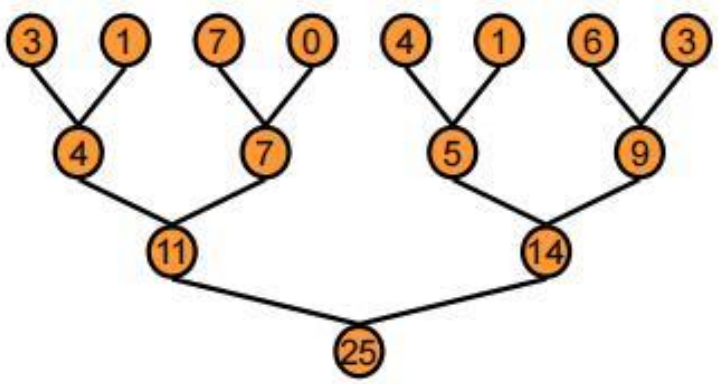
利用并行归约进行数组求和

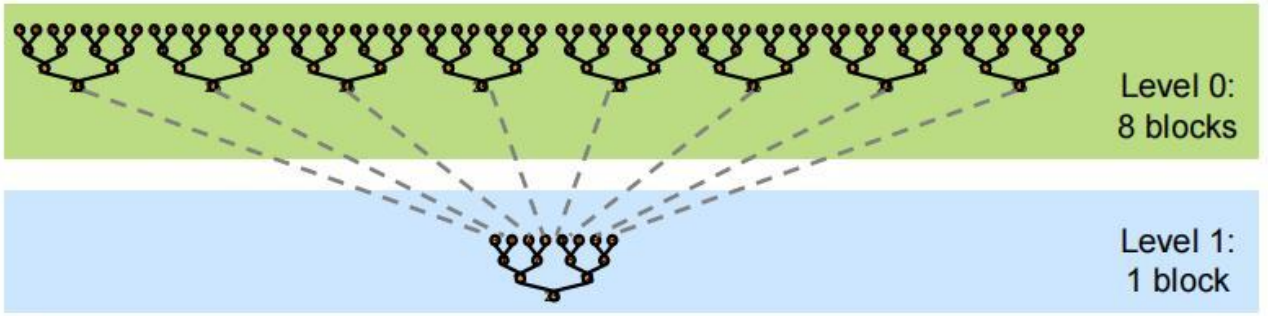
张武举

并行算法设计

- 在GPU中，reduce采用了一种树形的计算方式。如下图所示。从上至下，将数据不断地累加，直到得出最后的结果，即25。



- 但由于GPU没有针对global数据的同步操作，只能针对block的数据进行同步。所以，一般而言将reduce分为两个阶段，其示意图如下



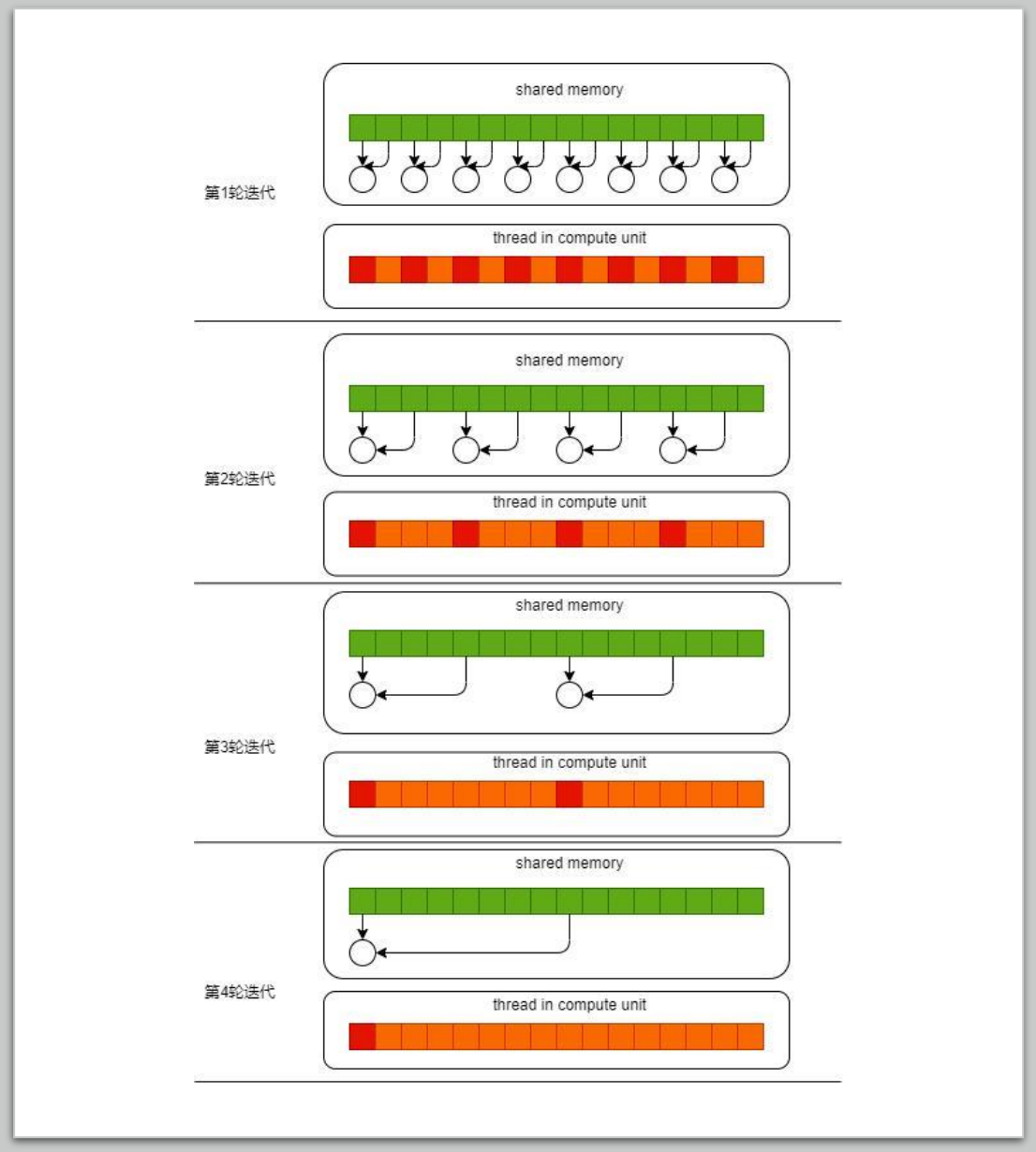
baseline算法

- Baseline算法比较简单，分为三个步骤。第一个步骤是将数据load至shared memory中，第二个步骤是在shared memory中对数据进行reduce操作，第三个步骤是将最后的结果写回global memory中

```
template <class T>
__global__ void reduce0(T* g_idata, T* sum_res, unsigned int n) {
  extern __shared__ T sdata[];

  // load shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
  sdata[tid] = (i < n) ? g_idata[i] : 0;
  __syncthreads();

  // do reduction in shared mem
  for (unsigned int s = 1; s < blockDim.x; s *= 2) {
    // modulo arithmetic is slow!
    if ((tid % (2 * s)) == 0) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }

  // write result for this block to global mem
  if (tid == 0)
    atomicAdd(sum_res, sdata[0]);
}
```

baseline算法

- 第二个阶段，block中需要计算的256个元素已经全部被存储在了shared memory中，此时需要对其进行reduce操作。这个过程需要进行多轮迭代：
  - 在第一轮迭代中，如果tid%2 ==0, 则第tid号线程将shared memory中第tid号位置的值和第tid+1号的值进行相加，而后放在第tid号位置。
  - 在第二轮迭代中，如果tid%4==0,则第tid号线程将shared memory中第tid号位置的值和第tid+2号的值进行相加，而后放在第tid号位置。
  - 不断迭代，则所有元素都将被累加到第0号位置。其示意图如下。其中，**红色的线程代表符合if条件的线程，只有它们有任务，需要干活**



