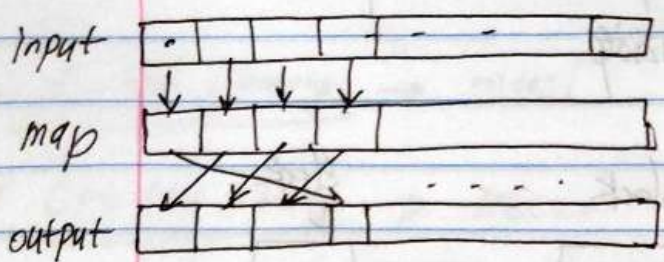
thrust-1.0.0

scatter:

input: (first, last)
(map, ...)
(output, ...)

```
while (first != last) {
    output[*map] = *first;
    first++;
    map++;
}
```



element-wise input

out[map[i]] = input[i]

element-wise op with functor

input: (first, last)
map: (map, ...)
stencil: (stencil, ...)
output: (output, ...)

```
while (first != last) {
    if (pred(*stencil))
        output[*map] = *first;
    first++;
    map++;
    stencil++;
}
```

---

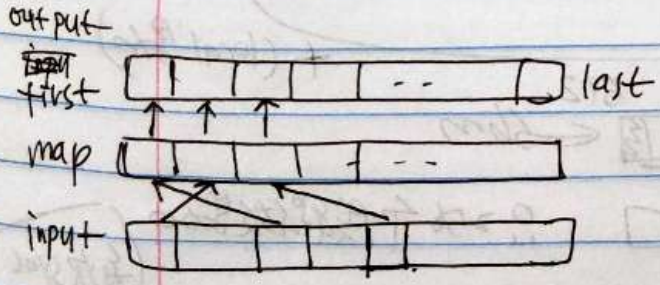
gather:

~~input~~ (first, last)
map: (map, ...)
~~output: (output, ...)~~
input: (input, ...)

```
while (first != last) {
    *first = input[*map];
    first++;
    map++;
}
```

op with functor

element-wise output

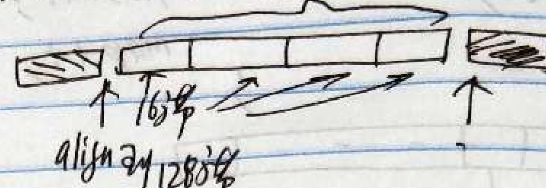



```
if (pred(stencil[i])) {
    out[i] = input[map[i]];
}
```

