CUDA 高性能计算经典问题：前缀和

CUDA高性能计算经典问题：前缀和

作者 | Will Zhang
来源 | OneFlow
编辑 | 极市平台

本文讨论一个经典问题Prefix Sum（前缀和），也被称为Scan/Prefix Scan等。Scan 是诸如排序等重要问题的子问题，所以基本是进阶必学问题之一。

1 问题定义

首先我们不严谨地定义这个问题，输入一个数组input[n]，计算新数组output[n]，使得对于任意元素output[i]都满足：

output[i] = input[0] + input[1] + ... input[i]

一个示例如下：

输入	0	1	2	...	9
输出	0	1	3	...	45

如果在CPU上我们可以简单地如下实现：

```
void PrefixSum(const int32_t* input, size_t n, int32_t* output) {
  int32_t sum = 0;
  for (size_t i = 0; i < n; ++i) {
    sum += input[i];
    output[i] = sum;
  }
}
```

问题来了，如何并行？而且是几千个线程和谐地并行？这个问题里还有个明显的依赖，每个元素的计算都依赖之前的值。所以第一次看到这个问题的同学可能会觉得，这怎么可能并行？

而更进一步地，如何用CUDA并行，Warp级别怎么并行，Shared Memory能装下数据的情况怎么并行，Shared Memory装不下的情况如何并行等等。

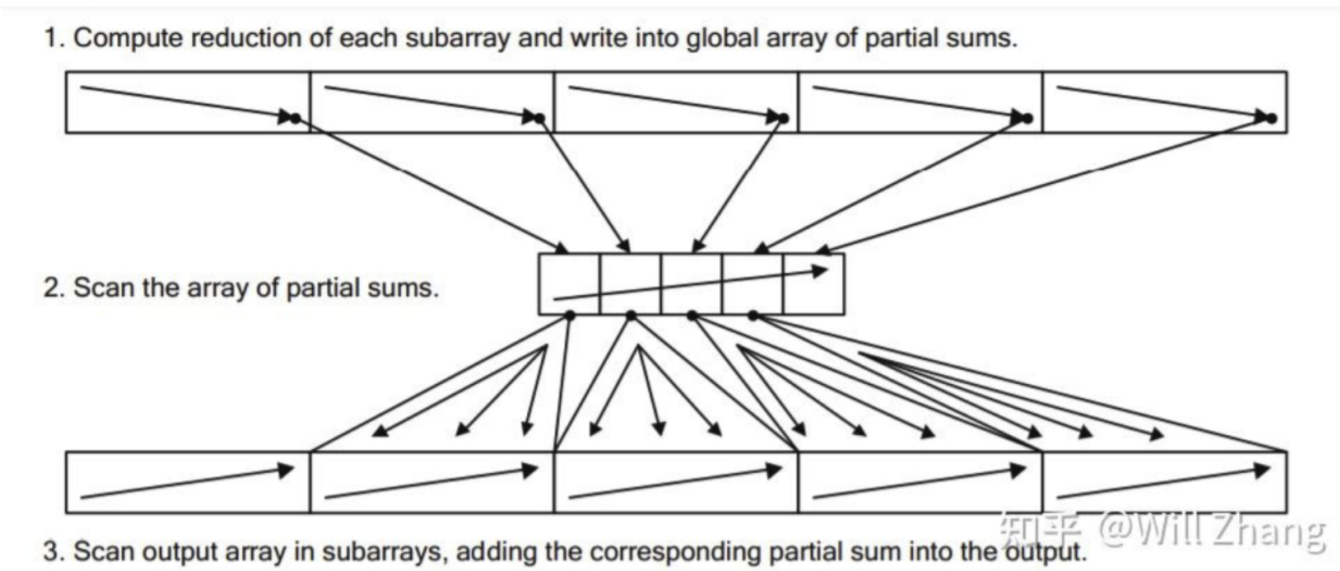
2 ScanThenFan

首先我们假设所有数据都可以存储到Global Memory中，因为更多的数据，核心逻辑也是类似的。

我们介绍的第一个方法称为ScanThenFan，也很符合直觉，如下：

- 将存储在Global Memory中的数据分为多个Parts，每个Part由一个Thread Block单独做内部的Scan，并将该Part的内部Sum存储到Global Memory中的PartSum数组中
- 对这个PartSum数组做Scan，我们使用BaseSum标识这个Scan后的数组
- 每个Part的每个元素都加上对应的BaseSum

