Parallel Reduce with CUDA

Reduce Operation

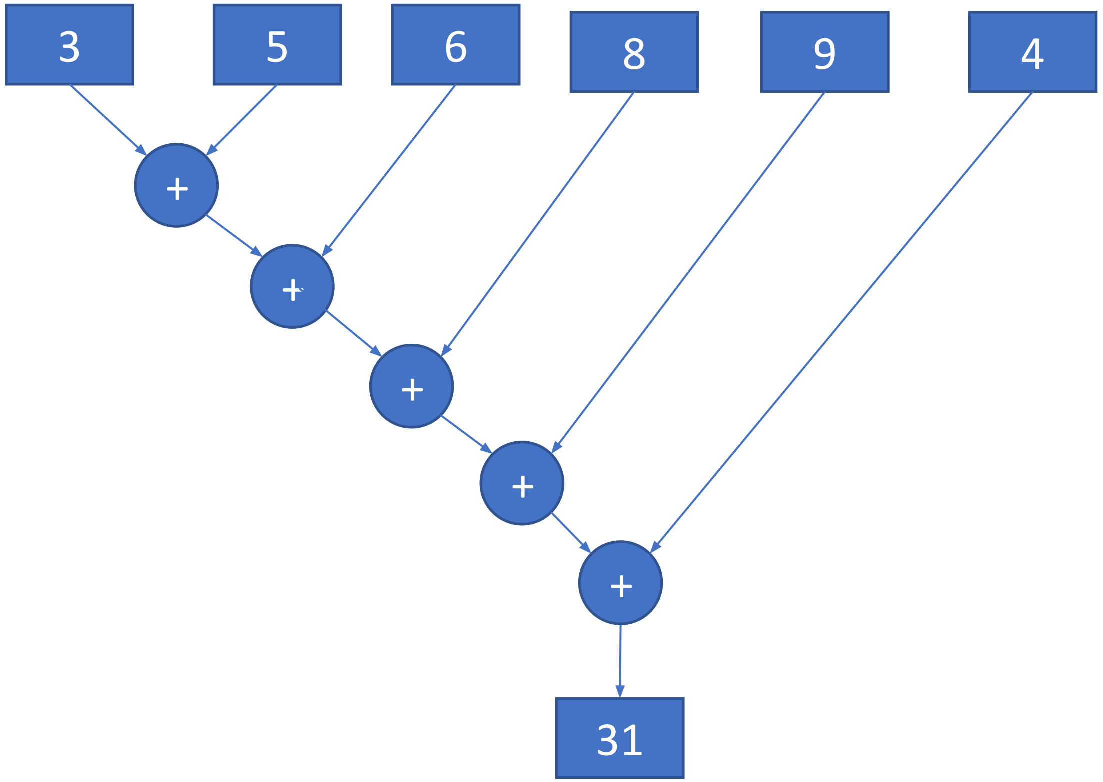
- The reduce operation “reduces” a set of elements according to a reduction operator
- For example:
  - an array of numbers can be reduced using sum operator to find the total sum
  - an array of numbers can be reduced using max operator to find the maximum

Reduce: Sum operator

- Assume we have an array [3, 5, 6, 8, 9,4]
- We can perform a reduction as follows
  - 3 + 5 = 8
  - 8 + 6 = 14
  - 14 + 8 = 22
  - 22 + 9 = 31
  - 31 + 4 = 35
- This operation can also be represented as:
  - (((((3+5)+6)+8)+9)+4) = 35

Reduce: Sum operator

- Let us visualize the sum reduction for the array [3, 5, 6, 8, 9, 4]



Reduce operator

- For the reduce operations, it is necessary that the operator are associative and sometimes commutative
- Associative:
  - Order of operations does not matter
  - For e.g.,
    - (1 + 2) + 3 = 1 + (2 + 3)
    - (1 x 2) x 3 = 1 x (2 x 3)
- Commutative
  - Order of operands does not matter
  - For e.g.,
    - 3 + 5 = 5 + 3
    - 3 x 5 = 5 x 3

