用于数据库分析的 GPU 和 CPU 的基本性能特征研究

摘要

本文是学习文章“A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics”的学习笔记，目的是总结文章中的要点，并归纳自己的理解。

GPU框架结构

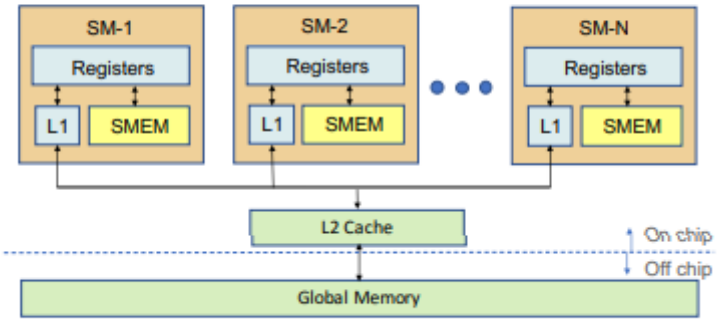

Figure 1: GPU Memory Hierarchy

要进行更加深入的阅读学习，首先需要了解现代GPU框架结构。其中图一是GPU的结构框架简图。

- 其中最底层最大的内存是全局内存。现代 GPU 可以拥有高达 32 GB 的全局内存容量和高达 $1200GBps$ 的内存带宽。
- 每一个GPU模块都有许多的计算单元，对应图中1到N个SM模块（$streaming \quad multiprocessors$）流多处理器。
- 每个 SM 都有多个内核和一组固定的寄存器。 每个 SM 还具有一个共享内存，用作由程序员控制的暂存器，并且可以被 SM 中的所有内核访问。 从 SM 对全局内存的访问被缓存在 L2 缓存中（L2 缓存在所有 SM 之间共享），也可以选择在 L1 缓存中（L1 缓存对于每个 SM 都是本地的，仅自己能访问）。

  PS:也就是说，内核要读取全局内存的数据要先经过L1和L2缓存器，以及自身的寄存器组。
- 程序使用GPU执行多线程任务，每一个SM对应一个线程，成为一个warp块。在对全局内存的存储和访问的时候，会将数据分存到每一个warp中，这是为了以便将多个加载/存储到同一高速缓存行组合成一个请求。 当 warp 对全局内存的访问导致相邻位置被访问时，可以实现最大带宽。

  PS：可以理解是把每一个SM中的内存读写过程合并成在一起执行。
- 随着摩尔定律放缓，CPU进步停滞不前。研究人员开始研究CPU-GPU系统，两个处理器之间通过PCIe协议连接，但是现代机器的PCIe的带宽非常小，远低于CPU或者是GPU的内存带宽。也就是说两种处理器放在一起使用反而因为无法有效联系在一起而卡脖子。
- 数据库社区过去的工作主要集中在使用 GPU 作为协处理器，我们称之为协处理器模型。 在此模型中，数据主要驻留在 CPU 的主内存中。 对于查询执行，数据通过 PCIe 从 CPU 传送到 GPU，因此（某些）查询处理可以在 GPU 上进行。 然后将结果传送回 CPU。

我们的目标

在本节中，我们将描述用于在 GPU 上高效执行查询的基于切片的执行模型。

- 首先说明为什么过去的协处理器模型是次优的设计。
- 给出一个示例，说明GPU上以大规模的并行方式运行查询的情况。
- 展示如何使用一组图元来表示我们的模型。

```
SELECT SUM(lo_extendedprice * lo_discount) AS revenue
FROM lineorder
WHERE lo_quantity < 25
AND lo_orderdate >= 19930101 AND lo_orderdate <= 19940101
AND lo_discount >= 1 AND lo_discount <= 3;
```

Figure 2: Star Schema Benchmark Q1.1

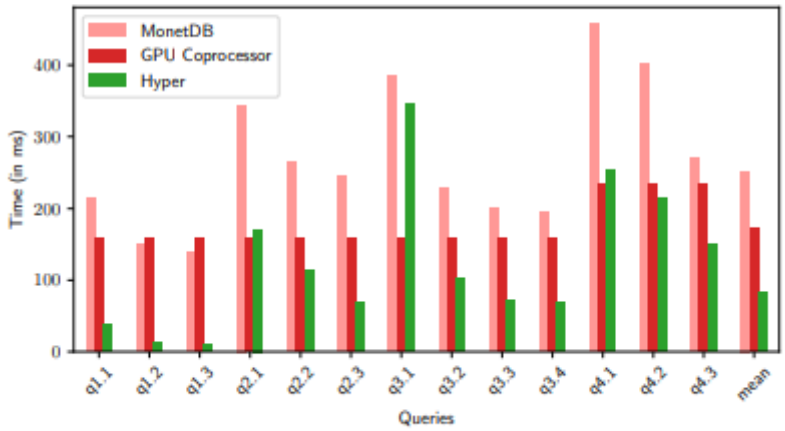

Figure 3: Evaluation on the Star Schema Benchmark

上面一大段话我没看懂。但是大致的意思应该是执行

```
select sum(lo_extendedprice * lo_discount) as revenue from lineorder
```