如下图



3 Baseline

我们先不关注Block内如何Scan，在Block内先使用简单的单个线程处理，得到如下代码：

```
__global__ void ScanAndWritePartSumKernel(const int32_t* input, int32_t* part,
                                          size_t part_num) {
  for (size_t part_i = blockIdx.x; part_i < part_num; part_i += gridDim.x) {
    // this part process input[part_begin:part_end]
    // store sum to part[part_i], output[part_begin:part_end]
    size_t part_begin = part_i * blockDim.x;
```

```
    size_t part_end = min((part_i + 1) * blockDim.x, n);
    if (threadIdx.x == 0) {  // naive implemention
      int32_t acc = 0;
      for (size_t i = part_begin; i < part_end; ++i) {
        acc += input[i];
        output[i] = acc;
      }
      part[part_i] = acc;
    }
  }
}
__global__ void ScanPartSumKernel(int32_t* part, size_t part_num) {
  int32_t acc = 0;
  for (size_t i = 0; i < part_num; ++i) {
    acc += part[i];
    part[i] = acc;
  }
}
__global__ void AddBaseSumKernel(int32_t* part, int32_t* output, size_t n,
                                 size_t part_num) {
  for (size_t part_i = blockIdx.x; part_i < part_num; part_i += gridDim.x) {
    if (part_i == 0) {
      continue;
    }
    int32_t index = part_i * blockDim.x + threadIdx.x;
    if (index < n) {
      output[index] += part[part_i - 1];
    }
  }
}
// for i in range(n):
//   output[i] = input[0] + input[1] + ... + input[i]
void ScanThenFan(const int32_t* input, int32_t* buffer, int32_t* output,
                 size_t n) {
  size_t part_size = 1024;  // tuned
  size_t part_num = (n + part_size - 1) / part_size;
  size_t block_num = std::min<size_t>(part_num, 128);
  // use buffer[0:part_num] to save the metric of part
  int32_t* part = buffer;
  // after following step, part[i] = part_sum[i]
  ScanAndWritePartSumKernel<<<block_num, part_size>>>(input, part, output, n,
                                                      part_num);
  // after following step, part[i] = part_sum[0] + part_sum[1] + ... part_sum[i]
  ScanPartSumKernel<<<1, 1>>>(part, part_num);
  // make final result
  AddBaseSumKernel<<<block_num, part_size>>>(part, output, n, part_num);
}
```

现在的代码里很多朴素实现，但我们先完成一个大框架，得到此时的耗时72390us作为一个Baseline。

4 Shared Memory

接着，我们看ScanAndWritePartSumKernel函数，我们先做个简单的优化，将单个Part的数据先Load到Shared Memory中再做同样的简单逻辑，如下

```
__device__ void ScanBlock(int32_t* shm) {
  if (threadIdx.x == 0) {  // naive implemention
    int32_t acc = 0;
    for (size_t i = 0; i < blockDim.x; ++i) {
      acc += shm[i];
      shm[i] = acc;
    }
  }
  __syncthreads();
}
```

```
__global__ void ScanAndWritePartSumKernel(const int32_t* input, int32_t* part,
                                          int32_t* output, size_t n,
                                          size_t part_num) {
  extern __shared__ int32_t shm[];
  for (size_t part_i = blockIdx.x; part_i < part_num; part_i += gridDim.x) {
    // store this part input to shm
    size_t index = part_i * blockDim.x + threadIdx.x;
    shm[threadIdx.x] = index < n ? input[index] : 0;
    __syncthreads();
    // scan on shared memory
    ScanBlock(shm);
    __syncthreads();
    // write result
    if (index < n) {
      output[index] = shm[threadIdx.x];
    }
    if (threadIdx.x == blockDim.x - 1) {
      part[part_i] = shm[threadIdx.x];
    }
  }
}
```

这个简单的优化把时间从72390us降低到了33726us，这源于批量的从`Global Memory`的读取。

5 ScanBlock

接下来我们正经地优化Block内的Scan，对于Block内部的Scan，我们可以用类似的思路拆解为

- 按照Warp组织，每个Warp内部先做Scan，将每个Warp的和存储到`Shared Memory`中，称为WarpSum
- 启动一个单独的Warp对WarpSum进行Scan
- 每个Warp将最终结果加上上一个Warp对应的WarpSum