版本1：解决warp发散

- 因为存在if语句，一个warp中只有线程id为偶数的线程执行红框内的加法，线程id为奇数的线程不执行加法任务。此处存在warp发散

```
template <class T>
__global__ void reduce0(T* g_idata, T* sum_res, unsigned int n) {
  extern __shared__ T sdata[];

  // load shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
  sdata[tid] = (i < n) ? g_idata[i] : 0;
  __syncthreads();

  // do reduction in shared mem
  for (unsigned int s = 1; s < blockDim.x; s *= 2) {
    // modulo arithmetic is slow!
    if ((tid % (2 * s)) == 0) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }

  // write result for this block to global mem
  if (tid == 0)
    atomicAdd(sum_res, sdata[0]);
}
```

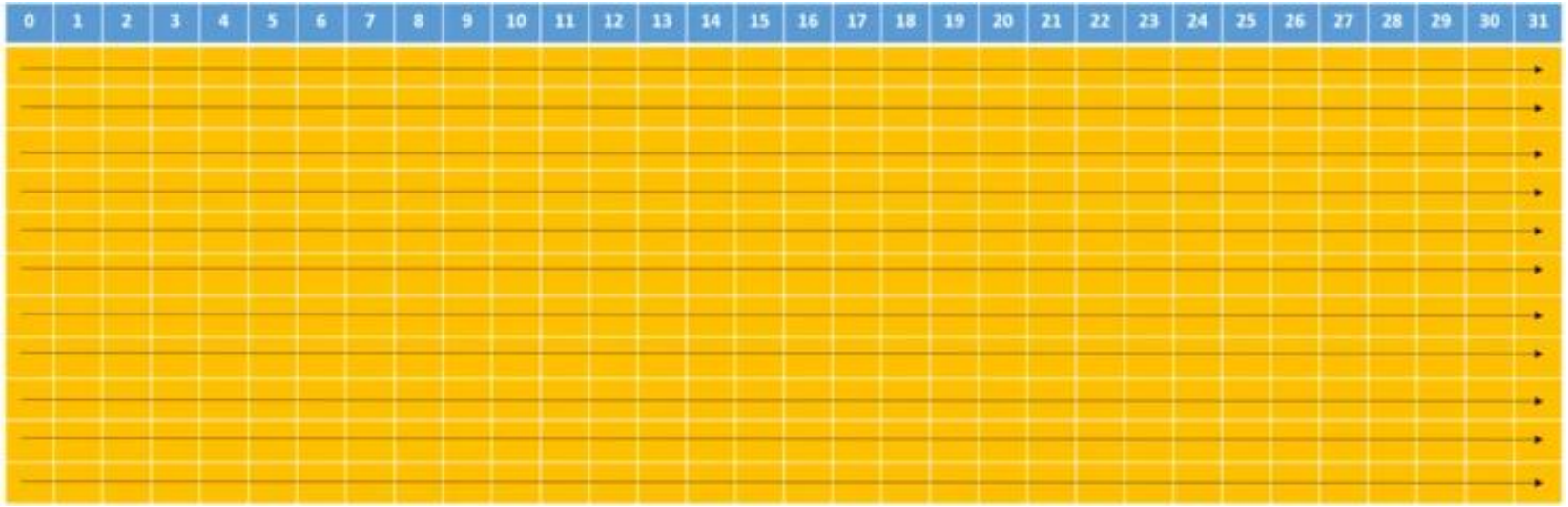
版本1：解决warp发散

```
template <typename T>
__global__ void reduce1(T* input_data, T* sum_res, unsigned int
data_size) {
  extern __shared__ T sdata[];
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
  sdata[tid] = (i < data_size) ? input_data[i] : 0;
  __syncthreads();
  // do reduction in shared mem
  for (unsigned int s = 1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;
    if (index < blockDim.x)
      sdata[index] += sdata[index + s];
    __syncthreads();
  }
  // write result for this block to global mem
  if (tid == 0) {
    atomicAdd(sum_res, sdata[0]);
  }
}
```

