NVIDIA

GPU Teaching Kit

Accelerated Computing

Module 5.2 – Parallel Computation Patterns (Reduction)

Parallel Reduction

目标

- 学习并行归约模式
  - —一类重要的并行计算
  - -工作**效率**分析
  - - 资源**效率**分析

“划分与总结”

- 一种处理大型输入数据集的常用策略

  在数据集中，处理元素没有必需的处理顺序（结合性和交换性）

  将数据集划分为较小的部分

  - 让每个线程处理一个数据块

  使用**归约树**将每个分块的结果汇总为最终答案

例如，**谷歌和 Hadoop MapReduce 框架**支持这一策略

- 目前我们将专注于**简化树**的**步骤**。

约简使其他技术成为可能

在一些常用的**并行化转换**之后，也需要进行简化以清理。

-私有化

- 多个线程写入一个输出位置
- 复制输出位置，以便每个线程都有一个单独的输出位置（私有化）

使用**归约树**将私有位置的值合并到原始输出位置

什么是归约计算?

使用“归约**操作**”将**一组输入值**归结为一个值

- 马克斯
- 最小
- 总和
- 产品

- 只要该操作:
  - 是结合律和交换律的
  - —具有明确的身份值（例如，总和为 0）

例如，对于三维坐标数据集，用户可能会提供一个自定义的“最大”函数，**其中每个坐标数据元**组的**幅值是从原点到该元组的距离**。

“集体操作”的一个例子

