Cooperative Groups in CUDA

Pawel Pomorski

Cooperative Groups in a nutshell

- CUDA before version 9.0 permitted synchronization only within thread blocks. There was no possibility of synchronization between thread blocks within a single kernel.

- CUDA 9.0 introduced Cooperative Groups to overcome this rather severe restriction by organizing groups of cooperating threads. These allow:

- intra-block synchronization, i.e. synchronization within sub-blocks.

- inter-block synchronization, subject to the restriction that all blocks in the kernel must be resident on the GPU

Brief review of CUDA

- CUDA launches massively multi-threaded functions called kernels which run on the GPU
- The threads are grouped into blocks of up to 1024 threads. Threads belonging to a given block can be synchronized.
- Works best for threads which are independent. As threads become more data dependent, complexity of code grows.

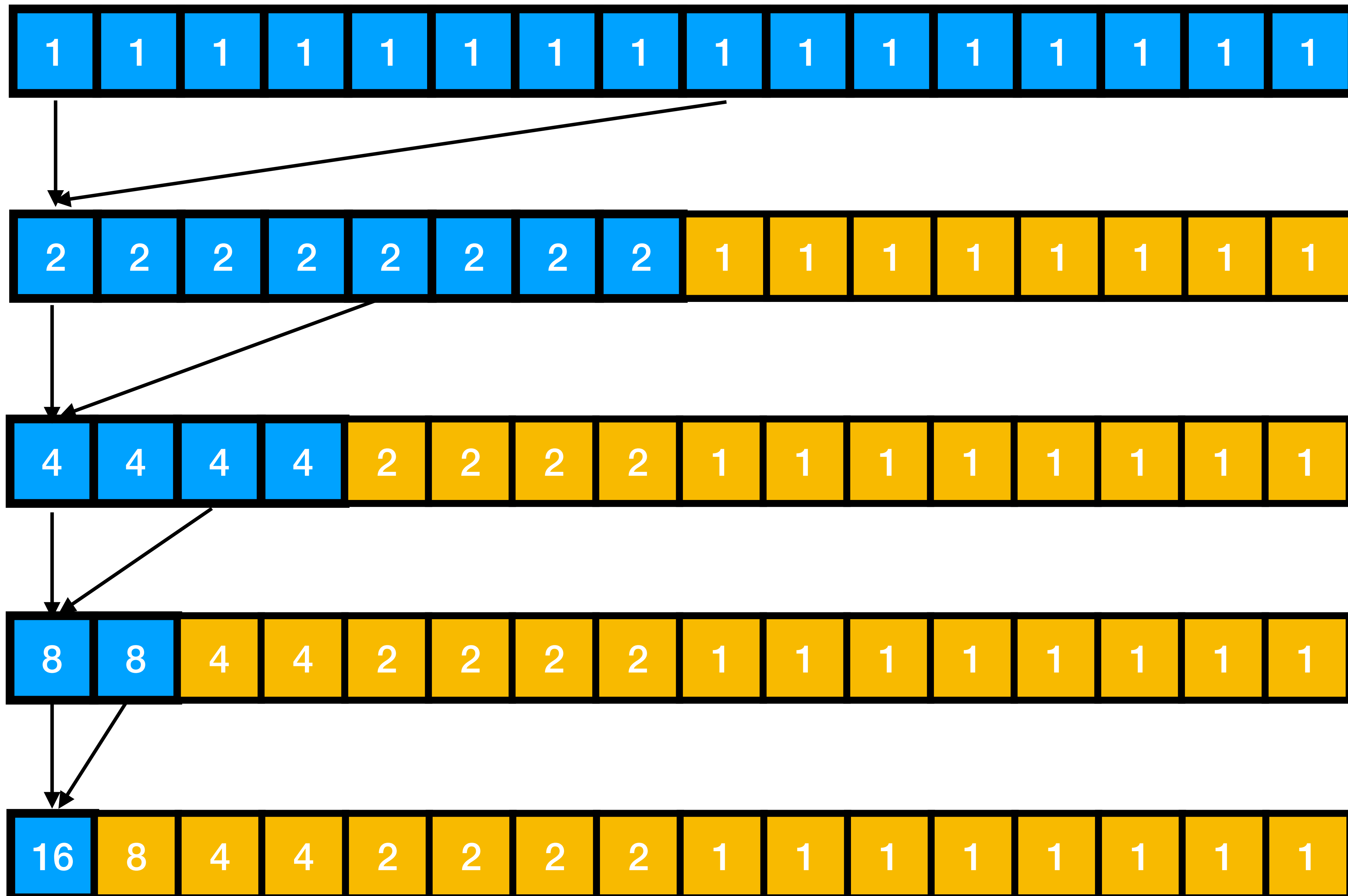
Reduction

$$S(\overrightarrow{x}) = \Sigma_i^n x_i$$

- Add all entries in a 1D array and return the sum.
- Highly data dependent, reduction requires an algorithm to be decomposed between threads. Limitations on thread synchronization have to be taken into account.
- Assign one thread per array element, then use binary reduction with synchronization to get the partial sum within a block.
- Get final sum using atomic operations or another kernel to run another binary reduction

```c
int main (int argc,char **argv)
{
    int i;
    int n = NBLOCKS * BLOCK_SIZE;

    float *input_host,*output_host;
    float *input_dev,*output_dev;

    size_t memsize;
    memsize = n * sizeof(float);

    input_host = (float *)malloc(memsize);
    output_host = (float *)malloc(sizeof(float));

    cudaMalloc((void **) &input_dev, memsize);
    cudaMalloc((void **) &output_dev, sizeof(float));

    for ( i = 0; i < n; i++) {
        input_host[i] = rand() / (float)RAND_MAX;
    }
    cudaMemcpy(input_dev, input_host, memsize, cudaMemcpyHostToDevice);
    reduce<<<NBLOCKS,BLOCK_SIZE>>>(input_dev,output_dev);
    cudaMemcpy(output_host, output_dev, sizeof(float),cudaMemcpyDeviceToHost);

    printf("sum %lf \n",output_host[0]);

    cudaFree(input_dev);
    free(input_host);
    return 0;
}
```

Binary reduction



```
#include "cuda.h" /* CUDA runtime API */
#include <stdio.h>

#define BLOCK_SIZE 1024
#define NBLOCKS 32

__device__ float reduction_sum_block(float *temp, float val)
{
    int thread_ind = threadIdx.x;

    for (int i = blockDim.x / 2; i > 0; i /= 2)
    {
    temp[thread_ind] = val;
    __syncthreads();
    if(thread_ind<i) val += temp[thread_ind + i];
    __syncthreads();
    }

    return val; //only thread with threadIdx.x=0 has the actual sum for block
}

__global__ void reduce(float *input, float *output)
{
    __shared__ float tempspace[BLOCK_SIZE];
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    float block_sum=reduction_sum_block(tempspace,input[i]);
    if (threadIdx.x==0) atomicAdd (&output[0], block_sum);
}
```

Cooperative Groups objects

- Datatype for representing a group of cooperating threads within a block

  thread_block

  or a group of cooperating blocks within a grid of blocks