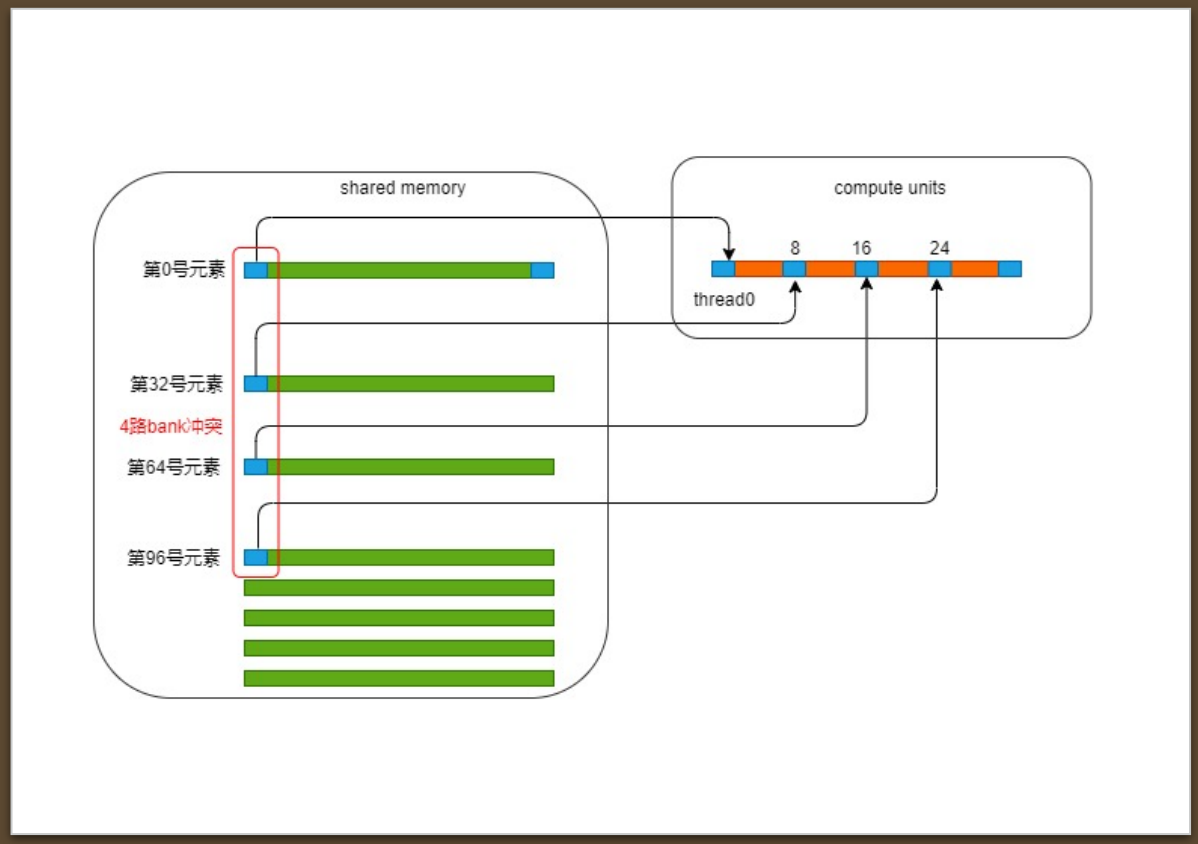
- 假设blockDim.x为256.
- 第一轮迭代时，前4个warp(每个warp为32个线程，4个warp总共128个线程)执行if语句，后4个warp不执行if里面的指令。但是对于单个warp来说，执行的指令是完全相同的
- 第二轮迭代时，前2个warp执行if中的指令，后2个不执行