在线性序列表中中执行这样一条SQL语句来计算商品的折后价格。我们观察图中的时间复杂度。

其中，$MonetDB$是CPU处理的开源数据库。它在三者中的效率是最低，其耗费的时间复杂度是最高的。而Hyper的速度要快一点。我们关心GPU（红色）的效率。**结论，过去的工作能够显示 GPU 协处理器的性能改进的原因是因为它们的优化实现与 CPU 上的低效基线（例如 MonetDB）进行了比较。**换句话说就是在过去把GPU作为协处理器是因为把GPU作为加速器后的效率和CPU低效计算比较，而没有意识到GPU作为主要执行引擎的效率。

随着 GPU 内存容量的显着增加，我们将会见到把GPU作为主要处理器而不是加速器。我们接下来见识到它有多快。

Tile-based Execution Model

虽然CPU也有很多的内核。但是我们注意到例如英伟达V100这样的GPU处理器可以拥有5000个内核。这样的大量并行性能力的增长的能力是很关键的的。考虑在一个CPU和一个GPU上执行下列查询语句

```
SELECT y FROM R WHERE y > v;
```

在CPU中执行该查询

数据被均每个内核均分。我们的目的是将查询结果写道一个连续的数组中去。系统使用一个原子计数器。这个计数器能够作为一个游标，作用是指向下一个要写入结果的位置。每一个核心计算它自己均摊的部分数据。这部分是一个向量，大约能够包含1000+条数据（足够小到能够放进一个L1缓存器），每一个核心能够对自己的向量快速过**第一遍**，计算出符合向量中要求的结果的条数d，然后在结果数组上分配d个空间。然后他又会从L1缓存器中再过一遍这些数据，在原子计数器指引下把数据写入到结果数组分配到的d个空间的位置上。由于第二次读取是经由L1缓存器，所以这样的读数据几乎是免费的（不耗费时间），这一次就是把符合要求的数据写入到结果数组。然而，受限于L1缓存器大小，每一个向量只有1000+条记录，**但是我们同时只能执行32个线程**。所以计数器不是限速的瓶颈。

实际的运行时间在：$\frac{D}{B_C}+\frac{D_\sigma}{B_C}$，其中D是列宽，$B_C$是CPU内存的带宽。

在GPU中执行该查询

我们可以在 GPU 上运行相同的计划，在数千个线程中划分数据。 但是，GPU 线程每个线程的资源要少得多（可以理解成僧多粥少）。 在 Nvidia V100 上，每个 GPU 线程在完全占用的情况下只能在共享内存中存储大约 24 个 4 字节条目，并行运行 5000 个线程。 在这里，全局原子计数器最终成为瓶颈，因为所有线程都试图增加计数器以找到输出数组的偏移量。 为了解决这个问题，现有的基于 GPU 的数据库系统将分 3 个步骤执行此查询，如图 4(a) 所示。

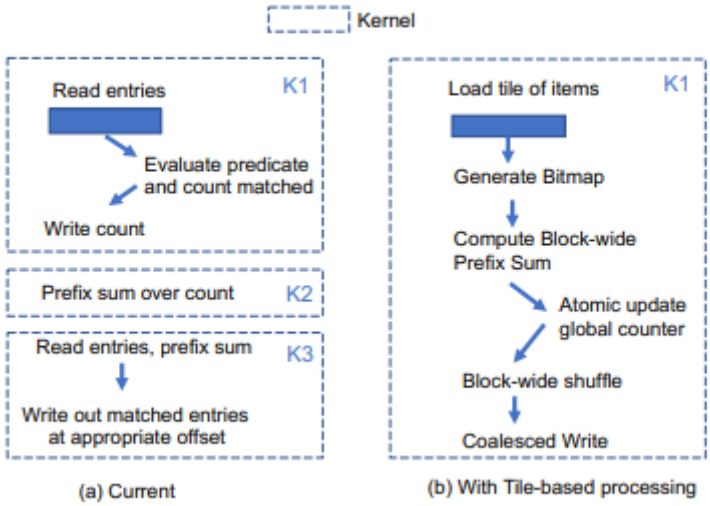

Figure 4: Running selection on GPU

在第一个内核K1中，会收取一大堆条目。其中每个条目以跨步（按线程编号交错）读取列条目，并且评估匹配的条目数目d。处理完所有元素后将每一个线程匹配的总数记录在count数组中，其中count[t]就是线程t中匹配的条目数目。第二个内核K2将使用计数数组来计算计数的前缀和并将其储存在另一个数组pf中。