代码如下

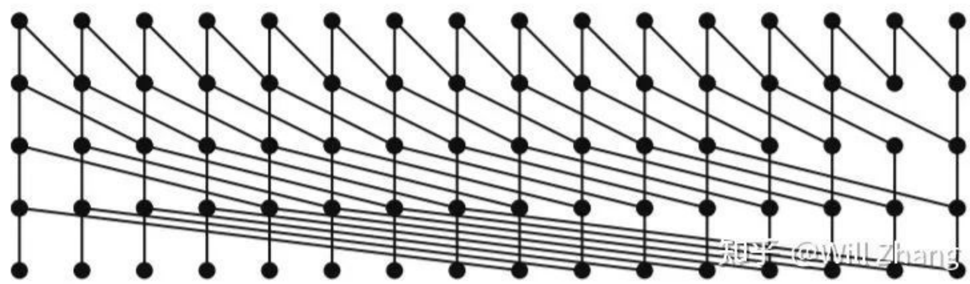
```
__device__ void ScanWarp(int32_t* shm_data, int32_t lane) {
  if (lane == 0) {  // naive implemention
    int32_t acc = 0;
    for (int32_t i = 0; i < 32; ++i) {
      acc += shm_data[i];
      shm_data[i] = acc;
    }
  }
}
__device__ void ScanBlock(int32_t* shm_data) {
  int32_t warp_id = threadIdx.x >> 5;
  int32_t lane = threadIdx.x & 31;  // 31 = 00011111
  __shared__ int32_t warp_sum[32];  // blockDim.x / WarpSize = 32
  // scan each warp
  ScanWarp(shm_data, lane);
  __syncthreads();
  // write sum of each warp to warp_sum
  if (lane == 31) {
    warp_sum[warp_id] = *shm_data;
  }
  __syncthreads();
  // use a single warp to scan warp_sum
  if (warp_id == 0) {
    ScanWarp(warp_sum + lane, lane);
  }
  __syncthreads();
  // add base
  if (warp_id > 0) {
```

```
    *shm_data += warp_sum[warp_id - 1];
  }
  __syncthreads();
}
```

这一步从33726us降低到了9948us。

6 ScanWarp

接着我们优化ScanWarp。为了方便解释算法，我们假设对16个数做Scan，算法如下图：



横向的16个点代表16个数，时间轴从上往下，每个入度为2的节点会做加法，并将结果广播到其输出节点，对于32个数的代码如下：

```
__device__ void ScanWarp(int32_t* shm_data) {
  int32_t lane = threadIdx.x & 31;
  volatile int32_t* vshm_data = shm_data;
  if (lane >= 1) {
    vshm_data[0] += vshm_data[-1];
  }
  __syncwarp();
  if (lane >= 2) {
    vshm_data[0] += vshm_data[-2];
  }
  __syncwarp();
  if (lane >= 4) {
    vshm_data[0] += vshm_data[-4];
  }
  __syncwarp();
  if (lane >= 8) {
    vshm_data[0] += vshm_data[-8];
  }
  __syncwarp();
  if (lane >= 16) {
    vshm_data[0] += vshm_data[-16];
  }
  __syncwarp();
}
```

这个算法下，每一步都没有bank conflict，耗时也从9948us降低到了7595us。

7 ZeroPadding

接下来我们想更进一步消除ScanWarp中的if，也就是不对lane做判断，warp中所有线程都执行同样的操作，这就意味着之前不符合条件的线程会访问越界，为此我们需要做padding让其不越界。