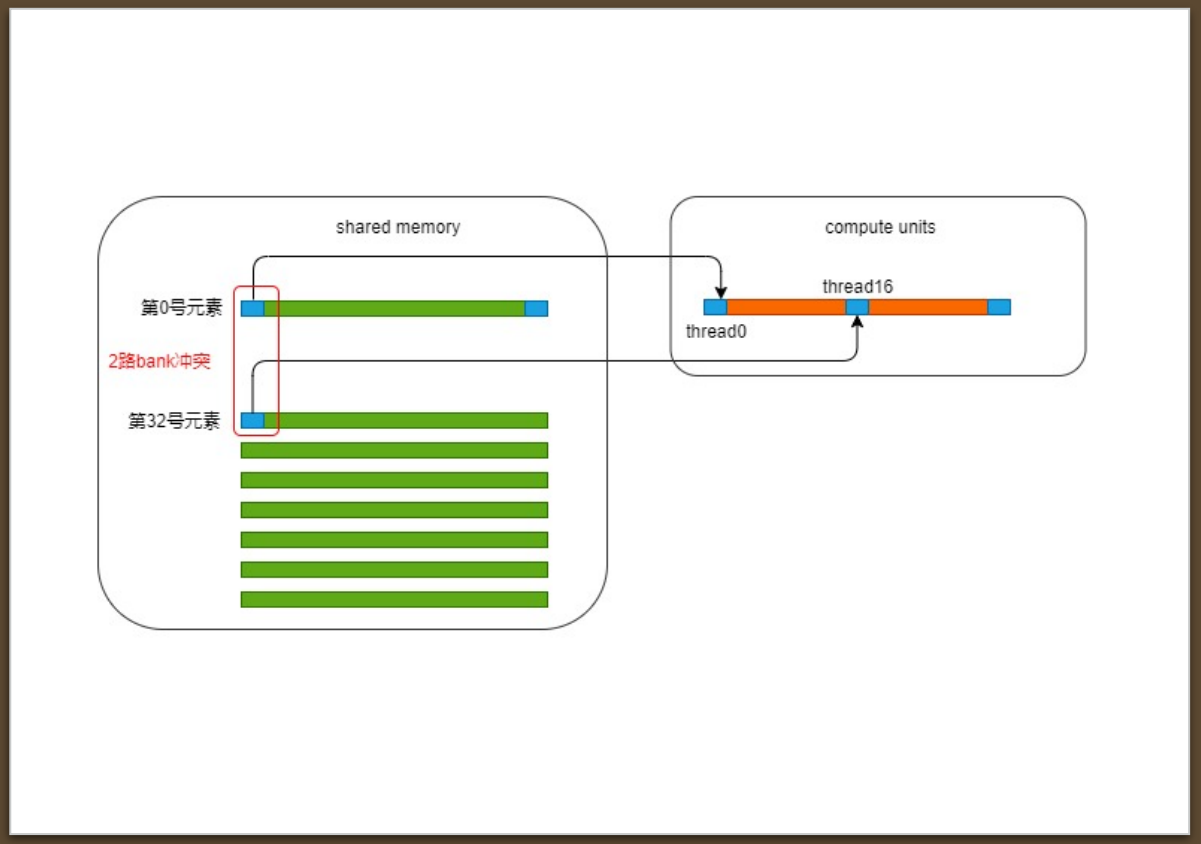
版本2：解决bank冲突

- 要解决bank冲突，首先我们要了解一下共享内存的地址映射方式。
- 在共享内存中，连续的32-bits字被分配到连续的32个bank中，这就像电影院的座位一样：一列的座位就相当于一个bank，所以每行有32个座位，在每个座位上可以“坐”一个32-bits的数据(或者多个小于32-bits的数据，如4个char型的数据，2个short型的数据)；而正常情况下，我们是按照先坐完一行再坐下一行的顺序来坐座位的，在shared memory中地址映射的方式也是这样的。下图中内存地址是按照箭头的方向依次映射的：
- 上图中数字为bank编号。这样的话，如果你将申请一个共享内存数组(假设是int类型)的话，那么你的每个元素所对应的bank编号就是地址偏移量(也就是数组下标)对32取余所得的结果，比如大小为1024的一维数组myShMem：
  - myShMem[4]: 对应的bank id为#4 (相应的行偏移量为0)
  - myShMem[31]: 对应的bank id为#31 (相应的行偏移量为0)
  - myShMem[50]: 对应的bank id为#18 (相应的行偏移量为1)
  - myShMem[128]: 对应的bank id为#0 (相应的行偏移量为4)