一种高效的顺序约简 O(N

将结果初始化为用于归约操作的恒等值

最大减小的可能最小值
- 最小减少量的最大可能值
- 0 表示求和简化
- 1 用于产品减量

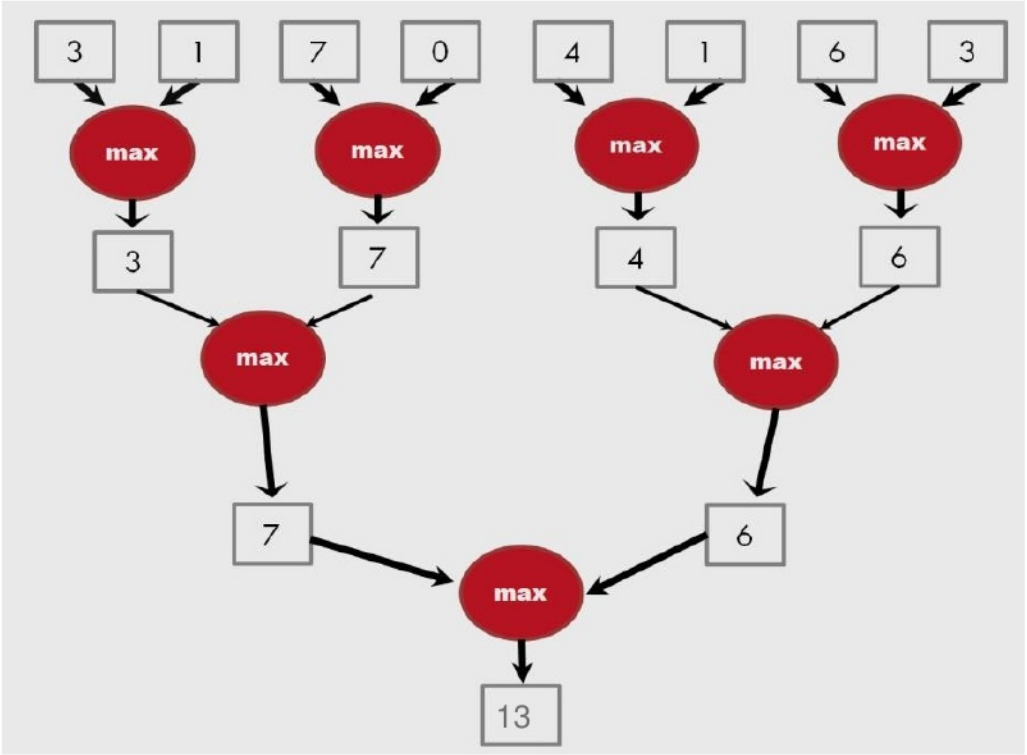
遍历输入并进行归约

对结果值和当前输入值进行运算

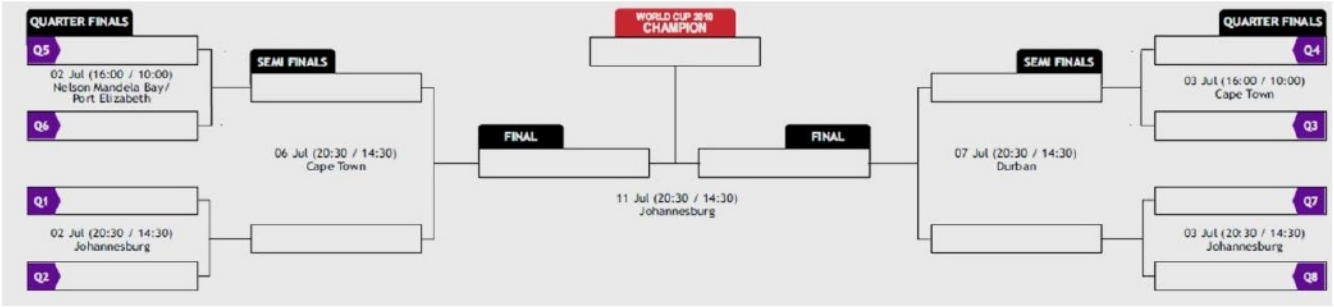
针对 N 个输入值执行了 N 次归约操作
- 每个输入值仅被访问一次 - 一个 O(N) 的算法

这是一种计算效率高的算法。

并行归约树算法在 log(N) 步内执行 N - 1 次操作



锦标赛是一棵具有“最大”运算的约简树



快速分析

对于 N 个输入值，归约树执行

– (1/2)N + (1/4)N + (1/8)N + … (1)N = (1- (1/N))N = **N-1 operations**

在 **Log(N) 步中，1,**048,576（2^20）个输入值需要 20 步。

——假设我们有足够的执行资源

- **平均并行度（N - 1）/ 对数（N）**

对于 N = 1,048,576，平均并行度约为 52,000 。

然而，资源需求高峰为 50 万。

这效率不**高**。

这是一种工作高效的并行算法。

完成的工作量与高效的顺序算法相当。

许多并行算法工作效率不高。

GPU Teaching Kit

Accelerated Computing

Module 9 – Parallel Computation Patterns (Reduction)