为了实现padding，回看ScanBlock函数，其定义的warp_sum并非为kernel launch时指定的。为了更改方便，我们将其更改为kernel launch时指定，如下

```
__device__ void ScanBlock(int32_t* shm_data) {
  int32_t warp_id = threadIdx.x >> 5;
```

```
  int32_t lane = threadIdx.x & 31;       // 31 = 00011111
  extern __shared__ int32_t warp_sum[];  // warp_sum[32]
  // scan each warp
  ScanWarp(shm_data);
  __syncthreads();
  // write sum of each warp to warp_sum
  if (lane == 31) {
    warp_sum[warp_id] = *shm_data;
  }
  __syncthreads();
  // use a single warp to scan warp_sum
  if (warp_id == 0) {
    ScanWarp(warp_sum + lane);
  }
  __syncthreads();
  // add base
  if (warp_id > 0) {
    *shm_data += warp_sum[warp_id - 1];
  }
  __syncthreads();
}
__global__ void ScanAndWritePartSumKernel(const int32_t* input, int32_t* part,
                                          int32_t* output, size_t n,
                                          size_t part_num) {
  // the first 32 is used to save warp sum
  extern __shared__ int32_t shm[];
  for (size_t part_i = blockIdx.x; part_i < part_num; part_i += gridDim.x) {
    // store this part input to shm
    size_t index = part_i * blockDim.x + threadIdx.x;
    shm[32 + threadIdx.x] = index < n ? input[index] : 0;
    __syncthreads();
    // scan on shared memory
    ScanBlock(shm + 32 + threadIdx.x);
    __syncthreads();
    // write result
    if (index < n) {
      output[index] = shm[32 + threadIdx.x];
    }
    if (threadIdx.x == blockDim.x - 1) {
      part[part_i] = shm[32 + threadIdx.x];
    }
  }
}
__global__ void ScanPartSumKernel(int32_t* part, size_t part_num) {
  int32_t acc = 0;
  for (size_t i = 0; i < part_num; ++i) {
    acc += part[i];
    part[i] = acc;
  }
}
__global__ void AddBaseSumKernel(int32_t* part, int32_t* output, size_t n,
                                 size_t part_num) {
  for (size_t part_i = blockIdx.x; part_i < part_num; part_i += gridDim.x) {
    if (part_i == 0) {
      continue;
    }
    int32_t index = part_i * blockDim.x + threadIdx.x;
    if (index < n) {
      output[index] += part[part_i - 1];
    }
  }
}
// for i in range(n):
//   output[i] = input[0] + input[1] + ... + input[i]
void ScanThenFan(const int32_t* input, int32_t* buffer, int32_t* output,
                 size_t n) {
```

```
  size_t part_size = 1024;  // tuned
  size_t part_num = (n + part_size - 1) / part_size;
  size_t block_num = std::min<size_t>(part_num, 128);
  // use buffer[0:part_num] to save the metric of part
  int32_t* part = buffer;
  // after following step, part[i] = part_sum[i]
  size_t shm_size = (32 + part_size) * sizeof(int32_t);
  ScanAndWritePartSumKernel<<<block_num, part_size, shm_size>>>(
      input, part, output, n, part_num);
  // after following step, part[i] = part_sum[0] + part_sum[1] + ... part_sum[i]
  ScanPartSumKernel<<<1, 1>>>(part, part_num);
  // make final result
  AddBaseSumKernel<<<block_num, part_size>>>(part, output, n, part_num);
}
```

注意在ScanAndWritePartSumKernel的Launch时，我们重新计算了shared memory的大小，接下来为了做padding，我们要继续修改其shared memory的大小，由于每个warp需要一个16大小的padding才能避免ScanWarp的线程不越界，所以我们更改ScanThenFan为：

```
// for i in range(n):
//   output[i] = input[0] + input[1] + ... + input[i]
void ScanThenFan(const int32_t* input, int32_t* buffer, int32_t* output,
                 size_t n) {
  size_t part_size = 1024;  // tuned
  size_t part_num = (n + part_size - 1) / part_size;
  size_t block_num = std::min<size_t>(part_num, 128);
  // use buffer[0:part_num] to save the metric of part
  int32_t* part = buffer;
  // after following step, part[i] = part_sum[i]
  size_t warp_num = part_size / 32;
  size_t shm_size = (16 + 32 + warp_num * (16 + 32)) * sizeof(int32_t);
  ScanAndWritePartSumKernel<<<block_num, part_size, shm_size>>>(
      input, part, output, n, part_num);
  // after following step, part[i] = part_sum[0] + part_sum[1] + ... part_sum[i]
  ScanPartSumKernel<<<1, 1>>>(part, part_num);
  // make final result
  AddBaseSumKernel<<<block_num, part_size>>>(part, output, n, part_num);
}
```