Serial implementation of sum operation

- The sum reduce operation can be implemented using a for loop

```
/*
 * Function - Reduce using serial loops
 * ----------------------------
 *   Use a for loop to add the elements
 *
 *   a: vector a
 *   sum: to store results
 *   n: maximum size of vector a
 */
 void serial_reduce(int *a, int *sum, int n) {
   for(int i=0;i<n;i++)
       *sum+=a[i];
}
```

Parallel implementation of sum operation

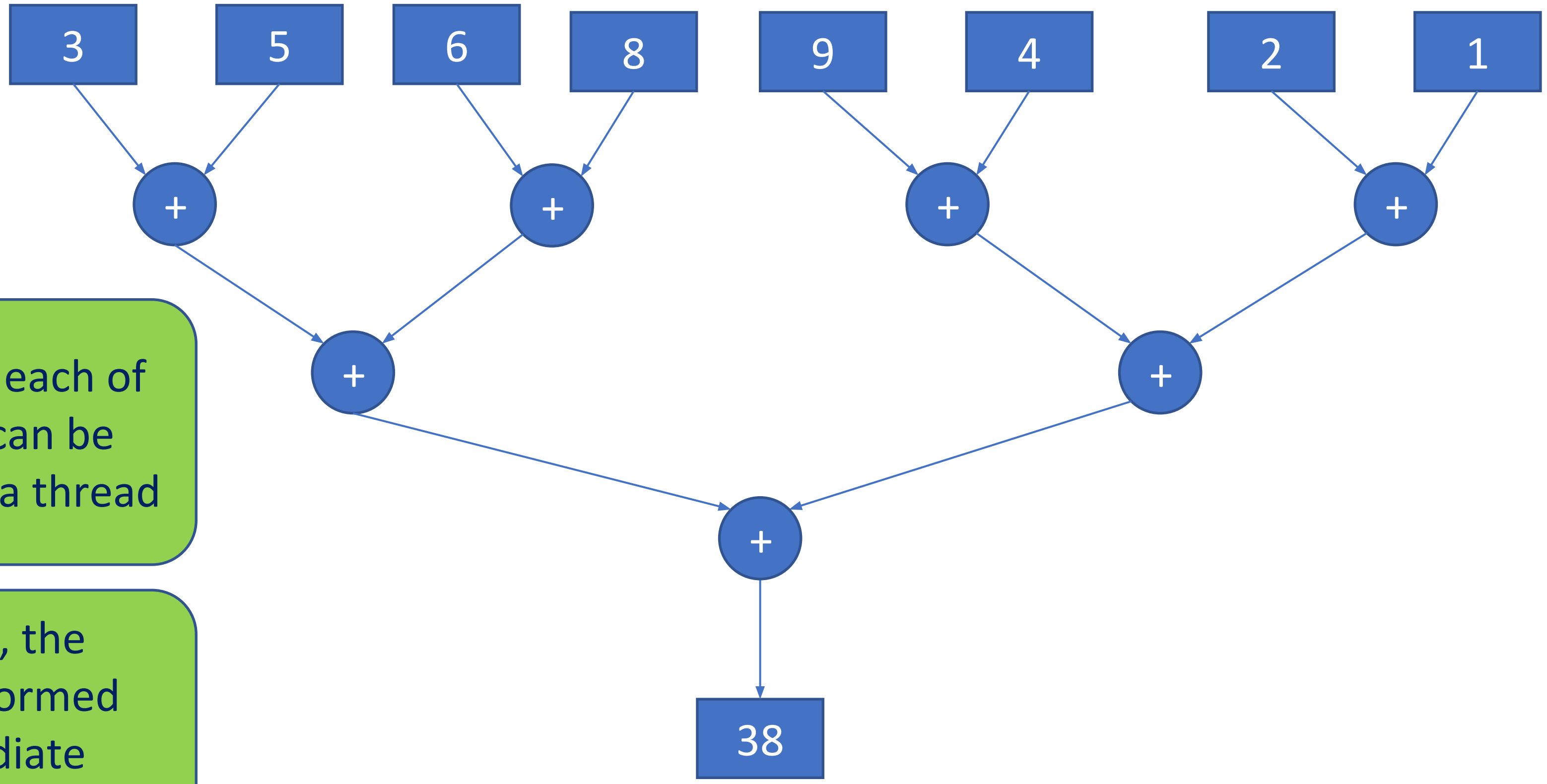
- Issue:
  - For large number of elements, it would take a lot of time for serial operation
- If we can execute this operation in parallel, it can be a lot faster
    - We can use then use CUDA to perform the parallel computation
- Can we parallelize this operation?
  - The associate and commutative properties of sum operation allows it to be added in any order
- How do we parallelize it?
  - Divide the elements into groups, perform the operation for each group in parallel
  - Arrange the results again into groups and perform the operation in parallel
  - Continue until there are only two results
  - Perform the operation to the two results to get the final result

