the heat dissipated when we try to increase the number of clock cycles of a CPU. It has been shown that we can theoretically increase the CPU clock frequency with newer architectures. However, we can't practically and efficiently handle the heat generated by the increase in frequency of CPU's. The second problem with the CPU is its dependency on the bandwidth and latency of memory (cache as well as primary memory). This increases the cost of the system as a whole.

The solution to the above single core CPU problem is to increase the number of cores (i.e., the number of CPUs) a system has. This leads to a system having a multi-core system, maybe having multiple CPUs with different primary memory or a CPU having multiple cores sharing the same memory space. This multi-core system solves the low latency and High Bandwidth problem partially. The current demands of applications cannot be solved by just increasing the no of cores in a CPU. Data intensive tasks such as Video processing, Gaming and DL tasks require huge computation power.

Data intensive applications often require huge amounts of arithmetic and floating point operations to be performed. These kinds of operations are often basic when compared to the operations that need to be performed by a CPU. Also, applications require large amounts(distinctively) of operations to be performed and output at the same time emphasizing on the need for quantity of no of operations over speed.

This is where GPU's, specifically GPGPUs, come into picture. GPU's consist of large no of processors, specifically processors that can perform simple operations like floating point arithmetic operations needed for video processing, gaming and DL tasks. These processors are often slower in speed when compared to CPU's. However, they are sufficient for real time operations that need to be run concurrently. This makes GPU's very well suited to applications where we need a high amount of operations to be performed at the same time.

There exists various parallel programming models to achieve concurrency in CPUs and GPUs. Some programming models are discussed below.

Parallel Programming Models

OpenMP

OpenMP (Open Multi-Processing) is an API (Application Programming Interface) available for C & C++ programming languages that provides parallel computing functionality on the CPU based on the shared memory multiprocessing model. In a shared memory model, multiple processes or threads access

a single shared memory space. OpenMP provides compiler directives to enable concurrent execution of code blocks across multiple threads. This programming model is only limited to programs that follow the SIMD (Single Instruction Multiple Data) programming model. We can't effectively write separate instructions for different data. Also the amount of processing is limited by the number of cores the CPU has along with the bandwidth of the primary memory.

MPI

MPI (Message Passing Interface) Programming model is often used where a computer system has multiple different processors and each processor has its own private memory and address space. Concurrency can be achieved in these types of systems when these processors communicate with one another through a communication channel to share information with each other during processing. The MPI programming model provides this communication channel between processes. Using MPI, both SPMD (single program multiple data) and MPMD (multiple program multiple data) programs can be solved since both processes run independently of each other. (Here programs and not instruction since multiple programs run independently of each other on different processors.) MPI provides a variety of coroutine/functions to achieve parallelism. The disadvantage of MPI is its dependency and the bottleneck of communication time depends on the data bus or the network between the processors.

Python Concurrency

Python provides several different advanced and comprehensive modules to achieve concurrency in programs written in that language. These modules provide different ways to write concurrent applications. Below are the main modules provided by python standard library:

1. MultiThreading - used usually for I/O Bound tasks. In Multithreading, even though there may be multiple threads, only one single thread is executed at any given time. This necessarily does not provide any parallelism. However, multithreading can be used to write network I/O and file I/O operations. Some concepts from the shared-memory programming model are used here.

2. Multiprocessing - used for CPU intensive tasks. This module provides actual functionality of concurrent parallelism. Multiprocessing involves multiple processes running concurrently, each with its own memory space. Processes do not share memory by default, so communication between processes usually involves more overhead, such as inter-process communication (IPC). The concepts of MPI are used to write concurrent programs in this module.

3. asyncio/async/await - This module provides threading using coroutines for I/O and network-related tasks where the processing is in the file system or on the network side.

4. concurrent.futures - This module provides a high level interface for asynchronously executing callables which can be used to write parallel execution programs.

5. queue - This is the data structure provided by python to transfer data between different separately running concurrent processes safely.

GPU Programming

CPUs along with the above programming models can be used to perform some complex tasks such as time slicing, virtual emulation, complex control flows, branching, security etc. However, some tasks such as gaming, video processing and DL tasks require large amounts of low level simple operations to be performed at the same time. This is where a GPU comes into play.

Anatomy of a GPU

Processors in GPUs differ from general-purpose CPUs in that they have a much simpler structure that is designed to execute hundreds of arithmetic instructions simultaneously. To increase the performance in a CPU, components such as Branch predictor, out-of-order executer, pre-fetcher are added. However, these components heavily rely on the I/O speed of main memory from which the instructions are transferred. Caches which work on the principle of either spatial or temporal locality are often used to reduce this lag. However, these systems do not scale well for large data intensive applications and are expensive to build.

