Design Doc

A20375099 Yuan-An Liu,
A20382007 Sharel Clavy Pereira

Introduction

This assignment aims to extract the benchmark for different parts like the CPU, GPU, memory, disk and network of a computer system. We have used the Chameleon testbed for KVM virtual machines with 8 virtual CPUs with CentOS 7 Linux for the OS.

Benchmarking for CPU, Disk and Network is programmed using Python. Memory benchmarking is done using C++ while GPU benchmark is calculated using CUDA programs. All these programs have individual bash scripts to enable execution for all test cases.

In these programs, concurrency is achieved using thread considering thread synchronization issues to avoid inconsistency and deadlocks.

1. How to start benchmarking CPU

The purpose here is to figure out the speed of the processor in order to measure the time length at which the CPU can handle multiple instruction execution. We are measuring double precision floating point operations and integer operations, per second, in terms of GFLOPS and GIOPS. Concurrency is achieved by spawning multiple threads with multiple iterations for each type of operations. A plot is generated by marking the time on the x-axis and FLOPS/IOPS on the y-axis, with 1-second samples. At the end a Linpack benchmark is run and both the results are depicted to compare the results.

2. How to start benchmarking GPU

GPU speed is measured for NVIDIA P100 for double precision floating point operations , integer operations, half-precision operations and quarter-precision operations. Concurrency is achieved by mapping 1 thread per core, a total of 3584 cores; for each of the four experiments. Units are measured using GFLOPS, GIOPS, GHOPS, GQOPS . Finally a Linpack benchmark is run and compared against the results obtained to measure efficiency achieved.

3. How to start benchmarking Memory

A program is created to measure the speed of the memory by performing various operations like read+write, sequential read access and random read access.Optimum concurrency is achieved by passing multiple threads and blocks of varying size (1B, 1KB, 1MB, 10MB), on a 20MB file.

Throughput and latency is measured for each iteration in MB/s and ms respectively.Stream benchmark is run and the results are compared to evaluate the efficiency achieved.

4. How to start benchmarking Disk

In this section we are measuring the speed of the disk using different parameter space like read+write, sequential read access and random read access. Concurrency is achieved by passing multiple threads with blocks of varying size (1B, 1KB, 1MB, 10MB), using a 20MB file. Throughput and latency is measured for each iteration in MB/s and ms respectively. IOZone benchmarking is performed, results are then compared to measure the efficiency obtained.

5. How to start benchmarking Network

A measurement of network speed over the loopback interface card is done for 1 node, between 2 processes on the same node and the network speed between two nodes. The parameter space here includes TCP protocol stack, UDP with fixed packet/buffer size of 64KB. Multiple threads are parallelly run achieve concurrency.Throughput and latency is measured for each iteration in Mb/s and ms respectively. At the end IPerf benchmark is performed and the results are compared to evaluate the efficiency achieved.

Discussion:

Trade off:

1. Machine was idle while running the benchmarks to improve reliability and consistency of the results.
2. Language issue, Python will be less efficient but faster and easier for development. Also, Python has less backwards compatibility. And for C++ has a working multi-threading support, but all Python implementation have a broken multithreading. This is because all Python implementations have the GIL, which makes multi-threading not very effective, to say the least.
3. Other processes might be perform while we are running our timers. This might affect the accuracy of calculating the duration.

Conclusion:

How we can do better:

1. Use C or C++: Although we already try to use Multiprocessing instead of threading in python: Takes advantage of multiple CPUs & cores.
   Avoids GIL limitations for Python
   Eliminates most needs for synchronization primitives unless if you use shared memory (instead, it's more of a communication model for IPC)
2. Try to put in only operations with threads themselves in the timer.