device/cuda/

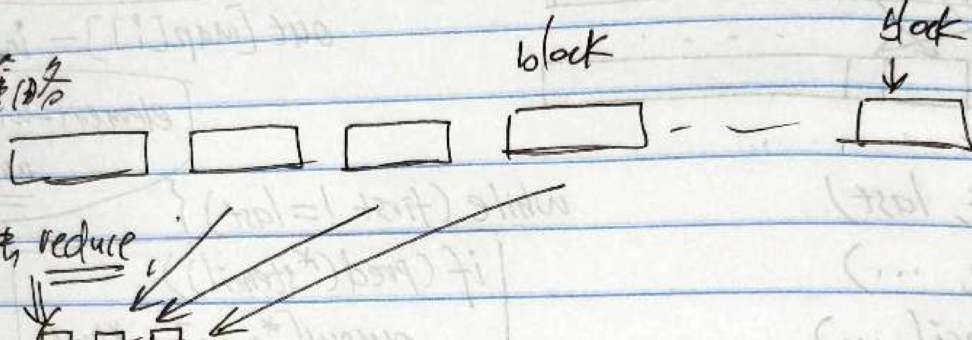
for_each

fill ⇒ vectorize 针对 sizeof = 1|2|4 大小的类型, 执行 wide fill



gather

reduce 策略

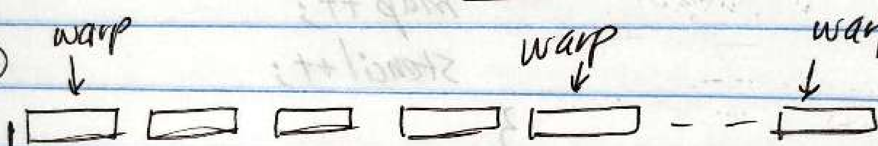


每个block单独 reduce

全局mem →

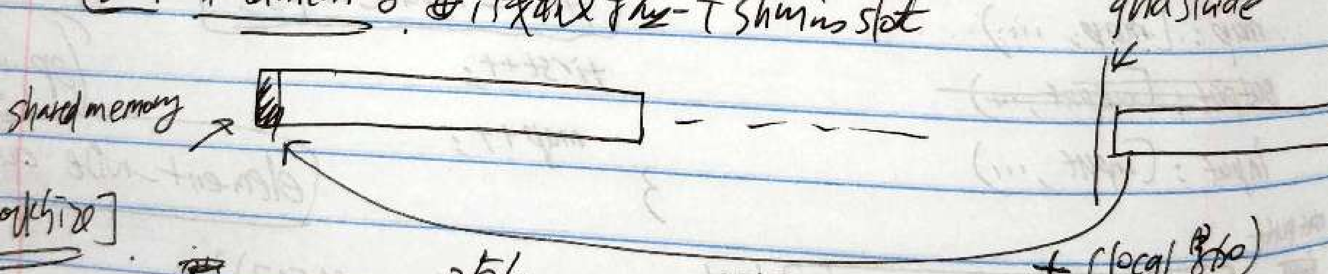
再copy到 host mem, 在host上做 reduce

block_reduce

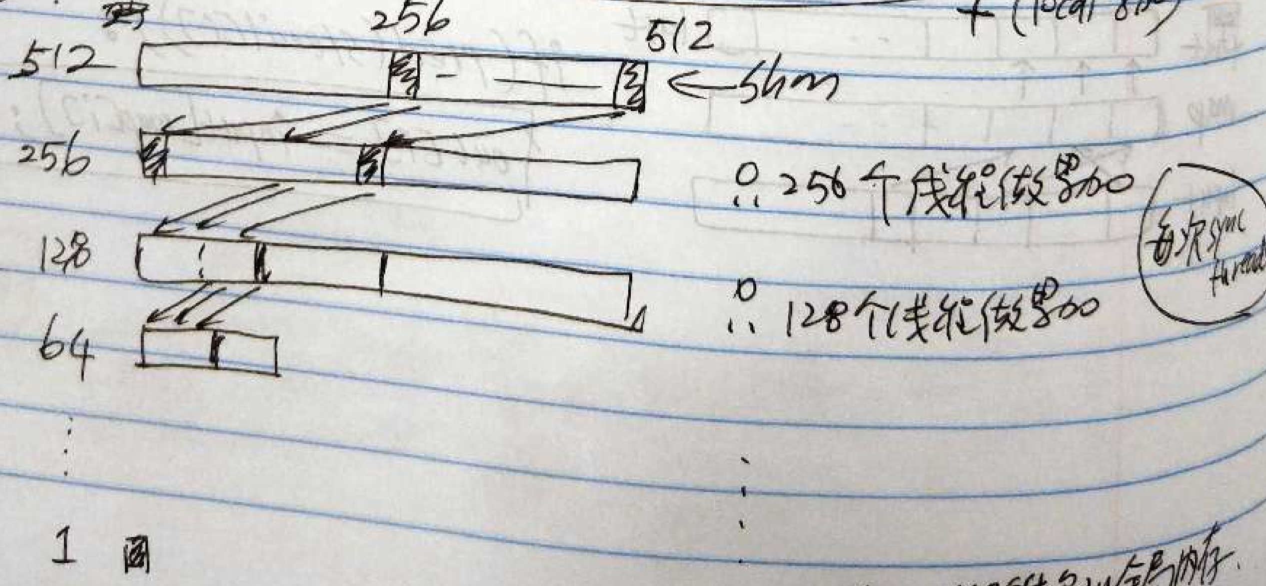


① 先 local reduce blak线程 grid stride 多个线程.