Lecture 9.2 - A Basic Reduction Kernel

目标

- 学习编写一个基本的归约内核
  - 线程到数据的映射
  - 关闭线程
  - 控制发散

并行求和归约

- 并行实现
  - 每个线程在每一步中增加两个值
  - 递归地将线程数量减半

  对于 n 个元素，需要 log(n) 步，需要 n/2 个线程

—假设使用共享内存进行就地归约

原始向量在**设备全局内存**中。

共享内存用于保存部分和向量

起初，部分和向量仅仅是原始向量。

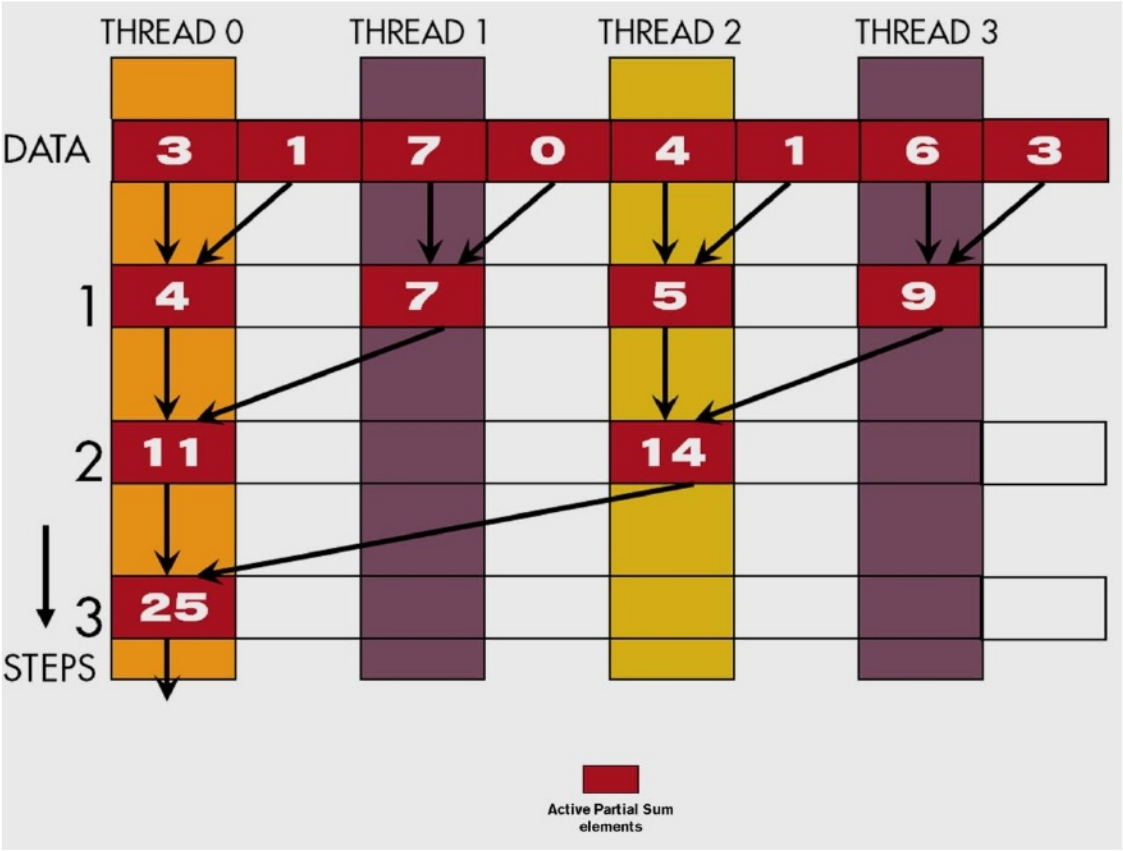
每一步都使部分和向量更接近总和。

最终的总和将出现在**部分总和向量**的**第 0 个元素**中。

由于读取和写入部分总和值，**减少了全局内存流量**

- **线程块大小**限制 n 小于或等于 2048 。

一个并行求和归约示例


THREAD 0
THREAD 1
THREAD 2
THREAD 3
DATA
3
1
7
0
4
1
6
3
1
4
7
5
9
2
11
14
3
25
STEPS
Active Partial Sum elements

一种简单的线程到数据的映射

每个线程负责部分和向量的**偶数索引位置**（责任位置）

每走一步，就**有一半的线**不再被需要。

其中一个输入总是来自**责任**所在的**位置**。

在每一步中，其中一个输入来自**距离越来越远**的地方。

一个简单的线程阻塞设计

每个线程块需要 2 * BlockDim.x 个输入元素
每个线程将 2 个元素加载到共享内存中

```
__shared__ float partialSum[2*BLOCK_SIZE];

unsigned int tx = threadIdx.x;
unsigned int start = 2*blockIdx.x*blockDim.x;
partialSum[tx] = input[start + tx];
partialSum[blockDim+tx] = input[start + blockDim.x+tx];
```

简化步骤

```
for (unsigned int stride = 1;
     stride <= blockDim.x;  stride *= 2)
{
  __syncthreads();
  if (tx % stride == 0)
    partialSum[2*tx]+= partialSum[2*tx+stride];
}
```

我们为什么需要 __syncthreads（）？

屏障同步

__syncthreads（） 是必需的，以确保在我们进入下一步之前，每个部分和版本的所有元素都已生成。

回到全球视角

在内核结束时，每个线程块中的**线程 0** 会将该线程块的总和写入由 **blockIdx.x** 索引的向量中 partialSum[0] 。

如果原始向量非常大，可能会有大量这样的总和。

主机代码可能会迭代并启动另一个内核。

-如果只有少量的金额，主持人可以简单地将数据转回并相加。

或者，每个块的线程 0 可以使用**原子操作**将结果累加到一个全局总和变量中。

GPU Teaching Kit

Accelerated Computing

Module 9 – Parallel Computation Patterns (Reduction)