注意shm_size的计算，我们为warp_sum也提供了16个数的zero padding，对应的Kernel改写如下：

```
__device__ void ScanWarp(int32_t* shm_data) {
  volatile int32_t* vshm_data = shm_data;
  vshm_data[0] += vshm_data[-1];
  vshm_data[0] += vshm_data[-2];
  vshm_data[0] += vshm_data[-4];
  vshm_data[0] += vshm_data[-8];
  vshm_data[0] += vshm_data[-16];
}
__device__ void ScanBlock(int32_t* shm_data) {
  int32_t warp_id = threadIdx.x >> 5;
  int32_t lane = threadIdx.x & 31;
  extern __shared__ int32_t warp_sum[];  // 16 zero padding
  // scan each warp
  ScanWarp(shm_data);
  __syncthreads();
  // write sum of each warp to warp_sum
  if (lane == 31) {
    warp_sum[16 + warp_id] = *shm_data;
  }
  __syncthreads();
  // use a single warp to scan warp_sum
  if (warp_id == 0) {
    ScanWarp(warp_sum + 16 + lane);
  }
  __syncthreads();
  // add base
```

```
  if (warp_id > 0) {
    *shm_data += warp_sum[16 + warp_id - 1];
  }
  __syncthreads();
}
__global__ void ScanAndWritePartSumKernel(const int32_t* input, int32_t* part,
                                          int32_t* output, size_t n,
                                          size_t part_num) {
  // the first 16 + 32 is used to save warp sum
  extern __shared__ int32_t shm[];
  int32_t warp_id = threadIdx.x >> 5;
  int32_t lane = threadIdx.x & 31;
  // initialize the zero padding
  if (threadIdx.x < 16) {
    shm[threadIdx.x] = 0;
  }
  if (lane < 16) {
    shm[(16 + 32) + warp_id * (16 + 32) + lane] = 0;
  }
  __syncthreads();
  // process each part
  for (size_t part_i = blockIdx.x; part_i < part_num; part_i += gridDim.x) {
    // store this part input to shm
    size_t index = part_i * blockDim.x + threadIdx.x;
    int32_t* myshm = shm + (16 + 32) + warp_id * (16 + 32) + 16 + lane;
    *myshm = index < n ? input[index] : 0;
    __syncthreads();
    // scan on shared memory
    ScanBlock(myshm);
    __syncthreads();
    // write result
    if (index < n) {
      output[index] = *myshm;
    }
    if (threadIdx.x == blockDim.x - 1) {
      part[part_i] = *myshm;
    }
  }
}
```

改动比较多，主要是对相关index的计算，经过这一步优化，时间从7595us降低到了7516us，看似不大，主要是被瓶颈掩盖了。对于ScanWarp还可以用WarpShuffle来优化，为了体现其效果，我们放在后面再说，先优化当前瓶颈。

8 Recursion

当前的一个瓶颈在于，之前为了简化，对于PartSum的Scan，是由一个线程去做的，这块可以递归地做，如下：

```
// for i in range(n):
//   output[i] = input[0] + input[1] + ... + input[i]
void ScanThenFan(const int32_t* input, int32_t* buffer, int32_t* output,
                 size_t n) {
  size_t part_size = 1024;  // tuned
  size_t part_num = (n + part_size - 1) / part_size;
  size_t block_num = std::min<size_t>(part_num, 128);
  // use buffer[0:part_num] to save the metric of part
  int32_t* part = buffer;
  // after following step, part[i] = part_sum[i]
  size_t warp_num = part_size / 32;
  size_t shm_size = (16 + 32 + warp_num * (16 + 32)) * sizeof(int32_t);
  ScanAndWritePartSumKernel<<<block_num, part_size, shm_size>>>(
      input, part, output, n, part_num);
  if (part_num >= 2) {
    // after following step
    // part[i] = part_sum[0] + part_sum[1] + ... + part_sum[i]
```

```
    ScanThenFan(part, buffer + part_num, part, part_num);
    // make final result
    AddBaseSumKernel<<<block_num, part_size>>>(part, output, n, part_num);
  }
}
```