Parallel reduce example

- Here is how we can do parallel reduce for the list [3, 5, 6, 8, 9, 4, 2, 1]
- In parallel we perform (3+5)=8 and (6+8)=14 and (9 + 4) = 13 and (2 + 1) = 3
- Now we perform (8 + 14) = 22 and (13 + 3) = 16 in parallel
- Finally we perform (22 + 16) =38

Parallel Reduce: Sum operator

- Let us visualize the sum reduction for the array [3, 5, 6, 8, 9, 4, 2, 1]

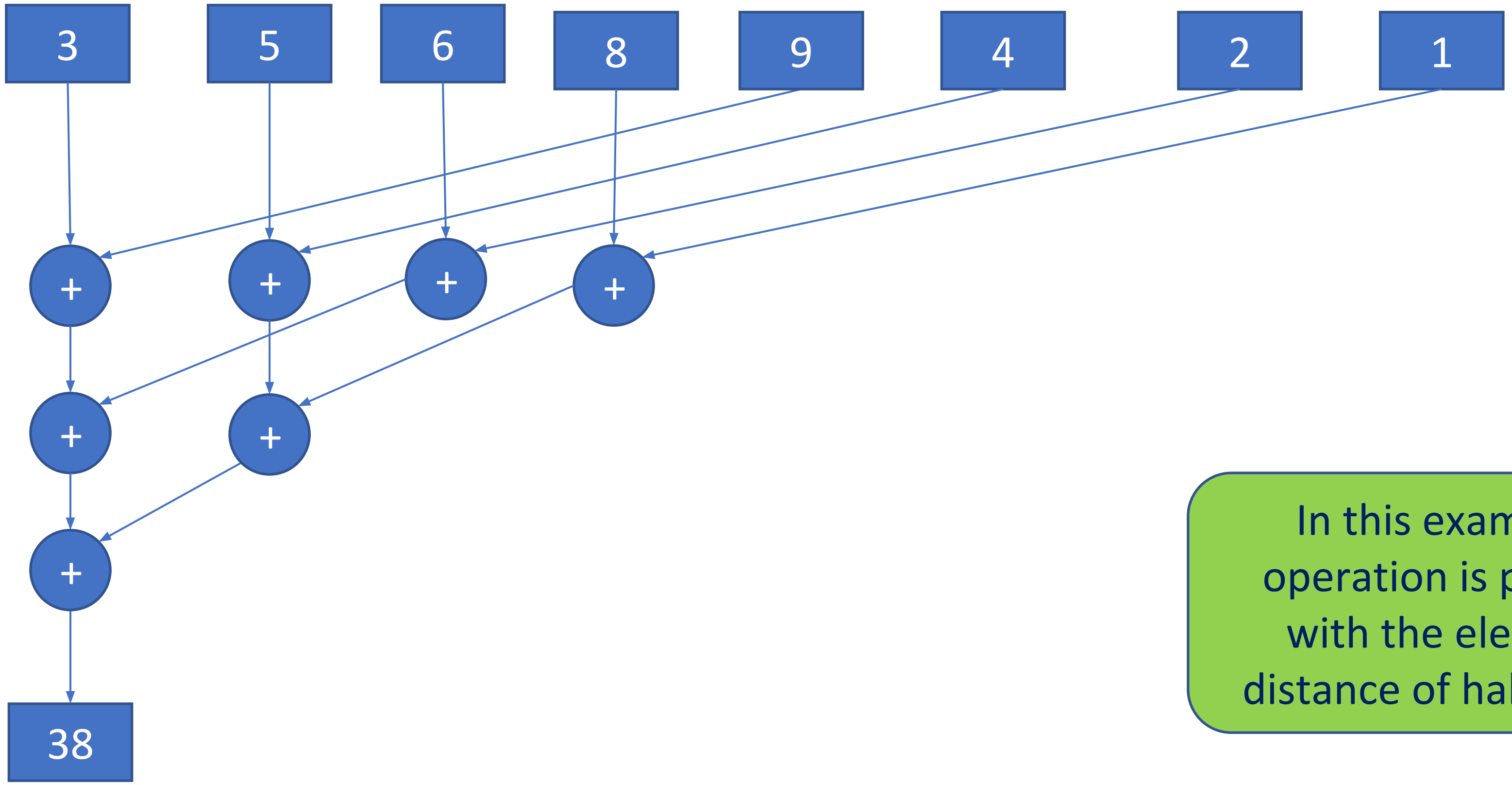


