GPU Acceleration for Databases - Lecture Summary

Wes Robbins, Rishi Borkar

December 6, 2021

Introduction

- GPUs was originally developed for graphics processing – reliving the the CPU of an expensive task.
- GPUs have since been adopted other use cases that benefit from parallelization such as deep learning, scientific computing, and database acceleration

GPU Architecture

- Acceleration is due to GPUs having 1000s of cores that can run in parallel
- Same instruction multiple data (SIMD) execution means that many of the cores have to be doing the same work on different data
- Hardware hierarchy on a GPU: Streaming Multiprocessor $\rightarrow$ Warp $\rightarrow$ Core.

Parallel Model of Computation

- A processor with X cores is X times faster than a processor with one core
- In the parallel MOC, linear scan gets a X speed-up!
- Reductions get a $\log X$ speed-up

Programming a GPU

Major programming languages:

- CUDA is created by NVIDIA and is a parallel computing platform and application programming interface (API) that allows software to use certain types of graphics processing unit (GPU) for general purpose processing.
- OpenCL is an open standard maintained by the non-profit technology consortium Khronos Group. Conformant implementations are available from AMD, ARM, Intel, NVIDIA.

CUDA

- First introduced as the acronym Compute Unified Device Architecture, but NVIDIA later dropped the common use of the acronym.
- Based on C/C++ with some extensions
- Has a large user commumity on NVIDIA forums and detailed documentation

CUDA Compute Hierarchy

The processing resources in CUDA are designed to help optimize performance for GPU use cases.Three of the fundamental components of the hierarchy are threads, thread blocks, and kernelgrids

- Threads: A thread – or CUDA core – is a parallel processor that computes floating point math calculations in an Nvidia GPU. All the data processed by a GPU is processed via a CUDA core. Each CUDA core has its own memory register that is not available to other threads.
- Thread blocks: As the name implies, a thread block – or CUDA block – is a grouping of CUDA cores (threads) that can be executed together in series or parallel. The logical grouping of cores enables more efficient data mapping. Thread blocks share memory on a per-block basis. Current CUDA architecture caps the amount of threads per block at 1024. Every thread in a given CUDA block can access the same shared memory
- Kernel grids: The next layer of abstraction up from thread blocks is the kernel grid. Kernel grids are groupings of thread blocks on the same kernel. Grids can be used to perform larger computations in parallel(e.g. those that require more than 1024 threads), however since different thread blocks cannot use the same shared memory,

Example: Vector addition

Standard C++ implementation

```
void add(int* a, int* b, int* c)
{
    int tID = 0;
    while (tID < 10)
    {
        c[tID] = a[tID] + b[tID];
        tID += 1;
    }
}

add(a, b, c)
```

Parallel CUDA implementation

```
__global__ void cuda_add(int* a, int* b, int* c)
{
    int tID = blockIdx.x * blockDim.x + threadIdx.x;
    if (tID < 10)
    {
        c[tID] = a[tID] + b[tID];
    }
}

cuda_add<<<5, 2>>>(a, b, c)
```

Memory Transfer

```
// Allocate host memory
h_t1 = (int *)malloc(size);
// Allocate device memory
cudaMalloc((void **)&d_t1,size);

cudaMemcpy(d_t1, h_t1, size, cudaMemcpyHostToDevice);
// Kernel invoked
thread_multi<<<4, N/4>>>(d_t1);
cudaMemcpy(h_t1, d_t1, size, cudaMemcpyDeviceToHost);

cudaFree(d_t1);
free(h_t1);
```

Cons

- GPU's feature less memory than CPU's, GPU memory max is currently 32GB, CPU max goes into 100's of GB's up to in to the Terabytes for some systems.This makes memory management is a complicated task.
- GPU's are relatively expensive compare to CPU's and the cost- speed trade-off is sometimes noy justifiable for many tasks
- OpenCL/CUDA is not very human readable and is hard to debug. It also has a steep learning curve.
- CUDA is a NVIDIA vendor lock in and eventhough OpenCL has implementations for several chip manufacturers, it isn't widely supported by many tools and slower on NVIDIA hardware.

GPU Databases

Why use a GPU database?

- It can parallelize queries resulting in significant query runtime acceleration.
- It can perform analytics processing and produce visualizations much faster than a standard CPU.

Problems & Open challenges

- Disk-IO Bottleneck: GPU-accelerated operators are of little use for disk-based database systems, where most time is spent on disk I/O. Since the GPU improves performance only once the data is in main memory, time savings will be small compared to the total query runtime.
- Reducing PCIe Bus Bottleneck: Data is typically cached in GPU memory, a GPU DB needs a multi-level caching technique, which is yet to be found.
- Generic Cost Model: From the query-optimization perspective, a GPU DB needs a cost model to perform cost-based optimization. In this area, two basic cost-model classes have emerged. The first class consists of analytical cost models and the second class makes use of machine-learning approaches to learn cost models for some training data. While analytical cost models excel in computational efficiency, learning-based strategies need no knowledge about the underlying hardware and can adapt to changing data. It is still open which kind of cost model is optimal for GPU DB.
- Increased Complexity of Query Optimization: The plan search space is significantly larger and a cost function that compares run-times across architectures is required.

OmniSciDB

OmniSciDB is SQL-based, relational, columnar and specifically developed to harness the massive parallelism of modern CPU and GPU hardware.

- Advanced Memory Management: Keeps hot data in GPU memory for the fastest access possible. OmniSci Core avoids this transfer inefficiency by caching up to 512GB of the most recently touched data in the GPU's ultra-fast video RAM.
- Parallelizes computation across multiple GPUs and CPUs. The SQL engine can also execute purely on CPUs, delivering superior performance even on CPU-only systems.
- Distributed Architecture: Single queries to span more than one physical host when data is too large to fit on a single machine. Each GPU processes a slice of data independently from other GPUs. The data is fanned out from CPU to multiple GPUs and then gathered back together onto the CPU.

A demo of the an OmniSciDB at work can be seen at OmniSci Demo

Brytlyt

Brytlyt is a British GPU database and analytics company that uses GPUs to support business intelligence tools. It was released in 2016 and is based on PostgreSQL 12

- Differentiating feature: Patented intellectual property enables parallel processing on GPU for JOINs which allows for millisecond query time on billions of rows of data, without pre-aggregation