写入 sharedmemory. 每个线程对应一个 shm中的slot



shm[Blocksize]

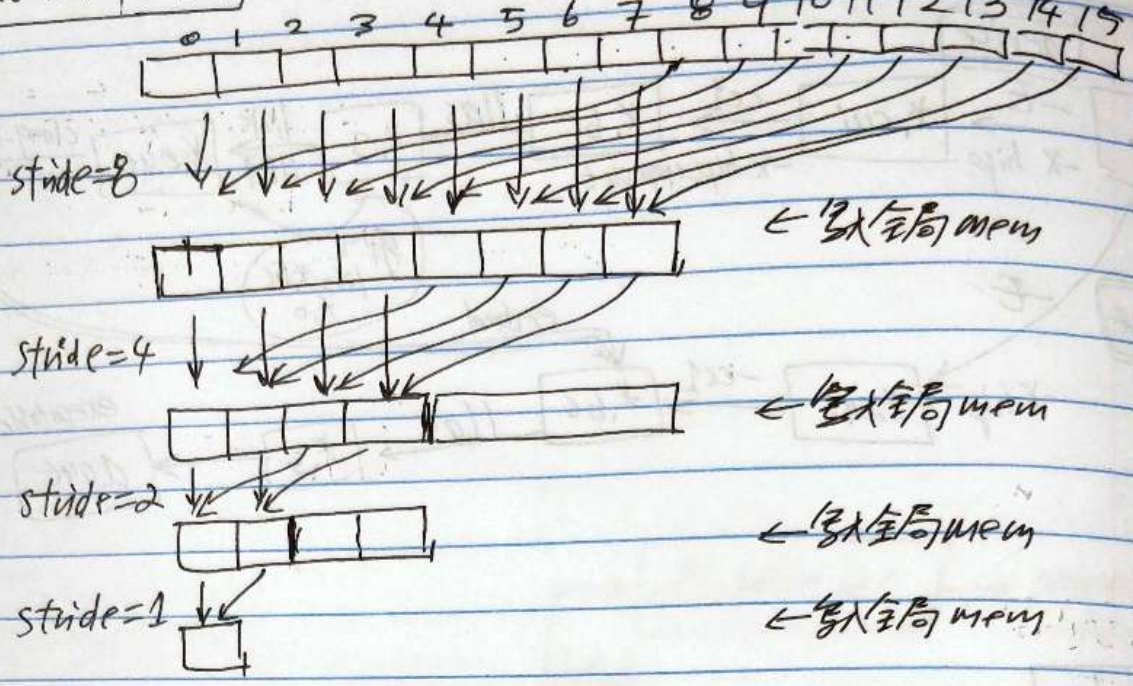
② 

1 图

→ 第一个线程将结果写到全局内存.