Processors in GPU altogether let go of these integrated components and have processors that only have ALU, Fetcher/Decoder and independent registers. We can further share fetcher/decoder logic among the ALU and execution context, such that all ALUs execute the same operations contained in an instruction stream/kernel with only change in the data on which the operation is being performed. (Note that the code that gets executed on a GPU device is called a kernel). A shared memory (DRAM) is added to transfer the data to the fetcher/decoder. Thus, now each thread has its own independent ALU and Execution context but that Fetcher/Decoder and DRAM is shared across all threads(processors). Sharing of Fetcher/Decoder also allows the instruction stream/kernel/parallel program to have identical program counter (PC) and the threadID can be used to identify the data which is to be used in the instructions. This architecture makes the GPU best suitable to run SIMD (Single Instruction Multiple Data) programs. The above makes up 1 processor in a GPU. This processor in a GPU is called a Compute Unit. Thus 1 compute unit consists of 1 Fetcher/Decoder, Multiple ALUs and Execution context and 1 shared memory. Each ALU can be referred to as a processing element(PE). GPU's usually have tens and hundreds of these compute units and each compute unit can consist of hundreds and thousands of processing elements.

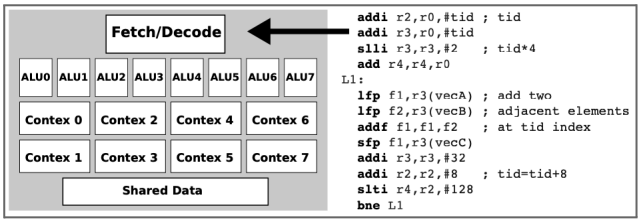


Fig: The picture above shows the architecture of a simple compute unit

For example, a Nvidia V100 GPU has 84 compute units, each having 64 processing elements, for a grand total of 5,396 processing elements.

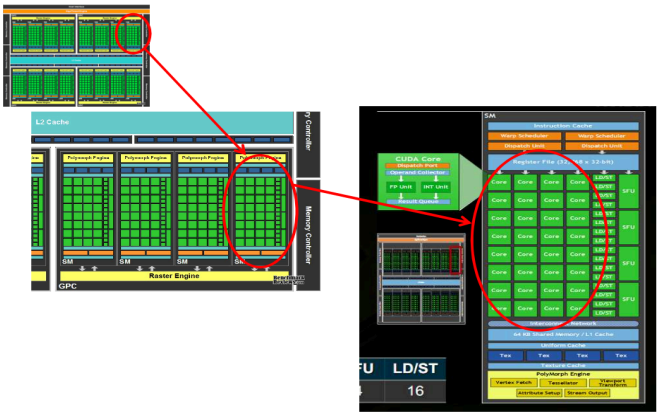


Fig: The picture above shows the architecture of a Nvidia GPU

Programming View of GPU

Generally, unit work done by a thread in a GPU is termed as a work-item (WI) and these work-items are grouped together into work-groups (WG). A work-group is simply a block of work-items that are executed on the same compute unit. However, note that all the work-items within a work-group are able to share local memory just like processing elements are able to share the same shared memory inside a compute unit. Work-groups execute independently from each other on their respective compute units. Also synchronization is possible only between the work-items in a work group.

The work-items and work-group of a GPU can be accessed using the concept of an NDRange (N-Dimensional Range). The NDRange represents the index space over which data can be iterated over. The host program (typically the CPU main() function) invokes the kernel over the above index space. The index space in a GPU usually has 1, 2, or 3 dimensions. Kernel functions are executed exactly one time for each point in the NDRange index space. However, note that unlike the index space of a for loop where the loop iterates sequentially over the indexes, a GPU device is free to access the index space in parallel and in any order indicating that the index space is for representing the data rather than the iteration.

Work-items have unique Global IDs and have unique Local IDs within a work-group. Work-groups also have unique work-group IDs. NDRange defines the total number of work-items that execute in parallel. The need for specifying unique work-group IDs is to define the no of

work-items inside a work-group and the no of work-group in total so that we can effectively define work-items in a work-group that can share data and synchronize with each other.

Memory model of GPU

GPUs provide 4 levels of memory for work-items.