移除了之前的简单操作后，耗时从7516us下降到了3972us。

9 WarpShuffle

接下来我们使用WarpShuffle来实现WarpScan，如下:

```
__device__ int32_t ScanWarp(int32_t val) {
  int32_t lane = threadIdx.x & 31;
  int32_t tmp = __shfl_up_sync(0xffffffff, val, 1);
  if (lane >= 1) {
    val += tmp;
  }
  tmp = __shfl_up_sync(0xffffffff, val, 2);
  if (lane >= 2) {
    val += tmp;
  }
  tmp = __shfl_up_sync(0xffffffff, val, 4);
  if (lane >= 4) {
    val += tmp;
  }
  tmp = __shfl_up_sync(0xffffffff, val, 8);
  if (lane >= 8) {
    val += tmp;
  }
  tmp = __shfl_up_sync(0xffffffff, val, 16);
  if (lane >= 16) {
    val += tmp;
  }
  return val;
}
```

时间从3972us降低到了3747us。

10 PTX

我们可以进一步地使用cuobjdump查看其编译出的PTX代码，我添加了点注释，如下:

```
__device__ int32_t ScanWarp(int32_t val) {
  int32_t lane = threadIdx.x & 31;
  int32_t tmp = __shfl_up_sync(0xffffffff, val, 1);
  if (lane >= 1) {
    val += tmp;
  }
  tmp = __shfl_up_sync(0xffffffff, val, 2);
  if (lane >= 2) {
    val += tmp;
  }
  tmp = __shfl_up_sync(0xffffffff, val, 4);
  if (lane >= 4) {
    val += tmp;
  }
  tmp = __shfl_up_sync(0xffffffff, val, 8);
  if (lane >= 8) {
    val += tmp;
  }
  tmp = __shfl_up_sync(0xffffffff, val, 16);
  if (lane >= 16) {
    val += tmp;
```

```
  }
  return val;
}
```

时间从3972us降低到了3747us。

10 PTX

我们可以进一步地使用cuobjdump查看其编译出的PTX代码，我添加了点注释，如下：

```
// 声明寄存器
.reg .pred %p<11>;
.reg .b32 %r<39>;

// 读取参数到r35寄存器
ld.param.u32 %r35, [_Z8ScanWarpi_param_0];
// 读取threadIdx.x到r18寄存器
mov.u32 %r18, %tid.x;
// r1寄存器存储 lane = threadIdx.x & 31
and.b32 %r1, %r18, 31;
// r19寄存器存储0
mov.u32 %r19, 0;
// r20寄存器存储1
mov.u32 %r20, 1;
// r21寄存器存储-1
mov.u32 %r21, -1;
// r2|p1 = __shfl_up_sync(val, delta=1, 0, membermask=-1)
// 如果src lane在范围内，存储结果到r2中，并设置p1为True, 否则设置p1为False
// r2对应于我们代码中的tmp
shfl.sync.up.b32 %r2|%p1, %r35, %r20, %r19, %r21;
// p6 = (lane == 0)
setp.eq.s32     %p6, %r1, 0;
// 如果p6为真，则跳转到BB0_2
@%p6 bra BB0_2;
// val += tmp
add.s32 %r35, %r2, %r35;
// 偏移2
BB0_2:
mov.u32 %r23, 2;
shfl.sync.up.b32 %r5|%p2, %r35, %r23, %r19, %r21;
setp.lt.u32     %p7, %r1, 2;
@%p7 bra BB0_4;
add.s32 %r35, %r5, %r35;
...
```

可以看到，我们可以直接使用__shfl_up_sync生成的p寄存器来做条件加法，从而避免生成的条件跳转指令，代码如下：

```
__device__ __forceinline__ int32_t ScanWarp(int32_t val) {
  int32_t result;
  asm("{"
      ".reg .s32 r<5>;"
      ".reg .pred p<5>;"

      "shfl.sync.up.b32 r0|p0, %1, 1, 0, -1;"
      "@p0 add.s32 r0, r0, %1;"

      "shfl.sync.up.b32 r1|p1, r0, 2, 0, -1;"
      "@p1 add.s32 r1, r1, r0;"

      "shfl.sync.up.b32 r2|p2, r1, 4, 0, -1;"
      "@p2 add.s32 r2, r2, r1;"

      "shfl.sync.up.b32 r3|p3, r2, 8, 0, -1;"
      "@p3 add.s32 r3, r3, r2;"

      "shfl.sync.up.b32 r4|p4, r3, 16, 0, -1;"
      "@p4 add.s32 r4, r4, r3;"
```

```
      "mov.s32 %0, r4;"
      "}"
      : "=r"(result)
      : "r"(val));
  return result;
}
```