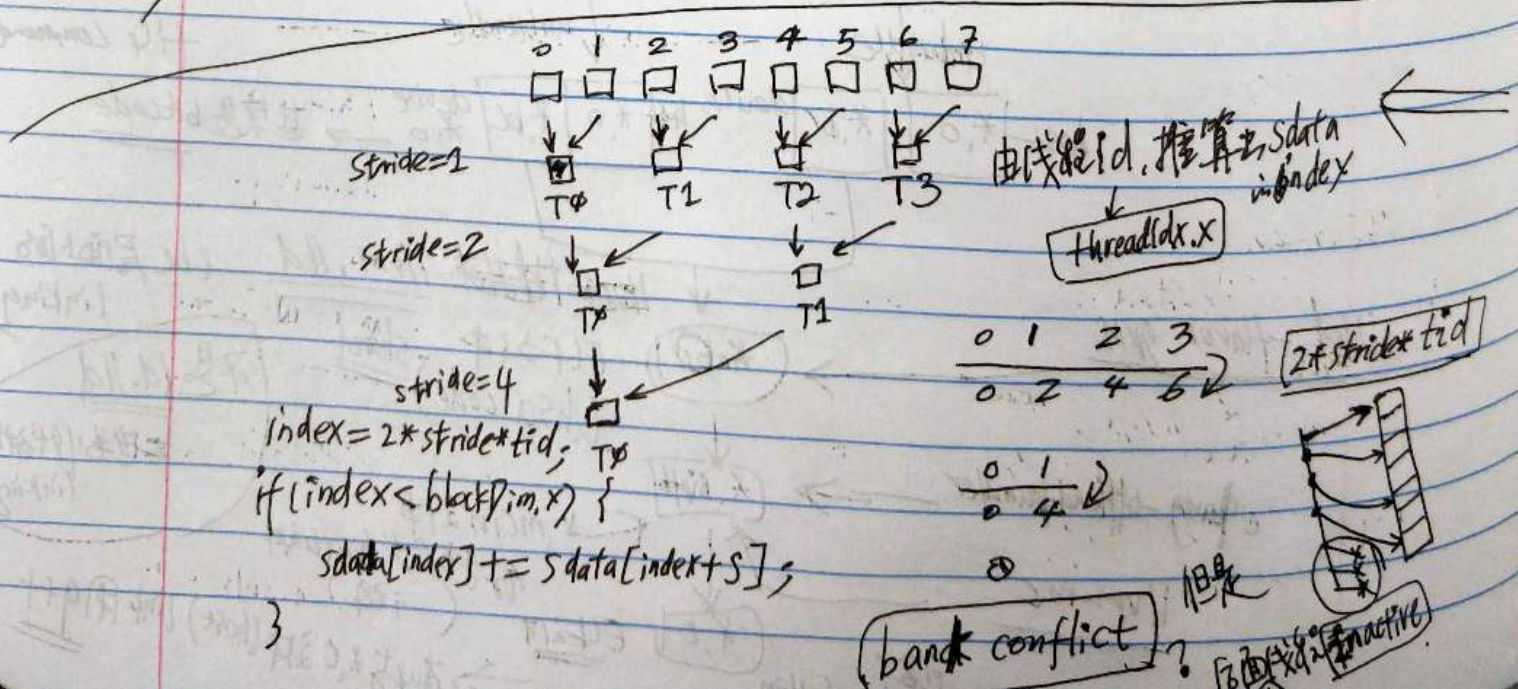
reduction kernel [CUDA]

- gmem



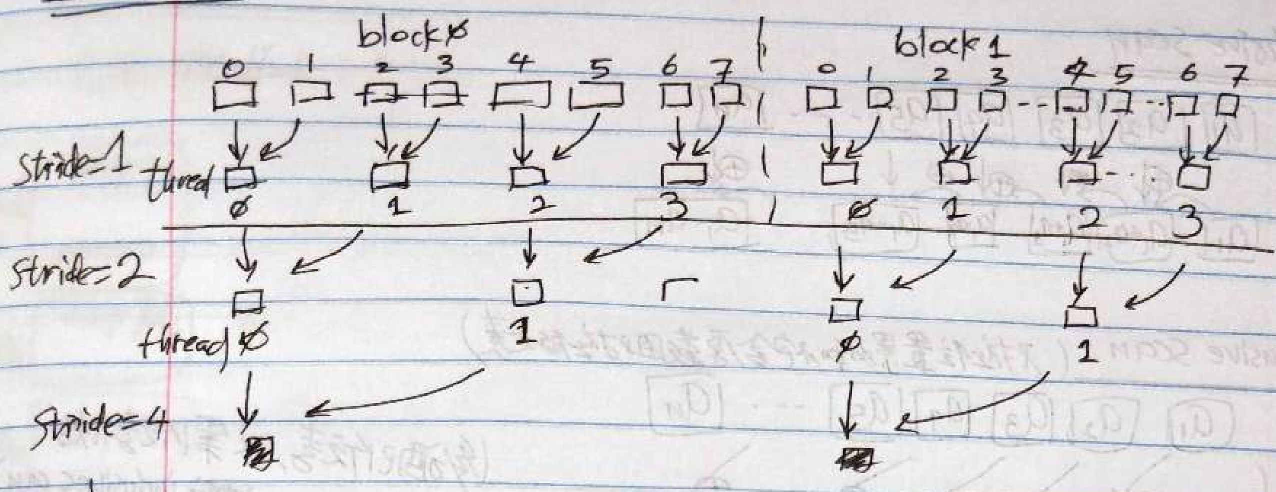
kernel:

```
auto idx = blockDim.x * blockIdx.x + threadIdx.x;
if (idx < size && idx + stride < size) {
    data[idx] += data[idx + stride];
}

for (int stride = size/2; stride > 0; stride >>= 1) {
    dim3 blockSz(256);
    dim3 gridSz((stride + blockSz.x - 1) / blockSz.x);
    reduce_gmem <<< gridSz, blockSz >>> (data, size, stride);
}
```




```
index = 2 * stride * tid;
if (index < blockDim.x) {
    sdata[index] += sdata[index + s];
}
```