In parallel reduce, each of the + operation can be done in parallel by a thread

In this example, the operation is performed with the immediate neighbors of the elements

Parallel Reduce: Sum operator

- The reduce operation can done in another way as well



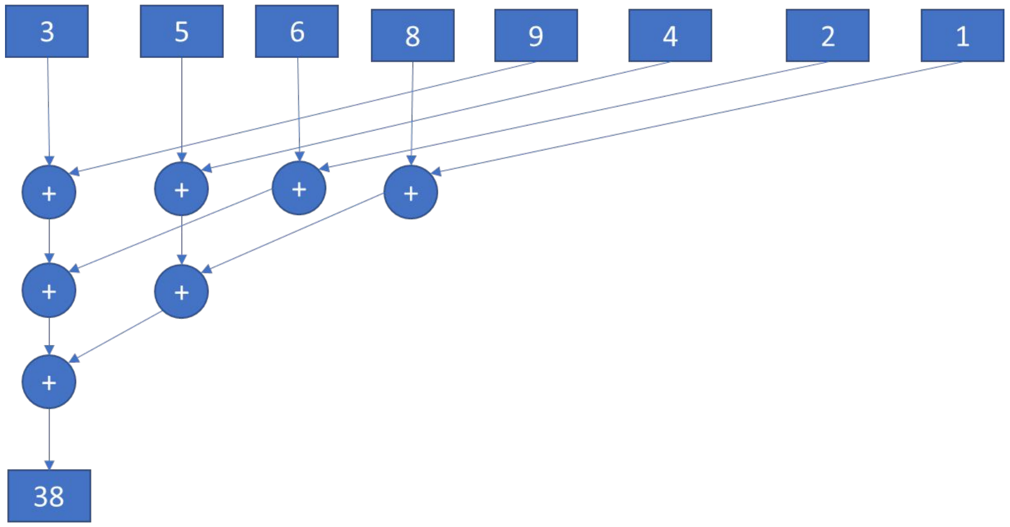
In this example, the operation is performed with the elements at distance of half array size