此外移除依赖的大量shared memory，如下：

```
__device__ __forceinline__ int32_t ScanBlock(int32_t val) {
  int32_t warp_id = threadIdx.x >> 5;
  int32_t lane = threadIdx.x & 31;
  extern __shared__ int32_t warp_sum[];
  // scan each warp
  val = ScanWarp(val);
  __syncthreads();
  // write sum of each warp to warp_sum
  if (lane == 31) {
    warp_sum[warp_id] = val;
  }
  __syncthreads();
  // use a single warp to scan warp_sum
  if (warp_id == 0) {
    warp_sum[lane] = ScanWarp(warp_sum[lane]);
  }
  __syncthreads();
  // add base
  if (warp_id > 0) {
    val += warp_sum[warp_id - 1];
  }
  __syncthreads();
  return val;
}
__global__ void ScanAndWritePartSumKernel(const int32_t* input, int32_t* part,
                                          int32_t* output, size_t n,
                                          size_t part_num) {
  for (size_t part_i = blockIdx.x; part_i < part_num; part_i += gridDim.x) {
    size_t index = part_i * blockDim.x + threadIdx.x;
    int32_t val = index < n ? input[index] : 0;
    val = ScanBlock(val);
    __syncthreads();
    if (index < n) {
      output[index] = val;
    }
    if (threadIdx.x == blockDim.x - 1) {
      part[part_i] = val;
    }
  }
}
__global__ void AddBaseSumKernel(int32_t* part, int32_t* output, size_t n,
                                 size_t part_num) {
  for (size_t part_i = blockIdx.x; part_i < part_num; part_i += gridDim.x) {
    if (part_i == 0) {
      continue;
    }
    int32_t index = part_i * blockDim.x + threadIdx.x;
    if (index < n) {
      output[index] += part[part_i - 1];
    }
  }
}
// for i in range(n):
//   output[i] = input[0] + input[1] + ... + input[i]
void ScanThenFan(const int32_t* input, int32_t* buffer, int32_t* output,
                 size_t n) {
  size_t part_size = 1024;  // tuned
  size_t part_num = (n + part_size - 1) / part_size;
```

```
  size_t block_num = std::min<size_t>(part_num, 128);
  // use buffer[0:part_num] to save the metric of part
  int32_t* part = buffer;
  // after following step, part[i] = part_sum[i]
  size_t shm_size = 32 * sizeof(int32_t);
  ScanAndWritePartSumKernel<<<block_num, part_size, shm_size>>>(
      input, part, output, n, part_num);
  if (part_num >= 2) {
    // after following step
    // part[i] = part_sum[0] + part_sum[1] + ... + part_sum[i]
    ScanThenFan(part, buffer + part_num, part, part_num);
    // make final result
    AddBaseSumKernel<<<block_num, part_size>>>(part, output, n, part_num);
  }
}
```

此时耗时下降到了3442us。

11 ReduceThenScan

不同于ScanThenFan，其在第一遍每个Part内部做Scan。在这一节中我们将在第一遍只算和，而在最后一步做Scan，代码如下：

```
__global__ void ReducePartSumKernel(const int32_t* input, int32_t* part_sum,
                                    int32_t* output, size_t n,
                                    size_t part_num) {
  using BlockReduce = cub::BlockReduce<int32_t, 1024>;
  __shared__ typename BlockReduce::TempStorage temp_storage;
  for (size_t part_i = blockIdx.x; part_i < part_num; part_i += gridDim.x) {
    size_t index = part_i * blockDim.x + threadIdx.x;
    int32_t val = index < n ? input[index] : 0;
    int32_t sum = BlockReduce(temp_storage).Sum(val);
    if (threadIdx.x == 0) {
      part_sum[part_i] = sum;
    }
    __syncthreads();
  }
}
__global__ void ScanWithBaseSum(const int32_t* input, int32_t* part_sum,
                                int32_t* output, size_t n, size_t part_num) {
  for (size_t part_i = blockIdx.x; part_i < part_num; part_i += gridDim.x) {
    size_t index = part_i * blockDim.x + threadIdx.x;
    int32_t val = index < n ? input[index] : 0;
    val = ScanBlock(val);
    __syncthreads();
    if (part_i >= 1) {
      val += part_sum[part_i - 1];
    }
    if (index < n) {
      output[index] = val;
    }
  }
}
void ReduceThenScan(const int32_t* input, int32_t* buffer, int32_t* output,
                    size_t n) {
  size_t part_size = 1024;  // tuned
  size_t part_num = (n + part_size - 1) / part_size;
  size_t block_num = std::min<size_t>(part_num, 128);
  int32_t* part_sum = buffer;  // use buffer[0:part_num]
  if (part_num >= 2) {
    ReducePartSumKernel<<<block_num, part_size>>>(input, part_sum, output, n,
                                                  part_num);
    ReduceThenScan(part_sum, buffer + part_num, part_sum, part_num);
  }
  ScanWithBaseSum<<<block_num, part_size, 32 * sizeof(int32_t)>>>(
```

```
    input, part_sum, output, n, part_num);
}
```