· shared mem



stride < blockDim.x

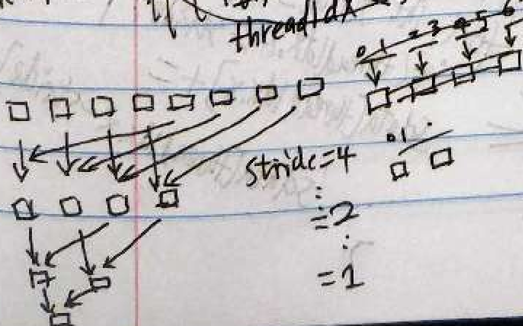
```
auto idx = blockIdx.x * blockDim.x + threadIdx.x;

extern __shared__ int s_data[];
sdata[threadIdx.x] = (idx < size) ? in[idx] : ∅;    } load gmem to shared mem
                                                     ^上个处可预先reduce 一些全局mem
__syncthreads();

for (int stride=1; stride < blockDim.x; stride <<= 1) {
    if (idx % (stride * 2) == ∅)
        sdata[threadIdx.x] += sdata[threadIdx.x + stride];

    __syncthreads();
}

if (threadIdx.x == ∅)
    data[blockIdx.x] = sdata[∅];
```

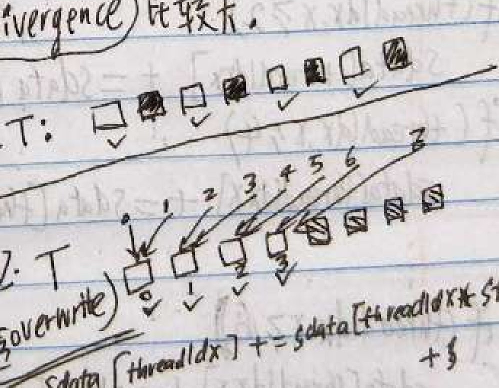
thread: ∅ ①2③4⑤6⑦ 21午 — stride=1
∅ 4 — stride=2
∅ — stride=4

divergence 比较大。

可以改为：if (threadIdx < stride) (stride = blockDim.x/2 → ∅)

```
if (threadIdx.x < stride) {
    sdata[threadIdx.x] += sdata[threadIdx.x + stride];
}
```



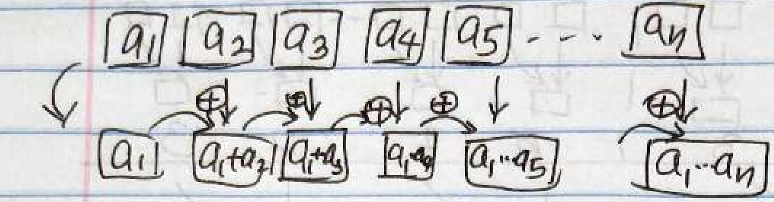


(会overwrite) sdata[threadIdx] += sdata[threadIdx + stride + 1]