1. Private memory: These are registers that are local to a specific work item/thread. Work-items can both read and write from these registers.
2. Local memory: This is a region that is shared within a work-group. All work-items in the same work-group have both read and write access. This is the shared memory within a compute unit.
3. Global memory: This is the dedicated memory of a GPU (usually DRAM, DDR5) in which all work-items and work-groups have read and write access.
4. Constant memory: This is a constant memory for the work items. Work-items have only read access to this region. The host is permitted to both read and write in this memory.

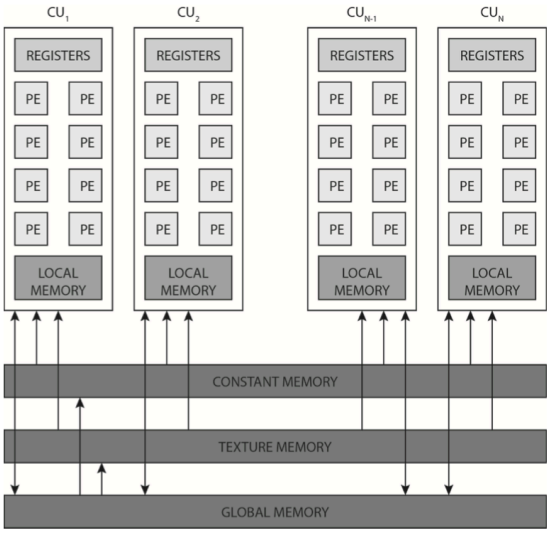


Fig: Memory Hierarchy in a GPU

GPU programming frameworks provide their own specifications to access the different NDRange index space and different levels of memory. The basics specification for CUDA for the Nvidia GPU will be discussed below.

When and Where to use GPU Programming models?

- ➔ Only on problems that can be broken down into many many small pieces and can be solved independently of each other.
- ➔ Only when we have very intense Data Parallel applications. We need not think about using GPUs for other applications.
- ➔ The requirement is not only for a few threads, but for thousands of threads.
- ➔ Where each thread executes the same program (called the kernel) but operation on different small piece of the overall data

CUDA Programming

Nvidia provides a high level API to the C++ programming language by which we can access the GPU device. The work-items in Nvidia GPUs are called threads and the work-groups are called blocks. Due to the limitation of the number of threads in a block (compute unit) we need to define both thread and block as per our requirement. This is also the case when we want to share data and synchronize between threads in a block. A thread block may contain up to 1024 threads. threadIDx and blockIDx are the variables provided inside the kernel function to access the NDRange index space. Note that threadIDx and blockIDx can be up to 3 dimensional vectors and can be used to access 3 dimensional elements sent inside the kernel function. CUDA also provides various other advanced features for efficiently programming GPUs which is outside the scope of this introduction report.

PyCUDA

PyCUDA provides Cython/Python wrappers for CUDA C++ drivers and runtime APIs allowing Python developers to use GPUs in their programs. However, in case of using PyCUDA, the kernel function needs to be written in C++ and needs to be embedded inside the python code. The main advantage of using PyCUDA is the ability to use python native libraries such as numpy, tensorflow in python and use GPUs for massive parallel computation if any custom computations required through the GPUs.

TPUs (Tensor Processing Units)

TPUs are specialized hardware accelerators focused on performing large scale matrix(tensor)

operations efficiently. The units are specially used in ML/DL training and inference tasks. TPUs offer significant performance advantages over traditional GPUs especially for deep learning tasks especially matrix multiplication operations due to their specialized hardware architecture and reduced precision support. Reduced precision refers to a bfloat16 format which is a 16 bit floating point format optimized for deep learning workloads which allows faster computation while maintaining reasonable numerical accuracy required for deep learning. Some other optimizations in TPUs include Fused-Multiply-Add arithmetic where matrix operations like multiplication of 2 matrices and addition to the result of the multiplication is done in 1 iteration. Frameworks such as Tensorflow/ Pytorch/ Keras are optimized to completely utilize the power of TPUs.

PERFORMANCE COMPARISON

The test consisted of the time taken to compute the matrix multiplication/product of 2 2-d Matrices of 1000 * 1000 dimensions with each element in the matrix set to an initial value of 55.55. The codes were written in C++ and Python to run on the different platforms. The configurations of the computing platforms on which the test was run is given in the respect programming codes attached below.