为了简化，我们在代码中使用cub的BlockReduce，这个版本的耗时为3503us， 略有上升。

之前的算法都存在递归，现在我们想办法消除递归，延续ReduceThenScan的想法，只需要我们把Part切得更大一些，比如让Part数和Block数相等，就可以避免递归，代码如下：

```
__global__ void ReducePartSumKernelSinglePass(const int32_t* input,
                                              int32_t* g_part_sum, size_t n,
                                              size_t part_size) {
  // this block process input[part_begin:part_end]
  size_t part_begin = blockIdx.x * part_size;
  size_t part_end = min((blockIdx.x + 1) * part_size, n);
  // part_sum
  int32_t part_sum = 0;
  for (size_t i = part_begin + threadIdx.x; i < part_end; i += blockDim.x) {
    part_sum += input[i];
  }
  using BlockReduce = cub::BlockReduce<int32_t, 1024>;
  __shared__ typename BlockReduce::TempStorage temp_storage;
  part_sum = BlockReduce(temp_storage).Sum(part_sum);
  __syncthreads();
  if (threadIdx.x == 0) {
    g_part_sum[blockIdx.x] = part_sum;
  }
}
__global__ void ScanWithBaseSumSinglePass(const int32_t* input,
                                          int32_t* g_base_sum, int32_t* output,
                                          size_t n, size_t part_size,
                                          bool debug) {
  // base sum
  __shared__ int32_t base_sum;
  if (threadIdx.x == 0) {
    if (blockIdx.x == 0) {
      base_sum = 0;
    } else {
      base_sum = g_base_sum[blockIdx.x - 1];
    }
  }
  __syncthreads();
  // this block process input[part_begin:part_end]
  size_t part_begin = blockIdx.x * part_size;
  size_t part_end = (blockIdx.x + 1) * part_size;
  for (size_t i = part_begin + threadIdx.x; i < part_end; i += blockDim.x) {
    int32_t val = i < n ? input[i] : 0;
    val = ScanBlock(val);
    if (i < n) {
      output[i] = val + base_sum;
    }
    __syncthreads();
    if (threadIdx.x == blockDim.x - 1) {
      base_sum += val;
    }
    __syncthreads();
  }
}
void ReduceThenScanTwoPass(const int32_t* input, int32_t* part_sum,
                           int32_t* output, size_t n) {
  size_t part_num = 1024;
  size_t part_size = (n + part_num - 1) / part_num;
  ReducePartSumKernelSinglePass<<<part_num, 1024>>>(input, part_sum, n,
                                                    part_size);
  ScanWithBaseSumSinglePass<<<1, 1024, 32 * sizeof(int32_t)>>>(
      part_sum, nullptr, part_sum, part_num, part_num, true);
  ScanWithBaseSumSinglePass<<<part_num, 1024, 32 * sizeof(int32_t)>>>(
```

```
        input, part_sum, output, n, part_size, false);
}
```

耗时下降至2467us。

12 结语

即使做了很多优化，对比CUB的时间1444us，仍然有较大优化空间。不过本人一向秉承“打不过就加入”的原则，而且CUB也是开源的，后面有时间再深入CUB代码写一篇代码解读。

Reference

https//:www.amazon.com/CUDA-Handbook-Comprehensive-Guide-Programming/dp/0321809467

（原文链接：https://zhuanlan.zhihu.com/p/423992093）