版本2：解决bank冲突



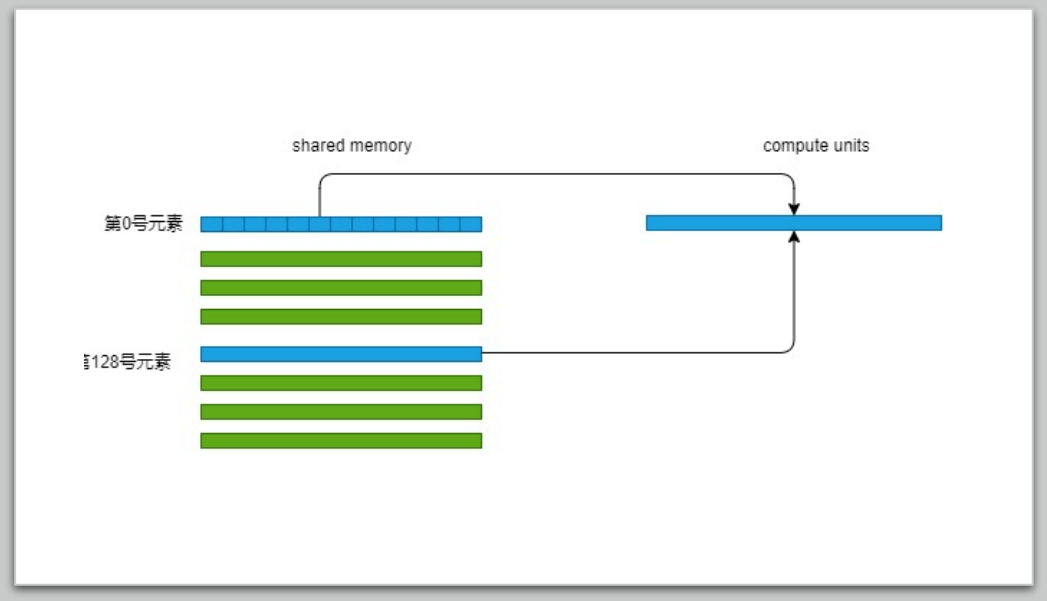


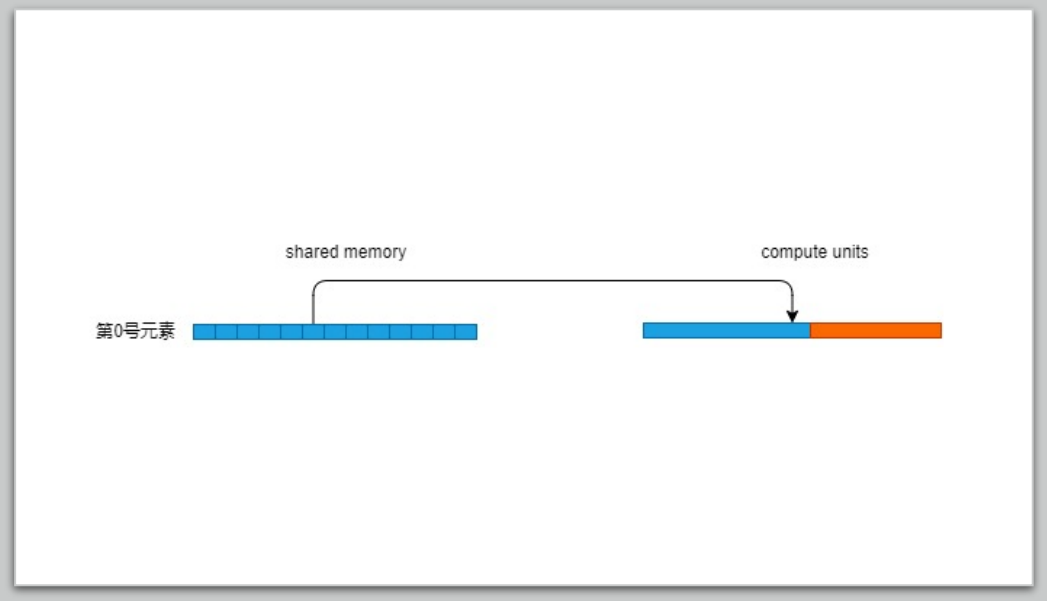
版本2：解决bank冲突

```
template <class T>
__global__ void reduce2(T* g_idata, T* sum_res, unsigned int n) {
  extern __shared__ T sdata[];
  // load shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
  sdata[tid] = (i < n) ? g_idata[i] : 0;
  __syncthreads();
  // do reduction in shared mem
  for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1) {
    if (tid < s) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }
  // write result for this block to global mem
  if (tid == 0) {
    atomicAdd(sum_res, sdata[0]);
  }
}
```