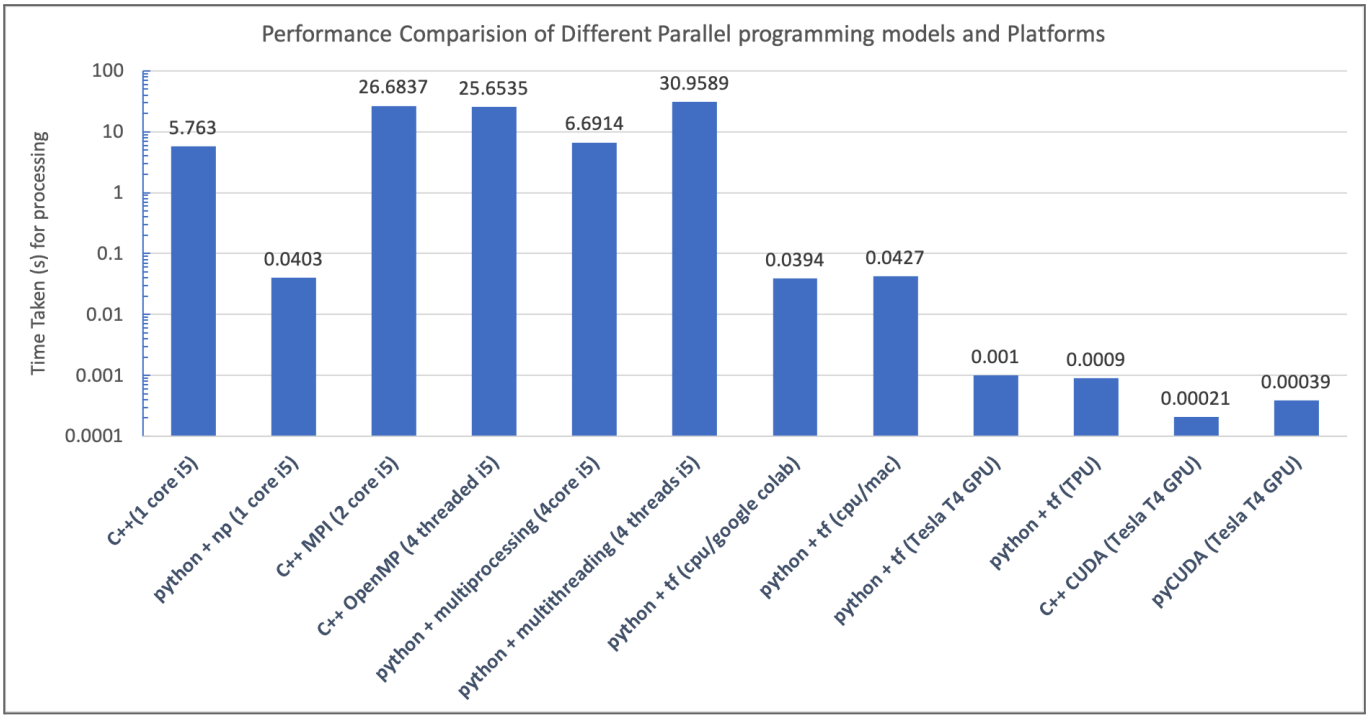
CODES

The github repository consists of all the codes and results of the tests conducted: https://github.com/fyr-repo/parallel_programming_intro

ANALYSIS

Note that this is not a full benchmarking report. A comprehensive analysis would require the tests to be done with different parameters such as speedup, scalability and efficiency with sufficiently large samples and various different types of problems. This is just a simple test to understand the difference between the performance of different parallel computing platforms.

- The performance of GPUs and TPUs is significantly higher than CPUs and its multithreading capabilities
- However, GPUs are only suitable when the program can be independently solved and CPUs are still the best when you need to do I/O and networking.
- The performance of TPUs is lower than GPU in this report. However, this is not the ideal case and TPUs are built to be faster and more efficient than GPUs. Tests on bigger data and TPU level optimization needs to be done to determine this.
- C++ OpenMP and MPI and Python Multithreading and Multiprocessing have lesser performance than single core performance in the same language. This maybe due to the need for coping of the data between the threads or processes and due to the unavailability of spatial or temporal locality provided using single core uses
- Python Tensorflow and Numpy implementations provide better performance than C++ single core implementations.



REFERENCES

1. Nvidia CUDA C++ Programming Guide https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
2. GPU Programming - CS6023 - IIT Madras http://www.cse.iitm.ac.in/~rupesh/teaching/gpu/jan24/
3. MPI Docs https://www.mpi-forum.org/docs/
4. PyCuda Documentation https://documen.tician.de/pycuda/
5. GPU 101 - Oregon State University https://web.engr.oregonstate.edu/~mjb/cs575/Handouts/gpu101.2pp.pdf
6. Python Standard Library - Concurrency https://docs.python.org/3/library/concurrency.html
7. Introduction to Parallel Computing: From Algorithms to Programming on State-of-the-Art Platforms - Monograph