  grid_group

- **#include <cooperative_groups.h>**
  using namespace cooperative_groups;

- Initialized from existing CUDA block or grid

  thread_block g = this_thread_block();
  grid_group grid = this_grid();

  Can spawn new groups by subdividing existing ones via tiling

- Numerous associated methods: **g.thread_rank(), g.size(), g.sync()** …

```
using namespace cooperative_groups;

__device__ float reduction_sum_block(thread_group g,float *temp, float val)
{
    int thread_ind = g.thread_rank();

    for (int i = blockDim.x / 2; i > 0; i /= 2)
    {
    temp[thread_ind] = val;
    g.sync();
    if(thread_ind<i) val += temp[thread_ind + i];
    g.sync();
    }

    return val; //only thread with threadIdx.x=0 has the actual sum for block
}

__global__ void reduce(float *input, float *output)
{
    __shared__ float temp[BLOCK_SIZE];
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    thread_group g = this_thread_block();
    float block_sum=reduction_sum_block(g,temp,input[i]);
    if (threadIdx.x==0) atomicAdd (&output[0], block_sum);
}
```

Cooperative Kernel

- Required to permit synchronization between blocks, special launch syntax
- All blocks must be resident on the GPU. This limits the number of threads in kernel to 1024 times the number of multiprocessors (eg. 56*1024 on P100 Pascal GPU).

```
 void *kernelArgs[] = {
      (void *)&input_dev, (void *)&partial_sum_dev, (void *)&output_dev
  };

   cudaLaunchCooperativeKernel((void*)reduce_with_barrier, NBLOCKS,
BLOCK_SIZE, kernelArgs);
```

```
__global__ void reduce_with_barrier(float *input, float *partial_sum, float
*output)
{
    __shared__ float temp[BLOCK_SIZE];
    grid_group grid = this_grid();
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    thread_group g = this_thread_block();
    float block_sum=reduction_sum_block(g,temp,input[i]);

    if (threadIdx.x==0) partial_sum[blockIdx.x]=block_sum;

    grid.sync();

    if(i==0){
        output[0]=0.0;
        for(int j=0;j<NBLOCKS;j++){
            output[0]=output[0]+partial_sum[j];
        }

    }
}
```

Tiling to create cooperative subgroups

- Generate new groups via tiling

  ```
  thread_group g = this_thread_block();
  thread_group tile32 = tiled_partition(g, 32);
  ```

- The new group tile32 can now be used by each group of 32 threads within a block to synchronize

- Can repeat this process, generating new tiling from existing tiling, permitting fine grained synchronization, down to individual warps

```
__global__ void reduce_with_barrier(float *input, float *partial_sum, float
*output)
{
    __shared__ float temp[BLOCK_SIZE];
    float block_sum,total_sum;
    grid_group grid = this_grid();
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    thread_group g = this_thread_block();
    block_sum=reduction_sum_block(g,temp,input[i]);

    if (threadIdx.x==0) partial_sum[blockIdx.x]=block_sum;

    grid.sync();

    thread_group tile32 = tiled_partition(g, 32);

    if(blockIdx.x==0 && threadIdx.x<32){
        total_sum=reduction_sum_block(tile32,temp,partial_sum[i]);
    }

    if(blockIdx.x==0 && threadIdx.x==0){
        output[0]=total_sum;
    }

}
```

Multiple levels of cooperative groups