Research Report: CPU-GPU Heterogeneous computing

Ting Li (SN:2017124076)

May 17, 2019

1 CPU & GPU

	CPU	GPU
Aim	Quickly execute a single instruction stream	Quickly execute a large number of parallel instruction streams
Compare	Cache is used to reduce memory access latency	Cache is used to amplify memory bandwidth
	Each core supports one or two threads	Run thousands of threads simultaneously
	Switching threads costs hundreds of clock cycles	Switching threads takes no time
	Vector data is processed through SIMD (single instruction multiple data)	through SIMT (Single Instruction Multiple Threads)
	The advantage is Integer operation	The advantage is floating point operation

Table 1: Compare of CPU & GPU

For the whole machine performance, CPU and GPU are the guarantee of performance. In order to give users the strongest comprehensive performance, reasonable collocation of CPU & GPU is the priority among priorities. (Called CPU-GPU Heterogeneous computing)

2 CPU-GPU heterogeneous platform

2.1 Introduction

Although we can gain great performance using CPU-GPU heterogeneous system, the programming complexity is much greater. There are some performance bottlenecks, for example, load balancing, synchronization and delay, data locality and task division. It is significant to solve these problems to achieve greater performance. There are some Parallel Programming Models.

Type	Parallel Programming Models
Traditional	MPI, OpenMP, Pthreads
New and developing	TBB, IBM X10, OpenCL
In researching	SWARM, Huckleberry, Skandium
Application-specific	Pub/Sub, MapReduce
Hardware-oriented	CUDA, StreamIt, NP-Click

Table 2: Parallel Programming Models

2.2 GPU and Compute Capability

For GPU, different device has different number of SM, per SM contains 8 SP(Scalar Stream Processor), a Instruction unit, registers, Shared memory, Constant memory, Texture Cache and so on. When excecuting CUDA program, each SM corresponds to a thread block. SM uses SIMT as its execute model. What's more, 32 consecutive threads are divided into a group called wrap. Warp block is the basic unit of thread scheduling.

The amount of SM is closely related to the computing power of GPU.

2.3 Performance optimization of CPU-GPU heterogeneous platform

Here are some Key factors affecting program performance: delay of accessing memory, load balancing and Global synchronization overhead.

Firstly, the bandwidth of memory has always been one of the major bottlenecks affecting computer performance. Whether the computing device is CPU or GPU, the computing power of the processor is much higher than the access bandwidth of the memory. For GPU, different types of memory differ greatly in capacity and access speed. In addition, even for the same memory, the effective bandwidth obtained by different access modes varies greatly. Therefore, choosing appropriate memory access mode and making full use of high memory on chip can effectively reduce memory access latency and improve the performance of the whole program.

Secondly, on the CPU-GPU heterogeneous parallel system, the task should be divided into 2 parts. One part to the CPU and the other part to the GPU. Secondly, because the kernel functions executed on the GPU are usually executed by thousands of threads, the tasks assigned to the GPU need to be further partitioned to these threads. Load distribution balance is related to the full utilization of computing resources, and also to the completion of a given computing task in the shortest time.

Finally, the CUDA runtime provides CudaThreadSynchronize () to synchronize hosts and devices, and _syncthreads () to synchronize threads in the same thread block. These basically have no overhead, but they do not provide functions to synchronize different thread blocks. Currently, the way to achieve synchronization between different thread blocks is to restart kernel functions. But it will increase the overhead and program complexity of multiple access to the global memory.

Here are some performance optimization strategies:

Type	Method	Description
Accessing Memory	Optimize data transfer between storages	Use Page-locked Memory. Transfer data between devices.
Optimize	Use joint access to global memory	Satisfy the conditions of joint access as far as possible.
	Use efficient Shared storage	Make full use of Shared memory Reduce access to global memory. Avoid bank conflicts.
Code Optimize	Optimization of arithmetic instructions	Add the - use_fast_math option when compiling CUDA programs.
	Avoid warp branching	For example: if, switch, for, while, do. Can use # pragma unroll.
Advanced Optimize	Computation overlaps with communication	1.The host uses Asynchronous communication function to transfer data to device, then CPU returns immediately and execute other tasks. 2.When CUDA program has several kernel functions or requires multiple starts of the same kernel function, the host can return and communicate with the device immediately after you start the kernel function
	Host and device parallel computing	Use cudaThreadSynchronize(). When dealing with large-scale data, a small part of the task is assigned to the CPU, so that the CPU and GPU work in parallel.
	Realize Global synchronization Using atomic functions	Use proper atomic functions. Like: atomicInc and atomicCAS.

Table 3: Performance Optimization Strategies