Lecture 9.3 - A Better Reduction Kernel

目标

- 学习编写一个更好的归约内核
  - 提高了资源效率
  - 改进的线程到数据的映射
  - 控制偏差减小

关于朴素归约核的一些观察

在每次迭代中，对于每个 warp，将依次遍历两条控制流路径。

执行加法的线程和不执行加法的线程

不执行加法的线程仍然会消耗执行资源。

在第一步之后，执行的线程数量将减少到一半或更少。

在第一步之后，**所有奇数索引**的**线程**都被禁用。

在第五步之后，每个块中的所有 warp 都将无法通过 if 测试，**资源利用率低，但没有发散**。

这可能会持续一段时间，多达另外 6 步（步长 = 32、64、128、256、512、1024），在此期间，**每个活动**的工作负载线程只有**一个生产线程**，直到一个块中的所有工作负载线程都退出。

```
for (unsigned int stride = 1;
     stride <= blockDim.x;  stride *= 2)
{
  __syncthreads();
  if (t % stride == 0)
    partialSum[2*t]+= partialSum[2*t+stride];
}
```

线程索引的使用至关重要

在某些算法中，人们**可以改变索引**的**使用**来改善发散行为。

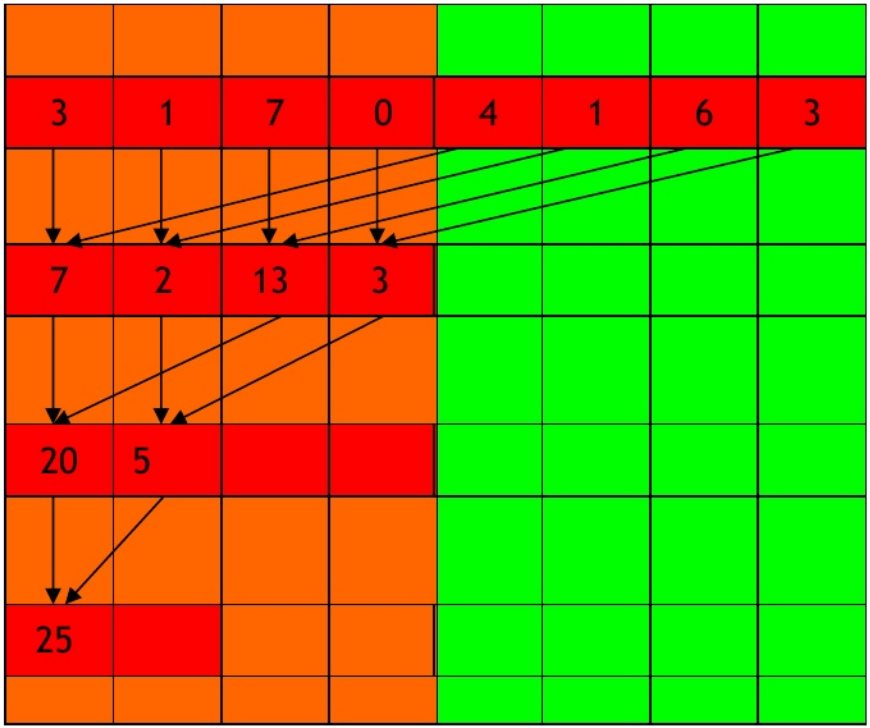
- 交换运算符和结合运算符

始终将部分总和压缩到 partialSum[ ] 数组的**前部位置**

保持活动线程**连续**

一个关于 4 个线程的示例

线程 0 线程 1 线程 2 线程 3



更优的约简内核

```
for (unsigned int stride = blockDim.x;
      stride > 0;  stride /= 2)
{
  __syncthreads();
  if (t < stride)
     partialSum[t] += partialSum[t+stride];
}
```

快速分析

- 对于一个 1024 线程的线程块

前 5 步没有分歧

- 在每一步中，1024、512、256、128、64、32 个连续的线程处于活动状态。

在每个 warp 中的所有线程要么全部处于活动状态，要么全部处于非活动状态。

最后的 5 个步骤仍会有分歧。

每一步中有连续的 16 个、8 个、4 个、2 个、1 个线程处于活动状态。

- 每个 warp 中少于 32 个线程

OnVIDIA

你好!

GPU 教学套件

加速计算