# PDC 2024 Homework 5 Report

210012945 寿晨宸

## Step 2: Avoid divergent branching and %

在 step 1 baseline 中，在进行条件判断时使用了 `if(tid % (2*s)==0)`，这会导致两个问题：

1. 在同一个 warp 中分支跳跃过多，可能的分支预测失误也多。
2. 使用了 %，较为消耗性能，进而使活跃线程比例较少。

因此，在 step 2 中，我更改了线程和条件判断间的映射，规避了使用 % 运算，尽量减少同一个 warp 内的 branching。

根据 omniperf profile 的结果：

| Metrix | Baseline | Step 2 |
| --- | --- | --- |
| Time(ns) | 1322721.00 | 1215200.00 |
| VALU Active Threads | 37.38 | 53.2 |
| Branch | 26.0 | 21.5 |

可以看出，step 2 增加了活跃线程，减少了 branching，并降低了运行时间。

## Step 3: Sequential addressing

在之前的算法中，因为 64 threads per wavefront/32 bank of LDS，因此在将 global memory load 到 local memory 时，会出现以下情况：在任意一个half-warp，第 k 个线程刚好把元素放进第 k 个 memory bank，即 `sdata[i].bank_id=i%32`。也就是说，在规约的每一个 iter，都会有两个 thread 同时访问同一个 bank 内的不同 memory，因此产生 bank conflict。

譬如，stride=1 时，thread 0 access sdata[0],sdata[1], thread 16 access sdata[32],sdata[33]，而 sdata[0].bank_id=sdata[32].bank_id=0，sdata[1].bank_id=sdata[33].bank_id=1。

step 3 更改了访存的方式，将交错寻址改成了线性寻址：

1. 在 iter 1 时，使一个 thread 只访问同一个 bank 内的元素，且不同的 thread 访问的 bank 之间没有重合。
2. 在之后的 iter 中，不存在属于同一个 bank 的不同元素。

根据 omniperf profile 的结果：

| Metrix | Step 2 | Step 3 |
| --- | --- | --- |
| Time(ns) | 1215200.00 | 1134880.00 |
| Bank Conflict | 45.00 | 0.00 |
| Branch | 21.5 | 20.5 |

可以看出 step 3 消除了 bank conflict，减少了运行时间。

## Step 4: First add during load

将 block 数量减半，从 global memory 加载数据到 local memory 时，一次性加载两个 element，相加之后放入 local memory 中。

根据 omniperf profile 的结果：

| Metrix | Step 3 | Step 4 |
| --- | --- | --- |
| Time(ns) | 1134880.00 | 948001.00 |
| VALU Active Threads | 48.38 | 52.34 |
| Branch | 20.5 | 20.5 |

可以看出 step 4 减少了运行时间。

## Step 5: Unroll the last iteration of the loop

step 5 展开了循环的最后一次迭代。

- step 1-4 中，每经过一次 iter，活跃的线程数都会减半，造成极大的浪费。
- 但事实上，我们并不需要所有 wavefronts 都执行循环的最后几次 iter。
- 另外，如果我们拥有少于 64 个线程(这 64 个线程在同一个 warp 内)，我们将不再需要 `__syncthreads()` 来同步，也不需要所有 if 语句。
- 在 step5 之后的 step 中，需要将最后循环展开的部分另放入 `__device__` 函数中，并将参数声明为 volatile，以免编译器做不必要的优化。

因此，我们只在循环中执行 stride>64 的 iter。并展开最后一层循环：若 tid<64，则令 $sdata[tid]=\sum_{i=0}^8 sdata[tid+2^i]$。

根据 omniperf profile 的结果：

| Metrix | Step 4 | Step 5 |
| --- | --- | --- |
| Time(ns) | 948001.00 | 993280.00 |
| VALU Active Threads | 52.34 | 59.15 |
| Branch | 20.5 | 5.75 |

可以看出 step 5 减少了 branching，增加了活跃线程，但因为将 sdata 声明为 volatile 而关闭了许多优化，所以增加了运行时间。

## Step 6: Complete unrolling using constexpr

如果在编译时就已经知道 iter 的信息，我们就能够展开整个循环。

- 将关于 BLOCK_SIZE 的判断转换成 constexpr，使得在编译时就能够确认分支走向。

根据 omniperf profile 的结果：

| Metrix | Step 5 | Step 6 |
| --- | --- | --- |
| Time(ns) | 993280.00 | 994241.00 |
| Branch | 5.75 | 3.25 |

step 6 降低了 branching，但运行时间的提升不明显。

## Step 7: Algorithm cascading: Mix parallel & Sequential execution

将 load/reduce two elements 的操作替换为一个循环，尽可能多得 reduce elements。将并行和串行相结合。

根据 omniperf profile 的结果：

| Metrix | Step 6 | Step 7 |
| --- | --- | --- |
| Time(ns) | 994241.00 | 990720.00 |
| VALU Active Threads | 59.15 | 59.9 |

step 7 略微降低了运行时间。

## Step 8: Using warp shuffles for inter-thread communication

shuffle 指令允许 thread 直接读其他 thread 的寄存器值，只要两个 thread 在同一个 warp 中，这种比通过 shared Memory 进行 thread 间的通讯效果更好，latency 更低，同时也不消耗额外的内存资源来执行数据交换。

step 8 运用 shuffle 指令实现了循环的最后几次在 warpSize 内的 iter。

根据 omniperf profile 的结果：

| Metrix | Step 7 | Step 8 |
| --- | --- | --- |
| Time(ns) | 990720.00 | 948959.00 |

step 8 降低了运行时间。