Scan

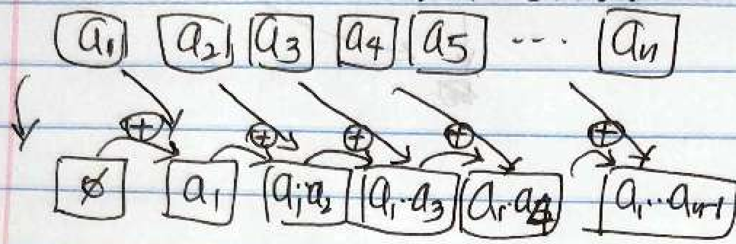
thrust-1.0.0

inclusive scan

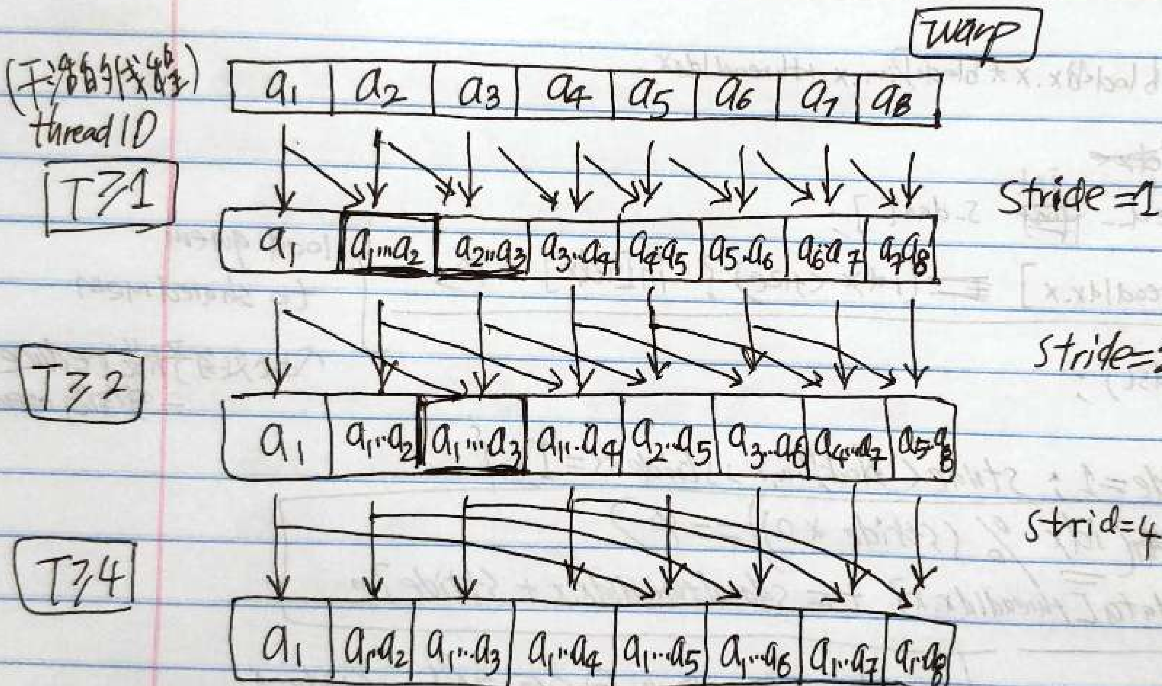


exclusive scan

(对每个位置累加和不包含原数组对应的元素)



(多了一个元素，从第1个元素开始进行 inclusives can)
第0个元素列入0，丢弃最后元素)

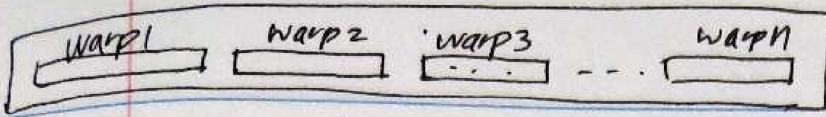



```
if(threadId.x ≥1)
    sdata[threadIdx.x] += sdata[threadIdx.x-1];
if(threadIdx.x ≥2)
    sdata[threadIdx.x] += sdata[threadIdx.x-2];
if(threadIdx.x≥4)
    sdata[threadIdx.x] += sdata[threadIdx.x-4];
    ⋮
if(threadIdx.x≥16)
    sdata[threadIdx.x] += sdata[threadIdx.x-16];
```

⇐ 每个线程(≥1)都在干活.