Let us see a parallel implementation in CUDA

- We can achieve it as following:
  - First launch 1024 blocks, each with 1024 kernels to reduce 1024 elements
  - Each of the block will produce 1 item, so a total of 1024 items
  - Then launch a block with 1024 kernels to reduce the 1024 items in previous step

Let us see a parallel implementation in CUDA

- Let us see how we can implement the reduce in kernel similar to the figure
- There are 1024 blocks, each with 1024 kernels
- So each block is responsible for 1024 elements from the input array
- We can perform reduction as follows
    - For each computation, we can divide the array into half so that we have two 512 element regions
    - Now each of 512 threads can add its element to corresponding element in 2nd half and write back to the array in first half
    - This results in 512 elements, where each element represents the addition (a reduced result) of two elements
    - Now again we divide into two 256-element regions and 256 threads can add corresponding elements from 1st and 2nd region and save them in the 1st region
    - We can continue dividing the region into half until a region with only one element remains
    - We can write that element back to the global memory



Let us see how the kernel looks like

Perform reduction with the element at a half the distance, continue again in the loop

Synchronization

Write back to global memory

```
__global__ void parallel_reduce(int *in, int *out) {
    //get the thread id from block and number of threads
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
    //get the thread id in the block
    int t_id = threadIdx.x;
    //using for loop break continuously into half (1024->512->256...1), until region with 1 element
    for(int i=blockDim.x/2;i>0;i >>= 1){
        if (t_id < i)
            in[thread_id] +=in[thread_id + i]; //add the corresponding element from other half
     //we need to synchronize so that all the threads complete the first operations
     __syncthreads();

    }
      //need to write the result from this block to global memory
    if(t_id==0){
        out[blockIdx.x] = in[thread_id];
    }

}
```

Here is the main function

```
void serial_reduce(int *a, int *sum, int n) {
    //use a serial for loop for addition operation
   for(int i=0;i<n;i++)
       *sum+=a[i];
}
```

This simple program uses 1024 threads to divide 1024*1024 = 2^20 elements into half and so continues working by dividing into half. The size needs to be divided into two equal parts to work properly in this program. For more information check (Harris 2007): https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

- Number of elements = 2^20
- Host variables
- Device variables
- Memory allocation
- Initialize to 1, you can initialize to any values, 1 used for simplicity
- A serial method to add values to test result
- Print serial result

```
int main(int argc, char **argv){
    //size of the array 2^20
    int size = 1<<20;
    printf("%d",size);

    //host vectors, h_in will contain the original array, we will use sum for result
    int *h_in, sum=0;
    //device vectors d_in will contain the original array, we will use d_inter to store intermediate results
    //d_out to store final result
    int* d_in, * d_inter, * d_out;

    size_t size_vect = size*sizeof(int); /* size of the total vectors necessary to allocate memory */
    h_in = (int*)malloc(size_vect);
    //allocate memory for the vectors on device (gpu)
    cudaMalloc((void **)&d_in, size_vect);
    cudaMalloc((void **)&d_inter, size_vect);
    cudaMalloc((void **)&d_out, sizeof(int));

    //initialize the vectors each with value 1 for simplicity
    for (int i = 0; i < size; i++) {
        h_in[i] = 1;
    }

    //use serial function for sum
    serial_reduce(h_in, &sum, size);

    //Verify the result by adding all the sum, should be size
    printf("Serial results:%d\n",sum);
    //Continued on next slide
```