前缀和：就是将该位置填写入原数组中该位置前面所有元素的和

其实这个前缀和数组pf就是每一个计算模块SM结果记录要写入的位置，比如SM1找到3条，SM2找到2条...，显然第一个前缀和是0，从位置0写入第一组答案，第二个前缀和是3，从3的位置处写入第二组查询记录，以此类推。同时线程本地还维护一个本地指针计数器c_i，它通常初始化成0。所以例程i的写入位置就是：$pf[i]+c_i$，然后然c_i自增长到$pf[i]+c_i$，这是为了下一轮使用该SM模块的时候能够继续写入对应位置答案。

上述方法处理输出数组的偏移量问题利用了前缀和来优化。它的运行时间是T的函数，T是线程数（T << n）该方法最终要比CPU到GPU方法转换要快很多。但是也同样存在许多问题。首先，它从全局内存中读取输入列两次，而不是像CPU一样只读一次（放在L1 缓存器中），它还要读取计算中间结构pf。最后，每个线程写入输出数组的不同位置，从而导致随机写入。

为了解决这些问题，我们引入TILE模型。

Tile-based execution module

这是基于图块处理的模型，把CPU上基于向量的处理扩展到GPU上，其中每个线程一次处理一个向量。图5说明该模型。

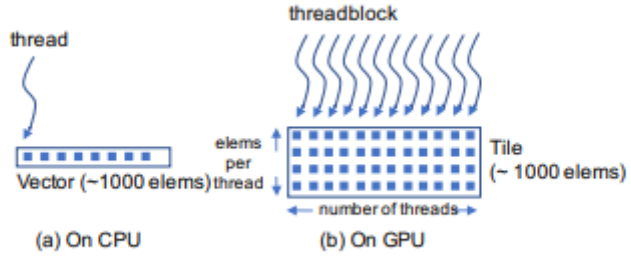

Figure 5: Vector-based to Tile-based execution models.

在同一个线程块中的线程们可以通过共享内存进行通信，并且可以通过屏障进行同步。因此，即使 GPU 上的单个线程在完全占用时最多只能在共享内存中保存 24 个整数，但单个线程块可以在共享内存中共同保存更大的一组元素。 我们称这个单元为 Tile。 在基于 Tile 的执行模型中，我们不是将每个线程视为一个独立的执行单元，而是将线程块视为基本执行单元，每个线程块一次处理一个条目 tile。 这种方法的一个关键优势是，在将 tile 加载到共享内存后，后续通过 tile 将直接从共享内存而不是从全局内存中读取，从而**避免了上述实现中描述的第二次通过全局内存**。

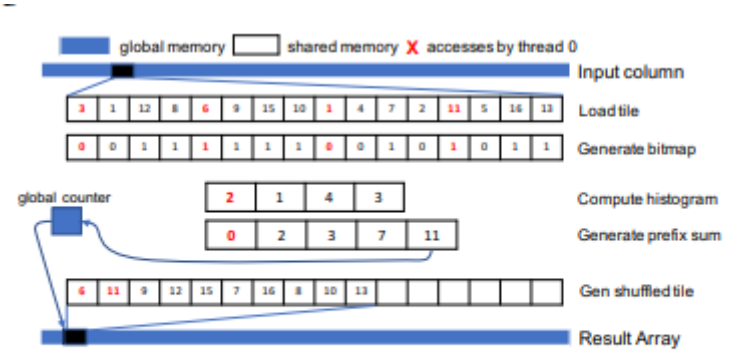

Figure 6: Query Q0 Kernel running y > 5 with tile size 16 and thread block size 4

图 4(b) 显示了如何使用基于图块的模型实现选择。 整个查询被实现为单个内核而不是三个。 图 6 显示了一个示例执行，其中一个大小为 16 的 tile 和一个由 4 个线程组成的线程块用于谓词 y >5。 请注意，这只是为了说明，因为大多数现代 GPU 使用的线程块大小是 32（warp 大小）的倍数，加载的元素数量将是线程块大小的 4-16 倍。 我们首先将全局计数器初始化为 0。内核将一组项目从全局内存加载到共享内存中。 然后线程将谓词并行应用于所有项目以生成位图。 例如，线程 0 计算元素 0、4、8、12 的谓词（以红色显示）。 然后每个线程计算每个线程匹配的条目数以生成直方图。 线程块协同计算直方图上的前缀和，以找到每个线程在共享内存中写入的偏移量。然后就是类似，上面的描述，写入结果。

Crystal库

##