版本2：解决bank冲突

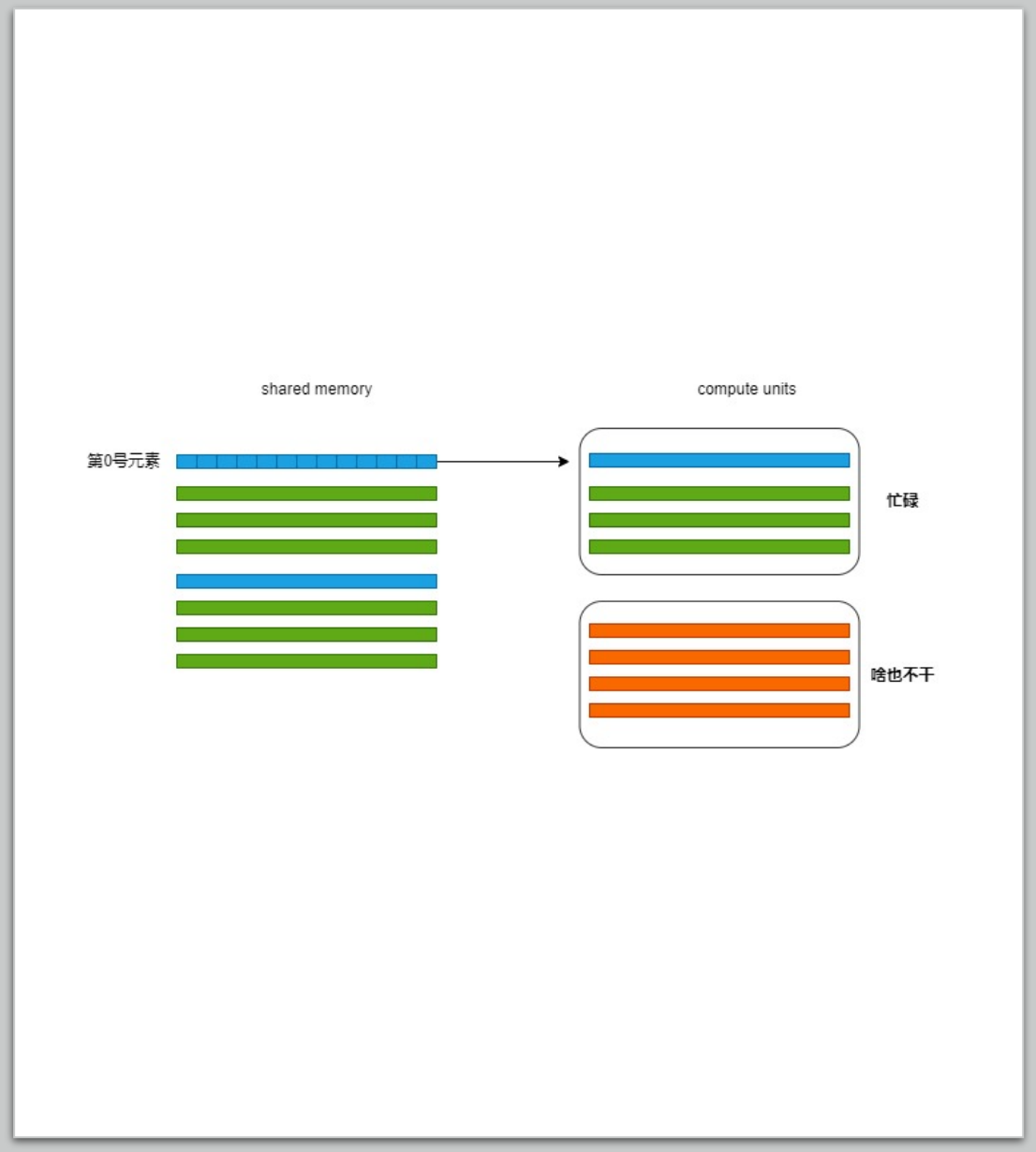
- 分析0号warp。0号线程需要load shared memory的0号元素以及128号元素。1号线程需要load shared memory中的1号元素和129号元素。这一轮迭代中，在读取第一个数时，warp中的32个线程刚好load 一行shared memory数据。
- 再分析第2轮迭代，0号线程load 0号元素和64号元素，1号线程load 1号元素和65号元素。每次load shared memory的一行。
- 再来分析第3轮迭代，0号线程load 0号元素和32号元素，接下来不写了，总之，一个warp load shared memory的一行。没有bank冲突。
- 到了4轮迭代，0号线程load 0号元素和16号元素。16号线程啥也不干，因为s=16，16-31号线程啥也不干，跳过去了