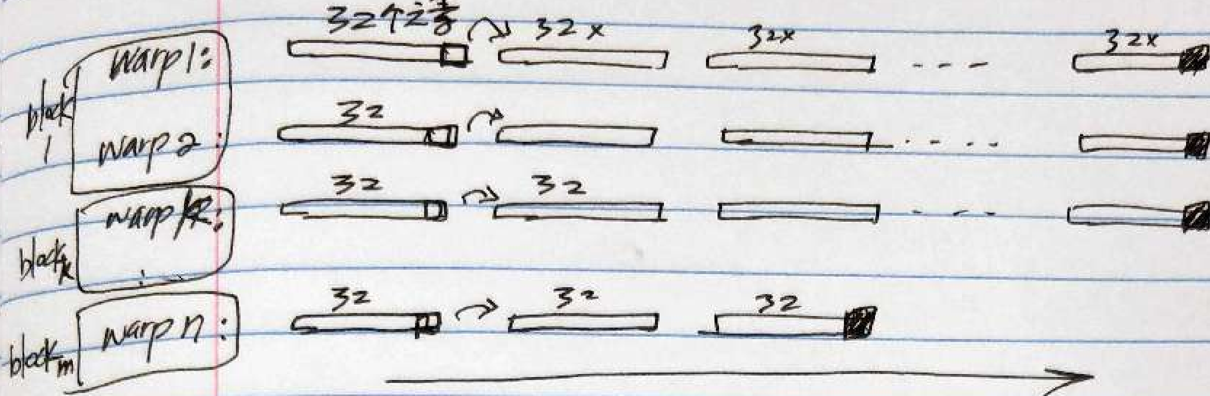
⇐ 每个线程(≥2)都在同时干活

```
//pragma unroll.
for( int stride=1; stride<warpsize; stride<<=1){
    if(threadIdx.x≥stride)+=
    sdata[threadIdx.x]+=
        sdata[threadIdx.x-stride];
}
```



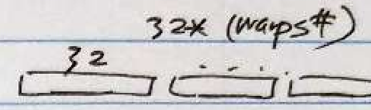
(thread block)

只进行一轮 block scheduling (1 wave), 让每个warp处理多个元素. 从而计算, 每个warp处理的数组个数。



(连续的数组序列)
数组大小: 32

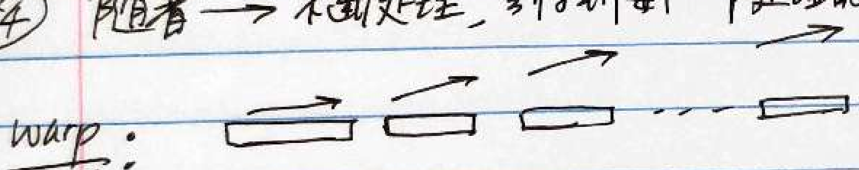
①: 每个warp先将32个元素 read in shared memory :



②. 每个warp调用 warpScan., 最后元素(每个warp 对应的 shm segment)
保存着 reduced sum (carry) (carry也放在shm)

③ 每个warp 进行下一个32元素的数组. 在之前, 将carry累加到待处理数组的第一个元素.

④ 随着 → 不断处理, 会得到每个warp处理的 scan 的数组序列.



(Note, warp处理的元素是连续的序)

每个warp最后元素(处理之后)就是 final carry, 写到 global memory