Here is the main function

- Copy the data to device from host
- Define the number of threads
- Define the block size
- First computation on device to reduce to 1024 elements
- In the second computation, 1024 threads add the result to a single output
- Copy the data back from device to host
- Print the parallel result
- Release the device memory
- Release the host memory

```
    //Start CUDA processing
    // Copy vector host values to device
    cudaMemcpy(d_in, h_in, size_vect, cudaMemcpyHostToDevice);

    //define number of threads
    int threads = 1024;
    //define block size in integer
    int block_size = (int)ceil((float)size / threads);
    //execute the kernel with block size and number of threads
    //this is the first phase where the computation results 1024 elements
    parallel_reduce<<<block_size,threads>>>(d_in, d_inter);

    //add the 1024 elements to a single result using
    parallel_reduce<<<1,block_size>>>(d_inter, d_out);

    // Copy result back to host
    cudaMemcpy(&sum, d_out, sizeof(int), cudaMemcpyDeviceToHost);

    //Verify the result by adding all the sum
    printf("Device sum global :%d\n",sum);


    // Release all device memory
    cudaFree(d_in);
    cudaFree(d_inter);
    cudaFree(d_out);

    // Release all host memory
    free(h_in);

}
```

Use of global memory in kernel

- In the kernel, we are accessing the global memory quite frequently
- In each iteration of the loop, we read n items from global memory at step 6 and write back n/2 items at step 10
- In next step, we read n/2 items at step 6 and write back n/4 items at step 10, and so on..
- Instead of writing to global memory, we can use shared memory and complete all computations in device and transfer only the final result

```
1.  __global__ void parallel_reduce(int *in, int *out) {
2.      int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
3.      int t_id = threadIdx.x;
4.      for(int i=blockDim.x/2;i>0;i >>= 1){
5.          if (t_id < i)
6.              in[thread_id] +=in[thread_id + i];
7.       }
8.      __syncthreads();
9.      if(t_id==0){
10.         out[blockIdx.x] = in[thread_id];
11.     }
12.
13. }
```

Global memory updates

Use of shared memory in kernel

Use a shared memory in device

Copy global memory data to shared memory

Update shared memory instead of global memory

Finally update global memory from shared memory

```
__global__ void parallel_reduce_shared(int *in, int *out) {
    //use a shared memory
    extern __shared__ int shared_data[];
   //get the thread id from block and number of threads
   int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
   //get the thread id in the block
   int t_id = threadIdx.x;

   //load data from global mem
   shared_data[t_id]=in[thread_id];
   //synchonize threads to load all data
   __syncthreads();
   //using for loop break continously into half (1024->512->256...1), until region with 1 element
   for(int i=blockDim.x/2;i>0;i >>= 1){
       if (t_id < i)
           shared_data[t_id] +=shared_data[t_id]; //add the corresponding element from other half
       //we need to synchronize so that all the threads complete the first operations
      __syncthreads();
   }

   //need to write the result from this block to global memory
   if(t_id==0){
       out[blockIdx.x] = shared_data[0];
   }
}
```

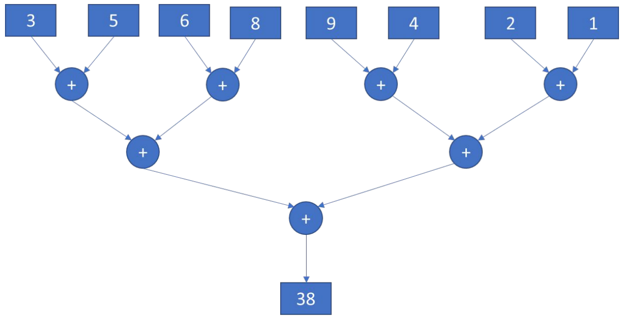
Change in the main function

Declare the amount of shared data (bytes needed) in main function during launching the kernel

```
//using shared memory
parallel_reduce_shared<<<block_size,threads,threads*sizeof(int)>>>(d_in, d_inter);
parallel_reduce_shared<<<1,block_size,block_size*sizeof(int)>>>(d_inter, d_out);
```

Exercise

1. Review the given source code for parallel reduction using sum operator for 2^20 elements
2. Write a kernel that performs reduction as shown in figure using shared memory. More details in the exercise document.



3. Write a program for parallel reduction using max operation to find the max value among 2^20 elements

References

- Harris, M. (2007). Optimizing parallel reduction in CUDA. *Nvidia developer technology*, *2*(4), 70.