版本3：解决idle线程

- reduce2最大的问题就是线程的浪费。可以看到我们启动了256个线程，但是在第1轮迭代时只有128个线程在干活，第2轮迭代只有64个线程在干活，每次干活的线程都会减少一半。第一轮迭代示意图如下，只有前128个线程在load数据。后128个线程啥也不干



版本3：解决idle线程

```
template <typename T>
__global__ void reduce3(T* input_data, T* sum_res, unsigned int
data_size) {
  extern __shared__ T sdata[];
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x * (blockDim.x * 2) + threadIdx.x;
  T mySum = (i < data_size) ? input_data[i] : 0;
  if (i + blockDim.x < data_size)
    mySum += input_data[i + blockDim.x];
  sdata[tid] = mySum;
  __syncthreads();
  // do reduction in shared mem
  for (size_t s = blockDim.x / 2; s > 0; s >>= 1) {
    if (tid < s) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }
  // write result for this block to global mem
  if (tid == 0) {
    // sum_res[blockIdx.x] = sdata[0];
    atomicAdd(sum_res, sdata[0]);
  }
}
```

- 每个线程累加两个元素，元素索引间隔为blockDim.x，也就是一个block的线程个数。这样可以把总的block数目减半。

版本4：展开最后一维减少同步

- reduce3中当进行到最后几轮迭代时，此时的block中只有warp0在干活时，线程还在进行同步操作。这一条语句造成了极大的浪费。

IMPORTANT:
For this to be correct, we must use the volatile keyword!

```
template <typename T>
__device__ void warpReduceSum(volatile T* sdata, int tid) {
  sdata[tid] += sdata[tid + 32];
  sdata[tid] += sdata[tid + 16];
  sdata[tid] += sdata[tid + 8];
  sdata[tid] += sdata[tid + 4];
  sdata[tid] += sdata[tid + 2];
  sdata[tid] += sdata[tid + 1];
}
```

版本5：完全展开减少计算

我们还可以将for循环进一步完全展开，减少for循环的消耗。

```
Template <unsigned int blockSize>
__device__ void warpReduce(volatile int* sdata, int tid) {
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid +  8];
    if (blockSize >=  8) sdata[tid] += sdata[tid +  4];
    if (blockSize >=  4) sdata[tid] += sdata[tid +  2];
    if (blockSize >=  2) sdata[tid] += sdata[tid +  1];
}
```

```
if (blockSize >= 512) {
    if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
if (blockSize >= 256) {
    if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
if (blockSize >= 128) {
    if (tid <  64)  { sdata[tid] += sdata[tid +  64]; } __syncthreads(); }

if (tid < 32) warpReduce<blockSize>(sdata, tid);
```

版本6：让一个线程多干点

- 如果一个线程被分配更多的work时，可能会更好地覆盖延时。如果线程有更多的work时，对于编译器而言，就可能有更多的机会对相关指令进行重排，从而去覆盖访存时的巨大延时。但是block的设定肯定需要合理的设置

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
sdata[tid] = 0;

while (i < n) {
    sdata[tid] += g_idata[i] + g_idata[i+blockSize];
    i += gridSize;
}
__syncthreads();
```